HW02 seletuskiri

1 Lihtsustus graafiks

Kogu batuutide väljakut võib vaadelda suunatud graafina, kus tippudeks on batuudid ning iga tipu naabriteks on need tipud (ehk batuudid), kuhu esialgselt hüpata saab.
Selle jaoks on loodud *Node* klass, mis esialgu hoiab endas naabreid massiivina ning infot batuudi kohta. Lõplik versioon lahendusest salvestab lahenduse käigus tipu külge ka infot selle kohta, kui hea on optimaalne lahendus sellest punktist alates ning infot selle kohta, millise naabrini tuleks hüpata, et saada optimaalne lahendus.

2 Definitsioonid ja notatsioon

Olgu suvaline graafi tipp a.
Olgu tipp, kuhu me tahame jõuda s. (antud juhul väljaku all paremal nurgas asuv tipp)

Olgu F^a lahendite hulk tipule a. Selle hulga elementi, ehk lahendit tähistame f_i^a ning lahendi all on mõeldud hüpete ennikut, mis viib tipust a tipuni s.

Olgu $H(f_i^a)$ hüpete arv lahendi f_i^a korral.
Olgu $P(f_i^a)$ trahv, mis kogutakse hüpete käigus lahendi f_i^a korral.

Vastavalt ülesandepüstitusele on üks lahend parem kui teine, siis kui ta teeb vähem hüppeid või sama palju hüppeid, aga kogub vähem trahvi.
Formaalsemalt f_i^a on parem kui f_j^a siis ja ainult siis, kui

$$H(f_i^a) < H(f_j^a) \vee \Big(H(f_i^a) = H(f_j^a) \wedge P(f_i^a) < P(f_j^a)\Big)$$

Ütleme, et $f_i^a > f_j^a$ tähendab, et f_i^a on parem kui f_j^a.

Olgu $G(a)$ kõige parem ehk optimaalne lahend tipule a.
Formaalsemalt, olgu $G(a)$ selline lahend tipule a, et $\forall f_i^a \in F^a : G(a) \geq f_i^a$

3 Lahendusidee

Olles tipus a, mille naabrite hulk on A, liigume sellele tipule $a_i \in A$, millelt optimaalne lahend on parem, kui ükskõik milliselt teiselt tipu a naabertipult, ehk $\forall a_j \in A : G(a_i) \geq G(a_j)$.

3.1 Optimaalsuse põhjendus

Liikudes tipult a_i tipule a_j tuleb lahendusse juurde üks hüpe ning tipu a_i trahvi kogus, olenemata sellest, millisele naabrile hüpata.
Ühtlasi kahe lahenduse võrdluse tulemus ei muutu kui mõlemale liita sama hüpete arv ning sama kogus

trahvi.
Seega optimaalne lahendus tipule a_i saadakse, kui võtta tema naabrite hulgast selline, mille jaoks optimaalne lahendus on kõige parem. (ning sellele lisada hüpe tipult a_i valitud naabertipule)

3.2 Pseudokood

Eelneva põhjal saab kirja panna rekursiivse algoritmi järgnevalt

```
function SOLVE(node)
    if node = endnode then
        return baseSolution
    bestSolution ← null
    for neighbour in node.neighbours do
        solution ← solve(neighbour)
        if solution > bestSolution then
            bestSolution ← solution
    return combine(node, bestSolution)
```

4 Meetodi keerukus

4.1 Parim juht

Parimal juhul saab algusest hüpata vaid rea või veeru lõppu ning rea või veeru lõpust alla paremale nurka ning sellisel juhul vaatab lahendus läbi ainult konstantse hulga teekondi ning keerukuseks on $O(1)$.
Punktis 7 kirjeldatud põhjustel tuleb tegelikult kogu keerukuseks $O(n)$, kuid siinne arutelu on siiski informatiivne sügavuti otsingu enda keerukuse hindamiseks.

4.2 Halvim juht

Olgu väljaku pikima külje pikkus n, ning teise külje pikkus an, kus $0 < a \leq 1$.
Eeldame, et keskmise hüppe pikkus on b.
Iga teekonna hüpete pikkuste summa peab olema $an + n$, järelikult keskmiselt on lahenduses $\frac{an+n}{b}$ hüpet.
Igal tipul on 0 kuni 6 naabrit. Kuna meid huvitab halvim juht, siis eeldame, et peaaegu alati 6.
Igale *solve* meetodi väljakutsele vastab ühe sammu tegemine, seega lahenduse leidmiseks peab *solve* meetod jõudma rekursiooniga $\frac{an+n}{b}$ sügavusele ning igal sügavuse suurenemisel kutsutakse meetodit välja 6 korda, seega kokku kutsutakse meetodit välja $6^{\frac{an+n}{b}}$ korda.
Siit on näha, et halvimal juhul $a = 1$ ning $b = 1$, ja meetodit kutsutakse seega välja $6^{2n} = 36^n$ korda, ehk halvima juhu keerukus on $O\left(36^n\right)$.

5 Edasiarendused

5.1 Vahetulemuste meelde jätmine, ehk memoization

Märkame, et tippe läbi käies jõuab lahendus ühe ja sama tipuni mitu korda ning, et mingist konkreetsest tipust algav lahenduse alamosa ei sõltu sellest, mis teed pidi sinna tipuni jõuti.
Seega kui mingi tipu kohta on lahendus leitud, saab selle meelde jätta ning järgnevatel väljakutsetel seda lahendust kasutada.

5.1.1 Keerukus peale edasiarendust

Parima juhu keerukus ei muutu.
Halvima juhu keerukuse leidmiseks enam ei sobi eelmine metoodika, kuna nüüd lõppeb rekursioon tihti varem ära (kuna jõuti tipuni, mille jaoks oli lahendus juba olema).

Nüüd tuleks märgata, et iga tipu jaoks leitakse lahendust vaid 1 kord ning selle leidmine on konstantse keerukusega operatsioon, kui kõikide naabrite alamlahendused on olemas. Kui tipu mõnel naabril veel puudub lahendus, siis tuleb see leida, kuid loeme selleks kuluva aja naabri enda lahenduse leidmise alla, mitte esialgse tipu lahenduse leidmise alla.
Selliselt mõeldes saab väita, et tipu jaoks alamlahenduse leidmine on konstantse keerukusega operatsioon, kuid tuleb meeles pidada, et halvimal juhul peame leidma alamlahendused iga tipu jaoks.
Kuna kokku on an^2 tippu, siis tuleb kogu ajaks can^2, kus c on ühe tipu alamlahenduse leidmiseks kuluv konstantne aeg.
Kokkuvõttes keerukuseks tuleb $O\left(n^2\right)$

5.1.2 Edasiarenduse jaoks vajalikud andmestruktuurid

$HW02$ klassi loodi massiiv *solved*, mis hoiab infot selle kohta, kas tipp on lahendatud või mitte.
Tipu enda kohta käivaid andmeid hoitakse tipu enda küljes $Node$ klassis.
$HW02$ klassis meetod $markJump$ sisaldab loogikat selle kohta, mida teha alamlahenduse meelde jätmiseks.

5.2 Kärpimine

Kui tipu naabrid järjestada selliselt, et väljaku peal kaugemad käiakse enne läbi, siis on lootust varakult leida mingi peaaegu optimaalne lahend.
Kui see meelde jätta, siis muude otsinguharude uurimisel saab vaadata mitu hüpet on siiamaani tehtud (ja kui palju trahvi kogutud) ning kui see on halvem kui seni leitud parim lahendus, siis otsingu sellest harust lõpetada, kuna sealt ei tule enam kindlasti parimat lahendust.

5.2.1 Keerukus peale edasiarendust

Parima juhu keerukus ei muutu.
Halvima juhu keerukus samuti ei muutu, kuna halvimal juhul peame ikkagi iga tipu jaoks alamlahenduse leidma.

5.2.2 Edasiarenduse jaoks vajalikud andmestruktuurid

$HW02$ klassi loodi muutujad, mis hoiavad infot seni leitud parima lahenduse kohta.
$HW02$ klassis meetod $markJump$ nüüd vajadusel kirjutab ka parima lahenduse üle.

Tipu naabrite järjestamiseks selliselt, et kaugemad tuleksid enne, tuleb need ära sorteerida. Naabreid on küll maksimaalselt 6, kuid seda sorteerimist tehakse potentsiaalselt n^2 korda, seega tuli tähelepanu pöörata ka sellele, et seda võimalikult efektiivselt teha.
$HW02$ klassis meetod $addNeighbours$ käib kummaski suunas batuute läbi järjest, seega paremal olevad naabrid on juba sorteeritud ning all olevad naabrid on juba sorteeritud. Sellepärast $Node$ klassis olevale $addNeighbours$ meetodile antakse argumentidena paremal ning all olevad naabrid eraldi ning alles siin järjestatakse kõik naabrid ühte massiivi kasvavalt lineaarse keerukusega.

5.3 Rekursiooni asendamine magasiniga

Kuna rekursioon töötab taustal analoogselt magasiniga, siis saab iga rekursiivse algoritmi ümber kirjutada iteratiivseks algoritmiks, mis lisab ning eemaldab asju magasinist.
Keerukust see ei mõjuta, kuid võib vahel algoritmi mingi konstantse kordaja võrra paremaks teha. Antud juhul viimaste testide läbimiseks oli just selline ümberkirjutamine vajalik.

5.3.1 Edasiarenduse jaoks vajalikud andmestruktuurid

Magasini implementeerimiseks luuakse $Node$ massiiv ning muutuja $stackCounter$, mis hoiab meeles, mitmendal indeksil on magasini pealmine element.

Elemendi lisamiseks lisatakse massiivi uus element ning suurendatakse *stackCounter* muutujat.
Elemendi eemaldamiseks vähendatakse *stackCounter* muutujat.

6 Erinevad variandid

Erinevaid lahendusvariante on siin kirjeldatud kas 2 või 8, sõltuvalt sellest, millal lahendusi "erinevateks" lugeda.
Kokku oli 3 üksteisest sõltumatut edasiarendust. Kui erinevateks lahendusteks lugeda erinevate edasiarenduste kombinatsiooni, siis on kokku erinevaid variante $2^3 = 8$. Alternatiivselt võib üheks variandiks lugeda esialgset rekursiivset lahendust, mille kohta ka pseudokood toodi, ning teiseks variandiks lõplikku lahendust, mis kõik testid läbis.
Lisatud on 3 erinevat koodi, üks neist vastab esialgsele lahendusele, üks lõplikule variandile ning ühes on võimalik igat edasiarendust vastavalt soovile sisse või välja lülitada.

7 Graafi moodustamise keerukus

Eelnev keerukuse analüüs ei arvestanud ajaga, mis kulub väljaku muutmisel graafiks. Täpsemalt siis *addNeigbours* nimeline meetod HW02 klassis, mis võtab batuudi ning käib läbi kõik batuudid, kuni jõuab hüppe pikkuse või seinani. Sellise meetodi keerukus halvimal juhul on $O(n)$ ning kui seda teha iga batuudi jaoks, tuleb kokku $O(n^3)$.
Sellepärast lõplikus variandis kutsutakse seda meetodit välja alles siis, kui batuudile on vaja naabreid leida. Sellisel juhul sõltub meetodi välja kutsumiste arv sellest, kui pikad hüpped on väljakul või kui tihedalt on asetatud seinad.
Siinkohal oleks mõistlik eristada kahte võimalust.

7.1 Konstantse pikkusega hüpped / konstantsete vahedega seinad

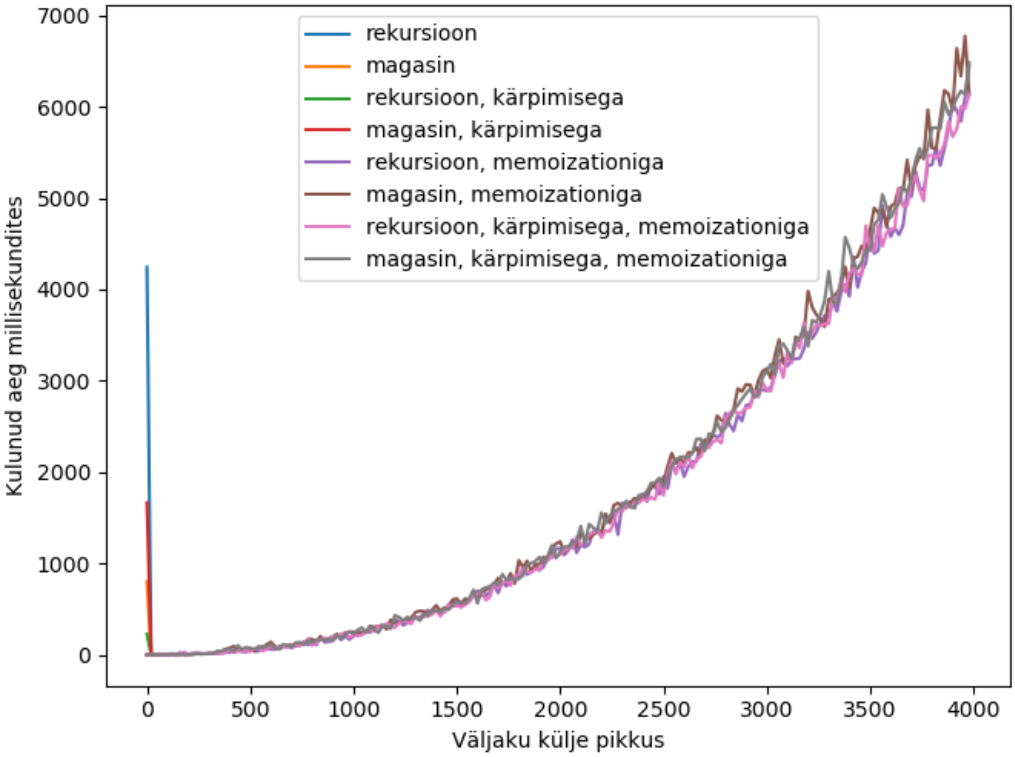
See on võimalus, mida varem vaadeldi halvima juhuna.
Selle korral $HW02$ klassi $addNeighbours$ meetodit kutsutakse välja küll n^2 korda, kuid iga väljakutse on konstantse keerukusega, seega halvima juhu keerukus on endiselt $O(n^2)$.

7.2 Konstante arv hüppeid lahenduses / konstante arv seinud

See on võimalus, mida varem vaadeldi parima juhuna.
Selle korral $HW02$ klassi $addNeighbours$ meetodit kutsutakse välja konstante arv kordi ning iga väljakutse keerukus on $O(n)$, seega parima juhu keerukus iga lahendusmeetodi puhul on tegelikult $O(n)$.

8 Katselise võrdluse tulemused

Algoritmide ajaliseks võrdluseks genereeriti väljakuid, kus iga batuudi hüppetugevus oli 1 ning mõõdeti aega, mis erinevatel algoritmidel selle lahendamiseks kulus.
Tulemused vastavad laias laastus sellele, mida keerukuse analüüs ennustas, ehk ainuke asi, mis kiirusele märgatavat mõju avaldab, on memoization.


rekursioon
magasin
rekursioon, kärpimisega
magasin, kärpimisega
rekursioon, memoizationiga
magasin, memoizationiga
rekursioon, kärpimisega, memoizationiga
magasin, kärpimisega, memoizationiga
Kulunud aeg millisekundites
Väljaku külje pikkus
0
1000
2000
3000
4000
5000
6000
7000
0
500
1000
1500
2000
2500
3000
3500
4000