MATH 566: Lecture 7 (09/10/2024)

Today: * Computational complexity
* Search algorithms – generic search

Def An algorithm is said to run in $O(f(n))$ time if for some numbers $c>0$ (real) and $n_0>0$ ($\in \mathbb{N}$), the time $T(n)$ taken by the algorithm is at most $c f(n)$ for all $n \geq n_0$

could be $f(n,m,p,\ldots)$ for multiple dimensions

We can define the notion of $O(\cdot)$ for functions:

Def A function $f(n)$ is $O(g(n))$ if there exist numbers $c>0$ and $n_0>0$ such that $f(n) \leq c\, g(n)$ $\forall\, n \geq n_0$.

Formally, $O(g(n))$ is the set of all such functions. Hence we say $f(n)$ is **in** $O(g(n))$.

$O(\cdot)$ as a function gives the most dominant term in the input, e.g.,

$$O\left(100n + n^2 + 0.00001 n^3\right) = O(n^3).$$

$O(\cdot)$ gives a convenient way to ignore coefficients and lower order terms in polynomial expressions.

<u>Size of a problem</u>: #bits needed to store the problem, i.e., represent all data of the problem.

For instance, let $0 \leq A_{ij} < 2^{51}$; then each A_{ij} takes upto 51 bits.

If each element needs K bits, then the algorithm uses $O(mnK)$ storage. Typically, $K = O(\log(A_{max}))$, where

$$A_{max} = \max_{i,j} \{|A_{ij}|, |B_{ij}|\}$$

for adding two matrices.

Big Ω and big Θ notations

→ while $O(\cdot)$ gives upper bound, we also talk about a lower bound.

Def An algorithm is said to run in $\Omega(f(n))$ time if for some numbers c' (>0, real) and $n_0'>0$, for all instances with $n \geq n_0$, the algorithm takes at least $c'f(n)$ time on **some** problem instance.

Notice the difference in the definitions of $O(\cdot)$ and $\Omega(\cdot)$.

$O(f(n))$: **every** instance takes at most $cf(n)$ time (for $n \geq n_0$)

$\Omega(f(n))$: **some** instance takes at least $c'f(n)$ time (for $n \geq n_0'$).

Def An algorithm is said to be $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$.

↳ $\equiv$ said to run in $\Theta(f(n))$ time

Note that the corresponding definition(s) for functions are a bit different.

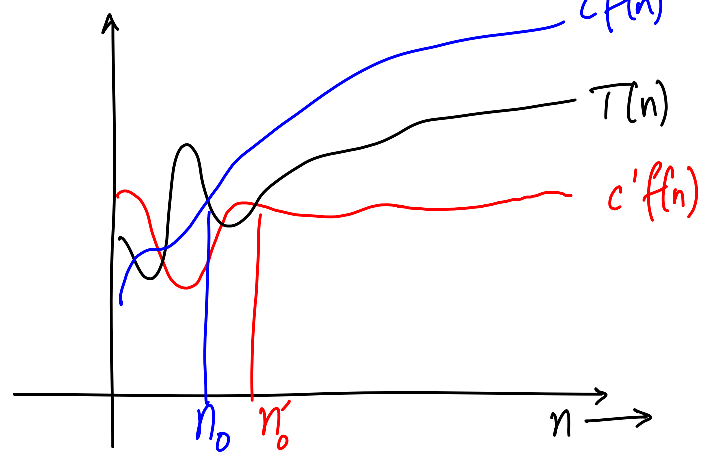
Def A function $f(n)$ is $\Omega(g(n))$ if there exist numbers $c'>0$ and $n_0'>0$ such that $f(n) \geq c'g(n)$ $\forall\, n \geq n_0'$.

Def A function $f(n)$ is $\Theta(g(n))$ if there exist numbers $c, c'>0$ and $n_0>0$ such that $c'g(n) \leq f(n) \leq cg(n)$ $\forall\, n \geq n_0$.

↳ can start with n_0, n_0' and take $\max(n_0, n_0')$ here.

Here is a typical function for running time and how it compares with lower and upper bounds as n becomes large.

T(n): running time



<u>Example</u> Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

{One n_0 is enough; we can use the larger of n_0 and n_0'}

To show this result, we want to find $c>0, c'>0$ and $n_0>0$ such that $c'n^2 \le \frac{1}{2}n^2 - 3n \le cn^2 \;\forall\; n \geqslant n_0$.

$$\Rightarrow \quad c' \le \frac{1}{2} - \frac{3}{n} \le c$$

at $n=6$, we get $\frac{1}{2} - \frac{3}{n} = 0$. We can look at n that are 7 or higher for possible choices.

We can choose $c = \frac{1}{2}$, $n_0 = 7$, $c' = \frac{1}{14}$.

We are interested in **polynomial time algorithms** where the worst-case complexity is bounded by a polynomial function of problem parameters $n, m, \log C, \log U$, where $C = \max_{(i,j)\in A} |c_{ij}|$, $U = \max u_{ij}$.

e.g., $O(mn)$, $O(m + n\log C)$.

Strongly polynomial time algorithm: worst case complexity is bounded by a polynomial in only m and n, i.e., it is independent of $\log C, \log U$.

e.g., $O(m^2 n)$. (strongly poly-time algos $\subset$ poly-time algos)

Exponential time algorithm: running time is $O(f(\cdot))$ where $f(\cdot)$ is exponential in $m, n, \log C, \log U$ → recall: size of problem varies as log of C_{ij}, etc.

e.g., $O(2^n)$, $O(n!)$,

e.g., an exact algorithm for the traveling salesman problem (TSP).

Pseudopolynomial time algorithm: running time is bounded by a polynomial in m, n, C, U. → and **not** $\log C$, $\log U$.

e.g., $O(m + nC)$. (pseudopoly-time algos $\subset$ exponential time algos)

Search Algorithms

Goal: find all nodes in G that satisfy a property.

e.g., 1. find all nodes in $G=(N,A)$ that are reachable by directed paths from node s;

2. find all nodes that can reach node t via directed paths.

We will discuss
- * generic search
- * breadth-first search (BFS)
- * depth-first search (DFS)

INPUT: $G=(N,A)$ and a node $s \in N$.

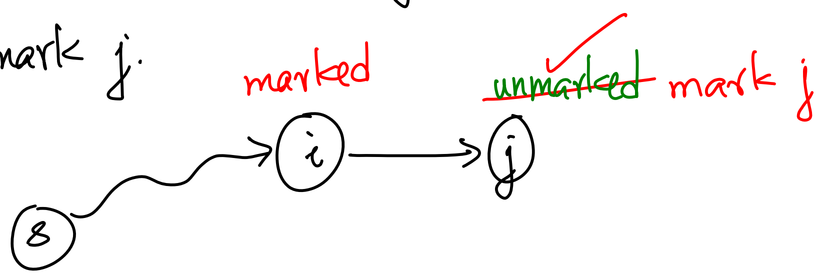
OUTPUT: $\{ j \in N \mid \text{a directed path exists in } G \text{ from } s \text{ to } j \}$.

At any stage of the algorithm, a node is either **marked** or **unmarked**.

marked → is reachable from s

unmarked → status of reachability from s yet to be determined

Critical step: If node i is marked, node j is unmarked, and $(i,j) \in A$, then mark j.



Such an arc (i,j) where i is marked and j is unmarked is said to be **admissible**.

We use **pred(i)** indices to store directed paths from s to i; and the order of traversal of the nodes in **order(i)**.

Here is the pseudocode from AMO:

order(s) = s; works only for s = 1.

```
algorithm search;
begin
    unmark all nodes in N;   <- maintain a binary n-vector (array)
    mark node s;
    pred(s) : = 0;
    next : = 1;   <- counter
    order(s): = s̶; -> next;
    LIST : = {s}
    while LIST ≠ Ø do
    begin
        select a node i in LIST;
        if node i is incident to an admissible arc (i, j) then
        begin
            mark node j;
            pred( j) : = i;
            next : = next + 1;
            order( j) : = next;
            add node j to LIST;
        end
        else delete node i from LIST;
    end;
end;
```

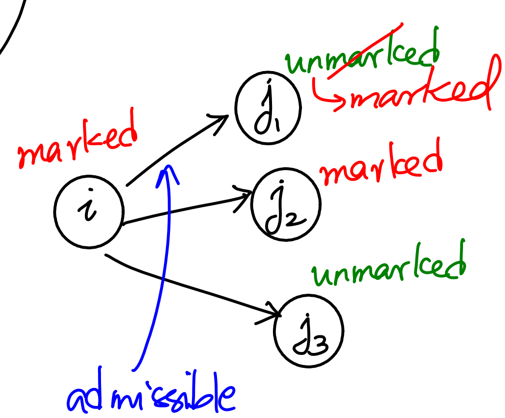
initialization

order(s) = 2 ⟹ node 5 is the 2nd node visited

Figure 3.4 Search algorith

look at $A(i)$ list

Run through $A(i)$ list: The **current arc** (i,j) is the next candidate arc from $A(i)$.



Depending on how we maintain and update the LIST, we get different versions of the generic search.

If we maintain LIST as a queue, i.e., select node i from the front, add node j to back of LIST, we get **breadth-first search** (BFS). The nodes are examined in a **first-in first-out** (**FIFO**) order.