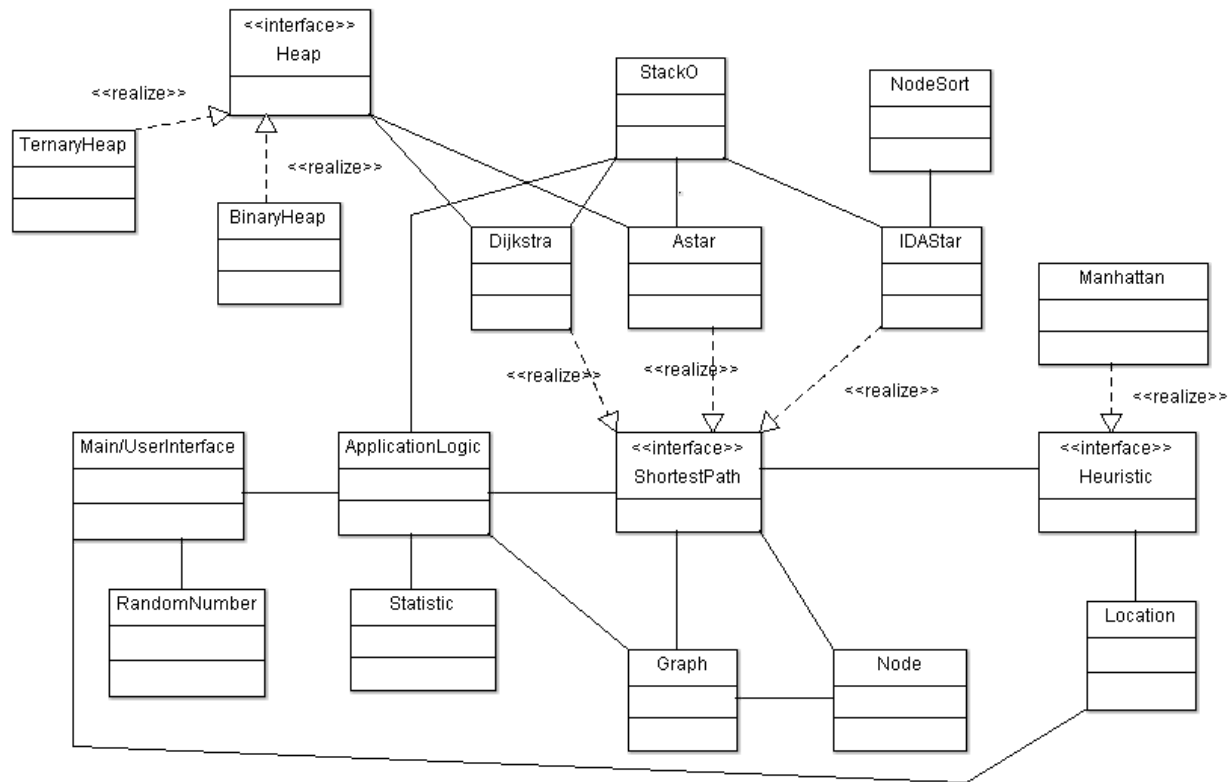


Ohjelman yleisrakenne

Alla ylätason luokkakaavio, joka kuvaa ohjelman rakennetta:



Saavutetut aika- ja tilavaativuudet

Dijkstra

Pseudokoodi:

```
Dijkstra-with-heap(G,start, goal)
  Initialize(G)
  heap-insert(H,start,distance[start])
  while not empty(H)
    u = heap-del-min(H)
    if(u==goal) lopetetaan
    for jokaiselle solmulle v vierus[u]
      Relax(u,v)
      heap-insert(H,v,distance[v])
```

- Alustus vie aikaa $O(|V|)$, kun kaikille koordinaatiston pisteille eli verkon solmuille asetetaan etäisyys (initialize-metodi)
- Algoritmi käyttää minimikekoa. Keko voi olla 2-keko (vanhemmalla max 2 lasta) tai 3-keko (vanhemmalla max 3 lasta). Algoritmi käyttää heap-insert (add-metodi) ja heap-del-min operaatioita (poll-metodi). Keko-operaatiot on nimetty Javan PriorityQueue:n mukaan, jotta tietorakennetta voidaan helposti vaihtaa
 - 2-keko
 - keko-operaatioiden aikavaativuus $O(\log n)$, kun keossa n alkia
 - Rivillä 45 tehdään $|V|$ kappaletta heap-del-min eli aikaa kuluu $O(|V| \log |V|)$
 - Rivillä 63 tehdään $|E|$ kappaletta heap-insert operaatioita
 - Kokonaisuudessa aikaa kuluu (while luopissa riveillä 43-69) $O(|E| + |V| \log |V|)$
 - 3-keko

Heapify-operaation viimeisellä rivillä rekursiokutsu, joten suorituksen aikavaativuus määräytyy suoritettujen rekursiokutsujen määrän mukaan. Keon solmulla on kolme lasta, joten keon korkeudeksi saadaan $O(\log_3 n)$. Rivillä 45 tehdään $|V|$ kappaletta heap-del-min operaatiota, joka kutsuu heapify-operaatiota. Aikaa kuluu siis $O(|V| \log_3 |V|)$

Rivillä 63 tehdään $|E|$ kappaletta heap-insert operaatioita

Kokonaisuudessa aikaa kuluu (while luopissa riveillä 43-69) $O(|E| + |V| \log_3 |V|)$
- Tilavaativuus on $O(|V|)$, koska kaikille taulukoille ja keolle varataan tilaa solmujen lukumäärän verran.

A*

Pseudokoodi:

Astar(G, start, goal)

Initialize(G, s)

heap-insert(H, start, distance[start] + loppuun[v])

while not empty(H)

u = heap-del-min(H)

if(u == goal) lopetetaan

for jokaiselle solmulle v vierus[u]

Relax(u, v, w)

heap-insert(H, v, distance[v] + loppuun[v])

- A* algoritmin "ohjelmarunko" on sama kuin Dijkstralla. Ainoa ero on, että A* käyttää heuristiikkaa ja kutsuu sen getToEnd-metodia etäisyysarvion saamiseksi ja se voidaan päätellä vakioajassa, $O(1)$.

- Myös tilavaativuus on sama kuin Dijkstralla, eli $O(|V|)$, koska kaikille taulukoille ja keolle varataan tilaa solmujen lukumäärän verran.

IDA*

Pseudokoodi:

node: current node
 g: cost to reach current node
 f : estimated cost of cheapest path (root..node..goal)
 h(node): estimated cost of cheapest path (node..goal)
 cost(node, succ): step cost function
 is_goal(node): goal test
 successors(node): node expanding function

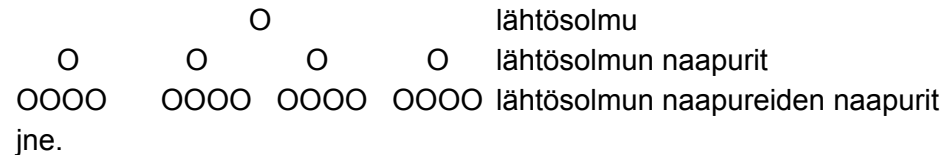
```
procedure ida_star(root)
  bound := h(root)
  loop
    t := search(root, 0, bound)
    if t = FOUND then return bound
    if t =  $\infty$  then return NOT_FOUND
    bound := t
  end loop
end procedure
```

```
function search(node, g, bound)
  f := g + h(node)
  if f > bound then return f
  if is_goal(node) then return FOUND
  min :=  $\infty$ 
  for succ in successors(node) do
    t := search(succ, g + cost(node, succ), bound)
    if t = FOUND then return FOUND
    if t < min then min := t
  end for
  return min
end function
```

- Algoritmi toimii kuten syvyysuuntainen algoritmi. IDA* algoritmi tekee syvyysuuntaisia hakuja lähtösolmusta alkaen uudelleen ja uudelleen kunnes maalisolmu löytyy tai

paluuarvona syvyysuuntaisesta hausta tulee ääretön, mikä kertoo ettei maalisolmua tavoiteta.

- Jos haut tehtäisiin käyttäen puuta, IDA* aikavaativuus on $O(b^d)$ ja tilavaativuus $O(d)$, kun d on syvyys ja b lapsien määrä.
- En toteuttanut puu-tietorakennetta, mutta rekursiokutsuista voi ajatella rakentuvan puu ja manhattan-heuristiikkaa käytettäessä lapsia on max 4 (2-4).



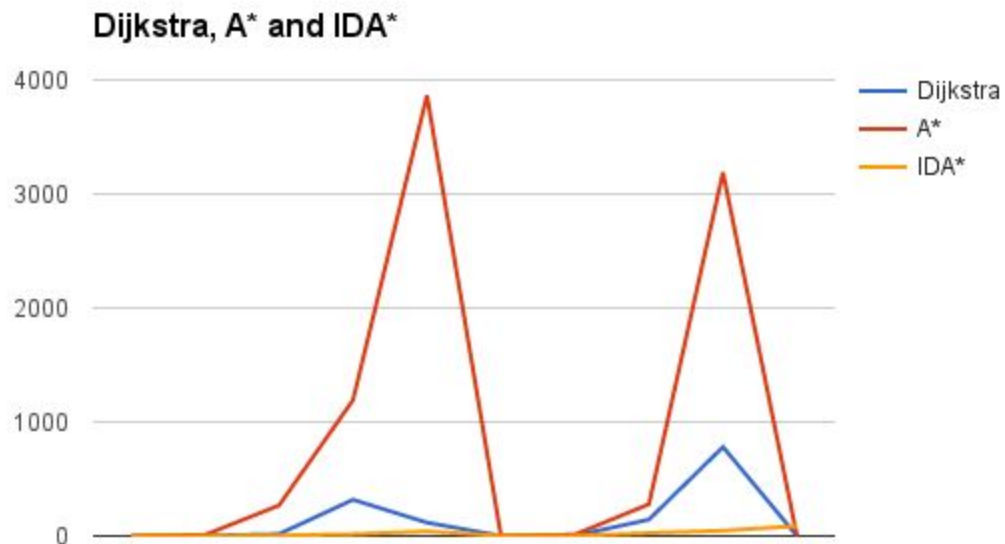
- Algoritmissa käytetään wikipedian pseukoodista poiketen visited-taulukkoa ja se alustetaan jokaisella search -operaation kutsukerralla, jotta solmuissa ei vierailta useaan kertaan yhdellä syvyysuuntaishauulla. Lisäksi naapuritsolmut järjestetään etäisyyden mukaan nousevaan järjestykseen. Näin algoritmia on hieman nopeutettu.
- Algoritmin aikavaativuus on siis $O(b^d)$
- Tilavaativuus on $O(|V|)$, koska kaikille taulukoille varataan tilaa solmujen lukumäärän verran. Näin jos ei oteta huomioon rekursiokutsujen varaamaa tilaa.

Suorituskyky- ja O-analyysivertailu

Kaavioiden pisteiden määrät ja solmujen etäisyydet (vakio, välillä 1-9). Kaavioissa aikayksikkö on millisekunti. Keko-kaaviossa (2,3) kolmas vertailukohde on Javan PriorityQueue. Lisätietoa Testaustilasto-dokumentissa.

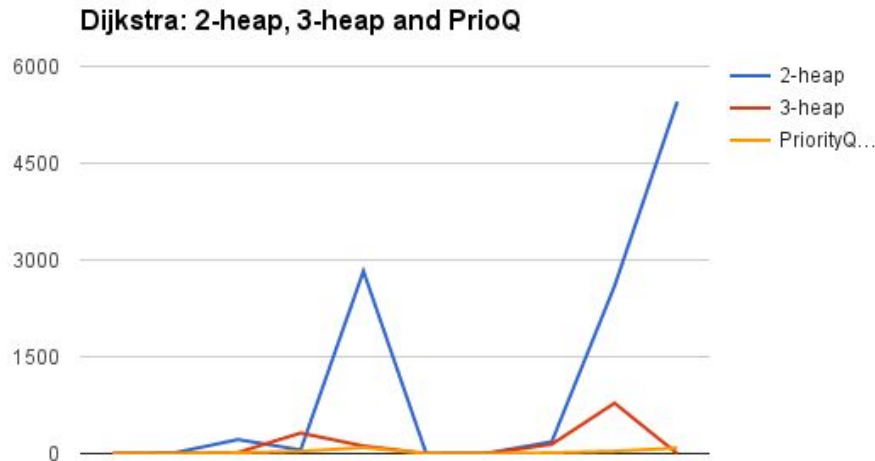
10*10	1
100*100	1
1000*1000	1
2000*2000	1
3000*3000	1
10*10	1-9
100*100	1-9
1000*1000	1-9
2000*2000	1-9
3000*3000	1-9

Kuvassa 1 näkyy miten koordinaatiston pisteiden kasvaessa sadasta 9 miljoonaan IDA* vasteajoissa ei näy suurta muutosta. Dijkstran (luvut 3-keolla) vasteajoissa on pieniä muutoksia ja 9 miljoonalla ajo vei jo liikaa aikaa (kaaviossa viiva käy nollassa virheellisesti). A*:n (luvut 3-keolla) vasteaika oli isoilla pistemäärillä epäilyttävän iso ja 9 miljoonalla ajo vei jo liikaa aikaa (kaaviossa viiva käy nollassa virheellisesti).



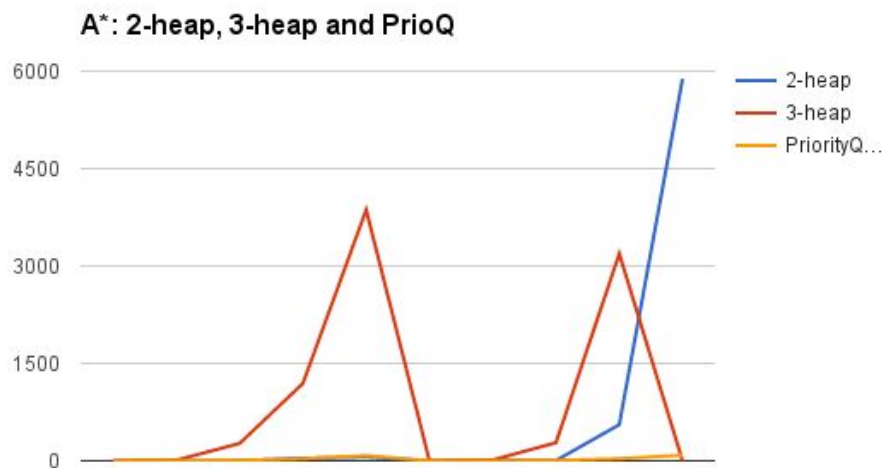
Kuva 1: Kaikki algoritmit ja niiden parhaat vasteajat.

Kuvassa 2 näkyy, että Dijkstra hyötyi 3-keosta. 2-keon suorituskky oli koetuksella kun pisteiden määrä kasvoi 9 miljoonaan. 3-keon (kun etäisyydet 1-9) viiva käy virheellisesti nollassa, kun lukuja ei enää saatu.



Kuva 2: Dijkstra vasteajat eri keko-ratkaisuilla.

Kuvassa 3 näkyy, että A* ei hyötynyt 3-keosta yhtä paljon kuin Dijkstra. 2-keon suorituskky oli koetuksella kun pisteiden määrä kasvoi 9 miljoonaan ja solmujen etäisyydet olivat välillä 1-9. 3-keon (kun etäisyydet 1-9) viiva käy virheellisesti nollassa, kun lukuja ei enää saatu.



Kuva 3: A* vasteajat eri keko-ratkaisuilla.

Työn mahdolliset puutteet ja parannusehdotukset

- Esteet tuottavat välillä ongelmia IDA* algoritmille, mutta en ole löytänyt yhteistä tekijää ongelmatapauksille. Ongelman huomaa, kun palautetun reitin varrella esiintyy este-koordinaatti.

Lähteet

- <https://www.cs.helsinki.fi/u/jkivinen/opetus/tira/k16/luennot.pdf> (Dijkstra, A*)
- https://en.wikipedia.org/wiki/D-ary_heap (3-keko)
- https://en.wikipedia.org/wiki/Iterative_deepening_A* (ida* pseudokoodi)
- https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search (aikavaativuus)