Eötvös Loránd Tudományegyetem

Természettudományi Kar

Moschnitzka Ádám

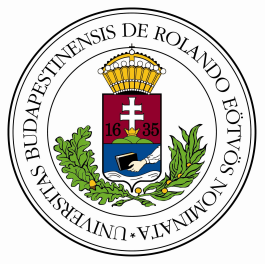
Útkereső algoritmusok négyzetrácsokon

BSc Szakdolgozat

Témavezető:

Király Tamás

Operációkutatási Tanszék



Budapest, 2016

Köszönetnyilvánítás

Ezúton szeretném megköszönni témavezetőmnek, Király Tamásnak aki mindig segítségemre volt a szakdolgozat írása során. További köszönetet szeretnék még mondani családomnak és barátaimnak akik egyetemi éveim alatt támogattak és segítettek.

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. Bevezetés

Az útkereső algoritmusok napjainkban egyre több területen kerülnek felhasználásra. Ennek megfelelően egyre több algoritmust fejlesztenek ki, hogy mindig a problémának megfelelő legyen. Az eredeti Dijkstra algoritmust sok irányba továbbfejlesztették és jelenleg az A^* algoritmus a legelterjedtebb a gyakorlatban. A két algoritmus között az a különbség, hogy A^* egy módosított súlyfüggvényt használ a keresés során, ami nem csupán a kezdőponttól vett távolságot veszi számításba, hanem azt is, hogy a végponttól milyen távolságban van a pont amit vizsgálunk. Ezért az A^* általában lényegesen gyorsabban találja meg a legrövidebb utat. Ebben a szakdolgozatban egy olyan problémáról lesz szó, amikor a térképünk egy négyzetrácsos háló, ahol az élek egyenlő hosszúak és ezen a térképen vannak járható és járhatatlan pontok (akadályok). A probléma speciálisnak tűnik, de a gyakorlatban egyre többet kerül elő. Leggyakrabban a mesterséges intelligencia kutatásában, illetve számítógépes játékokban ahol a térképet így szokták implementálni.

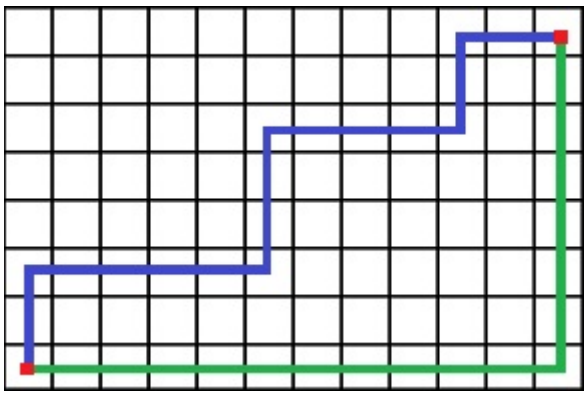
Dijkstra Algoritmus:
Edsger W. Dijkstra 1956-ban publikálta az algoritmust, ami róla lett elnevezve. Az eredeti verzió két pont közötti legrövidebb utat állapítja meg, de gyakori az a változat is, ami egy kezdőpontból megkeresi az összes pontba a legrövidebb utat. Adott a csúcsok és az élhalmaz, és az élekre van egy hosszfüggvény. Az algoritmus inícializálásához alkotunk egy Q csúcshalmazt. Ezek a csúcsok azok amiknek még nem számítottuk ki a hozzá tartozó legrövidebb utat. Kezdetben a távolságukat végtelenre állítjuk (kivéve a kezdőpontét, azt 0-ra) , mivel nem tudjuk milyen hosszú lesz a hozzájuk tartozó legrövidebb út. Számon tartjuk azt is melyik csúcsból jutunk el az eddig talált legrövidebb úton egy csúcsba, kezdetben ezt ismeretlennek állítjuk be. Az algoritmus során egy ciklus, addig amíg van még csúcs Q-ban vagy megtaláljuk a célpontot, mindig kiválasztja a legkisebb súlyú u pontot Q-ból, majd ellenőrzi u szomszédainak a súlyát és ahol tud javít rajta.

Dijkstra Pseudo Kód:
Data: G Gráf a csúcsok halmaza, s a kezdőpont, g a végpont

```
create set of vertices Q;          /*A csúcsok amikre az algoritmus még nem terjeszkedett*/
for ∀ v ∈ G do
    dist[v] ← INFINITY;                /*A csúcsok kezdeti súlyát végtelenre állítjuk*/
    prev[v] ← UNDEFINED;               /*Számon tartjuk minden csúcsnak a megelőzőjét*/
    add v to Q;                    /*Feltöltjük Q-t. Kezdetben minden csúcs az eleme lesz*/
end
dist(s) ← 0;                              /*A kezdő csúcs súlya 0 az algoritmus indításakor*/
while ∃ u ∈ Q do
    u ← min[dist(u)];              /*A legkisebb súlyú csúccsal terjeszkedik az algoritmus*/
    if u = g then
        return 0;                  /*Ha megtaláltuk a célpontot nem kell tovább keresnünk*/
    end
    for ∀ v ∈ neighbours(u) do
        if dist[u] + length(u, v) < dist[v] then
            dist[v] ← dist[u] + length[u, v]; /*Az új csúcs szomszédainak súlyát frissítjük*/
            prev[v] ← u;               /*Az új csúcs szomszédainak megelőzőjét frissítjük*/
        end
    end
    remove u from Q;
end
return dist[ ]; prev[ ];
```

1.2. Alap probléma

Tehát az alap probléma, hogy A^* túl sok felesleges lépést tesz a keresés során, nem optimális. Ennek fő oka, hogy a rácshálón sok a szimmetrikus egyenlő hosszú út, és ez nagyon lelassítja a keresést.



A probléma megértésére be kell vezetnünk néhány definíciót:

1.2.1. Definíció. ***Út:*** Vektorok sorozata, amik egy négyzetről egy másik szomszédos négyzetre mutatnak.

1.2.2. Definíció. ***Szimmetria:*** Azt mondjuk két út szimmetrikus, ha a kezdő és végpontjuk megegyezik és a vektorok sorrendjének megváltoztatásával kihozható az egyik a másikból.

1.2.3. Definíció. ***Járható út:*** Egy utat járhatónak vagy helyesnek nevezünk, ha minden pontja járható.

A gyorsításra több ötlet is született:

1. A keresés helyének csökkentése absztrakcióval. Ezek általában gyorsak és kevés memóriát igényelnek de gyakran csak egy közelítő megoldást adnak.
2. Heurisztikus függvények pontosítása. Előre kiszámoljuk és eltároljuk kulcspontok közötti távolságokat. Ez nagyon nagy többlet memória igénnyel járhat.
3. Zsákutca keresés és egyéb állapot-hely metsző metódusok. Ezek célja olyan területek megtalálása amik felfedezése nem szükséges. Ezeknek kicsi a többlet memória igénye, de nem is gyorsítanak sokat.

1.3. A JPS és RSR algoritmusok

A továbbiakban két algoritmusról és azok esetleges módosításairól lesz szó. Mindkét esetben a keresést lassító szimmetriát szeretnénk valahogy kezelni. Az RSR (Rectangular Symmetry Reduction) és a JPS (Jump Point Search) nem területeket keres amiken nem kell végigmennie, hanem olyan szimmetrikus csomópontokat keres és metsz ki, amik lassítják a keresést. Az első részben az RSR-ről lesz szó ami egy előfeldolgozó algoritmus és a keresési tér módosításával fog gyorsítani. Ennek két változata amikor 4-csatlakozású, illetve 8-csatlakozású a háló. 4-csatlakozásúnak fogjuk nevezni amikor csak egyenes lépéseket engedünk meg (fel-le-jobbra-balra) és 8-csatlakozásúnak amikor az átlós lépések is megengedettek. A második részben pedig a JPS algoritmusról ami az A^* módosítása és a keresés során ugró pontok segítségével fog tudni nagyobb mértékű gyorsulást elérni.

Forrás:
Szakdolgozatom alapjául az alábbi négy cikk szolgált: [1], [2], [3], [4].

2. fejezet

RSR 4-csatlakozású rácshálón

2.1. Az RSR algoritmus alap ötlete

Az RSR (Rectangular Symmetry Reduction) algoritmust először egy 4-csatlakozású rácshálóra fogjuk definiálni. Az alap ötlet az, hogy úgy csökkentjük a keresés során megtalálható szimmetrikus utakat, hogy azonosítunk négyszögeket, amiknek csak a külsejéből terjeszkedünk és sohasem a belsejéből. Az algoritmus a keresési tér előfeldologzásából áll alapvetően, ezért valamennyi többlet memória használattal jár. Mivel az algoritmusunk offline, ezért szinte bármelyik másik útkereső algoritmussal is kombinálható további gyorsulásért.

Videójátékokban gyakran feladat nagy terek, földalatti alagutak vagy szobák felfedezése, ilyenkor sok a szimmetrikus út amiket az RSR jól tud szűkíteni. Később észrevehetjük majd, hogy minél több az akadálymentes tér, annál többet gyorsít a keresésen az RSR.

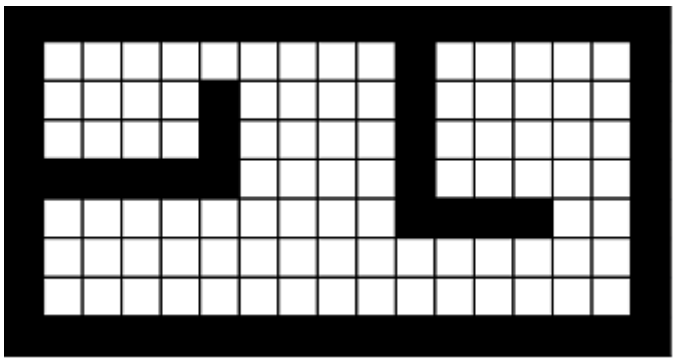
2.2. Keresés előtti szimmetria csökkentés

A videójátékokban előforduló üres szobák vagy terek sok szimmetrikus utat jelentenek a keresés során. Ha ezeket nem kezeljük jól, akkor nagy valószínűséggel növelni fogja a keresés idejét. A gráfunk átalakításával csökkentjük a szimmetriák számát. Csúcsokat hagyunk el az előfeldolgozás során és ezzel csökken a szimmetrikus utak száma. A keresés optimalitásának megőrzéséhez a csúcsok elhagyása után makró-éleket kell hozzáadnunk. Mivel a 4-csatlakozású RSR során csak egyenes lépések megengedettek, a makró-élek hosszához Manhattan-távolság függvényt fogunk használni.

2.2.1. Definíció. ***Makró-él:*** A gráfban eredetileg nem szereplő élek, amiket a keresés optimalitásának megőrzése érdekében helyezünk be a gráfunk átalakítása során.

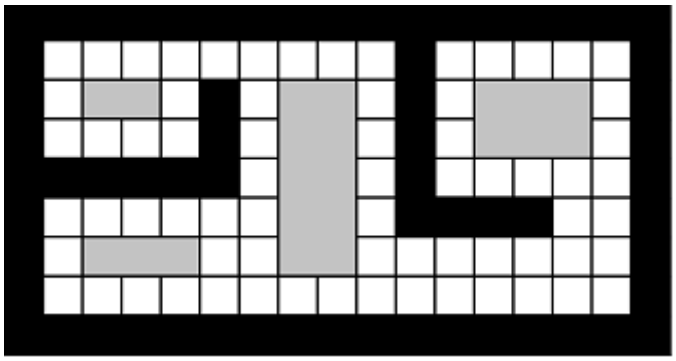
2.2.2. Definíció. ***Manhattan-távolság:*** 1-normán alapuló távolság: $\sum_{i=0}^{n} |x_i$ - $y_i|$. Ahol a két csúcs: $X(x_1, ..x_n)$ és $Y(y_1, .., y_n)$.

A következő módon fogjuk azonosítani és kiszűrni a szimmetrikus utakat a 4-csatlakozású rácshálón:

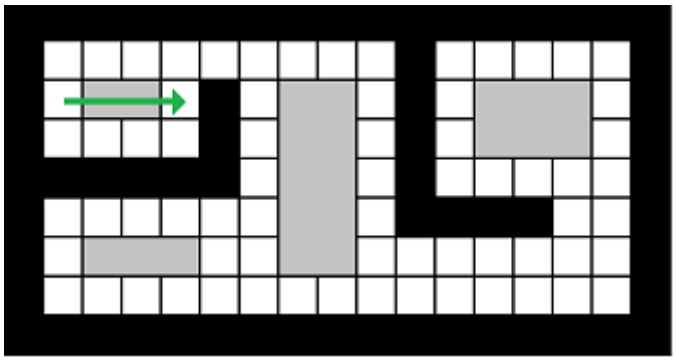


1. Felbontjuk a rácshálót üres szobákra. Ezek a szobák négyszögletűek és nem tartalmaznak akadályt. Az akadályok helyeitől függően a szobák mérete változó lehet.

2. Minden üres szoba belső pontjait kimetszük, a külső pontjait pedig érintetlenül hagyjuk.



3. Makró-éleket helyezünk el a szobák belsejében. Ezek az élek kötik össze az egymással szemben lévő szélső pontokat (A szobák szélein lévő pontok). Az ilyen makró-élek hossza megegyezik a két végpont közti Manhattan távolsággal.



Azokat a szobákat amiknek a magassága vagy a szélessége maximum kettő, a második és harmadik lépés nem módosítja. Ezeknek a szobáknak nincs belső pontja, amit ki lehetne metszeni. A nagyobb

szobák belső pontjainak kimetszésével csökkentettük a szimmetrikus utak számát a külső pontok között. Azt állítjuk, ez a módszer megőrzi az optimalitást, amikor egy tetszőleges szobában legrövidebb utat keresünk.

2.2.3. Lemma. *Legyen R egy tetszőleges négyszögletes szoba, amiben nincsenek akadályok. Valamint legyen m és n egy-egy pont a szoba szélén. Ekkor m és n optimálisan összeköthető olyan úttal, ami csak szélén lévő pontokból és egy makró-élből áll.*

Bizonyítás: Két különböző esetet kell megvizsgálnunk a bizonyítás során:

1. Amikor m és n ugyanazon az oldalon vagy két egymásra merőleges oldalon van. Ekkor optimálisan juthatunk el m-ből n-be a szoba szélső pontjain.

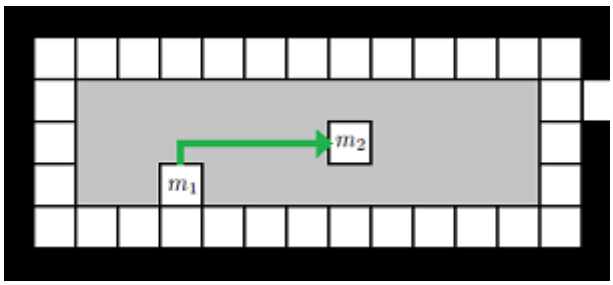
2. Ha m és n két egymással szemben lévő oldalán van a szobának. Optimális út megtalálásához induljunk el m-ből egy makró-élen ahonnan m'-be jutunk ami már n-el azonos oldalon van. Innen a szoba szélén eljutunk n-be. Ez a távolság optimális egy négycsatlakozású négyzetrács hálón. □

Az 2.2.3. lemma közvetlen következménye, hogy a továbbiakban a kimetszett belső pontokat nem kell vizsgálnunk, csak a szélső pontokat. Csupán azt az esetet kell még megfontolni, amikor a kezdő vagy a végpont egy kimetszett belső pont.

2.3. Keresés alatti beillesztés

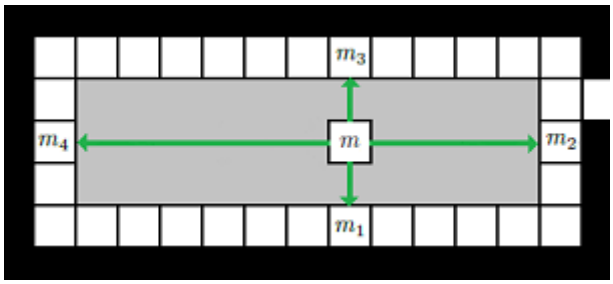
Előfordulhat, hogy egy kimetszett pontra szükség van egy kezdő vagy egy végpont miatt. Ekkor ideiglenesen visszahelyezzük a pontokat a keresés idejére.

1. Ha egy szobában van a kezdő és végpont, nem szükséges a pontok vissza helyezése. Mivel garantáltan nincs a két pont között akadály, a legrövidebb út triviálisan megtalálható.



2. Egyébként m-et a legközelebbi szélsőkhöz kötjük.

2.3.1. Lemma. *Legyen R egy üres négyszögletes szoba. Bármely m, n-hez ahol m egy újra beillesztett pont és n egy szélső pont, mindíg található optimális út úgy, hogy m-en kívül nem érintünk belső pontot.*



Bizonyítás: Visszahelyezzük m-et és csatlakoztatjuk m_1, m_2, m_3, m_4-hez, amik a legközelebbi szomszédok a szoba 4 szélén. Ezek távolsága m-től a Manhattan távolságuk. Az optimális út m-ből n-be az, ha elindulunk m-ből m_i-be (Ahol m_i azonos oldalon van n-el). Innentől R szélén eljutunk n-be. Ennek az útnak a távolsága optimális m és n között. □

A keresés végén ismét eltávolítjuk a keresés idejére ideiglenesen visszahelyezett pontokat.

2.4. Optimalitás

2.4.1. Állítás. *A korábban leírt metsző eljárások után is A^* egy optimális hosszúságú utat fog találni, amennyiben az létezik.*

2.4.2. Tétel. *Minden optimálisan hosszú $\pi^*(s,g)$ úthoz egy 4-csatlakozású négyzetrács hálón, van egyenlő hosszú út a metszett térképen is.*

Bizonyítás: Következik a 2.2.3. és 2.3.1. lemmákból. Felbonthatjuk $\pi^*(s,g)$-t optimális hosszú szakaszokra minden szobán amin áthalad. Ekkor minden m szélső pontból n szélső pontba haladó optimális útnak van egy megfelelője ami csak abban a szobában lévő szélső pontokat és maximum 1 makró-élt tartalmaz. □

Ennek következménye, hogy az RSR segítségével talált legrövidebb út könnyen átalakítható. (Például ha nem akarjuk, hogy az út végig a falat kövesse.)

2.5. Az RSR algoritmus pszeudokódja

Az inícializáció során a térképet megfeleltetjük egy mátrixnak. Minden csúcshoz hozzárendelünk egy koordinátát úgy, hogy a bal felső sarok lesz a (0,0). Továbbá minden csúcshoz egy állapot jelzőt, hogy nyomon követhethessük annak a csúcsnak a státuszát. A három esete:

1. Ha járhatatlan a csúcs, akkor legyen az értéke -1.
2. Ha járható, és még nincs szobához rendelve, akkor 1.
3. Ha járható, de már egy szobához van rendelve, akkor 0.

Ez a szoba építésnél fontos, hogy a szoba akadálymentes legyen és olyan csúcsokból álljon csak amik még nem lettek másik szobához rendelve. A szobákat a bal felső és a jobb felső csúcsokkal tároljuk el. A szoba építés során a téglalapokat egy maximum kupacba tesszük ahol a prioritásuk a belső pontjaik száma lesz.

RSR algoritmus:
Data: G Gráf a csúcsok halmaza
create T maxheap; `/*Maximum kupac a négyszögek tárolására*/`
create R set; `/*A szobák halmaza*/`
for $\forall\ v \in G$ **do**
  add $maxrectangle(v)$ *to* T; `/*Maximális prioritású négyszögek keresése*/`
end
while $\exists\ v \in G$ *with* $condition(v) = 1$ **do**
  maxroom(); `/*Maximális méretű szobák kiválasztása*/`
end
for $\forall\ r \in R$ **do**
  macroedge(r); `/*Makró-élek behelyezése a szobákba*/`
end

2.6. Szobák elkészítése

Az algoritmus során kulcsfontosságú a szobák kiválasztása, mivel ez határozza meg, hogy mennyit javít az RSR a keresés sebességén. Mivel minél nagyobb szobákat akarunk kiválasztani, ezért a nagyobb szobákat építjük a kisebbek előtt, az alapján, hogy mennyi belső pontot tartalmaz a szoba.

Szoba építés algoritmus:

1. Minden egyes járható t ponthoz építsünk egy maximális méretű négyszöget úgy, hogy t a bal felső sarka. Minden ilyen négyszög csak olyan járható pontot tartalmazhat, ami még nincs egy szobához se rendelve.
2. Maximális kupac segítségével listázzuk a járható t pontokat úgy, hogy a t-hez hozzá rendelt négyszögek belső pontjainak számát vesszük a prioritásának.
3. Vegyük ki a legnagyobb prioritású pontot ami még nincsen szobához rendelve.
4. Ellenőrizzük a prioritását t-nek úgy, hogy az 1. lépésben leírt módon újra építünk egy maximális méretű négyszöget.
5. Ha megegyezik akkor a szobák közé tesszük, egyébként frissítjük a prioritását.
6. Ismételjük a 3.-5. lépéseket amíg a kupac nem üres.

Látható, hogy A^* gyorsulása nagyban függ a kimetszett pontok számától. Ezért a nagy szobák felismerése kritikus. Erre az algoritmusunk nem optimális, de egyszerű és a gyakorlatban jó eredményeket ad.

A négyszögek építésére a $maxrectangle(v)$ algoritmust használjuk, a maximális priorítású kiválasztására pedig a $maxroom()$ algoritmust:

maxrectangle(v):
Data: v a bal felső csúcs amihez a lehető legnagyobb négyszöget építjük
$v_2 \leftarrow v$; /*A jobb alsó sarok a v_2 lesz*/
$maxborder \leftarrow infinity$; /*Számon tartjuk a legkorábbi jobb oldali akadályt*/
$maxt \leftarrow (0, v, v_2)$; /*A maximális prioritású négyszög. Két csúcsával azonosítjuk*/
while $maxborder \geq 3$ **do**
  if $condition(v_2) = 1$ *and* $x(v_2) < maxborder$ **then**
    if $priority(v, v_2) > prior(maxt)$ **then**
      $maxt \leftarrow (priority(v, v_2), v, v_2)$; /*Ellenőrizzük, hogy nagyobb négyszöget találtunk-e*/
    end
    $v_2 \leftarrow (x(v_2 + 1), y(v_2))$; /*A jobb alsú csúcsot tovább léptetjük jobbra*/
  if $x(v_2) = maxborder$ **then**
    $v_2 \leftarrow (x(v), y(v_2) + 1)$; /*Ha elértük a maxborder-t akkor a következő sorban folytatjuk a keresést*/
  if $condition(v_2) \neq 1$ **then**
    $maxborder \leftarrow x(v_2)$; /*Ha akadályt találtunk, hozzá igazítjuk a maxborder-t*/
    $v_2 \leftarrow (x(v), y(v_2) + 1)$; /*A következő sorban folytatjuk a keresést*/
  end
end
return *maxt;* /*A függvény a legnagyobb talált négyszöget adja vissza*/

A $maxrectangle(v)$ algoritmust egy v csúcsra meghívva, felépít egy olyan négyszöget aminek a bal felső csúcsa a v. Továbbá a szoba minden pontja járható és egyetlen pontja sincs már egy szobához rendelve, ez a maxroom() algoritmus miatt fontos. Az algoritmus a bal felső sarokból indulva jobbra terjeszkedik egészen addig amíg egy akadályba nem ütközik, vagy el nem éri a maxborder-t. Ezt az alapján állítjuk be az algoritmus futása során, hogy milyen távolságra találtuk az első akadályt. Ilyenkor az algoritmus a következő sorban folytatja a keresést elölről. Ez egészen addig történik, amíg akadályt nem találunk az első vagy a második oszlopban, ugyanis ezután már nem tudnánk javítani a maximális prioritású négyszögön. (A v oszlopát és sorát tekintjük az elsőnek)

Az algoritmus futása során 3 esetet nézünk a jobb alsó sarok helyzetétől függően:

1. Ha a csúcs szabad és még nem értük el a határt, akkor ellenőrizzük a négyszög prioritását, majd tovább lépünk jobbra.
2. Ha a elértük a maxborder-t akkor a következő sorban kezdjük elölről a keresést.

3. Ha a csúcs nem járható, akkor frissítjük a maxborder értéket és a következő sorban folytatjuk a keresést.

maxroom():

Data: T a négyszögek maximum kupaca, R a szobák halmaza
$r \leftarrow maxpriority(t)$; ; /*A legnagyobb prioritású négyszög kiválasztása, $t \in T$ */
if $\forall\ v \in r, condition(v) = 1$ **then**
  add r to R; /*Ha a négyszög minden pontja megfelelő, hozzá adjuk a szobákhoz*/
  for $\forall\ v \in r$ **do**
    $condition(v) \leftarrow 0$; /*A szoba pontjainak állapotát átírjuk*/
  end
else
  $t \leftarrow maxrectangle(cord_2(r))$; /*Ha a négyszögnek van már szobához adott pontja újra építjük*/
  if $priority(r) = priority(t)$ **then**
    add t to R; /*Ha nem változott a prioritás, akkor hozzá adjuk a szobákhoz*/
    for $\forall\ v \in t$ **do**
      $condition(v) \leftarrow 0$; /*A szoba pontjainak állapotát átírjuk*/
    end
  else
    add t to T; /*Egyébként az új négyszöget hozzá adjuk a maximum kupachoz*/
  end
end
remove r from T; /*Az eredeti négyszöget eltávolítjuk a kupacból*/

A *maxroom*() algoritmus kiválasztja a kupacból a legnagyobb prioritású szobát. Ellenőrzi, hogy van-e olyan pontja ami már korábban egy szobához lett rendelve. Ha nincs, akkor kiválasztja szobának és a csúcsainak az állapotát átállítja. Amennyibben van csúcsa, ami már egy másik szobához lett rendelve, az algoritmus meghívja a négyszög bal felső sarkára a $maxrectangle(v)$ függvényt és új négyszöget épít. Ha az újjáépített négyszögnek nem változik a prioritása akkor kiválasztja szobának. Ha változott, akkor visszarakja a kupacba az új négyszöget. Az algoritmus végén törli a kupacból az eredetileg kiválasztott négyszöget, mivel azt vagy kiválasztottuk szobának vagy újjáépítettük a hozzá tartozó négyszöget.

A $maxrectangle(v)$ és *maxroom* algoritmusok a 4-csatlakozásos és 8-csatlakozásos esetben is ugyanúgy működnek ezért nem kell rajta módosítanunk. Ha engedünk átlós lépéseket is, akkor az optimalitás megőrzése érdekében módosítanunk kell a $macroedge(r)$ algoritmusunkon.

Az előfeldolgozás során a *macroedge*(r) algoritmust minden szobára meghívjuk. A *macroedge*(r) behelyezi a makró-éleket a szobákba, hogy a keresés során az optimalitás továbbra is fennáljon. Az algoritmus hosszúsága miatt három részre bontva:

macroedge(r):
Data: r a szoba amibe behelyezzük a makró-éleket
$v_1 \leftarrow r$ bal felső csúcsa;
$v_2 \leftarrow r$ jobb alsó csúcsa;
for $(i = y(v_1) + 1$ **to** $y(v_2) - 1,\ i++)$ **do**
  create $< (x(v_1), i), (x(v_2), i) >$; `/*A vízszintes makró-élek létrehozása*/`
end
for $(i = x(v_1) + 1$ **to** $x(v_2) - 1,\ i++)$ **do**
  create $< (i, y(v_1)), (i, y(v_2)) >$; `/*A függőleges makró-élek létrehozása*/`
end

A *macroedge*(r) algoritmus első része az egyenes makró-éleket helyezi be. Ha az algoritmusunkat 4-csatlakozású rácshálón akarjuk futtatni akkor az első része kell csupán. A *create* függvény létrehozz a két csúcs között egy élt aminek a hosszát is egyből beállítja. A 4-csatlakozású esetben a két csúcs Manhattan-távolsága lesz a makró-élek hossza. A következő fejezetben a 8-csatlakozású esetre is kibővítjük az algoritmust, ekkor a *macroedge*(r) algoritmus további részei helyezik be az átlós makró-éleket.

Forrás:
A fejezet nagy része az [1] cikk alapján íródott.

3. fejezet

RSR 8-csatlakozású rácshálón

3.1. Az RSR algoritmus kibővítése

Az előző fejezetben tárgyalt RSR algoritmust ebben a fejezetben kiterjesztjük arra az esetre, ha a rácshálón az átlós lépések is megengedettek, továbbá néhány optimalizáló módszert is megvizsgálunk. Ennek megfelelően új távolság függvényt kell használnunk az átlós makró-élek miatt.

3.1.1. Definíció. ***Octile-távolság:*** $z = max(x, y) + (\sqrt{2} - 1) * min(x, y)$. Ahol z az átlós él hossza, x és y pedig az átlós él merőleges vetületeinek hossza.

8-RSR Algoritmus:

1. Felosztjuk a térképet diszjunkt, akadálymentes négyszögekre (szobákra), a korábban leírt **Szoba építés algoritmussal.**
2. Metszük ki a belső pontokat. (Optimálisabb eredményt kaphatunk ha néhány külsőt is kimetszünk, ennek szabályáról egy későbbi részben írok.)
3. Minden R négyszögben helyezzünk el makro-éleket a kiválasztott külső pont párok között. Minden ilyen él súlya legyen az Octile-távolsága a kezdő és végpontnak. (Manhattan-távolság ha 4-csatlakozású)
4. Keresés során ideiglenesen visszaillesztünk pontokat, hogy kezeljünk olyan eseteket is, amikor a cél vagy az induló pont ki lett metszve.

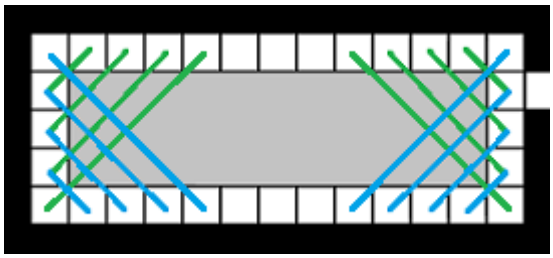
3.2. Makró-élek 8-csatlakozású rácshálón

A belső pontok eltávolítása után, makró-éleket kell behelyezni. Csak nem-dominált makró-éleket szeretnénk behelyezni.

3.2.1. Definíció. ***Nem-dominált él:*** Olyan él aminek a hossza szigorúan kisebb bármelyik másik lehetséges útnál ugyanarra a pontpárra.

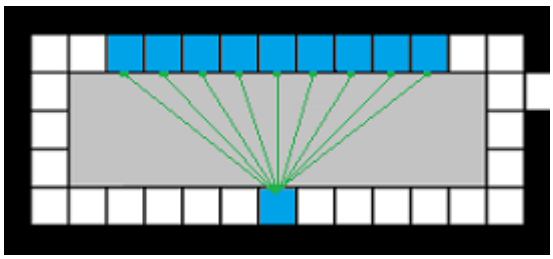
Három eset lehetséges:

1. Egymás mellett lévő oldalon van a két pont. Ekkor csak akkor van összekötve, ha a legrövidebb út köztük egy átlós (45°-os) egyenes. A pseudo kód a *macroedge*(r) algoritmus második része.



if $|x(v_1) - x(v_2)| \geq |y(v_1) - y(v_2)|$ **then**
  for $(i = y(v_1)$ **to** $y(v_2) - 1,\ i++)$ **do**
    $create < (x(v_1), i), (x(v_1) + y(v_2) - i, y(v_2)) >;$ /*A bal és az alsó oldal összekötése makró-éllel*/
    $create < (x(v_2), i), (x(v_2) - y(v_2) + i, y(v_2)) >;$ /*A jobb és az alsó oldal összekötése makró-éllel*/
  end
  for $(i = y(v_2)$ **to** $y(v_1) + 1,\ i--)$ **do**
    $create < (x(v_1), i), (x(v_1) - y(v_1) + i, y(v_1)) >;$ /*A bal és a felső oldal összekötése makró-éllel*/
    $create < (x(v_2), i), (x(v_2) + y(v_1) - i, y(v_1)) >;$ /*A jobb és a felső oldal összekötése makró-éllel*/
  end
else
  for $(i = x(v_1)$ **to** $x(v_2) - 1,\ i++)$ **do**
    $create < (i, y(v_1)), (x(v_2), y(v_1) + x(v_2) - i >;$ /*A felső és a bal oldal összekötése makró-éllel*/
    $create < (i, y(v_2)), (x(v_2), y(v_2) - x(v_2) + 1) >;$ /*A felső és a jobb oldal összekötése makró-éllel*/
  end
  for $(i = x(v_2)$ **to** $x(v_1) + 1,\ i--)$ **do**
    $create < (i, y(v_1)), (x(v_1), y(v_1) - x(v_1) + i) >;$ /*Az alsó és a bal oldal összekötése makró-éllel*/
    $create < (i, y(v_2)), (x(v_1), y(v_2) + x(v_1) - i) >;$ /*Az alsó és a jobb oldal összekötése makró-éllel*/
  end
end

2. A pontok szemköztiek. Minden ilyen ponthoz készítünk egy háromszög alakzatot a szemközti pontokkal. Először kiválasztjuk a szemközti pontot, majd ezt bővítjük mindkét irányba, addig amíg az 45°-ot nem zár be, vagy nem egy sarokpont. A pseudo kód a *macroedge*(r) algoritmus harmadik része.



for $(i = x(v_1)$ **to** $x(v_2),\ i++)$ **do**
  $j = i - 1$;
  while $j \geq x(v_1)$ *and* $angle[(j, y(v_2)), (i, y(v_1)), (i, y(v_2))] \leq 45°$ **do**
    $create < (j, y(v_2)), (i, y(v_1)) >$; /*A jobb és a bal oldal összekötése makró-éllel*/
    $j--$;
  end
  $j = i + 1$;
  while $j \leq x(v_2)$ *and* $angle[(j, y(v_2)), (i, y(v_1)), (i, y(v_2))] \leq 45°$ **do**
    $create < (j, y(v_2)), (i, y(v_1)) >$; /*A jobb és a bal oldal összekötése makró-éllel*/
    $j++$;
  end
end
for $(i = y(v_1)$ **to** $y(v_2),\ i++)$ **do**
  $j = i - 1$ **while** $j \geq y(v_1)$ *and* $angle[(x(v_2), j), (x(v_1), i), (x(v_2), i)] \leq 45°$ **do**
    $create < (x(v_2), j), (x(v_1), i) >$; /*A felső és az alsó oldal összekötése makró-éllel*/
    $j--$;
  end
  $j = i + 1$ **while** $j \leq y(v_1)$ *and* $angle[(x(v_2), j), (x(v_1), i), (x(v_2), i)] \leq 45°$ **do**
    $create < (x(v_2), j), (x(v_1), i) >$; /*A felső és az alsó oldal összekötése makró-éllel*/
    $j++$;
  end
end

3. A két pont azonos oldalon van. Ekkor ugyanúgy vannak összekötve, mint az eredeti gráfban, nem szükséges makró-él hozzá adása.

A többi pont elérhető a természetes élek és maximum 1 makró-él bevonásával.

A két algoritmus a *macroedge*(r) algoritmus kiterjesztése a 8-csatlakozású RSR-re. Ha engedjük az átlós lépéseket, akkor az optimalitás megőrzéséhez újabb makró-éleket is fel kell vennünk az előfeldolgozás során. A *create* függvény egy élt hoz létre a két csúcs között, aminek a hossza a két csúcs Octile-távolsága lesz. Az első esetben, amikor két szomszédos oldalt kötünk össze, megnézzük, hogy a téglalapunknak a szélessége vagy a magassága-e a nagyobb. Majd a rövidebb oldalakon végigmegyünk és összekötjük a csúcsait a szomszédos oldalak csúcsaival. A második esetben elég két oldal csúcsain (egy függőleges és egy vízszintes oldal) végigmenni, hogy az összes szükséges makró-él bekerüljön a szobába.

3.2.2. Lemma. *Legyen R egy üres szoba egy 8-csatlakozású négyzetrácshálón továbbá m és n két külső pont. Ekkor m és n összeköthető optimálisan egy olyan úton amiben nincsenek dominált makró-élek.*

Bizonyítás: A bizonyítás a korábban tárgyalt 3 esetre bontható:

1. Amennyiben m és n azonos oldalon találhatók. m-ből ellépünk n-ig szélső pontokon. Az optimalitás egyértelmű.
2. Ha m és n szomszédos oldalon van. A két pontot összeköthetjük egy optimális makró-éllel, és egyéb szélső csúcsokat összekötő élekkel.
3. Ha m és n szemközti oldalon van. Ez az eset az előzőhöz hasonlóan kezelhető. Egyik esetben sincs szükség dominált makró-élekre. □

3.2.3. Tétel. *Minden egyes optimális π úthoz az eredeti rácshálón, létezik egy optimális π' út a módosított hálón úgy, hogy π' és π ugyanolyan költségű.*

Bizonyítás: Tekintsünk egy R szobát amin a π optimális út áthalad az eredeti rácshálón. Legyen m és n két pontja π-nek és külső potjai R-nek. Ekkora a 3.2.2. lemma szerint, van optimális út a módosított gráfban m és n között. Cseréljük ki az eredeti $< m, .., n >$ szegmenst π-ben egy vele egyenlő költségűre a módosított rácshálón. Az az eset amikor m (vagy n) egy kezdő vagy végpont hasonlóan kezelhető. Ha ezt a cserét végrehajtjuk minden olyan négyszögre amin áthalad π, kapunk egy π'-t ami kielégíti az eredeti feltételeket. □

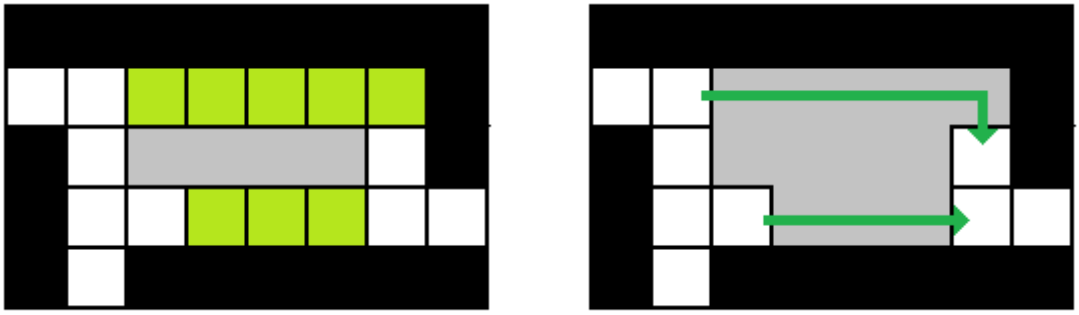
3.3. További ötletek a keresés gyorsítására

Online külső pont elimináció

Ha egy pont és a szülője azonos szobába tartozik, akkor a keresés során nem szükséges a szemközti oldalt új pontért vizsgálni, mivel ilyenkor a szülő csúcsból optimálisabban elérhető a másik oldal. Ha az aktuális csúcsnak nincs szülője vagy az egy másik szobához tartozik, akkor az összes csúcsot meg kell vizsgálni.

Offline külső pont metszés

Elhagyhatók azok a külső pontok, melyek nem kapcsolódnak másik szoba pontjához. Ilyenkor az optimallitás megőrzéséhez a metszett pontok szomszédait összekötjük. Az él hossza az Octile-távolságuk lesz. Az ábrán látható módon újabb makró-éleket adunk a szobához. Ezzel a módszerrel sok pontot tudunk lemetszeni így az algoritmusnak a keresés során lényegesen kevesebb csúcsra kell kiterjednie.



Memória szükséglet

Az RSR-nek szüksége van egy memória többletre ami lineáris a gráf méretével. Eltároljuk minden ponthoz a szülő négyszög (szoba) azonosítóját. Minden szobának eltároljuk a bal felső és a jobb alsó csúcsát. A további memória igény elkerülhető. A makró-élek kiszámíthatóak az algoritmus futása közben is a négyszög geometriai tulajdonságait kihasználva.

3.4. Előnyei és hátrányai

Az RSR a gyakorlatban jól használható egyszerű algoritmus, és akkor eredményezi a legnagyobb gyorsulást ha nagy üres területek vannak. Megtartja az optimalitást. Az A^*-al kombinálva jelentős gyorsulás érhető el és a pont beillesztés (abban az esetben ha a kezdő vagy a célpont ki lett metszve) konstans idő alatt elvégezhető. Mivel ez egy előfeldolgozó algoritmus, szükséges memória többlet felhasználása, de ez a gyakorlatban nem nagy ezért jól használható.

Forrás:

A fejezet nagy része a [2] cikk alapján íródott.

4. fejezet

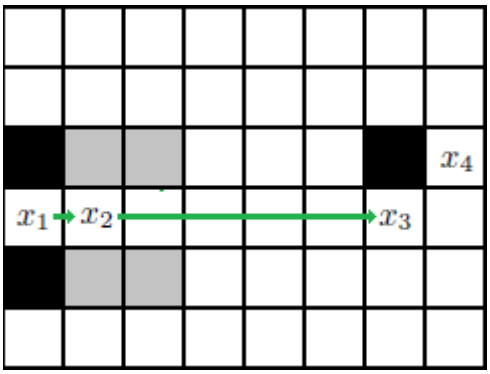
JPS algoritmus

4.1. A JPS algoritmus alapjai

A JPS (Jump Point Search) algoritmus az A^* algoritmus fejlesztése. Az alapötlet, hogy úgy csökkenti a keresés során a szimmetrikus utak miatti többlet keresést, hogy úgynevezett ugrópontokat azonosít. A Dijkstra algoritmusban, amikor a legkisebb súlyú csúcsot bevesszük azon csúcsok halmazába, amiknek a súlyát már nem változtatjuk, az új csúcs szomszédainak súlyát újra számoljuk. A JPS e helyett a lépés helyett, lemetszi a természetes szomszédait az új csúcsnak és azokat úgrópontokkal helyettesíti. Ezt egy rekurzív algoritmussal éri el, ami két metszési szabályt alkalmaz. Az algoritmust 8-csatlakozású négyzetrácsos hálón vizsgáljuk. Ez azt jelenti, hogy minden csúcsnak 8 szomszédja van ezek vagy járhatóak vagy nem. Az átlós élek hossza $\sqrt{2}$, az egyenes élek hossza pedig 1.

Alapvető jelölések:
A $\vec{d}$ valamely járható irány a 8-ból. Azt mondjuk $y = x + k\vec{d}$ ha y elérhető x-ből ha $\vec{d}$ irányba lépünk k-szor. Ha $\vec{d}$ átlós irány, akkor felbontjuk $\vec{d_1}$ és $\vec{d_2}$-re ahol azok egyenesek és 45°-al térnek el $\vec{d}$-től. A $\pi =< n_0, n_1, ..., n_k >$ út egy körmentes irányított séta ami n_0-ból indul és n_k-ban végződik. Legyen $\pi \backslash x$ olyan út aminek nem eleme x. Egy tetszőleges π út költségét jelölje $len(\pi)$, két pont távolságát pedig a $dist(n_0, n_k)$. Továbbá $p(x)$ az x csúcs megelőzőjét jelöli.

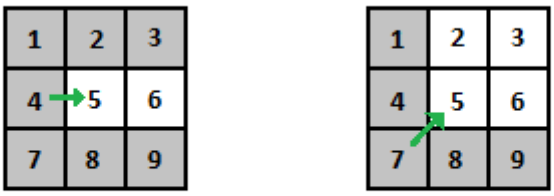
Ugrópontok alapötlete
A keresés során csak az ugrópontokra terjeszkedünk, ezért fontos, hogy ezeket jól definiáljuk. Először ábrák segítségével bemutatjuk az alapötletet és később definiáljuk pontosan a metszési szabályokat.



A keresés x_2 ponttal terjeszkedik, aminek a megelöző pontja x_1. A lépés iránya egyenes jobbra. Amikor x_2-re lépünk felesleges a szürkével jelölt szomszédokat vizsgálni, mivel azok az utak amik azokon keresztül vezetnének, dominálva vannak egy olyan úttal aminek x_2 nem eleme. Ezért csak a jobb oldali szomszédját kell vizsgálni. Ekkor ezt a szomszédot nem adjuk hozzá a listánkoz mint ahogy azt A^*-ban tennénk, hanem elindulunk abba az irányba, amíg nem találunk egy olyan pontot mint x_3 (ugró pont), aminek van legalább még 1 nem dominált szomszédja (sarok szomszéd). Ilyenkor x_3 egy rákövetkező csúcsa lesz x_2-nek és hozzá rendelünk egy g-értéket ahol $g(x_3) = g(x_2) + dist(x_2, x_3)$ (az odáig vett költség). Egyébként ha akadályba ütközünk, akkor további keresés ebben az irányban felesleges és nem adunk hozzá semmit.

4.2. Metszési szabályok

A keresés során azonosítani akarjuk azokat a szomszédos csúcsokat amiket nem kell megvizsgálnunk és ezért lemetszhetjük. Ezeket a metszési szabályokat alkalmazzuk rekurzívan az algoritmus során a keresés felgyorsításához. Két utat kell összehasonlítanunk. Ha π ami $p(x)$-el kezdődik és átmegy x-en és n-ben végződik, azzal a π' úttal ami szintén $p(x)$-el kezdődik és n-ben végződik de x nem eleme, továbbá az összes pont eleme $neighbours(x)$-nek (vagyis x szomszédai). Két lehetőséget kell megfontolni, attól függően, hogy milyen irányból lépünk x-be. (Ha x a kezdpont akkor $p(x) = NULL$ és nem metszünk semmit.)



Egyenes irány **Átlós irány**

A szürke mezők jelölik a lemetszett szomszédokat.

Egyenes irány:
Metszük ki x-nek minden olyan n szomszédját amire megfelel a következő feltétel:

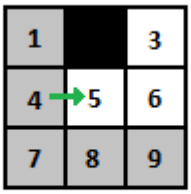
$$len(< p(x), .., n > \backslash x) \leq len(< p(x), x, n >)$$

Átlós irány:
Hasonlóan, metszük ki x-nek minden olyan n szomszédját amire megfelel a következő feltétel:

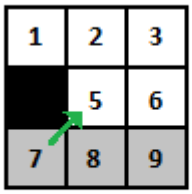
$$len(< p(x), .., n > \backslash x) < len(< p(x), x, n >)$$

4.2.1. Definíció. ***Természetes szomszéd:*** Ha x szomszédai mind járható pontok, akkor a metszés után megmaradt szomszédait természetes szomszédoknak nevezzük.

Ha x-nek nem minden szomszédja járható, lehet hogy nem tudjuk az összes nem-természetes szomszédot lemetszeni. Ilyenkor a megmaradt nem természetes szomszédot sarok szomszédnak (forced neighbour) nevezzük.



Egyenes irány



Átlós irány

Az ábrákon egyenes irány esetében a 3-as mező egy sarok szomszéd, átlós esetben pedig az 1-es mező sarok szomszéd.

4.2.2. Definíció. ***Sarok szomszéd:*** x-nek egy n szomszédja sarok szomszéd ha:

1. n nem természetes szomszédja x-nek.
2. $len(< p(x), x, n >) < len(p(x), .., n > \backslash x)$

4.2.3. Definíció. ***Ugró pont:*** Az y pont ugró pont x-ből, $\vec{d}$ irányban, ha y minimalizálja k-t úgy, hogy $y = x + k\vec{d}$ és a következő feltételek valamelyike teljesül:

1. y a célpont.
2. y pontnak legalább egy sarok szomszédja van az 4.2.2. definíció értelmében.
3. $\vec{d}$ egy átlós irány és létezik egy z pont, hogy $z = y + k_i\vec{d_i}$ ami $k_i \in \mathbb{N}$ lépésre van $\vec{d_i} \in \{\vec{d_1}, \vec{d_2}\}$ irányban úgy, hogy z egy ugrópont y-ból az első vagy a második feltétel szerint.

4.3. Ugrópontok keresése

Az ugrópontok keresése az egyik leglényegesebb része az algoritmusnak, ezt a keresést az $Identify(x, s, g)$ algoritmus végzi. Az algoritmus célja, hogy kicserélje az x csúcs eredeti szomszédait ugrópontokra.

Identify(x,s,g):
Data: x az aktuális pont, s a kezdőpont, g a célpont
$successors(x) \leftarrow 0$; /*Létrehozzuk a $successors(x)$-et*/
$neighbours(x) \leftarrow prune(x, neighbours(x))$; /*Alkalmazzuk x szomszédain a metszési szabályokat*/

for $\forall\ n \in neighbours(x)$ **do**
  $n \leftarrow jump(x, direction(x, n), s, g)$; /*Ugrás a szomszéd irányában*/
  add n to $successors(x)$; /*Ugrópont hozzáadása, amennyiben talált*/
end
return $successors(x)$;

Kezdésképp létrehozza az algoritmus a $successors(x)$-et, ami kezdéskor még üres, később ebbe gyűjti az x-hez tartozó ugrópontokat. Második lépésként a metszési szabályokat alkalmazva lemetszi a szomszédait, hogy a továbbiakban csak a természetes és a sarok szomszédokat kelljen vizsgálni. Ezután a megmaradt szomszédait nem adjuk hozzá a $successors(x)$-hez, hanem az irányukba elindulunk és megpróbálunk ugrópontokat találni. Az algoritmus addig fut, amíg a $neighbours(x)$ üres nem lesz. Előfordulhat, hogy a $successors(x)$ üres az algoritmus lefutása után, ez azt jelenti, hogy x-nek nincs ugrópont leszármazottja, ezért felesleges tovább vizsgálnunk.

jump($x, \vec{d}, s, g$):
Data: x az ugrás indulópontja, $\vec{d}$ az irány, s a kezdőpont, g a célpont
$n \leftarrow step(x, \vec{d})$;
if *n nem járható vagy nem része a térképnek* **then**
  return $NULL$; /*Sikertelen a keresés*/
if $n = g$ **then**
  return n; /*Ha a célpontot találja meg a keresés vele tér vissza*/
if $\exists\ n' \in neighbours(n)$ *úgy, hogy n' sarok szomszéd* **then**
  return n; /*Ha n ugrópont akkor őt adja vissza értékül*/
if $\vec{d}$ *átlós irány* **then**
  for $\forall i \in \{1, 2\}$ **do**
    if $jump(n, \vec{d_i}, s, g) \neq NULL$ **then**
      return n; /*Átlós irány esetén az egyenes irányok ellenőrzése */
  end
end
return $jump(n, d, s, g)$;

A $jump(x, \vec{d}, s, g)$ algoritmus hajtja végre az ugrást az ugrópontok keresése során. Egy adott $\vec{d}$ irányba elindul és kétféleképpen állhat le. Vagy ugrópontot talál, vagy zsákutcába ér a keresés.
Háromféleképp érhet véget a keresés egy adott irányban:

1. Ha n nem járható pont akkor a keresés sikertelen.
2. Ha n ugrópont, akkor azzal a ponttal fog visszatérni. n egy ugrópont leszármazottja lesz x-nek.
3. Ha n nem ugrópont és nem is akadály, akkor tovább lépünk $\vec{d}$ irányba.

Ha $\vec{d}$ átlós irány akkor a lépés előtt először meg kell vizsgálni az egyenes irányokat és ha ott nem talál ugrópontot, csak akkor lép tovább. Ez a 4.2.3. definícióból adódik és szükséges az optimalitás megőrzéséhez.

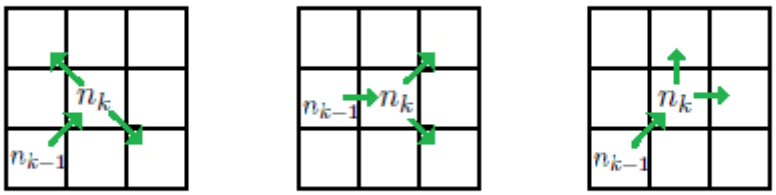
4.4. Optimalitás

4.4.1. Állítás. *A JPS optimalitás tartó keresés. Minden optimálisan hosszú úthoz létezik egy ugyanolyan hosszú út ami megtalálható úgy, hoy a keresés során csak ugró pontokon terjeszkedünk.*

Ötlet a bizonyításhoz: Minden optimális útnak vesszük egy szimmetrikus alternatíváját amit folytonos részekre bontunk. Ekkor belátható, hogy minden fordulópont egyben ugrópont is.

4.4.2. Definíció. ***Fordulópont:*** Olyan n_i pont egy úton, ahol a haladás irány az előző n_{i-1} pontból n_i-be különbözik az n_i-ből n_{i+1}-be vett haladási iránytól.

Háromféle fordulópont lehet egy optimális úton:



1. Átlósból-átlós
2. Egyenesből-átlós
3. Átlósból-egyenes

Egyébb forduló pont típus mint az egyenesből-egyenesbe triviálisan nem optimális, ezért nem kell vele foglalkoznunk. Ez a korábban bevezetett metszési szabályokból következik.

4.4.3. Definíció. ***Átlós-priorítású út (diagonal-first):*** Egy π utat átlós-priorításúnak nevezünk, ha nem tartalmaz egyenesből-átlós fordulópontot ($< n_{k-1}, n_k, n_{k+1} >$) úgy, hogy azt ki lehetne cserélni egy átlósból-egyenes fordulóponttal ($< n_{k-1}, n'_k, n_{k+1} >$) miközben π hossza nem változik.

Átlós-priorítású út algoritmus:
π egy tetszőleges optimális hosszúságú út.

1. Kiválasztunk két szomszédos élt π-n úgy, hogy (n_{k-1}, n_k) egyenes lépés és (n_k, n_{k+1}) átlós.
2. Felcseréljük n_k-t n'_k-el úgy, hogy (n_{k-1}, n'_k) átlós és (n'_k, n_{k+1}) egyenes lesz. A művelet akkor sikeres,ha (n_{k-1}, n'_k) és (n'_k, n_{k+1}) szabályos lépések, vagyis n'_k járható pont.
3. Az első és második lépést addig ismételjük amíg lehet változtatni π-n.
4. return π

4.4.4. Állítás. *Az átlós-priorítású út algoritmussal bármely optimális hosszú π út átalakítható egy átlsó-priorítású π' úttá.*

4.4.5. Lemma. *Minden fordulópont egy optimális π' átlós-priorítású úton egyben ugrópont is.*

Bizonyítás: Legyen n_k egy tetszőleges fordulópont π'-n. Három esetet kell megvizsgálni, melyek egybeesnek a korábban tárgyalt optimális fordulópontokkal.

1. Átlósból-átlós: Mivel π' egy optimális hosszúságú út, ezért lennie kell egy akadálynak n_k és n_{k-1} mellett, ami a kitérőt okozza. Ezt azért tudhatjuk, mert ha nem lenne akadály akkor a

   $$dist(n_{k-1}, n_{k+1}) < dist(n_{k-1}, n_k) + dist(n_k, n_{k+1})$$

   lenne, tehát lenne rövidebb út n_{k-1} és n_{k+1} között. Ez ellentmondás mivel π' optimális. Továbbá az is következik ebből, hogy n_{k+1} sarok szomszédja n_k-nak. A 4.2.3. definíció második feltétele miatt ekkor n_k egy ugrópont.

2. Egyenesből-átlós: Ebben az esetben egy akadálynak kell lennie n_k mellett. Ha nem így lenne n_k-t lehetne helyettesíteni egy átlósból-egyenes élekkel ami ellentmondás lenne azzal, hogy π' átlós-priorítású út. Mivel tudjuk, hogy π' átlós-priorítású út, következik belőle, hogy n_{k+1} sarokszomszédja n_k-nak. A 4.2.3. definíció második feltétele miatt ekkor n_k egy ugrópont.

3. Átlósból-egyenes: Kétféleképp lehetséges ez az eset, attól függően, hogy a cél elérhető-e n_k-ból egyenes lépések sorozatával vagy π'-nek van még további fordulópontja. Ha a cél elérhető egyenes lépésekkel, akkor n_k-nak van ugrópont leszármazottja, ami kielégíti a 4.2.3. definíció harmadik feltételét, így n_k ugrópont. Ha n_k-t követi még fordulópont, akkor annak egyenesből-átlósnak kell lennie, és ekkor a korábban látott okokból az egy ugrópont. Ekkor ismét a 4.2.3. definíció harmadik feltétele miatt n_k ugrópont. □

4.4.6. Tétel. *Az ugrópontos keresés mindig optimális utat talál. (Amennyiben létezik út)*

Bizonyítás: Legyen π egy tetszőlegesen választott optimális út két pont között és π' egy átlós-priorítású szimmetrikus változata amit az átlós-prioritású út algoritmussal készítettünk. Megmutatjuk, hogy minden fordulópont π'-n, olyan pont amire az ugrópontos keresés optimálisan kiterjeszkedik az algoritmus során.

Osszuk fel π'-t szomszédos szakaszok sorozatára úgy, hogy $\pi' = \pi'_0 + \pi'_1 + .. + \pi'_n$. Ekkor minden $\pi'_i =< n_0, n_1, .., n_k >$ egy útrészlet amin minden lépés ugyanolyan irányú. Vegyük észre, hogy a kezdő- és célpontot kivéve minden részlet kezdő- és végpontja fordulópont. Mivel minden π' csupán olyan lépésekből áll amik egy irányba tartanak (egyenes vagy átlós) ezért használhatjuk az jump() algoritmust, hogy az $n_0 \in \pi'_i$-ről ami egy részlet kezdőpontja átugorjunk $n_k \in \pi'_i$-re ami a végpontja anélkül, hogy feleslegesen megállnánk és terjeszkednénk közben. Közbülső terjeszkedések előfordulhatnak, de garantált, hogy n_k-t optimálisan érjük el n_0-ból.

Meg kell még mutatni, hogy n_0 és n_k ugrópontok és ezért az algoritmus kiterjeszkedik rájuk. A 4.4.5. lemma miatt minden fordulópont π'-n egyben ugrópont is, így minden fordulópontra kiterjed a keresés. Csupán a kezdő- és célpontokat kell még megvizsgálni. A kezdőpont az algoritmus kezdetekor kiterjesztésre kerül, a célpont pedig definíció alapján ugrópont. □

Forrás:
A fejezet nagy része a [3] cikk alapján íródott.

5. fejezet

JPS algoritmus fejlesztése

5.1. A JPS algoritmus pszeudokódja

Az algoritmus inícializációja megegyezik az A* algoritmuséval. A lényeges különbség a ciklusban van, amikor a legkisebb súlyú csúcsot kivesszük a Q halmazból.

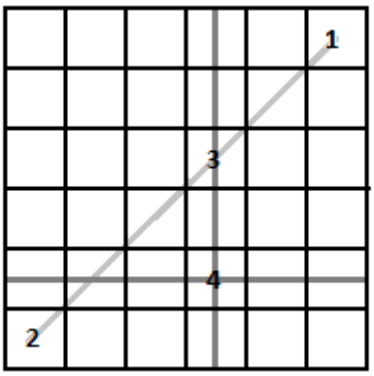
JPS algoritmus:

Data: G Gráf a csúcsok halmaza, s a kezdőpont, g a végpont
create set of vertices Q; /*A csúcsok amikre az algoritmus még nem terjeszkedett*/
for $\forall\ v \in G$ **do**
  $dist[v] \leftarrow INFINITY$; /*A csúcsok kezdeti távolságát végtelenre állítjuk*/
  $prev[v] \leftarrow UNDEFINED$; /*Számon tartjuk minden csúcsnak a megelőzőjét*/
  add v to Q; /*Feltöltjük Q-t. Kezdetben minden csúcs az eleme lesz*/
end
$dist(startingpoint) \leftarrow 0$; /*A kezdő csúcs súlya 0 az algoritmus indításakor*/
while $\exists\ u \in Q$ **do**
  $u \leftarrow min[dist(u)]$; /*A legkisebb súlyú csúccsal terjeszkedik az algoritmus*/
  if $u = g$ **then**
    return 0; /*Ha megtaláltuk a célpontot nem kell tovább keresnünk*/
  end
  $Identifysuccessors(u, s, g)$; /*Megkeressük az új csúcs ugrópont leszármazottait*/
  for $\forall\ v \in successors(u)$ **do**
    if $dist[u] + length(u, v) < dist[v]$ **then**
      $dist[v] \leftarrow dist[u] + length[u, v]$; /*Az új csúcs ugrópont leszármazottainak súlyát frissítjük*/
      $prev[v] \leftarrow u$; /*A megelőzőket frissítjük*/
    end
  end
  remove u from Q;
end

5.2. A JPS gyorsítása

Előfeldolgozás:

Észrevehető, hogy az algoritmus futási idejének nagy részét, nem a tényleges keresés, hanem az ugrópontok azonosítása teszi ki. Ezért tovább javíthatunk az algoritmusunk sebességén, ha ennek az idejét csökkentjük. Ezt különböző módokon érhetjük el. Az egyik ilyen ha az ugrás sebességét gyorsítjuk. A másik lehetőségünk az adatok előfeldolgozása. A JPS+ algoritmus ezen az ötleten alapul. Előre kiszámol minden járható csúcsból mind a 8 irányba egy ugrópontot. Ha egy irányban nem találnánk ugrópontot és a JPS nem generálna semmit, mi megjegyezzük az út végét. Mivel ilyenkor még nem ismerjük a célpontot így attól függő ugrópontokat nem fogunk találni. Az algoritmus futása során előfordulhat, hogy egy diagonális ugrásnál, nem azonosítunk egy csúcsot amiből elérhető lenne a célpont (az ábrán a 4-es csúcs). Ilyenkor mindig megvizsgáljuk, hogy áthalad-e a diagonális ugrásunk olyan soron vagy oszlopon aminek eleme a célpont, és ha igen, akkor megnézzük, hogy abból a pontból elérhető-e a célpont. Ez az eljárás elég ahhoz, hogy megőrizzük a keresés optimalitását. A JPS+ nagy előnye a gyorsasága, mivel nem kell keresnie az ugrópontokat futás közben. Az előfeldolgozásnak viszont két hátránya is van. Ha a gráfunk megváltozik, akkor újra kell generálnunk az ugrópontokat (elég lehet lokális javítás is adott esetben). Továbbá jelentős memória többletre van szükség, ugyanis minden csúcshoz eltárolunk 8 ugrópontot.



Metszőszabályok fejlesztése:

Az ugrópontokat két csoportra oszthatjuk aszerint, hogy van-e járhatatlan szomszédja vagy nincs. Ha egy olyan ugrópontot, aminek van akadály a szomszédjában elhagynánk, előfordulhatna, hogy az algoritmusunk nem optimális utat talál vagy egyáltalán nem is talál utat a célpontunkhoz. Azok az ugrópontok viszont amiknek minden szomszédjuk járható, csupán köztes pontja olyan ugrópontoknak amiknek van akadály a szomszédjukban. Elhagyhatjuk ezeket a köztes ugrópontokat, ha a leszármazott ugrópontjait eltároljuk és azt adjuk hozzá az eredeti ponthoz mint leszármazott ugrópontok. Ekkor az algoritmusunk optimális marad és tovább gyorsul.

Forrás:

A fejezet nagy része a [4] cikk alapján íródott.

Irodalomjegyzék

[1] Daniel Harabor, Adi Botea, *Breaking Path Symmetries on 4-connected Grid Maps*, 2010

[2] Daniel Harabor, Adi Botea, Philip Kilby, *Path Symmetries in Undirected Uniform-Cost Grids*, 2011

[3] Daniel Harabor, Alban Grastien, *Online Graph Prunning for Pathfinding on Grid Maps*, 2011

[4] Daniel Harabor, Alban Grastien, *Improving Jump Point Search*, 2014