HW02 analüüs

Lähteülesanne 2021

Batuudikeskusesse ehitati uus batuudiväljak, mis on ristkülikuline, tehtud kõrvuti olevatest ühesuurustest ruudukujulistest batuutidest. Igal batuudil on erinev põrkejõud, mida väljendatakse täisarvuga ja mis võimaldab hüpata edasi vastava arvu batuute. Sellel batuudiväljakul korraldatakse võistlusi, kus start on loodenurgas (NW) ja tuleb jõuda täpselt kagunurka (SE). Hüpata tohib igast kohast ainult ida- või lõunasuunas batuudi põrkejõule vastava batuutide arvu võrra. Üle serva hüppamisel tuleb alustada algusest. On kaks tüüpi batuute: tavalised ja trahviga. Trahviga batuudi peale hüppamisel saab hüppaja nii palju trahviühikut, kui selle batuudi põrkejõud. Batuudiväljakul on ka seinad. Vastu seina hüpates hüppaja maandub eelmisele batuudile. Võidab see, kes jõuab lõppu kõige väiksema arvu hüpetega. Kui kahel mängijal on samapalju hüppeid, võidab see, kel on väiksem trahvi kogusumma. = Hüpata tohib ainult batuudil oleva põrkejõu numbri pikkuseid hüppeid. +- Igal hüppel võib lisaks hoogu ühe batuudi võrra maha võtta või juurde panna, st saab hüpata kas põrkejõu, sellest ühe võrra vähema või ühe võrra rohkema batuudide arvu võrra. Samas kui batuudi põrkejõud on null, siis hüpata ei saa ja „+-“ reeglit ei rakendata. Selle reegli vaatamata ei tohi endiselt hüpata põhja- ja läänesuunas.

1 Ülevaade lahenduskäikudest

Antud töö lahendamiseks on kasutatud mitu meetodit - maksimaalselt hea tulemuse andis nii Breadth First Search (BFS) otsingualgoritm *meetod 1, 2, 3* kui ka Depth First Search (DFS) algoritm *meetod 4* . Klassikaliselt sobib mugandatud BFS ja Dijkstra ning selle edasiarendused lühima tee otsingu jaoks paremini.

- *Meetod 1* - Kasutatakse hüpete jaoks graafe, mis määravad hüppamise kõik võimalikud kombinatsioonid. Seejärel otsitakse graafidest BFS algoritmiga parimad teekonnad ning lõpuks leitakse uuesti DFS algoritmiga vähima trahviga teekond. See lahendus sobib ja leiab lõpliku teekonna, kuid suuremate siseandemetega tekib aegluse probleem. Seetõttu on meetodis 2 loobutud graafidest.
- *Meetod 2* - keskendub BFS otsingule graafe kasutamata. Lisaks arvestatakse ka kohe trahvisüsteemiga, et mitte korrata sama algoritmi mitu korda. See meetod jookseb võrreldes eelmisega poole kiiremini, kuid siiski jääb selle kiirusest väheks testides testLevel5, testLevel6. Selleks, et seda probleemi lahendada, on antud meetodit optimiseeritud ning proovitud meetodit 3.
- *Põhimeetod Meetod 3* - Optimeeritud on suuremad meetodi 2 kitsaskohad - näiteks seinte otsing on optimaalsem ja kiirem. Lisaks on võetud kasutusele primitiivsed struktuurid, ei kasuta *HashMap'e*, vaid pigem maatrikseid, mis annab veidi parema tulemuse. Meetod 3 on antud ülesande lahendi *põhimeetod*, mis küll läbib kõik testid, kuid testLevel5, testLevel6 läbimiseks lõpetab napilt enne lävendi lõppu. Seetõttu on ka proovitud veel üht lähenemist - meetod 4.
- *Meetod 4* - DFS algoritm on ülesehituselt sarnane BFSiga, kuid tööpõhimõte on teine. See algoritm tundub töötavat küll veidi efektiivsemalt, kuid programmi lõpetamise kiirus ei erine BFS-st suurel määral. DFS ei tundu klassikaliset antud ülesande lahendiks kõige paremini, see vajab häälestamist ja optimeerimist.

BFS leiab korrektse lahenduse, kuna algoritm kontrollib kõiki hüpete võimalusi, samal ajal hoiab meeles, milline lahend on parim. Otsing hargneb alguspunktist laiali, leitakse järgmised hüpped ja seejärel järgmised ... jne. Kontrolli käigus võrreldakse käesolevat teekonna pikkust parima teadaoleva pikkusega läbi mingi punti, mida hoitakse mälus. Juhul kui seda hüpet on juba külastatud, kontrollib kõigepealt teekonna pikkust, eelistatakse alati lühimat. Kui pikkused on võrdsed, siis kontrollib, läbi millise hüppe

on väiksem trahv. Õige lahenduse leidmiseks aitab kaasa parima pikkuse, väiksema trahvi ja parima teekonna meeleshoidmine ja uuendamine, kui otsing leiab midagi sobivamat. Kui BFS töö lõpetab, jääb järgi vaid teekond õigsesse formaati teisendada ja tulemus on käes.

Keerukuse hinnanguks kujuneb maatriksi andmestruktuuri tõttu halvimal juhul $O(n^2)$

2 Meetodite detailsemad kirjeldused

2.1 Meetod 1 kirjeldus

Klassid *Vertex, Graph, BFSGraph.* Otsingualgoritm BFS.

Vertex kirjeldab ühte *node'i* - tema tüüp, trahvi suurus ja ühendused. Vertexid on omavahel ühendatud. *Graph* klassis luuakse ühenduste struktuur. *BFSGraph* klassis olev otsingufunktsioon tegeleb sellest struktuurist otsinguga.
Idee on luua võimalike teekondade määramiseks graafi struktuur. Graafi loomisel arvestatakse hüpetega (=, +, -) ja luuakse seosed kuidas punktist edasi liikuda. Pluss-miinus variandi puhul on tõenäoline, et ühest graafist hargneb välja 6 ühendust järgmistesse graafidesse. Graafi ehitamine on antud meetodi keerukam osa, selles faasis kontrollitakse ka võimalikke seinu, et luua õiged seoses punktide vahel. Kui graaf on loodud, käivitatakse parimate/lühimate teekondade otsing ja kui need on leitud, käivitatakse väiksema trahviga teekonna otsing.

```
public class HW02 {

    @Override
    public Result play(Trampoline[][] map) {
        Graph graph = new Graph();
        graph.createGraph(map);
        var start = graph.getStartVertex();
        var end = graph.getEndVertex();

        BFSGraph bfs = new BFSGraph();
        var shortestRoutes = bfs.searchAllShortestRoutes(start, end);
        var routeWithLowestFine = bfs.searchRouteLowestFine(end, start, shortestRoutes);
        var finalRoute = bfs.getRoute(routeWithLowestFine, start, end);
        var jumps = routeToJumps(finalRoute, end);

        return new Result() {
            ...
            }
        };
    }
}
```

2.2 Meetod 2 kirjeldus

Klass *BFSMatrix* peafunktsioon straightSearch(Trampoline[][] map)

Graafiga ülesande lahendus võtab liiga palju aega, suurem osa kulub graafi ja vajalike ühenduste loomisele. Meetod 2 on ilma graafita variant. Idee BFSiga otsinguks jäi samaks, seega teekonna otsingu idee ei muutunud. Küll aga kontrollitakse kohe otsingu ajal nii parimat distantsi kui ka vähimat trahvi juhul, kui pikkus ühest ja teisest teekonnast on samad. See tähendab, et loobutakse ühest lisaotsingust, mis meetodis 1 tehakse. Kui parim teekond on teada, väljastatakse see sobivasse formaati. Meetod 2 on kiirem kui meetod 1, kuid leiduvad ka kitsaskohad, mida on meetodis 3 optimeeritud.

```
public class HW02 {

    @Override
```

```
    public Result play(Trampoline[][] map) {
    BFSMatrix bfs = new BFSMatrix();
        var res = bfs.straightSearch(map);
        return new Result() {
            ...
            }
        };
    }
}
```

2.3 Meetod 3 kirjeldus

Klass *BFSMatrix* peafunktsioon straightSearchWithoutMaps(Trampoline[][] map) koos abifunktsioonidega.

Lahenduskäiguks on kasutusse võetud maatriksid eelnevate HashMapide struktuuride asemel. Java IntelliJ Profiler tool näitas, et suure osa ajast veedab otsing programmi käivituse ajal hashmapist andmete otsingule. See probleem kaob maatriksite kasutuselevõtuga. Teine optimeerimine toimub seinte otsingus, kus iga punkti jaoks kontrolliti seinu ainult sobivas vahemikus otse sisendandmetest. Selle asemel hoitakse mälus info rea/veeru seinte kohta. Kui punkt esmakordselt otsingusse seinud otsima läheb, luuakse seinte info jooksvalt kogu rea/veeru ulatuses. Selline andmestik lubab programmil kiiremini seintest eelnevatele punktide jaoks arvutusi teha. Kasutatav algoritm ei ole muutunud, kasutatakse BFS, teatavate optimeerimistega. Mälus hoitakse info parima teekonna, trahvide ja parima distantsi kohta iga punkti kohta. Antud meetod läbib küll kõik testid, kuid vaevaliselt alla 5-sekundi lävendi, ilmselt on võimalik muu algoritmiga parem tulemus saavutada.

```
public class HW02 {

    @Override
    public Result play(Trampoline[][] map) {
        BFSMatrix bfs = new BFSMatrix();
        var res = bfs.straightSearchWithoutMaps(map);
        return new Result() {
            ...
            }
        };
    }
}
```

```
public class BFSMatrix {
    public Method2ResultWithoutMaps straightSearchWithoutMaps(Trampoline[][] map) {
        //null checks and var inits

        unvisitedQueue.add(start);
        while (!unvisitedQueue.isEmpty() && !found) { //where can still jump
            var point = unvisitedQueue.poll();
            if (point.equals(end)) found = true;

            var children = getLandingPoints(point, map); //new jumps
            var fine = getAccumulatedFine(point);
            var newChildDistance = getAccumulatedDistance(point) + 1;

            for (var child : children) {
                var currentChildDistance = getAccumulatedDistance(child);
                var newChildFine = fine + getTrampolineFine(child);
                var currentChildFine = getAccumulatedFine(child);

                if (notVisited(child) || newChildDistance < currentChildDistance) { //
                    shorter
                    updateDistance(child, newChildDistance);
                    updateRoute(child, point);
                    updateFine(child, newChildFine);
                    unvisitedQueue.add(child);

```

```
                } else if (newChildFine < currentChildFine && newChildDistance ==
                    currentChildDistance) { //same distance but better
                    updateRoute(child, point);
                    updateFine(child, newChildFine);
                    unvisitedQueue.add(child);
                }
            }
        }
        var totalFine = getAccumulatedFine(end);
        return new Method2ResultWithoutMaps(routeMap, totalFine, end, start);
    }
}
```

2.4 Meetod 4 kirjeldus

Klass *DFS* peafunktsioon search(Trampoline[][] map) koos abifunktsioonidega.

Viimaseks meetodiks kasutatakse DFS otsingualgortimi. See algoritm tundub küll efektiivsem, kuid selle optimeerimine lühima ja parima teekonna arvutuseks osutus arvatust keerulisemaks. Idee on sama, mis BFS, kuid DFSis kasutatakse stack andmestruktuuri, mis lubab teekonnal koheselt lõppu jõuda. DFS puhul läheb otsing sügavuti, mitte ei hargne laiali, otsib lõpp-punkti minnes ühte haru pidi kogu aeg. Kui ühte teed pidi ei leia, siis tuleb sammu tagasi ja proovib teist teed pidi lõppu leida. Kui child on läbi käidud, aga uus distants sellele on parem kui enne, siis tuleb ta uuesti stacki panna, et läbi selle childi uuesti tee lõppu arvutada ja uuendada lõpuni viivad distantsid väiksemaks. Üldiselt võib öelda, et DFS meetod 4 ja BFS meetod 3 on antud ülesande lahendamisel ajaliselt samaväärsed, keerukus on sama.

```
DFS dfs = new DFS();
        var res = dfs.search(map);
        return new Result() {
          ...
        };
```

3 Lahendusmeetodite ja algoritmide katsetused

Selleks, et võrralda kõiki meetodeid, on loodud sisendmaatriksid 1000, 1500 ja 2000 elemendiga. 2000 elemendi testis on maatriksis iga punkti (trampoline) väärtuseks 2, hüpete arv ühest punktis on 6. Selgelt on näha, et graafi loomine võtab liiga kaua aega, samas, algoritm BFS optimeerimisel on võimalik saavutada testri täispunktisumma. On ka näha, et DFS ja BFS algoritmide implementatsioon on enamvähem võrdväärse tulemusega.

Meetod 1 **BFS Graaf**	Meetod 2 **BFS HashMap**	Meetod 3 **BFS Maatriks**	Meetod 4 **DFS Maatriks**
1000x1000 ühed graafi loomine 8,650s otsing 4,593s parima otsing 3,984s teekond 0,001s	1000x1000 ühed otsing 10,586s teekond 0,012s	1000x1000 ühed otsing 2,059s teekond 0,013s	1000x1000 ühed otsing 1,750s teekond 0,013s
1500x1500 ühed graafi loomine 32,346s otsing 12,769s parima otsing 12,528s teekond 0,004s	1500x1500 ühed otsing 33,368s teekond 0,012s	1500x1500 ühed otsing 2,522s teekond 0,013s	1500x1500 ühed otsing 2,451s teekond 0,011s
2000x2000 kahed ei lõpetanud	2000x2000 kahed otsing 123,636s teekond 0,016s	2000x2000 kahed otsing 3,986s teekond 0,011s	2000x2000 kahed otsing 3,453s teekond 0,008s

Kokkuvõte

Ülesande lahendamisel on kasutatud mitmeid meetodeid, põhiliselt on kasutatud BFS ja DFS algoritme. Selleks, et läbida testid, on vaja BFS ja DFS optimeerida. Käesolev töö andis võimaluse proovida ja võrrelda erinevaid lähenemisi ja andmestruktuure. Antud lahenduse keerukus on halvimal juhul $O(n^2)$, testLevel5 on läbitud 4,648 sekundiga (timeout 5s), seega on ilmselt võimalik ülesannet ka paremini lahendada.