Algorithms

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

Computer Science Department
Boston University

Spring 2020

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.

Stable matching

- Matchings in a graph. Perfect matchings.
- Nodes: persons. Each orders all possible partners by preference.
- Women, say A, B, C. Men, say X, Y, Z.
- Instability in a matching: a non-matched pair that would be an improvement for both participants.

	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. If for example everybody can be paired with everybody (no sex distinction) the answer is no. (This is the stable roommate problem.)

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

Example (Stable roommate problem)

Rankings of each of A, B, C, D:
$A : (B, C, D), B : (C, A, D), C : (A, B, D), D : (A, B, C)$.

- D is ranked last by everyone else.
- Every non-D is ranked first by some non-D.

So in every matching, the partner of D will be part of an instability with the one by whom he/she is ranked first.

Brute force

- We can always find if there is a stable matching and find one, by going through all possible perfect matchings between n men and n women.
- What is the number of those matchings?

An algorithm

Question 1: we give an algorithm (invented by Gale and Shapley). This answers also Question 2.
The algorithm simply says: “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, each step benefits some woman (by giving her a match or improving her match), without harming any other 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.

Discussion

Questions

- Whom does the end result of this algorithm benefit: men or women?
- How unique is the matching M^* we found?

- "Politically correct" version: men-women replaced with hospitals-students. This is also a major real application.
- Example of an approach to problem solution: 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.)

Definition A valid partner of a person is one with which he/she is matched in some stable matching.

Theorem In M^*,

ⓐ each man gets the best valid partner,

ⓑ each woman gets the worst valid partner.

The proof refers to “hospitals-students” in place of “men-women”.

Hospital-optimality of M^*

- Suppose hospital h is the first one rejected by a student s with whom it is paired in some stable matching M.
- The rejection happens when h' proposes, whom s prefers to h.
- h' prefers s to its pair s' in M, else it would have been rejected by s' before proposing to s—and it had no rejection yet.
- Then $h'-s$ is unstable in M: contradiction.

Student-pessimality of M^*

- Suppose that h is not the worst valid partner of its pair s in M^*: it has a less preferred one, h', matched with it in a stable matching M.
- Let s' be paired with h in M.
- By hospital-optimality, h prefers s to s'.
- Then $h - s$ is unstable in M: contradiction.

Questions

- Example (when not doing "men propose, women dispose") of an infinite series of switches through a cycle of unstable matchings? Each switch joins an unstable pair and the partners they leave behind.
- Does every possible algorithm need $\Omega(n^2)$ steps?
  - This a lower bound question. Such questions can be very hard, they need the consideration of all possible algorithms.
  - Assume that inspecting each preference is an extra step. How many preferences do we have to inspect in the worst case? Think of this as a game between the algorithm asking about the preferences and an adversary supplying the data.

Solved exercises

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

Note that all these are variations on the theme of independent set.

Asymptotic analysis

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 $f(n) \le c \cdot g(n)$ 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 $f(n) \ge c \cdot g(n)$ 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 $\log 3$.

Examples

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

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:

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

Sums: the art of simplification

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:

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

Lower bound:

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

Review of data structures

RAM

RAM (random access machine): a simple model that corresponds reasonably to the real, complex machines we use.

Memory an array of n (large, fixed number) of memory locations $M[i]$, each containing a word of some width of w bits. Normally, $w \approx \log n$, so an address of any memory location fits into a word.

Primitive operations A machine-language program is a sequence of these:

- reading/writing a memory location (given by an address)
- simple arithmetic/logic on words
- branching based on a simple condition, say $M[0] > 0$.

Computing time Number of primitive operations taken during the execution of the program.

Higher-level constructs (like functions, recursion) must be translated by a compiler into a machine-language program.

What is a data structure

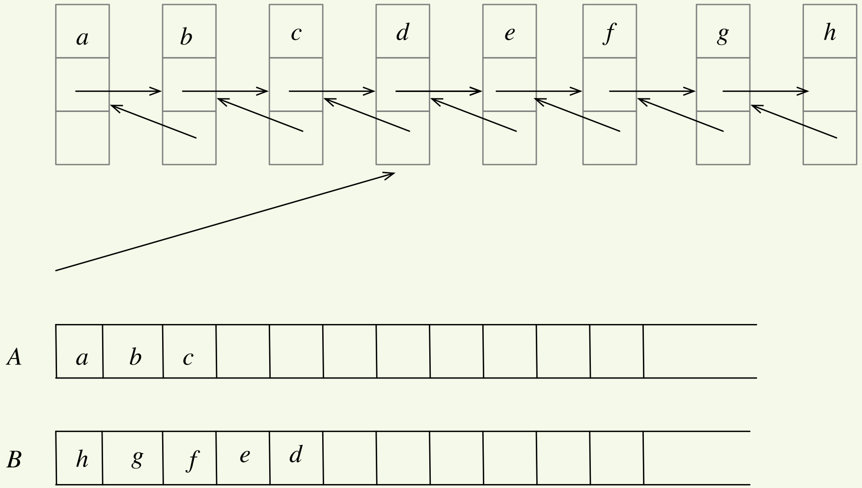
Abstract data type Does not say how it represents the data but does say what operations it is expected to carry out efficiently.
Example: stack (last-in-first-out), queue (first-in-first-out), priority queue, dictionary.

Data structure A way of representing data that is one level up from just words in memory.

- Most common: array. Can be fixed-length or variable-length. Has many-many uses.
  Implementation of variable length: copy into a new array of double size (or half size) if needed.
- Other data structures may consist of some records that contain fields, and some of these fields are links to other records. All these fit together in a particular way described by the structure.
  Examples: linked list, binary tree, heap, hash table.

Linked lists

- A (say, doubly) linked list has the advantage that an item can be inserted/deleted at a given position in it, at unit cost. It implements stacks and queues easily.
- In practice, implementation by an array (if needed, variable-length) is often easier and faster.
- If only one end changes (as for a stack), a single array suffices.
- If insertion/deletion at just one place is needed, use two arrays (the second one in reverse order).



$A.length = 3, \ B.length = 5$

Inverting an array

Suppose that an array A of length n contains all the numbers $1, \ldots, n$ in some order. This is also called a permutation. Example

$$A[1] = 3,\ A[2] = 2,\ A[3] = 4,\ A[4] = 1.$$

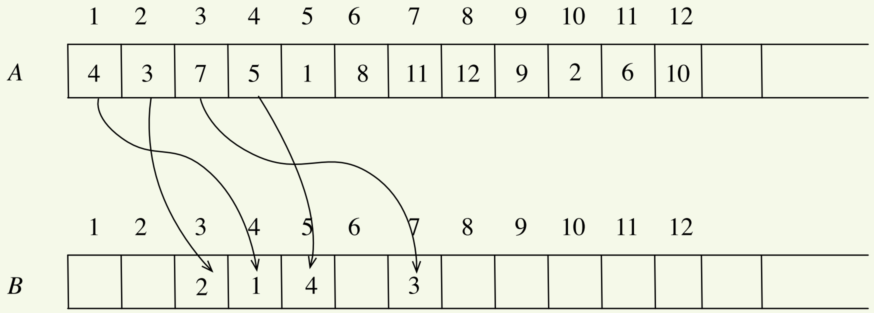
Example of A: Hospital h assigns students $1, 2, 3, 4$ the preferences $3, 2, 4, 1$.
We may frequently want another array B, its inverse (also a permutation), that is $B[i] = j$ iff $A[j] = i$. So we have

$$B[1] = 4,\ B[2] = 2,\ B[3] = 1,\ B[4] = 3.$$

Example of B: Hospital h lists the students in its order of preferences (starting with the most preferred) as $4, 2, 1, 3$.

The array A can be inverted into an array B by the command:

```
**foreach** $i$ **do**
    $B[A[i]] = i$
```



Implementing Gale-Shapley

Let us implement the algorithm just using arrays.

Input data in two 2-dimensional arrays (indexed from 1):

HrankS$[h,s]$ = the rank given by hospital h to student s.

SrankH$[s,h]$ = the rank given by student s to hospital h. Set SrankH$[s,0] \leftarrow n+1$.

Preprocess for each hospital, invert the array HrankS$[h,s]$:

HbestS$[h,k]$ shows the kth best student for hospital h.

Matching in two one-dimensional arrays:

Smatch$[s]$ = the hospital matched to student s, 0 if unmatched.

HmatchR$[h]$ = the rank of the student whom hospital h tries to match. Initially, HmatchR$[h] \leftarrow 1$.

Unmatched hospitals in a stack, represented by an array $U[i]$.

Initially $U[i] = i$. Variable L keeps the index of its last filled element $U[L]$. Initially, $L = n$.

```
GS(HrankS, SrankH):                    // the Gale-Shapley algorithm
    **foreach** $h, s$ **do**                            // invert student ranks
        HbestS[$h$, HrankS[$h, s$]] $\leftarrow s$
    **while** $L \neq 0$ **do**            // work on last unmatched hospital
        $h \leftarrow U[L]$
        $s \leftarrow$ HbestS[HmatchR[$h$]]
        $h' \leftarrow$ Smatch[$s$]
        **if** SrankH[$h$] $>$ SrankH[$h'$] **then**    // $h$ tries next best
            HmatchR[$h$] $\leftarrow$ HmatchR[$h$] $+ 1$
        **else**
            Smatch[$s$] $\leftarrow h$
            **if** $h' = 0$ **then**     // one fewer unmatched hospital
                $L \leftarrow L - 1$
            **else**         // $h'$ gets unmatched, will try next best
                $U[L] = h'$
                HmatchR[$h'$] $\leftarrow$ HmatchR[$h'$] $+ 1$
```

Comparing data structures

We may ask the cost (say, in time) to implement various operations of an abstract data type. Some examples:

Stack operations: push, pop, peek.

- *Linked list* implements each in constant, that is $O(1)$ time.
- *Variable-length array* implements these operations in constant amortized time. This means that the cost of n operations (starting from an empty stack) is $O(n)$, though some push operations may take more than constant time.

  Implementation: when the array reaches size 2^k then we double its size, at cost $O(2^k)$. Every increase of size 2^k can be charged to the last 2^k previous push operations.

Priority queue of size n. Each record has a key, and the "first" record is the one with the smallest key. Operations: Insert, find-first, delete, delete-first, change-key.

Sorted linked list find-first, delete-first in constant time. Others in time $O(n)$.

Binary heap (recall from the data structure course!): all these operations in $O(\log n)$ time.

We will use this as a component of some algorithms.

Dictionary of size n. Each item has an identifying key.

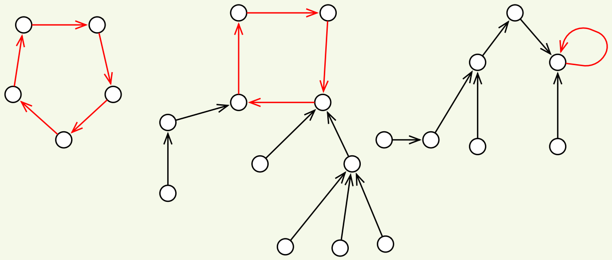
Operations: insert, find (by the key), delete.

Hash table On average, each operation costs $O(1)$. In the worst case, it may cost n.
Most software (including Python interpreters, Java compilers) implement dictionaries via hash tables. In Python, they are convenient even when containing just 3-4 items.

Balanced tree Each operation costs $O(\log n)$.
Large databases use balanced trees.

Graph representation

- A directed graph $G = (V, E)$ is given by a set V of vertices (nodes) and a set E of edges.
- Edge $e = (u, v)$, u is the start node and v the end node. If $u = v$ then e is a loop edge.
- Undirected graph: each edge (u, v) can also be written as (v, u), so it can be represented as the set $\{u, v\}$.
- We will always say whether the graph we are talking about is directed or undirected.
- Parallel edges: when there is more than one edge of the same form (u, v).
  Unless we say so, our graphs will have no parallel edges, and even no loop edges.
- Typically, we will use the notation $n = |V|$, $m = |E|$, that is our graph has n vertices and m edges. But there will be many other cases!



- In this graph, the number of vertices is $n = 23$. By the way, also the number of edges is $m = n = 23$, since here each vertex has exactly one outgoing edge: the outdegree is 1. The indegree varies from 0 to 3.
- In undirected graphs, we just talk about the degree of a vertex.

Representing a graph in the computer

- For simplicity, assume no parallel edges: for a pair u, v, at most one edge (u, v).
- With each node and edge, there can be some extra information to be stored, (call it a label).
  Example: vertices represent cities, edges represent roads between them; the roads have length.

Several possibilities for representing a graph: we will choose one based on:

- What questions to answer?
- How important is to save on storage?

Some questions:

1. Given vertices u, v, find out if there is an edge (u, v).
2. Given vertex u, find all edges leaving u, or all edges entering u.

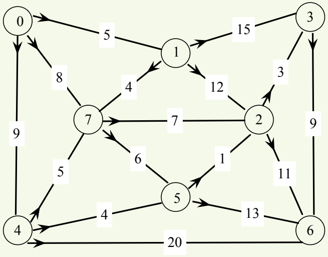
Let $V = \{v_1, \dots, v_n\}$. A natural data structure for answering question 1 is an adjacency matrix: an array $A[i, j]$ where $A[i, j] = 1$ if $(v_i, v_j) \in E$ and 0 otherwise.

$$A[i,j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Or, if edges (u, v) has label $l(u, v)$ representing its length, we could denote the lack of edge by the symbol ∞, and set

$$A[i,j] = \begin{cases} l(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ \infty & \text{otherwise.} \end{cases}$$

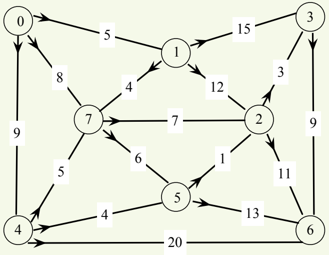
Example in which the vertices (and the rows and columns of the matrix) are numbered from 0, and we omit the entries with ∞:



	0	1	2	3	4	5	6	7
0		5			9			8
1			12	15				4
2				3			11	
3							9	
4						4	20	5
5			1				13	
6								
7			7			6		

A directed graph can have, even without loops, $n(n-1) \approx n^2$ edges. We call a graph dense if the number of edges is $m = \Omega(n^2)$, and sparse if it is $\ll n^2$.

It wastes storage to represent a sparse graph by an adjacency matrix. It may be better to represent it by an array of adjacency lists. This is an array B where for each i, $B[i]$ points to a (linked) list of all the edges leaving v_i. List i, below represents an edge (v_i, v_j) by the pair $(j : l(v_i, v_j))$.



0	(1:5), (4:9), (7:8)
1	(2:12), (3:15), (7:4)
2	(3:3), (6:11)
3	(6:9)
4	(5:2), (6:20), (7:5)
5	(2:1), (6:13)
6	
7	(2:7), (5:6)

Many algorithms actually rely on the adjacency lists, so this is not just an economical data structure but also a useful one. Its storage requirement is $O(n+m)$ which can be much less than the $O(n^2)$ needed by the adjacency matrix.

- Most practical examples of graphs are sparse. Think of a Netflix database, where each user rates each film she has seen. Say, 50 million users, up to 100 films seen by each.
- A graph that might be rather dense: consider a large collection of documents, say all articles published by the science publisher Elsevier (say, 3 million). Define a graph whose elements are all English words in some dictionary, say 20 thousand words. Two words are connected by an edge if there is some document in which they appear together. A label could be added, showing the number of documents in which they appear together.

- Answering the question: “is (u, v) an edge?”:
  In the adjacency list representation, worst case: up to n steps, if vertex u has a high degree. Indeed, we may have to scan the whole list.
- If adjacency matrix is still not economical, we may use a dictionary (implemented as a hash table). Two possibilities:
  - One dictionary for the whole graph, storing all edges (u, v).
  - An own dictionary for each vertex u, for all the edges leaving u. This makes it also easy to list all edges leaving u.

Sometimes we will not want to choose between data structures, but use several! For example, lists for incoming edges, lists for outgoing edges.

Breadth-first search

In a directed (or undirected) graph $G = (V, E)$, an algorithm relying on adjacency lists: breadth-first search. Passing through all vertices reachable from some source vertex s, in a certain order:

1. Find vertices reachable from s in one step.
2. Find vertices reachable from s in two steps.
3. And so on.

The algorithm proceeds in stages. Let L_k (kth layer) be the set of vertices reachable in exactly k steps. In stage k, we have found the set $I_k = \bigcup_{i<k} L_i$ (kth interior) of vertices reachable in $< k$ steps, and just finding L_k.

Visited vertices: $I_k \cup L_k = \bigcup_{i=0}^{k} L_i$.

Details of stage k : For each vertex in $L_{k-1} \subset I_{k-1}$, add its unvisited neighbors (using its adjacency list) to L_k.

- We can run the algorithm without dividing into stages, by creating a single queue

$$Q = L'_{k-1} \cup L'_k$$

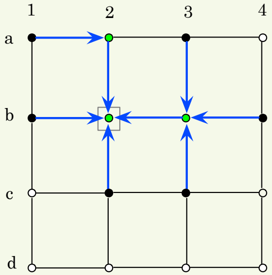
  where L'_{k-1} is the part of L_{k-1} yet to be processed, and L'_k the part of L_k already added, with L'_k in the back.
- The single step to run repeatedly:

  > Take off a vertex u from the front of queue Q, visit all its non-visited neighbors, add them to the end of Q.

- To every new vertex v visited, add the information on the node u from whose edge list it was taken: we will call it the parent $u = v.parent$ of v. This way we create a tree called the shortest path tree: following the parents repeatedly to the root s of the tree gives a shortest path.

BFS example

- Edge list of each node ordered by (east, north, west, south).
- The record $(u : v_1, v_2, v_3)$ means removing u from the front of the queue and adding its newly visited neighbors to the queue.
- Visited nodes green, black, queue Q black.
- Starting point $b2$, shown after processing $a2$.
- Blue arrows point to parents.



$(b2 : b3, a2, b1, c2), (b3 : b4, a3, c3),$

$(a2 : a1)$, This stage shown.

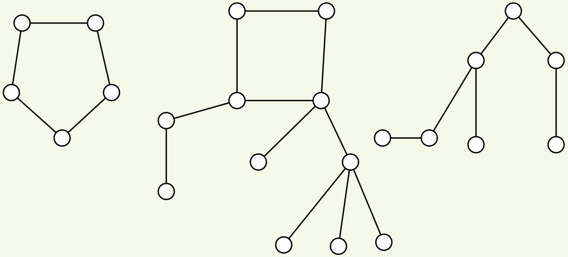
$(b1 : c1), (c2 : d2), (b4 : a4, c4),$

$(a3 :), (c3 : d3), (a1 :), (c1 : d1), (d2 :),$

$(a4 :), (c4 : d4), (d3 :), (d1 :), (d4 :)$

Connected components

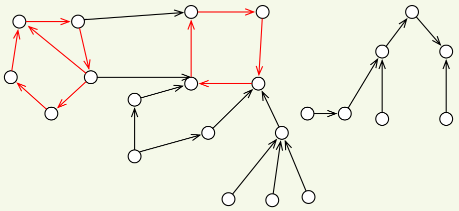
- Let $G = (V, E)$ be an undirected graph, and $s \in V$ a vertex. The set of vertices connected to s by a path (possibly by a "path" of length 0, so including s) is called the connected component of s.
- Clearly, if u is connected to v then v is also connected to u. Their components coincide, and if not then they are disjoint. So the set of vertices is a disjoint union $V = C_1 \cup \cdots \cup C_k$ of components: it is partitioned into the components.
- G is connected if it consists of a single component.
- We saw an algorithm that finds the connected component of a vertex s, namely all vertices reachable from s: breadth-first search.



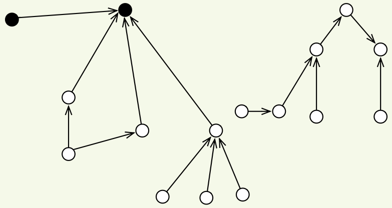
An undirected graph with 3 components.

Directed graphs

- The case of directed graphs is more complex: a directed path $(v_0, v_1, \ldots, v_k)$ is such that for each i, (v_i, v_{i+1}) is an edge. Unless saying otherwise, by a “path” in a directed graph we mean a directed path.
  We say that v is reachable from u if it is reachable on a directed path. We denote it as
  $$u \xrightarrow{*} v.$$
- This relation is transitive: $(u \xrightarrow{*} v) \wedge (v \xrightarrow{*} w) \Rightarrow (u \xrightarrow{*} w)$.
- In a directed graph, if $u \xrightarrow{*} v$ and $v \xrightarrow{*} u$ then we say that u and v are strongly connected. The strongly connected component of u is the set of all vertices strongly connected to it (including itself, connected by a “path” of length 0).
- Again, the set of vertices is a disjoint union of such components.



There are only 2 strong components of size > 1. The red edges run within the components. How many components are here?



Merging each component into a single vertex (shown black here), this new component graph is acyclic: has no directed cycle.

Acyclic graphs

- A graph is acyclic if and only if $u \xrightarrow{*} v$ and $v \xrightarrow{*} u$ implies $u = v$. (Prove it!) For an acyclic graph $G = (V, E)$ we will also use the notation $u \leq v$ for $u \xrightarrow{*} v$.
- A relation $\leq$ that is transitive with $(u \leq v) \wedge (v \leq u) \Rightarrow (u = v)$ is called a partial order. (Partial since it may happen that neither $u \leq v$ nor $v \leq u$ holds.)

Topological sort

- If the vertices of a graph $G = (V, E)$ are listed in an order $v_1, \ldots, v_n$ with the property that if (v_i, v_j) is an edge then $i < j$, then this list is called a topological sort.
- Clearly not possible if G has any directed cycle. But always exists if G is acyclic: we will even see an algorithm for it.
- The topological sort extends the partial order defined by G into a complete order.
- If the order is already complete then G contains a directed path going through all vertices. (Prove it!)

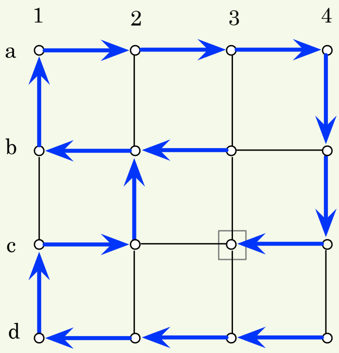
$$\begin{tikzcd}
 & d & \\
b \arrow[ur] & & c \arrow[ul] \\
 & a \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

This acyclic graph has two possible topological sorts: a, b, c, d and a, c, b, d. Adding the edge (b, c) turns it into a complete order (unique topological sort).

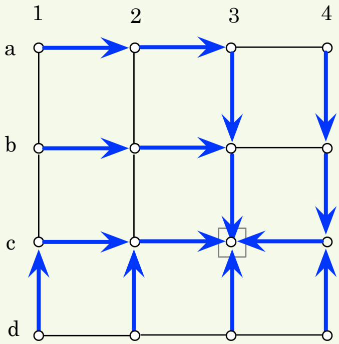
Depth-first search

Note For this material the textbook by CLRS (Cormen , Leiserson, Rivest, Stein) gives more useful detail than Kleinberg-Tardos.

- Another way to visit all vertices of a (directed or undirected) graph by following edges: from every newly visited vertex, follow an edge immediately to other unvisited vertices. Backtrack if there is none.
- Very simple to define using recursion.
- The path tree it defines is very different from shortest path trees: typically, it is narrow with long paths instead of wide with short paths.
- Its nice properties come handy sometimes (like here for topological sort).



- Depth-first search on an undirected graph. The adjacency lists are ordered (east, north, west, south). Starting from vertex $c3$.
- Produces a long skinny path tree (parent directions shown).

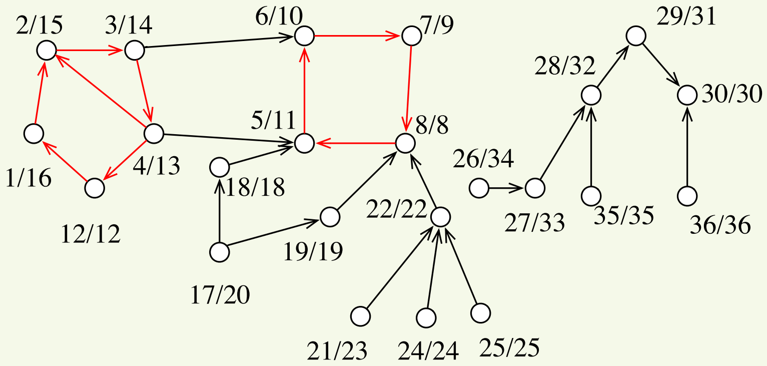


- Breadth-first search on an undirected graph. The adjacency lists are ordered (east, north, west, south). Starting from vertex $c3$.
- Produces a short bushy path tree (parent directions shown).

- Use the adjacency list representation, take the vertices from some *listIn* containing all elements of V in some order.
- The record u representing a vertex has a Boolean field *u.visited*.
- After processing each vertex, add it to the end of *listOut*.

```
DFS-visit(G, u, listOut):   // Visit all vertices reachable from u
    u.visited ← True
    foreach  out-neighbor v of u do
        if not v.visited then
            v.parent ← u
            DFS-visit(G, v, listOut)
    add u to listOut

DFS(G):
    listOut ← []
    foreach u in V do
        if not u.visited then
            DFS-visit(G, u, listOut)
    return listOut
```



Example discover/finish times of depth-first search. Output in order of finish times:
8,9,10,11,12,13,14,15,16,18,19,20,22,23,24,25,30,31,32,33,34,35,36.

Theorem If $u \xrightarrow{*} v$, but not $v \xrightarrow{*} u$, then depth-first search outputs v before u.

Proof.

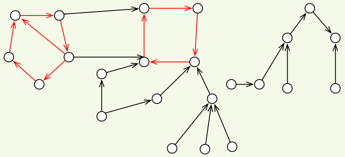
- Suppose that $\mathit{DFS\text{-}visit}(G, u, L')$ was called before $\mathit{DFS\text{-}visit}(G, v, L')$. Then $\mathit{DFS\text{-}visit}(G, v, L')$ was called recursively from within the call $\mathit{DFS\text{-}visit}(G, u, L')$, therefore v will be output before u, since u is output only at the end of $\mathit{DFS\text{-}visit}(G, u, L')$.
- Suppose that $\mathit{DFS\text{-}visit}(G, u, L')$ was called after $\mathit{DFS\text{-}visit}(G, v, L')$. Since not $v \xrightarrow{*} u$, call to $\mathit{DFS\text{-}visit}(G, v, L')$ ends before the call to $\mathit{DFS\text{-}visit}(G, u, L')$, and outputs v earlier.

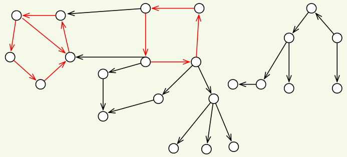
□

Corollary If G is acyclic then depth-first search outputs the vertices in reverse topological order.

Finding the strong components

- For directed graph G we can define the transpose graph $G^T = (V, E^T)$ by reversing the edges of G: same vertices, but $(u, v) \in E^T$ iff $(v, u) \in E$.
- The adjacency list of G^T represents G by listing for each vertex u the incoming edges of G (these are the outgoing edges of G^T).
- The graphs G, G^T are different, but have the same strong components.



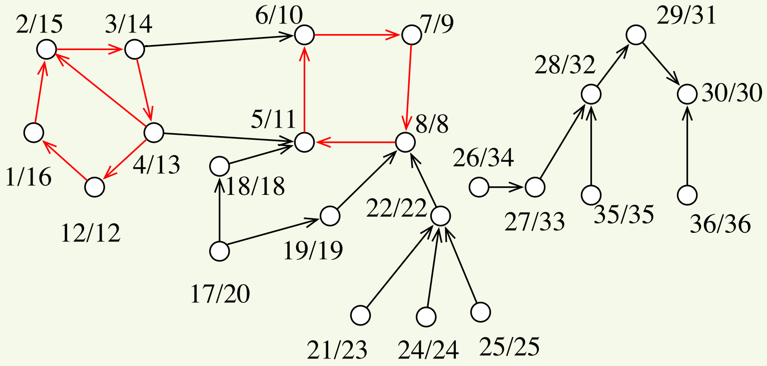


- We will see that taking the output list L' of $DFS(G, L, L')$ in reverse order and running a depth-first search of G^T on it, we obtain the strong components of G.
- Each strong component will be stored in its own list $strongComps[i]$: its elements are $strongComps[i][0]$, $strongComps[i][1]$, $\ldots$.

```
$findStrongComps(G)$:
    $listOut = DFS(G)$
    **foreach** $u$ in $V$ **do** $u.visited \leftarrow 0$
    $strongComps \leftarrow []$; $i \leftarrow 0$
    **for** $j = n$ **to** 1 **do**
        $u \leftarrow listOut[j]$
        **if** $u.visited = 0$ **then**
            $strongComps[i] \leftarrow []$
            $DFS\text{-}visit(G^T, u, strongComps[i])$
            add $strongComps[i]$ to $strongComps$
            $i \leftarrow i + 1$
    **return** $strongComps$
```

The program processes *listOut* in decreasing order.

- When $\mathit{DFS\text{-}visit}(G^T, u, \mathit{strongComps}[i])$ is called, the whole strong component of u is still unvisited (else u would have been visited, too). All its elements are added to $\mathit{strongComps}[i]$.
- Nothing else is added: indeed, suppose $v \xrightarrow{*} u$, but not $u \xrightarrow{*} v$. Then by the Theorem, v comes after u in *listOut*, so it has been processed before the call to $\mathit{DFS}(G^T, u, \cdot)$.



Strong components, as discovered by the algorithm (by their original finish times):

$\{36\}, \{35\}, \{34\}, \{33\}, \{32\}, \{31\}, \{30\}, \{25\}, \{24\}, \{23\}, \{22\},$
$\{20\}, \{19\}, \{18\}, \{16, 12, 13, 14, 15\}, \{11, 8, 9, 10\}.$

Greedy algorithms

A greedy algorithm generally assumes

- 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 $f(\cdot)$.)
- 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 makes the choice with the best contribution.

- Is generally fast.
- May not find the optimum, but sometimes it does.
- Even when it does not find the optimum it may find a good approximation to it.

Interval scheduling (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. Example:

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

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

Rationale This restricts our later choices least.

On the 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. 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 dominated by the cost of sorting.

Other possible strategies say, sorting by starting time we get only $(0,6),(6,10),(12,14)$.

Interval partitioning

Suppose our intervals are activities to be performed on some computers, and all must be scheduled. The goal is to schedule them on the smallest number of computers.

Lower bound If some point of time is contained in k activites then we need at least k computers. Example: activities $(1,5),(3,9),(6,8),(4,6)$. The activities $(1,5),(3,9),(4,6)$ all contain point 4, so we need at least 3 computers. 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 on the first available computer.

Could this need $> d$ computers? With this algorithm, at the first time when k computers do not suffice, there is an overlap of $k+1$.

Implementing interval partitioning

Input: Array A with $A[i] = (s_i, f_i)$, $i = 1, \dots, n$.
Output: List $C[j]$ of activities i assigned to computer j.
For computer j, let $F[j]$ be the finishing time of its last scheduled activity. To always find j with the smallest $F[j]$, can use a priority queue Q of records $(j, F[j])$ ordered by key $F[j]$.

```
intervalPartitioning(A):
    $k \leftarrow 1$; $F[1] \leftarrow 0$; insert $(1, F[1])$ into $Q$
    for $i = 1$ to $n$ do
        Let $F[j]$ be minimum key in $Q$
        if $s_i < F[j]$ then
            $k \leftarrow k + 1$; $j \leftarrow k$
        else
            delete the minimum $(j, F[j])$ from $Q$
        append $i$ to $C[j]$; $F[j] \leftarrow f_i$
        insert $(j, F[j])$ into $Q$
```

Cost of interval partitioning

Since Q can be implemented by a heap, each iteration costs at most $O(\log d)$ steps, so the total time cost is $O(n \log d)$.

Minimize maximum lateness

Now each task i has a duration t_i and a deadline d_i (and only one computer to process them). We choose start time s_i, then $f_i = s_i + t_i$. Minimize the maximum of all latenesses $L_i = f_i - d_i$ among all tasks.

“Greedy” algorithm Earliest deadline first.

Why optimal?

Consider a different order. Look at the first consecutive pair for which this order is different from the order of their deadlines: say $d_2 < d_3$, $s_3 < s_2$.

```
...............d2...............d3.....................
s3..............................f3..............f2
s2.............f2..................................f3
```

$$L_2 - L_3 = (s_3 + t_3 + t_2 - d_2) - (s_3 + t_3 - d_3) = t_2 - d_2 + d_3 > 0,$$

so $L_2 > L_3$. After swap, L_2 decreases by t_3, and L_3 increases by the same. Decreased the larger and increased the smaller—the maximum decreases.

(We used an exchange argument: if permutation π differs from the original order ι then has an inversion in a consecutive pair: swap brings π closer to ι.)

Other possible choices

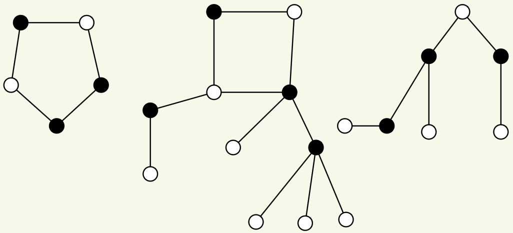
With counterexamples: order by length t_i, by slack $d_i - t_i$.
(We may leave this to the lab.)

Task partitioning

- Suppose that we have again tasks $i = 1, \dots, n$ with (integer) durations t_i, no deadlines, and 2 computers. What is the shortest time we can finish them all?
- It is a very difficult problem to solve this problem exactly: maybe all algorithms for it take exponential time.
- Suggest a greedy algorithm! How far can its result get from the optimum?

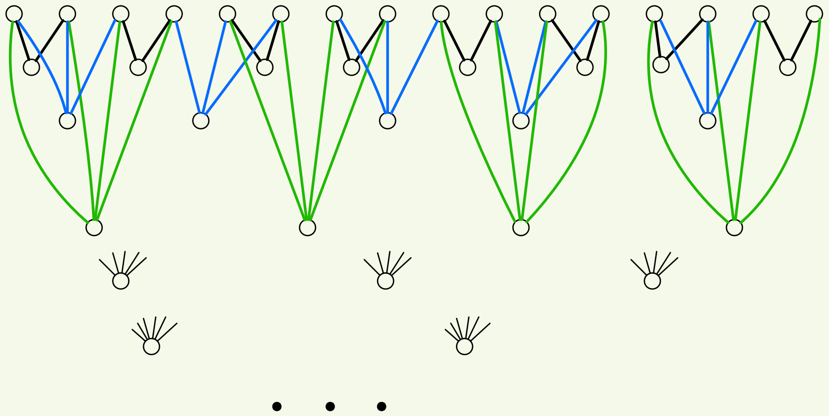
Vertex cover

For an undirected graph $G = (V, E)$, a vertex cover is a set S of vertices such that every edge has one of its ends in S. Example:



- Finding the smallest possible vertex cover is known to be hard. (Is the cover in the example optimal?)
- What would be a greedy algorithm giving at least a “rather good” vertex cover?

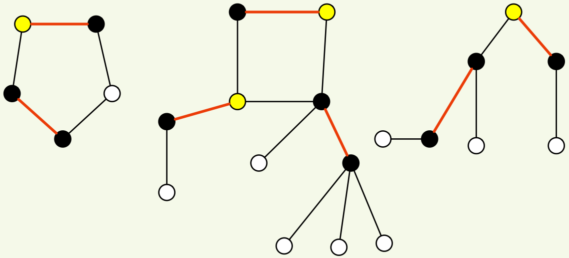
A bad graph for the greedy vertex cover algorithm



Row 1 has 16 points. Points of row i connect to disjoint groups of size i in the first row. The number of points in rows 2-16 is $8+5+4+3+2+2+2+1+\cdots+1=34$. The greedy algorithm picks all these, instead of just the 16 points of the first row.

- The example (with n in place of 16) shows that a vertex cover given by the greedy algorithm can be $\Omega(\log n)$ larger than the optimum. (Details below.)
- It is a theorem (proof skipped) that this is the worst possibility: it is at most $O(\log n)$ times larger than the optimum. (Also true for the more general problem of set cover.)

A non-greedy algorithm that is never that bad



- Repeatedly: pick an uncovered edge. Add both ends to the cover. (Not optimal: the yellow vertices are not needed.)
- If we picked k edges, these are disjoint, so every vertex cover must have size $\geq k$. Our vertex cover has size $2k$, so it is guaranteed to be at most twice worse than the optimum.
- The example does not say that the greedy vertex cover algorithm is always bad: only that in some bad cases it can be much worse than this non-greedy one.

Analyzing the counterexample

Each row $i \geq 2$ has $\lfloor n/i \rfloor$ points. If $i \leq n/2$ then

$$\lfloor n/i \rfloor \geq \frac{n}{i} - 1 = \frac{n-i}{i} \geq \frac{n/2}{i}.$$

So our sum is at least

$$\frac{n}{2}\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n/2}\right).$$

Let us lower-bound $\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$. Write it as

$$\frac{1}{2} + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \cdots + \frac{1}{8}\right) + \cdots .$$

There are $\lfloor \log k \rfloor$ full groups, each of size $\geq 1/2$.
The sum is $\geq \lfloor \log k \rfloor / 2 = \Omega(\log k) = \Omega(\log n)$ for $k = n/2$.

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.

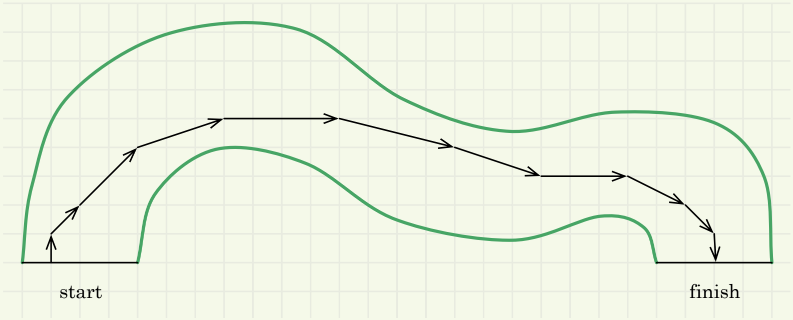
Edges if the word between the points is in the dictionary (maybe without the “s” at the end).

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.)

Car racing



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 $p_2 - p_1 = v_1$, $|v_2 - v_1|_\infty \le 1$.
Here $|(x, y)|_\infty = \max(|x|, |y|)$ is the so-called maximum norm.

Shortest paths with edge weights (lengths)

- Weight of edge $e = (u, v)$: $l(e) = l(u, v)$.
- 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.

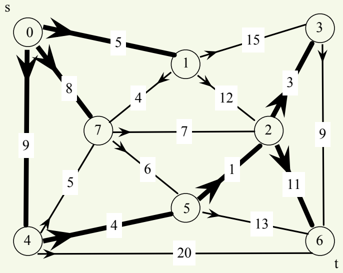
Dijkstra's algorithm

- 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 among those reachable by an edge from some $u \in S$: indeed, some such u is on any shortest path to v^*: $d(s) = d(u) + l(u, v^*)$.

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

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

  As we have seen, $d'(v^*) = d(v^*)$ (but generally $d'(v) \neq d(v)$ for $v \neq v^*$).
- The next element to take off Q and to add to S is $v^* = \arg\min_{v \in Q} d'(v)$. Set $d(v^*) = d'(v^*)$.
- Check all $v \notin S$ in the adjacency list of v^*. If $d(v^*) + l(v^*, v) < d'(v)$ set $d'(v) \leftarrow d(v^*) + l(v^*, v)$ and add v to Q if it is not there yet.



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).
$S' = \{1(5)0\}$, $Q = \{2(17)1, 3(20)1, 4(9)0, 7(8)0\}$
$S' = \{1(5)0, \mathbf{7(8)0}\}$, $Q = \{\mathbf{2(15)7}, 3(20)1, 4(9)0, \mathbf{5(14)7}\}$
$S' = \{1(5)0, \mathbf{4(9)0}, 7(8)0\}$, $Q = \{2(15)7, 3(20)1, \mathbf{5(13)4}, \mathbf{6(29)4}\}$
$S' = \{1(5)0, 4(9)0, \mathbf{5(13)4}, 7(8)0\}$, $Q = \{\mathbf{2(14)5}, 3(20)1, \mathbf{6(26)5}\}$
$S' = \{1(5)0, \mathbf{2(14)5}, 4(9)0, 5(13)4, 7(8)0\}$, $Q = \{\mathbf{3(17)2}, \mathbf{6(25)2}\}$
$S' = \{1(5)0, 2(14)5, \mathbf{3(17)2}, 4(9)0, 5(13)4, 7(8)0\}$, $Q = \{6(25)2\}$

Implementing Dijkstra's algorithm

Compute for every v distance $d[v]$ from s and parent $\pi[v]$.
To find $v^* = \arg\min_{v \in Q} d'(v)$ efficiently, use a priority queue Q of pairs (δ, v) with value δ.

```
Dijkstra($G, s$):
    **foreach** $v$ **do**
        $d[v] \leftarrow \infty$; $done[v] \leftarrow$ False
    $d[s] \leftarrow 0$
    Set up priority queue $Q$ with $(d[s], s) \in Q$
    **while** $Q$ is not empty **do**
        $(\delta, u) \leftarrow removeMin(Q)$
        **if** **not** $done[u]$ **then**
            $done[u] \leftarrow$ True
            **for** out-neighbors $v$ of $u$ **do**
                $\gamma \leftarrow d[u] + l(u, v)$
                **if** $\gamma < d[v]$ **then**
                    $\pi[v] \leftarrow u$; $d[v] \leftarrow \gamma$; add $(\gamma, v)$ to $Q$
```

- In this implementation, a vertex v gets onto Q in a new copy every time v is reached on a new edge. But Q is still at most as large as the number of edges, so $\log |Q| \leq 2 \log n$.
- In a version with just one copy of each vertex on the heap, we need a heap implementation allowing the removal of items (in some other applications they are necessary). Most implementations in the usual libraries don't have this function.

Minimum spanning trees

- 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.
- Negative weights are also allowed: this allows to ask for the heaviest tree, too.
- Generic algorithm: Repeatedly, add some 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.

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

Prim's algorithm

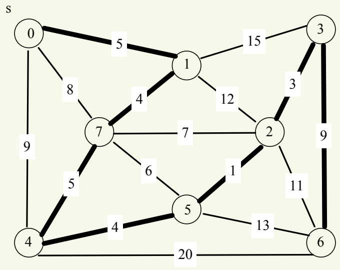
Keep adding a lightest edge adjacent to the already constructed tree.

Implement this similarly to Dijkstra's algorithm: maintain a set Q of neighbors v adjacent to the tree S, organize it as a priority queue.

The main difference to Dijkstra's algorithm 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.

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

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

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

$S' = \{1(5)0, 7(4)1, \mathbf{4(5)7}\}$,

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

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

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

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

Solved exercises

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.

Implementing Prim's algorithm

Compute for every v a parent $\pi[v]$.
(Meaning of $d[v]$ is different from Dijkstra.)

```
Prim(G, s):
    foreach v do
        done[v] ← False; d[v] ← ∞
    d[s] ← −∞                                          // 0 for Dijkstra.
    Set up priority queue Q with (d[s], s) ∈ Q
    while Q is not empty do
        (δ, u) ← removeMin(Q)
        if not done[u] then
            done[u] ← True
            for neighbors v of u do
                if not done[v] then
                    γ = l(u, v)     // d[u] + l(u, v) for Dijkstra.
                    if γ < d[v] then
                        π[v] ← u; d[v] ← γ; add (γ, v) to Q
```

Kruskal's algorithm

Another minimum spanning tree algorithm based on the same theorem:

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

Efficient implementation needs a smart way to track the components of the graph; see below.

Clustering

Example Netflix has a database of a lot of customers i, $i = 1, \dots, n$.. Each has a set F_i of films they have seen. For customers i, j, let

$$l(i, j) = |F_i \Delta F_j| = |F_i \setminus F_j| + |F_j \setminus F_i|$$

be the number of films that one of them has seen and the other one has not. This “distance” is an indicator of their difference in tastes. Netflix wants to classify its customers: create clusters of them. One way is (not the best, too rigid), for a given δ, to break them up into the largest number of subsets with the property that customers i, j in different groups are at distance $l(i, j) > \delta$ from each other.

Kruskal’s algorithm solves the clustering problem: just stop adding edges when they become larger than δ.

Union-find

- To implement Kruskal's algorithm, we need to keep track of a family of disjoint sets (in case of the Kruskal algorithm, the trees in the forest).
- Now concentrate on this task, abstracting away from the spanning tree problem.
- The Union-Find abstract data type has a family of disjoint sets $S_1, \dots, S_k$, $S_i \subset \{1, \dots, n\}$. Two operations:
  - *Find-Set*(x): Given an element $x \in \{1, \dots, n\}$, find S_i with $x \in S_i$.
  - *Union*(i, j): given i, j, replace sets S_i, S_j with their union $S_i \cup S_j$.

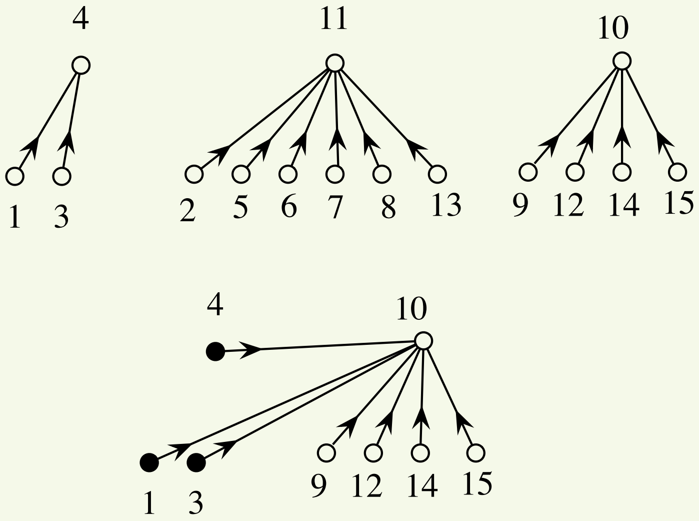
Implementing Union-Find

A common idea: represent each set S_i by some root element $u_i \in S_i$. We will write $S_i = S(u_i)$.
How to answer *Find-Set* questions?

First idea

- For each element v let $rep[v] = u$ where $v \in S(u)$.
- Also, for each set $S(u)$ a list (say, linked) of its elements.

- $\mathit{Find\text{-}Set}(x)$] returns $rep[x]$. Fast.
- $\mathit{Union}(u, v)$: for example $rep[u] \leftarrow v$.
  Add the list of $S(v)$ to the list of $S(u)$.
  $rep[w] \leftarrow u$ for all $w \in S(v)$. This part is slow.

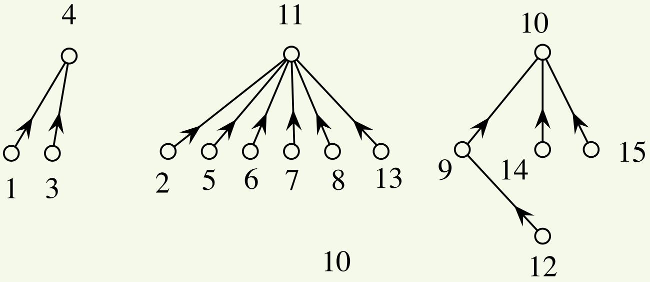


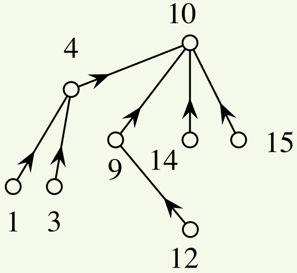
Union$(4, 10)$: the black elements *w* had *rep*$[w]$ reassigned.

Second idea A tree $T(u)$ with root u for each set $S(u)$.

- $\mathit{parent}[v]$ belongs to the set of v, $\mathit{parent}[u] = u$ for root u.
- $\mathit{rank}[u]$: (bound on) the height of the tree under u.

- $\mathit{Union}(u, v)$:
  if $\mathit{rank}[u] \le \mathit{rank}[v]$ then $\mathit{parent}[u] \leftarrow v$.
  If $\mathit{rank}[u] = \mathit{rank}[v]$ then $\mathit{rank}[v] \leftarrow \mathit{rank}[v] + 1$.
  Fast.
- $\mathit{Find\text{-}Set}(x)$ is slower: follow parents to the root.
  But each path has length $\le \log n$. Indeed, if a vertex has rank > 1 it has at least 2 children.





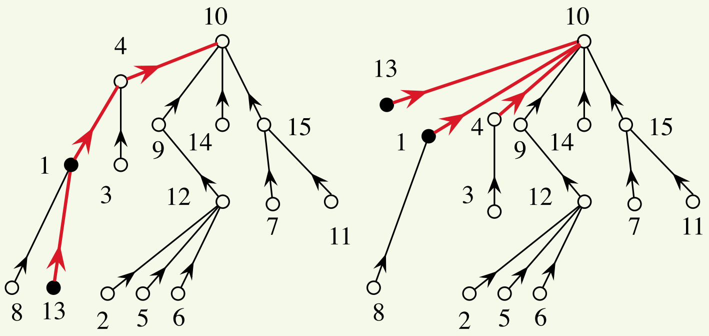
$Union(4, 10)$

Path compression

The $\log n$ cost per *Find-Set*() is not bad, but can be improved by the following trick.

- With each *Find-Set*(), set the parent of all passed elements to the root.

See the CLRS book for how much this saves you in in amortized cost: for n operations, the cost is $O(n\alpha(n))$ where $\alpha(n)$ grows much slower than even $\log n$.



Find-Set(13)

```
Find-Set($x$):                                      // With path compression
    if $x \neq parent[x]$ then
        $parent[x] \leftarrow$ Find-Set($parent[x]$)
    return $parent[x]$

Union($x, y$):
    $x' \leftarrow$ Find-Set($x$); $y' \leftarrow$ Find-Set($y$)
    if $rank[x'] > rank[y']$ then
        $parent[y'] \leftarrow x'$
    else
        $parent[x'] \leftarrow y'$
        if $rank[x'] = rank[y']$ then
            $rank[y'] \leftarrow rank[y'] + 1$
```

Implementing the Kruskal algorithm

```
Kruskal(G):
// Outputs the edges of a minimum spanning tree.

    Sort the edges of $G$ by weight into a list $E$
    foreach $v$ do
        $parent[v] = v$; $rank[v] = 0$
    while $E$ is not empty do
        remove the smallest-weight edge $(u, v)$ from $E$
        if $Find\text{-}Set(u) \neq Find\text{-}Set(v)$ then
            output edge $(u, v)$
            $Union(u, v)$
```

Note The forest of the Kruskal algorithm consists of trees, and the Union-Find data structure consists of trees on the same sets. But these trees have nothing to do with each other!

Divide and conquer

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.

Merge sort

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 $A[q+1\,..\,r]$.
    $n_1 \leftarrow q - p + 1$; $n_2 \leftarrow r - q$
    create array $L[1\,..\,n_1+1]$ and $R[1\,..\,n_2+1]$
    for $i \leftarrow 1$ to $n_1$ do $L[i] \leftarrow A[p+i-1]$
    for $j \leftarrow 1$ to $n_2$ do $R[j] \leftarrow A[q+j]$
    $L[n_1+1] \leftarrow \infty$; $R[n_2+1] \leftarrow \infty$
    $i \leftarrow 1$, $j \leftarrow 1$
    for $k \leftarrow p$ to $r$ do
        if $L[i] \le R[j]$ then $A[k] \leftarrow L[j]$; $i{+}{+}$
        else $A[k] \leftarrow R[j]$; $j{+}{+}$
```

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**
        $q \leftarrow \lfloor (p+r)/2 \rfloor$
        Merge-Sort($A, p, q$)
        Merge-Sort($A, q+1, r$)
        Merge($A, p, q, r$)
```

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

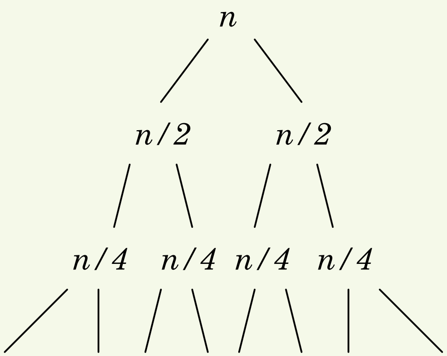
$$T(n) \leq \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{otherwise.} \end{cases}$$

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).

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

Recursion tree



Work on top level

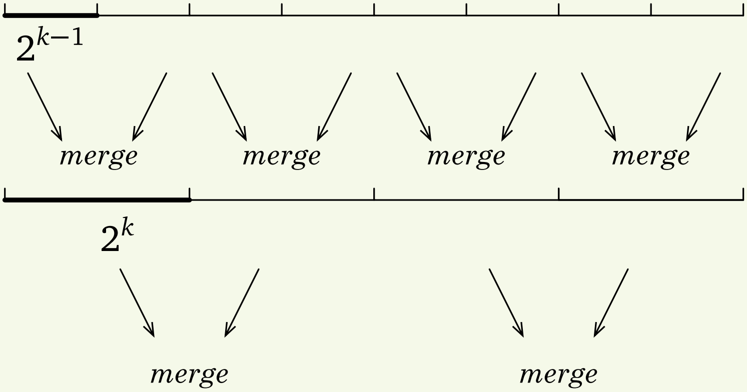
Total work on level 1

Total work on level 2

...

Nonrecursive version

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



Closest pair of points

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

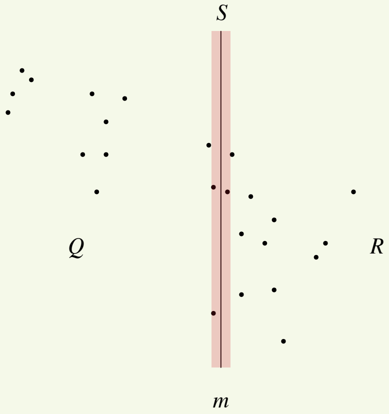
Question Given a set P of n points in the plane, find a pair $p, q \in P$ with $d(p, q)$ minimal.

A brute-force solution takes $O(n^2)$ steps. Can we do better?

For simplicity, let $n = 2^k$.

- Let P_x = the points of P, 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).

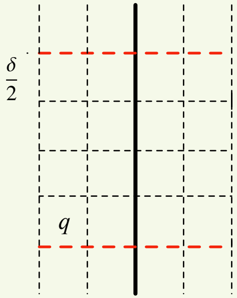


- Let $m =$ the maximal x coordinate of points in Q.
  Let $S =$ the set of points in P whose x coordinate differs from m by at most δ.
- If (q, r) is a boundary case then both q and r are in S. How to find them?
- Let S_y be the list in which the set S is sorted by the y coordinate. The crucial observation:

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: indeed, for every $q \in S_y$ we only have to check the next 15 ones to see if they can form a boundary pair (q, r).

Proof of the lemma. Vertical line: x coordinate m.



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

Running time analysis

- Sort in both the x and the y directions, to get P_x and P_y. No more sorting needed during the recursive calls.
- Let $F(n)$ be the running time of the algorithm after this sorting.

$$\begin{aligned} F(2) &= 1, \\ F(n) &\leq 2F(n/2) + cn. \end{aligned}$$

Same recursion as for merge sort, same resolution: total running time is $O(n \log n)$.

Integer multiplication

Some applications (say in cryptography) require the multiplication of large integers. We will use binary notation:

$$(10110)_2 = 0 \cdot 1 + 1 \cdot 2 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4,$$
$$(x_{n-1}x_{n-2} \dots x_1 x_0)_2 = x_0 + x_1 \cdot 2 + \dots + x_{n-1}2^{n-1}.$$

Multiplying two numbers:

$$XY = (x_0 + x_1 \cdot 2 + \dots + x_{n-1} \cdot 2^{n-1})(y_0 + y_1 \cdot 2 + \dots + y_{n-1} \cdot 2^{n-1}).$$

School method: needs at least to compute all products $x_i y_j$, so costs $\Omega(n^2)$. As old as the positional number system (thousands of years). Was first improved only around 1960.

Divide and conquer

$$\begin{aligned}
X_0 &= (x_{n/2-1} \dots x_1 x_0)_2 = x_0 + 2x_1 + \dots + 2^{n/2-1} x_{n/2-1},\\
X_1 &= (x_{n-1} x_{n-2} \dots x_{n/2})_2,\\
X &= X_0 + 2^{n/2} X_1, \quad Y = Y_0 + 2^{n/2} Y_1,\\
XY &= X_0 Y_0 + 2^{n/2}(X_0 Y_1 + X_1 Y_0) + 2^n X_1 Y_1.
\end{aligned}$$

- Gives a recursive inequality for running time:

  $$T(n) \leq 4T(n/2) + cn,$$

  since we computed $X_i Y_j$ and then made some additions.
- Does not save anything: Even the stronger inequality $T(n) \leq 4T(n/2)$ allows $T(n) = n^2$.

The Karatsuba trick

We don't need $X_0 Y_1$ and $X_1 Y_0$ separately, only their sum. This appears as part of a single product

$$(X_0 - X_1)(Y_0 - Y_1) = X_0 Y_0 + X_1 Y_1 - (X_0 Y_1 + X_1 Y_0)$$

of maximum n-bit numbers. Its other parts are already computed!
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** $p + 2^{n/2}(p + q - r) + 2^n q$
```

Running time analysis

Again assuming $n = 2^k$:

$$M(n) \le cn + 3M(n/2) \tag{3}$$

$$M(n/2) \le cn/2 + 3M(n/4) \tag{4}$$

$$M(n) \le cn + (3/2)cn + 9M(n/4) \quad \text{where we substituted (4) into (3)}$$

$$M(n) \le cn + (3/2)cn + (3/2)^2 cn + 27M(n/8),$$

$$\dots$$

$$M(n) \le cn(1 + (3/2) + (3/2)^2 + \cdots + (3/2)^{k-1}) + 3^k M(\overset{\overset{=1}{\downarrow}}{n/2^k}),$$

$$1 + (3/2) + (3/2)^2 + \cdots + (3/2)^{k-1}$$

$$\le (3/2)^{k-1}(1 + (2/3) + (2/3)^2 + \cdots) = (3/2)^{k-1} \cdot 3 = 3^k/2^{k-1},$$

$$M(n) \le c \cdot 2^k \cdot 3^k/2^{k-1} + 3^k = (2c+1)3^k,$$

$$3^k = 2^{k\log 3} = n^{\log 3}.$$

(Sum of geometric series: for $q < 1$, $1 + q + q^2 + \ldots = \frac{1}{1-q}$.)

- This algorithm is faster than the school method: $O(n^{\log 3}) \approx O(n^{1.585})$ in place of $O(n^2)$.
- Only the first step towards faster multiplication: the current best algorithm has complexity $O(n \log n)$, faster than $O(n^{\lambda})$ for any $\lambda > 1$.

Applications

- Same algorithm works for base 10; it applies even when X, Y are polynomials:

$$X(z) = x_0 + x_1 z + x_2 z^2 + \cdots + x_{n-1} z^{n-1}.$$

- The coefficient of z^k in the product $X(z)Y(z)$ is

$$x_0 y_k + x_1 y_{k-1} + \cdots + x_k y_0.$$

  Called the convolution of $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$.
- Take two independent random variables, A, B with integer values: A takes value $i \geq 0$ with probability p_i, B takes it with with probability q_i. Then $A + B$ takes value k with probability

$$p_0 q_k + p_1 q_{k-1} + \cdots + p_k q_1.$$

  So this is the convolution of $(p_0, p_1, \ldots)$ and $(q_0, q_1, \ldots)$, and can now be computed faster!

Solved exercises

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

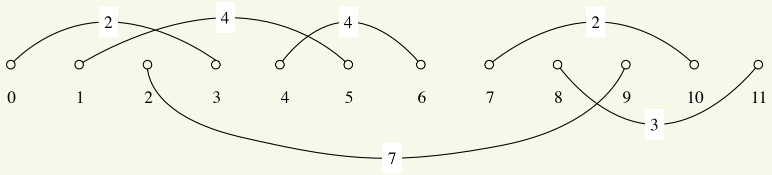
Dynamic programming

Weighted interval scheduling

Interval scheduling problem, but 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 Task $T_i = s_i(v_i)f_i$: starting time s, endtime f, value v.

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



Earliest-deadline-first would choose T_1, T_3, T_5. But its total value is smaller than that of T_1, T_3, T_6.

Recursion idea Compute the optimum not just for the whole system, but for many subsystems as well: $\mathrm{OPT}(i)$ = the maximum total value of all possible selections from tasks $1, 2, \ldots, i$.

Order by finish time again: $f_1 \le \cdots \le f_n$. The last task before i disjoint from it is

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

Recursion:

$$\begin{aligned}
\mathrm{OPT}(i) = \max \text{ of the two quantities below}:& \\
&\mathrm{OPT}(i-1), && \text{passing } T_i \\
&v_i + \mathrm{OPT}(p(i)). && \text{including } T_i.
\end{aligned}$$

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

Use a table

- Just calling the recursive formula is unwise, leads to exponential blowup.
  Example: with, say, $p(i) = i - 2$ for each i:

$$\begin{array}{cccccccc}
n & & & & & & & \\
n-1 & & & & n-2 & & & \\
n-2 & & n-3 & & n-3 & & n-4 & \\
n-3 & n-4 & n-4 & n-5 & n-4 & n-5 & n-5 & n-6
\end{array}$$

- Save the computed values $\mathrm{OPT}(1), \mathrm{OPT}(2), \dots, \mathrm{OPT}(n)$ in some array M. Instead of the recursive calls, just refer to M:

```
for $i = 0$ to $n$ do $M[n] \leftarrow 0$
for $i = 1$ to $n$ do
    $M[i] \leftarrow \max(M(i-1), v_i + M[p(i)])$
```

Memoization

General idea: memoization, (or caching).

- Applies when
  - Some recursive calls would be repeated many times.
  - There is enough memory to store the results for all of them.
- Store all these results in a table (in programming, a “static”, or “global” array).
- In each call, first check whether the result is already in the table: if yes, just return it, else compute it and store it.

```
RWS($i$) :                              // Recursive weighted scheduling
    if $M[i]$ is not defined then
        $M[i] \leftarrow \max(\mathrm{RWS}(i-1), v_i + \mathrm{RWS}(p(i)))$
```

Running time found by tracing: at most twice that of the non-recursive version above.

How to find which tasks are selected? Backwards (like following parents in Dijkstra's algorithm), collecting tasks in a set S:

```
$i \leftarrow n$; $S = \emptyset$
while $i > 0$ do
    if $M[i] = M[i-1]$ then $i$--
    else $S \leftarrow S \cup \{i\}$; $i \leftarrow p(i)$
```

Other examples

- 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.)

Sequence alignment

- A problem with many applications: for example the `diff` program, biology, voice recognition.
- Two sequences of symbols from some alphabet S:
  $X = (x_1, \dots, x_m)$, and $Y = (y_1, \dots, y_n)$.
  An alignment is given by two sets of indices
  $$1 \le i_1 < i_2 < \cdots < i_k \le m \text{ and } 1 \le j_1 < j_2 < \cdots < j_k \le n$$
  such that x_{i_p} is matched to y_{j_p}, for all $p = 1, \dots, k$.
- Example: match the words “kolor” and “colour”. We could match the bold characters: **kolor** with **colo**u**r**.

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

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

Example: Let $\delta = 1$, and $d(r,s) = 0$ if $r = s$ and 2 otherwise.
In matching **kolor** with **colour**, the total penalty is 1 + 2, since there is one unmatched symbol “u” and one mismatched pair (k,c).

A larger example

$$X = (r, p, q, q, r, a, x, y, b, b, x, y, a, b, w, v),$$
$$Y = (p, q, q, r, a, x, b, y, y, u, v, w, v).$$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
r	p	q	q	r	a	x	y	b	b	x	y	a	b	w	v
	p	q	q	r	a	x			b	y	y	u	v	w	v
	1	2	3	4	5	6			7	8	9	10	11	12	13

$$\mathbf{i} = 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 10 \quad 11 \quad 12 \quad 15 \quad 16$$
$$\mathbf{j} = 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 12 \quad 13$$

Black symbols matched with no penalty. Red symbols matched, penalty 1. Blue symbols passed, penalty 1 for each.
Total penalty: 8.

Dynamic programming algorithm

Let $A[i,j]$ be the minimum penalty for matching $x_1 \dots x_i$ with $y_1 \dots y_j$. Of course, $A[0,0] = 0$, $A[0,j] = j\delta$, $A[i,0] = i\delta$ for $i, j > 0$. Recursion:

$$
\begin{aligned}
A[i,j] = \min &\text{ of the three quantities below :} \\
&d(x_i, y_j) + A[i-1, j-1], && \text{matching } x_i, y_j \\
&\delta + A[i-1, j], && \text{passing } x_i \\
&\delta + A[i, j-1]. && \text{passing } y_j
\end{aligned}
$$

This allows to fill in the array $A[i,j]$ for example row-by-row.

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

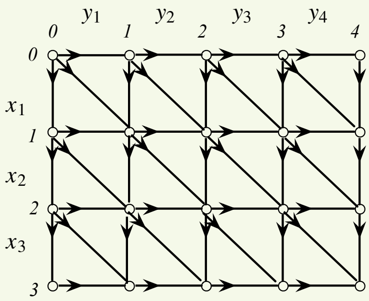
$$\begin{array}{c|ccccccccccccc}
 & & c & & o & & l & & o & & u & & r & \\
\hline
 & 0 & \leftarrow & 1 & & 2 & & 3 & & 4 & & 5 & & 6 \\
k & \uparrow & \nwarrow & \uparrow & & & & & & & & & & \\
 & 1 & \leftarrow & 2 & & 3 & & 4 & & 5 & & 6 & & 7 \\
o & & & & \nwarrow & & & & & & & & & \\
 & 2 & & 3 & & 2 & & 3 & & 4 & & 5 & & 6 \\
l & & & & & & \nwarrow & & & & & & & \\
 & 3 & & 4 & & 3 & & 2 & & 3 & & 4 & & 5 \\
o & & & & & & & & \nwarrow & & & & & \\
 & 4 & & 5 & & 4 & & 3 & & 2 & \leftarrow & 3 & & 4 \\
r & & & & & & & & & & & & \nwarrow & \\
 & 5 & & 6 & & 5 & & 4 & & 3 & & 4 & & 3
\end{array}$$

$$
\begin{array}{c|ccccccccccccc}
 & & c & & o & & l & & o & & u & & r & \\
\hline
 & 0 & \leftarrow & 1 & & 2 & & 3 & & 4 & & 5 & & 6 \\
k & \uparrow & \nwarrow & \uparrow & & & & & & & & & & \\
 & 1 & \leftarrow & 2 & & 3 & & 4 & & 5 & & 6 & & 7 \\
o & & & & \nwarrow & & & & & & & & & \\
 & 2 & & 3 & & 2 & & 3 & & 4 & & 5 & & 6 \\
l & & & & & & \nwarrow & & & & & & & \\
 & 3 & & 4 & & 3 & & 2 & & 3 & & 4 & & 5 \\
o & & & & & & & & \nwarrow & & & & & \\
 & 4 & & 5 & & 4 & & 3 & & 2 & \leftarrow & 3 & & 4 \\
r & & & & & & & & & & & & \nwarrow & \\
 & 5 & & 6 & & 5 & & 4 & & 3 & & 4 & & 3
\end{array}
$$

(Choice at position $(1,1)$.) Find the alignment S going backwards:

```
$i, j \leftarrow m, n$; $S = \emptyset$
**while** $i > 0, j > 0$ **do**
    $(i', j') \leftarrow$ one of $(i-1, j-1), (i, j-1), (i-1, j)$
    which gives the minimum $A[i, j]$ above.
    **if** $(i', j') = (i-1, j-1)$ **then** $S \leftarrow S \cup (i, j)$
    $(i, j) \leftarrow (i', j')$
```

Graph representation



For $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$. Points (i, j) where i represents the position between x_i and x_{i+1}.

- Horizontal and vertical edges have length δ.
- Diagonal edge ending in (i, 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.)
- Find the alignment going backwards using $A[i, j]$, or record the parents while computing the distances $A[i, j]$ (as in Dijkstra).

Linear space via divide and conquer

(Not covered in Spring 2020)

- Computing $A[\cdot,\cdot]$ can be done in linear space, 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.

Space-saving alignment algorithm

Let a linear-space algorithm AlignmentCosts(X, Y) return just the one-dimensional array $A[1:m, n]$.

```
DC-Align$(X, Y)$ :                    // returns a shortest path $P$
    **if** $m \leq 2$ or $n \leq 2$ **then** compute directly
    $f[1:m] \leftarrow$ AlignmentCosts$(X[1:m], Y[1:n/2])$
    $g[1:m] \leftarrow$ AlignmentCosts$(X[1:m], Y[n/2+1:n])$
    $q \leftarrow \arg\min f[q] + g[q]$
    $P_1 \leftarrow$ DC-Align$(X[1:q], Y[1:n/2])$
    $P_2 \leftarrow$ DC-Align$(X[q+1:n], Y[n/2+1:n])$
    **return** $P_1 + (q, n/2) + P_2$
```

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

Analysis

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

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

We took the worst possible q.
Let us guess that the total time is still $\le dmn$ for some constant d.
Find out how large must d be for the proof to work.
The base case works if $d \ge c/2$.
By the above inequality and the inductive assumption

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

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

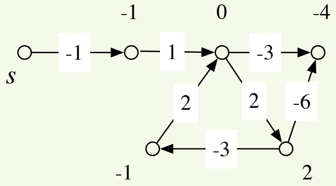
Shortest paths with possible negative costs

We have seen a shortest (smallest-cost) path algorithm already (Dijkstra), but it could not handle negative costs.

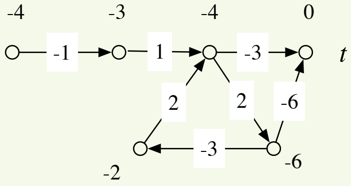
Negative cycle: Consider a merchant: on certain edges he spends money on traveling and buying merchandise, on others he makes profit by selling (negative cost). A cycle is profitable if the sum of edge costs along it is negative.

- If there is such a cycle then going around it repeatedly we can drive the cost towards $-\infty$.
- If there is no such cycle, then it is sufficient to look for paths of length $\leq n-1$.

Distances from s:



Distances to t:



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.

Improving space-efficiency

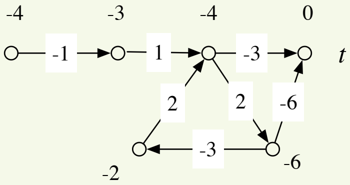
We can use the space of $M[i-1, v]$ to store $M[i, v]$, and call it simply $M[v]$. The number i is used now just as a counter of repetition.
first$[u]$ stores the first node of the current smallest-cost path from u (like a parent link).

```
$M[t] \leftarrow 0$
**for** $v \neq t$ **do** $M[v] \leftarrow \infty$
**for** $i = 1$ **to** $n$ **do**
    $didImprove \leftarrow$ False
    **for** $u \in V$ **do**
        **for** $v : (u, v) \in E$ **do**
            **if** $c_{uv} + M[v] < M[u]$ **then**
                $M[u] \leftarrow c_{uv} + M[v]$
                first$[u] \leftarrow v$
                $didImprove \leftarrow$ True
    **if not** $didImprove$ **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.
- Computing the shortest path $s = v_0, v_1, \dots, v_k = t$:

$$v_{i+1} = \arg\min_u w(v_i, u) + M[u].$$

Then of course $M[v_i] = w(v_i, v_{i+1}) + M[v_{i+1}]$.



The knapsack problem

Going to the market, to sell some items in my inventory. My knapsack has volume b.

- Given: volumes $b \geq a_1, \dots, a_n > 0$ of the items $1, 2, \dots, n$, and their integer values $v_1 \geq \cdots \geq v_n > 0$.
- Find a subset $i_1 < \cdots < i_k$ of them fitting into the knapsack: $a_{i_1} + \cdots + a_{i_k} \leq b$.
  Maximize the sum of their values $v_{i_1} + \cdots + v_{i_k}$.
- Expressed differently: $x_i \in \{0, 1\}$, where $x_i = 1$ means picking item i.

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

Special cases

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 $(a_1 + \cdots + a_n)/2$.

- The solution given here differs from the one in Kleinberg-Tardos; helps later with the approximation algorithm.
- Compute for each integer p the smallest volume

$$m_n(p)$$

  that gives value $\geq p$. (Upper bound on the total value: $V = v_1 + \cdots + v_n$.)
- The optimum is $\max\{\, p : m_n(p) \leq b \,\}$.

Dynamic programming solution

Subproblem using only the first k items:

$$m_k(p) = \min\{\, a_1x_1 + \cdots + a_kx_k \leq b : v_1x_1 + \cdots + v_kx_k \geq p\,\}.$$

If the set is empty the minimum is ∞.
Memoization: array $m[k,p] = m_k(p)$. Notation $|x|^+ = \max(x,0)$.

```
for $k = 0$ to $n$ do $m[k,0] \leftarrow 0$
for $p = 0$ to $V$ do
    if $p > 0$ then $m[0,p] = \infty$
    for $k = 1$ to $n$ do
        $m[k,p] \leftarrow \min(m[k-1,p], a_k + m[k-1, |p - v_k|^+])$
```

The min decides whether to include item k or not.

Another application: money changer problem. Produce the sum b using smallest number of coins of denominations $a_1, \dots, a_n$ (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.

Example

Let $\{v_1, \dots, v_5\} = \{1, 2, 4, 6, 10\}$, $a_i = v_i + 1$, $b = 13$.
Array $m[k, p]$. Recall

$$m[k,p] \leftarrow \min(m[k-1,p], a_k + m[k-1, |p - v_k|^+]).$$

The empty spaces in the array below have value ∞ (no items).

$k\backslash p$	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	11	11	13	14	

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

Is the running time polynomial?

Assume 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, $v_1, \ldots, v_n$:

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

  Assume $\log v_i, \log a_i \leq m$, then $L \leq (m+1)(2n+1)$.
- We compute a table of size $Vn > 2^m n$.
  In case $m = n$ this is $Vn \geq 2^m m$, exponential in input size L.

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, our algorithm can be adapted to approximate the optimum to within a factor of $1 + \varepsilon$, in time polynomial in (the input size and) $1/\varepsilon$.
  See later in the course.

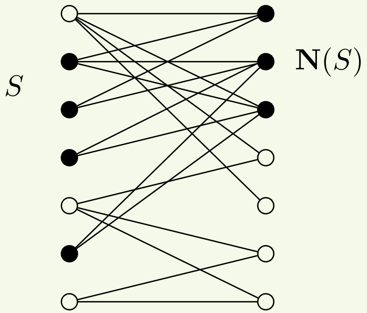
Matchings

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.

- Bipartite graph: left set A (workers), right set B (jobs).
- Matching, perfect matching.

Example with no perfect matching:



For $S \subseteq A$ let

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

be the set of all neighbors of the nodes of A. Perfect matching clearly needs the no bottleneck property:
For every $S \subseteq A$ we have $|\mathbf{N}(S)| \geq |S|$.

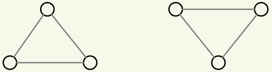
Example (No bottleneck) 6 tribes partition an island into hunting territories of 100 square miles each. 6 species of tortoise, with disjoint habitats of 100 square miles each.
Can each tribe pick a tortoise living on its territory, with different tribes choosing different totems?

No bottleneck here: the combined hunting area of any k tribes intersects with at least k tortoise habitats.

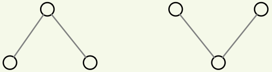
Example (No bottleneck) At a dance party, with 300 students, every boy knows 50 girls and every girl knows 50 boys. Can they all dance simultaneously so that only pairs who know each other dance with each other?

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

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



- Bipartiteness and positive degrees is insufficient.



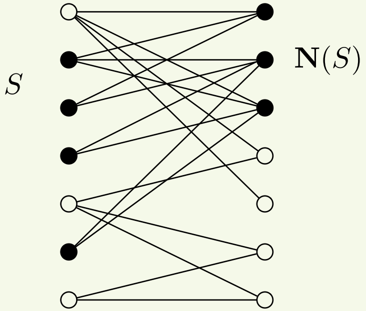
The no bottleneck property is also sufficient:

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 $|\mathbf{N}(S)| \geq |S|$.

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

Prove this as an exercise.



Flow networks

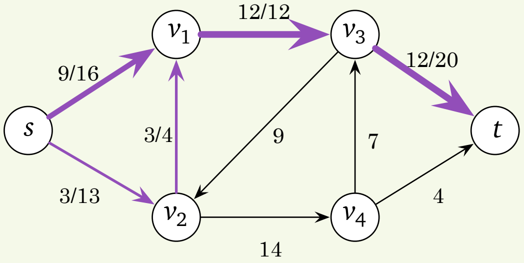
A model, generalizing the matching problem:

- Directed graph. Source s, sink t.
- Capacity: $c(u,v)$ on all edges (u,v) showing the amount of material that can flow from u to v. (We may have $c(u,v) \neq c(v,u)$.) If there is no edge (u,v) then set $c(u,v) = 0$.
- Flow function: $f(u,v) \leq c(u,v)$ on all edges (u,v) showing the amount of material going from u to v. No point of sending both from u to v and v to u: define $f'(u,v) = f(u,v) - f(v,u)$, then

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

  Our examples we will always show f' in place of f, and only in its positive direction.
- What goes into a vertex u different from s, t, also goes out:

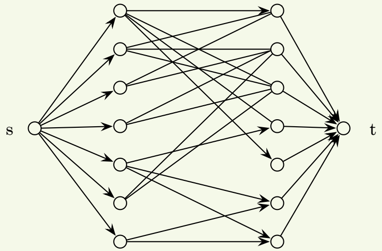
  $$\sum_v f'(u,v) = 0.$$



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

- Our goal is to maximize the value $|f| = \sum_v f(s,v)$.

Application to matching



- n points on left, n on right. Edges directed to right, with unit capacity from s to A and from B to t, and any capacity ≥ 1 (even ∞) between A and B.
- Perfect matching $\rightarrow$ flow of value n.
- Flow of value n $\rightarrow$ perfect matching?

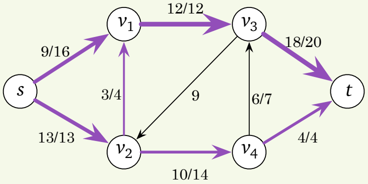
Example Recall the theorem saying that if the bipartite graph is regular then there is always a perfect matching.
I the dance party where every boy knows 50 girls and every girl knows 50 boys, in the corresponding flow network, we could just send a flow of 1/50 along every edge from boys to girls. This is maximum flow does not give us a matching.

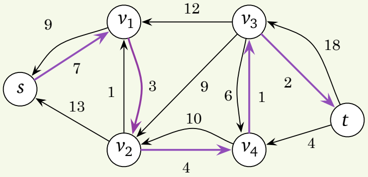
Fortunately (as will be seen), there is always an integer maximum flow.

Residual network, augmenting path

Given a flow f, residual capacity

$$\begin{aligned} c_f(u,v) &= c(u,v) - f'(u,v) \\ &= c(u,v) - f(u,v) + f(v,u). \end{aligned}$$



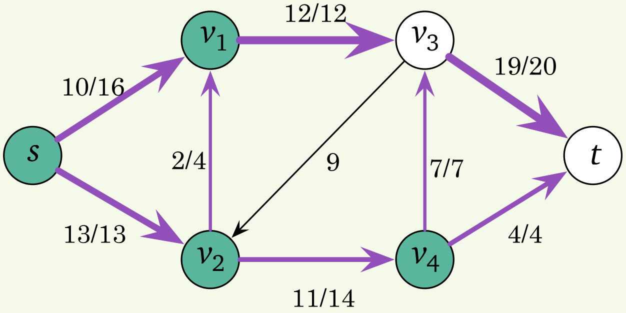


The residual network G_f may have edges (with positive capacity) that were not in the original network. An augmenting path is an s-t path in G_f (with some flow along it). (How does it change the original flow?)

G_f has the edges along which flow can still be sent.

If $f(u,v) > 0$ then sending from v to u means decreasing $f(u,v)$.

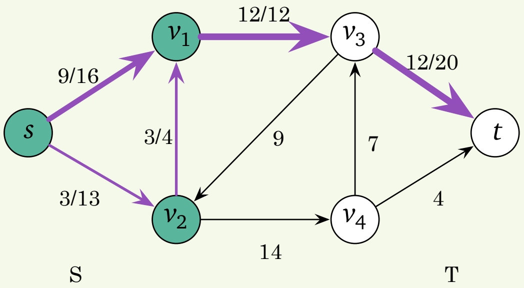
We obtained:



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

Cuts

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, $f(S,T) \le c(S,T)$.



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.

❶ $|f| = c(S,T)$ for some cut (S,T).

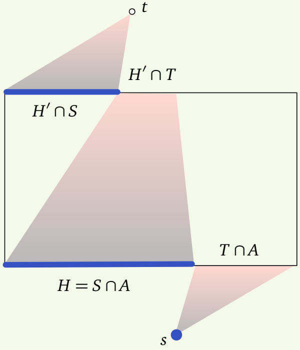
❷ f is a maximum flow.

❸ 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: ❶⇒❷ and ❷⇒❸ are obvious. The crucial step is ❸⇒❶. Given f with no augmenting paths, we construct (S,T): let S be the nodes reachable from s in the residual network G_f.

Proof of the marriage theorem

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 $H = S \cap A$, $H' = \mathbf{N}(H)$. We have $c(H, H' \cap B) \geq |H' \cap T|$ as each element of H' gets at least one edge from H.

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

$$|H| > |H'|.$$

Efficient flow algorithms

- 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. We will analyze it, but the following is more efficient:
- Goldberg: Push-relabel algorithm. Push as much pre-flow as the capacities bear, accumulating excess in the nodes. Push back the excesses. The process is regulated by a height function (labels).

Goldberg algorithm

- Goldberg's algorithm is "greedy": it will push so much along the edges that it will initially violate the flow property. Excess at point v: $e_f(v) = \sum_u f(u,v)$.
- The flow function is called an s-t pre-flow if all excesses other than at s are nonnegative. (It is a flow if $e_f(v) = 0$ for all $v \neq t$.)

Goldberg's algorithm:

- Will work with pre-flows without an augmenting path.
- Keeps adjusting the pre-flows until all excess is eliminated: with no augmenting path, we reach optimum.
- Integer labeling function $h(v)$ of nodes called the height.

Let n be the number of nodes.
Following properties will be maintained:

Source and sink heights $h(s) = n$, $h(t) = 0$.

Steepness bound If an edge (u, v) is in the residual network G_f then $h(u) \leq h(v) + 1$. (Thus if $h(v) - h(u) > 1$ the edge (u, v) is saturated.)

These imply that there is no augmenting path, since the height could not sink fast enough along it.

Initialization $h(s) \leftarrow n$, and $h(v) \leftarrow 0$ for all $v \neq s$.
$f(e) \leftarrow c(e)$ for every edge leaving s,
$f(e) \leftarrow 0$ for all other e.

Pushing or relabeling While there is a node $u \neq s, t$ with
$e_f(u) > 0$
(for additional efficiency choose one with the highest $h(u)$):

```
if there is an edge $(u, v)$ in the residual network $G_f$ with
 $h(u) > h(v)$ then
    push as much of the excess into $f(u, v)$ as $c(u, v)$ allows
else
    $h(u) \leftarrow h(u) + 1$
```

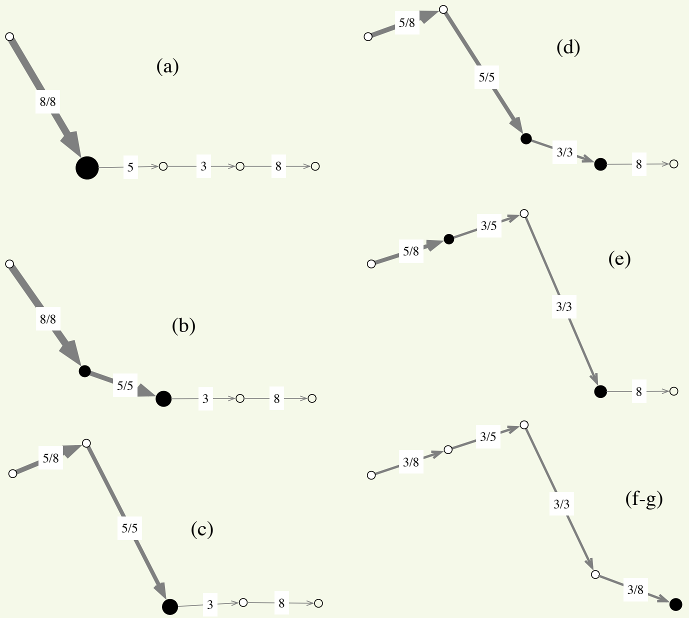
Maintains the source and sink heights and the steepness bound.

- Termination in $O(n^3)$ steps: proof below.

Example

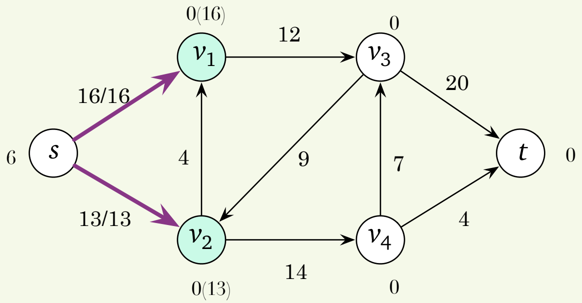
A path $a \xrightarrow{8} b \xrightarrow{5} c \xrightarrow{3} d \xrightarrow{8} e$.
Below, we will write 3(5) for a vertex with height 3 and excess 5.

$$
\begin{array}{ccccccccc}
a & \xrightarrow{8} & b & \xrightarrow{5} & c & \xrightarrow{3} & d & \xrightarrow{8} & e \\
5 & \xrightarrow{8/8} & 0(8) & \xrightarrow{5} & 0 & \xrightarrow{3} & 0 & \xrightarrow{8} & 0 \\
5 & \xrightarrow{8/8} & 1(3) & \xrightarrow{5/5} & 0(5) & \xrightarrow{3} & 0 & \xrightarrow{8} & 0 \\
5 & \xrightarrow{5/8} & 6 & \xrightarrow{5/5} & 0(5) & \xrightarrow{3} & 0 & \xrightarrow{8} & 0 \\
5 & \xrightarrow{5/8} & 6 & \xrightarrow{5/5} & 1(2) & \xrightarrow{3/3} & 0(3) & \xrightarrow{8} & 0 \\
5 & \xrightarrow{5/8} & 6(2) & \xrightarrow{3/5} & 7 & \xrightarrow{3/3} & 0(3) & \xrightarrow{8} & 0 \\
5 & \xrightarrow{3/8} & 6 & \xrightarrow{3/5} & 7 & \xrightarrow{3/3} & 0(3) & \xrightarrow{8} & 0 \\
5 & \xrightarrow{3/8} & 6 & \xrightarrow{3/5} & 7 & \xrightarrow{3/3} & 1 & \xrightarrow{3/8} & 0(3)
\end{array}
$$


(a)
8/8
5
3
8
(b)
8/8
5/5
3
8
(c)
5/8
5/5
3
8
(d)
5/8
5/5
3/3
8
(e)
5/8
3/5
3/3
8
(f-g)
3/8
3/5
3/3
3/8

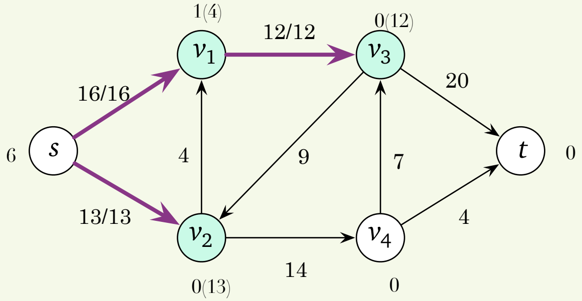
Push-relabel on the example

(Looks better in presentation mode.)



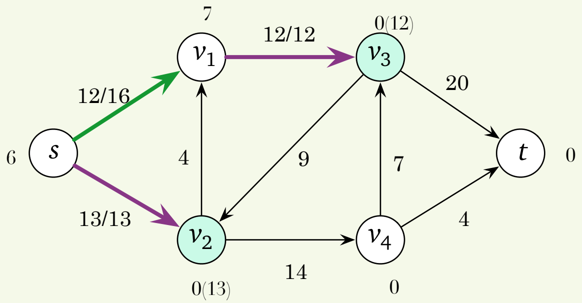
Push-relabel on the example

(Looks better in presentation mode.)



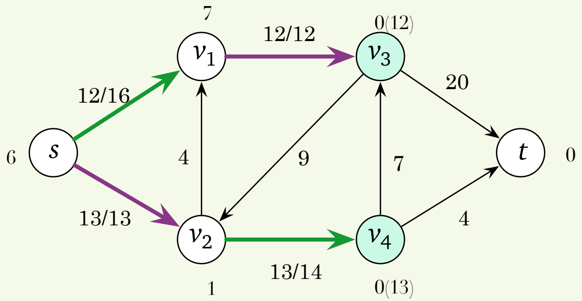
Push-relabel on the example

(Looks better in presentation mode.)



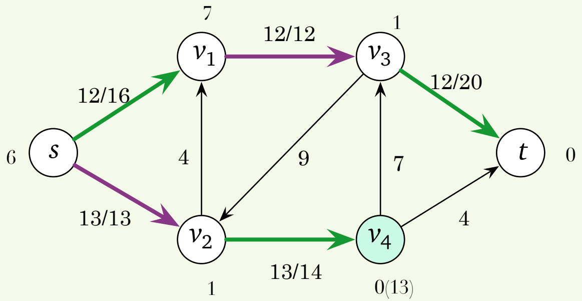
Push-relabel on the example

(Looks better in presentation mode.)



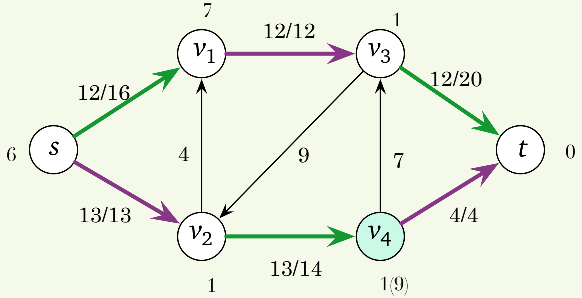
Push-relabel on the example

(Looks better in presentation mode.)



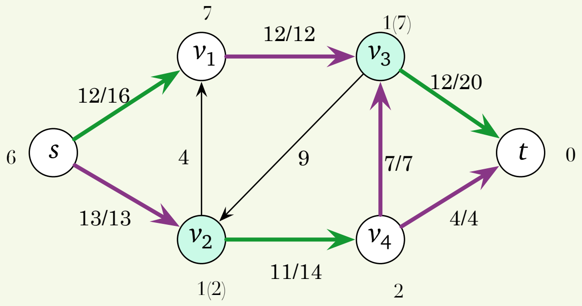
Push-relabel on the example

(Looks better in presentation mode.)



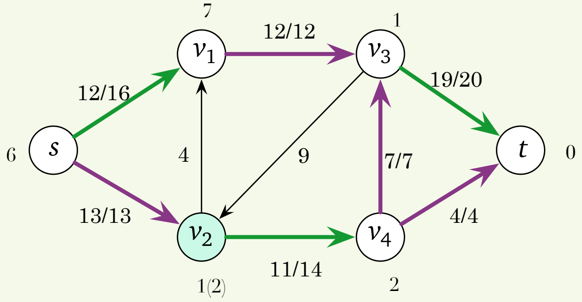
Push-relabel on the example

(Looks better in presentation mode.)



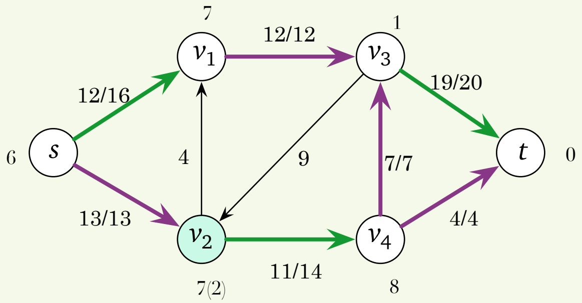
Push-relabel on the example

(Looks better in presentation mode.)



Push-relabel on the example

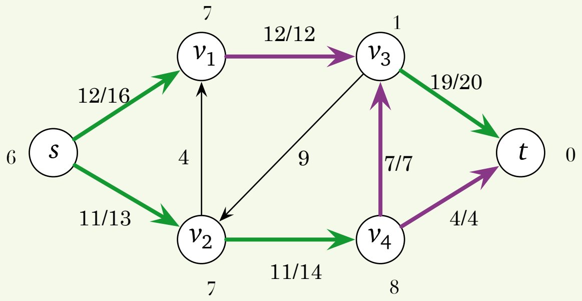
(Looks better in presentation mode.)



(After several back-and-forths between v_2 and v_4.)

Push-relabel on the example

(Looks better in presentation mode.)



Claim If $e_f(u) > 0$ then there is an augmenting path from u to s.

Indeed, let B be the set of vertices u with no augmenting path from u to s. No flow comes into B, but then no excess can be created in B, the sum of excesses is 0.

- The claim and the steepness bound imply $h(u) \leq h(s) + n - 1$, hence $h(u) \leq 2n - 1$. Hence the number of relabeling operations is bounded by $(n-2)(2n-1) \leq 2n^2$.

A push operation on edge (u,v) is saturating if it results in $f(u,v) = c(u,v)$.

Claim On each edge (u,v) there are at most n saturating pushes. So the total number of saturating pushes is $\leq 2mn$.

Indeed, between each two saturating pushes, the height must increase by at least 2 (at the push in the opposite direction $h(v) > h(u)$). Now recall the bound $2n-1$ on maximum height.

- We will bound by $4n^3$ the number of non-saturating pushes.

Claim

ⓐ At each value H of the maximum height of nodes with excess, from each node u of height H there is at most one non-saturating push.

ⓑ H changes at most $4n^2$ times.

To prove ⓐ: a nonsaturating push eliminates the excess of u, and u can get a new excess only from a neighbor with height above $h(u)$.
To prove ⓑ: H can also decrease; however, it can increase only by a relabel operation, of which there are $< 2n^2$.

- More sophisticated analysis shows a bound $O(n^2\sqrt{m})$ in place of $O(n^3)$.

Edmonds-Karp algorithm

(Not covered in Spring 2020)

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

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

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

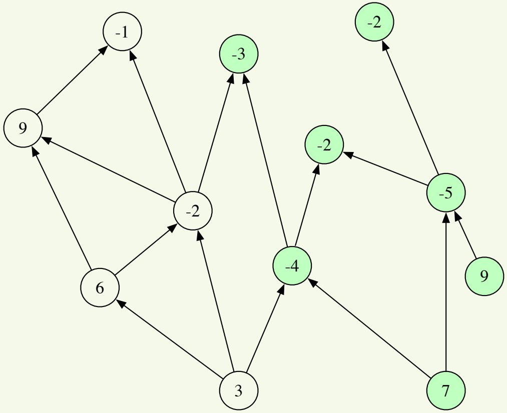
$$O(m^2 n).$$

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

Project selection

Network flow theory has many applications. Sometimes it takes ingenuity to apply it: see the following example.

- Set of possible projects to choose from: $P = \{1, 2, \ldots, n\}$.
  Project i brings profit p_i: positive or negative (then it is a cost).
- Dependencies: acyclic directed graph $G = (P, E)$.
  Edge (i, j): project i requires project j, too.
- Not a time ordering. Example:
  4: a wedding shower in which we could collect $p_4 = 1000$ dollars, but then we have to:
  - 5: buy food beforehand for $-p_5 = 200$ dollars, and
  - 8: clean up afterwards for $-p_8 = 250$ dollars.

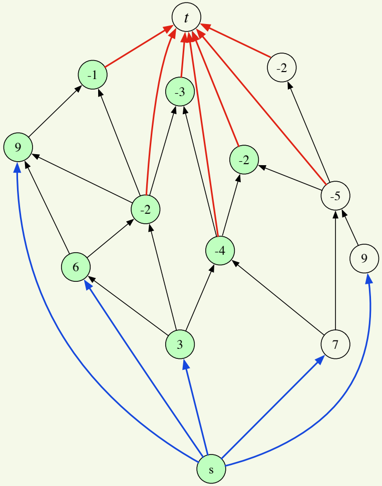


A subset $S \subseteq P$ is feasible if with every project in it, it contains all others on which it depends: $u \in S$, $(u, v) \in E \Rightarrow v \in S$.
Example: the set of green projects.
Goal: A feasible set S with maximum total profit $p(S) = \sum_{i \in S} p_i$.

The solution introduces a flow network. Add source and sink s, t. Capacities:

1. 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 $c(j, t) = -p_j$.

A cut S, T has finite capacity if and only if $S' = S \setminus \{s\}$ is feasible.



Cut with capacity $(7 + 9) + (1 + 2 + 3 + 4 + 2)$.

If S is feasible:

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

Obvious upper bound on the total profit: $C = \sum_{i : p_i > 0} p_i$.

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

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.

- To maximize the profit, find a minimum cut.

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.

Contention resolution

- Assume that n processes $P_1, \dots, P_n$ 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.

Events and their probabilities

- 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

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

  This becomes an equality for mutually exclusive events, $\mathcal{A}, \mathcal{B}$, that is if $\mathcal{A} \cap \mathcal{B} = \emptyset$ (see next).

- 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

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

Bayes's Theorem

Let $B_1, \ldots, B_n$ be mutually exclusive events of positive probability such that $\Pr(B_1) + \cdots + \Pr(B_n) = 1$. Then for an arbitrary event A the following holds:

$$\Pr(A) = \Pr(A \mid B_1)\Pr(B_1) + \cdots + \Pr(A \mid B_n)\Pr(B_n).$$

This fact is sometimes called Bayes's Theorem, or the theorem of total probability.

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

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

We will estimate this also using calculus.

Estimating $(1 + x)$ for products

Below we will explain the following two important inequalities from calculus:

$$e^{\frac{x}{1+x}} \le 1 + x \le e^x. \tag{6}$$

Now, let us just apply them. Writing $x = -1/n$ gives

$$e^{-\frac{1}{n-1}} \le 1 - \frac{1}{n} \le e^{-\frac{1}{n}}. \tag{7}$$

Hence

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

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

$$\Pr(\mathcal{F}(i,t)) = \prod_{r=1}^{t}(1 - \Pr(\mathcal{S}(i,r))) \le (1 - 1/en)^t.$$

Using the second inequality of (7):

$$\Pr(\mathcal{F}(i,t)) \le (1 - 1/en)^t \le e^{-t/en}.$$

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

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

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 $2en\ln n$ steps, then some rare disaster of probability $< 1/n$ happened.

Estimating $1+x$

The above application is typical for the calculations you encounter in probability theory. When many events are involved, the formulas needed to calculate the probabilities become complex. The tools of calculus are used to approximate them. Now let us prove the inequalities (6) used above.

The important inequality $1+x \le e^x$ comes from the convexity of the exponential function: the tangent line $y = 1+x$ of the curve $y = e^x$ is below it. Inverting the same inequality gives a bound from the other side:

$$\begin{aligned}
&\frac{1}{1+x} = 1 - \frac{x}{1+x} \le e^{-\frac{x}{1+x}}, \\
&1 + x \ge e^{\frac{x}{1+x}}.
\end{aligned}$$

Random variables

In a probability space, a random variable is some quantity X such that events of the kind $a \leq X < b$ 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
$\Pr\{Y = i\}$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Expected value

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 $\mathsf{E}X = p_1x_1 + p_2x_2 + \ldots$.

Examples

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

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

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

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

Sum theorem

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

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

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

$$\mathsf{E}(I_{\mathcal{A}} + I_{\mathcal{B}}) = \mathsf{E}I_{\mathcal{A}} + \mathsf{E}I_{\mathcal{B}} = \Pr(\mathcal{A}) + \Pr(\mathcal{B}) = 1/2 + 5/6 \text{ dimes}.$$

Chebyshev-Markov inequality

We want to estimate the probability that runtime is much larger than the expected value. The following theorem helps:

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

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

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

$$c\mu \cdot \Pr\{X > c\mu\} = \mathsf{E} X' \leq \mathsf{E} X = \mu.$$

Example: the probability that some randomized algorithm takes longer than $40n$ is less than $1/10$, since the expected time is bounded by $4n$. (There are much better estimates for this probability, but they use similar principles.)

Geometric random variable

- How many times do you have to toss a 6-sided die to have the number 2 come up? The probability that it comes up is 1/6 each time. The number of tosses needed is a random variable.
- In general, for some probability $p > 0$, consider repeated independent experiments in which the probability of success is always p. Let T be the (random) number of experiments needed until the first success. Then

  $$\Pr\{ T = k \} = p(1-p)^{k-1}.$$

  This is called the geometric random variable.
- Many applications need the expected value of T. It is

  $$\mathsf{E}\, T = \sum_{k=1}^{\infty} kp(1-p)^{k-1} = 1/p.$$

  So the expected number of throws of the die is 6.

Conditional expectation

Using conditional probabilities, one can also define the conditional expected value $\mathsf{E}[X \mid A]$ of some random variable X with respect to some event A of positive probability. The analogue of Bayes's Theorem holds for this case. Let $B_1, \dots, B_n$ be mutually exclusive events of positive probability such that $\Pr(B_1) + \cdots + \Pr(B_n) = 1$. Then for an arbitrary random variable X we have

$$\mathsf{E}\, X = \mathsf{E}[X \mid B_1]\Pr(B_1) + \cdots + \mathsf{E}[X \mid B_n]\Pr(B_n). \tag{8}$$

Example Toss a die. If 1 comes up (event E) then toss it until an even number comes up. Otherwise toss it until a 5 or 6 comes up. Let S be the number of tosses after the first one to finish.

$$\begin{aligned} \mathsf{E}\, S &= \mathsf{E}[S \mid E]\Pr(E) + \mathsf{E}[S \mid \neg E]\Pr(\neg E) \\ &= 2 \cdot (1/6) + 3 \cdot (5/6). \end{aligned}$$

Randomized caching

- A cache in a computer is a part of memory with very fast access time. It contains a limited number k of items of the same size.
- The runtime system faces a sequence of requests for memory items: $\sigma = s_1, s_2, \ldots, s_n$.
- If a memory item s is requested that is not in the cache then this is a cache miss. Then it takes much longer to bring s into the cache. Also, it will evict some other item from the cache.
- Next time when s is requested again, it can be accessed fast (provided it still has not also been evicted from the cache).
- Looking for a good eviction policy, minimizing the number of cache misses.

- In case the whole sequence σ of requests is known then there is an optimal policy: evict an item that will be requested farthest in the future. (Its optimality was proved in the greedy algorithms section of Kleinberg-Tardos.)
- This policy is offline: it must know all future requests in advance. Normally this sequence is not known in advance: we need an online policy, deciding only on the basis of the past, not the future.
- Example: evict the least recently used item (LRU).
- Every online policy will be good on some sequences and bad on others. How to compare them?

Idea: compare the performance to that of the ideal offline policy (evict the item requested farthest in the future). We call this number of cache misses the offline optimum.

Claim For every (deterministic) online policy there is a sequence σ that produces a number of cache misses k times larger than the offline optimum.

Indeed, let the sequence $\sigma = s_1 s_2, \ldots, s_n$ consist of repeated requests to items $1, 2, \ldots, k+1$.

- If s_{i+1} is made to be the item evicted at time i then every request results in a cache miss.
- But the farthest-in-future policy will can always answer at least $k-1$ requests before the next cache miss.

Here is a randomized eviction policy. We will prove that its expected number of cache misses is at most $O(\log k)$ times larger than the offline optimum.

- The policy will mark some of the items in the cache. It works as follows, when an item x is requested.

```
if $x$ is not in the cache and all items in the cache are marked
 then
    remove all marks
mark $x$
if $x$ is not in the cache then
    evict a random unmarked item
```

Example Cache size $k = 4$.

The requests: 1, 8, 9, 4, 1, 3, 4, 3, 5, 6, 1.

* shows marking. Cache and request:

$$\begin{aligned}
&(1,8,9,4) \leftarrow 1, \quad (1^*,8,9,4) \leftarrow \textcolor{red}{3},\\
&(1^*,8,3^*,4) \leftarrow 4, \quad (1^*,8,3^*,4^*) \leftarrow 3,\\
&(1^*,8,3^*,4^*) \leftarrow \textcolor{red}{5}, \quad (1,5,3,4) \leftarrow \textcolor{red}{6},\\
&(1,6^*,3,4) \leftarrow 1, \quad (1^*,6^*,3,4).
\end{aligned}$$

Analysis

Let us lower-bound the optimal offline number of cache misses.

- Divide the sequence of requests into phases. Each phase starts with a unmarked cache and ends once the marks are removed.
- A requested item is called fresh if it has not been marked in the previous phase. Else it is called stale.
- In phase j, let c_j be the number of fresh requests.

Let r be the total number of phases.

Claim The optimal number of cache misses is at least $\frac{1}{2}\sum_{j=1}^{r} c_j$.

This follows (with some reasoning) from the fact that no matter what policy is used, in phases $j-1$ and j together there are $k+c_j$ distinct element requests, hence at least c_j cache misses.

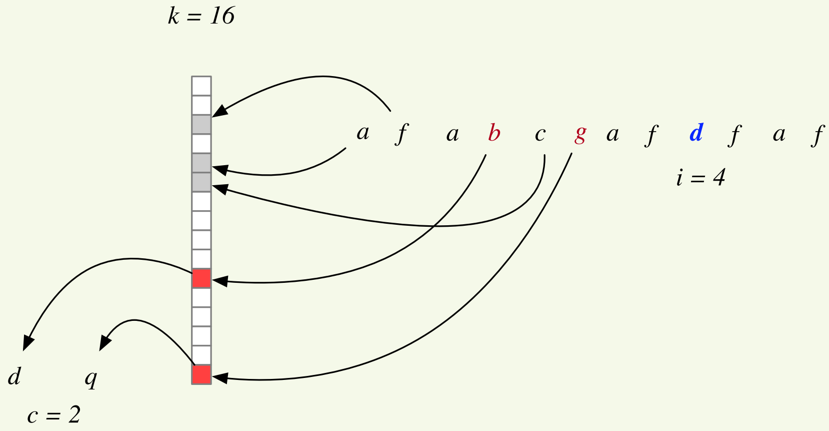
Claim The expected number of cache misses in our algorithm is at most

$$(H(k)+1)\sum_{j=1}^{r} c_j,$$

where $H(k) = 1 + 1/2 + \cdots + 1/k = O(\log k)$.

To prove it, estimate $\mathsf{E} X_j$ where X_j is the number of cache misses in phase j.

- Each of the c_j fresh requests is a cache miss.
- Consider the stale requests. For the ith request to some distinct stale item s, let $A_i = 1$ if it is a cache miss and 0 otherwise. The number of stale cache misses is $\sum_i A_i$. The expected value of this number is $\sum_i \mathsf{E} A_i$.
- $\mathsf{E} A_i = \Pr\{\text{the } i\text{th request is a cache miss}\}$. We will estimate this probability.



$k = 16$
$a \quad f \quad a \quad b \quad c \quad g \quad a \quad f \quad d \quad f \quad a \quad f$
$i = 4$
d
q
$c = 2$

Red requests are fresh. Blue request is a stale cache miss.

- From the k items in the cache in phase $j-1$ (these are now the stale ones), already $i-1$ have been marked in phase j, so $k-i+1$ are unmarked.
- Suppose that $c \le c_j$ fresh requests were already made. Then c elements among the $k-i+1$ have been evicted.
- The set of c evicted elements is chosen uniformly from all subsets of size c. So the probability for the ith request to belong to this set is

$$\frac{c}{k-i+1} \le \frac{c_j}{k-i+1}.$$

$$\mathsf{E} X_j \le c_j + c_j\left(\frac{1}{k} + \frac{1}{k-1} + \dots\right) = c_j + c_j \sum_{i=1}^{k} \frac{1}{i} = c_j(1 + H(k)).$$

Calculus shows $H(k) = \ln k + O(1)$, but upper bound is easier:

$$\begin{aligned} H(k) &= 1 + 1/2 + \dots + 1/k \\ &= 1 + (1/2 + 1/3) + (1/4 + \dots + 1/7) + \dots \\ &\le 1 + 1 + 1 + \dots (\log k \text{ times}). \end{aligned}$$

For a sequence of requests σ let $f(\sigma)$ be the optimal number of cache misses. We proved:

Theorem On σ, the randomized caching algorithm achieves an expected number $O(\log k) f(\sigma)$ cache misses.

Randomized selection

(Not covered in Spring 2020)

Let A be an array of n numbers $a_1, \dots, a_n$. For simplicity assume they are all distinct.

- The median is the $\lfloor n/2 \rfloor$th number in the sorted order of elements of A.
- The task of looking for the median is solved by sorting A. But that takes $n \log n$ steps (we count comparisons). Is not there a faster method?
- Some faster methods are based on divide-and-conquer. Since they are recursive, we first generalize: we want to compute $\mathit{Select}(A, k)$, the kth element in the sorted order.

Idea:

```
GenericSelect$(A, k)$:
    if $n = k = 1$ then return $a_1$
    Pick some index $r$ (by some method).
    $A_1 \leftarrow$ all elements of $A$ that are $\leq a_r$
    $A_2 \leftarrow$ all elements of $A$ that are $> a_r$
    if $k \leq |A_1|$ then
        return GenericSelect$(A_1, k)$
    else
        return GenericSelect$(A_2, k - |A_1|)$
```

If $\max(|A_1|, |A_2|) = O(n)$ then the recursive analysis promises an $O(n)$ estimate.

But how to pick such a number r? Idea: Pick a random r, hoping it is good enough on average.

RandomizedSelect(A, k) is essentially the same as *GenericSelect*(A, k), except that r is a random variable R with $\Pr\{R = i\} = 1/n$ for $i = 1, \dots, n$. Let X be the running time, and $T(n)$ its expected value.. Using Bayes's Theorem (8) for conditional expectations:

$$T(n) = \mathsf{E}X = \mathsf{E}[X \mid R = 1]\Pr\{R = 1\} + \cdots + \mathsf{E}[X \mid R = n]\Pr\{R = n\}.$$

If $R = i$ then the recursion is called either in A_1 or in A_2; in the worse case in the larger one.
Adding the cost n of partitioning into A_1 and A_2:

$$\begin{aligned} \mathsf{E}[X \mid R = i] &\le n + T(\max(i, n-i)), \\ T(n) &\le n + \frac{1}{n}(T(n-1) + T(n-2) + \cdots + T(n-2) + T(n-1)) \\ &= n + \frac{2}{n}(T(n-1) + T(n-2) + \cdots + T(n/2+1)). \end{aligned}$$

We know $T(1) =$. Instead of resolving the complex estimate

$$T(n) \leq n + \frac{2}{n}(T(n-1) + T(n-2) + \cdots + T(n/2))$$

try to prove $T(n) \leq cn$ for some constant c: try to choose a c that allows proving this by induction:

$$T(n) \leq n + \frac{2c}{n}((n-1) + (n-2) + \cdots + n/2).$$

The sum of an arithmetic series of k terms with starting term a and ending term b is $k\frac{a+b}{2}$. So the sum inside is $\frac{n}{2}\frac{3n-1}{4} < \frac{3n^2}{8}$. Substituting:

$$T(n) < n + \frac{2c}{n}\frac{3n^2}{8} = n(1 + 3c/4).$$

Choosing c with $1 + 3c/4 = c$, that is $c = 4$, completes the proof.

- The Kleinberg-Tardos book gives a different proof of $T(n) = O(n)$, also worth seeing.
- Similarly, it gives a different proof than the one given below that randomized Quicksort has expected running time $O(n \log n)$.

Average-case analysis of Quicksort

(Not covered in 2020)
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 $O(n \log n)$.
  However, it is not sufficiently reassuring to say that Quicksort works well on random orders. The typical arrays that we will have to sort are probably not random at all, for example they may be partially sorted.

- If we introduce randomness ourselves (we randomize), we don’t have to rely on the fact that the input array is random. 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 $\Pr\left\{ C_{ij} = 1 \right\} = \mathsf{E}\, C_{ij} = \frac{2}{j-i+1}$. The expected number of comparisons is

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

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 $\le 2n\ln n = O(n \log n)$.

Hash tables

(Not covered in Spring 2020)

Problem A very large universe U of possible items, each with a different key. The set $S \subset U$ of n of actual items that may eventually occur is much smaller. We want to store the items in a data structure in a way that they can be

- stored fast as they come, and found fast later.

Solution ideas
- Balanced search trees: you must have seen them in a data structures course.
- Hash function $h(k)$, hash table $T[0\,..\,m-1]$. Key k hashes to hash value (bucket) $h(k)$. This can be in practice faster, but its analysis is more complex.

Problem with hashing Collisions.

Resolution Chaining, open hashing, and so on.

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

Hash functions

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, a string of bytes can be treated as a base 256 integer, possibly adding up such integers for different segments of the word.

Randomization: universal hashing

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

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

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.

Using a universal hash function

Once we have chosen the random parameter r, we will keep it fixed.

- No matter how we fix r, if the universe U is big then there is some set $S_r \subset U$ with $|S_r| = n$ such that $h(k, r)$ maps all elements of S_r to the same position in the table.
- But if we assume that somehow the set $S \subset U$ was fixed before we choose r (we just don't know S), or chosen independently of r, then universality helps bounding the expected number of collisions.

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

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

(If x is a bit string, break it into segments of size $\log m$.) Fix the random coefficients $0 \le r_i < m$, $i = 1, \ldots, d$, 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, $d \not\equiv 0 \pmod{p}$, and $ad \equiv bd \pmod{p}$ 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 $(a-b)d$. It does not divide d, so it divides $a-b$.

Let us show that our random hash function is universal. Assume $(x_1, \dots, x_d) \neq (y_1, \dots, y_d)$. We show that $\Pr\{ h(x, r) = h(y, r) \} \leq 1/m$. There is an i with $x_i \neq y_i$, we might as well assume $x_1 \neq y_1$. If $h(x, r) = h(y, r)$ then

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

where A only depends on the random numbers $r_2, \dots, r_d$. No matter how we fix $r_2, \dots, r_d$, 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 $r_2, \dots, r_d$) is $1/m$. Since this probability is the same under all conditions, it is equal to $1/m$.

Closest pair of points revisited

(Not covered in Spring 2020)
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, shortest distance upper bound 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 $|k-k'|,|l-l'| \le 2$.
- 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 $\mathsf{E} X_i = 2/i$.

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

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

Polynomial reduction, formally

- The computational problems in question are given by some mapping from (say, binary) string inputs, or instances to string outputs: so it is a function $A : \Sigma^* \to \Sigma^*$ where Σ is some finite alphabet.
- Even if we talk about other objects as inputs or outputs (graphs, numbers), they will eventually be encoded into strings for processing by our machines.
- We say that a polynomial-time computable functions τ, ϕ reduce problem A to problem B if for every possible input x, we have

$$A(x) = \phi(B(\tau(x))).$$

  So we encode the input x of A into an input $\tau(x)$ of problem B, and then transform the solution $y = B(\tau(x))$ into a solution $\phi(y)$ of problem A. (If $A(x), B(x) \in \{0, 1\}$ then we don't need the transformation ϕ.)

- Sometimes we allow for a more general kind of reduction: a polynomial-time algorithm solving A in which asking and (magically) getting an answer $B(y)$ on any particular input y counts only as one step.
- The imaginary “black box” device that answers our queries is sometimes called an oracle, and the computation using it an oracle computation.

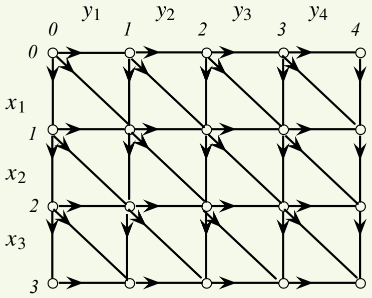
When a reduction exists from problem A to problem B we can write $A \leq B$. This relation is transitive (creates a partial order): if $A \leq B$ and $B \leq C$ then $A \leq C$. Indeed, suppose that

- A is reduced to B with the help of the polynomial-time functions τ_1, ϕ_1 by $A(x) = \phi_1(B(\tau_1(x)))$.
- Similarly B is reduced to C by $B(y) = \phi_2(C(\tau_2(y)))$.

The functions $\tau(x) = \tau_2(\tau_1(x))$ and $\phi(y) = \phi_1(\phi_2(y))$ are also polynomial-time, hence we have a polynomial reduction $A(x) = \phi(C(\tau(x)))$.

Some more examples:

- The sequence alignment problem was defined earlier.
  Input: two sequences $X = (x_1, \dots, x_m)$, $Y = (y_1, \dots, y_n)$, of symbols in some alphabet Γ, and penalties δ for deletion, and distance function $d(x, y)$ over the alphabet Γ (as penalty of mismatching symbols).
  Output: two sequences of indices $i_1, \dots, i_k$, $j_1, \dots, j_k$ at which the sequences must be aligned for minimizing the total penalty.
- Reduction: to the shortest path problem in a graph $G(X, Y, \delta, d)$ with edge lengths, with source and destination vertices s, t. Any algorithm finding a shortest path in G from s to t finds an optimal alignment between X and Y.



For $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$. Points (i, j) where i represents the position between x_i and x_{i+1}.

- Horizontal and vertical edges have length δ.
- Diagonal edge ending in (i, 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.)
- Find the alignment going backwards using $A[i, j]$, or record the parents while computing the distances $A[i, j]$ (as in Dijkstra).

There are two very different uses of reduction from a problem A to problem B.

1. We may have a good algorithm for solving B, and now the reduction provides also a good algorithm for solving A.
2. We know (or suspect) that there is no good algorithm for solving A. By the reduction now we know (or suspect) that there is no good algorithm for solving B either, since then the reduction would provide also a good algorithm for B.

In **NP**-completeness theory we mostly use reductions for the second purpose.

We discussed the subset sum problem as special case of the knapsack problem: Given positive integers $a_1, \dots, a_n, c$, decide whether there are $x_1, \dots, x_n \in \{0, 1\}$ such that

$$a_1 x_1 + \dots + a_n x_n = c. \tag{9}$$

A different problem where we have 2 equations to satisfy:

$$\begin{aligned} a_1 x_1 + \dots + a_n x_n &= c, \\ b_1 x_1 + \dots + b_n x_n &= d. \end{aligned} \tag{10}$$

Of course (9) can be reduced to (10) in a trivial way: just let the second equation be, say, 0=0 (or equivalent to the first one).
But we will reduce (10) to (9) also, so (9) is just as difficult as (10).

$$a_1 x_1 + \cdots + a_n x_n = c, \tag{11}$$

$$b_1 x_1 + \cdots + b_n x_n = d. \tag{12}$$

Multiply the second equation by $M = a_1 + \cdots + a_n + c + 1$ and add it to the first one:

$$(a_1 + M b_1) x_1 + \cdots + (a_n + M b_n) x_n = c + M d. \tag{13}$$

Let us see that if $x_1, \ldots, x_n$ solves (13) then it solves both (11) and (12). The remaindering distributes into the sum:

$$(u + v) \bmod M = ((u \bmod M) + (v \bmod M)) \bmod M. \tag{14}$$

Taking the remainder of the two sides of (13) modulo M and using (14) repeatedly, we get back (11). Indeed, for example $a_1 + M b_1 \bmod M = a_1$ since $a_1 < M$. Subtracting (11) from (13) and dividing by M we get back (12).

NP problems

Many of the problems we have seen, even if they are not solvable polynomially, possess a weaker but useful property: if a solution is offered, we can use it, we don't have to trust it blindly: 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 relation

  $$V(x, w)$$

  over strings x, w.
  x is the input, or instance.
  w is a potential witness, or certificate (or a “solution”).
- We require V to be computable in time polynomial in the length of the input x.
  In detail: there is a constant c such that for every n, every input string x of length n, the response $V(x, w)$ is computable in time $O(n^c)$. String w is a witness if $|w| = O(n^c)$, and $V(x, w) = \text{True}$.

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 $\geq k$.

Decision problems, search problems

To every **NP** problem defined by a verification relation $V(x, w)$ belong two related questions:

Decision problem Given x, is there a w satisfying $V(x, w)$?

Search problem Given input x, find a witness w satisfying $V(x, w)$ (or say if there is none).

Formally, a set S of strings is in **NP** it is the decision problem for some polynomial-computable verification relation $V(x, w)$, that is if

$$x \in S \Leftrightarrow \exists w V(x, w).$$

- Solving the search problem also solves the decision problem. An algorithm solving the decision problem sometimes helps solving the search problem (see one of your homework questions, on subset sum).
- But other times it does not.
  - The compositeness question on number x is the same as the question whether x is a prime number (that is not composite). There is a (nontrivial!) polynomial algorithm for deciding primality.
  - The search problem is asking to find a proper divisor of the number x. If there was a polynomial algorithm for this, then repeating it would give a polynomial algorithm of factorization of an input x (into prime divisors). Most of modern practical cryptography (the basis of a lot our secure internet transactions) is relying on the fact that no efficient factorization algorithm is known.

Optimization problems

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

- Clearly, a solution to the largest independent set problem would answer any question of the kind: "given G, k, does G have an independent set of size $\geq k$?".
- 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 which the answer is affirmative, is the size of the maximum independent set.
- The above reduction can be speeded up via binary search.
- Clearly to each optimization problem belongs also a search problem: asking not just for the size of the largest independent set but also for an independent set of maximum size.

Boolean formulas

$x_i \in \{0,1\}$.

$$\neg x = 1 - x,\ x \wedge y = \min(x,y),\ x \vee y = \max(x,y),$$

negation, conjunction, disjunction. Example formula:

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

Such a formula defines a Boolean function. An assignment (say $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 0$) allows to compute a value (in our example, $F(0,0,1,0) = 0$).

Some important rules transforming a formula without changing the Boolean function it represents: distributive and de Morgan rules,

$$\begin{aligned}
(x \vee y) \wedge z &= (x \wedge z) \vee (y \wedge z), \\
(x \wedge y) \vee z &= (x \vee z) \wedge (y \vee z), \\
\neg(x \wedge y) &= \neg x \vee \neg y, \ \neg(x \vee z) = \neg x \wedge \neg z.
\end{aligned}$$

Disjunctive normal form (DNF): a disjunction of clauses, each a conjunction of variables and negated variables (called literals). Example:

$$(x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_3 \wedge x_4) \vee \neg x_4$$

Conjunctive normal form (CNF): a conjunction of clauses, each a disjunction of literals. Example:

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

Fact Each Boolean formula is equivalent (as Boolean function) to a conjunctive normal form, and also to a disjunctive normal form.

You can find these normal forms by applying the distributivity and de Morgan rules.

Note If you start for example from a CNF, the equivalent DNF may be exponentially larger (applications of the distributive rule can double the size repeatedly).

Fact Every Boolean function $f(x_1, \dots, x_n)$ can be represented by a formula.

Indeed, a disjunctive normal form (of exponential size) can easily be read off from the table of values of $f(x_1, \dots, x_n)$, as we go over all possible assignments.

Satisfiability

- An assignment (a_1, a_2, a_3, a_4) satisfies F, if $F(a_1, a_2, a_3, a_4) = 1$.
  Example: $(\neg x \vee y) \wedge x$ is satisfied by $x = 1$, $y = 1$.
  And no assignment satisfies $(\neg x \vee y) \wedge x \wedge \neg y$.
- The formula is satisfiable if it has some satisfying assignment.
  So it is unsatisfiable if it is always false. Example: $x \wedge \neg x$.
- The formula is a tautology if it is always true, that is its negation is unsatisfiable. Example: $x \vee \neg x$.
- Satisfiability problem FSAT: given a formula $F(x_1, \ldots, x_n)$ decide whether it is satisfiable.

Special cases:

- 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. Asks to satisfy some very simple constraints: each a disjunction clause with up to three literals.

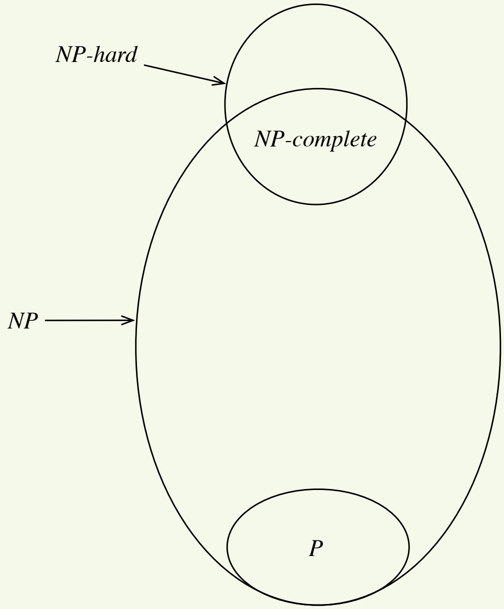
NP-completeness

Theorem (Cook-Levin) Every **NP** problem is reducible to 3SAT.

Significance of this result 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 to some problem (witnesses to a verification relation) would guarantee also a fast algorithm to decide whether there is any solution at all.

Usual conclusion This is unlikely, so probably SAT is hard.

- A problem to which every **NP** problem is reducible is called **NP**-hard. If it is also in **NP** (thus a decision problem for some verification relation) then it is called **NP**-complete.
  Now we know that SAT is NP-complete.
- A problem can be **NP**-hard and not **NP**-complete even just by its form: when it is not a decision problem but, say, an optimization problem or a search problem.
- If we reduce an **NP**-complete problem A to some problem B then this shows that B is also **NP**-hard. (If B is also in **NP** then B is also **NP**-complete.)

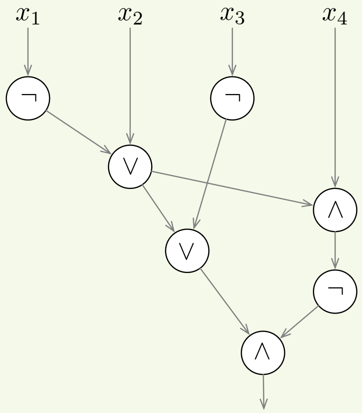

NP-hard
NP-complete
NP
P

Chains of reductions have shown the **NP**-completeness of many well-know problems, helping to explain their difficulty. We will show some of these reductions, starting with the large independent set problem.

Proof outline for the Cook-Levin theorem

Boolean circuits

Our computers are built up from Boolean circuits like this one:



It computes $((\neg x_1 \vee x_2) \vee \neg x_3) \wedge \neg((\neg x_1 \vee x_2) \vee x_4)$,
but is more economical: it reuses $\neg x_1 \vee x_2$.

Boolean circuits are missing an important computer component: memory units. Still, polynomial-time functions are computable by polynomial-size circuits:

Theorem (Poly-Sized Circuits) Suppose that a function $f(x)$ is computable in time $O(n^c)$ for some constant c—that is computable in polynomial time. Then there for every n there is a Boolean circuit C_n of size $O(n^{2c})$—that is polynomial size that from every x of length n computes $f(x)$.

The idea of the proof is that for every step of the computation, we represent the state of memory of our computer by a “layer” of the Boolean circuit.

Since circuits also compute Boolean functions, satisfiability is defined for them as well. The CSAT problem's inputs are circuits.

Lemma The CSAT problem is **NP**-complete.

- Consider any **NP** problem A with polynomial-time verification relation $V_A(x, w)$.
  Sketch of its reduction to CSAT:
- By the Poly-Sized Circuits Theorem, an $O(n^c)$ algorithm computing V_A gives rise for each n to a circuit $C_{A,n}$ of size $O(n^{2c}$ computing $V_A(x, w)$ for all $|x| = n$, $|w| \leq n^c$.
- Hardwiring the input string x we get a circuit $C_{A,n,x}$ with input w. Its satisfying assignments w are just the witnesses for $V_A(x, w)$.

CSAT to 3SAT

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 then we put the formula $x_i \vee x_j \Leftrightarrow y_k$ into conjunctive normal form:

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

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

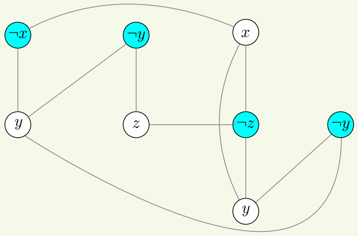
$$F_C(x_1, \ldots, x_n, y_1, \ldots, y_N) \wedge y_N$$

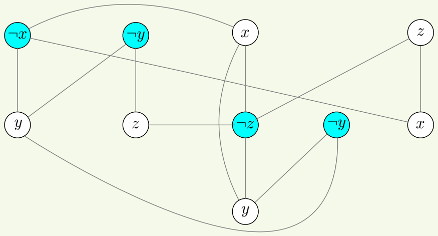
is satisfiable if and only if the circuit C is.

SAT to independent set

How to satisfy a CNF $C_1 \wedge C_2 \wedge \cdots \wedge C_n$? Each clause must be satisfied. For example if $C_1 = x \vee y \vee \neg z$ then one of $x, y, \neg z$ must be made true. If x appears in C_1 and $\neg x$ in C_3 then both cannot be made true.
So the task is this:
Pick one literal per clause, but if you picked a literal from some clause you cannot pick its negation from elsewere.

$$(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z \vee y) \wedge \neg y$$





$$(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z \vee y) \wedge \neg y \wedge (z \vee x)$$

Graph G_F for conjunctive normal form $F = C_1 \wedge \cdots \wedge C_m$:

- Assign a vertex to each occurrence of each literal.
- Connect literals within each clause.
- Connect each literal to each occurrence of its negation.

Each independent set corresponds to a set of literals that can be made true simultaneously, and which contains at most one literal per clause.

Integer programming

Reduction of 3SAT to 0-1 linear programming

- Equations of the form $x + x' = 1$ where x' represents $\neg x$.
- Turn a clause of the form $x \vee y \vee \neg z$ into an inequality

$$x + y + z' \geq 1.$$

- Turn inequalities into equations: $x + y + z' \geq 1$ into

$$x + y + z' + t_1 + t_2 = 3$$

  using some new variables t_i. So, satisfying a CNF is reduced to solving a set of equations

$$\sum_{j=1}^{n} a_{ij} x_j = b_i,\ i = 1, \ldots, m$$

  in 0-1 variables x_j where all $a_{ij} \in \{0, 1\}$.

- Reduction of many equations to one: same trick as used to reduce two equations to one. Multiply the ith equation by M^{i-1} where $M = 1 + \max_i b_i + \sum_j a_{ij}$, and add them all up:

$$A_1 x_1 + \cdots + A_n x_n = B$$

  where for example $A_1 = a_{11} + a_{21} M + \cdots + a_{m1} M^{m-1}$.
  This is a reduction to the subset sum problem, showing that it is **NP**-complete.
- We gave a dynamic programming algorithm for the subset sum problem (actually, for the more general knapsack problem). We remarked that it is not polynomial when the coefficients are large. Here the A_j will be large.

Traveling salesman

Another famous group of **NP**-complete (or **NP**-hard) problems:

- A Hamiltonian cycle of a graph is a cycle that passes through every vertex exactly once. The problem: given a graph G, does it have a Hamiltonian cycle?
- Given a graph G and two vertices s, t in it, what is the length of the longest path between s and t?
  In contrast, we have seen that the shortest path problem is in P.
- Given a set of cities with connecting roads between them that have lengths, what is the length of the shortest tour for a traveling salesman who needs to pass through all of the cities?

These problems reduce rather easily to each other, but it is a little tricky to show **NP**-completeness.

Similar problems in P and in NP

In a number of cases, changing a parameter turns a problem from polynomial to **NP**-complete.

- 3SAT is **NP**-complete.
  2SAT is in **P** (the set of polynomal-time solvable problems).
- 3-coloring is **NP**-complete.
  2-coloring (deciding whether a graph is bipartite) is in **P**.
- 3-partite matching (whether a 3-partite graph can be covered by disjoint triangles) is **NP**-complete.
  2-partite matching (whether a bipartite graph has a perfect matching) is in **P**.
- Traveling Salesman problem is **NP**-hard.
  Chinese Postman problem (finding the shortest route passing through all edges of a graph) is in **P** (not easy).

- Finding the largest independent set of vertices in a graph is **NP**-hard.
  Finding the largest independent set of edges (that is a matching) is solvable in polynomial time. We have seen a polynomial algorithm for bipartite graphs, but there is one also for general graphs (more complex).
  The Chinese Postman problem can be reduced to this.

- Finding a proper divisor of an integer x is probably hard (this is the factorization problem), even if not **NP**-hard.
  The trial division algorithm (just try all possible witnesses $1 < w < x$) is not polynomial, as the input length is $\log x$.
- Finding a common divisor of integers x, y (if it exists) is solvable in polynomial time by the Euclidean algorithm: using the identity

$$\gcd(x, y) = \gcd(y, x \bmod y).$$

  in a recursion.

Linear programming

(Not covered in Spring 2020)

Reformulation of vertex cover: given an undirected graph $G = (V, E)$, linear inequalities

$$\begin{aligned} x_u + x_v &\geq 1 \text{ for all } (u, v) \in E, \\ x_u &\geq 0, \\ \sum_u x_u &\leq k. \end{aligned}$$

When x_u must be integers, the solvability is equvalent to the question whether G has a vertex cover of size $\leq k$.

- Relaxation: of the above question: allow x_u to be real numbers: so we are looking for a fractional vertex cover.
  Is this still an **NP** problem? Not obvious, since maybe there are only witnesses x_u that cannot be expressed with a small number of bits.
- The fractional vertex cover problem is a special linear program.
  A general linear program for some (integer) coefficients a_{ij}, b_i, c_j:
  Find a solution $x_1, \dots, x_n$ for the linear inequalities

  $$a_{i1}x_1 + \cdots + a_{in}x_n \leq b_i, \quad i = 1, \dots, m$$

  (the constraints), maximizing $c_1x_1 + \cdots + c_nx_n$ (the objective function).

Theorem The solvability (in real numbers) of a set of linear inequalities in integer coefficients (just the constraints of the above program) is an **NP** problem: more precisely, if there is a solution then there is one in which $x_j = \frac{p_j}{q_j}$ with polynomial-length integers p_j, q_j.

The proof uses linear algebra.

Theorem (famous) There is a polynomial algorithm to solve (in real numbers) a set of linear inequalities in integer coefficients.

The maximum flow problem is a special linear program. Indeed, in it, given capacities $c(u,v)$ on the edges of a directed graph $G=(V,E)$ with source and target vertices $s,t\in V$, we are looking for a flow function $f(u,v)$ satisfying the inequalities

$$\begin{aligned} f(u,v)+f(v,u)=0, &\quad \text{for all edges } (u,v),\\ f(u,v)\le c(u,v), &\quad \text{for all edges } (u,v),\\ \sum_{v:(u,v)\in E} f(u,v)=0 &\quad \text{for all } u\in V\setminus\{s,t\}, \end{aligned}$$

and maximizing $\sum_{v:(s,v)\in E} f(s,v)$. This is a linear program for the variables $f(u,v)$.

- Known algorithms for general linear programs are more complex than those for just solving the maximum flow problem.

Overview: representative problems

Interval scheduling Solved by a greedy method, near-linear time.

Weighted interval scheduling Solved by dynamic programming, near-linear time.

Bipartite matching Solved by a more complex method, still polynomial time.

Large independent set **NP**-complete, takes possibly exponential time.

Competitive facility location Two players alternatingly select nodes in a graph, each of which has some value. Cannot select neighbor of an already selected node.
Question: Can player 1 achieve a total value of at least B, no matter how player 2 plays?
This is a PSPACE-complete problem, believed to be even much harder than **NP** problems. (It is **NP**-hard.)

Approximation algorithms

(Not covered in Spring 2020)

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

- In some example homework we may have shown that certain activity selection algorithms, though not optimal, approximate the optimal number of activities within a factor of 2.
- A simple algorithm for the knapsack problem 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.

Center selection

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 a 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 $d(u, w) \le d(u, v) + d(v, w)$.

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 $V = \{-1, 0, 1\}$, $U = \{-1, 1\}$, $k = 2$, 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.

An algorithm

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($k$)
    $C \leftarrow \{u\}$ for some arbitrary initial town $u \in U$
    **for** $i = 2$ **to** $k$ **do**
        find the town $v \in U$ farthest from $C$, that is
          $r(C) = d(v, C)$
        $C \leftarrow C \cup \{v\}$
```

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:

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

$k = 2$. If $u_1 = (x_1, y_1)$, $u_2 = (x_1, y_2)$ then

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

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

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

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 $u' \neq v'$, hence we cover all C^* this way. Indeed, otherwise by the triangle inequality $d(u,v) \leq d(u,u') + d(u',v) \leq 2r^*$. So 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. □

Vertex cover

See also the description under greedy algorithms.

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 $c_v \geq 0$. 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 $O(\log n)$ (see the example under greedy algorithms).
- 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 $p_e \geq 0$ is assigned to each edge. We say that these constitute a valid set of prices, if for each vertex u,

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

  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 $c(H) \geq s(p)$. Indeed,

$$c(H) = \sum_{u \in H} c_u \geq \sum_{u \in H} \sum_{e \ni u} p_e \geq \sum_{e \in E} p_e = s(p). \tag{15}$$

Our algorithm will just create some prices that push up the lower bound (15) 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:

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

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.

Note Our new algorithm for finding the prices p_e is very simple: it is just a greedy one. On the other hand, the idea to find a vertex cover via the dual approach of edge prices is non-trivial.

Strong duality

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:

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

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 $c(x) \geq s(p)$. The strong duality theorem says that $\min_x c(x) = \max_p s(p)$. We will not prove it here.

Approximating for the knapsack problem

(Not in a lecture in Fall 2013, can be skipped.)
Recall the knapsack problem. Given: volumes $b \geq a_1, \dots, a_n > 0$, and integer values $v_1 \geq \cdots \geq v_n > 0$.

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

Optimal solution x^*, giving $\mathrm{OPT} = \sum_i v_i x_i^*$.
Let $v = \sum_i v_i$. We solved this problem using dynamic programming, in $O(vn)$ arithmetic operations: this was not polynomial since v can be exponential in the input size.

Idea: break each v_i into a polynomial-size number of big chunks, for approximation. Let $r > 0$, $v_i' = \lfloor v_i / r \rfloor$.

$$\begin{aligned} \text{maximize } & \textstyle\sum_i v_i' x_i \\ \text{subject to } & \textstyle\sum_i a_i x_i \le \quad b, \\ & x_i = 0, 1,\ i = 1, \dots, n. \end{aligned}$$

Optimal solution x'.
For each ε, we will choose r in such a way that

- The value $\sum_i v_i x_i'$ approximates the original OPT within a factor $1 - \varepsilon$.
- The runtime is polynomial in $n, 1/\varepsilon$.

This is called an approximation scheme.

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

$$\sum_i (v_i/r)x'_i \ge \sum_i v'_i x'_i \ge \sum_i v'_i x^*_i \ge \sum_i (v_i/r)x^*_i - n,$$

$$\sum_i v_i x'_i \ge \mathrm{OPT} - r \cdot n = \mathrm{OPT} - \varepsilon v_1,$$

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

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

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

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

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