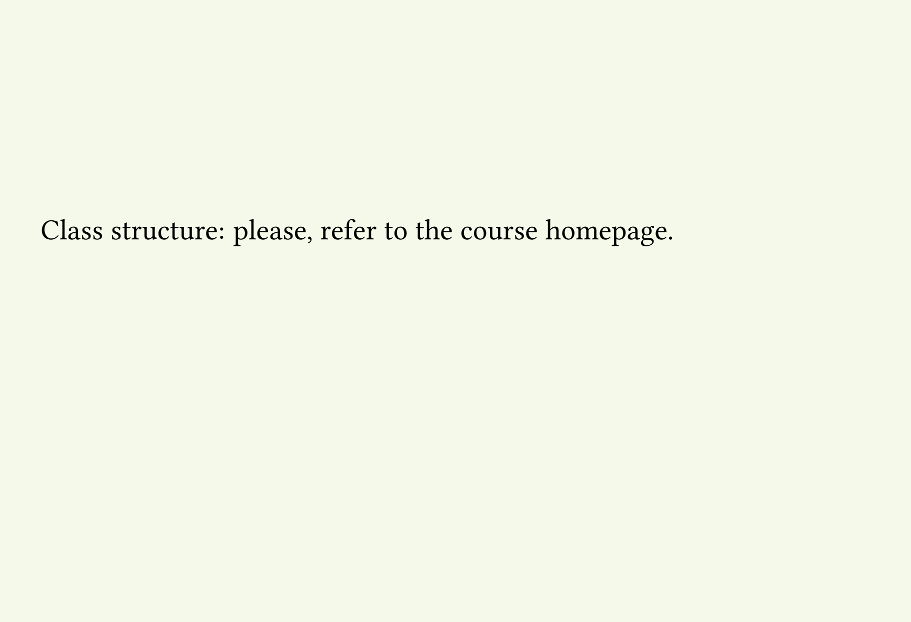
Algorithms

Péter Gács
Freely using the textbooks by
Kleinberg-Tardos and Cormen-Leiserson-Rivest-Stein,
and the slides of Kevin Wayne

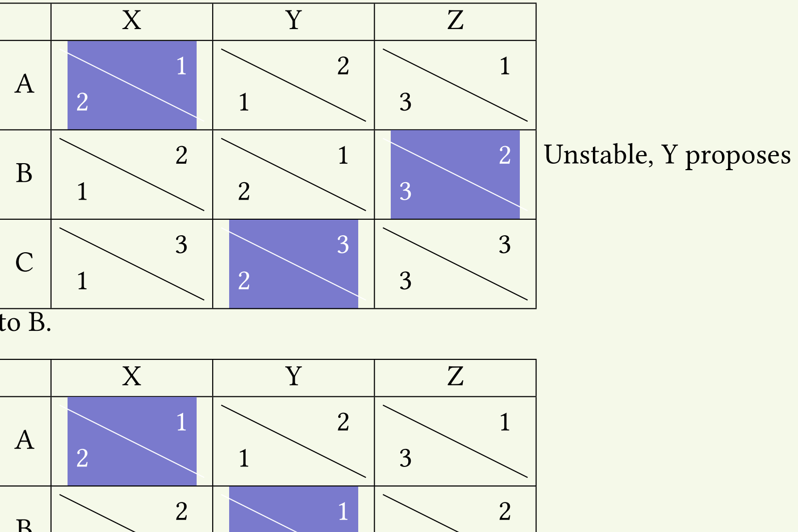
Computer Science Department
Boston University

Fall 2013

It is best not to print these slides, but rather to download them frequently, since they will probably evolve during the semester.


Class structure: please, refer to the course homepage.

Women, say A, B, C. Men, say X, Y, Z. Matchings.
Each orders the other sex by preference.
Instability in a matching: a non-matched pair that would be an improvement for both the man and the woman.



	X	Y	Z
A	**2 \ 1**	1 \ 2	3 \ 1
B	1 \ 2	2 \ 1	**3 \ 2**
C	1 \ 3	**2 \ 3**	3 \ 3

Unstable, Y proposes to B.

	X	Y	Z
A	**2 \ 1**	1 \ 2	3 \ 1
B	1 \ 2	**2 \ 1**	3 \ 2
C	1 \ 3	2 \ 3	**3 \ 3**

Stable, even if Z and C are unhappy …

Another stable matching:

	X	Y	Z
A	2 \ 1	**1 \ 2**	3 \ 1
B	**1 \ 2**	2 \ 1	3 \ 2
C	1 \ 3	2 \ 3	**3 \ 3**

Question 1 Is there always a stable matching?

The answer is not obvious. For example, in the stable roommate problem, where there is no sex distinction, but everybody can be paired with everybody, the answer is sometimes no.

Question 2 Provided there is a stable matching, how to find it (efficiently, by an algorithm)?

Interestingly, here we will answer Question 1 directly by an algorithm, answering therefore also Question 2: "men propose, women dispose". That is, starting from an empty matching, repeatedly an unmatched man proposes to the woman he prefers most (among the ones not matched to somebody they prefer better).

Theorem Terminates in $O(n^2)$ steps.

Indeed, since each step either increases the number of matches (without harming any woman) or benefits some woman.

Theorem When the algorithm stops, we have a stable matching.

Indeed, no man has any reason to switch, since no woman he prefers more is available to him.

This solution is an example of an approach to problem solution that Descartes recommended as often fruitful: introduce some new order to even where there is not any. Of course, there are many possible such ideas (tricks, heuristics). Generating them is only one part of the problem solution, frequently the smaller one: *checking whether the idea works* is just as important! (Example, RSA: Adleman's role was to break the cryptosystems proposed by Rivest and Shamir—not all, only the first 41.)

Question Is there an example (when not doing “men propose, women dispose”) of an infinite series of switches through unstable matchings?
Restrict to “reasonable” switches!

How unique is the matching we found?

Theorem Each man gets the woman most preferred by him among those available in some stable matching (valid partners).

Proof in class, look at the book and Kevin Wayne's slides.

Questions

- Is there an example where the Gale-Shapley algorithm indeed takes $\Omega(n^2)$ steps?
- Is it true that every possible algorithm needs $\Omega(n^2)$ steps? Assume that inspecting each preference is an extra step. How many preferences do we have to inspect?

1. Suppose that among the n women there are k rich ones, and also among the n men there are k rich ones. The preference lists are such that everybody prefers rich persons to the others. Show that in each stable matching, rich men are married with rich women.
2. Not all pairs are acquainted, and only acquainted pairs are allowed to match. There is a natural notion of stable partial matching for this case. Show that there is always such a matching.

Five example problems

Instead of writing, say, $O(n \log n)$, I will just write $n \log n$. But Big-Oh is always understood here, see below.

Interval scheduling A simple greedy algorithm gives $n \log n$.

Weighted interval scheduling Dynamic programming, $n \log n$

Bipartite matching n^k

Independent set NP-complete. No known algorithm is much better than brute force (exponential).

Competitive facility location (say Shell and Mobil) PSPACE-complete

Let $f(n), g(n)$ be some positive functions. The following asymptotic notation will be used.

- $f(n) = O(g(n))$, or $f(n) \overset{*}{<} g(n)$, if there is a constant $c>0$ with $f(n) \leqslant c \cdot g(n)$ for all n.
- $f(n) = o(g(n))$ or, equivalently, $f(n) \ll g(n)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) = \Omega(g(n))$ if there is a constant $c>0$ with $f(n) \geqslant c \cdot g(n)$ for all n. (This is the same as $g(n) = O(f(n))$, but we generally expect a simple formula on the right-hand side.)
- $f(n) = \Theta(g(n))$ or $f(n) \overset{*}{=} g(n)$, if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

The most important function classes: log, logpower, linear, power, exponential.

Some simplification rules.

- Addition: take the maximum. Do this always to simplify expressions. *Warning*: do it only if the number of terms is constant!
- An expression $f(n)^{g(n)}$ is generally worth rewriting as $2^{g(n)\log f(n)}$. For example, $n^{\log n} = 2^{(\log n)\cdot(\log n)} = 2^{\log^2 n}$.
- But sometimes we make the reverse transformation:

$$3^{\log n} = 2^{(\log n)\cdot(\log 3)} = (2^{\log n})^{\log 3} = n^{\log 3}.$$

  The last form is easiest to understand, showing n to a constant power $\log 3$.

$$n/\log\log n + \log^2 n \sim n/\log\log n.$$

Indeed, $\log\log n \ll \log n \ll n^{1/2}$, hence $n/\log\log n \gg n^{1/2} \gg \log^2 n$.

Order the following functions by growth rate:

$$
\begin{aligned}
&n^2 - 3\log\log n && \sim n^2,\\
&\log n/n, && \\
&\log\log n, && \\
&n\log^2 n, && \\
&3 + 1/n && \sim 1,\\
&\sqrt{(}5n)/2^n, && \\
&(1.2)^{n-1} + \sqrt{n} + \log n && \sim (1.2)^n.
\end{aligned}
$$

Solution:

$$
\begin{aligned}
\sqrt{(}5n)/2^n &\ll \log n/n \ll 1 \ll \log\log n\\
&\ll n/\log\log n \ll n\log^2 n \ll n^2 \ll (1.2)^n.
\end{aligned}
$$

Arithmetic series

Geometric series its rate of growth is equal to the rate of growth of its largest term.

Example

$$\log n! = \log 2 + \log 3 + \cdots + \log n = \Theta(n \log n).$$

Indeed, upper bound: $\log n! < n \log n$.
Lower bound:

$$\begin{aligned}\log n! &> \log(n/2) + \log(n/2 + 1) + \cdots + \log n > (n/2)\log(n/2) \\ &= (n/2)(\log n - 1) = (1/2)n \log n - n/2.\end{aligned}$$

Example

Prove the following, via rough estimates:

$$1 + 2^3 + 3^3 + \cdots + n^3 = \Theta(n^4),$$
$$1/3 + 2/3^2 + 3/3^3 + 4/3^4 + \cdots < \infty.$$

Example

$$1 + 1/2 + 1/3 + \cdots + 1/n = \Theta(\log n).$$

Indeed, for $n = 2^{k-1}$, upper bound:

$$\begin{aligned} & 1 + 1/2 + 1/2 + 1/4 + 1/4 + 1/4 + 1/4 + 1/8 + \dots \\ & \qquad = 1 + 1 + \cdots + 1 \ (k \text{ times}). \end{aligned}$$

Lower bound:

$$\begin{aligned} & 1/2 + 1/4 + 1/4 + 1/8 + 1/8 + 1/8 + 1/8 + 1/16 + \dots \\ & \qquad = 1/2 + 1/2 + \cdots + 1/2 \ (k \text{ times}). \end{aligned}$$

There is no general recipe. In general, greedy algorithms assume that

- There is an objective function $f(x_1, \dots, x_n)$ to optimize that depends on some choices $x_1, \dots, x_n$. (Say, we need to maximize $f()$.)
- There is a way to estimate, roughly, the contribution of each choice x_i to the final value, but without taking into account how our choice will constrain the later choices.
- The algorithm is greedy if it still makes the choice with the best contribution.

The greedy algorithm is frequently not the best, but sometimes it is.

Interval scheduling

Example: activity selection. Given: activities

$$[s_i, f_i)$$

with starting time s_i and finishing time f_i. Goal: to perform the largest number of activites.

Greedy algorithm Repeatedly choose the activity with the smallest f_i compatible with the ones already chosen.

Rationale This restricts our later choices least.

Other possible orders: construct counterexamples.

Example

$$(1,4),(3,5),(0,6),(5,7),(3,8),(5,9),(6,10),$$
$$(8,11),(8,12),(2,13),(12,14).$$

Chosen: $(1,4),(5,7),(8,11),(12,14)$.

Is this correct? Yes, by design, since we always choose an activity compatible with the previous ones.

Is this best? By induction, it is sufficient to see that in the first step, the greedy choice is best possible. And it is, since if an activity is left available after the some choice, it is also left available after the greedy choice.

How efficient? The cost is just the cost of sorting.

Interval partitioning

Suppose our intervals are activities to be performed in classrooms, and all must be scheduled. But we have more than one classroom. The goal is to schedule them in the smallest number of classrooms.

Lower bound If there is a time contained in k activites then we need at least k classrooms. Let the depth d be the size of the largest such overlap.

A "greedy" algorithm
- Sort the activities by starting time.
- Repeatedly, schedule the next activity in the first available classroom.

Why does it need only d classrooms? By contradiction, look at the first time when d classrooms do not suffice.

Minimize maximum lateness

Now each job i has a duration t_i and a deadline d_i. We may not be able to meet each deadline, but we want to mimize the maximum lateness among all jobs.

Greedy algorithm Earliest deadline first.

Why optimal? Consider a different order, say the original one:
: $i = 1, 2, \ldots, n$. Look at a consecutive pair for which this order is different from the order of their deadlines: $s_{i+1} < s_i$, $d_i < d_{i+1}$. After swap, the lateness of i increases by t_{i+1}, while the lateness of $i + 1$ decreases by the same. Since the lateness of i is smaller than that of $i + 1$, we decrease the larger and increase the smaller—the maximum lateness decreases.

Other possible choices order by length, by slack: $d_i - t_i$.
: Counterexamples.

Sometimes, it takes some thinking to see that a problem is a shortest path problem.

Example: how to break up some composite words?

> *Personaleinkommensteuerschätzungskommissionsmitglieds-reisekostenrechnungsergänzungsrevisionsfund* (Mark Twain)

With a German dictionary, break into relatively few components.

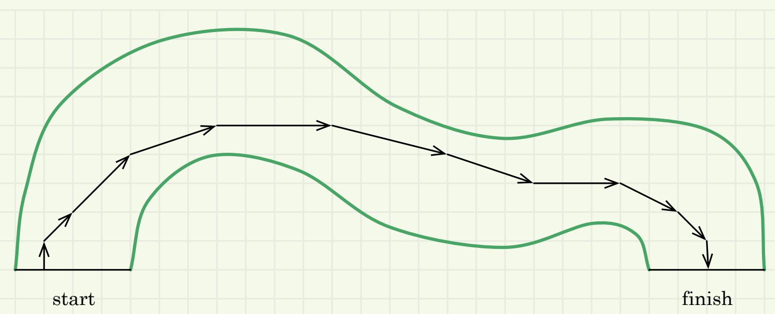
Graph points all division points of the word, including start and end.

Edges if the word between the points is in the dictionary (maybe without the “s” at the end).

Path between the start and end corresponds to a legal breakup.

(Note that this graph is acyclic.)

- The word breakup problem is an example where the graph is given only implicitly: To find out whether there is an edge between two points, you must make a dictionary lookup.
- We may want to minimize those lookups, but minimizing two different objective functions simultaneously (the number of division points and the number of lookups) is generally not possible.
  (For minimizing lookups, depth-first search seems better.)



In each step, the speed vector can change only by 1 in each direction. We have to start and arrive with speed 1, vertical direction. There is a graph in which this is a shortest path problem.

Vertices (point, speed vector) pairs (p, v).

Edges between (p_1, v_1) and (p_2, v_2): if $p_2 - p_1 = v_1$, $|v_2 - v_1|_\infty \leqslant 1$. Here $|(x, y)|_\infty = \max(|x|, |y|)$ is the so-called maximum norm.

Shortest paths with edge weights (lengths)

- Weight of edge $e = (u,v)$: $w(e) = w(u,v)$.
- Weight of path: the sum of the weights of its edges.
- Shortest path: lightest path.
- Distance $\delta(u,v)$ is the length of lightest path from u to v.

Variants of the problem:

- Single-pair (from source s to destination t).
- Single-source s: to all reachable points. Returns a tree of lightest paths, represented by the parent function $v \mapsto v.\pi$.
- All-pairs.

Negative weights? These are also interesting, but first we assume that all weights are nonnegative.

- Let $d(u)$ be the distance of point u from s.
- Follow the idea of breadth-first search. At any given time, for some value x, we will have already found the set S of all points u with $d(u) < x$ and possibly some with $d(u) = x$.
- Key observation: if v^* is next closest (to be added to S), then it is reachable by an edge from some $u \in S$ that is on a shortest path: $d(s) = d(u) + w(u, v^*)$.

How to find v^* efficiently?

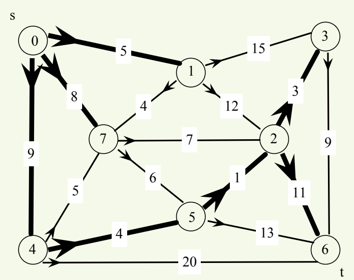
- Maintain the set Q of all points v not in S but reachable by an edge from S. Maintain on it a candidate distance

$$d'(v) = \min_{u \in S} d(u) + w(u,v).$$

  As we have seen, $d'(v^*) = d(v^*)$, but this is not necessarily so for $v \neq v^*$.

- In order to find $v^* = \arg\min_{v \in Q} d'(v)$ efficiently, set up a priority queue data structure on Q.

(I will use the book and Kevin Wayne's slides for the details.)



We show $S' = S \setminus \{0\}$ and Q as lists of $v(d)u$, where the (current) shortest path to v has length d and last link (u,v).

$S' = \{1(5)0\}$, $Q = \{2(17)1, 3(20)1, 4(9)0, 7(8)0\}$

$S' = \{1(5)0, \mathbf{7(8)0}\}$, $Q = \{\mathbf{2(15)7}, 3(20)1, 4(9)0, \mathbf{5(14)7}\}$

$S' = \{1(5)0, \mathbf{4(9)0}, 7(8)0\}$, $Q = \{2(15)7, 3(20)1, \mathbf{5(13)4}, \mathbf{6(29)4}\}$

$S' = \{1(5)0, 4(9)0, \mathbf{5(13)4}, 7(8)0\}$, $Q = \{\mathbf{2(14)5}, 3(20)1, \mathbf{6(26)5}\}$

$S' = \{1(5)0, \mathbf{2(14)5}, 4(9)0, 5(13)4, 7(8)0\}$, $Q = \{\mathbf{3(17)2}, \mathbf{6(25)2}\}$

$S' = \{1(5)0, 2(14)5, \mathbf{3(17)2}, 4(9)0, 5(13)4, 7(8)0\}$, $Q = \{6(25)2\}$

With respect to connectivity, another important algorithmic problem is to find the smallest number of edges that still leaves an undirected graph connected. More generally, edges have weights, and we want the lightest tree. (Or heaviest, since negative weights are also allowed.)

Generic algorithm: add a safe edge each time: an edge that does not form a cycle with earlier selected edges.

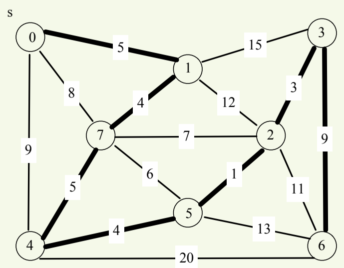
A cut of a graph G is any partition $V = S \cup T$, $S \cap T = \emptyset$. It respects edge set A if no edge of A crosses the cut. A light edge of a cut: a lightest edge across it.

Theorem If the edge set A is a subset of some lightest spanning tree, S a cut respecting A then after adding any light edge of S to A, the resulting A' still belongs to some lightest spanning tree.

Keep adding a light edge adjacent to the already constructed tree.
Implement this similarly to Dijkstra's algorithm: maintain a set Q points adjacent to the tree, orginize it as a priority queue.
The main difference to Dijkstra's algorithm is what is the key value $v.key$:

Prim: smallest edge length (so far) from the current tree T to v.

Dijkstra: smallest path length (so far) from the source s to v.



We show $S' = S \setminus \{0\}$ and Q as lists of $v(d)u$, where d is the length of the (current) smallest edge (u,v) from S.

$S' = \{\}$, $Q = \{1(5)0, 7(8)0, 4(9)0\}$

$S' = \{\mathbf{1(5)0}\}$, $Q = \{\mathbf{2(12)1}, \mathbf{3(15)1}, 4(9)0, \mathbf{7(4)1}\}$

$S' = \{1(5)0, \mathbf{7(4)1}\}$, $Q = \{\mathbf{2(7)7}, 3(15)1, \mathbf{4(5)7}, \mathbf{5(6)7}\}$

$S' = \{1(5)0, 7(4)1, \mathbf{4(5)7}\}$, $Q = \{2(7)7, 3(15)1, 5(6)7, \mathbf{5(4)4}, \mathbf{6(20)4}\}$

$S' = \{1(5)0, 7(4)1, 4(5)7, \mathbf{5(4)4}\}$, $Q = \{\mathbf{2(1)7}, 3(15)1, \mathbf{6(13)5}\}$

$S' = \{1(5)0, 7(4)1, 4(5)7, 5(4)4, \mathbf{2(1)7}\}$, $Q = \{\mathbf{3(3)2}, \mathbf{6(11)2}\}$

$S' = \{1(5)0, 7(4)1, 4(5)7, 5(4)4, 2(1)7, \mathbf{3(3)2}\}$, $Q = \{6(9)3\}$

Another minimum spanning tree algorithm based on the same theorem is this:

Kruskal's algorithm Keep increasing a forest (cycle-free graph), starting from the set of all points and no edges. Keep adding to the forest the shortest one among all edges of G that do not create a cycle.

After adding i points, this forest $F(i)$ has the following property:

> *There is some value $d(i)$ such that every pair of components is at "distance" $\geqslant d(i)$ from each other, (no shorter edge between them), and every chosen edge has length $\leqslant d(i)$.*

Kruskal's algorithm solves the following two important problems of clustering:

1. For a given δ, break up the points of G into the smallest number subsets with the property that they are at distance $>\delta$ from each other.
   Solution: grow the forest $F(i)$ to the largest i with $d(i)>\delta$.
2. For a given k, break up the points of G into k subsets with the property that the smallest distance between them is as large as possible.
   Solution: grow the forest $F(i)$ until it has k components. (What is i at this point?)

Question Given a graph with edge costs and an edge e, what is an algorithm to decide whether there is a minimum spanning tree containing e?

Answer Build a tree by Prim's algorithm but starting from e, and compare it with one built from scratch.

An efficient algorithm can frequently be obtained using the following idea:

1. Divide into subproblems of equal size.
2. Solve subproblems.
3. Combine results.

In order to handle subproblems, a more general procedure is often needed.

1. Subproblems: sorting $A[1 \,..\, n/2]$ and $A[n/2 + 1 \,..\, n]$
2. Sort these.
3. Merge the two sorted arrays of size $n/2$.

The more general procedures now are the ones that sort and merge arbitrary parts of an array.

```
MERGE(A,p,q,r)
Merges A[p . . q] and A[q + 1 . . r].
    n_1 ← q − p + 1; n_2 ← r − q
    create array L[1 . . n_1 + 1] and R[1 . . n_2 + 1]
    for i ← 1 to n_1 do L[i] ← A[p + i − 1]
    for j ← 1 to n_2 do R[j] ← A[q + j]
    L[n_1 + 1] ← ∞; R[n_2 + 1] ← ∞
    i ← 1, j ← 1
    for k ← p to r do
        if L[i] ⩽ R[j] then A[k] ← L[j]; i++
        else A[k] ← R[j]; j++
```

Why the ∞ business? These sentinel values allow to avoid an extra part for the case that L or R are exhausted. This is also why we used new arrays for the input, rather than the output.

```
MERGE-SORT(A, p, r)
Sorts A[p .. r].
   if p<r then
        q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q); MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

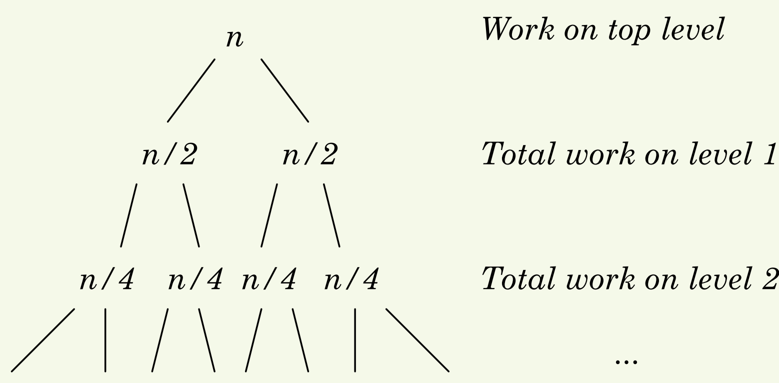
Analysis, for the worst-case running time $T(n)$:

$$T(n) \leqslant \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{otherwise.} \end{cases}$$

Resolving the recursive inequality

Assume that $n = 2^k$. When it is not, we just consider the smallest number $n' = 2^k > n$ (assuming that we sort a larger array).

$$\begin{aligned} T(n) &\leqslant c_2 n + 2T(n/2) && (1) \\ T(n/2) &\leqslant c_2 n/2 + 2T(n/4) && (2) \\ T(n) &\leqslant c_2 n + c_2 n + 4T(n/4) \quad \text{substituted (2) into (1)} \\ &\dots \\ T(n) &\leqslant k c_2 n + 2^k T(n/2^k) \leqslant n(c_1 + c_2 \log n) = O(n \log n). \end{aligned}$$

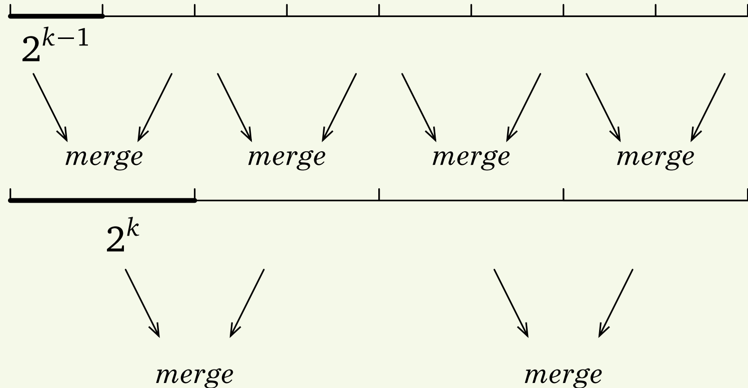


n
n/2
n/2
n/4
n/4
n/4
n/4
Work on top level
Total work on level 1
Total work on level 2
...

Assume $n = 2^k$:

$$
\begin{aligned}
&c_2 n(1 + 2 \cdot 1/2 + 4 \cdot 1/4 + \cdots + 2^k \cdot 2^{-k}) + c_1 n, \\
&= c_2 nk + c_1 n = n(c_2 \log n + c_1) = O(n \log n)
\end{aligned}
$$

Perform first the jobs at the bottom level, then those on the next level, and so on. In passes k and $k + 1$:



In the plane, for points p, q, let $d(p, q)$ denote the distance.

Question Given a set P of n points in the plane, find a pair $p, q \in P$ with $d(p, q)$ minimal.

A brute-force solution takes $O(n^2)$ steps. Can we do better? For simplicity, let $n = 2^k$.

- Let P_x be the list in which the points of P are sorted by the x coordinate. Let Q be the first $n/2$ points in P_x, and R the rest.
- Find the smallest distance δ_Q in Q and the smallest distance δ_R in R. Let $\delta = \min(\delta_Q, \delta_R)$.

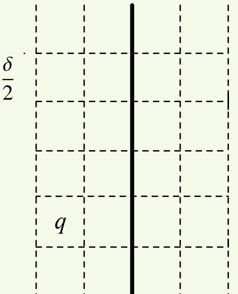
We found the closest pair unless it is some (q, r) with $q \in Q, r \in R$. Next we deal with this possibility, called the boundary case (q, r).

Let m = the maximal x coordinate of points in Q.
Let S = the set of points in P whose x coordinate differs from m by at most δ. If we have the boundary case (q, r) then both q and r are in S.
How to find them?
Let S_y be the list in which the set S is sorted by the y coordinate. The crucial observation:

Lemma q, r are within 15 positions of each other in S_y.

It follows that we can find all such q, r in linear time.

Proof of the lemma. The vertical line corresponds to x coordinate m.



Each little square contains at most one point of S, since two would be too close. Assume q has the smaller y coordinate, then r must be in one of the 16 little squares above. □

Running time analysis: First, we sort in time $O(n \log n)$ in both the x and the y directions, to get P_x and P_y. Since we do this at the beginning, sorting does not enter into the recursive calls. Let $F(n)$ be the running time of the algorithm after this sorting. We found

$$F(2) = 1,$$
$$F(n) \leqslant 2F(n/2) + cn.$$

So just as with merge sort, the total running time is $O(n \log n)$.

$$x = x_0 + 2x_1 + \cdots + 2^{n-1}x_{n-1} = (x_{n-1}x_{n-2}\ldots x_1x_0)_2,$$
$$x = X_0 + 2^{n/2}X_1, \quad y = Y_0 + 2^{n/2}Y_1,$$
$$xy = X_0Y_0 + 2^{n/2}(X_0Y_1 + X_1Y_0) + 2^nX_1Y_1.$$

Naive divide-and-conquer gives $T(n) \leqslant 4T(n/2) + cn$, leading to n^2. Trick: $(X_0 - X_1)(Y_0 - Y_1) = X_0Y_0 + X_1Y_1 - (X_0Y_1 + X_1Y_0)$. Both $X_0 - X_1$ and $Y_0 - Y_1$ are also maximum n-bit numbers. New algorithm:

RecMult(x, y)
    Find X_0, X_1, Y_0, Y_1
    $p \leftarrow$ RecMult(X_0, Y_0), $q \leftarrow$ RecMult(X_1, Y_1)
    $r \leftarrow$ RecMult$(X_0 - X_1, Y_0 - Y_1)$
    return $p + 2^{n/2}(p + q - r) + 2^nq$

Running time analysis, again assuming $n = 2^k$:

$$
\begin{aligned}
M(n) &\leqslant cn + 3M(n/2) && (3)\\
M(n/2) &\leqslant cn/2 + 3M(n/4) && (4)\\
M(n) &\leqslant cn + (3/2)cn + 9M(n/4) && \text{where we substituted (4) into (3)}\\
M(n) &\leqslant cn + (3/2)cn + (3/2)^2 cn + 27M(n/8),\\
&\quad \cdots\\
M(n) &\leqslant cn(1 + (3/2) + (3/2)^2 + \cdots + (3/2)^{k-1}) + 3^k M(n/2^k = 1).\\
&\quad 2^k(1 + (3/2) + (3/2)^2 + \cdots + (3/2)^k)\\
&\quad < 3^k(1 + 2/3 + (2/3)^2 + \cdots) = 3^k \cdot 3,\\
M(n) &\leqslant c \cdot 3^k \cdot 3 + 3^k = (3c+1)3^k,\\
3^k &= 2^{k \log 3} = n^{\log 3}.
\end{aligned}
$$

1. Find the maximum of a unimodal function.
2. Find the maximum difference between any two elements a_i, a_j of a sequence $a_1, \ldots, a_n$.

Interval scheduling problem, with additional complication: each task $i = 1, \ldots, n$ with start and finish times $s_i < f_i$ has some value v_i.
Instead of maximizing the number of scheduled tasks, we want to maximize the total value.

Example Let $T_i = s_i(v_i)f_i$ denote a task with starting time s, endtime f and value v. Consider the following 6 tasks:

$$T_1 = 0(2)3, T_2 = 1(4)5, T_3 = 4(4)6, T_4 = 2(7)9, T_5 = 7(2)10, T_6 = 8(1)11.$$

Assume that the tasks are ordered by finish time: $f_1 \leqslant \cdots \leqslant f_n$. For task i,

$$p(i) = \max\{\, j : f_j \leqslant s_i \,\},$$

that is the last task before i disjoint from it.
Let $\mathrm{OPT}(i)$ be the value of the sequence of tasks $1, 2, \ldots, i$. Dynamic programming tries to compute it recursively:

$$\mathrm{OPT}(i) = \max(\mathrm{OPT}(i-1), \mathrm{OPT}(p(i)) + v_i).$$

On the example:

i	1	2	3	4	5	6
T_i	0(2)3	1(4)5	4(4)6	2(7)9	7(2)10	8(1)11
$p(i)$	0	0	1	0	3	3
$\mathrm{OPT}[i]$	2	4	6	7	8	8

Just calling the recursive formula is unwise, it leads to an exponential blowup. Let us save the values $\mathrm{OPT}(1), \mathrm{OPT}(2), \dots, \mathrm{OPT}(n)$ after we computed them, in some array M. Then instead of the recursive calls, we just refer to this array.

```
NRWS(n)
    for i = 0 to n do M[n] ← 0
    for i = 1 to n do
        M[i] ← max(M(i − 1), v_i + M[p(i)])
```

The more general idea is memoization, or caching.

- Applies to a recursive procedure where the total number of subproblems is not too great.
- The results of all calls to subproblems must be stored in a table (in programming, a “static”, or “global” array).
- First check whether the result is already in the table: if yes, just return it, else compute it and store it.

```
RWS(i)
    if M[i] is not defined then
        M[i] ← max(RWS(i − 1), v_i + RWS(p(i)))
    return M[i]
```

How to find which tasks are selected? Tracing backwards, and collecting tasks in a set S:

```
    i ← n; S = ∅
    while i>0 do
        if M[i] = M[i − 1] then i−−
        else  S ← S ∪ {i}; i ← p(i)
```

So we add a task i (starting from the end) if $M[i] > M[i-1]$, but after that we skip over all the tasks between $p(i)$ and i, and continue from $p(i)$.

- Fibonacci numbers.
- Binomial coefficients.

(In both cases, the algorithm is not as bad as the naive recursion, but by far not as good as computing the known formulas.)

Given: volumes $b \geqslant a_1, \ldots, a_n > 0$, and integer values $v_1 \geqslant \cdots \geqslant v_n > 0$.

$$\begin{aligned} \text{maximize} \quad & v_1 x_1 + \cdots + v_n x_n \\ \text{subject to} \quad & a_1 x_1 + \cdots + a_n x_n \leqslant \quad b, \\ & x_i = 0, 1, \; i = 1, \ldots, n. \end{aligned}$$

In other words, find a subset $i_1 < \cdots < i_k$ of the set of items $1, \ldots, n$ (by choosing which $x_i = 1$) such that

- the sum of their volumes $a_{i_1} + \cdots + a_{i_k}$ is less than the volume b of our knapsack,
- the sum of their values $v_{i_1} + \cdots + v_{i_k}$ is maximal.

Subset sum problem find $i_1, \dots, i_k$ with $a_{i_1} + \cdots + a_{i_k} = b$. Obtained by setting $v_i = a_i$. Now if there is a solution with value b, we are done.

Partition problem Given numbers $a_1, \dots, a_n$, find $i_1, \dots, i_k$ such that $a_{i_1} + \cdots + a_{i_k}$ is as close as possible to $(a_1 + \cdots + a_n)/2$.

- We compute for each integer p the *smallest* volume that gives value $\geqslant p$. Upper bound on the total value: $V = v_1 + \cdots + v_n$.
- The subproblem in which only the first k items can be used:

$$m_k(p) = \min\{\, a_1 x_1 + \cdots + a_k x_k \leqslant b : v_1 x_1 + \cdots + v_k x_k \geqslant p \,\}.$$

If the set is empty the minimum is ∞.
The optimum is $\max\{\, p : m_n(p) \leqslant b \,\}$.

Array $m[k,p] = m_k(p)$. Notation $|x|^+ = \max(x, 0)$.

```
for k = 0 to n do m[k,0] ← 0
for p = 0 to V do
    if p>0 then m[0,p] = ∞
    for k = 1 to n do
        m[k,p] ← min(m[k − 1,p], a_k + m[k − 1,|p − v_k|^+])
```

Another application: money changer problem. Produce the sum b using smallest number of coins of denominations $a_1, \ldots, a_n$ (at most one of each). Corresponds to a case of the knapsack problem in which the volumes are all 1, and the values are a_i.

Let $\{a_1, \ldots, a_5\} = \{2, 3, 5, 7, 11\}$, $b = 13$, $v_i = a_i - 1$.
Array $m[k, p]$.

$k \backslash p$	1	2	3	4	5	6	7	8	9	10	11	12	...
1	2												
2	2	3	5										
3	2	3	5	5	7	8	10						
4	2	3	5	5	7	7	9	10	12	12	14		
5	2	3	5	5	7	7	9	10	12	11	13	14	

Complexity: $O(nV)$ steps, (counting additions as single steps).

Is this algorithm polynomial?

(Assume for simplicity that a_i and b are also integers.)

- A time complexity bound on an algorithm is generally given as a function of the length of input (measured in bits).
- Each number v_i written in binary has length $\lceil \log v_i \rceil$; a number of length m has size $>2^{m-1}$.
- The length of the input here is—essentially—the sum of the lengths of the numbers: $a_1, a_2, \ldots, a_n$, b, $v_1, \ldots, v_n$:

  $$L \leqslant \sum_i \log a_i + \log b + \sum_i \log v_i + 2n + 1.$$

  Let m be the maximum of all $\log v_i$, assume $\log a_i \leqslant m$, then $L \leqslant (m+1)(2n+1)$.
- The size of the table to compute is $Vn > 2^m n$. Simple special case $m = n$: then table size is $Vn \geqslant 2^m m$: this is exponential in the size L of the input.

So our algorithm is very expensive when the numbers v_i involved are large.

- An algorithm that is polynomial as a function of the size of the numbers in the input (as opposed to their representation length) is called pseudo-polynomial. The dynamic programming algorithm for the knapsack problem is such an algorithm.
- No polynomial algorithm is known for knapsack, not even for the subset sum problem: we will see that these problems are really hard (NP-complete).
- On the other hand, the dynamic programming algorithm can be used to achieve an approximation for the optimum to within a factor of $1 + \varepsilon$, in time polynomial in (the input size and) $1/\varepsilon$.

A problem with many applications: for example the `diff` program, biology, voice recognition.

Consider sequences of symbols from some alphabet S. Consider two such sequences, $X = (x_1, \ldots, x_m)$, and $Y = (y_1, \ldots, y_n)$. An alignment is given by two sets of indices

$$1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant m \text{ and } 1 \leqslant j_1 < j_2 < \cdots < j_k \leqslant n$$

such that x_{i_p} is matched to y_{j_p}, for all $p = 1, \ldots, k$.

There is a distance $d(r, s) \geqslant 0$ between symbols $r, s \in S$, which is also the penalty for matching them with each other. There is also a penalty $\delta \geqslant 0$ for every unmatched symbol, so the total penalty is

$$((m-k)+(n-k))\delta + \sum_{p=1}^{k} d(x_{i_p}, y_{j_p}).$$

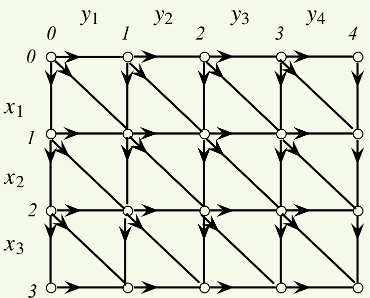
Example: match the words "kolor" and "colour", where $\delta = 1$, and $d(r, s) = 2$ if $r = s$ and 0 otherwise. We could match the bold characters: **kolor** with **colo**u**r**. Then the penalty is 1 + 2, since there is one unmatched symbol "u" and one mismatched pair (k,c).

Dynamic programming algorithm

Let $A[i,j]$ be the minimum penalty for matching $x_1 \dots x_i$ with $y_1 \dots y_j$. Recursion:

$$A[i,j] = \min(d(x_i, y_j) + A[i-1, j-1]), \delta + A[i, j-1], \delta + A[i-1, j]).$$

Of course, $A[0,0] = 0$, $A[0,j] = \delta$, $A[i,0] = \delta$ for $i,j > 0$. This allows to fill in the array $A[i,j]$ for example row-by-row.



The horizontal and vertical edges have length δ. The diagonal edge in row i and column j has length $d(x_i, y_j)$. Now $A[m, n]$ is the length of the shortest path from the left top to the right bottom. (Our algorithm is not slower than Dijkstra's for computing it.) To find the alignment, we just need to find the path.

Linear space via divide and conquer

- Computing $A[\cdot,\cdot]$ can be done in linear space by an algorithm $\textsc{AlignmentCosts}(X, Y)$, since we need only two neighboring rows at a time. But we don't only need the number $A[m, n]$, we also want to know the optimal alignment!
- New idea: first just find the crossing point of the optimal path in the middle column.
  $f(i, j)$ = the length of shortest path from $(0, 0)$ to (i, j).
  $g(i, j)$ = the length of shortest path from (i, j) to (m, n).
  Observation: for any k,

  $$A[m, n] = \min_q f(q, k) + g(q, k).$$

  The point (q, k) where the minimum is achieved belongs to a shortest path.

Divide-and-Conquer alignment algorithm.

```
DC-ALIGN(X, Y), returns a shortest path P
    if m ⩽ 2 or n ⩽ 2 then compute directly from A(X, Y)
    f[1 : m] ← ALIGNMENTCOSTS(X[1 : m], Y[1 : n/2])
    g[1 : m] ← ALIGNMENTCOSTS(X[1 : m], Y[n/2 + 1 : n])
    q ← the index minimizing f[q] + g[q]
    P1 ← DC-ALIGN(X[1 : q], Y[1 : n/2])
    P2 ← DC-ALIGN(X[q + 1 : n], Y[n/2 + 1 : n])
    return P1 + (q, n/2) + P2
```

The space requirement is clearly linear, since each recursive call can reuse the space of the earlier ones. (Exercise: prove it rigorously.)

The time analysis is more complex. We have (assuming n is a power of 2 for transparency)

$$T(m,2) \leqslant cm = (c/2)\cdot 2m,$$
$$T(m,n) \leqslant cmn + T(q,n/2) + T(m-q,n/2).$$

This is a little strange, since we don't know q.
Still, we guess that the total time is still $\leqslant dmn$ for some constant d.
Let us try to prove this, and see how large must d be for the proof to work.
The base case works if $d \geqslant c/2$.
By the above inequality and the inductive assumption

$$T(mn) \leqslant cmn + dqn/2 + d(m-q)n/2 = (c + d/2)mn.$$

So if $d \geqslant 2c$ then also $T(m,n) \leqslant dmn$ by the induction step.

Shortest paths with possible negative costs

We have seen a shortest (smallest-cost) path algorithm already (Dijkstra), but it could not handle negative costs. These can be important: an example is the arbitrage problem.

Negative cycle: If there is none, then we cannot decrease the cost a path by adding a loop to it: so, it is sufficient to look for paths of length $\leqslant n - 1$.

Wanted: an algorithm that in a graph $G = (V, E)$ with a cost function c_{uv} and no negative cycle, for any start and end nodes s, t, provides a smallest-cost path.

Idea: Compute for all v, i the length

$$M[i,v]$$

of the smallest-cost path from v to t using at most i edges.

- If $i = 0$ then $M[i,t] = 0$ and $M[v,t] = \infty$ for $v \neq t$.
- If $i>0$ then the following recursive relation holds:

$$M[i,u] = \min(M[i-1,u], \min_{(u,v)\in E}(c_{uv} + M[i-1,v])),$$

  allowing to fill the an array for $M[i,v]$.
- The running time is $O(nm)$ where m is the number of edges. Indeed, each edge is touched at most n times.

We can use the space of $M[i-1, v]$ to store $M[i, v]$, and call it simply $M[v]$. The number i is still used, but just as a counter of repetition. The value first$[u]$ stores the first node of the current smallest-cost path.

```
M[t] ← 0
for v ≠ t do M[v] ← ∞
for i = 1 to n do
    s ← 0                    // Any change in this iteration?
    for u ∈ V do
        for v : (u,v) ∈ E do
            if  c_uv + M[v]<M[u] then
                M[u] ← c_uv + M[v]
                first[u] ← v
                s ← 1
    if s = 0 then break
```

- This will stop at some $i<n$ if $M[v]$ does not change for any v.
- Following the $(u, \text{first}[u])$ edges we get a smallest-cost path to t.
- If we get into a cycle then it is a negative cycle.

Example (Workers and jobs) Suppose that we have n workers and n jobs. Each worker is capable of performing some of the jobs. Is it possible to assign each worker to a different job, so that workers get jobs they can perform?

It depends. If each worker is familiar only with the same one job (say, digging), then no.

Example At a dance party, with 300 students, every boy knows 50 girls and every girl knows 50 boys. Can they all dance simultaneously so that only pairs who know each other dance with each other?

- Bipartite graph: left set A (of girls), right set B (of boys).
- Matching, perfect matching.

Theorem If every node of a bipartite graph has the same degree $d \geqslant 1$ then it contains a perfect matching.

Examples showing the (local) necessity of the conditions:

- Bipartiteness is necessary, even if all degrees are the same.
- Bipartiteness and positive degrees is insufficient.

Example 6 tribes partition an island into hunting territories of 100 square miles each. 6 species of tortoise, with disjoint habitats of 100 square miles each.
Can each tribe pick a tortoise living on its territory, with different tribes choosing different totems?

For $S \subseteq A$ let

$$\mathbf{N}(S) \subseteq B$$

be the set of all neighbors of the nodes of A.
Special property: *For every $S \subseteq A$ we have $|\mathbf{N}(S)| \geqslant |S|$.*
Indeed, the combined hunting area of any k tribes intersects with at least k tortoise habitats.

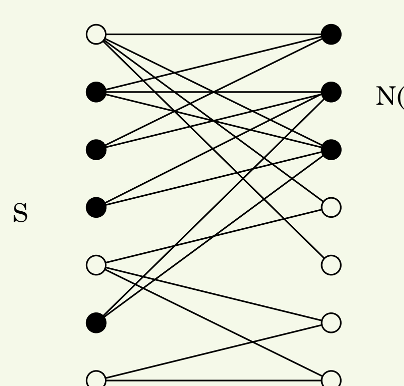
The above property happens to be the criterion also in the general case:

Theorem (The Marriage Theorem)
A bipartite graph has a perfect matching if and only if $|A| = |B|$ and for every $S \subseteq A$ we have $|\mathbf{N}(S)| \geqslant |S|$.

The condition is necessary.

Proposition The condition implies the same condition for all $S \subseteq B$.

Prove this as an exercise.



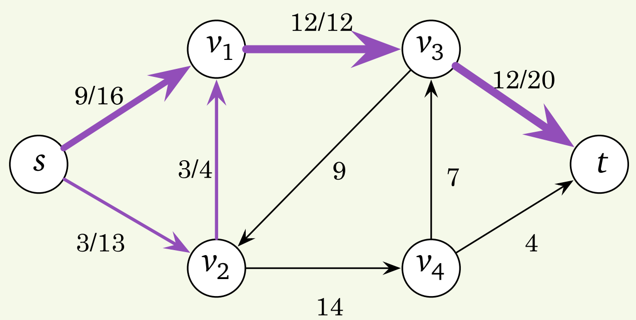
- Directed graph. Source s, sink t. Every vertex is on some path from s to t.
- Flow: function $f(u,v)$ on all edges (u,v) showing the amount of material going from u to v. We are only interested in the net flow $\hat{f}(u,v) = f(u,v) - f(v,u)$: then $\hat{f}(v,u) = -\hat{f}(u,v)$. So we simply require

$$f(v,u) = -f(u,v).$$

- The total flow entering a non-end node equals the total flow leaving it:

$$\text{for all } u \in V \setminus \{s,t\} \ \sum_v f(u,v) = 0.$$

- Each edge (u,v) imposes a capacity $c(u,v) \geqslant 0$ on the flow: $f(u,v) \leqslant c(u,v)$. (We may have $c(u,v) \neq c(v,u)$.)

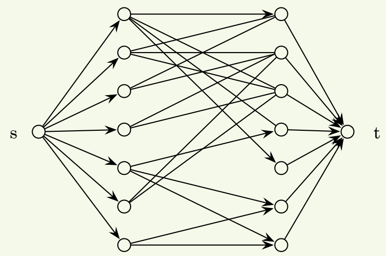


The notation f/c means flow f along an edge with capacity c.

- Our goal is to maximize the value of the flow f, that is

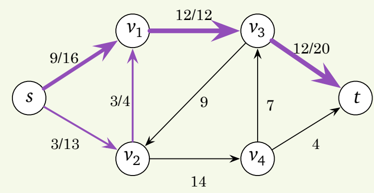
$$|f| = \sum_{v} f(s,v) = \sum_{v} f(v,t).$$

Application to matching

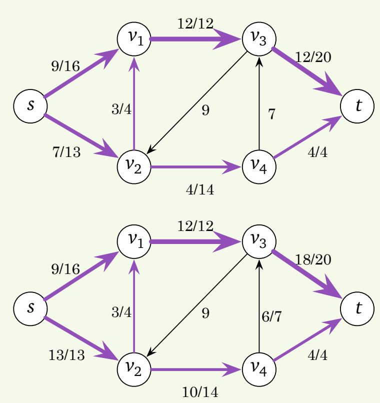


- n points on left, n on right. Edges directed to right, with unit capacity.
- Perfect matching → flow of value n.
- Flow of value n → perfect matching? Not always, but fortunately (as will be seen), there is always an integer maximum flow.

Idea for increasing the flow: do it along some path.

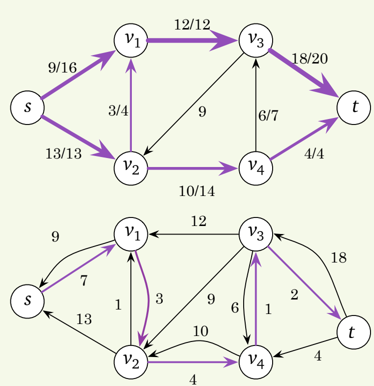


Increment it along the path s-v_2-v_4-t by 4.
Then increment along the path s-v_2-v_4-v_3-t by 6. Now we are stuck: no more direct way to augment.



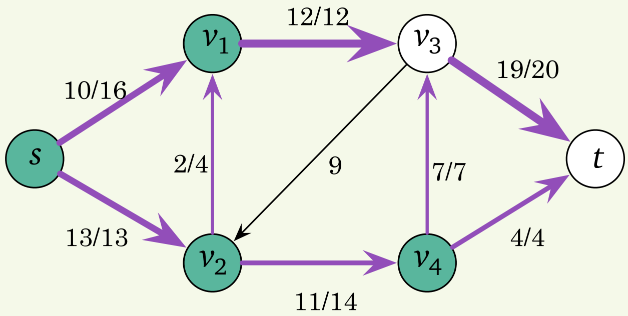
New idea: Increase (by 1) on (s, v_1), decrease on (v_1, v_2), increase on (v_2, v_4, v_3, t).

Generalization: Given a flow f, residual capacity $c_f(u,v) = c(u,v) - f(u,v)$. Makes sense even with negative $f(u,v)$. The residual network may have edges (with positive capacity) that were not in the original network. An augmenting path is an s-t path in the residual network (with some flow along it). (How does it change the original flow?)



Residual capacity of the path: the minimum of the residual capacities of its edges (in the example, 1).

We obtained:



This cannot be improved: look at the cut (S, T) with $T = \{v_3, t\}$.

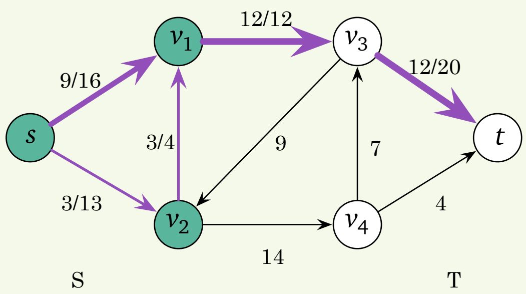
Keep increasing the flow along augmenting paths.

Questions

1. If it terminates, did we reach maximum flow?
2. Can we make it terminate?
3. How many augmentations may be needed? Is this a polynomial time algorithm?

Neither of these questions is trivial. The technique of cuts takes closer to the solution.

Cut (S,T) is a partition of V with $s \in S$, $t \in T$.
Net flow $f(S,T) = \sum_{u \in S, v \in T} f(u,v)$.
Capacity $c(S,T) = \sum_{u \in S, v \in T} c(u,v)$. Obviously, $f(S,T) \leqslant c(S,T)$.



In this example, $c(S,T) = 26$, $f(S,T) = 12$.

Lemma $f(S,T) = |f|$, the value of the flow.

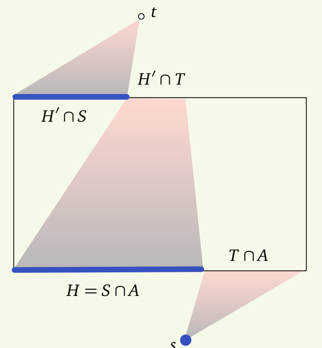
Corollary The value of any flow is bounded by the capacity of any cut.

Theorem (Max-flow, min-cut) The following properties of a flow f are equivalent.

1. $|f| = c(S, T)$ for some cut (S, T).
2. f is a maximum flow.
3. There are no augmenting paths to f.

The equivalence of the first two statements says that the size of the maximum flow is equal to the size of the minimum cut.
Proof: 1⇒2 and 2⇒3 are obvious. The crucial step is 3⇒1.
Given f with no augmenting paths, we construct (S, T): let S be the nodes reachable from s in the residual network G_f.

Using Max-Flow Min-Cut. Assume there is no perfect matching in the bipartite graph $G = (A \cup B, E)$, with $|A| = |B| = n$. We find a bottleneck $H \subseteq A$ with $|\mathbf{N}(H)| < |H|$.
Flow network over $A \cup B \cup \{s, t\}$ as before. Since there is no perfect matching, the maximum flow has size $< n$. So there is a cut (S, T), $s \in S$, $t \in T$, with $c(S, T) < n$.



Let $H = S \cap A$, $H' = \mathbf{N}(H)$.

$$
\begin{aligned}
n > c(S, T)& \\
&= c(\{s\}, T) + c(S \cap A, T \cap B) + c(S, \{t\}) \\
&\geqslant (n - |H|) + |H' \cap T| + |H' \cap S| \\
&= n - |H| + |H'|,
\end{aligned}
$$

$|H| > |H'|$.

- Does the Ford-Fulkerson algorithm terminate? Not necessarily (if capacities are not integers), unless we choose the augmenting paths carefully.
- Integer capacities: always terminates, but may take exponentially long.

  Network derived from the bipartite matching problem: each capacity is 1, so we terminate in polynomial time.
- Dinic-Edmonds-Karp: use breadth-first search for the augmenting paths. Why should this terminate?

Lemma In the Edmonds-Karp algorithm, the shortest-path distance $\delta_f(s,v)$ increases monotonically with each augmentation.

Proof: Let $\delta_f(s,u)$ be the distance of u from s in G_f, and let f' be the augmented flow. Assume, by contradiction $\delta_{f'}(s,v) < \delta_f(s,v)$ for some v: let v be the one among these with smallest $\delta_{f'}(s,v)$. Let $u \to v$ be be a shortest path edge in $G_{f'}$, and

$$d := \delta_f(s,u)(= \delta_{f'}(s,u)), \text{ then } \delta_{f'}(s,v) = d + 1.$$

Edge (u,v) is new in $G_{f'}$; so (v,u) was a shortest path edge in G_f, giving $\delta_f(s,v) = d - 1$. But $\delta_{f'}(s,v) = d + 1$ contradicts $\delta_{f'}(s,v) < \delta_f(s,v)$.

An edge is said to be critical, when it has just been filled to capacity.

Lemma Between every two times that an edge (u,v) is critical, $\delta_f(s,u)$ increases by at least 2.

Proof: When it is critical, $\delta_f(s,v) = \delta_f(s,u) + 1$. Then it disappears until some flow f'. When it reappears, then (v,u) is critical, so

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \geqslant \delta_f(s,v) + 1 = \delta_f(s,u) + 2.$$

Corollary We have a polynomial algorithm.

Proof: Just bound the number of possible augmentations, noticing that each augmentation makes some edge critical.

Let $n = |V|$, $m = |E|$. Each edge becomes critical at most $n/2$ times. Therefore there are at most $m \cdot n/2$ augmentations. Each augmentation may take $O(m)$ steps: total bound is

$$O(m^2 n).$$

There are better algorithms: Goldberg's push-relabel algorithm, also given in your book, achieves $O(n^3)$.

Network flow theory has many applications. Sometimes it takes ingenuity to see how to apply it, as the following example shows. Suppose that as a large company, we have a number of projects to choose from: their set is denoted by $P = \{1, 2, \dots, n\}$. Project i brings profit p_i. This may be positive or negative (then we see it as a cost). The projects have some dependencies, represented by an acyclic directed graph $G = (P, E)$. Edge (i, j) means that if we perform project i we have to perform project j, too. (This is not a time ordering: say, if 4 is a wedding shower in which we will collect $p_4 = 1000$ dollars, then for example we may have to perform 5, buying food beforehand for $-p_5 = 200$ dollars, and 8, cleaning up afterwards, for $-p_8 = 250$ dollars.)

Goal: A subset $H \subseteq P$ is feasible if with every project in it, it contains all others on which it depends: $u \in H$ and $(u,v) \in E$ implies $v \in H$. The total profit of a feasible set S is $p(S) = \sum_{i \in S} p_i$: we want to find a feasible set with the maximum possible profit. An obvious upper bound on the profit we can achieve is

$$C = \sum_{i : p_i > 0} p_i.$$

Idea: Introduce a flow network, reduce our problem to its minimum cut problem. New nodes: source and sink s, t. Capacities:

1. All edges $(i, j) \in E$ have capacity $c(i, j) = \infty$.
2. Edges (s, i) with $p_i > 0$ have capacity $c(s, i) = p_i$.
3. Edges (j, t) with $p_j < 0$ have capacity $c(j, t) = -p_j$.

Edges of type 1 imply that a cut S, T of the network has finite capacity if and only if $S' = S \setminus \{s\}$ is a feasible set (no infinite-capacity edge should go from S to T).

The capacity of a finite-capacity cut (S, T) is

$$c(S,T) = \sum_{i \in T : p_i > 0} p_i - \sum_{j \in S : p_j < 0} p_j. \tag{5}$$

We claim

$$c(S,T) = C - p(S'),$$

that is $c(S,T)$ is by how much the profit of the set S' is less than the upper bound. Indeed: the first sum of (5) is the amount of profits we lose, and the second sum is the amount of the costs we incur.
So, minimizing $c(S,T)$ maximizes $p(S')$. And minimizing $c(S,T)$ means finding a minimum cut, which can be done by the network flow algorithm we learned.

Randomized algorithms

- A randomized algorithm uses some source of randomness as its input, in addition to the input data. Surprisingly, this frequently helps.
- There are cases when no deterministic algorithm can solve a certain problem, but more frequently, bringing in randomness results in a more efficient algorithm.
- We will have to learn (or recall) some facts from basic probability theory along the way.

- Assume that n processes $P_1, \ldots, P_n$ must access a database. This happens in rounds. In each round, only one access is possible: if more than one process makes an attempt, they all fail. There is no communication between them, and no coordinator to help them.
- Idea: each process makes an attempt an in each round, with some probability $0<p<1$. (say $p = 0.1$). We want to estimate the time it takes for all processes to succeed with high probability.

- When we randomize, certain events acquire probabilities. The probability of event $\mathcal{A}$ is denoted generally by $\Pr(\mathcal{A})$.
- For example, let event $\mathcal{A}(i, t)$ happen if process P_i makes an attempt at time t. By definition, $\Pr(\mathcal{A}(i, t)) = p$.
- For an event $\mathcal{A}$, let $\neg A$ be the event that $\mathcal{A}$ does not happen. Then $\Pr(\neg\mathcal{A}) = 1 - \Pr(\mathcal{A})$. For example, the probability that process P_i does not make an attempt at time t is $1 - p$.
- For events $\mathcal{A}, \mathcal{B}$, let $\mathcal{A} \cup \mathcal{B}$ be the event that at least one of these happens. Knowing $\Pr(\mathcal{A})$ and $\Pr(\mathcal{B})$ does not generally suffice to know $\Pr(\mathcal{A} \cup \mathcal{B})$, but at least we know the union bound

$$\Pr(\mathcal{A} \cup \mathcal{B}) \leqslant \Pr(\mathcal{A}) + \Pr(\mathcal{B}).$$

This becomes and equality for mutually exclusive events, $\mathcal{A}, \mathcal{B}$, that is if $\mathcal{A} \cap \mathcal{B} = \emptyset$.

- For two events $\mathcal{A}, \mathcal{B}$ we write $\mathcal{A} \cap \mathcal{B}$ for the event that occurs if both $\mathcal{A}$ and $\mathcal{B}$ occur. If $\Pr(\mathcal{A})>0$ then we denote by

$$\Pr(\mathcal{B}|\mathcal{A}) = \frac{\Pr(\mathcal{A} \cap \mathcal{B})}{\Pr(\mathcal{A})}$$

  the conditional probability that $\mathcal{B}$ occurs provided that $\mathcal{A}$ occurs. For example, the conditional probability that the six-sided die comes up with an even number of points provided it shows more than 1, is 3/5.
- We say that $\mathcal{B}$ is independent of $\mathcal{A}$ if $\Pr(\mathcal{B}|\mathcal{A}) = \Pr(\mathcal{B})$, that is if

$$\Pr(\mathcal{A} \cap \mathcal{B}) = \Pr(\mathcal{A}) \cdot \Pr(\mathcal{B}).$$

  In general, knowing the probabilities of $\mathcal{A}$ and $\mathcal{B}$ does not allow yet finding the probability of $\mathcal{A} \cap \mathcal{B}$. But in this case it does.
  If $\mathcal{A}$ is independent of $\mathcal{B}$ then for example also $\neg\mathcal{A}$ is independent of $\mathcal{B}$: the answer of any question about $\mathcal{A}$ is independent on the answer of any question about $\mathcal{B}$.

- We say that event $\mathcal{C}$ is independent of events $\mathcal{A}, \mathcal{B}$ if its conditional probability is the same no matter what we assume about $\mathcal{A}, \mathcal{B}$. So

$$\Pr(\mathcal{C}) = \Pr(\mathcal{C}|\mathcal{A} \cap \mathcal{B}) = \Pr(\mathcal{C}|\mathcal{A} \cap (\neg\mathcal{B})) = \Pr(\mathcal{C}|\mathcal{B}) = \cdots$$

  We say that the set of events $\mathcal{A}, \mathcal{B}, \mathcal{C}$ is independent if each of them is independent of the rest.
- This is equivalent to saying that no matter what we ask about $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, the probability of the combined event is the product of the probabilities of its constituents.
  For example, we assume that each process attempts at time t independently with probability p. Then the probability that process 1 attempts and processes 2,3 don't is

$$\Pr(\mathcal{A}(1,t) \cap \neg\mathcal{A}(2,t) \cap \neg\mathcal{A}(3,t)) = p(1-p)(1-p).$$

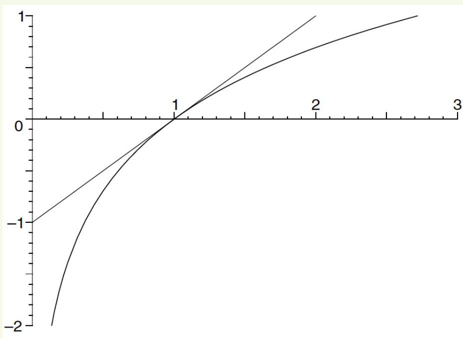
What is the probability of the event $\mathcal{S}(i,t)$ that at time t process P_i attempts and the other processes don't (so P_i succeeds)?

$$\Pr(\mathcal{S}(i,t)) = \Pr(\mathcal{A}(i,t) \cap \bigcap_{j \neq i} \mathcal{A}(j,t)) = p(1-p)^{n-1}.$$

The best strategy is to choose the value p that maximizes this. Calculus shows to choose $p = 1/n$, and then we get

$$\Pr(S(i,t)) = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}.$$

The important inequality $\ln(1 + x) \leqslant x$ comes from the concavity of the logarithm function:



This same inequality can be used to get a bound from the other side:

$$
\begin{aligned}
-\ln(1+x) &= \ln\frac{1}{1+x} = \ln\left(1-\frac{x}{1+x}\right) \leqslant -\frac{x}{1+x}, \\
\ln(1+x) &\geqslant \frac{x}{1+x}.
\end{aligned}
$$

Combining the two estimates:

$$\frac{x}{1+x} \leqslant \ln(1+x) \leqslant x.$$

Writing $x = 1/n$ gives

$$\frac{1}{n+1} \leqslant \ln\left(1+\frac{1}{n}\right) \leqslant \frac{1}{n},$$

while with $x = -1/n$ we get

$$-\frac{1}{n-1} \leqslant \ln\left(1-\frac{1}{n}\right) \leqslant -\frac{1}{n}.$$

Using the log inequality above,

$$-\frac{1}{n-1} \leqslant \ln(1-1/n) \leqslant -1/n. \tag{6}$$

Hence

$$\begin{aligned}
-1 &\leqslant \ln(1-1/n)^{n-1}, \\
e^{-1} &\leqslant (1-1/n)^{n-1}, \\
\frac{1}{en} &\leqslant \frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1} = \Pr(\mathcal{S}(i,t)).
\end{aligned}$$

Let $\mathcal{F}(i,t)$ be the event that process P_i does not succeed in any of the rounds $1,\ldots,t$: $\mathcal{F}(i,t) = \bigcap_{r=1}^{t} \neg\mathcal{S}(i,t)$. The events at different times are also independent of each other:

$$\Pr(\mathcal{F}(i,t)) = \prod_{r=1}^{t}(1 - \Pr(\mathcal{S}(i,r))) \leqslant \left(1 - \frac{1}{en}\right)^{t}.$$

Taking $t \geqslant k \cdot en$ and using the second inequality of (6):

$$\Pr(\mathcal{F}(i,t)) \leqslant \left(1 - \frac{1}{en}\right)^{t} \leqslant \left(\left(1 - \frac{1}{en}\right)^{en}\right)^{k} \leqslant e^{-k}.$$

Let $\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}(i,t)$ be the event that some process does not succeed by time $t \geqslant k \cdot en$. By the union bound:

$$\Pr(\mathcal{F}_t) \leqslant n \cdot e^{-k}.$$

Choosing for example $k = 2\ln n$, we get the probability bound $n \cdot n^{-2} = 1/n$. So if not every process succeeded within $2n\ln n$ steps, then some rare disaster of probability $<1/n$ happened.

In a probability space, a random variable is some quantity X such that events of the kind $a \leqslant X < b$ have probabilities assigned to them.

Example We toss a 6-headed die twice. Let X_i be the number of points coming up in the ith toss. Then $\Pr\{X_i = j\} = 1/6$ for $j = 1, \dots, 6$. Let $Y = X_1 + X_2$. Then

i	2	3	4	5	6	7	8	9	10	11	12
$\Pr\{Y = i\}$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

If the outcome of some experiment is a number X that can have values $x_1, x_2, \ldots$ with probabilities $p_1, p_2, \ldots$ respectively, then the expected value of X is defined as $\mathbb{E}X = p_1 x_1 + p_2 x_2 + \ldots$.

Examples

- If Z is a random variable whose values are the possible outcomes of a toss of a 6-sided die, then

$$\mathbb{E}Z = (1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5.$$

- If Y is the random variable that is 1 if $Z \geqslant 5$, and 0 otherwise, then

$$\mathbb{E}Y = 1 \cdot \Pr\{ Z \geqslant 5 \} + 0 \cdot \Pr\{ Z < 5 \} = \Pr\{ Z \geqslant 5 \}.$$

Theorem For random variables X, Y (on the same sample space):

$$\mathbb{E}(X + Y) = \mathbb{E}X + \mathbb{E}Y.$$

Example For the number X of spots on top after a toss of a die, let A be the event $2|X$, and B the event $X>1$. Dad gives me a dime if A occurs and Mom gives one if B occurs. What is my expected win? Let I_A be the random variable that is 1 if A occurs and 0 otherwise.

$$\mathbb{E}(I_A + I_B) = \mathbb{E}I_A + \mathbb{E}I_B = \Pr(A) + \Pr(B) = 1/2 + 5/6 \text{ dimes}.$$

Average-case analysis of Quicksort

Please, review the definition of deterministic and randomized Quicksort from your textbook. What is randomized is the choice of the pivot elements.
The meaning of "average" is different for the two cases.

- In the deterministic case the running time on the worst possible input order is $\Omega(n^2)$. Averaging is over all possible input orders: that average is $O(n \log n)$.
- In the randomized case, the expected running time means averaging over all possible choices of the pivots. We do not average over inputs, this works for every possible input. The average obtained is $O(n \log n)$. The worst case (taking the worst possible pivot sequence) is still $\Omega(n^2)$.

Let the sorted order be $z_1 < z_2 < \cdots < z_n$. If $i < j$ then let

$$Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}.$$

Let the random variable C_{ij} be defined to be 1 if z_i and z_j will be compared sometime during the sort, and 0 otherwise.
Every comparison happens during some partition, with the pivot element. Let π_{ij} be the first (random) pivot element entering Z_{ij}. A little thinking shows:

Lemma We have $C_{ij} = 1$ if and only if $\pi_{ij} \in \{z_i, z_j\}$. Also, for every $x \in Z_{ij}$, we have

$$\Pr\{\pi_{ij} = x\} = \frac{1}{j - i + 1}.$$

It follows that $\Pr\left\{ C_{ij} = 1 \right\} = \mathbb{E} C_{ij} = \frac{2}{j-i+1}$. The expected number of comparisons is

$$\sum_{1 \leqslant i < j \leqslant n} \mathbb{E} C_{ij} = \sum_{1 \leqslant i < j \leqslant n} \frac{2}{j - i + 1} \leqslant 2(n-1)\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right).$$

From analysis we know that the harmonic function $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + O(1)$. Hence the average complexity is $\leqslant 2n \ln n = O(n \log n)$.

Chebyshev-Markov inequality

We want to know also what is the probability that the number of steps is much larger than the expected value. Here is a theorem that helps:

Theorem (Chebyshev-Markov inequality) Let X be a random variable with $\mathbb{E}X = \mu$, and let $c>0$ be any constant. Then

$$\Pr\{X>c\mu\} \leqslant 1/c.$$

Indeed, let $X' = 0$ when $X \leqslant c\mu$ and $X' = c\mu$ when $X>c\mu$. Then

$$c\mu \cdot \Pr\{X>c\mu\} = \mathbb{E}X' \leqslant \mathbb{E}X = \mu.$$

So for example the probability that Quicksort takes longer than $20n \ln n$ is less than $1/10$. (There are much better estimates of this probability for Quicksort, but they use similar principles.)

Hash tables

Problem A very large universe U of possible items, each with a different key. The number n of actual items that may eventually occur is much smaller.

Solution idea Hash function $h(k)$, hash table $T[0 \mathinner{.\,.} m - 1]$. Key k hashes to hash value (bucket) $h(k)$.

New problem Collisions.

Resolution Chaining, open hashing, and so on.

Uniform hashing assumption Assumes that

- Items arrive “randomly”.
- Search takes $\Theta(1 + n/m)$, on average, since the average list length is n/m.

What do we need? The hash function should spread the (hopefully randomly incoming) elements of the universe as uniformly as possible over the table, to minimize the chance of collisions.

Keys into natural numbers It is easier to work with numbers than with words, so we translate words into numbers.
For example, as radix 128 integers, possibly adding up these integers for different segments of the word.

To guarantee the uniform hashing assumption, instead of assuming that items arrive "randomly", we we choose a random hash function, $h(\cdot, r)$, where r is a parameter chosen randomly from some set H.

Definition The family $h(\cdot, \cdot)$ is universal if for all $x \neq y \in U$ we have

$$\Pr\{\, h(x, r) = h(y, r) \,\} \leqslant \frac{1}{m}.$$

If the values $h(x, r)$ and $h(y, r)$ are pairwise independent, then the probability is exactly $\frac{1}{m}$ (the converse is not always true). Thus, from the point of view of collisions, universality is at least as good as pairwise independence.

An example universal hash function

We assume that our table size m is a prime number.
(There are tables of prime numbers, it will be easy to find one between, say, m and $4m$: by Chebyshev's theorem for every k there is a prime between k and $2k$.)
Let $d>0$ be an integer dimension. We break up our key x into a sequence

$$x = (x_1, x_2, \dots, x_d), \quad 0 \leqslant x_i < m.$$

(If x is a bit string, break it into segments of size $\log m$.) Fix the random coefficients $0 \leqslant r_i < m$, $i = 1, \dots, d$, therefore the number of possible random inputs is $|H| = m^d$.

$$h(x, r) = r_1 x_1 + \cdots + r_d x_d \bmod m.$$

We use the notation $a \equiv b \pmod{m}$ for $a \bmod m = b \bmod m$. This is the same as requiring $m|(a-b)$.

Fact Let p be a prime number, $r \not\equiv 0 \pmod{p}$, and $ar \equiv br \pmod{p}$ then $a \equiv b \pmod{p}$.

Indeed, by the fundamental theorem of arithmetic, if a prime number divides a product, it must divide one of its factors. Here, p divides $(a-b)r$. It does not divide r, so it divides $a-b$.

Let us show that our random hash function is universal. Assume $(x_1, \ldots, x_d) \neq (y_1, \ldots, y_d)$. We show that $\Pr\{ h(x,r) = h(y,r) \} \leqslant 1/m$. Let i be such that $x_i \neq y_i$, for example $x_1 \neq y_1$. If $h(x,r) = h(y,r)$ then

$$0 \equiv h(x,r) - h(y,r) \equiv r_1(x_1 - y_1) + A \pmod{m},$$
$$A \equiv r_1(y_1 - x_1) \pmod{m},$$

where A only depends on the random numbers $r_2, \ldots, r_d$. No matter how we fix $r_2, \ldots, r_d$, there are m equally likely ways to choose r_1. According to the Fact above, only one of these choices gives $r_1(y_1 - x_1) \equiv A \pmod{m}$, so the probability of this happening (conditionally on fixing $r_2, \ldots, r_d$) is $1/m$. Since this probability is the same under all conditions, it is equal to $1/m$.

Closest pair of points revisited

Using hashing, we will give a more efficient and more general algorithm to find the closest pair of points among n points.
Set of points P in the plane, say inside the unit square $[0, 1] \times [0, 1]$.
New strategy, using an appropriate data structure $P(\delta)$.

- We proceed by stages. In each stage, candidate shortest distance δ.
- Take points $p_1, p_2, \ldots$ from P in random order. For point p_i check whether it is closer than δ to any of the points $p_1, \ldots, p_{i-1}$ in the data structure $P(\delta)$.
  If yes, update the structure $P(\delta)$ with the new distance δ.
  Otherwise store p_i into $P(\delta)$.

Questions:

- What is the data structure?
- How long does this take, on average?

Partition the unit square into a grid of subsquares of sides $\delta/2$: they can be denoted as $S_\delta(k,l)$ for $k,l = 1,\dots,\lceil 1/2\delta \rceil$. To each point $p \in P$, let

$$Q_\delta(p)$$

be the square $S_\delta(k,l)$ in the partition where it belongs to.

- $P(\delta)$ is a hash table for the squares $Q_\delta(p_i)$, for $i = 1,2,\dots$ as keys. With each key $Q_\delta(p_i)$ we store p_i as a value. There is no p_j with $j<i$ in the same square, since then we would have $d(p_i,p_j)<\delta$.
- If for the new p_i we have $d(p_i,p_j)<\delta$ for some $j<i$, then p_j is in one of the 25 squares centered around $S_\delta(k,l) = Q_\delta(p_i)$: so we check only the 25 elements $S_\delta(k',l')$ in the table, where $|k-k'|,|l-l'| \leqslant 2$.
- If a new shortest distance δ is found then the new squares $Q_\delta(p_j)$ are reinserted into the new table $P(\delta)$ for $j = 1,\dots,i$.

Running time

Lookups and distance computations: At most 25 for each point.

Insertions: Let $X_i = 1$ if if stage i causes the shortest distance to change, 0 otherwise. Stage i has $1 + iX_i$ insertions, the total is

$$n + \sum_{i=1}^{n} iX_i.$$

Lemma $\mathbb{E}X_i = 2/i$.

Indeed, let p, q be the closest pair of points among $p_1, \dots, p_i$. $\Pr\{\, p \text{ or } q \text{ comes last} \,\} = 2/i$. Expected number of insertions:

$$n + \sum_{i=1}^{n} i \cdot 2/i = 3n.$$

For some problems exact solution seems hopeless. For those stated optimization problems, we can hope for a solution that approximates the optimum. We have seen already some examples:

- In homework problem 4 of set 1, we have shown that certain activity selection algorithms, though not optimal, approximate the optimal number of activities within a factor of 2.
- In problem 1 of set 4, we analyzed a polynomial algorithm for the knapsack problem, showing that it approximates the optimum within a factor of 2. (This is valuable since no polynomial algorithm is known for the knapsack problem.)
- A simple greedy algorithm for maximum matching also finds a matching that is not worse than half of the optimum.

Now we will see some more interesting examples—showing that even if a problem is proven difficult, giving up is not the right answer.

Suppose we have a set of places V, and a set of towns $U \subseteq V$. Our goal is to build a set $C \subseteq V$ of service centers in such a way that every town is close to at least one of these centers. More precisely, we call the distance $d(u,v)$ of two places the cost of going from place u to place v. If $C \subseteq V$ is a set of k centers, then let

$$d(v,C) = \min_{c \in C} d(v,c), \quad r(C) = \max_{u \in U} d(u,C).$$

So $r(C)$ is the distance of the town farthest from the set C: it is called the covering radius of C.

Center selection problem Given a distance function $d(\cdot,\cdot)$ over the set V of places, a set of towns $U \subseteq V$ and a number k, find the set of k centers C with minimum $r(C)$.

This is what it means that we want no town to be too far from C.

Properties of distance

We will consider a restricted version of the problem, in which the cost $d(u,v)$ satisfies the following two properties:

Symmetry $d(u,v) = d(v,u)$: the cost of going from u to v is the same as going from v to u.

Triangle inequality $d(u,w) \leqslant d(u,v) + d(v,w)$.

Though both requirements are reasonable, there are natural cases when they are not satisfifed. For example, $d(u,v)$ may not be symmetric in a city with one-way streets. And if going from town u to town w the only way is via town v, but this forces an overnight stay in a hotel for a price p, then it may happen that $d(u,w) = d(u,v) + d(v,w) + p$.

A bad greedy algorithm

Natural idea: Keep adding places in such a way that the covering radius is always smallest.

Example The above greedy algorithm can give an arbitrarily bad result.
Let $V = -1, 0, 1$, $U = \{-1, 1\}$, and the distance the difference. Then this algorithm chooses point 0 first, and for example point 1 second. This gives a covering radius 1, while the optimum is 0.

Here is an algorithm that is not too bad: we only use towns as centers. Keep adding the town to C that is farthest from it.

```
ADD-FARTHEST(k)
  C ← {u} for some arbitrary initial town u ∈ U
  for i = 2 to k do
      find the town v ∈ U farthest from C, that is r(C) = d(v,C)
      C ← C ∪ {v}
```

Algorithm ADD-FARTHEST(k) is not optimal.

Example Let $U = V$ be the set of 9 points on a 3×3 square grid with the ordinary Euclidean distance:

$$V = \{ (x, y) : x \in \{-1, 0, 1\}, y \in \{-1, 0, 1\} \}.$$

If $u_1 = (x_1, y_1)$, $u_2 = (x_1, y_2)$ then

$$d(u_1, u_2) = \left((x_1 - x_2)^2 + (y_1 - y_2)^2 \right)^{1/2}.$$

If we start with $v_1 = \{-1, -1\}$, then the algorithm chooses $v_2 = (1, 1)$, so $C = \{(-1, -1), (1, 1)\}$.
Now $r(C) = 2$, since $(1, -1)$ is at distance 2 from C. But even the single point $(0, 0)$ is better: $r(\{(0, 0)\}) = \sqrt{2}$.

Algorithm Add-Farthest(k) is not too bad:

Theorem If C^* is an optimal set and algorithm Add-Farthest(k) computes a set C then

$$r(C) \leqslant 2r(C^*).$$

Proof. Let $r = r(C^*)$. For a contradiction assume $r(C) > 2r$. Then the towns of C are at a distance $> 2r$ from each other. For each town u in C there is some place $u' \in C^*$ in the ball of radius r with center u. If $u \neq v$ then $u' \neq v'$. Indeed, otherwise by the triangle inequality $d(u,v) \leqslant d(u,u') + d(u',v) \leqslant 2r$. But then the balls of radius $2r$ around towns of C include all balls of radius r around the places of C^*, and so cover all towns. □

Consider an undirected graph $G = (V, E)$. A set of vertices H is a vertex cover if every edge has at least one end in H: if $e = \{u, v\}$ is an edge, then $H \cap \{u, v\} \neq \emptyset$.

- Our problem is to find a minimal-size vertex cover. We will see later that this is a really hard problem, so we can only hope to find an approximation.
- More generally, suppose that each vertex has some cost, or weight: the cost of vertex v is $c_v \geqslant 0$. We want to find the vertex cover whose cost is minimal, that is

$$c(H) = \sum_{v \in H} c_v$$

is as small as possible.

- The vertex cover question, even with unit costs, is interesting. Consider, for example, the following theorem.

Theorem In a bipartite graph, the size of the minimum vertex cover is equal to the size of the maximum matching.

  It is easy to see that the size of each vertex cover bounds the size of each maximum matching. On the other hand the equality is not trivial: it follows from the max-flow min-cut theorem.

- The vertex cover problem is probably very hard: we will see later that it is NP-hard, which makes it likely that even the best algorithm for solving it is not much faster than brute-force (exponential). On the other hand, there are some interesting approximation algorithms for it.

Greedy approach

- There is a natural greedy algorithm, repeating the following step:

  > *Choose a vertex with the largest degree, then delete it from the graph.*

  This is not a bad algorithm, but there are some nasty graph examples in which it approximates the optimum only within a factor of $O(\log n)$.
- On the other hand, the following non-greedy algorithm finds a vertex cover that is at most twice as large as the optimum, in the case where each vertex has unit cost. It repeats the following step:

  > *Find an edge that is not covered yet. Choose both of its endpoints. Delete the endpoints from the graph.*

- In what follows we develop an algorithm for the more general case, with possibly non-unit costs.

Dual approach via prices

- Sometimes it pays to start by finding a bound from the other side on our solution: this helps estimate how far we are from the optimum. This is similar to as if we approached the maximum flow problem from the side of trying to find a small cut: it is called a dual method.
- Here, the dual approach looks for a lower bound on the vertex cover cost. For this, we introduce the notion of prices.
- Suppose that a number $p_e \geqslant 0$ is assigned to each edge. We say that these constitute a valid set of prices, if for each vertex u,

$$\sum_{e \ni u} p_e \leqslant c_u.$$

  We call this the fairness condition for vertex u. The idea is that an edge e must pay to each of its end vertices p_e for covering it, but no vertex should receive more in total than its cost.

Prices help lowerbound the optimum. For a set of prices p, let $s(p) = \sum_{e \in E} p_e$. Let H be a vertex cover and p a set of prices obeying fairness. Then $c(H) \geqslant s(p)$. Indeed,

$$c(H) = \sum_{u \in H} c_u \geqslant \sum_{u \in H} \sum_{e \ni u} p_e \geqslant \sum_{e \in E} p_e = s(p). \tag{7}$$

Our algorithm will just create some prices that push up the lower bound (7) as much as possible. Every time it touches a price, it pushes it up until one of the two fairness inequalities at its ends becomes equality. In this case, we will call that vertex tight. When there is no more increase possible, the tight vertices form a vertex cover.

```
for each edge e do
    increase p_e until one of its two inequalities becomes tight
return the set H of tight vertices
```

Let us show that the cost $c(H)$ in the vertex cover obtained is at most 2 times the (lower bound to the) optimum:

$$c(H) = \sum_{u \in H} c_u = \sum_{u \in H} \sum_{e \ni u} p_e \leqslant 2 \sum_{e \in E} p_e = 2s(p).$$

The factor 2 is needed, since some edges on the left-hand side were counted twice. But we have seen that $s(p)$ is a lower bound to the cost of every vertex cover, in particular of the optimal ones.

Here is an exact theorem relating to the edge prices, similar to the max-flow min-cut theorem. Let a fractional vertex cover be a set of values $x = (x_v : v \in V)$ satisfying the following inequalities:

$$x_v \geqslant 0 \text{ for all vertices } v,$$
$$x_u + x_v \geqslant 1 \text{ for all edges } \{u,v\}.$$

Let $c(x) = \sum_{v \in V} c_v x_v$ be the cost of the fractional vertex cover. It is easy to see that for all fractional vertex covers x and prices p obeying fairness, we have $c(x) \geqslant s(p)$. The strong duality theorem says that $\min_x c(x) = \max_p s(p)$. We will not prove it here.

Assume $v_1 = \max_i v_i$. For the optimal solution x' of the changed problem, estimate $\frac{\sum_i v_i x'_i}{\mathrm{OPT}} = \frac{\sum_i v_i x'_i}{\sum_i v_i x^*_i}$. We have

$$\sum_i (v_i/r) x'_i \geqslant \sum_i v'_i x'_i \geqslant \sum_i v'_i x^*_i \geqslant \sum_i (v_i/r) x^*_i - n,$$
$$\sum_i v_i x'_i \geqslant \mathrm{OPT} - r \cdot n = \mathrm{OPT} - \varepsilon v_1,$$

where we set $r = \varepsilon v_1/n$. This gives

$$\frac{\sum_i v'_i x'_i}{\mathrm{OPT}} \geqslant 1 - \frac{\varepsilon v_1}{\mathrm{OPT}} \geqslant 1 - \varepsilon.$$

With $v = \sum_i v_i$, the number of operations is of the order of

$$nv/r \leqslant n^2 v / v_1 \varepsilon \leqslant n^3/\varepsilon,$$

which is polynomial in $n, 1/\varepsilon$.

In these lectures, we will talk about the kind of evidence we have when we say that a problem is hard.

- We may have a mathematical proof of its hardness, or even algorithmic unsolvability (examples in the course CS332).
- We just can say that many smart people tried it. . .
- We may say that its solution would solve a broad class of problems, that have been tried over many centuries.

We will concentrate on the last case: the class is the class of NP problems.

We saw examples when the solution of one problem also helped us solve another.

- Sequence alignment reduced to finding a shortest path in a certain graph.
- Maximum matching to network flow.

Definition An algorithm A polynomially reduces problem P is to problem Q if it solves P on every input x provided it gets an instant solution to every possible input y_1, y_2 to problem Q from some black box. We will write

$$P \leqslant_p Q$$

if there is such an algorithm A.

Algorithm A does not care how the black box (oracle) gets its answers, and how fast: it just uses them.
For example, the reduction algorithm for maximum matching created from any bipartite graph G a flow network F. And then, from an integer-valued maximum flow of F, it created a maximum matching for G. It did not care about how the integer-valued maximum flow was obtained: that part was used as a black box.

So far, we used reduction to help solve problem P provided somebody solves problem Q. But reductions can be applied also in the opposite direction. If we have evidence that P cannot be solved in polynomial time, and know $P \leqslant_p Q$ then this provides evidence that Q cannot be solved in polynomial time either.

Examples

- Minimum vertex cover to maximum independent set (and vice versa).
- Vertex cover to set cover. Simple, as we reduce a problem to a more general one.
- Two integer equations to one integer equation (subset sum). More interesting, as we reduce a problem to a less general one.

Two subset sum problems in one, for $x_i = 0, 1$:

$$a_{11}x_1 + \cdots + a_{1n}x_n = b_1 \tag{8}$$

$$a_{21}x_1 + \cdots + a_{2n}x_n = b_2 \tag{9}$$

Assume $D > a_{ij}, b_i$. Define $A_j = a_{1j} + a_{2j}D$, $B = b_1 + b_2D$. Now the two equations above have a solution if and only if the following one equation has:

$$A_1x_1 + \cdots + A_nx_n = B. \tag{10}$$

Indeed, for example taking remainders modulo D gives back (8). Then subtracting (8) from (10) and dividing by D gives back (9).

Many of the problems we have seen are, even if not solvable polynomially, have a weaker but useful property: if a solution is offered, it can be verified in polynomial time.

Examples

- Perfect matching.
- Compositeness of an integer.
- Large independent set in a graph.

An NP problem is given by a verification function

$$V(x, w)$$

computable in time polynomial in the length of x. Here x is the input, or instance. Argument w is a potential witness, or certificate.

Examples

Perfect matching problem: input is a bipartite graph G. Potential witness: a set of edges M. Verification function: checks whether M is a perfect matching for G (and thus a witness).

Compositeness problem: input is an integer x. Potential witness: an integer w. Verification function: checks whether w is a proper divisor of x.

Large independent set problem: input is a pair (G, k), graph G and integer k. Potential witness: a set U of vertices in G. Verification function: checks whether U is an independent set of size $\geqslant k$.

A number of NP problems are related to optimization problems. For example, the "large independent set problem" is related to the problem of finding a maximum-size independent set.

- Clearly, a solution to the optimization problem would answer any question of the kind: "given G, k, does G have an independent set of size $\geqslant k$?".
- But there is also a reduction in the other direction. Indeed, a black-box reduction just could ask questions about $(G, 1)$, $(G, 2)$,. . ., (G, n). The last k for the answer is affirmative, is the size of the maximum independent set.
- The above reduction can be speeded up via binary search.

- Let us use Boolean variables $x_i \in \{0, 1\}$, where 0 stands for false, 1 for true. A logic expression is formed using the connectives $\wedge, \vee, \neg$: for example

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee x_4).$$

- An assignment (say $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 0$) allows to compute a value (in our example, $F(0, 0, 1, 0) = 0$).
- An assignment (a_1, a_2, a_3, a_4) satisfies F, if $F(a_1, a_2, a_3, a_4) = 1$. The formula is satisfiable if it has some satisfying assignment.
- Satisfiability problem FSAT: given a formula $F(x_1, \ldots, x_n)$ decide whether it is satisfiable.

Special cases:

- A conjunctive normal form (CNF) $F(x_1, \ldots, x_n) = C_1 \wedge \cdots \wedge C_k$ where each C_i is a clause, with the form $C_i = \tilde{x}_{j_1} \vee \cdots \vee \tilde{x}_{j_r}$. Here each $\tilde{x}_j$ is either x_j or $\neg x_j$, and is called a literal.
  SAT: the satisfiability problem for conjunctive normal forms.
- A 3-CNF is a conjunctive normal form in which each clause contains at most 3 literals—it gives rise to 3SAT.

The 3SAT problem sounds especially basic. It asks to satisfy a number of very simple constraints: each a disjunction clause with up to three literals.

Theorem (Cook-Levin) Every NP problem is reducible to SAT.

So if a polynomial algorithm could be found for SAT then every NP problem could be solved in polynomial time. In other words, a fast algorithm to check solutions would guarantee also a fast algorithm to decide whether there is any solution at all. This is unlikely, so probably SAT is hard.
An NP-problem to which every NP problem is reducible is called NP-complete. Now we know that SAT is NP-complete.

First we define logic circuits. If C is such a logic circuit then let $\mathrm{CSAT}(C) = 1$ if C is satisfiable and 0 otherwise.

Lemma The CSAT problem is NP-complete.

Consider any NP problem A with polynomial-time verification function $V_A(x, w)$, we will reduce it to CSAT. There is a polynomial-size circuit C_A computing V_A. Hardwiring input string x we get a circuit $C_{A,x}$ with input w. Its satisfying assignments w are just the witnesses for $V_A(x, w)$.

Given a circuit C, we introduce a new variable y_i for the output of each node, along with a constraint saying it computes what it is supposed to. For example if it is an OR gate saying $x_i \vee x_j = y_k$, this is expressed using

$$(\neg x_i \vee y_k) \wedge (\neg x_j \vee y_k) \wedge (\neg y_k \vee \neg x_i \vee \neg x_j).$$

Let y_N be the output of the circuit. The 3SAT formula $F_C(x_1, \ldots, x_n, y_1, \ldots, y_N)$ that is the conjunction of all these constraints is true if and only if y_N is computed from $x_1, \ldots, x_n$ by the circuit C. Therefore the 3CNF

$$F_C(x_1, \ldots, x_n, y_1, \ldots, y_N) \wedge y_N$$

is satisfiable if and only if the circuit C is.

SAT to independent set

Karp's reduction of SAT to the independent set problem shows that the independent set problem is also NP-complete.