Solutions to Same Problems

CS 3AC3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Material partially based on *Algorithm Design* by Jon Kleinberg and Éva Tardos (Chapter 8)

The class **NP**

Definition (with verifiers)

The class **NP** (from *Nondeterministic Polynomial*) consists of those problems that are **verifiable** in polynomial time. More specifically, they are problems that can be *verified* in $O(n^k)$ for some constant k. where n is the size of the input to the problem.

Hamiltonian Path is such a problem!

Definition (with *nondeterministic* algorithms)

The class **NP** (from *Nondeterministic Polynomial*) consists of those problems that are *solvable* in polynomial time by *nondeterministic* algorithms. More specifically, they are problems that can be solved in $O(n^k)$ for some constant k. where n is the size of the input to the problem, by *nondeterministic* algorithms.

- The idea of *Nondeterministic Algorithms* is a simple consequence of *angelic* semantics.

NP-completeness

Definition

A problem Y is **NP**-complete if it satisfies the following two conditions:

1. $Y \in \mathbf{NP}$
2. every $X \in \mathbf{NP}$ is *polynomially reducible* to Y, i.e. $X \leq_P Y$.

Let **NPC** denote the class of all **NP**-complete problems.

Theorem

Suppose Y is **NP***-complete. Then* $Y \in \mathbf{P} \iff \mathbf{P} = \mathbf{NP}$.

NP-completeness: basic tool

Theorem

If X is **NP***-complete, $Y \in$* **NP***, and $X \leq_P Y$, then Y is* **NP***-complete.*

Algorithm (Showing NP-completeness of Y)

1. *First show that $Y \in$* **NP***. This is usually done by showing that an instance of Y has a* **polynomial verifier***.*
2. *Find a problem X that has been proven to be* **NP***-complete. For example, 3-SAT, VECTOR-COVER, HAMILTON-CYCLE, etc. If Y is a graph problem, try first X that is also a graph problem.*
3. *Show $X \leq_p Y$, i.e. X can be* **polynomially** *reduced to Y. While the fact that a transformation o X into an* instance *of Y is polynomial is often almost obvious,* **always** *mention it and explain.*

NP-completeness: frequent errors

Algorithm (Showing NP-completeness of Y)

1. *First show that $Y \in$ **NP**.*
2. *Find a problem X that has been proven to be **NP**-complete.*
3. *Show $X \leq_p Y$, i.e. X can be **polynomially** reduced to Y.*

Frequent Errors.

- It is NOT shown that $Y \in$ **NP**. Usually can be fixed.
- It is attempted to show that $Y \leq_P X$ instead of $X \leq_P Y$. This is the most serious error.
- It is not argued that transformation of X into an instance of Y is polynomial. Usually can be fixed.

Feedback Vertex Set

Let $G = (V, E)$ be a directed graph. A *feedback vertex set* is a subset $S \subseteq V$, such that every cycle of G contains a vertex in S.
A feedback vertex set problem is: Does a directed graph have a feedback set with k members?

1. To prove that the problem is in **NP** we need to show a polynomial time *verifier*.
2. All cycles are represented by *strongly connected components*. Finding strongly connected components can be done in $O(|V| + |E|)$ time (i.e. in $O(n^2)$ time) by using DFS (Depth First Search) technique (last year algorithm course).
3. The number of strongly connected components is smaller than n, we need to verify if each such component has a vertex in S, where $|S| \leq n$.
4. Each component has less than n^2 vertices. We need to verify if $C \cap S \neq \emptyset$ for each strongly connected component C.
5. By brute force we can do it in $O(n^2)$, so, altogether, we can do this verification in $O(n^3)$, i.e. polynomial time.

Feedback Vertex Set and Vertex Cover

- Let $G = (V, E)$ be a **directed graph**. A *feedback vertex set* is a subset $S \subseteq V$, such that every cycle of G contains a vertex in S.
  A feedback vertex set problem is: Does a directed graph have a feedback set with k members?
- Let $G = (V, E)$ be an **undirected graph**. A *vertex cover* of G is a subset $S \subseteq V$ such that every edge of G is incident upon some vertex in S.
  A vertex cover problem is: Does a directed graph have a vertex cover with k members? *It is NP-complete*.
- We show that vertex cover problem is polynomially transformable into the feedback vertex set.

Vertex Cover Polynomially to Feedback Vertex Set

- A rule: an arbitrary vertex cover and a specific feedback vertex set.
- Let $G = (V, E)$ be an undirected graph.
- Let D be a directed graph formed by replacing each edge of G by two directed edges.
- Specifically, let $D = (V, E')$, where $E' = \{(v, w) \mid \{v, w\} \in E\} \cup \{(w, v) \mid \{v, w\} \in E\}$. Since every edge in E has been replaced by a cycle in D. a set $S \subseteq V$ is a feedback set for D (every cycle of D contains a vertex in S) if and only if S is a vertex cover for G.
- Also, the representation of D can easily be found from G in polynomial time.
- Therefore, the feedback vertex set is NP-complete.

A store trying to analyse the behaviour of its customers will often maintain a two-dimensional array A, where the rows correspond to its customers and the columns correspond to the products it sells.
The entry $A[i, j]$ specifies the quantity of product j that has been purchased by customer I.
Here's a tiny example of such an array A.

	liquid detergent	beer	diapers	cat litter
Raj	0	6	0	3
Alanis	2	3	0	0
Chelsea	0	0	0	7

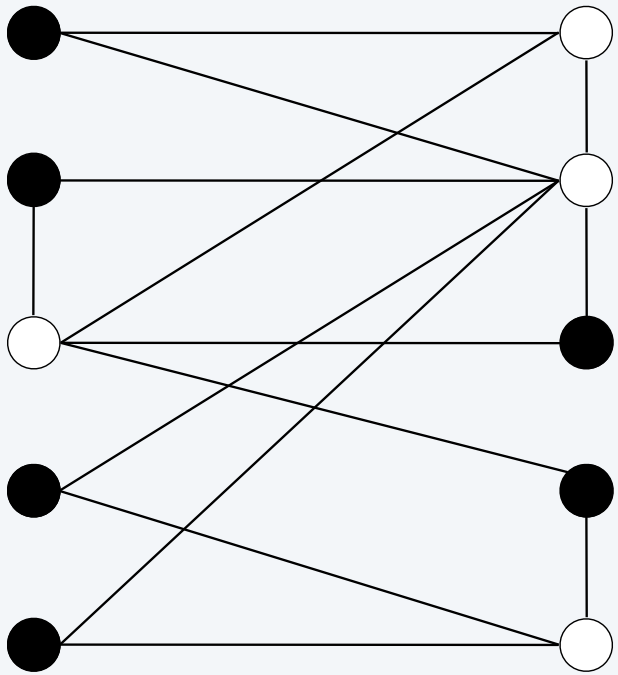
One thing that a store might want to do with this data is the following. Let us say that a subset S of the customers is **diverse** if no two of the of the customers in S have ever bought the same product (i.e., for each product, at most one of the customers in S has ever bought it).
A diverse set of customers can be useful, for example, as a target pool for market research.
We can now define the Diverse Subset Problem as follows: Given an $m \times n$ array A as defined above, and a number $k \leq m$, is there a subset of at least k of customers that is *diverse*?
Show that Diverse Subset is NP-complete.

INDEPENDENT-SET is **NP**-complete

INDEPENDENT-SET. Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

Ex. Is there an independent set of size ≥ 6?

Ex. Is there an independent set of size ≥ 7?



independent set of size 6

Diverse Subset is NP-complete

- The problem is in **NP** because we can exhibit a set of k customers, and in polynomial time it can be checked that no two bought the same common product.
- We now show that *Independent Set* $\leq_P$ *Diverse Subset*.
- A rule: an arbitrary independent set and a specific diverse subset.
- Given a grapg G and a number k, we construct a customer for each node of G, and a product for each edge of G.
- We then build and array that says customer v bought product e if edge e is incident to the nove n.
- Finally, we ask whether this array has a diverse subset of size k.
- We claim that this holds only if G has an independent set of size k.
- If there is a diverse subset of size k, then the corresponding set of nodes has the property that no two are incident to the same edge - so it is in an independent set of size k.
- Conversely, if there is an independent set of size k, then the corresponding set of customers has the property that no two bought the same product, so it is diverse.

Greedy Algorithms

- A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield optimal solution, but for many problems they do, and if they do, they are usually the most efficient.
- Popular Dijkstra's Shortest Paths algorithm is greedy.
- For most of algorithmic problems one might find some greedy solution, but it is optima only for some special.

One of many 'sum' problems

Given a set S of n positive integers and a positive number W, we want to produce the maximal number of pairs $\{a, b\}$ from S such that $a + b \leq W$. (No member of S may be used twice). For example, if $S = \{8, 9, 6, 1, 4, 7\}$ and $W = 10$), an optimal solution would be $\{\{1, 7\}, \{4, 6\}\}$.

Design an $O(n \log n)$ algorithm for this problem that uses the following *greedy* strategy - repeatedly pick a number a (any way you like) and pair it with the largest element b such that $a + b \leq W$.

Part of your task is to decide exactly how the choices of a will be made.

Prove your algorithm finds an optimal solution.

Solution to the 'sum' problem

Start by sorting the numbers in ascending order. This is the part that costs $O(N \log n)$. The reminder will run in $O(n)$.
For each number a_i (processed in ascending order), pick b_i as described in the problem statement, excluding, of course, umbers that have already been used.
Continue until no b_i is available to be chosen.
Let m be the number of pairs (a_i, b_i) so determined.
Notice that $a_i < b_i$ for each i (since when b_i is chosen, all items smaller than a_i have already been used)

Pseudo-code

```
/*  Given an array S of length n and an integer W
/*  Output the maximum number of pairs (a, b) such that a + b ≤ W

FindMaximumPairs( int W, array S[1..n])
   Sort(S);
   IndexA = 1; //points to the smaller element
   IndexB = n; //points to the larger element
   result = ∅
   while (IndexA ¡ IndexB )
      if (S[InxexA]+S[IndexB] ≤ W)
         result = result ∪ { S[InxexA],S[IndexB] }
         IndexA = IndexA + 1
         IndexB = IndexB - 1
      else IndexB = IndexB - 1
      endif
   endwhile
   return result
end FindMaximimPairs
```

Proof

We claim that this give an optimal solution. This is because the greedy solution " stays ahead" of any other competitive solution. Specifically, suppose that we have an optimal solution O that agrees with the greedy solution on the first $k-1$ pairs, for some integer k with $1 \leq k \leq m$. Here we agree to list the pairs of of the optimal solution in ascending order according to the smaller item of each pair - just like the greedy solution. Then we show that there is a (perheps different) optimal solution that agrees with the greedy solution on the first k points.

Let (a, b) be the kth pair in the optimal solution O.
If $(a, b) = (a_k, b_k)$, there is nothing to do.
Otherwise, $a_k \leq a$ because of the definition of the greedy algorithm, and thus $a_k + b \leq a + b \leq W$. Since b_k was the largest unused element satisfying the inequality, we have $b_k \geq b$.
If O does not use a_k (i.e. $a > a_k$), replace (a, b) by (a_k, b) in O.
We still have an optimal solution, but now (a_k, b) is part of it.
Now, let S_1 be what remains of S after removing the (greedy solution) pairs $\{a_i, b_1\}$ for $1 \leq i \leq k$.
Let S_2 be what remains of after removing the (optimal solution) pairs $\{a_k, b\}$ and $\{a_i, b_i\}$ for $1 \leq i \leq k - 1$.
S_1 and S_2 have $2k - 1$ elements in common and differ only in that S_1 has b and S_2 has b_k. The smallest of these is b.
Thus, for any set of pairs from S_2, there is an equally large set of pairs from S_1 whatever pair involves b_k, if any, can be replace by a pair involving b. Our "greedy pair" $\{a_i, b_i\}$ for $1 \leq i \leq k$ together with any optimal solution for S_1 will be an optimal solution for S.
This completes the proof by induction.

Because of the inductive poof, we now know that there is an optimal solution that agrees with the greedy solution, up to and including the mth pair. Can the optimal solution contain any additional pair? The answer is'no'.

If there were such a pair (a, b), we would have $a > a_m$. Let a' be the smallest unused number. Then $a' + b \leq a + b \leq W$. Let b' be the largest unused number such that $a' + b' \leq W$. Note that $a' \leq a < b \leq W$.

This contradicts the fact that the greedy algorithm terminates at $i = m$. It would have chosen $(m + 1)$st pair (a', b').

We conclude that the greedy algorithm indeed produces an optimal solution.

Algorithms paradigms

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.
- *Dynamic programming* is a fancy name for caching away intermediate results in a table for later reuse.

String similarity

Q. How similar are two strings?

Ex. ocurrance and occurrence.

o	c	u	r	r	a	n	c	e	–
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	–	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	–	u	r	r	–	a	n	c	e
o	c	c	u	r	r	e	–	n	c	e

0 mismatches, 3 gaps

Edit distance

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

cost = $\delta + \alpha_{CG} + \alpha_{TA}$

Applications. Unix diff, speech recognition, computational biology, ...

Sequence alignment

Goal. Given two strings $x_1\, x_2 \ldots x_m$ and $y_1\, y_2 \ldots y_n$ find min cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

$x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$, but $j > j'$

Def. The cost of an alignment M is:

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	–	G
–	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

an alignment of CTACCG and TACATG:

$$M = \{\, x_2\text{–}y_1, x_3\text{–}y_2, x_4\text{–}y_3, x_5\text{–}y_4, x_6\text{–}y_6 \,\}$$

Sequence alignment: problem structure

Def. $OPT(i,j)$ = min cost of aligning prefix strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

Case 1. OPT matches $x_i - y_j$.

Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_{j-1}$.

Case 2a. OPT leaves x_i unmatched.

Pay gap for x_i + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_j$.

Case 2b. OPT leaves y_j unmatched.

Pay gap for y_j + min cost of aligning $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_{j-1}$.

optimal substructure property
(proof via exchange argument)

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence alignment: algorithm

SEQUENCE-ALIGNMENT $(m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha)$

FOR $i = 0$ TO m

$M[i, 0] \leftarrow i\,\delta$.

FOR $j = 0$ TO n

$M[0, j] \leftarrow j\,\delta$.

FOR i = 1 TO m

FOR j = 1 TO n

$$M[i,j] \leftarrow \min \{ \alpha[x_i, y_j] + M[i-1, j-1],$$
$$\delta + M[i-1, j],$$
$$\delta + M[i, j-1]).$$

RETURN $M[m, n]$.

Sequence alignment: analysis

Theorem. The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length m and n in $\Theta(mn)$ time and $\Theta(mn)$ space.

Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▪

Q. Can we avoid using quadratic space?

A. Easy to compute optimal value in $O(mn)$ time and $O(m + n)$ space.

- Compute $\mathrm{OPT}(i, \bullet)$ from $\mathrm{OPT}(i - 1, \bullet)$.
- But, no longer easy to recover optimal alignment itself.

Stanley Cup Odds

Suppose two teams, A and B, are playing a match to see who first win n games for some particular n. The Stanley Cup Series is such a match with $n = 4$. Suppose A has a probability p_i of winning i games (so B has $1 - p_i$ probability of winning i games). Let $P(i, j)$ be the probability that if A needs i games to win, and B needs j games, thatA will eventually win the match. The set of all $P(i, j)$, $i, j = 1, ..., n$, is called a 'table of odds'. Use the dynamic programming technique to design an algorithm that produces such table of odds. Your algorithm should have time complexity not worse that $O(n^2)$. Show the solution for $n = 4$, $p_1 = 0.6$, $p_2 = 0.5$, $p_3 = 0.4$, $p_4 = 0.3$.

Solution

First note that $P(0,j) = 1$ for all j, since this means that A has won the match already, and $P(i,0) = 0$ for all i, since this means that B has won the match already.
Hence create an $n \times n$ array of $P(i,j)$ and fill it with 0 for all $P(i,0)$ and with 1 for all$P(0,j)$.
If team A needs$i$ games to win, then it has won $n - i$ games already. Thus, we obtain the following recurrence for the remaining i and j:

$$P(i,j) = p_{n-i+1}P(i-1,j) + (1 - p_{n-i+1})P(i,j-1).$$

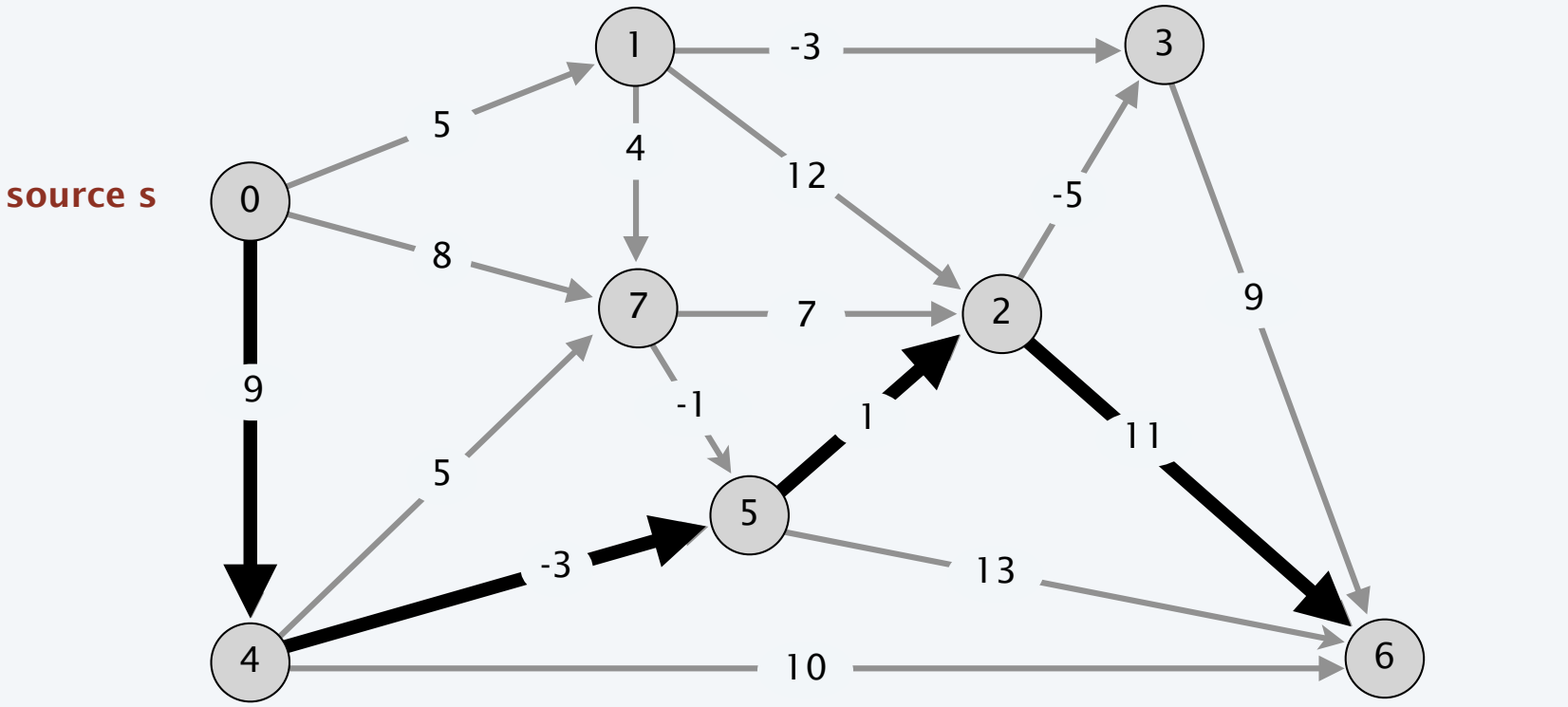
The rest is just a plain calculation.

Dynamic Programming

- Bellman-Ford Algorithm (negative weight allowed)

Shortest paths

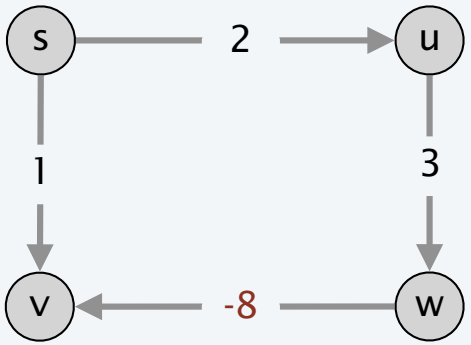
Shortest path problem. Given a digraph $G = (V, E)$, with arbitrary edge weights or costs c_{vw}, find cheapest path from node s to node t.



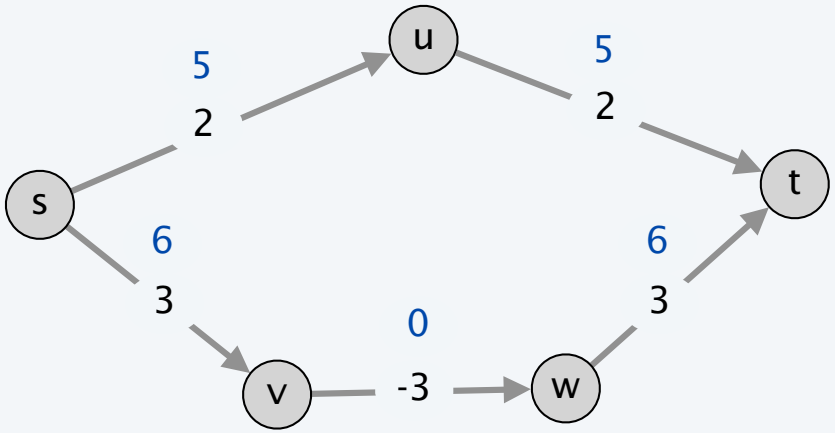
cost of path = 9 − 3 + 1 + 11 = 18

Shortest paths: failed attempts

Dijkstra. Can fail if negative edge weights.

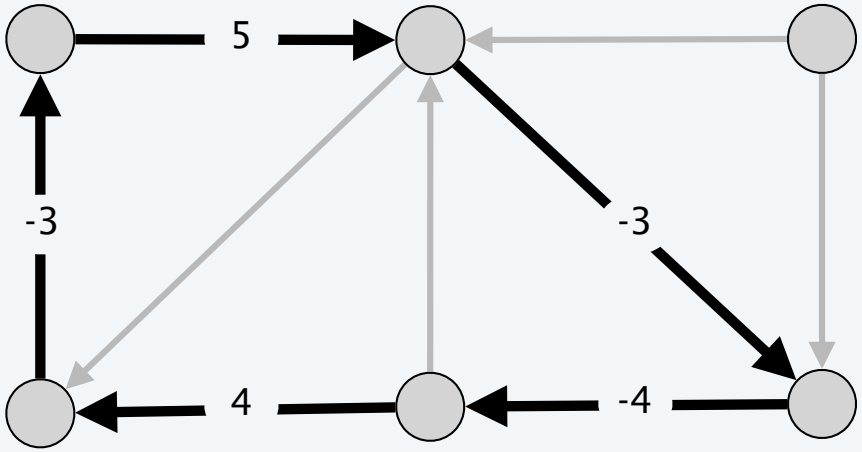


Reweighting. Adding a constant to every edge weight can fail.



Negative cycles

Def. A negative cycle is a directed cycle such that the sum of its edge weights is negative.

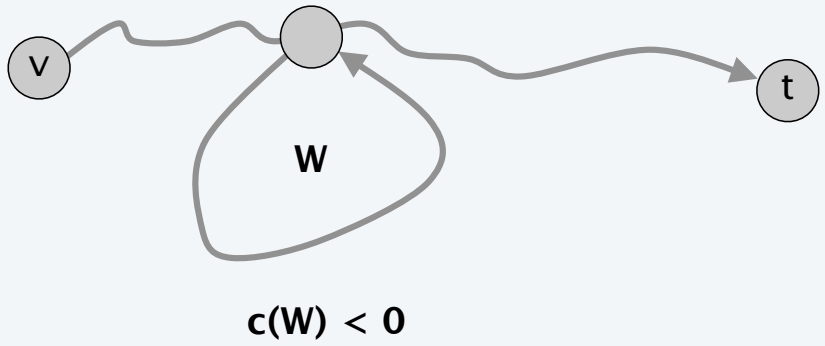


a negative cycle W : $c(W) = \sum_{e \in W} c_e < 0$

Shortest paths and negative cycles

Lemma 1. If some path from v to t contains a negative cycle, then there does not exist a cheapest path from v to t.

Pf. If there exists such a cycle W, then can build a $v \rightarrow t$ path of arbitrarily negative weight by detouring around cycle as many times as desired. ▪

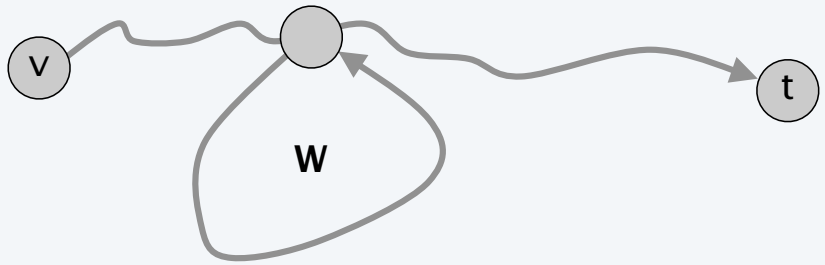


Shortest paths and negative cycles

Lemma 2. If G has no negative cycles, then there exists a cheapest path from v to t that is simple (and has $\leq n - 1$ edges).

Pf.

- Consider a cheapest $v \rightsquigarrow t$ path P that uses the fewest number of edges.
- If P contains a cycle W, can remove portion of P corresponding to W without increasing the cost. ▪

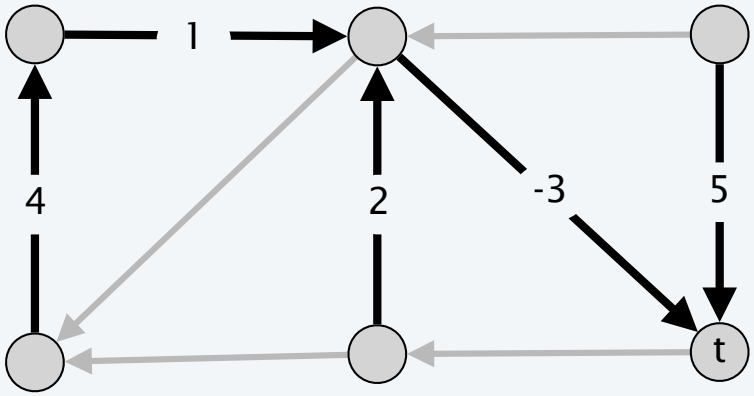


c(W) ≥ 0

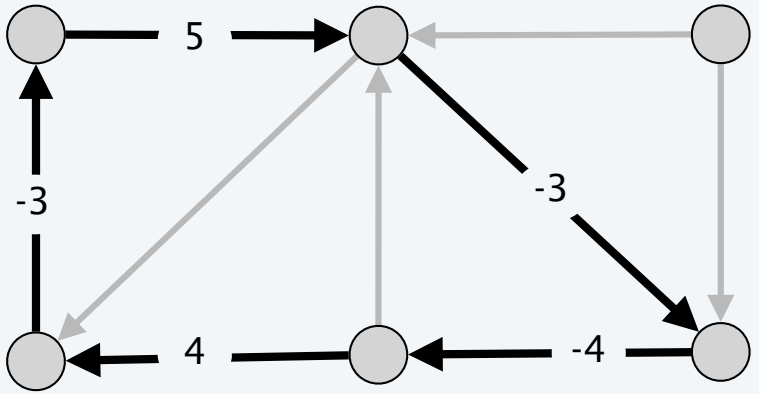
Shortest path and negative cycle problems

Shortest path problem. Given a digraph $G = (V, E)$ with edge weights c_{vw} and no negative cycles, find cheapest $v \rightarrow t$ path for each node v.

Negative cycle problem. Given a digraph $G = (V, E)$ with edge weights c_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Shortest paths: dynamic programming

Def. $OPT(i, v)$ = cost of shortest $v \to t$ path that uses $\le i$ edges.

- Case 1: Cheapest $v \to t$ path uses $\le i - 1$ edges.
  - $OPT(i, v) = \mathrm{OPT}(i - 1, v)$

optimal substructure property
(proof via exchange argument)

- Case 2: Cheapest $v \to t$ path uses exactly i edges.
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best $w \to t$ path using $\le i - 1$ edges

$$OPT(i, v) = \begin{cases} \infty & \text{if } i = 0 \\ \min\left\{ OPT(i-1, v),\ \min_{(v,w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

Observation. If no negative cycles, $OPT(n - 1, v)$ = cost of cheapest $v \to t$ path.

Pf. By Lemma 2, cheapest $v \to t$ path is simple. ▪

Shortest paths: implementation

SHORTEST-PATHS (V, E, c, t)

FOREACH node $v \in V$

  $M[0, v] \leftarrow \infty$.

$M[0, t] \leftarrow 0$.

FOR i = 1 TO $n - 1$

  FOREACH node $v \in V$

    $M[i, v] \leftarrow M[i-1, v]$.

    FOREACH edge $(v, w) \in E$

      $M[i, v] \leftarrow \min \{ M[i, v],\ M[i-1, w] + c_{vw} \}$.

Shortest paths: implementation

Theorem 1. Given a digraph $G = (V, E)$ with no negative cycles, the dynamic programming algorithm computes the cost of the cheapest $v \rightarrow t$ path for each node v in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Pf.

- Table requires $\Theta(n^2)$ space.
- Each iteration i takes $\Theta(m)$ time since we examine each edge once. ▪

Finding the shortest paths.

- Approach 1: Maintain a $successor(i, v)$ that points to next node on cheapest $v \rightarrow t$ path using at most i edges.
- Approach 2: Compute optimal costs $M[i, v]$ and consider only edges with $M[i, v] = M[i - 1, w] + c_{vw}$.

Shortest paths: practical improvements

Space optimization. Maintain two 1d arrays (instead of 2d array).

- $d(v)$ = cost of cheapest $v \rightsquigarrow t$ path that we have found so far.
- $successor(v)$ = next node on a $v \rightsquigarrow t$ path.

Performance optimization. If $d(w)$ was not updated in iteration $i - 1$, then no reason to consider edges entering w in iteration i.

Bellman-Ford: efficient implementation

BELLMAN-FORD (V, E, c, t)

FOREACH node $v \in V$

  $d(v) \leftarrow \infty$.

  $successor(v) \leftarrow null$.

$d(t) \leftarrow 0$.

FOR i = 1 TO $n - 1$

  FOREACH node $w \in V$

    IF ($d(w)$ was updated in previous iteration)

      FOREACH edge $(v, w) \in E$

        IF ($d(v) > d(w) + c_{vw}$)

          $d(v) \leftarrow d(w) + c_{vw}$.

          $successor(v) \leftarrow w$.

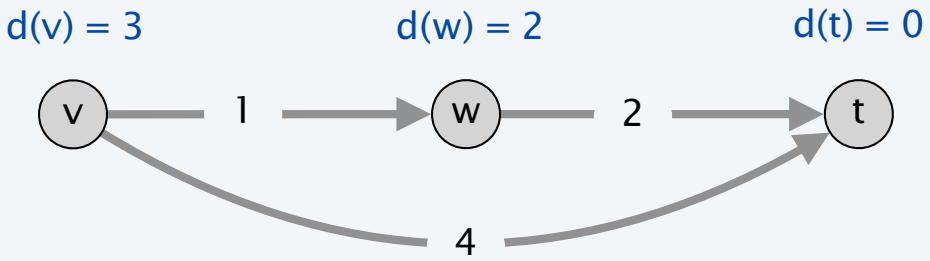
  IF no $d(w)$ value changed in iteration i, STOP.

1 pass

Bellman-Ford: analysis

Claim. After ~~the i^{th} pass of Bellman-Ford, $d(v)$ equals the cost of the cheapest $v \rightarrow t$ path using at most i edges~~.

Counterexample. Claim is false!



if nodes w considered before node v,
then d(v) = 3 after 1 pass

Bellman-Ford: analysis

Lemma 3. Throughout Bellman-Ford algorithm, $d(v)$ is the cost of some $v \rightarrow t$ path; after the i^{th} pass, $d(v)$ is no larger than the cost of the cheapest $v \rightarrow t$ path using $\leq i$ edges.

Pf. [by induction on i]

- Assume true after i^{th} pass.
- Let P be any $v \rightarrow t$ path with $i + 1$ edges.
- Let (v, w) be first edge on path and let P' be subpath from w to t.
- By inductive hypothesis, $d(w) \leq c(P')$ since P' is a $w \rightarrow t$ path with i edges.
- After considering v in pass $i+1$:

$$\begin{aligned} d(v) &\leq c_{vw} + d(w) \\ &\leq c_{vw} + c(P') \\ &= c(P) \quad \blacksquare \end{aligned}$$

Theorem 2. Given a digraph with no negative cycles, Bellman-Ford computes the costs of the cheapest $v \rightarrow t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.

Pf. Lemmas 2 + 3. ▪

can be substantially faster in practice

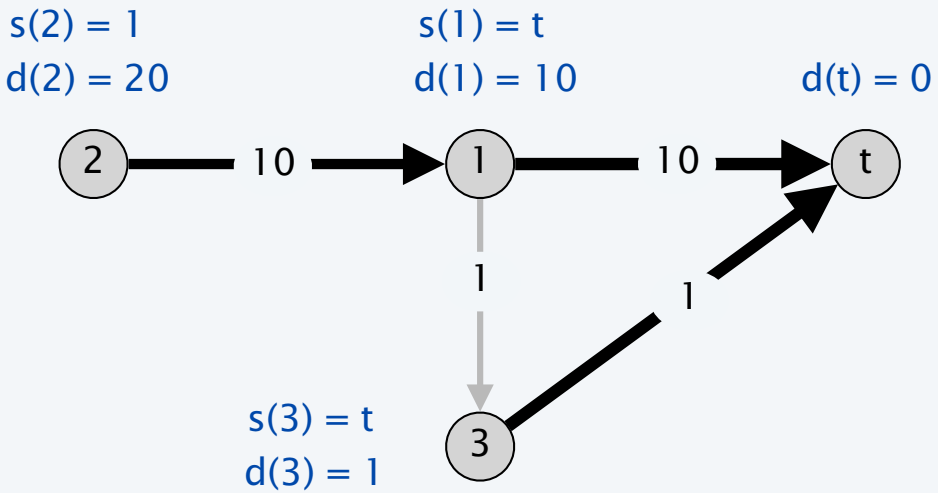
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(v) pointers gives a directed path from v to t of cost $d(v)$.~~

Counterexample. Claim is false!

- Cost of successor $v \rightarrow t$ path may have strictly lower cost than $d(v)$.

consider nodes in order: t, 1, 2, 3



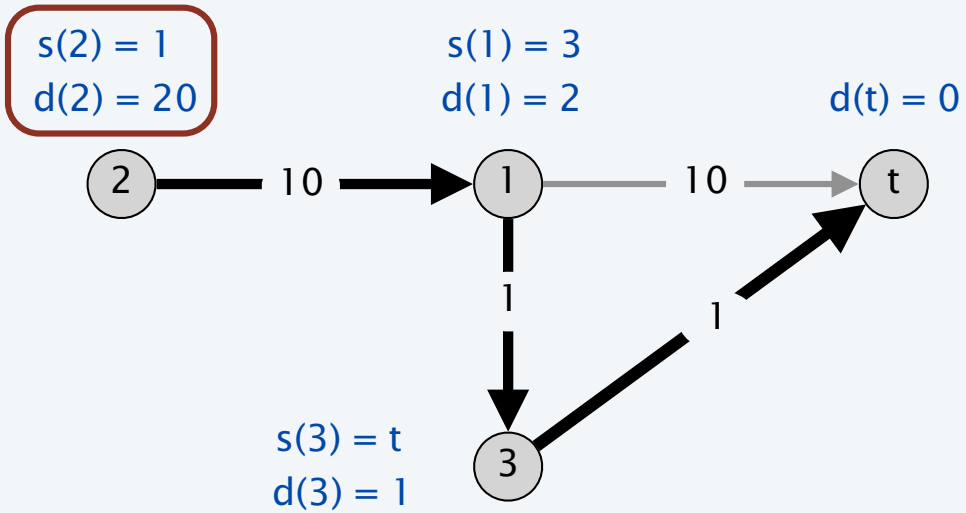
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(v) pointers gives a directed path from v to t of cost $d(v)$.~~

Counterexample. Claim is false!

- Cost of successor $v \rightarrow t$ path may have strictly lower cost than $d(v)$.

consider nodes in order: t, 1, 2, 3



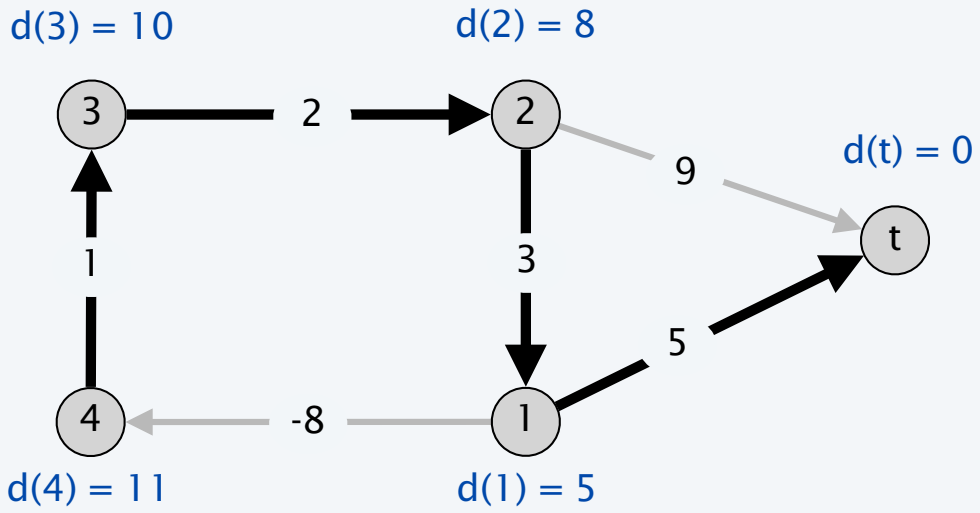
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(v) pointers gives a directed path from v to t of cost $d(v)$.~~

Counterexample. Claim is false!

- Cost of successor $v \rightarrow t$ path may have strictly lower cost than $d(v)$.
- Successor graph may have cycles.

consider nodes in order: t, 1, 2, 3, 4



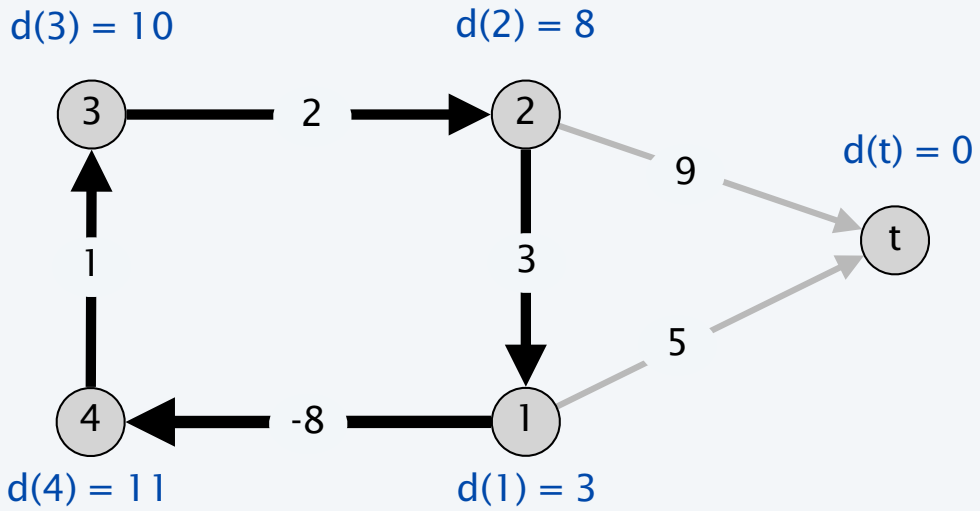
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(v) pointers gives a directed path from v to t of cost $d(v)$.~~

Counterexample. Claim is false!

- Cost of successor $v \rightarrow t$ path may have strictly lower cost than $d(v)$.
- Successor graph may have cycles.

consider nodes in order: t, 1, 2, 3, 4



Bellman-Ford: finding the shortest path

Lemma 4. If the successor graph contains a directed cycle W, then W is a negative cycle.

Pf.

- If $successor(v) = w$, we must have $d(v) \geq d(w) + c_{vw}$.
  (LHS and RHS are equal when $successor(v)$ is set; $d(w)$ can only decrease; $d(v)$ decreases only when $successor(v)$ is reset)
- Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ be the nodes along the cycle W.
- Assume that (v_k, v_1) is the last edge added to the successor graph.
- Just prior to that:

$$\begin{aligned}
d(v_1) &\geq d(v_2) + c(v_1, v_2) \\
d(v_2) &\geq d(v_3) + c(v_2, v_3) \\
&\vdots \quad\quad \vdots \quad\quad \vdots \\
d(v_{k-1}) &\geq d(v_k) + c(v_{k-1}, v_k) \\
d(v_k) &> d(v_1) + c(v_k, v_1)
\end{aligned}$$

holds with strict inequality since we are updating $d(v_k)$

- Adding inequalities yields $c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k) + c(v_k, v_1) < 0$. ▪

W is a negative cycle

Bellman-Ford: finding the shortest path

Theorem 3. Given a digraph with no negative cycles, Bellman-Ford finds the cheapest $s \rightsquigarrow t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.

Pf.

- The successor graph cannot have a negative cycle. [Lemma 4]
- Thus, following the successor pointers from s yields a directed path to t.
- Let $s = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = t$ be the nodes along this path P.
- Upon termination, if $successor(v) = w$, we must have $d(v) = d(w) + c_{vw}$.
  (LHS and RHS are equal when $successor(v)$ is set; $d(\cdot)$ did not change)
- Thus,
- $$\begin{aligned} d(v_1) &= d(v_2) + c(v_1, v_2) \\ d(v_2) &= d(v_3) + c(v_2, v_3) \\ &\vdots \quad\quad \vdots \quad\quad \vdots \\ d(v_{k-1}) &= d(v_k) + c(v_{k-1}, v_k) \end{aligned}$$

since algorithm terminated

Adding equations yields $d(s) = d(t) + c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k)$. ▪

min cost of any s→t path (Theorem 2)

0

cost of path P

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Most common usage.

- Divide problem of size n into two subproblems of size $n/2$ in linear time.
- Solve two subproblems recursively.
- Combine two solutions into overall solution in linear time.

Consequence.

- Other method: $\Theta(n^2)$.
- Divide-and-conquer: $\Theta(n \log n)$.

Counting inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: $1, 2, \ldots, n$.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j are inverted if $i < j$, but $a_i > a_j$.

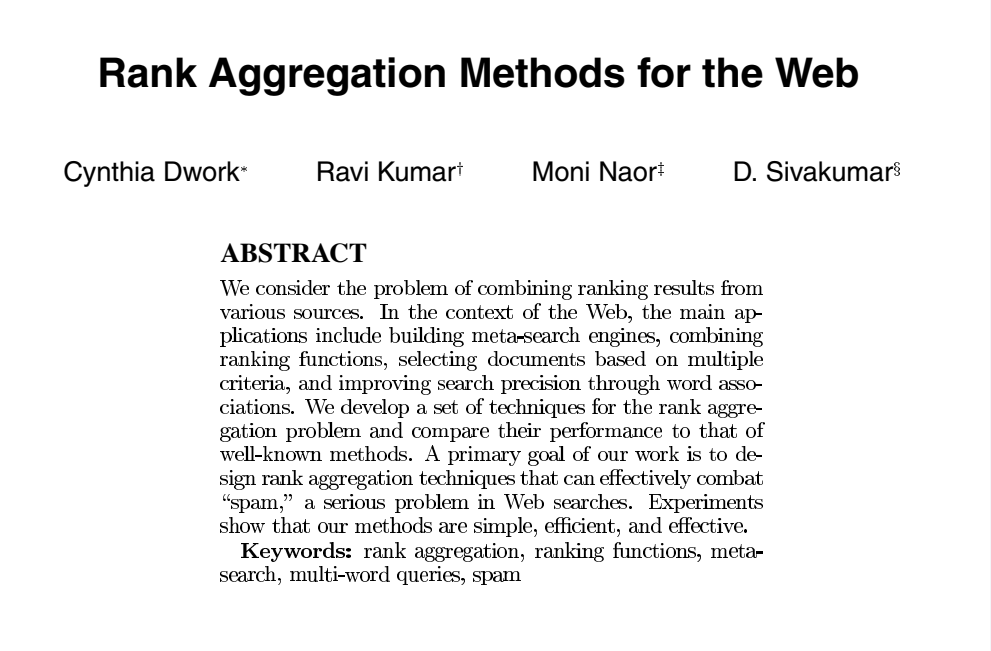
	A	B	C	D	E
me	1	2	3	4	5
you	1	3	4	2	5

2 inversions: 3-2, 4-2

Brute force: check all $\Theta(n^2)$ pairs.

Counting inversions: applications

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's tau distance).



Rank Aggregation Methods for the Web

Cynthia Dwork* Ravi Kumar† Moni Naor‡ D. Sivakumar§

ABSTRACT

We consider the problem of combining ranking results from various sources. In the context of the Web, the main applications include building meta-search engines, combining ranking functions, selecting documents based on multiple criteria, and improving search precision through word associations. We develop a set of techniques for the rank aggregation problem and compare their performance to that of well-known methods. A primary goal of our work is to design rank aggregation techniques that can effectively combat "spam," a serious problem in Web searches. Experiments show that our methods are simple, efficient, and effective.

Keywords: rank aggregation, ranking functions, meta-search, multi-word queries, spam

Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with $a \in A$ and $b \in B$.
- Return sum of three counts.

input

1	5	4	8	10	2	6	9	3	7

count inversions in left half A

1	5	4	8	10

5-4

count inversions in right half B

2	6	9	3	7

6-3 9-3 9-7

count inversions (a, b) with a ∈ A and b ∈ B

1	5	4	8	10

2	6	9	3	7

4-2 4-3 5-2 5-3 8-2 8-3 8-6 8-7 10-2 10-3 10-6 10-7 10-9

output 1 + 3 + 13 = 17

Counting inversions: how to combine two subproblems?

Q. How to count inversions (a, b) with $a \in A$ and $b \in B$?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element $b \in B$,
  - binary search in A to find how elements in A are greater than b.

list A

7	10	18	3	14

list B

17	23	2	11	16

sort A

3	7	10	14	18

sort B

2	11	16	17	23

binary search to count inversions (a, b) with a ∈ A and b ∈ B

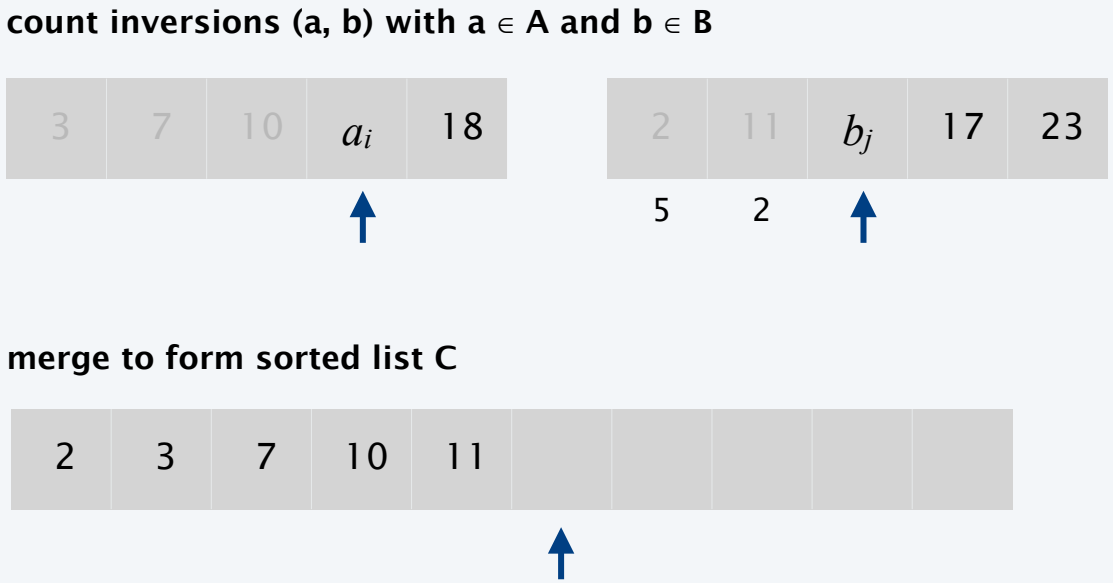
3	7	10	14	18

2	11	16	17	23
5	2	1	1	0

Counting inversions: how to combine two subproblems?

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.



Counting inversions: divide-and-conquer algorithm implementation

Input. List L.

Output. Number of inversions in L and sorted list of elements L'.

SORT-AND-COUNT (L)

IF list L has one element

    RETURN $(0, L)$.

DIVIDE the list into two halves A and B.

$(r_A , A) \leftarrow$ SORT-AND-COUNT(A).

$(r_B , B) \leftarrow$ SORT-AND-COUNT(B).

$(r_{AB} , L') \leftarrow$ MERGE-AND-COUNT(A, B).

RETURN $(r_A + r_B + r_{AB} , L')$.

Counting inversions: divide-and-conquer algorithm analysis

Proposition. The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.

Pf. The worst-case running time $T(n)$ satisfies the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{otherwise} \end{cases}$$

Integer multiplication

- Multiplication. Given two n-bit integers a and b, compute a × b.
- Grade-school algorithm. $\Theta(n^2)$ bit operations.

						1	1	0	1	0	1	0	1
					×	0	1	1	1	1	1	0	1
						1	1	0	1	0	1	0	1
					0	0	0	0	0	0	0	0	
				1	1	0	1	0	1	0	1		
			1	1	0	1	0	1	0	1			
		1	1	0	1	0	1	0	1				
	1	1	0	1	0	1	0	1					
0	0	0	0	0	0	0	0						
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1

- Question. Is grade-school algorithm optimal?
- No.

Divide-and-conquer multiplication

To multiply two n-bit integers x and y:

- Divide x and y into low- and high-order bits.
  Example. $x = \underbrace{1000}_{a}\underbrace{1101}_{b}$ $y = \underbrace{1110}_{c}\underbrace{0001}_{d}$
  $m = \lceil n/2 \rceil$
  $a = \lfloor x/2^m \rfloor \quad b = x \mod 2^m$
  $c = \lfloor y/2^m \rfloor \quad d = y \mod 2^m$
  Bit shifting can be used to compute a, b, c and d.
- Now we have: $x = 2^m a + b$ and $y = 2^m c + d$.
- Multiply four $\frac{1}{2}n$-bit integers, recursively.
- Add and shift to obtain result.

$$xy = (2^m a + b)(2^m c + d) = 2^{2m} \underbrace{ac}_{1} + 2^m (\underbrace{bc}_{2} + \underbrace{ad}_{3}) + \underbrace{bd}_{4}$$

Divide-and-conquer multiplication

MULTIPLY(x, y, n)

IF $(n = 1)$

  RETURN $x \times y$.

ELSE

  $m \leftarrow \lceil\, n / 2 \,\rceil$.

  $a \leftarrow \lfloor\, x / 2^m \,\rfloor$; $b \leftarrow x \bmod 2^m$.

  $c \leftarrow \lfloor\, y / 2^m \,\rfloor$; $d \leftarrow y \bmod 2^m$.

  $e \leftarrow$ MULTIPLY(a, c, m).

  $f \leftarrow$ MULTIPLY(b, d, m).

  $g \leftarrow$ MULTIPLY(b, c, m).

  $h \leftarrow$ MULTIPLY(a, d, m).

  RETURN $2^{2m}\, e + 2^m\, (g + h) + f$.

Divide-and-conquer multiplication analysis

Proposition

The divide-and-conquer multiplication algorithm requires $\Theta(n^2)$ bit operations to multiply two n-bit integers.

Proof.

Apply case 1 of the master theorem to the recurrence:

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \implies T(n) = \Theta(n^2)$$

□

Not better than grade-school algorithm!

Karatsuba trick

$x = \underbrace{1000}_{a}\underbrace{1101}_{b}$ $y = \underbrace{1110}_{c}\underbrace{0001}_{d}$

$m = \lceil n/2 \rceil$

$a = \lfloor x/2^m \rfloor \quad b = x \mod 2^m$

$c = \lfloor y/2^m \rfloor \quad d = y \mod 2^m$

$$xy = (2^m a + b)(2^m c + d) = 2^{2m} \underbrace{ac}_{1} + 2^m \overbrace{(\underbrace{bc}_{2} + \underbrace{ad}_{3})}^{\text{middle term}} + \underbrace{bd}_{4}$$

- To compute middle term $bc + ad$, use identity:
  $bc + ad = ac + bd - (a - b)(c - d)$

Now we have:

$$xy = 2^{2m} \underbrace{ac}_{1} + 2^m(\underbrace{ac}_{1} + \underbrace{bd}_{2} - \underbrace{(a - b)(c - d)}_{3}) + \underbrace{bd}_{2}$$

Bottom line. Only three multiplications of $\frac{1}{2}$-bit integers!

Karatsuba (divide-and-conquer) multiplication

KARATSUBA-MULTIPLY(x, y, n)

IF $(n = 1)$

  RETURN $x \times y$.

ELSE

  $m \leftarrow \lceil n / 2 \rceil$.

  $a \leftarrow \lfloor x / 2^m \rfloor$; $b \leftarrow x \bmod 2^m$.

  $c \leftarrow \lfloor y / 2^m \rfloor$; $d \leftarrow y \bmod 2^m$.

  $e \leftarrow$ KARATSUBA-MULTIPLY(a, c, m).

  $f \leftarrow$ KARATSUBA-MULTIPLY(b, d, m).

  $g \leftarrow$ KARATSUBA-MULTIPLY$(a - b, c - d, m)$.

  RETURN $2^{2m} e + 2^m (e + f - g) + f$.

Karatsuba analysis

Proposition

Karatsuba's multiplication algorithm requires $\Theta(n^{1.585})$ bit operations to multiply two n-bit integers.

Proof.

Apply case 1 of the master theorem to the recurrence:

$$T(n) = \underbrace{3T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \implies T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585})$$

□

Practice. Faster than grade-school algorithm for about 320-640 bits.

Dot product

Dot product. Given two length n vectors a and b, compute $c = a \cdot b$.

Grade-school. $\Theta(n)$ arithmetic operations.

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

$$a = \begin{bmatrix} .70 & .20 & .10 \end{bmatrix}$$

$$b = \begin{bmatrix} .30 & .40 & .30 \end{bmatrix}$$

$$a \cdot b = (.70 \times .30) + (.20 \times .40) + (.10 \times .30) = .32$$

Remark. Grade-school dot product algorithm is asymptotically optimal.

Matrix multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $C = AB$.

Grade-school. $\Theta(n^3)$ arithmetic operations.

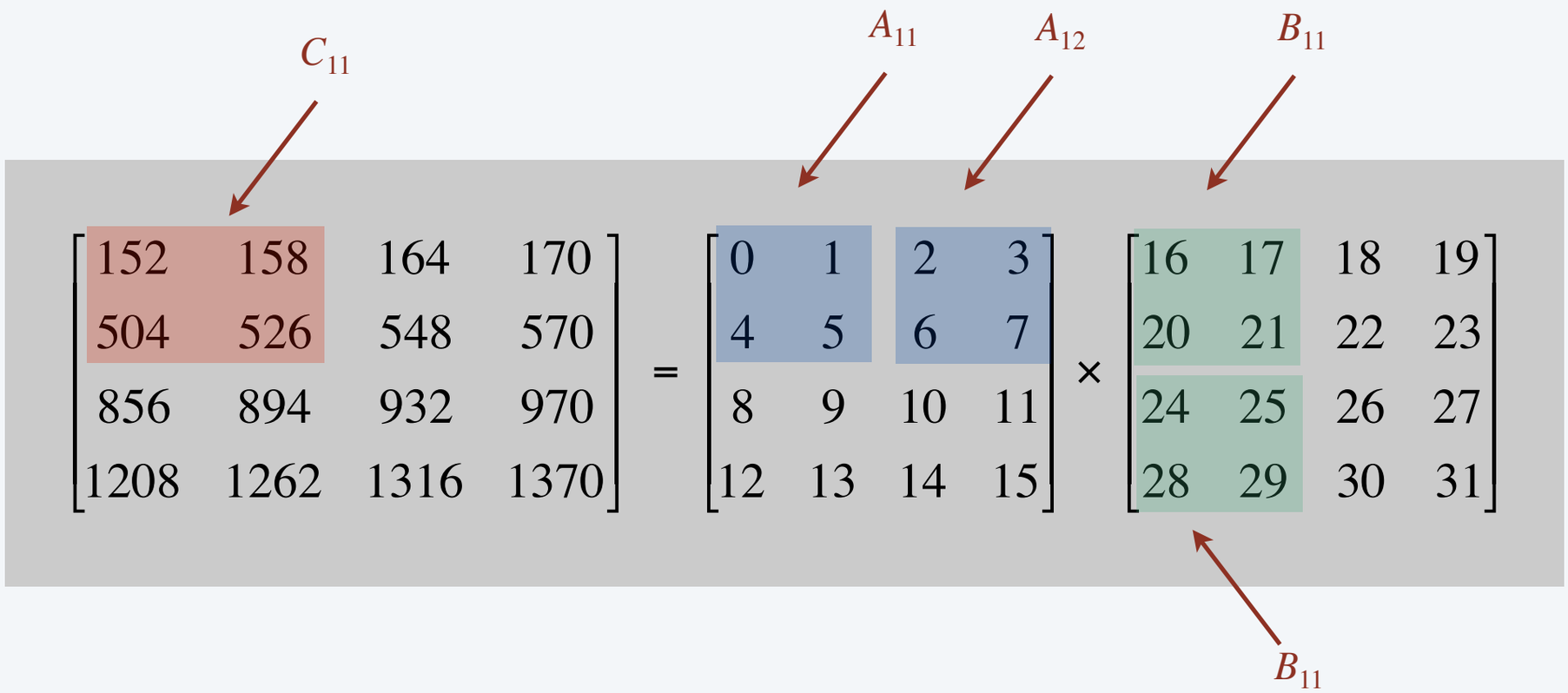
$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

Q. Is grade-school matrix multiplication algorithm asymptotically optimal?

Block matrix multiplication



$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

Matrix multiplication: warmup

To multiply two n-by-n matrices A and B:

- Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Conquer: multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

Running time. Apply case 1 of Master Theorem.

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Strassen's trick

Key idea. multiply 2-by-2 blocks with only 7 multiplications.
(plus 11 additions and 7 subtractions)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_1 + P_5 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &\leftarrow A_{11} \times (B_{12} - B_{22}) \\ P_2 &\leftarrow (A_{11} + A_{12}) \times B_{22} \\ P_3 &\leftarrow (A_{21} + A_{22}) \times B_{11} \\ P_4 &\leftarrow A_{22} \times (B_{21} - B_{11}) \\ P_5 &\leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &\leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &\leftarrow (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

Pf. $C_{12} = P_1 + P_2$

$= A_{11} \times (B_{12} - B_{22}) + (A_{11} + A_{12}) \times B_{22}$

$= A_{11} \times B_{12} + A_{12} \times B_{22}$. ✔

Strassen's algorithm

assume n is a power of 2

STRASSEN (n, A, B)

IF $(n = 1)$ RETURN $A \times B$.

Partition A and B into 2-by-2 block matrices.

$P_1 \leftarrow$ STRASSEN $(n / 2, A_{11}, (B_{12} - B_{22}))$.

$P_2 \leftarrow$ STRASSEN $(n / 2, (A_{11} + A_{12}), B_{22})$.

$P_3 \leftarrow$ STRASSEN $(n / 2, (A_{21} + A_{22}), B_{11})$.

$P_4 \leftarrow$ STRASSEN $(n / 2, A_{22}, (B_{21} - B_{11}))$.

$P_5 \leftarrow$ STRASSEN $(n / 2, (A_{11} + A_{22}) \times (B_{11} + B_{22}))$.

$P_6 \leftarrow$ STRASSEN $(n / 2, (A_{12} - A_{22}) \times (B_{21} + B_{22}))$.

$P_7 \leftarrow$ STRASSEN $(n / 2, (A_{11} - A_{21}) \times (B_{11} + B_{12}))$.

$C_{11} = P_5 + P_4 - P_2 + P_6$.

$C_{12} = P_1 + P_2$.

$C_{21} = P_3 + P_4$.

$C_{22} = P_1 + P_5 - P_3 - P_7$.

RETURN C.

keep track of indices of submatrices (don't copy matrix entries)

Analysis of Strassen's algorithm

Theorem. Strassen's algorithm requires $O(n^{2.81})$ arithmetic operations to multiply two n-by-n matrices.

Pf. Apply case 1 of the master theorem to the recurrence:

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Q. What if n is not a power of 2?

A. Could pad matrices with zeros.

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 10 & 11 & 12 & 0 \\ 13 & 14 & 15 & 0 \\ 16 & 17 & 18 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 84 & 90 & 96 & 0 \\ 201 & 216 & 231 & 0 \\ 318 & 342 & 366 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Strassen's algorithm: practice

Implementation issues.

- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm when n is "small" .

Common misperception. *"Strassen is only a theoretical curiosity."*

- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2{,}048$.
- Range of instances where it's useful is a subject of controversy.

Linear algebra reductions

Matrix multiplication. Given two n-by-n matrices, compute their product.

problem	linear algebra	order of growth
matrix multiplication	$A \times B$	$\Theta(MM(n))$
matrix inversion	A^{-1}	$\Theta(MM(n))$
determinant	$\lvert A \rvert$	$\Theta(MM(n))$
system of linear equations	$Ax = b$	$\Theta(MM(n))$
LU decomposition	$A = L\,U$	$\Theta(MM(n))$
least squares	$\min \lVert Ax - b \rVert_2$	$\Theta(MM(n))$

numerical linear algebra problems with the same complexity as matrix multiplication

Fast matrix multiplication: theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?

A. Yes! [Strassen 1969] $\Theta(n^{\log_2 7}) = O(n^{2.807})$

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?

A. Impossible. [Hopcroft and Kerr 1971] $\Theta(n^{\log_2 6}) = O(n^{2.59})$

Q. Multiply two 3-by-3 matrices with 21 scalar multiplications?

A. Unknown. $\Theta(n^{\log_3 21}) = O(n^{2.77})$

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, ...]

- Two 20-by-20 matrices with 4,460 scalar multiplications. $O(n^{2.805})$
- Two 48-by-48 matrices with 47,217 scalar multiplications. $O(n^{2.7801})$
- A year later. $O(n^{2.7799})$
- December 1979. $O(n^{2.521813})$
- January 1980. $O(n^{2.521801})$

History of asymptotic complexity of matrix multiplication

year	algorithm	order of growth
?	brute force	$O(n^3)$
1969	Strassen	$O(n^{2.808})$
1978	Pan	$O(n^{2.796})$
1979	Bini	$O(n^{2.780})$
1981	Schönhage	$O(n^{2.522})$
1982	Romani	$O(n^{2.517})$
1982	Coppersmith-Winograd	$O(n^{2.496})$
1986	Strassen	$O(n^{2.479})$
1989	Coppersmith-Winograd	$O(n^{2.376})$
2010	Strother	$O(n^{2.3737})$
2011	Williams	$O(n^{2.3727})$
?	?	$O(n^{2+\varepsilon})$

number of floating-point operations to multiply two n-by-n matrices

Approximation Algorithms

- Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation.
- They're currently doing a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.
- Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, w_2, \ldots, w_n$.
- Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.)
- You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimize the number of trucks that are needed in order to carry all the containers.
- This problem is NP-complete (you don't have to prove this).

- A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit.
- Now declare this truck "loaded" and send it off; then continue the process with a fresh truck.
- This algorithm, by considering trucks one at a time, may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

(a) Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

(b) Show, however, that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

- A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit.

(a) Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

- *Solution*

Let $w_1 = 1, w_2 = 2, w_3 = 1$ and $K = 2$.
Then the greedy algorithm here will use three trucks, whereas there is away to use just two, as $w_1 + w_3 = 2$, $w_1 + w_3 + w_2 = 4$ and $2 + 2 = 4$.
In a sense this is a typical case where greediness does not work!

- A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit.

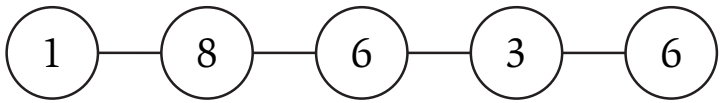
(b) Show, however, that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

- *Solution*

Let $W = \sum_i w_i$. Note that in *any* solution, each truck holds at most K units of weight, so W/K is a lower bound on the number of trucks needed.
Suppose the number of trucks used by our greedy algorithm is an odd number $m = 2q + 1$ (the case when m is even is essentially the same, a little easier).
Divide the trucks used into consecutive groups of two, for a total $q + 1$ groups (the last group contains only one truck).
In each group but the last, the total weight of containers must be *strictly* greater than K (else, the second truck in group would not have been started then) - thus, $W > 1K$, and so $W/K > q$.
It follows by our argument above that the optimum solution uses at least $q + 1$ trucks, which is within a factor of 2 of $m = 2q + 1$.

- Which strategy/approach should be used?

Let $G = (V, E)$ be an undirected graph with n nodes.
A subset of the nodes is called an independent set if no two of them are joined by an edge.
Finding large independent sets is difficult in general (the problem is NP-complete); but here we'll see that it can be done efficiently if the graph is "simple" enough.
Call a graph $G = (V, E)$ a path if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1.
With each node v_i, we associate a positive integer weight w_i.
Consider, for example, the five-node path drawn below.



The weights are the numbers drawn inside the nodes. The maximum weight of an independent set is 14.
The goal in this question is to solve the following problem:
Find an independent set in a path G whose total weight is as large as possible.

Greedy

- Give an example to show that the following algorithm does not always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
Start with S equal to the empty set
While some node remains in G
Pick a node v_i of maximum weight
Add v_i to S
Delete v_i and its neighbours from G
Endwhile
Return S
```

- *Solution*

Consider the sequence of weights $2, 3, 2$.
The greedy algorithm will pick the middle node, while the maximum weight independent set consists with the first and the third.
So, the greedy approach does not work.

Divide and Conquer

- Give an example to show that the following algorithm also does not always find an independent set of maximum total weight.

  Let S_1 be the set of all v_i where i is an odd number
  Let S_2 be the set of all v_i where i is an even number
  (Note that S_1 and S_2 are both independent sets)
  Determine which of S_1 or S_2 has greater total weight, and return this one

- *Solution*

Consider the sequence of weights $3, 1, 2, 3$.
The given algorithm will pick up the first and the third nodes, while the maximum weight independent set consists of the first and the fourth.

Dynamic Programming

Let S_i denote an independent set on $\{v_1, \ldots, v_i\}$, and let X_i denote its weight.
Define $X_0 = 0$ and note that $X_1 = w_1$.
Now, for $i > 1$, either v_i belongs to S_i, or it doesn't.
In the first case, we know that v_{i-1} cannot belong to S_i, and so $X_i = w_i + X_{i-2}$.
In the second case, $X_i = X_{i-1}$.
Thus we have the recurrence:

$$X_i = \max(X_{i-1}, w_i + X_{i-2}).$$

We thus compute the values of X_i, in increasing order from $i = 1$ to n.
X_n is the value want, and we can compute S_n by tracting back through the computation of the max operator.
Since we spend constant time per iteration, over n iterations, the total running time is $O(n)$.

- Randomized Algorithms
- They are often simple and efficient, but do not guarantee optimal solution.

Basic Probability

- A and B are *mutually exclusive* if $p(A \cap B) = \emptyset$.
- A and B are *independent* if $p(A \cap B) = p(A)p(B)$.
- *Conditional probability*: A and B are events and $p(B) > 0$
  The *probability of A given B* is defined as:

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

Random Variables

- A function $X : S \to \mathit{Reals}$ is called a *random variable*.
- The *expectation* (*expected value*) of X is:

$$E(X) = \sum_{x \in S} p(x)X(x)$$

- For $x \in \mathit{Reals}$,

$$p(\{x \in S \mid X(x) = r\}) = p(X^{-1}(r))$$

is often written as $p(X = r)$ and interpreted as "the probability that $X = r$".

Fact

$$E(X) = \sum_{r \in X(S)} r \cdot p(X = r)$$

Fact

Expected value is linear *i.e.*

1. $E(X+Y) = E(X) + E(Y)$
2. $E(cX) = cE(X)$
3. $E(\sum_i c_i X_i) = \sum_i c_i E(X_i)$

- Waiting for first success in independent trial:
$$P(X = j) = (1-p)^{j-1}p$$
- $E(X) = \sum_{j=1}^{\infty} jP(X=j) = \sum_{j=1}^{\infty} j(1-p)^{j-1}p =$
$$\frac{p}{1-p}\sum_{j=1}^{\infty} j(1-p)^j = \frac{p}{1-p}\cdot\frac{1-p}{p^2} = \frac{1}{p}$$
since $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ for $|x| < 1$.
- Expected number of trials to first success is $\frac{1}{p}$.

Maximal 3-Colouring

3-Coluoring is a yes/no question, but we can phrase it as an optimization problem as follows.
Suppose we are given a graph $G = (V, E)$, and we want to colour each node with one of three colours, even if we aren't necessarily able to give different colours to every pair of adjacent nodes.
Rather, we say that an edge (u, v) is *satisfied* if the colours assigned to u and v are different.
Consider a 3-colouring that maximizes the number of satisfied edges, and let c^* denote this number.
Give a polynomial-time algorithm that produces a 3-colouring that satisfies at least $(2/3)c^*$. edges.
If you want, your algorithm can be randomized; in this case, the expected number of edges it satisfies should be at least $(2/3)c^*$.

Solution

As this is a maximization problem, we need an upper bound of c^*, and the is an easy one: $c^* \leq m$, where $m = |E|$.
The algorithm is: colouring every node independently with one of the three colours, each with probability $\frac{1}{3}$.
Let random variable

$$X_e = \begin{cases} 1 & \text{edge } e \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}$$

Then for any given edge e, there are 9 ways to colour its two ends, each of which appears with the same probability, and 3 of them are not satisfying, i.e.

$$Exp[X_e] = Pr[e \text{ is satisfied}] = \frac{6}{9} = \frac{2}{3}.$$

Let Y be the random cariable denoting the number of satisfied edges, then by linearity of expectations,

$$Exp[Y] = Exp[\sum_{e \in E} X_e] = \sum_{e \in E} Exp[x_e] = \frac{2}{3}m \geq \frac{2}{3}c^*.$$

Hence, $Exp[Y] \geq \frac{2}{3}c^*$.