



Algorithm Design XV

NP Problem I

Guoqiang Li
School of Software



SHANGHAI JIAO TONG
UNIVERSITY

Efficient Problems, Difficult Problems



We have developed algorithms for

- FINDING SHORTEST PATHS IN GRAPHS,
- MINIMUM SPANNING TREES IN GRAPHS,
- MATCHINGS IN BIPARTITE GRAPHS,
- MAXIMUM INCREASING SUBSEQUENCES,
- MAXIMUM FLOWS IN NETWORKS,
-

All these algorithms are **efficient**, since their time requirement grows as a **polynomial function** (such as n , n^2 , or n^3) of the size of the input.

Exponential Search Space



In these problems we are searching for a solution (path, tree, matching) from among an **exponential** population of possibilities.

All these problems could in principle be solved in **exponential time** by checking through all candidate solutions, one by one.

An algorithm with running time 2^n , or worse, is useless in practice.

The **efficient algorithms** is to find clever ways to bypass **exhaustive search**, using clues from the input to narrow down the search space.

Are there “**search problems**” in which seeking a solution among an **exponential** chaos, and the fastest algorithms for them are **exponential**?

MINIMUM SPANNING TREES

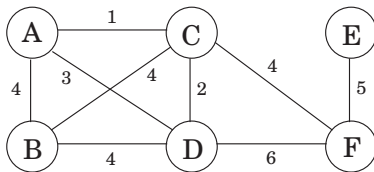
Build a Network



Suppose you are asked to **network** a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a **maintenance cost**.



A General Kruskal's Algorithm



```
 $X = \{ \};$   
repeat until  $|X| = |V| - 1;$   
    pick a set  $S \subset V$  for which  $X$  has no edges between  $S$  and  $V - S;$   
    let  $e \in E$  be the minimum-weight edge between  $S$  and  $V - S;$   
     $X = X \cup \{e\};$ 
```

A Little Change of the MST



SHANGHAI JIAO TONG
UNIVERSITY

WHAT IF THE TREE IS NOT ALLOWED TO **BRANCH**?

SATISFIABILITY PROBLEM

The instances of **SATISFIABILITY** or **SAT**:

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

a Boolean formula in conjunctive normal form (CNF).

It is a collection of clauses (the parentheses),

- each consisting of the disjunction of several literals;
- a literal is either a Boolean variable or the negation of one.

A satisfying truth assignment is an assignment of **false** or **true** to each variable so that every clause contains a literal of **true**.

Given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

Given a set of clauses, where each clause is the disjunction of two literals, looking for an assignment so that all clauses are satisfied.

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3) \wedge (x_1 \vee \bar{x}_2) \wedge (x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4)$$

Given an instance I of 2-SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I , G_I has an edge from the negation of α to β , and one from the negation of β to α .



Show that if G_I has a strongly connected component containing both x and \bar{x} for some variable x , then I has no satisfying assignment.

If none of G_I 's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable.

Conclude that there is a linear-time algorithm for solving 2-SAT.

A Little Change of the 2-SAT



SHANGHAI JIAO TONG
UNIVERSITY

3-SAT, SAT?

Various Problem Definitions and Reductions

The instances of **SATISFIABILITY** or **SAT**:

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

a Boolean formula in conjunctive normal form (CNF).

It is a collection of clauses (the parentheses),

- each consisting of the disjunction of several literals;
- a literal is either a Boolean variable or the negation of one.

A satisfying truth assignment is an assignment of **false** or **true** to each variable so that every clause contains a literal of **true**.

Given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

Decision Problem and Search Problem



SHANGHAI JIAO TONG
UNIVERSITY

Decision problem. Does there **exist** an assignment to satisfy the formula in CNF?

Search problem. **Find** an assignment to satisfy the formula.

Decision VS. Search



Turning an **decision problem** into a **search problem** does not change its difficulty, because the two versions reduce to one another.

The **decision problem** is a special version of the **search problem**.

The solution of the **search problem** is on the output tape of the Turing machine that encodes the **decision problem**.

For the SAT, we can ask the assignment of each variable one by one until all variables are finally assigned. The total execution time is bounded by the number of variables times the execution time of the decision problems.



SAT is a typical search problem.

We are given an instance I

- Some input data specifying the problem
- A Boolean formula in conjunctive normal form

we are asked to find a solution S

- An object that meets a particular specification
- An assignment that satisfies each clause

If no such solution exists, we must say so.



A search problem must have the property that any proposed solution S to an instance I can be quickly checked for correctness.

S must be concise, with length polynomially bounded by that of I .

- This is true for SAT, where S is an assignment to the variables.

There is a polynomial-time algorithm that takes as input I and S and decides whether or not S is a solution of I .

- For SAT, it is easy to check whether the assignment specified by S satisfies every clause in I .



A **search problem** is specified by an algorithm C that takes two inputs, an **instance** I and a **proposed solution** S , and runs in **time polynomial** in $|I|$.

We say S is a **solution** to I if and only if $C(I, S) = \text{true}$.



Researchers over the past 80 years have tried to find efficient ways to solve the SAT, but without success.

The fastest algorithms we have are still exponential on their worst-case inputs.

There are two natural variants of SAT with good algorithms.

- 2-SAT can be solved in linear time.
- All clauses contain at most one positive literal, say Horn formula, can be found by the greedy algorithm.

TRAVELING SALESMAN PROBLEM

The Traveling Salesman Problem

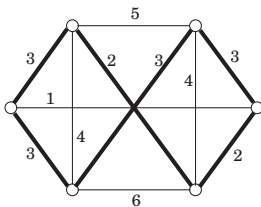


In the **traveling salesman problem (TSP)** we are given n vertices and all $n(n-1)/2$ distances between them, and a **budget b** .

To find a cycle that passes through every vertex exactly once, of total cost b or less - or to report that no such cycle.

A **permutation $\tau(1), \dots, \tau(n)$** of the vertices such that when they are toured in this order, the total distance covered is at most b :

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



Optimization Problem and Search Problem



SHANGHAI JIAO TONG
UNIVERSITY

Optimization problem. given an instance of TSP, find the minimum cost tour.

Search problem. given an instance of TSP, find a tour within the budget (or report that none exists).

Search VS. Optimization



Turning an **optimization problem** into a **search problem** does not change its difficulty, because the two versions reduce to one another.

Any algorithm that solves the **optimization** also solves the **search** problem:

- find the **optimum tour** and if it is within **budget**, return it; if not, there is no solution.

Conversely, an algorithm for the **search** problem can also be used to solve the **optimization** problem:

- First suppose that we knew the cost of the optimum tour; then we could find this tour by calling the algorithm for the **search** problem, using the **optimum** cost as the budget.
- We can find the optimum cost by **binary search**.

Search Instead of Optimization



Isn't any optimization problem also a search problem in the sense that we are searching for a solution that has the property of being **optimal**?

The solution to a **search problem** should be easy to recognize, or as we put it earlier, **polynomial-time** checkable.

Given a potential solution to the **TSP**, it is easy to check the properties "**is a tour**" (just check that each vertex is visited exactly once) and "**has total length $\leq b$** ."

But how could one check the property "**is optimal**"?



There are no known polynomial-time algorithms for the **TSP**, despite much effort by researchers over nearly a century.

There exists a faster, yet still exponential, **dynamic programming** algorithm.

The **MINIMUM SPANNING TREE (MST)** problem, for which we do have **efficient algorithms**, provides a stark contrast here.

The **TSP** can be thought of as a tough cousin of the **MST** problem, in which the tree is not allowed to **branch** and is therefore a **path**.

This extra restriction on the structure of the tree results in a much **harder problem**.

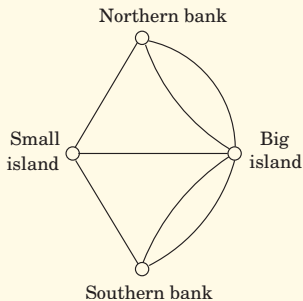
EULER AND RUDRATA

Euler Path



EULER PATH:

Given a graph, find a path that contains each edge exactly once



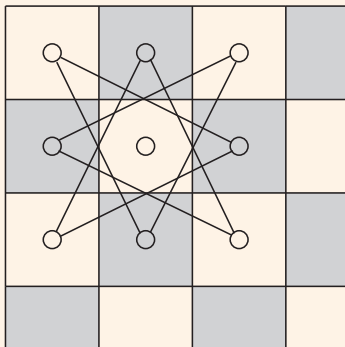


The answer is **yes** if and only if

- 1 the graph is **connected** and
- 2 every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has **even degree**.

There is a **polynomial time** algorithm for **EULER PATH**.

Rudrata Cycle



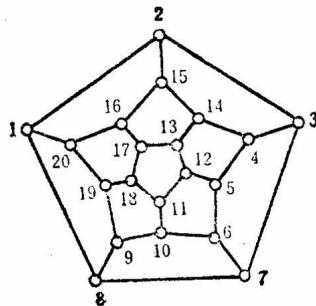
Rudrata Cycle



RUDRATA CYCLE:

Given a graph, find a cycle that visits each vertex exactly once.

In the literature this problem is known as the **Hamilton cycle problem**.



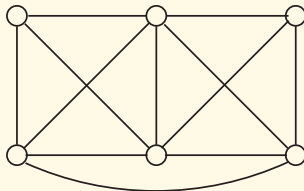
CUTS AND BISECTIONS

Minimum Cut



A **cut** is a set of edges whose removal leaves a graph disconnected.

MINIMUM CUT: given a graph and a budget b , find a cut with at most b edges.





This problem can be solved in polynomial time by $n - 1$ max-flow computations:

- give each edge a capacity of 1,
- and find the maximum flow between some fixed node and every single other node.

The smallest such flow will correspond (via the max-flow min-cut theorem) to the smallest cut.



In many graphs, the smallest cut leaves just a **singleton** vertex on one side - it consists of all edges **adjacent** to this vertex.

Far more interesting are small cuts that partition the vertices of the graph into nearly equal-sized sets.

BALANCED CUT: Given a graph with n vertices and a budget b , partition the vertices into two sets S and T such that $|S|, |T| \geq n/3$ and such that there are at most b edges between S and T .

INTEGER LINEAR PROGRAMMING



In a **LINEAR PROGRAMMING** problem we are given a set of **variables**, and to assign **real values** to them so as to

- 1 satisfy a set of **linear equations** and/or **linear inequalities** involving these variables, and
- 2 maximize or minimize a given **linear objective function**.



$$\max x_1 + 6x_2 + 13x_3$$

$$x_1 \leq 200$$

$$x_2 \leq 300$$

$$x_1 + x_2 + x_3 \leq 400$$

$$x_2 + 3x_3 \leq 600$$

$$x_1, x_2, x_3 \geq 0$$



INTEGER LINEAR PROGRAMMING (ILP): We are given a set of linear inequalities $A\mathbf{x} \leq \mathbf{b}$, where

- A is an $m \times n$ matrix and
- \mathbf{b} is an m -vector;
- an objective function specified by an n -vector \mathbf{c} ;
- a goal g .

We want to find a nonnegative integer n -vector \mathbf{x} such that $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{c} \cdot \mathbf{x} \geq g$.

Integer Linear Programming



SHANGHAI JIAO TONG
UNIVERSITY

$$\begin{aligned}\max & 2x_1 + 5x_2 \\ & 2x_1 - x_2 \leq 4 \\ & x_1 + 2x_2 \leq 9 \\ & -x_1 + x_2 \leq 3 \\ & x_1, x_2 \geq 0\end{aligned}$$

$$\begin{aligned}2x_1 + 5x_2 & \leq g \\ & 2x_1 - x_2 \leq 4 \\ & x_1 + 2x_2 \leq 9 \\ & -x_1 + x_2 \leq 3 \\ & x_1, x_2 \geq 0\end{aligned}$$

But there is a redundancy here:

- the last constraint $c \cdot x \geq g$ is itself a **linear inequality** and
- can be **absorbed** into $Ax \leq b$.



So, we define ILP to be following search problem:

Given A and b , find a nonnegative integer vector x satisfying the inequalities $Ax \leq b$.

3D-MATCHING

Bipartite Matching



SHANGHAI JIAO TONG
UNIVERSITY

BOYS

GIRLS

Al

Alice

Bob

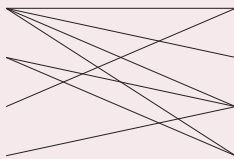
Beatrice

Chet

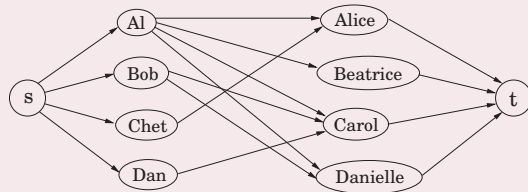
Carol

Dan

Danielle



Bipartite Matching



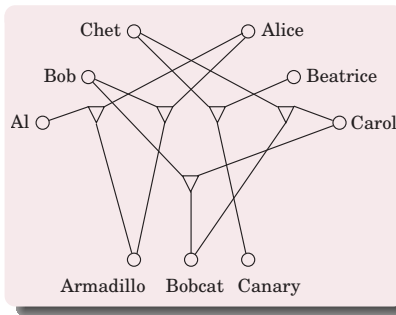
Three-Dimensional Matching



3D MATCHING: There are n boys, n girls, and n pets. The compatibilities are specified by a set of **triples**, each containing a boy, a girl, and a pet.

A triple (b, g, p) means that boy b , girl g , and pet p get along well together.

To find n disjoint triples and thereby create n harmonious households.



INDEPENDENT SET, VERTEX COVER, AND CLIQUE

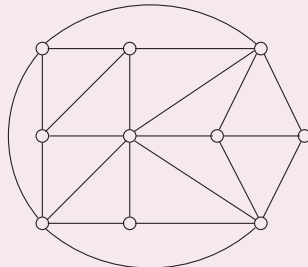


Independent Set, Vertex Cover, and Clique

INDEPENDENT SET: Given a graph and an integer g , find g vertices, no two of which have an edge between them.

VERTEX COVER: Given a graph and an integer b , find b vertices cover (touch) every edge.

CLIQUE: Given a graph and an integer g , find g vertices such that all possible edges between them are present.



LONGEST PATH

Longest Path



LONGEST PATH: Given a graph G with nonnegative edge weights and two distinguished vertices s and t , along with a goal g .

To find a path from s to t with total weight at least g .

To avoid trivial solutions we require that the path be simple, containing no repeated vertices.

KNAPSACK



KNAPSACK: Given integer weights w_1, \dots, w_n and integer values v_1, \dots, v_n for n items. We are also given a weight capacity W and a goal g .

Seek a set of items whose total weight is **at most** W and whose total value is **at least** g .

The problem is solvable in time $O(nW)$ by **dynamic programming**.

Is there a polynomial algorithm for KNAPSACK? Nobody knows of one.

A variant of the KNAPSACK problem is that the **unary** integers.

- by writing *IIIIIIIIII* for 12.
- It defines a legitimate problem, which we could call **UNARY KNAPSACK**.
- It has a **polynomial algorithm**.

A different variation:

- Suppose now that each item's value is equal to its weight, the goal *g* is the same as the capacity *W*.
- This special case is tantamount to finding a subset of a given set of integers that adds up to exactly *W*.
- **Q:** Could it be polynomial?

Subset Sum



SHANGHAI JIAO TONG
UNIVERSITY

SUBSET SUM: Find a subset of a given set of integers that adds up to exactly W .