O(n): Upper bound time complexity. Efficient: O(n²)
Ω(n): Lower bound time complexity
θ(n): Ger gränser på möjlig komplexitet
Example: f(n) = 123n, g(n) = 32n³ => f(n)∈O(n²). g(n)∈Ω(n²) De

---[1]

Matchning: Algorithm sida 124

Match:
- Ett set av par (x_i, y_i) s.t de element enbart finns i ett par

Perfect match:
- All x_i and y_i are matched

Stable:
- No two pairs are unstable

Unstable
- ((s_I, c_J), (s_K, c_L)) =
  - (s_I prefer c_L && c_L prefer s_I) || (s_K prefer c_J && c_J prefer s_K)

Gale shapley Algorithm explained:

1. Take a list p of students.
2. For each student, take the company they prefer most and havent applied to
3. add (s,c) if c has no student or prefers s over current (else add student to end of list)

Tip: Use **inverted list** (at index k is the students position in preference list) for companies preferences: Constant time to look up index i and index j and compare them

Time complexity:
- O(n²). n elever. n företag. En elev kan max söka på ett företag. Har n elever: O(n²)

Varför skapar perfekt, stabil matchning:
varför Perfekt lösning:
- Vet att om ett student ej har ett företag så finns det ett företag de ej sökt till (eftersom #student = #företag. Vet att företaget måste ta studenten om de ej är matchade. Dvs studenten har inte sökt alla företag)

varför Stabil lösning:
- Anta att vi har (s1, c1) och (s2, c2) som är ostabil efter terminering, pga s1 och c2 föredrar varandra
  - Case 1: s1 har sökt c2. Då hade s1 redan varit matchat med c2 - motsägelse
  - Case 2: s2 har inte sökt c2. Pga c2 prioriteras över c1, så hade han inte heller sökt c1.
    - Algoritmen kommer inte terminera då (se under varför) - motsägelse
  - Slutsats: antagande är felaktigt. Matchningen måste vara stabil

Förklaring varför det ej kommer terminera om student inte har ett företag:
Antag felaktigt att while-loopen terminerar med en student utan ett företag, för han har sökt till alla företag. Kommer alldrig ske pga punkt 1

---[2]

Länkad lista (single, double, circular): Sida 46
Hashmap:
loadfactor = #element/(total size)
Seperate chaining (öppen hastabell): Vid collision läggs elementet i en länkad lista på indexet
Open adressing (stängd hashtabell): Vid collision lägs elementet på nästa lediga plats:
- **Linear probing**: i = f(key) % n. if(!(a[i].empty)) then i++. Insert at i
- **Specialcase delete**: Kan inte bar sätta till null (förlorar kommande element
  1. Replace with "deleted"
  2. Move to left (flytta inte orelaterat element - beräkna nyckeln på elementet och kolla att den nyckeln stämmer överens med det vi flyttar)
     - Pseudokod sida 81

- **Quadratic probing (SIDA 84):** Vill undvika clustering

---

[1] Matching, galeshapley
[2] Länkad lista, hashmap, hashtabell

- h(k,i) = (h'(k) + i²) % m - tanken med i² (mot bara i) är att lämna en cluster snabbare
    - Om två olika nycklar har samma hashvarde så får vi cluster ändå
- Om det finns tom plats hittar vi den inte nödvändigtvist: Förutom om #tabell är primtal och load factor är mindre än ½ (sida 84)
- i är indexet som ökas vid kollision

- **Dubbel hashing.** Vill undvika andra hands clustering
    - h(k,i) = (h1(k)+ih2(k)
        - Risken att två nycklar har samma hashvärde är lägre
    - Garantera att alla postioner prövas (dvs kan nås av hashfunktionen: Sida 84

---

[3]

Graph representation:

- Note: Can use both at the same time

Adjancency matrix:

- Edge i to j is represented by m[i][j]=1
    - notera: dubbletter i oriktad graf. Förvara då bara hälften
    - For dense graph: Smaller options, and just as quick
    - Pros&cons

Adjacensy list:

- Edge i to j is represenated by i containing j (i contains its neighbors)

Path: Sequence of neighboaring nodes

Simple path: All nodes are distinct

Tree: (connected) undirected graph with no cycle

---

[4]

Depth first search (sida 91)

dfnum:

- Id på en nod

Ana (ancestor) & ättling (decendant):

- Om finns väg v->w, så är v ana, och w ättling

Strikt ana:

- om v är ana till w och v != w

Trädkant:

- Är en kant i sökträdet T. Den finns då: I varje dfs(w) - DVS undersök w - och vi har en kant v->w och w inte har besöks så finns en trädkant (v,w)

Framåtkant:

- kant (u,v) s.t v är strikt ättling till u i T men v är inte ett barn till u i T. Framåtkanten upptäcks när vi är i u och v har blivit besökt, men ät inte i sökstacken och u < v.

Korsande kant:

- kant (u,v) s-t u är inte en ättling till v och v är inte en ättling till u.

Algorithm. Tänk på att gå djupet först. DFS är depth first search

1. Gör dfs på startnod
2. Gå nästa nivå, gör dfs
3. Upprepra 2 tills vi når djupet.
4. Gå upp ett steg. Upprepra 3

Tidskomplexitet O(V+E)

- Givet konstant lookup på successors
- Pga

Breadth first seach (sidan 93)

1. Kolla på noder med avstånd 1 från start
2. Upperepa 1 för 2,3, …, n tills vi hittar slut eller vi har letat alla noder
    a. Den rekursion som hittar t först returnerar alla deras predecessors.

Tidskomplexitet

---

[3] Graph representatuin

[4] DFS, BFS

Mutably reachable: Path from u to v && path from v to u
Starkt sammankopplad: All nodes are mutably reachable
Starkt sammankopplad komponent: En delgraf som är stark sammankopplad
Starkt sammankopplat lemma:
s är en godtycklig nod i G. G är starkt sammankopplad ⇔ Varje nod är nåbar från s, s kan nå alla noder

- **Bevis =>** per definition av stark sammankopplad G
- <= för att vi vet att vägarna (u,....s,.....v) och (v,.......s,.......u) finns enligt lemma

Avgör stark sammankopplad med BFS:

1. Välj en godtycklig nod, s
2. Kolla att alla noder kan nås från s
3. Invertera G. Kolla att alla noder kan nås (ekvivalent med att alla kan nå s)

- Enligt lemma om 2 och 3 gäller ⇔ G är starkt sammankopplad

Tarjans algoritm: Hitta alla starkt sammankopplade komponenter: Tarjans

Low link: Indexet på roten i SCC
Algoritmen
Starta med en tom stack och en lista för att hålla reda på besöksordningen för varje nod i grafen.

1. För en godtycklig startnod i grafen, utför tarjans algoritm (2-4)
2. Tarjan(n): För noden, gör:
    a. Sätt visited till true
    b. tilldela ett nytt unikt index
    c. ge ett lowlink värde som är detsamma som indexet
    d. Lägg på stacken
3. För varje granne:
    a. not visited:
        i. utför tarjan(granne).
        ii. current.lowlink = min(current.lowlink, granne.lowlink)
    b. visited && granne har lägre index && granne är på stacken
        i. current.lowlink = min(current.lowlink, granne.lowlink)

4. Om lowlink == index
    - Poppa från stacken, tills vi når current
    - Dessa är alla en SCC

Notes:

- Stacken finns för att hålla koll på de aktiva noderna
    - Så enbart ett segment i stacken kan vara scc
- DFS säkerställer att noden i fråga kan nå den nod vars lowlink värde den tar
    - Då granne har ett lägre lowlink säger att den även kan nå current, för att den kan nå en nod som kan nå current

[5] SCC, starkt sammankopplad, tarjans
[6] greedy, interval scheduling, minimera förseningar

Greedy algorithm (sida 21):
- En algorithm som gör beslut utan all information

Bevisa att greedy är optimal
1. Greedy är alltid, åtminstoende, lika bra som en optimal (använder induktionsbevis)
2. Utbytesargument.
    - Se post-it

Utför 1 på interval scheduling (se nedan)
1. Antag vår lösning är request: r1, r2, .. r_n. Optimal är t1, t2 … t_m
2. Vill visa att n = m med induktionsbevis:
3. Basfall: f(r1) <= f(r2) - självklart, i och med att vi väljer första request
4. r_k > 1: Antag f(r_k-1) <= f(t_k-1). Vi vet kompatibelt: f(t_k-1) <= s(t_k)
5. Medför f(r_k-1) <= s(t_k) => dvs vi hinner alltid välja t_k
    a. Vi väljer den tidigaste tiden:
    b. Slutsats f(r_k) <= f(t_k)
6. Nu vill vi visa size(greedy)=size(optimal)
    a. Antag optimal.size > greedy.size & vet f(t_n) >= f(r_n)
    b. då finns t_n+1.
    c. iom f(t_n) >= f(r_n) så hade greedy hunnit schemalägga t_n+1 om optimal hinner
    d. DVS motsägelse, DVS greedy.size >= optimal.size

Interval scheduling: Hitta det största subsettet, så att alla element är compatibla

Compatible: Inga request överlappar

Algorithm:
1. Sortera listan baserat på klar först
2. Välj request som är klar först: Addera till res
3. Ta bort överlappande
4. Goto 1

Tidskomplexitet: O(n log n)
- Sortera: O(log n)
- Välj ut: maximalt O(n)

Minimera förseningar

Soft deadlines (d(r)):
- Kan failas, men föredras att inte - minimera totala förseningar

Output:
- Start s(r) och sluttiderna f(r) på requests

Input:
- tid(r), d(r). Ex r: t(r) = 2, d(r) = 3. Kan ex läggas s(r) = 1, f(r) = 3. Ger ingen försening

Algorithm:
1. Välj r_i med tidigast deadline: d(r_i)
2. Schemealägga så att början på r_i+1 är i slutet av r_i

[7]

Dijkstra: Hitta den kortaste vägen (s.104)

d(v) är kortaste vägen från start till v

Q: Alla noder

S: Alla noder vi kan den kortaste vägen till
1. Sätt d(s) till 0
2. Välj v som har kortast d(u)+w(e); u∈S, v∉S. e = (u,v)
    a. DVS gå via den edge som har minst distans till börjam, som gränsar till en nod vi kan vägen till
3. Gör predecessor(v) = u
4. d(v) = d(u)+w(e)
    a. Uppdaterar den vikt
5. Remove v from Q, add to S - tänk med cirklar
    a. Ser den som explored
6. Go to 1 while Q != null eller slutet är hittat
    a. Fortsätter tills vi har hittat v (eller explorat alla)

***algoritmen slut**

Tidskomplexitet O(m log n)
- Med antagandet att alla noder kan nås från s: #edges >= #nodes

---

[7] Dijkstra

- Samt antagandet att vi använder en heap för d(v)
- Skapa ny heap: O(n) eller O(n log n) beroende på metod
- Kolla granne + minska nyckel:
  - Varje nod måste kolla varje granne i Q och eventuellt minska grannen nyckel. Tiden för att minska en nyckel är O(log n) -> O(m log n)

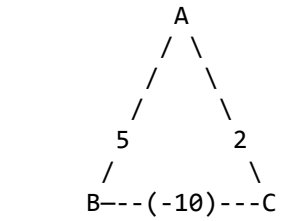
Why is dijkstra correct?

- Induktionsbevis

1. Basfalet då S=1. Dvs enbart s är i S. Då är d(s) är 0. Vilket vi vet stämmer
2. Inductive hypothesis: Assume theorem is true for $|S| \geq 1$.
3. Yttligare en nod v läggs nu till i S, då vi går från u (tillhör S) till v (tillhör ej S)
4. Antag motsäggelsefullt att det finns en kortare väg till v, via s … x y … v
   a. x i S, y inte i s
5. Detta kommer dock aldrig vara fallet, för då är distansen s->y vara kortare än s->v, dvs v hade inte valts. Så vi vet att kortaste vägen till v, är som vi har antagit, via u.

Dijkstra and negative edges

- Om man har en loop med negativ summa av vikter



- Kommer gå via 2. Säga att den är klar med C, trots att det är bättre att gå A-B-C
- Algoritmen kommer inte beräkna om distansen till noder den tror att den har hittat kortaste vägen till

---

[8]

Minimum spanning tree (MST)

Safe edge:

- En kant som kan läggas till i trädet utan att skapa en cyckel

Uppspänt träd:

- Anta att vi har en oriktad graf G
  - Ett uppspänt träd till G, är ett träd som innehåller alla noder från G, men inte nödvändigtvist alla kanter.

Minimalt uppspännande träd:

- Ett uppspännande träd där summan av kanternas vikter är minimal

Jarniks algorithm / prims algoritm:

Q: Utforskade noder

- Till en början alla noder, förutom start (en godtycklig nod)
- Representerar de noder som finns kvar att integrera in i trädet

T: Representerar vårt uppspänande träd (tom från början)

Algorithm:

1. Välj godtycklig nod v. Räknas som utforskad: upprepa sedan:

1. Välj en outforskad nod v, med minst vikt från utforskad nod u
2. Räkna v som utforskad
3. lägg till (u,v) till resultatträdet
4. gå till 1 medans Q inte är tom

Time complexity O(m log n) givet m >= n:

- O(n) iterationer av whilelooop
- O(log n) att ta bort minimala noden (tiden för delete min i en heaå
- Varje nod måste kolla varje av dess grannar
  - totalt m grannar (tillsammans för varje nod) i värsta fall.
  - Reduce key är O(log n) (reduce key för Q ändras - därmed ändras w(u,v) vilket vi sorterar listan utifrån - ger O(m log n)

Kruskals algoritm

1. Välj den kant med minimal vikt

---

[8] MST, Jarniks, prims, kruskals, union find

2. Om kanten inte skapar en cykel med T, lägg till den i T
3. ta bort kanten från möjliga conteneors för B
4. Gå till 1 medans vi har B != null

Detektera en cykel:

- Vi använder oss av union find
- om man övervägen lägga till kant (u,v)
    - gör find(u) och find(v)
    - om de är samma, så är det en cykel
    - Om de inte är samma är det inte en cykel.
        - Glöm inte göra union mellan de två mängdena

Union-find datastruktur

- Kan merge två mängden
- Skapa mängder med enbart ett element i varje mängd
- Kan hitta vilken mängd ett element är med i

Naive Union

- parrent(v) = u. Lägger in v och alla under v i datastrukturen u är i

Quicker union

- Håll koll på storleken på mängderna

1. Ta den minsta mängden, och lägg in den i den största mängden
2. Uppdatera storleken på den största mängden till summan av storleken av mängderna

- Detta är bättre, eftersom det undviker att man får höga träd. Då är det exempelvist snabbare att göra find nästa gång.

Find(Hitta vilken mängd ett element är en del av)

Naive find:

- Returnera v om v inte har en förälder
- Annars returna find av föräldern

Quicker find:

- Gör på samma sätt som naiva find, men under körningen, använd även path compression
    - DVS om v1 har föräldrer v2, v2 har föräldrer v3 osv upp till v_k
    - Sätt föräldrer till alla dessa till v_k, farfarfar….far. Höjden blir 2 på delträdet

Tidskomplexitet union-find:

- Då path compression och union by size:
- Union (by size) är konstant
- Givet m>=n find:
    - THETA(m*inv_ackerman(m,n)), vilket är mindre än fyra för alla rimliga värden på m och n
- Finns gånger då find blir linjär, men amorterad tid: essentially konstant

[9]

Rekursionsekvation

T(n): Runtime

example: T(n) = if(n == 1) return 0 else return 2T(n/2)+n

- Used to analyze runtime

Mergesort relation

Open form / implicit form:

- T(n) <= if(n == 1) return 0 else return T(roundup(n/2)) + T(rounddown(n/2))+n
    - So: Total T(n) är arbetet gjort på den nuvarande nivån + summan av arbetet gjort på nivåerna under

Closed form / explicit form: T(n) = nlog_2(n)

Att hitta den explicita formen:

1. Kolla hur det ser ut i de lägre nivåerna

---

[9] Mergesort, divide and conquere, master theorem

2. Gör en gissning
3. Bevisa mha induktionsbevis

- Vårt fall:
  - Base case: n = 1: T(1) = 1 log2 1 = 0 => stämmer
  - Antag T(n) = n log2 n
    - T(2n) = 2T(n) + 2n = 2n log2 n + 2n = 2n(log2 n + 1) =
    - 2n(log2 2n − 1 + 1) = 2n log 2n

The master theorem:

Nice way to find closed form tidskomplexitet for many recursive algorithms:

- T(1) = 1
- T(n) = aT(n/b)+n^s

Three solutions to master theorem:

1. T(n) = O(n²) if s > log_b(a)
2. T(n) = O(n^s log_2(n)) if s = log_b(a)
3. T(n) = O(n^(log_b(a)) if s<log_b(a)

log_b(a) = y ⇔ b^y = a

Convex hull (inte i boken)

- En delmängd av punkterna och kanter runtomkring dem, som omringar alla punkter

What is an leftturn

- Vi vill kolla om det är en vänstersväng då vi går från p0, över p1, till p2
- Gör vector från p0 till p1. Gör vector från p0 till p2
- **>0:** om kryssprodukten av p0p1 vector och p0p2 vektorn är positiv så är det en vänstersväng
- **=0:** samma linje

Jarvis march

1. Välj en punkt p_0 som helt säkert är en del av convex hull (ex leftmost)
2. Välj nästa punkt p_j
3. Om vi enbart gör vänstersvängar då vi går från p0, p_j till

Graham scan Algorithm

1. Ta punkten p0 med lägst y och gör den till origo
2. Sortera punkterna utefter deras vinkel till x-axeln (arctan)
   a. Specialfall, då två har samma vinkel. Vi vill inte ha den närmast origo, så lägg den först i listan, så att den senare kan pop:as av den längre bort från origo
3. Gå igenom de sorterade punkterna. För varje punkt, kontrollera om den gör en högersväng eller en vänstersväng jämfört med de två föregående punkterna i höljet. Om den gör en högersväng innebär det att den föregående punkten inte ska vara en del av det konvexa höljet och tas bort. Upprepa detta steg tills den aktuella punkten gör en vänstersväng.
4. Upprepra till vi har behandlat alla punkter

Dynamisk programmering:

- Uttryck en lösning utifrån lösningar av delproblem
- I jämfört med divide and conquer så finns överlappande problem
  - DVS memoization är viktigt för optimering

Weighted interval scheduling (sida 41)

---

[10] Convex hull, jarvis march, graham scan
[11] dynamisk programmering, weighted interval scheduling, string allignment

Input:
- Requests, with starts, finishes and weights

Output:
- Maximum the sum of the weights

Namn:
- f(r_1)=f(1)<=f(r_2)=f(2) <= ….
  - dvs f(1) har sluttid först

Algorithm weighted inverterval (utan memoization)

- Bilda en funktion rekursiv funktion OPT(n) som räknar hur den maximala värdet av alla requests från r0 till rn (genom att beräkna alla alternativ)

1. Välj det request, p(n) med tidigast sluttid som är kompatibel med nuvarande schemat
2. Med hjälp av OPT(n), kolla om p(n) borde vara en del av resultatet. DVS:
   a. Om OPT(n) = v(n) + OPT(p(n)) så läggs p(n) till i resultatet
   b. Om OPT(n) = OPT(n-1) så läggs p(n) inte till

Algortithm weighted inverterval (memoization)

- OPT(i) kommer aldrig att ändras. Finns bara en unik optimal lösning
- Kan göra OPT(i) bättre genom att göra så att den minns OPT(i)
  - DVS om OPT(i) inte har beräknats förut: Beräkna och skriv ner det
  - Om opt(i) har berknats förut, använd det värdet

Time complexity

String alignment

Problem: Beräkna hur lika två olika strängar är. Skillnaderna kvantifieras sedan. Detta motsvarar hur mycket det hade kostad att göra dem identiska

- Om man exempelvist en karaktär saknas, kan det kosta en viss summa
  - Motsvarar att sätta in ett wildcard i strängen
- Om man iställer behåller två mismatchade bokstäver kan det kosta en annan summa
  - Motsvarar att byta från en karaktär till en annan

Algoritmen String allign (med memoization)

1. Skapa en tvådimisionell matris M, där indexeringarna motsvarar lösningen på problemet på strängen från 0 .. i, 0 .. j
2. Populera kanterna på tabellen.
   a. DVS om en av strängarna är dom len(word1)*i och samma för ord två
3. Populera tabellen (i distinc ordning så vi använder memoization) Vi populerade därför grafen baklänges i labben med hjälp av:

```
for (int j = 1; j < words[1].length() + 1; j++) {
          for (int i = 1; i < words[0].length() + 1; i++) {
      var optimalCost = Math.max(
                        costOf(word1[i], word2[j]) + table[i - 1][j - 1],
                        //två bostäver

                        delta + table[i][j - 1]), //insert wildcard i word 1
                        delta + table[i - 1][j]); //insert wildcard i word 2
      table[i][j] = optimalCost;
         }
}
```

   a. Notera att vi har memoization, för vi använder våra tidigare lösningar från tablet
      i. DVS vi beräknar inte nytt i onödan
4. table[i][j] innehåller vår lösning

Sidenote: Nu kan vi även gå genom tabellen, se vilka insertade wildcards vi har gjort, genom gå igenom tabellen, välja nästa nod som lägst kostnad (up, höger, diagonalt) och bilda orden

[12]

[12] Bellman ford

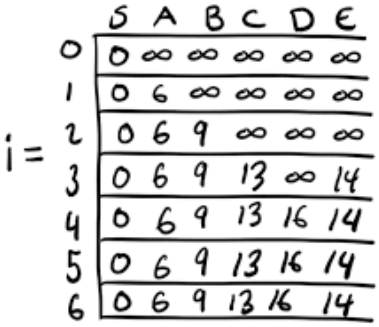
Bellman-ford: Shortest path.

- Kanter kan vara negativa & Minimal kostnad för att nå start till mål
- OPT(i,v) är en väg v->t som använder <= i kanter

Algorithm:

1. Tilldela alla noders avstånd från startnoden till oändligt, förutom startnoden själv som tilldelas avståndet 0.
2. Gå igenom varje kant i grafen och uppdatera avståndet till den angränsande noden om det finns en kortare väg genom den aktuella kanten.
3. Upprepa steg 2 för antalet noder minus 1 gånger.
    a. Detta är nödvändigt för att säkerställa att alla möjliga kortaste vägar har identifierats
4. Kontrollera om det finns några negativa cykler (dvs oändligt låg lösning) genom att utföra ytterligare en avslappning på kanterna. Om någon nod har en kortare väg genom en negativ cykel, kommer algoritmen att upptäcka detta och rapportera att det inte finns någon lösning.

- Implementeringsmässigt, grundar sig i dynamisk programmering
    - Vi räknar ut alla möjlig vägar
    - memoization är viktigt



(använd föregångarlistan för att hitta den kortaste vägen)

Tidskomplexitet: Antagandet att inte är en kompakt graf O(nm)

1. Init: O(n) - uppdaterar och intialiserar varje nod: O(n)
2. Relaxation - n iterationer. Vid varje iteration, undersöker vi varje edge en gång: o(m). Detta är nästlat, dvs O(nm)
3. Leta efter en negativ cykel: kolla alla edges = O(m)

- Notera, vi har n^3 i en kompakt graf:
    - n iterationer av : För varje nods granne (n noder med n grannar) så O(n^3)

Varför är det korrekt?

- Vi räknar ut alla möjligheter, och sätter den till den bästa möjligheten
    - Mer konkret förklaring
- Induktionsbevis: (kanske korrekt?)

1. Basfalet då S=1. Dvs enbart s är i S. Då är d(s) är 0. Vilket vi vet stämmer
2. Inductive hypothesis: Assume theorem is true for $|S| \geq 1$.
3. Yttligare en nod v läggs nu till i S, då vi går från u (tillhör S) till v (tillhör ej S)
4. Antag att det finns en kortare väg till v, via s …. x y … v i S, y outforskad, … x utforskad
5. Antag felacktigt att aldrig hittas och aldrig läggs i v
6. Detta är dock mostäelsefullt, för vi har #noder iterationer (maximala höjden på en väg) vilket betyder om den vägen existerar, och är kortare, kommer v tillsist tilldelas värdet på denna väg

---

Flow network

---

[13] Flow network, ford-fulerssonm goldberg-tarjan, preflow push

flöde:
- Hur mycket går över en kant

Capacitet:
- Hur mycket flöde får gå över en

Cut:
- En delmängd (A, B) där start tillhör A, slutet tillhör B

Kapcitet av en cut:
- Summan av kapaciteten av alla kanter som går ut från mängden

Maximum flow problem:
- Maximera flöded som går in till t

Ford fulekrsson algorithmen (s.112)

Delta(e) = utrymmet för förbättring på en kant
- Representeras av ett värde större än 0 i g_f

DELTA(path) = flaskhalsvärdet = min(delta(path(1)), delta(path(2)) …)

Residualgraf:
- En graf med samma noder som den ursprungliga grafen, men med kanter vars vikt representerar hur mycket mer flöde kan skickas mellan två givna noder i den ursprungliga grafen

Algoritm Ford fulkersson

- Framåtkant: alltid c(e)-f(e) (så mycket vi kan öka)
- Bakåtkant alltid f(e) (så mycket vi kan minska)
- I ford fulkersson har en kant en riktning, som sätts i början, men man har möjlighet att färdas åt båda riktningarna

1. Initiera G och G_f.
    a. G har till en början 0/c(e) på alla kanter
    b. G_f har till en början c(e) i riktningen av alla framåtkanter (inga bakåtkanter finns)
2. Kolla efter en enkel väg från s till t i residualgraf och beräkna DELTA(väg) != 0
    a. Öka flödet med DELTA i G
    b. Uppdatera G_f enligt
        i. Framåtkant: alltid c(e)-f(e) (så mycket vi kan öka)
        ii. Bakåtkant alltid f(e) (så mycket vi kan minska)
3. Upprepa 2 till ingen mer väg finns till slutet i G_f
    a. Om en sådan väg inte finns: har vi maximalt flow.

Tidskomplexitet:
- Vid varje iteration ökar flow med minst 1
    - om C är summan av alla kapaciteter, så kan vi maximalt ha C iterationer
- Vid varje iteration utförs en sökning till slutet.
    - Med antagandet att BFS används, tar denna sökning tiden O(m)
- Så vi får tidskomplexitet O(Cm)

Bipartit matchning:
- I en bipartit graf, kan noderna delas upp i två mängder så att det inte finns en kant mellan två noder i samma mängd

Preflow/förlöde:
- Flöde som enbart håller sig till kapacitetkravet (inte kravet för bevaring av flöde)

Överskridande förflöde: flöde in i kanten - flöde ut ur kanten
- Endast start kan ha negativt förflöde

Höjd: En nod kan enbart skicka till andra noder med lägre höjd
- h(s) = n
- h(t) = 0

Omättad push: Push som begränsas av excess preflow hos noden

- DVS vi pushar allt

Algoritmen Goldberg-tarjan / preflow-push

- Kan även jobba med en residualgraf, men jag väljer istället att tillåta att pusha negativt flöde

1. Sätt f(e) = c(e) för alla e (edges) som innehåller start
    a. Dessa får då excess preflow
2. Ta en godtycklig nod, n, exclusive slutnoden, som kan pusha flöde (med excess)
    a. pusha över en kant den får pusha på dvs vi kan öka flow i utåtriktningen
        i. Framåtkant: öka med min(excess(n), cap(e)-flow(e))
        ii. Bakåtkant: Minska med min(excess(n), flow(e))
3. Fortsätt med 2 till ingen mer kan pusha. Välj då att öka höjden med ett på en med execess preflow och gå till 2.
4. Upprepa 2 och 3 till enbart slutet har excess preflow

Tidskomplexitet:

- Mål är att räkna hur totalt många mättade push vi kan ha
- Höjden kan max vara 2n-1
- Varje nod kan maximalt pusha n-1 gånger innan den får relabel
- DVS maximalt $2n^2$ relabels.
- Utan att göra en relabel kan en mättad push göras åt båda hållen på en kant
    - dvs 2m gånger
- Detta ger totalt $4mn^2$ mättade pushar

Array based heap:

- Root stored at index 1
- k_j means the key stored at index j

Structure of array based heap: Parent is always smaller than children

- k_j <= k_2j, k_j <= k_2j+1
- We know nothing about k_2j vs k_2j+1

Delete min:

1. Replace the element at index 1, with the last element
2. Move it down to its correct position - with comparisons and switches (O(log n))

Insert:

1. Add element on the end
2. Move it upwards with comparsions and switches

Decrease-key / change priority:

- Minska värdet på en nyckel av ett specifikt element

1. Ändra värdet på nyckeln (nu bryter det mot heap egenskapen
2. Flytta den uppåt med jämförelser med föräldrar till man når den rätta platsen

Initialize a heap

ALT 1: n inserts - log n movedown : O(n log n)

ALT 2 (bättre): Heapify - O(n) - min heap förklarat, men funkar likadant för max heap

1. Starta med en osorterad array som du vill använda för att skapa en heap.
2. Börja från den sista föräldern i arrayen och arbeta dig bakåt till roten. För en array med n element är den sista föräldern på index (n/2) trunkerat

3. **FORTSÄTTNING array based heap**
4. För varje förälder, jämför dess värde med sina barn och byt plats om det behövs för att upprätthålla heapegenskapen. Genom att byta plats med det minsta barnet flyttar man successivt mindre element mot toppen av heapen.
5. Upprepa steg 3 för varje förälder tills roten i heapen är nådd.

Height of heap: log_2(n) trukerat

Antal noder på en viss höjd: roundup(n/2^i+1); i är höjden (höjd 0 är lövens nivå)

---

[14] Array based heap

- Specialfall. För löven kan man även ha mindre än detta antal

Hollow heap

Hard problem:
- No one knows an efficient solution (O(n^k)

Decision based problem:
- yes or no question

Optimazation based problem:
- Find an optimized solution to an problem

P class:
- Problems where there exists an O(n^k) solution to the problem

NP class:
- Problems where there exists a O(n^k) verification algorithm
- Notera att P är en delmängd av NP

Polynomial time reductions

- Consider problems P1 and P2 and solution A2, solving P2
  - We need P1. P1 has input x
- Consider f(x) which maps A1 input to A2 input
  - If A2(f(x)) = A1(x), you have found A1
  - If f(x) is efficient, we have created an polynomial time reduction
- Vi noterar detta som P1 <=p P2 för att P1, är lika svårt eller lättare än P2

Y <=p X
1. X easy => Y is easy
2. Y is hard => X is hard

X ≡p Y = Y <=p X && X <=p Y
- Y hard ⇔ X hard
- Y easy ⇔ X easy

NP-completedness (NPC)
1. X ∈ NP
2. For all Y ∈ NP we have Y ≤P X

NP-hard
1. For all Y ∈ NP we have Y ≤P X
- Ergo, doesnt necessarily need to be in NP
  - Ergo doesnt even have to have a polynomial verification

Proving NP-completedness
1. Show Y is NP (ergo has polynomial verification)
2. Take a problem X that is NPC and make a reduction from X, dvs show X <=p Y
   a. DVS Y är åtminstoende lika svårt som X
      i. Alternativt säga att ta X input och gör till Y input
   b. DVS Y är NPC

- För att Y blir åtminstoende NPC då

Exempel Hamilton NP-complete

Problem: Existerar det en enkel väg i en graf så att varje nod besöks exakt en gång

1. Trivialt att kolla att problemet är NP
2. Visa att vi kan ta ett NPC problem och reducera till hamilton problemet
   - VI väljer NPC problemet som 3SAt

---

[15] P, NP, <=p, time reductions, NPC, NP-hard,

- Dvs vi vill transformera en hamilton 3SAT input till hamilton input

3. 3SAT problemet kan ses som en mängd variabler och clauses C1 ∧ C2 …
    a. varje C = xi V xj V xk
4. Vi vill skapa en graf från våra clauses nu, som tolkas som NP problem
5. En clause kan representeras med grafen:
    a. 1 rad per variabel, 2 columner per clause.
    b. Alla dessa kopplas ihop, vilket tillåter 2^3 olika hamilton paths
    c. Beroende på ordningen av variablerna i vår clause så adderar vi edges som tillslut gör att om vi kan lösa hamiltonen så kan vi även lösa 3Sat
6. Gör detta för varje clause
7. Vi har nu transformerat inputen till ett 3SAT problem till input för ett hamilton
    a. 3Sat är sant om och bara om och hamilton existerar

Exmepel Traveling salesman problem NP-complete

Problem: Finns det en väg genom alla noder i en graf så att den totala vikten är maximalt X

1. Trivialt att kolla om vår lösning är NP. Linjärt att gå igenom noderna och summera
2. Vill visa att vi kan reducera från NPC problem hamiltonian problem till TSP
    a. Vi skapar en ny graf med kanter mellan alla noder
    b. Om motsvarande kant finns i hamilton problemet så sätter vi den kanten till
        1. Om den inte finns sätter vi vikten till 2
    c. Om och bara om det finns en lösning till TSP problemet med total max distans = #noder, finns en hamilton ccykel
        i. För att det kräver att vi bara använder går igenom varje nod max en gång. DVs de är antingen båda sanna eller båda falska
        ii. Vi har redducerat hamilton till TSP successfullly

The first NP-completedness proof

1. Visa Y är NP
2. For all X ∈ NP we have X ≤P Y

- The first problem to prove NPC was circuit satisfiability:

1. Prove 1. Trivially Polynomial to just stick in our solution in the variabels and check if its true or not
2. Prove 2:
    a. We know X is in NP with polynomial verification algorithm A(I,S)
        i. I is input to algorithm
        ii. S is the suggested solution
    b. Tolka I som n bitar och den suggested solution som bitar v1, v2 … vK
    c. Skapa en krets som implementerar A i polynomial tid
        i. Komplicerade delen som ignoneras
    d. Sätt in I (kända input värdenoch S = “k variabler” in i vårt krets
    e. Vi har nu visat att Circuit satisfiability problemet kan reduceras från circuitsatisfiability problemet till vilket NP problem som helst

Unit propagation

Linear programming

Maximize a linear function in a region

1. A region defined by lines, including x_i >= 0

[16] Linear programming, integer programming, the simplex algorithm

   a. i.e linear constraints
   b. Such as 3x0+x1=18 and -0.5*x0+x1=4
2. A linear objective function such as max z = x0 + 2x1

Feasible solutions (p)

- Varje begränsning, definierar ett halvplan (ett plan som delas av en linje) i n dimensioner
- Intersection av alla dessa områden är punkter som håller sig till alla begränsningar
- Området kommer alltid att vara konvext
  - DVS allla inre vinklar är mellan 0 och 180 grader
  - Kanter (som uppstår från korsande linjer) kallas för vertexes

Konsekvens av att det är konvext

- We can write a point as pk = λ · pi + (1 − λ) · pj, with 0 ≤ λ ≤ 1
  - pi och pj ligger i området

Infeasable: p is empty (no feasible solutions)
Unbounded: no finite solution exists

local optimal solution: "Lokal extrempunkt"

- x är ett lokal optimum om det inte finns någon annan punkt y i närheten av x som ger oss ett bättre resultat enligt vår målfunktion.

Theorem:

- A local optimum ofa linear program is also a global optimum

Theorem: F

- or a bounded feasible linear program with feasible region P, at least one vertex is an optimal solution.

Non basic:

- variblerna i ekvationen på höger sida i slack form

Basic:

- de ensammana variablerna på vänster sida i slack form

Simplex algorithm (usage):

- Determine feasibility. Lösa linjära program problem

The simplex algorithm (exampel)- maximera z

1. Våra begränsningar: (innehåller bara non basic variabels - ursprungliga)
   a. 3*x0+x1<=18
   b. -0.5*x0+x1 <=4
2. Sätt större än noll och ersätt med variabler, för varje begränsning (tillagt basic variables x2, x3)
   a. x2 = 4 − (−0.5*x0 + x1)
   b. x3 = 18 − (3x0 + x1)
3. max z = x0 + 2*x1 + y
4. Skriv om tills vi enbart har negativa coeficienter:
   1. Välj första non basic: x0 entering basic variabel
   2. basic begränsar hur mycket x0 kan ökas
   3. (om det finns är ingen begränsning, då är det linjära programmet obegränsat)
   4. x0 kan maximalt vara 6 (begränsas av x3 (tolkas som 0) = leaving basic variabel)
   5. Gör enkel algebraisk omskrivning för att byta plats på x3 och x0

- x2 = 7 − (0.167x3 + 1.167x1)
- x0 = 6 − (0.333x3 + 0.333x1)
- Ändra z
  - Bara skriv x0 utifrån x3 och x1 (se ovan x0 = 6-(...))
  6. Gör samma för x1 (för att alla coeficienter i z ekvationen är inte negativa)
- x1 = 6 - (...)

- x0 = 4 - (...)
- z = -0.6x3 -1.4x2 + 16
    - Set (x3,x2) = gives answer 16.

Integer programming:

- Linear programming with constraint x_i är helatal

branch-and-bound paradigm: Metod för att lösa integer problem

Branch: Dela upp problemet i flera, lättare problem
Bound: Evaluera, nå heltals gren (stoppa sökning) eller potentiellt överge vissa grenar

1. Anta att problemet kan ha icke integer svar (lös det som ett vanligt linjärt program)
2. Om exempelvist x_k har värde u (decimaltal) enligt simplex, då skapar vi två nya program

- Antag att vi har ett heltalsprogram och matar in det i simplex-algoritmen, och antag att (resultatet blir att) variabeln xk inte är ett heltal, utan får värdet u enligt simplex-algoritmen.
    - Då kan vi skapa två nya linjära program genom att förgrena:

1. Det första programmet har en extra begränsning $x_k \leq \lfloor u \rfloor$ (avrundat nedåt till närmaste heltal).
2. Det andra programmet har en extra begränsning $x_k \geq \lceil u \rceil$ (avrundat uppåt till närmaste heltal)

- Varje nya problem löses direkt med simplex-algoritmen.
    a. Om något av de nya programmen har en heltalslösning kan vi begränsa sökträdet och fokusera på den grenen (bound).
    b. Om något av de nya programmen har en icke-heltalslösning och den är bättre än den bästa heltalslösningen vi hittills har, lägger vi den i en kö för att eventuellt utforska senare.

- Fortsätt med detta till alla grenar antingen kastas bort eller når en heltalslösning