ALGOS PART 4 WEEK 11

SINGLE SOURCE SHORTEST PATH, WITH DP

Input: directed graph $G = (V, E)$, edge lengths c_e, source $= s$.
(assume no parallel edges)

Goal: for every destination $v \in V$, compute the length of the shortest s, v path.

Dijkstra + Heaps is very fast and good, but as mentioned in the Intro lecture, sometimes you can't apply it. Also fails on negative edge lengths. (So we'll use Bellman-Ford)

ON NEGATIVE CYCLES

Q: How do we even define shortest paths when G has a negative cycle? If we looped through it forever we'd get to $-\infty$.

We can compute the shortest cycle-free s-v path.

Problem: NP-hard (no polynomial algorithm, unless $P = NP$) (unless $P = NP$ is what)

Solution for now: Assume G has no negative cycles

Later: will show you how to quickly check this condition

It will either output "negative cycles!" or run fine.

2)

WEEK 2.2

SEQUENCE ALIGNMENT

Input: two strings over the alphabet {A, C, G, T} (portions of on or more genes)

Problem: How "similar" are they?

Application examples: ① Extrapolate function of genome substrings
② Similar genomes to infer proximity in evolutionary tree

Example:
A G G G C T
A G G C A
(one more match after gap; mismatches after gap; insert gap here)

"Penalties for mismatches and gaps!"

Input

PROBLEM

Input: 2 strings over {A,C,G,T}, penalties $pen_{gap} \geq 0$ and $pen_{AT} \geq 0$

Output: alignment of strings that minimizes the total penalty

total penalty is called Needleman-Wunsch score

We need an efficient way to find it, and we can't Brute Force it

A 500 letter string has more alignments than atoms in the universe (10^{125})

Some Algorithm designs:

- Divide and conquer
- Randomized algorithms
- Eg. Greedy algorithms
- Dynamic Programming.

Greedy algorithms make "myopic" decisions and hope everything works out at the end. Process every data point once

[redacted]

GREEDY VS DIVIDE & CONQUER

(1) Easy to propose multiple greedy algorithms for problems

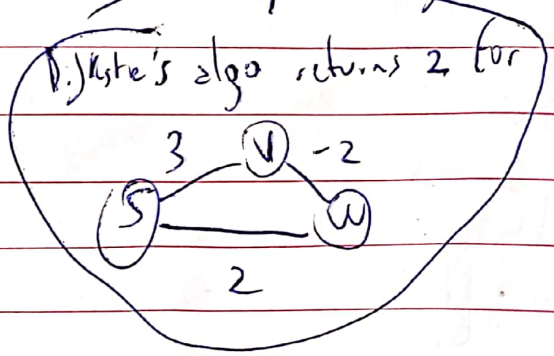
(2) And to calculate their running time too

(3) Hard to establish correctness, even ~~though~~ if you ~~might~~ find a correct one you'll have a hard time understanding why.

Dijkstra's algorithm ~~was~~ is easy to prove ish with positive lengths

It's easy to prove incorrectness!

Dijkstra's algo returns 2 for



Three strategies for Correctness:

(1) Induction

(2) Exchange (show it can be exchanged for an optimal solution)

(3) Whatever works!

1st WEEKLY

Recall: THE CACHING PROBLEM

- small fast memory (the cache)
- big slow memory
- process sequence of "page requests"
- on a fault, what do you evict from cache?

Example: Cache | a | b | c | d |

Request sequence: c ✓

d ✓

e ✗ what to do? We have to load e, but what do we evict?

Let's say we evict a: | e | b | c | d |

f ✗

we evict b: | e | f | c | d |

a ✗

b ✗

So in the end we had request sequence c d e f a b and 4 fails we could have gotten only 2 if we ~~evicted~~ evicted c and d

What eviction algorithm to choose?

THE OPTIMAL CACHING ALGORITHM

Theorem: [Belady 1960] The "furthest-in-future" algorithm is optimal (but what is it?)

Good benchmark for caching algorithms. LRU (least recently used) should work well enough as long as data exhibits locality of reference

WEEK 1 L2

OPTIMAL SUBSTRUCTURE

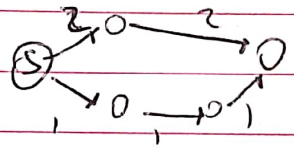
We'll study and understand how solutions are composed of subproblems

Intuition: we can exploit the sequential nature of paths
A subpath of a shortest path should itself be the shortest

Issue: It is not clear how to define "smaller" or "larger" subproblems until you've solved the problem itself

Key idea: artificially restrict number of edges in a path

Example:



If we restrict the max number of edges to 2 the shortest path increases from 3 to 4.

Lemma: Let $G(V,E)$ be a directed graph with edge lengths c_e and source vertex s. (We allow for negative cycles)

For every $v \in V$, $i \in \{1,2,3,\dots\}$, Let P = shortest path with at most i edges. (cycles permitted)

Case (1): if P has $\leq (i-1)$ edges, it is the shortest s-v path with $\leq (i-1)$ edges

Case (2): if P has i edges, delete the last one (w,v) to get P'.
P' is the shortest s-w path with $\leq (i-1)$ edges.

There are $(1 + \text{in-degree}(v))$ (computed by DFS) candidates for a path from s to v.

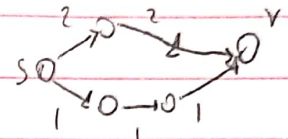
This can be deduced from (1)

WEEK 1 L3

THE RECURRENCE

Notation: Let $L_{i,v}$ = min length of s-v path with (i) edges

(i: permitted edges; v: destination)



$L_{3,v} = \min \{ 4, \min (\}$

Recurrence: for every $v \in V$, $i \in \{1, 2, 3, \ldots\}$

$$L_{i,v} = \min \begin{cases} L_{(i-1),v} & \text{(for case ①)} \\ \min_{(w,v) \in E} \{ L_{(i-1),w} + C_{wv} \} & \text{(for case ②)} \end{cases}$$

We are effectively brute forcing through all possible candidates that respect the opt. substructure lemma

IF NO NEG CYCLES,

- Shortest paths don't have cycles
- Paths have at most $\#e \leq (n-1)$

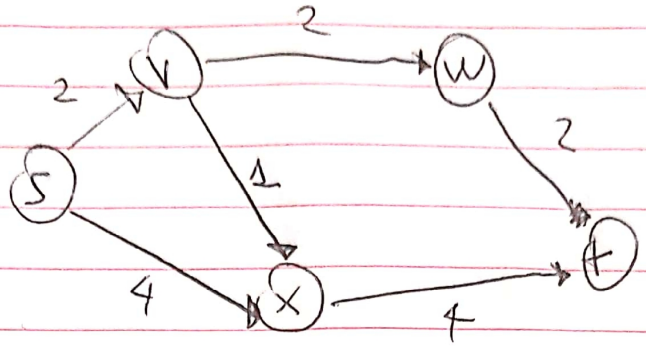
THE BELLMAN-FORD ALGORITHM

Let A = 2D Array (indexed by i and v)

Base case: $A[0,s] = 0$, $A[0,v] = +\infty$ (Host(inf)?) for all $v \neq s$.

- For i in range $(1, n)$:
  - For each $v \in V$
    - $A[i,v] = \min \begin{cases} A[i-1,v] \\ \min_{(w,v) \in E} \{ A[i-1,w] + C_{wv} \} \end{cases}$

Example:



$i=0$ nothing

$i=1$ $\min\{\min\{\infty, 0+2, 0+4\}\} = \min\{\infty, 2\} = 2$ $A[1,v]=2$, $A[1,x]=4$

$i=2$ for $v=s$, stays the same. for $v=x$, $\min\{4, \min(2+1, 2+2)\} \Rightarrow A[2,x]=3$

for $v=w$, $A[2,w]=4$

$i=3$ no change at t, $A[3,t]=\min\{6, 7\} = 6$.

The running time is $O(nm)$, n vertexes to consider, m edges "each time"

Note: suppose for some $j < n-1$, $A[j,v] = A[j-1,v]$ for all vertices v.
- can safely halt!

WEEK 1 L4

HOW TO CHECK FOR NEG CYCLES?

Claim: G has no neg cycle $\iff$ In the extended B-Ford Algo, $A[n-1, v] = A[n, v]$ for all $v \in V$

Consequence: we can check for neg cycles just by running BF again!

In a cycle C, $\sum_{(w,v)\in C} c_{wv} \geq \sum_{(w,v)\in C} (d(w) - d(v))$

If there was any in G, it would break $\min \begin{cases} A[n-1,v] \\ \min_{(w,v)\in E} \{A[n-1,w] + c_{wv}\} \end{cases}$

this bit

WEEK 1 LS

OPTIMIZING SPACE NEEDED FOR B-FORD

If we use an array with $i = (n-1)$ columns and n rows, $O(n^2)$ space

How can we do better? We want to have $O(n)$ space

We really only need the $A[i-1, v]$'s to be stored to compute $A[i, v]$'s

- We can remember the current and last round of subproblems in $O(n)$ space

To reconstruct the paths without a table,

- Compute in $O(n)$ a second table B, where $B[i,v]$ = second to last vertex on shortest sv path with $\leq i$ edges
- From the last column $B[i,v]$, we can find the previous ones from 2 cases!
  - if no hop (case 2) $B[i,v] = B[i-1, v]$
  - if hop subtract it! ← + hop.

WEEK 1 L8

ALL PAIRS SHORTEST PATHS - Problem Definition

Input: directed graph $G = (V, E)$ with edge costs c_e for each $e \in E$.

Goal: Compute the length of the shortest (v, v) path for all pairs of vertices $(u, v) \in V$

You need n invocations of a single vertex shortest path subroutine to do this.

Running Time (with nonnegative c_es)

$n \cdot$ Dijkstra $= O(nm \log n) = \begin{cases} O(n^2 \log n) & \text{if } m = \Theta(n) \\ O(n^3 \log n) & \text{if } m = \Theta(n^2) \end{cases}$

Run with general c_es

$n \cdot$ B-Ford $= O(n^2 m) = \begin{cases} O(n^3) & \text{if } m = \Theta(n) \\ O(n^4) & \text{if } m = \Theta(n^2) \end{cases}$

Are there any algos faster than $O(n^3)$ for the general case?

A quadratic number of distances have to be computed, for a given pair (u, v) the shortest path might have $O(n)$ edges, so at least n time

WEEK 1 L3

OPTIMAL SUBSTRUCTURE

We'll be attempting to find an algorithm without easy enumeration

Floyd–Warshall algorithm: $O(n^3)$ algorithm for APSP.
- It works even with negative edge lengths

Thus: ① At least as good as B-Ford, better in dense graphs

② With nonnegative edge costs, competitive with n Dijkstra's for dense graphs.

Important special case: transitive closure of binary operation (all ~~edges~~ pairs must have finite ~~paths~~ edges with finite weight)

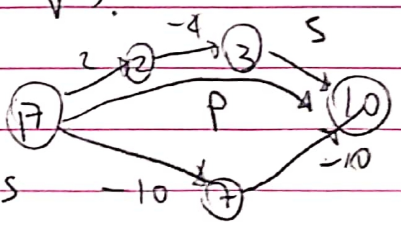
Open question: Can we solve APSP significantly faster than $O(n^3)$ in dense graphs?

It can be tricky to define ordering on subproblems with graphs

Let $V = \{1, 2, \dots, n\}$, $V^{(k)} = \{1, 2, \dots, k\}$ ~~the prefix~~ up until k, arbitrary number

Lemma: if G has no negative cycle, we can fix source $i \in V$, dest j, $k \in \{1, 2, \dots, n\}$. Let P = shortest cycle free i-j path with all internal nodes in $V^{(k)}$.

Example:



$i = 17$, $j = 10$, $k = 5$

because $k = 5$, we can't take the bottom (shorter) path.

OPTIMAL SUBSTRUCTURE LEMMA

Suppose G has no neg cost cycles. Let P be shortest cycle free i-j path with all internal nodes in $V^{(k)}$.

Case ①: if k is not internal to P, P is shortest cycle free i-j path $\subseteq V^{(k-1)}$

Case ②: if k is internal to P, split it:
- P_1 shortest cycle free i-k path in $V^{(k-1)}$
- P_2 " " " " k-j path in $V^{(k-1)}$

WEEK 1 L10

THE FLOYD-WARSHALL ALGORITHM

Setup: let $A[i,j,k]$ be 3D array — by k in $V^{(k)}$, all internal/middle vertices in $V^{(k)}$

Intent: $A[i,j,k]$ is the length of a shortest i-j path in $V^{(k)}$ (or $+\infty$ if not possible)

Question: What is $A[i,j,0]$?

if $i=j$	0
if $(i,j) \in E$	c_{ij}
if $i \neq j$ and $(i,j) \notin E$	$+\infty$

PSEUDOCODE.

- Initialize $A[i,j,0]$, fill with these
- For $k=1$ to n
  - For $i=1$ to n
    - For $j=1$ to n

$$A[i,j,k] = \min \begin{cases} A[i,j,k-1] \\ A[i,k,k-1] + A[k,j,k-1] \end{cases}$$

To recompute shortest i,j path, $B[i,j]$ = max label of internal node on shortest i,j path for all $i,j \in V$

WEEK 2 L11

A REWEIGHTING TECHNIQUE

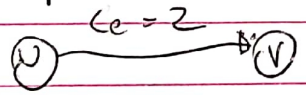
APSP reduces to n invocations of SSSP.

For non neg edge lengths, Dijkstra ⇒ SSSP
- $(n \cdot m \log(n)) = O(mn \log(n))$

if not B-Ford
- $(mn \cdot n) = O(mn^2)$

Johnson's Algorithm reduces APSP to:
- Invocation of B-Ford $(O(mn))$
- n invocations of Dijkstras $(O(mn \log n))$

Reweighting by adding the same to all edges to obtain G' only preserves the SP if all paths in G have the same number of edges



$p_u = -4$ $\quad$ $p_v = -3$

$c'_e = 2 + (-4) - (-3) = 1$

If we use c'_e instead of c_e, a path's P new length in G' is:

$$L(P') = \left(\sum_{e \in P} c_e\right) + p_s - p_t$$

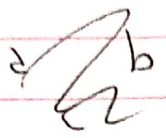
adds this to every s-t path.

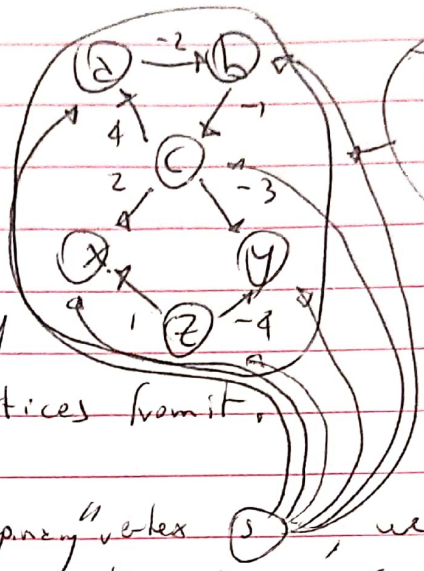
Vertex weights can be computed using B-Ford, and they always exist!

WEEK 1 LH & LI3

HOW TO FIND MAGICAL VERTEX WEIGHTS

Example

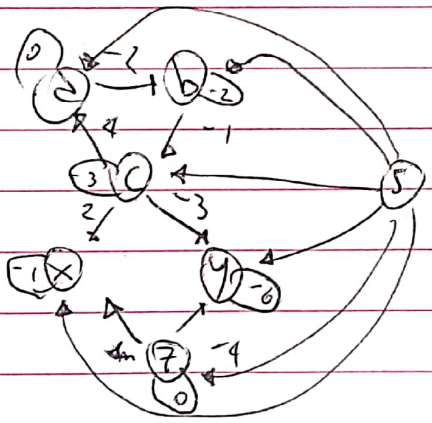




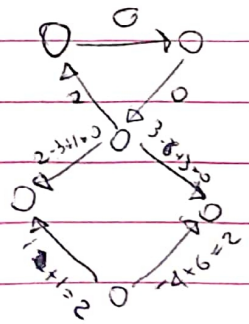
Source vertex is tricky: you need to be able to reach all other vertices from it.

① So we just add a new "imaginary" vertex s, we can do this because it doesn't add any u,v paths for u,v ∈ G

After ② computing shortest distances:



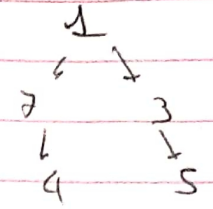
They are exactly the magical weights we want! ③ Add them:



Runtime = $O(n) + O(mn) + O(m) +$
① ② ③
$+ O(n \log n) + O(n^2) = O(n \log n)$
④ ⑤

We can now ④ run Dijkstra (#4) and ⑤ ~~iterate~~ compute for every u,v pair!

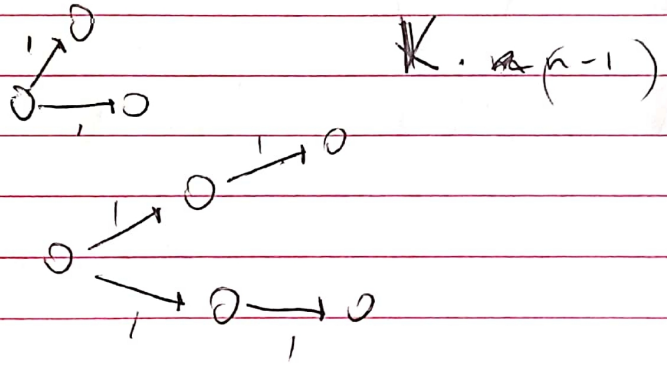
Problem Swibbling



S

B Food

$$A[i,v] = \min \begin{cases} A[i-1,v] \\ \min_{(u,v)\in E,\ u<v} A[i,u] + c_{uv} \end{cases}$$



WEEK 2 L1

NOT ALL PROBLEMS ARE SOLVABLE IN P TIME

We have focused on practical algorithms for important computational problems

Many of these problems, however, seem impossible to solve efficiently

How do we formalize intractability? How do we know when something is intractable? What are some approaches to NP-complete problems?

Formal definition: A problem is polynomial-time solvable, if there is an algorithm that correctly solves it in $O(n^k)$ time, for some constant k

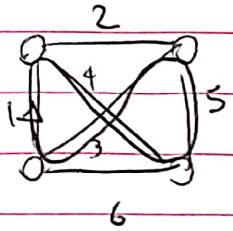
THE CLASS P IS DEFINED AS the set of poly time solvable problems

TRAVELING SALESMAN PROBLEM

Input: Complete undirected graph with non negative edges

Output: Min-cost tour, a cycle visiting each edge exactly once

THERE IS NO PTIME ALGO FOR TSP



WEEK 2 L2.

REDUCTION AND COMPLETENESS

Definition: we say problem Π_1 reduces to Π_2 if: given a ~~solution~~ subroutine for Π_2, we can use it to solve Π_1 in poly time

COMPLETENESS

Suppose: Π_1 reduces to Π_2

Contrapositive: if Π_1 not in P, neither is Π_2

Π_2 is at least as hard as Π_1

Definition: Let C = a set of problems

The problem Π is C-complete if:

1. $\Pi \in C$
2. Everything in C reduces to Π

That is: Π is the hardest problem in C

CHOICE OF THE CLASS C

Idea: Show TSP is C-complete for a really big set C

Why can't we just say C is all problems?

HALTING PROBLEM: Given a program and an input for it, will it eventually halt?

Can't be solved (Anyone who's read The Emperor's New Mind learned that)

TSP is solvable in finite time though Brute force though.

TSP is as hard as all brute-force-solvable problems

WEEK 2 L3 & L4

DEFINITION AND INTERPRETATION OF NP COMPLETENESS

Refined title: prove that TSP is as hard as all brute force solvable problems.

Definition: A problem is in NP if:
1. Solutions have length polynomial to the input size
2. Purported solutions can be verified in polynomial time.

Examples:

- Is there a TSP tour with length ≤ 1000?
- ~~POSSIBLE ANS~~ Knapsack

Note: every problem in NP can be solved by brute-force search in exponential time

By definition of completeness: A polynomial time algo for one NP complete problem solves every problem in NP.

Example:
Brute force try all solutions and compare at the end

HOW TO PROVE A PROBLEM IS NP COMPLETE

2) Find a known NP complete problem Π'
3) Prove Π' reduces to Π → implies Π at least as hard as Π'.

WEEK 2 L5

THE P VS NP QUESTION

Two classes of problem: P and NP

P: Polynomial Solvable

NP: Can verify solution (e.g. given) in polynomial time

P = NP ?

It is widely conjectured $P \neq NP$

Reasons to believe in $P \neq NP$:

- (Psychological) if $P = NP$, we'd have proved it by now
- (Philosophical) if $P = NP$, finding a proof always as easy as finding one

ULTIMATELY, NO PROOF!

NP STANDS FOR NONDETERMINISTIC POLYNOMIAL

WEEK 2 L6

NP COMPLETENESS IS A BEGINNING, NOT AN END

Three useful strategies to deal with NP completeness.

(1) Focus on computationally tractable special cases.

Examples: WIS in path graphs ~~A~~ (see? some NP problems are computationally tractable!)

(2) Heuristics: fast algorithms that are not always correct

(Greedy criterions for Knapsack)

(3) Solve in exponential time but faster than brute force search.

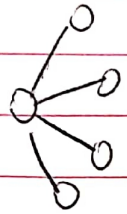
- Knapsack ($O(nW)$ instead of 2^n)
- TSP ($\neq 2^n$, $\neq n!$)

WEEK 2 L7

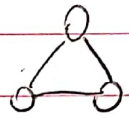
~~STARTER SEARCH FOR~~ VERTEX COVER PROBLEM

Input: undirected graph $G = (V, E)$

Goal: Compute min cardinality vertex cover, subset $\subseteq V$ that contains at least one endpoint of each edge of G.



size = 1



size = 2

Fact: In general, Vertex cover is NP Complete

STRATEGIES FOR NP-COMPLETE PROBLEMS

1. Identify comp. tractable special cases. For example, for VC, trees, and bipartite graphs can be solved. When solution is small (logn or less)
2. Heuristics
3. Exponential time, but smarter than brute force

WEEK 12 L8 & L9

SMARTER SEARCH FOR VERTEX COVER PROBLEM

Substructure Lemma: Consider graph G, edge $(u,v) \in G$, integer $k \geq 1$.

Let $G_v = G$ with v and its incident edges deleted.

Then G has a vertex cover of size k $\Leftrightarrow$ G_u or G_v has a vertex cover of size $(k-1)$.

This neat little trick allows us to reduce running time from considering all solutions ($O(\binom{n}{k}) \geq O(n^k)$ time).

IMPROVED ALGORITHM (G, k):

1. Random edge $(u,v) \in E$
2. Recursively search for vertex cover of size $k-1$ in G_u if found, return $S \cup \{u\}$
3. " " in G_v, " , return $S \cup \{v\}$
4. Fail (G has no vertex cover with size k)

NEW RUNNING TIME: $O(2^k)$ calls each considering $O(m)$ edges.

$O(2^k m)$ → if $k = O(\log n)$, it's polynomial time

→ remains feasible at $k = 20$, unlike $O(n^k)$

WEEK 2 L10 & L11

TSP OPTIMAL SUBSTRUCTURE?

Brute force $= \Omega(n!)$ → tractable to 12, 13

DP $= O(n^2 2^n)$ → tractable to ~30

~~Idea: Copy the format of B-Ford~~ Can't solve original problem from subproblems.

Some ~~guy~~ goes for next example.

The next one works!

$$L_{ij} = \min_{k \neq i,j} \{ L_{i-1,k} + C_{kj} \}$$

A DP ALGO FOR TSP

- Let A = 2D Array indexed by subsets $S \subseteq \{1, 2, \ldots, n\}$ containing 1 and destinations $j \in \{1, 2, \ldots, n\}$.

Base case: Fill it with $A[S, 1] = \begin{cases} 0 \text{ if } S = \{1\} \\ +\infty \text{ otherwise} \end{cases}$

- For $m = 2, 3, \ldots, n$ (m is subproblem size)
  - For each set S of size m containing 1
    - For each $j \in S$, $j \neq 1$
      - $A[S, j] = \min_{(k \in S, k \neq j)} \{ A[S - \{j\}, k] + C_{kj} \}$

Return $\min_{j=2,\ldots,n} \{ A[\{1, \ldots, n\}, j] + C_{j1} \}$

The running time for this beauty is:

$O(n 2^n) \; O(n) =$

$= O(n^2 2^n)$

choices of j | choices of S | work per subproblem

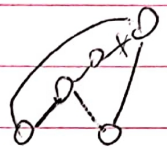
Problem Scribbling.

① $n^{\log n} = 2^{\log n \log n} = 2^{\log n^2}$?

$\Pi_1 \rightarrow \Pi_2$

TSP2 → TSP1

$\geq 1 \leq \log(n)$



✓

✓

×

WEEK 3 L1

APPROXIMATION ALGORITHMS

Heuristics are guaranteed to be fast, but not necessarily correct.

Strategies:

① ... if not special case,

② ← this

We can try several different greedy heuristics.

We could take the lightest or most valuable object first, or have a $\frac{\text{value}}{\text{weight}}$ (or $\frac{\text{weight}}{\text{size}}$) ratio.

Any greedy solution can be arbitrarily bad with respect to an optimal solution.

WEEK 3 L2 & L3

PERFORMANCE GUARANTEE ON OUR ALGO

Theorem: 3 step's greedy algo solution is always ≥ 50 % of an optimal solution.

If: What if we were allowed to fill the knapsack using a suitable fraction, say 78%, of the total number of items ($k+1$)

"We will call this "the greedy fractional solution"

Example: $W=3$, $v_1=3$, $v_2=2$, $w_1=2$, $w_2=2$

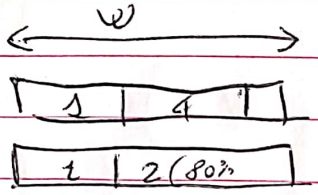
Get 100% — Get 50%.

Value $= 2, 2/50\% = 4$

So GFS $= 4$

GFS ≥ optimal solution

PROOF IT'S AS GOOD AS ANY NON-FRAC SOL



① Let S be an arbitrary feasible solution

② Suppose the I units of the knapsack are filled by S with items not packed by the greedy fractional solution.

③ There must be at least I units of knapsack filled with GFS not packed by S.

④ items in ③ have larger bang per buck (v/w) than those in 2.

⑤ Total value of GFS is at least that of S

ANALYSIS

Value of 3step $\geq$ total value of the first k items

Value of 3step $\geq$ Value of $(k+1)$ th item

2. " " $\geq$ sum (# first $(k+1)$ items) $\geq$ GFS $\geq$ OPTIMAL SOL

WEEK 3 L4

A DYNAMIC PROGRAMMING HEURISTIC

Goal: for arbitrarily close approximation. Let the user specify ε, fraction it can deviate from optimal.

Running time of will decrease if ε increases.

APPROACH: ROUNDING ITEM VALUES ~~SIMILAR~~

Recall we can solve knapsack in $O(nW)$ via dynamic programming for W and w_is integers.

If v_is are integers, we can solve knapsack in $O(n^2 v_{max})$, where $v_{max} = \max\{v_i\}$.

There we go, we have our poly-time algorithm

TO RECAP OUR HEURISTIC | Runtime: $O(n^2 v_{max})$

1. Round v_i to nearest multiple of m
   Divide by m to get smaller integers $\hat{v}_i$s
2. Use dynamic programming to solve with $\hat{v}_i$s and w_is

WEEK 3 LS

ANOTHER DP ALGO FOR KNAPSACK

DP#1 ~~(see L4)~~ $O(nW)$
w_i and W are integers

DP#2: $O(n^2 v_{max})$

Assume v_i's are integers

SUBPROBLEMS AND RECURRENCE

Subproblems: for $i = 0, 1, \ldots, n$ and $x = 0, 1, \ldots, nv_{max}$
define S_{ix} = minimum total size to achieve $\geq x$ using only the first i items.

Recurrence: $(i \geq 1)$

$$S_{ix} = \min \begin{cases} S(i-1, x) & \text{Case 1, } i \text{ not used in optimal solution} \\ w_i + S(i-1), (x - v_i) & \text{Case 2, } i \text{ used} \end{cases}$$

THE ALGORITHM

A = 2D Array, $(i = 0, \ldots, n \; ; \; x = 0, 1, \ldots, nv_{max})$

Fill it with $A[0, x] = \begin{cases} 0 & \text{if } x = 0 \\ +\infty & \text{otherwise} \end{cases}$

For i in range $(1, n+1)$: — $O(n) \cdot O(n) = O(n^2)$
  For x in range $(0, nv_{max} + 1)$: — $O(1)$
    $A[i,x] = \min \{ A[i-1, x], w_i + A[i-1, x - v_i] \}$

Return the largest x such that $A[n, x] \leq W$ — $O(nv_{max})$

Running time is $O(n^2 v_{max})$

WEEK 3 L6

DP HEURISTIC ANALYSIS

- Plan for analysis:

① Figure out how big we can take m, subject to achieving a $(1-\epsilon)$ approximation (or better)

② Use this value of m to determine running time.

If we round v_i to $\hat{v}_i$, $m\hat{v}_i < v_i$, $mv_i > v_i - m$.

So:

ACCURACY ANALYSIS

S = our solution.

S^* = optimal solution.

$$\sum_{i \in S} v_i \geq m \sum_{i \in S} \hat{v}_i \geq m \sum_{i \in S^*} \hat{v}_i \geq \sum_{i \in S^*} (v_i - m)$$

$v_i > m\hat{v}_i$

If S was the original solution, bigger than ours.

$m\hat{v}_i \geq v_i - m$

Normally $O(n^2 \hat{v}_{max})$

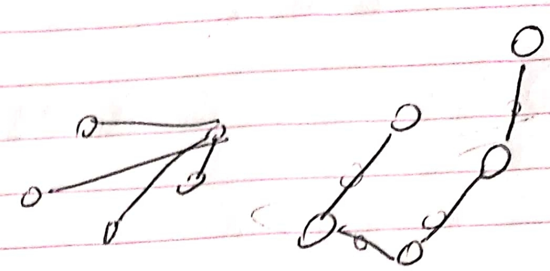
$mn \leq \epsilon \sum_{i \in S^*} v_i \rightarrow mn = \epsilon v_{max} \rightarrow \frac{v_{max}}{m} = \frac{n}{\epsilon}$

Running time $= O(n^3/\epsilon)$.

30.1

PROBLEM SCRIBBLING

Vertex cover at ...



$\sum_j p_j$ ~~B = max_j p_j~~ $B = \max_j p_j$

~~Min 29~~

26

LOCAL SEARCH

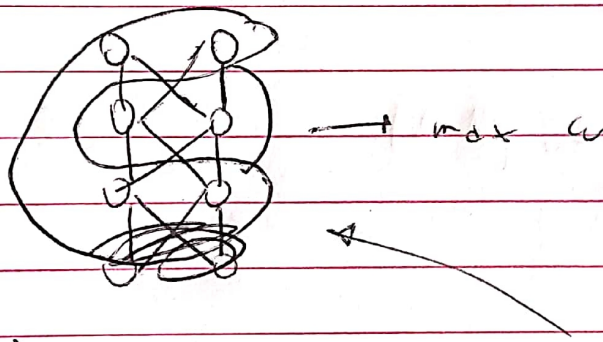
Max cut problem.

Input: undirected graph $G(V, E)$

Output/Goal: a cut (A, B) – a partition of V into two non-empty sets – that maximizes the number of crossing edges.

Sad fact: NP complete.

The case of bipartite graphs is a special case that can be solved by breadth first search.

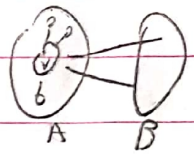


→ max cut is P

No edges internal to A or B, they all cross. So bipartite graphs are easy enough. Otherwise:

A LOCAL SEARCH ALGORITHM

Notation: for a cut (A, B) and vertex v, define:



$C_v(A,B)$ = # of edges incident on v that cross (A,B)
$D_v(A,B)$ = # of edges incident on v that cross (A,B)

Local Search Alg:

① Let (A,B) be arbitrary cut of G
② While there is a vertex v with $d_v(A,B) > C_v(A,B)$
– moving v to the other side makes the cut better. In the example, you gain 3 CEs and lose 2.
③ Return final cut (A,B)

Terminates in $\binom{n}{2}$ time ($O(n^2)$)

PERFORMANCE isn't guaranteed to be correct.

but: it's at least 50% of the true maximum, and also >50% of all the edges in the graph (which in turn upper bound the true maximum)

Expected number of CEs always $\geq 50\%$.

Proof: consider a random cut (A,B). For edge $e \in E$,

define $X_e = \begin{cases} 1 & \text{if } e \text{ crosses } (A,B) \\ 0 & \text{otherwise} \end{cases}$

$$E[X_e] = P_r[X_e = 1] = 1/2$$

PROOF OF PERFORMANCE GUARANTEE

Let A, B be a locally optimal cut.

For every vertex v, $d_v(A,B) \leq c_v(A,B)$.

Summing over all $v \in V$:

$$\sum_{v \in V} d_v(A,B) \leq \sum_{v \in V} c_v(A,B)$$

~~Crossing edges~~ Each non-CE counted twice

Each CE counted twice

So:

Add rhs to both sides

$$2 \cdot [\#\text{ non crossing edges}] \leq 2\,(\#\text{ of crossing edges}) \implies$$

$$2 \cdot (|E| - [\#CEs]) = 2|E| \leq 4\,(\#CEs] \implies [\#CEs] \geq \frac{1}{2}|E|.$$

Also holds with weighted graphs, though it's not guaranteed to converge in polytime anymore.

WEEK 4 L3 & L4

LET'S THINK ABOUT LSEARCH MORE GENERALLY

NEIGHBOURHOODS

Examples: cuts of a graph, TSP tours, CSP variable assignments.

Key ingredient: Neighbourhoods

For each $x \in X$, specify which $y \in X$ are its "neighbours"

Example: x, y are neighbouring cuts $\iff$ they differ by one vertex

A GENERIC LSEARCH ALGO

1. Let x = some initial solution
2. While the current solution x has a better neighbour y:
   $x = y$
3. Return the final (locally optimal) solution x

FAQ

How to pick first x?: either at random or, in cases where it would matter (Example: Two peaks) use the best heuristic you can think of

If two neighbours are superior and equal, how to choose?

1. At random?
2. more heuristics

How to define neighbourhoods?:

Make a decision between big, slow to verify neighbourhoods or small more approximate ones.

Is LSEARCH Guaranteed to terminate?

If X is finite, yes.

Is it ~~guaranteed to converge quickly~~?

Usually not (see smoothed analysis for examples of when it does)

Are locally optimal solutions generally good approximations to global?

Again, usually not. To mitigate, may run random search many times, remember best solution found.

WEEK 4 CO5

THE 2-SAT PROBLEM

Input: ① n Boolean variables $x_1, x_2, \ldots, x_n$
② m clauses of 2 literals each

Example: $(x_1 \vee x_2), (\bar{x}_1 \vee x_3), (x_3 \vee x_4), (\bar{x}_2 \vee \bar{x}_4)$

We are interested in knowing ~~from~~ if it is possible to satisfy all clauses.

Output: yes or no to this

2-SAT can be solved in polynomial time
Backtracking works in polynomial time
~~3-SAT~~ 3-SAT CANNOT, NP-COMPLETE
Brute-force search $\approx 2^n$ time
Can get time $\approx \left(\frac{4}{3}\right)^n$ via randomised local search.

PAPADIMITRIOU's 2-SAT ALGO:

Repeat $\log_2 n$ times
- Random initial assignment
- Repeat $2n^2$ times
  - if current assignment satisfies all clauses, halt, return solution
  - pick arbitrary unsatisfied clause and flip one of its variables. (randomly chosen)

report "unsatisfiable"

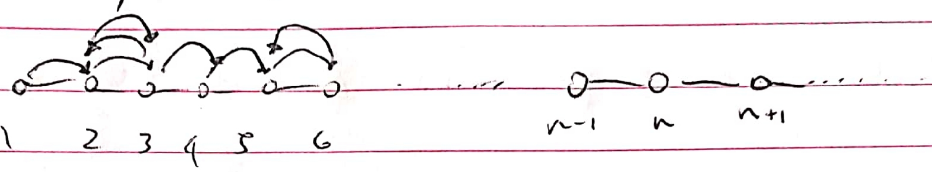
① Runs in poly time
② Always correct on unsatisfiable instances

WEEK 4 L6

RANDOM WALKS ON A LINE

To understand why Papadimitriou's algo works, we need to understand RWOAL

Setup: initially $t = 0$, position 0



At each step, up or down with 50% probability (except at 0)

Notation: for $n \geq 0$, T_n = number of steps before random walk gets there.

$E[T_n] = n^2$

PROOF (Think about it as a tree? ... MAYBE NOT)

Let Z_i = number of random walk steps to get from i to n.

Edge cases $E[Z_n] = 0$, $E[Z_0] = 1 + E[Z_1] = 1 + \ldots$

For $i \in \{1, 2, \ldots, n-1\}$ 1/2 $1 + E[Z_{i-1}]$ 1/2 $(1 + E[Z_{i+1}])$

$E[Z_i] = Pr[\text{go left}]\, E[Z_i \mid \text{go left}] + Pr[\text{go right}]\, E[Z_i \mid \text{go right}]$

$= 1 + \frac{1}{2}\left(E[Z_{i+1}] + E[Z_{i-1}]\right) \rightarrow E[Z_{i-1}] - E[Z_i] + 2$

$E[Z_0] - E[Z_1] = 1$

$E[Z_1] - E[Z_2] = 3$

$\vdots$ $\quad E[Z_n] = 0$ $\quad = 2n - 1$ $\rightarrow$

$E[T_n] = E[Z_0] = n^2$

A Corollary :

$$\Pr\left[T_n > 2n^2\right] \leq \frac{1}{2}.$$

WEEK 4 L7

LOCAL SEARCH: PAPADIMITRIOU'S ALGO.

*LS

Satisfiable Instances.

Theorem: For a satisfiable 2-SAT instance with n variables, Uddy D's Algo produces a satisfying assignment with probability $\geq \left(1 - \frac{1}{n}\right)$

What's baffling about the algorithm is that the measure of its success (how many statements are satisfied) can actually go down!

PROOF: first focus on a single iteration of the outer for loop.

Fix an arbitrary satisfying assignment a^*

Let a_t = assignment after t^{th} iteration t $(t = 0, 1, \ldots, 2n^2)$
Let X_t = number of variables on which a_t and a^* agree:

If $X_t = n$, algorithm ends with a^*

When a_t is not a satisfying assignment and the algo flips are random,

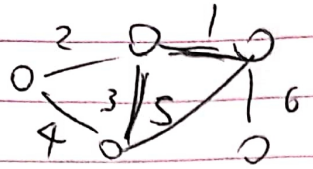
(1) If a^* and a_t differ on both x_i and x_j, $X_{t+1} = X_t + 1$
(2) If a^* and a_t differ on exactly one bool:

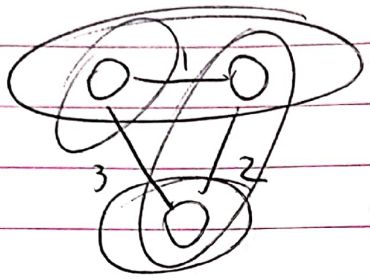
$$X_{t+1} = \begin{cases} X_{t+1} & (50\% \text{ prob}) \\ X_{t-1} & (50\% \text{ prob}) \end{cases}$$

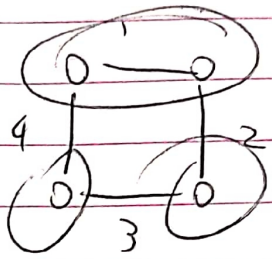
We get a random walk BUT
- it might stop early
- might have $X_0 > 0$ instead of $X_0 = 0$
- sometimes moves right with 100% probability

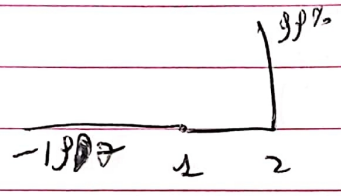
Which is why $\geq Pr\left[T_n \leq 2n^2\right] \geq \frac{1}{2}$ for each loop. $P_{Algo\ fails} = Pr\left[\text{[illegible]}\right] = \frac{1}{n}$

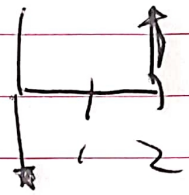
Scribbles











198 - 197 = 1