CSCI 3110

Exploiting Problem Structure and More

He

Alex Safatli

Monday, November 4, 2013

Contents

Exploiting the Problem's Structure

Problem Structure

Many problems have some manner or sort of structure. This could be, for instance, **algebraic**, **number-theoretic**, **topological**, **geometric**, etc. These commonly suggest some sort of solution. We can design a solution accordingly to exploit that structure. There is no fixed step in order to acquire a structure.

Example: Greatest Common Divisor

An example of a problem where we can exploit the problem structure is that of the **greatest common divisor**, or finding the gcd. Given nonnegative integers u, v, we would like to compute their greatest common divisor (`gcd(u,v)`). Before going further, we should review some mathematical notation.

$$d|u = d \text{ is a divisor of } u = d \text{ divides } u \tag{1}$$

In other words, there exists an integer k such that $u = kd$ if $d|u$.

Also, we can propose that d is a **common divisor** of u and v *iff* $d|u$ and $d|v$. Going further, $d = \gcd(u, v)$ means that d is a common divisor of u and v, and no other common divisor is $> d$. For example, $\gcd(24, 18) = 6$. This is especially useful when considering fractions:

$$\frac{18}{24} = \frac{18/6}{24/6} = \frac{3}{4} \tag{2}$$

Brute-Force Solution. A brute-force solution would involve calculating $t = \min(u, v)$ and trying $t, t - 1, t - 2, ...$ The running time of this function would be $O(\min(u, v))$. This is *not a linear* running time.

Recall that when we defined the running time, we did so on the basis of the size of the input. Even though it only requires two words to store these two values, we cannot say the size of the input is two — how would we describe the running time *v.* a constant? Instead, we use the number of bits; we would roughly require $\lg u + \lg v$ bits to encode u and v.

In the worst case, therefore, we could consider a situation where $u \approx v$. The input size in this case would be $\approx 2 \lg u$. The running time would be $O(u)$. Or, in other words, it would be $O(2^{\frac{\text{input size}}{2}})$. This is *exponential time.*

NOTE The reading for Friday is 31.2.

Euclid's Algorithm. So, how do we exploit the problem structure here? The idea is that if $d|u, d|v$ then $d|(au + bv)$ for all integers a, b. Let us choose a and b such as to minimize $au + bv$. One way to do this is to have $a = 1, b = -\lfloor u/v \rfloor$. Therefore,

$$au + bv = u - v\lfloor u/v \rfloor = u \mod v \tag{3}$$

We can then make the claim that $d|u$ and $d|v \iff d|v$ and $d|(u \mod v)$. Euclid's Algorithm, postulated around 300 BC, can be represented by the following pseudocode and makes use of this claim.

```
   Euclid(u,v)
1  while (v != 0) do
2    (u, v) <-- (v, u mod v)
3  return u
```

A theorem (Finck, 1841) claims that Euclid's Algorithm uses $O(\lg v)$ steps on input (u, v) where $u > v > 0$. If this latter condition is not true, switch the values. Proof. Consider replacing (u, v) by (u, r) where $r = u \bmod v$. There will be three cases here.

1. *Case.* $v > u/2$; then, $r = u - v < u/2$.
2. *Case.* $v = u/2$; then, $r = 0$.
3. *Case.* $v < u/2$; then, $r < v < u/2$.

In all three cases, we can always say that $r < u/2$. It follows that after two steps, say $(u, v) \to (v, r) \to (r, s)$, we have $s < v/2$ and $r < u/2$. The value of v has decreased by a factor of 2. After $O(\lg v) \leq 2 \lg v + 1$ steps, v will decrease to 0. □

Therefore, the running time of Euclid's Algorithm is $O(\min(\lg u, \lg v))$. It is *linear* (see above definition of linearity).

Binary Algorithm (Divide-and-Conquer, Exploits Problem Structure; J. Stein 1967, evidence in 1st Century China). The advantage of this algorithm is that it *does not perform division* (only right shifts). We can define the gcd as follows (with 5 cases), with $u \geq v$.

$$\gcd(u, v) = \begin{cases} u & \text{if } v = 0 \\ 2 \times \gcd(u/2, v/2) & \text{if } u, v \text{ both even} \\ \gcd(u, v/2) & \text{if } u \text{ is odd and } v \text{ is even} \\ \gcd(u/2, v) & \text{if } u \text{ is even and } v \text{ is odd} \\ \gcd((u - v)/2, v) & \text{if } u, v \text{ both odd} \end{cases} \tag{4}$$

The pseudocode for this algorithm is presented below.

```
    GCD(u,v)
// Direct translation of above.
1   if u < v then
2     return GCD(u,v)
3   if v = 0 then
4     return u
5   if u and v both even then
6     return 2*GCD(u/2,v/2)
7   if u is odd and v is even then
8     return GCD(u,v/2)
9   if u is even and v is odd then
10    return GCD(u/2,v)
11  if u and v both odd then
12    return GCD((u-v)/2,v)
```

Determining the running time of this is slightly more complicated than for Euclid's Algorithm. We can claim that the total number of subtractions, $s(u, v) \leq \lg(u + v)$. Proof. By induction on $u + v$. The base case has $u + v = 1$ means that $u = 0, v = 1$ or $u = 1, v = 0$. In both cases $s(u, v) = 0$. Assume true for $u + v < n$. Prove for $u + v = n$. Consider the five cases (without loss of generality $u \geq v$).

1. If $v = 0$, $s(u, v) = 0$.

2. If u and v are both even, $s(u,v) = s(u/2, v/2) \leq \lg(u/2 + v/2) = \lg(u+v) - 1 \leq \lg(u+v)$.
3. If u is even and v is odd, $s(u,v) = s(u/2, v) \leq \lg(u/2 + v) \leq \lg(u+v)$ because it is in an increasing function.
4. If u is odd and v is odd, $s(u,v) = s(u, v/2) \leq \lg(u + v/2) \leq \lg(u+v)$.
5. If u and v are both odd, $s(u,v) = 1 + s((u-v)/2, v) \leq \lg((u-v)/2+v)+1 = \lg((u+v)/2)+1 = \lg(u+v)$.

How about the total number of shifts? Each time we do a right shift, we merely discard one bit. The total number of right shifts is at most the total number of bits in both numbers, $\lg u + \lg v$.

The total running time for this algorithm is therefore $O(\max(\lg u, \lg v))$. This is *linear*.

Probabilistic and Randomized Techniques

Probabilistic Analysis

In this analysis, we make use of knowledge of, or make assumptions about: **distribution of input**. The average-case running time analysis is typically what is considered here — the average for the running time is acquired over the distribution of possible inputs.

For example, we can analyze the running time of quicksort by assuming a uniform distribution of values: the average case runtime can be considered to be $O(n \lg n)$. For interpolation sort, the average case runtime is $O(\lg \lg n)$.

Randomized Algorithms

Algorithm behaviour is determined by input here, and by values produced by a **random number generator**. Here, random means **truly random** — each time we ask for an integer between 1 and n, we get each integer equally likely (the probability of getting any integer is $1/n$). Furthermore, the **expected running time** here takes the expectation of the running time over the distribution of values returned by the random number generator (during the execution of the algorithm).

The focus will be on this manner of algorithm in this course; the former (probabilistic) analysis was covered in a previous Computer Science course.

Example: The Majority Element Problem

Given an array `A[1..n]` of n integers, where one element (integer) occurs $> \frac{n}{2}$ times, we wish to find this element.

Randomized Solution. Pseudocode for a randomized algorithm solution is found below.

```
   Find_Majority(A[1..n])
1  while true do
2    i <-- Random(1,n)
3    Get the number, j, of occurrences of A[i] in A
4    if j > (n/2) then
5      return A[i]
```

Analysis. Sketching the analysis out, we can observe that since there *is* a majority element (given), the probability of finding it in one try is $> \frac{1}{2}$ because they are the majority. The expected number of tries to find it is < 2 (to be proved later). Each try is $O(n)$ (count number of occurrences). Therefore, the expected running time is $O(n)$.

Proof. The only step we are uncertain about is the definition of the expected number of tries. The expectation of a random variable x that can take any value in a set S is, where $p_r(v)$ is the probability of acquiring some value v in the set:

$$\sum_{v \in S} p_r(v) \times v \tag{5}$$

For example, imagine tossing a die. The expected value of the die is $1 \times \frac{1}{6} + 2 \times \frac{1}{6} + ... + 6 \times \frac{1}{6} = 3.5$. Similarly, we make a guess at each iteration of the while loop in the majority element problem: in each try, we find the majority with probability $p > 1/2$. The expected number of tries $x = p \times 1 + (1-p)p \times 2 + (1-p)^2 p \times 3 + ...$ Therefore, $x = \sum_{i \geq 1} (1-p)^{i-1} pi$.

Let $y = \sum_{i \geq 1} (1-p)^i p = \frac{((1-p)p)}{1-(1-p)} = 1 - p$. What we want to compute is $x(1-p) + y = \sum_{i \geq 1} (1-p)^i pi + \sum_{i \geq 1} (1-p)^i p = \sum_{i \geq 1} (1-p)^i p(i+1) = x - p$. This last simplification is easily realized by setting some $j = i + 1$. This means, then, that $x(1-p) + (1-p) = x - p$ and that $x = \frac{1}{p}$. We have proved for certain that the expected number of tries is $< 2 = \frac{1}{1/2}$. □

What are some issues with this algorithm? It:

- is not robust — no majority: loop forever,
- *might* take a very long time to terminate; rare, and
- relies on **pseudorandom** numbers generated by a computer.

Realize that there is a deterministic approach or method of generating pseudorandom numbers. Most of them use a random **seed** to begin at a different place every time. For example, we may use the last few digits of a system clock, using the assumption that the current time will be close to a truly random number. For the remaining values, one approach involves a **linear congruential generator** (Lehmer 1949); this takes four parameters: x_0 (the seed), a modulus m, a multiplier a, and a constant c. Pseudorandom numbers are generated in $[0..m-1]$ by the formula:

$$x_{n+1} = (ax_n + c) \mod m \tag{6}$$

A good system could use $m = 2^{31} - 1, a = 16807, c = 0$ (Park & Miller 1988). This is used in GNU libraries.

Deterministic Solution. Boyer-Moore's voting algorithm. What if we wish to use a *deterministic* approach to this problem? There is indeed a solution of this nature that takes $O(n)$ time.

Graph Algorithms

Definitions, Cont'd

Loops or **self-loops** are edges from a vertex to itself. There is also this notion of a **multi-edge** where more than one edge can go from a vertex A to another vertex B. A graph can be denoted **simple** if it does not have any of these two notions (loops or multi-edges).

A **path** is a sequence of edges to travel; the length of a path would be the number of edges between the edges in that path. A path is simple if there are no repeated vertices or edges. A **cycle** is a special type of path where the first and last vertices are the same; it is simple if there are no repeated edges or vertices between the first and last vertices.

Connectivity of a graph can also be defined. A graph is **connected** if, for any two vertices, there is a path.

A **tree** is a special type of graph: it is a connected, acyclic, undirected graph. Recall the notion of the root: we can pick any node to be the root and define the parent-child relationship because there is no cycle. If none is picked, it is unrooted; otherwise, it is **rooted**.

Representation

Common approaches of representation are:

1. **adjacency matrix**: a $|v| \times |v|$ (where v is the set of vertices) matrix A where $A[i,j] = 1$ if $(i,j) \in E$ (the edge set) and 0 otherwise. If a graph is directed, the number of 1s is $|E|$. For an undirected graph with no loops, the number of 1s would be $2|E|$.
2. **adjacency list**: an array `Adj[1..|v|]` where each entry is a list; the uth entry would be for the vertex u where `Adj[u]` is a list containing all vertices v such that there is an edge between u and $v \in E$.

Comparing the two, we find that the space cost for the adjacency matrix would be $O(|v|^2)$ while the adjacency list would be $O(|v| + |E|)$. Determining adjacency, the matrix would merely be a lookup at $O(1)$, while for the list it would be $O(|v|)$ (in the worst case). Listing all neighbours of u for the adjacency matrix would be $O(|v|)$, while for the list it would be $O(d(u))$ (the degree of the node u); this could be much smaller.

NOTE We call a graph **sparse** if $|E| << |v|$; this means the adjacency list may be a lot more useful.

NOTE Traversing a Graph. Recall that the **breadth-first search** utilizes a queue and the **depth-first search** uses recursion or a stack; both are $O(|v|+|E|)$. In the textbook, review of these two algorithms are found at 22.3, 22.4.

Weighted Graphs

A **weighted graph** is where each edge has a **weight**, some value. Both representations are still usable here: in the adjacency matrix, we could store the actual weights instead of binary values (where ∞ is stored if no edge is present between two nodes). In the adjacency list, edge weight is stored together with vertices in a list (a new field for each item in the list).

Minimum Spanning Trees

A **spanning tree** is an unrooted tree that connects all vertices of a connected, undirected graph. Multiple could be found for the same graph. A **minimum spanning tree** (MST) is where we wish to minimize the sum of the weights of the edges used in a given spanning tree.

Why is this important? This has a large number of applications: in a computer network, which can be modelled using a graph, we can minimize the cost of using links in the network if a message is sought to be broadcast to all computers. This is also useful in clustering in data mining.

Minimum Spanning Trees continued on next page...

Kruskal's Algorithm. At each step of this algorithm, we choose the minimum-weight edge that does not form a cycle. We work with two sets:

1. a **forest** (set of trees), where initially each vertex is a tree of 1 vertex, and
2. a set of edges E', initially defined to be identical to E.

At each step, we remove and consider a minimum-weight edge from E'; if this edge connects two different trees, it is added to the forest and joins two trees. Otherwise, we discard it. When E' is empty, the forest becomes an MST.

Correctness. First, the algorithm produces a **spanning tree**. The resulting graph cannot have a cycle, because if it did, the edge added that formed the cycle would join together two vertices on the same tree. The resulting graph must be connected, for if it weren't, when we considered the edge that connected two components, we would have added it.

NOTE Reading for today is 22.1, 23.1; for next lecture, it is 23.2.

Next, we prove that the spanning tree T produced by the algorithm is a MST. Assume that it is *not*. Among all MSTs, let T' be the one with the largest number of edges in T (most shared edges with T). Consider the particular edge e in $T - T'$ which was the first to be added by the algorithm. Because T' is a spanning tree, $T' \cup \{e\}$ has a cycle C. T cannot contain all edges of C, so C contains an edge f not in T. Now, consider $T'' = (T' \cup e) - \{f\}$. T'' is also a spanning tree because the cycle was broken. Since T' is minimum, $\text{weight}(e) \geq \text{weight}(f)$; otherwise, T'' has a smaller total weight. If $\text{weight}(f) < \text{weight}(e)$, then Kruskal's Algorithm would have considered f before e. Adding f would not create a cycle (e was the first edge of T not in T'), so it would have been added. Since it was not added, $\text{weight}(f) \geq \text{weight}(e)$; therefore, $\text{weight}(e) = \text{weight}(f)$ and T'' is also a minimum spanning tree. T'' shares one more edge with T than T' does. This is a contradiction. □

Implementation. The representation of an edge can be denoted by (u, v). Can augment the use of a **Disjoint-Set Data Structure**. Note this is also associated with what is known as the "union find" problem. This structure would have operations:

1. `MAKE-SET` to create a set containing a single vertex,
2. `FIND` to, given a vertex, find the name of the set it is in, and
3. `UNION` to union two sets [given two vertices].

Note that this is a **Disjoint-Set Data Structure Problem**, or a **Union-Find Problem**. See Chapter 21 of the textbook for more information. Realize that we maintain a collection of **sets**; a set can be stored in a linked list of vertices which can then be augmented with several fields.

1. The *name* of the set would be the first element in this list.
2. A *tail pointer* would be contained in the structure.
3. A *number field* would be contained in the structure denoting the number of elements.
4. Each element is comprised of:
   - The data.
   - A pointer to the next element.
   - A *head pointer* to the list head.

Using this manner of data structure, `MAKE-SET` would be $O(1)$, `FIND` would be $O(1)$, and `UNION` would be $O(t)$ where t is the number of vertices in the smaller list. The sequence of operations needed for the latter function would be:

- Follow the head pointer of u to its head and v to its head.
- Say that u is in the smaller list; link that list to the end of the bigger list. Point the end of u to the beginning of v.
- Traverse u's list, updating each head pointer.
- Update the number field and tail pointer of v's list.

Pseudocode. The following is the pseudocode for this problem.

```
   Kruskal(G,w) // G = (V,E) ; w: weight function
1  A <-- empty set
2  for each v in V do:
3    MAKE-SET(v)
4  Sort the edges of E into increasing order of weight
5  for each edge (u,v) in E in increasing order of weight do:
6    if FIND(u) != FIND(v) then: // in different sets; joins diff trees
7      A <-- A U {u,v}
8      UNION(u,v)
9  return A
```

Using the aforementioned implementation of the data structure, this would possess an $O(|v|)$ process for the `MAKE-SET` loop, an $O(|E| \lg |E|)$ sorting process, and an $O(|E|)$ `FIND` and A-appending set of steps. Realize that we can observe that there are n `UNION`s of sets of initial size 1:

- Each union means a smaller list becomes part of a list of at least twice the size.
- An element's head pointer gets updated $O(\lg n)$ times (based on the above).
- Therefore, for n unions, the running time would be $O(n \lg n)$.

This means that the `UNION` step is computed $O(|E| \lg |E|)$ times. The total running time is therefore $O(|E| \lg |E|) = O(|E| \lg |V|)$.

Prim's Algorithm. This algorithm involves the following procedure.

1. At each step, choose a previously unconnected vertex that becomes connected by a lowest-weight edge. We work with:
   - A set of vertices V', initially containing an initial vertex arbitrarily chosen.
   - A set of edges E', initially empty.
2. At each step, choose (u, v) of minimal weight with $u \in V'$ and $v \notin V'$, add v to V' and (u, v) to E'. When $V' = V$, return E'.

Implementation. Keep vertices not in V' in a priority queue (*heap*) where `key[u]` is the weight of the lowest-cost edge connecting u to some vertex in V'. `EXTRACT-MIN` would be $O(\lg n)$ time, and `DECREASE-KEY` would have $O(\lg n)$ time.

We can also define another array $\pi[1..|v|]$ where $\pi[u]$ is the parent of u in MST; in the end, it would define the MST. See Chapter 6.

Pseudocode.

```
   PRIM(G,w,r) // r: root
1   for each u in V do:
2     key[u] <-- infinity
3     pi[u]  <-- null
4   key[r] <-- 0
5   Q      <-- V // heap
6   while Q != empty do:
7     u <-- EXTRACT-MIN(Q)
8     for each v in Adj[u] do: // for each v adjacent to u
9       if v in Q and w(u,v) < key[v] then: // determine in Q: a flag for vertex
10        pi[v] <-- u
11        key[v] <-- w(u,v)
```

Population of the arrays is $O(|V|)$, creating the heap is $O(|V| \lg |V|)$, `EXTRACT-MIN` step takes $O(|V| \lg |V|)$, the inner loop is $O(|E|)$, and the array updating step is $O(|E| \lg |V|)$. The total time is therefore $O(|E| \lg |V|)$.

NOTE The reading for this is 23.2. Next is 24.1.
NOTE See paper notes.

Dijkstra's Algorithm

The introduction, pseudocode, and explanation of this code is found on paper notes.

Time Complexity. If a heap is used, the running time would be $O((|V| + |E|) \lg |V|)$. If all vertices are reachable from s, have running time $O(|E| \lg |V|)$ because $|V| + |E| \approx 2|E|$. Using a Fibonacci heap, have $O(|V| \lg |V| + |E|)$. Recall that the Bellman-Ford algorithm had running time $O(|V||E|)$; the running time for this algorithm is more efficient – notice, though, that negative edges mean the result of this algorithm would not be correct, so in that case we use the former algorithm.

NP-Completeness

Some Problems are Unsolvable

There are some problems in Computer Science where there is no known solution to problems that can be solved efficiently or in polynomial time. Being aware of them means you can save time on trying to solve them.

Recall that running time is typically expressed as a function of input size. We consider an algorithm to be *fast* when it is bounded by a polynomial in input size. **Polynomial time** can be expressed as a running time $O(n^k)$ where n is input size and k is some constant number. Problems that are polynomial time include $O(n^3)$ and $O(n \lg n) = o(n^2)$.

Why is this the case?

- In *practice*, algorithms that are fast can almost always be expressed using a polynomial time. Let us take a look at a running time $O(n^{1000})$; by definition, this is still considered "fast" — in practice, though, these are rare.

Some Problems are Unsolvable continued on next page...

- *Models of computation* allow us to show, for any reasonable model, if we can solve the problem in polynomial time in one model, it can be solved in another model in polynomial time as well. The model we've used in this class is Random Access Machine (RAM) model. Another model is the Turing Machine (TM) model. One exception is the Quantum Computation model – if a problem can be solved in this model, it might not be able to be solved in other ones.
- These have nice *closure properties*. Their sum is still polynomial time, for instance, as an example of composition — using the result of one function for the variable of another.

The Complexity Class P

The complexity class P is a class of problems in computation that are characterized by being decision problems. These are problems that can be solved in **polynomial time**. An algorithm can be found to solve it in this time.

A **decision problem** is a problem that has a *yes* or *no* answer. For example, given an array of numbers — finding if a number exists in the array or not is a decision problem (it is either in the array or not in the array).

Realize that when we calculate the input size for an algorithm, we normally represent the number in binary. Usually, two different and reasonable encodings are polynomially-related. This means that there is a polynomial time algorithm that can convert one to the other. For example, we have seen two representations of the Graph: (i) an adjacency matrix ($O(|V|^2)$) and an (ii) adjacency list ($O(|V| + |E|)$) — these are polynomial in terms of $n = |V|$.

Decision Problems not in P. Consider the following. Extended Regular Expression problem: Given two extended regular expressions E and F over an alphabet Σ, decide if E and F specify the same set of strings. Any algorithmic solution would require a function with running time of at least $\Omega(2^{cn/\lg n})$.

Informal Definition of the NP Class. We can identify another class of decision problems that their "yes"-instance can be verified in polynomial time; this is the NP class. An **instance** is simply a set of input to an algorithm that solves the problem. A "yes"-instance is an instance of the problem with a "yes" answer. An example of such a problem (in graph theory) involves a **Hamiltonian cycle** of an undirected graph $G = (V, E)$ that is a simple cycle — it contains all of the vertices of the graph.

The problem is that, given a graph G, we want to determine if it has a Hamiltonian cycle. Nobody can solve this problem in polynomial time. However, if I claim a particular graph has a Hamiltonian cycle, and you demand proof, then there is a *certificate* one could produce that they can easily check. The certificate are the vertices that comprise the cycle, in the order by which they appear — from this, can see if the answer is correct by determining three things:

- is the cycle simple?
- are its edges $\in E$, and
- does this cycle have every vertex?

Let us contrast these two classes, P and NP. Problems in P can be solved quickly. Problems in NP are not necessarily solved quickly, but the "yes" answer can be verified quickly provided one has the certificate. Realize quickly that $NP \neq$ not polynomial. NP means **nondeterministic polynomial**. We do not know whether the hard problems in NP can be solved in polynomial time, but that does not mean we can say they cannot be.

The Complexity Class P continued on next page...

NOTE Analogy. In 1903, F. N. Cole proposed that $2^{67} - 1 = 193707721 \times 761838257287$. Determining this relation took years, but verifying it takes minutes.

NOTE 34.1, 34.2 are readings for today. Readings for next lecture is 34.3.

Formal Definition of the NP Class. A more formal definition for the NP class is first preceded by the definition of the term **verifier**. A verifier for a decision problem is an algorithm V that takes two inputs:

1. An instance I of the decision problem.
2. A certificate C.

V returns *true* if C verifies that the answer to I is *yes*. Otherwise, V returns *false*. For each *yes* instance I, there must be at least one such certificate. For each *no* instance there must be no certificate showing that it is *yes*.

A **polynomial-time verifier** is a V that runs in time bounded by a polynomial in the size of I. Therefore, a decision problem is NP if it possesses a polynomial-time verifier. Note that $P \subseteq NP$ — V allows us to solve the problem without even looking at C.

Example 1. For example, let us look at a polynomial-time verifier for detecting whether or not a graph has a Hamiltonian cycle, `HAM-CYCLE`. A polynomial-time verifier takes $I = G$ and $C =$ a claimed list of vertices forming the cycle as input. The verifier would check to ensure all vertices are represented, the edges actually exist in G, and the cycle is simple.

Example 2. The **Traveling Salesman Problem** (TSP) is a problem where we are given a directed, weighted graph G and a number B. We must find whether there exists a directed cycle passing through all the vertices with total combined weight $\leq B$.

To determine whether this problem is in NP, we must determine whether there is a verifier that can be computed in polynomial time. The verifier must determine a number of things. All vertices in the graph must be represented, as in the Hamiltonian cycle problem, the edges are in G, and the total weight of the edges are $\leq B$. This can be done in polynomial time. Therefore, this problem is in NP.

Polynomial-Time Reductions

Recall the section on reducing a problem to a known problem, a paradigm covered in this course. Formally, a decision problem A reduces to a decision problem B in polynomial time if there exists a polynomial-time computable function f that maps instances of A to instances of B, such that the answers to the instances are preserved.

- If x is a *yes* instance of A, then $f(x)$ is also a *yes* instance of B.
- If x is a *no* instance of A, then $f(x)$ is also a *no* instance of B.

This condition should hold for all possible inputs.

The formal notation for this possibility for reduction can be represented as: $A \leq_p B$.

Claim. If A and B are decision problems, $B \in P$, and $A \leq_p B$, then $A \in P$; A can also be solved in polynomial time. *Proof.* Since $A \leq_p B$, there is a polynomial-time computable function F mapping instances of A to B and preserving answers. Let x be an instance of A. To decide x, we compute $f(x)$ to acquire an instance of B which can be fed to our algorithm to solve B. Whatever B answers, we answer for

Polynomial-Time Reductions continued on next page...

A. Realize that the running time = cost of computing $f(x)$+ running B on $f(x)$. Computing $f(x)$ is done in polynomial time, and so is the latter operation: this is a sum of two polynomials, which is polynomial. □

Claim. If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

Claim. If $A \leq_p B$ and $B \in NP$, then $A \in NP$. The formal definition can be used to prove this claim.

NP-Complete

A decision problem A is *NP-hard* if $B \leq_p A$ for every $B \in NP$. Similarly, a decision problem A is *NP-complete* if A is *NP-hard* and $A \in NP$. We can then say that each *NP-complete* problem is the "hardest" problem in NP. If a single *NP-complete* problem is in P, then $P = NP$. However, we do not know if $P = NP$ because we do not know if there is a polynomial-time solution to the hardest problem in NP.

NOTE One can prove a problem is *NP-hard* by seeing if an *NP-complete* can reduce to that problem in polynomial time.

Some problems that are *NP-complete* are the following:

- The Hamiltonian Cycle Problem, `HAM-CYCLE` (using `VERTEX-COVER`),
- The Traveling Salesman Problem (TSP) (using above),
- `CIRCUIT-SAT` (see page 1072),
- `SAT` (using above; page 1079),
- `3-CNF-SAT` (using above; page 1082),
- `CLIQUE` (using above; page 1086),
- `VERTEX-COVER` (using above; page 1089), and
- `SUBSET-SUM` (using `3-CNF-SAT`; page 1097).

NOTE For the final, 30% will be algorithm design (DC, Greedy, DP, Graph problems).