Algoritmer

*"**algori´tm,** följd av instruktioner för ett beräkningsarbete som i ett ändligt antal steg löser ett beräkningsproblem och därmed kan utgöra grunden för ett datorprogram."* - Ne.se

---

Allmänt

Denna sammanfattning skrevs av flera personer våren 2014 till kursen EDAF05 - Algoritmer, datastrukturer och komplexitet. Dokumentet finner man här : https://docs.google.com/document/d/1U4h4F9AaCBm1ASAG98wRHFAR6VXka-u5E5AbImnf0zY/edit?usp=sharing

Breadth-First Search

BFS är en algoritm för att kontrollera om en graf är sammankopplad (Connected). Alltså om det för alla par av noder *u* och *v* finns en väg mellan *u* och *v* i grafen. Med BFS går man igenom samtliga noder i grafen genom att gå genom ett lager åt gången, med utgångspunkt från en nod S som sammankopplar samtliga noder.

Tidskomplexiteten för BFS är $O(m+n)$ om grafen representeras av en närbelägen lista (adjacency list).

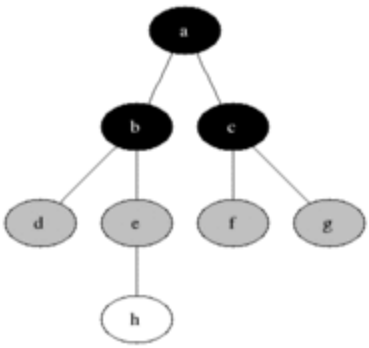


Bild på hur man går genom en graf m.h.a BFS.

Depth-First Search (DFS)

Med denna algoritmen går man genom en väg i grafen tills man kommer till sista noden och går sedan tillbaka till den första noden tidigare (dvs en nod längs vägen man precis besökt) som har en obesökt nod och går vidare längs dennes kanter. Denna algoritm används ofta för att hitta vägen genom en labyrint.

Tidskomplexiteten för DFS är samma som för BFS med samma argument.

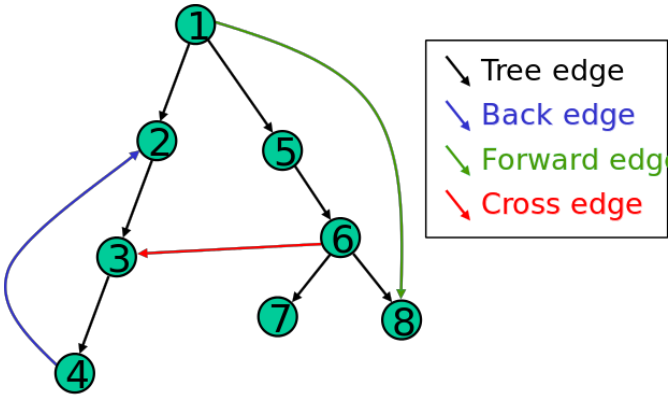


Bild över DFS

Laboration 2

Under laboration 2 (WORD LADDERS) använde vi oss av algoritmen BFS för att gå igenom en riktad graf som var uppbygd enligt följande anvisning. Du kan gå från ett ord till annan om var och en av de fyra sista bokstäverna av det förra ordet visas i senare ordet.

Divide and Conquer

Söndra och härska refererar till en klass av algoritmiska tekniker där man bryter problemet i flera delar, löser varje delproblem rekursivt och sedan kombinerar lösningar på dessa delproblem till en helhetslösning.

MergeSort

Dela indatan i två lika stora delar; lös de två delproblem på dessa delar med rekursion; och kombinera de två resultaten i en slutlig lösning genom att endast spenderar linjär tid för den första divisionen och sista sammanställningen.

Tidskomplexitet för mergesort

1. Metoden **merge**'s tidskomplexitet är *O(n)* där *n* är fältets storlek.
2. **MergeSort** har tidskomplexiteten *n log(n)* där *n* är antal element i listan man ska iterera igenom .

Merge and count är en form av Mergesort med den skillnaden att man i varje rekursivt anrop räknar antalet element *a* i lista A som är mindre än element *b* i lista B. Denna algoritm används för att beräkna hur stor skillnad det är mellan två olika mängder av objekt.

Användningsområden:

- Finna närmaste punkter i planet
- Bestäm skillnaden mellan två listor av objekt
- Sortera en lista, Ex) list.sort(); på en ArrayList list i Java.

Laboration 4

Under laboration 4 (CLOSEST PAIRS IN THE PLANE) implementerade vi en söndra och härska algoritm för att hitta ett närmaste par av punkter i planet.

Greedy Algorithms

En algoritm är girig om den bygger upp en lösning i små steg och vid varje steg väljer den **just då** bästa lösningen. För vissa *optimeringsproblem* hittar den giriga algoritmen en optimal lösning, medan för andra så kommer den inte att hitta någon optimal lösning.

Dijkstras algoritm

Dijkstras algoritm är en girig algoritm som hittar den billigaste vägen från en given nod till alla andra noder i en viktad och riktad graf med positiva bågkostnader.

Tidskomplexitet för Dijtstras algoritm

Om man implementerar prioritetskön med hjälp av en Fibonacci heap så har algoritmen tidskomplexiteten $O(E+VlogV)$, där V är antalet noder och E är antalet vägar i grafen.

Exempel på användningsområden för giriga algoritmer:

- Hitta den kortaste vägen i en graf ex. Handelsresandeproblemet (ibland förkortat **TSP** efter det engelska *the Traveling Salesman Problem)*
- Skapa ett minimalt uppspännande träd
  Minimalt uppspännande träd kan användas i många situationer där ett antal noder ska sammankopplas till minsta möjliga kostnad. Kostnaden för att ansluta två noder anges då som vikten för den kant som förbinder nodernas motsvarande hörn i en viktad graf. Den fullständiga grafen konverteras sedan med godtycklig algoritm till ett minimalt uppspännande träd där den billigaste lösningen åskådliggörs.
  Vanliga exempel är dragning av kablar för bredband, kabel-TV, telefoni eller elförsörjning. Hörnen motsvarar här hus och kanterna representerar de vägar som ledningen kan dras. De olika vikterna svarar mot vad det kostar att lägga en kabel efter en viss väg, vilket kan bero på faktorer som vägens längd, hur djupt man måste gräva ner ledningen och vilket bergart man tvingas gräva i. Eftersom det förefaller sig ganska osannolikt att två vägar skulle ha exakt samma kostnad kan man sedan entydigt bestämma var det blir billigast att dra ledningarna genom att ta fram ett minimalt uppspännande träd ur grafen.

Laboration 3

Greedy Algorithm använde vi oss av i laboration 3 (SPANNING USA) där vi skulle hitta ett minsta uppspännande träd mellan 128 amerikanska städer där längden av avståndet mellan städerna utgör vikten mellan noder.

Dynamic programming

"fancy name for caching away intermediate results in a table for later reuse"

Dynamisk programmering (DP) är en metod för att lösa komplexa problem genom att bryta ner dem i enklare delproblem. Genom att systematiskt beräkna lösningar till delproblem, spara dessa på ett effektivt sätt, samt att låta alla dellösningar beräknas genom att utnyttja andra dellösningar kan man hitta effektiva algoritmer för annars svårlösta problem.

Tanken bakom DP är ganska enkel. För att lösa ett givet problem så måste vi lösa olika mindre delar av problemet (delproblem), och sedan kombinera lösningarna av delproblemen för att nå en övergripande lösning. När man använder en mer naiv metod så är det vanligt att många av delproblemen blir genererade och lösta flera gånger. DP ämnar att lösa detta genom att bara lösa varje delproblem en gång, vilket minskar antal beräkningar (men ökar lagringsutrymmet, som ju används för att lagra dellösningarna). Så fort en lösning till ett delproblem har lösts så lagras lösningen. Nästa gång samma lösning behövs så hämtar man den bara. DP är extra användbart när antalet återkommande delproblem växer exponentiellt som en funktion av storleken på insignalen.

DP-algoritmer används för optimering (till exempel för att hitta den kortaste vägen mellan två punkter). En DP-algoritm kommer att undersöka alla möjliga sätt att lösa problemet och kommer sedan att plocka den bästa lösningen.
Ett alternativ till DP är att använda en girig algoritm, som alltid väljer den bästa vägen vid varje gren. En girig algoritm garanterar inte en optimal lösning men den är snabbare.

Som exempel, låt oss säga att du ska ta dig från punkt A till punkt B så snabbt som möjligt i en stad under rusningstid. En DP-algoritm kommer att granska hela trafikrapporten, kontrollera alla möjliga kombinationer av vägar du kan ta och kommer sedan tala om för dig vilken väg som är snabbast. Kanske måste du vänta ett tag tills algoritmen är klar, och först då kan du börja köra. Men vägen du tar kommer att vara snabbast (förutsatt att inget har hunnit ändras). Å andra sidan, om du hade använt en girig algoritm så hade du börjat köra direkt, och valt den väg som ser snabbast ut vid varje korsning. Den giriga algoritmen kanske inte tar dig till B snabbast, då du kanske hamnar i en trafikstockning efter att ha valt några vägar som ser snabbast ut.

Exempel på användningsområden för dynamiska algoritmer:

- Sequence Alignment
- Bellman-Ford algoritmen för att hitta det kortaste vägen i en graf

Laboration 5

Under laboration 5 (ME: GORILLA OR SEA CUCUMBER?) använde vi oss av dynamisk programmering.

Network flow / flödesnätverk

I grafteori är ett flow network en riktad graf där varje kant har en kapacitet och varje kant mottager ett flöde. Mängden flöde över en nod kan inte överskrida kapaciteten hos kanten. Ibland kallas ett flow network för ett nätverk. Ett nätverk kan användas för att modellera trafik i ett vägsystem vätskor i rör, ström i en elektrisk krets, eller något liknande där något färdas genom ett nätverk av noder.

> *The max-flow min-cut theorem states that finding a maximal network flow is equivalent to finding a cut of minimum capacity that separates the source and the sink.*

Ett vanligt problem att lösa m.h.a flödesnätverk är att hitta det maximala flödet (max flow).
Det maximala flödet är det största möjliga flödet från source till sink. Max-flow min-cut teoremet säger att hitta det maximala nätverksflödet är detsamma som att hitta ett "snitt" som separerar källan (source) och vasken (sink).
Max-flow min-cut teoremet säger att hitta det maximala nätverksflödet är det samma som att hitta ett "snitt" av minimikapacitet som separerar källan och vasken.

Olika algoritmer för att hitta max flow:

Algoritm	**Komplexitet**	**Beskrivning**
Ford–Fulkerson algorithm	$O(E \max\|f\|)$	As long as there is an open path through the residual graph, send the minimum of the residual capacities on the path. The algorithm works only if all weights are integers. Otherwise it is possible that the Ford–Fulkerson algorithm will not converge to the maximum value.
Edmonds–Karp algorithm	$O(VE^2)$	A specialization of Ford–Fulkerson, finding augmenting paths with breadth-first search.

Laboration 6

(Railroad planning) handlade om network flow.

NP-Hard

NP-naming convention

NP-hard problems do not have to be elements of the complexity class NP, despite having NP as the prefix of their class name. The NP-naming system has some deeper sense, because the NP family is defined in relation to the class NP and the naming conventions in the Computational Complexity Theory:

NP

Class of computational problems for which solutions can be computed by a non-deterministic Turing machine in polynomial time (or less). Or, equivalently, those problems for which solutions can be checked in polynomial time by a deterministic Turing machine.

NP-hard

Class of problems which are at least as hard as the hardest problems in NP. Problems in NP-hard do not have to be elements of NP, indeed, they may not even be decision problems.

NP-complete

Class of problems which contains the hardest problems in NP. Each element of NP-complete has to be an element of both NP and NP-hard.

NP-easy

At most as hard as NP, but not necessarily in NP, since they may not be decision problems.

NP-equivalent

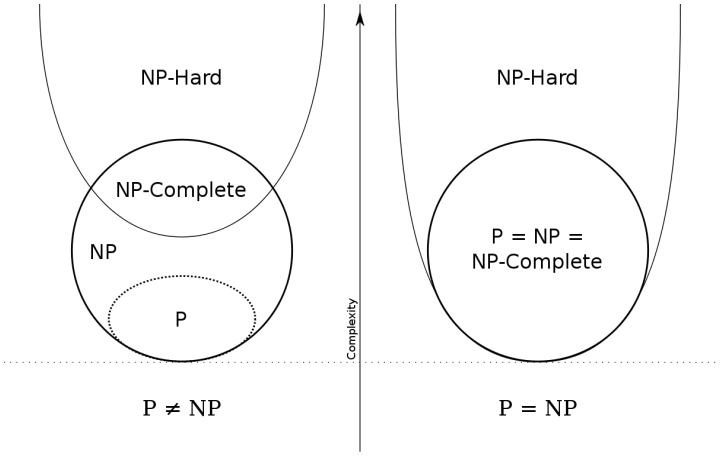
Exactly as difficult as the hardest problems in NP, but not necessarily in NP.

NP-hard (**N**on-deterministic **P**olynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A problem *H* is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H (i.e., $L \leq_T H$). In other words, *L* can be solved in polynomial time by an oracle machine with an oracle for *H*. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving *H*, and solves *L* in polynomial time, 2if the subroutine call takes only one step to compute. NP-hard problems may be of any type: decision problems, search problems, oroptimization problems.

As consequences of the definition, the following claims can be made:

- Problem *H* is at least as hard as L, because H can be used to solve *L*;
- Since *L* is NP-complete, and hence the hardest in class NP, also problem *H* is at least as hard as NP, but *H*does not have to be in NP and hence does not have to be a decision problem (even if it is a decision problem, it need not be in NP);
- Since NP-complete problems transform to each other by polynomial-time many-one reduction (also called polynomial transformation), all NP-complete problems can be solved in polynomial time by a reduction to *H*, thus all problems in NP reduce to *H*; note, however, that this involves combining two different transformations: from NP-complete decision problems to NP-complete problem *L* by polynomial transformation, and from *L* to *H* by polynomial Turing reduction;
- If there is a polynomial algorithm for any NP-hard problem, then there are polynomial algorithms for all problems in NP, and hence P = NP;
- If P ≠ NP, then NP-hard problems cannot be solved in polynomial time, while P = NP does not resolve whether the NP-hard problems can be solved in polynomial time;
- If an optimization problem H has an NP-complete decision version *L*, then *H* is NP-hard.

A common mistake is to think that the *NP* in *NP-hard* stands for *non-polynomial*. Although it is widely suspected that there are no polynomial-time algorithms for NP-hard problems, this has never been proven. Moreover, the class NP also contains all problems which can be solved in polynomial time.



Example of NP-Complete problems

Set-cover

http://en.wikipedia.org/wiki/Set_cover_problem

Givet en mängd {1, 2, …, m} (kallat universumet) och en mängd S av n mängder vars union är lika med universumet är problemet att hitta den minsta delmängden av S så att denna delmängd är lika med (täcker) universumet.

Exempel:
U = {1, 2, 3, 4, 5}
S = {{1, 2, 3}, {2,4}, {3, 4}, {4,5}}
Minsta delmängden av S som täcker U är {1,2,3} + {4, 5}.

Graph colouring

http://en.wikipedia.org/wiki/Graph_coloring
Färglägg varje vertex i en graf med ett givet antal färger så att ingen vertex delar färg med en granne.

3-dim. matching

http://en.wikipedia.org/wiki/3-dimensional_matching
Let X, Y, and Z be finite, disjoint sets, and let T be a subset of $X \times Y \times Z$. That is, T consists of triples (x, y, z) such that $x \in X, y \in Y$, and $z \in Z$. Now $M \subseteq T$ is a 3-dimensional matching if the following holds: for any two distinct triples $(x_1, y_1, z_1) \in M$ and $(x_2, y_2, z_2) \in M$, we have $x_1 \neq x_2$, $y_1 \neq y_2$, and $z_1 \neq z_2$. M is called a *maximal* 3-dimensional matching if it can not be extended by adding more triplets from T. If M maximizes |M| then it is called a *maximum* 3-dimensional matching.

Independent set

http://en.wikipedia.org/wiki/Maximum_independent_set#Finding_maximum_independent_sets
Ett set av vertices i en graf av vilka inga är grannar.

Vertex cover

http://en.wikipedia.org/wiki/Vertex_cover
Välj ut en så liten mängd som möjligt av vertices så att varje edge i grafen anknyter till någon vertex i mängden.

3-satisfiability (3-SAT)

Hamiltonian path

En path där varje nod i grafen besöks en och endast en gång.

Traveling salesman

Hitta den kortaste möjliga Hamiltonian vägen.