Toteutusdokumentti

Yleisrakenne

Ohjelman lähdekoodi on yritetty jakaa mahdollisimman loogisesti eri paketteihin. *Application*-paketti sisältää kaiken käyttöliittymäkoodin, ja *datastructures, graph, misc* ja *pathfinding* -paketit sisältävät nimiensä mukaisesti tietorakenteet, verkkoon liittyvät luokat, sekalaiset apuluokat ja itse reitinetsintäalgoritmit.

Ohjelmaan on implementoitu viisi reitinhakualgoritmia: A*, Dijkstran algoritmi, jump point search, breadth-first search ja depth-first search. Näistä kolme ensimmäistä antaa aina optimaalisen reitin, kun taas BFS lähes optimaalisen reitin ja DFS ensimmäisen reitin minkä sattuu löytämään. BFS:n epäoptimaalisuus johtuu tässä tapauksessa siitä, että verkossa on kahden pituisia kaaria, 1 ja √2.

Toteutetut tietorakenteet ovat lista (ArrayList), linkitetty lista (LinkedList), keko (Heap), hajautustaulu (joukko) (HashSet), jono (Queue), pino (Stack) ja assosiatiivinen hajautustaulu (myös kartta tai sanakirja) (HashMap). HashMapia ei käytetä projektissa ollenkaan, koska ajattelin sillä pystyväni tehostamaan Dijkstran algoritmin toimintaa, mutta myöhemmin se osoittautui huonoksi ideaksi. En kuitenkaan poistanut sitä, sillä se on täysin toimiva ja sille on myös kirjoitettu testit.

Aika- ja tilavaativuudet

Reitinhakualgoritmit

Taulukossa näkyvät aika- ja tilavaativuudet parhaimmassa ja huonoimmassa tilanteessa. Paras tilanne tarkoittaa sitä, että lähtöpiste on sama kuin lopetuspiste, ja huonoin sitä, että lopetuspistettä ei voi saavuttaa alkupisteestä sekä kaikki solmut ovat käytävä läpi.

Taulukoidut vaativuudet ovat minun implementaatioista, eli jossain muussa toteutuksessa esimerkiksi A*:n etäisyydet voi olla talletettu mappiin taulukon sijasta, jolloin parhaan tapauksen tilavaativuus olisi O(1).

Algoritmi	Aikavaativuus: Paras	Tilavaativuus: Paras	Aikavaativuus: Huonoin	Tilavaativuus: Huonoin
A*	O(1)	O(|V|)	O(|V|)	O(|V|)
Dijkstra	O(|V|)	O(|V|)	$O(\|V\|^2)$	O(|V|)
JPS	O(1)	O(|V|)	O(|V|)	O(|V|)
BFS	O(1)	O(1)	O(|V|)	O(|V|)
DFS	O(1)	O(1)	O(|V|)	O(|V|)

Jokaisen algoritmin (paitsi Dijkstran) huonoimman tapauksen aikavaativuus on O(|V|), sillä pahimmillaan koko syöte joudutaan käymään läpi. Tarkemmin ottaen jokaisen aikavaativuus olisi myös riippuvainen kaarien määrästä, mutta koska verkko on täydellinen ruudukko, kaarien määrä on lineaarisesti riippuvainen solmujen määrästä.

Vaikka jump point searchin ja A*:n aikavaativuus on teoriassa samaa luokkaa, niin niiden

huomattava nopeusero johtuu siitä, että A* käsittelee jokaista solmua tasavertaisesti tehden keko-operaatioita, kun taas JPS nimensä mukaisesti hyppää hyvin nopeasti useiden somujen yli tekemättä raskaita operaatioita.

Tietorakenteet

Taulukosta näkyvät kaikki tietorakenteet ja niiden yleisempien operaatioiden aikavaativuudet (average case). Jos tietorakenne ei tue operaatiota, niin se on merkitty viivalla.

Tietorakenne	insert(e)	remove(e)	find(e)	delete-min()	enqueue(e)	dequeue()	push(e)	pop()
Lista	O(1)	O(n)	O(n)	-	-	-	-	-
Linkitetty lista	O(1)	O(n)	O(n)	-	-	-	-	-
Hajautustaulu	O(1)	O(1)	O(1)	-	-	-	-	-
Hajautuskartta	O(1)	O(1)	O(1)	-	-	-	-	-
Keko	O(log n)	O(log n)	O(n)	O(log n)	-	-	-	-
Jono	O(1)	O(n)	O(n)	-	O(1)	O(1)	-	-
Pino	O(1)	O(n)	O(n)	-	-	-	O(1)	O(1)

Puutteet ja parannusehdotukset

Ohjelman suurin puute on varmaankin Dijkstran algoritmin käyttämä binäärikeko, jonka update-key() -operaatiota se kutsuu jokaisella askeleella. Update-key() kutsuu aina find() -operaatiota, jonka aikavaativuus on O(n) ja sen jälkeen päivittää avaimen, minkä aikavaativuus on O(log n). Tätä voisi parantaa marginaalisesti käyttämällä binäärikeon sijaan Fibonacci-kekoa, mutta se on liian iso työ suhteessa hyötyyn, sillä se ei poista find() -operaation tarvetta. Toinen vaihtoehto olisi pitää keon rinnalla jotain lookup-taulukkoa (esim HashMap, jonka teinkin valmiiksi), joka sisältäisi jokaisen elementin indeksin keossa. En kuitenkaan keksinyt tähän mitään käytännössä toimivaa ja järkevää ratkaisua, sillä yhden elementin muuttaminen keossa muuttaa myös monen muun elementin indeksiä.

Toinen huomionarvoinen asia on verkon toteutus. Voisin todennäköisesti nyt tehdä sen rakenteesta paljon paremman, kun tunnen tarkasti kuinka eri algoritmit toimivat ja mitä ominaisuuksia ne verkolta vaativat. Tässä vaiheessa projektia sen muuttaminen ei ole kuitenkaan ole kovin mielekästä.

Projektissa käytetyt lähteet

https://en.wikipedia.org/wiki/A*_search_algorithm
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
https://en.wikipedia.org/wiki/Breadth-first_search
https://en.wikipedia.org/wiki/Depth-first_search
https://harablog.wordpress.com/2011/09/07/jump-point-search/
http://zerowidth.com/2013/05/05/jump-point-search-explained.html
https://github.com/ClintFMullins/JumpPointSearch-Java
https://en.wikipedia.org/wiki/Binary_heap
https://en.wikipedia.org/wiki/Hash_table