- **No lecture** on Monday 5/26 (Memorial day),
- **Additional lecture:** probably Thursday 5/22 (let me know now if you can't)
- **Last Homework:** Wednesday 5/28.
- **Last Lecture:** Monday 6/2.
- **Final Exam:** Monday 6/9 (2 hours in the evening). You can bring the books, your notes, and your homework to the exam.

CSEP 521- Applied Algorithms

Coping with NP-hardness – Approximation Algorithms

Reading:

- Skiena, chapter 6.8
- CLRS, chapter 37 (1st Ed.)
  chapter 35 (2nd Ed.)

Review: NP-Completeness

- Some problems are intractable: as they grow large, we are unable to solve them in reasonable time.
- What constitutes reasonable time? Standard working definition: polynomial time
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

P and NP

P = problems that can be solved in polynomial time

NP = problems for which a solution can be verified in polynomial time = problems for which a solution can be find in polynomial time by a non-deterministic machine.

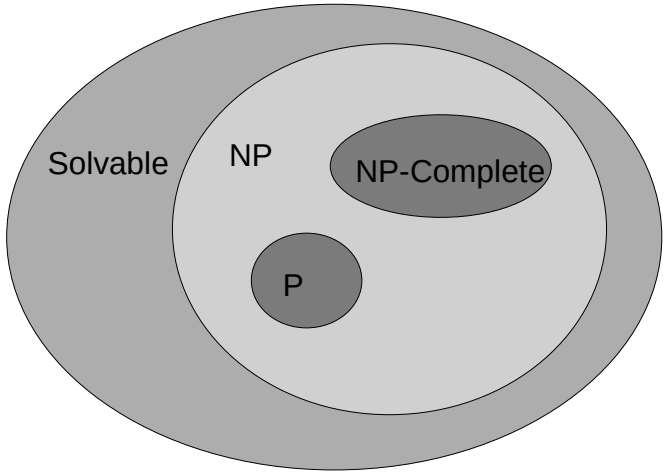
Unknown whether **P** = **NP** (most suspect not)

- Example: Hamiltonian-cycle problem is in **NP**:
  - Cannot be solved in polynomial time.
  - Easy to verify solution in polynomial time.

NP-Complete Problems

- We will see that NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P** = **NP**)
  - Thus: solve hamiltonian-cycle in $O(n^{100})$ time, you've proved that **P** = **NP**. Retire rich & famous.

NP Problems



Optimization v.s. Decision

To simplify things, we will worry only about *decision problems* with a yes/no answer

- Many problems are *optimization problems*, but we can re-cast them as decision problems

Example: Graph coloring.

- Optimization problem: what is the minimal number of colors needed to color *G*?
- Reporting problem: Can *G* be colored using k colors? If so, report a legal k-coloring.
- Decision problem: Can *G* be colored using k colors?

Why Prove NP-completeness?

- Though nobody has proven that **P** != **NP**, if you prove a problem NP-Complete, most people accept that it is probably intractable.
- Therefore it can be important to prove that a problem is NP-Complete
  - Don't need to come up with an efficient algorithm.
  - Can instead work on *approximation algorithms.*

Reduction

- A problem P can be reduced to another problem Q if *any* instance of P can be "easily rephrased" as an instance of Q, the solution to which provides a solution to the instance of P
  - Intuitively: If P reduces to Q, P is "no harder to solve" than Q.

Reducibility - An example

- P: Given a set of Booleans $\{x_i \in$ TRUE, FALSE$\}$, is at least one TRUE?
- Q: Given a set of integers, is their sum positive?
- Transformation: given $(x_1, x_2, \ldots, x_n)$ booleans, let $(y_1, y_2, \ldots, y_n)$ be a set of integers where $y_i = 1$ if x_i = TRUE, and $y_i = 0$ if x_i = FALSE.
- P is no harder than Q: if we can solve Q we can run the transformation to get a solution to P.

Using Reductions

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-complete:
  - P is NP-complete if $P \in$ NP and P is NP-hard.
- Definition of NP-Hard:
  - P is NP-hard if all problems R of NP are reducible to P. Formally: $R \leq_p P, \forall R \in$ NP
- If $P \leq_p Q$ and P is NP-hard, Q is also NP-hard.

Proving NP-Completeness

- How do we prove a problem P is NP-Complete?
  - Pick a known NP-Complete problem Q
  - Reduce Q to P:
    - Describe a transformation that maps instances of Q to instances of P, s.t. "True" for P = "True" for Q
    - Prove the transformation works
    - Prove it runs in polynomial time
  - and yeah, prove $P \in$ **NP**
- We need at least one problem for which NP-hardness is known. Once we have one, we can start reducing it to many problem.

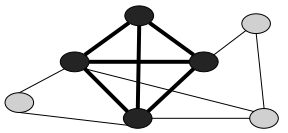
The SAT Problem

- The first problems to be proved NP-Complete was *satisfiability* (SAT):
  - Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?
  - Ex: $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
  - **Cook's Theorem:** The satisfiability problem is NP-Complete
    - Note: Argue from first principles, not reduction (any computation can be described using SAT expressions)
    - Proof: not here

The k-clique Problem

- *A clique in a graph G is* a subset of vertices fully connected to each other, i.e. a complete subgraph of G.
- The *clique problem*: how large is the maximum-size clique in a graph?
- *Can we turn this into a decision problem?*
- A: Yes, we call this the *k-clique problem*
- *Is the k-clique problem within* ***NP****?*

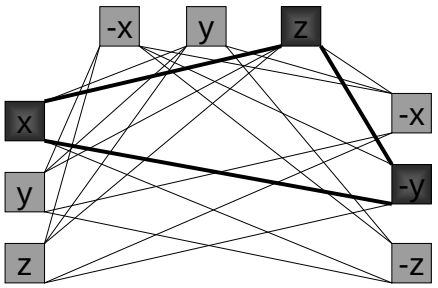
  Yes: Nondeterministic algorithm: guess k vertices then check that there is an edge between each pair of them.

4-clique: 

3-CNF → Clique

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

$$x = 1, y = 0, z = 1$$

G 

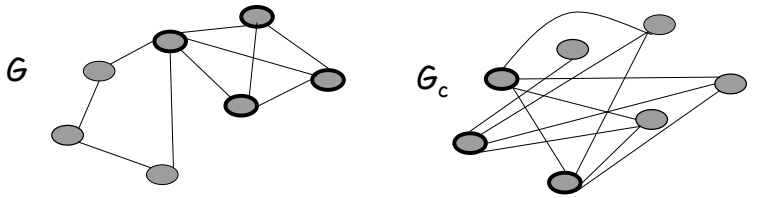
Any clique of size k must include exactly one literal from each clause.

The Vertex Cover Problem

- A vertex cover for a graph G is a set of vertices incident to every edge in G
- The vertex cover problem: what is the minimum size vertex cover in G?
- Restated as a decision problem: does a vertex cover of size k exist in G?
- Theorem: vertex cover is NP-Complete

Clique → Vertex Cover

- First, show vertex cover in NP (How?)
- Next, reduce k-clique to vertex cover:
  - The complement G_C of a graph G contains exactly those edges not in G
  - Compute G_C in polynomial time
  - Claim: G has a clique of size k iff G_C has a vertex cover of size |V| - k



The Traveling Salesman Problem:

- Optimization variant: a salesman must travel to *n* cities, visiting each city exactly once and finishing where he begins. How to minimize travel time?
- Model as complete graph with cost c(i,j) to go from city i to city j

- How would we turn this into a decision problem?
  - Answer: ask if there exists a path with cost at most k

Hamiltonian Cycle → TSP

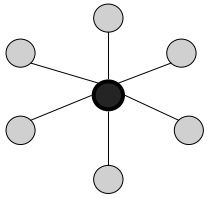
- The hamiltonian-cycle problem: given a graph G, is there a simple cycle that contains every vertex?
- To transform ham. cycle problem on graph G = (V,E) to TSP, create graph G' = (V,E'):
- G' is a complete graph
- Edges in E' also in E have cost 0
- All other edges in E' have cost 1
- TSP: is there a TS cycle on G' with cost 0?
  - If G has a ham. cycle, G' has a TS cycle with cost 0
  - If G' has TS cycle with cost 0, every edge of that cycle has cost 0 and is thus in G. Thus, G has a ham. cycle.

Other NP-Complete Problems

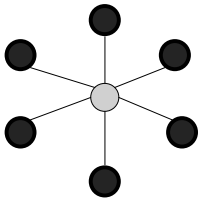
- *Partition:* Given a set of integers, whose total sum is 2S, can we partition them into two sets, each adds up to S?
- *Subset-sum*: Given a set of integers, does there exist a subset that adds up to some target *T*?
- *Graph coloring*: can a given graph be colored with *k* colors such that no adjacent vertices are the same color?
- *Steiner Tree:* Input: given a graph G=(V,E), a subset T of the vertices V, and a bound B, is there a tree connecting all the vertices of T of total weight at most B?

Independent Set

- Input: A graph G=(V,E), k
- Problem: Is there a subset S of V of size at least k such that no pair of vertices in S is connected by an edge.
- Maximum independent set problem: find a maximum size independent set of vertices.



Maxim**al** independent set



Maxim**um** independent set

Coping with NP-hardness

- O.K, I know that a problem is NP-hard. What should I do next?
- First, stop looking for an efficient algorithm.
- Next, you might insist on finding an optimal solution (knowing that this might take a lot of time), or you can look for approximate solutions with guaranteed performance.

Techniques for Dealing with NP-complete Problems

- Exactly
  - backtracking, branch and bound, dynamic programming.
- Approximately
  - approximation algorithms with performance guarantees.
  - heuristics with good average results.
- Change the problem (if possible…)

Approximation Algorithms

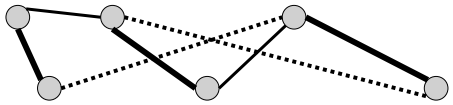
- The fact that a problem is NP-complete doesn't mean that we cannot find an approximate solution efficiently.
- We would like to have some guarantee on the performance – how far are we from the optimal?
- What is the best we can hope for (assuming $P \neq NP$)?

Approximation Algorithms with Additive Error.

- For few NP-hard problems, there are approximation algorithms that produce an **almost optimal** solution – one that is far only by an additive constant from the optimal.
- Minimization problems: $Alg(I) \leq opt(I) + c$
- Maximization problems: $Alg(I) \geq opt(I) - c$
- Example: Edge coloring.

Edge Coloring

- An Edge-coloring of a graph $G=(V,E)$ is an assignment, c, of integers to the edges such that if e_1 and e_2 share an endpoint then $c(e_1) \neq c(e_2)$.



- Let Δ be the maximal degree of some vertex in G.
- It is known that for any graph the minimal number of colors required to edge-color G is Δ or $\Delta+1$.
- It is NP-hard to distinguish between these two cases.
- There exists a polynomial time algorithm that colors any graph G with at most $\Delta+1$ colors.
- For this algorithm $Alg(I) \leq opt(I) + 1$.

r-approximation Algorithms

- Approximations with guaranteed additive error are rear.
- All other approximation algs we are going to see today are factor-r approximations:
  - Vertex cover
  - Traveling salesman
  - Bin packing
  - Knapsack
- An algorithm *Alg* is an r-approximation if, for any input, the solution that *Alg* outputs is within factor r from the optimal. ($r \geq 1$)

Approximation Algorithms (minimization)

- In minimization problems: Alg is r-approximation if $Alg(I) \leq r \cdot opt(I)$ for any instance I.

Example 1: Traveling Salesman is a minimization problem (the goal is to find a tour with minimal cost). If we have an algorithm, A, that finds, *for any graph*, a tour whose cost is at most 5 times the optimal, then A is 5-approximation to TSP.

Example 2: Minimum Spanning Tree is a minimization problem (the goal is to find an ST with minimal cost). Each of the optimal algorithms we've learnt is 1-approximation.

Approximation Algorithms (maximization)

- In maximization problems: Alg is r-approximation if $Alg(I) \geq (1/r) \cdot opt(I)$ for any instance I.

Example: Maximal clique is a maximization problem (the goal is to find a clique with maximal size). If we have an algorithm, A, that finds, for any graph, a clique whose size is at least $(\log n)^2/n$ times the optimal, then A is $n/(\log n)^2$-approximation to clique.

(remark: currently, this is the the best known approx ratio for clique.)

Example 1: Vertex Cover

- Given G=(V,E), find a minimum sized subset W of V such that for every (v,u) in E, at least one of v or u is in W.
- Last week we proved that this problem is NP-Hard.
- We are therefore ready to end up with a vertex cover W which is not of minimum size. But- we don't want it to be too large and we want to be able to find it in polynomial time.

Approximating Vertex Cover

VertexCover(G=(V,E)):

while (E≠∅)

1. select an arbitrary edge (u,v)
2. add both u and v to the cover
3. delete all edges incident to either u or v

1. This is a legal cover (why?)
2. This is a 2-approximation (its size is at most 2 times the minimal size vertex cover).

Proof: Let c be the number of iterations. The VC has size 2c. The edges selected in step 1 form a matching of size c (why?). Even if we only need to cover these edges we need at least c vertices.

Approximating Vertex Cover

A more natural algorithm: select in each iteration a vertex with maximum degree, add it to the cover and remove all its adjacent edges.

Looks promising!

However, we can end up with a vertex cover which is (ln n)-times the optimal.

A bad instance for Greedy VC:

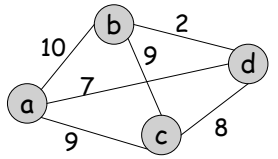
(To be drawn in class)

Example 2: Approximation Algorithm for Euclidean Traveling Salesman Problem

The Problem: Given n points in the plane (corresponding to locations of n cities) find a shortest traveling salesman tour – that passes exactly once in each of the points.

For each pair of cities a,b, we are given the distance dist(a,b) from a to b.

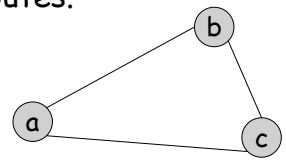
In other words, the input is given as a weighted complete graph.



Euclidean Traveling Salesman Problem

Distances in the plane satisfy the triangle inequality: dist(a,b) ≤ dist(a,c) + dist(c,b)

It means that direct routes are always shorter than indirect routes.



For this version, we will see a simple 2-approximation algorithm: we will find in poly-time a tour whose length is at most twice the optimal.

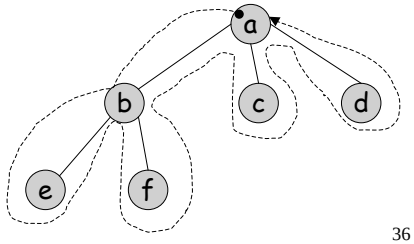
Approximating Euclidean TSP

Note: The weight of a minimum spanning tree is always less than the weight of the optimal tour.

Why? because by removing any edge from the optimal tour we get a spanning tree.

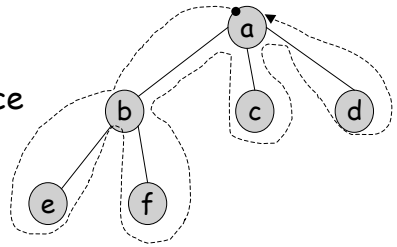
We will use this property to obtain an approximate solution.

Assume that this is our MST. Consider a DFS tour on this tree, let it be (w.l.o.g) a-b-e-b-f-b-a-c-a-d-a

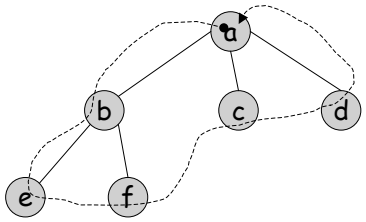


Approximating Euclidean TSP

- The DFS tour defines a TSP path that visits all the cities. The length of this tour is twice the total weight of the MST. However, we might visit some cities more than once.



- To get a legal solution we make shortcuts (move in the next step to the next unvisited vertex). This can only reduce the total length of the path.



Approximating Euclidean TSP

The resulting algorithm:

1. Find a minimum spanning tree of points
2. Convert to tour by following DFS and including edge in opposite direction when DFS backtracks.
3. Construct shortcuts by taking direct routes instead of backtracking.

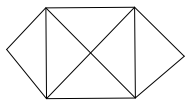
The length of the resulting tour is at most 2 times the optimal → this is a 2-approximation algorithm.

Better version of TSP algorithm

- Uses basic graph algorithms as subroutines: *Matching* and *Euler tour*.
  - a **matching** in a graph G=(V,E): a set of edges S from E such that each vertex in V is incident to at most one edge of S.
  - a **maximum** matching in G: a matching of maximum cardinality
  - a minimum weight matching in a weighted graph: a maximum matching of minimum total weight.
  - Finding a minimum weight matching can be solved in poly-time.

Better version of TSP algorithm

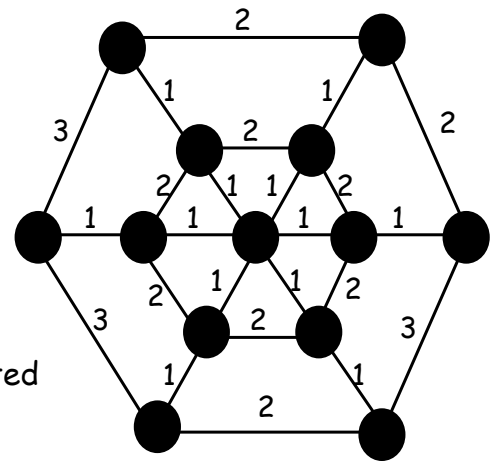
- Euler tours (Skiena, Section 8.4.6)
  - "draw without lifting your pen from the paper"
- An Euler tour in a graph is a tour of the graph that visits each edge exactly once. An Euler cycle begins and ends at the same vertex.
- Well known that an undirected graph contains an Euler cycle iff (1) it is connected and (2) each vertex has even degree.
- Easy to construct Euler tours efficiently.



Better version of TSP algorithm

1. Find an MST.
2. Find minimum weight matching of odd-degree vertices in the tree.
   There's an even number of them (why?).
3. Find Eulerian tour of MST plus edges in matching.
4. Make shortcuts.

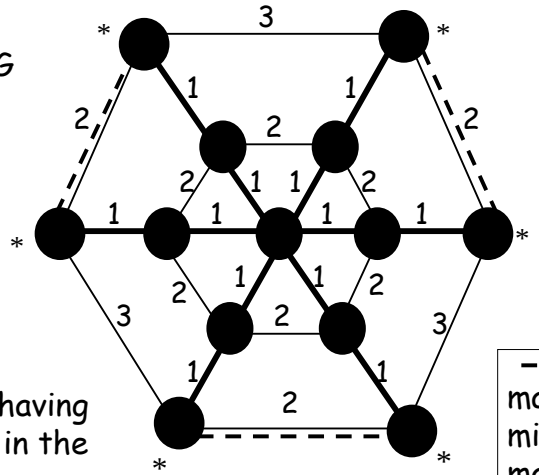
Approximating TSP, Example (1)



The weighted graph G.

Approximating TSP, Example (2)

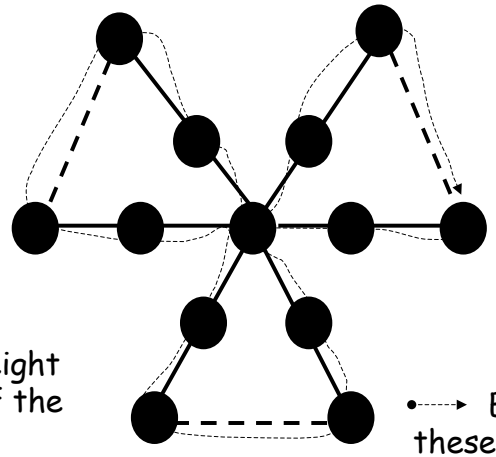
An MST in G



*– vertices having odd degree in the MST

max-size min-weight matching of *

Approximating TSP, Example (3)



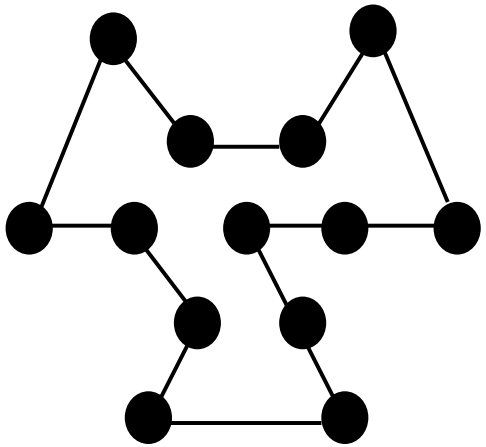
The MST + minimum weight matching of the odd-degree vertices.

Euler tour on these edges

Approximating TSP, Example (4)

Apply shortcuts.

The final tour:



This algorithm has provably performance guarantee

Theorem: The approximation algorithm for Euclidean TSP finds a tour of length at most 3/2 optimal.

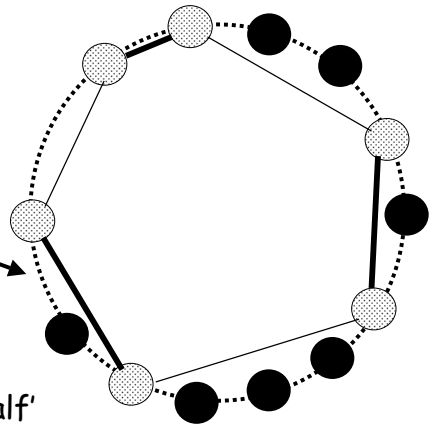
Proof:

- weight of MST ≤ weight of optimal tour
- weight of matching ≤ (weight of optimal tour)/2 (why? see next slide)
- shortcuts don't cost

Dotted vertices: odd-degree vertices in MST. (their # is even)

Any Optimal TSP tour (dotted line) visits these vertices in some order.

In bold: the lighter 'half' of the shortcut-path through these vertices.



A minimum weight matching is lighter than the **bold** shortcuts ≤ optimal TSP.

Non-Euclidean TSP

When did we use the assumption that for all a,b,c dist(a,b) ≤ dist(a,c) + dist(c,b) ?

Is it really needed?

Yes – get ready for our first non-approximability result!

Theorem: For any constant c, if there is a polynomial time c-approximation algorithm for TSP then P=NP.

Non-Euclidean TSP

Proof: Reduction from the Hamiltonian cycle problem. Assume that TSP is c-approximable.

There exists an algorithm *A* that gets a TSP instance *G'* and returns a TS tour whose cost is at most c times the optimal.

In particular, if *G'* has a TS tour of cost n, *A* finds a tour of cost at most cn.

We will use algorithm to solve the Hamiltonian-cycle decision problem.

Non-Euclidean TSP

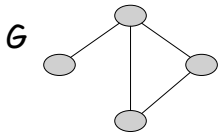
Given G=(V,E) and the question "Is there a HC in G" we construct a TSP instance G', such that there is a HC in G if there is a TS tour of cost at most n in G' and there is no HC in G if the minimal TS tour has cost > cn.

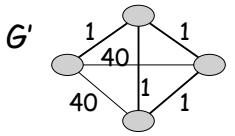
G'=(V,E') is a clique.

The weight of edges in E is 1

The weight of any additional edge is c·n.

- A HC in G corresponds to a TSP tour of weight n.
- Any tour that uses one or more additional edges has cost > cn.





n=4, c=10

Example 3: Bin Packing

The input: A sequence of items (numbers), $a_1,a_2,\ldots,a_n$, such that for all i, $0 < a_i < 1$

The goal: 'pack' the items in bins of size 1. Use as few bins as possible.

Example: The input: 1/2, 1/3, 2/5, 1/6, 1/5, 2/5.

Optimal packing in two bins:

(1/2, 1/3, 1/6), (2/5, 2/5, 1/5).

Legal packing in three bins:

(1/2, 1/3), (2/5, 1/6, 1/5), (2/5)

Approximating Bin Packing

Next-fit Algorithm:

1. Open an *active* bin.
2. For all i=1,2,…,n :
   - If possible, place a_i in the current active bin;
   - Otherwise, open a new active bin and place a_i in it.

Example: The input: {0.3, 0.9, 0.2}.

Next-fit packing (three bins): (0.3), (0.9), (0.2).

Theorem: Next-fit is 2-approximation to BP

Proof: Note: an optimal algorithm must use at least $\Sigma_i a_i$ bins (why?).

Approximating Bin Packing

Analysis of Next Fit: Assume that Next Fit uses h bins. The sum of items sizes in two consecutive bins is greater than 1 (otherwise, we can put them together).

Case 1: h is even:

$c(B_1) + c(B_2) > 1$

$c(B_3) + c(B_4) > 1$

$c(B_{h-1}) + c(B_h) > 1$

$\Sigma_i a_i > h/2$

Case 1: h is odd:

$c(B_1) + c(B_2) > 1$

$c(B_3) + c(B_4) > 1$

$c(B_{h-2}) + c(B_{h-1}) > 1$

$\Sigma_i a_i > (h-1)/2 + c(B_h)$

In both cases, we can obtain $h \leq \lceil 2\Sigma_i a_i \rceil \leq 2opt$

Remark: it can be shown that $h \leq 2opt-1$

Approximating Bin Packing

Is the analysis tight? consider an instance with 4n items {1/2, 1/2n, 1/2, 1/2n, …}.

Next-fit will put any two consecutive items in a bin.

Total number of bin used: 2n.

An optimal packing in n+1 bins: n bins, each with 1/2+1/2, one bin for the tiny items.

The ratio: 2n/(n+1) → 2 as n grows.

Approximating Bin Packing

First fit algorithm: place the next item in the first open bin that can accommodate it. Open a new bin only if no open bin has enough room.

Theorem: $h_{ff} \leq 1.7opt + 2$ (proof not here)

First fit Decreasing: sort the items from largest to smallest. Run FF according to the resulting order.

Theorem: $h_{ffd} \leq 1.222opt + 3$ (proof not here)

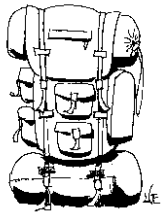
Note: This is not an online algorithm!

Example 4: The Knapsack problem

- You are about to go to a camp.
- There are many items you want to pack.
- You have one knapsack. The total weight you can carry is some fixed number W.
- Every item in your list has some weight, w_i, and some value (benefit), b_i, that measures how much you really need it.
- You need to pack the knapsack in a way that maximizes the total value of the packed items.

The Knapsack problem

Item #	Weight	Value
1	1	8
2	3	6
3	5	5
4	4	6



Max weight =8

A possible packing: Items 2 and 3. Value: 11

An optimal packing: Items 1,2,4. Value: 20

The Knapsack problem is NP-hard.

Greedy Algorithm for Knapsack

1. Consider the items in order of non-increasing b_i/w_i ratio
   $b_1/w_1 \geq b_2/w_2 \geq \ldots \geq b_n/w_n$
2. Add items to the knapsack as long as there is space.

Time Complexity:
O(n log n) (for sorting)
O(n) for packing loop.

→ O(n log n)

Greedy Algorithm for Knapsack

Claim: The approximation ratio of Greedy is not bounded.
Proof: To get ratio c, consider the following instance:
There are two items:
$b_1 = 2$, $w_1 = 1$
$b_2 = 2c$, $w_2 = 2c$

The knapsack has volume W = 2c

Greedy packs only the first item, value = 2.
Optimal: Pack the second item, value=2c
Ratio = c.

Improved Algorithm for Knapsack

Take the maximum of Greedy and the largest value that fits by itself (the most profitable item).

Theorem: The above algorithm is 2-approximation.
Proof: Suppose no weight of a single item exceeds W (these items can be removed in a preprocessing), and that $b_1/w_1 \geq b_2/w_2 \geq \ldots \geq b_n/w_n$
Let B be the largest value, and let G be the value computed by the greedy algorithm.
Let j be the first item that the greedy algorithm rejects

Improved Algorithm for Knapsack

$ALG = \max(B, G) \geq (B + G)/2$

$G = \sum_{i=1}^{j-1} b_i$ (item j is the first to be rejected)

$B \geq b_j$ (B is the most profitable)

$G+B \geq \sum_{i=1}^{j} b_i$ $\quad opt < \sum_{i=1}^{j} b_i$ (why?)

$\rightarrow ALG > opt/2$

Finding an Exact Solution

- Ready to run above-poly time.
- Might be useful for small instances, for problems that need to be solved only once in a while, and for which finding an optimal solution is critical.

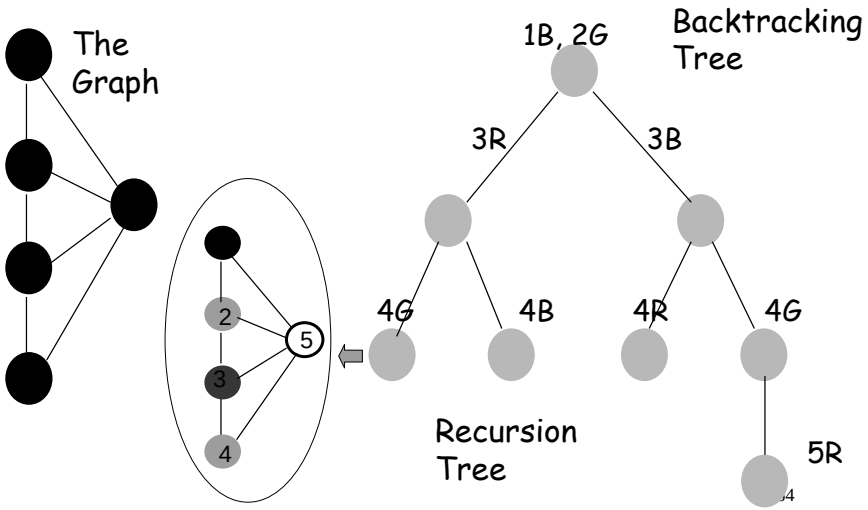
Backtracking

Example: Finding a 3-coloring of a graph

- Explore possibilities; backtrack when doesn't work.
- Start by assigning an arbitrary color to one of the vertices.
- Continue coloring while maintaining the constraints imposed by the edges.
- If reach a vertex that can't be colored, backtrack - go back up the recursion tree and explore other children.

Backtracking

Example: Finding a 3-coloring of a graph



Branch-and-Bound

- Variation for case where the goal is finding a minimum (or maximum) of some objective function
- Example: finding the minimum number of colors needed to color a graph.
- Idea: improve the performance of the algorithm by pruning search when it is known that it can't possibly be going down the optimal path.

Example: Minimum number of colors in graph coloring

- Suppose you build the possible- colorings tree, and at some point you find a valid coloring with k colors.
- Suppose later, after backtracking, you reach a vertex that requires a $(k+1)^{st}$ color → can backtrack.
- In this example, k serves as a bound for backtracking.
- Good B&B algorithms use heuristics that hopefully produce good bounds at early stages of the search.

Branch-and-Bound

Pruning technique

Lower bound computation