Toteutusdokumentti.

Ohjelma rakenne koostuu kolmesta paketista, käyttöliittämästä, polunetsintä algoritmeista ja avustavista tietorakenteista.

Käyttöliittymän päätarkoitus on visualisoida algoritmien toimintaa, ja antaa käyttäjälle mahdollisuus muokata kenttää jossa kyseiset algoritmit toimivat.

Tietorakenteet ovat toteutettu algoritmien toimintaa varten, ja tukevat suoraa niiden toimintaa toteuttamalla tarvittavia työkaluja.

Itse algoritmit ovat ohjelmiston runko ja tekevät suurimman osan työstä, hyödyntäen käyttöliittymää ja avustavia tietorakenteita.

Map olio on toteutettu käyttäen kahta Object taulukkoa ja hashCode metodia. Arvot tallennetaan olioon parina siten, että taulukkoon ”key” asetetaan avain arvo tiettyyn indeksiin, ja taulukkoon ”value” asetetaan avainarvoa vastaava arvo samalle indeksille kuin ”key” taulukossa. Taulukon indeksi määritellään hashCode:n avulla, jota kutsutaan HashC metodissa.

Map olion tilavaativuus on vakio O(1). Koodissa on kaksi Object taulukkoa molemmat koolla 100000 jotta kaikki tarvittava varmasti mahtuu taulukon sisälle.

Aikavaativuus lisättäessä uusi arvo Map olioon on O(n) hashCode funktion johdosta, mutta kun Map oliosta haetaan tietoa, niin silloin aikavaativuus on O(1) koska hashCode löydetään silloin välimuistista eikä sitä tarvitse generoida uudestaan.

Queue olio on toteutettu keon avulla, Queue vastaa hyvin pitkälti Javan PriorityQueue olioita. Queue oliossa ydintoiminta tapahtuu heapify, ja heap_del_min metodeissa. Heapify metodi varmistaa, että keko on oikeassa järjestyksessä, eli että minimiarvo oikeasti löytyy sieltä mistä sen kuuluisikin. Heapify saa arvokseen jonkin indeksin ja hakee sen perusteella tarvittavat korjaukset kekoon. Heap_del_min metodi puolestaan hakee keosta pienimmän arvon ja kutsuu keon pienennystä. Heap_insert puolestaan kasvattaa keon kokoa yhdellä ja lisää arvon, ja kutsuu heapify:n varmistamaan, että keko on oikeassa järjestyksessä.

Queue olion tilavaativuus on lineaarinen O(n) keon suhteen, koska keon koko vaihtelee jatkuvasti algoritmien suorituksen aikana.

Aikavaativuus oliolla puolestaan on O(log n). Aikavaativuus saadaan aikaan keon avulla. Keon avulla saadaan hyödynnettyä rekursiota varsin tehokkaasti, eikä aina kun lisätään uusi arvo kekoon tarvitse tarkistaa kaikkia arvoja. Heapify metodi osaa tarkistaa arvot vain alueelta jolla ne ovat saattaneet muuttua, ja isEmpty metodi tarvitsee tarkistaa vain ensimmäisen arvon keon määritelmän perusteella.

Astar Algoritmi toimii hakemalla naapureita ja valitsemalla niistä aina sen joka on heuristiikan perusteella lähimpänä maalia. Heuristiikassa algoritmi laskee kullekin nodelle prioriteetti arvon, jonka avulla saadaan selvitettyä nodet, joiden avulla päästään maaliin. Mieluusti kohtuullisen lyhyttä reittiä pitkin. Astar algoritmi voi ajoittain haarautua etsimään maalia kahdesta tai useammasta suunnasta, jos ne vaikuttavat yhtä hyviltä, mutta kun toinen reitti paljastuu paremmaksi, niin heikompi reitti jätetään kesken.

Astar algoritmi koostuu Queue ja Map olioista ja algoritmin omasta koodista. Algoritmi ei oman boolean taulukon lisäksi muita tietorakenteita, joten algoritmin tilavaativuus määräytyy Queue olion tilavaativuuden perusteella, eli O(n)

Astar algoritmin aikavaativuus ottamatta huomioon tietorakenteita koostuu while loopista, jonka sisällä on toinen for loop ja useampia vakioaikaisia toimintoja. Aikavaativuudeksi siis koostuu O(n^m) kun otetaan huomioon myös tietorakenteiden kuluttama aika.

JumpPointSearch toimii nimensä mukaan hakemalla uusia nodeja ”hyppimällä”. Algoritmi etsii uudet nodet aina sivusuunnassa, pystysuunnassa ja vinosuunnissa, ja valitsee näistä sen, joka on lähimpänä maalia. Algoritmi käyttää samankaltaista heuristiikkaa kuin Astar algoritmissa, ja laskee yksinkertaisesti etäisyyden maalista 2-ulotteisessa kentässä. Algoritmi yleisesti ottaen löytää melko yksinkertaisen polun maaliin, siinä kun Astar useasti tekee hieman siksakki liikettä.

JumpPointSearch Algoritmi toimii samoilla tietorakenteilla kuin Astar, joten tilavaativuus on sama O(n).

Aikavaativuus sen sijaan on hieman monimutkaisempi ja se koostuu while loopista, jonka sisällä on kaksi peräkkäistä for looppia jotka molemmat sisältävät melko paljon vakioaikaisia operaatioita, joten aikavaativuudeksi saadaan O(n^2m), jossa n on while looppi ja m sisällä olevat for loopit. JPS:n aikavaativuus siis näyttäisi olevan hieman korkeampi kuin Astar algoritmin, mutta hyvin lähellä samaa.

Joten periaatteessa Astar ja JPS toimivat samalla aikavaativuudella, ja optimoinnin avulla JumpPointSearch:in saisi todennäköisesti toimimaan vielä nopeammin kuin Astarin. Testausdokumentissa kuitenkin ilmenee, että tyhjällä kentällä molemmat algoritmit suoriutuvat hyvin samankaltaisessa ajassa ja eroa oli karkea 0.10ms, suoritusnopeuden ero alkaa vasta näkyä, kun ajetaan algoritmit kentällä, jossa on esteitä.

Käyttöliittymä on toteutettu yksinkertaisena ruudukkona, johon voidaan piirrellä seiniä ja asettaa alku ja loppupisteet. Käyttöliittymän ruudukot ovat muodostettu Node olioista. Node oliot sisältävät kaiken oleellisen tiedon siitä mitä kussakin ruudussa on, ja myös tiedon siitä mikä Node on kunkin Noden vanhempi.

Käyttöliittymän Node olion tilavaativuus on O(1), koska olio ei sisällä muuta kuin yksittäisiä vakioita.

Noden aikavaativuus on myös vakio O(1), koska kaikki getterit ja setterit toimivat suoraan eikä missään kohtaa tarvitse käyttää esimerkiksi for-looppia.

Itse käyttöliittymä eli Grid olio saa myös tilavaativuudeksi vakion taulukon koon suhteen, eli O(1).

Gridin aikavaativuus sen sijaan on O(n^2) konstruktorin perusteella jossa on for-looppiin upotettu for-looppi taulukon läpikäyntiä varten. Flush metodi toimii myös ajassa O(n^2). ja DrawPath toimii lineaarisessa ajassa o(n) väritettävien nodejen suhteen. Loput metodeista toimivat vakiollisessa ajassa, ja eivät täten hirveästi vaikuta algoritmien aikavaativuuksiin.

Mouse olio kattaa käyttöliittymän hiiren toiminnallisuuden.