UNIVERZA V LJUBLJANI

FAKULTETA
ZA RAČUNALNIŠTVO
IN INFORMATIKO

LOVRENC GREGORČIČ

ALGORITMI ZA ISKANJE POTI

DIPLOMSKO DELO NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

MENTOR: PROF. DR. IGOR KONONENKO

Ljubljana, 2012

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00233/2012
Datum: 04.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **LOVRENC GREGORČIČ**

Naslov: **ALGORITMI ZA ISKANJE POTI**
PATH SEARCH ALGORITHMS

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Naloga zajema pregled in analizo algoritmov za iskanje poti. Kandidat naj pregleda algoritme za iskanje poti in povzame najpomemnejše pristope k iskanju poti. Opiše naj osnovne probleme iskanja poti in pristope k njihovemu reševanju. V praktičnem delu naj vsak pristop uporabi na umetno generiranih podatkih oziroma na izbranih realnih domenah.

Mentor:
prof. dr. Igor Kononenko

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Kazalo

Povzetek

Ključne besede: algoritmi, iskanje poti, A*

Algoritmi za iskanje poti sodijo v tematiko umetne inteligence. Prvi algoritmi so se pojavili že v petdesetih letih dvajsetega stoletja, v tem delu je predstavljenih nekaj izbranih iz začetkov razvoja iskanja poti ter njihovih sodobnih naslednikov. Zelo splošno sta predstavljena delovanje samih algoritmov in njihove lastnosti. Iz opisov se je možno naučiti logike in delovanja algoritmov ter pridobiti dovolj podatkov, na podlagi katerih se lahko potem odločamo o primernemu algoritmu za dani problem.

Vsak algoritem pozna precej izboljšav, ki pa v delu niso opisana. Za rešitve se lahko obrnemo na uporabljene vire. Prav tako ni podrobno opisan del predstavitve iskalnega prostora. Uporabljali smo samo mreže, poznamo pa še veliko drugih možnosti (waypoints, mesh ...) [5], ki imajo določene prednosti pred našo mrežo. Poleg tega so opisani algoritmi neprimerni za implementacijo v aplikacijah, kjer želimo na omejenih sredstvih le imitirati pametno odločanje in iskanje poti [3]. Nam pa lahko podajo dober temelj, na katerem gradimo naprej.

Abstract

Keywords: algorithms, path finding, A*

Path finding algorithms are part of artificial intelligence. First algorithms were presented in the 1950s. This paper will present some of the algorithms from those days as well as their successors. The provided information about their logic and defining properties should suffice the reader to find the appropriate algorithm for his needs.

Each presented algorithm has already many modifications invented, but those modifications are out of scope of this paper. They are, however, described in some of the literature I used. Map presentation is another thing lacking sufficient attention in this paper. All I was using were grids, but there are many other possibilities (waypoints, mesh…) [5], which offer some advantages. Described algorithms are also not the best choice for imitating AI on limited resources [3]. However, they provide you a good basis.

1. UVOD

Pred časom sem razvijal enostavno igro za android, v kateri sem potreboval algoritem za iskanje najkrajše poti. Ker se sam v doglednem času nisem mogel domisliti dovolj dobrega algoritma, sem začel na internetu raziskovati v tej smeri in presenečen ugotovil, da je področje zelo živahno. Tako sem se odločil, da na to temo izdelam diplomsko nalogo. Raziskane algoritme sem implementiral za iskanje poti na mrežah, s katerimi tudi sicer velikokrat v igrah predstavljamo igralno površino.

Algoritmov je veliko in vsi imajo hibe in prednosti. Njihova učinkovitost je odvisna od mnogo faktorjev, med njimi tudi od samega prostora, ki ga preiskujejo. Cilj diplomske naloge je narediti kratek pregled algoritmov in na preprost način razložiti njihovo logiko. Vse opisane algoritme sem implementiral za iskanje na mreži. Najprej sem naredil aplikacijo, v kateri sem lahko v realnem času opazoval delovanje in obnašanje algoritmov (slika 1). Zaradi eliminacije vplivov sem za testiranje razvil ločeno aplikacijo.

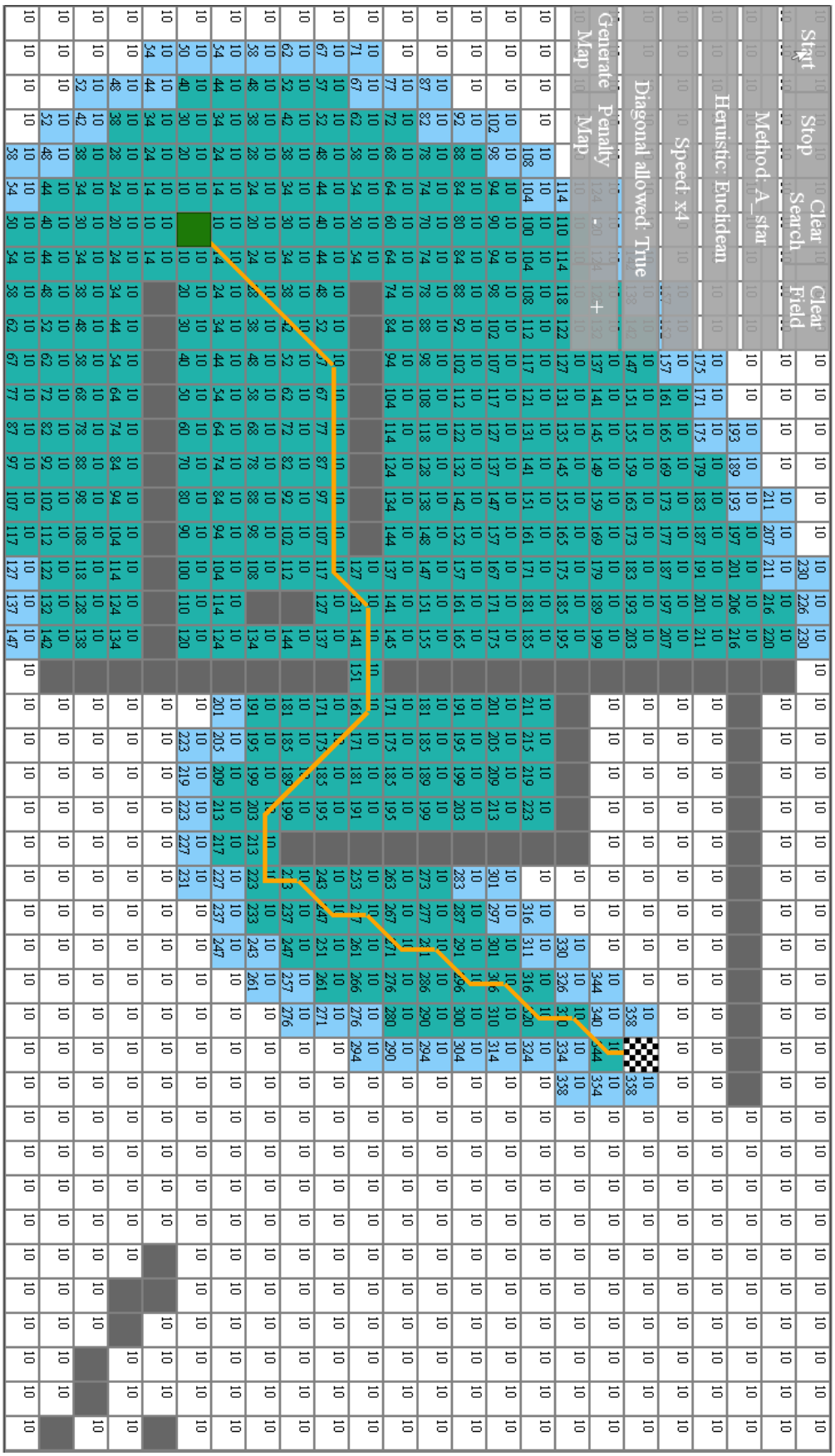
Algoritmi so opisani po kronološkem vrstnem redu. Ta razvrstitev se mi je zdela najbolj smiselna, saj se novejši algoritmi močno naslanjajo na starejše.

V prvem delu so na kratko opisani grafi in enostavne mreže, s katerimi opišemo naše iskalno polje. Sledi opis Dijkstrovega algoritma, ki je temelj mnogim novejšim algoritmom. Nato se logika nekoliko preseka s precej samosvojim Floydovim algoritmom. Nadaljeval sem z zelo prilagodljivim A*. Na tej točki se tudi začne iskanje poti z uporabo hevristike. Sledi primer hierarhičnega iskalnega algoritma, ki pa za dejansko iskanje uporablja A*. Zadnji opisani algoritem Jump-point pa temelji na simetrijah med enakovrednimi potmi. Čisto na koncu so opisane še nekatere osnovne hevristične funkcije za iskanje na mreži.

Na sliki 1 lahko vidite uporabniški vmesnik prve aplikacije. Levo zgoraj je menu, v katerem izberemo tip algoritma, hevristiko (za algoritme, ki jo uporabljajo), hitrost izvajanja ter določimo, ali se lahko premikamo v štirih ali osmih smereh (diagonalno).

V tem primeru smo izbrali algoritem A* z evklidsko hevristično funkcijo in omogočili tudi diagonalne premike. Zelen kvadratek na mreži predstavlja začetno točko, karirasti pa ciljno. Sivi kvadratki predstavljajo neprehodne ovire.

Slika prikazuje končno stanje. Svetlo modri kvadratki so kandidati za nadaljevanje iskanja, ki pa se jih tekom iskanja nismo odločili preiskati, ker smo imeli boljše opcije, zeleno obarvani pa so tisti kvadratki, ki smo jih dejansko tudi pregledali. Rumena črta označuje najdeno pot.



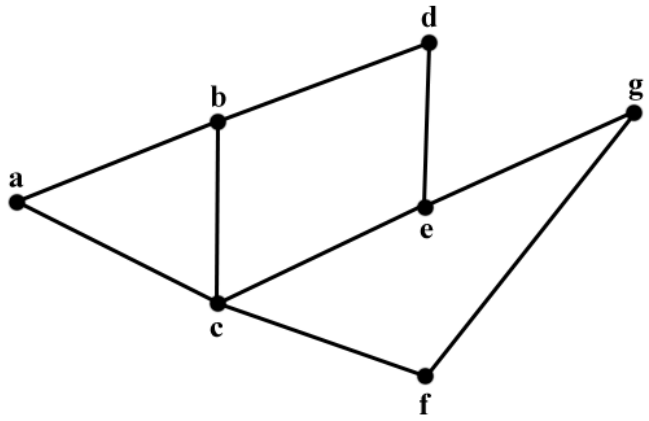
Slika 1: Primer iskanja v aplikaciji.

2. Grafi in mreže

Ker je delovanje nekaterih algoritmov predstavljeno tudi na grafih in mrežah, je prav, da jih na kratko predstavimo. Oglejmo si utežene in neutežene neusmerjene grafe ter 4- in 8-smerne mreže.

2.1. Graf

Graf je sestavljen iz neprazne, končne množice vozlišč V in množice povezav E, ki je podmnožica VxV. Po navadi ga zapišemo z $G = (V, E)$. Elementi množice V so vozlišča grafa G, v množici E pa imamo povezave. Na sliki 2 je primer grafa. [7]



Slika 2: Primer grafa $V = \{a, b, c, d, e, f, g\}$, $E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{c, e\}, \{c, f\}, \{d, e\}, \{e, g\}, \{f, g\}\}$.

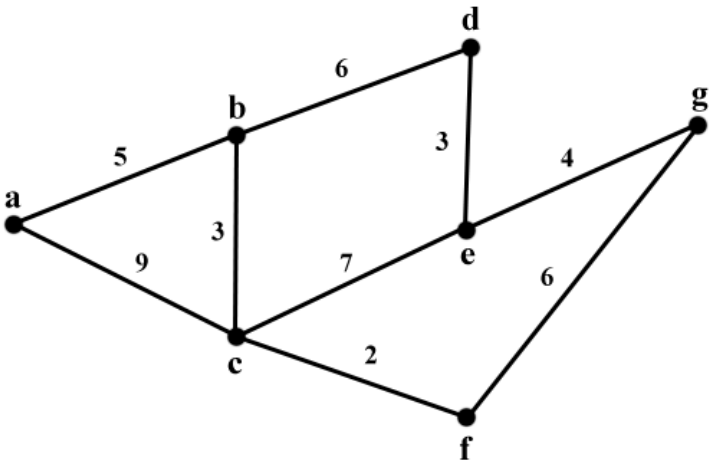
Vzemimo za primer vozlišči a in g. Med njima obstaja več različnih poti, in če dovolimo cikle, celo neskončno različnih. Vse povezave med vozlišči na našem grafu so enakovredne, zato je najkrajša pot tista, pri kateri obiščemo najmanj vozlišč. Torej $a \rightarrow c \rightarrow f \rightarrow g$, ali pa $a \rightarrow c \rightarrow e \rightarrow g$.

2.2. Uteženi graf

Povezave v grafu imajo lahko določene različne uteži (cene) za prehode med vozlišči. Takemu grafu pravimo, da je utežen. Sicer se tak graf v nobenem drugem pogledu ne razlikuje od grafa iz prejšnjega razdelka. Primer je podan na sliki 3.

Z utežmi se iskanje najkrajše poti spremeni. Najkrajša pot je tista, katere seštevek uteži je najmanjši[1]. Če pogledamo primer iskanja poti iz vozlišča a do vozlišča g na uteženem grafu na sliki 3, je najkrajša pot sedaj sledeča: $a \rightarrow b \rightarrow c \rightarrow f \rightarrow g$.

[1] Tudi v neuteženem grafu bi lahko rekli, da je najkrajša pot tista, ki ima najmanjši seštevek uteži. Če privzamemo, da imajo vse povezave enako težo (katerokoli pozitivno število), to hkrati pomeni, da bo pot z najmanjšim seštevkom uteži tista, ki bo vsebovala najmanj povezav.

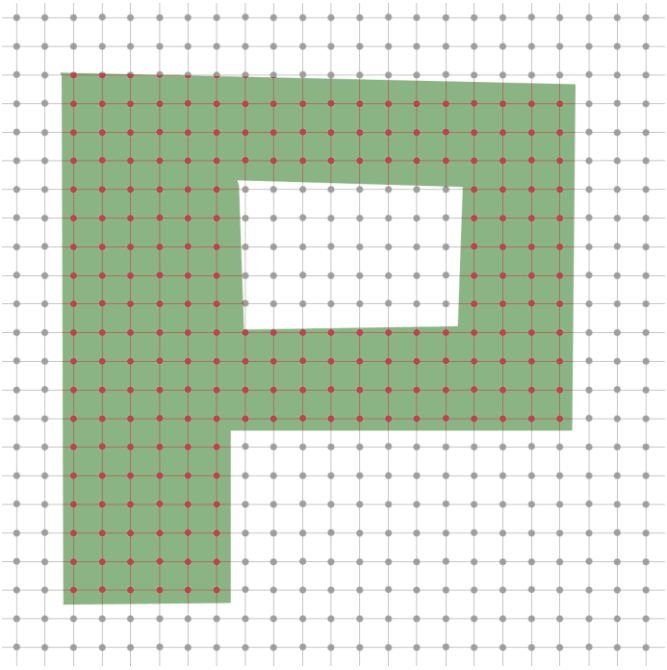


Slika 3: Primer uteženega grafa.

2.3. Mreža

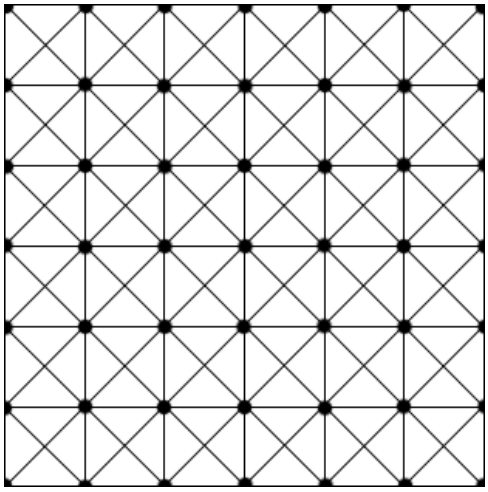
Preden lahko začnemo z iskanjem poti, moramo teren nekako predstaviti v za to primernem načinu. Za naše potrebe bo popolnoma zadostovala enostavna mreža. Z mrežo najenostavneje ponazorimo teren, saj jo je zelo enostavno zgraditi tudi avtomatsko. Mrežo ponazarja množica vozlišč, ki so enakomerno porazdeljena po površini in povezana s svojimi sosedi.

Na sliki 4 vidimo, kako lahko z mrežo prekrijemo prehodno površino, ki je označena z zeleno barvo. Pike predstavljajo vozlišča, črte pa so povezave med njimi. Vozlišča vijolične barve so prehodna, tista sive barve pa ne. Taka predstavitev površine je z vidika porabe sredstev zelo neučinkovita [5]. Veliko pomnilnih sredstev lahko prihranimo že s tem, da neprehodnih vozlišč sploh ne vključimo v naš graf.



Slika 4: Mreža, s katero prekrijemo prehodno površino.

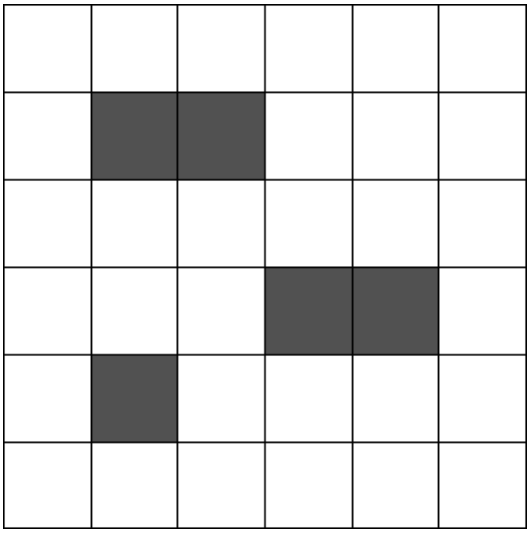
Na sliki 4 imamo samo horizontalne in vertikalne povezave, to pomeni, da se lahko premikamo v štirih smereh. Lahko pa bi vključili tudi diagonale, kot je to prikazano na sliki 5.



Slika 5: Mreža, ki omogoča 8-smerno premikanje.

Vendar pa je taka mreža za demonstracijo delovanja iskalnega algoritma precej nepregledna. Veliko lažje je algoritem predstaviti na mreži na sliki 6.

To je najpreprostejši primer mreže. Vsak kvadratek predstavlja svoje vozlišče. Sivi kvadratki so neprehodni. Samih povezav med vozlišči na tej mreži ne vidimo, je pa logično vsako vozlišče povezano s svojimi sosedami. Vse, kar potrebujemo vedeti, je, kako so vozlišča povezana s sosedami. Torej ali omogočamo 4- ali 8-smerno premikanje.



Slika 6: Preprosta mreža, kjer vsak kvadratek predstavlja vozlišče.

2.3.1. Pomanjkljivosti

Mreža zavzame veliko pomnilnih sredstev. Četudi v naš graf ne vključimo vozlišč, ki ne stojijo znotraj prehodnega območja, potrebujemo razmeroma veliko količino podatkov, da

predstavimo naš teren. Veliko vozlišč hkrati tudi pomeni počasnejše delovanje našega algoritma. Če nam ni mar za hitrost, to ni problem in lahko s podrobno mrežo zelo enostavno in natančno predstavimo naše polje gibanja. Toda po navadi imamo omejena sredstva in čas. Še posebej to velja pri igrah, kjer je le majhen del sredstev namenjen reševanju tega problema, rešitve pa bolj kot točnost predstavitve zahtevajo hitrost. Tedaj se je potrebno ozreti po drugih načinih predstavitve [5].

3. Dijkstrov algoritem

Algoritem je leta 1956 razvil nizozemski znanstvenik Edsger Dijkstra in je prvi algoritem za iskanje poti v grafu, s katerim se srečamo na fakulteti. Algoritem zgradi drevo najkrajših poti iz začetne točke (vozlišča) do vseh ostalih točk (pod pogojem, da so prehodi med vsemi povezanimi pari pozitivni) [4].

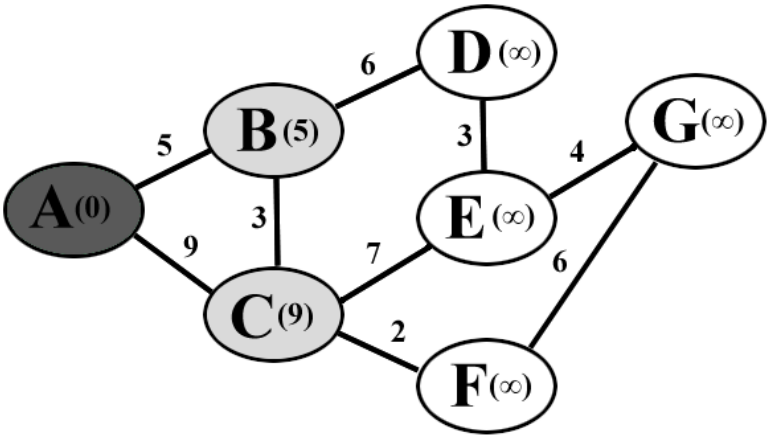
3.1. Lastnosti

- V grafu so dovoljeni cikli,
- algoritem vedno najde najkrajšo pot od začetnega do končnega vozlišča, oziroma do vseh raziskanih vozlišč,
- algoritem deluje po principu požrešne metode,
- v primeru enakovrednih povezav algoritem preiskuje na vse strani z enako hitrostjo (iskanje v širino),
- zahtevnost osnovnega algoritma je $O(n^2)$,
- izboljšana verzija, ki za iskanje naslednjega vozlišča, ki ga bomo pregledali, uporablja kopico, ima časovno zahtevnost $O(m\ log\ n)$.

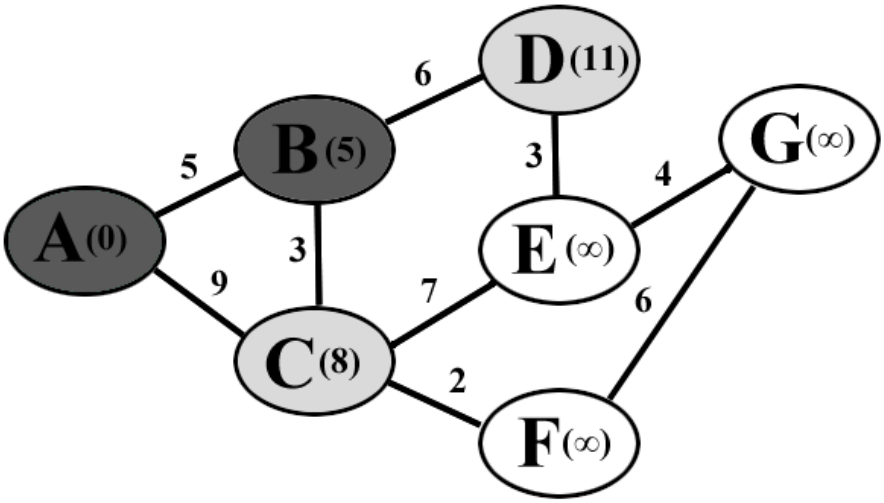
3.2. Delovanje

Algoritem v začetni točki pregleda vsa sosednja vozlišča ter jih označi za možne kandidate za nadaljevanje iskanja poti, začetna točka pa postane oče teh vozlišč. Vozlišče označi za pregledano. V danem primeru na sliki 7 iščemo najkrajšo pot iz vozlišča A do vozlišča G.

Nato se izmed kandidatov izbere tisto vozlišče, ki ima najcenejšo pot (v nadaljevanju vozlišče B). V vozlišču B se prav tako pregleda razdaljo do sosednjih vozlišč. Če je razdalja krajša od trenutne razdalje, postane B oče tega vozlišča (glej sliko 8).

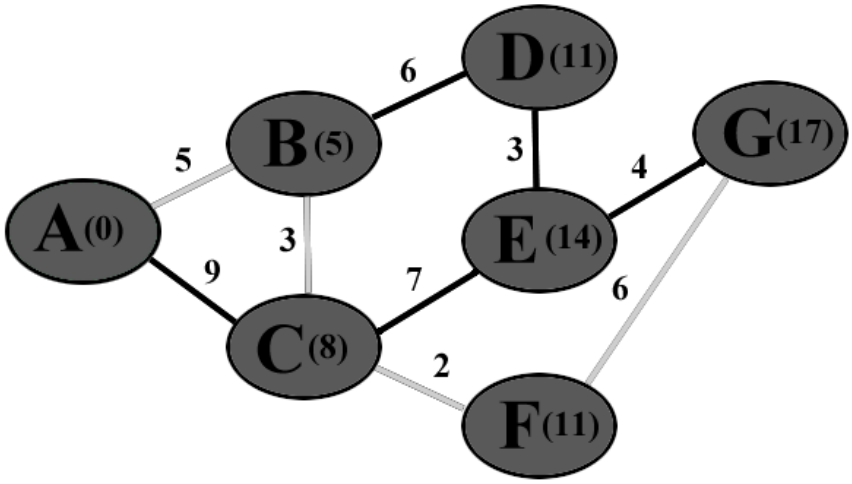


Slika 7: Izgled po prvem koraku Dijkstrovega algoritma. Vozlišče A je pregledano, vozlišči B in C sta možna kandidata za naslednje vozlišče.



Slika 8: V drugem koraku se razširi vozlišče B, saj je imel izmed vseh možnih vozlišč najcenejšo pot. Vozlišče C zamenja očeta, saj je pot preko vozlišča B cenejša kot pot neposredno iz vozlišča A.

Algoritem nato ponavlja korake, vse dokler ne zmanjka kandidatov (v tem primeru iz začetnega vozlišča nismo mogli priti do cilja), ali pa je izbrano vozlišče naš cilj. V tem primeru smo našli najkrajšo pot. Pot dobimo tako, da se iz ciljnega vozlišča preko očetovskih povezav sprehodimo do začetka. Rešitev za naš primer je podana na sliki 9. Shema algoritma Dijkstra je podana s Psevdokodo 1.



Slika 9: Najcenejša pot iz vozlišča A do G je torej sledeča: A, B, C, F, G.

Psevdokoda 1: Algoritem Dijkstra

```
VHODNI PODATKI : start, cilj
IZHODNI PODATKI: dolžina poti, potek poti
OPIS SPREMENLJIVK: Ga – dolžina poti iz začetnega vozlišča do vozlišča A
```

```
function Dijkstra(graf, začetek, cilj)

  FOREACH vozlišče A in graf
    razdalja[A] = INFINITY
    oče[A] = NULL

  sov[] = začetek  //seznam odprtih vozlišč
  WHILE (sov != EMPTY)

    A = vozlišče iz seznama odprtih vozlišč sov, ki ima najmanjšo razdaljo
    odstrani A s seznama odprtih vozlišč sov

    IF (A == cilj)
      return razdalja[A];

    FOREACH (sosed B vozlišča A)
      temp = razdalja[A] + teža_povezave (A, B)
      IF (temp < razdalja[B])
        oče[B] = A
        razdalja[B] = temp

  return null
```

4. Algoritem Floydov-Warshall

Floydov algoritem poišče najkrajše poti med vsemi pari vozlišč v usmerjenem grafu [4]. Dovoljene so negativne povezave[2]. V osnovi algoritem najde le dolžino poti med pari vozlišč. S preprosto modifikacijo lahko iz algoritma rekonstruiramo tudi vse poti. Vse, kar potrebujemo, je dodatna matrika, v katero za vsako izmed vozlišč, skozi katera moramo iti, za optimalno pot zapišemo tisto, ki ima najvišji indeks. Ko rekonstruiramo pot, potem rekurzivno beremo matriko.

4.1. Lastnosti

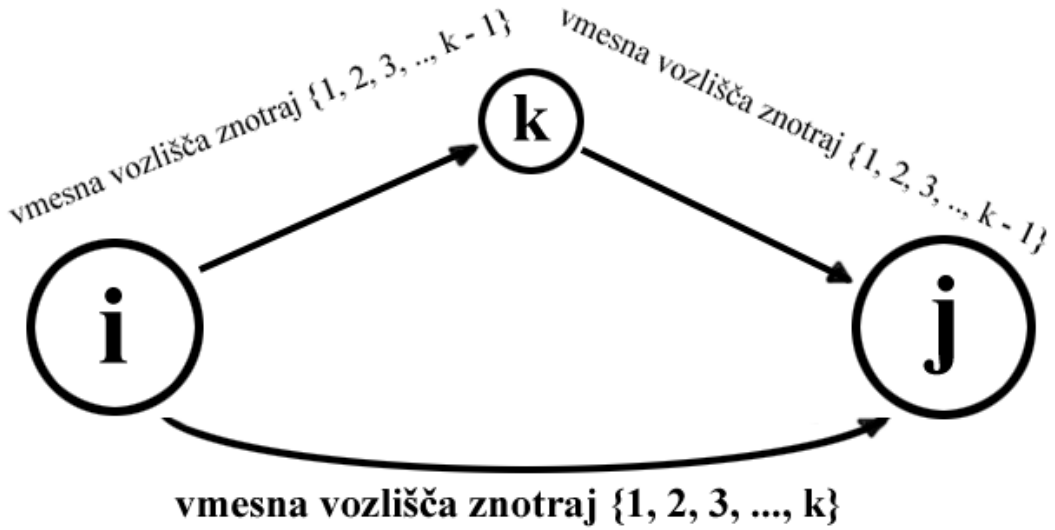
- Algoritem najde najkrajše poti za vse pare vozlišč v grafu,
- zahtevnost algoritma je $O(n^3)$.

4.2. Delovanje

Za vsak par vozlišč $i, j \in V(G)$ si ogledamo vse poti iz i do j, ki vsebujejo zgolj vozlišča iz $\{1, 2, 3, \dots, k\}$.

Če k ni vmesno vozlišče poti p iz i do j, potem gre pot p preko vozlišč $\{1, 2, 3, \dots, k-1\}$. Hkrati je najkrajša pot od i do j preko $\{1, 2, 3, \dots, k-1\}$ tudi najkrajša pot prek vozlišč $\{1, 2, 3, \dots, k\}$.

Če je **k** vmesno vozlišče poti p iz vozlišča **i** do **j**, potem pot **p** razdelimo na poti **p1** in **p2**, kjer je **p1** najkrajša pot iz vozlišča **i** do vozlišča **k**, **p2** pa najkrajša pot iz vozlišča **k** do vozlišča **j**. Obe poti vsebujeta le vozlišča $\{\mathbf{1}, \mathbf{2}, \mathbf{3}, \dots, \mathbf{k}-\mathbf{1}\}$ (glej sliko 10). Shema algoritma Floyd-Warshall je podana s Psevdokodo 2.



Slika 10: K je vmesno vozlišče poti iz i do j. Pot razdelimo na dva dela.

[2] Graf ne sme vsebovati ciklov z negativno vrednostjo. V tem primeru nikoli ne moremo najti najcenejše povezave. Kroženje po takem ciklu z vsakim krogom zniža ceno poti, zato najkrajše poti ne moremo najti.

Psevdokoda 2: Algoritem Floyd-Warshall

```
VHODNI PODATKI: dolz – inicializirana matrika, kjer so razdalje med sosedi
      enake teži povezav med njimi, ostale razdalje so nastavljene na
      neskoncno
IZHODNI PODATKI: floyds nastavi matriki dolzin najkrajsih poti ter naredi
      matriko razdalj (dolz) in matriko poti (pot)

funkcija najdi_pot vrne seznam vozlišč, ki sestavljajo najkrajšo pot
```

```
function floyds(dolz)

  FOR i = 1 do št_vozlišč
    FOR j = 1 do št_vozlišč
      dolz[i][j] = težaPovezave(i, j)

  FOR k = 1 do št_vozlišč
    FOR i = 1 do št_vozlišč
      FOR j = 1 do št_vozlišč
        tempDistance = min(dolz [i][j], dolz [i][k] + dolz [k][j])
        IF(tempDistance < dolz[i][j]) {
          dolz[i][j] = tempDistance
          poti[i][j] = k
        }

  return pot



function najdi_pot(start, cilj, poti, seznam_očetov)
  vmesno_vozl = pot[start, cilj]
  IF(dolz[start, cilj] != INFINITY)
    IF(start != seznam_očetov[cilj])
      return najdi_pot(start, vmesno_vozl) + vmesno_vozl +
        najdi_pot(vmesno_vozl, cilj)
  return prazen_seznam
```

5. A*

A* je zelo razširjen algoritem za iskanje poti, saj je zelo fleksibilen in učinkovito najde najkrajšo možno pot. Je razširitev Dijkstrovega algoritma in uporablja hevristiko. Hevristika je predstavljena v poglavju 9.

A* za izbiro najprimernejšega vozlišča za nadaljevanje iskanja poleg oddaljenosti od začetka upošteva tudi predvideno dolžino poti iz tega vozlišča do cilja (hevristika). Tako je iskanje bolj usmerjeno in preiščemo manj vozlišč. Pri iskanju algoritem za vsako raziskano vozlišče shranjuje tri vrednosti. Označimo jih z $g, h \; in \; f$. g je dolžina poti iz začetnega vozlišča, h je predvidena dolžina poti do končnega vozlišča, f pa je vsota teh dveh vrednosti [9].

Najpočasnejši korak algoritma A* je gotovo iskanje najprimernejšega vozlišča za nadaljevanje iskanja, še posebej pri dolgih poteh. Tedaj imamo mnogo odprtih kandidatov, kar zelo upočasnjuje iskanje. To lahko odpravimo z uporabo binarne kopice.

5.1. Lastnosti

- Algoritem s pravilno izbrano hevristično funkcijo vedno najde najkrajšo pot[3],
- ob dobri izbiri hevristike je algoritem zelo hiter,
- je zelo prilagodljiv in je zanj že napisanih mnogo modifikacij za različne probleme,
- časovna zahtevnost sledi tej formuli: $|h(x) - h^*(x)| = O(logh^*(x))$.

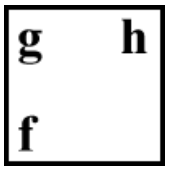
5.2. Delovanje

Algoritem je sam po sebi izredno preprost, hitrost in rezultati pa so odvisni od uporabljene hevristične funkcije. Če je funkcija neprimerna, bodo rezultati slabi. Lahko se namreč zgodi, da ne dobimo najkrajše poti, ali pa da je algoritem zelo počasen. Shema algoritma je podana s psevdokodo 3.

5.3. Predstavitev na mreži

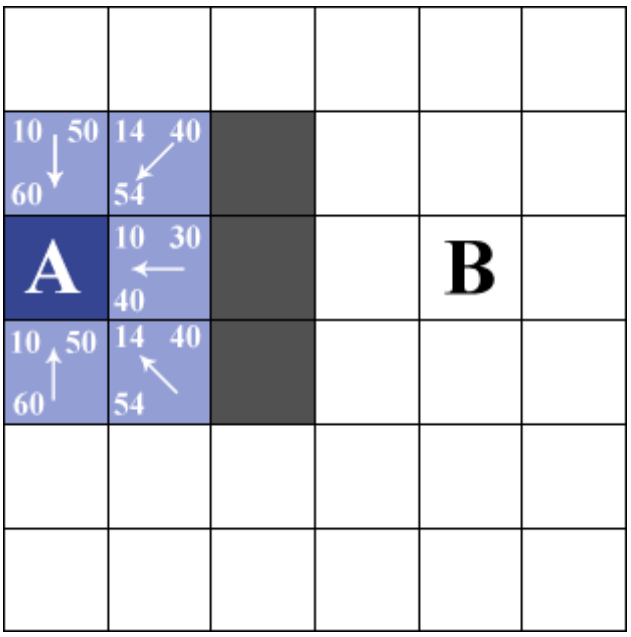
Verjetno si je težko predstavljati delovanje algoritma iz zgornje kode. Ker je A* precej pomemben algoritem in je prisoten tako v HPA* kot Jump-Point, si oglejmo njegovo delovanje še na mreži.

Iskali bomo pot iz vozlišča A do vozlišča B. Naša mreža omogoča 8-smerno premikanje. Cena vodoravnega in navpičnega premika je 10, cena diagonalnih premikov pa 14. Za hevristično funkcijo bomo zaradi preprostosti primera uporabili Manhattan[4] (glej slike 11-17). Vsak kvadratek na mreži prikazuje tudi vrednosti treh spremenljivk, ki so pomembne za naše iskanje. Njihova postavitev je prikazana na sliki 11.



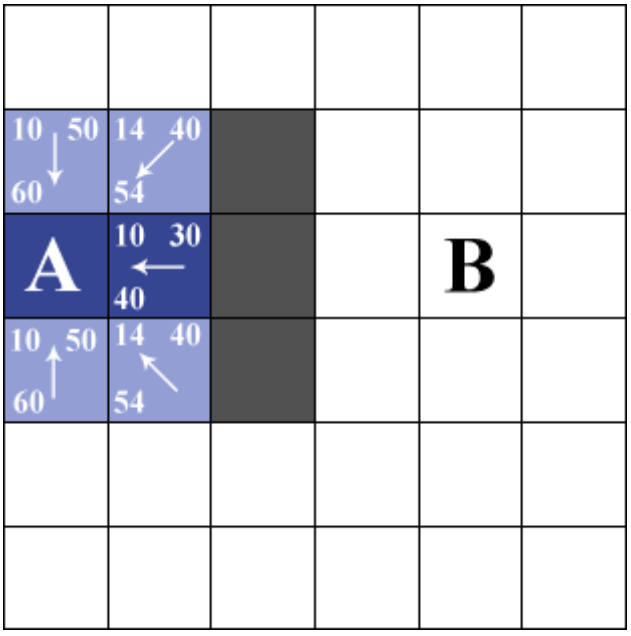
Slika 11: Pri demonstraciji algoritma so v vsakem polju zapisane vrednosti po tem razporedu.

[3] Več o vplivu izbire hevristične funkcije na algoritem si lahko preberete v poglavju 9.
[4] Glede na to da, dovolimo tudi diagonalne premike, Manhattan sicer ni optimalna izbira. Funkcija bo v nekaterih primerih vračala prevelike vrednosti, kar lahko povzroči, da A* vrne neoptimalno pot.



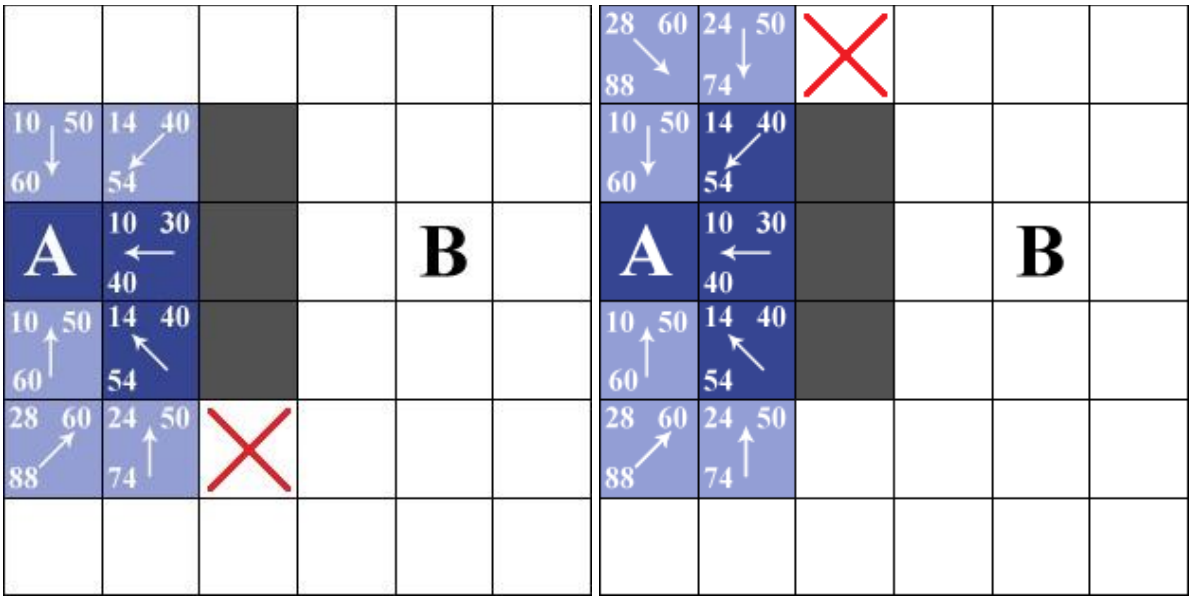
Slika 12: Stanje algoritma po prvi iteraciji.

V prvi iteraciji dobimo pet odprtih vozlišč (označena so s svetlo modro barvo). Puščice kažejo na očeta. Sedaj pride na vrsto izbira najugodnejšega izmed odprtih vozlišč. Že na prvi pogled vidimo, da bo to vozlišče, ki je desno od A. Njegova vrednost f je 40 in je manjša od vseh ostalih (slika 12).



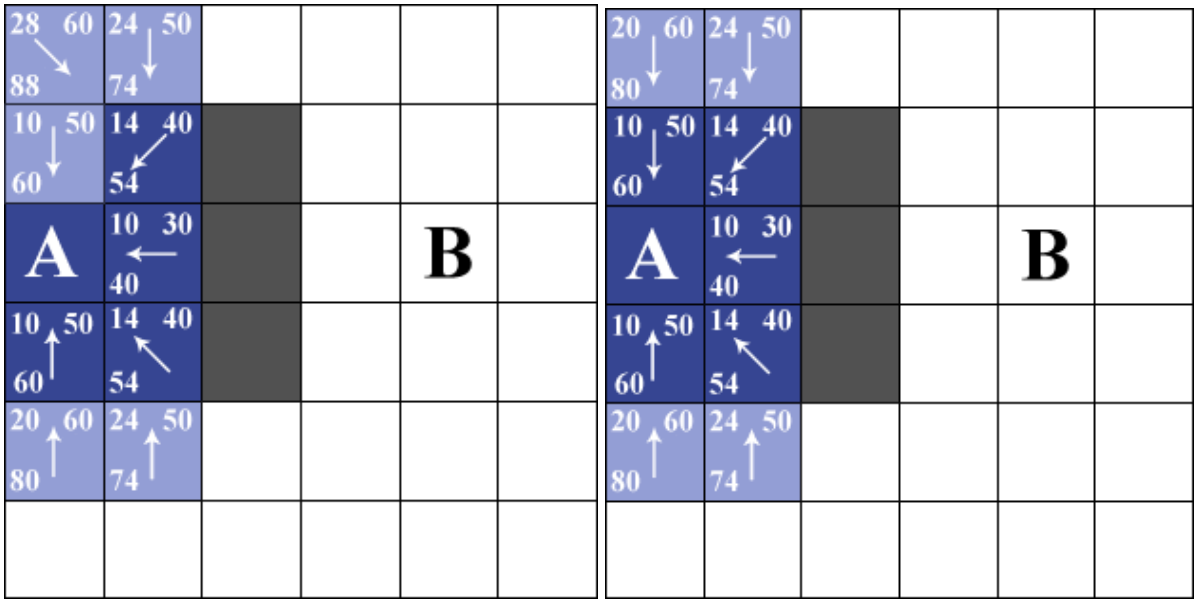
Slika 13: Druga iteracija nas pripelje do prepreke, ki je označena s sivimi polji.

Izbrano vozlišče se je izkazalo za slabo, saj smo se zaleteli v oviro (slika 13). Izberemo drugo najboljše polje in nadaljujemo pot preko njega. Sedaj imamo dve vozlišči z enako oceno. Nadaljevali bomo s spodnjim desnim vozliščem (slika 14).



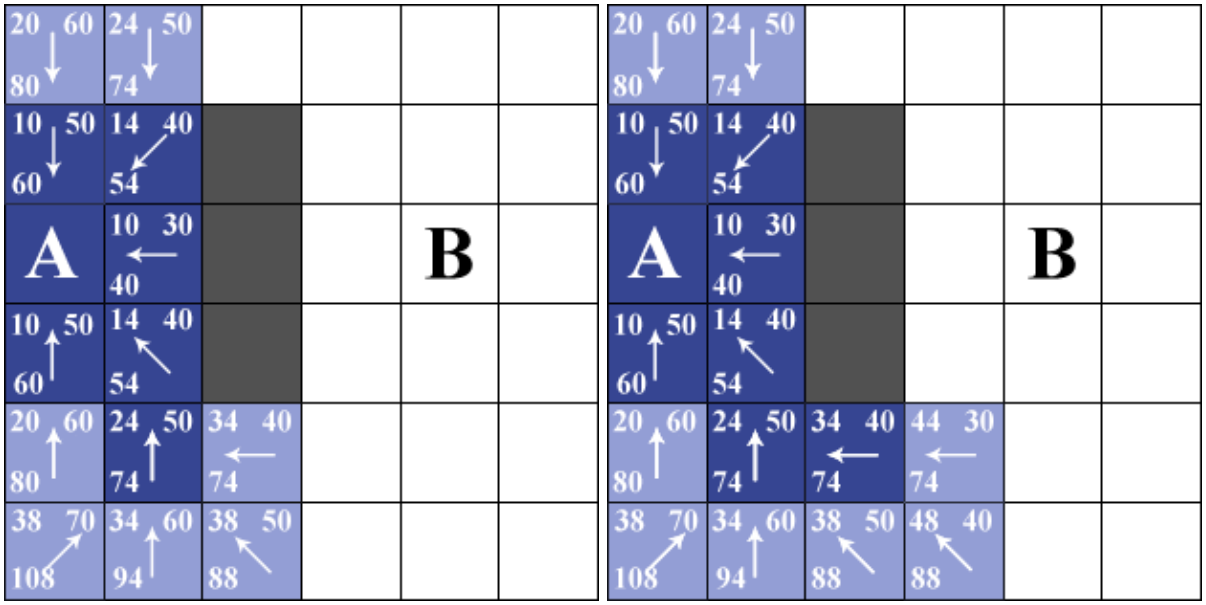
Slika 14: Stanje algoritma A* po tretji in četrti iteraciji.

Na listo odprtih vozlišč nismo dodali vozlišč takoj pod in nad oviro (slika 14). To je stvar izbire. V našem primeru smo ocenili, da bi tak diagonalni premik sekal oviro in je iz vozlišč, ki smo jih pregledovali, nemogoč.



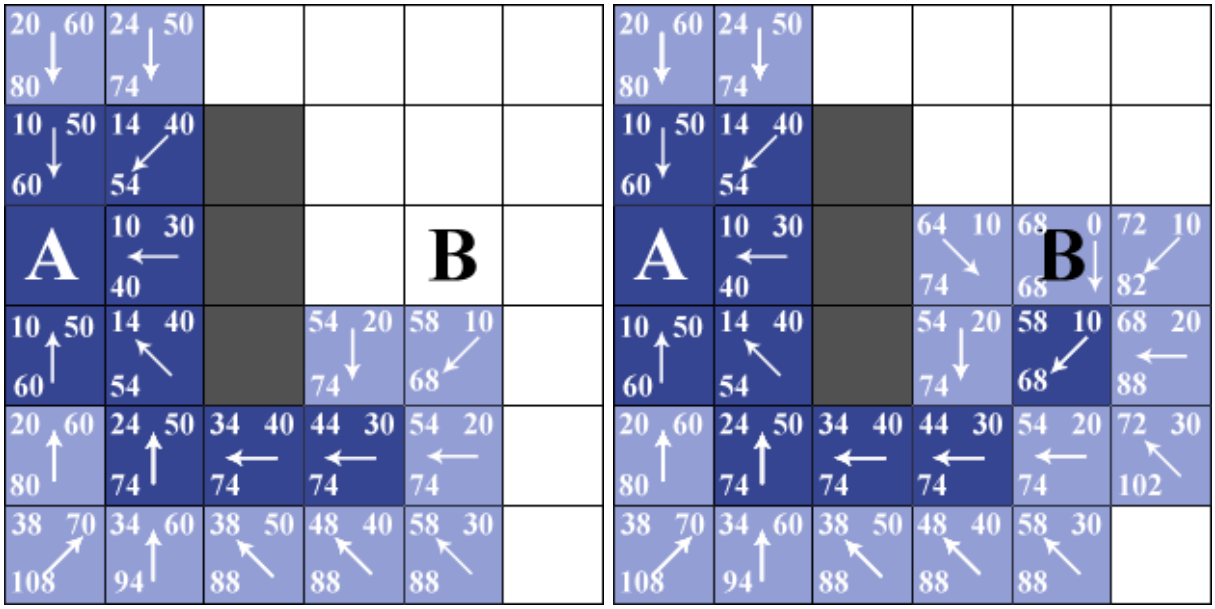
Slika 15: Stanje algoritma A* po peti in šesti iteraciji.

Na sliki 15 prvič vidimo primer, ko vozlišče zamenja očeta. To se zgodi v zgornjem levem in spodnjem levem vozlišču.



Slika 16: Stanje algoritma A* po sedmi in osmi iteraciji.

Nadaljujemo iskanje v smeri cilja. Kot vidite na slikah 16 in 17, smo imeli že več iteracij zapored več vozlišč z enako dobro oceno. Ker se držimo pravila, da v tem primeru vedno izberemo tisto vozlišče, ki je bilo zadnje dodano v množico odprtih vozlišč, hitro napredujemo proti cilju in raziščemo manj vozlišč.



Slika 17: Stanje algoritma A* po osmi in deveti iteraciji.

Iskanje napreduje in v množico odprtih vozlišč pride tudi naš cilj. Ker ima najnižjo vrednost f, bo v naslednji iteraciji vozlišče tudi izbrano, algoritem bo ugotovil, da je to naš cilj in iskanje se bo končalo. Iz ciljnega vozlišča le še rekonstruiramo pot preko očetovskih povezav.

Psevdokoda 3: Algoritem A*

VHODNI PODATKI: graf – graf, po katerem iščemo
start – začetno vozlišče
cilj - ciljno vozlišče
IZHODNI PODATKI: pot - seznam vozlišč, skozi katera poteka pot
OPIS SPREMENLJIVK:
trenutno_vozl - vozlišče, katerega v danem času raziskujemo
sosed - en izmed sosedov trenutnega vozlišča
korak - en korak najdene poti

```
function trenutno_vozl(graf, start, cilj)
   suv[] = start

   g[start] = 0    //seznam dolžin poti
   f[start] = 0    //seznam ocen poti
   sov[] = start   //seznam odprtih vozlišč
   oče[start] = null    //seznam očetov posameznih vozlišč
   pot[]                //seznam vozlišč na končni poti

   WHILE sov !== EMPTY
      trenutno_vozl = vozlišče iz sov, ki ima najmanjši f

      IF(trenutno_vozl == cilj)
         korak = cilj
         WHILE(oče[korak] != null)
            vstavi korak na začetek seznama pot
            korak = oče[korak]
         return pot


      odstrani trenutno_vozl iz sov
      FOREACH sosed sosed vozlišča trenutno_vozl
         nova_dolzina = g[trenutno_vozl] + teža_povezave(trenutno_vozl, sosed)

         IF(sosed ni v sov OR nova_dolzina < g[sosed])
            IF(sosed ni v sov)
               dodaj sosed v sov
            sov[] = sosed
            oče[sosed] = trenutno_vozl
            g[sosed] = nova_dolzina
            f[sosed] = g[sosed] + hevristična_ocena(sosed, cilj)

   return null
```

6. HPA*

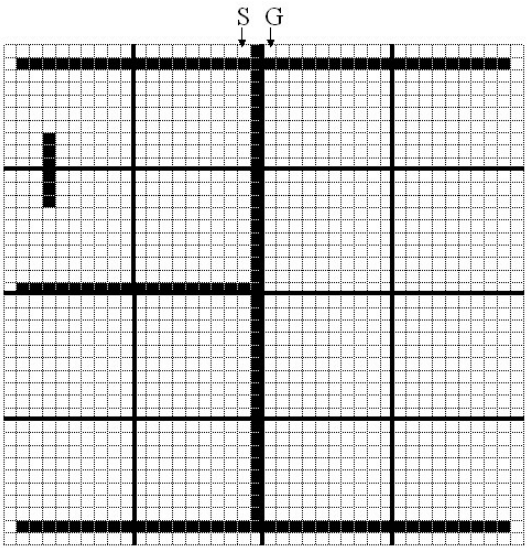
HPA* je algoritem za iskanje poti po mreži in je edini v diplomski nalogi obravnavani algoritem, ki uporablja predpripravljene podatke. Algoritem je hitrejši kot navaden A*, še posebej, če imamo zelo dolge poti, vendar pa ne vrne nujno najkrajše poti. Poti so z optimizacijo in glajenjem znotraj 1 % odstopanja od optimalne poti [1].

6.1. Lastnosti

- Iskanje poteka na abstraktnih podatkih,
- izdelava abstraktnega grafa je neodvisna od topologije, brez dodatnih informacij se ga enostavno izdela iz osnovne mreže,
- po ocenah avtorjev je algoritem do 10x hitrejši od A*, za ceno 1 % odstopanja od optimalne poti,
- HPA* pokaže največji preskok v hitrosti glede na A* šele, ko imamo razmeroma dolge poti,
- če imamo dovolj pomnilnih sredstev, lahko v njih shranimo vse poti med točkami v segmentih. V tem primeru iščemo le po abstraktnemu grafu, samo pot med temi točkami pa zgolj preberemo iz podatkov,
- omejen je na iskanje na mrežah.

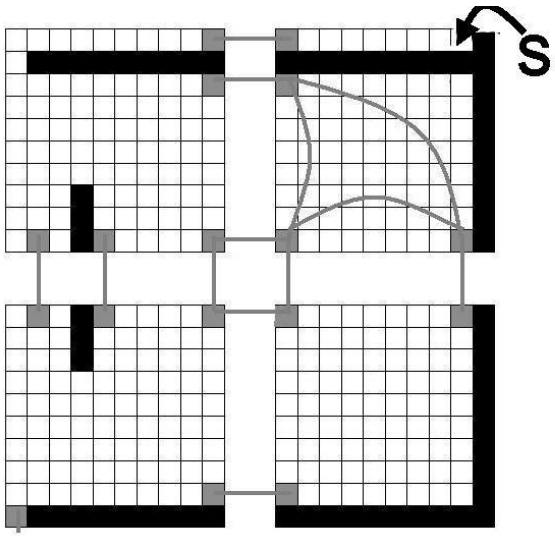
6.2. Priprava podatkov

Mrežo razdelimo na odseke. Vsak odsek vsebuje določeno število vozlišč (v implementirani verziji je vsak odsek velik 10 *10 polj). Potem pa poiščemo prehode med temi odseki. Iz prehodov sestavimo graf, po katerem potem poteka grobo iskanje (glej sliki 18 in 19).



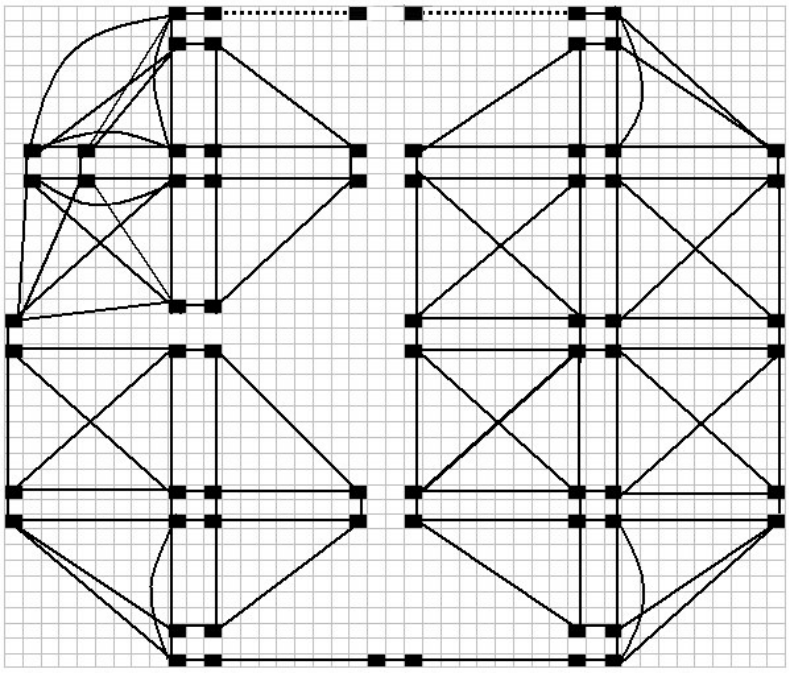
Slika 18: Odebeljene črte predstavljajo naše odseke, črni kvadratki pa predstavljajo ovire. S je začetna točka, G pa je cilj [1].

Graf je sestavljen iz točk, v katerih prehajamo iz enega odseka v drugega. Vse točke znotraj odseka moramo povezati med sabo ter s sosednjo točko iz drugega odseka (glej sliko 22, shema izdelave grafa je prikazana s psevdokodo 4). Dolžine teh medsebojnih poti potem predstavljajo uteži za grobo iskanje in jih moramo shraniti. Prav tako pa lahko ohranimo tudi same poti med točkami znotraj posameznega segmenta. Če imamo dovolj pomnilnih sredstev, lahko na ta način še dodatno pohitrimo algoritem, saj te poti uporabimo v naslednjem koraku.



Slika 19: Sivi kvadratki na sliki predstavljajo vozlišča abstraktnega grafa. Vidite lahko povezave med posameznimi odseki. V zgornjem desnem odseku pa so prikazane tudi povezave med vozlišči znotraj odseka [1].

Graf je sestavljen iz točk, v katerih prehajamo iz enega odseka v drugega. Vse točke znotraj odseka moramo povezati med sabo ter s sosednjo točko iz drugega odseka (glej sliko 20, shema izdelave grafa je prikazana s psevdokodo 4). Dolžine teh medsebojnih poti potem predstavljajo uteži za grobo iskanje in jih moramo shraniti. Prav tako pa lahko ohranimo tudi same poti med točkami znotraj posameznega segmenta. Če imamo dovolj pomnilnih sredstev, lahko na ta način še dodatno pohitrimo algoritem, saj te poti uporabimo v naslednjem koraku.



Slika 20: Graf, ki nastane po procesiranju mreže na sliki 18. S črtkanima črtama sta na tem grafu prikazana še začetna in končna točka [1].

Preden lahko sploh izvedemo grobo iskanje, moramo v graf vstaviti in pravilno povezati začetno in končno vozlišče. Ta korak zahteva dobršen del časa, ki ga algoritem porabi za grobo iskanje. Na podlagi časovne zahtevnosti vstavljanja začetka in cilja v graf je tudi izbrana velikost odsekov. Glede na raziskave so se odseki velikosti 10 x 10 izkazali za najbolj učinkovite. Pri večjih odsekih je potrebno začetek in cilj povezati s preveč vozlišči, da bi odtehtalo sicer hitrejše iskanje, pri manjših odsekih pa število vozlišč, na katerih poganjamo grobo iskanje, ni zadosti majhno.

V drugem koraku grobo iskanje preslikamo na dejansko mrežo. Za to moramo poiskati pot med vsakima sosednjima točkama grobega iskanja (ali pa jo zgolj prebrati, če smo si poti shranili pri predprocesiranju). Tako potem dobimo pot na izvornem grafu.

Taka pot še vedno ni najboljše, kar dobimo iz algoritma. Uporabimo lahko še zelo enostavno tehniko in pogledamo, ali lahko iz enega vozlišča na naši poti do naslednjega pridemo v ravni črti.

Psevdokoda 4: Algoritem HPA*

```
IZHODNI PODATKI: - abstrahirani graf
VHODNI PODATKI: maxLevel – maksimalna stopnja abstrakcije
```

```
function narediGraf(maxLevel) {
  pripraviSklope()
  zgradiGraf()
  FOR i = 2 to maxLevel
    dodajLevelAbstrakcije(i);

}

function pripraviSklope ()
  prehodi[]   //prehodi med posameznimi sklopi
  sklopi[1] = narediSklope(1)  //sklopi prvega nivoja
  FOREACH sklop1, sklop2 v sklopi[1]
    IF sta_soseda(sklop1, sklop2)
      E[] = ustvariPrehode(sklop1, sklop2)

function zgradiGraf()
  FOREACH prehod
    sklop1 = dobiSklop1(prehod, 1) //dobi sklopa, ki ju povezuje
    sklop2 = dobiSklop2(prehod, 1) //ta prehod
    vozlišče1 = novoVozlišče(prehod, sklop1) //naredi vozlišče na sredi
    vozlišče2 = novoVozlišče(prehod, sklop2) //prehoda
    dodajVozlišče(vozlišče1, 1) //dodaj vozlišče
    dodajVozlišče(vozlišče2, 1)
    dodajPovezavo(vozlišče1, vozlišče2, 1, 1, INTER) //dodaj notranjo
                                                     //povezavo

  FOREACH sklop
    FOREACH par vozlišč (v1, v2) v sklopu sklop
    d = razdaljaMedVozliščemaZnotrajSklopa(v1, v2, sklop)
    IF d < INFINITY
      dodajPovezavo(v1, v2, 1, d, INTRA)
```

```
function dodajLevelAbstrakcije(nivo) //dodajanja višjih nivojev
  sklopi[nivo] = narediSlope(nivo)
  FOREACH sklop1, sklop2 iz sklopi[nivo]
    IF !staSoseda(sklop1, sklop2)
      continue
    FOREACH povezava iz dobiPrehode(sklop1, sklop2)
      nastaviNivo(dobiVozlišče1(povezava), nivo)
      nastaviNivo(dobiVozlišče2(povezava), nivo)
      nastaviNivo(dobiPovezavo(povezava), nivo)
  FOREACH sklop iz sklopi[nivo]
    FOREACH par vozlišč(v1, v2) iz sklop
      d = razdaljaMedVozliščemaZnotrajSklopa(v1, v2, sklop)
      IF d < INFINITY
        dodajPovezavo(v1, v2, nivo, d, INTRA)
```

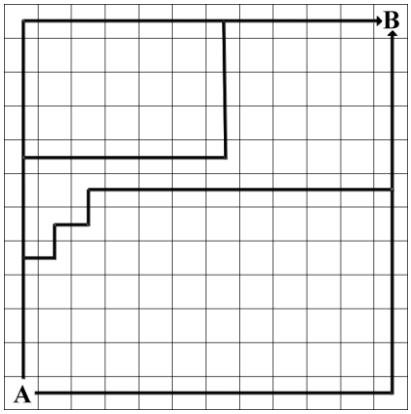
7. Jump-point search

Jump point je algoritem v razvoju in za svoje iskanje uporablja lastnost, ki jo imajo neutežene neusmerjene mreže [2]. Na taki mreži je po navadi veliko različnih poti do cilja, ki imajo enako dolžino. Drugi tu opisani algoritmi bi, razen v idealnih primerih, preverili velike odseke ostalih poti, Jump-point pa uporablja dve enostavni pravili, ki mu omogočata, da pregleda zelo malo polj in kljub temu najde optimalno pot [6].

7.1. Lastnosti

- Algoritem zaenkrat deluje le na 8-smerni kvadratni mreži, kjer imajo vsi prehodi med vozlišči enako težo (diagonalne povezave imajo težo $\sqrt{2} * D$),
- izkorišča simetrije med enakovrednimi potmi na taki mreži,
- je še hitrejši kot HPA* in ne zahteva predprocesiranja in dodatnih pomnilnih sredstev,
- vedno najde najkrajšo pot.

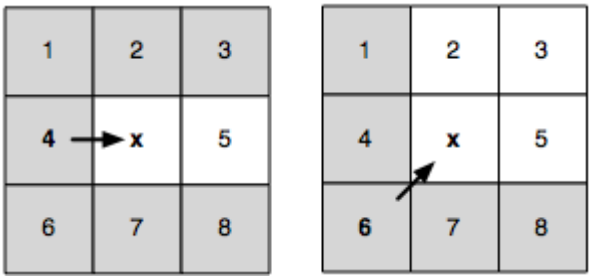
7.2. Delovanje



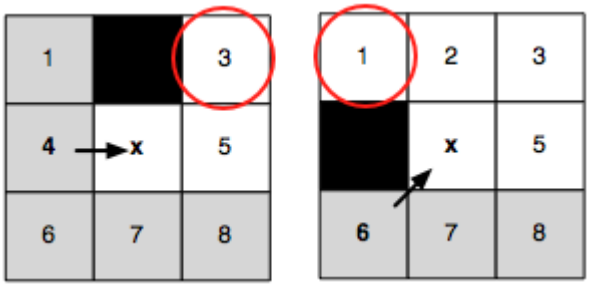
Slika 21: Vidimo 4 različne poti iz polja A do polja B. Vse so enakovredne. Seveda obstaja ogromno drugih enako dobrih poti, ki niso označene. Cena prehoda med polji je enaka za vse prehode.

Na sliki 21 lahko vidimo štiri enakovredne poti. Na mreži, na kateri imajo vsi prehodi v sosednja polja enako težo, so vse enakovredne poti sestavljene iz enakega števila horizontalnih in vertikalnih premikov. Edino, kar jih razlikuje, je vrstni red teh premikov. Jump point to izkorišča in za svoje iskanje uporablja dve enostavni pravili.

Slika 22 prikazuje prvo pravilo tega algoritma. Pregledujemo polje x. Puščice povedo, iz katere smeri prihajamo do polja, in na podlagi smeri lahko tudi izločimo vsa polja, do katerih lahko po enako dolgi ali krajši poti pridemo iz očeta polja x, brez da gremo skozi polje x samo. Pri horizontalnem premiku ob manjku zanimivih vozlišč zgolj nadaljujemo v dani smeri, pri diagonalnem, pa moramo pred nadaljevanjem iskanja v dani smeri pregledati še horizontalno in vertikalno smer.



Slika 22: Preučujemo polje X, puščica pa kaže, iz katere smeri prihajamo. Vsa siva polja preskočimo, saj lahko do njih enakovredno pridemo že iz očeta polja X.

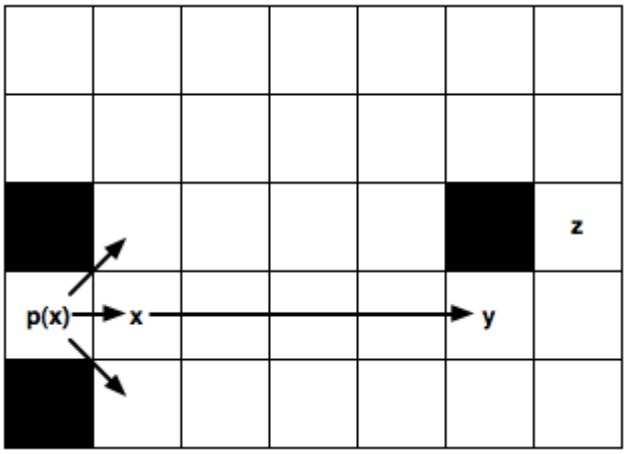


Slika 23: Črna polja so neprehodna. Za razliko od zgornje slike moramo tu v nadaljevanju iskanja dodatno raziskati tudi obe polji obkroženi z rdečo barvo.

Polja črne barve na sliki 23 predstavljajo neprehodne ovire. Če je x poleg ovire, potem obstajajo polja, do katerih iz starša ne obstaja ekvivalentna pot, ki ne bi vključevala polja x. Na sliki sta ti dve polji obkroženi. Vključiti ju moramo v možna polja za nadaljevanje iskanja poti.

Ti dve pravili rekurzivno izvajamo med samim iskanjem. Ideja je, da preskočimo vsa polja, ki jih lahko optimalno dosežemo, brez da gremo skozi polje, ki ga raziskujemo (na slikah polje x). Ustavimo se, ko naletimo na ovire (ne moremo več nadaljevati iskanja v dani smeri), ali pa pridemo do situacije, ki jo vidimo na sliki 24. V tem primeru si polje y zapomnimo, saj je za nas zanimivo, ker ima sosede, ki jih lahko optimalno dosežemo le preko njega.

Da algoritem vrača optimalne poti, je potreben le še pravi vrstni red iskanja. In sicer vedno najprej iščemo horizontalno in vertikalno, preden se za en korak premaknemo po diagonalah. Shema algoritma je predstavljena s psevdokodo 5.



Slika 24: Polje y je točka, do katere lahko preskočimo iskanje, ko horizontalno iščemo iz točke x. Polje y je za nas zanimivo, ker ima soseda z, do katerega najkrajša pot gotovo vodi skozi y.

Z nadaljevanjem razvoja bi avtorji algoritem radi še pohitrili s predpripravo mreže, ga mogoče priredili za grafe z utežmi, za heksagonalne mreže, ga uporabili v HPA* namesto A*, itd.

Psevdokoda 5: Algoritem Jump-Point

Za samo iskanje se uporablja A*, ta koda zgolj poišče zanimiva vozlišča, ki se jih dodaja v sov (seznam odprtih vozlišč).

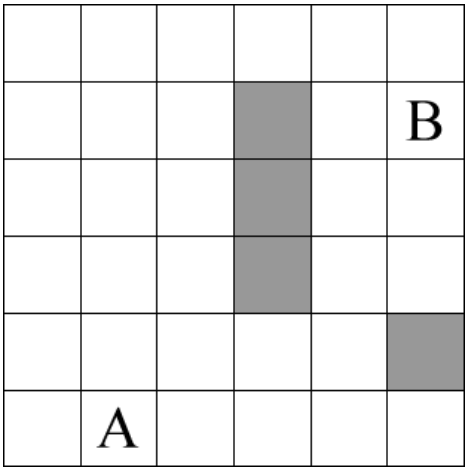
```
function najdiNaslednika(trenutnoVozlišče, start, cilj)
      nasledniki[];
      sosednjaVozlišča[];
      FOREACH vozlišče vozlišče iz sosednjaVozlišča
            n = skoči(n, dobiSmer(x, n), start, cilj)
            nasledniki[] = n
      return nasledniki;

function skoči(trenutrnoVozlišče, smer, start, cilj)
      n = korak(trenutnoVozlišče, smer)
      IF jeNedostopen(n)
            return NULL;
      IF n == cilj
            return n
      IF obstajaZanimivSosed(n)
            return n
      IF jeDiagonalna(smer)
            FOR i = 1 do 2
                  IF skoči(n, horVerSmer(i), start, cilj) != NULL
                        return n
      return skoči(n, smer, start, cilj)
```

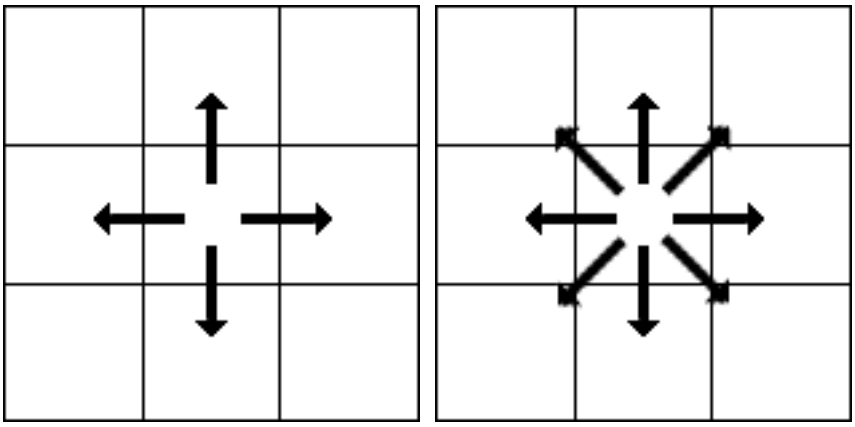
8. Hevristika

V vseh algoritmih se moramo odločati, v katero smer bomo raziskovali. Nekateri algoritmi se odločajo le na podlagi že preverjenih podatkov (Dijkstra), nekateri pa uporabijo tudi oceno dolžine poti iz trenutne točke do cilja. Take dolžine se seveda ne da točno napovedati, lahko pa jo grobo ocenimo. Temu rečemo hevristika [3].

Poznamo več vrst hevrističnih funkcij, mi pa moramo izbrati primerno našim omejitvam premikanja. Funkcije si bomo ogledali na mrežni predstavitvi zemljevida. Iskali bomo pot iz točke A do točke B (glej sliko 25).



Slika 25: Iščemo pot iz polja A do polja B. Siva polja so neprehodna.



Slika 26: Na taki mreži imamo lahko omogočeni dve vrsti premikanja. Na levi vidimo primer premikanja samo horizontalno in vertikalno, na desni pa je omogočeno tudi premikanje po diagonalah, v nadaljevanju 4-smerno in 8-smerno premikanje.

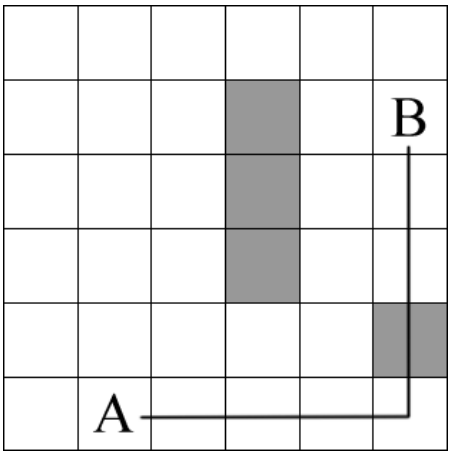
Lahko si tudi sami izmislimo hevristično funkcijo. Oglejmo si nekaj točk, na katere moramo biti pri tem pozorni, in na posledice, ki jih pustijo na delovanju A* [8].

- Če hevristična funkcija vrača $h(k) = 0$, potem šteje le $g(k)$ in se A* spremeni v Dijkstrov algoritem.
- Če je hevristična vrednost v vsakem vozlišču manjša ali enaka dejanski dolžini poti iz tega vozlišča do cilja, potem bo A* gotovo našel najkrajšo pot do cilja. Večja kot je razlika med hevristično oceno in dejansko ceno poti, več vozlišč bo moral A* preveriti.
- Če je $h(k)$ natančno enak dolžini poti iz vozlišča k, potem bo A* sledil le najboljši poti in ne bo nikoli pregledal nobenega vozlišča izven poti. V tem primeru bo A* zelo hiter. Tega se sicer ne da vedno doseči, a v nekaterih posebnih primerih je mogoče.
- Če je vrednost $h(k)$ v nekaterih vozliščih k večja od dejanske dolžine poti, potem A* ne bo vedno vrnil najkrajše možne poti, bo pa hitrejši (saj bo preveril manj vozlišč).
- Če je vrednost $h(k)$ relativno veliko večja od $g(k)$, potem je relavantna le še $h(k)$ in se A* spremeni v best-first search.

8.1. Hevristične funkcije

8.1.1. Manhattan

Manhattan je najbolj osnovna hevristična funkcija. Na sliki 27 vidimo, kako funkcija oceni dolžino poti iz točke A do točke B.



Slika 27: Primer hevristične funkcije Manhattan. Če privzamemo, da je cena premika iz enega vozlišča v naslednjega 10, nam funkcija za oceno premika iz vozlišča A do vozlišča B vrne 80.

Enačba za izračun hevristične vrednosti je sledeča:

$$h(n) = D * (abs(k.x - goal.x) + abs(k.y - goal.y))$$

Funkcija torej sešteje, koliko polj je cilj od trenutnega polja oddaljen horizontalno in vertikalno ter seštevek teh dveh vrednosti zmnoži s ceno premika med polji (D). Ta hevristična funkcija je najhitrejša od vseh funkcij, ki si jih bomo ogledali, a vseeno ni najboljša za vse primere. Če pogledamo četrto točko iz seznama stvari, na katere moramo biti pozorni pri izbiri hevristične funkcije, lahko opazimo, da bo Manhattan gotovo vračal

najkrajšo pot, le če imamo omogočeno samo 4-smerno premikanje. V primeru, da se premikamo v osmih smereh, pa lahko dobimo tudi kako daljšo pot.

D izberemo glede na ceno premika iz enega polja do drugega. Če imamo različne cene, izberemo najmanjšo možno vrednost premika. Tako zagotovimo, da nam algoritem vrne najkrajšo pot. Če povečamo D, bomo pohitrili algoritem, a ne bomo nujno dobili najkrajše poti.

8.1.2. Chebyshev

Chebysheva funkcija je nekoliko počasnejša, nam pa zagotavlja najkrajšo pot, tudi če imamo 8-smerno premikanje. Glede na to, da diagonalni premik nadomesti dve potezi, je formula sledeča.

$$h(n) = D * \max(abs(k.x - cilj.x), abs(k.y - cilj.y))$$

Ta hevristična funkcija bo uspešna le, če je cena vseh premikov enaka. V primeru, da je cena za diagonalni premik enaka npr. $D_{diag} = 1.41 * D$, ta funkcija ne bo zagotovila optimalne poti. Za take primere uporabimo naslednjo funkcijo:

$$hd(k) = \min(abs(k.x - cilj.x), abs(k.y - cilj.y))$$
$$hs(k) = abs(k.x - cilj.x) + abs(k.y - cilj.y)$$
$$h(k) = D2 * hd(k) + D * (hs(k) - 2 * hd(k)))$$

8.1.3. Evklidska

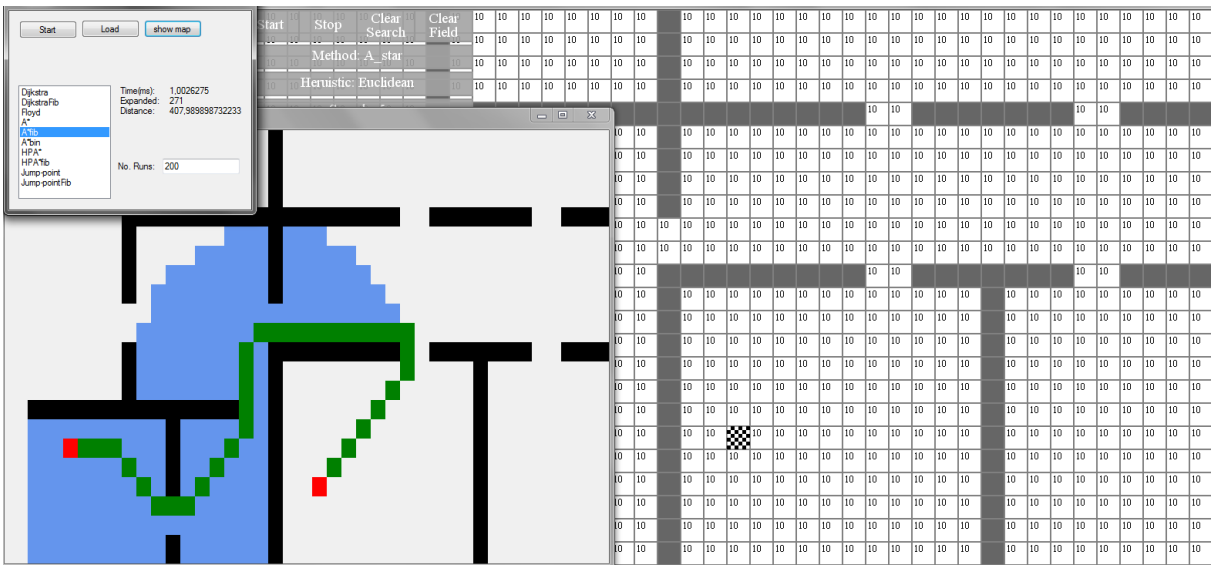
Če se lahko po mreži premikamo v katerikoli smeri, ne samo v navedenih osmih smereh, pa uporabimo evklidsko funkcijo:

$$h(k) = D * sqrt((k.x - cilj.x)^2 + (k.y - cilj.y)^2)$$

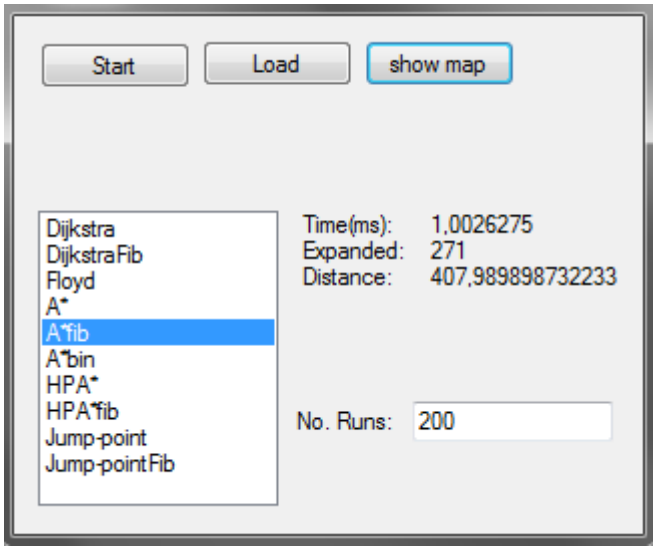
9. Implementacija testnega okolja in ovrednotenje

Za testiranje algoritmov sem razvil ločeno aplikacijo, s katero sem odstranil vpliv izrisovanja in vezanosti na platformo XNA. Tako sem lahko dobil bolj natančne čase in boljšo oceno učinkovitosti algoritmov. Algoritme sem na mrežah glede na zahtevnost poganjal od nekaj stokrat do nekaj tisočkrat, da sem izločil še problem interne ure.

Na sliki 28 sta vidni obe aplikaciji. Na desni strani je prvotna aplikacija, v kateri se lahko v realnem času sledi izvajanju algoritma. Ker tak sistem ni bil primeren za testiranje hitrosti, sem ločeno razvil program, čigar osnovno okno je vidno v zgornjem levem kotu in na sliki 29.



Slika 28: Izgled aplikacij.



Slika 29: Program za testiranje algoritmov.

V njem se lahko naloži mreži iz osnovne aplikacije, shranjeno v tekstovni datoteki ter poganja različne algoritme. Algoritme se izbira v izbirnem oknu na levi, na desni pa se izpišejo dolžina najdene poti, porabljen čas (v milisekundah) in število pregledanih vozlišč. Edini vnosni parameter programa je število izvajanj posameznega algoritma na isti mreži, s katerim izločimo vpliv interne ure računalnika. Program omogoča še poenostavljen izris mreže (vidno v spodnjem levem kotu slike 28).

9.1. Scenarij poizkusov

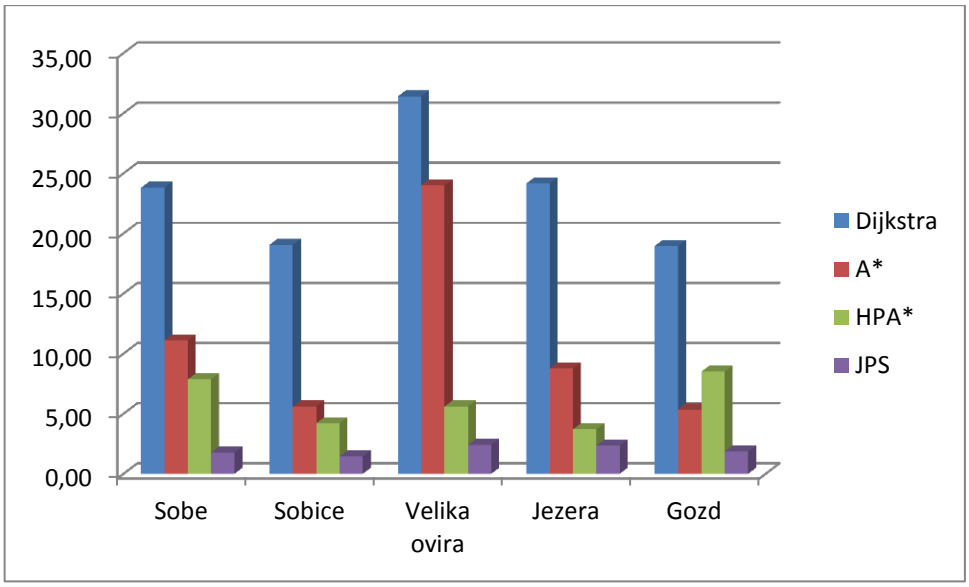
Za testiranje sem izbral le zanimivejše algoritme. Poleg Dijkstre, ki služi kot nekakšna orientacija, sem testiral še A*, HPA* in Jump-point, v enem testu pa sem pognal še Floydov algoritem.

Algoritme sem poganjal na različnih mrežah, s katerimi sem poizkušal zajeti različne probleme pri iskanju poti in tako razkriti šibkosti algoritmov. Znano je namreč, da se bo A* dobro odrezal na odprtih mrežah ali mrežah z majhnimi ovirami, ob velikih ovirah pa bo preiskoval ogromne količine terena med oviro in začetnim vozliščem. Podobne značilnosti sem poizkusil odkriti tudi pri drugih algoritmih.

Za testiranje sem uporabil pet mrež. Ena simulira iskanje v prostoru z veliko količino majhnih ovir (gozd), ena mreža je zelo odprta in ima le tri ovire (npr. iskanje poti okoli jezer), potem je ena mreža, ki simulira delovni prostor v celičnem delovnem okolju (sobice), ena mreža z veliko oviro, ter zadnja, ki predstavlja malce bolj odprt labirint (sobe).

9.2. Rezultati

Algoritme sem testiral po treh kategorijah: času, količini pregledanih vozlišč in dolžini najdene poti. Meritve so prestavljene s spodnjimi grafi.

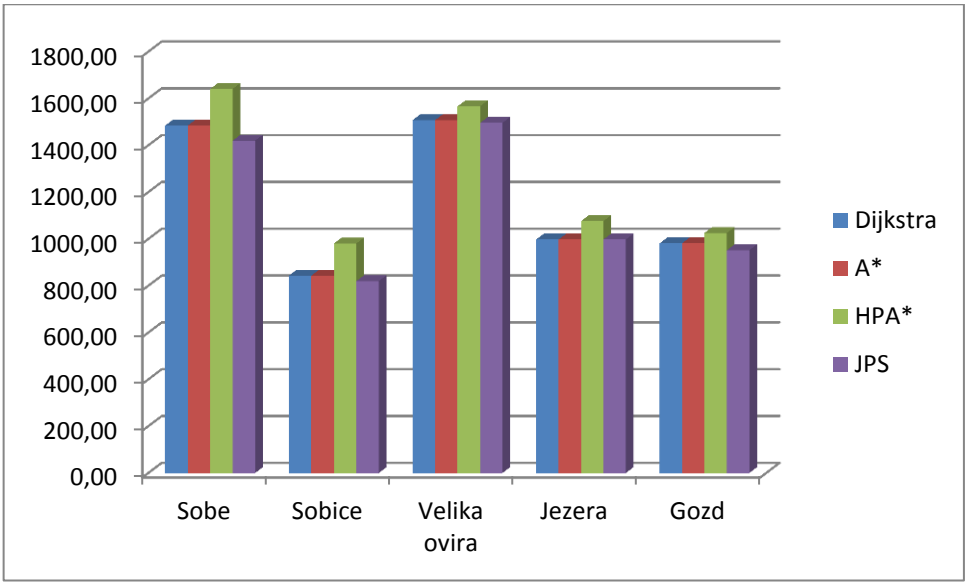


Slika 30: Časovna zahtevnost algoritmov v milisekundah.

Slika 30 prikazuje, koliko časa so porabili posamezni algoritmi za iskanje poti po posameznih mrežah. Edino presenečenje je bila hitrost algoritma HPA*, ki je bil v enem od primerov celo počasnejši kot A*, v ostalih pa tudi ni bil veliko hitrejši. Glede na to, da moja implementacija sploh ni poizkušala gladiti najdene poti, je to še toliko bolj presenetljivo. Ena od razlag bi lahko bila posledica razmeroma kratkih poti, saj vpenjanje začetne in končne točke v graf

vzame precej časa in glede na to, kolikšen kos mreže predstavlja posamezen odsek, je ta čas v mojem primeru relativno velik. Še posebej je to opazno na mreži gozda, kjer mora algoritem ti dve točki povezati z mnogo vozlišči, saj med odseki obstaja veliko prehodov.

Edini preostali rezultat, ki nekoliko izstopa, so merjenja na mreži z veliko oviro. Opazi se občutno manjšo razliko med algoritmoma Dijkstra in A*. A* je od Dijkstre hitrejši zaradi hevristike[5], ki se trudi čim bolj uspešno napovedati, v katero smer naj gremo, da pridemo najhitreje do cilja. V primeru ogromne ovire mora algoritem kljub hevristiki pregledati velike odseke mreže pred samo oviro, preden najde pot okoli nje.



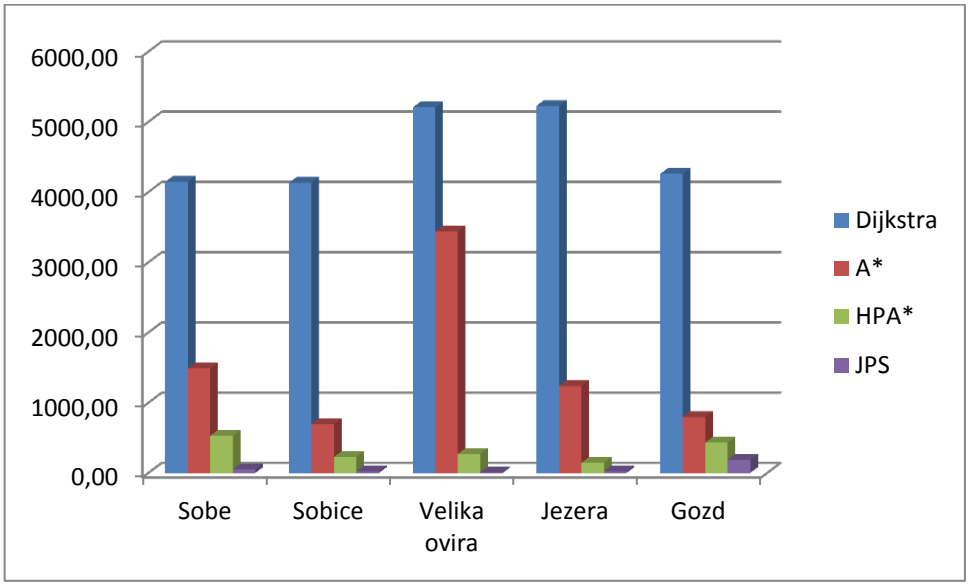
Slika 31: Dolžine najdenih poti.

Na sliki 31 so prikazane dolžine najdenih poti. Tukaj ni nobenih presenečenj. algoritem HPA*, ki sem ga implementiral, ni imel nobenih funkcij za glajenje najdene poti[6], s katerimi naj bi zmanjšal odstopanje razdalje na 1 %, pa se je vseeno v vseh primerih obnesel presenetljivo dobro. Dijkstra, A* in JPS vsi najdejo najkrajšo možno razdaljo. Do odstopanja razdalj poti funkcije JPS pride zgolj zaradi njenega dojemanja ovir, saj jih nekoliko[7] seka in se tega ne da spremeniti, lahko pa bi spremenili Dijkstro in A*, da bi iskala na enak način.

Nazadnje pa sem meril še število preiskanih vozlišč. Meritve so prikazane na sliki 32. Število pregledanih vozlišč se še posebej pri Dijkstri in A* že na prvi pogled zelo lepo pokriva z grafom na sliki 30. Nekoliko drugače je pri HPA* in JPS. Pri HPA* je potrebno začetno in končno točko pred iskanjem dodati v graf in ju po iskanju odstraniti. Kot vidite, to ni trivialna naloga. Če pogledamo na primeru gozd, je HPA* pri iskanju preiskal občutno manj vozlišč, a je zaradi vstavljanja in brisanja točk po času kar krepko zaostal za A*. Daleč najmanj vozlišč

[5] Na testu se je uporabljala Chebysheva hevristična funkcija.
[6] Način glajenja poti, opisan v članku [1], ni primeren za gibanje strogo znoraj mreže.
[7] JPS bi se na sliki 14 prestavil na polje, označeno z rdečim X.

razširi JPA, vendar pa nekaj časa pri svojih skokih porabi za ocenjevanje pomembnosti vozlišč. Vseeno je bil nesporno najhitrejši.



Slika 32: Število pregledanih vozlišč.

Rezultati testiranj se načeloma ujemajo s pričakovanji. Razlogi za to so morda premajhne mreže, ali pa preveč ovir na mrežah, saj se algoritem veliko bolje obnese na mrežah z velikimi odprtimi prostori in manj prehodi.

Floydov algoritem sem pognal le na prvi mreži, saj ga ni mogoče neposredno primerjati z ostalimi algoritmi. Algoritem namreč poišče najkrajše poti med vsemi odprtimi vozlišči. Časovna zahtevnost je n^3 in glede na to, da je algoritem neodvisen le od števila prehodnih vozlišč in postavitev mreže nanj nima vpliva, bi bilo nadaljnje testiranje potrata časa.

Našel je enako dolgo pot kot vsi ostali. Da je poiskal vse poti je porabil 1.2664 ure. Med tem časom je našel 14.453.376 najkrajših poti. Povprečno je torej porabil 0.3154 milisekunde na pot. Za primerjavo, JPS je rabil 0.6725 milisekunde.

10. Zaključek

Med potekom naloge sem implementiral osem algoritmov. Z implementacijo sem sicer imel nekaj težav, saj si algoritmov nisem natančno ogledal vnaprej in sem slabo zastavil osnovo svojega programa. Nekajkrat sem spremenil uporabniški vmesnik, predvsem zaradi nepoznavanja I/O mehanike platforme XNA.

Pred začetkom nisem imel znanja in predstave o algoritmih za iskanje poti. V diplomski nalogi sem se tako osredotočil predvsem na preiskovalni del, toda to je le ena plat samega iskanja poti. Enako pomembna je tudi predstavitev iskalne površine. S premišljeno

predstavitvijo je mogoče občutno zmanjšati čas in pomnilna sredstva, ki so potrebna za izvajanje iskanja. Različne predstavitve nosijo prav tako različno količino podatkov. Če površino zapolnimo s konveksnimi poligoni, imamo na vsakem mestu točne podatke o npr. širini prehoda. Tako lahko na isti mreži poganjamo iskanje za npr. pešca in tovornjak. Tovornjak jasno ne more skozi ozke prehode. Torej dobimo dodatno vrednost za manj sredstev, kot bi jih zavzela mrežna predstavitev. Ta del problema iskanja poti je zelo živahen in zanimiv in bi si zaslužil več pozornosti.

Gotovo je bil najbolj zanimiv algoritem Jump-point. Ideja je zelo samosvoja in se je sam bržda ne bi nikoli domislil. Algoritem ima potencial, in če bo avtorjem uspelo izvesti vsaj polovico svojih želja, ga bomo še videvali. V dani obliki je zelo omejen. Že to, da ne zmore delovati niti na uteženi mreži, je velika pomanjkljivost. Tega se tudi zavedajo in je to ena od stvari, ki bi jo radi rešili. Predvidevanje poti na uteženem grafu se zdi nemogoče. A morda se rešitev skriva prav v predstavitvi podatkov.

11. Viri

[1] A. Botea, M. Müller in J. Schaeffer, "Near Optimal Hierarchical Path-Finding", *Game Developement,* št. 1, zv. 1, str. 7 - 28, 2004.

[2] A. Botea, D. Harabor in P. Kilby, "Path Symmetries in Uniform-cost Grid Maps", v zborniku Symposium on Abstraction Reformulation and Approximation (SARA), Barcelona, 2011.

[3] D. Bourg in G. Seeman, AI for Game Developers, O'reilly, 2004, pogl. 6, 7.3.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest in C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009, pogl. 24.3, 25.2.

[5] M. Gernier, "Pathfinding concepts, the basics", objavljeno: 30.6.2011, http://mgrenier.me/2011/06/pathfinding-concept-the-basics/, obiskano: 15.6.2012.

[6] A. Grastien in D. Harabor, "Online Graph Pruning for Pathfinding on Grid Maps.", v zborniku 25th National Conference on Artificial Intelligence (AAAI), San Francisco, 2011.

[7] R. Merris, Graph Theory, John Whiley & sons, 2001, pogl. 1.

[8] A. Patel, "Amit's Game Programming Information", http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html, obiskano: 15.6.2012.

[9] S. Rabin, AI Game Programming Wisdom, Charles River Media, 2002, pogl. 3.