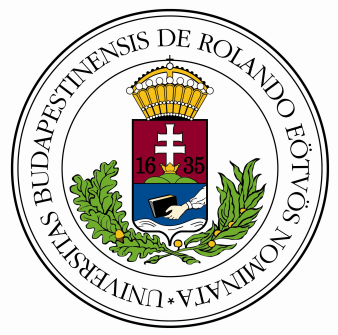
Eötvös Loránd Tudományegyetem
Matematika Intézet

Szabó Zsolt

Többszereplős útkeresési algoritmusok

BSc szakdolgozat

Témavezető: Bérczi Kristóf



ELTE Operációkutatási Tanszék

2014. Budapest

Tartalomjegyzék

1. fejezet

Bevezetés

A többszereplős útkeresési problémák egyre több helyen megjelennek, ahogy a technika fejlődik, és igény van a mesterséges intelligenciák együttműködésére. Robotok együttes mozgását, automatizált járművek ütközésmentes közlekedését, légiirányítási problémákat, számítógépes játékok egységeinek útkeresését egyaránt felírhatjuk többszereplős útkeresési feladatként, de más felhasználás is elképzelhető.

A feladatunk minden esetben az, hogy több szereplőnek (legyen az jármű, robot vagy játékbeli egység) egyszerre találjunk olyan utakat, melyeken végighaladva a szereplők nem ütköznek össze. Azt hogy ez a probléma most kezd igazán érdekessé válni jól mutatja, hogy az első igazi megoldási kísérlet kevesebb, mint tíz éve született [9], és a legtöbb megoldási módszer öt éves sincsen. Érdemes megjegyezni, hogy a többszereplős útkeresés problémája NP-teljes.

A dolgozatban először a 2.1 fejezetben szó lesz az A* keresésről [4], mivel a többszereplős algoritmusok is előszeretettel használnak A*-ot részfeladatok megoldására. Megemlítjük az A* két másik változatát is. Az egyik a változó gráfok módosulásai utáni újrakeresést gyorsítja [2], ezt a 2.2 fejezetben ismertetjük. A másik, a 2.3 fejezetben bemutatott algoritmus a gráf pontjait több szinten át csoportosítva egy hierarchikus gráfot hoz létre és azon keres [4].

Ezek után az említett többszereplős módszerekből mutatunk be néhányat, kezdve Silver [9] módszereivel a 3.1.1 alfejezetben, melyek az A* egyszereplős keresést alkalmazzák, majd rátérünk egy a gráf irányításának szabályozásával működő algoritmusra a 3.1.2 szakaszban [11], és ismertetjük a tolócserélő algoritmusnak elnevezett módszert a 3.1.3 alfejezetben [3]. Ezek a módszerek a szereplőket egyenként kezelik, és utólagos javításokkal kerülik el az ütközéseket.

Bemutatunk három olyan algoritmust is, amelyek a szereplőket együttesen kezelve próbálnak megoldást találni. Az első ilyen egy a szereplők lehetséges együttes helyzeteiből felépített gráfon keres az A* keresés segítségével és a 3.2.1 alfejezetben kerül bemutatásra [10]. A 3.2.2 alfejezetben ismertetett második módszer bizonyos útköltségek mellett keres ütközésmentes megoldást, miközben az útköltségeket változtatja [8], a harmadik pedig egy konfliktusokra koncentráló megoldás és annak egy továbbfejlesztett változata [6, 7]. Ezeket a 3.2.3 alfejezetben mutatjuk be.

2. fejezet

Az A* útkeresés és néhány változata

Ha útkeresési algoritmusokat szeretnénk vizsgálni, mindenképp érdemes említést tenni a talán legelterjedtebb algoritmusról, az A* útkeresésről. Rengeteg kereső rendszer használja olyannyira, hogy például a számítógépes játékokban szinte csak A*-ra épített technikákat alkalmaznak, és a többszereplős útkeresési algoritmusokban is előszeretettel oldanak meg részproblémákat A* segítségével. Népszerűségét többek között hatékonyságának, könnyű implementálhatóságának és sokrétű optimalizálási lehetőségeinek köszönheti. Az A*-ban rejlő lehetőségek érzékeltetése érdekében a következőkben röviden bemutatom magát az algoritmust, és két módosított változatát.

2.1. Az A* útkeresés

2.1.1. A feladat

A feladat azonos a jól ismert Dijkstra algoritmusnál kitűzöttel. Adott egy nemnegatív költségekkel élsúlyozott irányított gráf $G = (V, E)$, abban két csúcs s és t. Szeretnénk megtalálni az s-ből t-be vezető utak közül egy legolcsóbbat. A költségfüggvény legyen c. Az első fejezet további részében ezeket a jelöléseket használjuk.

2.1.2. Az algoritmus

Informálisan, az algoritmus nagyon hasonló Dijkstra algoritmusához, de a nyitott csúcsok közül nem a legkisebb jelenlegi költségűt választjuk, hanem egy heurisztikát használva azt, amelyik a legvalószínűbb, hogy egy legolcsóbb úthoz vezet. Ebből adódóan pontos heurisztikával nagyon hatékony módszert alkothatunk, de egy rossz heurisztika akár egy a Dijkstránál gyengébb algoritmust is szülhet.

Az A* esetén is iterációkban számolunk, de a csúcsoknál a jelenlegi költségen ($g(v)$) és megelőző csúcson kívül tárolunk egy becsült teljes költséget is ($f(v)$), ami a költség és a célcsúcstól vett, becsült távolság ($h(v)$) összege ($f(v) = g(v) + h(v)$). A $h(v)$ egy nemnegatív érték, amit a korábban említett heurisztika szerint határozunk meg, például az adott csúcs első érintésekor, és az algoritmus folyamán nem változik. Egy iterációban megvizsgáljuk az aktuális csúcs (u) kimenő élein (e) vett szomszédait (v)

és minden ilyen v-re a Dijkstránál megszokottak szerint frissítjük $g(v)$-t és a megelőző csúcs bejegyzést, ha $g(v) > g(u) + c(e)$. Ekkor frissül $f(v)$ is. A következő vizsgálandó csúcs a nyitott csúcsok közül a minimális f-értékű lesz (ami nagyon sokszor különbözik a minimális g-értékűtől).

Mivel az iterációk során a következő csúcsot f alapján választjuk megtörténhet, hogy már lezárt csúcshoz találunk új, rövidebb utat, ha egy korábbi becslés azt jó választásnak hitte, pedig valójában nem volt az. Ilyen esetekben ezt a csúcsot visszatesszük a nyitott csúcsok közé, mivel az ő új értéke alapján újra kell értékelnünk a kimenő élszomszédait is.

Felmerül a kérdés, hogy mikor érjen véget az algoritmus. Ha addig fut amíg a célcsúcs a nyitott listán a legkisebb f-értékű lesz, akkor a korábban írottak szerint nem garantált az optimalitás, viszont ha addig fut amíg a célcsúcs a nyitottak közt a legkisebb g-értékű lesz, akkor megmutatható, hogy az algoritmus ugyanannyi ideig fut, mint a Dijkstra, tehát ez a változat értelmetlen. A heurisztika különféle megválasztásával szerencsére ez irányítható, és biztosítható az optimalitás, vagy éppen megengedhetünk nem-optimális megoldást is, ha lényegesebb a gyorsaság. Sok A* implementáció a még gyorsabb futás érdekében a célcsúcs elérésekor rögtön leáll, és még azt sem várja meg, hogy a nyitott listán a legkisebb legyen.

2.1.3. Heurisztikák

Minél pontosabb egy heurisztika, annál gyorsabb algoritmust eredményez a használata. Ugyanakkor a pontosabb, okosabb becslés kiszámítása általában tovább tarthat. Emellett az A* keresés máshogy viselkedik attól függően, hogy a heurisztika alulról vagy felülről becsüli a céltól vett távolságot.

Alulbecslő heurisztika esetén az algoritmus hosszabb ideig fog futni, mert ekkor g jobban érvényesül f-ben, mint h, és ezért az algoritmus viszonylag lassan tart majd t felé, így tovább fut. Ugyanakkor ilyen esetben akkor is legrövidebb utat kapunk, ha a nyitottak közt legkisebb f-értékre alapozva állunk meg, mivel akkor a heurisztikánk minden távolságot alulról becsül, ezért minden v-re a nyitott listából $g(v) + h(v) < g(v) + c(vt)$ fennáll, azaz t nem lesz a legkisebb f-értékű a nyitott listán amíg a hozzá vezető út nem a lehető legrövidebb. Tehát ha az optimalitás fontosabb a sebességnél, akkor alulbecslő heurisztikát kell alkalmazni.

Felülbecslő heurisztikával az optimalitás nem biztosított, de az algoritmus hamarabb eléri t-t, mint az alulbecslő, mivel ilyenkor hamarabb értékelődnek ki a t-hez feltehetően közelebb eső csúcsok. Látható, hogy ha a heurisztika mindig legfeljebb x-el becsli felül a valódi távolságot, akkor a talált st út is legfeljebb x-el lesz hosszabb a valódi legrövidebbnél. Tehát ha ezt az x-et kis értéken tudjuk tartani, akkor az algoritmus majdnem pontos marad, ami bizonyos felhasználások esetén elegendő.

Nézzünk meg három egyszerű, játékokban használt heurisztikát.

Euklideszi távolság

A dolgozatban tekintett példák legtöbbjében a megalkotott gráf egy térkép vagy valós, esetleg virtuális terület leképezése, ezért van értelme a tényleges távolságon alapuló heurisztikáknak. Nyilvánvalóan

az Euklideszi távolság legfeljebb akkora, mint a valódi, így alulbecslő heurisztikát ad. Ahogy sejthető, nem minden gráfon működik egyformán jól. Nézzünk olyan gráfokat, melyek csúcsai egy valamekkora négyzetháló négyzetei és az élek a szomszédos (él vagy csúcs szerint) négyzetek közt vannak (nyolc-kapcsolt rács). Bizonyos négyzetek lehetnek átjárhatatlanok, így képezhetünk akadályokat, falakat. Ha egy ilyen térképen alig vannak akadályok, az euklideszi távolsággal képzett heurisztika nagyon jól teljesít, de egy kisebb szobákból álló térképen lehet, hogy szinte minden négyzetet érint, mivel túlságosan alulbecsli a távolságokat.

Manhattan távolság

A Manhattan távolság használatának például négy-kapcsolt rács (az átlós mozgás nem megengedett) esetén lehet értelme. Úgy kaphatjuk meg két mező Manhattan távolságát, hogy koordinátánként vesszük a távolságukat és ezeket összegezzük ($d = \sum_{i=1}^{n} |x_i - y_i|$). Négy-kapcsolt rács esetén ez is alulbecslő heurisztika.

Klaszter heurisztika

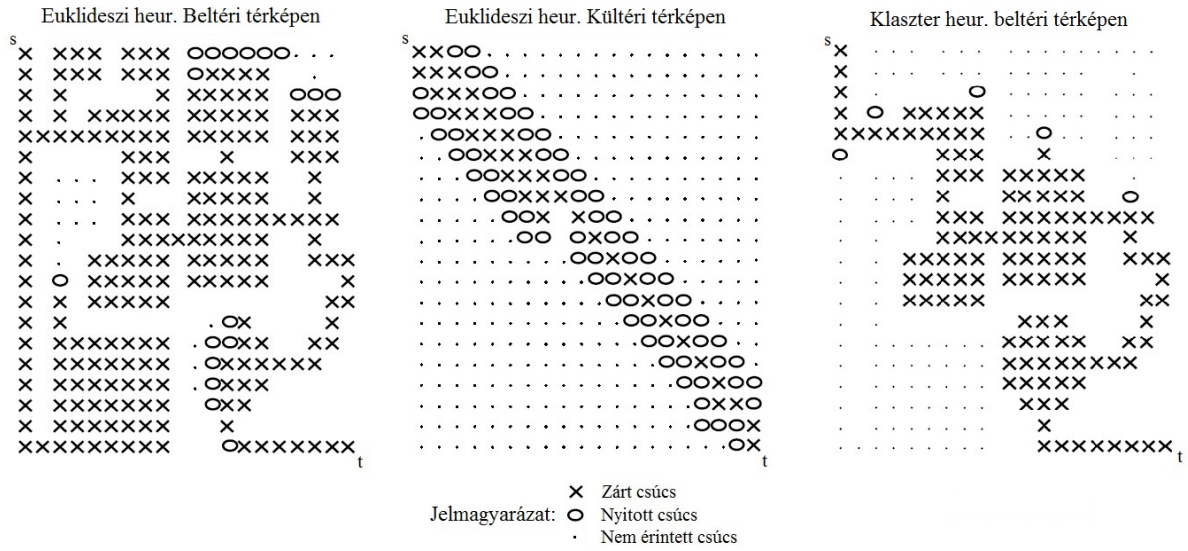
A klaszter heurisztika (cluster heuristic) az egy csomóban levő csúcsok összefogásával ér el jobb eredményt. A klaszterek közti távolságokat egy külön algoritmus előre megbecsüli és eltárolja (valós implementációkor egyszerre csak a klaszterek egy kellően kis részhalmazára). Ha s és t egy csoporton belül van, akkor euklideszi távolságot használ, ha külön csoportban, akkor pedig az eltárolt becslést. Kisebb szobákból álló térkép és általában kis csoportokat tartalmazó gráfok esetén ez a módszer határozottan jobban teljesít, mint az euklideszi távolságon alapuló, de a sok klaszter hosszú előfeldolgozást és nagy tárhelyet igényel. Hátránya, hogy mivel egy klaszterben minden csúcs h-értéke azonos, sokszor az elért klaszterek majdnem minden pontját érinti, bár ez klaszteren belüli külön útkereséssel javítható.

2.2. Újratervező A*

Mesterséges intelligenciáknak sokszor kell változó körülményekhez igazodni. Megtörténhet, hogy menet közben akadályok jelennek meg, utak válnak átjárhatatlanná vagy nehezebben járhatóvá, és ehhez alkalmazkodnunk kell.

2.2.1. A módszer

Az újratervező A* (Lifelong Planning A*, ezentúl LPA*) az A* algoritmust kombinálja egy DynamicSWSF-FP nevű algoritmussal [5]. Olyan véges gráfokra használható, melyeknek az élhosszai változhatnak. Ezáltal modellezhető a csúcsok eltűnése és újabbak bekerülése is. Az algoritmus folyamatosan újrakeresi a legrövidebb utat s és t csúcsok közt. A legelső keresés egy sima A*, ami az azonos f-értékű csúcsok közül a kisebb g-értékűt választja, de a későbbi keresések felhasználják a korábbi keresések eredményeit, amikor lehet, így ezek a keresések már sokkal gyorsabban futhatnak (lényegében



2.1. ábra. Heurisztikák összehasonlítása (s-ből t-be történt az útkeresés) [4]

ez a DynamicSWSF-FP). Ha a gráf nagy és gyakran, nagy mértékben változik, vagy tervezés során történik a változás, akkor még az így javított keresésünk sem lesz sokkal jobb, mintha mindig teljesen újra keresnénk, de a valóságban legtöbbször csak kis változások tapasztalhatók.

Jelölések

$succ(u) \subseteq V$ az u csúcs gyerekeit, $pred(u) \subseteq V$ az u őseit jelöli. $g^*(u)$ az s-ből u-ba menő legrövidebb út hossza, azaz:

$$g^*(u) = \begin{cases} 0 & \text{ha u=s} \\ min_{v \in pred(u)}(g^*(v) + c(v,u)) & \text{különben} \end{cases}$$

Az LPA* az A*-ban használt g-érték mellett fenntart egy rhs (right-hand side) értéket is a csúcsoknál, ami egy lépést előre tekint:

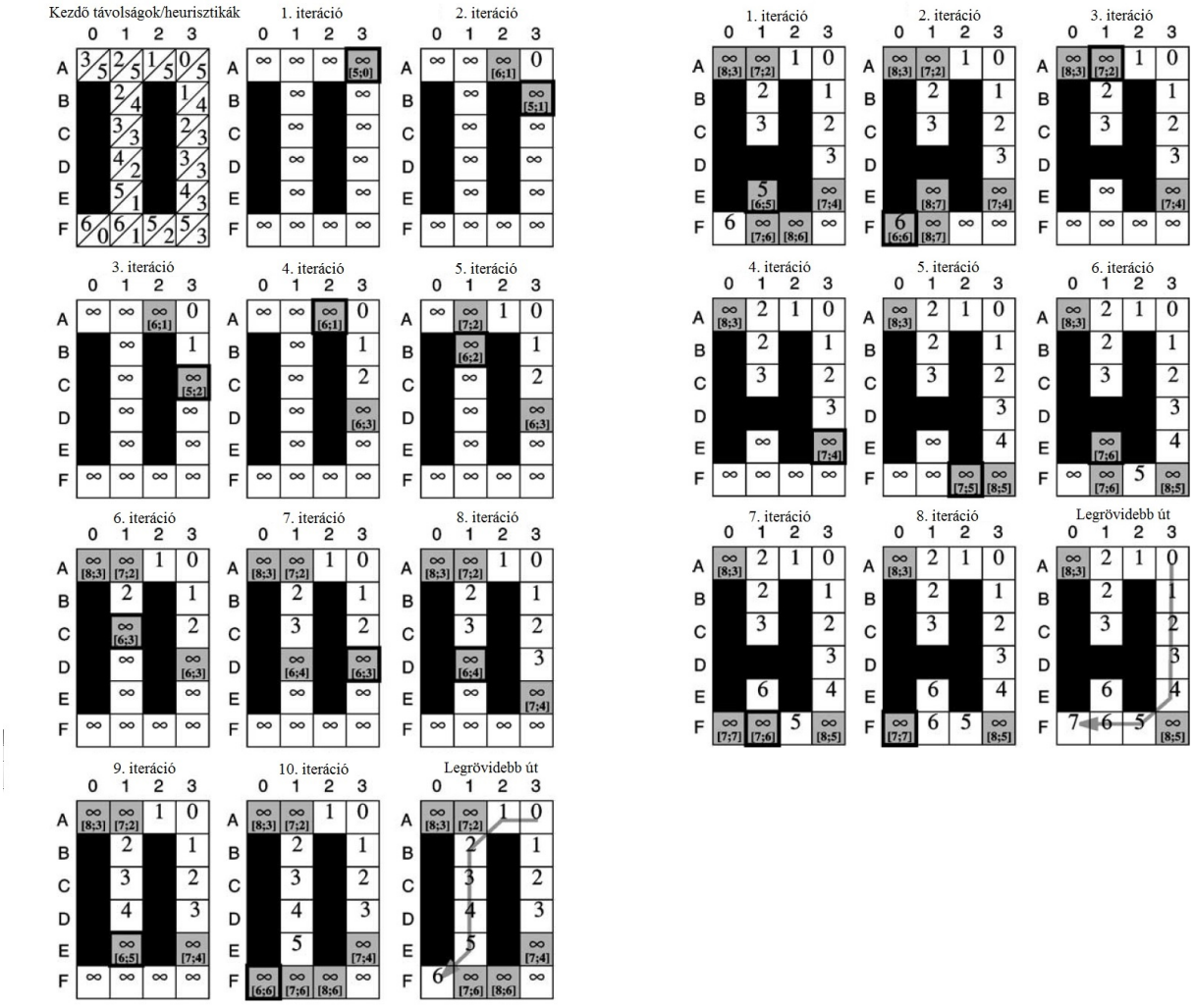
$$rhs(u) = \begin{cases} 0 & \text{ha u=s} \\ min_{v \in pred(u)}(g(v) + c(v,u)) & \text{különben} \end{cases}$$

A könnyebb átláthatóság érdekében az LPA* algoritmust nyolc-kapcsolt rácson vett példával szemléltetem majd, ahol a négyzetek átjárhatóból átjárhatatlanná válhatnak, vagy fordítva. Két szomszédos négyzet közt az áthaladás ennek megfelelően 1 vagy végtelen költségű. Az A*-hoz használt heurisztika pedig legyen a két négyzet x és y koordinátája közül a nagyobb.

Nyolc-kapcsolt rácsok esetén $pred(u) = \{u$ 8 szomszédja$\}$.

Ebből a felírásból következik, hogy ha $g(u) = rhs(u)$, akkor u-ra már kiszámoltuk a g^* értéket, azaz $g(u) = g^*(u)$. Ha $g(u) = rhs(u)$, akkor u-t lokálisan konzisztensnek nevezzük. Látható, hogy ha minden csúcs lokálisan konzisztens, akkor minden csúcsra $g(u) = g^*(u)$, azaz $g(u)$ az s-ből u-ba vezető legrövidebb út hossza. Ha valamelyik csúcs blokkolódik, akkor először ennek a csúcsnak a szomszédaira újraszámoljuk az rhs-értéket, és ha ez különbözik az aktuális g-értéktől, akkor a csúcs már nem konzisztens. Az inkonzisztenssé vált csúcsok adják az újratervezés kiindulópontját.

Az LPA* nem törekszik minden csúcs konzisztenssé tételére, ehelyett A*-szerűen a heurisztika alapján azon csúcsokat részesíti előnyben, amik a legrövidebb st megtalálását feltehetően elősegítik. Az A*-tól eltérően az LPA* nem tart fent külön listát a zárt csúcsoknak, mert a g és rhs-értékek összehasonlítása elég a zártság meghatározásához.



2.2. ábra. Az LPA* algoritmus futása először (bal), és változtatás után (jobb) [2]

A 2.2 ábrán bal oldalon nyomon követhetjük az LPA* első futását. Szögletes zárójelben $k_1(s) =$

$min(g(s), rhs(s)) + h(s)$ és $k_2(s) = min(g(s), rhs(s))$. A következő kiértékelendő csúcsot a k_1-érték alapján választjuk. Az ábrán jobb oldalon a D1 cella blokkolása utáni futást figyelhetjük meg. Látható, hogy először lefut az A*, majd blokkolás után először a változtatott cella elérhető szomszédait vizsgáljuk meg, és az A*-szerű keresést nem kell teljesen elölről kezdeni.

2.2.2. Az algoritmus

```
procedure KulcsotSzamol(s) {return [min(g(s), rhs(s)) + h(s); min(g(s), rhs(s))];}
procedure Inicializal()
    U = Ø;
    for all (v ∈ V) do rhs(v) = g(v) = ∞;
    rhs(s) = 0;
   U.Betesz(s, [h(s); 0]);
procedure CsucsotFrissit(u)
    if (u ≠ s) then rhs(u) = min_{v∈pred(u)}(g(v) + c(v, u));
    if (u ∈ U) then U.Torol(u);
    if (g(u) ≠ rhs(u)) then U.Betesz(u,KulcsotSzamol(u));
procedure LegrovUtatSzamol()
while U.LegnagyobbKulcs() < KulcsotSzamol(t) or rhs(t) ≠ g(t) do
    u = U.Kivesz();
    if (g(u) > rhs(u)) then
        g(u) = rhs(u);
        for all (v ∈ succ(u)) do CsucsotFrissit(v);
    else
        g(u) = ∞;
        for all (v ∈ succ(u) ∪ {u}) do CsucsotFrissit(v);
    end
end
procedure Main()
    Inicializal();
   forever
      LegrovUtatSzamol();
      Élhosszváltozásra vár;
      for all ((u, v) megváltozott költségű irányított él) do
        Élköltséget frissít;
        CsucsotFrissit(v);
```

2.3. ábra. Az Ujratervező A* algoritmus [2]

A Main() függvény először meghívja az Inicializal()-t, ami minden csucs g és rhs-értékét beállítja végtelenre, kivéve s-ét, aminek az rhs-értéke 0 kezdetben. Így s inkonzisztens lesz és bekerül a nyitott csúcsok listájába. Ez biztosítja, hogy a LegrovUtatSzamol() első futása az A*-ot hajtja végre. Tényleges megvalósításkor elég minden csúcs kezdőértékeit az első elérésekor beállítani. Ha egy élköltség megváltozik, a CsucsotFrissit() függvény frissíti az rhs, k_1 és k_2 értékeket a változtatás által érintett csúcsokon, és ha valamelyik inkonzisztenssé válik, akkor LegrovUtatSzamol() újraszámolja a legrövidebb utat az inkonzisztens csúcsok kiértékelésével kezdve.

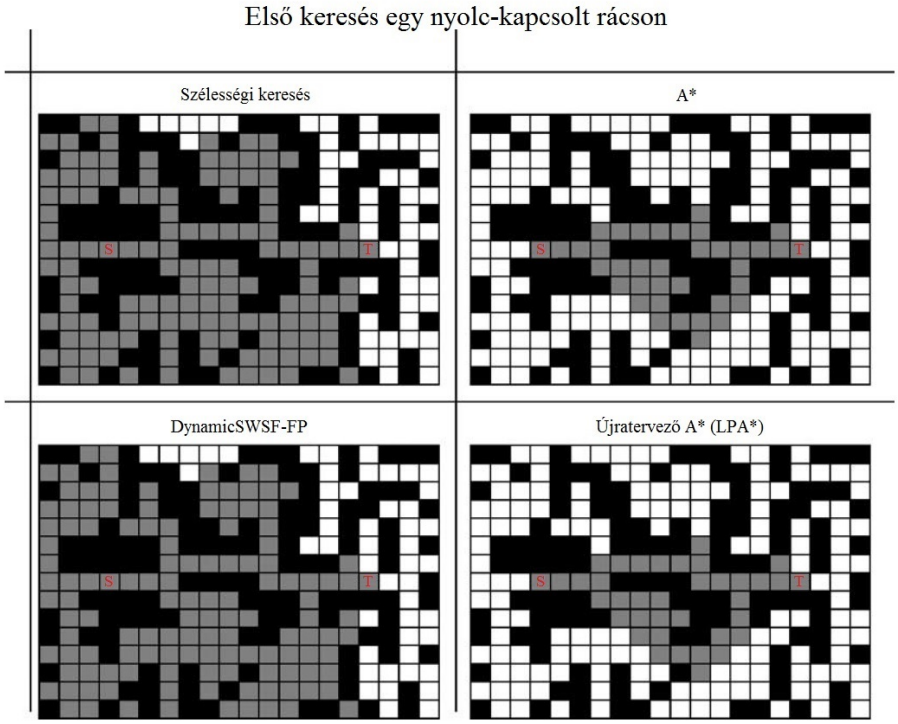
Egy u csúcs lokálisan túlkonzisztens, ha $g(u) > rhs(u)$, és alulkonzisztens, ha $g(u) < rhs(u)$. LegrovUtatSzamol() túlkonzisztens csúcs g-értékét az rhs-értékére állítja, ezzel konzisztenssé téve azt, míg alulkonzisztensét végtelenre, amitől már biztosan nem lesz alulkonzisztens. Ezek a változtatások befolyásolhatják $succ(u)$ elemeit, így azoknak is ellenőrizni kell a konzisztenciáját. Az LPA* addig fut, míg t lokálisan konzisztenssé válik és a következő kiértékelendő csúcs k_1-értéke nem kisebb mint t-é. Ha a keresés végén $g(t) = \infty$ akkor nincs véges hosszú út t-be, egyébként pedig visszakövethető a legrövidebb út t-től s-ig.

Az LPA* nyilvánvalóan jól használható a szállítási problémákhoz, robotok útkereséséhez, ahol a jármű útját egy folyamatosan változó környezetben kell meghatároznunk, vagy kommunikációs hálózatokban, ahol a jelenleg leginkább elterjedt módszer Dijkstra algoritmusával teljesen elölről keresi újra a legjobb utat.

2.2.3. Egy tesztfutás

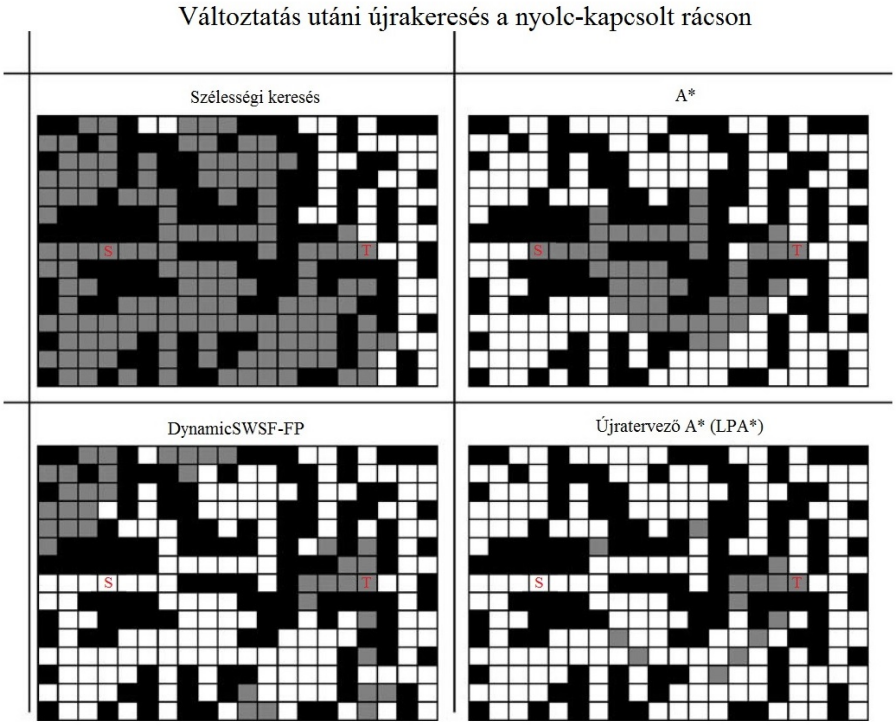
A 2.4 és 2.5 ábrákon egy véletlen generált térképen hasonlítjuk össze az újratervező A* keresést néhány korábbi algoritmussal. A 2.4 ábrán látható a keresések legelső futása.

Látszik, hogy az újratervező A*-nál ez megegyezik egy sima A* kereséssel. A 2.5 ábrán viszont már jól látható a különbség, ami egy mező blokkolása után az újrakeresésben mutatkozik. Míg az A* újrakeres s-től indulva, az újratervező változat láthatóan célorientáltan csak



2.4. ábra. Algoritmusok összehasonlítása változtatás előtt [2]

olyan mezőket vizsgál újra, amiknek változott az s-től vett távolsága.



2.5. ábra. Algoritmusok összehasonlítása változtatás után [2]

2.3. Hierarchikus A*

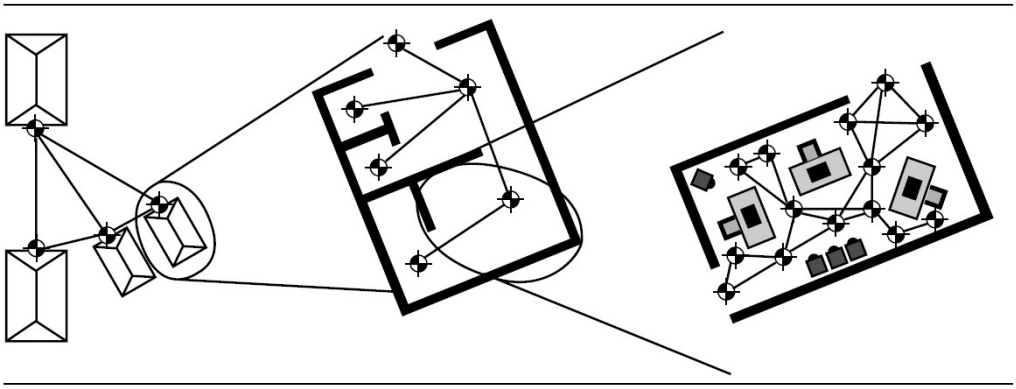
A hierarchikus útkeresés nagyon hasonlóan működik ahhoz, ahogy az emberek megtervezik az útjukat két pont között. Először tervezünk egy átfogó utat, aminek minden állomása egy valamivel részletesebb útiterv az adott régióra, és így tovább. Például ha el szeretnék jutni Amerikába, az átfogó tervem lehet, hogy kimegyek a reptérre, felszállok a gépre, Amerikában leszállok és elmegyek a célhelyre. De ezután részletesen meg kell terveznem hogyan jutok a reptérre, vagy a reptérről a célpontba.

2.3.1. A módszer

Az útkeresés minden szinten külön A*-gal megvalósítható, de alakítanunk kell a gráf adatstruktúrán. Ezt az A* Klaszter heurisztikájában látotthoz hasonló csoportosítással tesszük meg. Egy játék térképén például egy szoba belső pontjait vehetjük egybe, így az egész szobát egy pont jelképezi a magasabb szinten. Azután az épület így kialakult szobaszintű pontjait vehetjük egybe, majd az épületkomplexum épületeit, és így tovább. Ezzel kialakul egy hierarchikus gráf. Ahhoz, hogy ezen végre tudjuk hajtani az útkeresést, ki kell tudnunk bontani a pontokat az általuk reprezentált gráfrészletté.

Csúcsokon kívül természetesen éleknek is kell lenni a csoportok közt. Ha két klaszter közt lehetséges

az átjárás, akkor azokat magasabb szinten is össze kell kötni, és az élköltségnek mutatni kell az áthaladás nehézségét. Ezt elérhetjük manuálisan vagy kiszámolhatjuk az alacsonyabb szintű élek segítségével.



2.6. ábra. Hierarchikus elrendezés (1. példa) [4]

Az élszámolásra egységesen jó módszert találni nem egyszerű, mert például K klaszterből L-be átérni könnyebb vagy nehezebb lehet attól függően, hogy K-ba honnan léptünk be. A csoportok kialakításakor már érdemes ennek a problémának a minimalizálására törekedni, de ez sem oldható meg könnyen. Ugyanakkor van néhány sűrűn használt heurisztika a klaszterközi élhosszok számítására.

Az első a minimális távolság, ami értelemszerűen a két csoport közti legrövidebb élt veszi a csoportok közti távolságnak. Ez legtöbb esetben túl rövid, de ettől függetlenül lehet értelme ezzel számolni.

A második az úgynevezett maximin távolság, ami kiszámolja (általában valamilyen útkereséssel) minden bejövő élre a kimenő élhez vezető út költségét és ezek közül a legnagyobbat hozzáadja a kimenő él költségéhez, és ezt veszi csoportok közti távolságnak. Ez általában túl sok, de az elsőhöz hasonlóan nem feltétlenül értelmetlen.

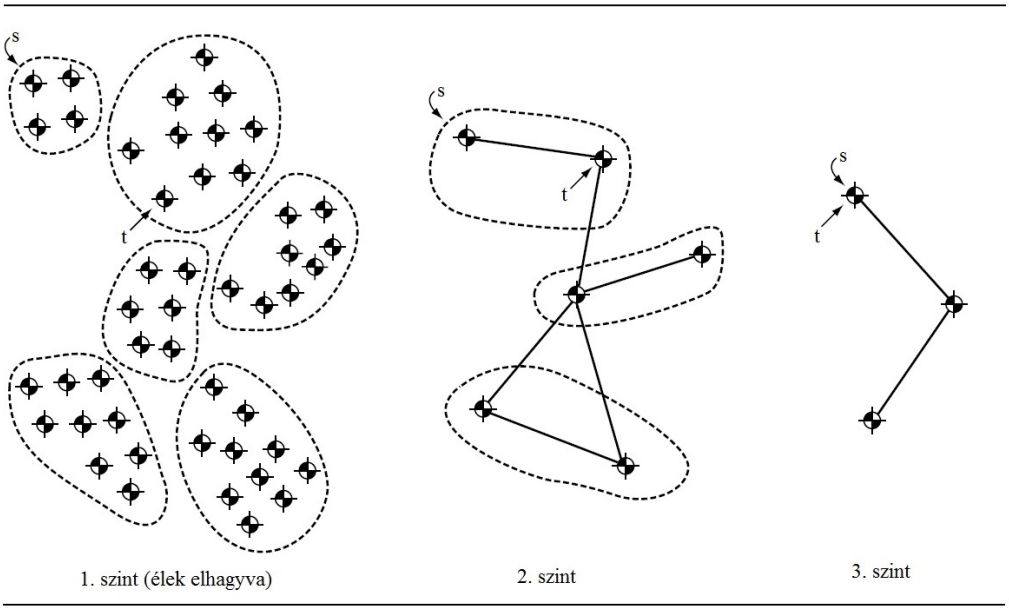
A harmadik, átlagos távolságot figyelő módszer az előzőhöz hasonlít, de az értékek átlagát veszi, nem a maximumát, így egy közepes becslést ad a valódi távolságokra.

A hierarchikus A* által talált út nagyban függhet a használt klaszterezési és élszámítási módszertől.

2.3.2. Az algoritmus

Mivel a gráfunk több szintből áll, először ki kell találnunk honnan kezdjük az algoritmust. A legjobb választás talán a legmagasabb olyan szint, ahol a kezdő és végcsúcs (s és t) már nincs egy klaszterben, hiszen ennél magasabb szinten a megoldás triviális, alacsonyabb szintről kezdve pedig felesleges többletmunkát hajtanánk végre. A 2.7 ábrán az útkeresést ennek megfelelően a 2 szinten fogjuk kezdeni.

Amikor a kezdő szinten megterveztük az utat, először csak az első pontját fejtjük ki, mivel a járművünk vagy karakterünk mozgása szempontjából az a legfontosabb. A végpontot átállítjuk a kifejtett klaszterbeli kilépési pontra és így tervezzük meg az alacsonyabb szintű utat. Ennek az útnak megint kifejtjük az első pontját, átállítjuk a végpontot, és így tovább a legalacsonyabb szintig. Így megkapjuk a részletes út elejét, amit a karakterünk már ténylegesen be tud járni. Például ha szobák sorozatán



2.7. ábra. Hierarchikus elrendezés (2. példa) [4]

akarunk átjutni, először megtervezzük a magas szintű utat: előszoba → nappali → hálószoba, majd csak az előszobában keresünk utat a bejárati ajtótól a nappaliba vezető ajtóig. Bizonyos esetekben érdemes lehet nem csak a legelső pontokat kifejteni, hanem rögtön az első párat.

Miután a karakterünk elérte a megtervezett útrészlet végét, az algoritmus újra meghívódik és megtervezi a következő részletet. Ha olyan megvalósítást választunk ahol tároljuk a már megtervezett utakat, ott a magasabb szintű utakat nem is mindig kell újratervezni, amivel tovább gyorsíthatjuk az algoritmust.

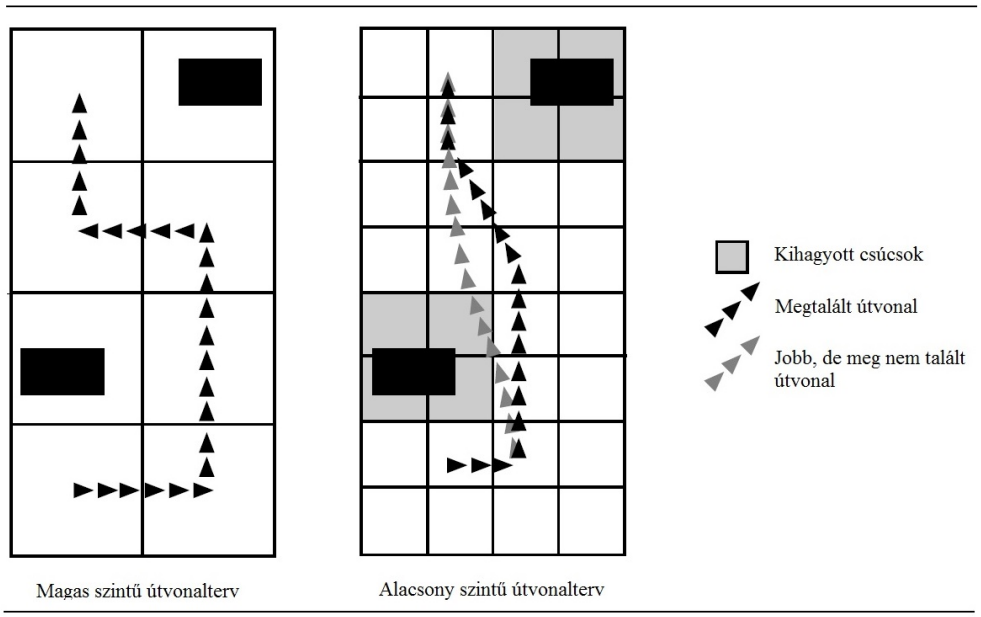
A hierarchikus útkeresés úgy is megoldható, hogy a végcsúcs helyzetét nem változtatjuk, és rögtön a teljes utat keressük meg, viszont mivel nem akarjuk az egész gráfot végignézni, csak azokat a csúcsokat vesszük bele a keresésbe, melyek részei a magasabb szinten talált út pontjainak.

2.3.3. Az algoritmus értékelése, gyengeségei

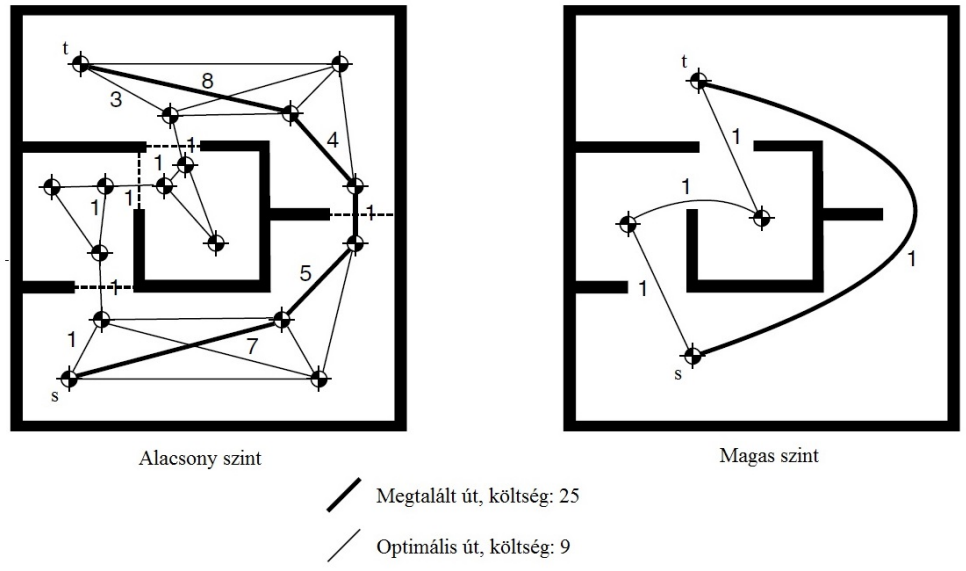
A hierarchikus A* algoritmus nem minden esetben gyorsabb, mint a sima A*, sőt egy rosszul megkonstruált hierarchikus gráf használatával még lassabb is lehet, de nagy gráfok esetén, jó klaszterezés mellett rendkívüli sebességnövekedés érhető el, mivel a kis részekre használt A*-oknak sokkal kevesebb iterációt kell végrehajtaniuk. A talált megoldás általában nem optimális, hiszen ez is egyfajta heurisztikán alapul, és ennek megfelelően bizonyos helyzetekben jobban, máskor rosszabbul teljesít.

Megtörténhet akár az is, hogy a magas szintű gráf élhosszainak valamilyen meghatározása miatt nem találunk meg egy rövidebb utat két pont közt, például ahogy a 2.9 ábrán látszik. A minimum heurisztikát használtuk, így mivel bármely két szoba közt a legrövidebb távolság 1, az algoritmus azt az utat választja ahol a legkevesebb szobán kell átmenni, pedig nem az a legjobb. Található olyan helyzet

is ahol az említett 3 módszer egyike sem találja meg a legjobb utat, és megeshet, hogy nagyon nehéz olyat találni, aminek sikerül, de ez ismert hátránya a heurisztikus módszereknek.



2.8. ábra. A hierarchikus keresés hosszú utat talál, mert kizár cellákat [4]



2.9. ábra. A hierarchikus keresés hosszú utat talál, mert a heurisztikánk nem megfelelő [4]

3. fejezet

Többszereplős útkeresési algoritmusok

Többszereplős útkeresési (Multi-Agent Pathfinding (MAPF)) problémák esetén adott egy $G = (V, E)$ gráfon k szereplő (például egységek egy stratégiai játékban), mindegyikhez adott egy start- (s_i) és egy célcsúcs (t_i) és a feladat megtervezni a szereplők útját a kezdőhelyüktől a céljukig úgy, hogy az útjuk során ne ütközzenek, azaz ne kerüljenek azonos időben azonos csúcsra és ne haladjanak át egy élen egyszerre egymással szemben. A dolgozatban erre a feladatra ismertetünk megoldó algoritmusokat, de léteznek egyéb változatok is.

Érdekes lehet az az eset, amelyikben nem fontos, hogy melyik szereplő melyik célcsúcsra érkezik meg, csak az, hogy mindegyik különböző célban végezze. Egy másik esetben, például valamiféle evakuációs szimuláció esetén, az is lehetséges, hogy több szereplő azonos célba jut. Ekkor a szereplő eltűnik a gráfról, ha elérte a célját.

Az úgynevezett kooperatív útkeresés feladat esetén a szereplők teljes ismerettel rendelkeznek a többi szereplő által megtervezett útról és egymáshoz igazodva próbálnak célba érni. Nem-kooperatív esetben nem tudnak semmit egymás útvonaláról és egy harmadik, versengő útkeresésnek (antagonistic pathfinding) elnevezett változatban pedig a többit akadályozva kell a szereplőknek elérni céljukat. Mi most csak a kooperatív változattal fogunk foglalkozni.

Bonyolíthatjuk a feladatot, ha megengedünk különböző méretű és sebességű szereplőket, vagy ha a gráf élein való áthaladást nem pillanatszerűnek vesszük, hanem az él hosszának megfelelő ideig a szereplőt az élen haladóként ábrázoljuk, de az élek hossz szerinti mennyiségű egységnyi szakaszra osztásával ezt elkerülhetjük, bár a felosztott élen való egymással szemben haladás problémáját valahogy kezelni kell. Ebben a dolgozatban általában négy- vagy nyolc-kapcsolt rácsokon fogunk keresni, ahol az élek egységnyi hosszúak és az áthaladás pillanatszerű.

Ilyen és ehhez hasonló térképábrázolási módszerek jellemzőek a valós idejű stratégiai játékokra, amikben sok egység együttes mozgását kell megtervezni. Erre tökéletesen alkalmas a többszereplős útkeresés, de alkalmazzák például robotikában, légiirányításban és teljesen vagy részben automatizált járművek útkereséséhez is.

Természetesen minél jobb megoldást szeretnénk adni a feladatra, de az sem teljesen mindegy, hogy milyen értéket szeretnénk a lehető legkisebbre csökkenteni. Nézhetjük például az utolsóként célba

érő szereplő lépésszámát (azaz célbaérési idejét), vagy útvonalhosszát (ha megengedünk várakozás műveletet, akkor ez különbözhet a lépésszámtól), vagy az összes szereplő együttes útvonalhosszát.

Megoldási módszerek közt két nagy csoportot szokás megkülönböztetni. Az egyik az elosztott (decoupled), a másik a centralizált (coupled) megközelítés.

Megjegyzés

A hivatkozott cikkekben szó esik egy másik megkülönböztetési szempontról is, aszerint, hogy a tényleges megvalósítás egy processzorra íródik, vagy megengedünk több processzoros párhuzamosítást. Ezeket centralizált (centralized) és elosztott (distributed) nevekkel látják el, de mi ezeket a korábban írt és később részletezett értelemben fogjuk használni. A dolgozatban egy processzoron futtatott, párhuzamosítás nélküli módszereket vizsgálunk.

3.1. Elosztott módszerek

Elosztott módszerekben a szereplők egyenként megtervezik saját útjukat, és különböző technikákkal utólag vagy még tervezés közben az ütközéseket eltüntetik. Hibája ezen módszereknek, hogy a megtalált megoldás általában nem lesz optimális az ütközések feloldásakor bekövetkező útvonalváltozások miatt, és megtörténhet, hogy nem is kapunk teljes megoldást, azaz nem minden egység ér célba. Cserébe ezek az algoritmusok gyorsabbak, mint a centralizáltak, főleg nagy számú szereplő esetén.

A következő három alfejezetben megnézünk néhány elosztott megközelítésen alapuló próbálkozást. Először szó lesz a Lokálisan Javító, a Kooperatív, a Hierarchikus Kooperatív és az Ablakos Hierarchikus Kooperatív A* algoritmusokról (Local Repair A*, Cooperative A*, Hierarchical Cooperative A*, Windowed Hierarchical Cooperative A*), majd egy a haladási irányok megkötésével variáló módszerről a Folyamszabályozott tervezésről (Flow Annotation Replanning), végül pedig a Toló-Cserélő (Push and Swap) algoritmust mutatjuk be, ami a nevében szereplő két műveletet ismételgetve viszi helyükre egyesével az egységeket.

3.1.1. A lokálisan javító A*-tól az ablakos hierarchikus kooperatív A*-ig

Lokálisan javító A*

A Lokálisan javító A* (Local Repair A*) tulajdonképpen nem egy algoritmust, hanem egy algoritmuscsaládot jelöl, amelyekben a szereplők külön-külön megtervezik saját optimális útjaikat sima A*-al a többi szereplőt figyelmen kívül hagyva, elkezdik követni az így megtalált utat és ha valamelyik lépésük ütközéshez vezetne, akkor újraterveznek az ütközési pontot kihagyva.

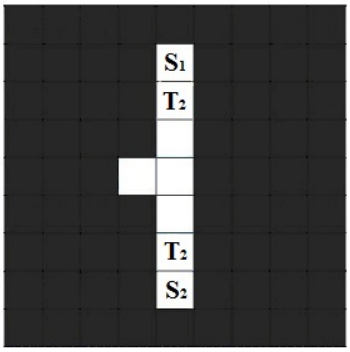
Ez a módszer könnyen vezethet ciklusokhoz, mivel az újratervezések egymástól függetlenül történnek és a várakozást (egy szereplő helyben marad) itt nem engedjük meg, mint lehetséges lépést. Szűk átjáróknál például jelentkezhet ez a probléma vagy tarthat túlságosan sokáig az áthaladás.

Többféle javítást is kitaláltak ezeknek az elkerülésére, például felállítható a szereplők közt egy elsőbbségi sorrend, vagy az újratervezésekkor egyre növő véletlen zajt vihetünk az adott szereplő heurisztikájába, így egy idő után egy másik úton akar majd haladni, mint az, akivel folyamatosan ciklikusan ütközik.

Kooperatív A*

A Kooperatív A* (Cooperative A*) algoritmusban már megjelenik a várakozás, mint lépés. Minden szereplő önállóan keresi az útját, de az algoritmus fenntart egy három dimenziós foglalási táblázatot, amiben saját útjuk megtervezése után a szereplők jelölik, hogy mikor hol fognak tartózkodni a gráfon és azokat a mezőket arra az időpillanatra többi szereplő már nem foglalhatja le. Például egy kétdimenziós rácshoz könnyen implementálható egy foglalási tábla egy egyszerű 3 dimenziós rácsként (az idő a 3. dimenzió), sőt mivel a méretéhez képest várhatóan kicsi lesz a lefoglalt mezők száma, akár egy hash-táblával is megoldható az ábrázolás, ahol a kulcsok a mezőket ábrázoló (a, b, t) számhármasok.

Ennek a módszernek is megvan az a hibája, ami a többi egyesével tervező algoritmusnak, hogy bizonyos problémákat nem tud megoldani, nem biztos, hogy minden szereplő célba jut, mert például két szereplő blokkolhatja egymást, mint a 3.1 ábrán. Itt is bevezethető a szereplők közti prioritási sorrend, ami valamelyest javít az ilyen helyzeteken.



3.1. ábra. Egy eset, amit a KA* nem tud megoldani [9]

Hierarchikus Kooperatív A*

A Hierarchikus Kooperatív A*-ot (Hierarchical Cooperative A*) a használt heurisztika teszi külön említésre méltóvá. Kooperatív A* útkereséskor bármilyen heurisztikát használhatunk. Bizonyos módszerekben a 2.3 fejezetben bemutatotthoz hasonló Hierarchikus A* kereséssel kiszámolnak egy legrövidebb utat a célig, és ezt használják heurisztikaként. HKA* esetén egy az időt és a foglalási táblát figyelmen kívül hagyó absztrakt térben való kereséssel találunk legrövidebb utat a megfelelő célcsúcsba. Ez minden szereplőnél egy jó alulbecslő heurisztika és pontatlansága csak attól függ, hogy az adott szereplő hány másikkal találkozik az útja során, és emiatt mennyire kell eltérnie saját megtervezett útjától.

Hierarchikus A* kereséseknél fontos, hogy mennyire tudjuk újrahasználni a már kiszámolt adatokat. Erre ad jó megoldást a $ReverseResumableA^*(RRA^*)$, amit a HKA* is használ. Az RRA* egy visszafele kereső A*-ot futtat Manhattan távolságheurisztikával, az adott szereplő céljától indulva, de nem a szereplő startpozíciójának, hanem egy adott u csúcsnak a kifejtéséig fut, és tárolja a kiszámolt távolságokat. Az RRA* kérés-alapon fut vagy folytatja futását. Ha egy u csúcshoz szeretnénk heurisztikát kérni, az RRA* megnézi, hogy a zárt listájában van-e, és ha igen, akkor visszaadja a már kiszámolt távolságot, ha nem akkor pedig folytatja a visszafele kereső A*-ot, amíg az az u csúcsot ki nem fejti.

Ablakos Hierarchikus Kooperatív A*

Az eddig ismertetett technikákban a szereplők megállnak, miután elérték céljukat, így ha az például egy szűk átjáróban van, akadályozhatják vagy teljesen blokkolhatják a többiek előrehaladását. Egy jó algoritmusban célba érés után is együtt kellene működniük a szereplőknek. Ha bevezetünk valami fix rangsort a szereplők közt, az is vezethet problémákhoz. Ezt a rangsor folyamatos változtatásával javíthatjuk, úgy hogy biztosítjuk, hogy minden szereplő néha egy időre a legmagasabb prioritással rendelkezzen. Gondot okozhat az is, hogy a fenti algoritmusok mind teljes utakat számolnak egy akár nagyon nagy három dimenziós térben. Egyszereplős keresésekben meg lehet tenni, hogy a tervezés és a kivitelezés egy időben történik. Ilyesmire törekszik az Ablakos Hierarchikus Kooperatív A* (Windowed Hierarchical Cooperative A*) is.

A módszer lényege, hogy a kooperatív keresést mindig csak a következő w lépésre hajtjuk végre, tehát fenntartunk egy w méretű ablakot, és azt toljuk arrébb, azaz bizonyos idő után (például $w/2$ lépésenként) mindig összesen w lépésig továbbkeresünk kooperatív módon. Az ablakon kívül csak az absztrakt téren keresünk, azaz a többi szereplőt figyelmen kívül hagyva. A w lépésen túli absztrakt keresés szükséges, hogy a szereplők biztosan a céljuk felé tartsanak.

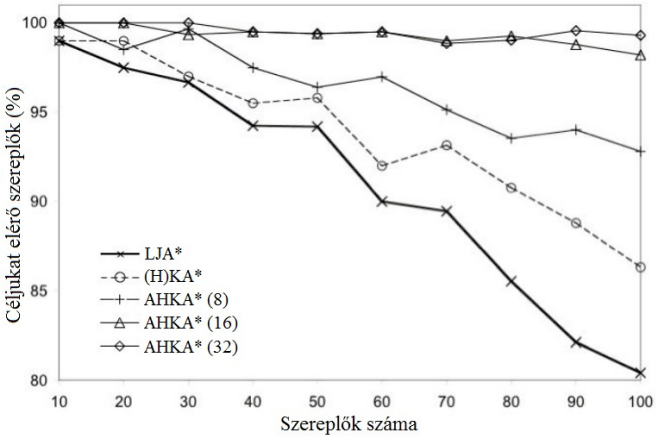
A w lépésben elért csúcsokhoz felveszünk egy speciális fiktív lezáró élt, ami az adott csúcsból közvetlenül a célcsúcsba vezet és hossza a két csúcs absztrakt tér-beli távolsága. Ezzel a trükkel a keresés leszűkül egy w-lépéses ablakra.

Ez az ablakos keresés folytatható a szereplő célba érése után is, és így tovább folyhat az együttműködés. Az RRA*-ban kapott eredmények újrahasználhatók az egymást követő ablakok számolásakor. Ehhez minden szereplőnek tárolni kell a saját nyitott és zárt csúcs listáját, illetve minden szereplőhöz futtatni kell egy kezdeti RRA*-ot a céljától a kezdőhelyéig, majd szükség esetén ezt az RRA*-ot lehet folytatni. Hogy a folytatott keresés konzisztens maradjon korábbi futásával, a keresésnek továbbra is a kezdőpozíció felé kell történnie, ami ronthat a sebességen, ha a szereplő nagyon eltérül az eredeti legjobb útjától, de az RRA* korábbi számításainak megtartásából adódó megtakarítás általában bőven ellensúlyozza ezt.

Teszteredmények

A teszteredmények nem saját mérésekből származnak. Több különböző mérés és részletesebb magyarázat található a [9] cikkben.

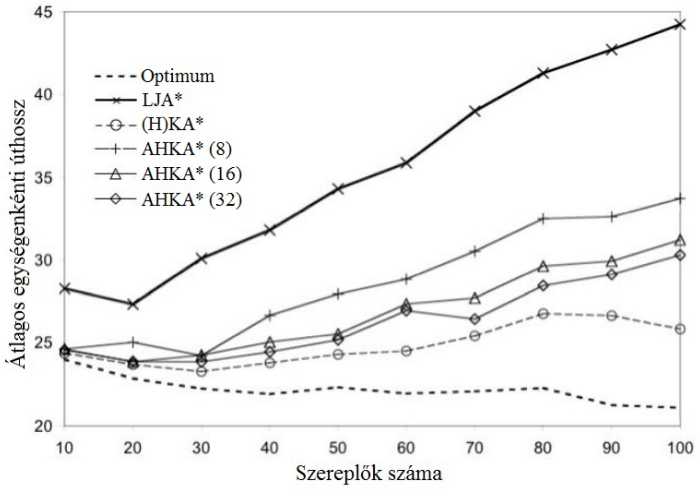
A tesztek 32x32-es négy-kapcsolt rácsokon futottak. A mezők 20%-át véletlenszerűen kiválasztva blokkolták. Egy szereplőt akkor vettek sikeresnek, ha az 100 lépésen belül célba ért és nem ütközött senkivel út



3.2. ábra. Algoritmusok célbaért szereplők száma szerinti összehasonlítása [9]

közben. A 3.2 grafikonon látható a különböző algoritmusokban célba ért szereplők száma százalékosan a szereplők számának függvényében.

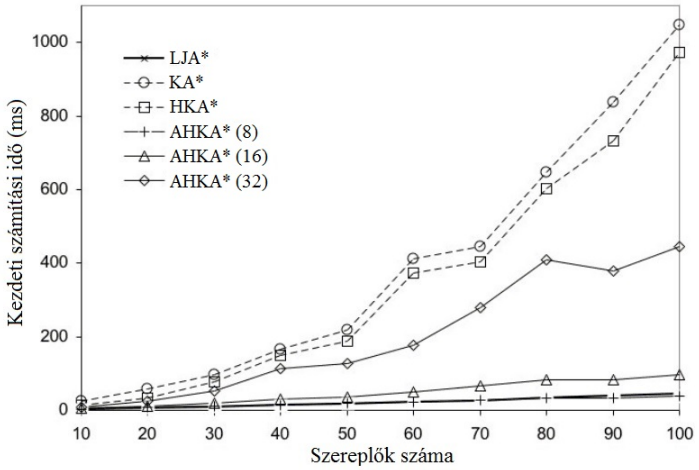
Láthatóan a lokálisan javító A* 40-50 szereplőnél már rosszul teljesít, mert nem kezeli jól a szűk átjárókat, míg az AHKA* (16) még 100 szereplővel is alig 2%-os hibával dolgozik, mivel a szűk helyzetekben kooperatív lévén elléptethet szereplőket mások útjából. (A zárójelbeli szám az ablakméretet jelöli.)



3.3. ábra. Algoritmusok átlagos úthossz szerinti összehasonlítása [9]

Megmérték a szereplők átlagos megtett úthosszát is (3.3 grafikon). Ebben a várakozás lépések is benne vannak a kooperatív algoritmusoknál. Jelölik az optimális átlagos úthosszt is (ha más szereplők nem lennének). Itt is látszik, hogy bár kevés szereplőnél minden algoritmus jól teljesít, ahogy nő az egységek száma, a lokálisan javító A* nagyon eltávolodik az optimumtól, és a többi algoritmus is valamennyire eltér a legjobb úttól, ahogy egyre többször találkoznak a szereplők.

Érdekes mutató még az algoritmusok számolással töltött ideje. Itt a kezdeti számolás idejét mutatjuk be a 3.4 ábrán. Ami észrevehető, hogy a lokálisan javító algoritmus nagyon gyors, hasonlóan a kis ablakkal futtatott ablakos módszerek is jól teljesítenek, mert a szereplők kezdeti útkeresésének nagy része a többi egységet nem veszi figyelembe. A nagy ablakos módszer valamivel lassabb, és sok szereplőre a teljesen előre tervező algoritmusok (KA*, HKA*) pedig már nagyságrendekkel rosszabbak, mert azok az egész utat megtervezik előre, viszont ennek megfelelően később nincs szükségük újraszámításokra.



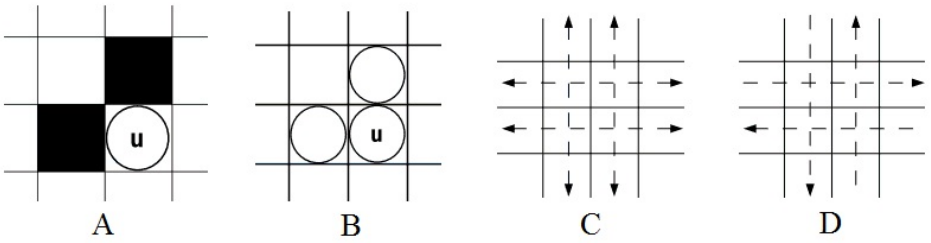
3.4. ábra. Algoritmusok számolási idő szerinti összehasonlítása [9]

3.1.2. Folyamszabályozott tervezés

Ahogy korábban szó volt róla, különböző terepek, térképek reprezentációjára sokszor használnak rácsot, így nem meglepő, hogy született ilyen gráfokra specializált többszereplős útkeresési algoritmus. Ilyen a Folyamszabályozott Tervezés (FSZT, Flow Annotation Replanning). Lényege, hogy a térképen a haladási irányok megkötésével nagyban csökkenti a szemtől-szembe ütközések valószínűségét és a csúcsok gyerekszámát is. A rács sorain felváltva csak jobbra vagy csak balra lehet haladni, az oszlopokon pedig felváltva csak fel vagy le. Néhány további szabállyal biztosítjuk, hogy bármely két mező közt, ha az eredeti rácson létezett út, akkor az újon is legyen mindkét irányban. Végrehajtás közben kialakulhat holtpont, amikor szereplők körben egymásra várnak. Ezeket egy heurisztikus metódussal lokálisan próbáljuk megoldani.

Probléma definíció

Nyolc-kapcsolt rácsban gondolkodunk. Egy mező akkor üres, ha nem tartózkodik rajta szereplő. A lépés pillanatszerű. Egy szereplő akkor léphet egy szomszédos mezőre, ha az üres és senki más nem fog oda lépni a következő lépésben. Átlós lépéshez kell még, hogy a két mező közül, amiken át kell haladni legalább az egyik szabad legyen (3.5/A,B ábra). Minden szereplőt egyforma méretűnek és sebességűnek tételezünk fel. A térkép széleit úgy tekintjük, mintha a rácsot kívülről blokkolt mezők vennék körül.

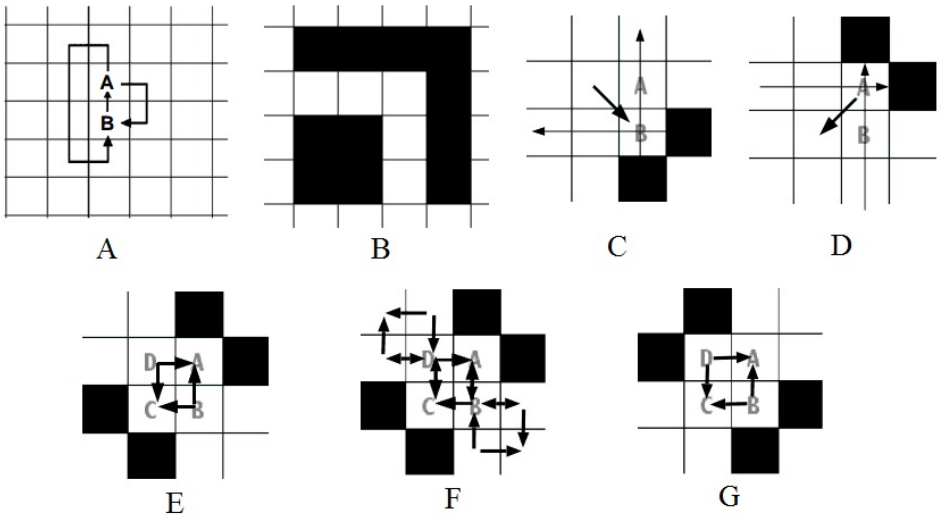


3.5. ábra. Átjárhatatlan átlók (A, B) és irányítások szabályozás előtt (C) és után (D) [11]

A gráf átalakítása

Először egy előfeldolgozó lépés keretében a rácsot egy folyamszabályozott keresőgráffá alakítjuk, amiben az addig irányítatlan éleket irányítottakra cseréljük, a korábban leírt iránymegkötéseknek megfelelően (3.5/C→D). Kezdetben átlós élek nem szerepelnek a gráfban, de a mezők közti kétirányú kapcsolatot biztosító lépések során majd bekerülhetnek.

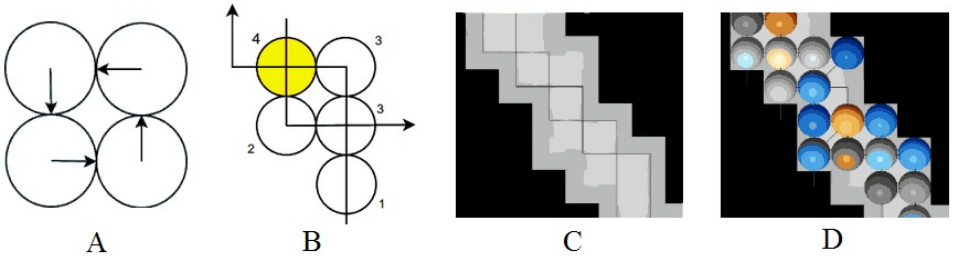
Így, ahogy a 3.6/A ábrán is látszik, megtörténhet, hogy két mező közt az egyik irányban már csak hosszabb utakon tudunk átjutni, vagy ha ezek blokkolva vannak, akkor kezdetben csak egy még hosszabb kerülő létezik, vagy egyáltalán nincs is. Ilyenkor egy újabb él behúzásával megoldhatjuk a problémát, például itt az A és B közti él kétirányúvá tételével. Ugyanezt tehetjük az alagutak (3.6/B ábra) mezőit összekötő élekkel is. Sarkokban kialakulhatnak forrás vagy nyelő mezők (3.6/C, D ábra).



3.6. ábra. A folyamszabályozás speciális helyzetei: Hosszabb visszautak (A), Alagút (B), Forrás (C), Nyelő (D), Két nyelő egymással szemben és feloldása (E, F), Két forrás egymással szemben (G) [11]

Ilyenkor egy megfelelő irányú átlós éllel biztosíthatjuk a be- vagy kijutás lehetőségét. Megtörténhet, hogy két forrás vagy két nyelő egymással szemben van (3.6/E, G ábra) és az átlós él behúzása ilyenkor értelmetlen. Ezeket a helyzeteket néhány már meglevő él kétirányúsításával oldhatjuk meg (3.6/F ábra). Már meglevő élet sosem törlünk a gráfból a kapcsolat-helyrehozás során.

Ezek a lokális javítások nem veszik figyelembe a térkép teljes elrendezését, így továbbra is maradhatnak kellemetlen helyzetek, például a 3.7/C ábrán látható szűk lépcsős folyosó. Itt is érdemes lenne bizonyos éleket, akár átlósokat is, kétirányúvá tenni, de ezt az algoritmus jelenlegi formájában nem veszi észre.



3.7. ábra. Holtpont (A), Holtpont feloldása (B), Szűk folyosó (C, D) [11]

Úttervezés

Az eddig leírtakat biztosítandó, az algoritmus egy előfeldolgozó lépéssel kezdődik ami a gráfot megfelelően átalakítja. Ezután minden szereplőnek A*-al, a többieket figyelmen kívül hagyva kiszámolunk

egy kezdeti útvonalat a céljához. Ha ezt mindenkinek megtettük, elindul az útvonaltervek végrehajtása.

Bár az irányok megszabásával erősen csökkentjük a szemtől-szembe ütközés esélyét, oldalra ütközések még mindig könnyen előfordulhatnak, ha a szereplők nem működnek együtt. Ez az együttműködés két fő pillérre épül. Először is arra törekszünk, hogy minél kevesebb újratervezésre legyen szükség. Másodszor, ha mégis újra kell terveznünk, azt megpróbáljuk rövid, lokális számításokkal megoldani. Az ütközések elkerülésének jó módja, ha megpróbáljuk minél egyenesebben tartani a szereplők útját. Ennek érdekében az A* szokásos $f = g + h$ szerinti irányválasztásánál az azonos f-értékű csúcsokból inkább azt választjuk, amelyik tartja a jelenlegi haladási irányt.

Az együttműködést itt is egy foglalási rendszerrel oldjuk meg, de nem tartunk fent egy nagy foglalási táblát. Minden szereplő csak k lépésre előre foglalja le az útján használt mezőket. Egy szereplő csak akkor kezdhet mozogni, amikor lefoglalta a következő k mezőjét (kivéve ha már k-nál közelebb van a céljához), és a k lépés megtétele után kezdheti el az újabb foglalást. Előfordulhat, hogy egy csúcsot sokszor egymás után többen is le szeretnének foglalni. Ilyenkor lépésenként felváltva a vízszintes majd függőleges lépést részesítjük előnyben, így egyenlően kezeljük a két irányt (mint egy forgalmas közúti kereszteződésben).

Ha egy szereplő elérte célját, folyamatosan várakozik. Ha ezzel blokkolná egy másik szereplő útját, akkor ellép a céljáról és A*-al megtervezi a visszautat, de közben a másik egység el tud haladni. A másik eset, amikor újratervezésre van szükség, a holtpontok kialakulása.

Mivel a teljes újratervezések helyett lokális javításokkal oldjuk meg az ütközéses és holtpont helyzeteket, a kezdeti A* kereséseknél használt nyitott és zárt csúcslistákat eldobhatjuk minden egység keresés végén, így sok memóriát spórolhatunk.

Holtpontok

Mivel a jelenlegi problémadefiníció szerint csak akkor engedünk meg egy lépést, ha a mező, ahova lépnénk üres, kialakulhatnak várakozási körök, például mint a 3.7/A ábrán látható. Ha sok egység van a térképen, ilyen helyzetek könnyen kialakulhatnak, és vezethetnek még nagyobb holtpontokhoz, mivel a beragadt egységek feltarthatják a mögöttük érkezőket is. Ennek elkerülése érdekében minél hamarabb fel kell fedeznünk, ha valahol holtpont alakult ki.

A holtpontkereső algoritmust olyankor indítjuk el, amikor egy szereplő nem tudja folytatni az útját, mert azt blokkolja egy másik, nem a célján elhelyezkedő szereplő. Az ellenőrzés során felépül az egymásra váró szereplők egy lánca, és akkor ér véget az ellenőrzés, ha találunk egy egységet, aki üres mezőre vár, ekkor nincsen holtpont, vagy találunk egyet, aki egy már ellenőrzött egységre vár, ekkor holtponthoz jutottunk.

A holtpont feloldásához kiválasztunk egy kritikus egységet és azt eltérítjük az eredetileg tervezett útjától. Jó esetben ezzel elérjük, hogy más egységek mozogni tudjanak, ami még több egységnek biztosít mozgási lehetőséget, és a holtpont megszűnik. Ennek ellenőrzésére, a kritikus egység elmozgatása után újrafuttatjuk a holtpontkeresést és ha valahol még mindig holtpont van, akkor egy újabb kritikus egységet választunk.

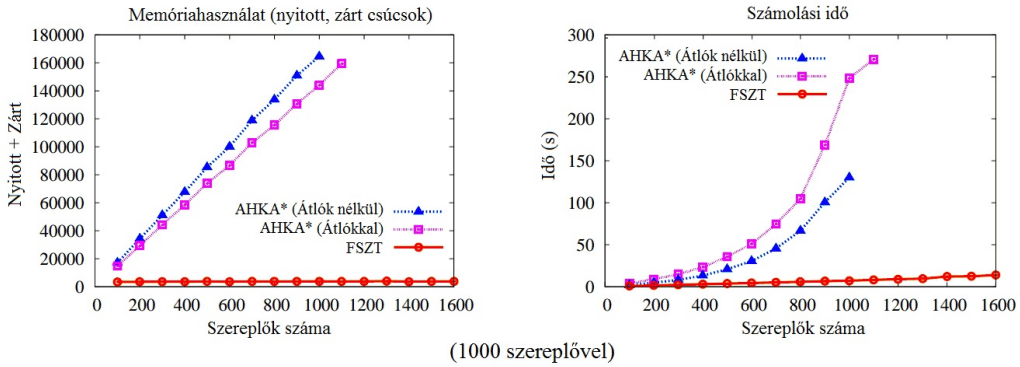
A kritikus egység kiválasztása útsűrűség heurisztika alapján történik. Egy csúcs útsűrűsége a rajta áthaladó tervezett utak száma, amit folyamatosan figyelnünk és frissítenünk kell, ha valamelyik szereplő újratervezésre kényszerül. Mivel várhatóan a legsűrűbb csúcs felszabadításával érjük el a legnagyobb hatást a holtpontra nézve, mindig próbáljuk a kritikus egységnek a legnagyobb útsűrűségű csúcson levőt választani (3.7/B ábra).

Ezt az egységet kiléptetjük a helyéről, majd keresünk neki egy utat ugyanoda ahonnan eljött. Mint korábban láttuk, ez egy legalább 3 lépéses út lesz, ami alatt a többi holtpontbeli egység tovább tud haladni. Ez a módszer hatékonynak bizonyult a gyakorlatban, kivéve szűk átjárókban (3.7/D ábra), ahol a kritikus egység elmozdítása nem feltétlenül okoz jelentős változást, így az áthaladás lassú lehet.

Teszteredmények

A tesztek nem saját mérések eredményeit mutatják. Bővebb tesztek és részletesebb magyarázatok a [11] cikkben találhatók.

A teszteket a Baldur's Gate című számítógépes játék térképein futtatták, ebből itt egynek az eredményei láthatók a 3.8 ábrán. Az FSZT algoritmust az AHKA* (8) átlós lépéseket megengedő és nem megengedő két változatával hasonlították össze. Az bal oldali grafikonról leolvasható, hogy az AHKA* változatok memóriahasználata lineáris a szereplők számában, míg az FSZT jóformán konstans, hiszen minden szereplő kezdeti A* számolása után eldobjuk a zárt és nyitott csúcslistákat. A jobb oldali grafikon a teljes számolási időt mutatja, ami az AHKA* esetén exponenciális a szereplők számában, hiszen a kezdeti útkeresésen felül folyamatosan újratervezések történnek. Az FSZT algoritmusban viszont a kezdeti A* dominál, hiszen később csak kis helyi újratervezések fordulnak elő, így a számolási idő nagyjából lineáris a szereplők száma szerint. A táblázatban az 1000 szereplővel futtatott méréseknél kapott számértékek láthatók.



(1000 szereplővel)

Átjárható mezők száma	Algoritmus	Keresési idő (s)	Összesen megtett távolság	Memória-használat	Kifejtett csúcsok száma	10 próbából sikeres	Legnagyobb megoldott egységszám
14098	FSZT	**7**	100767.8	**3874**	**1171818**	8	**1600**
	AHKA* (Átlók nélkül)	130	106335.3	164531	6247997	4	1000
	AHKA* (Átlókkal)	248	**86315.0**	143990	9554315	**10**	1100

3.8. ábra. FSZT algoritmus összehasonlítása az AHKA*-al [11]

3.1.3. Toló-cserélő algoritmus

A Toló-cserélő (Push and Swap) algoritmus két egyszerű lépést váltogatva próbálja célba juttatni a szereplőket, a céljukba vezető legrövidebb út mentén. Az egyik lépés, a tolás egyszerűen a legrövidebb út mentén eggyel előrébb tolja az aktuális szereplőt, ha nem áll az útjában másik egység. Ha a tolás nem lehetséges, akkor cserét hajtunk végre, aminek eredménye, hogy az aktuális szereplő helyet cserél az útjában állóval, és minden más a célját már elért szereplő a csere után is a célján áll. Ez a módszer nem feltételezi, hogy a gráf rács-szerű, az előző technikákhoz képest jóval több esetben ad teljes megoldást, számolási sebességben felveszi a versenyt a már ismertetett Ablakos Hierarchikus Kooperatív A* algoritmussal, és sokkal gyorsabb, mint az egyszerű, A*-ot használó centralizált módszer, amiről a centralizált módszereknél lesz szó.

Jelölések

Adott egy $G = (V, E)$ gráf, rajta n szereplő, R a szereplők halmaza, és $n \leq |V| - 2$ (majd látni fogjuk, hogy a cseréhez szükség van legalább két üres mezőre).

Egy kiosztás $A : [1, n] \to V$ lehelyezi a szereplőket különböző csúcsokra: $\forall i, j \in [1, n], j \neq i : A[i] \in V, A[i] \neq A[j]$. A kezdő kiosztást jelöljük S-el, a végsőt T-vel.

Egy akció $\pi(A_a, A_b)$ az átmenet A_a-ból A_b-be, ami során egy szereplő átlép egy vele szomszédos csúcsra, azaz $\exists i \in [1, n]$ hogy $\forall j \in [1, n], j \neq i : A_a[i] \neq A_b[i], (A_a[i], A_b[i]) \in E, A_a[j] = A_b[j]$.

Egy utazás $\Pi = \{A_0, ..., A_k\}$ kiosztások sorozata, úgy hogy bármely Π-beli egymást követő A_i, A_{i+1}-re létezik $\pi(A_i, A_{i+1})$ akció. A feladatunk találni egy $\Pi^* = \{S, ..., T\}$ utazást.

Az algoritmus

```
procedure ToloCserelo(G, R, S, T)
A ← S;
Π* ← {A};
U ← Ø;
for all r ∈ R do
  while A[r] ≠ T[r] do
    if Tol(Π*, G, A, T, r, U) == hamis then
      if Cserel(Π*, G, A, T, r, U) == hamis then
        return hiba;
  U ← U ∪ A[r];
return Π*;
```

3.9. ábra. Toló-cserélő algoritmus [3]

A 3.9 algoritmus a módszer magas szintű működését mutatja. Beállítja a kezdőkiosztást, inicializálja a Π^* utazást, beletéve az első kiosztást, és üresen létrehozza a céljukat elért szereplők halmazát (U). Ezután minden r szereplőt a Tol és $Cserel$ metódusokkal a megfelelő célba visz, ha ez lehetséges, majd betesz a céljukat elért szereplők halmazába. Ha valaki nem tudja így elérni a célját, akkor az algoritmus megoldhatatlannak jelzi a feladatot, ha mindenki a helyére jutott, akkor visszaadjuk a Tol és $Cserel$ metódusokkal felépített utazást.

Tol metódus

---

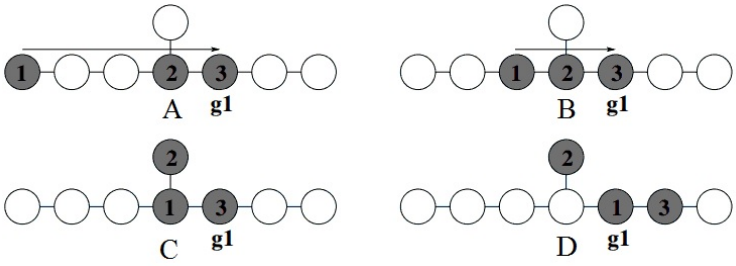
procedure Tol(Π^*, G, A, T, r, U)
$p^* \leftarrow$ LegrovidebbUt($G, A[r], T[r]$);
$v \leftarrow$ első csúcs p^*-ban $A[r]$ után;
while $A[r] \neq T[r]$ **do**
  while $\exists v$ *és* v *üres* G*-ben* **do**
    $A[r] = v$;
    $\Pi^* = \Pi^* + A$;
    $v \leftarrow$ következő csúcs p^*-ban;
  end
  if $A[r] \neq T[r]$ **then**
    Jelöljük $A[r]$-t és U elemeit blokkoltnak G-ben;
    $v_{ures} \leftarrow v$-hez legközelebbi üres csúcs G-ben;
    $p \leftarrow$ LegrovidebbUt(G, v, v_{ures});
    if $p == \emptyset$ **then**
      return hamis;
    end
    Jelöljük $A[r]$-t és U elemeit szabadnak G-ben;
    Toljuk el p mentén a sort v_{ures} felé, és ezt is jegyezzük Π^*-ba;
  end
end
return igaz;

3.10. ábra. Tol metódus [3]

A Tol algoritmus először kiszámol egy legrövidebb p^* utat az r bemeneti szereplőnek $A[r]$-ből $T[r]$-be, majd addig iterál, amíg csere nélkül tudja a szereplőt a célja felé mozgatni. Ha p^* mentén a mezők üresek, akkor egyszerűen halad előre és a keletkező kiosztásokat tárolja. Ha az iteráció után r még nem érte el $T[r]$-t, az azt jelenti, hogy a következő mező foglalt p^* mentén. Ekkor először megpróbáljuk eltolni az útból a mezőn álló szereplőt, úgy hogy se r-t se az U-ban levő egységeket ne kelljen mozgatni,

(de másokat lehet).

Ehhez $A[r]$-t és U elemeit blokkoltnak véve kiszámolunk a p^*-on foglalt mezőtől egy legrövidebb utat egy közeli üres csúcshoz. Ha van ilyen út, akkor a mentén eltolva az egységeket felszabadítjuk a következő mezőt p^* mentén és r tovább haladhat, ha nincs ilyen út, akkor cserélnünk kell.



3.11. ábra. Tol bemutatása [3]

Cserél metódus

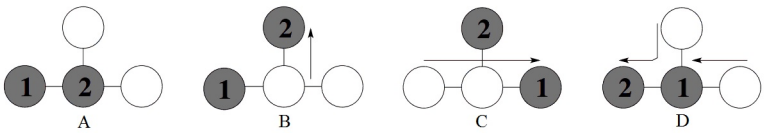
---

procedure Cserel(Π^*, G, A, T, r, U)
$p^* \leftarrow$ LegrovidebbUt($G, A[r], T[r]$);
$s \leftarrow$ első csúcs p^*-ban $A[r]$ után;
siker = hamis;
cserehelyek $\leftarrow$ {minden legalább 3 fokú csúcs G-ben};
while *cserehelyek* $\neq \emptyset$ *és siker* == *hamis* **do**
  v = cserehelyek.kivesz();
  $p \leftarrow$ LegrovidebbUt($G, A[r], v$);
  $\Pi \leftarrow \emptyset$;
  if *KettosTolas(*$\Pi, G, A, T, \{r, s\}, p$*)* == *igaz* **then**
    if *Kiurit(*$\Pi, G, v, A[r], A[s]$*)* == *igaz* $\emptyset$ **then**
      siker = igaz;
    end
  end
end
if *siker* == *hamis* **then**
  return hamis
end
Hajtsuk végre a cserét, majd fordított irányban vigyünk vissza mindenkit a helyére (r, s felcserélve), közben Π^*-ban tároljunk minden lépést;
if $T[s] \in U$ **then**
  return Kijavit(Π^*, G, A, T, r, s)
end
return igaz;

3.12. ábra. Cserél metódus [3]

A Cserel metódus kicseréli az r szereplőt a $T[r]$-hez vezető legrövidebb útján előtte álló s szereplővel, úgy hogy a már a céljukat elért egységek a csere után is a céljukon álljanak. A csere egy olyan csúcsnál valósulhat meg, aminek foka legalább 3. Az algoritmus egy ilyen v csúcsot keres, odatolja r-t és s-t, megcseréli őket, majd visszaviszi őket a kiindulási helyükre fordított sorrendben, és közben az esetleg céljukról eltolt szereplők is visszaállhatnak a helyükre. A két csúcs v-hez tolását a KettosTolas algoritmus teszi meg, ami a Tol-hoz analóg módon működik, de szimultán két egységet mozgat, és nem veszi blokkoltnak az U-beli szereplők helyét, így válogatás nélkül eltol bárkit az útból.

Ha r és s elérte v-t és egy vele szomszédos csúcsot, ki kell még üríteni v két szomszédját, hogy a csere megtörténhessen. Ezt a Kiurit metódusban tesszük meg, ha lehetséges. Ha nem tudunk elég csúcsot kiüríteni, akkor keresünk egy másik legalább 3 fokú csúcsot, ott is megpróbáljuk, és így tovább. Miután megtörtént a csere, az akciósorozatot, ami a KettosTolas-ok és az Kiurit-ek során keletkezett, fordítva le kell játszanunk, figyelve, hogy r és s szerepét felcseréljük. Megtörténhet, hogy s a csere előtt a célján állt, és így a csere miatt elkerült onnan. Az ilyen eseteket kezeli a Kijavit metódus.

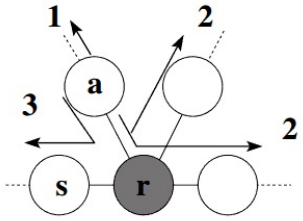


3.13. ábra. Cserel bemutatása [3]

A kiürít és kijavít metódusokról

A Kiurit es Kijavit funkciókat csak nagy vonalakban ismertetjük. A Kiurit meghívásakor r és s már v-n és egyik szomszédján áll, és el kell érni, hogy v még két szomszédja üres legyen. Több eset lehetséges. Az egyik, amikor a v szomszédján álló a szereplő tud Tolni kifele, ekkor a felszabadíthat egy helyet v mellett. A másik, ha a nem tud Tolni, de van egy másik üres hely v egy másik szomszédján. Ekkor r-t és s-t eggyel visszatolva a-t átvihetjük a másik üres helyre, ahol megint megpróbálhat Tolni kifele. Egy harmadik eset, ha a r és s irányába szeretne kitolódni, de ez nem érdekes, mert ahhoz, hogy ezt a megtegye helyet kell cserélnie r-el és s-el valahol, ami ha lehetséges, akkor ugyanott r és s is megcserélhető, tehát ezt az esetet felesleges vizsgálni.

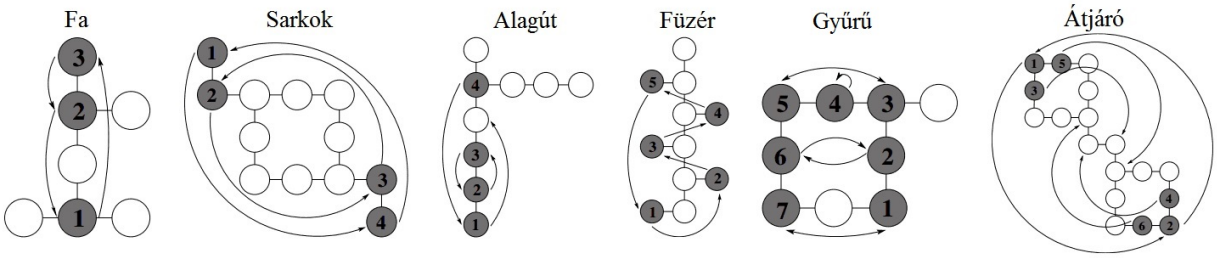
Mint korábban említettük, a Kijavit metódust akkor hívjuk, ha egy cserében a másik résztvevő már a célján állt, és a csere következtében elkerült onnan. Itt is több esetet kell megvizsgálni. Az első, ha a csere után r rögtön Tol-ni tud tovább, azaz s helye felszabadul és így s visszatérhet oda. A második eset, ha r következő lépése is Cserel. Ekkor egy újabb szereplő kerül s céljára. Ha r még a célja előtt Tol egyet, akkor azokat, akikkel s óta helyet cserélt szintén eltolhatjuk eggyel, így s célja felszabadul. Akkor van baj ha r a céljáig csak cseréket hajt végre. Ekkor az a szereplő folytatja a Kijavító algoritmust, aki addig r helyén állt. Ha valamiért egy második eset-beli Cserel nem hajtható végre, akkor a Kijavit elbukik, és így az őt kiváltó Cserel is.



3.14. ábra. Kiurit esetei [3]

Értékelés, teszteredmények

A toló-cserélő algoritmus igazi előnye, hogy a korábbi (Centralizált A*, AHKA*) módszerekhez képest nagyon sok többszereplős keresési problémát képes megoldani, miközben számítási időben nem marad el egyiktől sem, bár a megtalált megoldások általában hosszabb utakat eredményeznek. Megjegyezzük, hogy ha a gráf egyetlen körből áll, tehát nincsen legalább három fokú csúcs, ahol cserélni lehetne, az okozhat olyan helyzetet, ami megoldható, de az itt leírt algoritmus nem oldja meg, a többi pedig igen, de ezt nem lenne nehéz orvosolni.



3.15. ábra. Toló-cserélő algoritmus tesztjéhez használt speciális gráfok [3]

A 3.15 ábrán különböző speciális gráfok láthatók, amiken a toló-cserélő algoritmus összehasonlításra került az AHKA*-al és a centralizált A* módszerekkel, a 3.16 táblázatban pedig a különböző algoritmusok számítási ideje és az általuk talált megoldás hossza látható az egyes gráftípusokon. A ∞ idő jelentése, hogy a számítás nem ért véget egy adott időn belül. Látható, hogy a toló-cserélő módszer minden feladatot megoldott, míg mindkét másik algoritmus elbukott bizonyos esetekben, és a centralizált A* a megoldottaknál is sok számolást végzett. Az is leolvasható, amit mint a toló-cserélő algoritmus hátrányát vetettem fel, hogy bár mindig talál megoldást, az utak összhossza nagyobb, mint az eddigi módszereknél. Ugyanakkor véletlen generált, jórészt szabad mezőkből álló térképen ez az algoritmus is jól teljesít, hiszen az általa talált megoldás távolságtöbblete főleg a szükséges cserék számától függ.

	Centralizált A*		AHKA* (5)		Toló-cserélő	
Feladat	Idő	Méret	Idő	Méret	Idő	Méret
Fa	2.80	15	0.68	34.4	0.86	39
Sarkok	3882	36	0.53	36	1.30	60
Alagút	417	53	∞	n/a	1.36	145
Füzér	207	20	1.04	36.1	0.77	39
Gyűrű	∞	n/a	∞	n/a	2.99	523
Átjáró	∞	n/a	∞	n/a	3.57	108

3.16. ábra. Toló-cserélő algoritmus teszteredményei a speciális gráfokon, összehasonlítva korábbi algoritmusokkal [3]

A tesztek itt sem saját eredmények, egyéb tesztesetek és elemzés a [3] cikkben olvashatók.

3.2. Centralizált módszerek

A centralizált megoldások egy nagy feladattá egyesítik a problémát, általában valamilyen együttes állapotok gráfjának felírásával, és ezen a gráfon valamilyen egyszereplős útkeresési algoritmus segítségével (például A*) keresnek utat egyszerre minden szereplőnek. Ezek a módszerek optimális megoldást adnak vissza (általában lépésszám szerinti minimumot), viszont az együttes gráf mérete a szereplők száma szerint exponenciálisan nő. Például egy 10x10-es négy-kapcsolt rács esetén n egységgel az együttes gráf nagyjából 100^n csúcsú lesz, átlagosan 5^n gyerekszámmal (minden egység léphet 4 irányba vagy várhat). Emiatt még viszonylag kis számú egység esetén is futhatnak lassan ezek az algoritmusok, nagy egységszám esetén pedig szinte használhatatlanok.

Az optimális megoldás reményében mégis születtek a centralizált megközelítést használó megoldások is. Ezekből fogunk ismertetni hármat. Először a legegyszerűbb, de legkevésbé hatékony módszert ismertetjük röviden, és az ahhoz kitalált javítási módszerekről lesz szó. Utána egy kétszintű, a szereplők egyenkénti úthosszai alapján kereső algoritmust nézünk meg, végül pedig egyet, ami főleg a szereplők közti ütközésekre, konfliktusokra koncentrál.

3.2.1. A* alapú próbálkozások optimális megoldásra

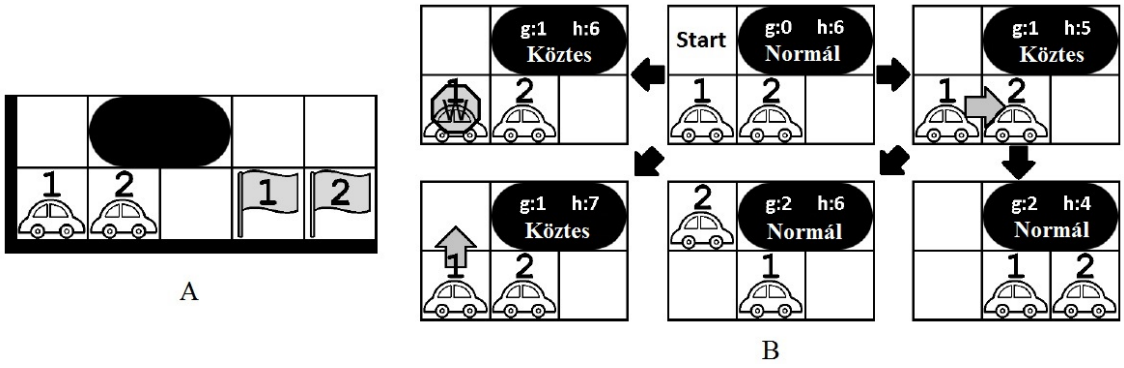
Az egyszerűség kedvéért itt is elsősorban nyolc-kapcsolt rácsokon fogjuk megvizsgálni a módszereket, de most megengedjük, hogy egy szereplő ugyanabban a lépésben lépjen egy másik helyére, amelyikben az ellépett onnan. Az első A*-ot használó centralizált módszerben felépítünk egy gráfot, amiben egy csúcs egy csúcs-n-esnek felel meg az eredeti gráfban, aszerint hogy az n szereplő hol tartózkodik. Élet húzunk két csúcs közt, ha olyan állapotokat jelölnek, amik közt minden egység egyszeri lépésével ütközés nélkül át lehet jutni. Egy lépés az eredeti gráfon történhet a nyolc irányba és lehet várakozás. Ennek megfelelően a csúcsok gyerekszáma az együttes gráfban 9^n nagyságrendű lesz és k mező esetén k^n nagyságrendű lesz a csúcsok száma. Látható, hogy ez már viszonylag kevés szereplő és kis térkép esetén is óriási memória és számításigényt jelent. Ezeken a rossz mutatókon javít Standley két ötlete.

Élfelbontás

Az első az élek felbontása, hogy csökkentsük a csúcsok fokát és ezzel az A* által a csúcsok kifejtésekor nyitott listába helyezett csúcsok számát. Eddig az élek egy időegységet ugrottak, ami alatt minden egység mozoghatott. Most felbontjuk az éleket, úgy hogy a szereplőket egy fix sorrend szerint egyesével mozgatjuk. A csúcsokban a szereplők helyzete mellett minden szereplőhöz jegyezzük, hogy merre akar menni. Így a 9^n-es fokszám 9-re csökken, viszont mivel minden él csak egy egység mozgását jelöli, a cél távolsága n-szeresére nő. Ezt a módszert nevezzük Élfelbontásnak (ÉF, Operator Decomposition).

Ebben a reprezentációban kétféle csúcs van. Normál csúcsok az eddig is létezők, amiknél nincs irány-hozzárendelés egyik egységhez sem. Köztes csúcsnak nevezzük az élek felbontásából származó új csúcsokat, amiknél legalább egy egységhez van irány hozzárendelve. Egy köztes csúcsnál, ahol $n-1$

szereplőhöz már rendeltünk irányt, az utolsó szereplőhöz való irányrendeléssel normál csúcsba jutunk, azaz ekkor lépünk egy időegységet. A futtatott A* algoritmus nem veszi különbözőnek a normál és köztes csúcsokat.



3.17. ábra. Példafeladat (A), és azon az algoritmus első lépései (B) [10]

Ahhoz, hogy az élfelbontás optimális megoldást eredményezzen meg kell engednünk az olyan irány-hozzárendeléseket, amiknél egy szereplő egy nála a sorrendben később jövő szereplő helyére lép. Ha például a 3.17/A ábrán látható helyzetben ezt nem engednénk meg, akkor az 1. jármű várna vagy felfele lépne, amik nem vezethetnek optimális megoldáshoz. Ha ráléphet a 2. helyére, akkor viszont egymást követve célba fognak érni, és az A* keresés meg is találja ezt a megoldást.

A 3.17/B ábra mutatja az első pár lépését az algoritmusnak az előbbi helyzetben. Először az A* kifejti a Start csúcsot az 1. autó szerint és generál 3 köztes csúcsot a megfelelő irány-hozzárendelésekkel, hiszen az 1. autó mehet jobbra, fel és várhat. Ezután kifejti a jobb felső csúcsot, aminek két normál csúcs utódja lesz, mivel a 2. az utolsó egység, akihez irányt kell rendelni. Ő mehet balra-fel, vagy jobbra. Ha vár vagy balra megy, az az 1. autó aktuális csúcsnál vett irány-hozzárendelése mellett ütközéshez vezetne, tehát azokat a csúcsokat (bal, vár) eldobja az algoritmus.

Tökéletes heurisztikával és f-érték-egyenlőség esetén tökéletes döntéshozással az A* $b \times d$ csúcsot fejt ki, ahol b az átlagos fokszám, d pedig a keresés mélysége. Az eredeti algoritmusnál ez $9^n \times t$, míg az élfelbontásos megoldásnál csak $9 \times n \times t$, ahol t az optimális megoldás hossza. Ez exponenciális nyereség tökéletes heurisztika mellett, és gyengébb heurisztikák esetén is feltehetően nagymértékű sebesség és memóriahasználat-beli javulást jelent.

Az élfelbontásos A* is le tudja generálni minden normál csúcs minden szabályos lépéssel elérhető normál leszármazottját, így ez az algoritmus is megtalálja az optimális megoldást, ha az létezik. Az együttes gráfon futó A* csúcslistáinak méretnövekedését valamennyire korlátozhatjuk, ha menet közben figyeljük, hogy ne generáljunk duplikátumot már vizsgált csúcsokból. Élfelbontás esetén pedig, mivel azonos köztes csúcsoknak azonos a normál őse, ezért elég a normál csúcsokat betenni a zárt listába.

Heurisztika

Egy jó alulbecslő heurisztika a szereplők valódi legrövidebb úthosszainak összege. Ezt például a korábban ismertetett visszafele kereső A*-al (RRA*) ki tudjuk számolni, de a köztes csúcsok között az éppen soron levő egység hozzárendelt iránya szerint, RRA* nélkül is kiszámolható, hogy miként változik a heurisztikus érték.

Független részproblémák

Az élfelbontás nagyban csökkenti a keresés által bejárt gráfrész méretét, de az algoritmus végeredményben még így is exponenciális a szereplők számában. Érezhető ugyanakkor, hogy nem biztos, hogy minden szereplő mindenki mással találkozik valahol, tehát lehetséges, hogy a szereplők feloszthatók egymástól független csoportokra, úgy hogy az egy csoportban levő egységek legfeljebb egymással találkoznak a keresés során. Ezekre a csoportokra külön futtatjuk a fent leírt algoritmust, és mivel az a csoportméretben exponenciális, ez a legnagyobb csoport szereplőszámában exponenciális megoldást fog eredményezni.

A függetlenség felismerésére egy egyszerű algoritmust fejlesztettek ki. Kezdetben minden szereplő egy önálló csoportot alkot, majd minden csoportra lefuttatjuk a fenti keresést és ha két csoport valahol ütközik, azokat összevonjuk és az összevont csoportra új megoldást számolunk. Ezt addig folytatjuk, amíg olyan csoportokat nem kapunk, amik egymással nem ütköznek, és ezzel egy időben minden csoportra egy megoldást is kaptunk, amik összessége adja a teljes megoldást.

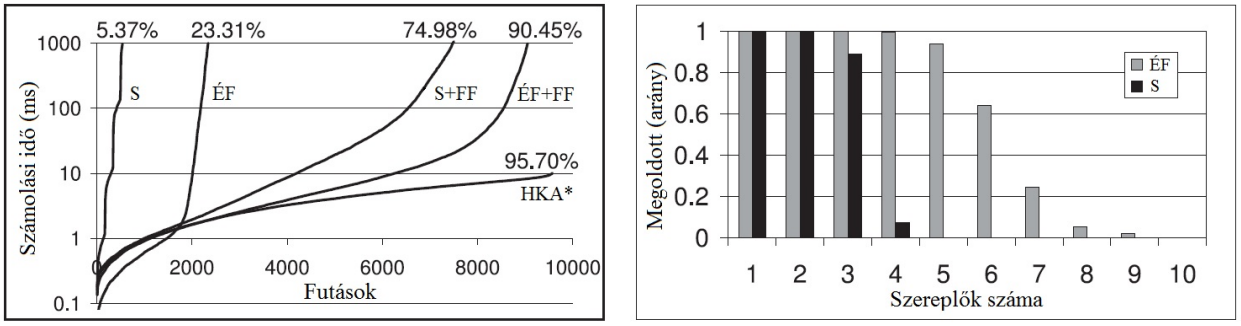
A függetlenségfelismerő (FF, Independence Detection) algoritmus egy továbbfejlesztett változatában nem vonjuk össze azonnal az ütköző csoportokat, hanem először megpróbálunk az egyiknek az éppen megtalált, de ütköző optimális megoldással azonos hosszúságú, de az ütközési pontokat elkerülő másik megoldást találni. Ha ez nem sikerül, akkor a másik csoporttal is hasonlóan újratervezünk. Csak akkor vonjuk össze a két csoportot, ha a másiknak sem sikerült optimális hosszúságú új megoldást találni.

Az FF önmagában nem oldja meg a keresési feladatot, csak kisebb alproblémákra bont egy centralizált keresőt annak többszöri lefuttatásával, de mivel a keresők futásideje általában a szereplők számától nagyban függ, így a legnagyobb csoportra végzett számítás fogja dominálni az egész számolási időt. Ennek megfelelően FF használható az eredeti A*-gal és az ÉF-t használó változattal is.

Teszteredmények

A tesztek nem saját eredmények, részletesebb leírás a [10] cikkben található. Ezek a mérések is 32x32-es 20%-ban blokkolt rácsokon történtek, és a most ismertetett algoritmusok különböző kombinációit hasonlították össze a korábban ismertetett HKA* algoritmussal 10000 térképen 2-60 szereplővel. A 3.18/A grafikonon láthatóak az egy másodpercnél rövidebb futások algoritmusonként, futásidő szerint növekvő sorrendben. Az első A*-os algoritmus (S) láthatóan a leggyengébb, és az ÉF-t használó változat is viszonylag gyengén teljesít, bár ahogy a 3.18/B grafikonon látszik, a teljesítményromlás jóval

lassabb, mint az első A*-é, ahogy a szereplők száma nő. FF-t használva viszont a régi A* is már majdnem a problémák 3/4-ét megoldotta 1 másodperc alatt. ÉF és FF együttes felhasználásával a könnyű feladatok gyorsabb megoldása mellett már nehezebb problémákon is időben túljut az algoritmus.



A B

3.18. ábra. Algoritmusok összehasonlítása

Bár a HKA* összességébe többször és gyorsabban talált megoldást, mint az ÉF+FF változat, sok (430-ból 361-szer) esetben az ÉF+FF talált megoldást, ahol az HKA* nem [10]. Emellett a legalább 40 egységgel futtatott kereséseknél, míg az HKA* megoldásai átlagosan 12.8 lépéssel hosszabbak voltak az elméleti alsó korlátnál, ez a szám az ÉF+FF-nél csak 4.4.

3.2.2. Költségnövelő fa keresés

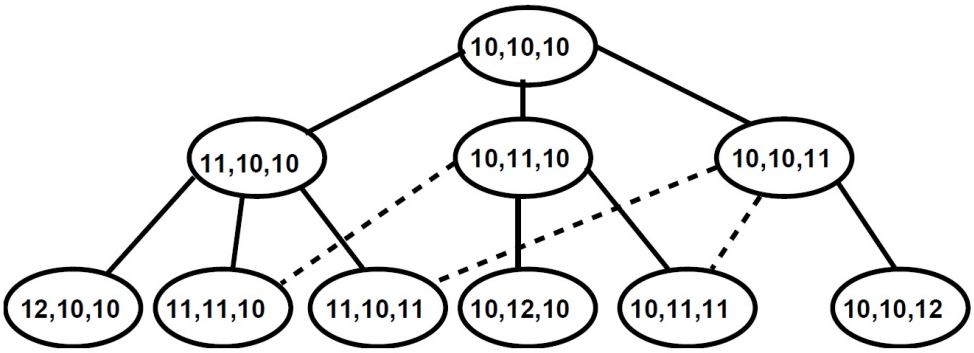
Míg az előzőekben ismertetett centralizált módszerek a teljes, szereplők helyét jegyző állapotokból álló téren kerestek, a költségnövelő fa keresés (KNFK, Increasing Cost Tree Search) egy más koncepcióval közelíti meg a problémát. Két fő kérdést vet fel. Mennyi az egyenkénti útköltsége az egységeknek egy optimális megoldásban? (Jelen esetben a szereplők útjainak összköltsége szerint optimalizálunk.) A második kérdés, hogy hogyan találunk adott költségű utat egy szereplő céljához?

Ezek megválaszolására egy két szintű algoritmust használ a KNFK. A magas szintű keresés egy költségkombinációkból felépített fát (Költségnövelő fa) jár végig. Az alacsony szintű keresés pedig megoldást keres adott költségek mellett. Megjegyzendő, hogy a KNFK algoritmus itt ismertetett változata nem ismeri fel, ha egy feladat megoldhatatlan.

Magas szintű keresés

A költségnövelő fa (KNF) a következőképpen épül fel:

Csúcsok: k szereplő esetén minden csúcs a KNF-ben egy k-hosszú vektor, ami a szereplőknek szánt költségeket tartalmazza sorban, $c = [C_1, C_2, ..., C_k]$. Egy ilyen csúcs reprezentál minden lehetséges megoldást, amiben az i-edik szereplő útköltsége éppen C_i. Egy csúcs teljes költsége $C_1 + C_2 + ... + C_k$, és ennek megfelelően a fában azonos szinten azonos költségű csúcsok vannak.



3.19. ábra. Egy költségnövelő fa [8]

Gyökércsúcs: A gyökércsúcsban a költség minden egységre az adott egységnek a többiek kizárásával számolt optimális út hossza.

Gyerekek: Minden $c = [C_1, C_2, ..., C_k]$ csúcsnak k gyereke van, az i-edikre: $s_i = [C_1, C_2, ..., C_{i-1}, C_i + 1, C_{i+1}, ..., C_k]$. Tehát a gyerek költsége mindig eggyel nagyobb mint a szülőé.

Célteszt: Egy csúcs a KNF-ben célcsúcs, ha létezik teljes, ütközésmentes megoldása a többszereplős útkeresés feladatnak, amelyben minden a_i egység útköltsége éppen C_i.

A 3.19 ábrán látható egy példa KNF-re, ahol az optimális egyéni megoldás mindhárom szereplőnek 10 hosszú. A szaggatott vonallal behúzott élek eldobhatók, hiszen nem érdemes egy csúcsot többször megvizsgálni. Könnyen látható, hogy az ezen a fán végrehajtott szélességi kereséssel (az alacsony szintű keresést használva) megtalálunk egy optimális megoldást.

A KNF mélységét jelöljük Δ-val. Nyilvánvaló, hogy Δ egyenlő az optimális együttes, és az elméleti optimum megoldás hosszának különbségével. A csúcsok gyerekszáma a duplikátumélek kidobása előtt pontosan k, tehát a fa csúcsszáma $O(k^{\Delta})$, tehát Δ-ban exponenciális, és nem a szereplők számában.

A magas szintű keresés tehát szélességi kereséssel bejárja (és közben építi) a KNF-t, és minden elért csúcsra lefuttatja az alacsony szintű keresést, hogy meghatározza, hogy az adott csúcs célcsúcs-e.

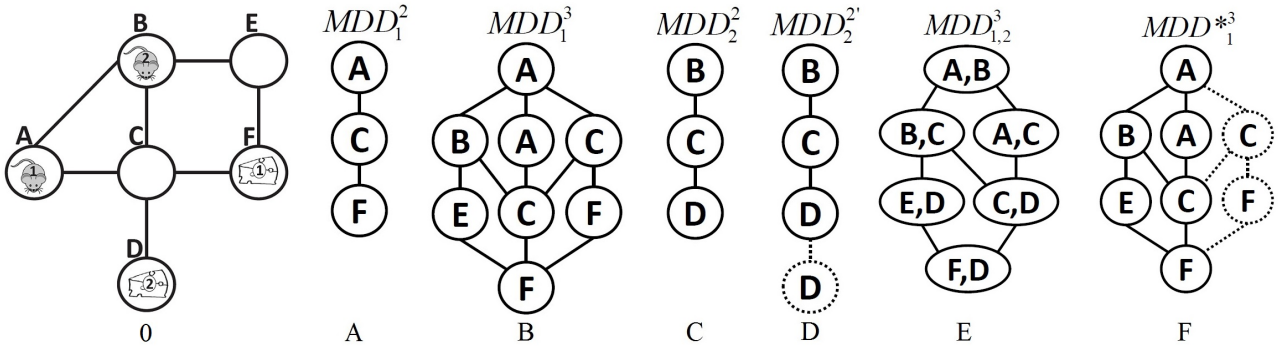
Alacsony szintű keresés

Hogy meghatározzuk egy csúcsról, hogy célcsúcs-e, az első ötletünk lehet, hogy minden egységre vegyünk minden megfelelő költségű utat és ezeket hasonlítsuk össze az összes többi szereplő összes útjával. Természetesen ez nem járható út, hiszen már az egy egységhez tartozó utak száma is túlnőhet az értelmes határon.

Tömör útreprezentáció MDD-kkel

Az utakat speciális tömör adatszerkezetben tároljuk. Ez az MDD (Multi-value Decision Diagram), ami önmaga is egy gráf. MDD_i^c legyen az a_i szereplőhöz tartozó MDD, ami a c költségű, s_i-ből t_i-be vezető utakat tárolja. Ennek megfelelően MDD_i^c-nek egy forrás csúcsa van a 0. szinten (s_i) és egy nyelő a c. szinten (t_i). Minden t mélységben levő csúcs MDD_i^c-ben a_i egy lehetséges helyzetét jelöli

a t időpontban egy s_i-ből t_i-de vezető c költségű úton. A 3.20/A,B ábrák MDD_1^2-t és MDD_1^3-t (a_1 egység), a 3.20/C ábra pedig MDD_2^2-t (a_2 egység) ábrázolja a 3.20/0 ábrán látható feladathoz, ahol az egerek feladata elérni saját sajtjukat.



3.20. ábra. A feladat és a hozzá tartozó MDD-k [8]

Vegyük észre, hogy míg a c hosszú utak száma c-ben exponenciális, addig MDD_i^c mérete maximum $|V| \times c$ (V az eredeti gráf csúcsszáma), hiszen MDD_i^c minden szintjén maximum $|V|$ csúcs lehet, és pontosan c szintje van.

MDD_i^c felépítése nagyon egyszerű. Az eredeti gráfon futtatunk egy szélességi keresést s_i gyökérrel c mélységig és megtartjuk az így bejárt részgráfnak azt a részét, amely (s_i-ből indul és) t_i-ben végződik. Továbbá MDD_i^c felhasználható MDD_i^{c+1} legenerálásakor.

$MDD_i^c(x,t)$-vel jelöljük a t mélyen levő x helyhez rendelt csúcsot MDD_i^c-ben. Például MDD_1^2 gyökere $MDD_1^2(a,0)$. Ha az MDD mélysége nem érdekes, akkor használhatjuk a MDD_i jelölést.

Célteszt MDD-kkel

A célteszt során a k szereplő MDD-i alapján ki kell találnunk, hogy létezik-e ütközésmentes megoldás, tehát minden MDD-n kell egy-egy utat találni úgy, hogy ezek az utak ne kerüljenek konfliktusba egymással. Például a 3.20/0 ábra feladatában, a KNF [2,2]-es gyökeréhez tartozó MDD_1^2 és MDD_2^2 minden útja c-re fut az első lépés után, így ezeken nincs teljes megoldás, azaz ez a KNF-csúcs nem célcsúcs. A [3,2] csúcshoz tartozó MDD_1^3 és MDD_2^2 MDD-ken már van nem ütköző út, <a-b-c-f> a_1-nek és <b-c-d> a_2-nek. Tehát ez a KNF-csúcs célcsúcs, és az algoritmus megáll, visszaadva 5-öt, mint optimális úthosszösszeget.

Ahhoz, hogy az MDD-ken hatékonyan tudjunk keresni bevezetjük az együttes MDD-ket, amiket több egység MDD-jének összeolvasztásával kapunk. MDD_{ij} az MDD_i és MDD_j összeolvasztásával jön létre. MDD_{ij} minden csúcsa az a_i, a_j szereplők egy szabályos elhelyezése. Nézhetjük csak azonos mélységű MDD-k összeolvasztását, hiszen egy kisebb MDD segédélekkel nagyobbá tehető (3.20/D ábra, $MDD_2^{2'}$).

Formálisan MDD_{ij} egy csúcsa, $MDD_{ij}([x_i,x_j],t)$ az a_i, a_j egységeknek egy $[x_i,x_j]$ elhelyezését tartalmazza a t időpontban és $MDD_i(x_i,t)$ és $MDD_j(x_j,t)$ egyesítésével jön létre. $MDD_{ij}([x_i,x_j],t)$

gyerekeit az őt felépítő csúcsok gyerekeinek egyesítésével kapjuk, de eldobjuk azokat a gyerekeket, amiken a két szereplő egy helyen van, illetve azokat az éleket, amik ütközéshez vezető mozgást jelentenek. Ezekből következően a kétszereplős MDD-k minden szintjén legfeljebb $|V|^2$ csúcs lehet, tehát a c magas MDD teljes mérete $|V|^2 \times c$. A 3.20/E ábrán látható MDD_1^3 és $MDD_2^{2'}$ összeolvasztása a konfliktusos csúcsok és élek eldobása után.

k szereplőre egyszerűen általánosítható ez az összeolvasztási módszer. Észrevehetjük, hogy a k-szereplős MDD-k az eddig használt k-szereplős keresési gráf kis részgráfjait alkotják. Szintenkénti csúcsszámuk $O(|V|^k)$.

A keresés

Az alacsony szintű keresés a $f = C_1 + C_2 + ... + C_k$ költségű KNF-csúcshoz a k-szereplős összeolvasztott $MDD^c_{12...k}$-n keres utat $MDD^c_{12...k}([s_1, ..., s_k],0)$-ből (ami az egyetlen csúcs a 0. szinten) $MDD^c_{12...k}([t_1, ..., t_k], c)$-be (ami az egyetlen csúcs a c. szinten). Ha talál utat akkor ez a KNF-csúcs célcsúcs, ha nem talál, azaz ilyen költségek mellett nem oldható meg konfliktusmentesen a feladat, akkor hibával tér vissza. Az MDD-n való kereséshez például mélységi keresés használható, mivel bár az ábrákon nincs jelölve, de az élek lefele mutató irányított élek, és egy szinten belül nem futnak élek, így a megtalált út a 0. szintről a c-edikre csak c-hosszú lehet (c itt a C_i-k maximuma).

---

procedure KNFK(...)
Felépítjük a KNF gyökerét egyéni keresésekkel;
foreach *KNF csúcs minimális f-érték szerint haladva* **do**
  foreach a_i *szereplő* **do**
    Építsük meg MDD_i-t;
  end
  Hajtsunk végre ritkítást; //opcionális
  Az egyszereplős MDD-ket olvasszuk össze, majd keressünk utat a k-szereplős MDD-ben;
  if *Célcsúcsot találtunk* **then**
    return Megoldás
  end
end

3.21. ábra. KNFK algoritmus [8]

Technikák az algoritmus gyorsítására

Az eddig leírtak szerint a KNF-en a megtalált célcsúcs előtt vizsgált csúcsok egyike sem célcsúcs. Ezeknél a csúcsoknál az alacsony szintű keresés végigfut a teljes k-szereplős MDD-n, ami az adott csúcshoz tartozik. Ez egyre hosszabb számolást igényel, ahogy a szereplők száma nő, ezért kitaláltak egy az MDD-k tulajdonságait kihasználó gyorsítási módszert, a páros ritkítást.

Páros ritkítás (PR, Pairwise Pruning)

Ennek lényege, hogy mielőtt a k-szereplős MDD-t megnéznénk, vesszük az összes egyszereplős MDD-t, összeolvasztjuk őket az összes lehetséges párosítás szerint és a párokra sorban megvizsgáljuk, hogy létezik-e út a 0. szintjétől a c-edikig. Ha létezik, megyünk a következő párra, ha viszont nem létezik, akkor már ezen a ponton biztos, hogy a k-szereplős MDD-ben sincsen jó út, hiszen a k szereplő közt a most vizsgált kettő is ott van. Mivel a használt mélységi keresés a kétszereplős MDD-ben gyorsan talál jó utat, ha az létezik, ezért ez a módszer jól használható, és nem jelent számottevő többletszámolás akkor se, ha minden párra talál jó utat, és a k-szereplős teret is végig kell nézni.

Fejlesztett páros ritkítás (FPR, Enhanced Pairwise Pruning)

A páros ritkítás tovább javítható, ha észrevesszük, hogy a párosával való összeolvasztáskor lehetséges, hogy az összeolvasztott MDD-k bizonyos csúcsait konfliktusok miatt egyáltalán nem használhatjuk. Tehát az összeolvasztással keletkezett kétszereplős MDD alapján a két egyszereplősből is kidobhatunk bizonyos csúcsokat. A további párosításokban, illetve ha odáig jutunk, akkor a k-szereplős MDD megalkotásakor is dolgozhatunk a csökkentett méretű egyszereplős MDD-kkel. Ezt a ritkítást minden párra elvégezzük úgy, hogy ha egy MDD-t megritkítunk, akkor onnan kezdve annak ritkított változatával dolgozunk tovább. A ritkítás a k-szereplős MDD-n való keresést is gyorsítja, hiszen kisebb MDD-knek kisebb az összeolvasztottja is. Emellett az FPR nagyobb eséllyel veszi észre már a páros hasonlításkor, ha nincs megoldás.

Ismételt fejlesztett páros ritkítás (IFPR, Repeated Enhanced Pairwise Pruning)

MDD_1 és MDD_2 összeolvasztása és ritkítása után kapjuk MDD_1^*-ot és MDD_2^*-ot, amik mindketten a további összeolvasztásaik során tovább ritkulnak MDD_1^{**}-á és MDD_2^{**}-á. Könnyű látni, hogy ezek után lehetséges, hogy MDD_1^{**} és MDD_2^{**} összeolvasztása még jobban megritkítja őket. Tehát lehet értelme az egész FPR-t újra végigjátszani, így tovább növelve az esélyt, hogy csak páros összehasonlításokkal kiderítjük, hogy nincs jó megoldás. Az ismétlést folytathatjuk addig amíg már egyik MDD sem ritkul tovább, vagy bizonyos iterációszámig. (3.20/F ábrán látható MDD_1^{*3}, amit MDD_{12}^3 szétszedésével nyerünk. A szaggatott rész, ami eltűnik belőle az eredeti MDD_1^3-hoz képest.)

A ritkítások hatékonysága

Nyilvánvalóan a sima PR fut a leggyorsabban, hiszen továbblép, amint talál egy jó utat a kétszereplős MDD-ben, míg az FPR ezeket is mind teljesen végignézi, hogy megtalálja hol ritkíthatók az egyszemélyes MDD-k, viszont ennek köszönhetően a későbbi páros és az esetleges k-személyes keresések gyorsulnak. Az IFPR még az FPR-nél is lassabb, de nagyobb mértékű ritkítást eredményez.

m-szereplős ritkítás

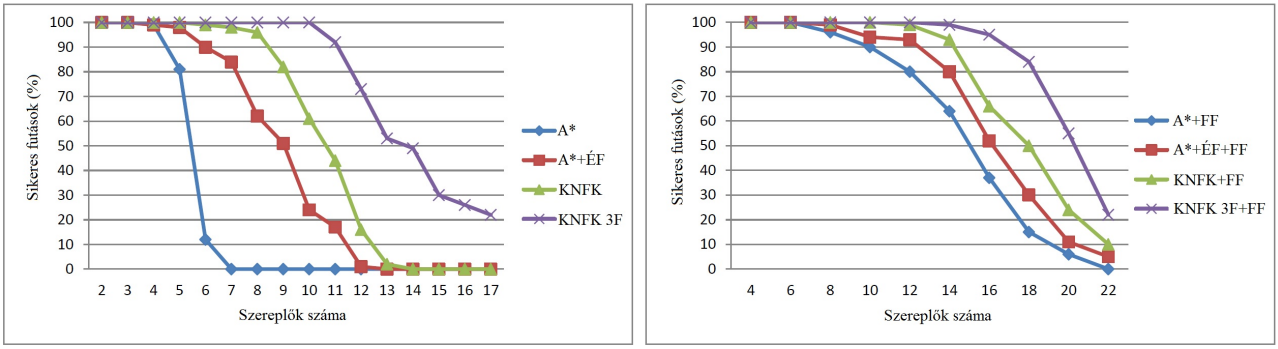
A páros ritkítások nem vesznek észre olyan konfliktusokat, amik csak több szereplő együttműködése esetén jelentkeznek. Értelmes lehet tehát az pároshoz hasonlóan nagyobb szereplőszámra is végrehajtani a ritkítást, ugyanúgy mint kétszereplős esetben, csak m-szereplős MDD-t használva, minden lehetséges szereplő-m-esre. Észrevehetjük, hogy egy m-szereplős ritkítás előtt van értelme végigjátszani az $m-1$

szereplős ritkítást, hiszen az az m-eshez képest gyors, és gyorsítja az m-szereplőset is.

Egy m-szereplős c magas MDD-ben legfeljebb $|V|^m \times c$ csúcs van, és $O(k^m)$ darab m-szereplős MDD van, ha az egységek teljes száma k. Tehát legrosszabb esetben (ha minden MDD-ben van jó út) az m-szereplős ritkítás futásideje $O(|V|^m \times c \times k^m)$.

Teszteredmények

A közölt eredmények nem saját mérésekből származnak. Részletesebb magyarázat és egyéb érdekes mérések és összehasonlítások a [8] cikkben olvashatók.



3.22. ábra. Algoritmusok összehasonlítása [8]

Az első kísérlet 8x8-as akadálymentes rácson futott, de a szereplők úgy lettek elhelyezve, hogy minél több konfliktus legyen az útjaik közt. A 3.22/bal grafikonon a legfeljebb 5 percig futó sikeres, teljes megoldások arányát láthatjuk, és jól látszik, hogy már a ritkítás nélkül használt KNFK is jobban teljesít, mint az A* és az annál valamivel hatékonyabb A*+ÉF, és ezeket mind felülmúlja a KNFK 3F, ami fejlesztett 3-szereplős ritkítást használ. De az is látszik, hogy még a KNFK 3F is csak elég kevés szereplő esetén talál mindig megoldást.

A második kísérlet szintén 8x8-as akadálymentes rácson futott, de itt a szereplők kezdeti helyzete és céljaik elhelyezkedése véletlen volt. Ahogy a függetlenségfelismerő (FF) bemutatásakor említettük, az használható bármely kereső mellett. Ennek megfelelően a KNFK és KNFK 3F mellett is. Az így mért eredményeket mutatja a 3.22/jobb grafikon. Bár a kísérlet kicsit más kiinduló helyzetből indult mint az előző, a javulás mértéke szemmel látható, de ezek az algoritmusok szereplőszámban így sem veszik fel a versenyt az elosztottakkal.

Amikor az A jobb*

Két speciális helyzetben bizonyult az A* jobbnak, mint a KNFK. Az egyik amikor hosszú folyosókon kell végigmenni az egységeknek, a másik pedig, amikor kis területen nagyon sűrűn helyezkednek el a szereplők. Ezen esetekben ugyanis a KNF Δ mélysége nagyra nő viszonylag kis szereplőszám mellett is, és mivel a KNFK Δ-ban exponenciális, ezért ilyenkor rosszul teljesít.

3.2.3. Konfliktus alapú keresés és továbbfejlesztése

A konfliktus alapú keresés (KAK, Conflict Based Search [6]) szintén egy kétszintű algoritmus. Alacsony szinten minden egységnek külön keresünk utat, megfelelve a magas szinten használt megkötési fa (MF, Constraint Tree) megkötéseinek, amiket az alacsony szintű keresés során tapasztalt ütközések alapján adunk a fához. Ez a módszer a centralizált és elosztott módszerek keveréke, hiszen magas szinten a szereplők együttes kezelésével optimális megoldást keres, de az alacsony szintű keresés az elosztott módszerekre jellemző módon, egyesével történik. Megjegyzendő, hogy ez az algoritmus csak azt veszi ütközésnek, ha két egység egyszerre lép egy helyre, azt nem, ha egymással szemben mennek át egy élen. Továbbá megengedi, hogy két egység kövesse egymást, azaz az egyik oda lépjen ahol a másik éppen tartózkodik, ha az ugyanekkor ellép onnan.

Jelölések

Az (a_i, v, t) hármas az a_i szereplő egy *megkötése*. Jelentése, hogy a t időpontban a_i-nek tilos a v helyen tartózkodnia. Az algoritmus során a szereplőknek egyre több megkötésük lesz. Az a_i szereplő egy útját *konzisztensnek* nevezzük, ha megfelel minden megkötésének, azaz sehova sem lép olyankor, amikor azt egy megkötés tiltja. Egy megoldás (k szereplő együttes útja) *konzisztens*, ha minden szereplő saját útja konzisztens benne. Az (a_i, a_j, v, t) négyes egy *konfliktus*, és azt jelenti, hogy t időpontban a_i és a_j ugyanarra a v csúcsra lépett. Egy megoldás *helyes*, ha nincs benne konfliktus. Egy konzisztens megoldás nem biztosan helyes, hiszen a megkötéseknek való megfelelés nem biztosítja, hogy máshol sem ütköznek az egységek.

A KAK algoritmus lényege, hogy minden egységhez létrehozzuk megkötések egy halmazát, és ezen megkötéseknek megfelelő utat találunk nekik. Ha ezek az utak valahol konfliktusba kerülnek, akkor az alapján új megkötéseket adunk az egységeknek. A magas szintű keresés megtalálja a konfliktusokat és nyilvántartja a megkötéseket, míg az alacsony szint az egységek útját számolja ki.

Magas szintű keresés

A magas szint egy megkötési fának (MF) nevezett bináris fában keres. Az MF egy N csúcsa három dolgot tartalmaz. Megkötések egy halmazát (N.megkot), egy megoldást (N.mego) és a teljes költséget (N.kolt). A gyökércsúcs megkötéshalmaza üres. Minden gyerek csúcs megörökli a szülő megkötéseit és hozzáad még egy megkötést a halmazhoz. A csúcsbeli megoldás k útból áll és az egyéni utak meg kell feleljenek az adott szereplő megkötéseinek. Ilyen utakat az alacsony szintű keresés találhat. Az MF egy csúcsa célcsúcs, ha a benne található megoldás helyes. A csúcsok teljes költsége a megoldásban szereplő egyéni utak hosszának összege. Ez az érték lesz az MF-csúcs f-értéke is, ami szerint a magas szintű keresés előrehalad (mindig a legkisebb f-értékű csúcsot veszi a még nem vizsgált nyitott csúcsok közül).

A keresés fenntart egy nyitott csúcs listát, amibe minden legenerált gyerek belekerül, és az algoritmus akkor áll le, ha talál megoldást, vagy a nyitott lista kiürül.

```
procedure MagasSzint(...)
R.megkot = ; //R a gyökér
R.mego = keressünk egyéni utakat az alacsony szintű kereséssel;
R.kolt = ossz(R.mego);
Tegyük be R-t a nyitott listába;
while A nyitott lista nem üres do
    P ← legkisebb költségű csúcs a nyitott listából;
    Szimuláljuk P útjait, amíg konfliktust nem találunk;
    if P-ben nincs konfliktus then
        return P.mego //P célcsúcs
    end
    C ← az első (a_i, a_j, v, t) konfliktus P-ben;
    foreach a_i szereplő C-ben do
        A ← új csúcs;
        A.megkot ← P.megkot + (a_i, v, t);
        A.mego ← P.mego;
        Frissítsük A.mego-t az alacsony szint meghívásával (a_i-re);
        A.kolt = ossz(A.mego);
        Tegyük A-t a nyitott listába;
    end
end
```

3.23. ábra. KAK algoritmus magas szintű keresése [6]

Az MF-csúcsok kifejtése

Az N csúcsban adott megkötésekkel meghívjuk az alacsony szintű keresést, ami talál minden egységhez egy optimális, konzisztens utat. Ezután betesszük N-t a nyitott csúcs listába. Ha az f-értéke alapján N-re kerül a sor, akkor szimuláljuk N.mego utait és, ha nem történik ütközés, akkor célcsúcsnak jelöljük N-t és visszaadjuk N.mego-t mint megoldás. Ha viszont valahol ütközést tapasztalunk, leállítjuk a szimulációt, az N-t nem célcsúcsnak jelöljük, és az ütközésnek megfelelő (a_i, a_j, v, t) konfliktust feloldjuk. (Lehet több egység között is konfliktus.)

Konfliktusok feloldása

Az N csúcsnál (a_i, a_j, v, t) konfliktust találtunk, így az nem célcsúcs, de a konfliktusból tudhatjuk, hogy egy helyes megoldásban a_i és a_j közül legfeljebb az egyik szabad, hogy v-n legyen t időpontban. Így az (a_i, v, t) és (a_j, v, t) megkötések egyikét hozzá kell adnunk a megkötéshalmazhoz. Mivel optimális megoldást keresünk, mindkét esetet meg kell vizsgálnunk, ezért N-nek létrehozzuk két gyerekét, akik mindketten öröklik N minden megkötését, és kapnak az újak közül egyet-egyet. Lehetséges, hogy egy

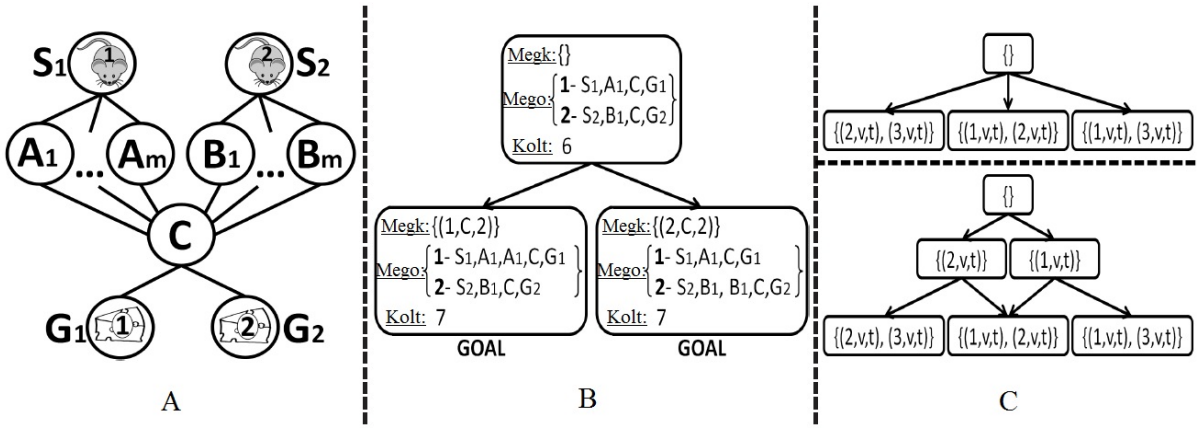
konfliktus 3 vagy több szereplőt tartalmaz, akik mind egyszerre akartak azonos helyre lépni. Ekkor ezt feloldhatjuk egyszerre, vagy kettesével is (3.24/C ábra).

Vegyük észre, hogy elég minden csúcsban csak az újonnan keletkezett megkötést eltárolni, hiszen a gyökérhez visszavezető úton megtaláljuk az örökölt megkötéseket. Hasonlóan, az alacsony szintű keresést is elég arra az egy egységre futtatni, aminek új megkötése keletkezett, hiszen a többire a keresés új megkötés hiányában ugyanazt az utat találná meg.

Alacsony szintű keresés

Az alacsony szintű keresés egy szimpla A* keresés minden egységre külön-külön a korábban már látott háromdimenziós, időt is figyelembe vevő gráfon, azzal a megkötéssel, hogy a megkötéseknek megfelelő csúcsokra az adott időpontban nem léphet a szereplő, tehát ha egy v csúcsra $g(v) = t$ és létezik (a_i, v, t) megkötés, akkor ezt a (v, t) csúcsot az a_i-re vonatkozó A* eldobja.

Példa



3.24. ábra. A feladat (A) és a hozzá tartozó megkötési fa (B) és egy 3-szereplős konfliktus két kifejtési módja (C) [6]

A 3.24/A ábrán látható egerek feladata elérni a saját sajtjukat. A feladathoz tartozó MF-t a 3.24/B ábra mutatja. A gyökérben nincsen megkötés, és mindkét egér talál egy utat magának, összesen 6 költséggel, de az útszimuláció kimutatja, hogy ezek a C csúcsban ütköznek ($(a_1, a_2, C, 2)$), ezért a gyökérről látjuk, hogy nem célcsúcs, és a konfliktus szerint létrehozunk két gyereket $(a_1, C, 2)$ és $(a_2, C, 2)$ megkötéssel. A bal gyerekben az alacsony szint új utat számol a_1-nek, ami egyet vár mielőtt C-re lépne. a_2 útja itt változatlan marad. Így ennek a csúcsnak a költsége 7. Hasonlóan a jobb gyerekre is kijön a 7-es költség és mindkettő bekerül a magas szintű keresés nyitott listájába. Ezután a bal gyerek kerül kifejtésre és az útszimuláció nem talál konfliktust, tehát ez egy célcsúcs és visszatérhetünk a benne tárolt megoldással.

Csoportosító konfliktus alapú keresés

A csoportosító konfliktus alapú keresés (CsKAK, Meta-Agent Conflict Based Search) a KAK egy továbbfejlesztése, és a már ismertetett függetlenségfelismerő általánosítása. Az alacsony szintű keresés itt nem egyenként a szereplőkre, hanem szereplők csoportjaira fut, bár kezdetben minden csoport egy szereplőből áll.

A magas szintű keresésben ezentúl, ha konfliktust találunk, két megoldás közül választhatunk. Vagy elágazunk kétfelé, mint eddig, vagy összevonhatjuk az ütközőket. Ha összevonunk, akkor létrejön egy új csoport, amit már később nem választhatunk szét, de összevonhatunk más csoportokkal. Összevonás után alacsony szinten az új csoportnak egy centralizált keresővel (bármelyikkel) keresünk optimális megoldást, eszerint frissítjük az MF éppen vizsgált csúcsának megoldás és költség adatait, és betesszük azt a nyitott listába. Kérdés, hogy mikor vonjunk össze csoportokat, de előbb nézzük meg, hogyan történik az összevonás.

Összevonás

Mivel csoportokkal dolgozunk, a megkötések és konfliktusok is csoportokra értelmezettek. (X, v, t) megkötés jelentése, hogy az X csoport semelyik tagja sem lehet v-n t időpontban. (X, Y, v, t) konfliktus jelentése pedig, hogy valamely $x \in X$ és $y \in Y$ egyszerre tartózkodik v-n t időpontban a megtalált konzisztens utak szerint. Az x, y szereplők pontos ismerete nem szükséges, hiszen az ebből a konfliktusból keletkező megkötések az adott csoport minden tagjának megtiltják, hogy v-n legyenek t időpontban.

Összevonáskor az összevont csoportok (a_i, a_j) már létező megkötéseivel is foglalkoznunk kell. Ehhez bevezetjük a külső és belső konfliktus, illetve a külső és belső megkötés elnevezéseket. Belső konfliktus, amiben csak a_i és a_j szerepel. Ezek és a belőlük származó belső megkötések nem érdekesek, hiszen mostantól az összevont csoportra centralizált algoritmust futtatunk. Külső konfliktus, ami a_i és a_k $(k \neq j)$, vagy a_j és a_m $(m \neq i)$ közt történt. A külső konfliktusból származó (a_i, v, t_1) és (a_j, w, t_2) megkötéseket az együttes csoportjukra vonatkozó (a_{ij}, v, t_1) és (a_{ij}, w, t_2) megkötésekké alakítjuk.

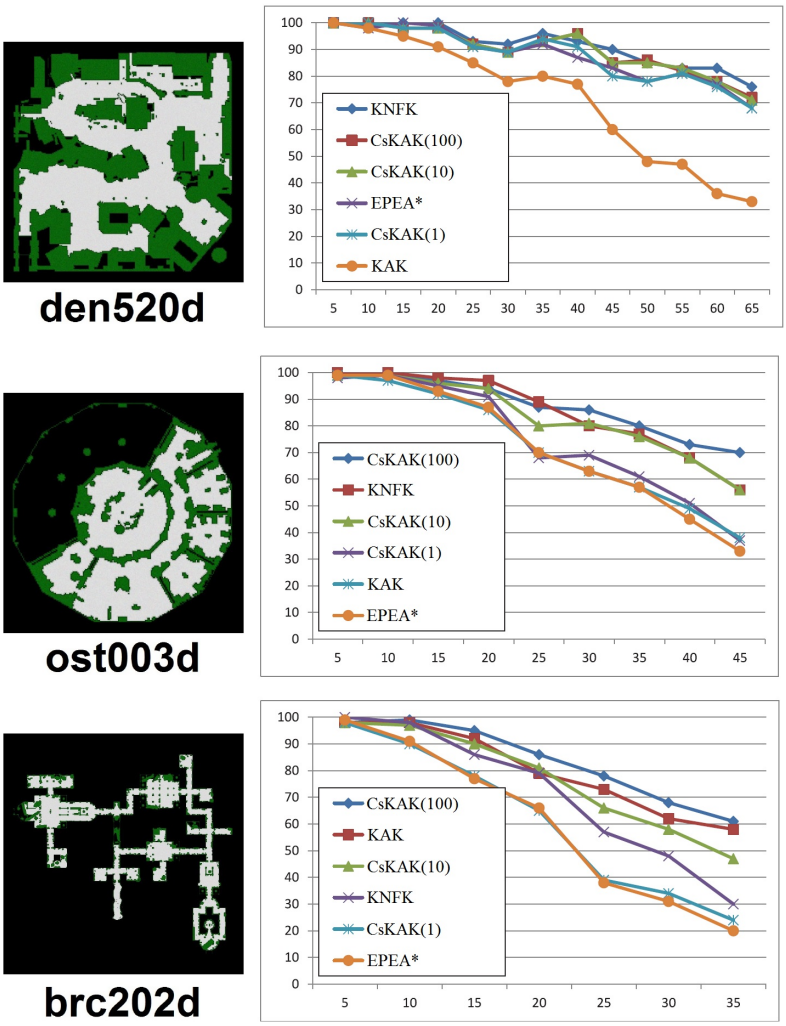
Összevonási stratégia

Sokféleképpen eldönthető, mikor érdemes összevonni. Ezek közül egyben bevezetünk egy határértéket. Két csoportot akkor vonunk össze, ha a köztük a futás során felfedezett konfliktusok száma meghaladja ezt a határértéket. Hogy ezt figyelni tudjuk fenntartunk egy mátrixot amiben jegyezzük a csoportpárok konfliktusainak számát. A H határárértékű algoritmust CsKAK(H)-val jelöljük.

Ilyen stratégia mellett észrevehető, hogy a CsKAK(0) éppen a függetlenségfelismerőnek felel meg, hiszen amint konfliktust talál összevon, ha pedig sosem vonunk össze (CsKAK(∞)), az a normál KAK megfelelője. Ugyanúgy, mint az FF, a CsKAK is használhatja bármely centralizált keresőt alacsony szintű keresésnek.

Teszteredmények

Az eredmények bővebb kifejtéséért és további tesztekért lásd a [6] és [7] cikkeket.



3.25. ábra. Teszteredmények 3 térképen, algoritmusok összehasonlítása [7]

A 3.25 ábrán látható a tesztek eredménye. A tesztek a Dragon Age: Origins című játék néhány speciális tulajdonságú pályáján futottak (grafikonok mellett), egyiken (legfelül) a nyílt területek a jellemzők, a másodikon (középen) szűkebb átjárók és nyílt területek vegyesen találhatóak, a harmadikon (alul) pedig a folyosók és szűk helyek dominálnak. A CsKAK algoritmus alacsony szintű keresőjeként a dolgozatban nem tárgyalt Enhanced Partial Expansion A* (EPEA*) [1] keresést használták, KNFK-nak pedig egy ritkításos verziója futott.

A vízszintes tengelyeken a szereplők száma látható, a függőlegesen a sikeres futások száma. Minden szereplőszámmal 100 teszt futott, a szereplőket és céljaikat véletlen felhelyezve. Ha egy algoritmus nem oldotta meg a feladatot 5 perc alatt, akkor leállították.

Látható, hogy ahol nagy nyílt területek vannak, ott a KNFK teljesít a legjobban, de egyedül a sima KAK (CsKAK(∞)) marad le jelentősebben. Ugyanakkor a sok szűk helyet tartalmazó térképen a KAK majdnem a legjobb eredményeket mutatja, míg itt a KNFK gyengébb. A legrosszabb mutatókat a szűk járatokkal is rendelkező pályákon az EPEA* adja.

4. fejezet

Végszó

Bár a dolgozat végéhez értünk, a többszereplős útkeresési módszerek kutatása közel sem ért véget. Láthattuk, hogy a gyors módszerek a sebesség növekedésével távolodnak az optimális megoldástól, az optimális módszerek pedig már viszonylag kicsi szereplőszám mellett is nagyon lassúak, tehát olyan helyzetben, amikor a másodperc törtrésze alatt kell dönteni, ezek nem használhatók.

Az ismertetett algoritmusok, bár egyre jobbak és jobbak, mindegyiknek megvan a gyengesége, mindegyik rosszul teljesít bizonyos helyzetekben, vagy csak speciális gráfokon, például rácsokon működik hatékonyan. Megtehetjük, hogy a mesterséges intelligenciával döntetjük el a helyzethez igazodva, hogy melyik algoritmust érdemes futtatni, de szebb megoldás lenne, ha találnánk egy minden helyzetben jól teljesítő technikát.

Természetesen léteznek a dolgozatban ismertetetteken kívül még más megoldási kísérletek, de mint a bevezetőben említettük, ezek legtöbbje új eredmény az utóbbi 5-10 évből. Ezek közül egyet említenénk meg, amiben a speciális gráfokon vett többszereplős útkeresés és a hálózati folyamok problémájának ekvivalenciáját bizonyítják [12]. A hasonlóság a két probléma között tényleg szembetűnő, de az általános gráfokon vett ekvivalencia bizonyítására nem sok remény van, hiszen a folyamprobléma P-beli, míg a többszereplős útkeresés NP-teljes.

Irodalomjegyzék

[1] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. R. Sturtevant, J. Schaeffer, and R. Holte. Partial-expansion a* with selective node generation. In *AAAI*, 2012. 40

[2] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning a. *Artificial Intelligence*, 155(1) :93–146, 2004. 1, 6, 7, 8, 9

[3] R. Luna and K. E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 294–300. AAAI Press, 2011. 1, 22, 23, 24, 25, 26

[4] I. Millington. *Artificial Intelligence for Games*. CRC Press, 2006. 1, 5, 10, 11, 12

[5] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2) :267–305, 1996. 4

[6] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. Conflict-based search for optimal multi-agent path finding. In *AAAI*, 2012. 1, 36, 37, 38, 39

[7] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. Meta-agent conflict-based search for optimal multi-agent path finding. In *SOCS*, 2012. 1, 39, 40

[8] G. Sharon, R. Stern, M. Goldenberg, and A. Felner. The increasing cost tree search for optimal multi-agent pathfinding. 2011. 1, 31, 32, 33, 35

[9] D. Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005. 1, 15, 16, 17

[10] T. S. Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, 2010. 1, 28, 29, 30

[11] K.-H. C. Wang and A. Botea. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008. 1, 18, 19, 21

[12] J. Yu and S. M. LaValle. Multi-agent path planning and network flow. In *Algorithmic Foundations of Robotics X*, pages 157–173. Springer, 2013. 41