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*), kaksi binäärikekoa (*BinaryHeap* ja *IndexedBinaryHeap*), hajautustaulu (*HashSet*), jono (*Queue*), pino (*Stack*) ja assosiaatiotaulu (tai kartta, hakurakenne) (*HashMap*). HashMapia ei käytetä projektissa ollenkaan, koska ajattelin sillä pystyväni tehostamaan Dijkstran algoritmin toimintaa, mutta myöhemmin toteutinkin sen HashSetillä, mikä osoittautui tehokkaammaksi. En kuitenkaan poistanut sitä, sillä se on täysin toimiva ja sille on myös kirjoitettu testit.

Syötteiden vaikutus suoritukseen

Seuraavaan taulukkoon on koottu algoritmit ja kuvailtu, millaisilla syötteillä ne toimivat parhaiten ja huonoiten.

Algoritmi	Toimii hyvin	Toimii huonosti
A*	Syötteet, joissa arvioitu matka maaliin vastaa hyvin todellista matkaa, esim. esteet pieniä ja ei vaadi pitkiä kiertoteitä.	Syötteet, jossa arvioitu matka maaliin vastaa huonosti todellista matkaa, esim. sokkelot monimutkaisilla reiteillä ja umpikujilla.
Dijkstra	Pienet syötteet ja syötteet, jotka rajoittavat etsintärintaman laajentumista liian suureksi, esim. kapeakäytäväiset sokkelot.	Suuret ja avoimet syötteet, sillä aikavaativuus ei ole lineaarinen ja ei sisällä mitään heuristiikkaa optimoidakseen etsintää.
JPS	Erittäin avoimet syötteet. Mitä suurempia tyhjiä alueita, sitä tehokkaammin algoritmi pystyy hyppimään niiden yli.	Paljon esteitä ja vähän tyhjää sisältävät esteet. Tällöin hyppyjen koko pienenee, ja suoritusnopeus lähenee A*:a
BFS	Samoin kuin Dijkstran tapauksessa, etsintärintaman laajenemista rajoittavat syötteet toimivat tehokkaammin, mutta syötteen koolla ei ole merkitystä aikavaativuuden lineaarisuudesta johtuen.	Avoimet syötteet, joissa etsintärintama pääsee kasvamaan suureksi vääriin suuntiin.
DFS	Syötteen laadulla ei ole juurikaan merkitystä, sillä algoritmi ei edes yritä löytää maalia tehokkaasti.	←

Pseudo- ja todellisen koodin väliset erot

Ainoa eteen tullut ongelma pseudokoodin ja konkreettisen toteutuksen välillä oli Dijkstran algoritmissa, jossa jokaisella iteraatiolla kutsutaan update-key(element) -operaatiota. Pseudokoodissa ei otettu kantaa prioriteettijonon toteutukseen, vaan siinä oletettiin, että operaation vaativuus on O(log n). Todellisuudessa päivitettävän elementti pitää myös ensin etsiä jonosta, minkä vaativuus on O(n). Tästä johtuen algoritmin kokonaisaikavaativuus oli $O(|V|^2)$. Myöhemmin keksin optimoida algoritmia käyttämällä prioriteettijonona kekoa, jossa jokainen alkio on indeksoitu hajautustauluun, jotta haku toimii vakioajassa. Se laskee algoritmin vaativuuden luokkaan O(|V| log |V|). Tämä optimointi toimi erittäin hyvin, eräälläkin syötteellä se laski suoritusaikaa jopa viiteen prosenttiin alkuperäisestä.

Aika- ja tilavaativuudet

Hakualgoritmit

Seuraavassa 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| log |V|)	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.

Tietorakenteet

Seuraavasta 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(n)	O(n)	O(log n)	-	-	-	-
Indeksoitu keko	O(log n)	O(log n)	O(1)	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)

Aikavaativuuden parantaminen

En usko, että hakualgoritmien aikavaativuuksia pystyy parantamaan nykyisestä lisäämällä tilavaativuutta. Sen sijaan tietorakenteiden osalta listan (ArrayList) operaatiot remove(e) ja find(e) on mahdollista parantaa toimimaan vakioajassa käyttämällä samanlaista sisäistä hajautusrakennetta kuin indeksoidussa keossa. Näin ollen myös jonon ja pinon vastaavat operaatiot saa toimimaan vakioajassa, jos muuttaa ne toimimaan ArrayListin päällä nykyisen LinkedListin sijaan.

Puutteet ja parannusehdotukset

Verkon toteutus tuntuu tässä vaiheessa projektia melko kömpelöltä. Voisin todennäköisesti nyt tehdä sen rakenteesta paljon paremman, kun tunnen tarkasti kuinka eri algoritmit toimivat ja mitä ominaisuuksia ne verkolta vaativat. Tässä vaiheessa sen muuttaminen ei ole kuitenkaan ole kovin järkevää.

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