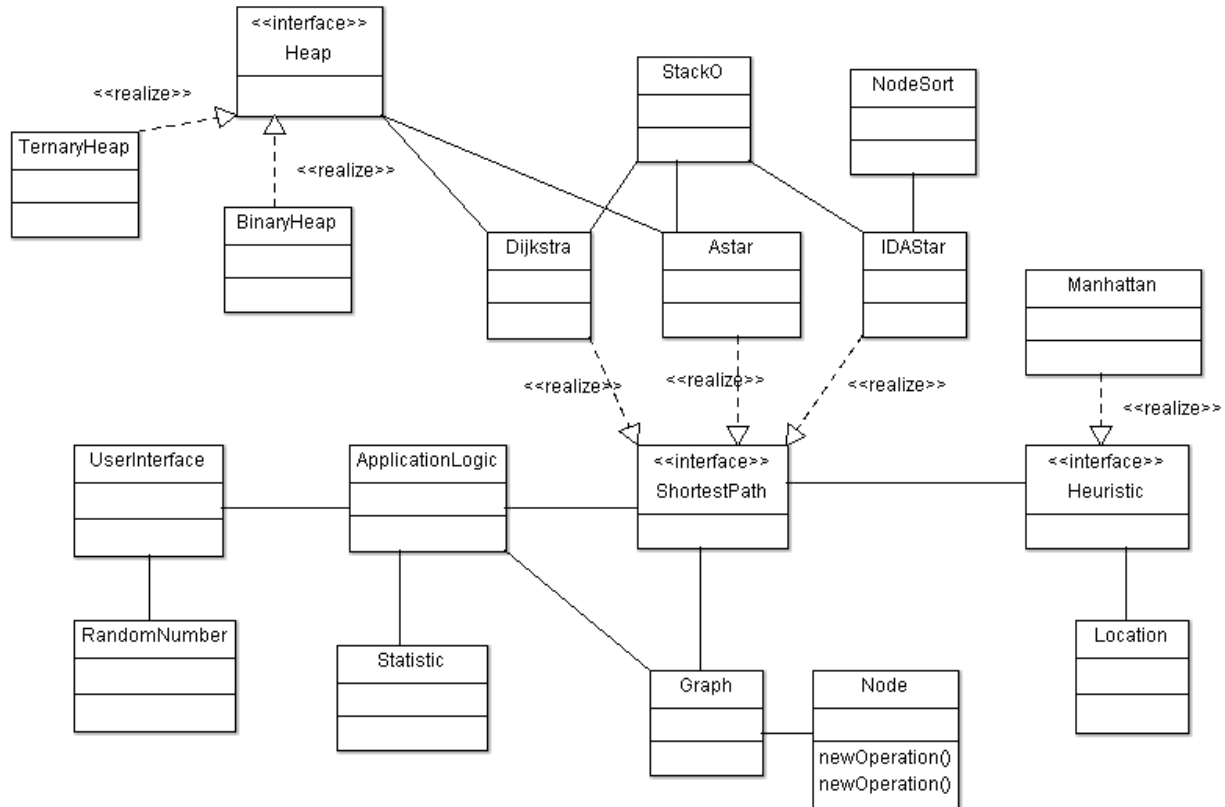


## Ohjelman yleisrakenne

Alla ylätasen luokkakaavio, joka kuvaa ohjelman rakennetta:



## Saavutetut aika- ja tilavaativuudet

### Dijkstra

- 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)  $O(|E| + |V| \log_3 |V|)$ .

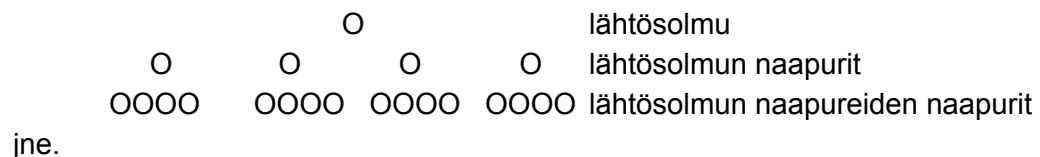
- Tilavaativuus on  $O(|V|)$ , koska kaikille taulukoille ja keolle varataan tilaa solmujen lukumäärän verran.

### A\*

- 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\*

- 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 naapurisolmut järjestetään etäisyyden mukaan nousevaan järjestykseen. Näin algoritmia on hieman nopeutettu.
- Algoritmin aikavaativuus on siis  $O(b^d)$  ja tilavaativuus  $O(d)$ , kun  $d$  on syvyys ja  $b$  lapsien määrä.

- Tilavaativuus on  $O(|V|)$ , koska kaikille taulukoille varataan tilaa solmujen lukumäärän verran.

Suorituskyky- ja O-analyysivertailu (mikäli työ vertailupainotteinen)

Työn mahdolliset puutteet ja parannusehdotukset

- IDA\* ja verkon sekä solmujen käyttäminen -> haun tekeminen puussa

Lähteet

- <https://www.cs.helsinki.fi/u/jkivinen/opetus/tira/k16/luennot.pdf> (Dijkstra, A\*)
- [https://en.wikipedia.org/wiki/D-ary\\_heap](https://en.wikipedia.org/wiki/D-ary_heap) (3-keko)
- [https://en.wikipedia.org/wiki/Iterative\\_deepening\\_A\\*](https://en.wikipedia.org/wiki/Iterative_deepening_A*) (ida\* pseudokoodi)
- [https://en.wikipedia.org/wiki/Iterative\\_deepening\\_depth-first\\_search](https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search) (aikavaativuus)