Contents

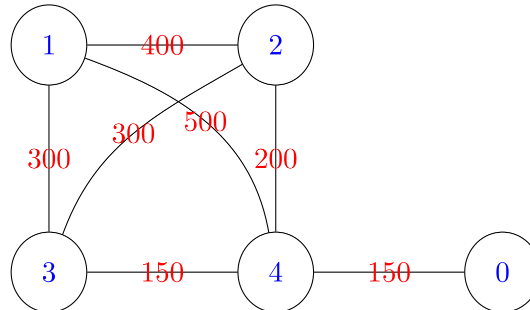
1 Uppgift 1 Elnät för Notställsbelysning **2**
Beskrivning . . . 3
Tidskomplexitet . . . 3
Korrekthet . . . 3

2 Uppgift 2 Viktiga möten **4**
Problem Modellering . . . 4
Rekursionsekvation för den optimala totala prioriteten . . . 5
Polynomisk Lösning . . . 6
Skapa Mötesgraf . . . 6
Skapa Mötesgraf: Tidskomplexitet . . . 7
Skapa Mötesgraf: Korrekthet . . . 7
Söka Mötesgrafen . . . 8
Söka Mötesgrafen: Tidskomplexitet . . . 8
Söka Mötesgrafen: Korrekthet . . . 9
Bäst schemaläggningen . . . 9
Bäst schemaläggningen: Tidskomplexitet . . . 9
Bäst schemaläggningen: Korrekthet . . . 10
Hitta Bästa Schemat . . . 10

3 Uppgift 3 Optimal Analys av Ringar i Vas **11**
Problem Modellering . . . 11
Den Triviala Vasen . . . 11
Den Godtyckliga Vasen . . . 12
Den Godtyckliga Vasen: Tidskomplexitet . . . 13
Den Godtyckliga Vasen: Korrekthet . . . 13
Alla Ringar i en Vas . . . 14
Alla Ringar i en Vas: Tidskomplexitet . . . 15
Alla Ringar i en Vas: Optimal . . . 15
Alla Ringar i en Vas: Korrekthet . . . 16

1 Uppgift 1 Elnät för Notställsbelysning

Givet grafen utav numrerade notställ 1-4 och eluttaget 0, där kanternas vikt representerar längden mellan notställ. Hitta det minsta längden total sladd som behövs för att koppla alla notställ samt hur många grenar varje uttag har och hur sladdarna bör kopplas.



Intuitivt så måste problemet lösas med hjälp av ett minimalt spännande träd eftersom om en cykel fanns så skulle en av sladdarna kunna tas bort utan att någon av lamporna skulle släckas, och det är det uppenbart enligt definition att det minimala spännande trädet är nödvändigt för att ha den kortaste totala längden på sladdarna, som besöker varje notställ.

Algorithm 1 Notställsbelysning

Input: M
Output: RES

1: $L \leftarrow M.length$
2: $C \leftarrow \{1\} \cdot L$ //Vektor av ettor av längd L
3: $V \leftarrow \{L\}$ //Lägg tubaspelarens notställ i listan av besökta notställ
4: $P \leftarrow \{(0, L)\}$ //Lägg sladden från eluttaget till tubaspelarens notställ i listan av sladdar
5: **while** $V.length < L$ **do**
6: $min \leftarrow \infty,\ s_{min} \leftarrow 0,\ t_{min} \leftarrow 0$
7: **for** $s \in V$ **do**
8: **for** $t \in (1, L)$ **do**
9: **if** $t \notin V \wedge M[s][t] < min$ **then**
10: $min \leftarrow M[s][t]$
11: $s_{min} \leftarrow s$
12: $t_{min} \leftarrow t$
13: **end if**
14: **end for**
15: **end for**
16: //min är längden av den kortaste sladden från ett besökt (s_{min}) till ett obesökt (t_{min}) notställ
17: $V \leftarrow V + \{t_{min}\}$ //lägg till t_{min} i V
18: $P \leftarrow P + \{(s_{min}, t_{min})\}$ //lägg till (t_{min}, s_{min}) i P
19: $C[s_{min}] \leftarrow C[s_{min}] + 1$ //inkrementera antalet uttag från s_{min}
20: **end while**
21:
22: $RES \leftarrow \emptyset$
23: **for** $(s, t) \in P$ **do**
24: $RES \leftarrow \{(M[s][t], C[s], s, t)\} + RES$
25: **end for**

Beskrivning

För att hitta den minsta spännande sladduppsättningen kan en modifierad version av Prims algoritm användas. I varje iteration av *while* slingan hittar den den kortaste sladden som kopplar ett upplyst notställ till ett icke-upplyst notställ. Efter detta sparas vilka notställ som kopplades och en räknare som räknat antalet grenar som behövs vid varje notställ inkrementeras. Slutligen så kombineras denna information till att ge ut en lista av sladdar som behövs. Indatan M är en ett-indexerad matris av avstånd mellan notsällen.

Rad 1-4 initialiserar variablerna L, C, V och P. L är antalet notställ som är samma som längden på ena dimensionen av M, C är en vektor av räknare av antalet uttag som behövs vid varje notställ som alla börjar på värdet 1. V är en lista av besökta notställ som börjar med tubaspelarens notställ. P är en lista med par av notställ som ska kopplas, första elementet it tupeln är där sladden ska kopplas från och andra elementet är vart den ska kopplas till, P initialiseras till att vara kopplingen från vägguttaget till tubaspelaren $(0, L)$.
Rad 5-21 beskriver uppbyggnaden av det minimala spännande trädet. En invariant av slingan kan beskrivas med ord som "V består av besökta notställ och P är sladdarna i det minimala spännande trädet av V"
Rad 7-15 i slingan består av två *for* slingor som för kollar alla stigar som går ut från de besökta notsällen till de ickebesökta och sparar den minsta sladden samt mellan vilka notställ den går mellan.
Rad 18-20 Använder informationen om den minsta sladden för att spara mellan vilka notställ den går samt räknar upp antalet grenar i varje sladd.
Rad 23-26 Med hjälp av vektorn C, listan P och matrisen M skapas listan av längden på sladden, antalet grenar och mellan vilka notställ som sladden bör dras i variabeln RES som också är utdata av algoritmen.

Tidskomplexitet

Låt n är antalet notställ. Slingan på rad 5 kommer upprepas n gånger då den efter efter varje genomgång kommer lägga till ytligare ett notställ i V och minska skillnaden mellan $V.lenght$ och L. *for* slingan på rad 7 kommer upprepas upp till n gånger eftersom det är max-längden på V. Rad 8 kommer också upprepas n gånger eftersom $L = n$. Slutligen kommer rad 9 också ta n steg eftersom hela listan V måste kollas för att utesluta att t inte är i V. Raderna 10,11,12 och 18,19,20 kan antas ske på konstant tid. Slingan på rad 24 kommer inte heller bidra till komplexiteten eftersom att gå igenom listan P är kort jämfört med operationerna i *while* slingan. Sammanfattning, den totala tidskomplexiteten är är produkten av 4 slingor som körs n gånger som ger en tidskomplexitet på $\mathcal{O}(n^4)$.

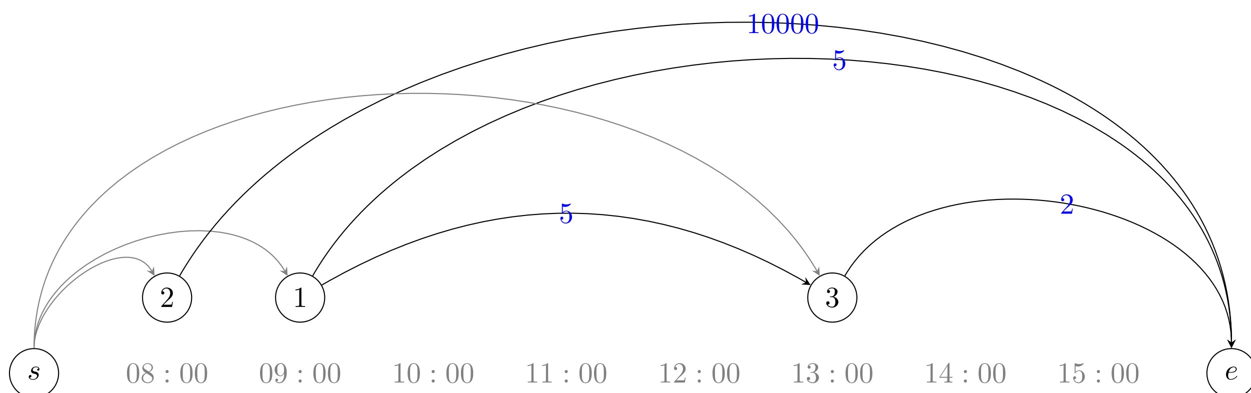
Korrekthet

För att motivera korrektheten bör man bevisa att sladdarna mellan notställen utgör ett minimalt spännande träd över alla notställ, och att min algoritm hittar detta träd i listan av kanter P. Man behöver också bevisa att min algoritm sedan riktar alla sladdar i rätt riktning från notställ till notställ och att den räknar antalet grenar på rätt sätt.

2 Uppgift 2 Viktiga möten

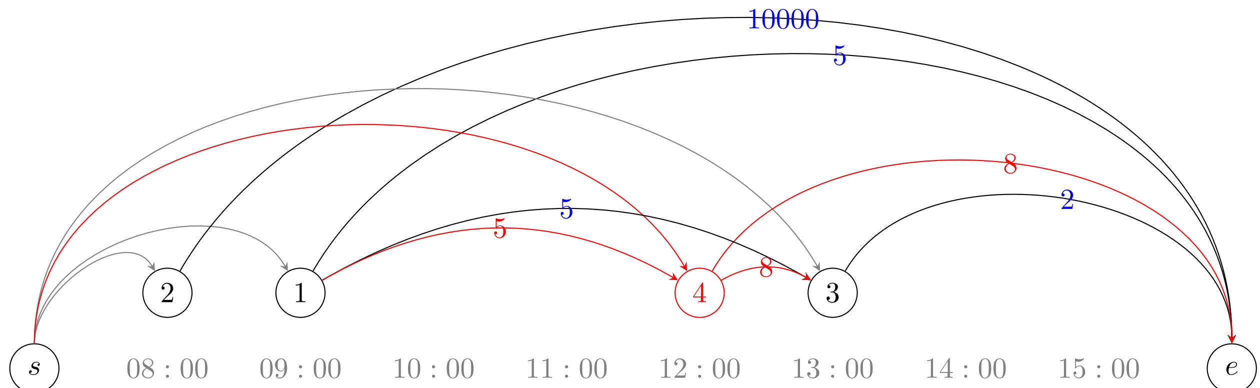
Problem Modellering

Problemet med att boka möten med olika prioriteter kan beskrivas som ett riktakt grafproblem där möten representeras av hörn och kostnad av kanterna mellan mötena motsvarar prioriteter. Det är möjligt att göra denna modellering eftersom att tiden bara går i en riktning vilket betyder att kanterna mellan möten måste vara riktade från ett tidigare möte till ett senare möte. Om man sedan lägger till ett övre hörn som har en kant till varje möte samt en undre kant som är riktad till från varje möte kan problemet slutligen ses som att problem att hitta stigen med den högsta summan prioriteter som börjar i det första hörnet och sluter i det sista hörnet. I uppgiftsbeskrivning står det också att mötena är sorterade på sluttid men algoritmen skapad i denna uppgift har ordningen på möteslistan ingen betydelse och kommer inte att användas.



Grafen ovan är det givna exemplet från uppgiftsbeskrivningen, tidsslagen är bara med för tydlighet och inte en del av grafen och kanter utan kostnad har kostnad 0. Denna graf kan skapas genom att gå igenom listan av möten och skapa ett hörn som man placerar ovanför mötets start tid, varje hörns värde motsvarar löpnumret på mötet. För varje möte m_i skapar man en kant till varje annat möte m_j vars starttid är större eller lika med $m_i s$ sluttid, kantvikten sätts till mötets prioritet. Därefter lägger man till två nya hörn, s och e och för varje möte i möteslistan ritar en kant från s mötet med vikt 0 och en kant från mötet till e med vilket motvarande mötets prioritet. Man kan notera att i grafen saknas informationen om sluttiden för varje möte, men för problemets lösning är sluttiden för ett möte oviktigt och det enda viktiga är att veta vilka möte som går att schemaläggas efter ett möte.

Om man till exempel skulle göra en ny graf med exemplet ovan med ett fjärde möte mellan klockan $12:00$ och $13:00$ med prioritet 8 skulle det följande skilja. Ett nytt hörn med värde 4 skulle skapas ovanför tiden $12:00$. En ny kant skulle gå mellan s och det nya mötet och en annan från det nya mötet till e. För alla möten som slutar innan klockan $12:00$ ritas en ny kant från det mötet till det nya mötet, i detta fall kanten $(1, 4)$. För alla möten som börjar efter eller samtidigt som sluttiden på det nya mötet d.v.s. kl $13:00$ dras en kant från från det nya mötet till början av nästa mötet i detta fall $(4, 3)$. Ifall det är mer än ett möte som börjar vid samma tid gör man precis på samma sätt men man sätter mer än ett hörn ovanför klockslaget, det är inga problem eftersom tiderna är bara där för visualisering och alla topologiskt ekvivalenta grafer tjänar samma syfte.



Grafen ovan är det givna exemplet från uppgiftsbeskrivningen med ett fjärde möte mellan klockan 12 : 00 och 13 : 00. Från denna modellering kan man dra följande slutsatser att om det finns minst ett möte gå gäller.

- Alla kanter riktade i kronologisk ordning och det är inte möjligt att gå tillbaka i tiden.
- Från varje möte är det möjligt att gå direkt via en kant till varje annat möte som man skulle kunna schemalägga efter det nuvarande mötet.
- Antalet hörn i grafen är proportionell mot antalet möten.
- Från varje möte har minst en kant nämligen till e.
- Till varje möte finns minst en kant nämligen från s.
- Grafen kommer alltid vara sammanhängande.
- Summan av kanternas kostnader för en stig från s till e är den totala prioriteten av alla möten som man passerar.
- Den dyraste stigen följer hörnen som representerar schemaläggningen av möten som ger högst prioritet.

Med denna problem formulering gäller det alltså att bygga grafen från en lista av möten samt att hitta den dyraste stigen från e till s för att hitta den bästa kombination av möten som get högst total prioritet.

Rekursionsekvation för den optimala totala prioriteten

Rekursivt kan det implementeras att man är vid ett godtyckligt hörn i grafen, man kollar om man är div slutet e, om inte så går man längst alla kanter och adderar varje kants värde till en summa och upprepar rekursivt stegen och returnerar summan prioriteten för den kanten som gav det högsta totala prioriteten. Denna Rekursionsekvation ska köras på hörnet s för att få totala prioriteten av den bästa schemaläggningen. Denna algoritm tar ett starthörn i och en grannlista M som argument där ickeexisterade kanter representeras med -1.

Algorithm 2 Rekursionsekvation

```
    Input: i, M
    Output: RES
 1: if i = e then
 2:     RES ← 0
 3:     return
 4: end if
 5:
 6: MAX ← 0
 7: for j ∈ [1, M.length] do
 8:     if M[i][j] ≠ −1 then
 9:         PATH ← M[i][j]+ Rekursionsekvation(j, M)
10:         if PATH ≥ MAX then
11:             MAX ← PATH
12:         end if
13:     end if
14: end for
15: RES ← MAX
```

Med den givna algoritmen skulle man anropa $Rekursionsekvation(s, M)$ där s kan antas vara 0 och e är antalet möten plus 1 och M är kantmatrisen för den givna mötesgrafen. Rad 7 kommer uppfyllas för minst ett möte eftersom alla möten har en kant till e. Det finns ingen risk att rad 8 skapar en oändlig rekursionsloop eftersom det inte finns några cykler i grafen. Algoritmen är korrekt eftersom den gör en totalsökning av alla stigar från s till e och väljer den dyraste, alltså den med högsta totala prioritet.

Polynomisk Lösning

Skapa Mötesgraf

För att hitta den bästa stigen i grafen bör man först skapa grafen från en lista med möten. Man kan antas att ett möte är ett objekt med 5 attribut. Löpnummer, beskrivning, starttid, sluttid och prioritet. I koden kommer jag hänvisa till n, *name*, *start*, *end* och *priority* för respektive attribut. För att hitta *billigaste* stigen i en graf mellan två punkter kan man enkelt använda sig av *Dijkstra's algoritm* som beskrivs på sida 137 i kursboken. I vårat grafproblem ska dock den dyraste stigen hittas. Dock kan man reducera problemet genom att hitta den dyraste kanten i grafen och spara den i konstanten d. Därefter sätter man all kanter e (inklusive 0 kanterna) till ett nytt värde $e = 1 + d - e$. Nu har grafen blivit en omvänd graf där de möten med högst prioritet har lägst kostnad och tvärt om. Eftersom d var den dyraste kanten så kommer ändå alla kanter i grafen fortfarande vara positiva heltal så Dijkstra's algoritm kan nu hitta stigen med den högsta totala prioriteten genom att välja kanterna med den lägsta totala kostnaden. Algoritmen kommer ta in en lista av möten och returnera en kantmatris som kan sökas med Dijkstra's algoritm. Från Dijkstra's algoritm kommer en stig räknas ut som används för att summera den totala prioriteten för den bästa schemaläggningen.

Algorithm 3 Skapa Mötesgraf

Input: M
Output: RES

1: $SIZE \leftarrow M.length + 2$
2: $RES \leftarrow [[+\infty : SIZE] : SIZE]$ // fyll en $SIZE \times SIZE$ matris med oändlighet.
3: $MAX \leftarrow$ 1+max($m.priority$ for $m \in M$)
4:
5: **for** $(n_1, _, _, end_1, priority) \in M$ **do**
6: $RES[0][n_1] \leftarrow MAX$ // Skapa kant från s
7: $RES[n_1][SIZE - 1] \leftarrow MAX - priority$ // Skapa kant till e
8: **for** $(n_2, _, start_2, _, _) \in M$ **do**
9: **if** $start_2 \geq end_1$ **then** // Hitta möten som börjar efter n_2 slutar
10: $RES[n_1][n_2] \leftarrow MAX - priority$ // Skapa kant mellan möjliga möten
11: **end if**
12: **end for**
13: **end for**

Om man kör algoritmen för möteslistan i det tidigare exemplet skulle RES vara lika med den följande kantmatrisen. Ett antagande man gör är att M inte är tomma listan. Om man körde denna algoritm på exempelgrafen skulle man få denna kantmatris.

	s	1	2	3	4	e
s 0	∞	10001	10001	10001	10001	∞
Budgetuppföljning 1	∞	∞	∞	9996	9996	9996
Börsintroduktion 2	∞	∞	∞	∞	∞	1
Semesterersättning 3	∞	∞	∞	∞	∞	9999
Nytt Möte 4	∞	∞	∞	9993	∞	9993
e 5	∞	∞	∞	∞	∞	∞

Skapa Mötesgraf: Tidskomplexitet

För en möteslista M av längd n kommer initialisering av RES ta $(n + 1)^2$ steg att utföra. Att hitta max prioriteten i möteslistan kan göras på linjär tid. Bägge *for* slingorna kommer köras n gånger och tilldelningen och jämförelsen på rad 6, 7, 9, 10 kan alla ske på konstant tid. Sammanlagt kommer algoritmen ha en tidskomplexitet på $\mathcal{O}(n^2)$.

Skapa Mötesgraf: Korrekthet

För att denna algoritm ska vara korrekt så ska en grannmatris skapas som följer beskrivningen på mötesgrafen och motsvarar möteslistan M. Denna kantmatris kan ges till Dijkstra's algoritm för att hitta den billigaste stigen från s till e, i exemplet är det $(0, 2), (2, 3)$.

Söka Mötesgrafen

Algorithm 4 Dijkstra's Algoritm

Input: M
Output: $PATH$

```
1: SIZE ← M.length, s ← 0, e ← M.length − 1
2: PATH ← [0 : SIZE] // Skapa stig vektor med ∞ av längd SIZE
3: D ← [∞ : SIZE] // Skapa distans vektor med ∞ av längd SIZE
4: D[s] ← 0 // Sätt s till att ha 0 distans
5: V ← {s, 1, 2 . . . e} // sätt V som listan av alla hörn
6:
7: while V.length > 0 do
8:    min ← ∞
9:    for i ∈ V do
10:      if D[i] < min then
11:         min ← D[i]
12:         x ← i
13:      end if
14:   end for
15:   // x är nu det billigaste ickebesökta hörnet som ligger granna med ett besökt hörn
16:   V ← V \ x // ta bort x från V
17:
18:   for y ∈ V do // uppdatera D vektorn efter att x är besökt
19:      if D[y] > D[x] + M[x][y] then
20:         D[y] ← D[x] + M[x][y]
21:         PATH[y] ← x // Spara att billigaste stigen till y är från x
22:      end if
23:   end for
24: end while
```

Söka Mötesgrafen: Tidskomplexitet

För en kantmatris med sidolängd n kommer *while* slingan att köras n varv eftersom V börjar med att innehålla n element och på rad 16 tas ett element ut ur V. Bägge *for* slingorna itererar också en gång för varje element i V. Resten av instruktionen i *while* slingan kan köras på konstant tid vilket ger en total tidskomplexitet på $\mathcal{O}(n^2)$ vilket stämmer överens med teoretiska värden för Dijkstra's algoritm.

Söka Mötesgrafen: Korrekthet

Innan algoritmen börjar gäller det att M är en kantmatris och efter alla variabler har skapas så kan en slinginvariant SI uttryckas som $D[x]$ är minsta totala kostnaden från s till x och att $PATH[x]$ är hörnet innan x i den optimala stigen för alla hörn $x \notin V$. Innan slingan börjar så är finns det inga hörn som inte ligger i V alltså gäller SI innan slingan börjar. Rad 9-14 hittar indexet x på det minsta värdet i D för alla index som ligger i V. Ett hörn sådant hörn kommer alltid hittas eftersom grafen är sammanhängande. Detta index tas bort från V vilket innebär att $D[x]$ måste vara den mista kostnaden för att ta sig till hörn x samt att $PATH[x]$ måste vara hörnet man kom ifrån för att ta sig till x. Detta krav är uppfylls i *for* slingan på rad 18-23 då alla distanser uppdateras ifall den är närmare från det nya hörnet x. Det gäller att för en graf med en mängd besökta hörn, den närmaste ickebesökta hörnet måste nödvändigtvis vara det som har den kortaste kanten till ett besökt hörn, givet att alla hörn har positiva kanter. Alltså gäller SI efter varje varv i slingan. När *while* är färdig är listan V tom vilket innebär att alla hörn i $PATH$ visar vilket det tidigare hörnet är för den optimala stigen enligt SI.

Bäst schemaläggningen

Den tidigare algoritmen kan kombineras med Dijkstra's algoritm för att räkna ut den totala prioriteten. Dijkstra's algoritm kan räkna ut hela stigen som get den minsta totala kostnaden. När man räknat ut alla kanterna i den kortaste stigen kan man summera deras totala prioritet genom att kolla upp den i den originella möteslistan. Man måste vara försiktig att inte lägga till den första kanten i stigen eftersom den går från s till det första mötet och egentligen inte har en prioritet.

Algorithm 5 Bäst schemaläggningen

Input: M
Output: SUM

1: $s \leftarrow 0,\ e \leftarrow M.length + 1$
2: $RES \leftarrow$ Skapa Mötesgraf (M) // Skapa kantmatris
3: $PATH \leftarrow$ Dijkstra's algoritm (RES) // Utför Dijkstra's algoritm på RES från s till e
4:
5: $SUM \leftarrow 0$
6: $i \leftarrow PATH[e]$
7: **while** $i \neq s$ **do** // Följ $PATH$ baklänges från e till s
8: $\quad SUM \leftarrow SUM + M[i].priority$ // Addera prioriteten till SUM
9: $\quad i \leftarrow PATH[i]$
10: **end while**

Algoritmen gör att antagande att listan M är 1 indexerad eftersom det första mötet har löpnummer 1. Men om så inte är fallet kan man enkelt på rad 8 lägga $i - 1$.

Bäst schemaläggningen: Tidskomplexitet

Algoritmen består löst av 3 olika steg, första är att skapa kantmatrisen, andra är Dijkstra's algoritm, och tredje är att räkna ut summan av prioriteter av den bästa stigen. För alla beräkning antas det att möteslistan består av n stycken möten.

Skapa Mötesgraf

Att skapa matrisen RES på rad 2 har redan en tidskomplexitet på $\mathcal{O}(n^2)$ på grund av dess storlek. De andra tilldragningarna går snabbare än n^2. De 2 nästlade for slingorna kommer båda upprepas n gånger och får därför en tidskomplexitet på $\mathcal{O}(n^2)$. alla tilldelningar samt jämförelsen i slingorna kan antas ske på konstant tid.

Tidskomplexitet: Dijkstra's algoritm

Analysen denna implementation av Dijkstra's algoritm kom fram till att tidskomplexiteten är $\mathcal{O}(n^2)$

Tidskomplexitet: Räkna Totala Prioriteten

Räkna den totala algoritmen är summan av att skapa mötesgraf och Dijkstra's algoritm samt att summera kostnaderna i den bästa stigen. $while$ slingan kan högst ta $n + 1$ varv eftersom det är den längsta möjliga stigen i grafen (stigen som besöker alla möten samt s och e). Att uppdatera summan och att göre jämförelsen sker på konstant tid, men att kolla $M[x].priority$ sker på linjär tid eftersom att M är en lista och hitta möte x måste linjärsöka genom listan. Alltså är tidskomplexiteten på denna algoritm också $\mathcal{O}(n^2)$ som då också är den totala tidskomplexitet på hela algoritmen.

Bäst schemaläggningen: Korrekthet

Korrektheten för de individuella delarna har redan beskrivits. Men för rad 5-10 gäller det att bevisa att om $PATH$ är resultatet av Dijkstra's algoritm att den kan användas för att summera den högsta totala prioriteten för den den bästa schemaläggningen. Om man börjar på rad 5 gäller det att $PATH$ är en vektor med ett värde för varje hörn i grafen och för varje hörn så representerar värdet i grafen vilket annat hörn som kommer innan i den optimala stigen från s till e. Efter rad 5 och 6 gäller det att $SUM = 0$ och $i = e$. Med hjälp av detta kan man skapa slinginvarianten SI som är att SUM är den totala prioriteten för det bästa schemat av möten som sker efter möte med löpnummer i. Innan slingan gäller detta eftersom $SUM = 0$ och $i = PATH[e]$ vilket är det sista mötet. I slingan adderas prioriteten av mötet i med SUM och i sätts till $PATH[i]$ vilket är mötet före i. Detta betyder alltså att man har hoppat tillbaka i tiden med ett möte och att SUM har ökat med mötets prioritet, alltså gäller SI fortfarande. Till slut kommer i vara lika med s och då hoppar man ur slingan. Eftersom s är innan det första mötet så kommer SUM vara summan prioriteterna av alla möten i det bästa schemat, alltså är algoritmen korrekt.

Hitta Bästa Schemat

Algoritmen kan lätt anpassas att räkna ut det bästa schemat istället för totala prioriteten med en liten förändring i sista delen av algoritmen. (rad 5-10)

Algorithm 6 Hitta Bästa Schemat

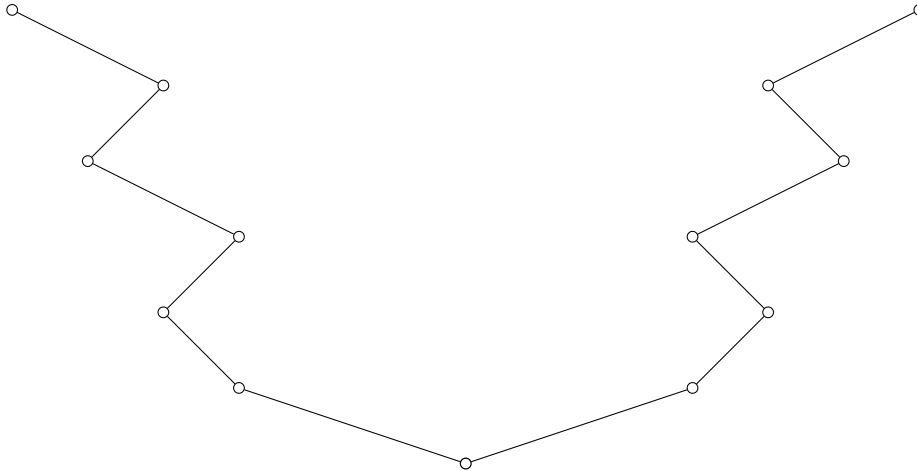
Input: M
Output: RES

1: **while** $i \neq s$ **do** // Följ $PATH$ baklänges från e till s
2:   **print** $M[i].n$+" "+$M[i].name$
3:   $i \leftarrow PATH[i]$
4: **end while**

Denna förändring kan utföras på den tiden det tar att kolla upp namnen i möteslistan vilket kan ske på linjär tid. Linjär tid är snabbar än algoritmen i sin helhet alltså har denna algoritm samma komplexitet som tidigare, d.v.s. $\mathcal{O}(n^2)$

3 Uppgift 3 Optimal Analys av Ringar i Vas

Problem Modellering



Givet är en vas som består av stycken segment från en mängd av m punkter (x_i, y_i) spelade runt origo. Var för sig släpps n antal ringar med radier $c_1, c_2, ...c_n$ i vasen. Problemet är att hitta hur långt ned i vasen varje ring av radie c_i skulle falla om den var släppt ned i vasen mätt från vasens bott och returnera dessa höjder i stigande ordning. Eftersom vasen är symmetrisk runt origo kommer bara den högra delen av vasen betraktas, vid användning av ett segment kan det antas att vara är segment med ickenegative x-värden som betraktas, detta gäller för hela uppgiften. Ett anat antagande är att bara ringar som kan ramla inuti vasen är betraktade. Jag gör detta antagande eftersom eftersom instruktionerna säger att ringarna landar *i vasen* och inte på.

Den Triviala Vasen

I fallet då vasen består av ett enda segment skulle vasen ta formen av en rund kon. Detta segment skulle ha bestå av en punkt (x, y) samt origo $(0, 0)$. Varje ring som släpptes is vasen skulle landa på en höjd som motsvarar segments y-värde för x-värdet som är lika med ringens radie r. Detta värde kan räknas ut genom att skapa en linje som går igenom bägge punkterna i segmentet och räkna ut linjens y-värde när x är lika med r. Denna teknik kan göra mera generaliserad för ett godtyckligt segment som inte nödvändigtvis innehåller origo, så länge ringens radie sammanfaller mellan de två punkternas x-värden.

Algorithm 7 Ring i vas med ett segment

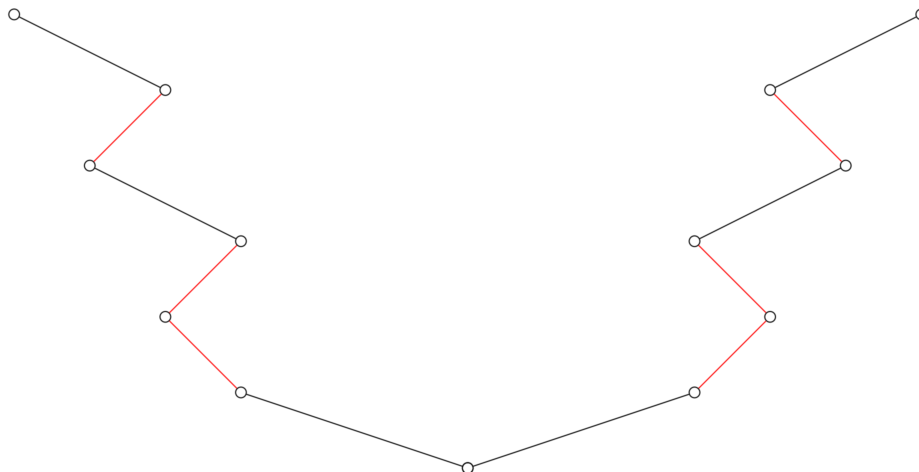
Input: r, $(x_1, y_1), (x_2, y_2)$
Output: h

1: // beskriv segmentet som en linje i form $y = m \times x + b$
2: $m \leftarrow \frac{y_2 - y_1}{x_2 - x_1}$ // räkna ut linjelutning
3: $b \leftarrow y_1 - m \times x_1$ // räkna ut y-skärning
4: $h \leftarrow m \times r + b$ // räkna ut höjden där ringen landar ringen

Körtiden av denna algoritm är konstant eftersom division, addition och multiplikation med rimligt stora tal sker på konstant tid. I ordo-notation skulle det beskrivas som $\mathcal{O}(1)$. Ett viktigt antagande som görs är att radien på varje ring r_i i C är inom intervallet av segmentets X-koordinater. I annat fall så skulle segmentet inte kunna fånga ringen. Matematiskt uttryckt så gäller $(x_1 < r_i < x_2) \vee (x_2 < r_i < x_1) \forall r_i \in C$. Detta gäller speciellt för segment som inte innehåller origo, detta är viktigare i mera generella fall. För att bygga en vas av flera segment kan denna princip utnyttjas.

Den Godtyckliga Vasen

Eftersom vasen kan bestå utav mer än ett segment så måste dessa fall behandlas. Som tur är så antas ringarna att falla rakt ned i vasen och inte kan falla på utsidan av den övre öppningen av vasen. Det finns också ickerelevanta segment som uppstår när vasen är konkav. Man kan tänka att de segment som inte vore synliga om man kollade rakt ned i vasen. En ring kan inte landa på delar av ett segment som ligger under ett annat segment för att det övre segmentet hade redan fångat ringen. Med hjälp av detta behöver man bara betrakta det översta segmentet för varje x-värde som är mindre än vasens öppning Segmenten som uppfyller dessa kriterier, dessa segment kommer kallas för *relevanta* segment. Man kan dela in alla relevanta segment till att vara olika *hinkar* som kan betraktas som individuella ensegmentiga vaser för alla ringar vars storlek faller inom segmentets x-värden.



I exemplet ovan visas alla ickerelevanta segment med röd färg eftersom att för alla punkter (x_1, y_1) längs segmentet ligger det minst en punkt (x_2, y_2) där $x_1 = x_2$ men $y_1 < y_2$. Att hitta dessa segment kan man göra med an algoritm som jämför x-värdena på alla segment och spara de relevanta segmentet i ett balanserat sökträd för att snabbt hitta senare, till exempel ett röd-svart träd som använder x-värdet som nyckel i trädet.

Algorithm 8 Skapa segment träd

Input: P
Output: SEG

1: $SEG \leftarrow$ Empty R-B Tree // Skapa tomt röd-svart träd
2: $x_{min} \leftarrow P[0].x$ // sätt x_{min} till det första x-värdet i P
3: **for** $p \in P$ **do** // för varje punkt i P
4: **if** $p.x < x_{min}$ **then** // kolla om segmentet innehåller av nya x-värden
5: $x_{min} \leftarrow p.x$
6: $SEG.insert((p, p_{prev}), key = p.x)$ // infoga de två punkterna som bildar segmentet i SEG
7: **end if**
8: $p_{prev} \leftarrow p$
9: **end for**

Denna algoritm tar en lista av vasens punkter som argument och ger tillbaka ett röd-svart binärt träd med de relevanta segmenten i vasen där det minsta x-värdet av de två punkterna i segmentet är nyckeln i trädet. Det är viktigt att notera att P är sorterad efter fallande y-värden och att det antas att origo är med i P. Om så inte är fallet måste man ge listan baklänges samt lägga till origo i listan ifall nödvändigt. Notera att på rad 4 att uttrycket $p.x < x_{min}$ kommer aldrig vara sant för det första elementet eftersom det första x_{min} sätts till att vara lika med x-värdet av den första punkten i P.

Den Godtyckliga Vasen: Tidskomplexitet

För en given lista P med n antal punkter. Tilldelningen rad 1-2 kan ske på konstant tid. *for* slingan på rad 3 kommer köras n gånger. Jämförelsen samt de två tilldelningarna i slingan kommer ske på konstant tid, men dock så kommer det enligt litteraturen i värsta fall att ta $\mathcal{O}(log(n))$ att infoga ett element i det rödsvarta trädet. Eftersom denna infogning i värsta fall sker en gång för varje element i listan så är den totala tidskomplexiteten $\mathcal{O}(n \times log(n))$.

Den Godtyckliga Vasen: Korrekthet

För att denna algoritm ska vara optimal krävs det att det röd-svarta trädet som ges ut ska innehålla alla relevanta segment från vasen representerat som de två ändpunkterna på segmentet. Dessa segment ska vara indexerade med x-värdet på av punkten som är till mest till vänster (har lägsta x-värdet). Innan algoritmen startar gäller det att P är en lista av ickenegativa punkter som beskriver vasen likt uppgiftsbeskrivningen, P måste vara sorterade på fallande y-värden och att P innehåller origo (eftersom P är sorterad och inte innehåller negativa tal måste origo ligga sist i listan). Efter rad två har SEG och x_{min} initialiserats, en slinginvariant S vid detta tillfälle är att alla relevanta segment som ligger helt till höger om x_{min} är tillagda i SEG och att inget ickerelevant segment är tillagda i SEG. Detta är sant eftersom alla segment som ligger helt utanför kantens öppning är ickerelevanta. Som en väluppfostrad slinginvariant ska göra så gäller S på första raden i *for* slingan, om *if* uttrycket inte är sant så betyder det att punkten p har ett x-värde som är större än x_{min}, vilket betyder att segmentet slutar till höger om ett relevant segment. Detta betyder att det nya segmentet antingen ligger utanför vasens kant eller ligger helt under ett annat segment och är med andra ord ett ickerelevant segment och bör inte läggas till i SEG. I motsatt fall när *if* uttrycket är sant kommer exekvering att sätta det nya x_{min} till att vara ps x-värde och segmentet kommer infogas i SEG. Eftersom segmentet slutade i ett x-värde som var mindre än det förra x_{min} betyder det att det inte finns ett segment ovanför segmentet från p_{prev} och p och att p ligger till vänster om vasens övre kant, detta betyder att segmentet är ett relevant segment och bör därför läggas till i SEG.

Efter detta sätts p_{prev} till att vara p eftersom att slutet av ett segment används som starten av nästa segment, men detta har inte en påverkan på slinginvarianten så efter iterationen i slingan är S fortfarande sant. Slutligen kommer S gälla efter *for* slingan, detta gäller eftersom att det sista elementet i P är $(0,0)$ vilket innebär att x_{min} måste ha sats till 0, vilket innebär att alla segment i P ligger helt till höger om x_{min}. Med andra ord gäller det att SEG innehåller exakt alla relevanta segment från P när algoritmen terminerar.

Algorithm 9 Trädsökning

Input: SEG, r
Output: (p_1, p_2)

1: **if** $SEG = null$ **then** // basfall
2: $(p_1, p_2) \leftarrow (null, null)$
3:
4: **else if** $SEG.key < r$ **then**
5: $(p_1, p_2) =$ trädsökning$(SEG.right, r)$
6: **if** $p_1 = null \wedge p_2 = null$ **then**
7: $(p_1, p_2) \leftarrow SEG.data$
8: **end if**
9: **else if** $SEG.key >= r$ **then**
10: $(p_1, p_2) \leftarrow$ trädsökning$(SEG.left, r)$
11: **end if**

Även fast algoritmer för sökning i ett binärträd finns så kommer min implementation kräva en liten förändring, men denna förändring bör inte ett sätt som kan motivera en ny analys av tidskomplexitet och korrekthet. När vi vill se vilket intervall en given ring radie r faller inom måste vi hitta det största x-värdet i SEG som är mindre än r. Detta gör vi genom att göra vanlig binäring med en lite förändring, nämligen på rad 6-7 kollar vi ifall trädsökningen i det högra trädet gav ett resultat, om inte så tar vi den nuvarande intervallen istället eftersom alla intervaller till höger i sökträdet är större än r. En annan förändring är att istället för att ha ett basfall när $SEG.key = r$ kollar vi det istället på rad 9 då vi vill hitta intervallen som innehåller r. Att tidskomplexiteten av detta sökande är samma som vanlig binärsökning går att informellt säga att den tillagda *if* satsen kommer exekveras en gång och kan ske på konstant tid, men andra ord, sök är $\mathcal{O}(log(n))$ där n antalet element i trädet. När det gäller korrekthet så kan man argumentera att för alla r som har ett intervall i SEG att den skulle hitta rätt intervall eftersom alla värden mellan origo och vasens övre kant är täckt av minst ett relevant intervall och efter varje rekursivt anrop av trädsökning kommer antalet intervaller minska, och ifall ett intervall inte hittas kommer som sagt det nuvarande intervallet returneras på rad 7.

Alla Ringar i en Vas

Nu när vi har en algoritm för att behandla en ring i en vas med ett segment samt en algoritm för att hitta detta segment för en godtycklig ring kan en algoritm som kan behandla alla ringar för en godtycklig vas.

Algorithm 10 Alla ringar

Input: P, R
Output: RES

1: $P \leftarrow \{(0,0)\} + P$ // lägg origo i P
2: $P \leftarrow P.reverse()$ // sätt P att vara baklänges
3: $SEG \leftarrow$ Skapa segment träd (P)
4:
5: $RES \leftarrow \emptyset$
6: **for** $r \in R$ **do**
7: $(p_1, p_2) \leftarrow$ trädsökning(SEG, r)
8: $h \leftarrow$ Ring i vas med ett segment(r, p_1, p_2)
9: $RES \leftarrow h + RES$
10: **end for**
11: $RES \leftarrow$ mergesort(RES)

Algoritmen ovan får en lista av punkter P som beskriver vasen samt en lista med ringradier R alla är tillräckligt små för att få plats i vasen. P antas kommer i sorterad ordning efter stigande y-värden. Det första som göra är att lägga till origo i början av P samt att ändra ordningen på P eftersom det behövs när man skapar segment trädet. Ett rödsvart träd av alla segment skapas med hjälp av P och sparas i SEG. analysen för röd-svarta träd finns i boken ”Introduction to Algorithms” skriven av Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest och Clifford Stein på sida 308. Sedan utförs en trädsökning för varje ring för att hitta det rätta segmentet som ringen passar i. Detta segment används för att hitta höjden där ringen skulle landa med hjälp av den generaliserade ring i vas med ett segment algoritmen. Resultatet är sparad i en lista som slutligen sorteras för att ge tillbaka värdera i den önskade ordningen. Sortering som används är mergesort som finns i boken ”Introduction to Algorithms” skriven av Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest och Clifford Stein på sida 210.

Alla Ringar i en Vas: Tidskomplexitet

För en vas som består av n antal punkter och m stycken ringar kan men beräkna tidskomplexiteten. Att lägga till origo i P kan ske på konstant tid eftersom det infogas i början av listan. Att vända ordningen på en lista kan göras på linjär tid. Att skapa segmentträdet är etablerat att ha en komplexitet på $\mathcal{O}(n \times log(n))$. *for* slingan på rad 6 kommer upprepas m gånger, en för varje ring. I slingan anropas trädsökning vilket har en komplexitet på $\mathcal{O}(log(n))$ och segmentberäkningen som sker på konstant tid, som också tilldelningen på rad 9. Slutligen så görs mergesort på RES som har en tidskomplexitet på $\mathcal{O}(n \times log(n))$. Alltså är den totala komplexiteten $n \times log(n) + m \times log(n) + m \times log(m) = \mathcal{O}((n + m) \times log(n) + m \times log(m))$.

Alla Ringar i en Vas: Optimal

Om vi antar att segmentet består av de två punkterna $(0,0)$ och $(10,10)$ enheten är oviktigt så länge vi antar att varje rings radie c_i följer $0 < c_i < 10$. I detta fall skulle varje ring landa på den höjd där ringens radie är lika med segmentets y-värde, och i detta fall är det när $y = c_i$ eftersom segmentet som består av de två triviala punkterna sammanfaller med linjen $y = x$ för alla värden $x \in [0,10]$. Eftersom höjderna motsvarar ringarnas radie och att dessa radier ska ges tillbaka i stigande ordning så kan detta problem reduceras till sorteringsproblemet. Detta ger oss $\mathcal{O}(nlog(n))$ som en undre gräns för komplexiteten på detta problem, eftersom det i annat fall skulle gå att sortera heltal snabbare än $\mathcal{O}(nlog(n))$. I ett mera allmänt fall så skulle ringar is vas problemet av en vas som består av ett enda godtyckligt segment inte kunna lösas på bättre än $\mathcal{O}(nlog(n))$ där n är antalet ringar som läggs i vasen.

Alla Ringar i en Vas: Korrekthet

korrektheten av trädskapandet samt sökningen har redan analyserats. Det är också etablerat innan att för varje segment att r kommer ligga mellan början på det första och slutet på det sista relevanta intervallet så det är garanterat att sökningen kommer hitta en passande intervall för alla $r \in R$. Eftersom att ringens radie ligger inuti segmentet så kommer rad 8 också att köras korrekt. När det gäller mergesort så kan analysen läsa om i den hänvisade boken.