CS483 Design and Analysis of Algorithms

Review: Chapters 4 - 8, DPV

Instructor: Fei Li

lifei@cs.gmu.edu with subject: CS483

Office hours: STII, Room 443, Friday 4:00pm - 6:00pm or by appointments

Course web-site:

http://www.cs.gmu.edu/∼lifei/teaching/cs483_fall08/

Figures unclaimed are from books "Algorithms" and "Introduction to Algorithms"

Announcements

1. Pick up solutions of assignment 8.
2. I add office hours: 2:00pm - 6:00pm December 5th, Friday.
3. The final exam is scheduled on December 10th, Wednesday. 1:30pm - 4:15pm. Innovation Hall 136 (this classroom). Wish you luck in all your finals!
4. You are allowed to have one-page cheat sheet (hand-written, front-and/or-back). No calculator. Closed textbook.
5. I will bring scratch paper for use.
6. Please take 5-min to fill in the evaluation form and return it to the TA, Cynthia. Thank you!

Final Covers:

1. 4.2, 4.4, 4.6
2. 5.1, 5.2
3. 6.1, 6.2, 6.3, 6.4
4. 7.1, 7.2, 7.6
5. 8.1, 8.2, 8.3

Requirements:

1. Definitions: P, NP, NP-complete, flow, cut
2. Algorithms:
    1. Dijkstra & Bellman-Ford (greedy)
    2. Kruskal & Prim (greedy)
    3. Huffman coding (greedy)
    4. Dynamic programming, knapsack (using dynamic programming approaches) (dynamic programming)
    5. linear formulation, linear programming in geometric interpretation
    6. maximum-flow min-cut (iterative approach)
    7. proving NP-completeness (reduction)
    8. (the simplex algorithms)
3. If necessary, I will give some well-known NPC problems for your use in your reduction.

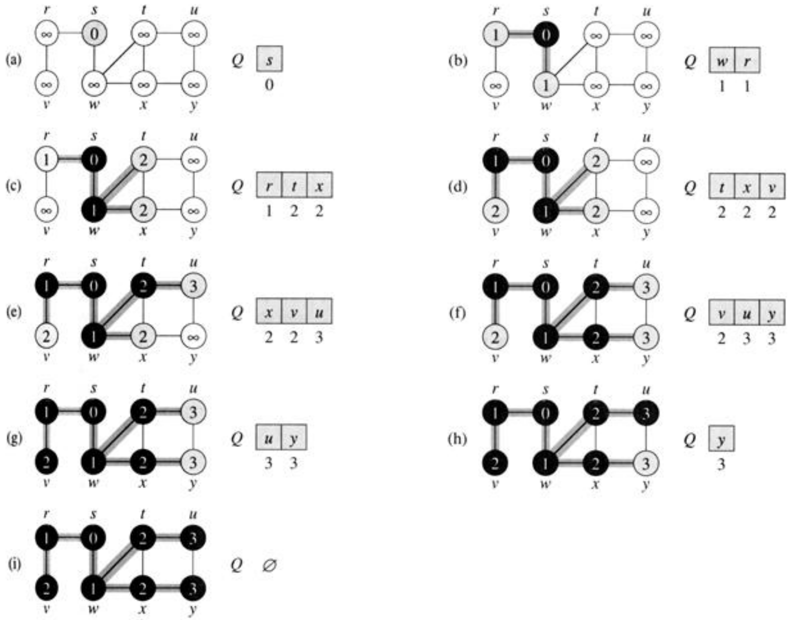
Chapter 4: Paths in Graphs

1. Breath-First Search
2. Dijkstra's Algorithm
3. Shortest path with negative edges

Breath-First Search

```
procedure bfs(G, s)
Input:  Graph G = (V, E), directed or undirected; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
   u = eject(Q)
   for all edges (u, v) ∈ E:
      if dist(v) = ∞:
         inject(Q, v)
         dist(v) = dist(u) + 1
```

Breath-First Search



Analysis of BFS

Theorem

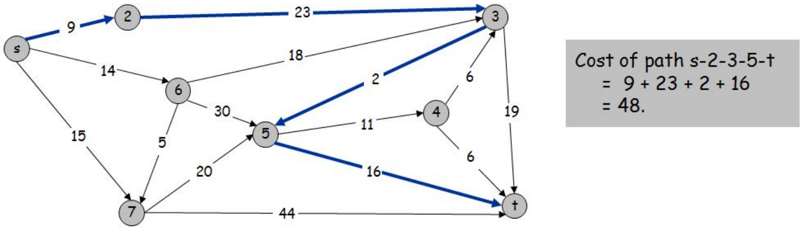
BFS *runs in* $O(m+n)$ *time if the graph is given by its adjacency representation,* n *is the number of nodes and* m *is the number of edges*

Proof.

When we consider node u, there are $deg(u)$ incident edges $(u,\ v)$. Thus, the total time processing edges is $\sum_{u \in V} deg(u) = 2 \cdot m$ $\square$

Dijkstra's Algorithm

Annotate every edge $e \in E$ with a *length* l_e. If $e = (u, v)$, let $l_e = l(u, v) = l_{uv}$
Input: Graph $G = (V,\ E)$ whose edge lengths l_e are *positive integers*
Output: The shortest path from s to t



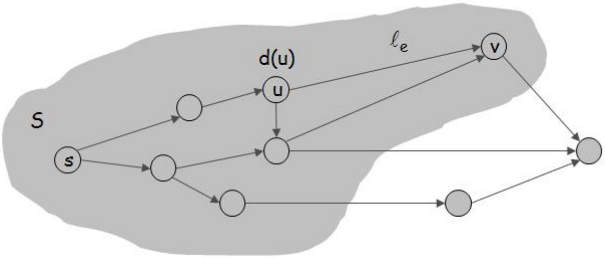
from Wayne's slides on "Algorithm Design"

Dijkstra's Algorithm

1. Maintain a set of explored nodes S for which we have determined the shortest path distance $d(u)$ from s to u
2. Initialize $S = \{s\}$, $d(s) = 0$
3. Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v), u \in S} d(u) + l_e$$

4. Add v to S, and set $d(v) = \pi(v)$



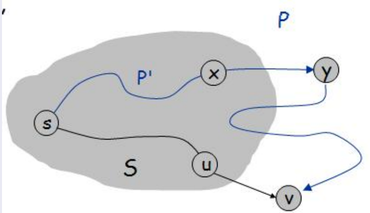
from Wayne's slides on "Algorithm Design"

Dijkstra's Algorithm

Theorem

Dijkstra's algorithm finds the shortest path from s to any node v: $d(v)$ is the length of the shortest $s \rightsquigarrow v$ path

Proof.



$\square$

from Wayne's slides on "Algorithm Design"

Theorem

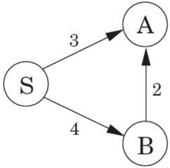
The overall running time of Dijkstra's algorithm is $O((|V| + |E|) \cdot \log |V|)$

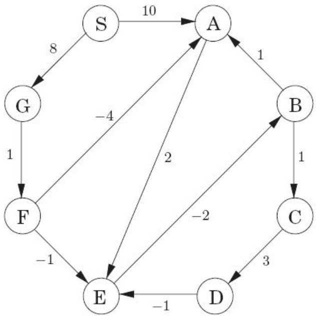
Show the figure in the textbook.

Shortest Paths in the Presence of Negative Edges

Simply update *all* the edges, $|V| - 1$ times

Dijkstra's algorithm will not work if there are negative edges





	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Chapter 5: Greedy Algorithms

1. Minimum Spanning Tree
2. Huffman Coding
3. Horn Formulas

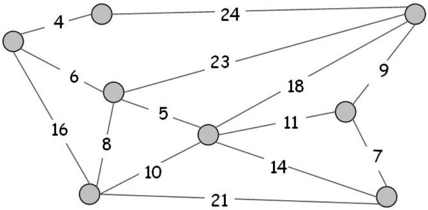
Greedy Approach

Idea. Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit

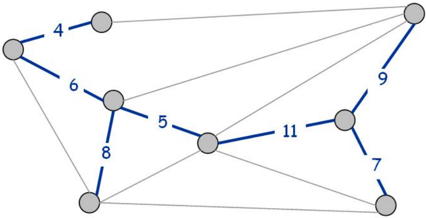
Minimum Spanning Tree

Definition

Minimum Spanning Tree (MST). Given a connected graph $G = (V, E)$ with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized



$G = (V, E)$



$T, \ \sum_{e \in T} c_e = 50$

from Wayne's slides on "Algorithm Design"

Greedy Algorithms

1. **Kruskal's algorithm**
   Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle
2. **Reverse-Delete algorithm**
   Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T
3. **Prim's algorithm**
   Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T

Kruskal's Algorithm

Figure 5.1 The minimum spanning tree found by Kruskal's algorithm.

Kruskal's Algorithm

Figure 5.1 The minimum spanning tree found by Kruskal's algorithm.

1. makeset(x): create a singleton set containing just x
2. find(x): to which set does x belong?
3. union(x, y): merge the set containing x and y

Kruskal's Algorithm

```
procedure kruskal (G, w)
Input:  A connected undirected graph G = (V, E) with edge weights w_e
output: A minimum spanning three defined by the edges X

for all u ∈ V:
   makeset (u)

X = {}
sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
    if find(u) ≠ find(v):
        add edge {u, v} to X
        union(u, v)
```

Running time $= |V|$ `makeset` $+ 2 \cdot |E|$ `find` $+ (|V| - 1)$ `union`

Correctness of Greedy Algorithm

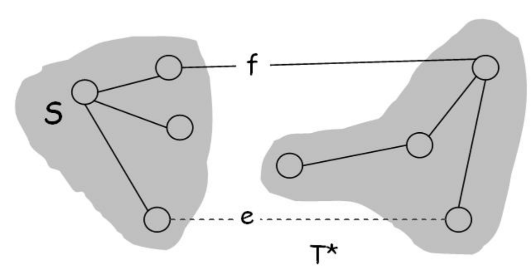
Definition

Cut. A *cut* is any partition of the vertices into two groups, S and $V - S$

Lemma

Let S be any subset of nodes, and let e be the min-*cost edge with exactly one endpoint in S. Then the MST contains e*

Proof.



□

from Wayne's slides on "Algorithm Design"

Correctness of Greedy Algorithm

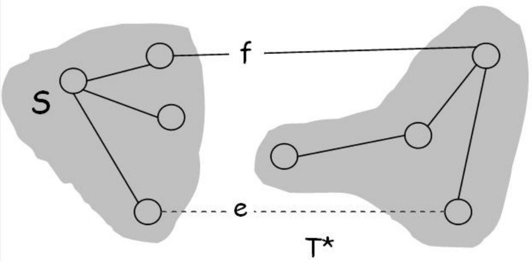
Definition

Cycle. Set of edges the form $(a, b), (b, c), (c, d), \ldots, (y, z), (z, a)$

Lemma

Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST does not contain f

Proof.



$\square$

from Wayne's slides on "Algorithm Design"

Prim's Algorithm

1. Initialize S = any node
2. Apply cut property to S
3. Add min-cost edge in cut-set corresponding to S to T, and add one new explored node u to S

Figure 5.1 The minimum spanning tree found by Kruskal's algorithm.

Huffman Coding

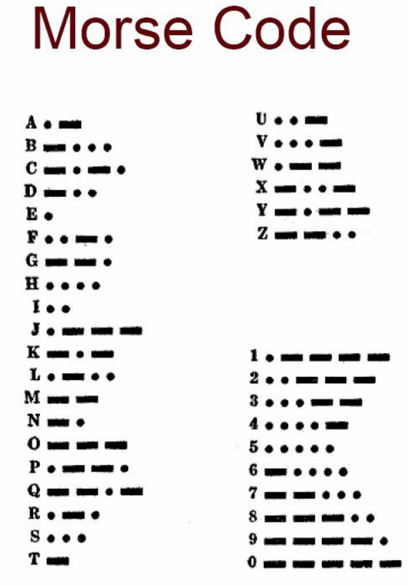


Image adapted from Wikipedia.

Huffman Coding

Definition

Prefix-free. No codeword can be a prefix of another codeword

$\{0,\ 01,\ 11,\ 001\}$?

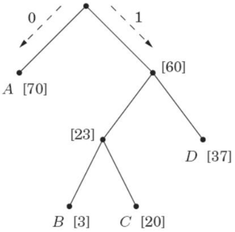
Remark

Any prefix-free encoding can be represented by a full *binary tree.*

$\{0,\ 100,\ 101,\ 11\}$?

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets

Symbol	Codeword
A	0
B	100
C	101
D	11

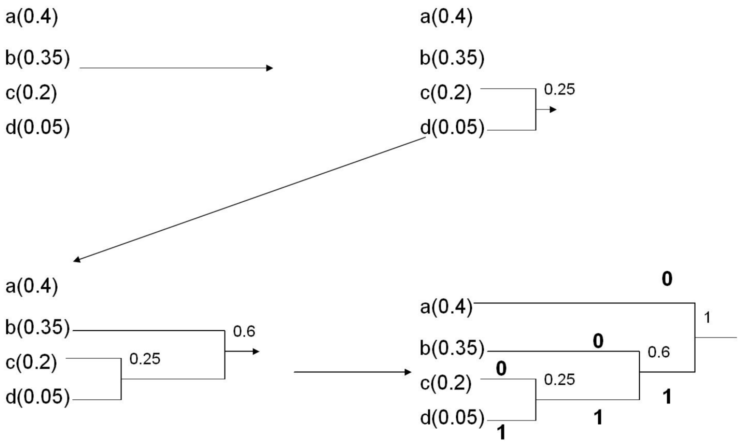


$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth of the } i\text{th symbol in tree})$$

Huffman Coding

```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output: An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n: insert(H, i)
for k = n+1 to 2n−1:
   i = deletemin(H),  j = deletemin(H)
   create a node numbered k with children i, j
   f[k] = f[i] + f[j]
   insert(H, k)
```

Huffman Coding



http://rio.ecs.umass.edu/ gao/ece665_08/slides/Rance.ppt

Horn Formula

The most primitive object in a Horn formula is a *Boolean variable*, taking value either `true` or `false`
A *literal* is either a variable x or its negation $\bar{x}$
There are two kinds of *clauses* in Horn's formulas

1. *Implications*

$$(z \wedge w) \Rightarrow u$$

2. Pure *negative clauses*

$$\bar{u} \vee \bar{v} \vee \bar{y}$$

Questions. To determine whether there is a consistent explanation: an assignment of `true`/`false` values to the variables that satisfies all the clauses

Satisfying Assignment

Input: A Horn formula
Output: A satisfying assignment, if one exists

```
function horn

    set all variables to false;

    while (there is an implication that is not satisfied)
        set the right-hand variable of the implication to true;

    if (all pure negative clauses are satisfied)
        return the assignment;
    else
        return ``formula is not satisfiable'';
```

$$(w \wedge y \wedge z) \Rightarrow x, (x \wedge z) \Rightarrow w, x \Rightarrow y, \Rightarrow x, (x \wedge y) \Rightarrow w, (\bar{w} \vee \bar{x} \vee \bar{y}), \bar{z}$$

Chapter 6: Dynamic Programming

1. Shortest Path
2. Longest Increasing Subsequences
3. Edit Distance
4. Knapsack

Algorithmic Paradigms

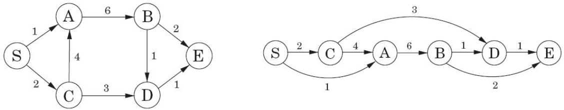
1. **Greed**
   Build up a solution incrementally, optimizing some local criterion in each step
2. **Divide-and-conquer**
   Break up a problem into two sub-problems, solve each sub-problem *independently*, and combine solution to sub-problems to form solution to original problem
3. **Dynamic programming**
   Identify a collection of subproblems and tackling them one by one, smallest first, using the answers to smaller problems to help figure out larger ones, until the whole lot of them is solved

Shortest Paths in Directed Acyclic Graphs (DAG)

Remark

The special distinguishing feature of a DAG is that its node can be linearized.

Figure 6.1 A dag and its linearization (topological ordering).




```
initialize all dist(·) values to ∞
dist(s) = 0
for each v ∈ V\{s}, in linearized order:
    dist(v) = min_{(u,v)∈E}{dist(u) + l(u,v)}
```

$$dist(D) = \min\{dist(B) + 1,\ dist(C) + 3\}.$$

This algorithm is solving a collection of *subproblems*, $\{dist(u) : u \in V\}$. We start with the smallest of them, $dist(s) = 0$.

Some Thoughts on Dynamic Programming

Remark

1. *In dynamic programming, we are not given a DAG; the DAG is implicit.*
2. *Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: If to solve subproblem B we need to answer the subproblem A, then there is a (conceptual) edge from A to B. In this case, A is thought of as a smaller subproblems than B — and it will always be smaller, in an obvious sense.*

Problem

How to solve/calculate above subproblems' values in Dynamic Programming?

Solution

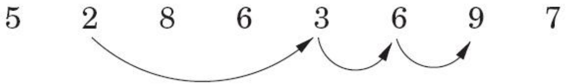
?

Longest Increasing Subsequences

Definition

The input is a sequence of numbers $a_1, \ldots, a_n$. A subsequence is any subset of these numbers taken in order, of the form $a_{i1}, a_{i2}, \ldots, a_{ik}$ where $1 < i_1 < i_2 < \ldots < i_k \leq n$, and an *increasing subsequence* is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length.

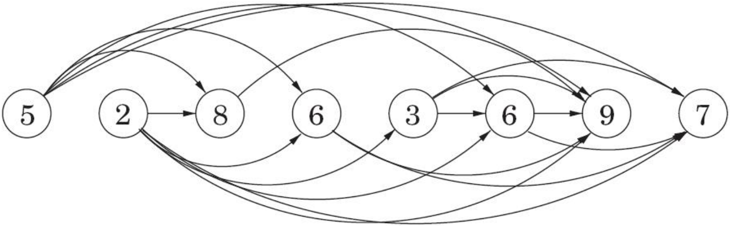
5, 2, 8, 6, 3, 6, 9, 7 is 2, 3, 6, 9:



Longest Increasing Subsequences

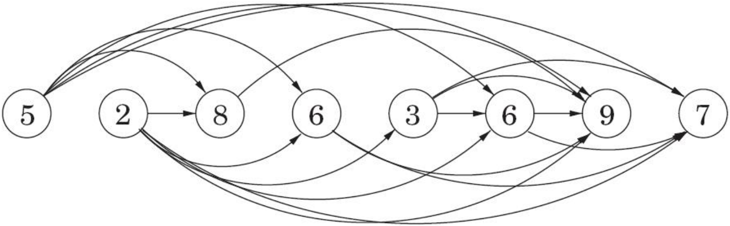
Longest Increasing Subsequences

Figure 6.2 The dag of increasing subsequences.



Longest Increasing Subsequences

1 **Figure 6.2** The dag of increasing subsequences.



2 function inc-subsequence(G = (V, E))

```
for j = 1, 2, ... n
    L(j) = 1 + maxL(i): (i, j) ∈ E;

return max_j L(j);
```

Longest Increasing Subsequences

Remark

There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

Theorem

The algorithm runs in polynomial time $O(n^2)$.

Proof.

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}.$$

□

Problem

Why not using recursion? For example,

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$

Solution

Bottom-up versus (top-down + divide-and-conquer).

Edit Distance

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	T	A	C	A	T

$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

-	C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	-	T	A	C	A	T

$2\delta + \alpha_{CA}$

from Wayne's slides on "Algorithm Design"

Problem

If we use the brute-force method, how many alignments do we have?

Solution

?

Edit Distance

1. **Goal**. Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$, find alignment of minimum cost. Call this problem $E(m, n)$.
2. **Subproblem** $E(i, j)$.
   Define $diff(i, j) = 0$ if $x[i] = y[j]$ and $diff(i, j) = 1$ otherwise

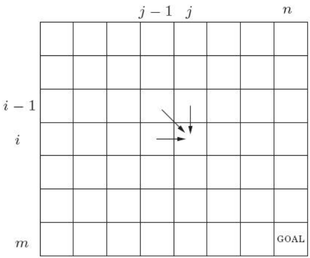
$$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), diff(i, j) + E(i-1, j-1)\}$$

```
function edit-distance(X, Y)

    for i = 0, 1, 2, ... m
        E(i, 0) = i;

    for j = 1, 2, ... n
        E(0, j) = j;

    for i = 1, 2, ... m
        for j = 1, 2, ... n
            E(i, j) = min { E(i - 1, j) + 1, E(i, j - 1) + 1,
                            E(i - 1, j - 1) + diff(i, j) };

    return E(m, n);
```

Edit Distance

Figure 6.4 (a) The table of subproblems. Entries $E(i-1, j-1)$, $E(i-1, j)$, and $E(i, j-1)$ are needed to fill in $E(i, j)$. (b) The final table of values found by dynamic programming.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Theorem

`edit-distance` *runs in time* $O(m \cdot n)$

The Underlying DAG

Remark

Every dynamic program has an underlying DAG structure: Think of each node as representing a subproblem, and each edge as a precedence constraint on the order in which the subproblems can be tackled.
Having nodes $u_1, \ldots, u_k$ point to v means "subproblem v can only be solved once the answers to $u_1, u_2, \ldots, u_k$ are known".

Remark

Finding the right subproblems takes creativity and experimentation.

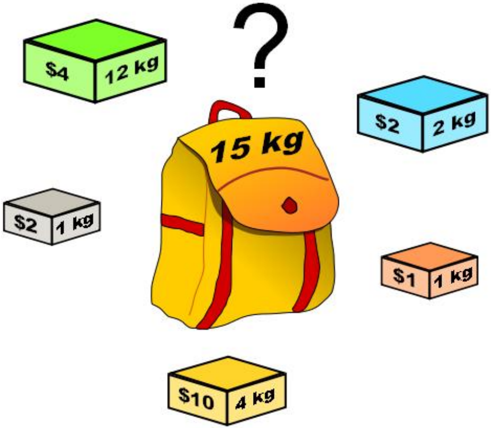
Solving Problems Using Dynamic Programming Approach

1. What is a subproblem?
   Can you define it clearly?
2. What is the relation between a smaller-size subproblem and a larger-size subproblem?
   Can we get the solution of the larger one from the smaller one?
   What is the dependency between them?
   What is the "DAG"?
   Is there a relationship between the optimality of a smaller subproblem and a larger subproblem?
3. How to solve this problem?
   What is the running-time complexity?

Knapsack

Knapsack Problem

Given n objects, each object i has weight w_i and value v_i, and a knapsack of capacity W, find most valuable items that fit into the knapsack



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

Knapsack Problem

Subproblem:

$K(w, j) =$ maximum value achievable using a knapsack of capacity w and items $1, 2, \ldots, j$

Goal: $K(W, n)$

```
function knap-sack(W, S)

    Initialize all K(0, j) = 0 and all K(w, 0) = 0;

    for j = 1 to n
        for w = 1 to W
            if (w_j > w)
                K(w, j) = K(w, j - 1);
            else
                K(w, j) = max{K(w, j - 1), K(w - w_j, j - 1) + v_j};

    return K(W, n);
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

from Wayne's slides on "Algorithm Design"

Traveling Salesman Problems

Definition

(TSP). Start from his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returned home. Given the pairwise distance between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Figure:

http://watching-movies-online.blogspot.com/2008/09/watch-mr-beans-holiday-hindi-dubbed.html

Traveling Salesman Problems

Definition

(TSP). Start from his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returned home. Given the pairwise distance between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Subproblem.
Let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.
Relation.

$$C(S, j) = \min_{i \in S, i \neq j} C(S - \{j\}, i) + d_{ij}.$$

```
C({1}, 1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1, 2, ..., n} of size s and containing 1:
        C(S, 1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S, j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1, ..., n}, j) + d_j1
```

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 2^n)$.

One of the Top 10 Algorithms in the 20th Century!

1. Formulate a problem using a linear program (Section 7.1)
2. Solve a linear program using the simplex algorithm (Section 7.6)
3. Applications: flows in networks; bipartite matching; zero-sum games (Sections 7.2 - 7.5)

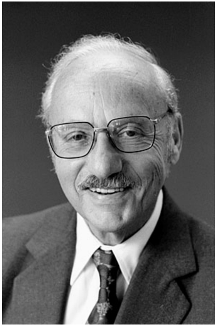


Figure: Father of Linear Programming and Simplex Algorithm: George Dantzig (1914 - 2005)

from http://en.wikipedia.org/wiki/George_Dantzig

Warm Up

Definition

Linear programming deals with satisfiability and optimization problems for linear constraints.

Definition

A linear constraint is a relation of the form

$$a_1 \cdot x_1 + \ldots + a_n \cdot x_n = b,$$

or

$$a_1 \cdot x_1 + \ldots + a_n \cdot x_n \leq b \quad \text{or} \quad a_1 \cdot x_1 + \ldots + a_n \cdot x_n \geq b,$$

where the a_i and b are constants and the x_i are the unknown variables.

Definition

Satisfiability: Given a set of linear constraints, is there a value $(x_1, \ldots, x_n)$ that satisfies them all?

Definition

Optimization: Given a set of linear constraints, assuming there is a value $(x_1, \ldots, x_n)$ that satisfies them all, find one which maximizes (or minimizes)

$$c_1 \cdot x_1 + \ldots + c_n \cdot x_n.$$

A Toy Example without Necessity of Calculation – from Eric Schost's Slides

Problem

You are allowed to share your time between two companies

1. *company C_1 pays 1 dollar per hour;*
2. *company C_2 pays 10 dollars per hour.*

Knowing that you can only work up to 8 hours per day, what schedule should you go for?

Of course, work full-time at company C_2.

1. **Linear formulation**:
   x_1 is the time spent at C_1 and x_2 the time spent at C_2.
2. **Constraints**:
   $$x_1 \geq 0, \ x_2 \geq 0, \ x_1 + x_2 \leq 8.$$
3. **Objective function**:
   $$\max \quad x_1 + 10 \cdot x_2.$$
4. **Solution**:
   $$x_1 = 0, \ x_2 = 8.$$

Another Example With Geometrical Solution

Problem

Two products are produced: A and B. Per day, we make x_1 of A with a profit of 1 each, we make x_2 of B with profit 6. $x_1 \leq 200$ and $x_2 \leq 300$, and the total A and B is no more than 400. What is the best choice of x_1 and x_2 at maximizing the profit?

$$
\begin{array}{rrcl}
\textbf{Objective}: \ \max & x_1 & + & 6 \cdot x_2 \\
\textbf{Subject to}: & x_1 & \leq & 200 \\
& x_2 & \leq & 300 \\
& x_1 + x_2 & \leq & 400 \\
& x_1, x_2 & \geq & 0
\end{array}
$$

Definition

The points that satisfy a single inequality are in a half-space.

Definition

The points that satisfy several inequalities are in the intersection of half-spaces. The intersection of (finitely many) half-spaces is a convex polygon (2D) — the feasible region.

Any Algorithmic Observation?

Definition

An extreme point p is impossible to be expressed as a convex combination of two other distinct points in the convex polygon.

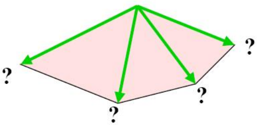
Theorem

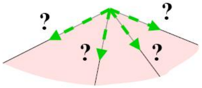
The optimal solution, if it exists, is at some extreme point p.

1. A naive algorithm (expensive!):
   1. List all the possible vertices.
   2. Find the optimal vertex (the one with the maximal value of the objective function).
   3. Try to figure out whether it is a global maximum.
2. Our approach (the simplex algorithm):
   1. Start at some extreme point.
   2. Pivot from one extreme point to a neighboring one.
   3. Repeat until optimal.

The Simplex Algorithm — Sketch

1. Start at some extreme point v_1.
2. Pivot from one extreme point v_1 to a neighboring one v_2.





   1. v_2 should increase the value of the objective function.
   2. Several strategies are available to select v_1.
3. Repeat until optimal — reach a vertex where no improvement is possible.

Correctness?

Complexity analysis?

The Simplex Algorithm

Consider a generic LP

$$\begin{array}{rcl} \max & & \vec{c}^T \vec{x} \\ \mathbf{A}\vec{x} & \leq & \vec{b} \\ \vec{x} & \geq & 0 \end{array}$$

One each iteration, simplex has two tasks:

1. Check whether the current vertex is optimal (and if so, halt).
2. Determine where to move next.

    1. Move from the origin by increasing some x_i for which $c_i > 0$. *Until we hit some other constraint*.

    That is, we release the tight constraint $x_i \geq 0$ and increase x_i until some other inequality, previously loose, now become tight. At that point, we are at a new vertex.

Remark

Both tasks are easy if the vertex happens to be at the origin. That is, if the vertex is elsewhere, we will transform the coordinate system to move it to the origin.

Theorem

The objective is optimal when the coordinates of the local cost vector are all zero or negatives.

Simplex in Action

Initial LP:

$$\max\ 2x_1 + 5x_2$$

$$\begin{array}{rcll} 2x_1 - x_2 & \leq & 4 & ① \\ x_1 + 2x_2 & \leq & 9 & ② \\ -x_1 + x_2 & \leq & 3 & ③ \\ x_1 & \geq & 0 & ④ \\ x_2 & \geq & 0 & ⑤ \end{array}$$

Current vertex: {④, ⑤} (origin).
Objective value: 0.

Move: increase x_2.
⑤ is released, ③ becomes tight. Stop at $x_2 = 3$.

New vertex {④, ③} has local coordinates (y_1, y_2):

$$y_1 = x_1, \quad y_2 = 3 + x_1 - x_2$$

Rewritten LP:

$$\max\ 15 + 7y_1 - 5y_2$$

$$\begin{array}{rcll} y_1 + y_2 & \leq & 7 & ① \\ 3y_1 - 2y_2 & \leq & 3 & ② \\ y_2 & \geq & 0 & ③ \\ y_1 & \geq & 0 & ④ \\ -y_1 + y_2 & \leq & 3 & ⑤ \end{array}$$

Current vertex: {④, ③}.
Objective value: 15.

Move: increase y_1.
④ is released, ② becomes tight. Stop at $y_1 = 1$.

New vertex {②, ③} has local coordinates (z_1, z_2):

$$z_1 = 3 - 3y_1 + 2y_2, \quad z_2 = y_2$$

Rewritten LP:

$$\max\ 15 + 7y_1 - 5y_2$$

$$\begin{array}{rcll} y_1 + y_2 & \leq & 7 & ① \\ 3y_1 - 2y_2 & \leq & 3 & ② \\ y_2 & \geq & 0 & ③ \\ y_1 & \geq & 0 & ④ \\ -y_1 + y_2 & \leq & 3 & ⑤ \end{array}$$

Current vertex: {④, ③}.
Objective value: 15.

Move: increase y_1.
④ is released, ② becomes tight. Stop at $y_1 = 1$.

New vertex {②, ③} has local coordinates (z_1, z_2):

$$z_1 = 3 - 3y_1 + 2y_2, \quad z_2 = y_2$$

Rewritten LP:

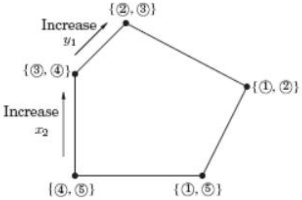
$$\max\ 22 - \tfrac{7}{3}z_1 - \tfrac{1}{3}z_2$$

$$\begin{array}{rcll} -\frac{1}{3}z_1 + \frac{5}{3}z_2 & \leq & 6 & ① \\ z_1 & \geq & 0 & ② \\ z_2 & \geq & 0 & ③ \\ \frac{1}{3}z_1 - \frac{2}{3}z_2 & \leq & 1 & ④ \\ \frac{1}{3}z_1 + \frac{1}{3}z_2 & \leq & 4 & ⑤ \end{array}$$

Current vertex: {②, ③}.
Objective value: 22.

Optimal: all $c_i < 0$.

Solve ②, ③ (in original LP) to get optimal solution $(x_1, x_2) = (1, 4)$.

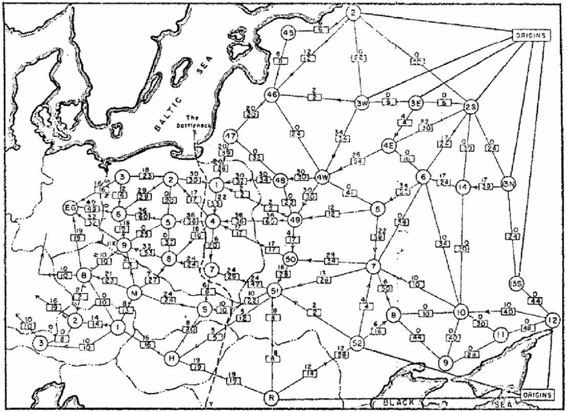


Complexity of the Simplex

1. **Worst case**.
   One can construct examples where the simplex algorithm visits all vertices (which can be exponential in the dimension and the number of constraints).
2. **Most cases**.
   The simplex algorithm works very well.

Flows in Networks

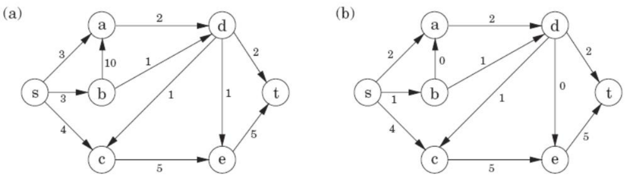
Soviet Rail Network, 1955



Reference: *On the history of the transportation and maximum flow problems.* Alexander Schrijver in Math Programming, 91: 3, 2002.

Flows in Networks

Figure 7.4 (a) A network with edge capacities. (b) A *flow* in the network.



Definition

Consider a directed graph $G = (V, E)$; two specific nodes $s, t \in V$. s is the *source* and t is the *sink*. The *capacity* $c_e > 0$ of an edge e.

Definition

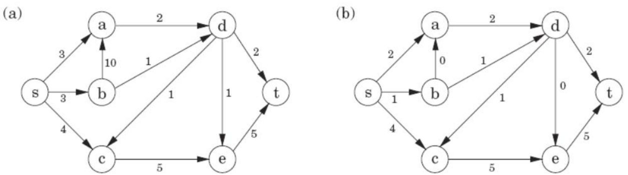
Flow. A particular shipping scheme consisting a variable f_e for each edge e of the network, satisfying the following two properties:

1. $0 \leq f_e \leq c_e, \forall e \in E$.
2. For all nodes $u \neq s, t$, the amount of flow entering u equals the amount leaving u (i.e., flows are conservative):

$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}.$$

Flows in Networks

Figure 7.4 (a) A network with edge capacities. (b) A *flow* in the network.



Definition

Size of a flow. The total quantity sent from s to t, i.e., the quantity leaving s:

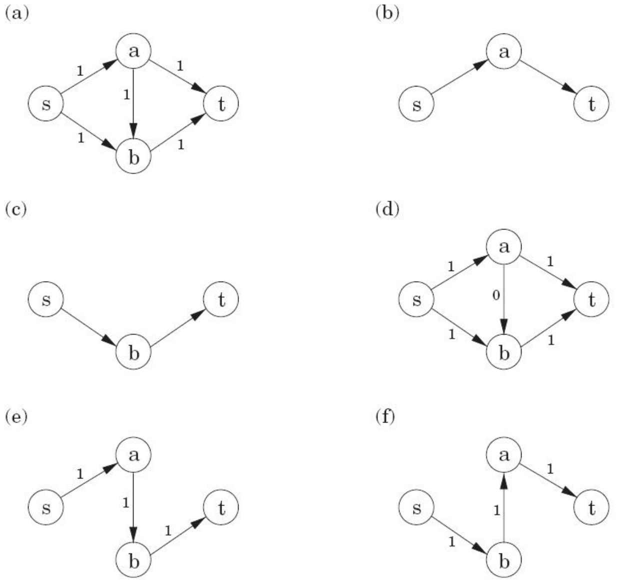
$$size(f) := \sum_{(s,u)\in E} f_{su}.$$

$$\begin{aligned}
\max size(f) \quad &:= \quad \sum_{(s,u)\in E} f_{su} \\
\text{subject to} \quad & \\
0 \quad \leq \quad f_e \quad &\leq \quad c_e, \quad \forall e \in E \\
\sum_{(w,u)\in E} f_{wu} \quad &= \quad \sum_{(u,z)\in E} f_{uz}, \quad u \neq s, t
\end{aligned}$$

Using the Interpretation of the Simplex Algorithm

1. Start with a zero flow.
2. Repeat: Choose an appropriate path from s to t, and increase flow along the edges of this path as much as possible.

Figure 7.5 An illustration of the max-flow algorithm. (a) A toy network. (b) The first path chosen. (c) The second path chosen. (d) The final flow. (e) We could have chosen this path first. (f) In which case, we would have to allow this second path.



Using the Interpretation of the Simplex Algorithm

1. Start with a zero flow.
2. Repeat: Choose an appropriate path from s to t, and increase flow along the edges of this path as much as possible. In each iteration, the simplex looks for an $s - t$ path whose edge (u, v) can be of two types:
    1. (u, v) is in the original network, and is not yet at full capacity. If f is the current flow, edge (u, v) can handle up to $c_{uv} - f_{uv}$ additional units of flow.
    2. The reverse edge (v, u) is in the original network, and there is some flow along it. Up to f_{vu} additional units (i.e., canceling all or part of the existing flow on (v, u)).

Definition

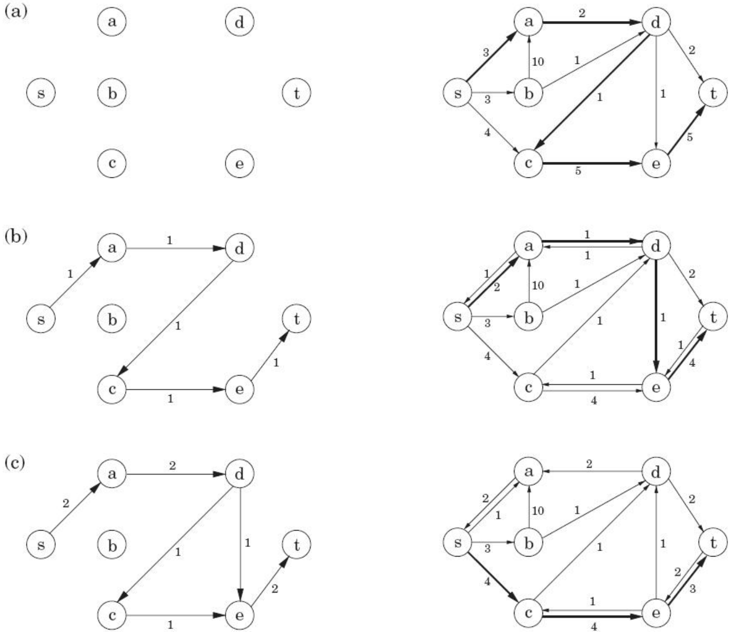
Residual network $G^f = (V, E^f)$. G^f has exactly the two types of edges listed, with residual capacity c^f:

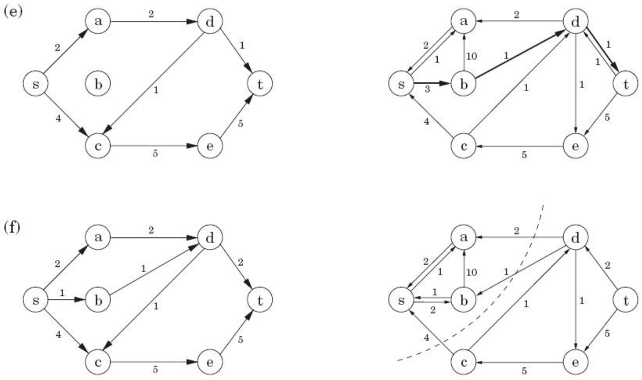
$$c^f := \begin{cases} c_{uv} - f_{uv}, & \text{if } (u, v) \in E \text{ and } f_{uv} < c_{uv} \\ f_{vu}, & \text{if } (v, u) \in E \text{ and } f_{vu} > 0 \end{cases} \tag{1}$$

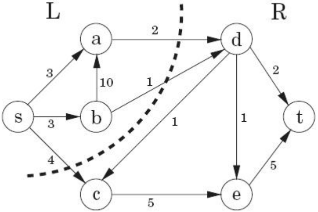
Definition

Augmenting path. An augmenting path p is a simple path from s to t in the residual network G^f.

Current flow | Residual graph




(e)
(f)


L
R

Flows in Networks

Definition

Cuts. A $s - t$ cut partitions the vertices into two disjoint groups L and R such that $s \in L$ and $t \in R$. Its *capacity* is the total capacity of the edges from L to R, and it is an upper bound on *any* flow from s to t.

Theorem

Max-flow min-cut theorem. *The size of the maximum flow in a network equals the capacity of the smallest (s, t)-cut.*

Proof.

? □

Theorem

The running time of the augmentation-flow algorithm is $O(|V| \cdot |E|^2)$ over an integer-value graph.

Proof.

? □

Chapter 8: Overview

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

Some Typical Hard Problems

1. Satisfiability
2. Traveling Salesman Problem
3. Independent Set, Vertex Cover, and Cliques
4. Knapsack and Subset Sum

Satisfiability — **SAT**

Definition

Literal: a Boolean variable x or $\bar{x}$

Definition

Disjunction: logical *or*, denoted $\wedge$

Definition

Clause: e.g., Boolean formula in conjunctive normal form (CNF)

$$(x \wedge y \wedge z)(x \wedge \bar{y})(y \wedge \bar{z})(z \wedge \bar{x})(\bar{x} \wedge \bar{y} \wedge \bar{z})$$

Definition

Satisfying truth assignment. An assignment of `false` or `true` to each variable so that every clause it to evaluate is `true`

Lemma

For formulas with n variables, we can find the answer in time 2^n. In a particular case, such assignment may not exist.

Proof.

? □

Satisfiability — **SAT**

Definition

Literal: a Boolean variable x or $\bar{x}$

Definition

Disjunction: logical *or*, denoted $\wedge$

Definition

Clause: e.g., Boolean formula in conjunctive normal form (CNF)

$$(x \wedge y \wedge z)(x \wedge \bar{y})(y \wedge \bar{z})(z \wedge \bar{x})(\bar{x} \wedge \bar{y} \wedge \bar{z})$$

Definition

Satisfying truth assignment. An assignment of `false` or `true` to each variable so that every clause it to evaluate is `true`

Lemma

In a particular case — Horn formula, *a satisfying truth assignment, if one exists, can be found by (?) in polynomial time*

Lemma

In a particular case (each clause has only two literals), **SAT** *can be solved in polynomial time (linear, quadratic, etc?) by (?) algorithm*

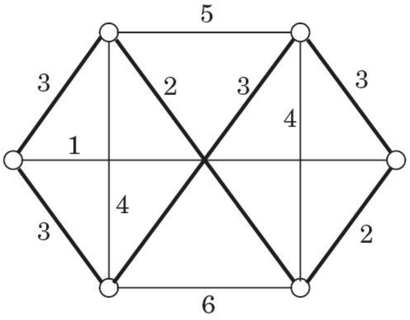
The Traveling Salesman Problem — **TSP**

Definition

TSP. We are given n vertices $1, 2, \ldots, n$ and all $(n \cdot (n-1))/2$ distances between them, as well as a budget b. We are asked to find a *tour*, a cycle that passes through every vertex exactly once, of total cost b or less — or to report that no such tour exists

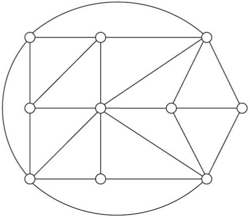
$$d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \ldots + d_{\tau(n),\tau(1)} \leq b$$

The optimal traveling salesman tour, shown in bold, has length 18



Independent Set, Vertex Cover, and Clique

What is the size of the largest independent set in this graph?



Definition

Independent Set. Find g vertices that are independent, i.e., no two of them have an edge between them

Definition

Vertex Cover. Find b vertices that cover every edge

Definition

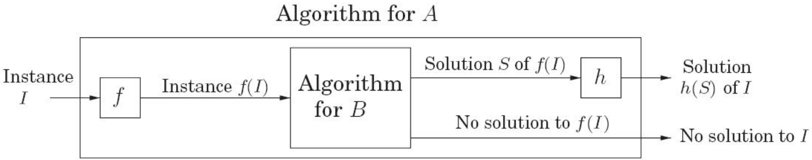
Clique. Find a set of g vertices such that all possible edges between them are present

Reduction

Definition

Problem A polynomial reduces to problem B if arbitrary instances of problem A can be solved using

1. Polynomial number of standard computational steps, plus
2. Polynomial number of calls to oracle that solves problem B



P, NP and NP-Complete

Definition

Search problems. Any proposed solution can be quickly (in polynomial time of the input size) checked for correctness

Definition

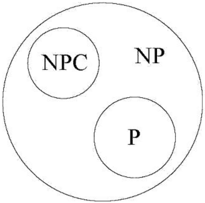
P. The class of all search problems that can be solved in polynomial time

Definition

NP. The class of all search problems

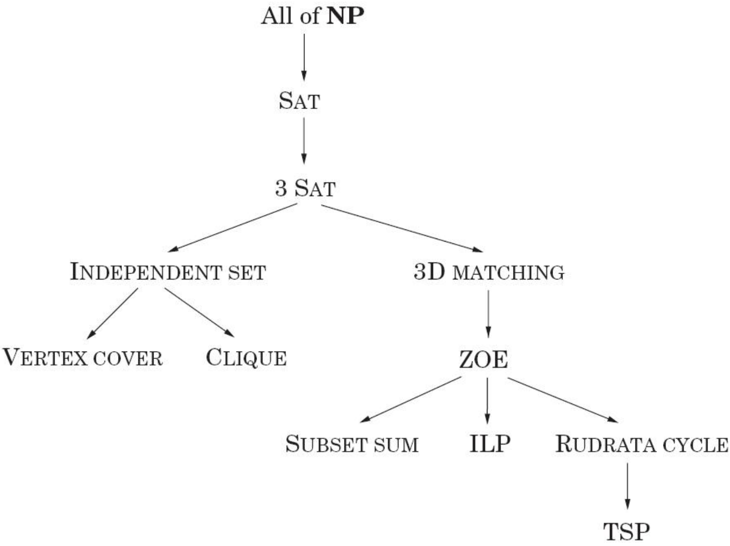
Definition

NP-complete. A problem is NP-complete if all other search problems reduce to it. (A hardest search NP problem.)



Reduction

Figure 8.7 Reductions between search problems.



3SAT $\rightarrow$ Independent Set

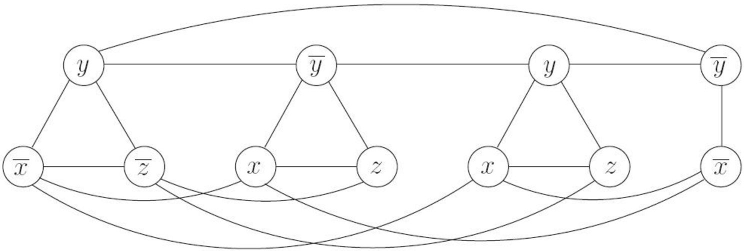
3SAT $\longrightarrow$ Independent Set

1. $$(\bar{x} \vee y \vee \bar{z})(x \vee \bar{y} \vee z)(x \vee y \wedge z)(\bar{x} \vee \bar{y})$$

3SAT $\longrightarrow$ Independent Set

1. $$(\bar{x} \vee y \vee \bar{z})(x \vee \bar{y} \vee z)(x \vee y \wedge z)(\bar{x} \vee \bar{y})$$

2. **Figure 8.8** The graph corresponding to $(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y})$

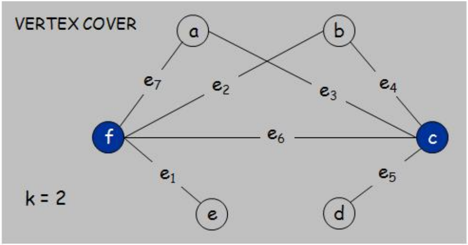


SAT $\rightarrow$ 3SAT

$$(a_1 \vee a_2 \vee \ldots \vee a_k) \rightarrow (a_1 \vee a_2 \vee y_1)(\bar{y}_1 \vee a_3 \vee y_2)(\bar{y}_2 \vee a_4 \vee y_3) \cdots (\bar{y}_{k-3} \vee a_{k-1} \vee a_k)$$

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Set Cover → Vertex Cover



SET COVER

$U = \{1, 2, 3, 4, 5, 6, 7\}$
$k = 2$

$S_a = \{3, 7\}$ $S_b = \{2, 4\}$
$S_c = \{3, 4, 5, 6\}$ $S_d = \{5\}$
$S_e = \{1\}$ $S_f = \{1, 2, 6, 7\}$

from Wayne's slides on "Algorithm Design"

Beyond NP-hard

Theorem

Any problem in NP $\rightarrow$ SAT

Proof.

? $\square$

Theorem

There exists algorithms running in exponential time for NP problems

```
function paradox(z: file)

    1: if terminates(z, z)
        go to 1;
```

Theorem

Some problems do not have algorithms

For example, find out x, y, z to satisfy

$$x^3yz + 2y^4z^2 - 7xy^5z = 6.$$