



Algorithms & Complexity

The Class NP

Anton Bryl - abryl@computing.dcu.ie

CA313@Dublin City University. 2009-2010.

November 12, 2009



Non-Deterministic *TIME*

- ▶ For nondeterministic Turing machine the time complexity is denoted *NTIME*.
- ▶ $NTIME(T, x)$ denotes the number of steps of the *non-deterministic* computation $T(x)$.
- ▶ We will say that $NTIME(T) = O(f(n))$ if $NTIME(T, x) = O(f(n))$ where $n = |x|$ (length of x).



Non-Deterministic *TIME*

- ▶ For nondeterministic Turing machine the time complexity is denoted *NTIME*.
- ▶ $NTIME(T, x)$ denotes the number of steps of the *non-deterministic* computation $T(x)$.
- ▶ We will say that $NTIME(T) = O(f(n))$ if $NTIME(T, x) = O(f(n))$ where $n = |x|$ (length of x).

- ▶ $NTIME(f(n)) = \{P \mid \exists T \text{ s.t. } T \text{ solves } P \text{ and } NTIME(T) = O(f(n))\}$



The class NP

Definition

$$NP = \cup_{k \in \mathbb{N}} NTIME(n^k).$$

NP is the (complexity) class of decision problems that can be solved using a non-deterministic Turing Machine and a polynomial amount of computation time.



The class NP

Definition

$$NP = \cup_{k \in \mathbb{N}} NTIME(n^k).$$

NP is the (complexity) class of decision problems that can be solved using a non-deterministic Turing Machine and a polynomial amount of computation time.

Obviously, $P \subseteq NP$. Does $P = NP$?



The class NP

Definition

$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k).$$

NP is the (complexity) class of decision problems that can be solved using a non-deterministic Turing Machine and a polynomial amount of computation time.

Obviously, $P \subseteq NP$. Does $P = NP$?

It is an open problem of major importance.



The class NP

Alternative definition

NP is the class of decision problems, for which the “yes”-answers have proofs verifiable in polynomial time by a deterministic Turing machine.



The class *NP*: Application

- ▶ Given this “easy to check” feature of the class NP, there is a natural application for the problems which belong to NP but are not proven to belong to P.



The class *NP*: Application

- ▶ Given this “easy to check” feature of the class NP, there is a natural application for the problems which belong to NP but are not proven to belong to P.
- ▶ No fast way to find the answer is known, but if somebody claims that he/she knows the answer, it is easy to verify this claim.

The subset sum problem: slightly better algorithm.

Example

$$S = \{-2, -3, 15, 14, 7, -10\}$$

1. Split the n elements into 2 sets of $\frac{n}{2}$
 $S_1 = \{-2, -3, 15\}$; $S_2 = \{14, 7, -10\}$
2. For each of the 2 subsets calculate the sums of all possible $2^{n/2}$ subsets of its elements and store them in an array of length $2^{n/2}$

$$\{\}, \{-2\}, \{-3\}, \{15\}, \{-2, -3\}, \{-2, 15\}, \{-3, 15\}, \{-2, -3, 15\}$$

$$\Rightarrow A_1 = (, -2, -3, 15, -5, 13, 12, 10)$$

$$\{\}, \{14\}, \{7\}, \{-10\}, \{14, 7\}, \{14, -10\}, \{7, -10\}, \{14, 7, -10\}$$

$$\Rightarrow A_2 = (, 14, 7, -10, 21, 4, -3, 11)$$

3. Sort each of these arrays
 $A_1^{\text{sorted}} = (, -5, -3, -2, 10, 12, 13, 15)$
 $A_2^{\text{sorted}} = (, -10, -3, 4, 7, 11, 14, 21)$

4.

The subset-sum problem: slightly better algorithm.

Example

- 1.
- 2.
3. $A_1^{\text{sorted}} = (-5, -3, -2, 10, 12, 13, 15)$
 $A_2^{\text{sorted}} = (-10, -3, 4, 7, 11, 14, 21)$
4. Check if an element of the first array and of the second array sum up to 0. To do that, the algorithm passes through the first array in decreasing order and the second array in increasing order and skipping empty elements.
 i_1 (i_2): index of current element in A_1 (A_2)
 if ($A_1[i_1] + A_2[i_2] > 0$): $i_1 = i_1 - 1$
 if ($A_1[i_1] + A_2[i_2] < 0$): $i_2 = i_2 + 1$
 if ($A_1[i_1] + A_2[i_2] = 0$): solution found. Stop

Slightly better algorithm: complexity

1. Split the n elements into 2 sets of $\frac{n}{2}$: $O(1)$
2. Calculate sum of all possible $2^{n/2}$ subsets of its elements and store them in an array of length $2^{n/2}$
 worst case: $O(\frac{n}{2}2^{n/2}) = O(n 2^{n/2})$ (definition of O)
3. Sort each of these arrays
 $O(t \log t)$; $t = 2^{n/2} \Rightarrow O(\log 2 \times \frac{n}{2}2^{n/2}) = O(n 2^{n/2})$
4. Check if an element of the first array and of the second array sum up to 0. To do that, the algorithm passes through the first array in decreasing order and the second array in increasing order. $O(2 \times 2^{n/2}) = O(2^{n/2})$

$$O(1) + O(n 2^{n/2}) + O(n 2^{n/2}) + O(2^{n/2}) = O(n 2^{n/2})$$

Slightly better algorithm: remarks

- ▶ for large n , n is negligible compared to $2^{n/2}$
- ▶ $O(2^n) \neq O(2^{n/2})$
 - ▶ if for large enough n , $\exists c, 0 \leq f(n) \leq c \times g(n)$, then $\exists c', 0 \leq f(n) \leq c' \times k \times g(n)$ (for example $c' = \frac{c}{k}$). Thus $O(g(n)) = O(k \times g(n))$ and multiplicative constants can be ignored.
 - ▶ E.g. for any $a, b > 1 : O(\log_a(n)) = O(\log_b(n))$
 - ▶ There exists no c such that for any n large enough $2^n \leq c \times 2^{n/2}$. Thus $O(2^n) \neq O(2^{n/2})$.
In general if $d \neq c$, $O(d^n) \neq O(c^n)$. (here $2^{n/2} = (2^{1/2})^n$)
- ▶ There exists an algorithm with gives an approximate solution in polynomial time.

Example 2: The Traveling Salesman Problem (TSP)

Definition (The Traveling Salesman Problem (TSP))

Given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city?

Definition

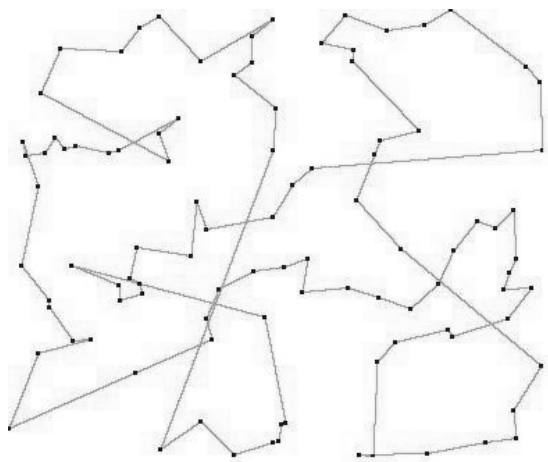
A salesman spends his time visiting n cities (or nodes) cyclically. In one tour he visits each city just once, and finishes up where he started. In what order should he visit them to minimise the distance traveled?

The Traveling Salesman Decision Problem (TSDP)

Definition (The Traveling Salesman Decision Problem (TSDP))

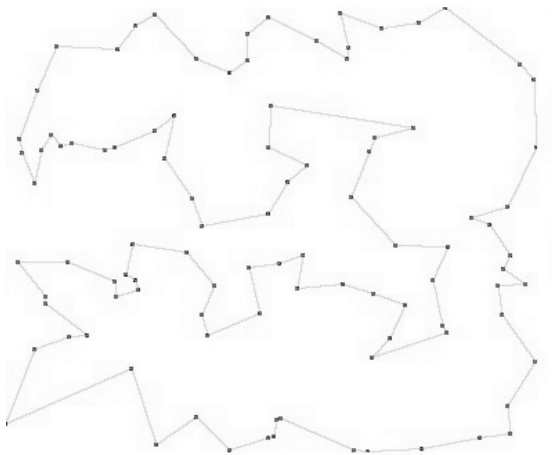
Given a number of cities and the costs of traveling from any city to any other city, and a number k , is there a round-trip route that visits each city exactly once and then returns to the starting city with a total cost less than k ?

The Traveling Salesman Problem (TSP)



Total cost = 196

The Traveling Salesman Problem (TSP)



Total cost = 161

The Traveling Salesman Decision Problem (TSDP)

An equivalent formulation in terms of graph theory is:

Definition

Given a complete weighted graph G (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), and a number k , does a cycle exist passing through all the nodes of G such that the sum of the weights associated with the edges of the cycle is, at most, k ?

The TSP: the brute-force approach

A path correspond to a sequence of cities. In order to enumerate the set of paths, we first choose one city, then another one, and so on.

The TSP: the brute-force approach

A path correspond to a sequence of cities. In order to enumerate the set of paths, we first choose one city, then another one, and so on.

Number of different paths?

If n is the number of cities, then, at each step k we can choose between $n - k$ cities. Direction does not matter.

$$\implies \text{number of paths} = \frac{1}{2} \times (n - 1) \times (n - 2) \cdots \times 1 = \frac{(n-1)!}{2}$$

Complexity

The complexity is in $O((n - 1)!)$.

As expected, this approach leads to an (at least) exponential algorithm. ($n! = O\left(\left(\frac{n}{2}\right)^n\right)$)

The TSP: non-brute-force exact algorithms

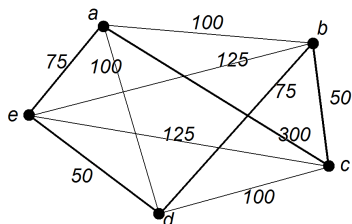
- ▶ Dynamic programming: $O(n^2 \times 2^n)$ (and requires lots of additional space)
- ▶ Various techniques for reasonably small n , e.g. “Branch and bound”

The TSP: an approximate algorithm

The Nearest Neighbor Heuristic

For each city, we continue the path by choosing the nearest city.

*An instance of the
Travelling Salesman Problem*



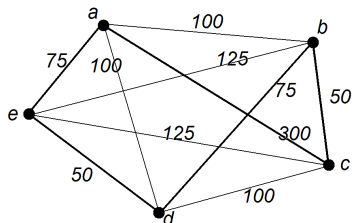
Cost of Nearest
Neighbor Path,
AEDBCA = 550

The TSP: an approximate algorithm

The Nearest Neighbor Heuristic

For each city, we continue the path by choosing the nearest city.

*An instance of the
Travelling Salesman Problem*



*Cost of Nearest
Neighbor Path,
AEDBCA = 550*

Complexity

We only have to explore one path, with $n - 1$ cities, so the complexity is linear: $O(n)$.

The TSDP

TSDP is in *NP*

If a solution (i.e. a sequence of cities) is given, how long is it to compute its cost?

The TSDP

TSDP is in *NP*

If a solution (i.e. a sequence of cities) is given, how long is it to compute its cost?

Complexity of verification

We only have to explore one path, with $n - 1$ cities, so the complexity of verification is linear: $O(n)$.

\implies *TSDP* is in *NP*.

The Bottleneck TSP

Another related problem

Find a route, which minimizes the cost of the most costly transition between two cities.

Polynomial-time reduction

Definition (Reduction)

Reduction is a transformation of one problem into another problem.

Intuitively, if a problem A is reducible to a problem B , a solution to B gives a solution to A . Thus solving A cannot be harder than solving B .

Definition (Polynomial-time reduction)

A polynomial-time reduction is a reduction which is computable by a deterministic Turing machine in polynomial time.

Polynomial-time reductions compose in a transitive fashion

Polynomial-time reduction: Example

Definition (Undirected Hamilton cycle problem)

Given an undirected graph G , is there a cycle that passes through each node of G exactly once ?

We want to reduce the Undirected Hamilton cycle problem to the Traveling Salesman Decision problem (TSDP).

Polynomial-time reduction: Example

Given an undirected graph G , we construct the following instance of the TSP:

- ▶ number of cities=number of vertices in graph
- ▶ distance between two cities i and j :

$$d_{ij} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } (v_i, v_j) \in G \\ 2 & \text{otherwise} \end{cases} \quad (1)$$

- ▶ budget $k = n$

The cost of an itinerary is n plus the number of intercity distances traversed that are not edges of G . Thus, a tour of cost k or less exists if and only if the number of “nonedges” used is zero, that is, if and only if the tour is a Hamilton cycle of G .

NP-Hard problems

Definition (*NP*-Hard Problem)

A problem is *NP*-hard (Non-deterministic Polynomial-time Hard) if solving it in polynomial time would make it possible to solve all problems in class *NP* in polynomial time. That is, a problem is *NP*-hard if an algorithm for solving it can be translated into one for solving any other *NP* problem. *NP*-hard therefore means “at least as hard as any *NP* problem”.

Definition (*NP*-Hard Problem)

NP-hard (Non-deterministic Polynomial-time hard) refers to the class containing all problems H , such that for every decision problem L in *NP* there exists a polynomial-time reduction to H .

NP-Complete problems

Definition (*NP*-Complete Problem)

A decision problem is *NP*-complete if:

1. it is in *NP*
2. it is *NP*-hard, i.e. every other problem in *NP* is reducible to it.

