SZAKDOLGOZAT

Juhász Gábor

Debrecen
2009

Debreceni Egyetem
Informatikai Kar

Valós idejű stratégiai játék útvonalkereső algoritmusai

Témavezető:
Espák Miklós
Egyetemi tanársegéd

Készítette:
Juhász Gábor
Programtervező informatikus

Debrecen
2009

Tartalomjegyzék

1. Bevezetés

A való idejű stratégiai játékok a számítógépes játékok egy olyan fajtái, amelyekben épületeket építhetünk, egységeket gyárthatunk, és ezek segítségével kell legyőzni az ellenfeleinket. Az épületek helyhez kötöttek, míg az egységek mozoghatnak a térképen. A térképen nyersanyagok is találhatóak, amelyek az épületek és egységek gyártásához szükségesek. A játékot úgy lehet elképzelni, mintha egy hadsereg parancsnokai lennénk, és mi irányítanánk az összes épület működését, és az összes egység cselekedeteit. Az egységeket tehát úgy irányítjuk, hogy „menj ide”, „lődd ezt” és hasonló utasításokat adunk nekik. Mindezt általában a billentyű és egér segítségével.

1. ábra: Pillanatkép a Warcraft nevű valós idejű stratégiai játék első részéből

Az ilyen játékokban tehát a játékos elvárja, hogy az egységek úgy mozogjanak, ahogyan egy ember tenné, tehát próbálják a lehető legegyszerűbb módon elérni a céljaikat. A játékélményt nagyban csökkenti, ha az egységek egy hosszú utat választanak a céljaikhoz, amikor egy sokkal rövidebb is létezik. Az út kiszámítását a játékokban útvonalkereső algoritmusok végzik. Tehát ezek az algoritmusok minden ilyen játék alapját képezik, és a fejlesztők célja minél hatékonyabb megoldásokat használni.

A dolgozatomban ismertetni szeretném az ilyen algoritmusok hátterét, működését, illetve egy olyan program elkészítését, amely alapja lehet egy valós idejű stratégiai játéknak, és implementál egy útvonalkereső algoritmust.

2. A probléma leírása

Amikor számítógéppel szeretnénk megoldani egy problémát, először olyan formába kell hoznunk, hogy az a számítógép számára feldolgozható legyen. Ezt a célt szolgálják a problémareprezentációs technikák, melynek egyik fajtája az állapottér-reprezentáció. Léteznek emellett másfajta probléma leíró technikák is, mint például probléma redukció, vagy logika alapú reprezentáció, de ezekkel a szakdolgozatomban nem kívánok foglalkozni. Az állapottér-reprezentációban csak a probléma szempontjából lényeges tulajdonságokat vesszük figyelembe. Például az útvonalkeresésben az egység pozíciója lényeges tulajdonságnak tekinthető, míg valószínűleg nem szeretnénk, ha például a játékos neve befolyásolná az útvonalkeresés eredményét.

2.1 Állapottér-reprezentáció

Legyen *p* egy probléma. Keressük meg azon tulajdonságokat *p* világában, amelyek lényegesek a *p* szempontjából. Legyen *n* db ilyen jellemző, melyek rendre $h_1 ... h_n$. Ekkor a $(h_1 ... h_n)$ érték n-est *p* világában egy állapotnak nevezzük. Jelölje az i. tulajdonság értékkészletét H_i *(i=1,…n).* Ekkor *p* világának állapotai a $H_1 \times ... \times H_n$ halmaz elemei. Nem biztos azonban, hogy a Descartes szorzat minden eleme előfordulhat állapotként, ezért definiálhatunk egy kényszerfeltételt, amely leszűkíti az állapotteret. A kényszerfeltétel egy függvény, amit jelöljünk *kf(a)*-val, ahol $a \in H_1 \times ... \times H_n$. A függvény igaz értékkel tér vissza, ha *a* állapot valódi állapot, és hamis értékkel, ha nem. Ezt a halmazt nevezzük állapottérnek, és *A*-val jelöljük:

$$A = \{a \mid a \in H_1 \times ... \times H_n \wedge kf(a)\}$$

A probléma világában léteznek még úgynevezett operátorok. Az operátorok egy állapotból egy másik állapotot hoznak létre, vagyis állapotból állapotba képező függvények. Jelöljük *O*-val az operátorok halmazát. Ekkor $o : A \rightarrow A$, ahol $o \in O$. Mivel nem biztos, hogy minden operátor alkalmazható minden állapotban, ezért megadhatjuk még az operátorok alkalmazási előfeltételét egy $ef_o(a)$ függvénnyel, amelyben $o \in O$ és $a \in A$. A függvény igaz értékkel tér vissza, ha *o* alkalmazható az *a* állapotra, tehát a képzett állapot szintén eleme az állapottérnek, és hamissal, ha nem. Így felírható az operátor értelmezési tartománya:

$$dom(o) = \{a \mid a \in A \wedge ef_o(a)\}.$$

Az operátornak megadhatunk alkalmazási költséget is, amely segítségével előnyben részesíthetjük bizonyos operátorok alkalmazását.

Kezdőállapotnak nevezzük azt a kitüntetett állapotot, amely állapotból indulunk. Ezt jelöljük *k*-val, $k \in A$.

A célállapot az az állapot, amelybe el szeretnénk jutni. Célállapotból több is lehet, ezért a célállapotok halmazát adjuk meg, amelyet *C*-vel jelölünk. Megadhatjuk explicit módon, vagyis felsoroljuk az elemeit, és implicit módon a következőképpen:

$$C = \{a \mid a \in A \wedge cf(a)\}$$

A *cf(a)* függvény a célfeltétel vizsgálatára szolgáló függvény, amely igaz értékkel tér vissza, ha *a* eleme a célállapotok halmazának, és hamissal, ha nem. Ez a megadási mód akkor is hasznos lehet, ha nem ismerjük pontosan a célállapotot, vagy ha túl sok célállapot létezik.

Tehát a fenti jelöléseket használva a *p* probléma állapottér-reprezentációját az $\langle A, k, C, O \rangle$ négyessel definiálhatjuk. [1]

2.2 Az állapottér-gráf

Az állapottér meghatároz egy irányított gráfot, melyet a probléma állapottér-gráfjának nevezünk. A gráf minden csomópontja egy állapotnak felel meg. Egy csomópontból pontosan annyi irányított él indul, ahány alkalmazható operátor létezik a csomópont által reprezentált állapotra, és ezek mindegyike a hozzátartozó operátor alkalmazása által generált állapotnak megfelelő csomópontra mutat. Útnak nevezzük a gráfban olyan élek sorozatát, amely egy csomópontot legfeljebb egyszer tartalmaz. Legyen *A* és *B* csomópontok egy gráfban. *A*-ból közvetlenül elérhető *B* pontosan akkor, ha *A*-ból vezet él *B*-be. Ekkor *B*-t *A* szomszédjának nevezzük. Ezen kívül *A*-ból elérhető *B*, ha létezik a gráfban olyan út, melynek kiindulópontja *A*, és végpontja *B*. A gráf tartalmazhat hurkokat és köröket. Hurok van a gráfban, ha egy csomópontból egy másik csomópont több úton is elérhető. Körnek nevezzük, ha egy út ugyanabban a csomópontban végződik, mint amelyből kiindult.

2.3 Keresés az állapottér-gráfban

Általában a keresés kezdetekor nem áll rendelkezésünkre a teljes állapottér gráf, de rendelkezésünkre állnak a szabályok, melyek segítségével felépíthetjük azt. A gráf kiinduló-csomópontjának tekintjük azt a csomópontot, amely a kezdőállapotot tartalmazza. A keresést tekinthetjük az állapottér-gráf fává alakításának.

A fa egy speciális gráf, amely nem tartalmaz hurkot és kört, valamint csak irányított éleket tartalmaz. Ha létezik *A*-ból *B*-be mutató irányított él, akkor *A*-t *B* szülőjének, *B*-t *A* gyermekének nevezzük. Kitüntetett elem a fában a gyökérelem, mert a gyökérelemen kívül minden elemnek pontosan egy szülője van. A fa bármely elemének akárhány gyermeke lehet. Ha egy elemnek nincs egyetlen gyermeke sem, azt levélelemnek hívjuk.

A gráf fává alakításakor problémát a körök és a hurkok jelentenek, mert ilyenkor egy elem többször is előfordulhat a fában, végtelen fát eredményezve. A keresőnek tehát érdemes körfigyeléssel rendelkeznie, és elmetszenie a gráfot az átalakításnál, amennyiben szükséges. Ezzel részletesebben a későbbiekben külön fogok foglalkozni (l. 3.3.1.). Egy elem mélységén értjük a gyökércsomópontból az elembe vezető út hosszát. Több elem közül a legsekélyebben fekvő az, amelyik a legkisebb mélységű.

A gráf kiinduló-csomópontjának a fában a gyökérelem felel meg. Egy állapottér-gráfbeli csomópont szomszédai a fában a csomópont gyermekeiként fognak megjelenni. A keresés során épülő fa a keresés kezdetekor csak a gyökérelemet tartalmazza. A többi elemet úgy kapjuk, hogy kiválasztunk egy még ki nem terjesztett csomópontot, és kiterjesztjük. A kiterjesztés az a folyamat, amikor a fába beillesztjük a csomópont szomszédait. A kiterjesztés előtt ellenőriznünk kell, hogy a kiválasztott csomópont célállapot-e. Ha igen, a keresés sikeresen fejeződik be. Ekkor a megoldás a gyökérelemtől a csomópontba vezető út lesz, amelyet fordított irányban kapunk, ha sorra vesszük a szülőket a gyökérelemig. A keresés sikertelenül áll le, ha elfogytak a kiterjeszthető csomópontok. Azt, hogy melyik csomópontot választjuk kiterjesztésre, a keresési stratégia határozza meg. [2]

3. Általános megoldáskeresők

Az általános megoldáskeresők egy paraméterül kapott állapottér-reprezentációhoz keresik a megoldást. A keresési stratégia alapján több különböző keresőt különböztetünk meg. Ezek mindegyike értékelhető teljesség, optimalitás, időigény és tárigény szerint.

- Teljesnek mondunk egy keresőt, ha minden esetben megtalálja a megoldást, amennyiben létezik.
- Optimális egy kereső, ha több különböző megoldás esetén az optimálisat találja meg (ez lehet a legkevesebb csomópontot tartalmazó megoldás, esetleg a legkisebb költségű megoldás)
- A kereső időigénye azt határozza meg, hogy mennyi idő alatt fejezi be a kereső a működését (akár talált megoldást, akár nem)
- A kereső tárigénye (memóriaigény) azt írja le, hogy a keresőnek mennyi memóriára van szüksége a kereséshez.

A gráf elágazási tényezője azt mutatja meg, hogy a gráfban egy csomópontból átlagosan hány él indul. Az idő- és tárigény becsléséhez a továbbiakban egy olyan gráfot tekintünk, amelynek minden csomópontjából *b* db él indul ki, így lesz ennek a gráfnak az elágazási tényezője *b*. A keresések időigénye n_i db kiterjesztett csomópont esetén $n_i \cdot t$ lesz, ahol *t* egy csomópont kiterjesztéséhez szükséges idő. A keresés tárigénye $n_t \cdot s$ ahol n_t az egy időben tárolt csomópontok maximális száma, és *s* egy csomópont tárolásához szükséges memória mérete. A továbbiakban az egyszerűség kedvéért *t*-t és *s*-et egységnyi értéknek tekintjük. [2]

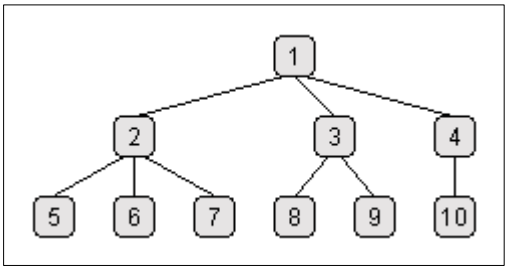
3.1 Nem informált keresők

Megkülönböztetünk informált és nem informált keresőket. Az informált keresők a kiterjesztendő csomópont kiválasztásához használnak valamilyen „tudást”, amelyet explicit módon kell megadnunk. A nem informált keresők ezzel szemben szisztematikusan bővítik a keresőfát. [2]

3.1.1 Szélességi kereső

A szélességi kereső kiterjesztésre azt a még ki nem terjesztett csomópontot fogja választani, amely a legsekélyebben fekszik. Amennyiben több ilyen csomópont is létezik, véletlenszerűen választ egyet. Vagyis első lépésként kiterjeszti a startcsúcsot, azután annak

összes gyermekét, majd azok gyermekeit, így haladva mindig egy szinttel mélyebbre a fában, míg megoldást nem talál, vagy el nem fogytak a csomópontok.



2. ábra: A csomópontok kiterjesztési sorrendje a szélességi keresés során

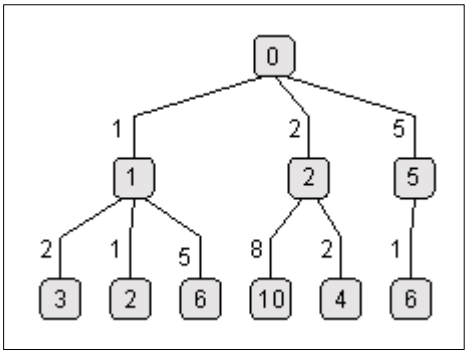
A szélességi kereső teljes, és optimális, mert amennyiben létezik megoldás, megtalálja, és több megoldás esetén azt a megoldást kapjuk, amelybe a legkevesebb operátor-alkalmazással juthatunk. Az optimalitás egyszerűen belátható. Legyen N_c a megtalált célállapotot tartalmazó csomópont. Ha a megoldás nem lenne optimális, az azt jelentené, hogy létezik olyan N csomópont, amely célállapotot tartalmaz, és mélysége kisebb, mint az N_c-é, mivel egy csomópont mélysége megegyezik a kiinduló állapotból a csomópont által reprezentált állapotig alkalmazott operátorok számával. Ebből viszont az következne, hogy létezik olyan csomópont, amely sekélyebben fekszik N_c-nél, és még nem lett kiterjesztve. Mivel az algoritmus mindig a legsekélyebben fekvő csomópontot terjeszti ki, ilyen eset nem fordulhat elő.

Az algoritmus a 0. szinten a egyetlen csomópontot, a gyökérelemet terjeszti ki, amely b db gyermeket generál. Az 1. szinten így b db csomópontot terjeszt ki, b^2 db csomópontot generálva, amit a 2. szinten fog kiterjeszteni. Vagyis az algoritmus az i. szinten b^i db csomópontot terjeszt ki. Ezeknek összege adja a kiterjesztett csomópontok számát. A keresés időigénye tehát $\sum_{i=0}^{d} b^i$ ahol d a legsekélyebb megoldás mélysége. A következőkben az egyszerűség kedvéért a komplexitást tüntetem fel, ami jelen esetben $O(b^d)$. Ez a legrosszabb esetben vett időigény, amikor azt feltételezzük, hogy a megoldás szintjén lévő csomópontok közül a megoldást tartalmazót utolsóként választja az algoritmus. Szerencsés esetben ezt elsőként választjuk kiterjesztésre, így b^d -vel csökkentve a kereséshez szükséges időt $O(b^{d-1})$-re.

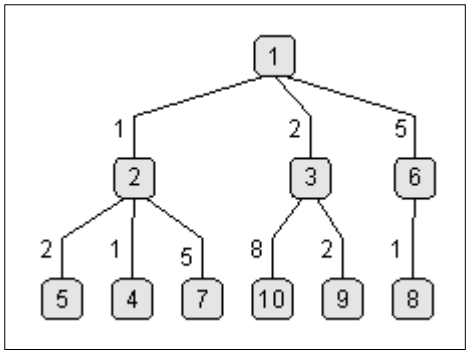
Az algoritmus tárigénye szintén $O(b^d)$, mert az összes kiterjesztett csomópontot tárolnunk kell, ahhoz, hogy vissza tudjuk állítani a megoldást. [2]

3.1.2 Egyenletes költségű kereső

A szélességi kereső „hibája”, hogy nem veszi figyelembe az operátorok alkalmazási költségét, így bár a legrövidebb megoldást szolgáltatja, nem biztos, hogy az egyben a legolcsóbb is. Az egyenletes költségű kereső a szélességihez hasonlóan működik, de ez a még ki nem terjesztett csúcsok közül azt fogja kiválasztani kiterjesztésre, amelyhez a gyökérelemből vezető út költsége a legkisebb. Az egyenletes költségű tekinthető a szélességi kereső általánosításának. Vagyis a szélességi kereső egy speciális egyenletes költségű kereső, amelyben minden operátor alkalmazási költsége egységes.



3. ábra: Az élek és csomópontok költsége



4. ábra: A csomópontok kiterjesztési sorrendje az egyenletes költségű keresés során

Ezzel a keresővel olyan szempontból kaphatunk optimális megoldást, hogy a legkisebb útköltségű megoldást kapjuk eredményként. Azonban ez csak akkor van így, ha igaz a fára, hogy minden csomópont útköltsége nagyobb vagy egyenlő a szülőjének az útköltségénél, vagyis nincs negatív költségű operátor. Ha feltételezzük ezt, akkor láthatjuk, hogy ha kiterjesztünk egy csomópontot, biztosan nem találunk később nála olcsóbb megoldást. Ha viszont van negatív költségű operátor, akkor semmilyen szempontból nem garantált az optimális megoldás. A teljesség garantált, ha nincs negatív költségű operátor. Ha mégis van, akkor a teljességet csak véges gráf esetén tudjuk garantálni.

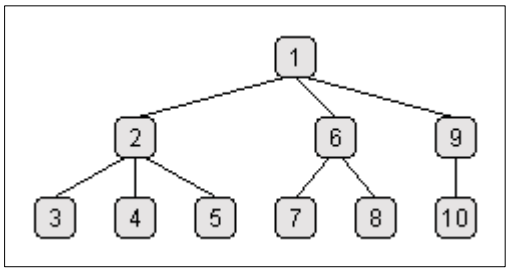
A kereső idő- és tárigénye nehezen becsülhető *b*-vel és *d*-vel, mert az útköltség teljesen eltérő lehet a mélységtől. Az viszont elmondható, hogy ez akár sokkal több is lehet $O(b^d)$-nél, mert az algoritmus viselkedéséből adódóan előbb járja be a kis lépésekből álló nagy fákat, mint a nagy lépéseket tartalmazó utakat. [2]

3.1.3 Mélységi kereső

A mélységi kereső azt a még ki nem terjesztett csomópontot fogja választani kiterjesztésre, amelyik a legmélyebben fekszik. Több azonos mélységű csomópontból pedig ez is

véletlenszerűen választ. A kereső így egy kiválasztott utat követ addig, amíg azt folytatni tudja, majd visszalép, és egy másig ágon halad tovább. Közben természetesen megállhat, ha célállapotot talált, vagy elfogytak a csomópontok.

A mélységi kereső hátrányos tulajdonsága, hogy elkezdhet követni egy utat, amely rossz irányba halad, így esetleg csak nagyon sokára ér el célállapotot. Valamint az is előfordulhat, hogy ez az út a végtelen hosszú, ilyenkor nem is találja meg a megoldást.



5. ábra: A csomópontok kiterjesztési sorrendje a mélységi keresés során

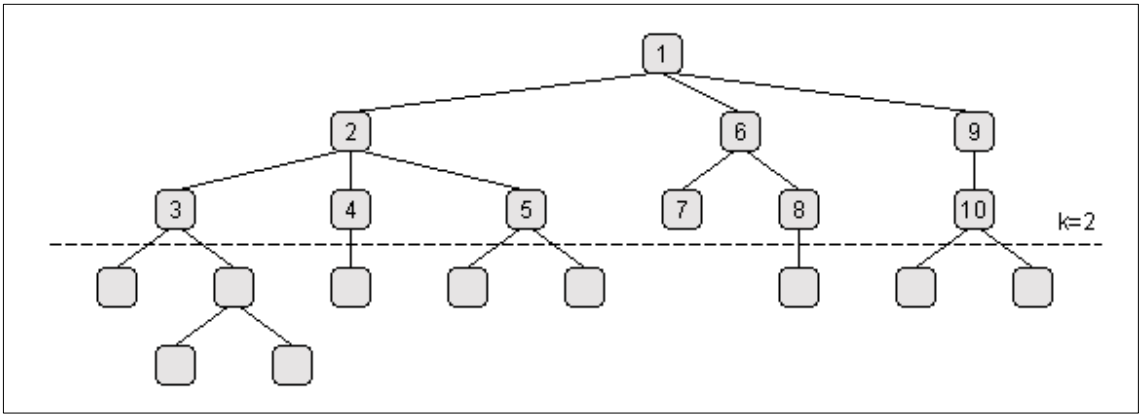
Tehát a kereső csak akkor teljes, ha a keresési fa véges. Ezen kívül semmi garanciát nem ad optimális megoldás megtalálására.

Időigénye legrosszabb esetben $O(b^m)$, ahol *m* a fa mélysége, legjobb esetben *d*, ami egy elég nagy intervallum lehet. A mélységi kereső akkor hatásos, ha például egy olyan fában keresünk, amelyben minden levélelem célállapot, és viszonylag mélyen vannak a fában, mert ilyenkor a mélységi kereső legfeljebb *m* db csomópontot terjeszt ki, hiszen a kereső minden kiterjesztésnél egyre mélyebb csomópontokat terjeszt ki, amíg el nem ér egy levélelemet.

A keresés során itt elég nyilván tartani az aktuális utat a lehetséges elágazási pontokkal, így a mélységi keresőnek a szélességihez képest általában sokkal kevesebb memóriára van szüksége. A tárigény a megoldás mélysége helyett a fa legmélyebb csomópontjának mélységétől függ. Így a kereső memóriaigénye $b \cdot m$, ahol *m* a fa legnagyobb mélysége. [2]

3.1.4 Mélységkorlátozott kereső

A mélységkorlátozott kereső a mélységi kereső egy változata. A még ki nem terjesztett csomópontok közül ugyanúgy választunk, mint a mélységi kereső esetén, viszont az algoritmus nem veszi figyelembe adott *k* mélységnél nagyobb mélységgel rendelkező csúcsokat. Ezáltal kiküszöböli a végtelen fák problémáját.



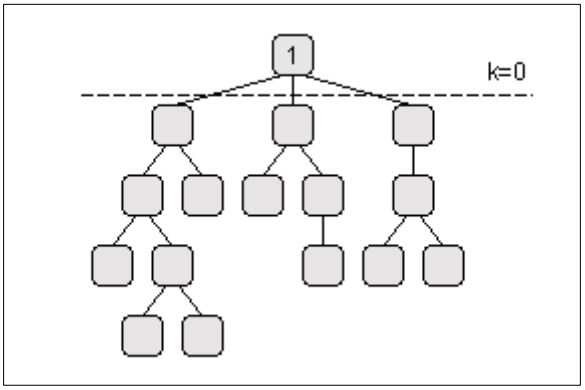
6. ábra: A csomópontok kiterjesztési sorrendje a mélységkorlátozott keresés során k=2 korláttal

A kereső azonban mindemellett is csak akkor lesz teljes, ha $k \geq d$, vagyis ha a mélységkorlát legalább akkora, mint a legsekélyebben fekvő megoldás mélysége. Amennyiben $k = d$, akkor a kereső optimális, egyéb esetben az optimalitás továbbra sem garantált.

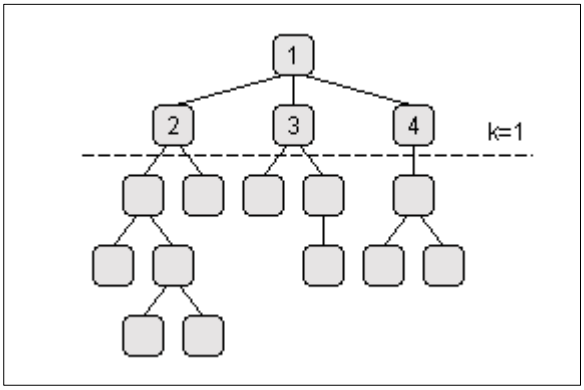
Az idő- és tárigénye hasonló a mélységi keresőhöz. Időigénye $O(b^k)$, tárigénye $b \cdot k$, ahol k a mélységkorlát. [2]

3.1.5 Iteratívan mélyülő kereső

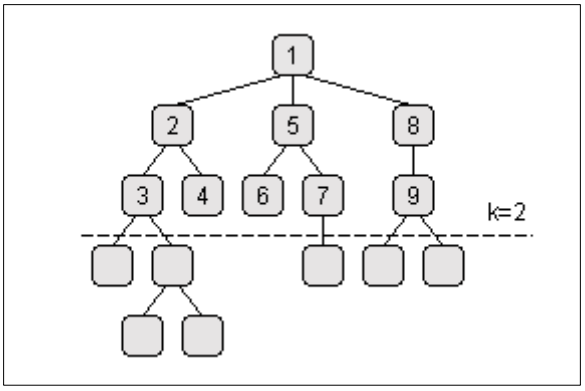
Azt láthattuk, hogy a mélységkorlátozott keresésnél teljesség és optimalitás szempontjából is lényeges egy jó mélységkorlát megválasztása. Az iteratívan mélyülő keresés erre ad megoldást azzal, hogy többször futtat egymás után mélységkorlátozott keresést, de folyamatosan növekedő mélységkorláttal, amíg megoldást nem talál. Vagyis kezdetben a mélységkorlát 0, majd 1, 2 és így tovább. Ahhoz, hogy azt is el tudjuk dönteni, hogy van-e megoldás, ellenőriznünk kell, hogy a mélységkorlát elérésekor van-e még ki nem terjesztett csomópont. Ha nincs, akkor bejártuk az egész fát, vagyis nem létezik megoldás.



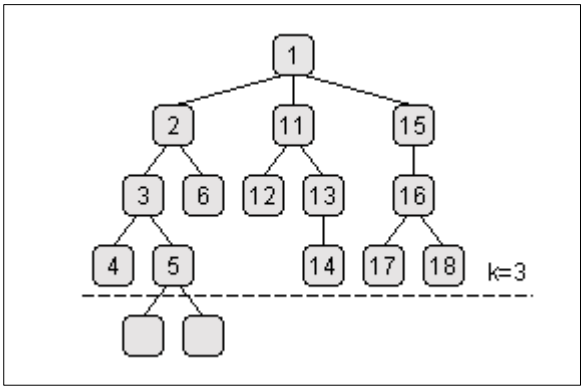
7. ábra: Csomópontok kiterjesztési sorrendje az iteratívan mélyülő keresés első iterációjában



8. ábra: Csomópontok kiterjesztési sorrendje az iteratívan mélyülő keresés második iterációjában



9. ábra: Csomópontok kiterjesztési sorrendje az iteratívan mélyülő keresés harmadik iterációjában



10. ábra: Csomópontok kiterjesztési sorrendje az iteratívan mélyülő keresés negyedik iterációjában

Ez a kereső tehát ebből adódóan teljes lesz, valamint optimalitását tekintve a szélességi keresőhöz hasonlít, vagyis több megoldás esetén azt a megoldást kapjuk, amelyik a legsekélyebben fekszik.

Ha az algoritmus időigényét figyeljük meg, érdekes dolgot tapasztalunk. Azt várnánk, hogy az iteratívan mélyülő kereső időigénye a szélességi keresőhöz képest lényegesen nagyobb, hiszen a megoldás feletti szinteken lévő csomópontokat újra és újra kiterjeszti. Szám szerint a k korlátig a k mélységű csomópontokat 1-szer, a $k-1$ mélységűeket 2-szer, és így tovább, míg végül 0 mélységű elemeket – ami egyedül a gyökérelem – $k+1$-szer. A két kereső időigényét tehát a következőképpen írhatjuk fel az átláthatóság kedvéért $\sum$ használata nélkül:

$$(k+1)b^0 + kb^1 + (k-1)b^2 + ... + 1b^k,$$

illetve

$$b^0 + b^1 + b^2 + ... + b^k.$$

Ha például $b = 10$ és $k = 5$, akkor az iteratívan mélyülő keresés időigénye 123 456, a szélességi keresőé pedig 111 111. Ez láthatóan nem nagy különbség. Ha meg szeretnénk érteni, miért is van ez így, vizsgáljuk meg a fát. Az 0. szinten 1 elem található, az 1. szinten 10, a 2. szinten 100, és így tovább 1 000, 10 000, 100 000. Megfigyelhetjük, hogy ha egy fa elágazási tényezője elég nagy, akkor lényegesen több elem van egy adott szinten, mint az azt megelőző összes szinten együtt. Minél nagyobb az elágazási tényezője egy fának, annál kevésbé lesz veszteséges az iteratívan mélyülő kereső. Az azonban nyilvánvaló, hogy sohasem fog kevesebb csomópontot kiterjeszteni.

Amiért mégis megéri a szélességi keresővel szemben az iteratívan mélyülő keresést választani, az a tárigénye, hiszen ez a tulajdonsága a mélységi kereső tárigényére hasonlít, vagyis $b \cdot k$,

hiszen ennek a keresőnek is elég az aktuális csomópontig vezető utat tárolni, ami viszont minden szinten *k*-val egyenlő, *k* pedig létező megoldás esetén legfeljebb *d*, illetve *m*, ha nincs megoldás. [2]

3.2 Informált keresők

Sok olyan probléma létezik, amelynek megoldásáról hasznos, esetleg triviális igazságokat állapíthatunk meg, akár első ránézésre. Például tudjuk, hogy egy sakktáblán a királlyal a tábla egyik sarkából a másikba 7 lépésnél kevesebb lépésben nem juthatunk el. Ezt bátran mondhatjuk, akkor is, ha nem tudjuk milyen egyéb bábuk vannak a táblán. Ilyen igazságok felismerése nekünk, emberek számára, ha nem is mindig egyszerű, de általában nem lehetetlen feladat. Azonban egy számítógép nem gondolkodik, csak számol, így ha szeretnénk, hogy a számítógép használja a tudásunkat, le kell írnunk a valamilyen feldolgozható formában, illetve meg kell adnunk, hogy azt hogyan kell fel használni. Az informált keresők valósítják meg a tudás felhasználásának a módját. Ez a tudás fogja befolyásolni a keresés „irányát”.

A tudást egy $h: A \rightarrow \mathbb{R}$ függvénnyel írjuk le, ahol *A* az állapotok halmaza. Ezt a függvényt nevezzük heurisztikának. A heurisztika értéke egy becslés lesz arra vonatkozóan, hogy adott állapotból milyen költséggel jutunk el egy célállapotba. A heurisztikát figyelembe véve a keresés alatt, a kiterjesztett csomópontok száma lényegesen lecsökkenhet, és ezzel együtt a keresés idő- és tárigénye is. A heurisztika értéke a célállapotban általában 0, és minél távolabbinak becsüli a heurisztika a célállapotot, az értéke annál nagyobb lesz. Ez nincs mindig így, de ez alapján megkülönböztethetünk heurisztikákat.

Elfogadhatónak (alulbecslő) mondunk egy heurisztikát, ha értéke minden állapotban kisebb, mint a vizsgált állapotból a célba jutás költsége.

Egy heurisztika monoton, ha a heurisztika értéke minden állapotban legfeljebb annyival kevesebb a szülőállapot heurisztikájától (ha van ilyen), mint a szülőállapotból a vizsgált állapotba juttató operátor alkalmazásának költsége. [2]

3.2.1 Legjobbat először kereső

Ez a kereső a kiterjesztésre kiválasztásnál a csomópont heurisztikája alapján dönt. Mindig azt a csomópontot fogja kiterjesztésre kiválasztani, amelyiknek legkisebb a heurisztikája. Tehát mindig azt az utat folytatja, amelyikben leginkább várható a megoldás. Viselkedése hasonló a mélységi keresőhöz, mert egy utat addig követ, amíg csak tud, majd ha nem talált megoldást, visszalép.

A kereső ugyancsak a mélységi keresőhöz hasonlóan csak akkor teljes, ha a fa véges, és optimalitást ez sem garantál.

Időigénye legrosszabb esetben megegyezik a mélységi keresőével, vagyis $O(b^m)$, tárigénye pedig szintén ugyanennyi, mert a kereséshez használt összes csomópontot a memóriában tartja. [2]

3.2.2 A*-kereső

Ez a kereső egy csomópont kiterjesztésnél figyelembe veszi az adott állapotba jutás addigi költségét, amit $g(n)$-el jelölünk, és az állapotból a célba jutás becsült költségét is, amit $h(n)$-el jelölünk, vagyis a kiértékelő függvénye a csomóponton átmenő teljes út becsült költsége alapján dönt, tehát

$$f(n) = g(n) + h(n)$$

A kereső azt a csomópontot választja kiterjesztésre, amelynél az $f(n)$ értéke a legkisebb.

Az A*-kereső teljes, de önmagában nem garantálja, hogy optimális költségű utat talál. Azonban alulbecslő heurisztika használatával az optimalitás is garantált.

A kereső idő- és tárigénye megegyezik, mert minden kiterjesztett csúcsot tárolnia kell. Legrosszabb esetben ez $O(b^m)$, de megfelelő heurisztika használatával ez jelentősen lecsökkenhet.

Ebből is látszik hogy a heurisztika megválasztása kulcsfontosságú. Megfigyelhető, hogy ha a heurisztika értéke túl kicsi, a kereső viselkedése az egyenletes költségű keresésre fog hasonlítani. Ha a heurisztika értéke felülbecsüli a tényleges költséget, az optimális megoldás nem garantált, de a kereséshez szükséges idő lecsökkenhet. Ha a heurisztika sokkal nagyobb a költségnél, a kereső elveszti hatékonyságát. [2]

3.2.3 Hierarchikus A*-kereső

Ez a kereső nem tartozik a legismertebb keresők közé, bemutatását viszont a dolgozat későbbi részeiben történő utalás miatt fontosnak tartom. Az A*-kereső fontos tulajdonsága, hogy minél pontosabb a heurisztika, annál kevesebb csomópontot terjeszt ki. A hierarchikus A*-kereső az A*-keresőnek egy olyan változata, amely heurisztikaként egy olyan A*-keresőt használ, mely egy olyan állapottérben keres, amely az eredeti állapottér egy egyszerűbb, absztraktabb módosított változata. A módosított állapottér létrehozása történhet egyes megszorítások elhagyásával, vagy csomópontok összevonásával is, így kapva egy elég pontos heurisztikát. A kereső így minden kiterjesztett állapothoz heurisztikaként az absztrakt

állapottérben vett legolcsóbb költséget veszi. Ez ugyanakkor azzal jár, hogy az eredeti állapottér minden kiterjesztett csomópontjához egy keresés fut le az absztrakt állapottérben. Így tehát nyilvánvaló veszélye a kereső használatának, hogy a pontosabb heurisztika kiszámításához szükséges idő túlnőhet a használatából származó nyereségen. Ezen úgy tudunk javítani, ha valamilyen módon felhasználjuk a már kiszámolt heurisztika értékeket. [3]

3.3 Keresők hatékonyságát növelő módszerek

Léteznek olyan keresési technikák, amelyeket szinte minden eddig említett kereső használhat, ezzel növelve a keresés sikerességének valószínűségét, esetleg csökkentve a kereséshez szükséges időt, vagy éppen egy problémára nyújtva megoldást. [2]

3.3.1 Körfigyelés

Az előzőekben bemutatott algoritmusok nagy részére jellemző, hogy végtelen keresési fa esetén nem biztosított a teljesség. Ha a fa végtelensége abból adódik, hogy a gráf köröket tartalmaz, akkor a körfigyelés megoldást nyújt a problémára. Emellett csökkenthetjük a keresés időigényét is, ha nem terjesztjük ki újra azokat a csomópontokat, amelyeket egyszer már kiterjesztettünk. A körfigyelés megvalósítására különböző megoldások léteznek. Az egyik, hogy egy csomópont kiterjesztésekor eltávolítjuk a gyermekei közül azokat, amelyek állapota megegyezik a csomópont szülőcsomópontjában tárolt állapottal. Másik, valamivel hatékonyabb megoldás, hogy a gyermekek közül azokat is eltávolítjuk, amelyek a csomópont bármelyik ősével megegyeznek. Végül a harmadik, legteljesebb megoldás, hogy a kiterjesztett csomópont gyermekei közül akkor távolítunk el csomópontot, ha az egész eddigi keresés során bármikor is előfordult. Ehhez viszont vizsgálnunk kell az összes eddigi csomópontot.
Látható tehát, hogy a körök megszüntetése az idő- és tárigény növekedésével járhat, azonban sok kört tartalmazó gráfok esetén mindenképpen megéri alkalmazni. [2]

3.3.2 Kétirányú keresés

A kétirányú keresés során nem csak a kiinduló-csomópontból indulunk a cél felé, hanem ugyanekkor a célból is indul egy kereső a start felé. Ez akkor gazdaságos, ha az elágazási tényező mindkét irányban elég nagy. Ha az elágazási tényezőt mindkét irányban b-nek vesszük, a keresés idő- és tárigénye $O(2b^{d/2})$-re csökkenhet, vagyis a komplexitása $O(b^{d/2})$ lesz, mivel mindkét keresőnek csak *d/2* mélységig kell keresni a fában. Például a szélességi kereső használatával, ha $b = 10$ és $d = 8$, akkor a kereső alap esetben a $\sum_{i=0}^{d} b^i$ képlet alapján

111 111 111 csomópontot terjeszt ki, míg kétirányú keresés használatával $\sum_{i=0}^{d/2} b^i$ -t, vagyis 22 222 darabot.

A módszer hátránya, hogy nem minden esetben használható. Ugyanis a visszafelé haladó keresésben a kiterjesztéskor ahelyett, hogy azokat az állapotokat generálnánk, amelyek az adott állapotból közvetlenül elérhetőek, azokat kell generálnunk, amelyekből az adott állapot közvetlenül elérhető. Ez nem mindig egyértelmű. Emellett kérdéses, hogy mit teszünk, ha több célállapot is létezik. Illetve nem használhatunk kétirányú keresést, ha feladatunk a célállapot előállítása, így nem ismerjük azt. [2]

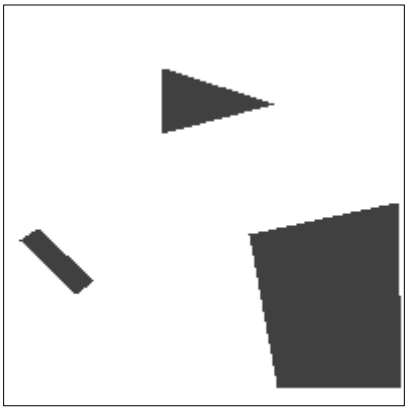
4. Az útvonalkeresés

A megoldáskeresők használatának egyik leglátványosabb területe az útvonalkeresésben nyilvánul meg. Az útvonalkeresés segítségével megtudhatjuk, hogyan jutunk el a világ egy pontjából egy másik pontjába. Ez lehet akár a valós világ is, vagy egy egyszerűbb, mesterségesen alkotott világ. Ilyen, egyszerű világokkal találkozhatunk a valós idejű stratégiai játékokban is. Ezekben a játékokban az egységek egy célpozíciót szeretnének elérni. Az egységek számára a céljuk felé vezető útvonalat az útvonalkereső szolgáltatja. A térképnek különböző minőségű területei lehetnek (pl. föld, fű, hegy, víz, stb.). Az útvonalkereső elsősorban azt veszi figyelembe, hogy az adott terület járható-e. Emellett az is előfordulhat, hogy egyes területeken gyorsabban, vagy lassabban tudnak haladni az egységek, és a kereső akár ezt is figyelembe veheti.

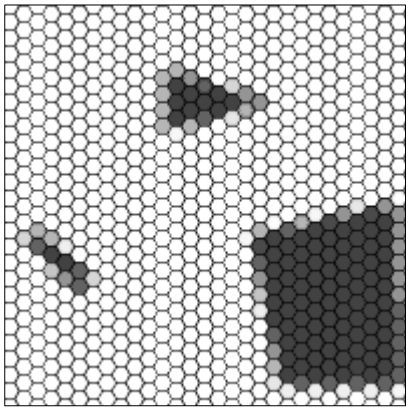
A kereső által használt állapottér tehát a térképen alapul. Ebben a fejezetben szeretném bemutatni, hogy milyen módszerrel alakíthatjuk át a játék térképét állapottér-gráffá, szeretnék ismertetni néhányat az útvonalkeresésre használt technikák közül, végül ismertetném az útvonalkeresésben használt leggyakoribb heurisztikákat.

4.1 A térkép gráffá alakítása

A játék térképe lehet diszkrét, vagy folytonos. Diszkrét a térkép akkor, ha az egységek és akadályok, csak a térkép jól meghatározott pontjain állhatnak, míg folytonos, ha nincs ilyen kikötésünk.



11. ábra: Folytonos térkép



12. ábra: Diszkrét térkép

4.1.1 Diszkrét térkép

Egyszerű a dolgunk, ha diszkrét térképpel dolgozunk. Ekkor a térkép „jól meghatározott pontjai” alkotnak egy-egy csomópontot, amelyeket azonosíthatunk a térkép egy pozíciójával. Így általában egy négyzetrácsos térképet kapunk, de előfordulhat háromszög vagy hatszög alapú rács is.

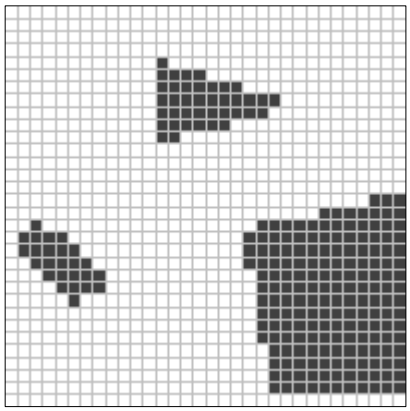
4.1.2 Folytonos térkép

A folytonos térkép gráffá alakítása már bonyolultabb, mert ilyenkor egy-egy területhez kell meghatároznunk csomópontot. Erre több különböző megoldás létezik.

4.1.2.1 Egyenletes rács

A legegyszerűbb megoldás, hogy egy rácsot fektetünk az eredeti térképre, és úgy kezeljük, mint egy diszkrét térképet, tehát a rácspontok által lefedett terület lesz egy csomópont. Ehhez minden ilyen rácspontot homogénné kell alakítanunk, mert egy rácspont nem jelölhet eltérő minőségű területeket (vagyis nem lehet egy pont egyszerre járható és nem járható). Ilyenkor vehetjük például a legnagyobb arányban tartalmazott területet, vagy a tartalmazott területek közül a legnehezebben járhatót. A módszer egyes változataiban a diszkrét megoldáshoz hasonlóan használhatunk négyzetek, háromszögeket, vagy hatszögeket is.

Ennek a módszernek a pontosságát nagyban befolyásolja, hogy mekkorára választjuk a rácspontokat. A kisebb rácspontok pontosabb térkép reprezentációt adnak, de ezáltal nagyobb lesz az állapottér-gráf.

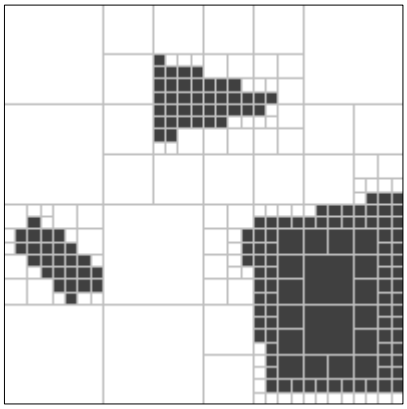


13. ábra: Folytonos térkép átalakítása négyzetráccsal

4.1.2.2 Kvadratikus fa[1]

Ez a módszer négyzetekre osztja fel a térképet, viszont itt a négyzetek oldalmérete változhat. A módszer megkülönböztet „járható” és „nem járható” területeket a térképen. Első lépésben megkeressük a legkisebb négyzetet, amelybe a térkép belefér. Amennyiben ez a négyzet homogén, vagyis tartalmaz „járható” és „nem járható” területet is, akkor ezt felosztjuk négy egyenlő négyzetre, és ezekre vizsgáljuk ugyanezt egyenként. Csak akkor osztjuk a négyzeteket kisebb részekre, ha szükséges, vagyis ha a négyzet homogén. Az eljárást addig folytatjuk, amíg el nem értünk egy megfelelő pontosságot. Ez a módszer folytonos térkép gráffá alakítása mellett egyszerűen alkalmazható négyzet alapú diszkrét térkép további absztrakciójaként is.

A módszer hátránya, hogy nehéz megfelelő heurisztikát használni hozzá, amely elfogadható, és elég közel becsli a tényleges távolságot. Ez abból adódik, hogy a csomópontok területe nagymértékben eltérhet. Cserébe sokkal kevesebb csomópontunk lesz, mint az egyenletes rács használatakor. Emellett ez a módszer térben is jól használható, ekkor Oktális fának[2] nevezzük.

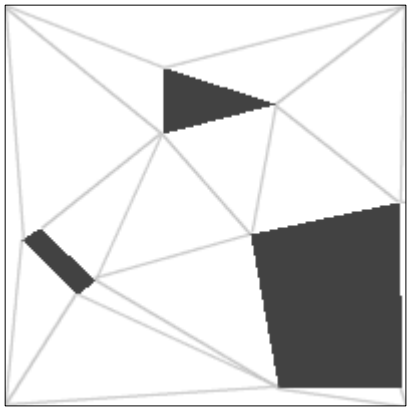


14. ábra: Folytonos térkép átalakítása kvadratikus fa használatával

4.1.2.3 Konvex poligonok

Ez a módszer a térképet konvex poligonokra bontja fel, amik egy csomópontot alkotnak. A módszer leggyakoribb változata háromszögeket használ. Ez a módszer is viszonylag kevés csomópontot generál. Hátránya, hogy a térképen lévő akadályokat is poligonként kell ismernünk a használatához.

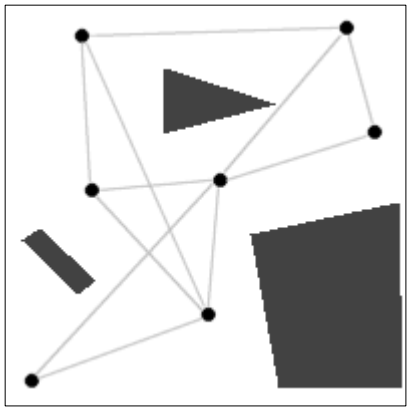
[1] Quadtree
[2] Octree



15. ábra: Folytonos térkép egy lehetséges átalakítása konvex poligonok használatával

4.1.2.4 Navigációs pontok

A módszer lényege, hogy figyelmen kívül hagyjuk a térkép egyes részeit, és csak egy-egy meghatározott pontot alakítunk csomóponttá, és a keresés csak ezeken a csomópontokon folyik. Ezek lesznek a navigációs pontok Ennél a módszernél azonban már nem egyértelmű, hogy mely csomópontokból melyeket érhetünk el. A legáltalánosabb megoldás, hogy egy navigációs pontból akkor érhető el egy másik, ha a két pont közötti egyenes szakasz nem ütközik akadályba. A módszer alkalmazásához az is hozzátartozik, hogy minden kereséskor a kiinduló- és célpozíciót is fel kell vennünk a gráfba, különben az egységek csak az előre meghatározott pontokat ismernék.



16. ábra: Folytonos térkép egy lehetséges átalakítása navigációs pontok segítségével

4.1.3 További absztrakciós technikák

Az előző eljárások segítségével egy csomópontokból és élekből álló gráfot kapunk eredményül, ahol a csomópontok a rács cellái, az élek pedig a köztük lévő szomszédsági viszonyok. Mint azt már korábban leírtam, a keresés hatékonyságának szempontjából érdemes lehet csökkenteni a gráf csomópontjainak számát. Ezt úgy is megtehetjük, hogy több

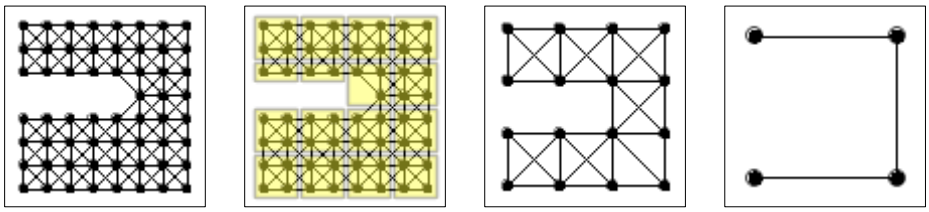
csomópontot bizonyos szempontok alapján egyesítünk, és az újonnan létrejött csomópontokból létrehozunk egy új absztrakciós réteget. Az absztrakciót ezután elvégezhetjük minden egyes újonnan előállt gráfra, így egy szint-hierarchiát képezve, ahol a 0. szint az eredeti gráf. A továbbiakban feltételezzük, hogy a gráf élei irányítatlanok. Legyen a leképezés függvénye ϕ, amely egy adott *G* gráfbeli csomópontot *G*' gráfbeli csomóponttá képez. Legyenek n_1' és n_2' csomópontok, *G*'-ben. Az absztrakt gráfban – vagyis *G*'-ben – pontosan akkor van él n_1' és n_2' között, ha létezik olyan n_1 és n_2 csomópont *G*-ben, hogy $\phi(n_1) = n_1'$ és $\phi(n_2) = n_2'$, valamint létezik él n_1 és n_2 között.

Az absztrakció előnye a csomópontok számának csökkenése mellett, hogy a szintek közötti hierarchikus kapcsolat miatt gyorsan meghatározható, hogy létezik-e út két pont között. Hátránya viszont a heurisztika inkonzisztenciája az absztrakt, és az eredeti állapottér-gráf között. Amikor az eredeti gráfban végezzük a keresést, akkor a heurisztika a csomópontok pozícióját használja fel. Az absztrakt állapottérben viszont egy csomópontnak több állapot felel meg az eredeti szinten. Itt használhatjuk heurisztikaként a tartalmazott csomópontok átlagos pozícióját. Viszont ez a heurisztika, míg alulbecslő lesz az absztrakt gráfban, nem biztos, hogy az eredeti gráfban is az. Így mint azt korábban kifejtettem, az A*-kereső nem garantálja az optimális megoldást az eredeti gráfra nézve.

Aszerint, hogy mi alapján választjuk ki az egyesítendő csomópontokat, több módszert különböztetünk meg. [4]

4.1.3.1 Klikk-absztrakció[3]

Ez az absztrakció klikkeket (csomópont-együttest) keres a gráfban, és ezek elemeit képezi le egyetlen új csomóponttá. A klikkben lévő csomópontokra jellemző, hogy egy csomópontból egyetlen élen keresztül el lehet jutni a klikk bármelyik másik csomópontjába. Egy olyan diszkrét, négyzetrácsos térképen, amelyen nyolc irányba léphetünk, a legnagyobb klikk négy elemből állhat. Ez az absztrakció nem támaszkodik a csomópontok térbeli pozíciójára, így nem csak útvonalproblémákra alkalmazható. [4]

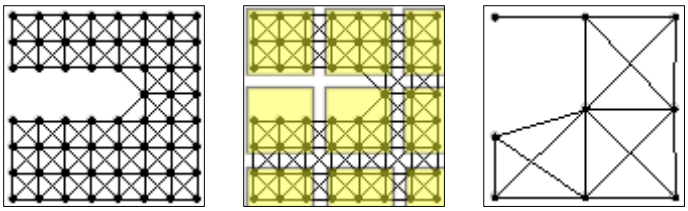


17. ábra: Klikk absztrakció

[3] Clique-abstraction

4.1.3.2 Szektor absztrakció[4]

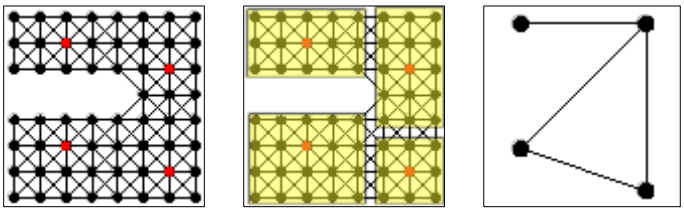
Ez a módszer jól meghatározott, egyenlő méretű szektorokra osztja fel a területet. Minden területen belül szélességi keresővel meghatározza a kapcsolódó csomópontokat, amelyek az új absztrakciós szinten egy csomóponttá válnak. A szektorok méretét k paraméter alapján határozzuk meg. Az *i.* absztrakciós szinten egy $k^i \times k^i$ méretű négyzet jelent egy szektort. Ez a módszer felhasználja a csomópontok térbeli helyzetét is. [4]



18. ábra: Szektor absztrakció k=3 esetén

4.1.3.3 Sugár absztrakció[5]

Ez az r paraméterű algoritmus először kiválaszt egy még nem absztrahált csomópontot, majd egyesíti azokkal a csomópontokkal, amelyek legfeljebb r élnyi távolságra vannak tőle. A nem absztrahált csomópont kiválasztására nincs külön szabály, történhet véletlenszerűen vagy valamilyen szabály alapján. A sugár absztrakció egyik változatának tekinthető a csillag absztrakció[6], amelyben $r = 1$, és a nem absztrahált csomópontok közül azt választja ki, amely csomópont a legnagyobb fokszámmal rendelkezik – vagyis a legtöbb szomszédja van. A sugár absztrakció független a csomópontok térbeli elhelyezkedésétől. Megfelelő csomópontkiválasztási algoritmussal az $r = 1$ paraméterű sugár absztrakció megegyezik a $k = 3$ paraméterű szektor absztrakcióval. [4]



19. ábra: Egy lehetséges sugár absztrakció r=2 esetén

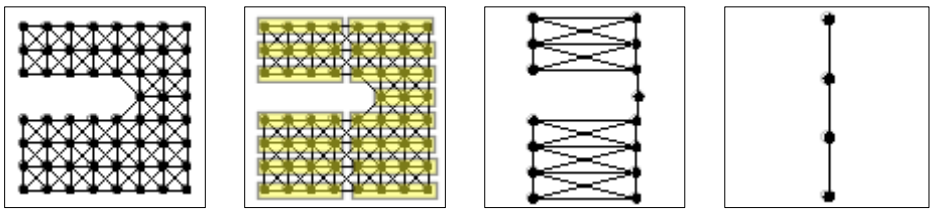
[4] Sector abstraction
[5] Radius abstraction
[6] Star abstraction

4.1.3.4 Sor absztrakció[7]

Az algoritmus a mélységi kereső mintájára egy még nem absztrahált csomópontból kiindulva keres *k* db nem absztrahált csomópontból álló sorozatot, és ezen sorozat elemeit egyesíti egy csomóponttá a következő szinten. A kiinduló-csomópont választása itt is történhet véletlenszerűen, vagy egy szabály alapján, például a csomópont térbeli pozíciójára alapozva. [4]

4.1.3.5 Csomópont-korlát absztrakció[8]

Ez az algoritmus az előző „szélességi-kereső változata", amely egy még nem absztrahált csomópontból szélességi keresőt futtat, amíg *k* db még nem absztrahált csomópontot nem talál, majd ezeket egyesíti. A kiinduló-csomópont kiválasztására itt is ugyanaz vonatkozik, mint az előző algoritmusnál. Amennyiben ugyanazt a csomópont-kiválasztó algoritmust használjuk, a csomópont-korlát absztrakció $k = 2$ paraméterrel megegyezik a sor absztrakcióval, valamint ha a *k* megegyezik a tőle legfeljebb *r* élnyi távolságra lévő csomópontok számával, akkor



20. ábra: Egy lehetséges sor absztrakció k=3 esetén

megegyezik a sugár absztrakcióval. [4]

4.2 Útvonalkeresésben használt algoritmusok

Az útvonal kereséséhez használhatunk olyan algoritmusokat, amelyek nem keresőkön alapulnak, hanem keresés nélkül csak a közvetlen környezetükre támaszkodva döntenek a következő lépésről. Ezek általában elindulnak a cél irányába, és ha akadályba ütköznek, akkor megpróbálják azt kikerülni. Ezen algoritmusok megoldásai viszont gyakran közel sem nevezhetőek optimálisnak, sőt a megoldást sem találják meg mindig.

Ehelyett megoldáskeresőket alkalmazunk. A különböző megoldáskeresők különböző mértékben szolgáltatnak optimális megoldást, illetve veszik figyelembe a környező egységeket. Ebben a részben be szeretném mutatni a David Silver által publikált ablakolt

[7] Line abstraction
[8] Node-limit abstraction

hierarchikus kooperatív A*-algoritmus [9] működését, amelyet a programomban implementáltam, illetve az ennek megértéséhez szükséges algoritmusokat.

4.2.1 A*-algoritmus az útvonalkeresésben

A legkézenfekvőbb megoldás, hogy A*-keresőt használunk a megoldás keresésére, hiszen optimális megoldást ad, és az eddig megismert általános keresők közül a leggyorsabb volt.

Egy valós idejű stratégiai játékban viszont, ahol több egység is mozoghat egyszerre, így a környezet dinamikusan változhat, önmagában nem bizonyul elég hatékonynak. A dinamikus változás miatt nem tudjuk, hogy egy már megtalált út nem vált-e időközben érvénytelenné, tehát minden lépés után újra kell terveznünk az utat. Ez a megoldás viszont túl nagy processzorteljesítményt kíván. A probléma megoldására jöttek létre az A*-algoritmus különböző változatai, amelyek többnyire az állapottér, az operátorok és a heurisztika megfelelő megválasztásában térnek el az eredetitől.

4.2.2 Helyi-javítású A*-algoritmus[10]

Ezt az algoritmust előszeretettel használták a korai stratégiai játékokban. Az algoritmus úgy működik, hogy először minden egység megkeresi A*-algoritmussal a legrövidebb utat a céljához, figyelmen kívül hagyva az összes többi egységet, kivéve a közvetlen szomszédait, majd elkezd haladni a saját útján. Ezután pedig minden lépés előtt ellenőrzi, hogy érvényes-e még a lépés, vagyis nem áll-e másik egység azon a pozíción. Ha tud lépni, akkor folytatja az útját, ha pedig nem, akkor az algoritmus újból lefuttatja a keresőt az aktuális pozíciója és a célja között, hogy elkerülje az ütközést.

A kereső gyorsan működik, hiszen nem fogja minden lépés után újratervezni az útvonalat. Emellett viszont ha egy torlódási ponthoz több egység egyszerre ér, előfordulhat, hogy túl hosszú utakat kapunk megoldásként, esetleg nem is talál megoldást. Emellett ilyenkor számolnunk kell azzal is, hogy esetleg minden lépésben újra lefut egy teljes keresés. Így összességében nem mondható hatékony algoritmusnak az újabb valós idejű stratégiai játékokban. [5]

4.2.3 Kooperatív A*-algoritmus[11]

Az algoritmus futása során az egységek egymás után keresik az útvonalat a céljuk felé. Az egységeknek ismerniük kell a többi egység tervezett útvonalát, és a keresésnek figyelembe

[9] Windowed Hierarchical Cooperative A* (WHCA*)
[10] Local-Repair A* (LRA*)
[11] Cooperative A* (CA*)

kell vennie az időtényezőt is, tehát azt is kell ellenőrizni, hogy abban a pillanatban, amikor az egység egy adott pozícióra lépne, lesz-e azon a pozíción egy másik egység. Ehhez az algoritmus használ egy foglaló-táblát, amelyet az egységek egymásról való ismeretének a reprezentálására használ. Ez a tábla arról szolgáltat információt, hogy egy adott időpillanatban áll-e egység adott pozíción. Amikor egy egység befejezte a keresését, regisztrálja a tervezett útvonalát a foglaló táblában. Így az egységek, amelyeknek ezután fut le a keresője, már tudni fogják az egység pontos pozícióját minden időpillanatban. Ezen kívül az egységek cselekvési lehetőségei minden állapotban kibővülnek egy várakozás operátorral is, amely hatására az egység következő cselekvésig tartja a pozícióját, így elkerülve a felesleges mozgást.
Ez az algoritmus érzékeny az egységek sorrendjére, vagyis hogy milyen sorrendben keresik az útvonalukat, mert előfordulhat, hogy egy egység a megtervezett útvonalával elzár minden megoldást egy másik egység elől. Heurisztikaként általában alapvetően Manhattan-heurisztikát használunk, bár egy bonyolultabb térképen ez a heurisztika elég gyenge eredményt produkálhat, sok csomópontot feleslegesen kiterjesztve, így érdemesebb lehet egy pontosabb heurisztikát választani. [5]

4.2.4 Hierarchikus kooperatív A*-algoritmus [12]

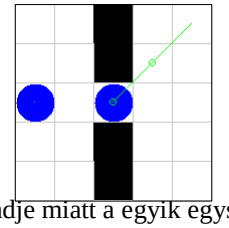
Tehát azt tudjuk, hogy minél jobb egy heurisztika, annál gyorsabb az algoritmus. Például tökéletes heurisztika esetén az A*-algoritmus csak az optimális úton lévő csomópontokat terjeszti ki. Egyik megoldás, hogy előre letároljuk a térkép összes lehetséges két pontja között az optimális távolságot. Ez elég pazarló megoldásnak mondható, mert például egy 512x512 mezős térképen, melynek az 50%-a járható, – ekkor 131 072 mezőt kell számításba venni. Körülbelül 8,5 milliárd utat kellene kiszámolni és letárolni. Ez egyrészt rengeteg memóriát foglalna, másrészt nagyon sokáig tartana legenerálni. Másik megoldásként viszont használhatunk hierarchikus A*-algoritmust. Ebben hierarchiaként az állapottér absztrakcióját használjuk. Ez az absztrakció abból áll, hogy elhagyjuk az időtényezőt, s ezzel együtt a foglaló-táblát is. Ez így tökéletesen becsüli a távolságot a kiinduló pozíció és a cél között az egység számára, amennyiben az közben nem ütközik más egységgel. A hierarchikus A-algoritmus egyik lényeges kérdése, hogy hogyan hasznosítsuk újra a már kiszámolt információt. Jelen esetben a megoldás egy visszafelé haladó folytatható A*-algoritmus [13] használata lesz. Ez a kereső az egység céljától indul, és az egység kezdeti pozíciója felé tart.

[12] Hierarchical Cooperative A* (HCA*)
[13] Reverse Resumable A* (RRA*)

Mivel visszafelé haladunk, ügyelnünk kell arra, hogy *N* állapot kiterjesztésekor nem azokat a csomópontokat szeretnénk kapni, amik *N*-ből elérhetőek, hanem amiből *N* elérhető. Az útvonalkeresésben viszont az a szerencsés helyzet áll fenn, hogy ha *A*-ból közvetlenül elérhető *B*, akkor *B*-ből is közvetlenül elérhető *A* és viszont. Az egyetlen dolog, amire figyelnünk kell, hogy az operátorok inverzét kapjuk az állapotok között. A keresés addig folyik, amíg egy adott csomópont ki nem lesz terjesztve, nem pedig addig, amíg a kiinduló-pozíciót elérjük, így amikor a keresés befejeződik, pontosan tudjuk az adott csomópont optimális távolságát a céltól (az A*-algoritmus tulajdonságaiból adódik). Az algoritmus azért folytatható, mert nem törli a nyílt és zárt csomópontok listáját. Ez azzal az előnnyel jár, hogy ha szeretnénk tudni egy pozíció távolságát a céltól, és az a csomópont már ki lett terjesztve, egyszerűen megkeressük a zárt csomópontok között, és visszaadjuk az értéket, ha pedig nem, akkor folytathatjuk a keresést, amíg a csomópontot ki nem terjesztjük. Tehát a hierarchikus kooperatív A*-algoritmus nem más, mint a kooperatív A*-algoritmus, amiben heurisztikaként hátrafelé haladó folytatható A*-algoritmust használunk. Amennyiben a célig optimális utat nem keresztezi egyetlen másik egység sem, a heurisztika az első hívásnál kiszámolja az összes szükséges távolságot a célig, és amikor szükség van rá, már tartalmazni fogja. Ha más egység kerül az útba, akkor pedig folytatja a keresést, amíg ki nem terjeszti az adott csomópontot. [5]

4.2.5 Ablakolt hierarchikus kooperatív A*-algoritmus

Az egyik közös probléma az előző három algoritmusban, hogy mindig végigszámolják az utat a kiindulástól a célig, ráadásul az utóbbi kettő ezt egy igen nagy állapottérben teszi figyelembe véve az idő-tényezőt is. Ezen kívül nagyban befolyásolhatja a keresés sikerességét, hogy melyik egység keresi előbb az útvonalát.



21. ábra: A keresés sorrendje miatt a egyik egység nem talál útvonalat

Például, tegyük fel, hogy U_1 egység egy olyan pozíción áll, amely az egyetlen átjáró két terület – *A* és *B* – között, U_2 egység pedig az *A* területen. Mindkét egység azt a parancsot kapja, hogy lépjen *B* terület egyik pontjára. Ekkor, ha U_2 egység keres először, a keresés eredménytelen lesz, hiszen ekkor még nincs információ arról, hogy U_1 el fog-e mozdulni az

útból. Ugyan ebben a szituációban, ha először U_1 egység keresése fut le, U_2 is megtalálja az útvonalat.

Az „ablakolás” megoldást jelent ezekre a problémákra. Az „ablakolás” alatt azt értjük, hogy a keresést lekorlátozzuk rögzített *w* mélységre, így az egységek az útvonaluknak csak egy részét keresik meg úgy, hogy eközben figyelembe veszik egymás mozgását, viszont nincs információjuk az „ablakon” kívül eső egységek mozgásáról, így felgyorsítva a keresést. A heurisztikaként használt RRA*- algoritmus viszont továbbra is a teljes útvonalat keresi, így garantálva, hogy a részútvonal végpontja az absztrakt optimális útvonalon fog elhelyezkedni. Az egységek ezután ezt a részútvonalat bizonyos időközönként újraszámolják. Erre azért van szükség, mert előfordulhat ütközés, például amikor egy akadálymentes környezetben két egység *t* mező távolságra van egymástól, egymás irányába tartanak az optimális útvonalon, és az ablakméret éppen $w = t - 1$.

Ahhoz, hogy az „ablakolt” keresést megvalósítsuk, meg kell változtatnunk az A*-algoritmus leállási feltételét. Van azonban egy másik megoldás is. A csomópontok kiterjesztése során csak akkor generáljuk le az elérhető állapotokat, ha a csomópont kevesebb, mint *w* lépésre van az egység aktuális pozíciójától. Ha *N* a kiterjesztendő csomópont, és *N* pontosan *w* lépésre van az egység aktuális pozíciójától – vagyis amikor elértük ablakméretet –, egyetlen speciális operátort alkalmazunk rá, amely rögtön a célállapotba juttat, és *G* célállapot esetén költsége megegyezik az *N* és *G* absztrakt távolságával.

Az algoritmus gyorsaságán tudunk még javítani azzal, ha ugyanazt az RRA* algoritmust használjuk heurisztikaként végig, az egész útvonalon. Ez az algoritmus végig az egység kezdeti pozíciója és a cél között folytatja a csomópontok kiterjesztését, különben inkonzisztenssé válhatnak a már tárolt csomópontok. [5]

4.3 Útvonalkeresésben használatos lényegesebb heurisztikák

Ezen heurisztikák közös tulajdonsága, hogy az adott állapot és a célállapot koordinátáiból próbálnak becslést adni a tényleges távolságra. A továbbiakban egy állapot *x* mezőjére hivatkozva az állapot által tartalmazott pozíció *x* koordinátáját, az *y* mezőjére hivatkozva az *y* koordinátáját értem. Emellett *g*-vel jelölöm a célállapotot. Az alábbi heurisztikák mindegyike elfogadható és monoton, ha a megfelelő tulajdonságú állapottéren alkalmazzuk.

4.3.1 Manhattan távolság

Ha egy mezőről négy irányban léphetünk, Manhattan távolság alapú heurisztikát ajánlott használnunk. A Manhattan távolság két pont között a koordináta tengelyek mentén mért távolság. A heurisztika értékét a következő képlet adja:

$$h_1(n) = c \cdot (abs(n.x - g.x) + abs(n.y - g.y))$$

Ahol *c* a legkisebb költségű operátor alkalmazási költsége.

4.3.2 Átlós távolság

A Manhattan távolság nyolc irányú megfelelője. Értékét a következő kifejezés adja:

$$h_2(n) = c \cdot \max(abs(n.x - g.x), abs(n.y - g.y))$$

Ahol *c* szintén a legkisebb költségű operátor alkalmazási költsége. Ha összesen kétfajta költséget használunk – egyet az átlós, és egyet a nem átlós mozgáshoz –, akkor a következő kifejezés hatékonyabb lehet:

$$h_2'(n) = c_2 h'(n) + c_1(h_1(n) - 2 \cdot h'(n))$$

Ahol, c_1 az egyenes, c_2 az átlós mozgás költsége.

4.3.3 Euklideszi távolság

Egy folytonos térképen bármilyen irányban megengedett a mozgás, ekkor az euklideszi heurisztika a megfelelő választás:

$$h_3(n) = c\sqrt{(n.x - g.x)^2 + (n.y - g.y)^2}$$

Habár a gyökvonás meglehetősen számolásigényes művelet, nem emelhetjük négyzetre az egész kifejezést, hogy eltüntessük, mert a négyzetre-emelt heurisztika értéke több lehet, mint a valós távolság, így pedig már nem lesz elfogadható a heurisztika.

4.3.4 Pontos heurisztika

Egy módszer arra, hogy pontos heurisztikát kapjunk, hogy előre letároljuk minden pontban az összes többi pontba vezető optimális távolságot. Ez viszont nagyobb térképek esetén egyrészt nagyon sok időbe telne, másrészt túl nagy memória kellene a tároláshoz. Ezen kívül dinamikus környezetben nem is feltétlenül marad pontos.

Ehelyett érdemes megfigyelnünk, hogy pontos heurisztikát kaphatunk olyan esetben, ha speciális a környezet. Tehát, ha például nincs akadály az útban, akkor a két pont közötti távolság lesz a legrövidebb távolság. Ekkor használhatjuk az előző heurisztikák közül a megfelelőt.

5. Implementáció

Egy valós idejű stratégiai játék elkészítése egy több tagból álló fejlesztőcsapat számára is hosszú időt vehet igénybe. Az én célom egy olyan játék létrehozása, amely megvalósítja az útvonalkeresést több egység esetén is, és meg is jeleníti a felhasználó számára. Mindezt valós időben, interaktívan teszi, a felhasználó beavatkozásának lehetőségével. Ezen felül elvárás, hogy a rendszer egyszerűen bővíthető legyen új funkciókkal.

A játék térképe diszkrét, melyen a rácspontok négyzet alakúak, és mérete legfeljebb 128x128 rácspont nagyságú lehet. A térképen nyolc irányba lehet mozogni. A program több egységet egy időben kezel. Az egységek maximális száma 20-ra van korlátozva. A keresést pedig egy ablakolt hierarchikus kooperatív A*-kereső implementálásával oldottam meg. Mint azt már említettem, ez egy visszafelé haladó folytatható keresőt használ. Ez pedig átlós távolságon alapuló heurisztikával dolgozik.

A program általános felépítése három fő részre bontható. Az első rész felelős a játék elemeinek, illetve viselkedésének a leírásáért (a játék logikája). A második rész a játék megjelenítéséért, és a harmadik rész a felhasználói interakció hatékony kezeléséért. Ez a felosztás megfelel a model-view-controller tervezési mintának. [6]

Az első feladat egy grafikus keretrendszer létrehozása volt, amely elvégzi a felhasználó számára is látható elemeinek megjelenítését. Mindezt úgy, hogy lehetőséget nyújt ablakos és teljes-képernyős megjelenítés, dupla-pufferelés és élsimítás használatára. Ezzel együtt egy olyan általános keretrendszert is létre kellett hozni, amely segítségével egyszerűen megvalósítható a megjelenítés és a játék logikája közötti kapcsolat. Ezután következett a játéktér leírása, illetve ennek az általános keretrendszerrel történő összekapcsolása, az az által nyújtott lehetőségek segítségével. A játéktér foglalja magában a térképet, az egységeket és mindent, amely a játék elemeinek viselkedését és tulajdonságait írja le. Végül pedig implementálni kellett az útvonalkereső algoritmust.

5.1 A grafikus keretrendszer

A grafikus keretrendszert a `framework.graphics` csomag tartalmazza. A keretrendszer olyan objektumokkal dolgozik, amelyek megjeleníthetőek. Erre a célra definiáltam két interfészt. Az egyik a `Drawable` interfész, amelyet azok az objektumok implementálnak, amelyeket ki lehet rajzolni. Az interfész tartalmazza a `draw` metódust, amely egy

`Graphics` objektumot paraméterül átadva lehetőséget nyújt az objektumnak, hogy a grafikus környezetre rajzolja magát. Ezen kívül tartalmaz egy metódust, amely igaz, vagy hamis értékkel tér vissza, annak függvényében, hogy az objektum jelenleg látszik-e a képernyőn, ezzel elkerülve a képernyőn nem látszó részek felesleges kirajzolását. Végül a `getPriority` metódus azt határozza meg, hogy az objektumnak milyen rajzolási prioritása van a többi objektumhoz képest. A nagyobb prioritású objektumok elfedik a kisebb prioritásúakat. A másik interfész a `Container` interfész. Ezt az interfészt azok az objektumok implementálják, amelyeket nem lehet közvetlenül kirajzolni, de tartalmaznak kirajzolható objektumokat. Egyetlen metódusa van, mely visszaadja a tartalmazott `Drawable` interfészt implementáló objektumokat.

A következő lépés volt a grafikus keretrendszer elkészítésénél, hogy definiáltam a megjelenítőt. A megjelenítést két osztály végzi. Az egyik az ablakos, míg a másik a teljes-képernyős megjelenítésért felelős. Ahhoz, hogy a játék keretrendszere mindkettőt egyszerűen kezelni tudja, egy közös absztrakt ősosztályból származtattam őket. Ez az osztály az `OOGDisplay`. Ez kiterjeszti a `JFrame` osztályt, hogy kívülről is el lehessen érni egyes metódusait, például így adhatunk a megjelenítőhöz `Listener` objektumokat is, amelyek a felhasználói interkaciókat figyelik, és kezelik. Az `init` metódus végzi a képernyő megjelenítéséhez szükséges beállítások végrehajtását, illetve a magát megjelenítést is. A `restoreScreen` metódus pedig lezárja a megjelenítőt, és visszaállítja a képernyőt. A legfontosabb metódus – ha lehet ilyet mondani – a `drawOOG` nevű, amely a paraméterül kapott kirajzolható objektumokat rajzolja ki. Az objektumok egy `PriorityQueue` típusú kollekcióba vannak szedve, amely az objektumok prioritása alapján van rendezve. Ebben a metódusban figyelni kell arra, hogy amennyiben van háttér-puffer, akkor először arra történjen a rajzolás, és annak tartalma ezután egyben kerüljön a képernyőre. A `reinitBackBuffer` metódus szolgál arra, hogy egy ilyen puffert újra létrehozzunk, például amikor megváltozik a képernyő mérete. Végül az interfész lehetőséget nyújt a megjelenítés felbontásának, illetve méretének beállítására a `setDisplayMode` metódussal, illetve a jelenlegi beállítások lekérdezésére a `getDisplayMode` metódus segítségével.

A csomag tartalmaz még két osztályt, az `OOGFullScreen` és az `OOGWindow` osztályokat, melyek rendre a teljes-képernyős illetve az ablakos megjelenítést végzik el. Ezek egyszerűen csak a funkciójuknak megfelelően implementálják az `OOGDisplay` interfészt.

5.2 A játék keretrendszer

Ez a keretrendszer, amely megvalósítja a program logikája, és a megjelenítés közötti kapcsolatot. Ez a `framework.game` csomagban található. Ahhoz, hogy a keretrendszer jól működjön, ugyanúgy kell tudnia kezelni egy olyan játékot, amelyen például csak egy labda pattog, mint egy olyat, amelyen például egy autóval kell egy versenypályán minél gyorsabban körbeérni. Ezt egy egyszerű általánosítással értem el, amelyet egy interfész létrehozásával tettem meg. Ez az interfész a `Game` interfész. Ez a játéklogika általánosítása. A játéklogika külön szálon fut, ezért ki kell terjeszteni a `Runnable` interfészt. A játéknak ismernie kell a keretrendszert, amely futtatja, hogy a felhasználói interakciót megfelelően kezelni tudja. Erre szolgál az `initGame` metódus, amely paraméterül adja át magát a keretrendszert. Ezen kívül tudnunk kell, hogy a játék mikor fejeződött be, hiszen ezután a keretrendszernek is felesleges futnia. Ezt az `isFinished` metódus meghívásával érhetjük el. Végül kell egy olyan metódus, amely segítségével elkérhetjük a játéktól azokat az objektumokat, amelyeket meg lehet jeleníteni. Ez a `getOOGs` metódus feladata, amelyet a `Containter` interfész implementálásából kap, ezáltal a játék maga egy olyan objektumnak tekinthető, amely kirajzolható elemeket tartalmaz.

A csomag másik osztálya a `GameEngine` osztály. Ez az osztály vezérli a megjelenítő osztályokat, tehát ez is egy külön szálon fog futni, vagyis implementálja a `Runnable` interfészt. Az osztály a `run` metódusában egy ciklus fut egészen addig, amíg a játék, amelyet kezel, be nem fejeződik. A ciklusban pedig `PriorityQueue`-ba rendezi a játék megjeleníthető objektumait, majd átadja őket a megjelenítőnek. Mindezt egy meghatározott másodpercenkénti frissítések számának betartása mellett. Az osztálynak a példányosításhoz egy `Game` objektumra van szüksége, amelyet referenciaként tárol el. Az osztály ezen kívül lehetőséget nyújt különböző tulajdonságok beállítására, mint például a már említett másodpercenkénti frissítések száma, ablakos, vagy teljes-képernyőn való futás, az ablak méretének, illetve teljes-képernyős megjelenítés esetén a használni kívánt képernyő felbontás megadása, valamint ezen tulajdonságok lekérdezése.

5.3 A játéktér

A legösszetettebb feladat a program írásakor a játéktér megfelelő megalkotása volt. A játéktér foglalja magába azon elemeket, amelyek a játék logikájának működéséhez szükségesek.

5.3.1 A `Terrain` és a `TerrainType` osztályok

A játék világának egyik fő alkotóeleme a térkép. Ezen mozognak az egységek, illetve ez határozza meg, hogy mely mezők járhatóak az egységek számára. A térkép tárolására egy 2-dimenziós tömböt használok, melynek elemei `TerrainType` objektumok. A `TerrainType` egy absztrakt osztály, melyet kiterjesztve lehet definiálni különböző tulajdonságú területeket, például földet, füvet, hegyet, vagy akár vizet. Az osztály kiterjeszti a `Drawable` interfészt, de a `draw` metódusát nem implementálja. Így a különböző területek különböző grafikával jelenhetnek meg. Konkrét térképterület-típusokat külön, nem publikus osztályokként, ugyanabban a csomagban hoztam létre. Egy *Ground* nevű járható, és egy *Wall* nevű nem járható területet implementáltam. Melyek fehér és fekete színű négyzetekként jelennek meg a játékban. Ezen kívül az osztály definiál két, ezeknek megfelelő konstans értéket, melyek segítségével egy 2-dimenziós `byte` tömböt meg tud feleltetni egy `TerrainType` objektumokból álló tömbnek.

A `Terrain` osztály kiterjeszti a `Containter` interfészt, hiszen kirajzolható, `TerrainType` objektumokat tartalmaz. A `getOOGs` metódus a tartalmazott `TerrainType` objektumok közül azokat adja vissza, amelyek láthatóak a képernyőn, hiszen felesleges kirajzolni azokat, amelyek nem látszanak.

Az osztály egyik konstruktora egy 2-dimenziós `byte` tömböt kap paraméterül, amelyből az előbb említett konstansok segítségével felépíti a `TerrainType` objektumokat tartalmazó tömböt. Ennek az egyszerűsítésére szolgál a két privát `newTerrain` metódus, amely elvégzi az objektumok létrehozását.

Az osztálynak van még egy konstruktora, amely paraméter nélküli. Ez az osztály `generateSimpleTerrain` metódusát használva generál egy véletlenszerű `byte` tömböt. Az osztály három darab `generateSimpleTerrain` metódust tartalmaz. Az első hét paramétert vár, így a generálást végző programkód teljesen „finom-hangolható”. A másik két metódus ezt hívja meg. A másodikban csak a térkép szélességét és a magasságát lehet beállítani, a többi előre beállított alap érték marad. A harmadik nem vár paramétert, és minden értéket beállít egy alap értékre.

A Terrain osztály végül szolgáltat néhány egyéb metódust, amelyekkel például a térkép szélességét és magasságát lehet lekérdezni (`getWidth`, `getHeight`), illetve egy adott pozíción lévő területet lehet beállítani, vagy járhatóságát lekérdezni (`getTerrainAt`, `setTerrainAt`).

5.3.2 A `Unit` osztály

A játéktér másik fontos elemei az egységek. Az egységeket irányíthatjuk a játék során. Az egységek tulajdonságainak és viselkedésének a megvalósítása a `Unit` osztály feladata. Minden `Unit` példány tartalmaz egy `int` típusú `id` nevű mezőt, mely egy azonosító az objektumnak. Ez minden példány számára különböző, és ennek segítségével gyorsan el lehet dönteni két egységről, hogy azonosak-e. Ezt a mezőt használja a `hashCode` metódus is. Az értékét pedig a `Unit` osztály statikus metódusa adja, a `getNextId`, amely minden hívásnál növel egy statikus változót és visszaadja annak értékét. Az osztály ezen kívül tartalmaz egy `defaultSpeed` példányváltozót is, amely segítségével az egység cselekvései közötti játékfrissítések számát lehet beállítani – más szóval az egység sebességét – egy statikus alap értékhez viszonyítva. Elméletben így létre lehet hozni különböző sebességű egységeket, illetve dinamikusan változtatni az egységek sebességét, de akkor az útvonalkereső algoritmusnak is figyelembe kellene vennie ezt, ami bonyolultabb keresőalgoritmus implementálását teszi szükségessé. Az én célom a program fejlesztésekor a gyorsaság mellett az egyszerűség is volt, ezért a jelenleg implementált keresőalgoritmus ezt nem támogatja.

Az osztály tartalmaz még két `LinkedList` adatszerkezetet, amely `Position` objektumokat tartalmaz. A `Position` osztály pedig egy csomagolóosztály, amely egy térképen levő pozíciót tartalmaz, pontosabban annak a sorát, és az oszlopát csomagolja be. A `Unit` osztály egyik listája a `route`, amely azt az útvonalat tartalmazza, amelyet az egység jelenleg követ. A lista első eleme mindig az egység aktuális pozíciója. A lista minden pozíciója szomszédos a listában őt követő és megelőző pozícióval. A másik lista a `waypoints`, amely nem közvetlen útvonalat tartalmaz az egység számára, hanem célok sorozatát, amelyeket el kell érnie.

A feldolgozást, vagyis az objektum működését két metódus vezérli, melyek egymás után hívódnak meg. Az első az `actPlan` metódus, amely ellenőrzi, hogy van-e még pozíció a `route` listában ahova léphet – ez azt jelenti, hogy egynél több eleme van – és ha már nincs közvetlen útvonala, akkor megnézi, hogy a `waypoints` lista tartalmaz-e elemet. Ha a `waypoints` lista üres, az azt jelenti, hogy az egység elérte a célját, ha nem üres, akkor pedig itt generálja le az útvonalat a `waypoints` lista első eleme alapján. Megjegyzendő, hogy az implementált útvonalkereső algoritmus nem feltétlenül keresi meg az egész útvonalat az egységtől egészen a célig, így vizsgálni kell, hogy a kereső által visszaadott útvonal a célig tart-e. Ha igen akkor a `waypoints` lista első elemét el lehet távolítani.

A kereső működéséből adódóan itt felmerülhetnek problémák. Az egyik ilyen, hogy az egység egészen addig futtatja a keresést, amíg el nem érte a célpozíciót. Ez akkor probléma, amikor már egy másik egység elfoglalta a helyet. Ilyenkor az egység minden lépés előtt lefuttatja a keresőt, és próbál utat keresni a céljához – természetesen eredménytelenül. Egy egyszerű megoldás lenne az, hogy vizsgáljuk, hogy foglalt-e a célpozíció, és ha igen, akkor nem folytatjuk a keresést. Ez viszont újabb problémákhoz vezetne, ugyanis előfordulhat, hogy két egység tart egy cél felé, és az egyik sokkal hamarabb eléri a célt. Ekkor a másik egység az előzőek alapján megállna, mert foglalt a pozíció, pedig a céltól még nagyon messze van. A következő megoldás, hogy ellenőrizzük, hogy a cél egy bizonyos sugarán belül vagyunk-e. Ekkor viszont a sugár megválasztása okozhat problémát. Ha túl kicsire választjuk a sugarat, akkor több egység esetén, a sugáron kívül tartózkodók továbbra is keresni fognak, míg túl nagy sugár esetén, ha egy egység a célpozícióban van, a többi egység a sugár mentén fog megállni, és nem megy közelebb a célhoz. A megoldás, amelyet implementáltam a probléma megoldására, egy kicsit összetettebb. Elindul a célpozícióból az egység pozíciójának vonalában, és megvizsgál minden pozíciót, hogy foglalt-e. A keresés csak akkor áll le, ha a céltól az egységig minden egyes pozíció foglalt.

A `Unit` osztály másik feldolgozó metódusa az `actMove`. Ez végzi el az egység „mozgatását”, vagyis lényegében annyit tesz, hogy a `route` lista első elemét eltávolítja. Ez előtt pedig ellenőrzi, hogy nem foglalt-e a pozíció ahová lépni kellene, mert ez is előfordulhat. Ha igen, akkor két lehetőséget lehet implementálni. Vagy halasztja a lépést egyel későbbre, hátha akkor felszabadul a pozíció, majd ha tudja, folytatja az útját. Vagy ami talán intelligensebbnek tűnik, újratervezi az útvonalat. Ezt egy egyszerű trükk segítségével valósítottam meg. A `route` lista utolsó elemét hozzáadtam a `waypoints` lista elejéhez, majd töröltem a `route` listát – pontosabban az első elem utáni elemeket, hiszen az első elem az aktuális pozíció, amit nem szabad törölni. Ezáltal a következő lépésben az `actPlan` metódusban újratervezi az útvonalat, feltöltve a `route` listát.

A vezérlést azért kellett két metódusra bontani, mert így megtehetjük azt, hogy minden egység először megkeresi az útvonalat, majd utána minden egység lép. Erre azért volt szükség, mert ha több egységgel dolgozunk, és minden egység sorban megkeresi az útvonalát, majd rögtön meg is lépi az első lépést, akkor a többi egység nem tudná pontosan kiszámítani az egység útvonalát.

Az osztály tartalmaz még lekérdezéseket is, például a `getPos` metódus, amely az egység pozícióját adja vissza, vagyis a `route` lista első elemét. A `getPosAfter` metódussal is az egység pozícióját kapjuk meg, de ez a paraméterül kapott lépésszám elteltével kialakult állapotra vonatkozik. Ez a `route` lista i. elemét adja vissza, ahol i a paraméter, de ha i nagyobb, mint a lista mérete, két lehetőségünk van. Vagy üres a `waypoints` lista, ekkor a `route` lista utolsó elemét kell visszaadni, vagy nem üres a `waypoints`, ekkor pedig nem adunk vissza pozíciót, mert azt nem tudjuk, hogy a `waypoints` listában szereplő pozíciókat hány lépés múlva éri el az egység.
Ezeken kívül a néhány segédmetódus is található az osztályban. Az `addWayPoint` metódus a `waypoints` lista végére illeszt egy elemet, a `clearRoute` metódus elvégzi a `route` lista kiürítését, úgy, hogy az első elemet benne hagyja, és a `stop` metódus, amely üríti a `waypoints` listát, majd a `route` listát is, a `clearRoute` meghívásával.
Végül az osztály implementálja a `Drawable` interfészt, hogy megjeleníthessük az egységeket a képernyőn.

5.3.3 A `World` osztály

A `World` osztály arra szolgál, hogy egybefoglalja, és kezelje az előzőekben leírt két osztályt, a `Terrain` és a `Unit` osztályokat. Egy `World` objektum egyetlen `Terrain` objektumot tartalmaz, amelyet egy publikus mezőben tárol. Emellett tartalmazhat több Unit osztály példányt, melyek tárolását egy `HashMap` végzi a `units` mezőben, amely kulcsként a Unit azonosítóját, vagyis az `id` mezőjét használja. Az osztály konstruktora paraméterül kaphat egy `Terrain` objektumot, amelyet eltárol, vagy ha paraméter nélkül hívjuk meg, létrehoz egyet.
Az osztály `act` metódusa annyit tesz, hogy sorban meghívja az összes tartalmazott egység `actPlan` metódusát, majd ezután az összes egység `actMove` metódusát.
Az egységek kezelésére különböző metódusok találhatóak az osztályban. Az `addUnit` hozzáad egy egységet a `units` kollekcióhoz. A `removeUnit` elveszi a kollekcióból a paraméterül kapott egységet, vagy a paraméterül kapott pozíción lévő egységet. A `getUnit` visszaadja a paraméterül kapott pozíción lévő egységet, és a `getUnitAfter` pedig visszaadja azt az egységet, amely paraméterül kapott lépésszám megtétele után a paraméterül kapott pozíción lesz – ha van ilyen. Ezen kívül itt található a `freeLOS` metódus is, amely a `Unit` osztálynál említett foglaltság vizsgálatot végzi az egység célja, és pozíciója közötti egyenes vonal mentén.

Az osztály még egy eszközt nyújt az útvonalkeresés felgyorsítására. Ez az úgynevezett *kereső tartalék*. Ez egy olyan lista, amely az eddigi keresések heurisztikáit tárolja, mivel a kereső jellegéből adódóan a heurisztika felépítése időigényes folyamat lehet, és mint azt már a kereső leírásánál is említettem, jelentősen le lehet csökkenteni a kereséshez szükséges időt, ha a heurisztikát megfelelően újra tudjuk használni. A heurisztikát a célpozíció jellemzi. Tehát amikor egy egység futtatni szeretné a keresőt, először megnézi, hogy a `searchPool` lista tartalmaz-e olyan heurisztikát, amely célpozíciója megegyezik az egység célpozíciójával. Ha igen, akkor azt használja heurisztikaként. Ha nincs megfelelő heurisztika a kereső tartalékban, akkor egy új heurisztikát használ a kereséshez, ami bekerül a tartalékba. Mivel ezek a heurisztikák egy idő után nagyon sok memóriát foglalhatnak, nem tároljuk az összes eddigit, csak a legutolsó néhányat. Ezt a korlátot állítja be a `World` osztály statikus konstansa, a `SEARCHPOOL_MAX_SIZE`.

Végül a `World` osztály implementálja a `Containter` interfészt, melynek a `getOOGs` metódusában visszaadja azokat az egységeket a `units` kollekcióból, amelyek látszanak a képernyőn, illetve a tartalmazott `Terrain` példány összes kirajzolható objektumát.

5.3.4 A `Player` osztály

A `gamespace` csomag tartalmaz még egy `Player` osztályt, amely a tulajdonképpen a kapcsolatot teremti meg a játék elemei, és a felhasználói interakciók között. Itt kapnak helyet azok a metódusok, amelyek az egységeket vezérlik. A `cmdUintsTo` metódus a paraméterül kapott összes egység útvonalát törli, és a paraméterül kapott pozíciót beállítja az egységek `addWaypoint` metódusa segítségével. A `cmdUnitsToAfter` metódus a hasonló az előzőhöz, de ez a metódus nem törli az egységek útvonalát, így a kapott pozíció felé csak azután indulnak, miután befejezték az előző útvonalukat. Végül a `cmdUnitStop` metódus az összes paraméterül kapott egységet megállítja, vagyis törli a tervezett útvonalukat, az egységek `stop` metódusa segítségével.

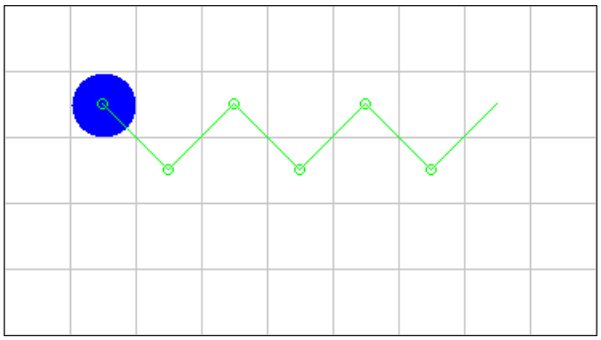
A `Player` osztályt azért érdemes használni ahelyett, hogy közvetlenül kezelnénk az egységeket, mert így egyszerűbbé válik több játékos kezelése is. Bár jelenleg a program nem használja ki a több játékos kezelésének lehetőségét, a kód továbbra is átlátható marad, és lehetőséget nyújt a későbbi fejlesztésre.

5.3.5 A `Move` osztály

Ez az osztály egy `enum` osztály, és egyedül arra szolgál, hogy definiálja a játéktéren való lehetséges mozgásokat az egységek számára. Kilenc példánya van, melyek megfelelnek a nyolc iránynak, kibővítve egy semleges cselekvéssel.

A példányoknak két mezője van, az egyik az iránynak megfelelő sor-koordináta változásának értékét, a másik az iránynak megfelelő oszlop-koordináta értékét tárolja.

A `getCost` metódus pedig egyet ad vissza értékül, ha a mozgás a koordinátatengelyek mentén történik, illetve „valamivel nagyobb”, mint egyet, ha a mozgás átlós irányú. A „valamivel nagyobbat” én 1.00001-nek választottam. Erre azért van szükség, hogy a kereső előnyben részesítse az egyenes vonalú mozgást. Ha ezt nem tennénk meg, „cikk-cakkos” mozgást kaphatnánk egy egyszerű egyenes szakaszon.



22. ábra: A kereső véletlenszerűen választ azonos költségű utak közül

5.4 Az absztrakció implementálása

A `game.abstraction` csomag két, előzőekben tárgyalt absztrakciót implementál. Mindkettőre jellemző, hogy az `AbstractTerrainFactory` osztály hozza létre a kapott térképből az absztrahált csomópontokat. A csomópontokat mindkét esetben a Field objektumokként kapjuk meg, de a két absztrakciónál eltérő adatszerkezetet használ.

5.4.1 Klikk absztrakció

A `game.abstraction.clique` csomag tartalmazza a klikk absztrakciót előállító, illetve az ahhoz szükséges osztályokat. A klikk absztrakció `Field` osztálya tartalmaz három `HashMap`-et, amelyekben tárolja a csomópont szomszédait (`neighbours`), gyermekeit (`children`), illetve az összes pozíciót (`base`), amelyet absztrahál. Ezen kívül egy szülőre mutató referenciát is (`parent`), és tárolja az absztrakciós csomópont szintjét is. Ezen kívül tartalmaz egy azonosításra szolgáló `id` mezőt, amely minden tekintetben megegyezik a `Unit`

osztálynál már bemutatott `id` mezővel, így erre itt most nem térek ki még egyszer. Az osztály metódusai ezen mezők kezelésére szolgálnak.

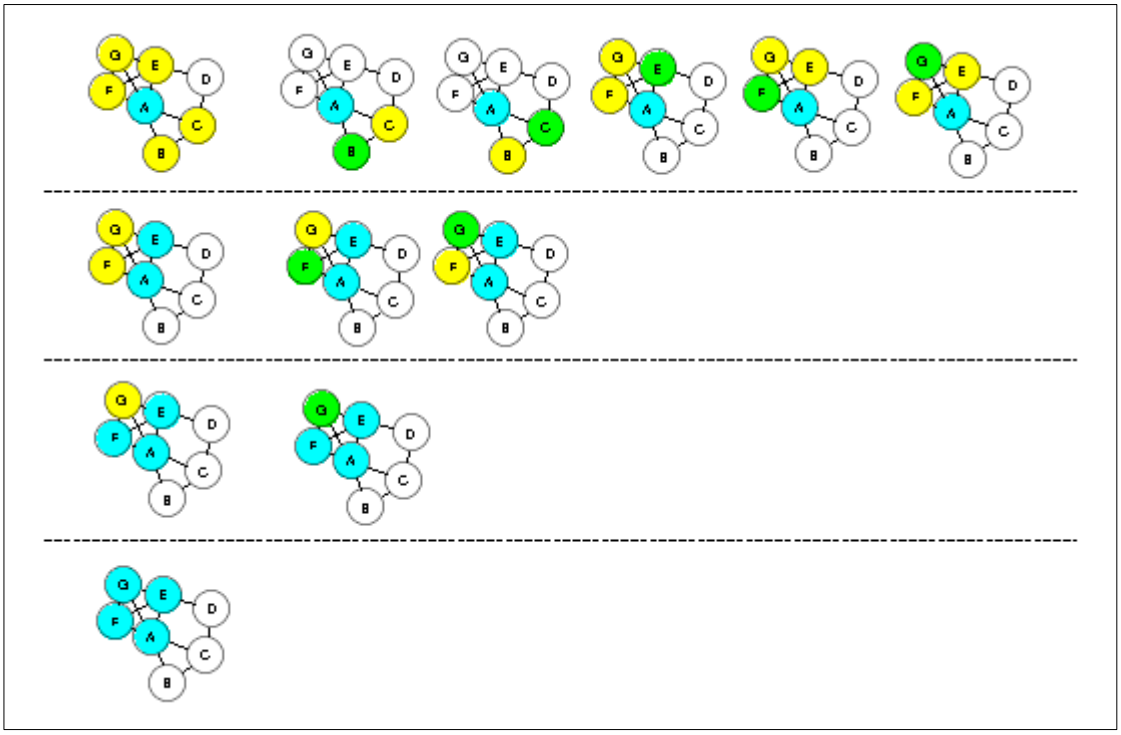
A klikk absztrakció a következőképpen működik:

Először létrehozzuk a legalsó szintű absztrakciót, melynek során a térkép minden járható területéhez létrehozunk egy megfelelő tartalmazó csomópontot, és a beállítjuk a szomszédi kapcsolatokat, a térképen szomszédos mezőknek megfelelő csomópontokkal. Ezután a kapott gráfból, a klikk absztrakció szabályai szerint új gráfot képzünk, és ezt addig folytatjuk, amíg már nem lehet magasabb absztrakciót létrehozni.

Az új gráf képzésekor a régi gráf elemeinek két listát hozunk létre, az egyik az absztrahált, a másik a nem-absztrahált csomópontokat tartalmazza majd. Kezdetben mindkét lista üres. Azután a régi gráf elemeit beletesszük a nem-absztrahált listába és végigmegyünk ennek azon elemein, amelyeket nem tartalmaz az absztrahált lista, és mindhez legenerálunk egy új csomópontot, amely az aktuális elemhez tartozó legnagyobb klikk elemeit tartalmazza gyermekekként, és egy ideiglenes listába tesszük, amely a csomópontok által reprezentált terület nagysága szerint van növekvő sorrendbe rendezve.

A nem-absztrahált lista bejárása után az ideiglenes listán megyünk végig, és ellenőrizzük, hogy az aktuális elem gyermekei közül tartalmazza-e valamelyiket az absztrahált lista. Ha nem, akkor a csomópont bekerül a végleges listába, a gyermekei pedig az absztrahált listába. Ha bármelyik gyermeke szerepel az absztrahált listában, akkor a gyermekek bekerülnek egy új nem-absztrahált listába, és a csomópontot pedig eldobjuk. Ezután a nem-absztrahált lista lesz az új-nem absztrahált lista, és az egész addig folytatódik, amíg ez üres nem lesz.

Egy csomóponthoz tartozó legnagyobb klikk előállítását a következő módon értem el. A kapott csomópont legyen *C*. Legyen C szomszédainak a halmaza S_C. A jelenlegi legnagyobb klikk legyen a *Q* halmaz, amely egyedül *C*-t tartalmazza. A lehetséges klikk-elemeket jelöljük *Qs*-el. Kezdetben $Q_S = S_C$. Végigmegyek Q_S elemein. Az aktuális elem legyen *N*, a szomszédai pedig S_N. S_N halmaz Q_S-el vett metszete legyen M_N. Amelyik *N*-re M_N a legnagyobb volt, azt hozzáadom a legnagyobb klikk halmazához, és újra megismétlem a ciklustól úgy, hogy $Q_S = M_{MAX}$. Ezt addig folytatom, amíg M_{MAX} üres nem lesz.



23. ábra: A klikk absztrakció működése

Az algoritmus könnyebb megértéséhez tekintsük a következő példát, ahol *A*-hoz tartozó legnagyobb klikket keressük. A jelenlegi legnagyobb klikk $Q = \{A\}$. Az *A* jelű csomópont szomszédai $S_A = Q_S = \{B, C, E, F, G\}$. Veszem S_A és $S_B = \{A, C\}$ metszetét, amiből megkapjuk, hogy $M_B = \{C\}$. Ekkor $M_{MAX} = M_B$. Hasonlóképpen $M_C = \{B\}$, $M_E = \{G, F\}$. Mivel M_E elemeinek a száma nagyobb, így $M_{MAX} = M_E$ lesz. Ezután $M_F = \{E, G\}$, $M_G = \{E, F\}$, de ezek nem nagyobbak, mint az eddigi legnagyobb metszet, így nem történik változás. Tehát a ciklus végén $Q = \{A, E\}$ és $Q_S = \{G, F\}$. A következő iterációban $M_F = \{G\}$ és $M_G = \{F\}$, tehát *F* bekerül a *Q* halmazba, és $Q_S = \{G\}$ lesz. A következő iterációban $M_{MAX} = M_G = \{\}$, így *G* is bekerül a *Q* halmazba, és a ciklus nem fut le többször, mert Q_S üres halmaz lett. Végeredményként a $Q = \{A, E, F, G\}$ halmaz tartalmazza az *A*-hoz tartozó legnagyobb klikket.

5.4.2 Kvadratikus felosztású rács

A kvadratikus felosztású rács létrehozásához szükséges osztályokat a `game.abstraction.quad` csomag tartalmazza. A `Field` osztály itt egy négyzetes területet reprezentál, amelyet egyszerűen meghatározhatunk a bal-felső és a jobb-alsó határmező pozíciójának megadásával. Ezen kívül tároljuk még a csomópont szomszédait is.

Az absztrakció menete itt a következő. Megkeressük a legkisebb olyan négyzetet, melynek oldalmérete 2 valamely hatványa, és nagyobb, vagy egyenlő, mint a térkép legnagyobb kiterjedése. A négyzet egy csomópont lesz, melynek bal-felső sarka a (0,0) pozíció, jobb-alsó sarkát pedig a kiterjedése egyértelműen meghatározza. Megnézzük, hogy ez a csomópont csak járható, vagy csak nem járható mezőket tartalmaz-e. Ha nem, akkor felosztjuk 4 darab, fele akkora oldalhosszúságú négyzetre, és mindegyikre ellenőrizzük ugyanezt a feltételt. Ezt addig folytatjuk, amíg az aktuális oldalhosszúság el nem érte az 1-et. Csak azokat a négyzeteket osztjuk további négyzetekre, amelyek nem homogének, vagyis járható és nem-járható területeket is tartalmaznak.

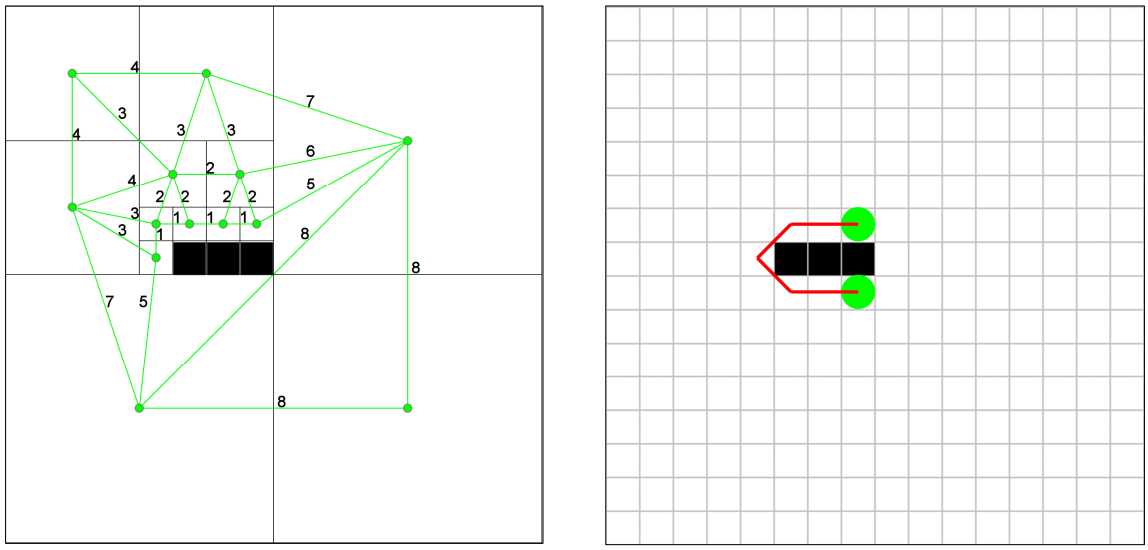
5.4.3 Megjegyzés az absztrakciókról

Mindkét fentebb ismertetett absztrakciós módszert implementáltam a programban, de teszteléskor mindkettő használatakor problémával kerültem szembe.

A klikk absztrakció felgyorsítaná a keresést, különösen annak eldöntését, hogy elérhető-e egymásból két pont a térképen. Mindemellett kiderült, hogy egy nagyobb térképen a csomópontok legenerálása nagyon hosszú időt vesz igénybe, így nem éri meg implementálni.

Ezután próbáltam ki a Kvadratikus rács alapú felosztást, amely generálása viszonylag gyors volt, és a keresést is gyorsította, de ugyanakkor a használata egyes esetekben olyan útvonalak generálását eredményezte, amelyeket nem tartottam elfogadhatónak a programban.

Ezek alapján úgy döntöttem, hogy a keresést absztrakció nélkül futtatom.



24. ábra: Az absztrakcióval talált optimális útvonal vizuálisan zavaró lehet a felhasználó számára

5.5 A játék menü

A `game.menu` csomag két osztályt tartalmaz. Az egyik a `Menu` osztály, amely tartalmazza a játékmenü elemeit. Ehhez használja fel a `MenuItem` osztályt, amely egy menüpontnak felel meg. A `MenuItem` tartalmaz egy x, és y koordinátát, szélességet, magasságot, egy szöveget, amelyet megjelenít, és egy `boolean` típusú változót, amely meghatározza, hogy jelenleg aktív-e a menüpont. Ezek a mezők határozzák meg a menüpont megjelenését. Az osztály tehát implementálja a `Drawable` interfészt, mert a menüpont a felhasználó számára látható. Az osztály `isAt` metódusa arra szolgál, hogy eldöntse, hogy egy paraméterül kapott koordináta az elem kiterjedésén belül van-e, így eldönthető egy kattintás alkalmával, hogy az adott menüponton történt-e a kattintás. Végül az osztály tartalmazza még az `operation` mezőt, amely arra szolgál, hogy azonosítani tudjuk egy menüpont funkcióját. A funkciókat a `Menu` osztály definiálja statikus konstans értékekként.

A `Menu` osztály implementálja a `Containter` interfészt, és a `getOOGs` metódusában a tartalmazott `MenuItem` objektumokat adja vissza. Ezek az objektumok az osztály konstruktorában jönnek létre. Az osztály tartalmazza még a `getItemAt` metódust, amely annak eldöntésére szolgál, hogy egy adott ponton történt kattintás egy menüponton történt-e. Ha igen, akkor a megfelelő menüpontot aktívvá teszi, a legutóbb kiválasztottat pedig inaktiválja.

Három menüpontot hoztam létre, Ha a „kijelölés” menüpont aktív, a felhasználó kijelölhet egységeket, és célt adhat meg nekik. Ha a „térkép szerkesztés” menüpont aktív, a felhasználó akadályokat tehet a térképre, illetve vehet el róla. Ha az „egység kezelés” menüpont aktív, a felhasználó egységeket tehet le a térképre, illetve távolíthat el onnan.

5.6 Segédosztályok

Mielőtt bemutatnám a játék vezérlő osztályának a működését, kitérnék azokra az osztályokra, amelyek nem kapcsolódnak szorosan a program egyik szemantikai egységéhez sem, de fontos szerepet töltenek be a kód egyszerűsítésében.

A `Controls` osztály csak statikus konstansokat tartalmaz. Ezek mindegyike egy-egy játékbeli eseményt jelöl, melyhez egy billentyű, vagy egérgomb van társítva. Ezáltal a például kilépéskor, nem azt ellenőrizzük, hogy az *Escape* billentyűt ütötte-e le a felhasználó, hanem, hogy a kilépéshez rendelt billentyűt ütötte-e le. Így ha egy eseményhez rendelt billentyűt meg szeretnénk változtatni, itt kell átírni.

A `Position` osztály egy sor- és oszlopkoordinátát csomagol be. Így sokkal egyszerűbbé teszi egyes metódusok működését. Ezen kívül `manhattanDistanceFrom` metódusa visszaadja a paraméterül kapott másik pozíciótól mért manhattan-távolságot, az `eightWayDistanceFrom` metódusa pedig a paraméterül kapott másik pozíciótól mért átlós távolságot.

A `HashMappedQueue` egy `PriorityQueue` és egy `HashMap` objektumot csomagol be. Viselkedése továbbra is megfelel egy `PriorityQueue`-nak, de a tartalmazás vizsgálatot nem a PriorityQueue-ban végzi, hanem a `HashMap`-ben, ezáltal felgyorsítva a műveletet.

Végül a `Viewer` osztály szolgál arra, hogy a játék térképén levő pozíciók, és a képernyő pontjai között megfeleltetést végezzen. Az osztály minden mezője és metódusa statikus, így bármelyik osztály hozzáférhet. Az osztály tárolja a képernyő – illetve az ablak – méretét (`displayWidht`, `displayHeight` mezők), azt hogy képernyőn a térkép mely pontja van jelenleg a képernyő bal felső sarkában (`displayFromX`, `displayFromY` mezők), és a csempeméretet, amely azt határozza meg, hogy a térkép egy pozíciója mekkora a képernyőn (`tileWidth`, `tileHeight` mezők). Ezen mezők segítségével ki tudja számolni hogy a térkép egy pozíciója a képernyő mely pontjához tartozik (`transformToScreen` metódus) illetve, hogy a képernyő egy pontja a térkép mely pozíciójára esik (`transformToWorld` metódus). A `zoom` metódust akkor hívjuk meg, amikor a térképet nagyítjuk, vagy kicsinyítjük. Ekkor a metódus a csempeméret növelésén illetve csökkentésén kívül még egy dolgot tesz. Paraméterül kapja a nagyítás „középpontját” is, amely egy képernyő koordináta, és a nagyítást úgy végzi el, hogy ez a pont a nagyítás után is a térkép ugyanazon a pozíciójára mutasson. Az osztály többi metódusa pedig mezőinek lekérdezésére, illetve beállítására szolgál.

5.7 A játékvezérlő

Az előzőekben bemutattam a `game` csomag összes alcsomagját a `game.search` kivételével, de a `game` csomag egyetlen osztályáról még nem esett szó. Ez az osztály a `PathFinder` osztály, amely a `Game` interfészt implementálja. A `PathFinder` osztály végzi az előzőekben ismertetett osztályok vezérlését, összekapcsolását. Az osztály `ups` mezője határozza meg, hogy a játék másodpercenként hányszor frissítse a játék elemeit. A `ge` mező a `GameEngine` objektumot tartalmazza, amelyen keresztül el lehet érni a megjelenítést végző ablakot is. Az oogLock egy a feltételes szinkronizációhoz szükséges mező, ezt később fejtem ki részletesebben (l. 5.10). A `finished` mező értéke `true` mindaddig, amíg a játék fut, és

`false`, amikor befejeződik. Ennek értékét adja vissza a `Game` interfész `isFinished` metódusa. A játéktérben definiált elemeket a `world`, a `menu` és a `player` mezők tartalmazzák. Mint azt már említettem, a játék jelenleg egy játékost kezel, amennyiben több játékost kellene kezelni, a `player` mező egy lista lenne. Az osztály belső osztályként definiál több `Listener` osztályt, amelyek a felhasználó interakciókat kezelik. Ezeket az osztályokat a megjelenítő `JFrame` objektumhoz kell hozzáadni, amely az `initGame` metódusban történik meg. Az osztály `run` metódusa egyetlen ciklust tartalmaz, amely addig fut, amíg a `finished` mező `true` értékű. A ciklusmag meghívja a `world` mező `act` metódusát, jelzi a megjelenítő szálnak (l. 5.10), hogy frissült a tartalom, és vár a következő frissítésig.

Végül ismertetném a `PathFinder` osztály által definiált `Listener` osztályokat. A `SelectAndCommandListener`, a `UnitAddRemoveListener`, és a `MapEditListener` a három menüpontnak megfelelő viselkedést implementálja. Mindhárom osztály rendelkezik egy `active` mezővel, amelyre minden esetben igaz, hogy a három osztály közül csak az egyiknél `true` az értéke. Ennek megvalósítása a `MenuListener` feladata, amely figyeli a felhasználó kattintásait, hogy történt-e kattintás valamelyik menüponton, és ennek megfelelően teszi aktívvá A fentebb említett három `Listener` osztály egyikét. A `SelectAndCommandListener` végzi a kijelöléskor szükséges rajzok megjelenítését. Emellett bal egérkattintáskor az egységek kijelölését, a kijelölt egységek nyilvántartását, és jobb egérkattintásra a kijelölt egységeket utasítja, hogy haladjanak a kattintás helyére. A `UnitAddRemoveListener` feladata, hogy bal kattintáskor egy egységet helyezzen el az egér pozíciójában, illetve jobb kattintáskor törölje az egér alatt található egységet – ha van. Végül a `MapEditListener` bal egérkattintáskor a térképen, a kattintás pozícióján lévő mezőt a nem járható `Wall` típusúra állítja, míg jobb kattintáskor járható, `Ground` típusú mezőt „tesz le” a térképre.

Az `ExitListener1` az alkalmazás ablakának bezárásakor a `finished` mező értékét `false`-ra állítja, ezáltal leállítja az alkalmazás futását. Az `ExitListener2` ugyanezt teszi, de az *Escape* billentyű leütésére. Az `AddKeyListener` a `SelectAnDCommandListener` működését változtatja meg annyiban, hogy a *SHIFT* billentyű nyomva tartása alatt a kijelölést bővíteni lehet, illetve jobb kattintás esetén az egységek csak az aktuális céljuk elérése után indulnak a kattintás helyére. A

`StopKeyListener` a jelenleg kiválasztott egységeket „megállítja”, ha a felhasználó leüti az *S* billentyűt. A `MapDragListener` feladata, hogy a felhasználó az egér középső gombjának nyomva tartásával tudja a térképet pozícionálni. A `ZoomListener` az egérgörgő használatakor a térképet nagyítja, illetve kicsinyíti, úgy, hogy a kurzor eközben mindig a térkép ugyanazon pontjára mutasson. Végül a `ResizeListener` szolgál arra, hogy ha a program futása közben átméretezzük az ablakot, akkor meghívja az ablak `reinitBackBuffer` metódusát, így generálva az alak új méretének megfelelő háttér-puffert. Az osztályok által használt billentyűparancsok, a *Controls* osztály segítségével módosíthatóak.

5.8 A kereső

A kereséshez szükséges osztályokat három csoportra lehet osztani. Az első csoport azokat az interfészeket és osztályokat tartalmazza, amelyekből létre lehet hozni egy általános keresőt. A második csoport az általános keresők implementációja. A harmadik csoport pedig a játékhoz legközelebb álló osztályok, amelyek csak létrehozzák a keresők számára szükséges objektumokat, és meghívják a megfelelő keresőt.

5.8.1 A kereséshez szükséges interfészek és osztályok

A `search.common` csomag interfészeket, és egy osztályt tartalmaz, melyek segítségével implementálni lehet egy általános, vagy akár egy specifikusabb keresőt. Ahhoz, hogy az osztályok közötti megfelelő kapcsolatokat definiáljam, nagy hangsúlyt fektettem a típusparaméterekre.

A csomag legegyszerűbb interfésze az `Operator` interfész. Az állapottér-reprezentációban definiált operátornak felel meg. Egyetlen metódust definiál, a `getCost` metódust, amely az operátor alkalmazási költségét adja vissza.

A State interfész megfelel egy állapotnak. Típusparamétere lehet bármely osztály, amely implementálja az `Operator` interfészt. A `getApplicableOperators` metódus visszaadja az állapotban alkalmazható operátorokat a típusparaméternek megfelelő objektumokként egy kollekcióban. Az `apply` metódus a paraméterül kapott operátort alkalmazza az állapotra, amely megfelel a típusparaméternek. Az állapotoknak klónozhatóaknak kell lenniük, hogy a tartalmazott állapot ne változhasson. Ezért az interfész kiterjeszti a `Cloneable` interfészt. Ugyanakkor a `clone` metódusnak mindenképpen ugyanolyan típusú objektumot kell visszaadnia, mint az interfészt kiterjesztő osztály. Ez a

Java nyelv korlátain kívül esik, de egy `State` interfészt kiterjesztő típusparaméterrel egyértelművé tehető a visszatérési típus. Az implementáláskor pedig ügyelni kell arra, hogy ez a típusparaméter az implementáló osztály típusa legyen. Ezen kívül a kereső megfelelő és gyors működéséhez szükséges implementálni a `hashCode` és `equals` metódusokat.

A `Heuristic` interfész felel meg a heurisztikának. Típusparaméterül egy `State` interfészt kiterjesztő objektumot kap. A `getHeuristicValue` metódusa paraméterül ennek megfelelő állapotot kap, és visszatérési értékként szolgáltatja az állapotban becsült heurisztikát. A heurisztika dönti el azt is, hogy egy állapot célállapot-e. Erre szolgál az `isGoal` metódus. Ezt a két vizsgálatot kényelmes egy osztályban kezelni, mert mindkét metódus a célállapotot vizsgálja

A `Node` a csomag egyetlen osztálya. Az osztály egy példánya az állapottér-gráf egy csomópontját jelöli. Típusparaméterként kap egy állapotot, egy operátort és egy heurisztikát is, és ennek megfelelő objektumokkal dolgozik. A `state` mező tartalmazza az állapotot, a `cost` mező a csomópontba vezető útköltséget, a `heuristicValue` mező a tartalmazott állapotban becsült heurisztika értékének felel meg. Az `operator` mező tartalmazza azt az `Operator` példányt, amely a tartalmazott állapotot képezte. Az `applicableOperators` mező azon operátorok kollekciója, amely alkalmazhatóak a tartalmazott állapotra. A `parent` mező pedig a csomópont szülőcsomópontjára mutat. A mezők mindegyike a konstruktorban állítódik be létrehozáskor. A két konstruktor közül az egyik csak az állapotot és a heurisztikát állítja be. Ez csak a gyökér-csomópont létrehozására szolgál. A másik konstruktor segítségével pedig a többi csomópontot lehet létrehozni. Ez a konstruktor privát, mert újabb csomópontok csak egy csomópont kiterjesztésével jöhetnek létre, ezt pedig az osztály `getSuccessors` metódusa végzi. A `Node` objektumokat az A*-kereső egy összköltség alapján növekvő sorrendbe rendezett listában tárolja. Ezért szükség van egy rendezettség definiálására. Ehhez implementáljuk a `Comparable` interfészt, melynek `compareTo` metódusa határozza meg a rendezettséget. Ebben az összköltséget vizsgálom, mivel leggyakrabban A*-algoritmust használunk. Ha viszont mégis ettől eltérő rendezettséget szeretnénk, egy származtatott osztály segítségével egyszerűen megtehetjük. Végül az osztály `getSuccessors` metódusának feladata a csomópont gyermekeinek legenerálása, és visszaadása egy kollekcióban. Ehhez egy ciklusban végigmegyek a csomópontban alkalmazható operátorokon, és minden elemnél létrehozok egy másolatot a csomópont által tartalmazott állapotról, majd alkalmazom rá az operátort, végül létrehozok egy csomópontot,

amely szülőállapota az aktuális csomópont, és az imént kapott állapotot tartalmazza. Ezeket teszem bele egy kollekcióba, amelyet a ciklus lefutása után visszaadok. Megjegyendő még, hogy az osztály `equals` metódusa csak a tartalmazott állapotok egyenlőségét vizsgálja.

A csomag utolsó interfésze a `Search` interfész, amely a keresőt definiálja. Egyedül a `search` metódust tartalmazza, amely egy kapott kiinduló-állapotból visszaadja a célállapot csomópontját, amelyből a szülők követésével visszakereshető a teljes útvonal. Az interfész funkciója abból a szempontból jelentősebb, hogy rögzíti a típusparamétereket, így az ezt kiterjesztő kereső osztály nem használhat inkompatibilis típusokat.

5.8.2 Az általános keresők

Az előző részben felsorolt interfészek lehetőséget nyújtanak arra, hogy implementálásukkal „testre szabhatjuk” a keresőket. Emellett még mindig megtehetjük, hogy csak általános célú keresőket írunk. Én ezt a megoldást választottam. A program az ablakolt hierarchikus kooperatív A*-algoritmust implementálja, amely – mint azt már korábban ismertettem (l. 4.2.5) – egy visszafelé haladó folytatható A*-algoritmust használ heurisztikaként, így ezt is implementálni kellett. Ezeket az általános keresőket a `search.whca` illetve a `search.rra` csomagok tartalmazzák.

5.8.2.1 Az A algoritmus implementálása*

Az alap A*-algoritmust a programom nem tartalmazza. Azért tartom mégis szükségesnek az ismertetését, mert a következő két kereső működése alig tér el az alap A*-algoritmustól, így azoknál csak a különbségekre térnék ki.

Az A*-algoritmus `search` metódusa egy kiinduló állapotot, és egy heurisztikát kap paraméterül, melyekből létrehozza a kiinduló állapotnak megfelelő `Node` objektumot. A kereső két listát használ, amelyek a nyílt, illetve a zárt csomópontokat tartalmazzák. A nyílt lista implementációja egy `HashMappedQueue`, amelyet korábban már ismertettem. Ez a csomópontok összköltsége alapján van rendezve, ami a `Node` objektumok természetes sorrendje. A zárt lista egy `HashMap`, melyben minden érték kulcsa önmaga. Ezután a kiinduló állapot csomópontját rögtön bele is tesszük a nyílt csomópontokat tartalmazó listába.

Ezután egy ciklus következik, melynek első lépése, hogy aktuális csomópontnak vesszük a nyílt lista első elemét, és ellenőrizzük, hogy létezik-e egyáltalán, és ha igen, akkor ellenőrzi, hogy célállapot-e. Bármelyik feltétel teljesülésekor leáll a ciklus. Ezután az aktuális csomópont minden gyermekére ellenőrizzük, hogy tartalmazza-e már valamelyik lista – az

egyenlőségvizsgálat a tartalmazott állapotokat fogja vizsgálni. Ha a vizsgálat egyezést talált, akkor ellenőrizzük, hogy a már meglévő elem költsége-e a kisebb, vagy az újonnan létrejötté. Ha az újonnan létrejött csomópont olcsóbb, akkor a már meglévő csomópontot töröljük a listából, az újat pedig a nyíltak közé tesszük. A gyermekek legenerálása után az aktuális csomópontot eltávolítjuk a nyílt listából, beletesszük a zárt listába, és a ciklus kezdődik elölről.

A ciklus lefutása után a metódus az aktuális csomópontot adja vissza.

5.8.2.2 A visszafelé haladó folytatható A-algoritmus implementálása*

Ez a kereső a `search.rra` csomagban található, melynek egyetlen osztálya az `RRASearch` osztály. Ez implementálja a `Search` interfészt, de a típusparaméterei még mindig általánosak, így a keresőbármilyen problémára alkalmazható.

Az osztály példányváltozóként tartalmazza a nyílt és a zárt listát, az aktuális csomópontot, és a heurisztikát is. Az osztály konstruktora a kapott állapotból egy csomópontot képez, és beleteszi a nyílt listába, tehát ez lesz a kezdőállapot, a célállapotot pedig a heurisztika definiálja, amit szintén paraméterül kap, és példányváltozóként tárol. A kereső `search` metódusa abban különbözik az alap A*-algoritmusétól, hogy a metódus elején ellenőrizzük, hogy a paraméterül kapott állapot szerepel-e már a zárt csomópontok között – ehhez létrehozunk belőle egy csomópontot. Ha már szerepel, akkor a metódus rögtön visszaadja a már meglévő csomópontot. A metódus többi része pedig nem változott.

5.8.2.3 Az ablakolt hierarchikus kooperatív A-algoritmus implementálása*

Ezt a keresőt a `search.whca` csomag tartalmazza. A kereső osztálya a `WHCASearch` osztály, amely kiterjeszti a `Search` interfészt. Az osztály a heurisztikát és az ablakméretet tartalmazza példányváltozóként, amelyek példányosításkor állítódnak be a konstruktorban kapott paraméterek alapján. A `search` metódusban a ciklus eleje kiegészül még egy feltétellel, amely azt vizsgálja, hogy a csomópont mélysége elérte-e már az ablakméretet. Ha igen, akkor a ciklus befejeződik.

A kereső implementálásának összetettségét az adja, hogy egy másik keresőn alapuló heurisztikát használ. A megfelelő működéshez így típusparaméterként nem csak azt az állapotot és operátort kell megadni, amelyre meghívjuk a keresőt, hanem azokat is, amellyel a visszafelé haladó folytatható A*-kereső dolgozik majd. Ezért típusparaméterként fel kell venni még egy állapotot, egy operátort, és egy heurisztikát is a heurisztika keresőjének

számára, hiszen a két kereső nem ugyanabban az állapottérben végzi a keresést. Az ablakolt hierarchikus kooperatív A*-kereső heurisztikájának típusát viszont nem kell paraméterként átadni, mivel az mindig ugyanazt a heurisztikát használja. Ez az `RRAHeuristic` osztály.

Az `RRAHeuristic` osztály a `Heuristic` interfészt implementálja. A heurisztika működéséhez használja a visszafelé haladó folytatható A*-keresőt, amelyet példányváltozóként tart nyílván. Konstruktorában kapja meg a célállapotot, amelyet szintén példányváltozóban tárol. Ugyanitt hozza létre a belső keresőt is, melynek példányosításához szüksége van egy kiinduló állapotra, és egy heurisztikára. A heurisztikát paraméterként kapja, de a kezdőállapottal más a helyzet. Tudjuk, hogy a két kereső állapotai nem ugyanazok, de van közöttük kapcsolat. A heurisztika célállapota tulajdonképpen megegyezik a kereső kiinduló-állapotával, csak épp más állapottérben vannak. Ezért létrehoztam egy interfészt, amely egyetlen metódust definiál, amely paraméterül egy állapotot kap, és visszatérési értéke szintén egy állapot, de ez lehet más típusú. Ez a `Converter` interfész, amely a két állapot típusát típusváltozókként kapja. Ezt az interfészt kiterjesztve, és a `convert` metódusát implementálva definiálhatjuk a kapcsolatot a két kereső állapotai között. A heurisztika konstruktora tehát egy ilyen `Converter` objektumot is kap, melynek segítségével létre tudja hozni a belső keresőt. Ezután pedig eltárolja a `Converter` objektumot is, mert a `getHeuristicValue` metódusban szintén használni kell. Az osztálynak így összesen négy típusparaméter van. Az állapot, amelyet a külső kereső használ, valamint az állapot, az operátor, és a heurisztika, amelyet a belső kereső használ.

5.8.3 A játék-specifikus kereső implementálása

Ezeket az osztályokat a `game.search` csomagban találjuk. Itt már csak annyi a dolgunk, hogy létrehozzuk a keresők által használt állapotokat, operátorokat, és heurisztikákat.

Az `game.search.rra` csomagban a visszafelé haladó keresőszámára szükséges osztályok találhatóak. Az `RRAOperator` osztály implementálja az operátort, és egyetlen `Move` típusú példányváltozója van. Ennek költségét pedig egyszerűen visszaadja a `getCost` metódusban.

Az `RRAState` implementálja a `State` interfészt, és az `RRAOperátort` használja operátorként. Példányváltozóként egy `Position` objektumot, és egy World objektumot tárol, ahol a `world` mindig referencia lesz. Az `apply` metódusában a tartalmazott pozícióra alkalmazza a paraméterül kapott operátorból nyert `Move` objektumot. A `getApplicableOperators` metódus a `Move` osztály értékein halad végig, és

mindegyiket becsomagolja egy `RRAOpertor` objektumba, a *`STAY`* példány kivételével, amennyiben az általa generált pozíción a térkép járható. Ezeket pedig egy kollekcióban adja vissza.

A `DiagonalHeuristic` osztály a `Heursitic` interfészt implementálja `RRAState` típusparaméterrel. Tárolja a célpozíciót, és a `getHeuristicValue` metódusban a kapott állapot által tartalmazott pozíció ettől mért átlós távolságát adja vissza.

A `game.search.whca` csomag az ablakolt kereső állapotát, és operátorát tartalmazza. Mint azt már említettem, a heurisztika megadására nincs szükség. A `WHCAOperátor` osztály tartalmazza az operátort, amely nagyon hasonlít az `RRAOperátor` osztályra, azzal a különbséggel, hogy ebben a költség mindig 1 lesz, mivel a heurisztika értékébe bele lesz kalkulálva a költség.

Az `WHCAState` osztály implementálja a `State` interfészt. Ez az állapot is tartalmaz egy pozíciót, és egy World referenciát, de emellett tartalmazza azt is, hogy hány lépésre van a kiinduló állapottól (`step`), ezzel implementálva az idő tényezőt. Végül tartalmaz egy változót, amely korlátozza a várakozások számát (`maxWait`). Ennek a segítségével megadható, hogy például az egység ne álljon meg 5-nél többször az útja során, így csökkenthetjük az állapotteret. Az `apply` metódus a pozíció megváltoztatásán kívül növeli a `step`, és csökkenti a `maxWait` példányváltozót. A `getApplicableOperators` metódusban szintén a `Move` osztály példányain megyünk végig, de ha a `maxWait` kisebb, mint 0, akkor annak alkalmazását mellőzzük. Ezen kívül azokból a példányokból képzünk operátort, amelyek alkalmazásával a kapott pozíció járható, és egység sem lesz rajta, amikor az egység odaér. Ezt a `step` mező és a `world.getUnitAfter` metódus segítségével tesszük, amely a foglaló-tábla implementációjának tekinthető.

A `game.search` csomag utolsó osztálya a `SearchManager` osztály, amely létrehozza a kapcsolatot az egység, és a kereső között, valamint elvégzi a keresések újrahasznosítását is. Konstruktora egy `World` és egy `Unit` objektumot kap, amelyet el is tárol példányváltozóként. Emellett egy `WHCASearch` típusú mezője is van. Egyetlen publikus metódust tartalmaz, a `getRoute` metódust, amely egy célállapotot kap paraméterként. Ez frissíti a tartalmazott kereső célállapotát, majd elindítja a keresést a tartalmazott Unit példány aktuális pozíciójából. A keresés eredményeként kapott csomópontból pedig előállítja a Pozíciók listáját, amit a metódus visszaad. A kereső célállapotának frissítését az

`updateSearch` metódus végzi, amely először ellenőrzi, hogy van-e egyáltalán kereső. Ha van, akkor ellenőrzi a célállapotát, és ha nem megfelelő, akkor létrehoz egy új heurisztikát a kapott célállapottal, és beállítja a keresőnek. Ha egyáltalán nincs kereső, akkor pedig létrehoz egyet, a kapott célállapottal létrehozott heurisztikával, és egy statikus konstansként tárolt alap ablakmérettel. Az új heurisztika létrehozásakor itt ellenőrzi, hogy a kapott `World` objektum kereső tartalékában létezik-e használható heurisztika. Ha igen, akkor azt használja, ha nem, akkor létrehoz egyet, amit bele is tesz a kereső tartalékba.

5.9 A `main` metódus

A program `main` metódusát a `PathFinder` osztály tartalmazza. Ez a metódus létrehoz egy `PathFinder` és egy `GameEngine` objektumot úgy, hogy a `GameEngine` konstruktorának a létrehozott `Pathfinder` objektumot adja át. Ezután létrehoz egy-egy szálat a két objektumhoz, és elindítja mindkettőt.

A `main` metódusban történik a parancssori argumentumok feldolgozása is. A program „-?”, vagy bármilyen érvénytelen paramétert kapva megjeleníti a program által felismert kapcsolókat, amelyekkel be lehet állítani a térkép méretét, típusát, és hogy ablakban, vagy teljes képernyőn fusson a program.

5.10 Feltételes szinkronizáció

A program megjelenítő szála a `Pathfinder` osztály `getOOGs` metódusában kéri el a játék kirajzolható objektumait. Ezt elég minden játékfrissítés alkalmával egyszer megtennie. Ezt úgy oldottam meg, hogy létrehoztam egy `oogLock` nevű példányváltozót, és a `getOOGs` metódusban meghívom a `wait` metódusát. Ez a metódus addig vár, amíg valahonnan meg nem hívódik az objektum `notify`, vagy `notifyAll` metódusa. Tehát annyit kell tenni, hogy a `run` metódus ciklusának végén meghívjuk az `oogLock.notifyAll` metódust.

6. Összefoglalás

A szakdolgozat segítségével az olvasó betekintést nyerhet a mesterséges intelligencia alapjaiba, illetve a megoldáskereső algoritmusok felépítésébe és működésébe. Emellett egy olyan program felépítésébe is, amely ezeket az ismereteket a gyakorlatban alkalmazza.

A szakdolgozathoz mellékelt program készítése közben folyamatosan fejlesztettem ismereteimet a Java nyelv terén, mélyebb ismereteket szereztem a szinkronizáció és grafika Javában való megvalósításának területén, valamint a generikus osztályok megfelelő implementálására is sok időt fordítottam, így kialakítva egy véleményem szerint átgondolt szerkezetet a kereső osztályoknak. Emellett kihívás volt egy összetett program megtervezése, és fejlesztése. A fejlesztés során számos megoldandó feladat állt elém, amelyeket igyekeztem a legegyszerűbb, és legjobb módon megoldani. Ugyan csak egyetlen útvonalkeresőt implementáltam, és csak néhányat mutattam be részletesen, a választás előtt megismertem és kipróbáltam több különböző útvonalkereső algoritmust, és útvonalkeresésben használt technikát. Ugyanakkor a dolgozat írása közben kibővítettem ismereteimet a mesterséges intelligencia megoldáskereső módszerekkel kapcsolatos területen.

A jövőben tervezem a program kibővítését, más útvonalkeresők implementálását, illetve szeretném kipróbálni saját ötleteimet is ezen a téren, amelyekhez a szakdolgozathoz készített program egy jó kiindulási alap.

7. Köszönetnyilvánítás

Szeretném megköszönni Espák Miklósnak, témavezetőmnek segítségét, és hasznos tanácsait, amelyek hozzájárultak a szakdolgozat, és a szakdolgozathoz készült program sikeres megírásához. Emellett szeretném megköszönni Dr. Várterész Magdának és Jeszenszky Péternek, akik a mesterséges intelligencia területéről elméleti, illetve gyakorlati tudásukat átadták az egyetemi oktatás során, ami nélkül a szakdolgozat nem jöhetett volna létre.

8. Irodalomjegyzék

[1] Várterész M., Kósa M.: Mesterséges intelligencia 1. Mobidiák könyvtár, 2003

[2] Stuart J. Russel, Peter Norvig: *Mesterséges intelligencia modern megközelítésben*. Panem, 2000.

[3] Holte, R. C., Perez, M. B., Zimmer, R. M.; MacDonald, A. J.: *Hierarchical A*: Searching abstraction hierarchies efficiently.* Proceedings of the 13th National Conference on Artificial Intelligence 1996 p.530-535.

[4] Sturtevant, N., Jansen, R.: An *Analysis of Map-Based Abstraction and Refinement*. Lecture Notes in Computer Science Vol. 4612/2007 p.344-358.

[5] Silver, D. *Cooperative pathfindig*. Proceedings of the 1st Conference on Artificial Intelligence and Interactive Digital Entertainment 2005, p.117-122.

[6] Gamma, E. és tsai: *Programtervezési minták*. Kiskapu, 2004.