

Algorithms & Complexity

Complexity Classes

Nicolas Stroppa

Patrik Lambert - plambert@computing.dcu.ie

CA313@Dublin City University. 2008-2009.

November 20, 2008

Example 1: The subset-sum problem

Definition (The subset-sum problem (SSP))

Given a set of integers $S = \{i_1, i_2, \dots, i_n\}$, does any subset $A \subseteq S$ sum to 0?

Or: is there a subset $A \subseteq S$ s.t.

$$\sum_{i_j \in A} i_j = 0.$$

Application

In cryptography, Subset Sum problem comes up when a codebreaker attempts, given a message and ciphertext, to deduce the secret key.

Example 1: The subset-sum problem

Definition (The subset-sum problem (SSP))

Given a set of integers $S = \{i_1, i_2, \dots, i_n\}$, does any subset $A \subseteq S$ sum to 0?

Or: is there a subset $A \subseteq S$ s.t.

$$\sum_{i_j \in A} i_j = 0.$$

Application

In cryptography, Subset Sum problem comes up when a codebreaker attempts, given a message and ciphertext, to deduce the secret key.

For instance, does a subset of the set $\{-2, -3, 15, 14, 7, -10\}$ add up to 0?

Example 1: The subset-sum problem

For instance, does a subset of the set $\{-2, -3, 15, 14, 7, -10\}$ add up to 0?

The answer is *YES*, though it may take a little while to find a subset that does - and if the set were larger, it might take a very long time to find a subset that does. . .

However, if someone claims that the answer is *YES*, because “ $\{-2, -3, -10, 15\}$ add up to zero”, then we can quickly check that with a few additions. Verifying that the subset adds up to zero is much faster than finding the subset in the first place.

The subset-sum problem. The brute-force approach

Definition (The subset-sum problem)

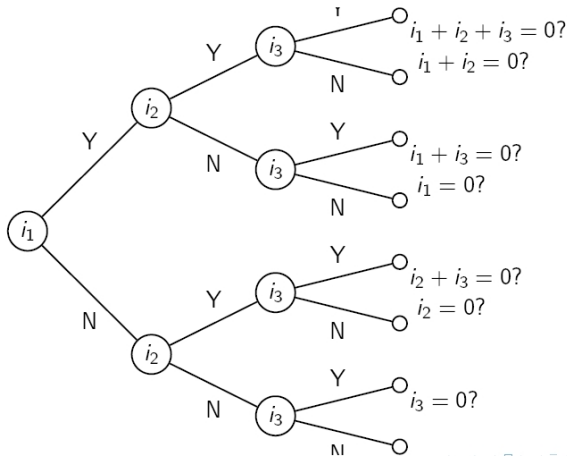
Given a set of integers $S = \{i_1, i_2, \dots, i_n\}$, does any subset of them sum to 0?

The brute-force approach

For each integer i in the set, consider the two following options: “I add i to the subset”, “I do *not* add i to the subset”.

You have a *decision tree*: at each node of the tree, you are making a decision about the inclusion of the integer in the subset.

The subset-sum problem. The brute-force approach



The subset-sum problem. Complexity

Definition (The subset-sum problem)

Given a set of integers $S = \{i_1, i_2, \dots, i_n\}$, does any subset of them sum to 0?

Complexity of the brute-force algorithm

For each integer, there are two choices. There are n integers, so 2^n paths. For each path, you compute at most n addition, so the final complexity is $O(n \times 2^n)$.

Proof of $SSP \in NP$

If I have the solution, how difficult is it to verify it?

⇒ at most n addition, so the complexity of the verification is $O(n)$.

⇒ with a Non-deterministic Turing Machine, we can explore all the paths at the same time, hence the linear complexity in this case.

The subset-sum problem: slightly better algorithm.

1. Split the n elements into 2 sets of $\frac{n}{2}$
2. Calculate sum of all possible $2^{n/2}$ subsets of its elements and store them in an array of length $2^{n/2}$
3. Sort each of these arrays according to sum of elems in subset
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.

The subset-sum problem: slightly better algorithm.

Example

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

- Split the n elements into 2 sets of $\frac{n}{2}$
 $S_1 = \{-2, -3, 15\}$; $S_2 = \{14, 7, -10\}$
- Calculate sum of all possible $2^{n/2}$ subsets of its elements and store them in an array of length $2^{n/2}$
 $\{\}, \{-2\}, \{-3\}, \{15\}, \{-2, -3\}, \{-2, 15\}, \{-3, 15\}, \{-2, -3, 15\}$
 $\Rightarrow A_1 = (-2, -3, 15, -5, 13, 12, 10)$
 $\{\}, \{14\}, \{7\}, \{-10\}, \{14, 7\}, \{14, -10\}, \{7, -10\}, \{14, 7, -10\}$
 $\Rightarrow A_2 = (14, 7, -10, 21, 4, -3, 11)$
- Sort each of these arrays according to sum of elems in subset
 $A_1^{\text{sorted}} = (-5, -3, -2, 10, 12, 13, 15)$
 $A_2^{\text{sorted}} = (-10, -3, 4, 7, 11, 14, 21)$
-

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 according to sum of elems in subset
 $O(t \log t)$; $t = 2^{n/2} \Rightarrow O(\log 2 \times \frac{n}{2}2^{n/2}) = O(n 2^{n/2})$
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 $2^{n/2}$
- ▶ 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.
- ▶ thus for any $a, b > 1$: $O(\log_a(n)) = O(\log_b(n))$
- ▶ if $0 \leq f(n) \leq c' \times 2^{n/2}$, we cannot find c' such that for any n large enough, $0 \leq f(n) \leq c' \times 2^n$. Thus $O(2^n) \neq O(2^{n/2})$.
In general if $d \neq c$, $O(d^n) \neq O(c^n)$. (here $2^{n/2} = (2^{1/2})^n$)

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

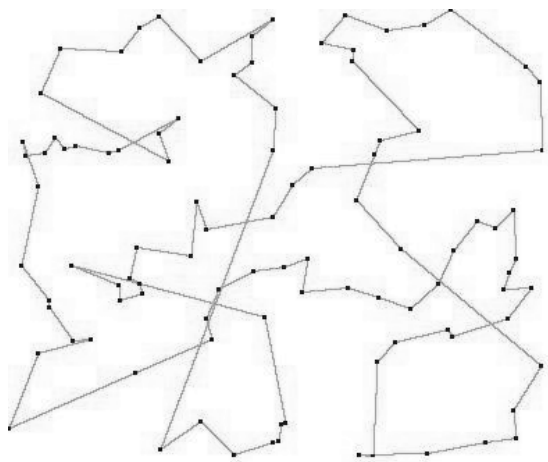
A salesman spends his time visiting n cities (or nodes) cyclically. In one tour he visits each city just once, and finishes up where he started. In what order should he visit them to minimise the distance traveled?

The Traveling Salesman Decision Problem (TSDP)

Definition (The Traveling Salesman Decision Problem (TSDP))

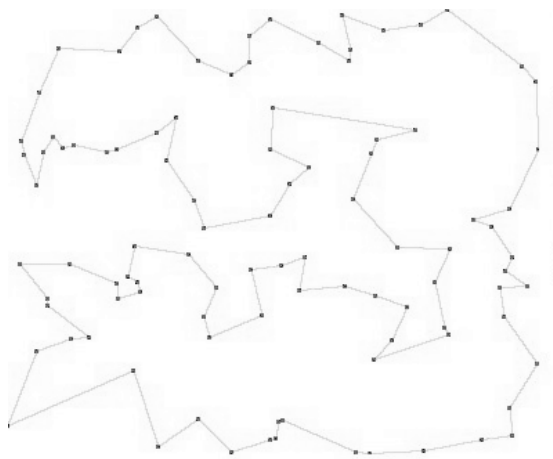
Given a number of cities and the costs of traveling from any city to any other city, and a number k , is there a round-trip route that visits each city exactly once and then returns to the starting city with a total cost less than k ?

The Traveling Salesman Problem (TSP)



Total cost = 196

The Traveling Salesman Problem (TSP)



Total cost = 161

The TSP: Applications

- ▶ Transportation and logistics
- ▶ printed circuit manufacturing: scheduling of a route of the drill machine to drill holes in a circuit board
- ▶ Meals delivery (pizza)
- ▶ Newspaper delivery
- ▶ Scheduling the collection of coins from payphones throughout a given region

The Traveling Salesman Problem (TSP)

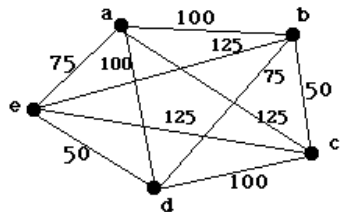
An equivalent formulation in terms of graph theory is:

Definition

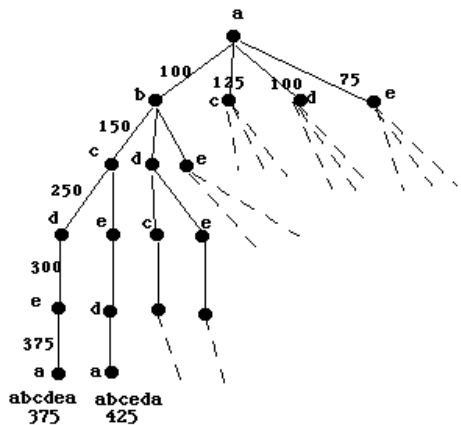
Given a complete weighted graph G (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), and a number k , does a cycle exist passing through all the nodes of G such that the sum of the weights associated with the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

The TSP: the brute-force approach

An Instance of the Traveling Salesman Problem



Search Space



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.

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

Complexity

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

As expected, this approach leads to an (hyper-)exponential algorithm. (the factorial function is hyper-exponential:

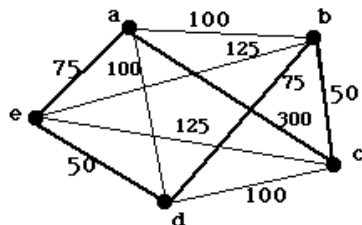
$$n! = O\left(\left(\frac{n}{2}\right)^n\right)$$

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
Traveling Salesman Problem**



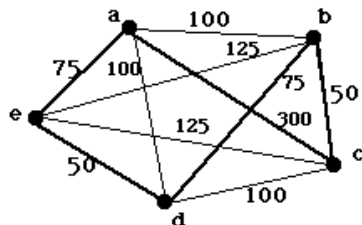
**Cost of Nearest
Neighbor Path,
AEDBCA = 550**

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 Traveling 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*.

If we had a non-deterministic Turing Machine, we could have explore all the paths in one go, with a linear complexity.

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 problem (TSP).

