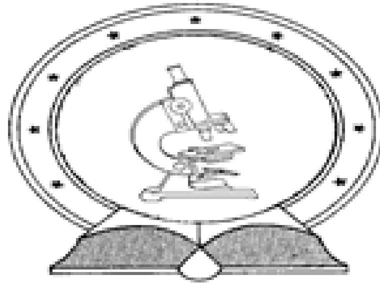
DE TTK



1949

The Appearance of Multiparadigm Programming Languages in the Teaching of Artificial Intelligence

Egyetemi doktori (Ph.D.) értekezés

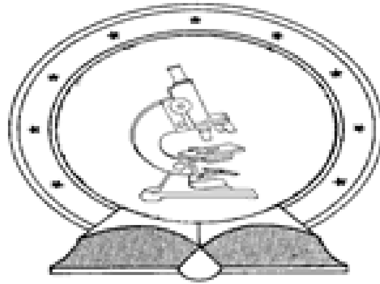
Pánovics János

Témavezető: Dr. Fazekas Gábor

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika- és Számítástudományok Doktori Iskola

Debrecen, 2013

DE TTK



1949

The Appearance of Multiparadigm Programming Languages in the Teaching of Artificial Intelligence

Egyetemi doktori (Ph.D.) értekezés

Pánovics János

Témavezető: Dr. Fazekas Gábor

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika- és Számítástudományok Doktori Iskola

Debrecen, 2013

Ezen értekezést a Debreceni Egyetem Természettudományi Doktori Tanács Matematika- és Számítástudományok Doktori Iskola *Szakmódszertan* programja keretében készítettem a Debreceni Egyetem természettudományi doktori (Ph.D.) fokozatának elnyerése céljából.

Debrecen, 2013. április 23.

Pánovics János
doktorjelölt

Tanúsítom, hogy Pánovics János doktorjelölt 1999–2002 között a fent megnevezett Doktori Iskola *Informatika* programjának keretében irányításommal végezte munkáját. Az értekezésben foglalt eredményekhez a jelölt önálló alkotó tevékenységével meghatározóan hozzájárult. Az értekezés elfogadását javasolom.

Debrecen, 2013. április 23.

Dr. Fazekas Gábor
témavezető

The Appearance of Multiparadigm Programming Languages in the Teaching of Artificial Intelligence

Értekezés a doktori (Ph.D.) fokozat megszerzése érdekében
a matematika- és számítástudományok tudományágban

Írta: Pánovics János okleveles programtervező matematikus

Készült a Debreceni Egyetem
Matematika- és Számítástudományok Doktori Iskolája
(Szakmódszertan programja) keretében

Témavezető: Dr. Fazekas Gábor

A doktori szigorlati bizottság:

elnök:	Dr. Páles Zsolt	....
tagok:	Dr. Fekete István	....
	Dr. Várterész Magda	....

A doktori szigorlat időpontja: 2012. március 9.

Az értekezés bírálói:

Dr.	....
Dr.	....
Dr.	....

A bírálóbizottság:

elnök:	Dr.	....
tagok:	Dr.	....
	Dr.	....
	Dr.	....
	Dr.	....

Az értekezés védésének időpontja: 2014. ....

Contents

Acknowledgments **1**

1 Introduction **2**

2 Overview of Functional Programming **5**
2.1 The Lambda Calculus 6
2.2 Functional Approach in Teaching Artificial Intelligence 9
2.2.1 Search Algorithms in Different Programming Languages 13
2.3 Multiparadigm Languages and F# 14

3 Some Implementations of Search Algorithms for Single-Agent Problems **16**
3.1 A Class Hierarchy for Search Algorithms 17
3.2 Various Implementations of Search Algorithms 20
3.2.1 The C# Implementation 20
3.2.2 The First F# Implementation 32
3.2.3 The Second F# Implementation 41
3.2.4 The Third F# Implementation 46
3.2.5 Comparing the Four Implementations 55
3.3 Implementing Specific Problems 58
3.3.1 Towers of Hanoi 58
3.3.2 Funfair Puzzle 68

4 Some Implementations of Search Algorithms on Game Trees **94**
4.1 A Class Hierarchy for Search Algorithms 95
4.2 Various Implementations of Search Algorithms 98
4.2.1 The C# Implementation 98
4.2.2 The First F# Implementation 106
4.2.3 The Second F# Implementation 112
4.2.4 Comparing the Three Implementations 118
4.3 Implementing Specific Problems 119

5 A New Methodology for Teaching Computer Science **130**
5.1 Overview . . . 130
5.2 Current Program Requirements . . . 132
5.3 Some Possible Projects . . . 134
5.3.1 Project #1: Creating a *Reversi* Application . . . 134
5.3.2 Project #2: Creating a Library Information System . . . 140
5.3.3 Further Subjects and Projects . . . 141
5.4 Conclusion . . . 142

6 Supporting Programming Contests with the ProgCont Application **143**
6.1 About Programming Contests . . . 144
6.2 The Architecture and Operation of ProgCont . . . 146
6.3 Our Experience with the Application and Possibilities for Future Development . . . 149

7 Summary **156**

8. Összefoglalás **158**
8.1. Új módszertan az informatikaoktatásban . . . 159
8.2. Multiparadigmás programozási nyelvek használata a mesterséges intelligencia oktatásában . . . 161
8.3. Programozó versenyek lebonyolítása a ProgCont alkalmazással . . . 164

Bibliography **170**

List of Publications **171**

Acknowledgments

First of all, I would like to thank my supervisor, Fazekas Gábor, for his guidance and encouragement. I am also very grateful for the invaluable help and support given by Várterész Magda, who acted like a mentor throughout my whole career. Many thanks to my colleague and friend Kósa Márk for all the useful advice he has given me, as well as Sterbinszky Nóra for the base idea of this thesis. Finally, I thank my family for their support and long-suffering patience.

The work is supported by the TÁMOP-4.2.2/B-10/1-2010-0024 project.
The project is co-financed by the European Union and the European Social Fund.

CHAPTER 1

Introduction

Until recently, programming was about using pure, single-paradigm techniques. However, nowadays programming languages tend to converge to one another, i.e., features of one paradigm are appearing in languages based on another paradigm [23, 24]. This process can be observed primarily in the area of imperative, object-oriented, and functional programming languages. For example, C# or D are based on the imperative C language, extended with object-oriented and functional language elements. Another example is CLOS, which is a functional language based on Lisp, with added object-oriented features. Such languages are called *multiparadigm programming languages*.

The advantage of these languages is that the programmer may select the paradigm best suited for solving a particular problem. We may even choose to create different parts of the same program using techniques from different programming paradigms. If a small part of a large application requires high efficiency, imperative code is used, reusable types are created in an object-oriented way, while in some domains of computation, we choose the functional approach because solutions to problems in these domains can be expressed in a more succinct way compared to using imperative code.

I believe that multiparadigm languages can be a great help in the education of computer science. Instructors may show different versions of the same algorithm depending on the paradigm used to implement it, and later students may decide which paradigm to choose in a homework assignment, an exam, or at work, depending on the problem specification and the student's programming experience. Of course, we don't necessarily need to use multiparadigm languages for this purpose, we can also use multiple single-paradigm languages, but this way, we don't have to teach and students don't have to learn yet another language just for coding a particular algorithm.

During my research, I examined the possibility of using F# as a new multiparadigm programming language for coding different algorithms in the area of artificial intelligence (AI). I chose this area for three main reasons. First, I used to be an instructor of the seminars of the *Introduction to Artificial Intelligence* course at the University of Debrecen [28]. Second, AI and search algorithms, in particular, seemed to be a field of computation where we can make use of functional programming because some substantial parts of these algorithms (like, for example, checking operator preconditions) are essentially functional. And third, I had a couple of imperative and object-oriented

implementations of the algorithms in question, which proved to be a good starting point for creating the F# versions [13, 12].

The other main area of my research was trying to find an answer to the following question: "Why do most of our students majoring Software Information Technology BSc have a hard time fulfilling the requirements of most nonbasic courses?" Based on my ten years of teaching experience at the University of Debrecen, I can say that our students have to face a number of difficulties during their studies. I think these difficulties root from two main problems: students are unmotivated and cannot sense the coherence between the knowledge taught in the various courses. I found that we could alleviate both of these problems by assigning long-term projects to students, which they can work on throughout their studies, dealing with a particular aspect of the project in each course.

Programming contests may be another motivating factor. We have been organizing at least one in-house contest per semester for more than ten years now, and I can say that there are at least a couple of students whose interest is piqued by those contests. During these ten years, we developed two applications for evaluating the contestants' submissions on-line. The first one is called Programming Contest Result Manager (PCRM), and it is an e-mail-based console application, while the second one is called ProgCont, which is a web application with a client/server architecture. PCRM is described in detail in [14, 12], now I introduce ProgCont along with our experience with it.

To summarize the above, my thesis sets the following goals:

- to create different sample implementations of the most popular AI search algorithms in C# and F#, including those that compute the next move in a two-player game,
- to compare the different implementations of these algorithms (from purely object-oriented to mostly functional),
- to give some examples of how to create C# and F# implementations of specific problems and games that can be fitted into the above-mentioned algorithms,
- to argue for using multiparadigm programming languages in the teaching of artificial intelligence based on the above implementations,
- to design some sample long-term projects to be assigned to students as part of a new methodology for teaching computer science,
- to share our experience with the programming contests organized by the Faculty of Informatics at the University of Debrecen as well as the applications managing them.

Chapter 2 contains a brief overview of functional programming, the basics of lambda calculus, and the possible use of FP in the teaching of artificial intelligence. Chapters 3 and 4 provide a possible course guide for the practical course of the subject *Introduction to Artificial Intelligence*. In Chapter 3, I present a couple of implementations of search algorithms for single-agent, state-space-repesented problems, compare them with one another, and give the state-space representation and various implementations of two

such problems (a simple and a complex one) as examples. The structure of Chapter 4 is similar to that of Chapter 3, but instead of single-agent problems, it deals with two-player games and search algorithms on game trees. Chapter 5 is about some problems our students have to face with during their studies and a possible "cure" for these problems. In Chapter 6, I discuss programming contests, the way ACM contests are conducted, and a new application developed for managing ACM-like and other kinds of contests. Finally, I summarize the results of my thesis in Chapter 7.

CHAPTER 2

Overview of Functional Programming

Chapters 3 and 4 describe a possible course guide that can be used in the practical courses of *Introduction to Artificial Intelligence* or other AI-related subjects. Chapter 3 presents a couple of multiparadigm implementations of single-agent problems, while chapter 4 deals with two-player games. Before going into the details, let's discuss shortly the basics of functional programming and multiparadigm languages.

The functional programming (FP) paradigm was born together with the first functional programming language, IPL, in 1955, one year before Fortran. The second FP language, Lisp, was invented in 1958; its variants are still in use today. Despite the fact that many FP languages have been developed since 1955, FP remained a programming language primarily used only in academic areas until recently. One reason for this is the immediate success of the first two major imperative languages, Fortran and Cobol. These two languages and their derivatives (and thus, imperative paradigm) dominated business programming for more than 30 years, when object-oriented languages took the leading role. Today, however, the promise of FP is finally being realized as enterprises recognize the need for more sophisticated computing solutions.

In pure functional programming, programs consist of expressions, among which the most important are function definitions. The functions in a functional program are very much like mathematical functions—they accept arguments, return values, but they cannot have side effects, and because of this, do not change the state of the program. Instead of changing values, functions create new values by copying them and applying modifications to the copies. Of course, this is not always efficient during runtime, but allows programmers to use really neat programming constructs, like, for example, treating functions themselves as values. Once a value is no longer required, it is usually automatically disposed of by a garbage collector.

The main differences between imperative and functional programming are the following:

- With an imperative approach, the programmer writes step by step *how* a particular problem can be solved. In contrast, using a functional approach, the programmer only declares *what* the problem is by decomposing it to simple function calls.

- The *state* of an imperative program is an important factor, whereas a purely func-

tional program does not have states, because it uses only *immutable* data and special functional data structures for storing them. For example, *lists* are used instead of *arrays*.

- Because of stateless programming, the execution of a purely functional program does not have *side effects*. This also implies that the order in which the expressions are evaluated is not important; neither is the place of their occurrence in the source code. The program will yield the same output on the same input in any evaluation order and at any place of the program. This is called *referential transparency*.
- The main building blocks of an imperative program are *statements*, while a functional program consists almost exclusively of *expressions*. Some basic control structures (such as conditionals or loops), which are statements in an imperative language, may also appear in functional programs but only as expressions.
- In imperative programming, the primary flow controls are conditionals, loops, and function (or method) calls. In functional programming, only function calls exist as flow controls, including *recursion*, which plays a much more significant role than in imperative programs.

The biggest advantage of functional programming over imperative programming is that in most cases, FP requires much less and clearer code to achieve the same result than imperative programming because of language constructs of a higher abstraction level. Less code also means less chance of errors, less testing, and due to this, more productivity. Functional programs are less error-prone, can be more easily parallelized, and they can be developed in a shorter time. Because of these advantages, functional programming is becoming more and more popular nowadays; even software industry is looking for more and more programmers with expertise in functional programming.

2.1 The Lambda Calculus

Functional programming languages are based on a formal system for expressing computation, called the *Lambda Calculus* [4]. It was designed to formalize mathematics in terms of functions, variables, variable binding, and substitution. In this section, I present the most basic concepts of Lambda Calculus as defined by Alonzo Church in 1936 [2], and then give some examples of using these concepts in F#.

> We select a particular list of symbols $\{$, $\}$, $($, $)$, λ, $[$, $]$, and an enumerably infinite set of symbols $a, b, c, \ldots$ to be called *variables*. And we define the word *formula* to mean any finite sequence of symbols out of this list. The terms *well-formed formula*, *free variable*, and *bound variable* are then defined by induction as follows. A variable x standing alone is a well-formed formula and the occurrence of x in it is an occurrence of x as a free variable in it; if the formulas F and X are well-formed, $\{F\}(X)$ is well-formed, and an occurrence of x as a free (bound) variable in F or X is an occurrence of x as a free (bound) variable in $\{F\}(X)$; if the formula M is well-formed and contains an occurrence of x as a free variable

in M, then $\lambda x[M]$ is well-formed, any occurrence of x in $\lambda x[M]$ is an occurrence of x as a bound variable in $\lambda x[M]$, and an occurrence of a variable y, other than x, as a free (bound) variable in M is an occurrence of y as a free (bound) variable in $\lambda x[M]$.

When writing particular well-formed formulas, we adopt the following abbreviations. A formula $\{F\}(X)$ may be abbreviated as $F(X)$ in any case where F is or is represented by a single symbol. A formula $\{\{F\}(X)\}(Y)$ may be abbreviated as $\{F\}(X, Y)$, or, if F is or is represented by a single symbol, as $F(X, Y)$. And $\{\{\{F\}(X)\}(Y)\}(Z)$ may be abbreviated as $\{F\}(X, Y, Z)$, or as $F(X, Y, Z)$, and so on. A formula $\lambda x_1[\lambda x_2[\ldots \lambda x_n[M]\ldots]]$ may be abbreviated as $\lambda x_1 x_2 \ldots x_n \cdot M$ or as $\lambda x_1 x_2 \ldots x_n M$.

The expression $S^x_N M|$ is used to stand for the result of substituting N for x throughout M.

We consider the three following operations on well-formed formulas:

I. To replace any part $\lambda x[M]$ of a formula by $\lambda y[S^x_y M|]$, where y is a variable which does not occur in M.

II. To replace any part $\{\lambda x[M]\}(N)$ of a formula by $S^x_N M|$, provided that the bound variables in M are distinct both from x and from the free variables in N.

III. To replace any part $S^x_N M|$ (not immediately following λ) of a formula by $\{\lambda x[M]\}(N)$, provided that the bound variables in M are distinct both from x and from the free variables in N.

Any finite sequence of these operations is called a *conversion*, and if B is obtainable from A by a conversion, we say that A is *convertible* into B, or "$A \operatorname{conv} B$." If B is identical with A or is obtainable from A by a single application of one of the operations I, II, III, we say that A is *immediately convertible* into B.

A conversion which contains exactly one application of Operation II, and no application of Operation III, is called a *reduction*.

A formula is said to be *in normal form* if it is well-formed and contains no part of the form $\{\lambda x[M]\}(N)$. And B is said to be a *normal form of* A if B is in normal form and $A \operatorname{conv} B$.

The originally given order $a, b, c, \ldots$ of the variables is called their *natural order*. And a formula is said to be *in principal normal form* if it is in normal form, and no variable occurs in it both as a free variable and as a bound variable, and the variables which occur in it immediately following the symbol λ are, when taken in the order in which they occur in the formula, in natural order without repetitions, beginning with a and omitting only such variables as occur in the formula as free variables.[1] The formula B is said to be the *principal normal*

[1]For example, the formulas $\lambda ab \cdot b(a)$ and $\lambda a \cdot a(\lambda c \cdot b(c))$ are in principal normal form, and $\lambda ac \cdot c(a)$, and $\lambda bc \cdot c(b)$, and $\lambda a \cdot a(\lambda a \cdot b(a))$ are in normal form but not in principal normal form. Use of the principal normal form was suggested by S. C. Kleene as a means of avoiding the ambiguity of determination of the normal form of a formula, which is troublesome in certain connections.

form of A if B is in principal normal form and $A \operatorname{conv} B$.

For practical reasons, we can also consider operators (such as +) and constants (numbers, for example) as variables.[2] We can write the lambda expression $\lambda x \cdot +(x, 1)$ in F# as `fun x -> x + 1`. Here, we can consider the symbol `1` as the numeric constant 1, and the symbol `+` as the symbol of the arithmetic addition operator. This way, we created an anonymous function that takes a parameter (x) and returns $x + 1$. If we want to call this function, we can write the lambda expression $\{\lambda x \cdot +(x, 1)\}(3)$ or in F#, `(fun x -> x + 1) 3`. Using reduction, this lambda expression can be converted into $+(3, 1)$. If we now consider the + symbol as the arithmetic addition operator, which returns the sum of its two arguments, another reduction yields 4, which is the normal form of the expression $\{\lambda x \cdot +(x, 1)\}(3)$.

The definition of the lambda expression implies that each lambda function may only take exactly one parameter. For example, the expression $\lambda xy \cdot +(x, y)$ is equivalent to $\lambda x \cdot \lambda y \cdot +(x, y)$, or in F#, `fun x -> fun y -> x + y`. This function can then be called with two arguments, which will result in a number: the sum of the two arguments. But we can also call the function with only one argument, as in $\{\lambda x \cdot \lambda y \cdot +(x, y)\}(3)$ or in `(fun x -> fun y -> x + y) 3`. The result of this function call is another function: $\lambda y \cdot +(3, y)$ or `fun y -> 3 + y`. We call this *partial function application* or *currying*; and we call the form of functions in which they take exactly one argument, the *curried form of functions*. Strictly speaking, F# also uses curried functions, but we can simulate functions that take multiple arguments with the help of the *tuple* type. For example, the function `fun x -> fun y -> x + y` can be abbreviated as `fun x y -> x + y`, which means the same thing: a function that takes one argument and returns another function that takes one argument and returns a number. However, if we write `fun (x, y) -> x + y`, we get a function that takes one tuple argument (in this case, a pair of numbers) and returns a number. This function can only be called with exactly one argument of type tuple containing two numbers, as in `(fun (x, y) -> x + y) (3, 4)`. As this function does not return another function, we cannot use partial application here.

Currying is possible because functions are treated as values in functional programming. This means that a function can be an argument or the return value of another function. Functions that take other functions as parameters or return functions are called *higher-order functions*. For example, $\lambda fxy \cdot f(x, y)$, or in F#, `fun f x y -> f x y` is a function that takes another function as its first parameter and applies it to its second and third parameters. We can call this function like this: $\{\lambda fxy \cdot f(x, y)\}(\lambda xy \cdot +(x, y), 3, 4)$, or in F#, `(fun f x y -> f x y) (+) 3 4`, which both yield the sum of 3 and 4. If we now replace the argument function with `(-)`, we will get the difference of 3 and 4.

Lambda Calculus is only the pure mathematical background of FP. Programming, however, is impure in this sense—there are a lot of functional programming techniques that make FP really usable and powerful in some computing domains. Such techniques include *type inference*, *closures*, *continuations*, *monads*, just to mention but a few.

[2] In programming language context, variables are called *values* because they never change.

2.2 Functional Approach in Teaching Artificial Intelligence

Teaching search algorithms to our students is a great pedagogical challenge. At our university, they first meet artificial intelligence during the course *Introduction to Artificial Intelligence*, which is one of the core subjects of our three main undergraduate programs, Software Information Technology, Business Information Technology, and Engineering Information Technology. In the lectures, the pseudocode of these algorithms is presented, together with some examples, but this is not always enough for students to understand what is going on behind the scenes. In the seminars, the instructors show the same algorithms written in a high-level programming language, which used to be Pascal and C, but nowadays we use Java and C#. However, the high-level language code is merely another representation of the pseudocode, so students with little programming background do not find it useful in understanding the operation of the algorithms. The idea is that we should try presenting the search algorithms also by using a very different approach, namely, functional programming. A functional program can hide the unimportant steps of searching, and focuses only on the problem itself. It may be useful even if students do not have any former knowledge of functional programming, because a functional program is just another way for describing a problem, and not for solving it (although the problem description usually incorporates at least parts of the solution).

Besides coding in an object-oriented language, I also propose using the functional approach for programming the solutions of state-space-represented problems for the following reasons:

- These are complex problems. We do not teach programming in the frame of this course anymore; instead, we teach how the previously learned programming knowledge can be combined with the theory of search algorithms. The more complex a problem is, the more elegantly it can be implemented using functional programming.
- Some parts of the AI search algorithms are functional by their very nature. The source code of these parts simply looks better in a functional language.
- It is worth implementing a couple of problems and search algorithms with both paradigms so that students can see the difference between them. Later they can decide which approach to use in their homework or during a test.
- Functional programming is an exciting challenge for the students, and challenge can be a great motivating force. They prefer dealing with challenging problems even if those problems are difficult or abstract.

As a proof of the succinctness of a functional program solving an AI problem, I present a short C code and a purely functional F# code of the solution of the well-known n-queens puzzle. Here is the C code first:

```
1   #include <stdio.h>
2   #include <stdlib.h>
```

```
3
4   typedef enum {FALSE, TRUE} BOOL;
5
6   #define N 8
7
8   int board[N + 1];
9
10  void print_array()
11  {
12    int i;
13    for (i = 1; i <= N; ++i)
14      printf("%d ", board[i]);
15    putchar('\n');
16  }
17
18  BOOL conflicting(int col, int row)
19  {
20    int i;
21    for (i = 1; i < col; ++i)
22      if (board[i] == row || col - i == abs(row - board[i]))
23        return TRUE;
24    return FALSE;
25  }
26
27  void find_solutions(int col)
28  {
29    static int num = 0;
30    if (col > N)
31    {
32      printf("Solution #%02d: ", ++num);
33      print_array();
34    }
35    else
36    {
37      int row;
38      for (row = 1; row <= N; ++row)
39        if (!conflicting(col, row))
40        {
41          board[col] = row;
42          find_solutions(col + 1);
43        }
44    }
45  }
46
47  int main()
48  {
49    find_solutions(1);
50    return EXIT_SUCCESS;
51  }
```

And here is the F# code:

```
1  let N = 8
2
3  let conflicting col row (queen : int list) =
4      let rec checkCol c =
5          c < col &&
6              let r = queen.[c]
7              r = row || col - c = abs (row - r) || checkCol (c + 1)
8      checkCol 0
9
10 let nextCol col newSolutions solution =
11     seq {1 .. N}
12         |> Seq.filter (fun row -> not (conflicting col row solution))
13         |> Seq.fold (fun solutions row ->
14             solutions @ [solution @ [row]]) newSolutions
15
16 let rec findSolutions col allSolutions =
17     if col = N then
18         allSolutions
19     else
20         findSolutions (col + 1)
21             (allSolutions |> List.fold (nextCol col) [])
22
23 findSolutions 0 [[]]
24     |> List.iteri (fun i solution ->
25         printfn "Solution #%02d: %A" (i + 1) solution)
```

Both programs find the 92 possible solutions of the 8-queens problem, although they are not fully equivalent. The C code uses recursive backtracking, while the F# code is more like an optimized recursive breadth-first search, in which most of the work is done by built-in functions such as `Seq.fold`.

The C code works the following way: It takes a one-dimensional array of `N` elements (actually $\texttt{N}+1$ so we do not have to bother with the zero index), and calls the recursive function `find_solutions`, which tries to find an appropriate (nonconflicting) row for a queen in the next column of the table in a `for` loop. The index of the next column is stored in `col`, which is 1 at the beginning. If there is no such row, a backtracking is performed, i.e., `find_solutions` returns to its previous instance in the call stack (where the value of `col` was one less than its current value), and so it tries to find the next good row in the previous column inside the `for` loop. While we can find a good place for a queen in the current column, we continue calling `find_solutions` with an incremented `col` value. When `col` reaches $\texttt{N}+1$, all `N` queens have been placed on the table, i.e., a solution is found. We print the solution, and continue with the search by backtracking (i.e., returning to the previous function in the call stack) until we return to the main function, which means that a backtracking was performed from the initial state.

The F# code uses a list of lists to store all the solutions. Each of the inner lists will finally contain `N` numbers with the row values for each column just like the array

in the C version. At the beginning, we start with a one-element list of an empty list (`[[]]`), and then try to place a queen to all the possible rows in the column indicated by `col`. For example, when `col` is 0 and `allSolutions` is `[[]]`, the result of the expression `allSolutions |> List.fold (nextCol col) []` will be a list of 8 one-element list containing the numbers 1 to 8: `[[1], [2], [3], ...]`. After this, the `findSolutions` function is called recursively with an incremented `col` value, which results in a list containing only 2-element lists with all the possible layouts of two queens in two columns. This is repeated until `col` reaches `N`, when the resulting list will contain all possible solutions.

Of course, we could also have written here the recursive breadth-first search in C. The reason I chose backtracking instead is that it is much shorter because in backtracking, we only have to store the current path, and it is done for us by the call stack of the `find_solutions` function. In breadth-first search, however, we would have to keep track of the partial solutions (those in which one less columns are already filled than we are currently dealing with), which would require us to handle some kind of data structure (a linked list, for example). The F# version does this with the built-in `list` data type. Just for comparison, here is the C# implementation of the recursive breadth-first search, which resembles the most the F# program in functionality (C# also has a built-in `List` data type):

```
 1  using System;
 2  using System.Text;
 3  using System.Collections.Generic;
 4
 5  class Board
 6  {
 7    public static int N = 8;
 8
 9    private int[] board;
10    private int col;
11
12    public Board()
13    {
14      board = new int[N + 1];
15      col = 1;
16    }
17
18    public Board(Board parent, int row)
19    {
20      board = (int[])parent.board.Clone();
21      board[parent.col] = row;
22      col = parent.col + 1;
23    }
24
25    public bool Conflicting(int row)
26    {
27      for (int i = 1; i < col; ++i)
28        if (board[i] == row || col - i == Math.Abs(row - board[i]))
```

```
29          return true;
30        return false;
31      }
32
33      public override string ToString()
34      {
35        StringBuilder sb = new StringBuilder();
36        for (int i = 1; i <= N; ++i)
37          sb.Append(board[i] + " ");
38        return sb.ToString();
39      }
40    }
41
42    class Program
43    {
44      static List<Board> findSolutions(int col, List<Board> allSolutions)
45      {
46        if (col > Board.N)
47          return allSolutions;
48        List<Board> newSolutions = new List<Board>();
49        foreach (Board solution in allSolutions)
50          for (int row = 1; row <= Board.N; ++row)
51            if (!solution.Conflicting(row))
52              newSolutions.Add(new Board(solution, row));
53        return findSolutions(col + 1, newSolutions);
54      }
55
56      static void Main()
57      {
58        int num = 0;
59        List<Board> initialList = new List<Board>();
60        initialList.Add(new Board());
61        foreach (Board solution in findSolutions(1, initialList))
62          Console.WriteLine("Solution #{0:00}: {1}", ++num, solution);
63      }
64    }
```

As you can see, the C# code is still much longer and, in my opinion, less expressive than the F# version of the very same algorithm.

2.2.1 Search Algorithms in Different Programming Languages

The first implementations of AI search algorithms were programmed using the first popular high-level imperative programming language, Fortran, and the first functional programming language, IPL. The General Problem Solver, created in 1959, was able to solve theoretically any formalized symbolic problems [17]. Later, as newer and newer imperative programming languages (such as C or Pascal) dominated business computing, search algorithms were rewritten in a number of imperative languages.

With the appearance of object-oriented paradigm, programmers had the possibility to easily create more abstract and general implementations of these algorithms.

Meanwhile, functional programming languages were undergoing vigorous development too. Lisp, for example, was invented in 1958, and its variants (Common Lisp, Scheme, and Clojure among others) are still in use today. As an example, there is a Lisp implementation for solving the "farmer, wolf, goat, and cabbage problem" in [16]. Another popular functional language is Haskell, which was used for implementing depth-first search in [11].

Prolog, designed specifically for logic programming in 1972, is a natural choice when it comes to programming AI search algorithms. If we would like to create the most concise implementation, we should probably use Prolog.

2.3 Multiparadigm Languages and F#

As I already mentioned, imperative programming dominated the first few decades of commercial computing. The problem with imperative languages lies in their verboseness. Even a very simple algorithm can take a lot of lines of code to implement. On the other hand, functional and logic programming languages require programmers to acquire a very special way of thinking about things, which may be appropriate for some sorts of real-world problems, but is unnatural for most problems. Some think it is a better way to combine the advantages of the different paradigms by merging their features in one programming language. Others say that this way we ruin the pure conceptual background of the language. In my opinion, we gain more than we lose with this multiparadigm approach. Nowadays, programming languages tend to converge to one another, i.e., functional features are appearing in imperative languages and vice versa (as in D, Python, C#, or F#). This way, developers may choose to do some kinds of computation functionally instead of the "traditional way." A good example of this is LINQ in C#.

My choice fell on F#, Microsoft's first truly functional programming language. F# was designed to be compatible with the .NET framework, including its object-oriented concepts. This is not a full compatibility, however, as some OO features would interfere with the functional part of the language. For example, the `protected` accessibility modifier might cause problems in lambda functions. F# is basically a special version of Objective Caml (OCaml), an object-oriented FP language, extended to support .NET interoperability. It combines all three major programming paradigms (imperative, object-oriented, and functional), so programmers may choose the most suitable way to solve a particular problem. They may write a program purely functionally, partly imperatively, using object-oriented tools, such as classes and objects, or by mixing any or all of these techniques [19, 26]. In F#, programmers may use object states, and this way, we do not have to write mystic code, for example, for handling complex data structures. Another drawback of pure functional programming is the inefficiency of the executable code: copying data requires more memory and more runtime than just performing a small modification of existing data. Additionally, F# can help students realize that no single programming paradigm is best for everything. These are the

reasons why F# seemed to be a good choice to implement the search algorithms, which we already had at our disposal in Java and C# [13, 12].

CHAPTER 3

Some Implementations of Search Algorithms for Single-Agent Problems

In this chapter, I present various implementations of AI search algorithms for solving arbitrary single-agent, state-space-represented problems. The implemented algorithms are the following:

- backtracking (with optional cycle check and depth bound check)
- branch-and-bound algorithm (with optional cycle check and initial cost bound)
- breadth-first search
- depth-first search
- Dijkstra's algorithm (uniform-cost search) [6]
- best-first search
- A algorithm

These algorithms and their derivatives are the most widespread in practice nowadays [8, 22, 9], and these algorithms form the core part of the subject *Introduction to Artificial Intelligence*—both in the lectures and in the seminars [28]. In the code listings, I will only present two of the above-listed algorithms throughout this chapter: backtracking and A algorithm.

First, I introduce the class hierarchy created by Kósa Márk and myself [13] when we started to use Java in the seminars of the aforementioned subject for coding the search algorithms. Next, I present a pure object-oriented C# implementation and three multiparadigm F# implementations of these algorithms, comparing them with one another from various aspects and emphasizing the main differences between them. Finally, I show some possible F# implementations of a couple of examples of specific problems that can be solved using the presented algorithms.

3.1 A Class Hierarchy for Search Algorithms

This section covers the classes and their members related to the state-space representation in an abstract level and three of the listed search algorithms with the help of a UML class diagram. Figure 3.1 shows two abstract classes for the state-space representation itself (`State` and `Operator`) as well as four `Node` classes, which represent the graph nodes used by the search algorithms:

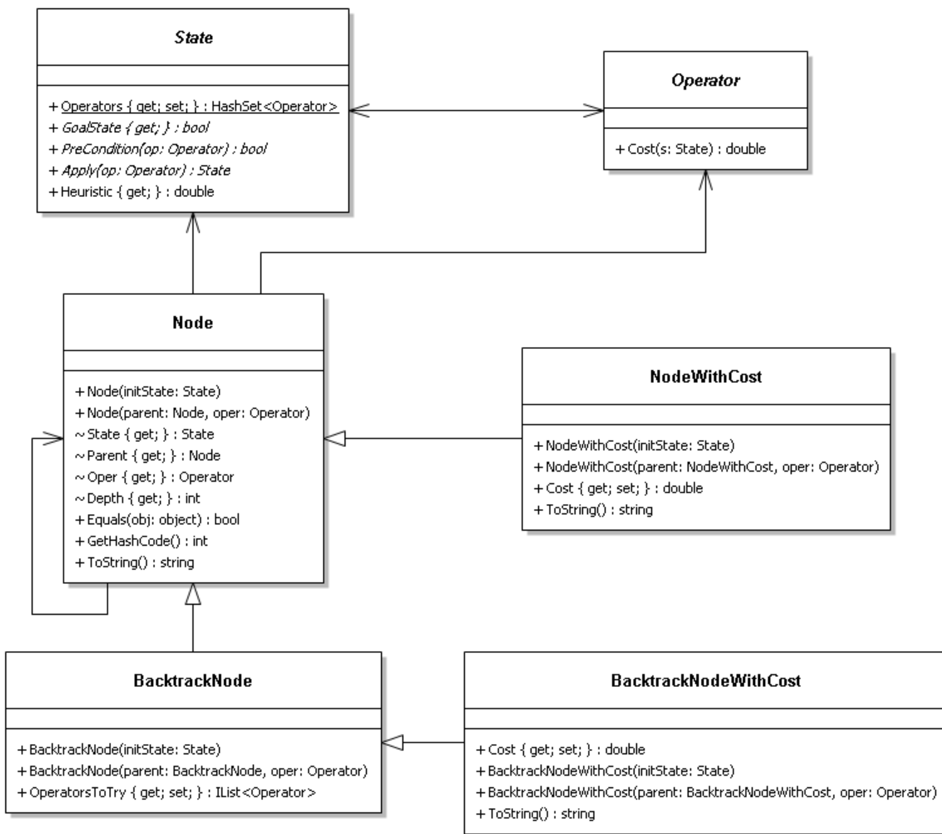


Figure 3.1. Classes representing the state space and the graph nodes.

`State` is used as a base class for the classes representing the states of concrete problems, while the `Operator` class serves as a base class for the concrete operators, which transform our problem from one state to another. The members of these classes are the following:

- `Operators`: the set of all operators relevant to the problem.
- `GoalState`: true if the current state is a goal state.

- `PreCondition`: true if the argument operator is applicable to the current state.
- `Apply`: applies the argument operator to the current state and returns the resulting state.
- `Heuristic`: a heuristic value that is an estimation of the cost of reaching the nearest goal state from the current state.
- `Cost`: the cost of applying the current operator to the argument state.

The four `Node` classes contain the following important members:

- `State`: the state represented by the node.
- `Parent`: the node to which an operator was applied to reach the current node.
- `Oper`: the operator that was applied to the parent node.
- `Depth`: the current node's depth in the spanning tree of the graph.
- `Cost`: the total cost of reaching the current node from the start node.
- `OperatorsToTry`: a list of operators applicable to the current node and not tried yet.

`Node` is used with non-cost-based graph search algorithms (e.g., breadth-first search), `NodeWithCost` is used with cost-based graph search algorithms (e.g., A algorithm), `BacktrackNode` is used with backtracking, and `BacktrackNodeWithCost` is used with the branch-and-bound algorithm.

Figure 3.2 shows another part of the UML class diagram, which contains the classes representing some of the search algorithms and their relations to other classes.

Here you can see two enumeration types. `SearchProp` contains some flags which control the operation of the search algorithms. If `AllSolutionsFlag` is set, the algorithm will search for all solutions, otherwise it will stop when the first solution is found. If `SolutionIsStateFlag` is set, the algorithm considers the goal state as the solution, otherwise the solution is considered to be the operator sequence leading from the initial state to the goal state. `CycleCheckFlag` is used only with backtracking and branch-and-bound search. If it is set, the algorithm will check for cycles in the current path during the search, otherwise it may enter an infinite loop. `Verbosity` contains three verbosity levels which control the amount of information printed to the output during the search. The caller may pass as an argument any combination of the search property flags as well as one of the verbosity levels to the constructor of a particular search algorithm.

`SearchAlg` is an abstract class which is the base of all search algorithms and contains the following members:

- `AllSolutions` and `SolutionIsState` are two logical values which are relevant to all search algorithms. They provide easy access to two of the flags so that we do not have to mask the flags argument every time they are needed.

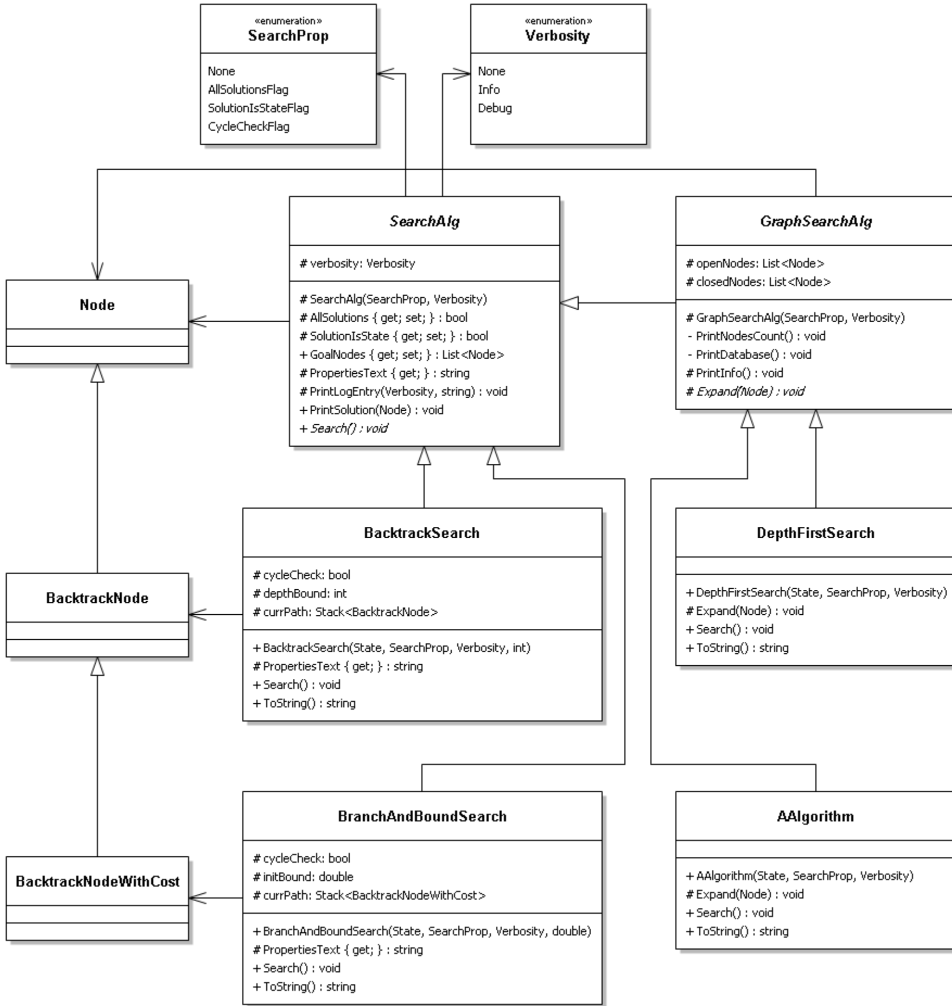


Figure 3.2. Classes representing some search algorithms.

- `GoalNodes`: a list of the goal nodes found during the search.
- `PropertiesText`: a string representation of the search properties.
- `PrintLogEntry`: prints the given text to the output if the verbosity level of the search algorithm is greater than or equal to the given level.
- `PrintSolution`: prints the solution taken as an argument to the output.
- `Search`: an abstract method that does the actual work; it must be overridden by the concrete search algorithms.

In addition to these members, `BacktrackSearch` and `BranchAndBoundSearch` also store the current path as a stack of nodes, while `GraphSearchAlg` stores the open and closed nodes as lists of nodes. `GraphSearchAlg` contains an abstract `Expand` method, which must be overridden by the concrete graph search algorithms.

3.2 Various Implementations of Search Algorithms

In this section, I show four possible implementations of AI search algorithms: a purely object-oriented and three multiparadigm versions with different amount of imperative code in them.

3.2.1 The C# Implementation

The following code contains no functional elements—it served as a starting point for the F# implementations. It consists of the classes introduced in Section 3.1. The presented code is not complete; it is listed here mainly for serving as a base for comparison, so I won't go into details explaining it. You can find a full explanation in [13, 12].

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace StateSpace
5   {
6     public abstract class State
7     {
8       public static ICollection<Operator> Operators { get; set; }
9       public abstract bool GoalState { get; }
10      public abstract bool PreCondition(Operator op);
11      public abstract State Apply(Operator op);
12      public virtual double Heuristic
13      {
14        get { return 0; }
15      }
16    }
17
18    public abstract class Operator
19    {
20      public virtual double Cost(State state)
21      {
22        return 1;
23      }
24    }
25
26    public class InvalidOperatorException : Exception
27    {
28      public InvalidOperatorException()
29        : base("No such operator!")
```

```
30          {
31          }
32        }
33      }
```

This part of the code defines the two abstract classes (`State` and `Operator`) required for the state-space representation itself and the `InvalidOperatorException` class, which will be thrown by the concrete problems in their `PreCondition` and `Apply` methods if they are given an `Operator` object that is not relevant to that particular problem. The two abstract classes will be inherited and their abstract methods will be implemented by specific problems (see Section 3.3). Note that there is a default implementation of the `Heuristic` property so that we can use heuristic search algorithms (e.g., best-first search) even with states which do not override this property. Similarly, we gave a default implementation for the `Cost` method of the `Operator` class, this way ensuring that any operator may participate in a cost-based search (e.g., Dijkstra's algorithm).

```
1     using System.Collections.Generic;
2     using StateSpace;
3
4     namespace SearchAlg
5     {
6       public class Node
7       {
8         internal State State { get; set; }
9         internal Node Parent { get; set; }
10        internal Operator Oper { get; set; }
11        internal int Depth { get; set; }
12
13        public Node(State initState)
14        {
15          State = initState;
16          Parent = null;
17          Oper = null;
18          Depth = 0;
19        }
20
21        public Node(Node parent, Operator oper)
22        {
23          State = parent.State.Apply(oper);
24          Parent = parent;
25          Oper = oper;
26          Depth = parent.Depth + 1;
27        }
28
29        public override bool Equals(object obj)
30        {
31          return obj is Node && State.Equals(((Node)obj).State);
32        }
```

```
33
34      public override int GetHashCode()
35      {
36        return 0;
37      }
38
39      public override string ToString()
40      {
41        string s = string.Format("{0}{1} (depth={2}",
42          Oper == null ? "" : Oper + " => ", State, Depth);
43        System.Reflection.PropertyInfo heurProp =
44          State.GetType().GetProperty("Heuristic");
45        if (heurProp.DeclaringType == heurProp.ReflectedType)
46          return s + string.Format(", heuristic={0})", State.Heuristic);
47        else
48          return s + ")";
49      }
50    }
51
52    public class NodeWithCost : Node
53    {
54      public double Cost { get; private set; }
55
56      public NodeWithCost(State initState)
57        : base(initState)
58      {
59        Cost = 0;
60      }
61
62      public NodeWithCost(NodeWithCost parent, Operator oper)
63        : base(parent, oper)
64      {
65        Cost = parent.Cost + oper.Cost(parent.State);
66      }
67
68      public override string ToString()
69      {
70        return base.ToString() + ", cost=" + Cost;
71      }
72    }
73
74    public class BacktrackNode : Node
75    {
76      public IList<Operator> OperatorsToTry { get; private set; }
77
78      private void Init()
79      {
80        OperatorsToTry = new List<Operator>();
81        foreach (Operator op in State.Operators)
```

```
82            if (State.PreCondition(op))
83              OperatorsToTry.Add(op);
84        }
85
86        public BacktrackNode(State initState)
87          : base(initState)
88        {
89          Init();
90        }
91
92        public BacktrackNode(BacktrackNode parent, Operator oper)
93          : base(parent, oper)
94        {
95          Init();
96        }
97      }
98
99      public class BacktrackNodeWithCost : BacktrackNode
100     {
101       public double Cost { get; private set; }
102
103       public BacktrackNodeWithCost(State initState)
104         : base(initState)
105       {
106         Cost = 0;
107       }
108
109       public BacktrackNodeWithCost(BacktrackNodeWithCost parent,
110                                    Operator oper)
111         : base(parent, oper)
112       {
113         Cost = parent.Cost + oper.Cost(parent.State);
114       }
115
116       public override string ToString()
117       {
118         return base.ToString() + ", cost=" + Cost;
119       }
120     }
121   }
```

These are the four `Node` classes used by the different search algorithms. There are three interesting things to note in this code:

- Each of these classes has two constructors: one for creating the start node and one for creating a child node from a parent node during an operator application.
- It is very important to override the `Equals` method this way because the search algorithms will lookup nodes in different data structures (stacks or lists), and it is

crucial for them to find a node with the same state, regardless of its parent or its depth in the spanning tree.

- The `ToString` method uses reflection to determine whether the concrete problem has overridden the default implementation of the `Heuristic` property of the `State` class. If so, it's value is added to the end of the string representation of the state.

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace SearchAlg
5   {
6     [Flags]
7     public enum SearchProp : byte
8     {
9       None                = 0,
10      AllSolutionsFlag    = 1,
11      SolutionIsStateFlag = 2,
12      CycleCheckFlag      = 4
13    }
14
15    public enum Verbosity : byte
16    {
17      None, Info, Debug
18    }
19
20    public abstract class SearchAlg
21    {
22      protected Verbosity verbosity;
23      protected bool AllSolutions { get; private set; }
24      protected bool SolutionIsState { get; private set; }
25      public IList<Node> GoalNodes { get; private set; }
26
27      protected SearchAlg(SearchProp properties = SearchProp.None,
28                          Verbosity verbosity = Verbosity.Info)
29      {
30        this.verbosity = verbosity;
31        AllSolutions = (properties & SearchProp.AllSolutionsFlag) !=
32          SearchProp.None;
33        SolutionIsState = (properties & SearchProp.SolutionIsStateFlag) !=
34          SearchProp.None;
35        GoalNodes = new List<Node>();
36      }
37
38      protected virtual string PropertiesText
39      {
40        get
41        {
42          return (AllSolutions ? "Searching for all solutions.\n" :
```

```
43                                  "Searching for the first solution.\n") +
44              (SolutionIsState ? "The goal state" :
45                "The operator sequence leading to the goal state") +
46              " is considered to be the solution.\n" +
47              "Verbosity level: " + verbosity + "\n";
48          }
49        }
50
51        protected void PrintLogEntry(Verbosity minLevel, string entry)
52        {
53          if (verbosity >= minLevel)
54            Console.WriteLine(entry);
55        }
56
57        public void PrintSolution(Node node)
58        {
59          if (SolutionIsState)
60            try
61            {
62              Console.WriteLine(node.State);
63            }
64            catch (NullReferenceException)
65            {
66              Console.WriteLine("Null as a solution???");
67            }
68          else if (node != null)
69          {
70            PrintSolution(node.Parent);
71            Console.WriteLine(node);
72          }
73        }
74
75        public abstract void Search();
76      }
77    }
```

Some issues worth noting here are the following:

- The `Flags` attribute is applied to the `SearchProp` enumeration type to denote that the listed properties may occur in combination. Their numeric values are the powers of 2 for the same reason.
- The constructor of `SearchAlg` clears all search properties by default and sets the default value of the verbosity level to `Info` unless the caller gives arguments to the constructor. Furthermore, an empty list is created for the goal nodes.
- The virtual property `PropertiesText` initially contains the string representation of those two search properties (`AllSolutions` and `SolutionIsState`) that are common to all search algorithms, as well as the verbosity level. It may be overridden by the concrete algorithm classes to extend it with further properties.

- The `PrintSolution` method prints just the goal state represented by the goal node given as a parameter if the `SolutionIsState` search property is set. Otherwise, it recursively prints a sequence of operator/state pairs from the start node to the goal node.

```
1  using System;
2  using System.Collections.Generic;
3  using StateSpace;
4
5  namespace SearchAlg
6  {
7    public class InvalidBoundException : Exception
8    {
9      public InvalidBoundException()
10       : base("Negative bound!")
11     {
12     }
13   }
14
15   public class BacktrackSearch : SearchAlg
16   {
17     protected bool cycleCheck;
18     protected int depthBound;
19     protected Stack<BacktrackNode> currPath;
20
21     public BacktrackSearch(State initState,
22                            SearchProp properties = SearchProp.None,
23                            Verbosity verbosity = Verbosity.Info,
24                            int depthBound = 0)
25       : base(properties, verbosity)
26     {
27       if (depthBound < 0)
28         throw new InvalidBoundException();
29       this.depthBound = depthBound;
30       cycleCheck = (properties & SearchProp.CycleCheckFlag) !=
31         SearchProp.None;
32       currPath = new Stack<BacktrackNode>();
33       currPath.Push(new BacktrackNode(initState));
34     }
35
36     protected override string PropertiesText
37     {
38       get
39       {
40         return base.PropertiesText +
41           (cycleCheck ? "Cycle check is on.\n" :
42                         "Cycle check is off.\n") +
43           (depthBound > 0 ? "Depth bound: " + depthBound + "\n" :
44                             "Depth bound check is off.\n");
```

```
45        }
46      }
47
48      public override void Search()
49      {
50        while (currPath.Count > 0)
51        {
52          BacktrackNode currNode = currPath.Peek();
53          string depthText = depthBound > 0 ?
54            " (depth=" + currNode.Depth + ")" : "";
55          PrintLogEntry(Verbosity.Debug,
56            "Current state: " + currNode.State + depthText);
57          if (currNode.State.GoalState)
58          {
59            if (!(SolutionIsState && GoalNodes.Contains(currNode)))
60              GoalNodes.Add(currNode);
61            if (AllSolutions)
62            {
63              PrintLogEntry(Verbosity.Info,
64                "Found a solution, backtracking.");
65              currPath.Pop();
66              continue;
67            }
68            else
69              break;
70          }
71          if (depthBound > 0 && currNode.Depth == depthBound)
72          {
73            PrintLogEntry(Verbosity.Info,
74              "Reached depth bound, backtracking.");
75            currPath.Pop();
76            continue;
77          }
78          if (currNode.OperatorsToTry.Count == 0)
79          {
80            PrintLogEntry(Verbosity.Info,
81              "No more applicable operators, backtracking.");
82            currPath.Pop();
83            continue;
84          }
85          Operator op = currNode.OperatorsToTry[0];
86          PrintLogEntry(Verbosity.Debug, "Applying operator: " + op);
87          BacktrackNode newNode = new BacktrackNode(currNode, op);
88          PrintLogEntry(Verbosity.Debug, "New state: " + newNode.State);
89          if (cycleCheck && currPath.Contains(newNode))
90            PrintLogEntry(Verbosity.Info, "Found a cycle.");
91          else
92            currPath.Push(newNode);
93          currNode.OperatorsToTry.Remove(op);
```

```
 94          }
 95        }
 96
 97        public override string ToString()
 98        {
 99          return verbosity == Verbosity.None ? "" :
100            "Searching using backtracking.\n" + PropertiesText;
101        }
102      }
103    }
```

This is a possible C# implementation of the backtracking algorithm with optional cycle check and depth bound check. First, `InvalidBoundException` is defined, which is thrown if a negative bound is given to the constructor of `BacktrackSearch`. The algorithm itself is written to be as close to the pseudocode presented in the lectures as possible. It starts in the constructor with creating an empty stack of nodes (actually `BacktrackNodes`) for storing the current path (`currPath`) and pushing the start node onto it. Creating a new `BacktrackNode` involves generating a list of operators applicable to that node (`OperatorsToTry`).

The control flow of the `Search` method is a bit complicated because of the different search properties we have to consider. The main control structure is a `while` loop that runs until the current path becomes empty, which occurs when a backtrack operation is performed in the start node. First, the current node (the top element in the stack of the current path) is checked whether it contains a goal state. If so, it is added to the list of the goal nodes, and the algorithm will either stop (if only one solution is required), or continue with a backtrack operation. If the current node is not a goal node, then two other conditions are checked which imply backtracking: (1) the depth of the current node reaches the depth bound, and (2) there are no more applicable operators left in the current node. If neither of these conditions are true, the next applicable operator is applied to the current state, and the applied operator is removed from the list of the applicable operators. Finally, we check for a cycle if required and push the new node generated by the operator application to the stack of the current path.

```
 1  using System;
 2  using System.Collections.Generic;
 3  using StateSpace;
 4
 5  namespace SearchAlg
 6  {
 7    public abstract class GraphSearchAlg : SearchAlg
 8    {
 9      protected List<Node> openNodes, closedNodes;
10
11      protected GraphSearchAlg(SearchProp properties = SearchProp.None,
12                               Verbosity verbosity = Verbosity.Info)
13        : base(properties, verbosity)
14      {
```

```
15        openNodes = new List<Node>();
16        closedNodes = new List<Node>();
17      }
18
19      private void PrintNodesCount()
20      {
21        Console.WriteLine("Open nodes: {0}, closed nodes: {1}.",
22                          openNodes.Count, closedNodes.Count);
23      }
24
25      private void PrintDatabase()
26      {
27        Console.WriteLine("Open nodes:");
28        foreach (Node node in openNodes)
29          Console.WriteLine(node);
30        Console.WriteLine("Closed nodes:");
31        foreach (Node node in closedNodes)
32          Console.WriteLine(node);
33        Console.WriteLine();
34      }
35
36      protected void PrintInfo()
37      {
38        if (verbosity == Verbosity.Info)
39          PrintNodesCount();
40        else if (verbosity == Verbosity.Debug)
41          PrintDatabase();
42      }
43
44      protected abstract void Expand(Node node);
45    }
46
47    public class AAlgorithm : GraphSearchAlg
48    {
49      public AAlgorithm(State initState,
50                        SearchProp properties = SearchProp.None,
51                        Verbosity verbosity = Verbosity.Info )
52        : base(properties, verbosity)
53      {
54        openNodes.Add(new NodeWithCost(initState));
55      }
56
57      protected override void Expand(Node node)
58      {
59        foreach (Operator op in State.Operators)
60          if (node.State.PreCondition(op))
61          {
62            NodeWithCost newNode =
63              new NodeWithCost((NodeWithCost)node, op);
```

```
64            int index;
65            if ((index = openNodes.IndexOf(newNode)) != -1)
66            {
67              NodeWithCost oldNode = (NodeWithCost)openNodes[index];
68              if (newNode.Cost < oldNode.Cost)
69              {
70                openNodes.Remove(oldNode);
71                openNodes.Add(newNode);
72              }
73            }
74            else if ((index = closedNodes.IndexOf(newNode)) != -1)
75            {
76              NodeWithCost oldNode = (NodeWithCost)closedNodes[index];
77              if (newNode.Cost < oldNode.Cost)
78              {
79                closedNodes.Remove(oldNode);
80                openNodes.Add(newNode);
81              }
82            }
83            else
84              openNodes.Add(newNode);
85          }
86      }
87
88      public override void Search()
89      {
90        while (openNodes.Count > 0)
91        {
92          PrintInfo();
93          NodeWithCost currNode = (NodeWithCost)openNodes[0];
94          if (currNode.State.GoalState)
95          {
96            GoalNodes.Add(currNode);
97            if (AllSolutions)
98            {
99              PrintLogEntry(Verbosity.Info, "Found a solution.");
100             openNodes.Remove(currNode);
101             closedNodes.Add(currNode);
102             continue;
103           }
104           else
105             break;
106         }
107         openNodes.Remove(currNode);
108         closedNodes.Add(currNode);
109         Expand(currNode);
110         openNodes.Sort(new AAlgComparer());
111       }
112       PrintInfo();
```

```
113       }
114
115       public override string ToString()
116       {
117         return verbosity == Verbosity.None ? "" :
118           "Searching using A algorithm.\n" + PropertiesText;
119       }
120
121       class AAlgComparer : IComparer<Node>
122       {
123         int IComparer<Node>.Compare(Node n1, Node n2)
124         {
125           return (((NodeWithCost)n1).Cost + n1.State.Heuristic).CompareTo
126                  (((NodeWithCost)n2).Cost + n2.State.Heuristic);
127         }
128       }
129     }
130   }
```

This is the implementation of the abstract base class of all graph search algorithms (`GraphSearchAlg`) and one of the concrete algorithm classes (`AAlgorithm`). The abstract `GraphSearchAlg` class contains the database in the form of two lists of nodes (`openNodes` and `closedNodes`), which store the open nodes and closed nodes of the search tree, respectively. Both of these lists are initialized with empty lists at this point. There are also a couple of methods printing some information about the open and closed nodes for debugging purposes, as well as an abstract `Expand` method, which has to be overridden by the concrete algorithm classes.

A algorithm is one of the most popular search algorithms. It starts with adding the start node to the list of open nodes in the constructor. Instead of `Node`, `NodeWithCost` is used to store the nodes of the search tree because this algorithm also takes into account the cost of reaching the current node from the start node when choosing the open node to expand. The `Search` method consists again of a `while` loop that runs until there are no more open nodes in our database, i.e., the entire state-space graph has been discovered. We take the first node of the list of open nodes (with the lowest value of the evaluation function among the open nodes), check whether it is a goal node, and if so, add this node to the list of goal nodes. If only one solution is needed, the algorithm stops, otherwise the current node is moved to the closed nodes (without expansion), and the algorithm continues. In case the current node does not contain a goal state, it is expanded and moved to the closed nodes.

Expansion in A algorithm works as follows: Each operator that is applicable to the node is applied. If the newly created node contains a state that is already present in the database, we have to check whether this state is now reached with less cost than earlier. If so, the old node is replaced with the new one, even if the old node is a closed node, in which case it is also moved to the list of open nodes to reexpand it later. After expanding the current node, the new list of open nodes is sorted by the sum of their cost and heuristic values. A private class implementing the `IComparer` interface is used for this purpose.

The classes of the omitted search algorithms contain only minor modifications to those listed above. Branch-and-bound algorithm is very similar to backtracking, whereas graph search algorithms are much like simplified versions of A algorithm.

The Main Program

Finally, let's see how we could write a main program that calls one of these algorithms to find a solution to a specific problem. Suppose that we already have the implementation of the *Towers of Hanoi* problem at our disposal (see Section 3.3). A possible main program may then look like the following:

```
1   using System;
2   using SearchAlg;
3
4   class Program
5   {
6     static void Main()
7     {
8       SearchAlg.SearchAlg searchAlg =
9         new BacktrackSearch(new Hanoi.HanoiState(),
10        SearchProp.CycleCheckFlag | SearchProp.AllSolutionsFlag,
11        Verbosity.Debug, 10);
12      Console.WriteLine(searchAlg);
13      searchAlg.Search();
14      int num = 0;
15      foreach (Node solution in searchAlg.GoalNodes)
16      {
17        Console.WriteLine("\nSolution #{0}:", ++num);
18        searchAlg.PrintSolution(solution);
19      }
20      Console.WriteLine("\nNumber of solutions: " +
21        searchAlg.GoalNodes.Count);
22    }
23  }
```

As you can see, this sample main program will search for all solutions of the Towers of Hanoi problem using backtracking with cycle check and a depth bound of 10 and printing all debug information to the screen. After instantiating the `BacktrackSearch` class, the search properties are printed, and then the `Search` method is called, which will generate the solutions in the `GoalNodes` list. Finally, the solutions are printed one by one using a `foreach` loop, together with the number of solutions found.

3.2.2 The First F# Implementation

The aim of first F# version of these algorithms I present here was to be as close to the C# version as possible, considering the peculiarities of the F# language. This version uses almost the same class hierarchy as discussed above, with a number of imperative language constructs and mutable data structures. However, no mutable

variables or fields were required for this implementation. The most notable difference is the lack of jump statements (such as `break` or `continue`), which had to be replaced with tail-recursive function calls. Let's now examine the code in details.

`State` and `Operator` are two classes that are common to all three implementations. Here is the code which defines these classes:

```
1   namespace StateSpace
2
3   open System.Collections.Generic
4
5   type [<AbstractClass>] State() =
6       static let operators = HashSet<Operator>()
7       static member Operators = operators
8       abstract GoalState : bool
9       abstract PreCondition : Operator -> bool
10      abstract Apply : Operator -> State
11      abstract Heuristic : double
12      default this.Heuristic = 0.0
13
14  and [<AbstractClass>] Operator() =
15      abstract Cost : State -> double
16      default this.Cost(_) = 1.0
17
18  exception InvalidOperator
```

As I wanted to avoid mutable fields, an initial value had to be assigned to all nonabstract fields and properties. That is why `operators` is initialized with an empty `HashSet` already in the abstract `State` class (instead of the concrete class of a specific problem). Although this seems to be a restriction, no part of the code depends on the exact type of this collection.

```
1   namespace SearchAlg
2
3   open System
4   open System.Collections.Generic
5   open StateSpace
6
7   type Node =
8       val private state  : State
9       val private parent : Node option
10      val private oper   : Operator option
11      val private depth  : int
12
13      member internal this.State  = this.state
14      member internal this.Parent = this.parent
15      member internal this.Oper   = this.oper
16      member internal this.Depth  = this.depth
17
18      new(initState : State) =
```

```
19            { state = initState; parent = None; oper = None; depth = 0 }
20
21        new(parent : Node, oper : Operator) =
22            { state  = parent.state.Apply(oper)
23              parent = Some parent
24              oper   = Some oper
25              depth  = parent.depth + 1 }
26
27        override this.Equals(other) =
28            match other with
29                | :? Node as otherNode ->
30                    this.state.Equals(otherNode.state)
31                | _ ->
32                    false
33
34        override this.GetHashCode() =
35            hash this.state
36
37        override this.ToString() =
38            let s =
39                sprintf "%s%O (depth=%d"
40                    (if this.oper = None then "" else
41                         this.oper.Value.ToString() + " => ")
42                    this.state this.depth
43            let heurProp = this.state.GetType().GetProperty("Heuristic")
44            if heurProp.DeclaringType = heurProp.ReflectedType then
45                s + sprintf ", heuristic=%g)" this.state.Heuristic
46            else
47                s + ")"
48
49    type NodeWithCost =
50        inherit Node
51
52        val private cost : double
53        member this.Cost = this.cost
54
55        new(initState : State) =
56            { inherit Node(initState); cost = 0.0 }
57
58        new(parent : NodeWithCost, oper : Operator) =
59            { inherit Node(parent, oper);
60              cost = parent.cost + oper.Cost(parent.State) }
61
62        override this.ToString() =
63            base.ToString() + ", cost=" + this.cost.ToString()
```

The implementation of the two `Node` classes is almost the same as in the C# version. The only difference is that the `Parent` property is of type `Node option` and the `Oper` property is of type `Operator option`. I used F#'s `option` type to simulate C#'s

reference type. It is needed here because both properties may or may not have an object value; namely, the start node does not have a parent, and there is no operator used to reach it. As the `null` value cannot be assigned to `Parent` and `Oper` in F#, `None` is used instead.

Another interesting thing to note here is that there is no `BacktrackNode` or `BacktrackNodeWithCost` class. It is because of the different operation of the backtracking algorithm—I will cover it later in more detail.

```
1   namespace SearchAlg
2
3   open System
4   open System.Collections.Generic
5
6   [<Flags>]
7   type SearchProp =
8       | None               = 0b00000000
9       | AllSolutionsFlag   = 0b00000001
10      | SolutionIsStateFlag = 0b00000010
11      | CycleCheckFlag     = 0b00000100
12
13  type Verbosity =
14      | None  = 0
15      | Info  = 1
16      | Debug = 2
17
18  [<AbstractClass>]
19  type SearchAlg(?properties, ?verbosity) =
20      let properties = defaultArg properties SearchProp.None
21      let verbosity = defaultArg verbosity Verbosity.Info
22      let allSolutions =
23          properties &&& SearchProp.AllSolutionsFlag <> SearchProp.None
24      let solutionIsState =
25          properties &&& SearchProp.SolutionIsStateFlag <> SearchProp.None
26      let goalNodes = List<Node>()
27
28      member this.AllSolutions = allSolutions
29      member this.SolutionIsState = solutionIsState
30      member this.GoalNodes = goalNodes
31
32      abstract PropertiesText : string
33      default this.PropertiesText =
34          (if allSolutions then
35               "Searching for all solutions.\n"
36           else
37               "Searching for the first solution.\n") +
38          (if solutionIsState then
39               "The goal state"
40           else
41               "The operator sequence leading to the goal state") +
```

```
42              " is considered to be the solution.\n" +
43              "Verbosity level: " + verbosity.ToString() + "\n"
44
45          member this.PrintLogEntry(minLevel, entry) =
46              if verbosity >= minLevel then
47                  printfn "%s" entry
48
49          member this.PrintSolution(node : Node option) =
50              if solutionIsState then
51                  try
52                      printfn "%O" node.Value.State
53                  with
54                      :? NullReferenceException ->
55                          printfn "Null as a solution???"
56              elif node.IsSome then
57                  this.PrintSolution(node.Value.Parent)
58                  printfn "%O" node.Value
59
60          abstract Search : unit -> unit
```

There is only one minor difference here from the C# version: in the `PrintSolution` method, the `node` parameter is of type `Node option` instead of just `Node`. The reason for this is the same as for `Parent` being of type `Node option`.

```
1   namespace SearchAlg
2
3   open System.Collections.Generic
4   open StateSpace
5
6   exception InvalidBound
7
8   type BacktrackSearch(initState, ?properties, ?verbosity, ?depthBound) =
9       inherit SearchAlg(defaultArg properties SearchProp.None,
10                        defaultArg verbosity Verbosity.Info)
11
12      let properties = defaultArg properties SearchProp.None
13      let verbosity = defaultArg verbosity Verbosity.Info
14      let depthBound = defaultArg depthBound 0
15      let cycleCheck =
16          properties &&& SearchProp.CycleCheckFlag <> SearchProp.None
17      let currPath = Stack<Node>()
18
19      do
20          if depthBound < 0 then
21              raise InvalidBound
22          currPath.Push(Node(initState))
23
24      override this.PropertiesText =
25          base.PropertiesText +
```

```
26                (if cycleCheck then
27                     "Cycle check is on.\n"
28                 else
29                     "Cycle check is off.\n") +
30                (if depthBound > 0 then
31                     "Depth bound: " + depthBound.ToString() + "\n"
32                 else
33                     "Depth bound check is off.\n")
34
35        override this.Search() =
36            let currNode = currPath.Peek()
37            let depthText =
38                if depthBound > 0 then
39                    sprintf " (depth=%d)" currNode.Depth
40                else
41                    ""
42            if currNode.State.GoalState then
43                this.PrintLogEntry(Verbosity.Debug,
44                    sprintf "Current state: %O%s" currNode.State depthText)
45                if not (this.SolutionIsState &&
46                        this.GoalNodes.Contains(currNode)) then
47                    this.GoalNodes.Add(currNode)
48                if this.AllSolutions then
49                    this.PrintLogEntry(Verbosity.Info,
50                        "Found a solution, backtracking.")
51                    currPath.Pop() |> ignore
52            elif depthBound > 0 && currNode.Depth = depthBound then
53                this.PrintLogEntry(Verbosity.Debug,
54                    sprintf "Current state: %O%s" currNode.State depthText)
55                this.PrintLogEntry(Verbosity.Info,
56                    "Reached depth bound, backtracking.")
57                currPath.Pop() |> ignore
58            else
59                State.Operators
60                    |> Seq.filter (fun op -> currNode.State.PreCondition(op))
61                    |> Seq.takeWhile (fun _ ->
62                        this.AllSolutions || this.GoalNodes.Count = 0)
63                    |> Seq.iter (fun op ->
64                        this.PrintLogEntry(Verbosity.Debug,
65                            sprintf "Current state: %O%s"
66                                currNode.State depthText)
67                        this.PrintLogEntry(Verbosity.Debug,
68                            sprintf "Applying operator: %O" op)
69                        let newNode = Node(currNode, op )
70                        this.PrintLogEntry(Verbosity.Debug,
71                            sprintf "New state: %O" newNode.State)
72                        if cycleCheck && currPath.Contains(newNode) then
73                            this.PrintLogEntry(Verbosity.Info,
74                                "Found a cycle.")
```

```
75                      else
76                          currPath.Push(newNode)
77                          this.Search())
78              if this.AllSolutions || this.GoalNodes.Count = 0 then
79                  this.PrintLogEntry(Verbosity.Debug,
80                      sprintf "Current state: %0%s"
81                          currNode.State depthText)
82                  this.PrintLogEntry(Verbosity.Info,
83                      "No more applicable operators, backtracking.")
84                  currPath.Pop() |> ignore
85
86      override this.ToString() =
87          if verbosity = Verbosity.None then
88              ""
89          else
90              "Searching using backtracking.\n" + this.PropertiesText
```

The implementation of the backtracking algorithm has the biggest difference from that of the C# version. It would have made no sense to simulate the `while` loop with tail-recursive function calls as backtracking is recursive by itself, even if its pseudocode is iterative. When I first created the "simulated" version, I saw that it contained a lot of unnecessary overhead. First, each `continue` statement in the `Search` method had to be replaced with a recursive call, which is not required in the presented version. Second, we can get rid of the `OperatorsToTry` property of the `BacktrackNode` class because we apply each applicable operator one after the other inside the lambda function of the `Seq.iter` function in line 63. This way, there is no need to store the operators not tried yet, and since the `OperatorsToTry` property is the only difference between the `Node` and `BacktrackNode` classes, we can also get rid of the `BacktrackNode` class itself.

Let's see how the `Search` method works in this implementation. The first couple of steps are the same as in the C# version. We take the top element in the stack of the current path, check whether it contains a goal state, and if so, add it to the list of the goal nodes. If we are searching for all solutions, we pop this element from the stack, and simply return from the `Search` method to its previous instance in the call stack to line 77, where the next operator will be tried by the `Seq.iter` function. The same will happen if depth bound is reached. In all other cases, we take all operators relevant to the problem (line 59) and filter out those that are not applicable to the current state (line 60). If there are no applicable operators, this gives us an empty sequence, and the control will jump to line 78. Otherwise, we have to check if a solution has already been found and only one solution is required. If this condition is true, then again we can "jump" to line 78.

Then we iterate through all applicable operators using the `Seq.iter` function (line 63), whose argument lambda function applies the operator (line 69), checks if the new state occurs already in the current path (line 72), and if not, pushes the new node onto the current path (line 76) and calls the `Search` method recursively (line 77). After the control returns back here, the next operator in the sequence is processed. When all operators have been tried, then again a backtracking operation is performed by

popping the current node from the current path and returning to the caller. If there are no more instances of the `Search` method in the call stack, the search is finished.

```
1   namespace SearchAlg
2
3   open System.Collections.Generic
4   open StateSpace
5
6   [<AbstractClass>]
7   type GraphSearchAlg(?properties, ?verbosity) =
8       inherit SearchAlg(defaultArg properties SearchProp.None,
9                         defaultArg verbosity Verbosity.Info)
10
11      let verbosity = defaultArg verbosity Verbosity.Info
12      let openNodes = List<Node>()
13      let closedNodes = List<Node>()
14
15      member this.OpenNodes = openNodes
16      member this.ClosedNodes = closedNodes
17
18      member this.PrintInfo() =
19          let printNodesCount () =
20              printfn "Open nodes: %d, closed nodes: %d."
21                  this.OpenNodes.Count this.ClosedNodes.Count
22
23          let printDatabase () =
24              printfn "Open nodes:"
25              for node in openNodes do
26                  printfn "%O" node
27              printfn "Closed nodes:"
28              for node in closedNodes do
29                  printfn "%O" node
30              printfn ""
31
32          if verbosity = Verbosity.Info then
33              printNodesCount ()
34          elif verbosity = Verbosity.Debug then
35              printDatabase ()
36
37      abstract Expand : Node -> unit
38
39  type AAlgorithm(initState, ?properties, ?verbosity) as this =
40      inherit GraphSearchAlg(defaultArg properties SearchProp.None,
41                             defaultArg verbosity Verbosity.Info)
42
43      let verbosity = defaultArg verbosity Verbosity.Info
44
45      do this.OpenNodes.Add(NodeWithCost(initState))
46
```

```
47        override this.Expand(node) =
48            State.Operators
49                |> Seq.filter (fun op -> node.State.PreCondition(op))
50                |> Seq.iter (fun op ->
51                    let newNode = NodeWithCost(node :?> NodeWithCost, op)
52                    let index = this.OpenNodes.IndexOf(newNode)
53                    if index <> -1 then
54                        let oldNode = this.OpenNodes.[index] :?> NodeWithCost
55                        if newNode.Cost < oldNode.Cost then
56                            this.OpenNodes.Remove(oldNode) |> ignore
57                            this.OpenNodes.Add(newNode)
58                    else
59                        let index = this.ClosedNodes.IndexOf(newNode)
60                        if index <> -1 then
61                            let oldNode =
62                                this.ClosedNodes.[index] :?> NodeWithCost
63                            if newNode.Cost < oldNode.Cost then
64                                this.ClosedNodes.Remove(oldNode) |> ignore
65                                this.OpenNodes.Add(newNode)
66                        else
67                            this.OpenNodes.Add(newNode))
68
69        override this.Search() =
70            this.PrintInfo()
71            if this.OpenNodes.Count > 0 then
72                let currNode = this.OpenNodes.[0]
73                if currNode.State.GoalState then
74                    this.GoalNodes.Add(currNode)
75                    if this.AllSolutions then
76                        this.PrintLogEntry(Verbosity.Info,
77                            "Found a solution.")
78                        this.OpenNodes.Remove(currNode) |> ignore
79                        this.ClosedNodes.Add(currNode)
80                        this.Search()
81                else
82                    this.OpenNodes.Remove(currNode) |> ignore
83                    this.ClosedNodes.Add(currNode)
84                    this.Expand(currNode)
85                    this.OpenNodes.Sort({ new IComparer<Node> with
86                        member this.Compare(n1, n2) =
87                            let f1 = (n1 :?> NodeWithCost).Cost +
88                                        n1.State.Heuristic
89                            let f2 = (n2 :?> NodeWithCost).Cost +
90                                        n2.State.Heuristic
91                            f1.CompareTo(f2) })
92                    this.Search()
93
94        override this.ToString() =
95            if verbosity = Verbosity.None then
```

```
96              ""
97          else
98              "Searching using A algorithm.\n" + this.PropertiesText
```

The implementation of these classes differs from that of the C# version in the `Expand` and `Search` methods of the concrete algorithm class. First, instead of the `foreach` loop, the `Expand` method uses the `Seq.iter` higher-order function to apply all applicable operators to the argument node. Second, the `Search` method uses two tail-recursive calls to simulate the `while` loop. Moreover, to sort the list of open nodes, the `IComparer` object is created using an object expression, instead of instantiating a private class.

The Main Program

To achieve the same result as in the C# implementation, we can write the following main program (as a function):

```
1   open SearchAlg
2
3   let main () =
4       let searchAlg =
5           BacktrackSearch(Hanoi.HanoiState(),
6               SearchProp.CycleCheckFlag ||| SearchProp.AllSolutionsFlag,
7               Verbosity.Debug, 10)
8       printfn "%O" searchAlg
9       searchAlg.Search()
10      searchAlg.GoalNodes
11          |> Seq.iteri (fun i solution ->
12              printfn "\nSolution #%d:" (i + 1)
13              searchAlg.PrintSolution(Some solution))
14      printfn "\nNumber of solutions: %d" searchAlg.GoalNodes.Count
```

Apart from the syntax, this main program is just like that of the C# version. The difference is the same here as in the `Search` method: the solutions are printed using the `iteri` function from the `Seq` module, instead of a `foreach` loop.

3.2.3 The Second F# Implementation

After creating the first F# implementation of the search algorithms, we can realize that it is actually not a functional code, even though it contains some functional elements, such as using sequences and higher-order functions from the `Seq` module to simulate `foreach` loops. The first version can be used as a starting point for students who are not familiar with functional programming but are familiar with object-oriented programming.

So, the obvious next step is to find a way to make the code more functional. One possibility for this is to get rid of classes which do not hold data (or do not hold *much* data) but are used only to provide some functionality, and to replace such classes with functions that return an object implementing the abstract methods (or overriding the

concrete methods) of a "base class" or interface. In our case, such classes are those representing the specific search algorithms, such as `BacktrackSearch` or `AAlgorithm`. We can, for example, substitute `BacktrackSearch` with a function which takes the same parameters as the constructor of the `BacktrackSearch` class, creates a `SearchAlg` object using an object expression, and returns this object:

```
1   module Backtrack
2
3   open System.Collections.Generic
4   open StateSpace
5   open SearchAlg
6
7   exception InvalidBound
8
9   let backtrackSearch initState properties verbosity depthBound =
10      let properties = defaultArg properties SearchProp.None
11      let verbosity = defaultArg verbosity Verbosity.Info
12      let depthBound = defaultArg depthBound 0
13      if depthBound < 0 then
14          raise InvalidBound
15      let cycleCheck =
16          properties &&& SearchProp.CycleCheckFlag <> SearchProp.None
17      let currPath = Stack<Node>()
18      let backtrack =
19          { new SearchAlg(properties, verbosity) with
20              override this.PropertiesText =
21                  base.PropertiesText +
22                      (if cycleCheck then
23                           "Cycle check is on.\n"
24                       else
25                           "Cycle check is off.\n") +
26                      (if depthBound > 0 then
27                           "Depth bound: " + depthBound.ToString() + "\n"
28                       else
29                           "Depth bound check is off.\n")
30
31              override this.Search() =
32                  let currNode = currPath.Peek()
33                  let depthText =
34                      if depthBound > 0 then
35                          sprintf " (depth=%d)" currNode.Depth
36                      else
37                          ""
38                  if currNode.State.GoalState then
39                      this.PrintLogEntry(Verbosity.Debug,
40                          sprintf "Current state: %O%s"
41                              currNode.State depthText)
42                      if not (this.SolutionIsState &&
43                              this.GoalNodes.Contains(currNode)) then
```

```
44                          this.GoalNodes.Add(currNode)
45                      if this.AllSolutions then
46                          this.PrintLogEntry(Verbosity.Info,
47                              "Found a solution, backtracking.")
48                          currPath.Pop() |> ignore
49                  elif depthBound > 0 && currNode.Depth = depthBound then
50                      this.PrintLogEntry(Verbosity.Debug,
51                          sprintf "Current state: %O%s"
52                              currNode.State depthText)
53                      this.PrintLogEntry(Verbosity.Info,
54                          "Reached depth bound, backtracking.")
55                      currPath.Pop() |> ignore
56                  else
57                      State.Operators
58                          |> Seq.filter (fun op ->
59                              currNode.State.PreCondition(op))
60                          |> Seq.takeWhile (fun _ ->
61                              this.AllSolutions ||
62                              this.GoalNodes.Count = 0)
63                          |> Seq.iter (fun op ->
64                              this.PrintLogEntry(Verbosity.Debug,
65                                  sprintf "Current state: %O%s"
66                                      currNode.State depthText)
67                              this.PrintLogEntry(Verbosity.Debug,
68                                  sprintf "Applying operator: %O" op)
69                              let newNode = Node(currNode, op )
70                              this.PrintLogEntry(Verbosity.Debug,
71                                  sprintf "New state: %O" newNode.State)
72                              if cycleCheck &&
73                                  currPath.Contains(newNode) then
74                                  this.PrintLogEntry(Verbosity.Info,
75                                      "Found a cycle.")
76                              else
77                                  currPath.Push(newNode)
78                                  this.Search())
79                      if this.AllSolutions ||
80                          this.GoalNodes.Count = 0 then
81                          this.PrintLogEntry(Verbosity.Debug,
82                              sprintf "Current state: %O%s"
83                                  currNode.State depthText)
84                          this.PrintLogEntry(Verbosity.Info,
85                             "No more applicable operators, backtracking.")
86                          currPath.Pop() |> ignore
87
88              override this.ToString() =
89                  if verbosity = Verbosity.None then
90                      ""
91                  else
92                      "Searching using backtracking.\n" +
```

```
93                            this.PropertiesText }
94          currPath.Push(Node(initState))
95          backtrack
```

Similarly, we can create functions for the graph search algorithms, but they will have `GraphSearchAlg` serving as the "base class" in the object expression:

```
1   module GraphSearchAlg
2
3   open System.Collections.Generic
4   open StateSpace
5   open SearchAlg
6
7   let aAlgorithm initState properties verbosity =
8       let properties = defaultArg properties SearchProp.None
9       let verbosity = defaultArg verbosity Verbosity.Info
10      let aAlg =
11          { new GraphSearchAlg(properties, verbosity) with
12              override this.Expand(node) =
13                  State.Operators
14                      |> Seq.filter (fun op -> node.State.PreCondition(op))
15                      |> Seq.iter (fun op ->
16                          let newNode =
17                              NodeWithCost(node :?> NodeWithCost, op)
18                          let index = this.OpenNodes.IndexOf(newNode)
19                          if index <> -1 then
20                              let oldNode =
21                                  this.OpenNodes.[index] :?> NodeWithCost
22                              if newNode.Cost < oldNode.Cost then
23                                  this.OpenNodes.Remove(oldNode) |> ignore
24                                  this.OpenNodes.Add(newNode)
25                          else
26                              let index = this.ClosedNodes.IndexOf(newNode)
27                              if index <> -1 then
28                                  let oldNode =
29                                      this.ClosedNodes.[index]
30                                          :?> NodeWithCost
31                                  if newNode.Cost < oldNode.Cost then
32                                      this.ClosedNodes.Remove(oldNode)
33                                          |> ignore
34                                      this.OpenNodes.Add(newNode)
35                              else
36                                  this.OpenNodes.Add(newNode))
37
38              override this.Search() =
39                  this.PrintInfo()
40                  if this.OpenNodes.Count > 0 then
41                      let currNode = this.OpenNodes.[0]
42                      if currNode.State.GoalState then
```

```
43                          this.GoalNodes.Add(currNode)
44                          if this.AllSolutions then
45                              this.PrintLogEntry(Verbosity.Info,
46                                  "Found a solution.")
47                              this.OpenNodes.Remove(currNode) |> ignore
48                              this.ClosedNodes.Add(currNode)
49                              this.Search()
50                      else
51                          this.OpenNodes.Remove(currNode) |> ignore
52                          this.ClosedNodes.Add(currNode)
53                          this.Expand(currNode)
54                          this.OpenNodes.Sort({ new IComparer<Node> with
55                              member this.Compare(n1, n2) =
56                                  let f1 = (n1 :?> NodeWithCost).Cost +
57                                           n1.State.Heuristic
58                                  let f2 = (n2 :?> NodeWithCost).Cost +
59                                           n2.State.Heuristic
60                                  f1.CompareTo(f2) })
61                          this.Search()
62
63              override this.ToString() =
64                  if verbosity = Verbosity.None then
65                      ""
66                  else
67                      "Searching using A algorithm.\n" +
68                          this.PropertiesText }
69      aAlg.OpenNodes.Add(NodeWithCost(initState))
70      aAlg
```

To summarize: the only difference from the first version is that we did not create separate classes for all search algorithms, only the two abstract classes (`SearchAlg` and `GraphSearchAlg`) remained. Instead, we used functions for creating the objects representing each of the search algorithms. Although this code has the same number of lines as the first implementation, using functions and object expressions instead of classes seems to be a good first step in the process of making our code more functional.

The Main Program

The main program in this implementation becomes the following:

```
1   open SearchAlg
2   open Backtrack
3
4   let main () =
5       let searchAlg =
6           backtrackSearch (Hanoi.HanoiState())
7               (Some (SearchProp.CycleCheckFlag |||
8                      SearchProp.AllSolutionsFlag))
9               (Some Verbosity.Debug) (Some 10)
```

```
10      printfn "%O" searchAlg
11      searchAlg.Search()
12      searchAlg.GoalNodes
13          |> Seq.iteri (fun i solution ->
14              printfn "\nSolution #%d:" (i + 1)
15              searchAlg.PrintSolution(Some solution))
16      printfn "\nNumber of solutions: %d" searchAlg.GoalNodes.Count
```

As you can see, the only difference is that we now have to call the `backtrackSearch` function to get the object that executes the backtracking algorithm with the same properties as earlier. The first argument of the function is the same as before, but the other three arguments are now of `option` type so that we can use `None` to work with the default values. It was not necessary in the first version because methods may have optional arguments whose names begin with a question mark (`?`), and the type of such arguments are automatically converted to `option` type by the F# compiler. If an optional argument is omitted by the caller, the value of the corresponding parameter will be `None`. F# functions, however, may not have optional arguments, so we need to use `option` types explicitly.

3.2.4 The Third F# Implementation

Apart from the way of creating the objects representing the different search algorithms, the second implementation does not differ from the first one. We can make our code really functional by placing the functionality of all abstract and concrete algorithm classes into one `search` function, which takes the following parameters:

- the initial state of a specific problem (`initState`)
- search properties (`properties`)
- verbosity level (`verbosity`)
- the algorithm to be used for searching along with further properties specific to that algorithm (`algorithm`)

So, instead of creating different functions for the different search algorithms, we will create only one function that takes care of everything and returns the list of the goal nodes. For the `algorithm` parameter, we need a discriminated union covering the possible algorithm types and their special properties:

```
1  type AlgorithmType =
2      | Backtrack of int option
3      | BranchAndBound of double option
4      | BreadthFirstSearch
5      | DepthFirstSearch
6      | Dijkstra
7      | BestFirstSearch
8      | AAlgorithm
```

Besides the properties handled with flags (i.e., one bit in the `properties` parameter: `AllSolutionsFlag`, `SolutionIsStateFlag`, and `CycleCheckFlag`), backtracking may have a depth bound parameter, whereas branch-and-bound search may have an initial cost bound parameter. None of the other algorithms require special properties.

Inside the `search` function, local values are used instead of properties like `AllSolutions` or `GoalNodes`, and local helper functions will play the role of the `ToString` or `PrintLogEntry` methods. Local functions will also be used to simulate inheritance, for example, `backtrack` and `graphSearchAlg` will be local functions of `search`.

I will now use a top-down approach to introduce the `search` function. Let's have a look at the outermost level first:

```
1   let search initState properties verbosity algorithm =
2       let properties = defaultArg properties SearchProp.None
3       let verbosity = defaultArg verbosity Verbosity.Info
4       let allSolutions =
5           properties &&& SearchProp.AllSolutionsFlag <> SearchProp.None
6       let solutionIsState =
7           properties &&& SearchProp.SolutionIsStateFlag <> SearchProp.None
8       let cycleCheck =
9           properties &&& SearchProp.CycleCheckFlag <> SearchProp.None
10      let goalNodes = List<Node>()
11
12      let printSearchInfo () =
13          let printCommonProperties () =
14              if allSolutions then
15                  printfn "Searching for all solutions."
16              else
17                  printfn "Searching for the first solution."
18              if solutionIsState then
19                  printf "The goal state"
20              else
21                  printf "The operator sequence leading to the goal state"
22              printfn " is considered to be the solution."
23              printfn "Verbosity level: %O" verbosity
24
25          match algorithm with
26              | Backtrack depthBound ->
27                  printfn "Searching using backtracking."
28                  if verbosity <> Verbosity.None then
29                      printCommonProperties ()
30                      if cycleCheck then
31                          printfn "Cycle check is on."
32                      else
33                          printfn "Cycle check is off."
34                      if depthBound.IsSome then
35                          printfn "Depth bound: %d" depthBound.Value
36                      else
37                          printfn "Depth bound check is off."
38              | BranchAndBound initBound ->
```

```
39                  printfn "Searching using branch-and-bound algorithm."
40                  if verbosity <> Verbosity.None then
41                      printCommonProperties ()
42                      if cycleCheck then
43                          printfn "Cycle check is on."
44                      else
45                          printfn "Cycle check is off."
46                      if initBound.IsSome then
47                          printfn "Initial cost bound: %g" initBound.Value
48                      else
49                          printfn "No initial cost bound."
50                | BreadthFirstSearch ->
51                  printfn "Searching using breadth-first search."
52                  if verbosity <> Verbosity.None then
53                      printCommonProperties ()
54                | DepthFirstSearch ->
55                  printfn "Searching using depth-first search."
56                  if verbosity <> Verbosity.None then
57                      printCommonProperties ()
58                | Dijkstra ->
59                  printfn "Searching using Dijktra's algorithm."
60                  if verbosity <> Verbosity.None then
61                      printCommonProperties ()
62                | BestFirstSearch ->
63                  printfn "Searching using best-first search."
64                  if verbosity <> Verbosity.None then
65                      printCommonProperties ()
66                | AAlgorithm ->
67                  printfn "Searching using A algorithm."
68                  if verbosity <> Verbosity.None then
69                      printCommonProperties ()
70            printfn ""
71
72        let printLogEntry minLevel entry =
73            if verbosity >= minLevel then
74                printfn "%s" entry
75
76        let backtrack depthBound =

    ⋮

136       let branchAndBound initBound =

    ⋮

196       let graphSearchAlg () =

    ⋮
```

```
335         printSearchInfo ()
336         match algorithm with
337             | Backtrack depthBound     -> backtrack depthBound
338             | BranchAndBound initBound -> branchAndBound initBound
339             | _                        -> graphSearchAlg ()
```

At the beginning of the function, values common to all search algorithms are defined, which were properties of `SearchAlg` or one of its subclasses. `goalNodes` is initialized again with an empty list of nodes. Then comes the local function `printSearchInfo`, which is a substitute for the former `ToString` methods. It uses the `algorithm` parameter to decide what information is to be printed to the screen. `printLogEntry` is another local function that was a method in the `SearchAlg` class. One method is missing though: `PrintSolution`. It will also become a function but not a local function, because it will be called outside the `search` function (see later).

Before the actual body of the function, three more local functions are defined: one for backtracking (`backtrack`), one for branch-and-bound search (`branchAndBound`), and one for the graph search algorithms (`graphSearchAlg`). Finally, the actual work is done in a couple of lines: after printing all the search information, we call one of the local functions depending on the value of the `algorithm` parameter, and the return value of these functions will also become the return value of the `search` function.

Let's continue with the `backtrack` function:

```
 76     let backtrack depthBound =
 77         let depthBound = defaultArg depthBound 0
 78         if depthBound < 0 then
 79             raise InvalidBound
 80         let currPath = Stack<Node>()
 81         currPath.Push(Node(initState))
 82
 83         let rec doWork () =
 84             let currNode = currPath.Peek()
 85             let depthText =
 86                 if depthBound > 0 then
 87                     sprintf " (depth=%d)" currNode.Depth
 88                 else
 89                     ""
 90             if currNode.State.GoalState then
 91                 printLogEntry Verbosity.Debug
 92                     (sprintf "Current state: %O%s" currNode.State depthText)
 93                 if not (solutionIsState &&
 94                         goalNodes.Contains(currNode)) then
 95                     goalNodes.Add(currNode)
 96                 if allSolutions then
 97                     printLogEntry Verbosity.Info
 98                         "Found a solution, backtracking."
 99                     currPath.Pop() |> ignore
100             elif depthBound > 0 && currNode.Depth = depthBound then
101                 printLogEntry Verbosity.Debug
```

```
102                     (sprintf "Current state: %O%s" currNode.State depthText)
103                 printLogEntry Verbosity.Info
104                     "Reached depth bound, backtracking."
105                 currPath.Pop() |> ignore
106             else
107                 State.Operators
108                     |> Seq.filter (fun op -> currNode.State.PreCondition(op))
109                     |> Seq.takeWhile (fun _ ->
110                         allSolutions || goalNodes.Count = 0)
111                     |> Seq.iter (fun op ->
112                         printLogEntry Verbosity.Debug
113                             (sprintf "Current state: %O%s"
114                                 currNode.State depthText)
115                         printLogEntry Verbosity.Debug
116                             (sprintf "Applying operator: %O" op)
117                         let newNode = Node(currNode, op)
118                         printLogEntry Verbosity.Debug
119                             (sprintf "New state: %O" newNode.State)
120                         if cycleCheck && currPath.Contains(newNode) then
121                             printLogEntry Verbosity.Info "Found a cycle."
122                         else
123                             currPath.Push(newNode)
124                             doWork ())
125                 if allSolutions || goalNodes.Count = 0 then
126                     printLogEntry Verbosity.Debug
127                         (sprintf "Current state: %O%s"
128                             currNode.State depthText)
129                     printLogEntry Verbosity.Info
130                         "No more applicable operators, backtracking."
131                     currPath.Pop() |> ignore
132
133         doWork ()
134         goalNodes
```

A new recursive function (`doWork`) has been introduced to do the work of the former `Search` method. After checking the value of the `depthBound` parameter and initializing the current path with the start node, we just have to call the `doWork` function and return the list of the goal nodes.

The `branchAndBound` function is very similar, but `graphSearchAlg` is a little more complicated, so let's see how it works:

```
196     let graphSearchAlg () =
197         let openNodes = List<Node>()
198         let closedNodes = List<Node>()
199
200         let printInfo () =
201             let printNodesCount () =
202                 printfn "Open nodes: %d, closed nodes: %d."
203                     openNodes.Count closedNodes.Count
```

```
204
205         let printDatabase () =
206             printfn "Open nodes:"
207             for node in openNodes do
208                 printfn "%O" node
209             printfn "Closed nodes:"
210             for node in closedNodes do
211                 printfn "%O" node
212             printfn ""
213
214         if verbosity = Verbosity.Info then
215             printNodesCount ()
216         elif verbosity = Verbosity.Debug then
217             printDatabase ()
218
219     let rec graphSearch sortOpenNodes =
```

⋮

```
296     let breadthFirstSearch () =
297         openNodes.Add(Node(initState))
298         fun () -> ()
299
300     let depthFirstSearch = breadthFirstSearch
301
302     let dijkstra () =
303         openNodes.Add(NodeWithCost(initState))
304         fun () -> openNodes.Sort({ new IComparer<Node> with
305             member this.Compare(n1, n2) =
306                 (n1 :?> NodeWithCost).Cost.CompareTo(
307                     (n2 :?> NodeWithCost).Cost) })
308
309     let bestFirstSearch () =
310         openNodes.Add(Node(initState))
311         fun () -> openNodes.Sort({ new IComparer<Node> with
312             member this.Compare(n1, n2) =
313                 n1.State.Heuristic.CompareTo(n2.State.Heuristic) })
314
315     let aAlgorithm () =
316         openNodes.Add(NodeWithCost(initState))
317         fun () -> openNodes.Sort({ new IComparer<Node> with
318             member this.Compare(n1, n2) =
319                 let f1 = (n1 :?> NodeWithCost).Cost + n1.State.Heuristic
320                 let f2 = (n2 :?> NodeWithCost).Cost + n2.State.Heuristic
321                 f1.CompareTo(f2) })
322
323     let sortOpenNodes =
324         match algorithm with
325             | BreadthFirstSearch -> breadthFirstSearch ()
```

```
326             | DepthFirstSearch   -> depthFirstSearch ()
327             | Dijkstra           -> dijkstra ()
328             | BestFirstSearch    -> bestFirstSearch ()
329             | AAlgorithm         -> aAlgorithm ()
330             | _                  ->
331                 failwith "Invalid graph search algorithm"
332         graphSearch sortOpenNodes
333         goalNodes
```

openNodes, closedNodes, and printInfo are the respective counterparts of the two properties and the method with similar names in the GraphSearchAlg class.

The most interesting part of the code follows. It is easy to see that all graph search algorithms have almost a common control flow with only three major differences between them (there is one more minor difference between Dijkstra's algorithm and the other algorithms, see later):

- the type of nodes in the database (Node versus NodeWithCost),
- the operation of expansion, and
- the sort criteria when sorting the list of open nodes after an expansion.

Because of this, it seemed reasonable to write only one recursive function (graph-Search) for all graph search algorithms that calls two other functions for expanding a node and for sorting the open nodes after it. For this to work, we need as many functions for expanding and sorting as many graph search algorithms we want to support. We can use two different approaches to select the appropriate function:

1. We can write these functions outside graphSearch and pass the appropriate one as an argument to it (in which case graphSearch becomes a higher-order function).
2. The other alternative is to place these functions inside graphSearch as local functions and select the appropriate one before calling it.

In this code, I chose the first alternative for sorting, and the second for expansion—just for the sake of variety.

The sortOpenNodes parameter of graphSearch is the function that will be called after each expansion. Before calling graphSearch, we have to assign a correct value to this parameter. This is achieved in lines 323–331, where the value of sortOpenNodes is set using a match expression. For each possible value of the algorithm parameter, a function is called, which does two things:

- initializes the list of open nodes with the start node of the appropriate type, and
- returns a lambda function which does the sorting the usual way, i.e., using an IComparer object, or in case of breadth-first search and depth-first search, it is an empty function.

Now, `graphSearch` can be called with the returned lambda function as an argument, and then we can return the list of the goal nodes.

And finally, here is the last part of the `search` function containing the "common graph search algorithm":

```
219    let rec graphSearch sortOpenNodes =
220        let expand (node : Node) =
221            let processNodeBFS op =
222                let newNode = Node(node, op)
223                if not (openNodes.Contains(newNode) ||
224                         closedNodes.Contains(newNode)) then
225                    openNodes.Add(newNode)
226
227            let processNodeDFS op =
228                let newNode = Node(node, op)
229                if not (openNodes.Contains(newNode) ||
230                        closedNodes.Contains(newNode)) then
231                    openNodes.Insert(0, newNode)
232
233            let processNodeDijkstra op =
234                let newNode = NodeWithCost(node :?> NodeWithCost, op)
235                let index = openNodes.IndexOf(newNode)
236                if index <> -1 then
237                    let oldNode = openNodes.[index] :?> NodeWithCost
238                    if newNode.Cost < oldNode.Cost then
239                        openNodes.Remove(oldNode) |> ignore
240                        openNodes.Add(newNode)
241                elif not (closedNodes.Contains(newNode)) then
242                    openNodes.Add(newNode)
243
244            let processNodeAAlg op =
245                let newNode = NodeWithCost(node :?> NodeWithCost, op)
246                let index = openNodes.IndexOf(newNode)
247                if index <> -1 then
248                    let oldNode = openNodes.[index] :?> NodeWithCost
249                    if newNode.Cost < oldNode.Cost then
250                        openNodes.Remove(oldNode) |> ignore
251                        openNodes.Add(newNode)
252                else
253                    let index = closedNodes.IndexOf(newNode)
254                    if index <> -1 then
255                        let oldNode = closedNodes.[index] :?> NodeWithCost
256                        if newNode.Cost < oldNode.Cost then
257                            closedNodes.Remove(oldNode) |> ignore
258                            openNodes.Add(newNode)
259                    else
260                        openNodes.Add(newNode)
261
262            let processNode =
```

```
263                 match algorithm with
264                     | BreadthFirstSearch
265                     | BestFirstSearch    -> processNodeBFS
266                     | DepthFirstSearch   -> processNodeDFS
267                     | Dijkstra           -> processNodeDijkstra
268                     | AAlgorithm         -> processNodeAAlg
269                     | _                  ->
270                         failwith "Invalid graph search algorithm"
271             State.Operators
272                 |> Seq.filter (fun op -> node.State.PreCondition(op))
273                 |> Seq.iter processNode
274
275         printInfo ()
276         if openNodes.Count > 0 then
277             let currNode = openNodes.[0]
278             if algorithm <> AlgorithmType.Dijkstra ||
279                 not (goalNodes.Count > 0 &&
280                     (currNode :?> NodeWithCost ).Cost >
281                         (goalNodes.[0] :?> NodeWithCost).Cost) then
282                 if currNode.State.GoalState then
283                     goalNodes.Add(currNode)
284                     if allSolutions then
285                         printLogEntry Verbosity.Info "Found a solution."
286                         openNodes.Remove(currNode) |> ignore
287                         closedNodes.Add(currNode)
288                         graphSearch sortOpenNodes
289                 else
290                     openNodes.Remove(currNode) |> ignore
291                     closedNodes.Add(currNode)
292                     expand currNode
293                     sortOpenNodes ()
294                     graphSearch sortOpenNodes
```

The `graphSearch` function begins with the definition of `expand`, which is an inner function responsible for expansion. As you can see, there is only one `expand` function for all graph search algorithms. This is because expansion itself works the same way in all algorithms (lines 271–273 contain its code): we iterate through all operators applicable to the current state and process the current node by applying each operator and doing something else, which is algorithm-dependent. So, only this algorithm-dependent part of the code needs to be separated from the `expand` function and placed in distinct functions like `processNodeDFS` or `processNodeAAlg`. Then, the appropriate function is selected again with a `match` expression in lines 262–270, and the selected function is given as an argument to the `Seq.iter` higher-order function (in line 273).

Lines 275–294 contain the actual body of the `graphSearch` function. Expansion of the current node can be found in line 292, sorting the open nodes is in the next line, both as function calls. The other parts of the code is the same in all graph search algorithms, except for the condition in lines 278–281, which became a little complicated because Dijkstra's algorithm should stop if we already have a solution with a cost less

than that of the current node.

There is only one missing function left: `printSolution`. Here it is:

```
1   let rec printSolution (node : Node option) solutionIsState =
2       if solutionIsState then
3           try
4               printfn "%O" node.Value.State
5           with
6               :? NullReferenceException -> printfn "Null as a solution???"
7       elif node.IsSome then
8           printSolution node.Value.Parent solutionIsState
9           printfn "%O" node.Value
```

As `printSolution` is now an external function, it needs one more parameter besides the goal node: `solutionIsState`. Otherwise, it works the same way as the previous implementations.

The Main Program

For the third implementation of the search algorithms, we can use a similar main program as earlier:

```
1   open SearchAlg
2
3   let main () =
4       let solutions = search (Hanoi.HanoiState())
5                              (Some (SearchProp.CycleCheckFlag |||
6                                     SearchProp.AllSolutionsFlag))
7                              (Some Verbosity.Debug)
8                              (Backtrack (Some 10))
9       solutions
10          |> Seq.iteri (fun i solution ->
11              printfn "\nSolution #%d:" (i + 1)
12              printSolution (Some solution) false)
13      printfn "\nNumber of solutions: %d" solutions.Count
```

The `main` function differs from the previous versions mainly in that now there is no `GoalNodes` property, it is replaced with the return value of the `search` function. The algorithm to be used for searching is given as the last argument to `search`, along with its special properties (depth bound in our case). And finally, `printSolution` now requires a second argument, which denotes whether the goal state itself or the operator sequence leading to it is considered to be the solution.

3.2.5 Comparing the Four Implementations

My goal with creating the presented implementations of AI search algorithms was to give students more approaches to understand the same pseudocode. Although I do not

know if they can better acquire the operation of these algorithms by thinking functionally, giving more than just one implementation cannot be harmful. Now, I present a partly subjective comparison of the four implementations from various aspects.

- *Code metrics*: Despite that we cannot use the same metrics in case of a functional language as in case of imperative languages, I will use three metrics here that may be relevant to both C# and F#: lines of code, number of classes, and number of functions (including methods). Table 3.1 summarizes the values of these metrics in the different implementations:

	C#	F# ver. 1	F# ver. 2	F# ver. 3
Lines of code	842	537	536	417
Number of classes	15	13	6	4
Number of functions	49	43	43	34

Table 3.1. Some code metrics.

The *lines of code* metric denotes the number of lines in all source files, including empty lines, but not including the source code of any specific problems. The *number of classes* includes all classes defined in the source code but does not include exception classes, enumerations, and `IComparer` classes. The *number of functions* includes all functions and nonabstract method implementations, including constructors, but does not include lambda expressions.

As you can see, the third F# implementation is half the size of the C# implementation. Of course, this difference comes mainly from the compact syntax of the F# language. The other reason for the F# implementations being shorter is that they do not contain two classes from the C# code (`BacktrackNode` and `BacktrackNodeWithCost`), which take 57 lines of code.

The decrease in the number of classes is a result of making the code more functional. The first F# implementation has two classes less than the C# version because it lacks the above-mentioned two classes. In the second version, the seven concrete algorithm classes are replaced with seven functions. In the third version, the two abstract classes `SearchAlg` and `GraphSearchAlg` are also converted to functions. The four remaining classes are `State`, `Operator`, `Node`, and `NodeWithCost`. To make the code purely functional, we would have to get rid of these classes too, but it would not result in a shorter or more readable code. For example, with these four classes, it is easy to write *reusable* code for an operator application: we just have to call the `Apply` method of the abstract `State` class without knowing how it is implemented in the concrete class representing the states of a specific problem. And this is exactly what the constructors of the two `Node` classes do. Actually, the two `Node` classes might be replaced with record types because these classes are not inherited by any other classes, but again, it would not be more readable, and on top of that, we would not be able to use such .NET methods as `Contains`.

The number of functions are very similar in each implementation. It is because each method in a class maps to a function in a classless implementation. Even constructors have corresponding functions; for example, there is a `backtrackSearch` function corresponding to the constructor of the `BacktrackSearch` class. The third F# implementation, however, contains somewhat less functions than the other three. The reason for this is that it lacks the seven overrides of the `ToString` method in each of the concrete algorithm classes (or objects). There is only one `printSearchInfo` function instead. The same is true for the `Expand` and `Search` methods in the graph search algorithms. All of the corresponding functions of these methods contain `match` expressions based on the algorithm used for searching. This shows well the difference between the object-oriented and the functional approach to the problem of introducing a new subclass of a base class and introducing new functionality to the subclasses. The OO approach is better if we want to introduce a new subclass because we do not have to touch the existing subclasses, just write the new class with all the functionality inherited from the base class. In FP, we need to add a new branch to all of the `match` expressions. However, FP is better if we want to add new functionality to the existing subclasses because we just have to write a new function with a similar `match` expression to the existing ones. In OO, we need to extend the base class with a new method and all of its subclasses with method overrides.

- *Mutable data structures used*: There is no difference in this aspect between the implementations. Although purely functional programs use no mutable data at all, I used the following mutable data structures in each implementation:
  - `State.Operators` of type `HashSet<Operator>`
  - `SearchAlg.GoalNodes` of type `List<Node>`
  - `BacktrackSearch.currPath` of type `Stack<Node>`
  - `BranchAndBoundSearch.currPath` of type `Stack<NodeWithCost>`
  - `GraphSearchAlg.OpenNodes` of type `List<Node>`
  - `GraphSearchAlg.ClosedNodes` of type `List<Node>`

  If we want, we can replace the type of any or all of these data collections with F#'s immutable `list` or `seq` data type. The resulting code would be of the same size, but it would be less efficient because the recursive functions in the `List` and `Seq` modules are slower than the corresponding .NET methods. It is particularly true when elements are added to or deleted from these collections: in case of an immutable data structure, we have to copy the original collection with a slight modification in its elements. This is the reason why I used .NET collections instead of F#'s immutable data types for storing the collections listed above.

- *Functional language constructs used*: The C# implementation does not contain any functional constructs, it is purely object-oriented. The first F# implementation uses tail-recursive functions and sequence operations as a replacement for loops. Although object expressions are not functional language constructs, the second

F# implementation is full of them as a substitute for subclasses. The third F# implementation contains the most functional elements: a discriminated union is used for storing the algorithm type, `match` expressions are used to deal with it, some functions are used as first-class values, and `graphSearch` is a higher-order function.

- *Efficiency*: As functional languages are more abstract than object-oriented languages, they need a more complicated runtime environment. This is the main reason why functional programs are generally less efficient than object-oriented programs, even if they are compiled and not interpreted. I ran the presented main programs of all implementations on the same computer, a Gigabyte T1028X TouchNote netbook with Intel Atom N280 CPU at 1.33 GHz and 1 GB of RAM, and all programs compiled with Microsoft Visual Studio 2010: the C# program finished in less than half a second, while the F# programs all ran for about 7 seconds. As I wrote, it would have been even worse if immutable F# data types were used.

As a final conclusion, my opinion is that it is not worth insisting on one or the other paradigm if we can use more of them within one program. Functional code is sometimes more abstract, more readable, or just shorter than its object-oriented counterpart. On the other hand, OO code is usually more efficient and sometimes more reusable than its functional counterpart. This is why I think multiparadigm languages like F# can be more advantageous mainly in large-scale applications but also in smaller programs.

3.3 Implementing Specific Problems

Implementing the search algorithms themselves is not the only area where we can benefit from functional (or multiparadigm) programming. Creating an implementation for a specific state-space-represented problem has its own peculiarities too. In this section, I present a simple and a more complex problem, and give some possible implementations of them.

3.3.1 Towers of Hanoi

The Problem

We will now consider a simplified version of the original puzzle. It consists of three pegs and three discs of different sizes which can slide onto any peg. The puzzle starts with the discs in a neat stack in ascending order of size on one peg, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another peg, obeying the following rules:

- Only one disc may be moved at a time.
- Each move consists of taking the upper disc from one of the pegs and sliding it onto another peg, on top of the other discs that may already be present on that peg.
- No disc may be placed on top of a smaller disc.

The solution of the problem is a sequence of moves.

State-Space Representation

Let's denote the three pegs with the letters A, B, and C, and the discs with the numbers 1, 2, and 3 in ascending order of their sizes (i.e., 1 denotes the smallest disc, and 3 denotes the largest). Let's consider a relevant property of the problem the pegs on which each disc can be found. This seems to be the most efficient representation with very little memory required for storing a state. As each disc can occur on any of the pegs, we can assign the same base set to each of the discs:

$$H_1 = H_2 = H_3 = \{A, B, C\}$$

The states of the problem will be elements of the Cartesian product of these base sets:

$$\begin{aligned}\mathcal{S} \subseteq H_1 \times H_2 \times H_3 = \{&(A,A,A),\ (A,A,B),\ (A,A,C),\\ &(A,B,A),\ (A,B,B),\ (A,B,C),\\ &(A,C,A),\ (A,C,B),\ (A,C,C),\\ &(B,A,A),\ (B,A,B),\ (B,A,C),\\ &(B,B,A),\ (B,B,B),\ (B,B,C),\\ &(B,C,A),\ (B,C,B),\ (B,C,C),\\ &(C,A,A),\ (C,A,B),\ (C,A,C),\\ &(C,B,A),\ (C,B,B),\ (C,B,C),\\ &(C,C,A),\ (C,C,B),\ (C,C,C)\}\end{aligned}$$

As all the elements of the $H_1 \times H_2 \times H_3$ set are valid states of our problem, there is no need for any constraints to narrow this set, i.e., the *state space* of the problem will be exactly this set:

$$\mathcal{S} = H_1 \times H_2 \times H_3$$

At the *initial state*, all discs are on peg A:

$$start = (A, A, A) \in \mathcal{S}$$

The *set of goal states* consists of two elements, which denote that all discs are on peg B or all discs are on peg C:

$$\mathcal{G} = \{(B, B, B), (C, C, C)\} \subset \mathcal{S}$$

The *set of operators* contains nine elements:

$$\mathcal{O} = \{\text{Move}(disc,\ peg)\}$$

where

$$disc \in \{\ A, B, C\ \}$$
$$peg \in \{\ 1, 2, 3\ \}$$

The Move(*disc*, *peg*) operator is applicable to state $h = (h_1, h_2, h_3) \in \mathcal{S}$ if all of the following preconditions are met:

- *disc* is the smallest one on its peg:

$$\forall i\,(i < disc \supset h_i \neq h_{disc})$$

- there are no smaller discs on *peg* than *disc*:

$$\forall i\,(i < disc \supset h_i \neq peg)$$

- *disc* is currently not on *peg*:

$$h_{disc} \neq peg$$

The application of the Move(*disc*, *peg*) operator to state $h = (h_1, h_2, h_3) \in \mathcal{S}$ results in a new state $h' = (h'_1, h'_2, h'_3) \in \mathcal{S}$ where

$$h'_i = \begin{cases} peg & \text{if } i = disc, \\ h_i & \text{otherwise.} \end{cases}$$

By defining the $\mathcal{S}$ state space, the *start* initial state, the $\mathcal{G}$ set of goal states, and the $\mathcal{O}$ set of operators, we have given a possible $p = \langle \mathcal{S}, start, \mathcal{G}, \mathcal{O} \rangle$ state-space representation of our problem. In figure 3.3, you can see the whole state-space graph of this puzzle, M denoting the Move operator:

The C# Implementation

Figure 3.4 shows the two classes representing the states and operators of this particular problem.

The only extra member in `HanoiState` is `discs`, which is an array of pegs where each disc can be found. This field holds all information contained by a state in the representation. `Move` comes with two members in addition to the inherited ones: `Disc` tells us which disc to move, while `Peg` is the destination peg. These two members correspond to the parameters of the Move operator.

Let's see now the source code of the `Move` and `HanoiState` classes:

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using StateSpace;
5
6   namespace Hanoi
7   {
8     class Move : Operator
9     {
10      internal int Disc { get; private set; }
11      internal char Peg { get; private set; }
12
13      public Move(int disc, char peg)
14      {
```

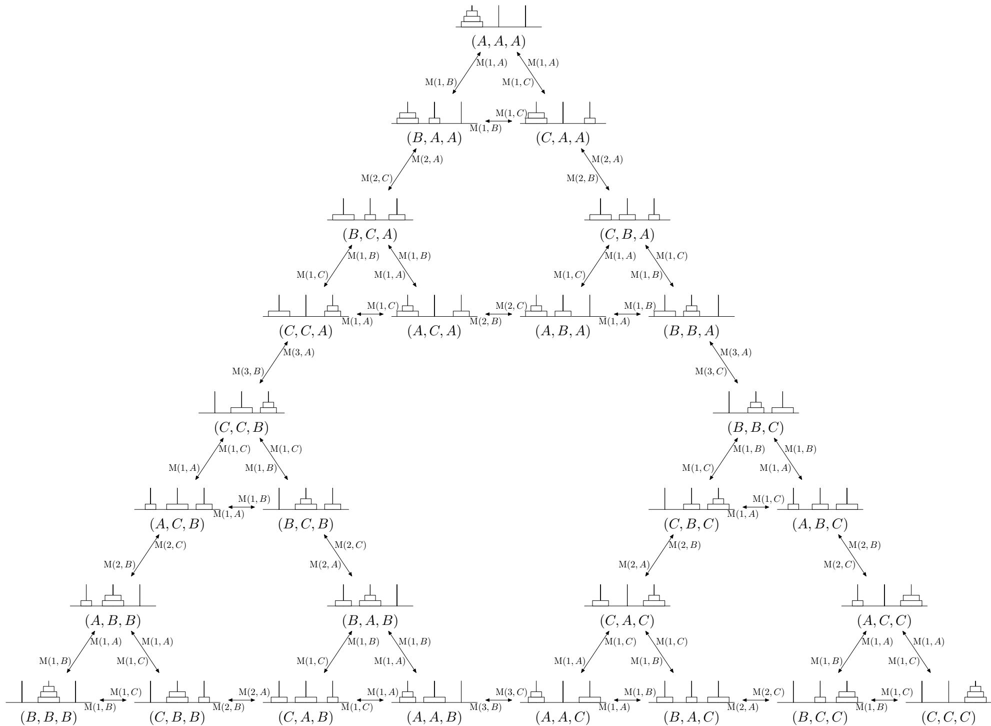


Figure 3.3. State-space graph of the Towers of Hanoi problem.

```
15          Disc = disc;
16          Peg = peg;
17        }
18
19        public override string ToString()
20        {
21          return "HanoiMove[ disc=" + Disc + ", peg=" + Peg + " ]";
22        }
23
24        public override double Cost(State state)
25        {
26          return Disc;
27        }
28      }
29
30      class HanoiState : State
31      {
32        const int N = 3;
33
34        static HanoiState()
```

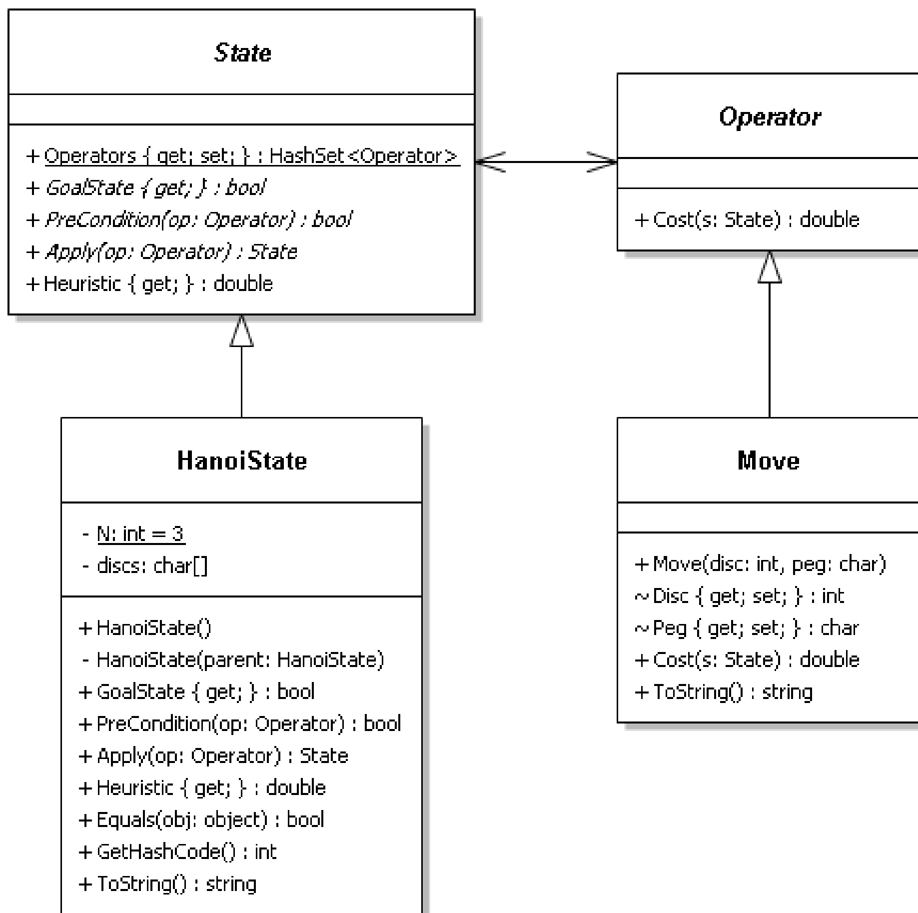


Figure 3.4. Classes representing a specific problem.

```
35        {
36          Operators = new HashSet<Operator>();
37          for (int disc = 1; disc <= N; ++disc)
38            for (char peg = 'A'; peg <= 'C'; ++peg)
39              Operators.Add(new Move(disc, peg));
40        }
41
42        char[] discs;
43
44        public HanoiState()
45        {
```

```
46          discs = new char[N];
47          for (int i = 0; i < N; ++i)
48            discs[i] = 'A';
49        }
50
51        HanoiState(HanoiState parent)
52        {
53          discs = new char[N];
54          parent.discs.CopyTo(discs, 0);
55        }
56
57        public override bool GoalState
58        {
59          get
60          {
61            if (discs[0] == 'A')
62              return false;
63            foreach (char peg in discs)
64              if (peg != discs[0])
65                return false;
66            return true;
67          }
68        }
69
70        public override bool PreCondition(Operator op)
71        {
72          if (op is Move)
73          {
74            Move m = (Move)op;
75            for (int i = 0; i < m.Disc - 1; ++i)
76              if (discs[i] == discs[m.Disc - 1] ||
77                  discs[i] == m.Peg)
78                return false;
79            return discs[m.Disc - 1] != m.Peg;
80          }
81          else
82            throw new InvalidOperatorException();
83        }
84
85        public override State Apply(Operator op)
86        {
87          if (op is Move)
88          {
89            HanoiState newState = new HanoiState(this);
90            Move m = (Move)op;
91            newState.discs[m.Disc - 1] = m.Peg;
92            return newState;
93          }
94          else
```

```
 95          throw new InvalidOperatorException();
 96      }
 97
 98      public override bool Equals(object obj)
 99      {
100        return obj is HanoiState && ToString() == obj.ToString();
101      }
102
103      public override int GetHashCode()
104      {
105        return discs.GetHashCode();
106      }
107
108      public override string ToString()
109      {
110        StringBuilder sb = new StringBuilder("HanoiState[ discs=(");
111        for (int i = 0; i < N; ++i)
112        {
113          if (i > 0)
114            sb.Append(",");
115          sb.Append(discs[i]);
116        }
117        return sb.Append(") ]").ToString();
118      }
119
120      public override double Heuristic
121      {
122        get
123        {
124          double value1 = N, value2 = N;
125          foreach (char peg in discs)
126            if (peg == 'B')
127              --value1;
128            else if (peg == 'C')
129              --value2;
130          return Math.Min(value1, value2);
131        }
132      }
133    }
134  }
```

The implementation of the Move operator is fairly straightforward. The Cost method has been overridden: in our implementation, the cost of moving a disc to another peg is proportional to its size, independently of the state to which the operator is applied.

The HanoiState class defines a one-dimensional array of characters for storing the pegs of each disc. The size of this array is N which stands for the number of discs in the problem. The smaller the index of an array element, the smaller the disc it represents.

The static constructor is responsible for creating all the possible operator instances

and adding them to the static Operators property. The class has two constructors: the public constructor creates the initial state with each disc being on peg A, while the private constructor is actually a copy constructor, which creates a clone of the HanoiState object taken as a parameter.

The GoalState property first makes sure that the smallest disc is not on peg A, then checks whether all the discs are on the same peg (with the help of a foreach loop). The skeletons of the PreCondition and Apply methods are the same: they both go through all the possible operator types (there is only one in our case: Move) and throw an exception if the argument is an operator of an unknown type. The PreCondition method has to make sure that none of the discs smaller than the one to be moved are on the source or the destination pegs and that the disc to be moved is not on the destination peg. The Apply method is very simple: it copies the current state and replaces the peg of the disc to be moved with the one given by the operator. Finally, the Heuristic property determines the number of discs not being on peg B and the same for peg C, and returns the smaller of the two numbers because at least that many moves are required to reach one of the two goal states.

The F# Implementation

A possible F# implementation of this problem, which uses imperative, object-oriented, and functional elements, may look like the following:

```
 1  module Hanoi
 2
 3  open System.Text
 4  open StateSpace
 5
 6  type Move(disc, peg) =
 7      inherit Operator()
 8      member this.Disc = disc
 9      member this.Peg = peg
10
11      override this.ToString() =
12          sprintf "HanoiMove[ disc=%d, peg=%c ]" disc peg
13
14      override this.Cost(_) = double disc
15
16  type HanoiState() =
17      inherit State()
18
19      static let N = 3
20      let discs = Array.create N 'A'
21
22      static do
23          for disc in 1 .. N do
24              for peg in 'A' .. 'C' do
25                  State.Operators.Add(Move(disc, peg)) |> ignore
26
```

```
27        member private this.Discs = discs
28
29        private new(parent : HanoiState) as this =
30            HanoiState() then
31                parent.Discs.CopyTo(this.Discs, 0)
32
33        override this.GoalState =
34            let rec allTheSame index =
35                index >= discs.Length - 1
36                    || discs.[index] = discs.[index + 1]
37                    && allTheSame (index + 1)
38            discs.[0] <> 'A' && allTheSame 0
39
40        override this.PreCondition(op) =
41            match op with
42                | :? Move as move ->
43                    let rec checkSmallerDiscs index =
44                        index >= move.Disc - 1
45                            || discs.[index] <> discs.[move.Disc - 1]
46                            && discs.[index] <> move.Peg
47                            && checkSmallerDiscs (index + 1)
48                    checkSmallerDiscs 0 && discs.[move.Disc - 1] <> move.Peg
49                | _ ->
50                    raise InvalidOperator
51
52        override this.Apply(op) =
53            match op with
54                | :? Move as move ->
55                    let newState = HanoiState(this)
56                    newState.Discs.[move.Disc - 1] <- move.Peg
57                    upcast newState
58                | _ ->
59                    raise InvalidOperator
60
61        override this.Equals(other) =
62            match other with
63                | :? HanoiState as otherHanoiState ->
64                    this.Discs = otherHanoiState.Discs
65                | _ ->
66                    false
67
68        override this.GetHashCode() =
69            hash discs
70
71        override this.ToString() =
72            let sb = StringBuilder("HanoiState[ discs=(")
73            for i in 0 .. N - 1 do
74                if i > 0 then
75                    sb.Append(',') |> ignore
```

```
76                  sb.Append(discs.[i]) |> ignore
77              sb.Append(") ]").ToString()
78
79          override this.Heuristic =
80              let value1 = ref N
81              let value2 = ref N
82              for peg in discs do
83                  if peg = 'B' then
84                      decr value1
85                  elif peg = 'C' then
86                      decr value2
87              double (min !value1 !value2)
```

There is not much difference between the C# and the F# implementations, yet the F# code is much shorter, but this difference in size is again due to the concise syntax of the F# language. Some interesting points about the code worth noting are the following:

- Unlike in the C# code, a `Discs` property had to be defined here because we cannot refer to the `discs` field of a `HanoiState` object other than the current instance (see, for example, `parent.Discs` in the explicit constructor).
- Both the `GoalState` property and the `PreCondition` method use recursive functions instead of loops to iterate through the `discs` array.
- The `Equals` method uses structural equality to compare the arrays in the two objects. The C# code compares the string representations of the objects because in C#, the equality operator between arrays implies reference equality. Since the string representations are different if the arrays of the two objects are different, and the equality operator is overloaded for strings, the use of the equality operator seems adequate in this situation (although it makes comparison rather slow). (We could also use the `SequenceEqual` extension method of LINQ.)
- I used two reference cells (i.e., mutable data) and a `for` loop in the `Heuristic` property. In this case, these imperative language elements do not make the code longer or less readable. We could also write this property purely functionally, like this:

```
79          override this.Heuristic =
80              let rec noOfOtherPegs peg i acc =
81                  if i = N then
82                      acc
83                  else
84                      let otherPeg = if discs.[i] = peg then 0 else 1
85                      noOfOtherPegs peg (i + 1) (acc + otherPeg)
86
87              double (min (noOfOtherPegs 'B' 0 0) (noOfOtherPegs 'C' 0 0))
```

3.3.2 Funfair Puzzle

The Problem

Happy is a family with five children. This year, on the birthdays of each child, they went to the amusement park, and from there, to a confectionery where they had the favorite cake of the birthday kid. Based on the given information, determine

- which month each kid has their birthday,
- which game they tried first in the park, and
- what kind of cake they ordered in the confectionery.

These are the facts:

1. The names of the children are the following: Alex, Carol, Paula, Robert, and Tim.
2. The months of the birthdays are the following: March, April, July, August, and November.
3. The games in the park are the following: dodgem, enchanted castle, roller coaster, carousel, and scenic railway.
4. The cakes are the following: chocolate, walnut, caramel, almond, and vanilla.
5. Carol was not born in March.
6. Neither Alex nor Carol ran from the entrance straight to the enchanted castle.
7. Scenic railway was the first "stop" in the park either in March or in April, and this happened either on Paula's birthday or when they had the vanilla cake.
8. Chocolate cake is neither Alex's nor Paula's favorite, and they didn't have it on the November birthday.
9. Paula's birthday is in a month with the same number of days as the month of Tim's birthday.
10. They had the almond cake before Paula's birthday.
11. Carol is not very fond of the almond cake, but she ate some for her sibling's sake. This happened on a different day from when they rode the roller coaster first.
12. Neither Alex nor Paula chose the walnut cake, and they had it on a different day from when they went to the carousel first in the park.
13. Carol was born either in March or in April.
14. The chocolate cake was asked for either in August or in November by one of the *Happy* kids.
15. The day when the carousel was the first stop in the park, they weren't celebrating Paula's birthday, nor did they have almond cake in the confectionery.

State-Space Representation

The relevant properties of our problem are the *Happy* children and, related to them,

- the months of their birthdays,
- the games they tried first in the park, and
- the kinds of cake they ordered in the confectionery.

This approach implies that we will try to assign the months, the games, and the types of cake to the children.

For a convenient notation, let's assign sequence numbers to the children:

Child's name:	Alex	Carol	Paula	Robert	Tim
Number:	1	2	3	4	5

Similarly, let's assign sequence numbers to the categories:

Category:	month	game	cake
Number:	1	2	3

From now on, we will refer to the children and the categories with their sequence numbers.

Let's now define the base sets that contain all possible months that can be assigned to each child:

$$H_{i,1} = \{\,3, 4, 7, 8, 11\,\} \cup \{\,0\,\}, \quad i \in \{\,1, 2, 3, 4, 5\,\}$$

The 0 symbol denotes that no month is assigned yet to child i.

Let's now define the base sets that contain all possible games that can be assigned to each child:

$$H_{i,2} = \{\,\text{dodgem}, \text{castle}, \text{roller}, \text{carousel}, \text{scenic}\,\} \cup \{\,0\,\}, \quad i \in \{\,1, 2, 3, 4, 5\,\}$$

The 0 symbol denotes that no game is assigned yet to child i.

Let's now define the base sets that contain all possible types of cake that can be assigned to each child:

$$H_{i,3} = \{\,\text{choco}, \text{walnut}, \text{caramel}, \text{almond}, \text{vanilla}\,\} \cup \{\,0\,\}, \quad i \in \{\,1, 2, 3, 4, 5\,\}$$

The 0 symbol denotes that no cake is assigned yet to child i.

The Cartesian product of these sets is the following:

$$H_{1,1} \times H_{2,1} \times \ldots \times H_{5,1} \times H_{1,2} \times H_{2,2} \times \ldots \times H_{5,2} \times H_{1,3} \times H_{2,3} \times \ldots \times H_{5,3} =$$

$$= \left\{ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \ldots, \right.$$

$$\begin{pmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \text{roller} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \ldots, \begin{pmatrix} 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & \text{roller} & 0 \\ 0 & 0 & \text{choco} & 0 & 0 \end{pmatrix}, \ldots,$$

$$\begin{pmatrix} 3 & 0 & 0 & 0 & 8 \\ \text{carousel} & 0 & 0 & \text{roller} & 0 \\ \text{vanilla} & 0 & \text{choco} & 0 & \text{walnut} \end{pmatrix}, \ldots,$$

$$\begin{pmatrix} 3 & 4 & 7 & 8 & 11 \\ \text{dodgem} & \text{castle} & \text{roller} & \text{carousel} & \text{scenic} \\ \text{choco} & \text{walnut} & \text{caramel} & \text{almond} & \text{vanilla} \end{pmatrix}, \ldots,$$

$$\left. \begin{pmatrix} 3 & 4 & 7 & 11 & 8 \\ \text{dodgem} & \text{scenic} & \text{roller} & \text{castle} & \text{carousel} \\ \text{almond} & \text{vanilla} & \text{caramel} & \text{walnut} & \text{choco} \end{pmatrix}, \ldots \right\}$$

The elements of this set are ordered 15-tuples, or matrices of size 3×5 if the elements are arranged in the form of a matrix. The number of elements in this set is $6^{15} = 470\,184\,984\,576$. However, if we take into consideration that the same month, game, or type of cake cannot be assigned to more than one child at a time, we get a lot less 15-tuples. We can also define additional constraints: The months, games, and types of cake should be assigned to the children in this order, i.e., first all the months are assigned, then all the games, and finally, all the cake types. Furthermore, we can also assign a value from each category to the children in order, e.g., again in the order of their sequence numbers. We can also define a number of constraints based on the information given in the problem description, like "the month of Carol's birthday cannot be March," or formally, $h_{2,1} \neq 3$.

There are only 839 15-tuples satisfying all of these constraints, so our *state space* consists of that many states:

$$\mathcal{S} = \left\{ h \mid h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \wedge \mathit{constraints}(h) \right\}$$

The *initial state* describes the situation when no values are assigned to any of the children:

$$\mathit{start} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \in \mathcal{S}$$

The *set of goal states* contains such elements in which the lower right element of the matrix is not zero, i.e., the type of cake is already assigned to Tim:

$$\mathcal{G} = \left\{ h \mid h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \wedge h_{5,3} \neq 0 \right\} \subseteq \mathcal{S}$$

The *set of operators* is defined using three operator identifiers, each with two parameters, resulting in 75 operators altogether:

$$\mathcal{O} = \{\text{Month}(ch,\ month),\ \text{Game}(ch,\ game),\ \text{Cake}(ch,\ cake)\}\,,$$

where

$$\begin{aligned} ch &\in \{\,1, 2, 3, 4, 5\,\} \\ month &\in \{\,3, 4, 7, 8, 11\,\} \\ game &\in \{\,\text{dodgem, castle, roller, carousel, scenic}\,\} \\ cake &\in \{\,\text{choco, walnut, caramel, almond, vanilla}\,\} \end{aligned}$$

The Month(ch, $month$) operator is applicable to state

$$h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \in \mathcal{S}$$

if all of the following preconditions are met:

- child ch has not been assigned a month yet:

$$h_{ch,1} = 0$$

- if $month$ is not to be assigned to Alex (the first child), then the child with a sequence number one less than ch must already have a month assigned:

$$ch \neq 1 \supset h_{ch-1,1} \neq 0$$

- $month$ has not been assigned to any child with a sequence number less than ch:

$$\forall i\,(i < ch \supset h_{i,1} \neq month)$$

- March must not be assigned to Carol (child #2):

$$ch = 2 \supset month \neq 3$$

- March must not be assigned to Paula (child #3):

$$ch = 3 \supset month \neq 3$$

- if we know that Paula (child #3) was born in April or November, then Tim (child #5) must also be assigned April or November:

$$(h_{3,1} = 4 \lor h_{3,1} = 11) \land ch = 5 \supset month = 4 \lor month = 11$$

- if we know that Paula (child #3) was born neither in April nor in November, then Tim (child #5) must not be assigned April or November either:

$$(h_{3,1} = 3 \lor h_{3,1} = 7 \lor h_{3,1} = 8) \land ch = 5 \supset month \neq 4 \land month \neq 11$$

- if Carol (child #2) is to be assigned a month, then *month* must be either March or April:

$$ch = 2 \supset month = 3 \lor month = 4$$

- if March is to be assigned to someone, and we know that it is not Carol (child #2) who was born in April, then March may only be assigned to Carol:

$$\forall i \, (month = 3 \land h_{i,1} = 4 \land i \neq 2 \supset ch = 2)$$

- if April is to be assigned to someone, and we know that it is not Carol (child #2) who was born in March, then April may only be assigned to Carol:

$$\forall i \, (month = 4 \land h_{i,1} = 3 \land i \neq 2 \supset ch = 2)$$

The application of the Month(ch, $month$) operator to state

$$h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \in \mathcal{S}$$

results in a new state

$$h' = \begin{pmatrix} h'_{1,1} & h'_{2,1} & h'_{3,1} & h'_{4,1} & h'_{5,1} \\ h'_{1,2} & h'_{2,2} & h'_{3,2} & h'_{4,2} & h'_{5,2} \\ h'_{1,3} & h'_{2,3} & h'_{3,3} & h'_{4,3} & h'_{5,3} \end{pmatrix} \in \mathcal{S}$$

where

$$h'_{i,j} = \begin{cases} month & \text{if } i = ch \land j = 1, \\ h_{i,j} & \text{otherwise.} \end{cases}$$

The Game(ch, $game$) operator is applicable to state

$$h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \in \mathcal{S}$$

if all of the following preconditions are met:

- child ch has not been assigned a game yet:

$$h_{ch,2} = 0$$

- Alex (child #1) may only be assigned a game if Tim (child #5) has already been assigned a month:

$$ch = 1 \supset h_{5,1} \neq 0$$

- if *game* is not to be assigned to Alex (the first child), then the child with a sequence number one less than ch must already have a game assigned:

$$ch \neq 1 \supset h_{ch-1,2} \neq 0$$

- *game* has not been assigned to any child with a sequence number less than ch:

$$\forall i \, (i < ch \supset h_{i,2} \neq game)$$

- Alex (child #1) must not be assigned the enchanted castle:

$$ch = 1 \supset game \neq \text{castle}$$

- Paula (child #3) must not be assigned the enchanted castle:

$$ch = 3 \supset game \neq \text{castle}$$

- scenic railway may only be assigned to a child who was born in March or April:

$$game = \text{scenic} \supset h_{ch,1} = 3 \lor h_{ch,1} = 4$$

- if we know that the favorite game of the child who was born in April is not the scenic railway, and now a game is to be assigned to the child who was born in March, then that *game* must be scenic railway:

$$\forall i \, (h_{i,1} = 4 \land h_{i,2} \neq 0 \land h_{i,2} \neq \text{scenic} \land h_{ch,1} = 3 \supset game = \text{scenic})$$

- if we know that the favorite game of the child who was born in March is not the scenic railway, and now a game is to be assigned to the child who was born in April, then that *game* must be scenic railway:

$$\forall i \, (h_{i,1} = 3 \land h_{i,2} \neq 0 \land h_{i,2} \neq \text{scenic} \land h_{ch,1} = 4 \supset game = \text{scenic})$$

- Paula (child #3) must not be assigned the carousel:

$$ch = 3 \supset game \neq \text{carousel}$$

The application of the Game(ch, $game$) operator to state

$$h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \in \mathcal{S}$$

results in a new state

$$h' = \begin{pmatrix} h'_{1,1} & h'_{2,1} & h'_{3,1} & h'_{4,1} & h'_{5,1} \\ h'_{1,2} & h'_{2,2} & h'_{3,2} & h'_{4,2} & h'_{5,2} \\ h'_{1,3} & h'_{2,3} & h'_{3,3} & h'_{4,3} & h'_{5,3} \end{pmatrix} \in \mathcal{S}$$

where

$$h'_{i,j} = \begin{cases} game & \text{if } i = ch \wedge j = 2, \\ h_{i,j} & \text{otherwise.} \end{cases}$$

The Cake(ch, $cake$) operator is applicable to state

$$h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \in \mathcal{S}$$

if all of the following preconditions are met:

- child ch has not been assigned a cake yet:

$$h_{ch,3} = 0$$

- Alex (child #1) may only be assigned a cake if Tim (child #5) has already been assigned a game:

$$ch = 1 \supset h_{5,2} \neq 0$$

- if $cake$ is not to be assigned to Alex (the first child), then the child with a sequence number one less than ch must already have a cake assigned:

$$ch \neq 1 \supset h_{ch-1,3} \neq 0$$

- $cake$ has not been assigned to any child with a sequence number less than ch:

$$\forall i \, (i < ch \supset h_{i,3} \neq cake)$$

- Paula (child #3) must not be assigned the vanilla cake:

$$ch = 3 \supset cake \neq \text{vanilla}$$

- if we know that the favorite game of Paula (child #3) is not the scenic railway, then vanilla cake may only be assigned to the child whose favorite game is the scenic railway:

$$h_{3,2} \neq 0 \wedge h_{3,2} \neq \text{scenic} \wedge cake = \text{vanilla} \supset h_{ch,2} = \text{scenic}$$

- if we know that the favorite game of Paula (child #3) is not the scenic railway, and the favorite game of the child to whom a cake is to be assigned is the scenic railway, then *cake* must be vanilla cake:

$$h_{3,2} \neq 0 \wedge h_{3,2} \neq \text{scenic} \wedge h_{ch,2} = \text{scenic} \supset \mathit{cake} = \text{vanilla}$$

- Alex (child #1) must not be assigned the chocolate cake:

$$\mathit{ch} = 1 \supset \mathit{cake} \neq \text{choco}$$

- Paula (child #3) must not be assigned the chocolate cake:

$$\mathit{ch} = 3 \supset \mathit{cake} \neq \text{choco}$$

- chocolate cake must not be assigned to the child who was born in November:

$$\mathit{cake} = \text{choco} \supset h_{ch,1} \neq 11$$

- almond cake must be assigned to a child who was born in an earlier month than Paula (child #3):

$$\mathit{cake} = \text{almond} \supset h_{ch,1} < h_{3,1}$$

- Carol (child #2) must not be assigned the almond cake:

$$\mathit{ch} = 2 \supset \mathit{cake} \neq \text{almond}$$

- almond cake must not be assigned to the child whose favorite game is the roller coaster:

$$\mathit{cake} = \text{almond} \supset h_{ch,2} \neq \text{roller}$$

- Alex (child #1) must not be assigned the walnut cake:

$$\mathit{ch} = 1 \supset \mathit{cake} \neq \text{walnut}$$

- Paula (child #3) must not be assigned the walnut cake:

$$\mathit{ch} = 3 \supset \mathit{cake} \neq \text{walnut}$$

- walnut cake must not be assigned to the child whose favorite game is the carousel:

$$\mathit{cake} = \text{walnut} \supset h_{ch,2} \neq \text{carousel}$$

- chocolate cake may only be assigned to a child who was born in August or November:

$$\mathit{cake} = \text{choco} \supset h_{ch,1} = 8 \vee h_{ch,1} = 11$$

- if a cake is to be assigned to the child who was born in August, and the child who was born in November did not order chocolate cake, then *cake* must be chocolate cake:

$$\forall i\,(h_{ch,1} = 8 \wedge h_{i,1} = 11 \wedge h_{i,3} \neq 0 \wedge h_{i,3} \neq \text{choco} \supset \mathit{cake} = \text{choco})$$

- if a cake is to be assigned to the child who was born in November, and the child who was born in August did not order chocolate cake, then *cake* must be chocolate cake:

$$\forall i\,(h_{ch,1} = 11 \wedge h_{i,1} = 8 \wedge h_{i,3} \neq 0 \wedge h_{i,3} \neq \text{choco} \supset \mathit{cake} = \text{choco})$$

- almond cake must not be assigned to the child whose favorite game is the carousel:

$$\mathit{cake} = \text{almond} \supset h_{ch,2} \neq \text{carousel}$$

The application of the Cake(ch, $cake$) operator to state

$$h = \begin{pmatrix} h_{1,1} & h_{2,1} & h_{3,1} & h_{4,1} & h_{5,1} \\ h_{1,2} & h_{2,2} & h_{3,2} & h_{4,2} & h_{5,2} \\ h_{1,3} & h_{2,3} & h_{3,3} & h_{4,3} & h_{5,3} \end{pmatrix} \in \mathcal{S}$$

results in a new state

$$h' = \begin{pmatrix} h'_{1,1} & h'_{2,1} & h'_{3,1} & h'_{4,1} & h'_{5,1} \\ h'_{1,2} & h'_{2,2} & h'_{3,2} & h'_{4,2} & h'_{5,2} \\ h'_{1,3} & h'_{2,3} & h'_{3,3} & h'_{4,3} & h'_{5,3} \end{pmatrix} \in \mathcal{S}$$

where

$$h'_{i,j} = \begin{cases} \mathit{cake} & \text{if } i = ch \wedge j = 3, \\ h_{i,j} & \text{otherwise.} \end{cases}$$

By defining the $\mathcal{S}$ state space, the *start* initial state, the $\mathcal{G}$ set of goal states, and the $\mathcal{O}$ set of operators, we have given a possible $p = \langle \mathcal{S}, \mathit{start}, \mathcal{G}, \mathcal{O} \rangle$ state-space representation of our problem.

The C# Implementation

Implementing the operators is easy, here is the source code:

```
1   using StateSpace;
2
3   namespace FunFair
4   {
5     enum Child
6     {
7       Alex, Carol, Paula, Robert, Tim
8     }
```

```
 9
10      enum GameType
11      {
12        Dodgem = 1, EnchantedCastle, RollerCoaster, Carousel, ScenicRailway
13      }
14
15      enum CakeType
16      {
17        Choco = 1, Walnut, Caramel, Almond, Vanilla
18      }
19
20      class FunFairMonth : Operator
21      {
22        internal Child Child { get; private set; }
23        internal int Month { get; private set; }
24
25        public FunFairMonth(Child child, int month)
26        {
27          Child = child;
28          Month = month;
29        }
30
31        public override string ToString()
32        {
33          return "Month[ child=" + Child + ", month=" + Month + " ]";
34        }
35      }
36
37      class FunFairGame : Operator
38      {
39        internal Child Child { get; private set; }
40        internal GameType Game { get; private set; }
41
42        public FunFairGame(Child child, GameType game)
43        {
44          Child = child;
45          Game = game;
46        }
47
48        public override string ToString()
49        {
50          return "Game[ child=" + Child + ", game=" + Game + " ]";
51        }
52      }
53
54      class FunFairCake : Operator
55      {
56        internal Child Child { get; private set; }
57        internal CakeType Cake { get; private set; }
```

```
58
59      public FunFairCake(Child child, CakeType cake)
60      {
61        Child = child;
62        Cake = cake;
63      }
64
65      public override string ToString()
66      {
67        return "Cake[ child=" + Child + ", cake=" + Cake + " ]";
68      }
69    }
70  }
```

The types Child, GameType, and CakeType are defined as enumeration types. Birthday months are stored as integers because I found unnecessary to work with the names of the months instead of their numbers. It is important that no identifiers with a value of 0 exist in GameType and CakeType because zero will be used to denote that no game or cake is assigned to a child yet. That is why the value 1 is assigned to the first identifier in both enumeration types. The other option would be to introduce a new identifier (e.g., None) as the first one with an underlying value of 0, but then, care should be taken to exclude this value when constructing the Operators collection (see the second F# implementation).

All three operator classes have two properties representing the parameters of the operators. They also have a constructor and a ToString method, which is fairly simple because we use the identifiers of the values defined in the enumeration types as their string representations.

The states of the problem are represented by the FunFairState class:

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Text;
 4  using StateSpace;
 5
 6  namespace FunFair
 7  {
 8    enum Category
 9    {
10      Month, Game, Cake
11    }
12
13    class FunFairState : State
14    {
15      const int N = 5, M = 3;
16
17      const int MONTH = (int)Category.Month;
18      const int GAME  = (int)Category.Game;
19      const int CAKE  = (int)Category.Cake;
```

```
20
21        const int ALEX   = (int)Child.Alex;
22        const int CAROL  = (int)Child.Carol;
23        const int PAULA  = (int)Child.Paula;
24        const int ROBERT = (int)Child.Robert;
25        const int TIM    = (int)Child.Tim;
26
27        const int DODGEM   = (int)GameType.Dodgem;
28        const int CASTLE   = (int)GameType.EnchantedCastle;
29        const int ROLLER   = (int)GameType.RollerCoaster;
30        const int CAROUSEL = (int)GameType.Carousel;
31        const int SCENIC   = (int)GameType.ScenicRailway;
32
33        const int CHOCO   = (int)CakeType.Choco;
34        const int WALNUT  = (int)CakeType.Walnut;
35        const int CARAMEL = (int)CakeType.Caramel;
36        const int ALMOND  = (int)CakeType.Almond;
37        const int VANILLA = (int)CakeType.Vanilla;
38
39        static FunFairState()
40        {
41          Operators = new HashSet<Operator>();
42          foreach (Child child in Enum.GetValues(typeof(Child)))
43          {
44            foreach (int month in new int[] {3, 4, 7, 8, 11})
45              Operators.Add(new FunFairMonth(child, month));
46            foreach (GameType game in Enum.GetValues(typeof(GameType)))
47              Operators.Add(new FunFairGame(child, game));
48            foreach (CakeType cake in Enum.GetValues(typeof(CakeType)))
49              Operators.Add(new FunFairCake(child, cake));
50          }
51        }
52
53        int[,] h;
54
55        public FunFairState()
56        {
57          h = new int[N, M];
58        }
59
60        FunFairState(FunFairState parent)
61        {
62          h = new int[N, M];
63          Array.Copy(parent.h, h, N * M);
64        }
65
66        public override bool GoalState
67        {
68          get
```

```
 69         {
 70           return h[N - 1, M - 1] != 0;
 71         }
 72       }
 73
 74       public override bool PreCondition(Operator op)
 75       {
 76         if (op is FunFairMonth)
 77         {
 78           FunFairMonth m = (FunFairMonth)op;
 79           int ch = (int)m.Child;
 80           int month = m.Month;
 81           if (h[ch, MONTH] != 0)
 82             return false;
 83           if (ch != 0 && h[ch - 1, MONTH] == 0)
 84             return false;
 85           for (int i = 0; i < ch; ++i)
 86             if (h[i, MONTH] == month)
 87               return false;
 88           if (ch == CAROL && month == 3)
 89             return false;
 90           if (ch == PAULA && month == 3)
 91             return false;
 92           if ((h[PAULA, MONTH] == 4 || h[PAULA, MONTH] == 11) &&
 93             ch == TIM && month != 4 && month != 11)
 94             return false;
 95           if ((h[PAULA, MONTH] == 3 || h[PAULA, MONTH] == 7 ||
 96             h[PAULA, MONTH] == 8) && ch == TIM &&
 97             !(month != 4 && month != 11))
 98             return false;
 99           if (ch == CAROL && month != 3 && month != 4)
100             return false;
101           for (int i = 0; i < N; ++i)
102             if (month == 3 && h[i, MONTH] == 4 && i != CAROL &&
103               ch != CAROL)
104               return false;
105           for (int i = 0; i < N; ++i)
106             if (month == 4 && h[i, MONTH] == 3 && i != CAROL &&
107               ch != CAROL)
108               return false;
109           return true;
110         }
111         else if (op is FunFairGame)
112         {
   ⋮
144           return true;
145         }
```

```
146         else if (op is FunFairCake)
147         {
      ⋮
199           return true;
200         }
201         else
202           throw new InvalidOperatorException();
203       }
204
205       public override State Apply(Operator op)
206       {
207         if (op is FunFairMonth)
208         {
209           FunFairMonth month = (FunFairMonth)op;
210           FunFairState newState = new FunFairState(this);
211           newState.h[(int)month.Child, MONTH] = month.Month;
212           return newState;
213         }
214         else if (op is FunFairGame)
215         {
216           FunFairGame game = (FunFairGame)op;
217           FunFairState newState = new FunFairState(this);
218           newState.h[(int)game.Child, GAME] = (int)game.Game;
219           return newState;
220         }
221         else if (op is FunFairCake)
222         {
223           FunFairCake cake = (FunFairCake)op;
224           FunFairState newState = new FunFairState(this);
225           newState.h[(int)cake.Child, CAKE] = (int)cake.Cake;
226           return newState;
227         }
228         else
229           throw new InvalidOperatorException();
230       }
231
232       public override bool Equals(object obj)
233       {
234         return obj is FunFairState && ToString() == obj.ToString();
235       }
236
237       public override int GetHashCode()
238       {
239         return h.GetHashCode();
240       }
241
242       public override string ToString()
```

```
243      {
244        StringBuilder sb = new StringBuilder("FunFairState[ h=(\n");
245        for (int ch = 0; ch < N; ++ch)
246          sb.AppendFormat("{0,-16}", (Child)ch);
247        sb.Append('\n');
248        for (int ch = 0; ch < N; ++ch)
249          sb.AppendFormat("{0,-16}", h[ch, MONTH]);
250        sb.Append('\n');
251        for (int ch = 0; ch < N; ++ch)
252          sb.AppendFormat("{0,-16}", (GameType)h[ch, GAME]);
253        sb.Append('\n');
254        for (int ch = 0; ch < N; ++ch)
255          sb.AppendFormat("{0,-16}", (CakeType)h[ch, CAKE]);
256        return sb.Append("\n) ]").ToString();
257      }
258    }
259  }
```

First, a number of named constants are defined to make the code more readable. Then, in the static constructor, the `Operators` static property is initialized with a `HashSet` containing all the 75 operators relevant to the problem.

The only field of the class (not counting the named constants) is `h`. For the sake of simplicity, it is a 5×3 matrix of integers. It is easier to refer to the elements using two indexes (the first for the child, the second for the category) than finding the appropriate element in a 15-tuple (represented by a 15-element vector or list). The type of elements is `int`, which means that the months, the games, and the types of cake are all represented by integers. We could also use elements of `Object` type, but then, a lot of type conversions would be necessary: months to `int`, games to `GameType`, and types of cake to `CakeType`. This is why the type `int` seemed to be more convenient, and this is why I defined the named constants to ease the work with the matrix elements. The values associated with the elements of the `Child` and `Category` enumeration types are of great importance because they are used as indexes of the matrix. In our case, all of the declared identifiers have default values, which means that the first identifier has a value of 0, and the value of all other identifiers is one greater than that of the previous one. This is exactly what we need when it comes to indexing an array. The drawback of using a matrix as the type of `h` is that we lose compile-time type checking for the matrix elements because of arrays being homogeneous data types.

There are two constructors in the class: the public constructor creates the initial state, with all elements set to 0 in the matrix implicitly, while the private constructor is used by the `Apply` method to clone an existing state by copying the original matrix to the new state. The `GoalState` property is pretty simple in this representation: if the lower right element of the matrix is already assigned a value, then each element is assigned a value that satisfies all preconditions, so the state is a goal state.

The type of this problem implies that `PreCondition` is a rather complex method with a lot of conditions. In our case, the representation lists 40 formulae as operator preconditions, some of which contain universal quantifiers. The body of the method uses the same skeleton as in the previous subsection: the runtime type of the `op`

parameter is checked and the appropriate block is run unless the operator is not relevant to our problem, in which case an exception is thrown. Instead of using one compound condition for each operator, I broke it up to smaller parts, just like in the representation. This way, I got 40 `if` statements, with some of them embedded in a `for` loop.

Since almost all of the formulae have an implication as their main logical operator, and C# does not have such an operator, I used the following logical law to implement it:

$$A \supset B \equiv \neg A \vee B$$

or rather its negated counterpart:

$$\neg(A \supset B) \equiv A \wedge \neg B$$

The negated implication is necessary because each formula is checked one after the other, and if either one is false, then the whole precondition is false, so we can return false. The precondition is satisfied only if all formulae are true.

Similarly, universal quantification is implemented using a `for` loop which checks for all possible values of the quantified variable (`i`) whether the subformula is false, and if so, it returns false. For this, I used one of De Morgan's laws:

$$\neg\forall i(P(i)) \equiv \exists i(\neg P(i))$$

The `Apply` method has the same skeleton as the `PreCondition` method. Applying an operator to a state involves copying that state and changing the value of one element of the matrix in the new state. The first index of the element to be changed is determined by the `Child` property of the operator, while the second index is determined by the type of the operator.

In this problem, overriding the `Equals` method is not so important, because there are no cycles in the representation graph. We could just as well omit the `Equals` method if we only considered the search algorithms, thus improving efficiency. And finally, the `ToString` method mimics the appearance of the matrices in the representation, printing an additional header with the names of the children above the first row.

The First F# Implementation

Here is the listing of the F# version that uses the same principles as the C# version—with a couple of exceptions:

```
 1  module FunFair
 2
 3  open System
 4  open System.Text
 5  open StateSpace
 6
 7  [<Literal>]
 8  let N = 5
 9  [<Literal>]
10  let M = 3
```

```
11
12  type Category =
13      | Month = 0
14      | Game  = 1
15      | Cake  = 2
16
17  let MONTH = int Category.Month
18  let GAME  = int Category.Game
19  let CAKE  = int Category.Cake
20
21  type Child =
22      | Alex   = 0
23      | Carol  = 1
24      | Paula  = 2
25      | Robert = 3
26      | Tim    = 4
27
28  let ALEX   = int Child.Alex
29  let CAROL  = int Child.Carol
30  let PAULA  = int Child.Paula
31  let ROBERT = int Child.Robert
32  let TIM    = int Child.Tim
33
34  type GameType =
35      | Dodgem          = 1
36      | EnchantedCastle = 2
37      | RollerCoaster   = 3
38      | Carousel        = 4
39      | ScenicRailway   = 5
40
41  let DODGEM   = int GameType.Dodgem
42  let CASTLE   = int GameType.EnchantedCastle
43  let ROLLER   = int GameType.RollerCoaster
44  let CAROUSEL = int GameType.Carousel
45  let SCENIC   = int GameType.ScenicRailway
46
47  type CakeType =
48      | Choco   = 1
49      | Walnut  = 2
50      | Caramel = 3
51      | Almond  = 4
52      | Vanilla = 5
53
54  let CHOCO   = int CakeType.Choco
55  let WALNUT  = int CakeType.Walnut
56  let CARAMEL = int CakeType.Caramel
57  let ALMOND  = int CakeType.Almond
58  let VANILLA = int CakeType.Vanilla
59
```

```
60  type Month(child : Child, month : int) =
61      inherit Operator()
62
63      member this.Child = child
64      member this.Month = month
65
66      override this.ToString() =
67          sprintf "Month[ child=%O, month=%d ]" child month
68
69  type Game(child : Child, game : GameType) =
70      inherit Operator()
71
72      member this.Child = child
73      member this.Game = game
74
75      override this.ToString() =
76          sprintf "Game[ child=%O, game=%O ]" child game
77
78  type Cake(child : Child, cake : CakeType) =
79      inherit Operator()
80
81      member this.Child = child
82      member this.Cake = cake
83
84      override this.ToString() =
85          sprintf "Cake[ child=%O, cake=%O ]" child cake
86
87  type FunFairState() =
88      inherit State()
89
90      static do
91          for ch in Enum.GetValues(typeof<Child>) do
92              let child = downcast ch
93              for month in [3; 4; 7; 8; 11] do
94                  State.Operators.Add(Month(child, month)) |> ignore
95              for game in Enum.GetValues(typeof<GameType>) do
96                  State.Operators.Add(Game(child, downcast game)) |> ignore
97              for cake in Enum.GetValues(typeof<CakeType>) do
98                  State.Operators.Add(Cake(child, downcast cake)) |> ignore
99
100     let h = Array2D.zeroCreate N M
101
102     member private this.H = h
103
104     private new(parent : FunFairState) as this =
105         FunFairState() then
106             Array.Copy(parent.H, this.H, N * M)
107
108     override this.GoalState =
```

```
109             h.[N - 1, M - 1] <> 0
110
111         override this.PreCondition(op) =
112             let inline (=>) antecedent consequent =
113                 if antecedent then consequent else true
114
115             let checkForAll cond =
116                 let rec checkForAll cond ch =
117                     ch = N || (cond ch) && checkForAll cond (ch + 1)
118                 checkForAll cond 0
119
120             match op with
121                 | :? Month as month ->
122                     let ch = int month.Child
123                     let month = month.Month
124                     h.[ch, MONTH] = 0
125                         && ch <> 0 => (ch <> 0 && h.[ch - 1, MONTH] <> 0)
126                         && checkForAll (fun i ->
127                             i < ch => (h.[i, MONTH] <> month))
128                         && ch = CAROL => (month <> 3)
129                         && ch = PAULA => (month <> 3)
130                         && ((h.[PAULA, MONTH] = 4 || h.[PAULA, MONTH] = 11)
131                             && ch = TIM) => (month = 4 || month = 11)
132                         && ((h.[PAULA, MONTH] = 3 || h.[PAULA, MONTH] = 7 ||
133                              h.[PAULA, MONTH] = 8) && ch = TIM) =>
134                             (month <> 4 && month <> 11)
135                         && ch = CAROL => (month = 3 || month = 4)
136                         && checkForAll (fun i ->
137                             (month = 3 && h.[i, MONTH] = 4 &&
138                                 i <> CAROL) => (ch = CAROL))
139                         && checkForAll (fun i ->
140                             (month = 4 && h.[i, MONTH] = 3 &&
141                                 i <> CAROL) => (ch = CAROL))
142                 | :? Game as game ->
       ⋮
163                 | :? Cake as cake ->
       ⋮
199                 | _ ->
200                     raise InvalidOperator
201
202         override this.Apply(op) =
203             match op with
204                 | :? Month as month ->
205                     let newState = FunFairState(this)
206                     newState.H.[int month.Child, MONTH] <- month.Month
```

```
207                 upcast newState
208             | :? Game as game ->
209                 let newState = FunFairState(this)
210                 newState.H.[int game.Child, GAME] <- int game.Game
211                 upcast newState
212             | :? Cake as cake ->
213                 let newState = FunFairState(this)
214                 newState.H.[int cake.Child, CAKE] <- int cake.Cake
215                 upcast newState
216             | _ ->
217                 raise InvalidOperator
218
219     override this.Equals(other) =
220         match other with
221             | :? FunFairState as otherFunFairState ->
222                 this.H = otherFunFairState.H
223             | _ ->
224                 false
225
226     override this.GetHashCode() =
227         hash h
228
229     override this.ToString() =
230         let sb = StringBuilder("FunFairState[ h=(\n")
231         for ch in 0 .. N - 1 do
232             sb.AppendFormat("{0,-16}", enum<Child> ch) |> ignore
233         sb.Append('\n') |> ignore
234         for ch in 0 .. N - 1 do
235             sb.AppendFormat("{0,-16}", h.[ch, MONTH]) |> ignore
236         sb.Append('\n') |> ignore
237         for ch in 0 .. N - 1 do
238             sb.AppendFormat("{0,-16}", enum<GameType> h.[ch, GAME])
239                 |> ignore
240         sb.Append('\n') |> ignore
241         for ch in 0 .. N - 1 do
242             sb.AppendFormat("{0,-16}", enum<CakeType> h.[ch, CAKE])
243                 |> ignore
244         sb.Append("\n) ]").ToString()
```

Some notable differences between this F# code and the C# code are the following:

- Enumeration types in F# are discriminated unions in which all identifiers must be assigned a value explicitly. I used the same values here as in the C# version, although some of them are implicitly assigned by the C# compiler.
- In contrast to the C# version, the `PreCondition` method uses one complex formula as the precondition of each operator. We can do this because we now use the `checkForAll` recursive function instead of `for` loops to implement universal quantifications. The argument of this function is a lambda expression representing

the immediate subformula of the quantifier as a function of the quantified variable. Using the `checkForAll` function, all formulae can be easily connected by logical `&&` operators to form one formula as the return value.

- For implementing implication, a new user-defined operator (`=>`) is introduced, making the formulae containing implications more similar to those in the representation. Of course, this also makes the source code of the `PreCondition` method much more succinct. However, we have to be careful when using this operator: as F# uses eager evaluation by default, both the antecedent and the consequent are evaluated before the function is called, i.e., the consequent is evaluated even if the antecedent is false. Consider, for example, the following formula:

  $$ch \neq 1 \supset h_{ch-1,1} \neq 0$$

  This formula cannot be implemented with the expression

  ```
  ch <> 0 => (h.[ch - 1, MONTH] <> 0),
  ```

  because it would cause an exception if `ch` $= 0$. The antecedent must be repeated in the consequent with a short-circuit `&&` operator if we want to avoid this exception:

  ```
  ch <> 0 => (ch <> 0 && h.[ch - 1, MONTH] <> 0)
  ```

  Another alternative is to use lazy evaluation in case of the consequent. As there are only three formulae in our problem affected by this issue, I chose not to use lazy evaluation in each implication but rather repeat the antecedent in these formulae.

- Static type conversion is an area where F# needs more overhead than C#. In particular, F# requires static upcast in some situations where C# does not. You can see such situations in the `Apply` method, where `newState` must be explicitly converted from `FunFairState` to `State`. Additionally, in the static constructor, an extra downcast is necessary when using the `Enum.GetValues` method because in F#, the objects in the array returned by the method are not automatically converted to any type.

- The structural equality check used in the `Equals` method is much more efficient than comparing the string representations of the states as in the C# version. However, as stated earlier, the override of the `Equals` method could also be omitted.

The Second F# Implementation

As I mentioned earlier, the use of a matrix to store the relevant data of our problem has the drawback of losing compile-time type checking of the matrix elements: theoretically, we could assign any integer value to any of the elements, e.g., we could assign chocolate cake as the favorite game of Alex. The second F# implementation differs from the first one mainly in that it uses a five-element vector of structures to store the data of a state, thus providing type safety:

```
1   module FunFair2
2
3   open System
4   open System.Text
5   open StateSpace
6
7   [<Literal>]
8   let N = 5
9
10  type Child =
11      | Alex   = 0
12      | Carol  = 1
13      | Paula  = 2
14      | Robert = 3
15      | Tim    = 4
16
17  let ALEX   = int Child.Alex
18  let CAROL  = int Child.Carol
19  let PAULA  = int Child.Paula
20  let ROBERT = int Child.Robert
21  let TIM    = int Child.Tim
22
23  type GameType =
24      | None            = 0
25      | Dodgem          = 1
26      | EnchantedCastle = 2
27      | RollerCoaster   = 3
28      | Carousel        = 4
29      | ScenicRailway   = 5
30
31  let GNONE    = GameType.None
32  let DODGEM   = GameType.Dodgem
33  let CASTLE   = GameType.EnchantedCastle
34  let ROLLER   = GameType.RollerCoaster
35  let CAROUSEL = GameType.Carousel
36  let SCENIC   = GameType.ScenicRailway
37
38  type CakeType =
39      | None    = 0
40      | Choco   = 1
41      | Walnut  = 2
42      | Caramel = 3
43      | Almond  = 4
44      | Vanilla = 5
45
46  let CNONE   = CakeType.None
47  let CHOCO   = CakeType.Choco
48  let WALNUT  = CakeType.Walnut
49  let CARAMEL = CakeType.Caramel
```

```
50  let ALMOND  = CakeType.Almond
51  let VANILLA = CakeType.Vanilla
52
53  [<Struct>]
54  type ChildData =
55      val mutable month : int
56      val mutable game  : GameType
57      val mutable cake  : CakeType
58
59  type Month(child : Child, month : int) =
60      inherit Operator()
61
62      member this.Child = child
63      member this.Month = month
64
65      override this.ToString() =
66          sprintf "Month[ child=%O, month=%d ]" child month
67
68  type Game(child : Child, game : GameType) =
69      inherit Operator()
70
71      member this.Child = child
72      member this.Game = game
73
74      override this.ToString() =
75          sprintf "Game[ child=%O, game=%O ]" child game
76
77  type Cake(child : Child, cake : CakeType) =
78      inherit Operator()
79
80      member this.Child = child
81      member this.Cake = cake
82
83      override this.ToString() =
84          sprintf "Cake[ child=%O, cake=%O ]" child cake
85
86  type FunFairState() =
87      inherit State()
88
89      static do
90          for ch in Enum.GetValues(typeof<Child>) do
91              let child = downcast ch
92              for month in [3; 4; 7; 8; 11] do
93                  State.Operators.Add(Month(child, month)) |> ignore
94              for game in Enum.GetValues(typeof<GameType>) do
95                  let game = downcast game
96                  if game <> GNONE then
97                      State.Operators.Add(Game(child, game)) |> ignore
98              for cake in Enum.GetValues(typeof<CakeType>) do
```

```
 99                 let cake = downcast cake
100                 if cake <> CNONE then
101                     State.Operators.Add(Cake(child, cake)) |> ignore
102
103     let h = Array.create N (ChildData())
104
105     member private this.H = h
106
107     private new(parent : FunFairState) as this =
108         FunFairState() then
109             Array.Copy(parent.H, this.H, N)
110
111     override this.GoalState =
112         h.[N - 1].cake <> CNONE
113
114     override this.PreCondition(op) =
115         let inline (=>) antecedent consequent =
116             if antecedent then consequent else true
117
118         let checkForAll cond =
119             let rec checkForAll cond ch =
120                 ch = N || (cond ch) && checkForAll cond (ch + 1)
121             checkForAll cond 0
122
123         match op with
124             | :? Month as month ->
125                 let ch = int month.Child
126                 let month = month.Month
127                 h.[ch].month = 0
128                     && ch <> 0 => (ch <> 0 && h.[ch - 1].month <> 0)
129                     && checkForAll (fun i ->
130                         i < ch => (h.[i].month <> month))
131                     && ch = CAROL => (month <> 3)
132                     && ch = PAULA => (month <> 3)
133                     && ((h.[PAULA].month = 4 || h.[PAULA].month = 11) &&
134                         ch = TIM) => (month = 4 || month = 11)
135                     && ((h.[PAULA].month = 3 || h.[PAULA].month = 7 ||
136                          h.[PAULA].month = 8) && ch = TIM) =>
137                         (month <> 4 && month <> 11)
138                     && ch = CAROL => (month = 3 || month = 4)
139                     && checkForAll (fun i ->
140                         (month = 3 && h.[i].month = 4 && i <> CAROL) =>
141                         (ch = CAROL))
142                     && checkForAll (fun i ->
143                         (month = 4 && h.[i].month = 3 && i <> CAROL) =>
144                         (ch = CAROL))
145             | :? Game as game ->
```

⋮

```
166                 | :? Cake as cake ->

        ⋮

201                 | _ ->
202                     raise InvalidOperator
203
204         override this.Apply(op) =
205             match op with
206                 | :? Month as month ->
207                     let newState = FunFairState(this)
208                     newState.H.[int month.Child].month <- month.Month
209                     upcast newState
210                 | :? Game as game ->
211                     let newState = FunFairState(this)
212                     newState.H.[int game.Child].game <- game.Game
213                     upcast newState
214                 | :? Cake as cake ->
215                     let newState = FunFairState(this)
216                     newState.H.[int cake.Child].cake <- cake.Cake
217                     upcast newState
218                 | _ ->
219                     raise InvalidOperator
220
221         override this.Equals(other) =
222             match other with
223                 | :? FunFairState as otherFunFairState ->
224                     this.H = otherFunFairState.H
225                 | _ ->
226                     false
227
228         override this.GetHashCode() =
229             hash h
230
231         override this.ToString() =
232             let sb = StringBuilder("FunFairState[ h=(\n")
233             for ch in 0 .. N - 1 do
234                 sb.AppendFormat("{0,-16}", enum<Child> ch) |> ignore
235             sb.Append('\n') |> ignore
236             for ch in 0 .. N - 1 do
237                 sb.AppendFormat("{0,-16}", h.[ch].month) |> ignore
238             sb.Append('\n') |> ignore
239             for ch in 0 .. N - 1 do
240                 sb.AppendFormat("{0,-16}", h.[ch].game) |> ignore
241             sb.Append('\n') |> ignore
242             for ch in 0 .. N - 1 do
243                 sb.AppendFormat("{0,-16}", h.[ch].cake) |> ignore
244             sb.Append("\n) ]").ToString()
```

Here, I used the `ChildData` structure type to store information related to each child. This way, we can substitute a one-dimensional array of this type for the integer matrix in the previous implementation. As type safety is now provided, two new values had to be introduced in the two enumeration types to take the role of zero: `GNONE` and `CNONE`. Note that these values must be omitted in the static constructor when creating the operator objects. The implicit constructor creates the array with `N` elements, each containing 0 as the month, `GNONE` as the game, and `CNONE` as the type of cake. It is because structures are value types, so they have an implicit default constructor that sets the values of their fields to zero. The explicit constructor creates a shallow copy of the array. If we used a record instead of a structure, we would have to replace the shallow copy with a deep copy because records are reference types. All other parts of the code differ from the corresponding parts of the first implementation only in the way of referring to a specific piece of information about the children. For example, instead of `h.[ALEX, CAKE]`, we have to use `h.[ALEX].cake`.

CHAPTER 4

Some Implementations of Search Algorithms on Game Trees

Besides single-agent problems, artificial intelligence also covers two-player games, and provides search algorithms on game trees. Games, in general, can be classified into two main categories: *gambles*, in which the players do not have influence to the outcome of the game, and *strategy games*, where the outcome of the game is actively affected by the players. Strategy games can be further classified based on the following aspects:

- Considering the number of players, there are *two-player*, *three-player*, ..., *n-player* games.
- Considering the length of the game, there are *finite* games, in which each player can choose from a finite set of moves, and each game terminates in a finite number of moves. Games that are not finite are called *infinite* games.
- Considering the sum of the players' gains and losses, there are *zero-sum* and *non-zero-sum* games. In zero-sum games, the sum of the players' gains and losses is zero.
- If a game has random factors, it is called *stochastic*, otherwise *deterministic*.
- In a game with *perfect information*, the players have all information related to the game at their disposal. A game with *imperfect information* does not have perfect information.

The algorithms presented in this chapter are capable of computing the next move in arbitrary state-space-represented, finite, deterministic, zero-sum, two-player strategy games with perfect information. (From now on, I will refer to such games shortly as two-player games.) The implemented algorithms are the following [18]:

- minimax algorithm
- negamax algorithm
- the above algorithms with alpha-beta pruning

As in the previous chapter, I present first the class hierarchy serving as a base for the C# implementation. Then, I give three implementations of the aforementioned algorithms: one in C# and two in F# with different amount of functional elements. I will use *Nim*, a simple two-player game, as an example to demonstrate how these implementations can be applied to play a particular game.

4.1 A Class Hierarchy for Search Algorithms

Figure 4.1 shows a UML class diagram containing all classes related to the implementation of the state-space representation, the game tree search algorithms, and the control of the game.

The `State` and `Operator` abstract classes are similar to those used in the case of single-agent search algorithms. The `Operator` class has no members, as the cost of operator applications is usually not relevant in two-player games. The `Operators`, `PreCondition`, and `Apply` members of the `State` class have exactly the same role here as in the state-space representation used by single-agent path-finding algorithms. The additional members are the following:

- `Player`: the player in turn in the current state, represented by a single character (A or B). Player A is always the starting player, who makes the opening move.
- `SwitchPlayer`: switches the player in turn in the current state, typically invoked by the `Apply` method as the final step of an operator application.
- `EndState`: true if the current state is an end state, i.e., the game has come to an end.
- `AWon` and `BWon`: true if the current state is an end state, and the game is won by player A or player B, respectively. If `EndState` is true, but neither `AWon` nor `BWon` is true, then the game is a tie.
- `MinimaxGoodness` and `NegamaxGoodness`: the goodness value of the current state to be used by the minimax and negamax algorithms, respectively. A positive number represents a "good state" for player A (in case of minimax algorithm) or the player in turn (in case of negamax algorithm). A negative number represents a "bad state", and zero represents an even position. Typically, `MinimaxGoodness` and `NegamaxGoodness` are equal if the player in turn is player A (i.e., the starting player), otherwise, they are the negation of each other.
- `ReadMove`: reads the next move of a human player from the standard input and returns the corresponding `Operator` object.

`GameProp` is an enumeration type used for setting the game properties. It contains the following flags:

- `AgainstHumanFlag`: if set, two human players will play against each other, the program will just control the game and possibly give hints to the players. If not set, a human player will play against the computer.

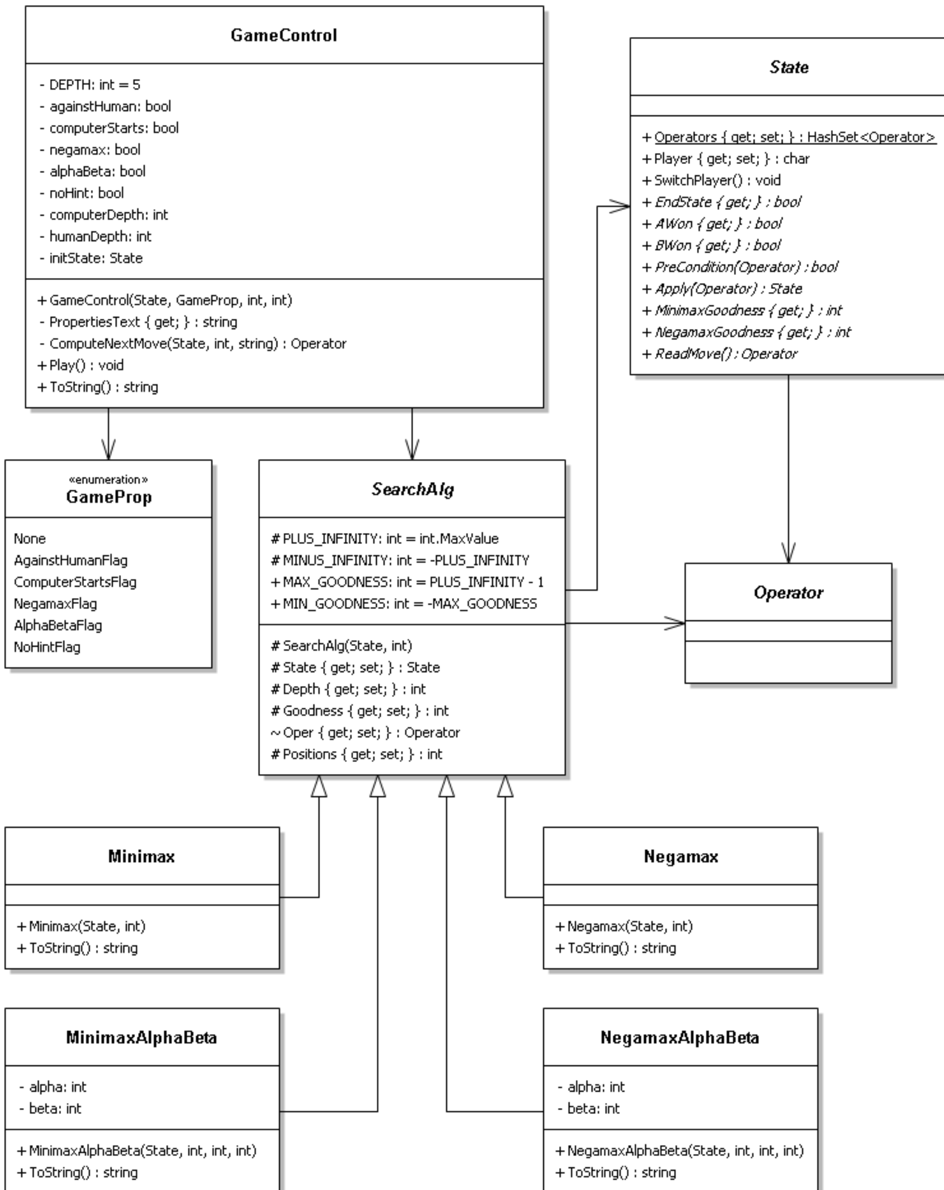


Figure 4.1. Classes representing state space, search algorithms, and game control.

- `ComputerStartsFlag`: if set, the computer will start the game, otherwise, the human player is the starting player. Relevant only if `AgainstHumanFlag` is set.

- `NegamaxFlag`: if set, the search algorithm used by the computer as a player and for hints will be the negamax algorithm instead of the default minimax algorithm.

- `AlphaBetaFlag`: if set, alpha-beta pruning will be used during the search.

- `NoHintFlag`: if set, the program will not give hints to the human player(s).

As the name suggests, `GameControl` is the class responsible for controlling the game. Its constructor takes the following arguments:

- the initial state

- game properties (as a combination of flags)

- a depth value used by the computer as a player

- a depth value used when computing a hint for the human player

The initial state is mandatory, the other three are optional arguments. By default, there are no flags set, and both depth values are 5. `PropertiesText` gives the string representation of the game properties; it is used by the `ToString` method. `ComputeNextMove` computes the next move for the computer as a player or for giving a hint to the human player. It takes the current state as its first argument, and uses one of the four search algorithms to build a part of the game tree with the current node as root and choose the most promising next move. The method returns the operator corresponding to this move. The depth in which the game tree is to be explored by the search algorithm is controlled by the second argument. After instantiating a new `GameControl` object, the `Play` method should be called to start the game.

Each search algorithm is represented by a class derived from the abstract `SearchAlg` class, which has the following members:

- `PLUS_INFINITY` and `MINUS_INFINITY`: these two integer constants are used internally as initial values for selecting the minimum or maximum of goodness values that belong to the children of a particular node in the game tree. By default, `PLUS_INFINITY` is `int.MaxValue` and `MINUS_INFINITY` is the negation of it. Note that we cannot use `int.MinValue` as `MINUS_INFINITY`, because the negamax algorithm may negate this value, which would cause an arithmetic overflow.

- `MAX_GOODNESS` and `MIN_GOODNESS`: these integers mark the lower and upper bounds between which all concrete implementations of `MinimaxGoodness` and `NegamaxGoodness` should return a value. By default, they have an absolute value one less than `PLUS_INFINITY`, ensuring that they do not interfere with `PLUS_INFINITY` and `MINUS_INFINITY` during minimum and maximum selection.

- `State`: the current game state represented by the current node in the game tree.

- `Depth`: the depth value in which the game tree is to be explored from the current node.

- `Goodness`: the goodness value of the current state after the relevant part of the game tree (with the current node as root) has been fully explored.
- `Oper`: the operator that leads to the state to which `Goodness` is assigned, i.e., the operator chosen as the next move.
- `Positions`: the total number of positions evaluated during graph exploration; used mainly for debugging purposes.

As you can see, a `SearchAlg` object represents not only a particular search algorithm but also a node in the game tree as well as the explored part of the game tree with the current node as root. `State` and `Depth` may be considered as input data, whereas `Goodness`, `Oper`, and `Positions` as output data, which may be queried after the search has finished.

The four concrete algorithm classes are very similar to one another. As you can see in Section 4.2, they all have a recursive constructor, which actually does the search. `MinimaxAlphaBeta` and `NegamaxAlphaBeta` have two additional private fields for storing the current alpha and beta values, but they are only required by the `ToString` method if we want them to be part of the string representation of the algorithm objects.

4.2 Various Implementations of Search Algorithms

This section covers three possible implementations of two algorithms: minimax algorithm without alpha-beta pruning and negamax algorithm with alpha-beta pruning. The C# version conforms to the class diagram presented in the previous section, while the two F# versions contain multiparadigm code in two different approaches.

4.2.1 The C# Implementation

Let's see first the implementation of the abstract classes constituting the state-space representation:

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace StateSpace
5   {
6     public abstract class Operator
7     {
8     }
9
10    public class InvalidOperatorException : Exception
11    {
12      public InvalidOperatorException()
13        : base("No such operator!")
14      {
```

```
15      }
16    }
17
18    public abstract class State
19    {
20      public static ICollection<Operator> Operators { get; set; }
21      public char Player { get; set; }
22
23      public void SwitchPlayer()
24      {
25        Player = Player == 'A' ? 'B' : 'A';
26      }
27
28      public abstract bool EndState { get; }
29      public abstract bool AWon { get; }
30      public abstract bool BWon { get; }
31      public abstract bool PreCondition(Operator op);
32      public abstract State Apply(Operator op);
33      public abstract int MinimaxGoodness { get; }
34      public abstract int NegamaxGoodness { get; }
35      public abstract Operator ReadMove();
36    }
37  }
```

These three classes resemble those with the same names presented in Section 3.2. The main difference is in the members of the `State` class. As the `Player` property is common to all concrete two-player games, it is more convenient to be defined here. So is `SwitchPlayer`, a very simple method, used to switch the player in turn in the current state. `EndState`, `AWon`, `BWon`, `MinimaxGoodness`, and `NegamaxGoodness` are defined as abstract properties, while `ReadMove` is an abstract method.

The implementation of the four search algorithms follows:

```
1   using System;
2   using StateSpace;
3
4   namespace Game
5   {
6     abstract class SearchAlg
7     {
8       protected const int PLUS_INFINITY = int.MaxValue;
9       protected const int MINUS_INFINITY = -PLUS_INFINITY;
10      public const int MAX_GOODNESS = PLUS_INFINITY - 1;
11      public const int MIN_GOODNESS = -MAX_GOODNESS;
12
13      protected State State { get; set; }
14      protected int Depth { get; set; }
15      protected int Goodness { get; set; }
16      internal Operator Oper { get; set; }
17      protected int Positions { get; set; }
```

```
18
19      protected SearchAlg(State state, int depth)
20      {
21        State = state;
22        Depth = depth;
23        Positions = 1;
24      }
25    }
26
27    class Minimax : SearchAlg
28    {
29      public Minimax(State state, int depth)
30        : base(state, depth)
31      {
32        if (state.EndState || depth == 0)
33          Goodness = state.MinimaxGoodness;
34        else
35        {
36          Goodness = state.Player == 'A' ? MINUS_INFINITY : PLUS_INFINITY;
37          foreach (Operator op in State.Operators)
38            if (state.PreCondition(op))
39            {
40              State newState = state.Apply(op);
41              Minimax newAlg = new Minimax(newState, depth - 1);
42              bool betterState = state.Player == 'A' ?
43                newAlg.Goodness > Goodness :
44                newAlg.Goodness < Goodness;
45              if (betterState)
46              {
47                Goodness = newAlg.Goodness;
48                Oper = op;
49              }
50              Positions += newAlg.Positions;
51            }
52        }
53      }
54
55      public override string ToString()
56      {
57        return "Minimax[ state=" + State + ", depth=" + Depth +
58               ", operator=" + Oper + ", goodness=" + Goodness +
59               ", number of evaluated positions=" + Positions + " ]";
60      }
61    }
62
63    class NegamaxAlphaBeta : SearchAlg
64    {
65      int alpha, beta;
66
```

```
67        public NegamaxAlphaBeta(State state, int depth,
68          int _alpha = MINUS_INFINITY, int _beta = PLUS_INFINITY)
69          : base(state, depth)
70        {
71          alpha = _alpha;
72          beta = _beta;
73          if (state.EndState || depth == 0)
74            Goodness = state.NegamaxGoodness;
75          else
76          {
77            foreach (Operator op in State.Operators)
78            {
79              if (alpha >= beta)
80                break;
81              if (state.PreCondition(op))
82              {
83                State newState = state.Apply(op);
84                NegamaxAlphaBeta newAlg =
85                  new NegamaxAlphaBeta(newState, depth - 1, -beta, -alpha);
86                if (-newAlg.Goodness > alpha)
87                {
88                  alpha = -newAlg.Goodness;
89                  Oper = op;
90                }
91                Positions += newAlg.Positions;
92              }
93            }
94            Goodness = alpha;
95          }
96        }
97
98        public override string ToString()
99        {
100         return "NegamaxAlphaBeta[ state=" + State + ", depth=" + Depth +
101                ", operator=" + Oper + ", goodness=" + Goodness +
102                ", alpha=" + alpha + ", beta=" + beta +
103                ", number of evaluated positions=" + Positions + " ]";
104       }
105     }
106   }
```

The `SearchAlg` abstract class has only four constant fields and five auto-implemented properties; there are no methods common to the search algorithms. The constructor initializes `Positions` to 1, which represents the root node of the subtree to be explored. `State` and `Depth` are also initialized, although they are only used in the `ToString` methods of the concrete algorithm classes.

As you can see, the presented two algorithm classes have the same structure: they have a recursive constructor and a `ToString` method. In case of alpha-beta pruning, two additional private fields are defined (`alpha` and `beta`), but only for debugging

purposes. They can be omitted if we do not want to print their values in the `ToString` method.

If we consider recursion as a functional programming element, then this code is multiparadigm in itself. Exploring the game tree means that for each child node, a new object is instantiated with the child node as root and with one less depth than in the parent. If the current node is an end node or `depth` reaches zero, then recursion is stopped, and the goodness value is set to whatever the evaluation function returns. Otherwise, a minimum or maximum selection is performed among the goodness values of each child node. `Oper` is set to the operator which results in the child node with the best goodness value, and `Positions` will be the total number of nodes explored from the current node as root (unless there is more than one path to reach a node, in which case it will be counted multiple times).

The code listing clearly shows the difference between minimax and negamax algorithms. Negamax is somewhat simpler because we do not have to check which player is in turn in the current state—it works the same way in both cases. However, in games where players can make several successive moves, negamax algorithm cannot be used. For negamax algorithm to work, the game tree must be such that the distance from the root to a node in which player A is in turn is even, and the distance from the root to a node in which player B is in turn is odd. Minimax algorithm does not have this restriction, because it works differently if player A is in turn or player B.

Alpha-beta pruning improves minimax and negamax algorithms by breaking the `foreach` loop that traverses the children of the current node for finding the best move when it turns out that there is no use exploring the remaining children.

Finally, here is the implementation of the `GameControl` class:

```
 1  using System;
 2  using StateSpace;
 3
 4  namespace Game
 5  {
 6    [Flags]
 7    public enum GameProp : byte
 8    {
 9      None               = 0,
10      AgainstHumanFlag   = 1,
11      ComputerStartsFlag = 2,
12      NegamaxFlag        = 4,
13      AlphaBetaFlag      = 8,
14      NoHintFlag         = 16
15    }
16
17    public class GameControl
18    {
19      const int DEPTH = 5;
20
21      bool againstHuman, computerStarts, negamax, alphaBeta, noHint;
22      int computerDepth, humanDepth;
```

```
23      State initState;
24
25      public GameControl(State initState,
26                         GameProp properties = GameProp.None,
27                         int computerDepth = DEPTH, int humanDepth = DEPTH)
28      {
29        this.initState = initState;
30        this.computerDepth = computerDepth;
31        this.humanDepth = humanDepth;
32        againstHuman   = (properties & GameProp.AgainstHumanFlag)
33                         != GameProp.None;
34        computerStarts = (properties & GameProp.ComputerStartsFlag)
35                         != GameProp.None;
36        negamax        = (properties & GameProp.NegamaxFlag)
37                         != GameProp.None;
38        alphaBeta      = (properties & GameProp.AlphaBetaFlag)
39                         != GameProp.None;
40        noHint         = (properties & GameProp.NoHintFlag)
41                         != GameProp.None;
42      }
43
44      private string PropertiesText
45      {
46        get
47        {
48          string s = "";
49          if (againstHuman)
50            s += "A human plays against a human.\n";
51          else
52          {
53            s += "A human plays against the computer.\n";
54            if (computerStarts)
55              s += "The computer starts the game.\n";
56            else
57              s += "The human starts the game.\n";
58          }
59          if (negamax)
60            s += "Searching using the negamax algorithm.\n";
61          else
62            s += "Searching using the minimax algorithm.\n";
63          if (alphaBeta)
64            s += "Using alpha-beta pruning.\n";
65          else
66            s += "Not using alpha-beta pruning.\n";
67          if (noHint)
68            s += "No hints.\n";
69          else
70            s += "With hints.\n";
71          return s;
```

```
72       }
73     }
74
75     private Operator ComputeNextMove(State state, int depth, string text)
76     {
77       SearchAlg searchAlg;
78       if (!negamax && !alphaBeta)
79         searchAlg = new Minimax(state, depth);
80       else if (negamax && !alphaBeta)
81         searchAlg = new Negamax(state, depth);
82       else if (!negamax && alphaBeta)
83         searchAlg = new MinimaxAlphaBeta(state, depth);
84       else
85         searchAlg = new NegamaxAlphaBeta(state, depth);
86       Console.WriteLine("Current search algorithm: " + searchAlg + "\n");
87       Console.WriteLine(text + ": " + searchAlg.Oper + "\n");
88       return searchAlg.Oper;
89     }
90
91     public void Play()
92     {
93       State state = initState;
94       while (!state.EndState)
95       {
96         Operator op;
97         Console.WriteLine("Current state: " + state);
98         if (!againstHuman && (state.Player == 'A' && computerStarts ||
99                               state.Player == 'B' && !computerStarts))
100          op = ComputeNextMove(state, computerDepth, "Computer's move");
101        else
102        {
103          if (!noHint)
104            ComputeNextMove(state, humanDepth, "Recommended move");
105          else
106            Console.WriteLine();
107          Console.WriteLine("The move of player '" + state.Player +
108                            "':\n");
109          op = state.ReadMove();
110        }
111        state = state.Apply(op);
112      }
113      Console.Write("The game is over. ");
114      if (state.AWon || state.BWon)
115        Console.WriteLine("The game is won by player '" +
116          (state.AWon ? 'A' : 'B') + "'.");
117      else
118        Console.WriteLine("The result is a tie.");
119    }
120
```

```
121      public override string ToString()
122      {
123        return "Two-player game.\n" + PropertiesText;
124      }
125    }
126  }
```

`GameProp` is defined in a way as flags are usually defined: the values assigned to the identifiers are the powers of 2. `DEPTH` is a named constant used by the constructor if the caller omits the third and fourth argument. The constructor sets the initial state of the game, the five game properties, and the look-ahead depths: one for the computer player and one for the hints for the human player. `PropertiesText` gives the string representation of the five game properties, which will be utilized by the `ToString` method. The private `ComputeNextMove` method first determines the search algorithm to be used for finding the next move, based on two of the game properties, then creates an algorithm object of the appropriate type. The constructor of the algorithm object selects the most promising move in the current game state, which is first printed and then returned. The algorithm object itself too is printed after the search for debugging purposes with all the relevant information regarding the search (such as the number of evaluated positions).

The `Play` method is responsible for controlling the game flow. It first initializes the current state with the initial state, then runs a `while` loop until the current state becomes an end state. If it is the computer's turn, the `ComputeNextMove` method is invoked with the appropriate look-ahead depth, otherwise, a move may be recommended for the human player (again, using the `ComputeNextMove` method) unless the hints are disabled, and then the human player's move is read by the `ReadMove` method. Finally, if an end state is reached, the result of the game is printed, and the game is over.

The Main Program

The main program is simpler than those in the previous chapter because now only the `Play` method must be called after instantiating a `GameControl` object with the initial state of a specific game (Nim, in this case):

```
 1  using System;
 2  using Game;
 3
 4  class Program
 5  {
 6    static void Main()
 7    {
 8      GameControl game = new GameControl(new Nim.NimState(),
 9        GameProp.NegamaxFlag | GameProp.AlphaBetaFlag);
10      Console.WriteLine(game);
11      game.Play();
12    }
13  }
```

This sample program will play a game of Nim using the negamax algorithm with alpha-beta pruning. The user will play against the computer, the user will be the starting player, the program will give hints to the user, and it will use the default depth of 5 for computing the next move in each state for both the human player (i.e., for the hints) and itself.

4.2.2 The First F# Implementation

As in the previous chapter, the first F# version is very similar to the C# implementation, though there are some differences. Let's see first the abstract classes `Operator` and `State`:

```
1   namespace StateSpace
2
3   open System.Collections.Generic
4
5   [<AbstractClass>]
6   type Operator() =
7       class
8       end
9
10  [<AbstractClass>]
11  type State(?player) =
12      let player = defaultArg player 'A'
13      static let operators = HashSet<Operator>()
14      static member Operators = operators
15      member this.Player = player
16      member this.OtherPlayer =
17          if player = 'A' then 'B' else 'A'
18      abstract EndState : bool
19      abstract AWon : bool
20      abstract BWon : bool
21      abstract PreCondition : Operator -> bool
22      abstract Apply : Operator -> State
23      abstract MinimaxGoodness : int
24      abstract NegamaxGoodness : int
25      abstract ReadMove : unit -> Operator
26
27  exception InvalidOperator
```

The differences between this code and its C# counterpart are the following:

- The static collection of operators is instantiated right here instead of the concrete class of a game so that it does not have to be mutable.
- The implicit constructor of `State` now has an optional argument (`player`). This way, the `player` field does not have to be mutable, because its value is only "changed" when it is created.

- For the same reason, an `OtherPlayer` property is introduced as a substitution for the `SwitchPlayer` method. It does not change the value of `player` but returns the player that is not the current one. It is used as an argument of the constructor when creating a new state from the current state.

  Here comes the source code of the search algorithms:

```
1   namespace Game
2
3   open StateSpace
4
5   [<AbstractClass>]
6   type SearchAlg(state : State, depth : int) =
7       [<DefaultValue>]
8       val mutable private goodness : int
9       [<DefaultValue>]
10      val mutable private oper : Operator
11
12      let mutable positions = 1
13
14      static member internal PLUS_INFINITY = System.Int32.MaxValue
15      static member internal MINUS_INFINITY = -SearchAlg.PLUS_INFINITY
16      static member MAX_GOODNESS = SearchAlg.PLUS_INFINITY - 1
17      static member MIN_GOODNESS = -SearchAlg.MAX_GOODNESS
18
19      member internal this.State = state
20      member internal this.Depth = depth
21      member internal this.Goodness
22          with get () = this.goodness
23          and set value = this.goodness <- value
24      member internal this.Oper
25          with get () = this.oper
26          and set value = this.oper <- value
27      member internal this.Positions
28          with get () = positions
29          and set value = positions <- value
30
31  type internal Minimax(state, depth) as this =
32      inherit SearchAlg(state, depth)
33
34      do
35          if state.EndState || depth = 0 then
36              this.Goodness <- state.MinimaxGoodness
37          else
38              this.Goodness <- if state.Player = 'A'
39                                   then SearchAlg.MINUS_INFINITY
40                                   else SearchAlg.PLUS_INFINITY
41              State.Operators
42                  |> Seq.filter (fun op -> state.PreCondition(op))
```

```
43                 |> Seq.iter (fun op ->
44                     let newState = state.Apply(op)
45                     let newAlg = Minimax(newState, depth - 1)
46                     let betterState =
47                         if state.Player = 'A'
48                             then newAlg.Goodness > this.Goodness
49                             else newAlg.Goodness < this.Goodness
50                     if betterState then
51                         this.Goodness <- newAlg.Goodness
52                         this.Oper <- op
53                     this.Positions <- this.Positions + newAlg.Positions)
54
55     override this.ToString() =
56         sprintf "Minimax[ state=%O, depth=%d, operator=%O, \
57             goodness=%d, number of evaluated positions=%d ]"
58             this.State this.Depth this.Oper this.Goodness this.Positions
59
60 type internal NegamaxAlphaBeta(state, depth, ?alpha, ?beta) as this =
61     inherit SearchAlg(state, depth)
62
63     let mutable alpha = defaultArg alpha SearchAlg.MINUS_INFINITY
64     let mutable beta = defaultArg beta SearchAlg.PLUS_INFINITY
65
66     do
67         if state.EndState || depth = 0 then
68             this.Goodness <- state.NegamaxGoodness
69         else
70             State.Operators
71                 |> Seq.takeWhile (fun _ -> alpha < beta)
72                 |> Seq.filter (fun op -> state.PreCondition(op))
73                 |> Seq.iter (fun op ->
74                     let newState = state.Apply(op)
75                     let newAlg = NegamaxAlphaBeta(newState, depth - 1,
76                                                   -beta, -alpha)
77                     if -newAlg.Goodness > alpha then
78                         alpha <- -newAlg.Goodness
79                         this.Oper <- op
80                     this.Positions <- this.Positions + newAlg.Positions)
81             this.Goodness <- alpha
82
83     override this.ToString() =
84         sprintf "NegamaxAlphaBeta[ state=%O, depth=%d, operator=%O, \
85             goodness=%d, alpha=%d, beta=%d, \
86             number of evaluated positions=%d ]" this.State this.Depth
87             this.Oper this.Goodness alpha beta this.Positions
```

The abstract `SearchAlg` class has three mutable fields, two of which (`goodness` and `oper`) having no explicit initial values; they are initialized with the default values of their types (0 and `null`, respectively). Besides them, the class contains nine members:

four constant fields, two immutable properties, and three mutable properties. The immutable properties serve as input, the mutable properties serve as output of the search algorithms.

The two algorithm classes are very much like their respective C# counterparts. The only difference is that they use higher-order functions from the `Seq` module to simulate imperative control structures: `takeWhile` is substituted for `break`, `filter` for `if`, and `iter` for `foreach`. Although the mechanisms are different, these functions—applied in chain using the forward pipe (`|>`) operator—give the same result as the imperative language constructs.

And now, let's see the implementation of the `GameControl` class:

```
1   namespace Game
2
3   open System
4   open StateSpace
5
6   [<Flags>]
7   type GameProp =
8       | None              = 0b00000000
9       | AgainstHumanFlag   = 0b00000001
10      | ComputerStartsFlag = 0b00000010
11      | NegamaxFlag        = 0b00000100
12      | AlphaBetaFlag      = 0b00001000
13      | NoHintFlag         = 0b00010000
14
15  type GameControl(initState, ?properties, ?computerDepth, ?humanDepth) =
16      static let DEPTH = 5
17
18      let properties = defaultArg properties GameProp.None
19      let computerDepth = defaultArg computerDepth DEPTH
20      let humanDepth = defaultArg humanDepth DEPTH
21
22      let againstHuman   = properties &&& GameProp.AgainstHumanFlag
23                                <> GameProp.None
24      let computerStarts = properties &&& GameProp.ComputerStartsFlag
25                                <> GameProp.None
26      let negamax        = properties &&& GameProp.NegamaxFlag
27                                <> GameProp.None
28      let alphaBeta      = properties &&& GameProp.AlphaBetaFlag
29                                <> GameProp.None
30      let noHint         = properties &&& GameProp.NoHintFlag
31                                <> GameProp.None
32
33      member internal this.PropertiesText =
34          (if againstHuman then
35               "A human plays against a human.\n"
36           else
37               "A human plays against the computer.\n" +
38               (if computerStarts then
```

```
39                    "The computer starts the game.\n"
40              else
41                    "The human starts the game.\n")) +
42          (if negamax then
43               "Searching using the negamax algorithm.\n"
44           else
45               "Searching using the minimax algorithm.\n") +
46          (if alphaBeta then
47               "Using alpha-beta pruning.\n"
48           else
49               "Not using alpha-beta pruning.\n") +
50          (if noHint then
51               "No hints.\n"
52           else
53               "With hints.\n")
54
55      member this.Play() =
56          let rec doWork (state : State) =
57              let computeNextMove depth text =
58                  let searchAlg : SearchAlg =
59                      if not negamax && not alphaBeta then
60                          upcast Minimax(state, depth)
61                      elif negamax && not alphaBeta then
62                          upcast Negamax(state, depth)
63                      elif not negamax && alphaBeta then
64                          upcast MinimaxAlphaBeta(state, depth)
65                      else
66                          upcast NegamaxAlphaBeta(state, depth)
67                  printfn "Current search algorithm: %O\n" searchAlg
68                  printfn "%s: %O\n" text searchAlg.Oper
69                  searchAlg.Oper
70
71              if state.EndState then
72                  printf "The game is over. "
73                  if state.AWon || state.BWon then
74                      printfn "The game is won by player '%c'."
75                          (if state.AWon then 'A' else 'B')
76                  else
77                      printfn "The result is a tie."
78              else
79                  printfn "Current state: %O" state
80                  if not againstHuman &&
81                      (state.Player = 'A' && computerStarts ||
82                       state.Player = 'B' && not computerStarts) then
83                      let op = computeNextMove computerDepth
84                                   "Computer's move"
85                      doWork (state.Apply(op))
86                  else
87                      if not noHint then
```

```
88                          computeNextMove humanDepth "Recommended move"
89                              |> ignore
90                      else
91                          printfn ""
92                      printfn "The move of player '%c':\n" state.Player
93                      doWork (state.Apply(state.ReadMove()))
94
95          doWork initState
96
97      override this.ToString() =
98          "Two-player game.\n" + this.PropertiesText
```

The major difference from the C# version lies in the `Play` method. While the main control structure in the C# code is a `while` loop, in F#, a recursive inner function (`doWork`) is responsible for the control of the game. The `Play` method itself does only one thing: it calls `doWork` with `initState` as its argument; the rest is done by `doWork`. All recursive functions consist of branches, some of which stop recursion. In our case, `doWork` has three branches. If the parameter state is an end state, the result is printed, and recursion is stopped. The second branch is evaluated if it is the computer's turn in the current state: `computeNextMove` computes the best move for the computer player, the corresponding operator is applied to the current state, and the game is continued by a tail-recursive call with the new state as an argument. If the human player is in turn, the third branch is evaluated. If hints are on, then again `computeNextMove` is called, but its return value is discarded—the recommended move will be printed as a side effect. The `ReadMove` method reads the human player's next move, which is executed, and the game continues again with a tail-recursive call.

`computeNextMove` is now a local function of `doWork`. This way, it needs only two arguments, since `state`, as a parameter of the enclosing function, is also accessible from the inner function. The other notable difference here is the excessive use of the `upcast` operator for converting the various concrete algorithm types to their common base type of `SearchAlg`.

The Main Program

A very simple main function that behaves exactly like the C# main program may look like this:

```
1   open Game
2
3   let main () =
4       let game = GameControl(Nim.NimState(),
5                       GameProp.NegamaxFlag ||| GameProp.AlphaBetaFlag)
6       printfn "%O" game
7       game.Play()
```

4.2.3 The Second F# Implementation

We can make the same modifications to the first F# version as in the previous chapter to make our code more functional:

- We can get rid of the concrete algorithm classes and replace them with functions. As the constructors of these classes were recursive, the replacement functions will be recursive too.
- The `ToString` methods of the concrete algorithm classes can be made inline in the `computeNextMove` function because they were only used once in the code.
- The `GameControl` class can be replaced with a `play` function, which takes the same parameters as the constructor of the `GameControl` class. No extra `algorithm` parameter is required in this case, because the search algorithm to be used is now determined based on the game properties.
- A private enumeration type will be used to deal with the possible search algorithms.
- Two new immutable types will be introduced to store the data used by the search algorithms: one for the minimax and negamax algorithms, and another for the algorithms using alpha-beta pruning. Both the parameter and the return value of each algorithm function will be of these types.

The `StateSpace` namespace remains the same in this implementation as in the first one. Let's see now the listing of the functions implementing the different search algorithms:

```
 1  module Game
 2
 3  open System
 4  open StateSpace
 5
 6  let internal PLUS_INFINITY = Int32.MaxValue
 7  let internal MINUS_INFINITY = -PLUS_INFINITY
 8  let MAX_GOODNESS = PLUS_INFINITY - 1
 9  let MIN_GOODNESS = -MAX_GOODNESS
10
11  type internal SearchAlgData(state : State, depth, goodness,
12                              oper, positions) =
13      member internal this.State = state
14      member internal this.Depth = depth
15      member internal this.Goodness = goodness
16      member internal this.Oper = oper
17      member internal this.Positions = positions
18
19  type internal AlphaBetaData(state, depth, goodness, oper,
20                              positions, alpha, beta) =
21      inherit SearchAlgData(state, depth, goodness, oper, positions)
22
```

```
23        member internal this.Alpha = alpha
24        member internal this.Beta = beta
25
26    let private initMinimaxGoodness (state : State) =
27        if state.Player = 'A' then MINUS_INFINITY else PLUS_INFINITY
28
29    let rec internal minimaxAlg (data : SearchAlgData) =
30        if data.State.EndState || data.Depth = 0 then
31            SearchAlgData(data.State, data.Depth, data.State.MinimaxGoodness,
32                          data.Oper, data.Positions)
33        else
34            State.Operators
35                |> Seq.filter (fun op -> data.State.PreCondition(op))
36                |> Seq.fold (fun acc op ->
37                    let newState = acc.State.Apply(op)
38                    let newAlg =
39                        minimaxAlg
40                            (SearchAlgData(newState, acc.Depth - 1,
41                                 initMinimaxGoodness newState, None, 1))
42                    let newPos = acc.Positions + newAlg.Positions
43                    let betterState = if data.State.Player = 'A'
44                                      then newAlg.Goodness > acc.Goodness
45                                      else newAlg.Goodness < acc.Goodness
46                    if betterState then
47                        SearchAlgData(acc.State, acc.Depth,
48                            newAlg.Goodness, Some op, newPos)
49                    else
50                        SearchAlgData(acc.State, acc.Depth,
51                            acc.Goodness, acc.Oper, newPos)
52                    ) data
53
54    let rec internal negamaxAlphaBeta (data : AlphaBetaData) =
55        if data.State.EndState || data.Depth = 0 then
56            AlphaBetaData(data.State, data.Depth, data.State.NegamaxGoodness,
57                data.Oper, data.Positions, data.Alpha, data.Beta)
58        else
59            State.Operators
60                |> Seq.filter (fun op -> data.State.PreCondition(op))
61                |> Seq.fold (fun acc op ->
62                    if acc.Alpha >= acc.Beta then
63                        acc
64                    else
65                        let newState = acc.State.Apply(op)
66                        let newAlg =
67                            negamaxAlphaBeta
68                                (AlphaBetaData(newState, acc.Depth - 1, 0,
69                                     None, 1, -acc.Beta, -acc.Alpha))
70                        let newPos = acc.Positions + newAlg.Positions
71                        if -newAlg.Goodness > acc.Alpha then
```

```
72                              AlphaBetaData(acc.State, acc.Depth,
73                                            -newAlg.Goodness, Some op,
74                                            newPos, -newAlg.Goodness, acc.Beta)
75                          else
76                              AlphaBetaData(acc.State, acc.Depth, acc.Alpha,
77                                  acc.Oper, newPos, acc.Alpha, acc.Beta)
78                     ) data
```

SearchAlgData is a class with the same members as SearchAlg in the previous implementation. A big difference is that now all members are immutable. AlphaBetaData is derived from SearchAlgData and extends it with two additional members (Alpha and Beta). They contain both input and output data of search algorithms. They could also be record types, but then it would be cumbersome to refer to the common part in an AlphaBetaData record. On the other hand, it would be more convenient to copy objects of these types using record expressions.

initMinimaxGoodness is just a helper function to make the code more readable in functions minimaxAlg and computeNextMove. It returns the initial goodness value of a state depending on the player in turn needed by the minimax algorithm.

The main differences between the first and second implementations regarding the search algorithms themselves are the following:

- The parameter of the algorithm functions in the second implementation is an object of type SearchAlgData or AlphaBetaData, which contains all the input data required by the algorithm. The parameters of the constructors in the first implementation are only the input data as individual parameters.

- The return value of the algorithm functions in the second implementation is an object of type SearchAlgData or AlphaBetaData, which contains all the output data produced by the algorithm. The constructors in the first implementation return the algorithm object storing the output data in its mutable members.

- The minimum or maximum selection in the second implementation is performed by the Seq.fold higher-order function generating a number of intermediate SearchAlgData objects but without changing state. The same result is achieved by the first implementation using the Seq.iter higher-order function by changing the values of the mutable properties of the same object without generating intermediate objects.

- In alpha-beta pruning, the first implementation uses Seq.takeWhile to stop iteration over operators if necessary. The second implementation does not stop iteration in this case, but the inner lambda function will return the same intermediate object (acc) until the iteration finishes. We cannot use Seq.takeWhile, because the alpha and beta values of the intermediate objects are not accessible outside the inner lambda function of Seq.fold.

- The initial values of goodness, alpha, and beta (PLUS_INFINITY or MINUS_INFINITY) are set at the beginning of the constructors in the first implementation. In the second implementation, these initial values are set by the computeNextMove function when calling the appropriate algorithm function.

Here is the remaining part of the Game module:

```
140  type private AlgorithmType =
141      | Minimax          = 1
142      | Negamax          = 2
143      | MinimaxAlphaBeta = 3
144      | NegamaxAlphaBeta = 4
145
146  [<Flags>]
147  type GameProp =
148      | None              = 0b00000000
149      | AgainstHumanFlag   = 0b00000001
150      | ComputerStartsFlag = 0b00000010
151      | NegamaxFlag        = 0b00000100
152      | AlphaBetaFlag      = 0b00001000
153      | NoHintFlag         = 0b00010000
154
155  let play initState properties computerDepth humanDepth =
156      let DEPTH = 5
157
158      let properties = defaultArg properties GameProp.None
159      let computerDepth = defaultArg computerDepth DEPTH
160      let humanDepth = defaultArg humanDepth DEPTH
161
162      let againstHuman   = properties &&& GameProp.AgainstHumanFlag
163                                <> GameProp.None
164      let computerStarts = properties &&& GameProp.ComputerStartsFlag
165                                <> GameProp.None
166      let negamax        = properties &&& GameProp.NegamaxFlag
167                                <> GameProp.None
168      let alphaBeta      = properties &&& GameProp.AlphaBetaFlag
169                                <> GameProp.None
170      let noHint         = properties &&& GameProp.NoHintFlag
171                                <> GameProp.None
172
173      let printGameInfo () =
174          printfn "Two-player game."
175          if againstHuman then
176              printfn "A human plays against a human."
177          else
178              printfn "A human plays against the computer."
179              if computerStarts then
180                  printfn "The computer starts the game."
181              else
182                  printfn "The human starts the game."
183          if negamax then
184              printfn "Searching using the negamax algorithm."
185          else
186              printfn "Searching using the minimax algorithm."
```

```
187            if alphaBeta then
188                printfn "Using alpha-beta pruning."
189            else
190                printfn "Not using alpha-beta pruning."
191            if noHint then
192                printfn "No hints."
193            else
194                printfn "With hints."
195            printfn ""
196
197        let rec doWork (state : State) =
198            let computeNextMove depth text =
199                let algorithm =
200                    if not negamax && not alphaBeta then
201                        AlgorithmType.Minimax
202                    elif negamax && not alphaBeta then
203                        AlgorithmType.Negamax
204                    elif not negamax && alphaBeta then
205                        AlgorithmType.MinimaxAlphaBeta
206                    else
207                        AlgorithmType.NegamaxAlphaBeta
208                let data =
209                    match algorithm with
210                        | AlgorithmType.Minimax ->
211                            minimaxAlg
212                                (SearchAlgData(state, depth,
213                                     initMinimaxGoodness state, None, 1))
214                        | AlgorithmType.Negamax ->
215                            negamaxAlg (SearchAlgData(state, depth,
216                                                MINUS_INFINITY, None, 1))
217                        | AlgorithmType.MinimaxAlphaBeta ->
218                            minimaxAlphaBeta
219                                (AlphaBetaData(state, depth, 0, None, 1,
220                                     MINUS_INFINITY, PLUS_INFINITY))
221                                     :> SearchAlgData
222                        | AlgorithmType.NegamaxAlphaBeta ->
223                            negamaxAlphaBeta
224                                (AlphaBetaData(state, depth, 0, None, 1,
225                                     MINUS_INFINITY, PLUS_INFINITY))
226                                     :> SearchAlgData
227                        | _ ->
228                            failwith "Invalid game tree search algorithm"
229                printf "Current search algorithm: %O" algorithm
230                printf "[ state=%O, depth=%d, operator=%O, goodness=%d, "
231                    data.State data.Depth
232                    data.Oper.Value data.Goodness
233                if alphaBeta then
234                    let alphaBetaData = data :?> AlphaBetaData
235                    printf "alpha=%d, beta=%d, "
```

```
236                         alphaBetaData.Alpha alphaBetaData.Beta
237                 printfn "number of evaluated positions=%d ]\n"
238                         data.Positions
239                 printfn "%s: %O\n" text data.Oper.Value
240                 data.Oper.Value
241
242             if state.EndState then
243                 printf "The game is over. "
244                 if state.AWon || state.BWon then
245                     printfn "The game is won by player '%c'."
246                             (if state.AWon then 'A' else 'B')
247                 else
248                     printfn "The result is a tie."
249             else
250                 printfn "Current state: %O" state
251                 if not againstHuman &&
252                     (state.Player = 'A' && computerStarts ||
253                      state.Player = 'B' && not computerStarts) then
254                     let op = computeNextMove computerDepth "Computer's move"
255                     doWork (state.Apply(op))
256                 else
257                     if not noHint then
258                         computeNextMove humanDepth "Recommended move"
259                             |> ignore
260                     else
261                         printfn ""
262                     printfn "The move of player '%c':\n" state.Player
263                     doWork (state.Apply(state.ReadMove()))
264
265         printGameInfo ()
266         doWork initState
```

`AlgorithmType` is not a discriminated union but an enumeration for two reasons:

- there are no special properties of the individual search algorithms,
- we use the identifiers defined in the enumeration as the names of the algorithms when printing the search information in the `computeNextMove` function, and for that, it is easier to use the `ToString` method of the `Enum` class than writing our own `ToString` implementation.

The `play` function begins with initializing some local values that were private fields of the `GameControl` class in the first implementation. Then two helper functions follow: `printGameInfo` plays the role of the `ToString` method and the `PropertiesText` property of the `GameControl` class, while `doWork` is a recursive function actually responsible for the game control (just like in the first implementation). The body of the `play` function consists only of calls to these helper functions.

The `doWork` function works exactly the same way in both implementations. The differences lie between the two local `computeNextMove` functions:

- The second implementation first sets the `algorithm` value to the appropriate identifier of the `AlgorithmType` enumeration. This value is then used in a `match` expression to determine which search algorithm is to be used, and one more time later, when printing the results of the search to the output: the identifier will serve as the name of the used algorithm.

- As I previously mentioned, the algorithm functions must be given an argument of type `SearchAlgData` or `AlphaBetaData` with all the initial values the algorithm requires. In the first implementation, only `state` and `depth` were given as arguments to the appropriate constructor; all other values were initialized by the constructor.

- After the search has finished, all relevant search information is printed to the output. In the second implementation, this is done inline, whereas the first implementation used the `ToString` method of the algorithm object to achieve the same result.

The Main Program

The main program now consists only of a call to the `play` function with four arguments, as F# functions may not have optional arguments:

```
1   open Game
2
3   let main () =
4       play (Nim.NimState())
5            (Some (GameProp.NegamaxFlag ||| GameProp.AlphaBetaFlag))
6            None None
```

4.2.4 Comparing the Three Implementations

The implementations presented in this chapter resemble one another to a greater extent than those in the previous chapter. It is because now even the original C# version used recursion in the implementation of the search algorithms, so the code could be made more functional only by replacing mutable data and assignments with immutable data and higher-order functions working with them. However, these modifications did not make the code shorter or more readable.

The most conspicuous difference between the second F# implementation and the other two is that it contains only one mutable data structure: the `Operators` property of the `State` class. It could be easily replaced with a sequence, but it would cause no increase in the abstraction level; `Operators` is used as a sequence everywhere in the code anyway.

The algorithms are the same, the approach is different, but students too are different, so some of them may better understand the algorithms based on a more functional approach than on a pure object-oriented code. For example, the `Seq.fold` function using a lambda expression with an accumulator parameter (`acc`) may better describe how minimum or maximum selection is performed for students who think recursively (functionally).

The observations made regarding the various implementations of search algorithms for single-agent problems are valid for search algorithms on game trees too. Table 4.1 shows the same code metrics discussed in Section 3.2:

	C#	F# ver. 1	F# ver. 2
Lines of code	359	252	261
Number of classes	8	8	4
Number of functions	14	14	11

Table 4.1. Some code metrics.

It is interesting to see that the number of lines in the second F# implementation is a little greater than in the first one. This too shows that we could not benefit much from converting the algorithm classes and the `GameControl` class into functions, and from replacing mutable data with immutable data. On the other hand, there are fewer classes and fewer functions in the second F# implementation. The reason for fewer classes is the substitution of functions for six classes; only `State` and `Operator` remained as classes. However, two new classes were introduced for storing search information, this is why there are four classes instead of two. The decrease in the number of functions comes mainly from eliminating the `ToString` methods of the four algorithm classes.

4.3 Implementing Specific Problems

Just like single-agent problems, two-player games can also be challenging when it comes to creating or implementing their state-space representation. However, the most challenging job in this case is typically not the representation of the precondition or application of operators but finding a usable evaluation function for computing the goodness value of states. If it is simple, it may not give a correct estimation of the goodness; if it is too complicated, it may take a lot of time to execute. We have to find a compromise between the two, which may be easier thinking functionally.

In this section, I present the state-space representation and two implementations of a simple and well-known two-player game: Nim. It is usually used as an example in the seminars of *Introduction to Artificial Intelligence* as well at our university because of its simplicity.

The Problem

Nim is a sort of "take-away game," in which players alternately remove objects from distinct heaps, and the player who takes the last object wins (or loses). There are a number of variants of this game, depending on how many heaps of objects exist initially, how many objects a player may take away in one move, and whether the player who makes the last move wins or loses. We will now consider the following game:

- There are initially 3 heaps of objects, containing 2, 4, and 3 objects, respectively.

- Players move alternately, and they can remove any positive number of objects from exactly one heap.
- The player who cannot move, loses. This happens when there are no more objects left in any of the heaps. In other words, the player who makes the last move wins.

State-Space Representation

Let's denote the heaps with the numbers 1, 2, and 3. A relevant property of the game is the number of objects in each heap. We can limit the maximum number of objects in each heap to $MAX = 10$. As each heap may contain any number of objects between 0 and MAX, we can assign the same base set to each heap:

$$H_1 = H_2 = H_3 = \{0, 1, \dots, MAX\}$$

Another relevant property of the game is the player in turn. Let's denote the players with the letters A and B. We can then define the following base set:

$$P = \{A, B\}$$

The states of the game will be elements of the Cartesian product of these base sets:

$$\begin{aligned} \mathcal{S} \subseteq H_1 \times H_2 \times H_3 \times P = \{&(0,0,0,A),\ (0,0,0,B),\\ &(0,0,1,A),\ (0,0,1,B),\\ &\dots,\\ &(MAX, MAX, MAX, A),\ (MAX, MAX, MAX, B)\} \end{aligned}$$

As all the elements of the $H_1 \times H_2 \times H_3 \times P$ set are valid states of our problem, there is no need for any constraints to narrow this set, i.e., the *state space* of the problem will be exactly this set:

$$\mathcal{S} = H_1 \times H_2 \times H_3 \times P$$

At the *initial state*, we have 2, 4, and 3 objects in the respective heaps, and player A is in turn:

$$start = (2, 4, 3, A) \in \mathcal{S}$$

The *set of end states* consists of two elements, in which each heap is empty:

$$\mathcal{E} = \{(0, 0, 0, p)\} \subset \mathcal{S}$$

If $p = A$, the winner is player B, otherwise the winner is player A. There is no tie in this game. The *set of operators* contains $3 \cdot MAX$ elements:

$$\mathcal{O} = \{\text{Move}(heap,\ number)\}$$

where

$$\begin{aligned} heap &\in \{\,1, 2, 3\,\}\\ number &\in \{\,1, 2, \dots, MAX\,\} \end{aligned}$$

The Move(*heap*, *number*) operator is applicable to state $h = (h_1, h_2, h_3, p) \in \mathcal{S}$ if the following precondition is met:

- there are at least *number* objects in heap *heap*:

$$h_{heap} \geq number$$

The application of the Move(*heap*, *number*) operator to state $h = (h_1, h_2, h_3, p) \in \mathcal{S}$ results in a new state $h' = (h'_1, h'_2, h'_3, p') \in \mathcal{S}$ where

$$h'_i = \begin{cases} h_i - number & \text{if } i = heap, \\ h_i & \text{otherwise,} \end{cases}$$

and

$$p' = \begin{cases} A & \text{if } p = B, \\ B & \text{otherwise.} \end{cases}$$

By defining the $\mathcal{S}$ state space, the *start* initial state, the $\mathcal{E}$ set of end states, and the $\mathcal{O}$ set of operators, we have given a possible $g = \langle \mathcal{S}, start, \mathcal{E}, \mathcal{O} \rangle$ state-space representation of our game.

The C# Implementation

Figure 4.2 shows the two classes representing the states and operators of this particular problem.

`NimState` has two constant fields: one for the number of heaps and another for the maximum number of objects in each heap. `heaps` is an array of integers that stores the current number of objects in each heap. Note that the player in turn is declared in the `State` class. The `NimMove` class has two properties (`Heap` and `Number`), which correspond to the parameters of the Move operator.

A possible C# implementation of these two classes is the following:

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using StateSpace;
5   using Game;
6
7   namespace Nim
8   {
9     class NimMove : Operator
10    {
11      internal int Heap { get; private set; }
12      internal int Number { get; private set; }
13
14      public NimMove(int heap, int number)
15      {
16        Heap = heap;
17        Number = number;
18      }
19
```

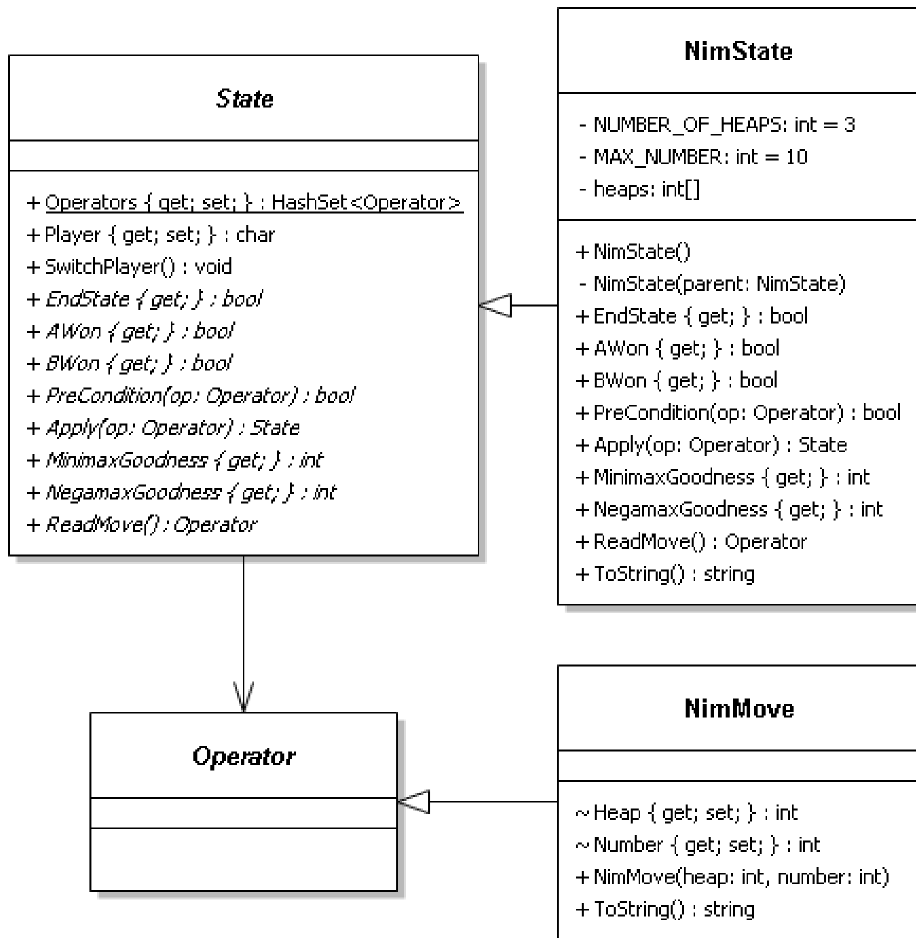


Figure 4.2. Classes representing a specific game.

```
20        public override string ToString()
21        {
22          return "NimMove[ heap=" + Heap + ", number=" + Number + " ]";
23        }
24      }
25
26      class NimState : State
27      {
28        const int NUMBER_OF_HEAPS = 3;
29        const int MAX_NUMBER = 10;
```

```
30
31        static NimState()
32        {
33          Operators = new HashSet<Operator>();
34          for (int h = 1; h <= NUMBER_OF_HEAPS; ++h)
35            for (int num = 1; num <= MAX_NUMBER; ++num)
36              Operators.Add(new NimMove(h, num));
37        }
38
39        int[] heaps;
40
41        public NimState()
42        {
43          Player = 'A';
44          heaps = new int[] {2, 4, 3};
45        }
46
47        NimState(NimState parent)
48        {
49          Player = parent.Player;
50          heaps = new int[NUMBER_OF_HEAPS];
51          parent.heaps.CopyTo(heaps, 0);
52        }
53
54        public override bool EndState
55        {
56          get
57          {
58            foreach (int num in heaps)
59              if (num != 0)
60                return false;
61            return true;
62          }
63        }
64
65        public override bool AWon
66        {
67          get
68          {
69            return EndState && Player == 'B';
70          }
71        }
72
73        public override bool BWon
74        {
75          get
76          {
77            return EndState && Player == 'A';
78          }
```

```
 79      }
 80
 81      public override int MinimaxGoodness
 82      {
 83        get
 84        {
 85          int nega = NegamaxGoodness;
 86          return Player == 'A' ? nega : -nega;
 87        }
 88      }
 89
 90      public override int NegamaxGoodness
 91      {
 92        get
 93        {
 94          int sum = 0;
 95          foreach (int num in heaps)
 96            sum ^= num;
 97          return sum != 0 ? SearchAlg.MAX_GOODNESS
 98                          : SearchAlg.MIN_GOODNESS;
 99        }
100      }
101
102      public override bool PreCondition(Operator op)
103      {
104        if (op is NimMove)
105        {
106          NimMove move = (NimMove)op;
107          return heaps[move.Heap - 1] >= move.Number;
108        }
109        else
110          throw new InvalidOperatorException();
111      }
112
113      public override State Apply(Operator op)
114      {
115        if (op is NimMove)
116        {
117          NimState newState = new NimState(this);
118          NimMove move = (NimMove)op;
119          newState.heaps[move.Heap - 1] -= move.Number;
120          newState.SwitchPlayer();
121          return newState;
122        }
123        else
124          throw new InvalidOperatorException();
125      }
126
127      public override Operator ReadMove()
```

```
128        {
129          Operator op;
130          while (true)
131          {
132            Console.WriteLine("Enter your move:");
133            Console.Write("Heap: ");
134            int heap = int.Parse(Console.ReadLine());
135            Console.Write("Number: ");
136            int number = int.Parse(Console.ReadLine());
137            if (heap < 1 || heap > NUMBER_OF_HEAPS ||
138                number < 1 || number > MAX_NUMBER)
139            {
140              Console.WriteLine("Invalid operator!");
141              continue;
142            }
143            op = new NimMove(heap, number);
144            if (PreCondition(op))
145              break;
146            else
147              Console.WriteLine("This operator cannot be applied!");
148          }
149          Console.WriteLine();
150          return op;
151        }
152
153        public override string ToString()
154        {
155          StringBuilder sb = new StringBuilder("NimState[ heaps=(");
156          for (int h = 0; h < heaps.Length; ++h)
157          {
158            if (h > 0)
159              sb.Append(',');
160            sb.Append(heaps[h]);
161          }
162          return sb.Append("), player='" + Player + "' ]").ToString();
163        }
164      }
165    }
```

The `NimMove` class has no interesting elements.

`NimState` starts with the definition of the two named constants, followed by the static constructor, which generates all the possible operator objects (30 in this case) and adds them to the `Operators` set. Then comes the definition of `heaps` and two constructors: the public default constructor constructs the initial state, while the private copy constructor constructs a clone of its parameter. `EndState`, `AWon`, and `BWon` are very simple properties in this game. The current state is an end state if all the heaps are empty, and a player has won the game if the current state is an end state and the other player is in turn.

In the case of this game, it is easier to compute the goodness value of a state

based only on the position, i.e., independently on the player in turn. This is why `MinimaxGoodness` is defined in terms of `NegamaxGoodness`. If the player in turn is player A, they are equal, otherwise, they are the negative of each other. For the player in turn, we can give a perfect heuristic. This means that we can tell about each possible position whether it is a winner or a loser position. It is a well-known fact that if the "nim-sum" (exclusive or) of the numbers of objects in each heap is zero, then it is a loser position for the player in turn, otherwise, it is a winner position. As a consequence, the search algorithms will find the winner move even if the depth of the search is set to 1.

The `PreCondition` and `Apply` methods are fairly simple in this game, and they have the same skeleton as the same methods in the previous chapter. Note that the `Apply` method uses the inherited `SwitchPlayer` method to change the player in turn. We can do this as a part of all operator applications because in Nim, players always move alternately, i.e., no two consecutive moves are made by the same player.

The `ReadMove` method reads from the input the heap number and the number of objects to remove in an infinite loop until the user enters a valid move. Then, the corresponding operator object is created, and its precondition is checked. If the operator is applicable, the method breaks the infinite loop and returns the operator. Finally, the `ToString` method is very simple too: the number of objects in each heap is printed one after the other, followed by the player in turn.

The F# Implementation

Let's see now a multiparadigm implementation of the `NimMove` and `NimState` classes:

```
1   module Nim
2
3   open System
4   open System.Text
5   open StateSpace
6   open Game
7
8   type NimMove(heap, number) =
9       inherit Operator()
10
11      member this.Heap = heap
12      member this.Number = number
13
14      override this.ToString() =
15          sprintf "NimMove[ heap=%d, number=%d ]" heap number
16
17  type NimState private (player) =
18      inherit State(player)
19
20      static let NUMBER_OF_HEAPS = 3
21      static let MAX_NUMBER = 10
22
23      let heaps = [|2; 4; 3|]
```

```
24
25      static do
26          for h in 1 .. NUMBER_OF_HEAPS do
27              for num in 1 .. MAX_NUMBER do
28                  State.Operators.Add(NimMove(h, num)) |> ignore
29
30      member private this.Heaps = heaps
31
32      new() =
33          NimState('A')
34
35      private new(parent : NimState, move : NimMove) as this =
36          NimState(parent.OtherPlayer) then
37              parent.Heaps.CopyTo(this.Heaps, 0)
38              this.Heaps.[move.Heap - 1] <-
39                  this.Heaps.[move.Heap - 1] - move.Number
40
41      override this.EndState =
42          Array.fold (fun acc num -> acc && num = 0) true heaps
43
44      override this.AWon =
45          this.EndState && this.Player = 'B'
46
47      override this.BWon =
48          this.EndState && this.Player = 'A'
49
50      override this.MinimaxGoodness =
51          let nega = this.NegamaxGoodness
52          if this.Player = 'A' then nega else -nega
53
54      override this.NegamaxGoodness =
55          let sum = Array.fold (fun acc num -> acc ^^^ num) 0 heaps
56          if sum <> 0 then MAX_GOODNESS else MIN_GOODNESS
57
58      override this.PreCondition(op) =
59          match op with
60              | :? NimMove as move ->
61                  heaps.[move.Heap - 1] >= move.Number
62              | _ ->
63                  raise InvalidOperator
64
65      override this.Apply(op) =
66          match op with
67              | :? NimMove as move ->
68                  upcast NimState(this, move)
69              | _ ->
70                  raise InvalidOperator
71
72      override this.ReadMove() =
```

```
73            printfn "Enter your move:"
74            printf "Heap: "
75            let heap = Int32.Parse(Console.ReadLine())
76            printf "Number: "
77            let number = Int32.Parse(Console.ReadLine())
78            if heap < 1 || heap > NUMBER_OF_HEAPS ||
79                number < 1 || number > MAX_NUMBER then
80                printfn "Invalid operator!"
81                this.ReadMove ()
82            else
83                let op = NimMove(heap, number)
84                if this.PreCondition(op) then
85                    printfn ""
86                    upcast op
87                else
88                    printfn "This operator cannot be applied!"
89                    this.ReadMove ()
90
91        override this.ToString() =
92            let sb = StringBuilder("NimState[ heaps=(")
93            for h in 0 .. heaps.Length - 1 do
94                if h > 0 then
95                    sb.Append(',') |> ignore
96                sb.Append(heaps.[h]) |> ignore
97            sb.Append("), player='" + this.Player.ToString() +
98                      "' ]").ToString()
```

Although this code works similarly to the C# code (for example, it uses an array for storing the number of objects in the heaps), there are some substantial differences:

- The `NimState` class has now three constructors. The private implicit constructor sets the player in turn to the given argument and initializes the `heaps` array to the initial position. The public default constructor constructs the initial state of the game by calling the implicit constructor with player A as an argument. The third constructor is private too; it is responsible for the actual operator application. It takes two parameters: a parent state and an operator to be applied to the parent state. First, the implicit constructor is called with the `OtherPlayer` property of the parent state as an argument, as we have to switch players as part of the operator application. Then, the `heaps` array is copied from the parent to the current state and modified according to the operator. This way, the only job of the `Apply` method will be to call this constructor and return the newly constructed state, so it does not have to change any state. Because of this, we could easily replace the type of `heaps` with an immutable list and make the entire code purely functional; although it would not result in a shorter or more readable code.

- The `EndState` and `NegamaxGoodness` properties use the `Array.fold` higher-order function instead of `foreach` loops to perform different kinds of summation on the `heaps` array. It results in a more succinct code but also less efficient in the case

of `EndState` because `Array.fold` traverses the whole array, whereas the `foreach` loop stops if the current element is not zero.

- The `Apply` method is much simpler than in the C# code because after checking whether the argument operator is of the correct type, it just calls the third constructor, which actually performs the operator application.

- Instead of an infinite loop, the `ReadMove` method now recursively calls itself if the user enters an invalid or inapplicable move.

CHAPTER 5

A New Methodology for Teaching Computer Science

Based on my ten years of teaching experience at the University of Debrecen, I can say that students majoring Software Information Technology BSc have to face a number of difficulties during their studies. I think these difficulties root from two main problems: students are unmotivated and cannot sense the coherence between the knowledge acquired in the various courses. This chapter tries to give some remedy to both of these problems by the idea of introducing some long-term projects to students, which they can work on throughout their studies, dealing with a particular aspect of the projects in each course.

5.1 Overview

For the last few years, most of the students majoring in some area of computer science at our university have been having a hard time fulfilling the requirements of most nonbasic courses. This is partly because of the big number of students. First, we have to launch many practical courses for the same subject with many students in each of them. Due to this, we need many instructors (including student instructors), who have to deal with a lot of students and have much less time to deal with each of them individually. Second, a lot of the students come to our university not because of their interest in computer science but for other reasons (like parental pressure, the popularity of information technology, good job prospects, or simply because they misunderstand the program objectives), and therefore they are often undermotivated.

However, mass education is not the only reason for "mass failure" and poor performance. I believe that we, the instructors, do have some influence on the efficiency of the education. The key is to find a way to pique students' interest. We can do this by assigning them tasks in which they are interested. Creating a two-player game with competitive artificial intelligence and a graphical front-end, writing a library information system that keeps track of data about books, patrons, and loans, or creating a web-based network analysis tool which computes different statistical data about network traffic may be such tasks. For example, we can read about the idea of using *Reversi* as a teaching tool in [27], teaching fundamental programming concepts via

two-dimensional game development in [15], or using physical and virtual models of discrete games to help students learn the fundamental concepts and problem solving strategies in computer science in [1]. In particular, two computer games are used to teach the concepts of boolean expressions and recursion in [7]. Even abstract knowledge of mathematical logic can be presented through playful tasks [25]. In fact, nowadays one can hardly find a renowned journal or conference on computer science education without a publication about teaching via real-world projects. As examples, we can mention the journals *Computers & Education* (impact factor: 2.621, publisher: Elsevier) or the *Journal of Computer Assisted Learning* (impact factor: 1.464, publisher: Wiley), and conferences like the *3rd Annual International Conference on Computer Science Education: Innovation and Technology* (held on November 19–20, 2012, in Singapore) or the *Consortium for Computing Sciences in Colleges — Northeastern Region* (held on April 12–13, 2013, in Albany, New York).

Whatever the task is about, the secret is that it should be a large-scale project, which covers more (or even most) of the subjects students encounter during their studies. Throughout the project work, they apply the knowledge discussed in the lectures and practical courses of the related core subjects. They learn the applicability and usability of the topics of each subject as well as the problems emerging during the application of that knowledge. I think that if we can find appropriate assignments, we may achieve better performance not only in solving the assignments but also in the final examinations of the courses.

Defining assignments related to developing real-world applications has the following benefits:

- Compelling examples increase students' motivation.
- Via a complex project, students can practice a number of aspects of computer science.
- Using the same complex project in more courses will help students better understand the relationship between the knowledge behind those courses.
- Projects make computer science education more practice-oriented.
- Projects validate the theoretical knowledge acquired during the lectures and answer the question of how to use that knowledge.

I would like to emphasize that the idea of project-oriented education is already applied in most graduate (master) programs of computer science at most Hungarian universities. According to the current act on higher education, even undergraduate programs must contain some amount of project work. We can see a good example of this at the Eötvös Loránd University, Faculty of Informatics, where students can participate in a cooperative training for 16 credits [3]. The goal of the cooperative training is to provide students with the possibility of getting acquainted with the practical side of computer science under the supervision of experienced professionals at real companies in the software industry. Another example is the subject titled *Project Laboratory* at the Budapest University of Technology and Economics, Faculty of Electrical Engineering

and Informatics [21] or at the University of Debrecen, Faculty of Informatics, where students may deepen their knowledge and get some experience in a specific field of computer science. Project-based education is also applied in foreign institutions; for instance, at the University of Paderborn, teams of three have to develop operating systems during one of the undergraduate courses.

There are, however, some important differences between the proposed approach of "teaching with projects" and the above-mentioned examples:

- Both cooperative training and the *Project Laboratory* courses are independent from the core subjects of an undergraduate program in the sense that they are separate educational units. According to my proposal, the projects would form a part of the course materials of most core subjects, so students can work on projects in the frame of the existing subjects, and no separate subjects or trainings are required.
- While cooperative training and the *Project Laboratory* courses focus on only one or two areas of computer science, the proposed approach covers the topics of most core subjects.
- Cooperative training and the *Project Laboratory* subject both have strict prerequisites, i.e., they are based on knowledge acquired during earlier studies. However, using the proposed approach, students start working on projects as early as the first semester. This also means that instructors have to deal with a lot more students, who have not participated in project works before. On the other hand, instructors may also be inexperienced in project management, and they need to cooperate with one another in order to achieve a better result.
- Although cooperative training is a part of education, the institute forfeits its right to control the flow of the training and the assessments. Another drawback is that it is not so easy to find the necessary number of companies with appropriate projects outside the capital.

5.2 Current Program Requirements

Let's first have a look at the requirements of the Software Information Technology BSc major. The program lasts for 6 semesters, and students have to gather a total of 180 credits according to the following list:

- 120 credits from core subjects
- 29 credits from compulsory elective subjects
- 11 credits from elective nonvocational subjects
- 20 credits from the thesis

Table 5.1 contains the full list of the core subjects with credit numbers, contact hours, prerequisites, and recommended semesters [5].

Subject code	Subject name	Credit	Lectures	Seminars	Labs	Prerequisites	Recommended semester
CS101	Discrete Mathematics 1	5	2	2			1
CS111	Calculus 1	5	2	2			1
CS401	Logic in Computer Science	5	2	2			1
CS201	Introduction to Informatics	5	2		2		1
CS202	HTML, XML	2			2		1
CS711	Computer Architectures	5	2		2		1
CS102	Discrete Mathematics 2	5	2	2		CS101	2
CS112	Calculus 2	5	2	2		CS111	2
CS131	Probability Theory and Statistics	5	2		2	CS101 CS111	2
CS421	Data Structures and Algorithms	5	2	2		CS201	2
CS301	Programming Languages 1	5	2		2	CS201	2
CS211	Operating Systems 1	5	2		2	CS201	2
CS411	Automata and Formal Languages	5	2	2		CS101	3
CS302	Programming Languages 2	5	2		2	CS301	3
CS212	Operating Systems 2	5	2		2	CS211	3
CS501	Database Systems	5	2		2	CS301	3
CS601	Introduction to Computer Graphics	5	2		2	CS101 CS301	3
CS141	Numerical Methods	5	2		2	CS102	3
CS441	Introduction to Artificial Intelligence	5	2	2		CS302 or (CS301 and CS401)	4
CS311	Programming Environments	2			2	CS302	4
CS321	Programming Technologies	5	2		2	CS302	4
CS721	Computer Network Architectures and Protocols	5	2		2	CS711 CS212	4
CS511	Database Administration	3	2			CS501	5
CS521	Technology of System Development	5	2		2	CS321	5
CS001	Thesis 1	10				CS321	5
CS451	Algorithm Design and Analysis	5	2	2		CS401 CS411	6
CS231	Internet Tools and Services	3	2			CS521	6
CS002	Thesis 2	10				CS321	6

Table 5.1. Core subjects.

The compulsory elective subjects are divided into five subject groups (called blocks), each containing four subjects with a total of 16–18 credits:

- Block A: Artificial Intelligence
- Block B: Database Systems
- Block C: Operating Systems and Networks
- Block D: Computer Graphics
- Block S: Information Theory and Applied Mathematics

Students have to complete at least one subject from each of these blocks. The remaining credits needed for the 29 credits can be earned by completing other subjects from the blocks or additional elective vocational subjects launched by the faculty at the beginning of each semester.

As you can see, students have to take and complete at least 37 subjects during their studies, which is a rather big number for only 6 semesters. I think that some of these subjects should be a part of graduate programs, and others (the basic subjects) should get more emphasis with more contact hours.

However, even if we insist on this study plan, we may still find projects that involve a number of the listed areas. Let's now have a look at a couple of examples.

5.3 Some Possible Projects

If we take a closer look at Table 5.1, we can soon realize that even a medium-sized software development project needs some knowledge from at least three core subjects. We can state this based merely on the names of the subjects, without knowing the detailed topics of them and without knowing the goal of the application to be developed. The three most basic subjects, which are involved in every software development project, are *Introduction to Informatics*, *Data Structures and Algorithms*, and *Programming Languages 1*.

In this section, I would like to present two projects of medium difficulty, parts of which may be used as assignments from as early as the first semester.

5.3.1 Project #1: Creating a *Reversi* Application

Of course, *Reversi* may be replaced by any (not too difficult) two-player game here. The main goals of this project are the following:

- To learn some basic programming idioms in at least one programming language.
- To learn how to represent the data structures used during the implementation as well as the algorithms that work with them.
- To learn some basic artificial intelligence methods that are good enough to beat at least a weak human player.

For the assignments to make sense, I briefly introduce the rules of the game [8]. *Reversi* (or *Othello*) is a strategy board game for two players, played on an 8×8 uncheckered board with 64 pieces colored differently on each side, which correspond to the opponents of a game. The color facing up indicates which of the two players controls the square occupied by the piece. The game begins with the central four squares occupied: each player controls one of the diagonals. Players take turns placing pieces on the board with their assigned color facing up until neither can make a move (usually when all 64 squares are occupied). The player who controls the most squares is the winner. A legal move must be to an empty square and must bracket at least one of the opponent's pieces in a straight line between an existing piece of the player in turn and the newly played piece. Upon moving to that square, all of the bracketed opponent's pieces are flipped, in all 8 directions. If a player has no legal move, they must pass, and their opponent will move. If a player has a legal move, they must make it even if it hurts their game.

Here is the list of subjects that may be affected by this project, along with topics of interest and example assignments regarding each subject:

- *Discrete Mathematics 1*: This is a subject with topics from set algebra, linear algebra, number theory, and combinatorics. We can say that almost all projects require some mathematical knowledge, though not necessarily in-depth knowledge. In the seminars, students can be asked combinatorial questions regarding the *Reversi* game. Example assignments for this subject include the following:
  - *How many arrangements of an* 8×8 *board are possible?*
  - *How many games in a* 6×6 *board are possible?*
- *Introduction to Informatics*: During the lectures, students learn the basic concepts that are essential for everyone with a degree in computer science. They learn, among others, about hardware and software, data representation, and basic searching and sorting algorithms. In the laboratories, they practice data representation and start writing simple programs in C. As for the project, the instructors may show how integers, real numbers, characters, strings, or other basic data may be represented in the memory. Although this knowledge is not essential for creating our application, I agree with those who say that data representation is a basic building stone of informatics without which the operation of a computer cannot be understood. Another significant result of this subject is that students write their first C programs so they can begin experimenting with the language. I think it is very important to start writing programs as soon as possible because it takes some time to get accustomed to using the language features for someone who has never seen a high-level program code before. Example assignments for this subject include the following:
  - *Suppose we later want to write a function* $H(b,p) = P(b,p)+8E(b,p)+64C(b,p)$ *where* b *is the board,* p *is one of the players,* $P(b,p)$ *computes the number of pieces* p *has on board* b*,* $E(b,p)$ *computes the number of edge pieces* p *has on board* b*, and* $C(b,p)$ *computes the number of corner pieces* p *has on board* b*. Suggest a representation of the value of* $H(b,p)$ *considering its minimum and maximum possible value.*

  - *Create a well-formed text document containing the rules of* Reversi *as well as the schedule of a* Reversi *tournament.*

- *HTML, XML*: As the name suggests, this subject deals with the syntax and use of these two important markup languages. Students learn how HTML can be used to create simple, static web pages, and how XML can be used to store almost any kind of hierarchically organized data. The knowledge provided by this subject can be useful for any project because all projects may make use of a simple web page or an XML database. In our case, we can store, for example, a game flow in an XML file. Example assignments for this subject include the following:

  - *Create a simple static HTML website containing information about our future* Reversi *game (e.g., the game rules).*
  - *Design an XML data structure (DTD) for storing the game flow.*

- *Logic in Computer Science*: Mathematical logic is used in a number of areas in computer science. In this introductory course, students learn about first-order predicate calculus. It is a big problem that a lot of students do not see at this point why this subject is important, and where they can use the acquired knowledge in the future. The instructors should explain them that all programming languages use conditions, and that conditions are actually logical formulae with all of their properties. Students should know how logical operators (such as implication or the universal quantifier) can be implemented in a programming language that does not implicitly contain those operators. As no laboratory belongs to this subject, these tasks are usually completed only during *Introduction to Artificial Intelligence* lessons. Other concepts that also occur during programming are those of free and bound variables, which may be implemented using parameters and local variables, respectively. Mathematical logic also plays a role in other subjects like *Database Systems* or *Introduction to Artificial Intelligence*. Example assignments for this subject include the following:

  - *Create a new first-order language with syntax and semantics which can be used to express different elements of the* Reversi *game. The language may include functions like the number of each player's pieces or the number of empty squares, and atomic formulae, e.g., for deciding whether a particular square of the board contains a particular player's piece, the game is over, the player in turn has won, the player in turn can win in the next move, or one of the players has more pieces than the other.*
  - *Formalize some interesting assertions about the game as compound formulae such as "if there are no empty squares left, the game is over" or "all nonempty squares contain a piece of either Player 1 or Player 2."*

- *Data Structures and Algorithms*: The goal of this subject is to present the most popular abstract data structures (including files) and their different implementations to the students. In the seminars, students first learn about three searching and at least five sorting algorithms in detail with C implementations, and then practice

the use of the most important data structures that have been talked about during the lectures. This is the first subject in time that has a greater direct influence on our project. From the application's point of view, one of the most important data structures is the array or its two-dimensional version, the matrix. It is because the board of the *Reversi* game and the pieces of the players can most conveniently be represented using an 8×8 matrix. It is interesting to mention how to implement the matrix in a language (like C) that does not support multidimensional arrays. Of course, matrix is not the only data structure used in this project. We will later need, among others, a record to store the states of the game, which can be implemented as a class in an object-oriented language, or a nonbinary tree to represent a part of the game tree, which again can be implemented as a class. Example assignments for this subject include the following:

  - *Implement an 8×8 matrix representing the board in C language. Write functions that execute simple operations on the board like setting all squares in a given row between two given columns to a given piece.*
  - *Implement a stack for storing the board states after successive moves for undoing/redoing the moves.*

- *Programming Languages 1*: From a programming aspect, this is the most important subject, which has the most influence on any software development project. The lectures teach students all the concepts related to high-level programming languages, while in the laboratories, students should acquire the use of a specific procedural language. At our university, this language has been C for ages now because of its significance and because it serves as a base for other, object-oriented languages like Java. Learning a programming language via small sample programs is a good method for the beginning, but they are not enough for learning how to *use* that language. This is where a larger-scale application comes in handy. It not only inspires students to spend some time with programming but also makes them meet situations that otherwise would not come to the front. So in the laboratories, after learning the language itself (which should not take more than 4 weeks), students can create the first version of the *Reversi* application with the help of the instructor. Of course, students have to use the data structures learned in the parallel course *Data Structures and Algorithms*. After finishing the second semester, they may be entitled to say that they are able to create simple (but usable) applications in C. Example assignments for this subject include the following:

  - *Write a function in C that takes a board and a player as parameters and returns the number of pieces the given player has on the given board.*
  - *Write a procedure that takes a board and a player as parameters, reads the given player's next move from the keyboard in a loop until the user enters a legal move, and updates the board according to the move. The code that checks the legality of a move should be placed in a distinct function.*

After successfully completing a number of such assignments, students will have their first working version of the *Reversi* game, which is able to store the state of the

game, draw the board to a character-based console, read the players' input from the keyboard, check the legality of the moves, check if the game is over, and print the result. As an optional assignment, they may improve the first version of the game with the ability to play against the computer. The computer may choose its move randomly in this version. The user should have the possibility to decide whether they want to play with the computer and if so, select the starting player.

- *Programming Languages 2*: The aim of this subject is to get students know the ins and outs of the object-oriented (OO) paradigm. The lectures also touch functional programming, but in the laboratories, they only have to learn one or two object-oriented languages (currently Java and C#). As newer and newer concepts are introduced in the lectures (classes, inheritance, polymorphism, interfaces, etc.), students can gradually rewrite the code of our game application in Java or C#. This way, they can compare the procedural and OO version of the same program and much better see the benefits of the OO paradigm. Example assignments for this subject include the following:
  - *Recode the first version of the game in Java and/or C#.*
  - *Try to rework the result so that it uses more and more OO programming idioms, classes, inheritance, interfaces, and OO data types (especially for collections).*
- *Introduction to Artificial Intelligence*: The lectures of this subject are about state-space representation, various search algorithms, problem reduction representation, and look-ahead algorithms for finding the best move in a two-player game. In the seminars, students first create state-space representations for various problems. After this, they learn how to implement logical formulae in Java or C#, then create a class hierarchy for the most popular search algorithms, and finally implement the minimax and negamax algorithms for two-player games. Although theoretically the subject has no laboratory courses, students still use computers and write programs during the seminars in the second part of the semester. Needless to say, this subject is of great importance concerning our project. By the end of the semester, students are able to build the minimax algorithm into the application so that human players can play against the computer. The instructors may even organize a contest among the students' programs to further motivate them to write the best possible heuristic function. Example assignments for this subject include the following:
  - *Create the Java or C# code implementing all the logical operators of first-order logic.*
  - *Augment the latest version of your program by integrating a minimax (or negamax) algorithm and eventually, alpha-beta pruning.*
- *Programming Environments*: This is a laboratory-only subject, which focuses on the usage of integrated development environments, debugging, CASE tools, the control of compilation, and using libraries. The instructors can show students how to detect different semantic errors in the *Reversi* application with the help of the debugger of Netbeans or Visual Studio. Students can also learn how to use a CASE

tool, for example, to create the Java code from a UML class diagram and thus shorten the coding time. As you can see, there are quite a few possibilities to experiment with our project throughout this course. Example assignments for this subject include the following:

- *Create a statically/dynamically linked library from the AI part of the* Reversi *application so that it can be used later with other applications as well.*

- *Programming Technologies*: The lectures of this subject deal with different programming methodologies, reuse-oriented development, the role of abstraction, programming idioms, design patterns, good programming styles, refactoring, testing, validation and verification, software metrics, and software quality assurance. In the laboratories, students learn about UML, advanced exception handling, using C# delegates, multithreading, reflection, working with metadata (annotations in Java or attributes in C#), using the Java API or the .NET framework, creating graphical user interfaces (e.g., using Swing), JavaBeans, database connectivity from Java and C#, network handling, processing XML files, internationalization (i18n), and using regular expressions. It can be seen from this enumeration that this subject covers a very wide area of software development. Because of this, instructors can show students a lot of exciting aspects of programming. Example assignments for this subject include the following:
  - *Create a GUI for your application using some visual form designer.*
  - *Make the program multilingual using i18n.*
  - *Extend the application with the capability of loading games from and saving games to XML files or a database (e.g., with JDBC).*

- *Computer Network Architectures and Protocols*: The lectures of this subject cover the theory of networking based on the ISO OSI model and the most popular protocols of each layer. Laboratories are used, among others, to create and implement new application layer protocols. Example assignments for this subject include the following:
  - *Create a client/server version of your application. This means that the server side runs on some host, and clients connect to it through TCP/IP and a new application layer protocol, which controls the game flow. As a bonus, you may write the server side using multithreading so that each client connection starts a new thread, which is responsible for the communication with that client.*

- *Technology of System Development*: The lectures are about the process of software development, process models, functional and nonfunctional requirements, system models, requirement analysis, design, standards (UML, MDA), service-oriented architecture, and agile software development. In the laboratories, students create different UML diagrams and ISO documents. They also learn how to use a version control system and developing in teamwork. Example assignments for this subject include the following:

  - *Create different kinds of UML diagrams (e.g., a class diagram, use case diagrams, or state diagrams) concerning our project.*

5.3.2 Project #2: Creating a Library Information System

This project differs from the *Reversi* project in that it does not require artificial intelligence but requires much deeper database knowledge. Of course, we can again replace the word "library" by any other institution (e.g., a hospital, a shop, a school, etc.); the point is that we need to keep track of a big amount of data and be able to execute (possibly complicated) queries via a user-friendly web-based interface or a thick client program.

Most of the depicted connections between each subject and the *Reversi* project apply to this project too. The differences are the following:

- *Data Structures and Algorithms*: For this project, instructors may show students how to create abstract record data structures for storing the books' data, the patrons' data, etc., and how to implement them using the *struct* type in C. The other thing students may learn is the different abstract file formats (serial, sequential, direct, indexed, etc.) with which these records can be stored.

- *Programming Languages 1*: The first version of the application can work with files instead of databases so we can end up with a C program that can read and write data from and to text files or binary files. This way, students will see the big difference between file management and database management during the *Database Systems* course.

- *Programming Languages 2*: This project probably needs a little more complicated class hierarchy than the *Reversi* project so students can better practice inheritance, polymorphism, or interfaces. On the other hand, they can also realize that file management is somewhat more convenient in Java or C# than in C.

- *Database Systems*: The *Reversi* application did not use databases (unless we added the capability of loading and saving games). However, database management is a crucial building stone in the Library project. In the lectures of this subject, students learn about the basic concepts of database systems as well as the relational, ER, EER, and ODMG data models, with special emphasis on the relational model. In the laboratories, students use Oracle SQL to acquire the usage of SQL DML, DDL, and DCL. This course is very important for our project. Students have to practice complex SELECT statements in our Library database to be able to build arbitrary queries into our application.

- *Database Administration*: This is a lecture-only subject about the role of the database administrator, creating a database environment, handling metadata, storage management, distributed databases, database security, archiving and recovery, preparing for catastrophes, database performance, and change management. Although this subject lacks laboratories, the lecturer can show examples of the above topics concerning our Library database. Examples from a well-known system always

helps better understand the underlying knowledge than examples from different, independent, unknown systems (or even from one single but unknown system).

5.3.3 Further Subjects and Projects

Subjects in Table 5.1 not mentioned so far are less important from a programmer's point of view, or at least the knowledge behind them is less used in real-world applications. There are two more subjects with laboratories during which students write programs. One is *Introduction to Computer Graphics* where they create programs, among others, for drawing simple graphic shapes like lines or circles, or for drawing three-dimensional shapes using parallel or central projection. Students can make use of the knowledge acquired during this subject in projects like creating a computer game with advanced graphics. If a student wants to work in this area, then this subject is an essential base for them, which must be followed by other, advanced subjects dealing with computer graphics like those in the Computer Graphics block.

The other core subject where students have to write programs is *Numerical Methods*. In this subject, they learn about function approximation, numerical differentiation, numerical quadrature, various methods for solving linear and nonlinear equations and equation systems, matrix factorization and inversion, computing determinants, and approximation of the eigenvectors and eigenvalues of matrices. They also learn to use software like MATLAB or the LINDO API. Some of the methods mentioned in the lectures are coded in the laboratories, while others are homework assignments. The knowledge provided by this subject along with other mathematical subjects like *Discrete Mathematics 1/2*, *Calculus 1/2*, or *Probability Theory and Statistics* can be applied in projects with some mathematical background. As an example, an application for various kinds of statistical analyses may be such a project. To further narrow it, someone may want to write a program that provides different statistical data from the electronic administration system used by the institution. This example also has to do with data mining or even data warehouses, which are areas covered in one of our graduate programs.

There is some sort of programming in the laboratories of *Computer Architectures* too. This subject overlaps with *Introduction to Informatics* because both deal with data representation, but *Computer Architectures* is more about the abstract architecture and operation of a computer. To help students better understand how computers work at lower levels, they learn some assembly programming during the laboratories. Students majoring Engineering Information Technology could make more use of this knowledge, although, interestingly enough, they do not have a laboratory course for this subject. Nevertheless, assembly programming comes in handy in projects requiring low-level programming such as writing drivers for different hardware components.

Subjects like *Automata and Formal Languages*, *Algorithm Design and Analysis*, and *Internet Tools and Services* have rather theoretical significance from a programming aspect. For example, if someone wants to write a compiler or just a parser for some language, then they can use the knowledge acquired during the courses of *Automata and Formal Languages*. However, automata can also be used in everyday programming, e.g., when coding an event loop using state machines.

Last but not least, subjects *Operating Systems 1/2* are about the architecture and functions of operating systems. In the laboratories, students learn to use and a little to administer Windows and Linux, today's two most popular operating systems. These are more practical subjects, but they have only little to do with programming. However, students learn during these subjects how to write scripts using batch files in Windows or shell scripts in Linux. They can also benefit from this knowledge when programming in other script languages like JavaScript.

5.4 Conclusion

I believe that learning all the aforementioned knowledge can be much more entertaining for the students by developing one or two larger-scale applications throughout their studies (even if in teamwork) than just writing small sample programs for every different area of software development. With one complex project or with two or three medium-sized applications, we can cover nearly every aspect of the development process and give students a comprehensive example of software engineering. If our faculty introduced this "learning via projects" approach of teaching, students would be a little more motivated and would more likely see the coherence between the topics of the wide range of subjects. On the other hand, this approach requires some extra work on the instructors' part: they need to find appropriate real-world applications that could become the projects, cooperate with one another on distributing the different parts of the projects among the various courses, and a lead instructor should be designated as the person in charge of these tasks, who has an oversight on all subjects in the study plan.

Of course, we cannot expect a radical improvement in students' performance just because of such a minor change in our teaching methodology. Decreasing the number of students or redesigning the program's study plan would have a much bigger effect on it. Although the faculty has little or no influence on the number of enrolled students, we could still initiate the supervision of the program requirements of the Software Information Technology BSc major.

CHAPTER 6

Supporting Programming Contests with the ProgCont Application

The role of educational contests for students is to lead them into a deeper acquaintance with a specific field of their studies. Contests give students personal objectives that stimulate them to work on their own. Apart from being stimulating, contests have a positive effect on students' educational results—participating in a remarkable contest or finishing in a good position a couple of times may contribute to their professional experience. Additionally, educational contests may help nurture professional relationships.

Students of the University of Debrecen have been participating in the Central European regional rounds of ACM International Collegiate Programming Contest since 1995 (although the university did not enter for the contest between 1998 and 2000). As a student in the second year of my studies, I was lucky enough to be a member of the team that advanced from the local round to the regional in 1995. Since 2001, I have been acting as an organizer and a member of the judge of local rounds of ACM ICPC as well as other programming contests.

Every year, two teams of three members represent our university at the ACM Central European Regional Contest. Before the regional, there are two preliminary rounds for selecting the advancing teams: a local and a national round. The local rounds are organized by my colleague Kósa Márk and myself as well as Kádek Tamás, who joined us two years ago. The national round is usually organized by the Budapest University of Technology and Economics and Eötvös Loránd University.

In earlier times, organizing the local university rounds was encumbered by the fact that we had to check the solutions submitted by the contestants "by hand," which, beyond inconvenience, hindered the efficient work of the judge and involved a number of possibilities of making mistakes. To find a solution to these problems, we decided to create an application that can process a large amount of submissions both in real time and off-line. Together with Kósa Márk and an agile student, Gunda Lénárd, we made an e-mail-based console application called *Programming Contest Result Manager* (PCRM) in 2004, which helped us evaluating submissions not only from contestants during a contest but also from students submitting solutions to homework assignments of a particular course, such as *Introduction to Artificial Intelligence*. You can read more about PCRM in [14, 12].

We used PCRM for a couple of years, but later, it failed to comply with our newer and newer expectations. For example, we wanted to have a user-friendly graphical interface for the application, and would have preferred a web interface for both the contestants and the judge. Another drawback was that it used POP3 protocol for retrieving the submitted solutions, which made it difficult to manage and use. We decided not to rework our existing application but to test some third-party programs (by that time, we could find a couple of programs on the Internet that seemed to meet our needs) and at the same time, get some talented students to develop a brand-new web application as a thesis work. As a result, an ASP.NET-based web application was born in 2009, created by a student, which was quite usable but contained some bugs, and after the student left our institution, no one could maintain it anymore. The best third-party program found and tested was PC^2 (Programming Contest Control System) developed by the California State University, but it too lacked some needed features.

When Kádek Tamás joined the Faculty of Informatics as an assistant lecturer, he undertook to implement a web application that suits our needs. With more than 10 years of experience behind our back, we could precisely describe the requirements of the application. This is how *ProgCont* came to life in 2011. It is by far the most usable and most robust utility for supporting programming contests we have ever met. The application is continuously under development—last time we had to extend it to support a new kind of contest, the Regional Team Contest of the Faculty of Informatics, organized for the first time in November 2012 for high schools and colleges of five nearby counties, which used a different evaluation system from ACM-like contests.

6.1 About Programming Contests

Before presenting the ProgCont application, I describe what kinds of programming contest we would like to manage with it, and how these contests are regulated. The primary goal of the application is to manage ACM-like contests, in which contestants (teams or individuals) have to solve problems from a problem set common to all participants in a predefined time interval (typically 5 hours). A problem set usually consists of 8–12 problems (in a 5-hour contest) from the following areas (you can find a lot of examples in our book with Juhász István and Kósa Márk [10]):

- combinatorics
- number theory (e.g., prime numbers)
- arithmetic and algebra (e.g., modular arithmetic, big integers)
- computational geometry
- backtracking
- graph theory (traversal, single-pair/single-source/all-pairs shortest path, minimum spanning tree, articulation point, flood fill, network flow, maximum bipartite matching, etc.)

- sorting
- string processing (e.g., pattern matching)
- greedy algorithms
- dynamic programming
- divide-and-conquer algorithms
- advanced data structures (e.g., Fibonacci heap, dictionary, binary indexed tree)

For solving the problems, contestants (even teams) may use one computer and arbitrary paper-based resource materials. They can compile, test, and submit their solutions on this computer. Solutions are always in the form of source code created in some programming language. Currently, the following languages are supported: ANSI C, C++, C#, Java, and Free Pascal, but this may vary from contest to contest, and even from problem to problem in a contest. The executable program generated from the source code by the respective compiler reads some input and produces some output. Input is usually the standard input, and output is usually the standard output. Occasionally however, input is read from a file, and output is written to another file.

When the judge receives a solution to a problem from a contestant, they compile it with the appropriate compiler, and run the executable for different test cases. Each problem is assigned at least one file containing the test cases. The submitted solutions must process these files and produce the correct output for each of them. (Very rarely but once in a while a problem may have no input data. These situations may be considered as if the program had to process an empty file.) The output is correct either if it is equal to a pregenerated file, or if it is correct according to an external evaluator program. The correctness of a solution may also depend on some predefined limitations regarding certain resources, such as the size of the source code, execution time, the size of memory used, or the use of prohibited library functions. Knowing the correctness of the output, the judge evaluates the submitted source code, typically by giving a score to the solution, and replies with one of the following messages regarding the solution:

- accepted
- compile error
- runtime error
- time limit exceeded
- memory limit exceeded
- wrong answer
- contest rules violation

It is also the judge's responsibility to provide the contestants and observers with the final ranking after the contest has ended.

In ACM contests, a problem is considered to be solved only if the program submitted by the contestant gives a correct result for each of the test cases. However, a contestant may retry to solve the problem any number of times after an unsuccessful attempt. Without this possibility, if there were too many difficult problems and tricky test cases in a contest, then it might end with too few accepted solutions, making it hard to determine the ranklist, not to mention that contestants (and the judge) would have no sense of achievement after the contest. This situation is similar to reality: if a program is unable to solve the problem in question (i.e., it does not meet the client's requirements), there is a possibility to refine the solution.

The order of contestants in the ranklist is determined mainly based on the number of correctly solved problems. If two or more contestants solve the same number of problems, then they are ranked based on their scores. The score is the sum of two components: the time elapsed from the beginning of the contest till the submission of the first accepted solution of each correctly solved problem and the penalty scores given for each incorrect solution of the correctly solved problems submitted before the first accepted solution of those problems. Both components are measured in minutes. Of course, the smaller this score, the better position the contestant will have in the ranklist among contestants with the same number of correctly solved problems.

During the preparation to the first Regional Team Contest of the Faculty of Informatics, we found that it would be better to use a different ranking algorithm and take into account the partially solved problems too. We introduced a new reply message from the judge: *partially solved*. A problem is considered to be partially solved if the submitted solution produces a correct output for at least 60% of the test cases but results in some kind of error in the rest of the cases. ProgCont was designed so that the percentage of test cases for which the program should produce a correct output for the problem to be considered partially solved can be parameterized for each problem in each contest.

The ranking algorithm was modified the following way: If two or more contestants have the same number of accepted problems, then they are ranked based on the number of partially solved problems. If these numbers are the same too, then the ranking is determined by the score of the contestants. The score is computed the same way as in ACM contests. This way, contestants may decide whether they are satisfied with a partially solved problem or continue to work on the same problem until it is accepted.

6.2 The Architecture and Operation of ProgCont

The ProgCont system consists of four key components: problem catalog, contest database, controller web application, and solution evaluator clients (see Figure 6.1).

The *problem catalog* contains all resources related to each problem, such as the problem description (sometimes in multiple languages), the figures in it, further (possibly downloadable) content, and the test cases used to evaluate solutions. Test cases are stored in one or more files. The way of testing the correctness of solutions can be

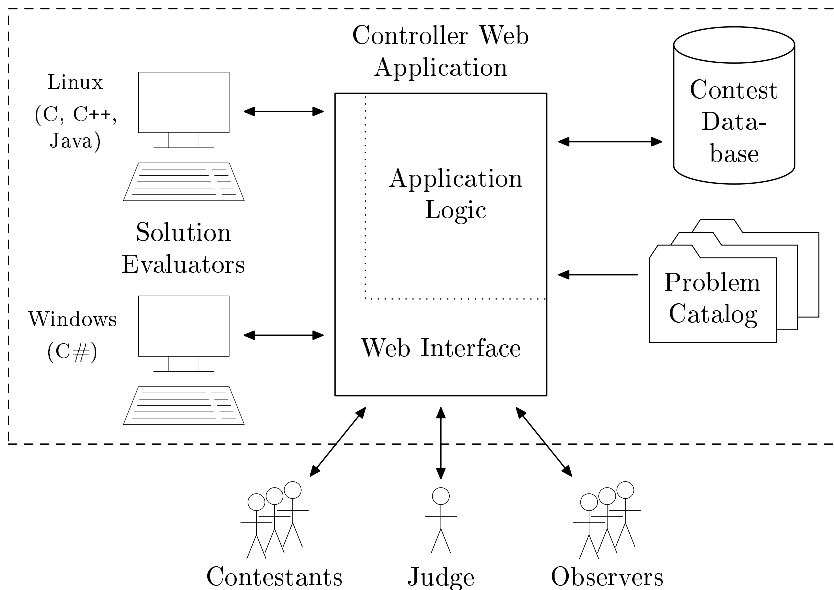


Figure 6.1. The architecture of the application.

configured to one of the following options: the program can either check if the output produced by the submitted solution is completely equal to a pregenerated output file, or call an external tool that analyzes the output and returns whether it can be considered correct. Time limits associated with the execution of the submitted program can be set for each test case separately. The problem catalog can be formed using the directory and file structure of the operating system. Each problem has a distinct folder, in which an XML file contains the problem description and a ZIP file contains the test cases, the pregenerated output files, and the testing parameters. Additionally, all other files referred to in the problem description are in this folder.

The *contest database* describes the relationship between the contest, the problems, the solutions, and the solution evaluations. Each contest comprises some problems selected from the problem catalog, in case of team contests, the members constituting the teams, and technical data regarding the flow of the contest, such as the start time and end time of the contest, or the allowed programming languages for each problem. The database stores the solutions submitted by the contestants and the result of each evaluation, which comes from one of the solution evaluator clients. All of this information is stored in a PostgreSQL database.

The flow of the contests is controlled by the *controller web application*. Contestants can browse the problems, submit the solutions, and learn about the results of evaluations and the current ranking using the web application. Observers (e.g., coaches of the contestants or guests) may also follow the current ranklist. The judge can set certain parameters of the contest via this web interface too. These parameters include the supported programming languages for each problem, the number of points each

problem is worth, or the amount of penalty time used when computing the score of the contestants. In addition, the web application is responsible for scheduling the evaluation of the submitted solutions, i.e., distributing them among the solution evaluator clients currently connected to the system. This distribution is based mainly on the information sent by the clients about the programming languages they support. For running the controller web application, Apache Tomcat web application server is used. Communication between the web application and the users (contestants and the judge) is secured by the SSL (HTTPS) protocol. The web application uses JDBC for accessing the contest database.

Solution evaluator clients are separate applications which periodically (every 5 seconds) check whether there is an available solution waiting for evaluation that they can handle. If so, they first try to compile the program code passed on to them by the controller web application using a preset compiler with preset options. If the compilation succeeds, the program is run for each test case using a preset runtime environment with preset options. The current test cases are downloaded from the problem catalog via the web application whenever the ZIP file containing them changes (and of course, the very first time they are used). During the execution of the program, the solution evaluator client takes into consideration the time limit set for the given test case. If the program stops within the time limit, its output is analyzed depending on the preset method of testing its correctness: it is either compared to the downloaded output file, or passed on to the external evaluator tool. Finally, the cumulative result is sent to the controller web application.

As different programming languages suit different operating systems, the solution evaluator clients were implemented as platform-independent Java applications. This way, they can (and should) be run on distinct (possibly virtual) computer(s) from that of the controller web application and the other evaluator clients, thus not endangering their operation, should a harmful code disrupt the runtime environment of a particular client. For example, a Windows-based client will compile and run C# code, while for C, C++, Java, or Pascal, a Linux-based client may be used. The more solution evaluator clients are used, the more evaluations can be performed at the same time.

During the design of the system's components, it was a primary objective that all communication between the external and internal elements of the system should be conducted over the standard HTTP protocol. The contest database can only be accessed through the controller web application, i.e., indirectly over HTTP protocol too. The evaluator clients use HTTP requests secured by HTTP digest authentication when communicating with the web application. This way, there is no network setup necessary for the users and the solution evaluator clients other than providing them with Internet access. However, in case of some contests, it may be important to restrict contestants' Internet access only to the controller web application.

In the ProgCont system, contests can be parameterized by numerous aspects:

- We can set the languages in which problem descriptions may be browsed. If a contestant selects a language, only the sections marked with the given language and sections not marked with any languages will be displayed from the XML file containing the problem description.

- In case of team contests, contestants may be organized into teams, and any team member may submit solutions to the problems in the name of the team. The results of submission evaluations associated with different members of the same team will be accumulated.

- The duration of the contest can be set arbitrarily; ProgCont is capable of controlling contests lasting from a few hours to weeks or even months.

- A contest can be set to remain visible after the contest session has expired. In this case, contestants may submit solutions even after the end of the contest. These submissions will be evaluated but will not count in the contestant's score or in the result of the contest.

- For each problem, we can set the number of points it is worth in case of an accepted solution, the rate of test cases for the problem to be considered partially solved, and the penalty time in seconds that will be given to the contestants for each incorrect solution after the problem has been correctly solved.

6.3 Our Experience with the Application and Possibilities for Future Development

We organized the first contest controlled by the ProgCont application on October 2, 2011. With respect to our original goal, it was the local round of ACM ICPC of that year. Making use of the ability to parameterize the contests, a short-term individual contest took place on February 6, 2012, among students of the course *Problems in Programming Contests*. In the same semester, we could help students deepen their knowledge in three different courses using contests lasting for more than a month: *Problems in Programming Contests*, *Programming Languages 1*, and *Introduction to Artificial Intelligence*. After that, we organized an ACM-like contest on May 6, 2012 (which was the preliminary round of ECN International Programming Contest in Târgu Mureş, Romania), another local ACM contest on October 7, 2012, and last but not least, the Regional Team Contest of the Faculty of Informatics on November 25, 2012. You can see the list of all the contests managed by ProgCont so far in Figure 6.2.

In the future, we would like to extend the functionality of the ProgCont system with the following features:

- We could introduce new functionality to the system that would assist the work of the organizers and participants both before and after the contest session. The most important activity before a contest is to recruit participants and to inform them about the details of the event. Information regarding the teams and their members are required during the contest too, so it is already a part of the database. (User names and passwords are assigned to team members, but the results of evaluations are assigned to the team.) Collecting and registering this information, i.e., the process of registration, might also be supported by the web application. In other words, an on-line registration interface would allow contestants to register for the

contest and to form a team. Registration and team composition must be approved by the judge, which may be another feature on the administrators' side. Based on the data collected during on-line registration, an interface could provide the organizers with such information as how many computers and how many computer labs will be required for the contest, or how many copies of the problem set will be necessary.

- After the contest has ended, information regarding the results remains in the database. Based on this information, the application might create an on-line "table of honor" containing the contestants (or teams) along with their accumulated results from their previous contests. This cumulative ranklist could motivate contestants for further participation. This table might show, for example, who has been the most successful contestant so far, what is the maximum number of problems that were solved in a contest, or which contestant was the fastest to solve a problem. It might be a source of information for contestants too, showing in which types of problems they have to improve their skills compared to others, and which types of problems they are good at.
- During the contest session, the application might perform further tests regarding the submitted solutions, such as searching for prohibited language elements or library functions in the source code, or enforcing the adherence to the resource limitations (like the amount of memory used or the number of parallel processes run). It would also be an interesting feature to find plagiarism in the submitted source code files.
- PC^2 has a good feature for providing communication between the contestants and the judge. If a contestant has a question about a specific problem or a question in general, they can send a clarification request to the judge, who will then receive a notification about this request, and can answer the question either to the contestant who asked it or to all of the contestants. The judge can also send a clarification without a question. Although ProgCont provides the ability of communication between the contestants and the judge, it lacks the feature of notification and the organization of messages and replies.

And finally, some screenshots of the web interface can be seen in the following figures.

Figure 6.2. Contests managed by ProgCont.

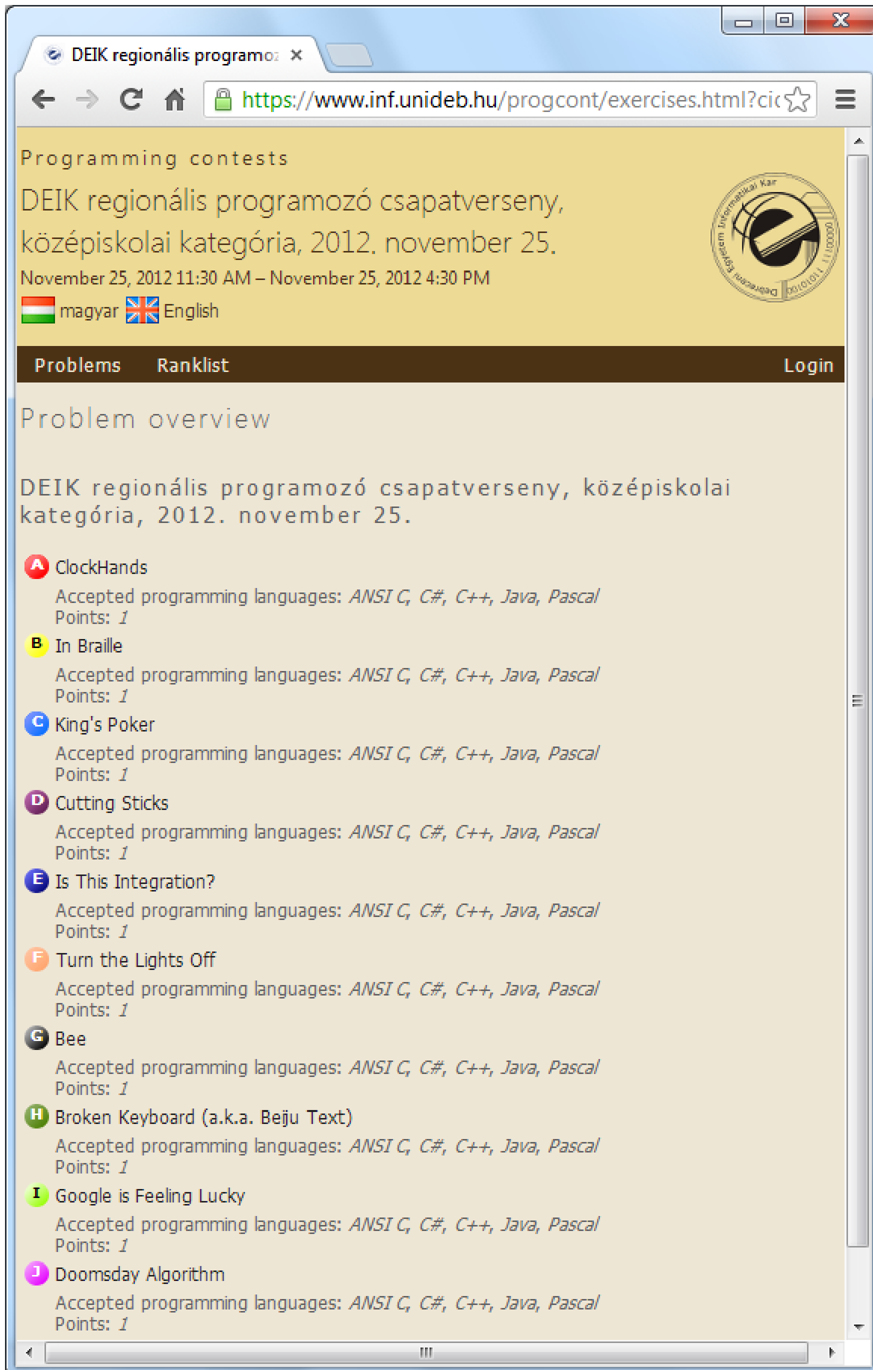


Figure 6.3. Problem set of a contest.

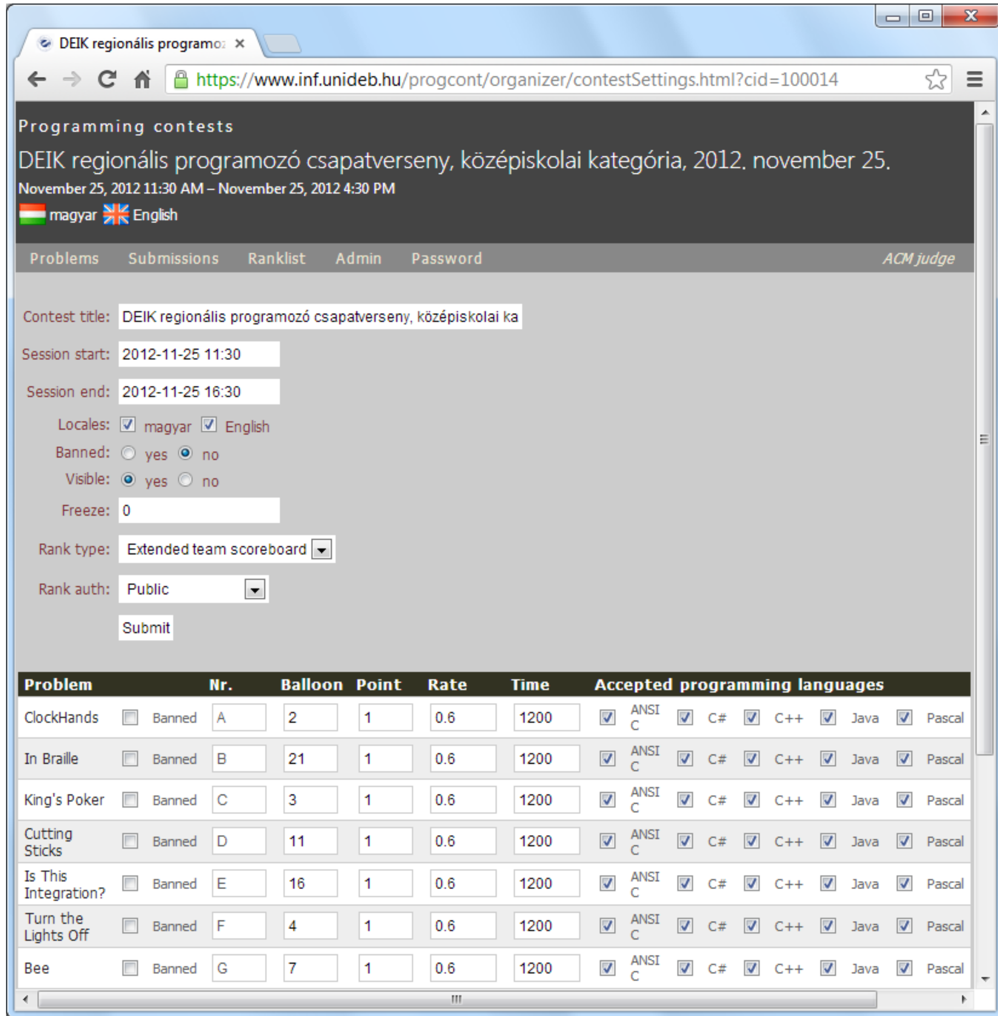


Figure 6.4. Contest settings.

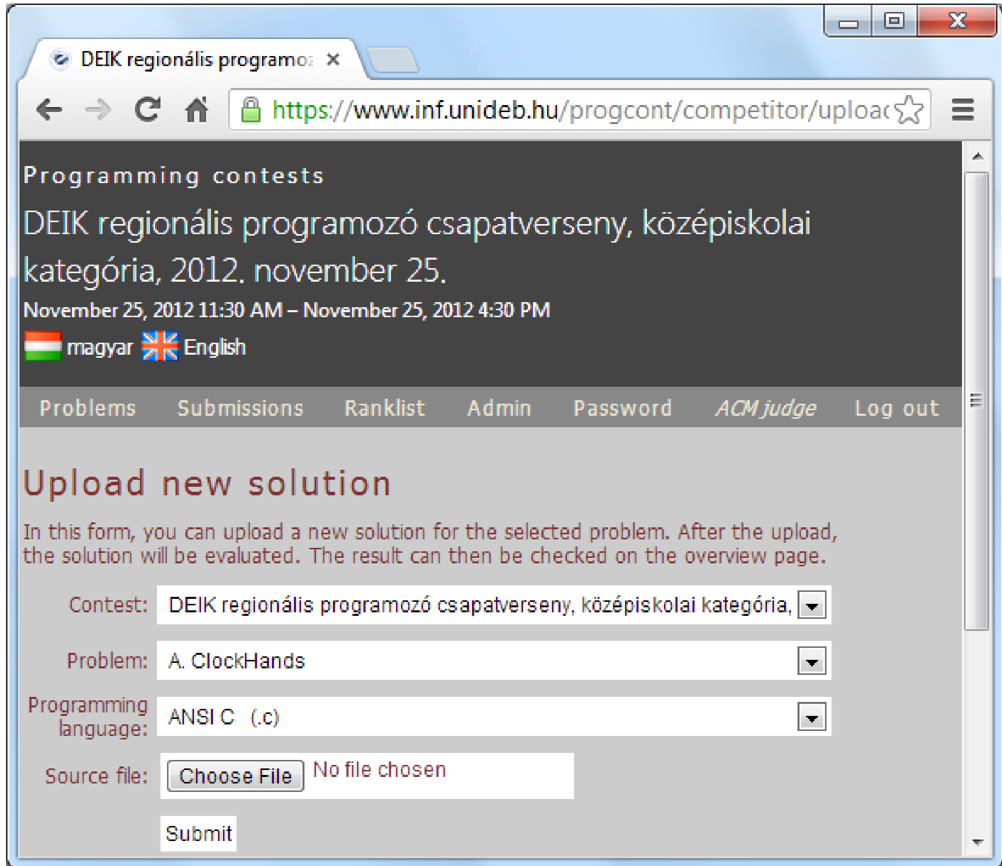


Figure 6.5. Submission of a solution.

DEIK regionális programoz ×

https://www.inf.unideb.hu/progcont/ranklist.html

Programming contests

DEIK regionális programozó csapatverseny, középiskolai kategória, 2012. november 25.

November 25, 2012 11:30 AM – November 25, 2012 4:30 PM

magyar English

Problems Ranklist Login

DEIK regionális programozó csapatverseny, középiskolai kategória, 2012. november 25.

#	Team	A	B	C	D	E	F	G	H	I	J	PT	PR	Time
1.	Aritmetikai Egység	1 1:13	2 4:58	1 2:40	1 2:09	12		2 2:14	8 6:51	1 0:20	1 0:38	8	0	21:03
2.	9^(2^-1) kocka	1 1:50	1 2:43	4 4:33		3 4:41		8	1 1:00	1 0:38	1 0:30	7	1	15:55
3.	GGCG NyiTo	4 1:56	3 2:40	1 3:30	1	1	1	1 1:05	10	1 2:35	1 2:22	6	0	14:08
4.	lila	5 4:06	3 3:32	2 2:59	1 3:48			2	2	1 2:19	1 4:51	6	0	21:35
5.	HTTP 404	2 1:04	4	2 4:29				20	4 3:46	4 4:47	1 3:58	5	1	18:04
6.	TeamTag	1 1:28		1 4:20				2 2:25	3	2 2:42	1 2:33	5	0	13:28
7.	Földes I.	11 7:20		3 4:58				12	4	3 3:25	2 1:38	4	2	17:21
8.	Anonymus	2 1:06	3	5 4:02				6	6	1 4:04	1 1:14	4	0	10:26
9.	maxdb	1 0:33	2	3				4 5:57		2 1:27	1 2:45	4	0	10:42
10.	Zsuzsiék	1 1:01		1 4:52				1	1	1 3:45	1 3:23	4	0	13:01
11.	KékHalál	1 1:08	4 4:00	3				4 5:46			3 3:52	4	0	14:46
12.	Gabika becsajozott	8 7:10		4 5:56				4 4:44	3		1 3:23	4	0	21:13
13.	Fatal Error	1 1:49		1 2:36				5	1	3	1 2:56	3	0	7:21
14.	Kőrösi	4		2				2 1:16	7	1 3:22	1 3:37	3	0	8:15
15.	Befőttek	1						4	2	1 1:53	2 1:55	2	1	3:48

Figure 6.6. Part of the final ranklist of a contest.

CHAPTER 7

Summary

To summarize my results, I hope I managed to create some aids and tools that can make students' life easier during their studies. One such aid can be to assign them long-term projects, which they can work on from as early as the first semester. A real-world application is always more exciting than small program snippets, so a long software development process may improve students' motivation for learning and programming. They can create the relevant parts of the application in each course related to the project under the instructors' supervision. Another beneficial effect of this approach is that students will better see the coherence between the knowledge acquired in the various courses. I presented two such long-term projects with some possible assignments in each related subject.

I created a course guide for *Introduction to Artificial Intelligence* or other AI-related courses. It is made up of various C# and F# implementations of search algorithms for single-agent problems and for two-player games. The difference between these implementations lies in the amount of functional programming constructs used. I also created some C# and F# implementations of some specific problems and games. Students and instructors may select the version best suited for them, depending on their way of thinking and their knowledge of programming. This way, students may better understand the operation of search algorithms presented in the lectures.

In my opinion, it is not worth insisting on one or the other paradigm if we can use more of them within one program. Functional code is sometimes more abstract, more readable, or just shorter than its object-oriented counterpart. On the other hand, OO code is usually more efficient and sometimes more reusable than its functional counterpart. This is why I think multiparadigm languages like F# can be more advantageous mainly in large-scale applications but also in smaller programs. Students are different, so some of them may better understand the algorithms based on a more functional approach than on a pure object-oriented code. For example, the `Seq.fold` function using a lambda expression with an accumulator parameter may better describe how minimum or maximum selection is performed for students who think recursively (functionally).

Programming contests can also be a great motivating factor for students. We have been organizing ACM-like contests for more than ten years now, and I can say there are quite a few students who would never miss an opportunity to try a fall with others

in a competition. Not just for their sake, but we needed an application for managing ACM-like and possibly other kinds of programming contests as well. As we could not find a contest management software satisfying all our needs, two of my colleagues and I have decided to develop a brand-new web application for managing programming contests. *ProgCont* is designed to have a modular architecture and be parameterizable and easily extensible with new supported programming languages.

We organized the first contest controlled by the ProgCont application on October 2, 2011. With respect to our original goal, it was the local round of ACM ICPC of that year. Making use of the ability to parameterize the contests, a short-term individual contest took place on February 6, 2012, among students of the course *Problems in Programming Contests*. In the same semester, we could help students deepen their knowledge in three different courses using contests lasting for more than a month: *Problems in Programming Contests*, *Programming Languages 1*, and *Introduction to Artificial Intelligence*. After that, we organized an ACM-like contest on May 6, 2012 (which was the preliminary round of ECN International Programming Contest in Târgu Mureş, Romania), another local ACM contest on October 7, 2012, and last but not least, the Regional Team Contest of the Faculty of Informatics on November 25, 2012.

8. FEJEZET

Összefoglalás

Összefoglalva az eredményeimet, remélem, sikerült elkészítenem néhány olyan eszközt, amelyek megkönnyíthetik a hallgatók életét a tanulmányaik során. Az egyik ilyen eszköz az lehet, hogy olyan hosszú távú projekteket jelölünk ki számukra, amelyeken már az első félévtől kezdve dolgozhatnak. Egy valós világbeli alkalmazás mindig izgalmasabb, mint a kis programrészletek, egy hosszú szoftverfejlesztési folyamat tehát növelheti a hallgatók tanulási és programozási hajlandóságát. Az alkalmazás megfelelő részeit a projekthez kötödő kurzusok keretein belül, az oktató irányítása mellett készíthetik el. A másik előnyös hatása ennek a megközelítésnek az, hogy a hallgatók jobban átlátják az összefüggéseket a különböző kurzusokon elsajátított ismeretek között. Két ilyen hosszú távú projektet ismertettem, a kapcsolódó tantárgyakhoz kötődő néhány lehetséges feladattal együtt.

Módszertani útmutatást adtam *A mesterséges intelligencia alapjai* című vagy egyéb, a mesterséges intelligenciához kapcsolódó tantárgy oktatásához. Ez egyszereplős problémák és kétszemélyes játékok esetén alkalmazható kereső algoritmusok különböző C# és F# nyelvű implementációiból áll, amelyek között a különbség abban rejlik, hogy mennyi funkcionális programozási elemet tartalmaznak. Elkészítettem néhány konkrét probléma és játék C# és pár F# implementációját is. A hallgatók és az oktatók kiválaszthatják közülük azt a változatot, amelyik a leginkább illik hozzájuk a gondolkodási módjuk és a programozási ismereteik alapján. Ezáltal a hallgatók könnyebben megérthetik az előadásokon ismertetett kereső algoritmusok működését. Az elkészült implementációk összehasonlításából azt a következtetést vontam le, hogy a multiparadigmás programozási nyelvek használata a mesterséges intelligencia oktatásában mind az oktatók, mind a hallgatók számára hasznos lehet.

A programozó versenyek is erős motivációs tényezőt jelenthetnek a hallgatók számára. Már több mint tíz éve szervezünk ACM jellegű versenyeket, és elmondhatom, hogy van jónéhány olyan hallgató, akik egyetlen alkalmat sem szalasztanának el, hogy megmérettessék magukat másokkal egy-egy versenyen. Nemcsak az ő kedvükért, de szükségünk volt egy olyan alkalmazásra, amely ACM jellegű és esetleg más típusú programozó versenyek támogatására is képes. Mivel nem találtunk olyan versenykezelő szoftvert, amely minden igényünket kielégítette volna, két kollégámmal közösen úgy határoztunk, hogy kifejlesztünk egy vadonatúj webalkalmazást a programozó versenyek vezérlésére. A *ProgCont* rendszert úgy terveztük, hogy moduláris felépítésű és paraméterezhető legyen, valamint hogy könnyen bővíthető legyen új támogatott programozási nyelvekkel.

Az alábbiakban röviden ismertetem az elért eredményeket.

8.1. Új módszertan az informatikaoktatásban

Kutatásaim egyik fő területe az volt, hogy megpróbáltam választ találni az alábbi kérdésre: „Miért okoz problémát a legtöbb programtervező informatikus BSc szakos hallgatónknak a legtöbb haladó tantárgy követelményeinek a teljesítése?” A Debreceni Egyetemen szerzett tízéves oktatási tapasztalatom szerint hallgatóinknak számos nehézséggel kell szembenézniük tanulmányaik során. Ez részben a hallgatók nagy száma miatt van. Egyrészt sok kurzust kell indítanunk az egyes tárgyak gyakorlataiból, amelyeken így is túl sok a hallgató. Emiatt sok gyakorlatvezetőre (akár demonstrátorra) van szükség, akiknek sok hallgatóval kell foglalkozniuk, ezért sokkal kevesebb idő jut arra, hogy egyénenként foglalkozzanak velük. Másrészt sok hallgató nem az informatika iránti elkötelezettsége miatt jelentkezik az egyetemünkre, hanem más okokból (például szülői nyomás, az IT szakma népszerűsége, a jó elhelyezkedési lehetőségek vagy egyszerűen a szak céljainak félreértése miatt), és ebből fakadóan gyakran alulmotiváltak.

A tömegképzés azonban nem az egyetlen oka a „tömeges bukásnak” és a rossz teljesítménynek. Úgy vélem, hogy nekünk, az oktatóknak is van némi befolyásunk az oktatás eredményességére. A kulcs az, hogy találnunk kell egy módszert a hallgatók érdeklődésének a felkeltésére. Ennek egyik módja, hogy olyan feladatokat tűzünk ki számukra, amely érdekli őket. Ilyen feladat lehet például egy grafikus felülettel ellátott kétszemélyes játék elkészítése versenyképes mesterséges intelligenciával, egy könyvtári információs rendszer megírása, amely nyilvántartja a könyvek, olvasók és kölcsönzések adatait, vagy egy webalapú hálózatelemző segédprogram készítése, amely különböző statisztikai adatokat számít ki a hálózati forgalomról. Példaként olvashatunk a *Reversi* játék oktatási eszközként való felhasználásáról a [27] cikkben, az alapvető programozási fogalmak kétdimenziós játékok fejlesztésén kesztül történő oktatásáról a [15] tanulmányban, vagy az informatika alapfogalmainak és a problémamegoldó stratégiáknak a diszkrét játékok fizikai és virtuális modelljeinek segítségével történő oktatásáról az [1] cikkben. Konkrétan a logikai kifejezések és a rekurzió fogalmának a tanításához két számítógépes játékot fejlesztett a szerző a [7] dolgozatban. Még a matematikai logika absztrakt tételeit is be lehet mutatni játékos feladatokon keresztül [25]. Manapság szinte lehetetlen olyan neves, informatikaoktatással foglalkozó folyóiratot vagy konferenciát találni, amely ne tartalmazna legalább egy, valós világból vett projekteken keresztül történő oktatásról szóló közleményt. Példaként említhetjük a *Computers & Education* (impakt faktor: 2,621, kiadó: Elsevier) vagy a *Journal of Computer Assisted Learning* (impakt faktor: 1,464, kiadó: Wiley) folyóiratokat, illetve az olyan konferenciákat, mint a *3rd Annual International Conference on Computer Science Education: Innovation and Technology* (amely 2012. november 19–20-án került megrendezésre Szingapúrban) vagy a *Consortium for Computing Sciences in Colleges — Northeastern Region* (amelyre 2013. április 12–13-án került sor a New York állambeli Albanyban).

Bármiről szóljon is a feladat, a lényeg, hogy egy nagyobb projekt legyen, azaz több olyan tárgyat lefedjen, amellyel a hallgató találkozik a tanulmányai során (de lefedheti akár a legtöbb kötelező tárgyat is). A projektmunka során alkalmazásra kerülnek a projekthez kötődő tárgyak keretein belül tárgyalásra kerülő ismeretek. A hallgatók megismerik a tárgyalt témakörök alkalmazhatóságát, hasznosságát, valamint az alkalmazás során felmerülő problémákat. Azt gondolom, hogy ha megfelelő feladato-

kat tudunk találni, akkor jobb teljesítményt érhetünk el nemcsak a kiadott feladatok megoldásában, hanem a kurzusok vizsgái során is.

Ha vetünk egy pillantást a Debreceni Egyetem programtervező informatikus BSc szakának kötelező tantárgyainak listájára [5], hamar észrevehetjük, hogy még egy közepes méretű szoftverfejlesztési projekthez is szükség van legalább az alábbi három kötelező tárgyon tanult ismeretekre: *Bevezetés az informatikába*, *Adatszerkezetek és algoritmusok* és *Magas szintű programozási nyelvek 1*. Egy hosszú távú projektben azonban a legtöbb kötelező tárgy érintett lehet. Könnyen találhatunk olyan izgalmas, valamely valós világbeli alkalmazáshoz hasonló feladatokat, amelyek hosszú távú projektként szolgálhatnak. Ha már meghatároztuk a projekt célját, nincs más hátra, mint olyan feladatokat találni, amelyeket felhasználhatunk a kapcsolódó tantárgyak oktatásában. Véleményem szerint ha a hallgatók olyan összetett projekten dolgoznak, amelynek a fejlesztési folyamata felöleli a tanulmányaik csaknem teljes időtartamát, akkor motiváltabbak lesznek, és jobban fogják látni az egyes kurzusokon megtanult ismeretek közötti kapcsolatot.

Valós világbeli alkalmazások fejlesztéséhez kötődő feladatok kitűzése a következő előnyökkel jár:

- Az érdelődést felkeltő példák növelik a hallgatók motiváltságát.
- Egy összetett projekten keresztül a hallgatók az informatika számos területét gyakorolhatják.
- Ha több kurzuson is ugyanazzal az egy nagyobb projekttel találkoznak, az segít a hallgatóknak jobban megérteni a különböző kurzusok mögött rejlő ismeretek közötti összefüggéseket.
- A projektek gyakorlatorientáltabbá teszik az informatikaoktatást.
- A projektek hitelessé teszik az előadások során elsajátított ismereteket, és választ adnak arra a kérdésre is, hogy hogyan alkalmazzuk ezt a tudást.

Szeretném hangsúlyozni, hogy a projektorientált oktatás ötletét a magyar egyetemek legtöbb informatikai mesterképzésén már alkalmazzák. A jelenleg hatályos felsőoktatási törvény szerint még az alapképzéseknek is kell tartalmazniuk valamennyi projektmunkát. Kitűnő példája ennek az Eötvös Loránd Tudományegyetem Informatikai Kara, ahol a hallgatók részt vehetnek egy úgynevezett kooperatív képzésben 16 kreditért [3]. A kooperatív képzés célja, hogy a hallgatóknak lehetőséget biztosítson arra, hogy közelebbről is megismerkedhessenek az informatikus szakma gyakorlati oldalával valós informatikai cégek gyakorlott szakembereinek az irányításával. Egy másik példa az *Önálló laboratórium* című tárgy a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karán [21], illetve a Debreceni Egyetem Informatikai Karán, amelynek keretében a hallgatók elmélyíthetik ismereteiket és tapasztalatokat szerezhetnek az informatika egy bizonyos területén. Külföldi intézmények is alkalmazzák a projektalapú oktatást; a Paderborni Egyetemen például háromfős csoportoknak kell operációs rendszert fejleszteniük az egyik BSc-s kurzus keretében.

Van azonban néhány fontos különbség a javasolt projektalapú megközelítés és a fent említett példák között:

- Mind a kooperatív képzés, mind az *Önálló laboratórium* tárgyak függetlenek az alapképzések kötelező tárgyaitól abban az értelemben, hogy különálló oktatási egységnek számítanak. A javaslatom szerint a projektek a legtöbb kötelező tárgy oktatási anyagának szerves részét képeznék, így a hallgatók a már meglévő tárgyak kurzusain dolgozhatnak a projekteken, nincs szükség tehát újabb tárgyakra vagy képzésekre.

- Míg a kooperatív képzés és az *Önálló laboratórium* tárgyak az informatikának csak egy-két területére koncentrálnak, addig a javasolt megközelítés a legtöbb kötelező tárgy tematikáját felöleli.

- Mind a kooperatív képzésnek, mind az *Önálló laboratórium* tárgyaknak szigorú előfeltételei vannak, azaz a korábbi tanulmányok alatt elsajátított ismeretekre épülnek. A javasolt megközelítés használatával azonban a hallgatók már az első félév folyamán elkezdenek projekteken dolgozni. Ez azt is jelenti, hogy az oktatóknak sokkal több hallgatóval kell foglalkozniuk, akik korábban még nem vettek részt projektmunkában. Másrészt az oktatók sem rendelkeznek feltétlenül nagy tapasztalatokkal a projektmenedzsment területén, és együtt kell működniük egymással a jobb eredmény elérése érdekében.

- Bár a kooperatív képzés az oktatás részét képezi, az intézmény elveszíti a jogát a képzés és a számonkérések teljes körű kézben tartására. További hátránya, hogy a fővároson kívül nem olyan egyszerű a szükséges számban olyan céget találni, amely megfelelő projektekkel tud szolgálni.

8.2. Multiparadigmás programozási nyelvek használata a mesterséges intelligencia oktatásában

Kutatásaim során megvizsgáltam az F# mint egy új multiparadigmás programozási nyelv használatának lehetőségét a mesterséges intelligencia (MI) területén használt különféle algoritmusok kódolására. Három fő okból választottam ezt a területet. Egyrészt korábban oktatója voltam *A mesterséges intelligencia alapjai* című tárgy gyakorlatainak a Debreceni Egyetemen [28]. Másrészt az MI, azon belül konkrétan a kereső algoritmusok olyan számítási területet képviselnek, amely esetén jól alkalmazható a funkcionális programozás, mivel ezeknek az algoritmusoknak bizonyos részei (mint például az operátoralkalmazási előfeltételek ellenőrzése) alapvetően funkcionálisak. Harmadrészt rendelkezésemre állt a kérdéses algoritmusok néhány imperatív és objektumorientált implementációja, amelyek jó kiindulópontnak bizonyultak az F# verzió elkészítéséhez [13, 12].

Nem nehéz belátni, hogy az MI problémák megoldására a funkcionális programozás előnyösebb, mint az imperatív programozás. Ahogy a [20] cikkemben bemutattam, még az egyszerű 8 királynő problémának is sokkal tömörebb megoldását tudjuk megadni F#-ban (kizárólag funkcionális programozási elemek felhasználásával), mint C-ben vagy C#-ban. Ha azonban a hatékonyság vagy az újrafelhasználhatóság is szerepet

játszik, nem biztos, hogy a tisztán funkcionális kód lesz a legjobb megoldás bizonyos problémák esetén.

Egy-egy objektumorientált C# implementáción kívül elkészítettem három F# implementációt is az egyszereplős problémák kereső algoritmusaihoz, illetve kettőt a kétszemélyes játékok lépésajánló algoritmusaihoz. A célom az volt a különböző implementációk elkészítésével, hogy több megközelítést biztosítsunk a hallgatók számára ugyanazon pszeudokódok megértéséhez. Bár azt nem tudom, hogy funkcionálisan gondolkodva könnyebben megértik-e az algoritmusok működését, egynél több implementáció megismerése biztosan nem árthat. Íme egy különböző szempontokon alpuló, részben szubjektív összehasonlítása az egyszereplős problémák esetén használatos kereső algoritmusok négyféle implementációjának:

- *Forráskód metrikák*: Annak ellenére, hogy funkcionális nyelvek esetén nem alkalmazhatók ugyanazok a metrikák, mint imperatív nyelvek esetén, most három olyan metrikát fogok használni, amelyek mind C#-ban, mind F#-ban relevánsak lehetnek: a sorok száma, az osztályok száma és a függvények száma (beleértve a metódusokat is). Az 8.1. táblázat összefoglalja ezeknek a metrikáknak az értékeit a különböző implementációk esetén.

	C#	F# v1	F# v2	F# v3
Sorok száma	842	537	536	417
Osztályok száma	15	13	6	4
Függvények száma	49	43	43	34

8.1. táblázat. Néhány forráskód metrika.

A *sorok száma* metrika az összes forrásállományban szereplő kódsorok számát adja meg, beleértve az üres sorokat is, nem beleértve viszont a konkrét problémákhoz tartozó forráskódokat. Az *osztályok száma* a forráskódban definiált osztályokra vonatkozik, kivéve a kivétel osztályokat, a felsorolásos típusokat és az `IComparer` osztályokat. A *függvények száma* tartalmazza az összes függvényt és konkrét metódusimplementációt, beleértve a konstruktorokat, nem beleértve viszont a lambdakifejezéseket.

Ahogy látható, a harmadik F# implementáció feleakkora méretű, mint a C# implementáció. Ez a különbség persze főleg az F# nyelv kompakt szintaxisából következik. A másik oka annak, hogy az F# implementációk rövidebbek, az az, hogy hiányzik belőlük két osztály a C# kódból, amelyek 57 sort tesznek ki.

Az osztályok számának csökkenése az egyre „funkcionálisabb” kód eredménye. Az első F# implementáció – ahogy fent említettem – két osztállyal kevesebbet tartalmaz, mint a C# verzió. A második változatban a hét konkrét algoritmus osztályt hét függvénnyel helyettesítettük. A harmadik változatban a két absztrakt kereső osztályt is függvénnyé konvertáltuk. Ha tisztán funkcionális kódot szeretnénk, a maradék osztályoktól is meg kellene szabadulnunk, de az nem eredményezne rövidebb vagy olvashatóbb kódot. Az osztályok használatával könnyű *újrafelhasználható*

kódot írni például az operátoralkalmazásra: csupán meg kell hívnunk az absztrakt `State` osztály `Apply` metódusát, anélkül hogy tudnánk, hogyan is implementálta azt valamely konkrét probléma állapotait reprezentáló konkrét osztály. A két `Node` osztály konstruktorai pedig pontosan ezt csinálják. A két `Node` osztályt valójában lecserélhetnénk rekord típusokra, mivel belőlük nem származnak más osztályok, de így sem kapnánk olvashatóbb kódot, ráadásul nem tudnánk használni az olyan .NET metódusokat, mint például a `Contains`.

A függvények száma nagyon hasonló az egyes implementációkban. Ez azért van, mert az osztályok metódusai egy-egy függvénynek felelnek meg az osztálymentes implementációkban; még a konstruktoroknak is függvényeket feleltetünk meg. A harmadik F# implementáció azonban valamivel kevesebb függvényt tartalmaz, mint a másik három. Ennek az az oka, hogy hiányzik belőle a `ToString` metódusnak a konkrét algoritmus osztályokban (illetve objektumokban) megtalálható hét implementációja. Ezeket mindössze egyetlen függvény, a `printSearchInfo` helyettesíti. Ugyanez érvényes a gráfkereső algoritmusok `Expand` és `Search` metódusaira. A három metódusnak megfelelő függvények mindegyike `match` kifejezéseket tartalmaz, amelyeknek az alapja a kereséshez használt algoritmus. Ez jól mutatja a különbséget annak a problémának az objektumorientált és funkcionális megközelítése között, hogy hogyan tudunk egy alaposztályhoz új alosztályt, illetve az alosztályokhoz új funkcionalitást bevezetni. Ha új alosztályt szeretnénk bevezetni, akkor az OO megközelítés jobb, mert nem kell hozzányúlnunk a már meglévő alosztályokhoz, csak meg kell írnunk az új osztályt az alaposztályból örökölt összes funkcionalitással. Funkcionális megközelítés esetén ezzel szemben az összes `match` kifejezéshez egy-egy új ágat kell hozzávennünk. Ha viszont új funkcionalitással szeretnénk bővíteni a már meglévő alosztályainkat, akkor a funkcionális módszer a jobb, hiszen csak egy új függvényt kell írnunk a meglévőkhöz hasonló `match` kifejezéssel. OO megközelítést használva az alaposztályt ki kell egészítenünk egy új metódussal, valamint az összes alosztályt az új metódus egy-egy implementációjával.

- *A felhasznált módosítható adatszerkezetek*: Ebben a tekintetben nincs különbség az implementációk között. Bár a tisztán funkcionális programok egyáltalán nem alkalmaznak módosítható adatokat, mindegyik implementációban használtam néhány módosítható adatszerkezetet. Ha szeretnénk, ezen adatkollekciók bármelyikének a típusát lecserélhetnénk az F# nem módosítható `list` vagy `seq` adattípusára. Az így kapott kód ugyanolyan méretű lenne, viszont kevésbé hatékony, mivel a `List` és `Seq` modulok rekurzív függvényei lassabbak, mint a megfelelő .NET metódusok. Ez különösen akkor igaz, amikor elemeket adunk hozzá ezekhez a kollekciókhoz, vagy amikor elemeket törlünk belőlük: egy nem módosítható adatszerkezet esetén le kell másolnunk az eredeti kollekciót, egy apró módosítást végrehajtva az elemein. Ez az oka annak, hogy .NET kollekciókat használtam az F# nem módosítható adattípusai helyett a kérdéses adatszerkezetek tárolására.

- *A felhasznált funkcionális nyelvi eszközök*: A C# implementáció egyetlen funkcionális elemet sem tartalmaz, tisztán objektumorientált. Az első F# implementáció farokrekurzív függvényekkel és szekvenciaműveletekkel helyettesíti a ciklusokat.

Bár az objektumkifejezések nem funkcionális nyelvi eszközök, a második F# implementáció sokat tartalmaz belőlük az alosztályok kiváltására. A harmadik F# implementáció tartalmazza a legtöbb funkcionális elemet: diszkriminált uniót használunk az algoritmus típusának tárolására, `match` kifejezéseket a kezelésére, néhány függvényt első osztályú értékként használunk, és van benne egy magasabb rendű függvény is.

- *Hatékonyság*: Mivel a funkcionális nyelvek absztraktabbak, mint az objektumorientált nyelvek, összetettebb futtató rendszerre van szükségük. Ez a fő oka annak, hogy a funkcionális programok kevésbé hatékonyak, még akkor is, ha lefordítjuk, nem pedig interpretáljuk őket. Lefuttattam az összes implementáció bemutatott főprogramjait ugyanazon a számítógépen, egy Gigabyte T1018X TouchNote netbookon Intel Atom N280 processzorral 1,33 GHz-en és 1 GB memóriával, az összes programot Microsoft Visual Studio 2010-ben fordítva: a C# program kevesebb, mint fél másodperc alatt végzett, míg az F# programok mindegyike nagyjából 7 másodpercig futott. Ahogy írtam, ennél is rosszabb eredményt kaptunk volna, ha az F# nem módosítható adattípusait használtuk volna.

Az egyszereplős problémák esetén alkalmazott algoritmusok különböző implementációira vonatkozó megfigyelések a kétszemélyes játékok lépésajánló algoritmusaira is érvényesek, bár a különbség az utóbbi esetben kisebb. Ugyanez mondható el a konkrét problémák és játékok implementációival kapcsolatban is.

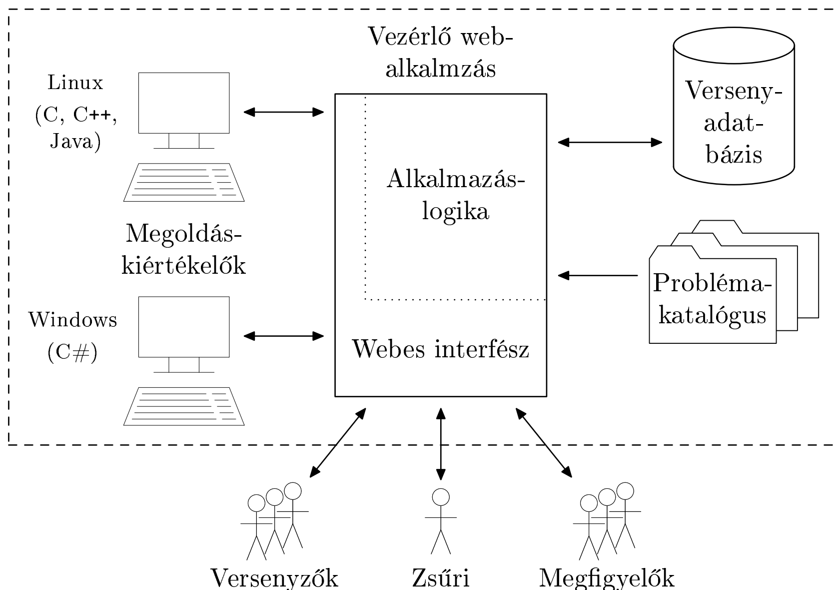
Végső konklúzióként az a véleményem, hogy nem érdemes az egyik vagy a másik paradigmához ragaszkodni, ha többet is használhatunk egy programon belül. A funkcionális kód néha absztraktabb, olvashatóbb vagy egyszerűen csak rövidebb, mint az objektumorientált megfelelője. Másrészről az OO kód általában hatékonyabb és néha újrafelhasználhatóbb, mint a funkcionális megfelelője. Ezért azt gondolom, hogy a multiparadigmás nyelvek, mint amilyen az F#, előnyösebbek lehetnek főleg nagyobb szabású alkalmazások esetén, de akár kisebb programokban is. A hallgatók különbözőek, így néhányuk jobban megértheti az algoritmusokat egy funkcionálisabb megközelítésre, mint egy tisztán objektumorientált kódra alapozva. Például a `Seq.fold` függvény egy akkumulátor paraméterrel ellátott lambda-kifejezéssel jobban leírhatja egy minimum- vagy maximumkiválasztás működését az olyan hallgatók számára, akik rekurzívan (funkcionálisan) gondolkodnak.

8.3. Programozó versenyek lebonyolítása a ProgCont alkalmazással

A programozó versenyek erős motivációs tényezőt jelenthetnek. Több mint tíz éve rendezünk félévenként legalább egy házi versenyt, és elmondhatom, hogy legalább néhány hallgató érdeklődését felkeltik ezek a versenyek. A tíz év alatt két alkalmazást is kifejlesztettünk a versenyzők által beküldött megoldások on-line kiértékelésére. Az első neve *Programming Contest Result Manager* (PCRM), és egy e-mail-alapú konzolos alkalmazás, a másodikat pedig *ProgCont*-nak hívják, amely egy kliens/szerver archi-

tektúrájú webalkalmazás. A PCRM-et részletesen tárgyalják a [14, 12] tanulmányok, most a ProgContot mutatom be a vele szerzett tapasztalatainkkal együtt.

A ProgCont rendszer négy, egymástól jól elkülöníthető összetevőből épül fel: problémakatalógus, versenyadatbázis, vezérlő webalkalmazás és megoldáskiértékelő kliensek (lásd az 8.1. ábrát).



8.1. ábra. A ProgCont rendszer felépítése.

A *problémakatalógus* tartalmazza a később összeállított versenyekre szánt anyagot, a problémák – esetenként többnyelvű – leírását a hozzá tartozó ábrákkal és további (akár letölthető) segédanyagokkal együtt, valamint szintén itt találhatók a megoldások ellenőrzéséhez szükséges tesztesetek is. Egy-egy problémához több teszteseteket tartalmazó állomány is készíthető. A megoldások helyességének ellenőrzése az alábbi módokon történhet: a program ellenőrizheti, hogy a beküldött megoldás által előállított kimenet karakterenként megegyezik-e egy előre generált kimenettel, illetve a kimenet helyessége vizsgálható külső programmal. Tesztesetenként határozható meg a kimenet előállításának időkorlátja is. A problémakatalógus az operációs rendszer könyvtár- és fájlszerkezetének segítségével alakítható ki. Minden probléma külön könyvtárban kap helyet. A probléma szövege egy XML fájlban, a hozzá tartozó tesztesetek, az előre generált kimeneti állományok és a tesztelési paraméterek pedig egy tömörített (ZIP) fájlban tárolódnak. A szöveges leírásban hivatkozott további dokumentumok szintén ugyanebben a könyvtárban kapnak helyet.

A *versenyadatbázis* írja le a versenyek, a feladatok, a versenyzők, a megoldások és a kiértékelések kapcsolatát. Egy-egy verseny a problémakatalógusból válogatott feladatok, csapatverseny esetében a csapatokat alkotó versenyzők, valamint a verseny lebonyolítására vonatkozó technikai adatok (például a verseny kezdetének és befejezé-

sének az ideje vagy az egyes feladatok esetében engedélyezett programozási nyelvek) összessége. Az adatbázis tárolja a versenyzők által beküldött megoldásokat, majd a megoldások értékelésének az eredményét, amelyeket egy-egy megoldáskiértékelő kliens szolgáltat. Mindezeket a ProgCont alkalmazásban egy PostgreSQL adatbázisban tároljuk.

Egy-egy verseny lebonyolítását a *vezérlő webalkalmazás* irányítja. A versenyzők a webalkalmazás segítségével böngészhetik a feladatokat, tölthetik fel az egyes feladatok megoldásait, és értesülhetnek a megoldáskiértékelések eredményéről és a verseny állásáról. A megfigyelők (például a versenyzők felkészítői vagy a vendégek) szintén követhetik az aktuális eredménylistát. A zsűri a webes felületen tudja beállítani a verseny bizonyos paramétereit, például azt, hogy az egyes feladatokra milyen programozási nyelven várja a megoldásokat, hogy hány pontot érnek az egyes feladatok, vagy hogy mennyi büntetőidő jár egy-egy rossz megoldásért. A webalkalmazás ütemezi a beküldött megoldások kiértékelésének sorrendjét is, ami azt jelenti, hogy ő osztja ki azokat az egyes, a rendszerhez éppen csatlakozó megoldáskiértékelő kliensek számára. A programkódok kiosztása a kliensek által küldött azon információ alapján történik, hogy milyen programnyelvű kódok kezelésére alkalmasak. A vezérlő webalkalmazás futtatására Apache Tomcat webalkalmazás-szervert használunk. A webalkalmazás és a felhasználók (a versenyzők és a zsűri) közötti kommunikáció biztonságát az SSL (HTTPS) protokoll szavatolja. A webalkalmazás JDBC-n keresztül éri el a versenyadatbázist.

A *megoldáskiértékelő kliensek* olyan önálló alkalmazások, amelyek periodikusan (5 másodpercenként) ellenőrzik, hogy van-e olyan kiértékelésre várakozó megoldás, amelyet kezelni tudnak. Ha van, a kliens első lépésben megpróbálja lefordítani a vezérlő webalkalmazás által neki kiosztott programkódot. Amennyiben a fordítás sikeres, lefuttatja a programot minden tesztesetre egy előre beállított futtatási környezetben. Az aktuális teszteseteket a problémakatalógusból tölti le a webalkalmazáson keresztül, valahányszor megváltozik az őket tartalmazó ZIP állomány (és persze az első alkalommal, amikor szükség van rájuk). A tesztesetek kiértékelésekor a megoldáskiértékelő kliens figyelembe veszi a tesztesetre beállított maximális futási időt. Ha a program időben megáll, a kliens ellenőrzi a kimenetét az előre beállított módszernek megfelelően: vagy összehasonlítja a letöltött kimeneti állománnyal, vagy továbbítja a külső kiértékelő programnak. Az összesített eredményt végül továbbítja a vezérlő webalkalmazás felé.

Mivel az egyes programozási nyelvekhez különböző operációs rendszerek passzolnak inkább, a megoldáskiértékelő klienseket platformfüggetlen Java implementációval készítettük. Így lehetőség van arra, hogy a vezérlő webalkalmazástól és a többi kiértékelő klienstől különböző (lehetőleg virtuális) számítógépeken fussanak, ezáltal nem veszélyeztetve azok működését, amennyiben egy kártékony kód tönkretenné egy konkrét kliens futtatási környezetét. A C# kódokat például fordíthatja és futtathatja egy Windows-alapú kliens, míg a C, C++, Java és Pascal kódokhoz egy másik, Linux-alapú klienst használhatunk. Minél több megoldáskiértékelő klienst használunk, annál több kiértékelést hajthatunk végre egyidőben.

2011. október 2-án rendeztük az első olyan versenyt, amelyet a ProgCont alkalmazással vezéreltünk. Az eredeti célunknak megfelelően ez az adott év ACM versenyének helyi fordulója volt. Kihasználva a versenyek paraméterezhetőségének a lehetőségét,

2012. február 6-án egy rövid egyéni versenyre került sor az *Informatikai versenyfeladatok* című kurzus hallgatói között. Ugyanabban a félévben három különböző tantárgy ismereteinek az elmélyítésében is tudtunk segíteni a hallgatóknak olyan versenyekkel, amelyek több mint egy hónapig tartottak: *Informatikai versenyfeladatok*, *Magas szintű programozási nyelvek 1* és *A mesterséges intelligencia alapjai.* Ezután egy ACM jellegű versenyt szerveztünk 2012. május 6-án (amely a marosvásárhelyi ECN Nemzetközi Programozó Verseny selejtezője volt), egy újabb helyi ACM versenyt 2012. október 7-én, végül, de nem utolsósorban pedig az Informatikai Kar Regionális Programozó Csapatversenyét 2012. november 25-én.

Bibliography

[1] Juan Albornoz Bueno and Raúl Alfredo Chaparro Aguilar. The learning of fundamental concepts and problem solving strategies in computer science, through the experimentation and classroom research with discrete games. *Proceedings of the 9th International Conference on Engineering Education*, July 23–28, 2006.

[2] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):354–363, 1936.

[3] Cooperative training at the Eötvös Loránd University, Faculty of Informatics. http://www.inf.elte.hu/karunkrol/oktatas/kepzeseink/kooperativkepzes/Lapok/altalanosleiras.aspx.

[4] Zoltán Csörnyei. *Lambda-kalkulus – A funkcionális programozás alapjai*. Typotex, Budapest, 2007.

[5] Degree requirements for the Software Information Technology BSc major at the University of Debrecen.
http://www.inf.unideb.hu/oktatas/?cat=&site=hallgato/nappali/oklevel_kovetelmeny/pti_2007.

[6] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] Jeffrey Michael Edgington. Toward using games to teach fundamental computer science concepts. Doctoral dissertation, University of Denver.

[8] István Fekete, Tibor Gregorics, and Sára Nagy. *Bevezetés a mesterséges intelligenciába*. ELTE Eötvös Kiadó, Budapest, 2006.

[9] Iván Futó, editor. *Mesterséges intelligencia*. Aula, Budapest, 1999.

[10] István Juhász, Márk Kósa, and János Pánovics. *C példatár*. Panem, Budapest, 2005.

[11] David J. King and John Launchbury. Structuring depth-first search algorithms in Haskell. *Proceedings of the 22nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 344–354, 1995.

[12] Márk Kósa. Korszerű információtechnológiai módszerek bevezetése a mesterséges intelligencia oktatásába. Doctoral dissertation, University of Debrecen.

[13] Márk Kósa and János Pánovics. Keresőalgoritmusok objektumorientált megközelítése a Mesterséges intelligencia tárgy bevezető kurzusán. *Proceedings of the 17th International Conference on Computers and Education*, pages 94–97, 2007.

[14] Márk Kósa, János Pánovics, and Lénárd Gunda. An evaluating tool for programming contests. *Teaching Mathematics and Computer Science*, 3(1):103–119, 2005.

[15] Scott T. Leutenegger and Jeffrey Edgington. A games first approach to teaching introductory programming. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, 39(1):115–118, 2007.

[16] George F. Luger and William A. Stubblefield. *AI algorithms, data structures, and idioms in Prolog, Lisp, and Java*. Pearson Education, Boston, 2009.

[17] Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. *Proceedings of the International Conference on Information Processing*, pages 256–264, 1959.

[18] Bernd Owsnicki-Klewe. Search algorithms. http://users.informatik.haw-hamburg.de/~owsnicki/search.html.

[19] Tomas Petricek and Jon Skeet. *Real-world functional programming*. Manning Publications, Greenwich, 2010.

[20] János Pánovics. A functional programming approach to AI search algorithms. *Journal of Information Technology Education: Innovations in Practice*, 11:353–376, 2012.

[21] Project Laboratory at the Budapest University of Technology and Economics. https://www.vik.bme.hu/kepzes/targyak/VIAUA354.

[22] Stuart J. Russell and Peter Norvig. *Mesterséges intelligencia modern megközelítésben*. Panem, Budapest, 2005.

[23] Michael L. Scott. *Programming language pragmatics*. Morgan Kaufmann, San Francisco, 2009.

[24] Robert W. Sebesta. *Concepts of programming languages*. Pearson Education, New Jersey, 2012.

[25] Raymond M. Smullyan. *Gödel's incompleteness theorems*. Oxford University Press, New York, 1992.

[26] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 2.0.* Apress, New York, 2010.

[27] D. W. Valentine. Playing around in the CS curriculum: Reversi as a teaching tool. *Journal of Computing Sciences in Colleges*, 20(5):214–222, 2005.

[28] Magda Várterész, Benedek Nagy, Márk Kósa, and János Pánovics. A Mesterséges intelligencia tárgy bevezető kurzusának gyakorlatai a Debreceni Egyetemen. *Proceedings of the Conference on Informatics in Higher Education 2002*, pages 1103–1109, 2002.

List of Publications

Peer-Reviewed Papers in the Subject of the Dissertation

1. Kádek Tamás, **Pánovics János**: Extended breadth-first search algorithm, *International Journal of Computer Science Issues* (2013) **10** (6), No. 2, pp. 78–82. Impact Factor: 0.242.

2. **Pánovics János**: Motivating students with projects encompassing the whole duration of their studies, *Teaching Mathematics and Computer Science* (2013) **11** (2), pp. 165–180.

3. **Pánovics János**: A functional programming approach to AI search algorithms, *Journal of Information Technology Education: Innovations in Practice* (2012) **11**, pp. 353–376. Impact Factor: h-index: 28.

   Ref. number: Education Resources Information Center ERIC #EJ990988.
   (http://www.eric.ed.gov)

4. Kósa Márk, **Pánovics János**: Search algorithms at ACM contests, *Proceedings of the 7th International Conference on Applied Informatics (ICAI 2007)*, Eger, Hungary, January 28–31, 2007, Vol. II., pp. 367–375 (2009).

   Ref. number: Zentralblatt MATH Database Zbl 1183.68722.
   (http://www.zentralblatt-math.org/zbmath/)

5. Kósa Márk, **Pánovics János**, Gunda Lénárd: An evaluating tool for programming contests, *Teaching Mathematics and Computer Science* (2005) **3** (1), pp. 103–119.

   Ref. number: Zentralblatt MathEduc Database ME 2005f.02643.
   (http://www.zentralblatt-math.org/matheduc/)

Book in the Subject of the Dissertation

6. Juhász István, Kósa Márk, **Pánovics János**: *C példatár* (in Hungarian), Panem, Budapest, 2005.

Conference Proceedings

7. Kádek Tamás, **Pánovics János**: *Általános állapottér modell* (in Hungarian), 23rd International Conference on Computers and Education, 2013, Sibiu, Romania, pp. 214–218.

8. Kósa Márk, **Pánovics János**: *Megoldáskereső algoritmusok programozása funkcionális megközelítésben* (in Hungarian), 22nd International Conference on Computers and Education, 2012, Alba Iulia, Romania, pp. 294–299.

9. Kósa Márk, **Pánovics János**: *Programming artificial intelligence search algorithms in a functional approach*, New Technologies in Science, Research and Education, XXV. DIDMATTECH 2012 International Scientific and Professional Conference, 2012, Komárno, Slovakia, pp. 144–151.

10. Kádek Tamás, Kósa Márk, **Pánovics János**: *A ProgCont szoftverrel támogatott programozó versenyek tapasztalatai* (in Hungarian), New Technologies in Science, Research and Education, XXV. DIDMATTECH 2012 International Scientific and Professional Conference, 2012, Komárno, Slovakia, pp. 152–157.

11. Kádek Tamás, Kósa Márk, **Pánovics János**: *Programozó versenyek támogatása webes alkalmazással* (in Hungarian), 21st International Conference on Computers and Education, 2011, Cluj-Napoca, Romania, pp. 184–187.

12. Kósa Márk, **Pánovics János**: *Kétszemélyes játékok optimalizálásának lehetőségei mobil eszközökre* (in Hungarian), Conference on Informatics in Higher Education 2011, Debrecen, pp. 200–206.

13. Kósa Márk, **Pánovics János**, Tőzsér Tamás: *Az amőba játék optimalizálásának lehetőségei mobil eszközökre* (in Hungarian), 20th International Conference on Computers and Education, 2010, Satu Mare, Romania, pp. 150–154.

14. Kósa Márk, **Pánovics János**: *Object-oriented approach of search algorithms for two-player games*, 8th International Conference on Applied Informatics (ICAI 2010), Eger, Vol. II., pp. 29–34.

15. Kósa Márk, **Pánovics János**: *Kétszemélyes játékok lépésajánló algoritmusai objektumorientált megközelítésben* (in Hungarian), 19th International Conference on Computers and Education, 2009, Târgu Mureş, Romania, pp. 263–266.

16. Kósa Márk, **Pánovics János**: *Szoftverfejlesztés a Qt keretrendszer használatával* (in Hungarian), 18th International Conference on Computers and Education, 2008, Şumuleu Ciuc, Romania, pp. 179–184.

17. Kósa Márk, **Pánovics János**: *Keresőalgoritmusok objektumorientált megközelítése a Mesterséges intelligencia tárgy bevezető kurzusán* (in Hungarian), 17th International Conference on Computers and Education, 2007, Oradea, Romania, pp. 94–97.

18. Kósa Márk, Nagy Benedek, **Pánovics János**: *Megoldáskereső algoritmusok hatékonyságának vizsgálata az állapottér-reprezentációk függvényében* (in Hungarian), 16th International Conference on Computers and Education, 2006, Sovata, Romania, pp. 76–81.

19. Kósa Márk, **Pánovics János**, Gunda Lénárd: *An evaluating tool for programming contests*, 6th International Conference on Applied Informatics (ICAI 2004), Eger, Vol. I., pp. 163–172.

20. Várterész Magda, Nagy Benedek, Kósa Márk, **Pánovics János**: *A Mesterséges intelligencia tárgy bevezető kurzusának gyakorlatai a Debreceni Egyetemen* (in Hungarian), Conference on Informatics in Higher Education 2002, Debrecen, pp. 1103–1109.

Conference Talks in the Subject of the Dissertation

21. Kádek Tamás, **Pánovics János**: *Általános állapottér modell* (in Hungarian), 23rd International Conference on Computers and Education, 2013, Sibiu, Romania, October 10–13, 2013.

22. **Pánovics János**: *Projektközpontú szemlélet az informatikaoktatásban* (in Hungarian), Sapientia MatInfo Conference, Târgu Mureş, Romania, May 25–26, 2013.

23. Kósa Márk, **Pánovics János**: *Megoldáskereső algoritmusok programozása funkcionális megközelítésben* (in Hungarian), 22nd International Conference on Computers and Education, 2012, Alba Iulia, Romania, October 11–14, 2012.

24. Kósa Márk, **Pánovics János**: *Programming artificial intelligence search algorithms in a functional approach*, XXV. DIDMATTECH 2012 International Scientific and Professional Conference, Komárno, Slovakia, September 10–13, 2012.

25. Kádek Tamás, Kósa Márk, **Pánovics János**: *Our experiences on the programming contests supported by the ProgCont software*, XXV. DIDMATTECH 2012 International Scientific and Professional Conference, Komárno, Slovakia, September 10–13, 2012.

26. Kádek Tamás, Kósa Márk, **Pánovics János**: *Programozó versenyek támogatása webes alkalmazással* (in Hungarian), 21st International Conference on Computers and Education, 2011, Cluj-Napoca, Romania, October 6–9, 2011.

27. Kósa Márk, **Pánovics János**: *Object-oriented approach of search algorithms for two-player games*, 8th International Conference on Applied Informatics, Eger, January 27–30, 2010.

28. Kósa Márk, **Pánovics János**: *Kétszemélyes játékok lépésajánló algoritmusai objektumorientált megközelítésben* (in Hungarian), 19nd International Conference on Computers and Education, 2009, Târgu Mureş, Romania, October 8–11, 2009.

29. Kósa Márk, **Pánovics János**: *Keresőalgoritmusok objektumtumorientált implementációja a Mesterséges intelligencia tárgy gyakorlati kurzusain* (in Hungarian), Conference on Informatics in Higher Education 2008, Debrecen, August 27–29, 2008.

30. Kósa Márk, **Pánovics János**: *Keresőalgoritmusok objektumorientált megközelítése a Mesterséges intelligencia tárgy bevezető kurzusán* (in Hungarian), 17th International Conference on Computers and Education, 2007, Oradea, Romania, October 11–14, 2007.

31. Kósa Márk, **Pánovics János**: *Search algorithms at ACM contests*, 7th International Conference on Applied Informatics, Eger, January 28–31, 2007.

32. Kósa Márk, Nagy Benedek, **Pánovics János**: *Performance analysis of search algorithms depending on the state space representation*, 6th Joint Conference on Mathematics and Computer Science, Pécs, July 12–15, 2006.

33. Kósa Márk, Nagy Benedek, **Pánovics János**: *Megoldáskereső algoritmusok hatékonyságának vizsgálata az állapottér-reprezentációk függvényében* (in Hungarian), 16th International Conference on Computers and Education, 2006, Sovata, Romania, May 25–28, 2006.

34. Kósa Márk, **Pánovics János**, Gunda Lénárd: *PCRM: kiértékelő szoftver programozói versenyekhez* (in Hungarian), Conference on Informatics in Higher Education 2005, Debrecen, August 24–26, 2005.

35. Kósa Márk, **Pánovics János**, Gunda Lénárd: *An evaluating tool for programming contests*, 6th International Conference on Applied Informatics, Eger, January 27–31, 2004.

36. Várterész Magda, Nagy Benedek, Kósa Márk, **Pánovics János**: *A Mesterséges intelligencia tárgy bevezető kurzusának gyakorlatai a Debreceni Egyetemen* (in Hungarian), Conference on Informatics in Higher Education 2002, Debrecen, August 28–30, 2002.

Further Conference Talks

37. Kósa Márk, **Pánovics János**: *Kétszemélyes játékok optimalizálásának lehetőségei mobil eszközökre* (in Hungarian), Conference on Informatics in Higher Education 2011, Debrecen, August 24–26, 2011.

38. Kósa Márk, **Pánovics János**, Tőzsér Tamás: *Az amőba játék optimalizálásának lehetőségei mobil eszközökre* (in Hungarian), 20th International Conference on Computers and Education, 2010, Satu Mare, Romania, October 7–10, 2010.

39. Kósa Márk, **Pánovics János**: *Szoftverfejlesztés a Qt keretrendszer használatával* (in Hungarian), 18th International Conference on Computers and Education, 2008, Şumuleu Ciuc, Romania, October 9–12, 2008.

40. Kósa Márk, **Pánovics János**, Heinrich du Toit, Nyakóné Juhász Katalin, John Pilcher: *Felhasználói felület fejlesztése az EPICS rendszerhez* (in Hungarian), Conference on Informatics in Higher Education 2008, Debrecen, August 27–29, 2008.

41. Nyakóné Juhász Katalin, Kósa Márk, **Pánovics János**, Lajkó Károly: *Öt éves múlt a TMCS* (in Hungarian), Conference on Informatics in Higher Education 2008, Debrecen, August 27–29, 2008.

Further Publications and Software Products

42. Kádek Tamás, Kósa Márk, **Pánovics János**: *ProgCont*, 2011 (7585 lines of Java code, 2724 lines of JSP code, 2499 lines of SQL code).

43. Kósa Márk, **Pánovics János**: *Programming Contest Result Manager*, Technical reports, 2009/9, Preprints No. 369, Faculty of Informatics, University of Debrecen.

44. Kósa Márk, **Pánovics János**: *Példatár a Programozás 1 tárgyhoz* (in Hungarian), e-learning material, MobiDiák project, 2005.

45. Gunda Lénárd, Kósa Márk, **Pánovics János**: *Programming Contest Result Manager*, 2003 (10588 lines of C++ code).