Øving 1

Del 1 og 2

Spørsmål, mulige feil og annen hjelp tas via Piazza.

Oppgave 1

Tatt fra Williamson & Schmoys – exercise 1.2.

In the *directed Steiner tree problem*, we are given as input a directed graph $G = (V, A)$, nonnegative costs $c_{ij} \geq 0$ for arcs $(i, j) \in A$, a root vertex $r \in V$, and a set of terminals $T \subseteq V$. The goal is to find a minimum-cost tree such that for each $i \in T$ there exists a directed path from r to i.

Prove that for some constant c there can be no $c \log |T|$-approximation algorithm for the directed Steiner tree problem, unless $P = NP$.

Oppgave 2

Tatt fra Williamson & Schmoys – exercise 1.4.

In the *uncapacitated facility location problem*, we have a set of clients D and a set of facilities F. For each client $j \in D$ and facility $i \in F$, there is a cost c_{ij} of assigning client j to facility i. Furthermore, there is a cost f_i associated with each facility $i \in F$. The goal of the problem is to choose a subset of facilities $F' \subseteq F$ so as to minimize the total cost of the facilities in F' and the cost of assigning each client $j \in D$ to the nearest facility in F'. In other words, we wish to find F' so as to minimize $\sum_{i \in F'} f_i + \sum_{j \in D} \min_{i \in F'} c_{ij}$.

(a) Show that there exists some c such that there is no $(c \ln |D|)$-approximation algorithm for the uncapacitated facility location problem unless $P = NP$.

(b) Give an $O(\ln |D|)$-approximation algorithm for the uncapacitated facility location problem.

(Her må det selvsagt også bevises at algoritmen som oppgis er en $O(\ln |D|)$-approksimeringsalgoritme.)

Oppgave 3

Tatt fra Williamson & Schmoys – exercise 2.1.

The *k-suppliers* problem is similar to the k-center problem given in Section 2.2. The input to the problem is a positive integer k, and a set of vertices V, along with distances d_{ij} between any two vertices i, j that obey the same properties as in the k-center problem. However, now the vertices are partitioned into *suppliers* $F \subseteq V$ and *customers* $D = V - F$. The goal is to find k suppliers such that the maximum distance from a supplier to a customer is minimized. In other words, we wish to find $S \subseteq F$, $|S| \leq k$, that minimizes $\max_{j \in D} d(j, S)$

(a) Give a 3-approximation algorithm for the k-suppliers problem.

(Her må det selvsagt også bevises at algoritmen som oppgis er en 3-approksimeringsalgoritme.)

(b) Prove that there is no α-approximation algorithm for $\alpha < 3$ unless $P = NP$.

Oppgave 4

(a) Formuler ryggsekk-problemet som et heltallprogram (IP problem). La w_i og v_i være henholdsvis vekten og verdien til element i, og la W være kapasiteten til ryggsekken. Et element kan enten tas med en gang eller ikke i det hele tatt.

(b) *Tatt rett fra kapittel 29 i Cormen – prøv før du ser!*

Formuler maks-flyt-problemet som et lineærprogram. Du er gitt en rettet graf $G = (E, V)$. Kilden er s, sluket er t, flyten fra node u til v er f_{uv} og kapasiteten fra node u til v er $c(u, v)$. For enkelthets skyld lar vi $c(u, v) = 0$ når $(u, v) \notin E$.

Oppgave 5 – Programmering

I denne oppgaven skal du prøve å løse mengdedekkeproblemet (*set cover*) med ulike metoder beskrevet i pensum. Vi har laget 15 instanser av ulik størrelse som du kan bruke til å teste implementasjonen din mot. I tillegg gir vi ut et Python-script som kan sjekke om en løsning er gyldig og hva kostnaden er, så du kan være sikker på at implementasjonen din produserer gyldige løsninger.

Utdelte instanser

De ulike probleminstansene ligger i OneDrive-mappen til faget (se Piazza). Hver instans er en tekstfil med heltall og flyttall på formatet:

$$
\begin{array}{l}
\mathtt{N}\ \mathtt{M} \\
\mathtt{w}_1\ \mathtt{e}_1^1\ \mathtt{e}_2^1\ \mathtt{e}_3^1\ \ldots\ \mathtt{e}_{|\mathtt{S}_1|}^1 \\
\mathtt{w}_2\ \mathtt{e}_1^2\ \mathtt{e}_2^2\ \mathtt{e}_3^2\ \ldots\ \mathtt{e}_{|\mathtt{S}_2|}^2 \\
\vdots \\
\mathtt{w}_\mathtt{j}\ \mathtt{e}_1^\mathtt{j}\ \mathtt{e}_2^\mathtt{j}\ \mathtt{e}_3^\mathtt{j}\ \ldots\ \mathtt{e}_{|\mathtt{S}_\mathtt{j}|}^\mathtt{j} \\
\vdots \\
\mathtt{w}_\mathtt{M}\ \mathtt{e}_1^\mathtt{M}\ \mathtt{e}_2^\mathtt{M}\ \mathtt{e}_3^\mathtt{M}\ \ldots\ \mathtt{e}_{|\mathtt{S}_\mathtt{M}|}^\mathtt{M}
\end{array}
$$

Legg merke til at tallene er separert med mellomrom og linjeskift.

Første linje gir:

- **Antall unike elementer:** N.
- **Antall mengder:** M.

Deretter følger M linjer der den j'te linjen gir:

- **Vekten til mengde** S_j**:** w_j. Vekten er i intervallet $[0.5, 1.5]$ og oppgis med fire desimaler etter komma.
- **Elementene i mengde** S_j**:** $e_1^j\ e_2^j\ e_3^j\ \ldots\ e_{|S_j|}^j$. Elementene er representert med heltall fra 1 til og med $|S_j|$.

Sjekke løsning

På OneDrive ligger også `check.py`. Den tar inn navn på instansfil og navn på løsningfil. Eksempel på bruk:

```
$ python check.py 1.in 1.out
```

der `1.in` er en instansfil og `1.out` er en løsningfil.

En løsningsfil (til en tilhørende instans) er en tekstfil med M linjer der den j'te linjen inneholder kun $x_j \in \{0, 1\}$. Se det utdelte eksempelet `1.out` – som er en gyldig (og optimal) løsning på instansen `1.in`.

Verktøy

Du står helt fritt hva gjelder programmeringsspråk, verktøy, solvere, etc. Men vi anbefaler spesielt språket Julia[1]. Det er enkelt, raskt og har god støtte for solvere gjennom pakken JuMP[2].

Pass på å installere nyeste versjon av Julia (0.6). Hvis du har installert Julia trenger du bare installere JuMP og en solver (f.eks. Cbc). Det gjøres enkelt i Julia:

```
Pkg.add("JuMP")
Pkg.add("Cbc")
```

JuMP har mange eksempler på bruk på GitHub[3]. Vi anbefaler å ta en titt på knapsack-eksempelet[4], mens du samtidig ser på det du kom fram til i oppgave 4a.

Aktiviteter

Her lister vi opp noen forslag til aktiviteter – det er **ikke** meningen å gjøre alle. Husk at øvingsopplegget er frivillig og du styrer selv hva du ønsker å gjøre.

(a) Formuler problemet som et heltallsprogram og la en solver løse det.

(b) Implementer den deterministiske avrundingsalgoritmen fra seksjon 1.3.

(c) Implementer dual-avrundingsalgoritmen fra seksjon 1.4.

(d) Implementer primal-dual-algoritmen fra seksjon 1.5.

(e) Implementer den grådige algoritmen fra seksjon 1.6.

(f) Implementer randomiserte avrundings-algoritmen fra seksjon 1.7.

(g) Implementer en annen algoritme du selv har funnet på eller funnet!

(h) Hvem greier å finne best løsning på de ulike instansene? Opprett en mappe med brukernavnet ditt på OneDrive og last opp dine beste løsninger og konkurrer med andre studenter!

(i) Lag vanskeligere instanser! Det er ikke trivielt å lage vanskelige mengdedekke-instanser som også er små. Mange av de utdelte instansene er gjort store for å gjøre dem vanskelige. Hvor små instanser som du ikke klarer å finne optimal løsning på klarer du å lage? Opprett en mappe med brukernavnet ditt på OneDrive og last opp probleminstansene dine og dine beste løsninger på dem. Post på Piazza og utfordre andre til å løse instansene dine.

[1] `https://julialang.org/`
[2] https://github.com/JuliaOpt/JuMP.jl
[3] https://github.com/JuliaOpt/JuMP.jl/tree/release-0.18/examples
[4] https://github.com/JuliaOpt/JuMP.jl/blob/release-0.18/examples/knapsack.jl

Legg merke til at flere av de utdelte instansene er så store at det er svært utfordrende å finne en optimal løsning (eller gode tilnærmeringer) med algoritmene i pensum. Ikke fortvil om noen av de ikke lar seg knekke – det er meningen.