Algorithms and Complexity

Algorithms and Complexity

CE 377K

February 12, 2015

REVIEW

HW 1 due today

Nodes, links, path, cycle, acyclic

Strongly/weakly connected, tree, spanning tree

$A(i)$ and $B(i)$

n and m

ALGORITHMS AND COMPLEXITY

In network optimization problems, the time is usually expressed in terms of the number of links m and the number of nodes n, although it can depend on other problem features as well.

A few difficulties:

- The number of steps can vary, even for problems of the same size. (Sometimes we are lucky, sometimes we are not.)
- It can be difficult to calculate the exact number of steps required.

The solution to these difficulties is the use of "big O" notation.

The big O notation is used to express the *worst case* behavior as the problem grows asymptotically larger and larger.

Formally, a function $f(x)$ is $O(g(x))$ if $f(x) \leq Cg(x)$ for some constant $g(x)$ and when x is sufficiently large.

Examples.

- $x^2 + 2x + 100$ is $O(x^2)$
- $3(x-2)^3$ is $O(x^3)$
- $3(x-2)^3$ is $O(x^4)$
- $x^{1000} + e^x$ is $O(e^x)$

It is often much easier to calculate the number of steps an algorithm needs in terms of "big O" notation, for a few reasons:

- We don't have to keep track of details, only the most significant terms.
- If something is difficult to calculate, we can work with upper bounds.

Examples.

- If a network has no "parallel" links, m is $O(n^2)$.
- If the number of links adjacent to each node is no more than a constant, m is $O(n)$.

Over the next few weeks, as we see different kinds of network algorithms we will be comparing their big-O complexity as well.

(This is surprisingly important. Even switching from an $O(n^2)$ to an $O(n \log n)$ algorithm can make a huge difference for large problems.)

A "brute force" solution to the shortest path problem is to list off all possible paths between two nodes in a network. What is the complexity of this?

This method is $O(n!)$ — you really don't want to do this. In fact you can solve this problem easily in $O(n^2)$ time, and can reduce this further by being clever.

If the big-O complexity of an algorithm is polynomial in the problem size, it is said to be a *polynomial* algorithm (in P).

Intuitively, problems in P are "easy" while problems not in P are considered "hard."

Interestingly: the shortest path problem is in P, but not the longest path problem or the traveling salesperson problem.

The minimum spanning tree, shortest path, maximum flow, and minimum cost flow problems are also in P.

This is related to one of the biggest open problems in computer science. If you can find a polynomial algorithm for the traveling salesperson problem, you will win a \$1 million prize and almost certainly win the Nobel prize in economics.

BASIC SEARCH ALGORITHM

Let's say we want to determine whether a path exists connecting node r to node s (both given).

(If we know how to do this, we can determine if a network is strongly connected. How?)

Surprisingly, this problem is not much easier than determining whether node r is connected to *all* of the other nodes.

The search algorithm works by determining the set of all nodes reachable from r via some path. Call this set $C(r)$

Algorithm

1. Set $C(r) \leftarrow \{r\}$
2. Initialize the *scan eligible list* $SEL \leftarrow \{r\}$
3. Choose some node $i \in SEL$ and remove it from SEL.
4. "Scan" node i by repeating the following steps for each link $(i, j) \in A(i)$:
   1. If node j is not in $C(r)$, add it to SEL: $SEL \leftarrow SEL \cup \{j\}$
   2. Mark node j as reachable: $C(r) \leftarrow C(r) \cup \{j\}$
5. If SEL is empty, terminate. Otherwise, return to step 3.

Example

Proof of correctness

To show that an algorithm is "correct" we must show two things:

- It must eventually terminate.
- When it terminates, it must have the correct answer.

How do we know that our search algorithm terminates?

At each iteration, one node is removed from SEL.

If the algorithm does not terminate, this means that some node must be added to SEL infinitely many times.

However, a node is only added to SEL if it is not already in $C(r)$, and then it is added to $C(r)$ immediately afterwards.

Therefore, no node can enter SEL more than once.

So the algorithm must terminate.

How do we know that it terminates with the right answer?

By contradiction, assume that when the algorithm terminates $C(r)$ is not the set of all nodes connected to r.

There are two possibilities. $C(r)$ either contains nodes which are not connected to r, or there are nodes connected to r not in $C(r)$.

For the first case, we can show that at any point in time $C(r)$ only contains nodes connected to r. (It does so upon initialization, and nodes are only added to $C(r)$ when there is a link to that node from another node connected to $C(r)$)

For the second case, assume that there is some path between r and i but $i \notin C(r)$ when the algorithm terminates. Find the last node j in the path which is in $C(r)$, so the next node in the path k is not in $C(r)$.

When j was added to $C(r)$, it was also added to SEL. Since the algorithm terminated, j must have been removed from SEL and scanned.

When this happened, k would have been added to both SEL and $C(r)$, a contradiction.

Complexity

We have already shown that nodes are scanned at most once.

When node i is scanned, we must perform $3|A(i)|$ steps (checking if a node is in SEL, adding it to SEL if not, and adding it to $C(r)$.)

So at worst $\sum_{i \in N} 3|A(i)| = 3m$ computations are performed, which is $O(m)$.

As stated, the algorithm only determines whether or not paths exist between i and other nodes, it does not actually tell you what the paths are.

Is there some way to modify the algorithm to provide specific connecting paths as well?

TOPOLOGICAL ORDER

A *topological order* is an assignment of numbers to nodes so that links always connect lower-numbered nodes to higher-numbered nodes.

Not all networks have a topological order. It turns out that a topological order exists if and only if the network is acyclic.

Easy part: if a network has a cycle it cannot have a topological order.

Harder part: if a network has no cycles it *always* has a topological order.

The easiest way to do this is by giving an algorithm which will produce a topological order given any acyclic network.

Algorithm

1. Initialize the set SEL with all nodes with no incoming links ($|B(i)| = 0$)
2. Set the current node label $k \leftarrow 1$
3. Choose any node i from SEL and delete it from the set.
4. Assign node i the label k
5. Delete node i from the network, along with any links in $A(i)$.
6. As a result if this deletion, if there are any new nodes j such that $|B(j)| = 0$, add them to SEL.
7. If SEL is empty, terminate. Otherwise, increase k by 1 and return to step 3.

Example

Correctness

Why must the algorithm eventually terminate?

Nodes are deleted from the network after being scanned, so can never re-enter *SEL*. So eventually the list must be empty.

For the topological ordering to be correct, any incoming links to a node must come from a lower-labeled node.

If a node initially has no incoming links, its label can be set arbitrarily.

If a node initially has incoming links but later does not (after deleting some nodes and links), then we just need to use a label which is higher than any used so far.

Complexity

Nodes enter the scan list exactly once.

After scanning each node, we have to delete that node and all of its outgoing links ($1 + |A(i)|$ steps).

So, in total we perform $\sum_{i \in N}(1 + |A(i)|) = n + m$ steps.

So this algorithm is $O(n + m)$, or simply $O(m)$ (why?)

What would happen if we tried to use this algorithm on a network with cycles?