Advanced Algorithms (I)

Yijia Chen
Fudan University

About me

Professor in Computer Science.

Ph.D. in Computer Science (Shanghai Jiaotong), Ph.D. in Mathematics (Freiburg).

Research interests: logic in computer science, computational complexity, and graph theory.

Contents

Edsger W. Dijkstra

Computer Science is no more about computers than astronomy is about telescopes.

Basics

Big-O notation

Big-O notation

$f(n)$ and $g(n)$ are the running times of two algorithms on inputs of size n.

> *Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ (which means that "f grows no faster than g") if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.*

$f = O(g)$ is a very loose analog of "$f \leq g$." It differs from the usual notion of because of the constant c, so that for instance $10n = O(n)$.

Why disregard the constant?

We are choosing between two algorithms: One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps.

Which is better? The answer depends on n:

- If $n \leq 5$, then $f_1(n) \leq f_2(n)$.
- If $n > 5$, then $f_2(n) < f_1(n)$.

f_2 scales much better as n grows, and therefore it is superior.

This superiority is captured by the big-O notation: $f_2 = O(f_1)$:

$$\frac{f_2(n)}{f_1(n)} = \frac{2n + 20}{n^2} \leq 22$$

for all $n \in \mathbb{N}$. On the other hand, $f_1 \neq O(f_2)$, since *the ratio*

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n + 20}$$

can get arbitrarily large.

Why disregard the constant? (cont'd)

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$.
Suppose we have a third algorithm which uses

$$f_3(n) = n + 1$$

steps. Is this better than f_2? Certainly, but only by a constant factor.

The discrepancy between f_2 and f_3 is tiny compared to the huge gap between f_1 and f_2. In order to stay focused on the big picture, we treat functions as equivalent if they differ only by *multiplicative constants*.

Returning to the definition of big-O, we see that $f_2 = O(f_3)$:

$$\frac{f_2(n)}{f_3(n)} = \frac{2n + 20}{n + 1} \leq 20$$

and $f_3 = O(f_2)$ with $c = 1$.

Other similar notations

Just as $O(\cdot)$ is an analog of $\leq$, we also define analogs of $\geq$ and $=$ as follows:

$f = \Omega(g)$ means $g = O(f)$,

$f = \Theta(g)$ means $f = O(g)$ and $f = \Omega(g)$.

Recall $f_1(n) = n^2$, $f_2(n) = 2n + 20$ and $f_3(n) = n + 1$. Then

$$f_2 = \Theta(f_3) \quad \text{and} \quad f_1 = \Omega(f_3).$$

Divide and conquer

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

The algorithm

```
MERGESORT(a[1 ... n])
// Input: an array of numbers a[1 ... n]
// Output: A sorted version of this array
1.  if n > 1 then
2.    return MERGE(MERGESORT(a[1 ... ⌊n/2⌋]),
3.                 MERGESORT(a[⌊n/2⌋ + 1 ... n]))),
4.         else return a.
```

```
MERGE(x[1 ... k], y[1 ... ℓ])
// Input: two sorted arrays x and y
// Output: A sorted version of the union of x and y
1.  if k = 0 then return y[1 ... ℓ]
2.  if ℓ = 0 then return x[1 ... k]
3.  if x[1] ≤ y[1]
4.    then return x[1]∘MERGE(x[2 ... k], y[1 ... ℓ])
5.    else return y[1]∘MERGE(x[1 ... k], y[2 ... ℓ]).
```

The time analysis

The recurrence relation:

$$T(n) = 2T(n/2) + O(n);$$

By Master Theorem

$$T(n) = O(n \log n).$$

Master theorem

If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

Median

Median

The median of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller.
If the list has *even length*, there are two choices for what the middle element could be, in which case we pick the smaller of the two, say.

The purpose of the median is to summarize a set of numbers by a single, typical value.

Computing the median of n numbers is easy: just sort them. The drawback is that this takes $O(n \log n)$ time, whereas we would ideally like something *linear*.

We have reason to be hopeful, because sorting is doing far more work than we really need – we just want the middle element and don't care about the relative ordering of the rest of them.

Selection

Input: A list of numbers S; an integer k.
Output: The kth smallest element of S.

A randomized divide-and-conquer algorithm for selection

For any number v, imagine splitting list S into three categories:

- elements smaller than v, i.e., S_L;
- those equal to v, i.e., S_v (there might be duplicates);
- and those greater than v, i.e., S_R respectively.

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \leq |S_L| \\ v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

How to choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$|S_L|, |S_R| \approx \frac{|S|}{2}.$$

If we could always guarantee this situation, we would get a running time of

$$T(n) = T(n/2) + O(n) = O(n).$$

But this requires picking v to be the median, which is our ultimate goal!

Instead, we follow a much simpler alternative: we pick v randomly from S.

How to choose v? (cont'd)

Worst-case scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \cdots + 1 = \Theta(n^2)$$

Best-case scenario: $O(n)$.

Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the average running time lie? Fortunately, it lies very close to the best-case time.

The efficiency analysis

v is *good* if it lies within the 25th to 75th percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good,

Lemma
On average a fair coin needs to be tossed two times before a "heads" is seen.

Proof.

$$E := \text{expected number of tosses before head is seen}.$$

We need at least one toss, and it's heads, we're done.
If it's tail (with probability $1/2$), we need to repeat. Hence

$$E = 1 + \frac{1}{2}E,$$

whose solution is $E = 2$ □

The efficiency analysis (cont'd)

Let $T(n)$ be the **expected running time** on an array of size n, we get

$$T(n) \leq T(3n/4) + O(n) = O(n).$$

Depth-first search

The goal of depth-first search

What parts of the graph are reachable from a given vertex?

Exploring graphs

EXPLORE(G, v)

Input: $G = (V, E)$ is a graph; $v \in V$

Output: `visited`(u) is set to TRUE for all nodes u reachable from v

1. `visited`$(v) =$ TRUE
2. **for** each edge $(v, u) \in E$ **do**
3. **if** not `visited`(u) **then** EXPLORE(u)

Exploring graphs (cont'd)

Theorem

EXPLORE(G, v) *is correct, i.e., it visits exactly all nodes that are reachable from* v*.*

Proof.

Every node which it visits must be reachable from v:

EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v.

Every node which is reachable from v must be visited eventually:

If there is some u that EXPLORE misses, choose any path from v to u, and look at the last vertex v on that path that the procedure actually visited. Let w be the node immediately after it on the same path.

So z was visited but w was not. This is a contradiction: while EXPLORE was at node z, it would have noticed w and moved on to it. □

Depth-first search

DFS(G)

1. **for all** $v \in V$ **do**
2. `visited`(v) = FALSE
3. **for all** $v \in V$ **do**
4. **if** not `visited`(v) **then** EXPLORE(v)

Running time of DFS

Because of the `visited` array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.
2. A loop in which adjacent edges are scanned, to see if they lead somewhere new.

This loop takes a different amount of time for each vertex. The total work done in step 1 is then $O(|V|)$. In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly *twice*, once during EXPLORE(x) and once during EXPLORE(y).

The overall time for step 2 is therefore $O(|E|)$ and so the depth-first search has a running time of $O(|V| + |E|)$.

Breath-first search

Distances

DFS does not necessarily find the shortest paths.

Definition

The distance between two nodes is the length of the shortest path between them.

The algorithm

BFS(G, s)

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

```
1.  for all u ∈ V do
2.          dist(u) = ∞
3.  dist(s) = 0
4.  Q = [s] (queue containing just s)
5.  while Q is not empty do
6.          u = eject(Q)
7.          for all edge (u, v) ∈ E do
8.                  if dist(v) = ∞ then
9.                          inject(Q, v)
10.                         dist(v) = dist(u) + 1
```

Correctness and efficiency

Lemma

For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$ from s have their distances correctly set; (2) all other nodes have their distances set to ∞; and (3) the queue contains exactly the nodes at distance d.

Lemma

BFS has a running time of $O(|V| + |E|)$.

Dynamic programming

Longest increasing subsequences

The problem

In the longest increasing subsequence problem, the input is a sequence of numbers $a_1, \ldots, a_n$.

A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and an increasing subsequence is one in which the numbers are getting strictly larger.

The task is to find the increasing subsequence of *greatest* length.

Graph reformulation

Let's create a graph of all *permissible transitions*: establish a node i for each element a_i, and add directed edges (i, j) whenever it is possible for a_i and a_j to be consecutive elements in an increasing subsequence:

$$i < j \text{ and } a_i < a_j$$

- This graph G = (V,E) is a dag, since all edges (i, j) have $i < j$
- There is a one-to-one correspondence between increasing subsequences and paths in this dag.

Therefore, our goal is simply to find *the longest path in the dag*!

The algorithm

for $j = 1$ **to** n **do**
    $L(j) = 1 + \max\{L(i) \mid (i,j) \in E\}$
return $\max_j L(j)$

$L(j)$ is the length of the longest path – *the longest increasing subsequence* – ending at j plus 1.

Any path to node j must pass through one of its predecessors, and therefore $L(j)$ is 1 plus the maximum $L(\cdot)$ value of these predecessors.
If there are no edges into j, we take the maximum over the empty set, zero.

The final answer is the largest $L(j)$, since any ending position is allowed.

This is dynamic programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in *a single pass*:

> *There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.*

In our case, each subproblem is solved using the relation

$$L(j) = 1 + \max\{L(i) \mid (i,j) \in E\}$$

Independent sets in trees

It is well known that many NP-hard problems can be solved in polynomial time on trees, i.e., Independent-Set, Dominating-Set, 3-Colorability, etc.

Independent sets in trees

Recall that a subset $S \subseteq V$ is an independent set of graph $\mathcal{G} = (V, E)$ if there are no edges between vertices in S.

Simple dynamic programming on trees:

Given a tree $\mathcal{T}$ we first fix an arbitrary $r \in V(\mathcal{T})$ as the root. Then we compute from leaves to the root r for each node t

$$I(t) := \text{size of a largest independent set of the subtree hanging from } t$$

$$= \max\left\{1 + \sum_{\text{grandchildren } t' \text{ of } t} I(t'), \sum_{\text{children } t' \text{ of } t} I(t')\right\}.$$

Treewidth

Tree decompositions of graphs

Definition

Let $\mathcal{G}$ be a graph. A tree decomposition of $\mathcal{G}$ is a tuple $\left(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\right)$, where $\mathcal{T}$ is a tree and B_t the bag at t such that the following conditions are satisfied:

(T1) For every $v \in V(\mathcal{G})$ the set

$$T_v := \left\{ t \in V(\mathcal{T}) \;\middle|\; v \in B_t \right\}$$

is nonempty and connected in $\mathcal{T}$, i.e., $\mathcal{T}[T_v]$ is a subtree of $\mathcal{T}$.

(T2) For every $e \in E(G)$ there exists a $t \in V(\mathcal{T})$ such that $e \subseteq B_t$.

Tree decomposition of graphs, examples

1. The complete graphs $\mathcal{K}_n$ for $n \in \mathbb{N}$.
2. The trees.
3. The grids $\mathcal{G}_{n \times n}$ for $n \in \mathbb{N}$.

Treewidth

The width of a tree decomposition $\big(\mathcal{T}, (B_t)_{t\in V(\mathcal{T})}\big)$ is

$$\mathrm{width}\big(\mathcal{T}, (B_t)_{t\in V(\mathcal{T})}\big) := \max\big\{|B_t| - 1 \;\big|\; t \in V(\mathcal{T})\big\}.$$

The treewidth of $\mathcal{G}$ is

$$\mathrm{tw}(\mathcal{G}) := \min\Big\{\mathrm{width}\big(\mathcal{T},(B_t)_{t\in V(\mathcal{T})}\big) \;\Big|\; \big(\mathcal{T}, (B_t)_{t\in V(\mathcal{T})}\big) \text{ is a tree decomposition of } \mathcal{G}\Big\}.$$

Treewidth, examples

1. $\mathrm{tw}(\mathcal{K}_n) = n - 1$ for the complete graphs $\mathcal{K}_n$.
2. $\mathrm{tw}(\mathcal{T}) = 1$ for every tree $\mathcal{T}$ of size at least 2.
3. $\mathrm{tw}(\mathcal{G}_{n \times n}) = n$ for every grid $\mathcal{G}_{n \times n}$.

Smooth tree decomposition

Definition

A tree decomposition $\left(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\right)$ is smooth if for every $\{t, t'\} \in E(T)$ we have

$$|B_t \setminus B_{t'}| = |B_{t'} \setminus B_t| = 1.$$

Theorem

Every tree decomposition can be efficiently transferred to a smooth one of the same width.

Theorem

Every graph $\mathcal{G}$ has a smooth tree decomposition of width $\mathrm{tw}(\mathcal{G})$.

Make tree decomposition smooth

Let $\left(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\right)$ be a tree decomposition of width w.

1. Make bags equal size: We choose a node $r \in V(\mathcal{T})$ with $|B_r| = w + 1$ as the root. Let t be a child of r with $|B_t| \leq w$. Clearly

$$|B_r \setminus B_t| + |B_t| \geq w + 1.$$

   We add $w + 1 - |B_t|$ vertices in $B_r \setminus B_t$ to B_t. After repeating this procedure recursively from the root to leaves, every bag has size $w + 1$.

2. Remove repetition: If there is an edge $\{t, t'\} \in E(\mathcal{T})$ with $B_t = B_{t'}$, then we merge t' with t.

3. Interpolation: Let $\{t, t'\} \in E(\mathcal{T})$ with $|B_t \cap B_{t'}| < w$, i.e.,

$$B_t \setminus B_{t'} = \{u_1, \ldots, u_\ell\} \quad \text{and} \quad B_{t'} \setminus B_t = \{v_1, \ldots, v_\ell\}$$

   for some $\ell > 2$ and pairwise distinct $u_1, \ldots, u_\ell, v_1, \ldots, v_\ell$. We insert new nodes $t_1, \ldots, t_{\ell-1}$ between t and t' with

$$B_{t_i} := \left(B_t \cap B_{t'}\right) \cup \{v_1, \ldots, v_i, u_{i+1}, \ldots, u_\ell\}$$

   for every $i \in [\ell - 1]$.

The size of smooth tree decompositions

Theorem

For every smooth tree decomposition $\left(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\right)$ *of* $\mathcal{G}$ *we have*

$$|V(\mathcal{T})| \leq |V(\mathcal{G})|.$$

Theorem

$$|E(\mathcal{G})| \leq \mathsf{tw}(\mathcal{G}) \cdot |V(\mathcal{G})|.$$

$\mathrm{tw}(\mathcal{K}_n) = n - 1$

$\mathrm{tw}(\mathcal{K}_n) \leq n - 1$: Take a tree decomposition with a singleton tree.

$\mathrm{tw}(\mathcal{K}_n) \geq n - 1$: Let $\left(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\right)$ be a smooth tree decomposition of $\mathcal{K}_n$ of width $\mathrm{tw}(\mathcal{K}_n)$. We show that there exists a B_t with $|B_t| = n$.

Trivial if $|V(\mathcal{T})| = 1$. Otherwise choose a leaf t and let t' be its parent in $\mathcal{T}$. By the smoothness

$$B_t \setminus B_{t'} = \{v\} \text{ for some } v \in V(\mathcal{K}_n).$$

Since v is adjacent to every other vertex in $\mathcal{K}_n$, we see that

$$B_t = V(\mathcal{K}_n).$$

Helly property for trees

Theorem

Let $\mathcal{T}$ be a tree and $\mathcal{T}_1, \ldots, \mathcal{T}_n$ subtrees of $\mathcal{T}$ such that

$$V(\mathcal{T}_i) \cap V(\mathcal{T}_j) \neq \emptyset$$

for every $i, j \in [n]$. Then

$$\bigcap_{i \in [n]} V(\mathcal{T}_i) \neq \emptyset.$$

Proof (Yaokun Wu, 2006)

We prove by induction on the size of $\mathcal{T}$.

Trivial for $|V(\mathcal{T})| = 1$.

Otherwise, let t be a leaf of $\mathcal{T}$. If $t \in V(\mathcal{T}_i)$ for every $i \in [n]$, then we are done.

Now assume $t \notin V(\mathcal{T}_i)$ for some $i \in [n]$. Consider

$$\mathcal{T} \setminus \{t\}, \mathcal{T}_1 \setminus \{t,\}, \ldots, \mathcal{T}_n \setminus \{t\}.$$

Then

- every $\mathcal{T}_i \setminus \{t\}$ is a subtree of $\mathcal{T} \setminus \{t\}$;
- $V(\mathcal{T}_i \setminus \{t\}) \cap V(\mathcal{T}_j \setminus \{t\}) \neq \emptyset$ for every $i, j \in [n]$.

The result follows from the induction hypothesis.

$\mathrm{tw}(\mathcal{K}_n) = n - 1$, again

Theorem

Let $\mathcal{G} = (V, E)$ be a graph and $S \subseteq V$ a clique. Then for every tree decomposition $\left(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\right)$ there is a node $t \in V(\mathcal{T})$ with $S \subseteq B_t$.

Proof.

For every $v \in V$ recall

$$T_v := \{t \in V(\mathcal{T}) \mid v \in B_t\}$$

induces a subtree $\mathcal{T}_v := \mathcal{T}[T_v]$ of $\mathcal{T}$.

Clearly for every $u, v \in S$ we have

$$V(\mathcal{T}_u) \cap V(\mathcal{T}_v) = T_u \cap T_v \neq \emptyset,$$

since there is an edge $\{u, v\}$ in $\mathcal{G}$.

The result follows from Helly property. □

Computing the treewidth

Theorem (Bodlaender, 1996)

The problem

TREEWIDTH	
Input:	A graph $\mathcal{G}$ and a number $k \in \mathbb{N}$.
Problem:	Decide whether $\mathrm{tw}(\mathcal{G}) \leq k$ and if so output a tree decomposition of $\mathcal{G}$ with width $\leq k$.

can be computed in time

$$2^{k^{O(1)}} \cdot \|\mathcal{G}\|.$$

Corollary

For every $k \in \mathbb{N}$ there is a linear time algorithm which on every graph $\mathcal{G}$ either outputs a tree decomposition of $\mathcal{G}$ of width $\leq k$ or reports that $\mathrm{tw}(\mathcal{G}) > k$.

Independent sets via tree decompositions

Independent sets via tree decompositions (1)

Let $\mathcal{G}$ be a graph and $\big(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\big)$ a smooth tree decomposition of $\mathcal{G}$. And let $w := \text{width}\big(\mathcal{T}, (B_t)_{t \in V(\mathcal{T})}\big)$.

We fix an arbitrary node $r \in V(\mathcal{T})$ as the root of $\mathcal{T}$.

Let $t \in V(\mathcal{T})$. We define $\mathcal{G}_t$ as the induced subgraph of G on vertices in B_t.

Furthermore, $\mathcal{G}_{\leq t}$ is the induced subgraph of $\mathcal{G}$ on vertices in

$$B_t \cup \bigcup_{\text{descendants } t' \text{ of } t} B_{t'}.$$

Independent sets via tree decompositions (2)

By dynamic programming we compute for every $t \in V(\mathcal{T})$ and every $X \subseteq B_t$ independent in $\mathcal{G}_t$:

$$I(t, X) := \text{size of a largest independent set } I \text{ of } \mathcal{G}_{\leq t} \text{ with } I \cap B_t = X$$
$$= |X| + \sum_{\text{children } t' \text{ of } t} \max\left\{ I(t', X') - |X' \cap X| \;\middle|\; X' \cap B_t = X \cap B_{t'} \right\}.$$

Note there are at most

$$|V(\mathcal{T})| \cdot 2^{w+1} \leq |V(\mathcal{G})| \cdot 2^{w+1}$$

many $I(t, X)$.

Independent sets via tree decompositions (3)

Theorem

For every $k \in \mathbb{N}$ there is a linear time algorithm which on every graph $\mathcal{G}$ with $\mathsf{tw}(\mathcal{G}) \leq k$ outputs a largest independent set in $\mathcal{G}$.