Contents

1 Matching **1**
- 1.1 Bipartitt Matching . . . 1
- 1.2 Vektet Bipartitt Matching . . . 2
- 1.3 Vektet (Ikke-Bipartitt) Matching . . . 2

2 Lineær Programmering **3**
- 2.1 Kanonisk Form . . . 3
- 2.2 Heltallsprogram . . . 3
- 2.3 Dualitet . . . 4
- 2.4 Komplementær Slakkhet . . . 4

3 Den ungarske metoden **6**
- 3.1 Prinsippet Bak Algoritmen . . . 6
- 3.2 Algoritmen . . . 7

4 Approksimasjon **8**
- 4.1 Maximum Independent Set/Største Uavhengige Set . . . 8
- 4.2 Set Cover/Mengdedekke . . . 8
  - 4.2.1 Approksimering ved avrunding . . . 9
  - 4.2.2 Approksimering ved dualbasert avrunding . . . 9
  - 4.2.3 Approksimering ved grådighet . . . 10

5 Primal-Dual Metoden **11**
- 5.1 Mengdedekke Revisited . . . 11
- 5.2 Feedback Vertex Set/Sykelkritiske Noder . . . 11
- 5.3 Korteste Vei . . . 12
- 5.4 Minimum Knapsack Problem . . . 13

6 Grådige Algoritmer og Lokale Søk **14**
- 6.1 Sekvensering (Parallelle Maskinjobber) . . . 14
- 6.2 Klynge-Analyse . . . 14
- 6.3 Metric TSP eller Traveling Salesman IRL . . . 15

7 Grådighet og Matroider **16**
- 7.1 Polymatroider og Submodularitet . . . 16
- 7.2 Approksimering og Snitt Over Flere Matroider . . . 17
- 7.3 Eksempler . . . 17
- 7.4 "Greedoids" . . . 18

8 Flyt **19**
- 8.1 Bakgrunn . . . 19
- 8.2 Praktiske Eksempler . . . 20
  - 8.2.1 Carpool Driver . . . 20
  - 8.2.2 Baseball Elimination Problem . . . 20

8.2.3 Maximum Density Subgraph . . . 20
8.2.4 Project Planning . . . 21
8.2.5 k-Orientering . . . 21
8.3 Most Improving / Shortest Augmenting Path . . . 21

9 Preflyt **23**
9.1 Push-Relabel . . . 23
9.2 Kjøretid . . . 24
9.3 Forbedringer . . . 24

10 Multiflyt og Multiplikativ Vektoppdatering (MVO) **25**
10.1 LP Formulering . . . 25
10.2 Cut Conditions og $K = 2$ . . . 26
10.3 Multiplikativ Vektsoppdatering (MVO) og Packing Problems . . . 26
10.4 Garg-Könemann Algoritmen . . . 27

11 Randomisering **28**
11.1 Max-SAT, Max-Cut og Min Cut . . . 28
11.2 Derandomisering . . . 29
11.3 Randomisert Avrunding av Lineærprogrammer . . . 29

12 Onlinealgoritmer **31**
12.1 Onlineproblem og Kompetitiv Analyse . . . 31
12.2 Paging . . . 31
12.2.1 Øvre grense . . . 32
12.2.2 Nedre grense . . . 32
12.2.3 Markeringsalgoritmer . . . 33

13 Parametrisering **34**
13.1 The Good, the Bad and the Ugly . . . 34
13.2 Kernelization . . . 35
13.2.1 Vertex Cover/Nodedekke . . . 35
13.2.2 Feedback Arc Set In Tournaments . . . 36
13.2.3 Edge Clique Cover . . . 36
13.3 Bounded Search Tree . . . 37

1. Matching

Matching er et ganske greit problem, men som går igjen mye på starten og i eksempler av forskjellige konsepter, så greit å ha god kontroll på det. Generelt så er det et av de få graf-optimaliseringsprobleme som kan løses i polynomisk tid. Problemet er formulert som følger: Gitt en graf $G = (V, E)$, finn en $M \subseteq E$ slik at grafen $G' = (V, M)$ ikke inneholder noen noder av grad > 1. Det finnes fire forskjellige tilfeller, som alle kan sees på som ganske forskjellige problemer. De kommer av hvorvidt grafen er bipartitt og/eller vektet.

Trolig det viktigste konseptet for å oppnå optimale løsninger på matching-problemer er forøkende stier. Først må vi ha litt terminologi: En maksimal matching er en løsning der vi ikke kan legge til flere kanter og fremdeles ha en matching. Et maksimum er den optimale løsningen. Åpenbart er et maksimum også en maksimal løsning. En forøkene sti er en sti der kantene veksler mellom å være kanter som er med i en matching, og ikke med i en matching. Dersom en slik sti starter å slutter i noder som ikke er med i matchingen, så er den forøkende. For å se dette, bare "flip" alle kantene i stien (fjern de som er med i matchingen, legg til de som ikke er det). Dette vil naturligvis øke matchingen med 1 kant (husk at stien startet og sluttet i noder som ikke var med i matchingen).

Det kanskje første teoremet i algkons er da at en løsning er et globalt maksimum hvis og bare hvis det ikke finnes noen forøkende stier. Ganske åpenbart den ene veien, men litt mer overraskende at ingen forøkende stier er et tilstrekkelig krav for å oppnå globalt maksimum. For å se dette, si at vi har en graf med en matching M uten noen forøkende stier. I tillegg antar vi at det finnes en M' slik at $|M'| > |M|$. Dersom vi da ser på grafen der kantene er i $M \oplus M'$, kan vi si et par ting om alle stiene. Her må naturlig nok alle stier alternere mellom M og M'. Dermed finnes det ingen odde sykler. Videre, siden $|M'| > |M|$ må det finnes minst en vei som både starter og slutter i M'. Dette vil da være en alternerende sti i $G = (V, M)$, så vi har oppnådd en kontradiksjon. Dermed kan vi naturlig nok generelt basere uvektet matching algoritmer på slike forøkende stier (detaljer i boka).

1.1 Bipartitt Matching

Når grafen er bipartitt ($G = (U \cup V, E)$) blir matching problemet noe "enklere". Men, i flere virkelighetstilfeller modellerer vi faktisk bipartitt matching, og ikke bare matching (typiske "assignement" problemer). Naturlig nok kan vi her også basere løsninger på forøkende stier, og det gir også opphav til et nytt teorem. Først, for å se hvordan man lager en slik forøkende sti i en bipartitt graf, start me en hvilken som helst umatchet node i U. Stien vil naturligvis alternere mellom U og V, og må også ende i V for å være forøkende. Dette gir oss da **Hall's Marriage Theorem**, som forteller oss at en bipartitt graf $G = (U \cup V, E)$ har en perfekt matching hvis og bare hvis $|S| \leq |N(S)|, \forall S \subseteq U$, der en perfekt matching er en mathing der alle noder har grad nøyaktig lik 1, og $N(S) = \{u \in V : (u, v) \in E, v \in S\}$ (eller sagt med ord, "naboene" til nodene $v \in S$). Dette er åpenbart nødvendig (ellers ville det ikke eksistert nok noder i $N(S)$ til å matche alle nodene i $S \subseteq U$), men det er også tilstrekkelig. For å se dette, la oss si at vi har en udekt node $u \in U$ (ikke perfekt altså), og at det ikke eksisterer noen forøkende stier (løsningen er med andre ord globalt maksimum). For at dette skal stemme må da $S \setminus \{u\}$ være matchet med T, og vi har $|T| = |N(S)| = |S| - 1 \iff |S| > |N(S)|$.

For uvektet bipartitt matching er altså disse forøkende stiene den naturlige måten å optimere problemet på. Videre finnes det et par andre smarte ting å gjøre, f.eks. Hopcroft-Karp som istedenfor å finne en og en forøkende sti finner den alle disjunkte forøkende stier. Dette forbedrer kjøretiden fra $\mathcal{O}(nm)$ til $\mathcal{O}(\sqrt{n}m)$. Andre varianter er å legge til en kilde og et sluk på hver "side" av den bipartitte grafen, og løse med maks flyt.

1.2 Vektet Bipartitt Matching

Dette løses med den ungarske metoden (se kapittel 3). Men det gis en liten teaser til tema om grådighet i forelesning 1, så jeg legger det ved her og. En helt naiv grådig løsning, alltid velge den tyngste/letteste (avhengig av om det er et maksimerings eller minimeringsproblem) kanten som er lovlig, gir en 2-approksimasjon, ettersom optimeringsproblemet bipartitt matching er snittet av to transversalmatroider. Les videre for å kunne dekryptere den forrige setningen (Det viser seg faktisk at en grådig løsning gir en 2-approksimasjon i det generelle, ikke-bipartitte tilfelle også, og ja, vi kommer til det og).

1.3 Vektet (Ikke-Bipartitt) Matching

Løses gjerne med Lineær Programmering. Se neste seksjon. Kan evt. også løses ved snittet over to matroider, ved å lage en k-orientering og ta snittet av hale- og hode-partisjonsmatroiden. Se seksjon 7.

2. Lineær Programmering

Dette er et av de viktigste konseptene i faget, og grunnlaget for mye av pensum, særlig første halvdel. Lineære programmer er en veldig generell måte å modellere/sette opp optimaliseringsproblemer over $\mathbb{Q}^n$, der løsningen består av å optimalisere en lineær målfunksjon, samtidig som vi holder oss innen lineære restriksjoner. Lineær program har en veldig visuel løsningsform, der restriksjonene tilsammen lager en (konveks) polytop av gyldige løsninger, og målfunksjonen bestemmer retningen hvor den optimale løsningen ligger. Dessverre så kan også restriksjonene være umulige, eller ubegrenset. Men dersom det ikke gjelder, kan vi generelt finne den optimale løsningen i polynomisk tid!

2.1 Kanonisk Form

Generelt ser den kanoniske formen på lineær program slik ut:

$$\begin{aligned} \text{Maximize:} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to:} \quad & A\ \boldsymbol{x} \leq \boldsymbol{b} \\ & x_i \geq 0 \quad \forall x_i \in \boldsymbol{x} \end{aligned}$$

Her er A en matrise, mens c, x, b er vektorer. Grunnen til at en slik "kanonisk form" er mulig er ganske enkelt fordi alle lineærprogram kan omformuleres til denne formen. Det gjøres slik: $min(cx) = max(-cx), x \geq b \iff -x \leq b, x = b \iff x \leq b \wedge x \geq b$. Merk at både gjennom boka og kompendiet så brukes min, max om hverandre (begge er definert som kanonisk form). Dette fordi at enkelte problem er naturlig å tenke på som maksimeringsproblemer, mens andre er mer naturlig å tenke på som minimeringsproblemer. Uansett, det er ønskelig å få alle ulikhetene samme vei for restriksjonene, så dette gjøres stort sett også.

2.2 Heltallsprogram

Veldig mange praktiske (og teoretiske) problemer har en form for heltallighet som krav. Hvis $\boldsymbol{x}$ representerer antal av forskjellige komponenter i en optimal løsning, er det naturlig å tenke seg at alle $x_i \in \boldsymbol{x}$ må være heltallige, avhengig av optimaliseringsproblemet (hvis $\boldsymbol{x}$ representerer mennesker fra forskjellige grupper f.eks. så er det jo uheldig å få en fraksjonell løsning). Dermed kan dette være en naturlig restriksjon på lineærprogrammet vårt. Da ender vi opp med det som kalles et heltallsprogram. Fra nå av bruker vi IP og LP (IP = Integer Program, LP = Linear Program). En annen variant av IP kan modelers som at $x_i \in \{0, 1\}$. Dette gir en veldig naturlig måte å formulere "decision"-type problemer, der $0, 1$ representerer valg (typisk ta med i løsningen eller ikke). Som vi vet er slike "decision"-problemer ofte NP-harde. Det er også enkelt å modellere mange NP-harde problemer som IPer, noe som betyr at det å løse et IP generelt er NP-hardt også.

Likevel så er det flere ting vi kan gjøre for å jobbe rundt dette, så helt håpløst er det ikke. For det første har enkelte lineær-program alltid heltallige løsninger (selvom det ikke har heltallskrav), og for det andre kan ofte løsningen på slakkingen av IPet fortelle oss noe om løsningen (hvis vi klarer å argumentere for en øvre grense på forskjellen mellom løsningen på IPet og det tilhørende LPet). En slakking av et heltallsprogram er når vi fjerner heltallskravet og ender opp med et tilhørende lineærprogram.

2.3 Dualitet

Også et utrolig viktig konsept som går igjen i første halvdel av pensum. For ethvert lineærprogram, kan vi enkelt konstruere det såkalte "duale" lineærprogrammet. Dette omtales som dualen, mens det originale lineærprogrammet da omtales som primalen. Vi lager dualen ved å sette en variabel for hver restriksjon i primalen, og en restriksjon i dualen for hver variabel i primalen. En intuitiv forståelse av dualen er litt vanskelig å skrive ned, men det kommer når det jobbes med. Det som er målet er å finne en øvre grense for den optimale løsningen for primalen, og det viser seg at denne øvre grensen også er et lineær program. Dualen til den kanoniske formen nevnt tidligere ser slik ut, og forsøker å skvise en øvre grense så nært som mulig:

$$\begin{array}{ll} \text{Mimimize:} & \boldsymbol{b}^T \boldsymbol{y} \\ \text{subject to:} & A^T \boldsymbol{y} \geq \boldsymbol{c} \\ & y_i \geq 0 \quad \forall y_i \in \boldsymbol{y} \end{array}$$

Hadde vi tatt dualen av dette lineær-programmet igjen, hadde vi endt opp med den kanoniske formen vi starta med. Dette er generelt også alltid sant (dualen til dualen er igjen primalen).

For gyldige løsninger på primalen og dualen kan vi generelt skrive følgnde ulikheter

$$\begin{array}{ll} \text{Primal:} & A\boldsymbol{x} \leq \boldsymbol{b} \\ \text{Dual:} & \boldsymbol{c} \leq \boldsymbol{y} A \\ x, y \geq 0 & \boldsymbol{c}\boldsymbol{x} \leq \boldsymbol{y} A \boldsymbol{x} \leq \boldsymbol{y}\boldsymbol{b} \\ \text{Ved optimum (!):} & \boldsymbol{c}\boldsymbol{x} = \boldsymbol{y} A \boldsymbol{x} = \boldsymbol{y}\boldsymbol{b} \end{array}$$

Den nest nederste kalles ulikheten kalles svak dualitet, mens den nederste likheten kalles sterk dualitet. Beviset for sterk dualitet kan sees i boken.

Dette gjelder naturlig nok kun når både primalen og dualen er endelig optimale (det holder å vise at en av dem er det, for da må også den andre være det). Generelt finnes det 4 tilfeller, som jeg lister raskt opp nå; 1) PRIM endelig optimal, DUAL endelig optimal, 2) PRIM ubegrenset, DUAL umulig, 3) PRIM umulig, DUAL ubegrenset, 4) begge umulig.

2.4 Komplementær Slakkhet

Dette er et viktig teorem, og en basis for primal-dual metoden som det står skrevet om litt lenger nede. Komplementær slakkhet er følgende "betingelse" for et PRIMAL/DUAL par:

$$\begin{aligned} y_i = 0 &\Longleftarrow (Ax)_i \neq b_i \quad \forall i \\ x_j = 0 &\Longleftarrow (yA)_j \neq c_j \quad \forall j \end{aligned}$$

Det er lett å se at komplementær slakkhet impliserer likhet mellom målfunksjonene (at det er en tilstrekkelig betingelse for likhet). Men at kanskje noe mer overraskende resultat er at komplimentær slakkhet også er nødvendig for å oppnå likhet! Dvs. likhet mellom målfunksjonene impliserer komplimentær slakkhet. Her følger et bevis:

Se for oss at vi har likhet mellom målfunksjonene, altså $\boldsymbol{c}\boldsymbol{x} = \boldsymbol{y} A \boldsymbol{x} = \boldsymbol{y}\boldsymbol{b}$. Den siste halvdelen kan vi skrive om som $\boldsymbol{y}(\boldsymbol{b} - A\boldsymbol{x}) = 0$, og vi vet at $\boldsymbol{b} - A\boldsymbol{x} \geq 0$, siden restriksjonen var at $A\boldsymbol{x} \leq \boldsymbol{b}$. Dette gir oss følgende:

$$\begin{aligned} \boldsymbol{y}(A\boldsymbol{x} - \boldsymbol{b}) = & \sum_{i=1}^{n} y_i(Ax - b)_i = 0 \\ & (Ax - b)_i \geq 0 \qquad \forall i \in \{1, .., m\} \\ & y_i \geq 0 \qquad \forall i \in \{1, .., m\} \end{aligned}$$

og det er helt tydelig at vi må ha komplimentær slakkhet (dette viser strengt tatt bare komplimentær slakkhet den ene veien, men et helt symmetrisk argument gir beviset andre veien og).

3. Den ungarske metoden

Denne algoritmen er et praktisk eksempel (historisk sett forløperen) til Primal-Dual metoden, som det står om i kapittel 5. Jeg går litt raskt igjennom dette kapittelet, siden det er veldig greit om en har kontroll på Primal-Dual metoden. Den løser som nevnt tidligere vektet bipartitt matching. Den har og en matrise-versjon som ikke er del av pensum, men ganske enkel å utføre. Videre ser vi på graf versjonen. Vi har en bipartitt graf $G = (U \cup V, E)$. Vi starter med å sette opp lineærprogrammet vårt:

$$\begin{aligned} \text{Maximize:} \quad & \sum_{j \in E} c_j x_j \\ \text{subject to:} \quad & \sum_{j:i \in j} x_j = 1, \quad \forall i \in U \cup V \\ & x_j \geq 0 \quad \forall j \in E \end{aligned}$$

Restriksjonen her kan tolkes som at summen av kanter tilkoblet node i må være nøyaktig 1. Med andre ord er det kun perfekte matchinger som er tillat. Videre er den sentrale idéen i algoritmen basert på å betrakte dualen, så vi setter opp den og:

$$\begin{aligned} \text{Minimize:} \quad & \sum_{v \in U \cup V} y_v \\ \text{subject to:} \quad & y_u + y_v \geq c_j, \quad u, v : j = (u, v), \forall j \in E \\ & y_v \geq 0 \qquad \forall v \in U \cup V \end{aligned}$$

Dualvariabler kan tolkes prislapper på nodene, der prisen på to nodepar som har en kant mellom seg må overskride prisen av kanten. Dette gir opphav til en ny graf basert på G kalt likhetsgrafen basert på disse såkalte nodeveingene. I likhetsgrafen tar vi kun med de kantene der dualrestriksjonen er stram (der nodeveingen er lik kantvekten).

3.1 Prinsippet Bak Algoritmen

Idéen er at vi ønsker å oppnå komplimentær slakkhet (som vi har sett betyr det at vi oppnår optimum). Derfor får vi kun lov til å skru på x_j dersom dualrestriksjonen for kant j er stram, altså om $y_u + y_v = c_j$. Dermed bygger vi matchingen på en måte som gjør at dersom vi oppnår en gyldig løsning for primalen vil den oppfylle komplimentær slakkhet, og dermed være optimal. For å se at dersom vi har komplimentær slakkhet har også likhetsgrafen en perfekt matching, anta at den ikke har det. Da har vi et subset $S \subseteq U$ med $|S| > |N(S)|$ (fra Halls teorem). Det vi ønsker å gjøre nå er å senke alle y_v for $v \in S$ mest mulig. La $\epsilon = min\{y_u + y_v - j : u \in S, v \notin N(S)\}$ Vi kan nå sette $y_u = y_u - \epsilon, \forall u \in S$ og $y_v = y_v + \epsilon, \forall v \in N(S)$. Dette kan vi åpenbart gjøre uten å bryte noen dualrestriksjoner (se på hvordan ϵ er konstruert), men det leder også til en bedre dual. Dette er en kontradiksjon (det bryter med svak dualitet), siden vi antok at vi hadde komplimentær slakkhet. Dermed må det være slik at $|S| \leq |N(S)|, \forall S \subseteq U$ i likhetsgrafen, og vi vet at likhetsgrafen har en perfekt matching.

3.2 Algoritmen

Analysen av hvorfor komplimentær slakkhet leder til en perfekt matching blir også utgangspunktet for algoritmen. Vi starter med å lage en vilkårlig lovlig dual (typisk settes $y_u = max\{c_j : u \in j\}, \forall u \in U$, mens vi setter $y_v = 0, \forall v \in V$). Deretter finner vi en ikke utvidbar (maksimum) matching i likhetsgrafen. Dersom den ikke er perfekt, finn en $S \subseteq U$ der $|S| > |N(S)|$ (typisk ved å traversere fra en ikke matchet node). Deretter gjør som i forrige avsnitt, og sett $y_u = y_u - \epsilon, \forall u \in S$ og $y_v = y_v + \epsilon, \forall v \in N(S)$. Vi får nå (minst) en ny kant i likhetsgrafen (merk, kanter kan forsvinne og), og vi kan dermed gå tilbake til å finne ikke-utvidbare matchinger, og vi gjør dette i hver iterasjon til vi finner en perfekt matching. Som nevnt vil da denne være den maksimale perfekte matchingen i G på grunn av komplimentær slakkhet.

For detaljer, se forelsning / boka. Det viktige her er den generelle idéen bak algoritmen. For en generalisering av denne idéen, se kapittel 5.

4. Approksimasjon

Approksimasjonsalgoritmer er algoritmer for å løse optimaliseringsproblemer. Konseptet som introduseres er litt løst basert på følgende utsagn: For å løse optimaliseringsproblemer trenger man I) Optimal løsning, II) I rask tid, II) for alle problemer. Velg 2. Approksimeringsalgoritmer slacker på kravet om optimal løsning, ved å heller gi et svar som er så nært som mulig optimalt, men ikke nødvendigvis optimalt.

For å få noe av verdi må vi kunne si noe om hvor nære approksimasjonen er, eller approksimasjonsgraden. La APX og OPT være hhv. approksimasjonen og den optimale løsningen. Da sier vi at algoritmen er en α-approksimasjon dersom.

$$APX \geq \frac{1}{\alpha} \cdot OPT \text{ For maksimeringsproblemer} \tag{1}$$
$$APX \leq \alpha \cdot OPT \text{ For minimeringsproblemer} \tag{2}$$

Et viktig konsept som går igjen er PTAS vs FPTAS. Definisjonen på en PTAS er en familie med approksimeringsalgoritmer $\{A_\epsilon\}_{\epsilon>0}$, der alle $A \in \{A_\epsilon\}_{\epsilon>0}$ er $(1+\epsilon)$-approksimasjoner, med polynomisk kjøretid i n. En FPTAS er en PTAS der kjøretiden også er polynomisk i $\frac{1}{\epsilon}$. For å illustrere forskjellen oppfyller $O(n^{\frac{1}{\epsilon}})$ kun kravet til PTAS, mens $O(n^c + (\frac{1}{\epsilon})^d)$ oppfyller kravet til FPTAS også.

4.1 Maximum Independent Set/Største Uavhengige Set

For en graf $G = (V, E)$, kalles en $I \subseteq V$ independent/Uavhengig dersom ingen to noder i I er koblet sammen.

Å finne det største kan skrives som et lineærprogram på følgende måte

$$\begin{array}{lll} \text{Maximize:} & \sum_{v \in V} x_v & \\ \text{subject to:} & x_u + x_c \leq 1, & \text{for alle kanter } (u,v) \in E \\ & x_v \in \{0,1\}, & \text{for alle noder } v \in V \end{array}$$

Dette er hovedsaklig tatt med som et eksempel på problemer som ikke kan approksimeres (Ikke 100% sikker på dette, men tror det er et teorem om at det ikke finnes noen PTAS for største uavhengige sett, med mindre $P = NP$). Det samme gjelder for Maksimum Clique problemet (Disse problemene er ganske tydelig relatert, for å se dette ta komplimentet av grafen til en instans av det ene problemet og sett det som en instans av det andre).

4.2 Set Cover/Mengdedekke

Mengdedekke er et veldig generelt problem, formulert som følgende: For et grunsett $E = \{e_1, e_1, ..., e_m\}$ har vi flere subset $S_1, S_2, ..., S_n$ der $S_i \subseteq E$, samt en tilknyttet vekt w_i for hver S_i. Målet er å finne den sammensetningen av subset som dekker hele grunsettet med lavest vekt. Dersom $w_i = 1, \forall i \in \{1, .., n\}$ har vi et *uvektet* mengdedekke problem. Formulert som et lineær program har vi altså følgende:

$$
\begin{aligned}
\text{Minimize:} \quad & \sum_{i=1}^{n} w_i x_i \\
\text{subject to:} \quad & \sum_{j:e\in S_j} x_j \geq 1, \qquad \text{for alle } e \in E \\
& x_i \in \{0,1\}, \quad i \in \{1,...,n\}
\end{aligned}
$$

F.eks. er da et nodedekke et spesialtilfelle av mengdedekke. For å redusere fra en generell instans av et nodedekke til mengdedekke, la grunsettet være alle kanter. For hver node, lag et subset med alle tilkoblede kanter (slik at $n = |V|$), og la $w_i = v_i$.

I øving 2 er det et annet eksempel der mengdedekke reduseres til *uncapacitated facility problem* og motsatt.

4.2.1 Approksimering ved avrunding

Den første taktikken for å approksimere mengdedekke er å approksimere ved avrunding. Det fungerer ved først å løse slakkingen til lineærprogrammet, finne $f = \max(|\{j : e \in S_j\}|)$, og deretter runde alle $x_i \geq \frac{1}{f}$ opp til 1, og sette resten til 0. Dette gir en f-approksimering. En intuitiv tolkning av f her er maksimalt antal subset et element opptrer i. Det er tydelig at dette fremdeles er en gyldig løsning til lineærprogrammet, ettersom $\sum_{j:e\in S_j} x_j \geq 1$ i den optimale løsningen (siden summen ikke har mer enn f ledd må minst et av dem være mer enn $\frac{1}{f}$).

Dette er grunnen til at relaksering med avrunding løser nodedekke som en 2-approksimasjon. Hver kant (hvert element) er intil nøyaktig to noder (del av to subset). Dermed har vi $f = 2$.

4.2.2 Approksimering ved dualbasert avrunding

Dualen til (LP relakseringen av) lineærprogrammet til mengdedekket ser ut som følger:

$$
\begin{aligned}
\text{Maximize:} \quad & \sum_{i=1}^{n} y_i \\
\text{subject to:} \quad & \sum_{i:e_i\in S_j} y_i \leq w_j, \quad \text{for alle } j \in \{1,...,n\} \\
& y_i \geq 0 \qquad \text{for alle } i \in \{1,...,n\}
\end{aligned}
$$

Husk: I primalen hadde vi en restriksjon pr. element. Dvs. at hver y_i tilsvarer en restriksjon, eller et element. I tillegg har vi i dualen en restriksjon pr. delmengde i primalen. Den intuitive moten å forstå dualen på kan være at hvert element har en pris. Prisen til et sett (summen av prisen til alle elemntene i settet) kan ikke overskride vekten av settet. Målet er da å maksimere prisene y_i.

Ideen er nå å bruke noe av konseptet bak komplimentær slakkhet, samt den optimale løsningen til dualen for å konstruere en god approksimasjon til primalen. Vi gjør det på følgende måte: Løser dualen og får $\boldsymbol{y_i^*}$. Deretter løser vi primalen ved å velge alle de S_j som er slik at $\sum_{i:e_i\in S_j} \boldsymbol{y_i^*} = w_j$, dvs der dualrestriksjonen er stram. Dette fungerer ettersom hvis det skulle ha fantes et element som ikke var dekket, ville det betydd restriksjonen ikke var stram for noen av mengdene som elementet var del av. Dermed er løsningen heller ikke optimal siden vi kunne uten problem økt prisen på elementet.

Hva får vi fra dette? Jo, det blir en ny f-approksimasjon, f samme som i forrige seksjon. Det er hovedsaklig to ting som er sentrale for å se dette. For det første, siden $DUAL \leq OPT$, har vi at $\sum_{i=1}^{n} y_i \leq OPT$. Videre er S_j kun med i løsningen dersom $\sum_{i:e_i \in S_j} y_i = w_j$. Resten klarer du å sette sammen selv (eller se i boka...).

4.2.3 Approksimering ved grådighet

Den siste metoden for approksimering av mengdedekke er presentert som en grådig løsning (og her den suverent beste!). Dette er en H_n-approksimering ($H_n = \sum_{k=1}^{n} \frac{1}{k}$, altså summen av de n første leddene i den harmoniske rekken), der $n = |V|$. Den er ganske enkel å forklare. Approksimasjonsgraden er litt mer å forklare. Algoritmen fungerer slik: Ved hvert steg, regn ut det settet som gir mest lavest stykkpris på udekte elementer, og legg til dette settet i løsningen. Itererer til alle elemter er dekket (maks n ganger). Litt mer formelt, la I være den nåverende løsningen, og la $\hat{S}_i = S_i \setminus \{e \in I\}$. Ved hver iterasjon gjør $I = I \cup S_j : j = min_i(\frac{|S_i|}{w_i}), \forall i$.

For bevis av approksimasjonsgraden her, se boka, side 108.

5. Primal-Dual Metoden

Dette er den generelle ungarske metode/dualbasert avrunding i mengdedekke metoden, og gir ofte opphav til gode approksimeringsalgoritmer. I tillegg er approksimeringsgraden ofte lett å analysere.

Metoden er veldig generell, og går ut på å starte med en gyldig dual-løsning (ikke nødvendigvis opptimal), og så gradvis forbedre dualen samtidig som man bygger løsningen til primalen (ofte basert på komplimentær slakkhet). Analysen av løsningen går da ofte ut på at vi har komplimentær slakkhet en vei, mens en approksimert komplimentær slakkhet den andre veien. Det enkleste er nesten bare å se på eksempler, så vi kjører på.

5.1 Mengdedekke Revisited

Se over for en beskrivelse av mengdedekke. Primal-dual metoden for mengdedekke gir opphav for løsningen approksimering ved dualbasert avrunding, kun med en liten endring. I steden for å starte med å konstruere en optimal løsning til dualen $\boldsymbol{y_i^*}$, så starter vi heller med en gyldig løsning, f.eks $y_i = 0, \forall i$. Deretter gjør vi som i analysen av gyldigheten, og ser på et vilkårlig udekt element e_i. Siden elementet er udekt, betyr det at $\sum_{k:e_k \in S_j} y_k < w_j, \forall j : e_i \in S_j$, og dermed kan vi øke y_i med en $\epsilon > 0$, slik at minst én dualrestriksjon blir stram. Da legger vi til S_j i løsningen for alle $j : \sum_{k:e_k \in S_j} y_k = w_j$, altså der dualrestriksjonen er stram. Slik kan vi bygge løsningen vår med maks $n =$ antall elementer iterasjoner.

Her ser vi at komplimentær slakkhet åpenbart er oppfylt den ene veien, siden vi valgte å sette $x_j = 1 \iff \sum_{k:e_k \in S_j} y_k = w_j$. For alle andre har vi $x_k = 0$. Andre veien derimot har vi kun approksimert komplimentær slakkhet. Dette fordi at for en $y_i > 0$ så vil $\sum_{j:y_i \in S_j} x_j \leq f$, der f er som tidligere (maks antall ganger et element opptrer i forskjellige set). Dermed er det også lett og se at dette blir en f-approksimasjon.

5.2 Feedback Vertex Set/Sykelkritiske Noder

Dette problemet kan tolkes på flere måter. Personlig er den mest intuitive måten som følger: Vi har en urettet graf $G = (V, E)$, med tilhørene vekter w_i til nodene. Vi ønsker å fjerne noder billigst mulig slik at G blir asyklisk. Som et IP (der $\boldsymbol{C}$ betegner settet av sykler i G):

$$\begin{aligned} \text{Minimize:} \quad & \sum_{i \in V} w_i x_i & & \\ \text{subject to:} \quad & \sum_{i \in C} x_i \geq 1, & & \forall C \in \boldsymbol{C} \\ & x_i \in \{0, 1\} & & \forall i \in V \end{aligned}$$

Her er det viktig å se at $\boldsymbol{C}$ vokser eksponensielt med antall noder, så det blir mange restrikssjoner etterhvert. Tar vi relakseringen av IP formuleringen vår, og ser på dualen har vi

$$\begin{aligned} \text{Maximize:} \quad & \sum_{C \in \boldsymbol{C}} y_c & & \\ \text{subject to:} \quad & \sum_{C \in \boldsymbol{C} : i \in C} y_c \leq w_i, & & \forall i \in V \\ & y_c \geq 0 & & \forall C \in \boldsymbol{C} \end{aligned}$$

Nå har vi naturlig nok eksponensielt mange dualvariabler (siden vi hadde eksponensielt mange primalrestriksjoner), men er ikke et stort problem, ettersom vi kun trenger å øke et polynisk antall av dem). Den naturlige måten å løse dette på er å forsøke nøyaktig samme strategi som i forrige seksjon, starte med gyldig dual, også øke en y_c slik at minst en restriksjon blir stram. Samme analyse som tidligere oppnår vi da følgende approksimasjonsgrad (her betegner S løsningssettet):

$$\sum_{i \in S} w_i = \sum_{i \in S} \sum_{C : i \in C} y_c = \sum_{C \in \boldsymbol{C}} |S \cap C| y_c \tag{3}$$

Her ser vi altså at vi får en approksimasjonsgrad på $max_C(|S \cap C|)$ (maks antall sykler en node i løsningen er med i). Dessverre vil samme naive "bare øk hvilken som helst dualvariabel" strategi som tidligere føre til at dette tallet kan bli veldig høyt. Men ved litt smartere valg av variabler vil vi kunne oppnå en ganske god approksimasjonsgrad.

Løsningen er å i et hvert steg kun øke dualvariabler som tilhører sykler med mindre enn $2\lceil \log_2 n \rceil$ noder av grad 3 eller mer. Det er en del analyse bak dette, men her kommer en rask oppsummering: Vi fjerner alltid noder av grad < 2 siden disse umulig kan være del av en sykel. Stier av noder av grad 2 vil utslettes ved å velge en vilkårlig node i stien (dette gjøres iterativt siden naboer da vil ha grad < 2). Dermed kan disse stiene slåes sammen til en kant. Deretter har vi et lemma som sier at i en graf uten noder av grad 1 finnes det alltid en sykel med maks $2\lceil \log_2 n \rceil$ noder av grad 3 eller mer. Dette sees ved å se for seg et binærsøk gjennom grafen, der vi slår sammen stier av noder med grad 2. Her vil entall noder minst doble seg per nivå gjennom treet, og vi kan ha maksimal dybde på $\lceil \log_2 n \rceil$. Dermed (om det finnes minst en sykel), vil det finnes minst en sykel som også har lengde mindre enn $2\lceil \log_2 n \rceil$ (se for deg at den knyttes sammen helt i bånn av treet).

Denne strategien gir oss en $4\lceil \log_2 n \rceil$-approksimasjon. For å se dette, se først at størrelsen på sykler som "legges til" (dualvariablen økes) i hvert trinn av algoritmen ikke overskrider $2\lceil \log_2 n \rceil$, hvis vi kun teller noder av grad > 2. Teller vi bare disse nodene, så har vi $|S \cap C| \leq 2\lceil \log_2 n \rceil$. Men, det kan også være stier med noder av grad 2 i mellom hver av nodene. Uansett, fra disse stiene er maksimalt en node med (se forrige avsnitt). Dermed kan har vi $|S \cap C| \leq 4\lceil \log_2 n \rceil$, og som tidligere argumentert for gir dette en øvre grense på approksimasjonsgraden.

Dette er en viktig lekse i primal-dual algoritmen. Ofte ønsker vi å øke den "minste" (på et eller annet vis) dual-variabelen.

5.3 Korteste Vei

Mer av det samme som i forrige seksjon, det vanskeligste her er oppsettet. Hopper over det formelle, kan evt. se i boka. Her har vi primalvariabler som kanter som skal skrus av/på, mens dualvariablene er s-t kutt, dvs. $S = \{v \in V : s \in S, v \notin S\}$. Er egentlig ingen ny lekse her, men igjen øker vi alltid den y_s som tilhører den "minste" $S \in \boldsymbol{S}$. Minste her betyr det subsettet av noder vi kan nå med løsningen vi har bygget til ved den nåværende iterasjonen, og øker til en av dualrestriksjonene blir stramme. Da tar vi med den kanten i løsningen. Tilslutt, når vi når t så stripper vi vekk alt som ikke er del av s-t veien og returnerer løsningen P.

Resultatet her blir da en velkjent algoritme, Dijkstra's algoritme, som finner den optimale løsningen. Dette sees ved standard primal-dual analyse, som gir oss at dette er en $max(|P \cap \delta(S)|)$-approksimasjon, der $\delta(S) = \{e = (u, v) : u \in S, v \notin S\}$. Deretter kan man se at $|P \cap \delta(S)| = 1, \forall y_c \neq 0$, siden ellers ville vi på et punkt lagt til en sykel i løsningen vår. Dermed får vi en optimal løsning (det er jo velkjent at Dijkstra's returnerer den optimale løsningen, så dette burde vel ikke være noen bombe).

5.4 Minimum Knapsack Problem

Denne seksjoner viser et annet fenomen som kan være et hinder for approksimeringsalgoritmer. Problemet er som et slags motsatt knapsack-problem, der man ønsker å bære lavest mulig vekt, men der verdien av tingene er høyere enn en satt terskel. En naturlig formulering av IP-et gir:

$$\begin{aligned} \text{Minimize:} \quad & \sum_{i \in I} s_i x_i \\ \text{subject to:} \quad & \sum_{i \in I} v_i x_i \geq D \\ & x_i \in \{0, 1\} \quad \forall i \in I \end{aligned}$$

Her er s_i og v_i hhv. vekten og verdien til objektet $i \in I$. Dette oppsettet gir opphav til et arbitrært dårlig "Integrality gap", eller heltallsavvik. Heltallsavvik er betegnelsen på den øvre grensen for et tall α slik at $\frac{OPT}{PRIM} \geq \alpha$ (for et minimeringsproblem). I primal-dual analysen er det helt klart nødvendig at dette tallet ikke er for høyt, ettersom det gjør at løsningen på relakseringen forteller oss lite om den faktiske løsningen.

I vårt nåværende oppsett kan man gi problemet $I = \{1, 2\}$, hvor $v_1 = D - 1, v_2 = D, s_1 = 0, s_2 = 1$. Her er det lett og se at heltallsløsningen er å kun ta med objekt 2, og gir oss summen 1, mens LP-relakseringen vil ta med objekt 1, og fylle opp siste lille biten med en fraksjon av objekt 2. Dette gir summen $\frac{1}{D}$, og gir oss heltallsavvik $\frac{OPT}{PRIM} = D$.

Det er en annen lur måte å sette opp problemet på som unngår det skrekkelige heltallsavviket ved å bytte ut verdi-begrepet med "avstand" fra å fylle opp til kravet D, og gir opphav til en 2-approksimering. Se boka (s. 127)/forelesning 5 for detaljer.

6. Grådige Algoritmer og Lokale Søk

Grådige algoritmer er noe som bør være kjent (kanskje til og med som noe av det enklere) fra tidligere fag. Lokalt søk har noen likheter med grådige algoritmer. Begge har den store fordelen av at de ofte er svært enkle å implementere, og gir også ofte grunnlag for teoretisk analyse av approksimeringer.

Grådige algoritmer, formulert med språket som vi er blitt vant med nå, er algoritmer som starter med en ugyldig primal, og bygger løsningen gradvis. Alle valg som tas er bindende. Lokale søk derimot starter med en (gjerne arbitrær) gyldig primal og forsøker å opptimalisere den, mens den holder seg gyldig. Her er ingen valg bindende. Vi går i denne seksjonen raskt gjennom tre greie eksempler og analyserer approksimasjonsgraden.

6.1 Sekvensering (Parallelle Maskinjobber)

Dette er et greit problem, hvor vi har m maskiner som skal fullføre n jobber $j = 1...n$, der jobb j har prosesseringstid p_j. En maskin kan kun holde på med en jobb om gangen, og må fullføre den når den er startet på. Målet er å minimere tiden det tar å fullføre alle jobbene, omtalt som C_{max}.

Et enkelt lokalt søk er som følger: Start med et arbitrært oppsett av jobber. Deretter flytt den jobben som er ferdig sist til den maskinen som i øyeblikket er ferdig tidligst, slik at C_{max} synker. Fortsett å gjøre dette til det ikke lenger går (at jobben blir ikke blir flyttet), og output oppsettet.

For analysen av approksimasjonsgraden her trenger vi to nedre grenser for $OPT = C^*_{max}$. De to åpenbare er at $C^*_{max} \leq max_j p_j$, og at $C^*_{max} \leq \sum_j \frac{p_j}{m}$. Vi deler så inn i to intervaller for algoritmens output, før og etter siste jobb p_j begynner (vi kaller det tidspunktet s_j. Ved tidspunktet s_j har alle m maskinene jobbet konstant, og jobben j har ikke blitt startet på enda, altså har vi at $s_j \leq \sum_{i \neq j} \frac{p_i}{m}$. Videre er $p_j \leq max_i p_i$. Dermed er det ikke vanskelig å se at $C_{max} = s_j + p_j = \sum_{i \neq j} \frac{p_i}{m} + p_j = (1 - \frac{1}{m})p_j + \sum_i \frac{p_i}{m} \leq (2 - \frac{1}{m})C^*_{max}$. Altså er vårt lokale søk en $(2 - \frac{1}{m})$-approksimasjon.

For den grådige varianten av problemet, bare plasser ut jobber så snart en maskin har blitt ledig. Analysen her er nå mega-enkel, siden vi ser at når den grådige algoritmen er ferdig, vil den ha samme output-form som det lokale søke (altså hvis vi hadde forsøkt å kjøre det lokale søke på outputen til den grådige algoritmen, ville den bare ha outputta med en gang). Dermed er også dette en $(2 - \frac{1}{m})$-approksimasjon. Dette kan videre forbedres til en $\frac{4}{3}$-approksimasjon ved å sortere lista av jobber for de mates til den grådige algoritmen. Detaljer finnes i boka/forelesning 6.

6.2 Klynge-Analyse

Her er oppgaven rimelig grei, målet er å finne k senter blandt n punkter. Slike senter er da noder som er "nærmest de andre". Litt mer formelt er målet å finne de punktene som gjør at radiusen til de sirklene som har punktene i midten og omfavner alle noder i grafen blir minst mulig. Den sentrale antagelsen i problemet er at vi befinner oss i såkalte metriske rom, der grafen er komplett, og kantene har vekter som kan sees på som "distanse" mellom nodene, altså at $d_{ij} \geq 0, d_{ij} = d_{ji}$ og at $d_{ik} + d_{kj} \geq d_{ij}$ (The triangle-inequality) holder for alle noder.

Her konstruerer vi enkelt en grådig algoritme som starter med å velge en arbitrær node å legger til i løsningen S. Så, for hver iterasjon tar den den noden som ligger lengst unna alle andre noder i S og legger til i S. Dette fortsetter til $|S| = k$. Dette gir enkelt og greit en 2-approksimasjon. For å se dette, anta først at alle noder i S befinner seg i hver sin sirkel, med hensyn til $OPT = S^*$. Da ser vi med en gang at dette er en 2-approksimasjon (en radius på $2r^*$ dekker hele sirkelen uansett hvor noden er plassert). Dersom S inneholder to noder fra samme sirkel, betyr det at da det var den lengste avstanden i systemet, var maksimal avstand $\leq 2r^*$.

Det mest interessante her er at selvom det var såpass enkelt å oppnå en 2-approksimasjon, ville det å oppnå så mye som en mikroskopisk smule bedre implisere at $P = NP$. Grunnen til dette er at vi kunne brukte en $(2-\epsilon)$-approksimasjon for dette problemet til å bestemme om en graf har en dominerende mengde av k noder (der $N(S) \cup S = V, S \subseteq V, |S| = k$) , et problem som er NP komplett. For å se dette, set $d_{ij} = 1$ dersom $(i, j) \in E$, ellers set $d_{ij} = 2$. Da holder fortsatt de metriske kravende, men en strengt bedre enn 2-approksimasjon løser også dominerende set korrekt.

6.3 Metric TSP eller Traveling Salesman IRL

TSP er et velkjent problem, arguably det mest kjente av alle NP-komplette problem. Det er vist at så mye som en $\mathcal{O}(2^n)$-approksimasjon vil implisere at $P = NP$ (ikke så veldig vanskelig å se, reduser fra hamilton sykel). Men hva om vi legger på det metriske kravet fra forrige seksjon? Det er jo overhode ikke helt urimelig, ettersom det f.eks vil gjelde for TSP i "virkeligheten".

Vi lager en ganske straight-forward grådig algoritme, som starter med de to nærmeste nodene. Deretter legger den til den noden som nærmest disse to igjen, og utvider løsningen (som i Prims MSP). En fint bilde her er en strikk som strekker seg over en og en ekstra spiker, til den til slutt er innom alle noder.

Vi kan faktisk bruke Prim's MSP algoritme enda mer direkte. Dette kalles "Twice Around the Tree"-algoritmen (iallfall av Hetland...). Først, se at løsningen $MSP < OPT$, siden OPT er den minste Hamilton syklen (fjern en kant, så har du et spenntre). Så vi starter med å bruke Prim's grådige algoritme for å finne et MSP. Deretter dobler vi alle kanter, og lager en euler-krets. Løsningen vår er nå rekkefølgen nodene traverseres i euler-kretsen (dvs. hopp over noder når de besøkes for ikke-første gang). Dette gir en 2-approksimasjon, fordi ved å hoppe over noder (short-cutting) blir veien iallfall ikke lengre (siden $d_{ik} + d_{kj} \geq d_{ij}$). Videre er har vi bare doblet MSPen vår så løsningen vår $C \leq 2MSP \leq 2OPT$.

En slightly bedre algoritme har navn Christofides' algoritme. Poenget med kant-doblingen isted var å få partalls grad på alle noder. Men det er litt over-kill å doble alle nodene, så isteden ser vi kun på de nodene av odde grad. For disse nodene finner vi en perfekt matching av minimal kost (det må gå, siden det finnes et partall antall av dem, fordi summen av graden til alle nodene må være partallsgrad, siden hver kant legger på 2). Denne matchingen har da kost mindre enn $\frac{OPT}{2}$. Legger vi denne matchingen oppå MSPet vårt, så vil alle noder nå ha partalls grad, og vi kan lage en eulerkrets, og løse på samme måte som før. Vi får nå $C \leq MSP + \frac{OPT}{2} \leq \frac{3}{2}OPT$, så Christofides' algoritme er en $(\frac{3}{2})$-approksimasjon.

7. Grådighet og Matroider

Når vi forsøker å abstrahere konseptet bak den grådige algoritmer, åpner det seg en hel ny verden, som har gitt opphave til et helt forskningsfelt, nemlig matroide-teori. For å forstå hva matroider i det hele tatt handler om starter vi med et såkalt *Independent system*, eller uavhengig mengdesystem. Det består av mengde paret $(E, \boldsymbol{S})$, der E er et ikke-tomt sett, og $\boldsymbol{S} \subseteq 2^E$ er lukket under inklusjon (for $A \in \boldsymbol{S}$ og $B \subseteq A$ har vi $B \in \boldsymbol{S}$).

Dersom vi ser på $M = (E, \boldsymbol{S})$, har vi følgende ekvivalente utsagn:

- M er en matroide
- M er et uavhengighetssystem med en submodulær rangfunksjon (se polymatroider)
- for alle $J, K \in \boldsymbol{S}$ med $|J| = |K| + 1$ har vi at det eksisterer en $a \in J \setminus K$, slik at $K \cup \{a\} \in \boldsymbol{S}$
- For alle $A \subseteq E$, har alle maksimale uavhengige set i A samme kardinalitet.
- ...og mange fler

For beviser på ekvivalens se i boka (eller tenk litt, med utgangspunkt i at den grådige algoritmen alltid finner det globalt maksimale uavhengige settet i M). Merk at maksimale uavhengige set kalles også basiser.

Stort sett når vi ser på matroider antar vi at det underliggende problemet er maksimering, og at vi kun ser på positive vekter. Dette er uten tap av generalitet, siden dersom vi har en vekt-funksjon w med negative vekter, kan vi erstatte den med en annen vekt-funksjon w' der $w' = w + c$, slik at alle vekter er positive. Dette blir naturlig nok samme optimale sett som løsning, ettersom alle maksimale sett er like store ($w'(A) = w(A) + |A|c, \forall A$). Videre, om vi har et minimeringsproblem med vektfunksjon w_1, kan vi naturlig nok se på maksimering over vektfunksjonen $w_1' = -w_1$.

7.1 Polymatroider og Submodularitet

I forelesning ble matroider egentlig introdusert som polymatroider der x er binær. Jeg synes vel det var greiere å gå rett på matroider, men kan vel være verdifull innsikt for det. Polymatroider er igrunn de lineær-programmene som lar seg løse grådig. For å se hvilke egenskaper som da trengs, så må vi ha et bilde på en grådig algoritme for lineær-program. Det blir som følger: sorter først etter c (som her betegner målfunksjonen) i synkende rekkefølge deretter øk alltid den mest verdifulle x_j så mye som mulig. Så videre til kravene.

Det enkleste er å se for seg et ligningsystem på nedre triangulær form. Dersom strategien for skal fungere, må $b_i \leq b_{i+1}$ (der i betegner radene, husk at vi å for oss at restriksjonene var på nedre triangulær form). Vi kan da løse dualen på samme måte, og oppnår komplimentær slakkhet (og dermed optimalitet). Men! dessverre så har vi jo ekstremt sjeldent et ligningsystem på en slik triangulær form, så vi må opprette et krav til for restriksjonen $\boldsymbol{b}$, slik at det ikke ødelegger for strategien vår i det generelle tilfelle. Dette kravet kalles submodularitet og er som følger:

$$b(a_i \cup a_j) + b(a_i \cap a_j) \leq b_i + b_j$$

I forelesning 7 er det veldig godt og visuelt forklart hvorfor dette holder. Kort forklart fordi om vi ser for oss to restriksjoner i, j, så kan vi flytte kolonnene i lineær programmet slik at $a_i \cap a_j$ ligger først, deretter $a_i \setminus a_j$ og tilslutt $a_j \setminus a_i$. For at optimum må finnes grådig har vi at $a_j x \leq b_j$. Videre kan vi skrive $a_j x = ((a_i \cup a_j) - a_i + (a_i \cap a_j))x$. Dermed blir kravet $b(a_i \cup a_j) - b_i + b(a_i \cap a_j) \leq b_j$, som skrevet om blir submodularitetskravet.

Et visuelt bilde på en polymatroide følger. Kort forklart og muligens litt forenklert former løsningsettet til en polymatroide en n-dimensjonal konveks polytop (n er størrelsen på lineærprogrammet i antall variabler), der optimum kan nåes ved kun å vandre langs en dimensjon (dvs. øke en variabel) av gangen! Et par greie egenskaper å ta med: Dersom $\boldsymbol{b}$ er heltallig, har også polymatroiden heltallshjørner. Videre gjelder det samme for snittet av to polymatroider (dette snittet er såkalt TDI, *totally dual integral*). Mer om snitt av matroider under.

7.2 Approksimering og Snitt Over Flere Matroider

Den grådige algoritmen løser som sagt matroider optimalt. Men hva skjer om vi forsøker å benytte den grådige algoritmen over et uavhengig system som ikke nødvendigvis er en matroide? Da ønsker vi å finne en approksimasjonsgrad.

Starten på å analysere en slik approksimasjonskrad er for en mengde $A \subseteq E$ å definere en øvre rang $ur(A)$ og en nedre rang $lr(A)$ som er definert lik kardinaliteten til henholdsvis største og minste basis (merk at for en matroide er da naturlig nok $ur(A) = lr(A), \forall A \subseteq E$. Denne kalles da bare en rang, notert ved $\rho(A)$). Dette gjør analysen av approksimasjonsgraden ganske grei. For et mengdesystem M, og en $A \subseteq E$, kan det være at den grådige algoritmen finner $lr(A)$, mens opt naturlig nok er $ur(A)$. For M definerer vi rangkvtienten $rq(M) = min_A\{\frac{lr(A)}{ur(A)}\}$, og for et uvektet optimaliseringsproblem kan vi dermed få $\frac{APX}{OPT} \leq rq(M)$. Videre kan det vises ganske greit (den grådige algoritmen velger jo alltid de enkelt-elemntene med høyest mulig verdi osv., se boka/forelesning for et mer rigorøst bevis) at for et mengdesystem vil vi også alltid ha $\frac{APX}{OPT} \geq rq(M)$.

La oss se for oss et mengdesystem $M = (E, S)$, der $S = \cap_{i=1}^{k} S_i$, og der $M_i = (E, S_i)$ er matroider. Dette kalles maksimering over snittet av k matroider. Det kan løses eksakt for $k = 2$ i polynomisk tid, men er NP-hardt for $k \geq 3$. Men, vi kan naturlig nok bruke grådighet som approksimasjon. Som vi har sett, fungerer rangkvotienten som en god nedre grense for approksimasjonsgraden, og det kan vises at $rq(M) \geq \frac{1}{k}$. Dette beviset kan sees i boka/forelesning 7.

7.3 Eksempler

Denne seksjonen har vært ganske "der ute" så langt, men med litt praktiske tilfeller og mye teori i boks, kan vi se effekten av arbeidet her i denne seksjonen. Vi starter med å se på et par ganske generelle matroider (de som er sett på i pensum):

- **Grafmatroiden:** E er kanter i en urettet graf, $\boldsymbol{S}$ er sett av kanter uten sykler.
- **Hale- og hodepartisjonsmatroider:** E er kanter i en rettet graf, $\boldsymbol{S}$ er sett av kanter med ulike start- og sluttnoder.

- **Vektormatroider:** E er en mengde med vektorer, $\boldsymbol{S}$ er lineært uavhengige mengder (dette er den ikke-algoritmiske inngangen til matroideteori).
- **Uniforme k-matroider:** E er kanter i en mengde, $\boldsymbol{S}$ er delmengder opp til kardinalitet k.
- **Transversalmatroider:** E er venstrenoder i en bipartitt graf, $\boldsymbol{S}$ er sett av noder som kan matches samtidig.

Ettersom disse mengdesystemene er matroider, vet vi nå at alle disse maksimeringsproblemene kan løses optimalt med den grådige algoritmen (eksempelvis er den grådige algoritmen over grafmatroiden enkelt og greit Prim's algoritme med negativ vekt-funksjon).

Videre kan vi endelig forstå den fantastiske setningen fra seksjon 1! Snittet av to transversal-matroider er naturlig nok mere kjent som problemet matching i en bipartitt graf (en matroide for hver side av den bipartitte grafen). Dermed vet vi at den grådige algoritmen løser bipartitt matching med approksimasjonsgrad $\alpha = \frac{1}{2}$. Videre vet vi også at det kan løses eksakt i polynomisk tid, siden det er snittet over $k = 2$ matroider. Ellers kan vi egentlig like lett finne approksimasjonsgraden til den grådige algoritmen for matching i generelle grafer. Ved å starte med å gi enhver kant i grafen en vilkårlig retning, ser vi at snittet av hale- og hodepartisjonsmatroider for grafen vil tilsvare matchinger i den original grafen. Dermed vet vi at den grådige algoritmen faktisk løser generelle matchinger med approksimasjonsgrad $\alpha = \frac{1}{2}$.

Et annet eksempel er spenntrær i en rettet graf. Det kan modelleres som snittet av grafmatroiden og hode- ELLER halepartisjoner, og viser oss at dette problemet kan løses i polynomisk tid, selvom den grådige algoritmen ikke gir optimal løsning. Videre kan vi betrakte snittet av grafmatroiden og hode- OG halepartisjoner, og med litt tenking ser vi fort at de maksimale uavhengige mengdene her må tilsvare hamiltonstier. Dette viser at optimalisering over snittet av $k \geq 3$ matroier er NP-hardt, men at den grådige algoritmen løser det tilsvarende optimaliseringsproblemet med approksimasjonsgrad $\alpha = \frac{1}{3}$. Så mye approksimeringsteori! Og så mye enklere enn tidligere nå som vi kan anvende matroider! Fantastisk.

7.4 "Greedoids"

Helt til slutt kan det nevnes at kravet om inklusjon i uavhengige mengdesystemer faktisk er litt for strengt i forhold til hva som trengs for å kunne løse et optimaliseringsproblem med den grådige algoritmen. Dermed kan vi bytte ut det kravet heller med kravet at dersom $A \in \boldsymbol{S}$, så finnes det alltid en $e \in A$ slik at $A \setminus \{e\} \in \boldsymbol{S}$. Legg merke til at dette kravet, kalt tilgjengelighet, er mye mindre strengt enn kravet om inklusjon. Et slikt tilgjengelig system er en "greedoid" (en generalisering av matroide) hvis og bare hvis kravet for matroider i uavhengige mengdesystemer "for alle $J, K \in \boldsymbol{S}$ med $|J| = |K| + 1$ har vi at det eksisterer en $a \in J \setminus K$, slik at $K \cup \{a\} \in \boldsymbol{S}$" utvides til at også $J \setminus \{a\} \in \boldsymbol{S}$ (i uavhengige mengdesystemer var dette ikke et nødvendig krav, siden for alle $A \subseteq J$ hadde vi $A \in \boldsymbol{S}$). Legg også merke til at $\varnothing \in \boldsymbol{S}$ for alle tilgjengelige systemer så vel som alle uavhengige mengdesystemer.

8. Flyt

Flyt brukes mye i modellering, og har vist seg å være en av de viktigste teori-områdene for å modellere praktiske problemer. Vi starter med litt bakgrunn som vanlig, og så ser vi på litt praktiske problemer som kan løses ved flyt. Flyt burde være litt kjent fra algdat. Videre har vi alt sett at flyt som et lineærprogram har tilhørende dual minste s-t kutt (det motsatte av korteste vei problemet).

8.1 Bakgrunn

Et flytnettverk er en graf $G = (V, A)$, med kapasiteteter på kantene $u(i, j)$, samt to markerte noder $s, t \in V$, en kilde s og et sluk t. Videre må flyten følge to krav, nemlig kravet "capacity constraint", at $0 \leq f(i, j) \leq u(i, j), \forall (i, j) \in A$, samt kravet "flow conservation", at $\sum_{k:(i,k)\in A} f(i, k) = \sum_{k:(k,i)\in A} f(k, i), \forall i \neq s, t$. Verdien på flyten $|f|$ er da $|f| = \sum_{k:(s,k)\in A} f(s, k) - \sum_{k:(k,s)\in A} f(k, s)$ (flyt ut av kilden minus flyt inn til kilden), eller ekvivalent (pga. flow conservation) $\sum_{k:(k,t)\in A} f(k, t)$. Målet er da generelt å finne størst mulig $|f|$.

Mye av den teoretisk analysen baserer seg på såkalt "skew symmetry", hvor vi $\forall (i, j) \in A$ setter $f(j, i) = -f(i, j), u(j, i) = 0$. Det er lett å se at vi fremdeles opprettholder kravene om flyt når vi legger på skew symmetry, om vi tillatter negativ flyt. Faktisk kan vi nå sette definere flow conservation enda enklere som $\forall i \neq s, t, \sum_{k:(i,k)\in A} f(i, k) = 0$. Tilsvarende setter vi $|f| = \sum_{k:(s,k)\in A} f(s, k)$.

Vi har tidligere vært innom notasjonen $\delta(S)$ for et s-t kutt S. Som nevnt er minste mulige $\delta(S)$ dualen til lineærprogrammet for flytnettverk, og dermed får vi med en gang at $|f| = min_S \delta(S)$. Det kan også vises direkte uten kunnskap om lineærprogram ganske greit, se forelesning eller boka. Det direkte beviset viser i første omgang at $|f| \leq \delta(S), \forall S$. For likhet må vi definere residual grafen. Den er definert for en flyt f på en graf $G = (V, A)$ med kapasiteter $u(i, j)$ som $G_f = (V, A)$, der $u_f(i, j) = u(i, j) - f(i, j)$. Dersom det finnes en path P på G_f fra s til t kalles dette en augmenting path, og eksistensen av en slik P impliserer at $|f|$ ikke er maksimal (fordi vi kan da dytte $min_{(i,j)\in P} u_f(i, j)$ enheter langs P). Med denne forkunskapen kan vi bevise at følgende uttrykk er ekvivalente:

- f er en maksimal flyt
- Det finnes ingen augmenting path P i G_f
- $|f| = u(\delta(S))$ for et min. s-t kutt S

Bevisene her er ikke så ille, se evt. boka/forelesning. En til egenskap som er viktig å nevne er heltalls-egenskapen ved flyt, som sier at dersom alle kapasiteter $u(i, j)$ er heltallige, vil også den maksimale f være heltallig. Dette er kjempenyttig, siden da holder det alltid å vise at en ikke-heltallig f' eksisterer, og dermed eksisterer en heltallig f med $|f| \geq |f'|$. Dette er ikke en veldig vanskelig egenskap å se, kan f.eks vises ved å finne den maksimale f ved å kun bruke augmenting paths (disse må jo alltid være heltallige).

8.2 Praktiske Eksempler

Vi ser raskt (litt slurvete) på tre praktiske problemer som kan modelleres med flyt, og håper at disse inspirerer til å løse lignende oppgaver på eksamen.

8.2.1 Carpool Driver

Over en periode på n dager skal m forskjellige mennesker sette opp en carpool-kalender. Hver av dagene er k folk med i bilen. Tanken er da at disse er ansvarlige for å kjøre $\frac{1}{k}$-del av bilen. La r_i være det totale ansvaret person i fått på seg. Målet er at person i skal måtte kjøre maksimalt $\lceil r_i \rceil$ ganger. Løses ved å sette opp en rad med noder for alle personene, og kanter fra s til person i med kapasitet $\lceil r_i \rceil$. Videre, sett opp en rad med noder for alle dagene, og en kant fra hver person til de dagene den personen skal sitte i bilen med ubegrenset kapasitet. Tilslutt sett kanter fra dagene til t med kapasitet 1.

En liten sidenote her er at dette alltid er løslig, siden det er helt trivielt løslig fraksjonelt, og pga. heltallsegenskapen (ubegrenset kapasitet kan bestemmes å være heltallig) får vi at dette er heltallig-løslig.

8.2.2 Baseball Elimination Problem

Litt mye notasjon å skrive opp her som kun er relevant for akkurat dette problemet, så outliner det heller litt grovt, og overlater beviser til boka. Vi ønsker å "eliminere" lag fra en baseball-liga (eliminere betyr i denne settingen at de ikke lenger kan vinne), og setter det opp ved å lage et flytnetthverk for hvert lag k vi ønsker å skjekke om er eliminert. Det kan gjøres ved å sette en rad med noder som representerer alle de forskjellige parene av lag i ligaen utenom k, og en kant fra s til disse med kapasitet "gjenværende kamper mellom disse to". Deretter en kant fra dette paret til hvert av lagene i paret med uendelig kapasitet. Til sist, en kant fra alle lag i til t med kapasitet "antall mulige poeng for k (seiere + gjenværende spill) minus seiere for lag i". Dersom $\{s\}$ nå er et min s-t kutt, er ikke laget eliminert. En grei intuisjon for dette er at vi klarer å dytte en flyt tilsvarende gjenstående kamper mellom alle andre lag enn k gjennom nettverket uten at noen får flere seiere enn k teoretisk kan få.

8.2.3 Maximum Density Subgraph

Ganske generelt graf-problem, gitt en graf $G = (V, E)$, finn det settet $S \subseteq V$ slik at grafen indusert av S, $G(S) = (V, E(S))$ har høyest mulig "tetthet", $D = \frac{|E(S)|}{|S|}$. Det nøyaktige oppsettet her leder til litt knotete notasjon og en del bokføring for bevis av korrekthet osv., så hopper greit over det, kan evt. se i boken/forelesning. Men heller er det kule konseptet her idéen om å bruke maks-flyt + binærsøk for å finne svaret, når maks-flyt modellen kan brukes som et orakel for å skjekke om en parameter er en gyldig løsning. Litt mer formelt, om D^* er den maksimale tettheten, vet vi at D^* er i intervallet $[0, m], m = |E(S)|$. Deretter kan sette $\gamma = \frac{(l+u)}{2}$, der l, u betegner nedre og øvre grense i intervallet, og benytte maks-flyt oppsettet vårt til å skjekke om γ er lavere enn den maksimale tettheten. Hvis den er lavere blir det nye intervallet $[\gamma, u]$, ellers blir det $[l, \gamma]$. Deretter, siden D^* er rasjonal, med begrenset størrelse på nevneren (maksimal størrelse er jo $n = |V|$), vet vi at ved å utføre

binærsøket et endelig antall ganger ($\mathcal{O}(log(n))$) har vi begrenset intervallet til kun ett mulig tilfelle.

8.2.4 Project Planning

Oppgaven er tatt fra øving 4. Her har vi tre prosjekter, P1, P2 og P3 som krever så så mange "arbeidsmåneder" å fullføre, og som bare kan utføres under visse måneder, 4 måneder å utføre dem på og 8 arbeidere. Videre kan kun 6 arbeidere jobbe på hvert prosjekt pr. måned. Vi setter ganske greit opp problemet ved å ha kanter fra s til arbeiderne med uendelig kapasitet. Deretter en kant fra arbeiderne til hver måned med kapasitet 1. Så en kant fra hver måned til hvert prosjekt, tilsvarende de månedene prosjektene kan utføres på, alle med kapasitet 6. Til slutt en kant fra alle prosjektene til t med kapasitet tilsvarende antall arbeidsmåneder det krever å fullføre prosjektet.

8.2.5 k-Orientering

Tatt fra konteksamen 2018. Oppgaven lyder som følger "En orientering av en urettet graf $G = (V, E)$ er en tilordning av retning til hver kant $e \in E$, som resulterer i en ny rettet graf $D = (V, A)$, der hver urettet kant $(u, v) \in E$ tilsvarer en rettet kant enten $(u, v) \in A$ eller $(v, u) \in A$. Videre er en k-orientering en orientering der hver node v har maksimalt $k(v)$ inn-kanter, for en funksjon $k : V \longrightarrow \mathbb{N}$. Hvordan kan du effektivt finne en k-orientering v.hj.a. maks flyt?".

Dette løses ganske greit ved å sette opp en rad med noder for hver kant $e \in E$, og en kant fra s til denne raden med kapasitet 1. Deretter sett opp en rad med noder for hver node $v \in V$, og legg til en kant mellom disse radene slik at hver kant kobles med sine ende-punkter i G. Disse kantene kan ha kapistet 1 (eller uendelig, spiller ingen rolle). Til slutt sett opp en kant fra hver node v til t med kapasitet $k(v)$.

8.3 Most Improving / Shortest Augmenting Path

Så langt har vi jo faktisk ikke gått veldig inn på noen algoritme for å løse flyt-problemet, selvom vi i introduksjonen hintet litt til å bruke disse augmenting path'ene. Problemet her er at ved å ta en tilfeldig augmenting path, får vi ikke en polynomisk kjøretid. Hver augmenting path øker flyten med minimum 1, så vi trenger maksimalt $\mathcal{O}(mU)$ slike paths, hvor $U = max(u(i, j))$. En kjøretid på $\mathcal{O}(mU)$ er kun **pseudo-polynomisk**, siden numerisk data, i dette tilfelle $u(i, j)$, ser vi på som binær input. Med andre ord trenger vi $log(U)$ bits for å inpute U, og kjøretiden blir dermed eksponensiell. Vi viser straks en algoritme som har polynomisk kjøretid. Videre er det mulig å oppnå det som kalles **Strongly polynomial**, dvs. at kjøretiden kun er avhengig av antall elementer i inputen, og ikke størrelsen av dem (vi ser på det og).

En naturlig idé er å finne den mest forbedrene ("most improving") augmenting pathen i G_f. For å analysere en slik algoritme, bruker vi veldig grei teori om flyt-dekomponering, $f = f' + f''$. Det er lett å bevise at summen av to flyt $f' og f''$ blir en ny flyt som overholder både capacity constraint og flow conservation, og at $|f| = |f'| + |f''|$. Se boka/forelesning for detaljer. Videre har vi at viktig lemma som sier at for enhver s-t flyt f så finnes det flyter $f_1, f_2, ..., f_l, l \leq m$ slik at $f = \sum_{i=1}^{l} f_i$, hvor for alle i, har vi at de kantene med positiv

flyt i f_i enten danner en s-t path eller en sykel. Dette lemma kan vises med induksjon. Videre følger ganske naturlig gitt en maksimal flyt f^* og en hvilken som helst s-t flyt f, så har den maksimal flyten i G_f verdi $|f^*| - |f|$, og videre fra det første lemma kan denne maks-flyten i G_f bygges av augmenting paths, der den most improving augmenting pathen har verdi minst $\frac{1}{m}(|f^*| - |f|)$. Dermed ved å alltid velge den "most improving" får vi etter k iterasjoner $|f^*| - |f|^{(k)} \leq (1 - \frac{1}{m})^k(|f^*| - |f|)$, som etter $k = mlog(mU)$ vil gi oss maks flyten, altså har vi en polynomisk kjøretid (f.eks. gitt en ganske naiv most improving path algoritme med $\mathcal{O}(m^2)$ kjøretid, oppnår vi en total kjøretid på $\mathcal{O}(m^3log(mU)$. Denne kan forbedres litt ved enten en bedre "most improving" algoritme, eller å istedenfor å alltid finne den most improving augmenting path'en, nøyer vi oss med en path som er "bra nok", eller som har verdi mer enn Δ.

Som lovt er det og mulig å oppnå en strongly polynomial kjøretid, altså helt uavhengig av U. Ideen er veldig lite revolusjonerende, men analysen blir veldig bra. Vi velger isteden alltid den korteste augmenting pathen. Analysen på kjøretiden her går på å se på at hver kant kan kun bli saturert ($f(i,j) = u(i,j)$) $\mathcal{O}(n)$ ganger (dette gjøres ved å sette på distanse labels osv. osv., se neste seksjon for mer info), og videre siden det kun eksisterer m kanter får vi at vi må finne $\mathcal{O}(mn)$ slike paths. Korteste vei kan finnes i $\mathcal{O}(m)$ tid (ikke trivielt!), så totalt får vi da en kjøretid på $\mathcal{O}(m^2n)$.

9. Preflyt

De forrige algoritmene brukte på et eller annet vis alle augmenting paths for å finne maksimal flyt. Men ved kun å bruke slike paths kan vi få ganske dårlig kjøretid ved å konstruere spesifikke eksempler (der vi trenger mange augmenting paths, som alle kun øker flyten med 1). I denne seksjonen skal vi se en litt annen maks-flyt variant, som unngår dette problemet, og som er en av de raskere maks-flyt algoritmene idag, både teoretisk og praktisk.

9.1 Push-Relabel

Den algoritmen baserer seg på distance labels (så vidt nevnt i slutten av forrige seksjon), og såkalt preflyt. En preflyt har samme krav om capacity constraint samt skew symmetry fra flyt, men har et løsere krav enn flow conservation: nemlig at for alle $i \in V, i \neq s$ så har vi $\sum_{k:(k,i)\in A} f(k,i) \geq 0$ (sagt med ord at det kommer mer flyt inn enn det går ut av hver node, husk vi har skew symmetry). Vi kaller denne mengden *excess at i for a preflow f*, og skriver $e_f(i) = \sum_{k:(k,i)\in A} f(k,i)$. Legg også merke til at en flyt er og en gyldig preflyt.

Videre setter vi på distance labels $d(i), \forall i \in V$. Disse representerer på et vis en nedre grense for distansen til sluket. En intuisjon som blir nyttig etterhvert er og at de representerer "høyden" til node i. Vi har en gyldig distance labeling om følgende krav er oppfylt:

- $d(s) = n$
- $d(t) = 0$
- $d(i) \leq d(j) + 1, \forall (i,j) \in A_f$(residualgrafen)

Videre har vi at for en preflyt f og en valid distance labeling d, finnes ingen augmenting paths i G_f, der vi kun får gå over en kant der $d(i) = d(j) + 1$. For å se dette, anta at det finnes en s-t path P i G_f (altså en augmenting path). P inneholder maks n noder, og dermed maks $n-1$ kanter. Da får vi $d(s) = d(t) + |P| \leq 0 + n - 1 = n - 1$. Men $d(s) = n$, så dette er en kontradiksjon.

Dette gir oss alt vi trenger for å outline push-relabel algoritmen. Vi starter med en valid distance labeling og en preflyt. Deretter forsøker vi gradvis å gjøre om preflyten til en flyt, mens vi holder distance labelingen valid. Dette resulterer da i en maks flyt på grunn av resultatet over.

Litt mer spesifikt ønsker vi å dytte så mye *excess at i* mot sluket, over de kantene som er på den *tilsynelatende* korteste veien. Med det menes at vi kun dytter flyt over en kant (i,j), dersom $d(i) = d(j) + 1$ (om dette og $u_f(i,j) > 0$ holder, kaller vi kanten admissable). Når vi dytter dytter vi alltid mest mulig, dvs. $min(e_f(i), u_f(i,j))$. Om vi ikke har noen admissable kanter for en node i med $e_f(i) > 0$, da tolker vi det som at labelen er feil, og gjør en relabel. Mer spesifikt øker vi $d(i)$ til $d(i) = min_{j:(i,j)\in A_f}(d(j) + 1)$, dvs. den laveste j som i har residual-flyt til. Merk at en slik j alltid finnes, pga. skew symmetry kan vi alltids dytte flyt tilbake igjen. Her er $d(i)$ som høyde en god intuisjon, der flyt alltid går nedover.

Om vi nå initialiserer push-relabel algoritmen vår som $d(s) = n, d(i) = 0, i \neq s$, og setter $f(s,k) = u(s,k), \forall k$, så ser vi at algoritmen holder en gyldig preflyt og en valid distance labeling hele veien. Dette er ikke vanskelig å se, se evt. boka/forelesning. Det er og lett å

se at når algoritmen stopper (da $e_f(i) = 0, \forall i \neq s$) har vi en flyt, og som argumentert for tidligere er den maksimal.

9.2 Kjøretid

Analysen av kjøretiden baserer seg hovedsaklig på to resultater. Den ene sier at for enhver preflyt f, for en node $i, e_f(i) > 0$, så finnes det alltid en path fra i til s i G_f. Det andre resultatet er at for en node i har vi $d(i) \leq 2n - 1$. Dette fordi at for at vi skal relable så må $e_f(i) > 0$, og da finnes det en path P til s. Dette gir at $d(i) \leq d(s) + |P| \leq 2n - 1$.

Se nå på antall relabel operasjoner. På starten har vi $d(i) = 0, \forall i \neq s$. Videre er som vist alltid $d(i) \leq 2n - 1$. Hver relable øker $d(i)$ med minst 1. Siden vi aldri endrer på s, t har vi $n - 2$ noder som kan relables. Da får vi et maks antall relable operasjoner som $(n-2)(2n-1) = \mathcal{O}(n^2)$. Analysen av antall pushes er hakket mer tricky, men fortsatt greit. Vi deler inn i saturerende ($u(i, j)$ blir dyttet) og ikke-saturerende (mindre enn $u(i, j)$ blir dyttet) pushes. Samme som for i shortest path har vi en øvre grense på $\mathcal{O}(mn)$ ganger å saturere kanter. For å se det, se at for at en path (i, j) skal satureres 2 ganger, må $d(i)$ øke med minst to, men vi har og $d(i) \leq 2n - 1$. Analysen av antall ikke-saturerende pushes er smart. Vi konstruerer en funksjon $\phi = \sum_{iactive} d(i)$. Den slutter som 0, og er aldri negativ. Det sentrale er at et ikke-saturerende push senker ϕ med minst 1. For å se dette, se at selv om pushet gjør j aktiv hadde vi jo $d(i) = d(j) + 1$ (ikke-saturerende betyr jo at vi dytter $e_f(i)$, så i blir jo nødvendigvis inaktiv). Dermed blir en øvre grense for ikke-saturerende dytt gitt av hvor mye ϕ kan øke med under algoritmen. Ved relabeling kan vi øke ϕ med $\mathcal{O}(n^2)$, siden vi har $n - 2$ noder som kan økes med $2n - 1$. Videre vil også saturerende push øke ϕ, med opp til $2n - 1$ (i blir ikke nødvendigvis inaktiv, mens j kan bli aktiv). Som vi har sett kan vi ha $\mathcal{O}(mn)$ slike dytt og vi får en øvre grense for saturerende dytt på $\mathcal{O}(n^2m)$ Dermed blir også en øvre grense for ikke saturerende dytt $\mathcal{O}(n^2m)$. Alle operasjoner tar $\mathcal{O}(1)$ tid, så total kjøretid på push-relabel algoritmen blir dermed også $\mathcal{O}(n^2m)$.

9.3 Forbedringer

Vi har en forbedrert versjon av push-relabel kalt highest label push-relabel, som forbedrer kjøretiden til $\mathcal{O}(n^2\sqrt{m})$, der vi alltid dytter fra den aktive noden i med høyest $d(i)$. Kjøretidsanalysen her er hakket mer teknisk, men går frem stort sett på likt vis som i forrige avsnitt.

Vi har også små forbedringer, som ikke forbedrer den teoretiske kjøretiden, men som fremdeles kan bety mye for den praktiske kjøretiden. Det første er at vi tidlig kan finne $|f|$ uten å vite f (f betegner her den maksimale flyten). Den baserer seg på å redefinere aktive noder til kun å være aktive hvis de også har $d(i) < n$. Dette gir $|f|$ på grunn av et resultat som sier at "om vi terminererer push-relabel algoritmen når $e_f(i) > 0$ impliserer $d(i) \geq n, i \neq t$, da er settet S av noder som ikke kan nå t i A_f et min s-t cut". Se detaljer i boka/forelesning. Et annet raskt triks da er "gap relabeling", som sier at om det opstår et gap i distance labelsene, typ en $k < max_i(d(i))$, så setter vi $d(j) = n, \forall j : k < d(j) < n$. Et siste triks som eksperimentelt har vist seg å være lurt å gjøre av og til (ca. hver n relabel) er å regne faktisk distanse til t for alle noder, og relable $d(i)$ til den korrekte distansen.

10. Multiflyt og Multiplikativ Vektoppdatering (MVO)

Multiflyt er en naturlig vei videre fra flyt, men som viser seg å være betydelig vanskeligere. Mer spesifikt mister vi mange av de kjære egenskapene som integralitet (heltallsegenskapen) og maks flyt min kutt teoremet. Multiflyt problemet er formulert som følger: Vi har en rettet graf $G = (V, A)$ med kapasiteter $u(i, j)$, samt K kilde-sluk par $s_1 - t_1, ..., s_k - t_k$. Disse kan være overlappende (f.eks $s_i = t_j$). I noen tilfeller har vi også K demands til de K flytene notert som $d_1, ..., d_k$. Målet er da å finne K flyter $f_1, ...f_k$ slik at $\sum_{k=1}^{K} f_k(i, j) \leq u(i, j), \forall (i, j) \in A$ (kravet for hver av flytene hver for seg er det samme som tidligere). Målet kan være forskjellig, f.eks finne maks total flyt, eller bare oppnå $|f_k| \geq d_k, \forall k$.

10.1 LP Formulering

En naturlig (og helt gyldig) LP formulering av multiflyt problemet er som følger:

$$
\begin{aligned}
\text{Maximize: } & \sum_{k=1}^{K} \Big(\sum_{j:(s_k,j)\in A} f_k(s_k, j) - \sum_{j:(j,s_k)\in A} f_k(j, s_k) \Big) & & \\
\text{subject to: } & \sum_{j:j,i)\in A} f_k(j, i) - \sum_{j:i,j)\in A} f_k(i, j) & = 0, & \quad K = 1, ..., K, i \neq s_k, t_k \\
& f_k(i, j) & \geq 0, & \quad K = 1, ..., K, (i, j) \in A \\
& \sum_{k=1}^{K} f_k(i, j) & \leq u(i, j) & \quad \forall (i, j) \in A
\end{aligned}
$$

Men vi ønsker å se på en annen formulering også, blandt annet for Garg-Könemann algoritmen (se siste seksjon i dette kapitellet). Her har vi en notasjon $\mathcal{P}_k$ som betegner settet av alle $s_k - t_k$ paths i G. Videre har vi $\mathcal{P} = \cup_k \mathcal{P}_k$. Da vet vi fra tidligere kapitler at en flyt f_k kan dekomponeres til flyter langs $P \in \mathcal{P}_k$. Vekten av denne flyten skriver vi som $x(P)$. Da får vi:

$$
\begin{aligned}
\text{Maximize: } & \sum_{P\in\mathcal{P}} x(P) & & \\
\text{subject to: } & \sum_{P:(i,j)\in P} x(P) & \leq u(i, j), & \quad \forall (i, j) \in A \\
& x(P) & \geq 0, & \quad \forall P \in \mathcal{P}
\end{aligned}
$$

Og tar vi dualen her får vi:

$$
\begin{aligned}
\text{Minimize: } & \sum_{(i,j)\in A} u(i, j) & l(i, j) & \\
\text{subject to: } & \sum_{(i,j)\in P} & l(i, j) & \geq 1, \quad \forall P \in \mathcal{P} \\
& & l(i, j) & \geq 0, \quad \forall (i, j) \in A
\end{aligned}
$$

Her kan variablen $l(i, j)$ tolkes som lengden på kanten (i, j). Da sier dualen at lengeen på hver path P må være lengre enn 1, men samtidig at vi ønsker å minimere lengden på

paths som har mye kapasitet. Dette er som nevnt nyttig for forståelsen av Garg-Könemann algoritmen.

10.2 Cut Conditions og $K = 2$

Vi skulle gjerne brukt kutt til å avgjøre om vi kan møte demands, men som nevnt bryter maks flyt min kutt teoremet sammen for multiflyt. Ignorerer vi det et øyeblikk, kunne vi gjerne ha formulert en cut condition som $\sum_{k:s_k \in S, t_k \notin S} d_k \leq u(\delta^+(S)), \forall S \subseteq V$ ($\delta^+(S)$ betegner de utgående kantene av s-t kuttet S). For $K = 1$ er det naturlig at dette impliserer at vi kan møte demands, pga. maks flyt min kutt teoremet. Men, det viser seg at med en liten modifikasjon kan vi faktisk brukte dette når $K = 2$ også (enkle eksempler på hvorfor det ikke fungerer direkte kan konstrueres ganske lett).

Det første vi må gjøre er å sette opp skew symmetry igjen. Men denne gangen setter vi og $u(i, j) = u(j, i), \forall (i, j) \in A$. For å hindre at skew symmtery gjør at to flows kan utligne hverandre setter vi nå opp capacity constraint som $\sum_{k=1}^{K} |f_k(i, j)| \leq u(i, j)$. På grunn av skew symmetry og absoluttverdier påvirker nå $u(\delta^+(S))$ både s-t kutt og "t-s kutt", så vi må endre slightly på cut conditionen vår og får isteden kravet $\sum_{k:|S \cap \{s_k, t_k\}=1|} d_k \leq u(\delta^+(S))$. Og det viser seg at om $K = 2$ så er denne cut conditionen nok til å garantere at $|f_k| \geq d_k, \forall k$. Faktisk gir det også et resultat til om at dersom alle kapasiterer $u(i, j)$ er heltall har vi halv-integralitet, dvs. at for alle flyter f_k har vi $2|f_k| \in \mathbb{Z}$. Dette følger direkte fra beviset om at denne cut-conditionen er nok. Beviset er litt mye å skrive opp, men det går på å definere to nye grafer med en super-kilde s kobla til hver av s_1, s_2, og et super-sluk t koble til t_1, t_2. Deretter antar man at denne cut conditionen holder, også ser man at flytene klarer demands d_1, d_2.

10.3 Multiplikativ Vektsoppdatering (MVO) og Packing Problems

Dette problemet formulert som følger: Vi har T tidssteg, og for hvert steg $t = 1, ..., T$, velg en av N forskjellige verdier. For alle $i \in \{1, .., N\}$ verdiene og $t \in \{1, .., T\}$ har vi verdier $v_t(i) \in [0, 1]$. Problemet er at vi ved tid t kun vet de verdiene fra tidssteg $0, ..., t - 1$. Målet er å velge best mulig hele veien, og med MVO får vi et resultat som er nesten like bra som å velge en den beste fixed $j \in \{1, .., N\}$, dvs. $max_j \sum_{t=1}^{T} v_t(j)$.

Algoritmen fungerer ved å ha et set vekter $w(i)$ for hver i, som oppdateres under hvert tidssteg ($w_1(i) = 1, \forall i$). Ved hvert tidssteg t velger vi i med sansynlighet $p_t(i) = \frac{w_t(i)}{W_t}$ der $W_t = \sum_{i=1}^{N} w_t(i)$. Etter et valg er tatt får vi vite alle $v_t(i)$, så vi oppdaterervektene $w_{t+1}(i) = w_t(i)(1 + \epsilon v_t(i)), \forall i$. Enkel sannsynlighet gir da at forventningsverdien til det algoritmen returnerer blir $\sum_{t=1}^{T} \sum_{i=1}^{N} v_t(i) p_t(i)$.

Vi får et resultat som sier at $\sum_{t=1}^{T} \sum_{i=1}^{N} v_t(i) p_t(i) \geq (1 - \epsilon) \sum_{t=1}^{T} v_t(j) - \frac{1}{\epsilon} ln(N)$ for $\epsilon \leq \frac{1}{2}$. Dette finner vi ved å finne øvre å nedre grenser for det første uttrykket, og så trikse litt med ligningene. Ikke spesielt vanskelig å forstå, men ganske mye å skrive, så se evt. boka/forelesning. Det sentrale å merke seg er at denne grensen bryr seg ikke om fordelinger eller lignende. Dermed holder denne grense selv om vi velger verdier "adversarily".

Denne algoritmen kan brukes til mye, blandt annet multiflyt. Men før vi ser på det ser vi på et annet eksempel litt raskt, kalt et "Packing Problem". Kort sagt har vi et system ulikheter der vi ønsker å finne løsninger som er så riktig som mulig. Litt mer formelt har

vi et system

$$Mx \leq e, x \in \mathcal{Q} \tag{4}$$

Der M betegner en mxn matrise, e er en vektor med kun enere, og $\mathcal{Q}$ er et conveks set. Videre har vi $Mx \geq 0, \forall x \in \mathcal{Q}$. Tror ikke det er nødvendig å gå inn i for mye detalj her, men ideen er at vi indekserer $x_i \in \mathcal{Q}$, og oppretter en p_t vektor i hvert steg som består av sansynligheter $p_t(i)$. Vi finner så en $x_t \in \mathcal{Q}$ slik at $p_t^T M x_t \leq p_t^T e$, og oppdaterer vektene $w_{t+1}(j)$ slik at de radene j som bryter restriksjonen $M_j x \leq 1$ for mer vekt. Til slutt returnerer vi $x^* = \frac{1}{T}\sum_{t=1}^{T} x_t$ (legg merke til at det er her vi bruker at $\mathcal{Q}$ er konveks). Ved å sette $T = \frac{\rho}{\epsilon^2}$, der $\rho = max_i, x(M_i x)$, eller den øvre grensen for hvor mye vi kan klare å bryter en restriksjon, får vi et resultat som sier at for $\epsilon \leq \frac{1}{2}$, finner algoritmen en løsning $x^* \in \mathcal{Q}$, slik at $Mx^* \leq (1 + 4\epsilon)e$, med kjøretid på $\mathcal{O}(\frac{m\rho}{\epsilon^2} ln(m))$, pluss $\mathcal{O}(\frac{\rho}{\epsilon^2} ln(m))$ matriseprodukt utregninger og $\mathcal{O}(\frac{\rho}{\epsilon^2} ln(m))$ søk etter $x_t \in \mathcal{Q}$. Se boka for detaljer/bevis.

10.4 Garg-Könemann Algoritmen

Denne algoritmen baserer seg på MVO, samt LP formuleringen nevnt på starten av kapitellet. Idéen baserer seg på å bruke en vektsoppdatering som gjør at de kantene som har høy "congestion" (dvs. mye flyt i forhold til kapasitet), får mindre sjanse for å bli valgt i hver iterasjon. Den initialiseres med $x(P) = 0, \forall P \in \mathcal{P}$, dvs. ingen flyt, samt vekter $w(i,j) = 1, \forall (i,j) \in A$. Så finner den i hver iterasjon den korteste $P \in \mathcal{P}$, der lengden på hver kant $l(i,j) = \frac{w(i,j)}{u(i,j)}$. Deretter øker den flyten langs denne med $u = min(u(i,j))$ langs P. Legg her merke til at dette kan naturlig nok overskrive capacity constraints (vi brukte $u(i,j)$, ikke $u_f(i,j)$). Dette øker både $x(P)$ og $f(i,j), \forall (i,j) \in P$ med u. deretter oppdateres vekter til $w(i,j)(1 + \epsilon \frac{u}{u(i,j)}), \forall (i,j) \in P$ (altså øker den også kun vekten til de kantene langs pathen vi nettopp økte flyten. Dette gjøres til det finnes en kant slik at $\frac{f(i,j)}{u(i,j)} < \frac{ln(m)}{e^2}$ Løsningen vår x' er nå åpenbart ikke basert på en gyldig flyt, men merk at det kun er capacity constraints som er brutt. Dermed trenger vi bare å skalerere løsningen vår ned. Mer formelt finner vi $C = max(\frac{f(i,j)}{u(i,j)})$, og returnerer $x = \frac{x'}{C}$.

Resultatet vi får er at Garg-Könemann løser multiflyt som en $(1 + 2\epsilon)$-approksimasjon. Kjøretiden blir generelt mye bedre enn den teoretisk polynomiske løsningen vi får fra å løse lineærprogrammet. Generelt kan lineærprogram være eksponensielle i størrelse i forhold til instansen av problemet den skal løse. I dette tilfelle vil antall s-t paths være eksponensiell i antall noder. Merk at dette IKKE betyr at LP formulering + løsing fører til eksponensiell kjøretid, men "dårlig" polynomisk tid. I noen tilfeller kan vi altså være villig til å forbedre den polynomiske kjøretiden, i bytte mot en sub-optimal løsning.

11. Randomisering

Randomiserte algoritmer er definert som en randomisert Turing maskin, altså en Turing maskin som også tar inn en bitstream av tilfeldige bits, og bruker dette i algoritmen. Resultatet av dette er at algoritmen ikke lenger er deterministisk, men kan produsere forskjellig output, basert på forskjellig input. MVO fra forrige kapittel er et eksempel på en randomisert algoritme. Som vi skal se, kan vi generelt gjørerandomiserte algoritmer om til deterministiske. Men, grunnen til at vi fremdeles er intereserte i randomiserte algoritmer er at de ofte er enklere å analysere. Det ødelegger f.eks. for en adversary som ønsker å lage dårligst mulig input for en algoritme. Et formål som vi skal se er random sampling, som ofte gir en nyttig i approksimering.

To raske definisjoner er gjennomgått på Las Vegas og Monte Carlo algoritmer. Den enkle, men unøyaktige huskereglen er at Las Vegas algoritmer er "alltid korrekte, men ikke alltid raske", mens Monte Carlo algoritmer er "Alltid raske, men ikke alltid korrekte". En litt mer formell definisjon sier at Las Vegas algoritmen alltid terminerer, og outputter da enten et korrekt svar, eller ingen svar. Kjøretiden her er avhengig av tilfeldigheter. Monte Carlo algoritmer derimot terminerer enten uten svar, eller med et svar, men svaret er ikke nødvendigvis riktig. Kjøretiden til en Monte Carlo algoritme er også deterministisk. Vi kan ha en-sidig eller to-sidige Monte Carlo algoritmer som hhv. betyr at den kan ta feil på både ja/nei, eller at en av svarende er garantert riktige (f.eks. om den returnerer nei vet vi at svaret er nei, mens om den returnerer ja kan det fremdeles hende svaret er nei).

11.1 Max-SAT, Max-Cut og Min Cut

Dette er to enkle problemer, som begge har en helt triviell randomisert algoritme, men som gir oss en helt ok approksimering. Merk at når vi snakker om randomisert approksimering, så er det snakk om en nedre grense for hvor nære *forventningsverdien* er en optimal løsning. I Max-SAT problemet har vi variabler $x_1, ..., x_n$, og vekter $w_1, ..., w_m$ tilhørende bolske funksjoner $C_1, ...C_m$, formulert som $C_j = x_k \vee x_{k+1} \vee ... \vee x_{k+l}$, der $k + i \in \{1, ..., n\}$ (ikke nødvendigvis etterfølgende), og l betegner lengden på klausul C_j. Ved å sette hver x_i til $0, 1$ med like stor sannsynlighet, får vi en randmoisert $\frac{1}{2}$-approksimasjon. Om vi produserer løsningen W, og y_i betegner hvorvidt klausul C_i er true, så får vi $E[W] = \sum_{j=1}^{m} w_j E[y_j] = \sum_{j=1}^{m} w_j P(C_j true)$. Det er lett å se at på $P(C_j true) = 1 - (\frac{1}{2})^{l_j}$. Siden $l_j \geq 1$ får vi altså $E[W] \geq \frac{1}{2} \sum_{j=1}^{m} w_j \geq \frac{1}{2} OPT$. Som en dirkete konsekvens av denne analysen ser vi at om alle klausuler oppfyller $l_j \geq k$ så får vi faktisk en $(1 - (\frac{1}{2})^k)$-approksimasjon.

Det andre problemet er et Max-Cut problem i en hvilken som helst graf (se definisjon av kutt i Flyt kapitlene). Dette gjør vi på nøyaktig samme måte (legger inn en node i løsningen vår med 0.5 sannsynlighet), og analysen er mer eller mindre og helt lik, så vi får også her en $(\frac{1}{2})$-approksimasjon.

Det motsatte problemet, Min-Cut, har også en tilsvarende enkel approksimasjon, men vi utfører den litt anderledes. Det blir raskt klart hvorfor. I hvert steg trekker vi sammen to noder i grafen (fjerner alle kanter mellom dem, og merger dem), og beholder alle andre kanter. Legg merke til at med mindre kanten er i det optimale min cutet (en node i settet, en node utenfor), så endrer da ikke min-cuttet seg i den nye grafen. Dermed er idéen at vi gjør dette helt til det kun gjenstår to noder. Hvis vi har vært heldig, og ungått å fjærne kanter

med en endenode i kuttet og en utenfor, har vi nå det optimale min-cuttet. La $\delta(S)$ betegne kantene i min cuttet, og la $|\delta(S)| = k$. Vi har dermed at alle noder må ha grad $\geq k$, siden ellers hadde den noden vært et min cut. Sannsynligheten for at når vi fjerner en kant, så er den kanten i $\delta(S)$ er altså $\leq \frac{k}{\frac{nk}{2}} = \frac{2}{n}$. Da blir sansyntligheten for at vi algoritmen fjærner en kant ila. sine $n-2$ iterasjoner $\leq 1 - \frac{2}{n^2}$ (regnes ut som avhengige sannsynligheter). Nøkkelen er nå at denne algoritmen er mega-rask, så vi kan kjøre den mange ganger for å oppnå en bedre nedre grense for sannsynligheten for at løsningen er riktig. Legg merke til at $(1 - \frac{2}{n^2})^{\frac{n^2}{2}} < \frac{1}{e}$ Dermed har vi for eksempel etter å ha kjørt algoritmen $\frac{n^2}{2}$ ganger mindre enn $\frac{1}{e}$ sannsynlighet for at den beste løsningen ikke er optimal.

11.2 Derandomisering

Som nevnt følger et lite eksempel på derandomisering. Målet er å gjøre en randomisert algoritme deterministisk, uten at det går på bekostning av ytelsen. Vi tar Max-SAT som et raskt eksempel, se evt. boka for mer detaljer.

Når vi skal velge x_1, så er tanken at vi ser på de to verdiene $E[W|x_1 = true]$ og $E[W|x_1 = false]$. Det er ikke gitt at denne forventningsverdien generelt er lett å regne ut, men antar vi at den er det, velger vi den høyeste av disse. Antar vi at *true* gir best resultat, er det er klart at $E[W|x_1 = true] \geq E[W]$, siden $E[W] = \frac{1}{2}(E[W|x_1 = true] + E[W|x_1 = false])$. Dette kan vi så gjøre, og vise med induksjon at holder for alle variablene. I Max-SAT er naturlig nok denne forventningsverdien også enkel og regne ut, så dette er nok for å derandomisere Max-SAT algoritmen vår fra isted.

11.3 Randomisert Avrunding av Lineærprogrammer

En veldig intuitiv måte å benytte randomisering for å løse 0-1 IPer (lineærprogram der variabler kun tar verdier $\{0, 1\}$), er å løse LP-relakseringen, oppnå løsningen X^*, og deretter sette $x_i = 1$, med sannsynlighet x_i^*. Dette vil generelt føre til at restriksjoner kan bli brutt, men ikke med så mye. Om vi kan finne en sannsynlighet for at ingen restriksjoner blir brutt som er polynomisk i input størrelsen, har vi sett at vi kan kjøre algoritmen et polynomisk antal ganger for å oppnå en grei sannsynlighet for suksess. Har vi denne egenskapen, sier boken at algoritmen fungerer med *high probability*.

Vi ser på randomisert avrunding for mengdedekke problemet fra tidligere, og utfører taktikken fra forrige avsnitt. Vi ønsker å finne en sannsynlighet for at algoritmen returnerer et ugyldig mengdedekke. Vi ser at sannsynligheten for at element e_i ikke er dekt kan skrives som $P[e_i \text{ ikke dekt}] = \prod_{j:e_i \in S_j}(1 - x_j^*)$. Som tidligere vet vi at $(1 - x) \leq e^{-x}$, så vi får $P[e_i \text{ ikke dekt}] \leq \prod_{j:e_i \in S_j} e^{-x_j^*} = e^{(-\sum_{j:e_i \in S_j} x_j^*)} \leq e^{-1}$, der den siste ulikheten kommer av at x^* ikke bryter LP restriksjonene. Forventningsverdien til algoritmen vår nå er jo $PRIM$, men tar vi nå sannsynligheten for at det finnes et udekt element, blir den for høy. Men, ved å øke forventningsverdien (husk, dette er et minimeringsproblem) kan vi få en randomisert algoritme som løser mengdedekke med *high probability*. For å gjøre dette, si at vi setter $x_i = 1$ med sannsynlighet $1 - (1 - x_i^*)^{cln(n)}$ (intuisjonen her er at x_i^* gir en bias til en mynt som vi flipper $cln(n)$ ganger, der 1 kron er nok til å sette $x_i = 1$). Ved samme analyse for isted ser vi nå at $P[e_i \text{ ikke dekt}] = n^{-c}$, som holder når vi ser på sannsynligheten for at det eksisterer et udekt element. Analyse av forventningsverdien gir

nå $E[\sum_{j=1}^{m} w_j x_j \leq \sum_{j=1}^{m} w_j (cln(n)) x_j^* = (cln(n))PRIM \leq (cln(n))OPT$ (for bevis på den første ulikheten se boka), som gir at denne algoritmen er en $\mathcal{O}(ln(n))$-approksimering av mengdedekke problemet.

For en lignende analyse/strategi på Max-SAT får vi en $(1 - \frac{1}{e})$-approksimering. Se boka for detaljer (strategien er den samme, løs LP relakseringen, og bruk løsningen til å avrunde). Men, det kule resultatet her er at denne strategien fungerer best når l_j er lav, mens som vi har sett er den trivielle algoritmen fra starten av dette kapitellet best når l_j er høy. La W_1 betegne løsningen til den randomiserte avrundingen, mens W_2 betegner den trivielle algoritmen. Ved å nå konstruere en ny algoritme som kjører begge de to og alltid velger $max(W_1, W_2)$ får vi en algoritme med bedre teoretisk randomisert approksimasjon en begge de forrige! En slik algoritme er nemlig en randomisert $(\frac{3}{4})$-approksimasjon. Dette er ikke så vanskelig å se, vi får $E[max(W_1, W_2)] \geq \frac{1}{2}E[W_1] + \frac{1}{2}E[W_2] \geq f(l_j)OPT$, der $f(l_j) = [\frac{1}{2}(1 - (1 - \frac{1}{l_j})^{l_j}) + \frac{1}{2}(1 - 2^{-l_j})] \geq \frac{3}{4}, \forall l_j \in \mathbb{N}$ (her betegner l_j lengden på den lengste klausulen).

12. Onlinealgoritmer

I denne seksjonen ser vi på noe ganske annet enn resten av pensum. For å sammenligne med approksimasjoner som vi har hatt mye fokus på, hadde vi da typisk et optimeringsproblem som kun kunne løses eksakt i eksponensiell tid (med mindre $P = NP$), så vi ønsket å oppnå en approksimasjonsrate, som sa noe om hvor bra løsningen vår (som kunne oppnås i polynomisk tid) var i forhold til den eksakte løsningen. Som vi skal se er definisjonene i et online-problem basert på samme strategi, men med en litt annen problemstilling.

12.1 Onlineproblem og Kompetitiv Analyse

Et onlineproblem er er en et problem der problem kommer som en sekvens av forespørsler $I = (x_1, x_2, ..., x_n)$. En onlinealgoritme er da en algoritme som outputter et svar $O = (y_1, y_2, ..., y_n)$, men der svar y_i kun er avhengig av $x_1, ..., x_i, y_1, ..., y_{i-1}$. Onlineproblemer kan på lik linje med det vi har sett tidligere være minimerings eller maksimeringsproblemer. For å få en tilsvarende metrikk til approksimasjonsrate for å si noe om hvor "bra" onlinealgoritmen er, ser vi på det som kalles *competitive ratio*. En onlinealgoritme er c-competitive dersom:

- $cost(ALG(I)) \leq c(cost(OPT(I))) + \alpha$, for et minimeringsproblem
- $gain(OPT(I)) \leq c(gain(ALG(I))) + \alpha$, for et maksimeringsproblem

for en konstant α, og der $OPT(I)$ betegner den optimale løsningen som kunne vært oppnådd dersom man viste hele $I = (x_1, ..., x_n)$ på forhånd. Vi sier at ALG er strictly c-competitive dersom $\alpha = 0$, og strongly c-competitive, dersom en c-competitive algoritme er det beste som kan oppnås for dette problemet. Videre er ALG not competitive dersom det ikke eksisterer en slik konstant c. Dette impliserer typisk at c er avhengig av n. For å se at disse definisjonene gir mening, se at dersom $\frac{cost(ALG(I_i))}{cost(OPT(I_i))} \leq c, \forall i \in \mathbb{N}^+$ og $lim_{i\to\infty} cost(OPT(I_i)) = \infty$, så kan ikke ALG være $(c-\epsilon)$-competitive, siden impliserer at $cost(ALG(I_i)) \leq (c-\epsilon)cost(OPT(I_i)) + \alpha$ som vil medføre at $\frac{cost(ALG(I_i))}{cost(OPT(I_i))} - \frac{\alpha}{cost(OPT(I_i))} \leq c - \epsilon$ ($\frac{\alpha}{cost(OPT(I_i))}$ går mot 0). Dette kan også overføres til strongly c-competitive, ved å si at den første ulikheten gjelder for alle ALG.

12.2 Paging

Paging er et klassisk onlineproblem, og faktisk det eneste vi skal se på. Det er motivert at vi må ha en smart måte å cache minne på. Vi ser på en veldig forenklet modell som følger. Hovedminne består av såkalte pages $p_1, ..., p_m$, der alle tar like mye plass. I tillegg har vi en cache som har plass til k slike pages. Problemet $I = (x_1, x_2, ..., x_n)$ er page requests slik at $x_i \in \{p_1, ..., p_m\}$ blir requesta ved tid T_i. En onlinealgoritme ALG for dette problemet styrer da strategien for hvilke pages som skal være i cachen ved hvert tidsteg. Cachen (notert som en tuppel B_t for tidsteg T_t) er ved T_0 satt til $B_0 = (p_1, p_2, ..., p_k)$. Dersom $x_i \in B_{i-1}$ (omtalt som page hit), outputter algoritmen $y_i = 0$. Motsatt, dersom $x_i \notin B_{i-1}$ (omtalt som page miss) må ALG velge en $p_j \in B_{i-1}$ som den fjerner før den legger til x_i, slik at $B_i = (B_{i-1} \setminus \{p_j\}) \cup \{x_i\}$. Da setter den også $y_i = p_j$. Målet med paging er å minimere $cost(ALG(I)) = |\{i | y_i \neq 0\}|$. Merk at denne definisjonen setter visse restriksjoner for ALG.

For det første må den initialiseres til de k første pagene, og for det andre får den ikke lov å bytte ut pages utenom når det kommer en $x_i \notin B_{i-1}$ (denne egenskapen gjør at ALG kalles en *demand paging algorithm*). Det kan vises at ingen av disse restriksjonene egentlig er noen restriksjoner for den teoretiske analysen (hvordan algoritmen initialiseres kan fanges opp av α i fra competitive rate definisjonen, mens alle paging algoritmer kan konverteres til en *demand paging algorithm* uten å øke competitive rate).

Vi skal se på en rekke veldig intuitive algoritmer for paging. De er $FIFO$ (First In First Out), $LIFO$ (Last In First Out), LFU (Least Frequently Used), LRU (Least Recently Used), FWF (Flush When Full, merk at denne ikke oppfyller kravet til en *demand paging algorithm*) og LFD (Longest Forward Distance, en offlinealgoritme). Alle disse algoritmene fungerer slik du tror utifra navnet, se evt. boka for mer formelle definisjoner. Målet vil bli å finne hvem av disse som er strongly c-competitive.

12.2.1 Øvre grense

Et konsept som blir sentralt i competitive-analysen vår er såkalt k-fase partisjoneringer. De er definert som følger: For en $I = (x_1, ..., x_n)$ kan vi k-partisjonere I i etterfølgende faser $P_1, ..., P_N$ slik at fase P_1 starter når den første page missen intreffer, og deretter starter etterfølgende faser når den $k+1$ distinkte pagen requestes. Med andre ord inneholder hver fase k distinkte pages (muligens utenom den siste), og videre er de av maksimal lengde.

Med en k-fase partisjonering kan vi enkelt analysere f.eks. $FIFO$. Det er ikke vanskelig å se at i løpet av en fase P_i vil $FIFO$ maksimalt få k page misses (siden hver fase inneholder k distinkte pages). Videre må vi analysere hvor bra det er mulig å oppnå (altså $cost(OPT(I))$). I en fase MÅ OPT få minst 1 page miss (vi antar alltid uten tap av generalitet at x_1 er en page miss for begge). Dermed får vi $cost(OPT(I)) \geq N$ og samtidig $cost(FIFO(I)) \leq kN$, og det følger at $FIFO$ er en strictly-k-competitve online algoritme for paging. Lignende argumenter gir samme resultat for LRU og FWF.

12.2.2 Nedre grense

Hvor bra er det generelt mulig å oppnå? Dersom vi ønsker å vise at $FIFO$ (og LRU og FWF) er strongly-k-kompetitive, må vi klare å bevise at ingen online algoritme kan prestere bedre enn k-kompetitive. Heldigvis er dette ikke vanskelig!

For å vise dette introduserer vi en adversary som konstruerer I på værst mulig måte for ALG. Vi ser for oss et enkelt tilfelle der $m = k + 1$. Idéen er at vår adversary, som vet hvordan ALG fungerer konstruerer I slik at den alltid requester den som ALG nettopp byttet ut (altså $x_i = y_{i-1}$). Vi ser på $|I| \equiv 0 \pmod{k}$, slik at vi kan nøyaktig partisjonere I i faser på lenge k. ALG har da k page misses under en fase, mens en OPT vil naturlig nok ha maksimalt 1 miss. Dette kan formaliseres som at OPT ved en page miss bytter med en p_j som ikke er del av samme fase (dette er mulig, siden fasene kun er k lange). Det kan naturlig nok eksistere flere slike, det er ikke viktig for analysen, men for ordens skyld bytter da OPT ut med den som er lengst til blir requesta neste gang (OPT implementerer altså strategien LFD). Dermed ser vi at det teoretisk beste tilfelle for en online algoritme er å være k-competitive, og vi har dermed vist at $FIFO$ (og LRU og FWF) er strongly-k-competitive.

Denne adversary-analysen gir oss også helt trivielle beviser for at både $LIFO$ og LFU er not competitive. Se evt. boka.

12.2.3 Markeringsalgoritmer

Vi ønsker å generaliserer enkelte av de strongly-k-competitive paging algoritmene. For å gjøre dette introduseres konsepte Markeringsalgoritmen (for paging). De fungerer slik at når en request x_i kommer, så markeres den i cachen (om den ikke allerede er markert). Dersom $x_i \notin B_{i-1}$ har vi kun lov til å fjerne umarkerte pages. Dersom alle pagene i cachen er markerte, så unmarkes alle sammen, og vi kan velge fritt. Når vi legger inn x_i i B_i markeres den med en gang.

Analysen av markingeringsalgoritmen er veldig enkel. Vi ser for oss en partisjonering, der hver fase starter hver gang vi unmarker hele cahcen. Det er ikke vanskelig å se at en slik partisjonering tilsvarer k-partisjoneringen, lik den omtalt i tidligere avsnitt (se detaljer i boka). Videre er det helt åpenbart at markeringsalgoritmen ikke kan ha mer enn k page misses i en fase. Dermed er også markeringsalgoritmen strongly-k-kompetitive.

For å vise at en algoritme er en markeringsalgoritme holder det å bevise at algoritmen aldri fjærner en page som er markert. Med rask tenking ser vi da f.eks at LRU og FWF er markeringsalgoritmer, mens f.eks $FIFO$ er ikke det.

13. Parametrisering

Litt som i forrige seksjon, ønsker vi å se på noen litt andre definisjoner enn i resten av pensum. Dette er riktignok fremdeles relatert til kjøretid, men motivasjonen er at vi ønsker å kunne si noe mer om kjøretiden enn kun hvorvidt problemet kan løses polynomisk eller ikke. Litt mer spesifikt ønsker vi å se om problemer kan løses polynomisk, dersom vi fikserer visse parametre av problemet. Dette introduserer kompleksitetsklassene FPT (Fixed Parameter Tractable), og XP (Slicewise Polynomial). For å beskrive disse kompleksitetsklassene, trenger vi først definisjonen på et parametrisert problem. Et parametrisert problem er $L \subseteq \sum^* \times \mathbb{N}$ der vi typisk skriver det som (x, k), hvor x beskriver problemet, og k er en fiksert parameter av problemet. Vi introduserer da følgene kompleksitetsklasser:

- **FPT:** Problemets kjøretid er begrenset av $f(k) \cdot |(x,k)|^c$, for en konstant c
- **XP:** Problemets kjøretid er begrenset av $f(k) \cdot |(x,k)|^{g(k)}$

For ikke-synkende, og komputerbare funksjoner f, g. Merk at selvom de ser uskyldig like ut, er en FPT *enormt* mye bedre enn en XP. Vi kaller en algoritme en FPT/XP-algoritme, dersom den viser at problemet den løser tilhører klassen FPT/XP. Vi ser på et par raske eksempler, og plasserer dem i kompleksitetsklasser.

13.1 The Good, the Bad and the Ugly

Nodedekke er et kjent NP-komplett problem, som vi har diskutert litt rundt tidligere. Men hva om vi forsøker å fiksere en k, slik at vi spør om det eksisterer et nodedekke av størrelse k? Altså vi ønsker å se på en instans av nodedekke som et parametrisert problem (G, k), der $G = (V, E)$ beskriver grafen, og k er maksimal størrelse på nodedekket vi ønsker å finne. Vi skal i de to neste seksjonene se at vi kan redusere problemet til å løses i tid $\mathcal{O}(2^k k \cdot n)$, for $n = |V|$ (faktisk skal vi se at vi kan klare enda litt bedre også). Ser vi på definisjonene, er det dermed lett å se at dette er en FPT-algoritme (for $c = 1$).

Videre følger et eksempel der vi ikke klarer å oppnå de ønskelige resultatene våre fullt så lett for et slikt naturlig valg av k (sannsynligvis ikke i det hele tatt). Betrakt nå heller problem nodefargelegging (G, k), der vi ønsker å fargelegge nodene i en graf $G = (V, E)$ i k forskjellige farger, slik at for hver kant $(u, v) \in E$ har vi forskjellige farger på u, v. Dette problemet er i det generelle tilfelle NP-komplett, men kanskje vi kunne håpet på en lignende situasjon som i forrige avsnitt, der vi finner polynomiske løsninger for fikserte verdier av k? Dessverre viser det seg at k-fargelegging av noder er NP-komplett for $k \geq 3$, noe som gjør at dersom node-fargelegging var i enten FPT eller XP, ville det med en gang implisert at $P = NP$. Dermed finnes det trolig ingen XP-algoritme engang for node-fargelegging.

Det siste eksempelet for nå er et til klassisk NP-komplett problem, nemlig CLIQUE problemet. Om vi ser på det parametriserte problemet (G, k), der k betegner størrelsen på klikken vi ønsker å finne, finnes det en ganske triviell XP-algoritme (som løser problemet i $\mathcal{O}(n^k)$ tid). Vi ser på alle de $\binom{n}{k} = \mathcal{O}(\frac{n^k}{k^2})$ subsettene av størrelse k, og skjekker om hver av dem er en klikk (i tid $\mathcal{O}(k^2)$). Dessverre finnes det så langt ingen FPT-algoritmer for dette parametriserte problemet. Men, vi kan heller ikke vise på lignende måte som isted at det ville implisert $P = NP$. For å se dette, se at isted brukte vi det at k-node-fargelegging var NP-komplett for $k \geq 3$. En lignende grense for CLIQUE kan ikke eksistere, med mindre

$P = NP$, for da kunne vi brukt XP-algoritmen til å løse problemet i polynomisk tid. I følge boka er dette heller regelen enn unntaket, at vi ikke kan bruke NP-hardhet til å forklare hvorfor vi ikke finner FPT algoritmer, når vi har XP algoritmer.

Men, CLIQUE illustrerer også et annet viktig poeng. Valg av parametrisering er ekstremt viktig. Ser vi heller på den parametriserte versjonen for CLIQUE som (G, Δ), der $\Delta = max\{d(v) : v \in V\}$ (den maksimale node-graden), kan vi finne en FPT-algoritme som kjører i $\mathcal{O}(2^{\Delta}\Delta^2 \cdot n)$. Se boka for detaljer.

13.2 Kernelization

Idéen bak Kernelization er å redusere en problem instans mest mulig, til kun "kjernen" av problemet står igjen. Vi lager en preprosseserings algoritme som reduserer probleminstansen (I, k) til (I', k') på "safe" måte, dvs. at svaret ikke endrer seg. En slik algoritme $\mathcal{A}$ kalles en *kernelization algorithm* eller bare en *kernel* dersom $size_A(k) \leq g(k)$ for en komputerbar funksjon $g : \mathbb{N} \to \mathbb{N}$, hvor $size_A(k)$ betegner størrelsen på outputen eller problemets *kernel*, og $\mathcal{A}$ kjører i polynomisk tid. Med andre ord trenger vi at $\mathcal{A}$ reduserer problemet til et ekvivalent problem, der størrelsen er begrenset av en funksjon av k. Videre har vi et resultat som viser at at et problem har en *kernel* hvis og bare hvis den er i FPT. Se detaljer i boken. Det påpekes at selvom dette er et viktig resultat teoretisk resultat, er denne *kernelen* som regel ikke veldig praktisk anvendbar (den er gjerne eksponensiell i k, mens som vi skal se kan vi ofte klare å oppnå kvadratiske, eller til og med lineære kjerner i k). Merk at vi bedømmer hvor bra $\mathcal{A}$ er utifra $size_A(k)$, og ikke kjøretiden til $\mathcal{A}$. Vi krever kun at den er polynomisk.

13.2.1 Vertex Cover/Nodedekke

Vi ser litt mer nøye på nodedekke. Vi ønsker å finne såkalte reduksjonsregler, slik at vi kan redusere nodedekke til sin *kernel*. Vi skriver et par observasjoner, og fra de deduserer vi reduksjonsreglene. Det første å merke seg er at dersom v er en isolert node ($N(v) = \varnothing$), så vet vi at v åpenbart ikke er med i noen nodedekker. Motsatt vei kan vi tenke oss at dersom node v har $d(v) \geq k + 1$, så må v være i alle nodedekker av størrelse k eller mindre (alltid må enten v eller $N(v)$ være i nodedekke). Etter å ha fjærnet disse nodene fra problemet, se på antall noder og antall kanter i grafen, gitt at vi skal kunne finne et nodedekke av maks k noder. La S betegne dette nodedekke. Hver node i S har grad maksimalt k, og S selv består av maksimalt k noder. Da er det ikke vanskelig å se at et k-nodedekke er umulig dersom $|V| > k^2 + k$. Videre kan S kantene dekke maks $|S|k$ kanter. Dermed vet vi også at det er umulig dersom $|E| > k^2$. Vi er nå klare for å skrive ned reduksjonsreglene.

- **VC.1:** Dersom G inneholder en node isolert node v, lager vi en ny instans av problemet (G, k), der $G = (V \setminus \{v\}, E)$.
- **VC.2:** Dersom det finnes en node v med $d(v) \geq k + 1$, lager vi en ny instans av problemet $(G, k - 1)$, der $G = (V \setminus \{v\}, E')$, der E' er kantene etter at vi har fjernet de som lå intil v.
- **VC.3:** La (G, k) være en instans, der hverken VC.1 eller VC.2 er relevante. Dersom $k > 0$, og $|V| > k^2 + k$ eller $|E| > k^2$, vet vi at et nodedekke er umulig (output "nei").

Alle disse reduksjonsreglene kjører helt åpenbart i lineær tid, så som en konsekvens (særlig av VC.3) får vi at nodedekke har en *kernel* med $\mathcal{O}(k^2)$ noder og $\mathcal{O}(k^2)$ kanter.

13.2.2 Feedback Arc Set In Tournaments

Dette problemet er tilsvarende "Sykelkritiske noder" som tidligere omtalt, bortsett fra at vi er interessert i en løsning bestående av kanter. Videre er grafen vår $T = (V, E)$ en såkalt turnering. En turnering er en komplett graf med orientering (for alle noder $u, v \in V$ har vi nøyaktig en av (u, v) eller (v, u)). Vi ønsker da å fjærne maksimalt k kanter som fjærner alle sykler i T. Analysen her er litt mer teknisk enn for nodedekke, så jeg skriver heller kun opp reduksjonsreglene. Se evt. boka. Med triangler menes sykler bestående av nøyaktig 3 kanter.

- **FAST.1:** Dersom en kant e er del av minst $k + 1$ triangler, lager vi en ny instans av problemet $(T, k - 1)$ der $T = (V, E \setminus \{e\})$
- **FAST.2:** Dersom en kant v ikke er del av noe triangel, lag en ny instans (T, k) der $T = (V \setminus \{v\}, E')$ hvor E' er de kantene som er igjen når vi "shortcutter" v.

Merk at i analysen i boka snur de kanter fremfor å fjærne dem. Det at FAST.2 er "safe" er ikke trivielt, se evt. boka for bevis. Videre gir analysen vår at FEEDBACK ARC SET IN TURNAMENTS har en *kernel* bestående av maksimalt $k^2 + 2k$ noder.

13.2.3 Edge Clique Cover

Et tredje problem som vi ønsker å finne en kjerne til. Målet i dette problemet er for en graf $G = (V, E)$ å si om vi kan dekke hele E med maksimalt k klikker. Det viser seg at den optimale *kernelen* til EDGE CLIQUE COVER er eksponensiell i k (utenfor pensum). Vi gir igjen simpelthen reduksjonsreglene.

- **ECC.1:** Dersom G inneholder en node isolert node v, lag en ny instans (G, k), der $G = (V \setminus \{v\}, E)$.
- **ECC.2:** Dersom det finnes en isolert kant $e = (u, v)$ (u, v har begge grad 1), lag en ny instans $(G, k - 1)$, der $G = (V \setminus \{u, v\}, E \setminus \{e\})$
- **ECC.3:** Dersom det finnes en kant $e = (u, v)$ der $N[u] = N[v]$ ($N[x] = N(x) \cup \{x\}$), lag en ny instans $(G, k - 1)$, der $G = (V \setminus \{v\}, E')$ hvor E' er kantene etter at vi har fjærnet de som lå intill v.

Her er ECC.1 og ECC.2 ganske trivielle, mens ECC.3 er den smarte. To slike såkalte "true twins" kan behandles helt likt for optimum (alle klikkene u er med i er også v med i), så de kan reduseres. Slike noder gir også en god analyse for størrelsen på kjernen, som består av $\mathcal{O}(2^k)$ kanter. For å se dette, la $C_1, ..., C_k$ være klikkene i løsningen. Ingen to noder kan være del av nøyaktig de samme klikkene i løsningen (ellers hadde de vært "true twins" og hadde blitt redusert), og det finnes kun 2^k forskjellige kombinasjoner. Dermed, om $|V| \geq 2^k + 1$ må en node v (husk $d(v) \geq 1$) ikke være med i noen av klikkene, og dermed kan ikke de kantene som er intil v være dekte av løsningen.

13.3 Bounded Search Tree

Idéen her er veldig enkel, men effektiv for parametriserte problemer. Vi gjetter på et hvis en liten del av en løsning, og får en ny, mindre instans av problemet. Dette gjør vi rekursivt, til problemet er trivielt å løse. Dersom det ikke er løsbart "backtracker" vi, og forsøker å gjette anderledes. Dette kan tolkes som å konstruere og søke gjennom et søketre for problemet vi forsøker å løse. Det enkleste er nok å gå igjennom et eksempel, men først ser vi på et par formelle krav.

Vi har en instans I av et problem. I en branch reduserer vi til $I_1, ..., I_l$, hvor iallfall en av subproblemene inneholder løsningen, om den finnes. l må være liten nok, og videre må hver branch redusere størrelsen problemet nok. Litt mer formelt må $l \leq f(|I|)$, og videre har vi at $|I_i| \leq |I| - c$ for en konstant $c \geq 0$. Det vi gjerne ønsker å gjøre for å finne kjøretiden til en slik algoritme, er å finne en grense for antall noder i søketreet, ofte basert på antall løvnoder. Videre må vi vite noe om hvor lang tid vi bruker i hver node.

Vi ser igjen mer på nodedekke, med målet om å nå kjøretiden nevnt i introduksjonen. Vi baserer algoritmen på to enkle fakta: For en node v er som nevnt tidligere alltid enten v eller $N(v)$ med i løsningen. Videre er nodedekke helt trivielt når den maksimale nodegraden i en graf er 1. Dermed er en naturlig branching i grafen å enten velge v eller $N(v)$. Ved hver node velger vi alltid den v med maksimal grad. Videre benytter vi reduksjonsreglene i hver node. I den branchen hvor vi velger v senker vi k med 1, mens i den andre branchen senker vi k med $|N(v)| \geq 2$. Som tidligere diskutert er tid brukt i hver node $\mathcal{O}(n^{\mathcal{O}(1)})$. Det gjenstår dermed å finne et uttrykk for antall noder i søketreet. Vi kan sette opp en rekursiv formel for antall noder som $T(i) \geq T(i-1) + T(i-2), i \geq 2$, og $T(1) = T(0) = 1$ Dette er klart fra måten vi brancher. Dette gir en grenseverdi på $T(k) \leq 1.6181^k$ (se boka for utregning). Dette gir resultatet om at nodedekke kan løses i $\mathcal{O}(n\sqrt{m} + 1.6181^k k^{\mathcal{O}(1)})$. Faktisk kan vi forbedre dette ved å se at nodedekke også er trivielt løslig når maksimal nodegrad er 2. Samme utregning oppnår da en kjøretid på $\mathcal{O}(n\sqrt{m} + 1.4656^k k^{\mathcal{O}(1)})$