Combinatorische Optimalisatie
Samenvatting Periode 4

Koen Benjamins

1 Introduction, graphs, paths and trees

1.1 Graven

Een graaf bestaat uit punten, ook wel nodes of knopen. Lijnen die punten combineren heten lijnen of edges, ook wel arcs als er maar in een richting over de lijn gelopen kan worden. Het aantal lijnen dat in een punt komt heet de degree of graad van dat punt. In het geval van arcs wordt er een tweedeling gemaakt tussen de in-degree en de out-degree van een punt. Twee punten zijn verbonden indien je van het eerste punt in het tweede punt kan komen, ongeacht of daar andere punten voor nodig zijn. Een Euler pad is een pad dat elk lijnstuk van de graaf precies één keer gebruikt. Voor dat pad mogen het begin en eindpunt verschillend zijn. Een Euler circuit is een circuit dat elk lijnstuk van de graaf precies één keer doorloopt en waarvan het beginpunt het eindpunt is.
Stelling: *Een verbonden (multi)graaf heeft een Eulerpad als en slechts dan als het aantal punten met een oneven graad is 0 of 2. Als het aantal 0 is dan is het pad een circuit.*
Stelling: *Een Hamiltonpad is een pad langs knooppunten in een graaf waarbij elk knooppunt precies één keer op het pad ligt. Een gesloten Hamiltonpad noemt men een Hamiltoncykel of Hamiltoncircuit.*
Wanneer nu niet de vraag is of er een Eulerpad of Hamiltonpad bestaat, maar wat het goedkoopste of kortste pad is, dan is er sprake van een optimalisatieprobleem. Een voorbeeld van een algoritme om dit op te lossen is het Algoritme van Dijkstra.

1.2 Greedy Algoritmes

Het Greedy Algoritme is een algoritme dat bij iedere stap op weg naar de oplossing de stap kiest met de hoogste directe toevoeging. Bijvoorbeeld bij het knapsack probleem, kiest Greedy iedere keer het item met de kleinste waarde/gewicht verhouding. Greedy is vaak niet optimaal, maar sub-optimaal. In het geval de minimaal opspannende boom is Greedy wél optimaal.

1.3 Minimaal opspannende boom

De minimaal opspannende boom van een verbonden, gewogen graaf is de verbonden subgraaf daarvan met het kleinste totale gewicht.
Algoritme 1 - Kruskal:
Bij dit algoritme worden alle zijdes op volgorde van lengte (gewicht) gezet. Bij iedere iteratie wordt de zijde met de laagste lengte toegevoegd. Indien door het toevoegen van die zijde een cykel ontstaat, wordt de zijde overgeslagen en wordt dus de zijde met de op een na laagste lengte toegevoegd, mits deze natuurlijk geen cykel vormt. Deze iteratie wordt herhaald, totdat alle punten in de boom zitten.

Algoritme 2 - Prim:
Begin bij een willekeurig punt. Voeg iedere keer de zijde met de laagste waarde toe die in dat punt uitkomt. Verplaats dan naar de andere kant van die lijn en doe vanuit dat punt hetzelfde, zolang er geen cykel ontstaat.
Algoritme 3 - Boruvka:
Verbind elk punt met het dichtsbijzijnde (i.e. laagste waarde) punt. Herhaal de vorige stap, maar lees dan component in plaats van punt.

1.4 Stromen

Een stroom (flow) van s naar t, kortweg een $s-t$ stroom, is een functie $f : A \to \mathbb{R}_+$ met de eigenschap, dat voor elk punt $v \neq s$ geldt:

$$\sum_{a \in \delta^-(v)} f(a) = \sum_{a \in \delta^+(v)} f(a)$$

Met $\delta^-(v) :=$ de verzameling pijlen die in v aankomen en $\delta^+(v) :=$ de verzameling pijlen die uit v vertrekken. Het betreft hier dus een stroom met een maximale capaciteit. Deze conditie zegt dat in elk punt $v \neq s, t$ de hoeveelheid stroom die in v aankomt gelijk is aan de hoeveelheid stroom die uit v vertrekt. De waarde van de stroom $s-t$ is gelijk aan de netto hoeveelheid stroom die vertrekt uit s. Dus:

$$waarde(f) := \sum_{a \in \delta^+(s)} f(a) - \sum_{a \in \delta^-(s)} f(a)$$

1.4.1 Maximale stroom probleem

Een stroom van maximale waarde heet een maximum stroom. Om deze te vinden moet het *maximale stroomprobleem* worden opgelost. Die is als volgt gedefinieerd:

Gegeven: een gerichte graaf $D = (V, A)$, een bron s en een put t, en een capaciteitsfunctie $c : A \to \mathbb{R}_+$.

Bepaal: een $s-t$ stroom f onder c, met waarde(f) zo groot mogelijk.

Met f *onder* c wordt bedoeld, dat de stroom f een gegeven 'capaciteitsfunctie' (meestal een waarde) $c : A \to \mathbb{R}_+$ niet overschrijdt, ofwel $f(a) \leq c(a)$ voor elke pijl a.

1.4.2 Snede

Als $U \subseteq V$, definieer $\delta^+(U)$ als de verzameling pijlen welke U verlaten, dus de staart zin in U en de kop zin in $V \backslash U$. En, definieer $\delta^-(U)$ als de verzameling pijlen die in U aankomen, dus de staart zin in $V \backslash U$ en de kop in U. Als $s \in U$ en $t \notin V$
U, dan heet $\delta^+(U)$ een $s-t$ snede. De capaciteit van een snede $\delta^+(U)$ is gelijk aan de som van $c(a)$ genomen over alle pijlen $a \in \delta^+(U)$.
Stelling: De maximale waarde van een $s-t$ stroom onder c is gelijk aan de minimale capaciteit van de $s-t$ snedes. *Opm: als de capaciteiten geheeltallig zijn, dan bestaat er een geheeltallige maximum stroom.*

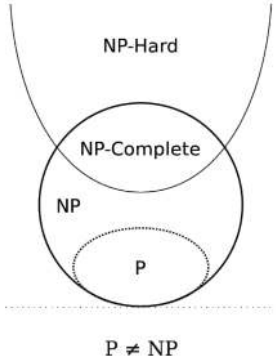
2 Complexity classes and reduction

2.1 Grote O notatie

Hetgeen binnen de O staat is de dominerende functie van een eenvoudige vorm, zodat een overzichtelijke indruk verkregen wordt van het asymptotische gedrag van de doelfunctie. Dus waar neigt de functie $f(x)$ naar voor $x \to \infty$. Vb. $f(x) = x^3 + \sqrt{x} + 1 \in O(x^3)$. Let op dat $f(x) \in O(x^{10})$ is ook waar, maar $f(x) \in O(\sqrt{x})$ niet.
Als je kijkt naar de rentijd van een algoritme, ga je uit van de hoogste O. Dus is een bepaalde stap in een algoritme $O(n)$, maar een andere stap $O(1 + 2 + ... + n)$, dan is $O(1 + 2 + ... + n)$ leidend en dus de indicator voor de rentijd.
Merk op dat $O(1 + 2 + ... + n) = O(n^2)$*, want* $1 + 2 + ... + n = \frac{n(n+1)}{2}$.

2.2 P of NP probleem

De klasse P bevat de problemen die opgelost kunnen worden met een algoritme dat in polynomiale tijd uitgevoerd kan worden (de tijdsduur van het algoritme wordt begrensd door een polynoom). Let wel dat het bewijzen van het niet bestaan van een oplossing ook een oplossing is.
De klasse NP bevat alle problemen waarvoor een algoritme bestaat dat, gegeven een voorgestelde oplossing, in polynomiale tijd kan controleren of de voorgestelde oplossing correct is. NP-volledige problemen zijn problemen die in de complexiteitsklasse NP liggen en waarvoor bovendien geldt, dat ieder probleem in NP in polynomiale tijd ertoe gereduceerd kan worden. Ze kunnen dus ook onderling tot elkaar gereduceerd worden. Daarmee vormen de NP-volledige problemen een complete graaf van onderlinge reduceerbaarheid.
Een gegeven probleem A is NP-moeilijk als ieder beslissingsprobleem in NP in polynomiale tijd tot A gereduceerd kan worden; het is dus minstens zo moeilijk als ieder probleem in NP. Als het probleem zelf ook tot de klasse NP behoort, dan is het NP-volledig. Niet ieder probleem dat NP-moeilijk is, is NP-volledig; het omgekeerde is - per definitie - wel het geval. If we denote a reduction from A to B by $A \to B$ then we can say that difficulty flows in the direction of the arrow, while efficient algorithms move in the opposite direction. It is through this propagation of difficulty that we know NP-complete problems are hard: all other search problems reduce to them, and thus each NP-complete problem contains the complexity of all search problems. And now we can finally define the class of the hardest search problems. A search problem is NP-complete if all other search problems reduce to it. Elk probleem in NP is te reduceren tot een NP-compleet probleem. Ook is het geval dat per definitie $P \subseteq NP$. Zie linker afbeelding voor overzicht.



Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

3 Dynamic programming

3.1 Definitie en eigenschappen

Dynamisch programmeren is een techniek voor het optimaal nemen van een rij van afhankelijke beslissingen. Het kan worden toegepast als de oplossing van een probleem gevonden kan worden door de oplossingen van subproblemen. Er moet dus een recursieve formule voor de oplossing worden gevonden. Dan kan er dus recursief een oplossing worden gevonden op de subproblemen, startend vanaf een triviaal/bekend beginpunt. *Let op: dynamisch programmeren is niet hetzelfde als het implementeren van een recursieve functie.*
Dit betekent dus dat het probleem de volgende eigenschappen moet hebben:

1. Simpelere subproblemen: het probleem moet kunnen worden opgedeeld in kleinere en makkelijker op te lossen subproblemen, die dezelfde structuur hebben.
2. Optimale substructuur van de problemen: de oplossing van het probleem moet zijn samengesteld uit de oplossing van de subproblemen.
3. Subprobleem overlap: subproblemen mogen elkaar overlappen.

3.2 Oplossing door middel van Dynamisch Programmeren

Een probleem kan worden opgelost door middel van DP door de volgende stappen af te lopen:

1. Definiëren van de op te lossen subproblemen;
2. De optimale waarde, weergegeven in termen van de subproblemen;
3. Beginwaarden;
4. De recursie;
5. Analyse van de benodigde ruimte (op computer);
6. Analyse van de rentijd van het probleem.

4 Approximation algorithms

4.1 Approximatie ratio

De approximatie ratio van een algoritme A geeft de maximale deviatie, die gevonden kan worden tussen de oplossing gevonden door middel van algoritme A en de optimale waarde. De approximatie ratio is als volgt gedefinieerd:

$$\alpha_A = \max_I \frac{OPT(I)}{A(I)}$$

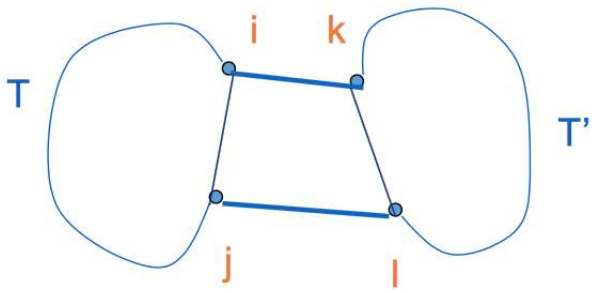
Let op dat soms de ratio andersom wordt berekend en dus een 2-approximatie algoritme hetzelfde is als een 1/2-approximatie algoritme. De volgende punten moeten in aanmerking worden genomen, onder de veronderstelling dat $P \neq NP$:

- There are NPC problems for which no efficient algorithm exists with a finite approximation ratio!
- There are NPC problems where the approximation ratio is finite but increases with the dimension of the instance!
- There are NPC problems for which finite approximation ratios are possible but these cannot be arbitrarily small!
- There are NPC problems that once decided the value for the approximation ratio an algorithm can be found that yields it!

A final point on approximation algorithms: often these algorithms, or their variants, per- form much better on typical instances than their worst-case approximation ratio would have you believe.

4.2 Nearest merger

Dit is een algoritme voor het Travelling Salesman Problem (TSP). Gegeven twee subtours T en T', vind $(i, j) \in T$ en $(k, l) \in T'$ waarvoor: $\min\{c_{ik} + c_{lj} - c_{ij} - c_{kl}\}$ Vervang (i,j) en (k,l) voor (i,k) en (l,j) zodat er een nieuwe grotere (sub)tour ontstaat.Als in de onderstaande afbeelding. *Opmerking: dit algoritme bestaat net als het double tree en nearest addition algoritme alleen voor zgn. 'metric TSP', ofwel een TSP met driehoeksongelijkheid. Voor een TSP zonder driehoeksongelijkheid bestaat geen approximatiealgoritme.* In woorden: in het begin is elk punt een tour. In elke iteratie worden de twee tours met de kleinste onderlinge afstand samengevoegd. Dit wordt gedaan door een lijn uit elke tour weg te laten (tenzij de tour een enkel punt is) en de twee tours met twee nieuwe lijnen te verbinden zodanig dat de kostentoename minimaal is.



5 Scheduling

5.1 Notatie

Scheduling concerns optimal allocation or assignment of resources, over time, to a set of tasks/activities/jobs. Where the resources (e.g. machines or people) are denoted by M_i, with $i = 1, 2, ..., m$ and the tasks (e.g. jobs) are denoted by J_j, with $j = 1, 2, ..., m$. Jobs parameters are:

p_j: processing time of job j

p_{ij}: processing time of job j on machine i (when processing time of job j depends on machine i)

r_j: release date of job j (earliest starting time)

d_j: due date (deadline) (committed completion time)

w_j: weight of job j (importance)

(Most) scheduling problems can be described by a three field notation $\alpha|\beta|\gamma$, where

$\boldsymbol{\alpha}$: describes the machine environment

(a) Single machine ($\alpha = 1$)

(b) Identical parallel machines (α = P or Pm)

- midenticalmachinesrunninginparrallel;
  * If α = P, then the number of machines, m, is part of the input
  * If α = Pm, the value m is considered a constant
- each job consists of a single operation and this may be processed by any of the machines for p_j time units

(c) Unrelated parallel machines (α = R or Rm)

- m different machines in parallel:
  p_{ij} is the process time of job j if scheduled completely on machine i

$\boldsymbol{\beta}$: describes the job characteristics

- release dates (r_j in β field)
  * if r_j in β field, jobs may not start processing before their release date
  * if r_j is not in β field, jobs may start at any time
- deadlines (d_j in β field)
  * if d_j is in β field, each job j should finish before time d_j
- preemption (pmtn in β field)
  * processing of a job on a machine may be interrupted and resumed at a later time even on a different machine
- unit processing times (pj = 1 or pij = 1 in ? field)
  * each job (operation) has unit processing times
- precedence constraints (prec in β field)
  * A job cannot start before some other job(s) are finished
  * May be represented by an a cyclic graph (vertices=jobs, arcs= precedence relations)

$\boldsymbol{\gamma}$: describes the objective criterion to be minimized (or max.) Some examples:

- Makespan: $\gamma = C_{max}$
- Maximum lateness: $\gamma = L_{max}$
- Total completion time: $\gamma = \sum_j C_j$

$\boldsymbol{C_j}$: is the completion time van job j vanaf tijdstip 0.

5.2 Algoritmes

Hieronder staat een aantal algoritmes voor **één** machine problemen.

5.2.1 Earliest Due Date (EDD)

Dit algoritme wordt gebruikt voor het inroosteren van taken bij een probleem om de *lateness* L_j te optimaliseren, met $L_j = C_j - d_j$. Het EDD-algoritme houdt in dat de taken op volgorde van *due date* worden ingeroosterd. Dit algoritme is optimaal voor dit probleem.

5.2.2 Shortest Process Time

Dit algoritme is optimaal voor het minimaliseren van de gemiddelde *flowtime*. De volgorde is gebaseerd op de procestijd van de taken. De volgorde van overblijvende taken is gesorteerd op basis van toenemende procestijd. Dus de taken worden op volgorde van procestijd gesorteerd en uitgevoerd, dus in de volgorde: $p_1 \leq p_2 \leq \ldots \leq p_n$.

5.2.3 Shortest Remaining Process Time

Dit algoritme is eigenlijk hetzelfde als 5.2.2, maar wordt gebruikt in gevallen waar preemption is toegestaan. Dit algoritme zet telkens job j op de machine met de kortste resterende procestijd.

6 Routing

Het TSP heeft ten doel het vinden van een route door alle punten met minimale kosten. Een generalisatie van dit probleem wordt gevonden door te stellen dat de punten verdeeld moeten worden in m routes. Door het toevoegen van een grootte per punt en een capaciteit aan de 'voertuigen' die gebruikt worden, dan wordt een capaciteitsrestrictie toegevoegd: de aantallen die vervoerd moeten worden naar de punten per route moeten in het 'voertuig' passen. Dit heet ook wel het Capacitated Vehicle Routing Problem of CVRP.
Two classes of often used solution methods for the CVRP use the fact that the problem generalizes the TSP: the route-first-cluster-second and cluster-first- route-second methods. These methods decompose the CVRP problem in two subproblems: a clustering problem and a TSP, where the difference between the two classes is determined by the order in which the problems are solved. If first the clusters are made, then the second subproblem is a TSP which can be solved to create a route for each cluster separately. If first the routing is done, then a single TSP is solved with all the locations. In the second step, the solution is broken down into sections for single vehicles, which need to be transformed into feasible routes by connecting the beginning and end to the depot, respecting the capacity restrictions.