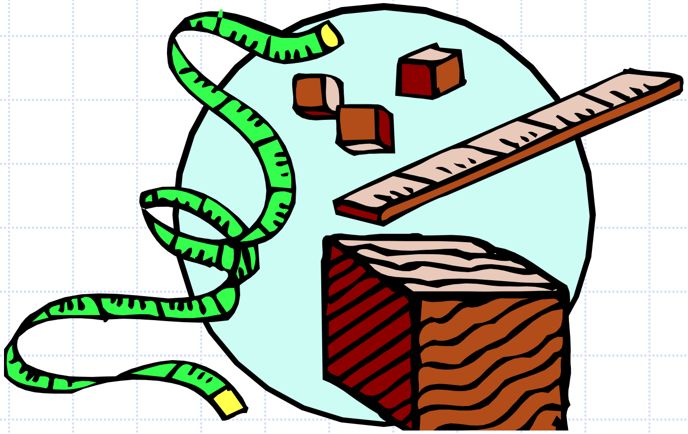
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Coping with NP-completeness



NP-completeness

- Your problem is NP-complete. Now what?
- Is your problem a special case that actually *can* be solved?
  - HORN SAT arises in logic programming and can be solved efficiently
  - If your graph is a tree, many of the NP-complete problems can be solved efficiently
    - E.g., INDEPENDENT SET

NP-completeness

- But your problem isn't one of these
- Perhaps some form of *intelligent* exponential search
  - Backtracking
  - Branch and bound
- These are exponential in the worst case, but could be very efficient for your particular problem

NP-completeness

- OR, you can develop an algorithm that may not find the optimal solution, but will find a solution that falls short of optimum, *but never by too much*
- An algorithm that provides such a guarantee (as in, a quantifiable guarantee) is called an *approximation algorithm*

NP-completeness

- Finally, *heuristics*
  - Algorithms with no guarantees on runtime or degree of approximation
  - Rely on ingenuity, intuition, a good understanding of the application, careful experimentation, and often insights from other fields (e.g., biology, physics) to attack a problem

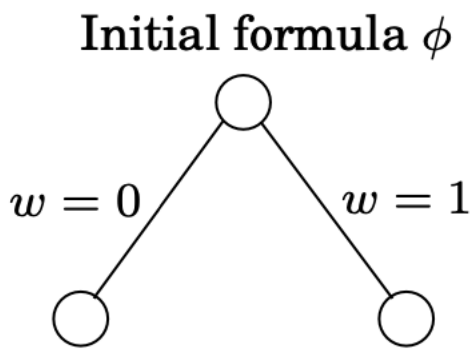
Backtracking

- The idea: sometimes you can eliminate a significant portion of the solution space just by knowing a little information
- E.g. An instance of SAT that contains the clause $(x_1 \vee x_2)$
  - You can eliminate all assignments with $x_1 = x_2 =$ false

Backtracking

Here's how it is done. Consider the Boolean formula $\phi(w, x, y, z)$ specified by the set of clauses

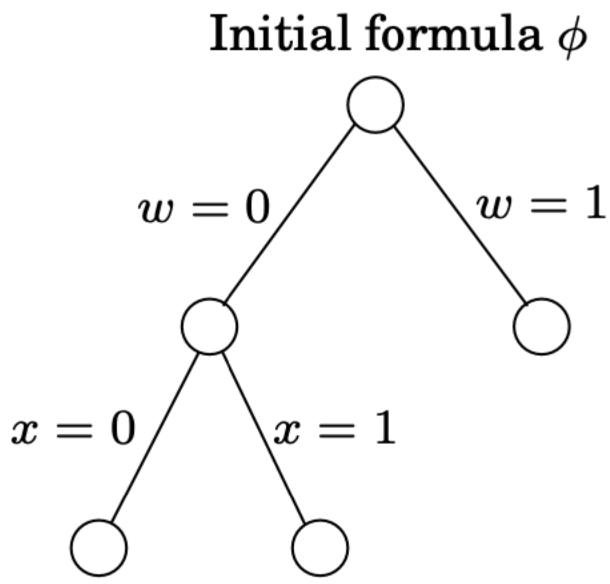
$$(w \vee x \vee y \vee z),\ (w \vee \overline{x}),\ (x \vee \overline{y}),\ (y \vee \overline{z}),\ (z \vee \overline{w}),\ (\overline{w} \vee \overline{z}).$$



Backtracking

Here's how it is done. Consider the Boolean formula $\phi(w,x,y,z)$ specified by the set of clauses

$$(w \vee x \vee y \vee z),\ (w \vee \overline{x}),\ (x \vee \overline{y}),\ (y \vee \overline{z}),\ (z \vee \overline{w}),\ (\overline{w} \vee \overline{z}).$$



Backtracking for SAT

- Explore the space of assignments
- Back out of a cul-de-sac if no solution possible, and continue down a remaining active node
- Grow the tree only if there is uncertainty at a node
- Stop at any stage where a satisfying assignment is found

Backtracking

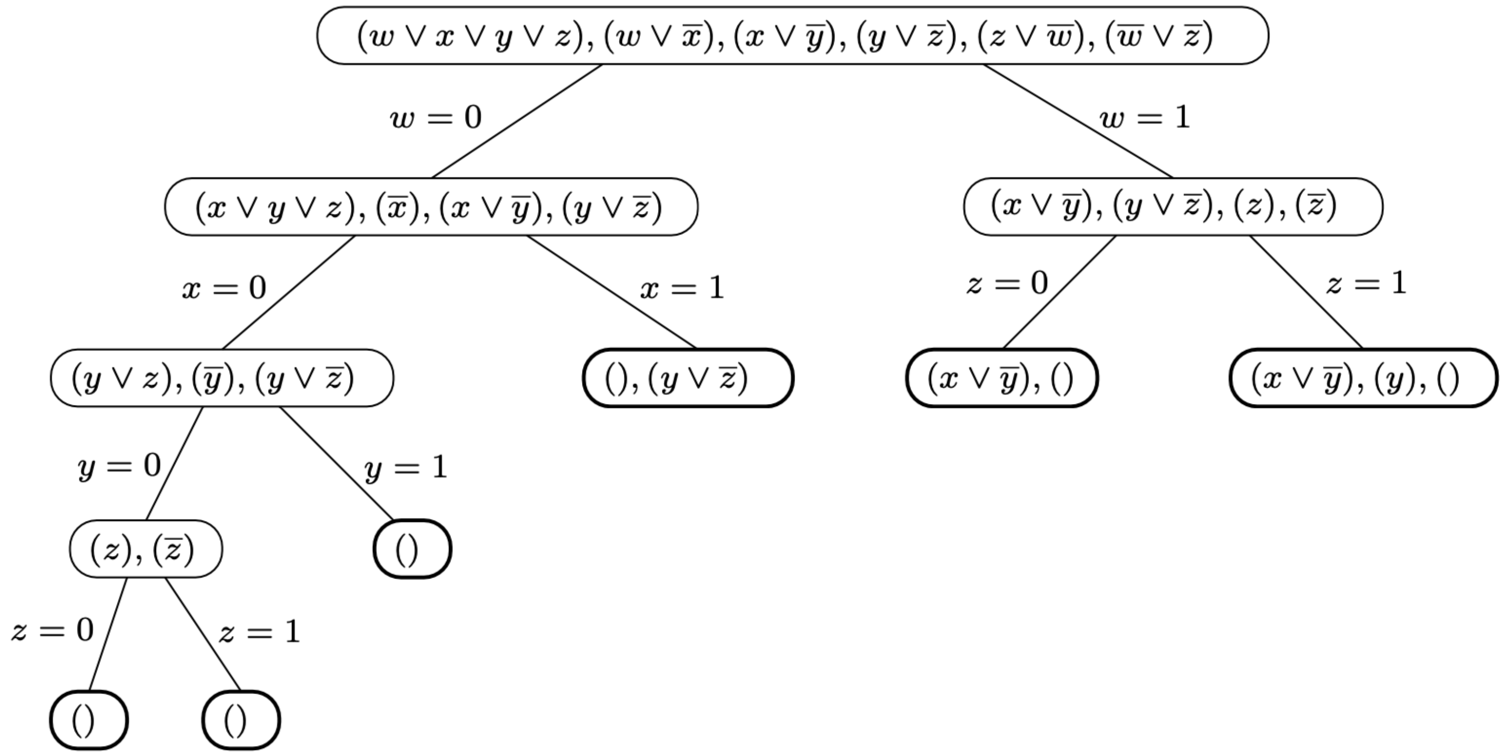
- For SAT each node can be described by the current variable assignments or by the remaining clauses
- The nodes of the subtree are subproblems
- The "empty clause", (), rules out satisfiability
- Given this, how would you choose which variables to consider first?

Backtracking in general

- Backtracking algorithm requires a test of subproblems, quickly declaring one of three possibilities
- Failure: the subproblem has no solution
- Success: A solution to the subproblem is found
- Uncertainty

Backtracking: our SAT example

Figure 9.1 Backtracking reveals that ϕ is not satisfiable.



Backtracking

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
Repeat while S is nonempty:
  choose a subproblem P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If test(P_i) succeeds:  halt and announce this solution
    If test(P_i) fails:  discard P_i
    Otherwise:  add P_i to S
Announce that there is no solution
```

Branch and Bound

- Similar to Backtracking, but applied to optimization problems
- The general idea: if in your search along the solution space you reach a subproblem that you know can’t give you the optimum, you chuck the problem
  - E.g.: TSP problem and you find a subproblem where best possible solution has length 30, but you already know a circuit that has total length 28

Branch and Bound

- Often it is not possible to efficiently determine the exact solution to a subproblem, but can bound that solution (either high or low, depending on the type of optimization)

Branch and Bound

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
bestsofar = ∞
Repeat while S is nonempty:
  choose a subproblem (partial solution) P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If P_i is a complete solution:  update bestsofar
    else if lowerbound(P_i) < bestsofar:  add P_i to S
return bestsofar
```

Branch and Bound: TSP

Let's see how this works for the traveling salesman problem on a graph $G = (V, E)$ with edge lengths $d_e > 0$. A partial solution is a simple path $a \leadsto b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b. We can denote such a partial solution by the tuple $[a, S, b]$—in fact, a will be fixed throughout the algorithm. The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \leadsto a$ with intermediate nodes $V - S$. Notice that the initial problem is of the form $[a, \{a\}, a]$ for any $a \in V$ of our choosing.

Branch and Bound: TSP

Let's see how this works for the traveling salesman problem on a graph $G = (V, E)$ with edge lengths $d_e > 0$. A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b. We can denote such a partial solution by the tuple $[a, S, b]$—in fact, a will be fixed throughout the algorithm. The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V - S$. Notice that the initial problem is of the form $[a, \{a\}, a]$ for any $a \in V$ of our choosing.

At each step of the branch-and-bound algorithm, we extend a particular partial solution $[a, S, b]$ by a single edge (b, x), where $x \in V - S$. There can be up to $|V - S|$ ways to do this, and each of these branches leads to a subproblem of the form $[a, S \cup \{x\}, x]$.

Branch and Bound: TSP

Let's see how this works for the traveling salesman problem on a graph $G = (V, E)$ with edge lengths $d_e > 0$. A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b. We can denote such a partial solution by the tuple $[a, S, b]$—in fact, a will be fixed throughout the algorithm. The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V - S$. Notice that the initial problem is of the form $[a, \{a\}, a]$ for any $a \in V$ of our choosing.

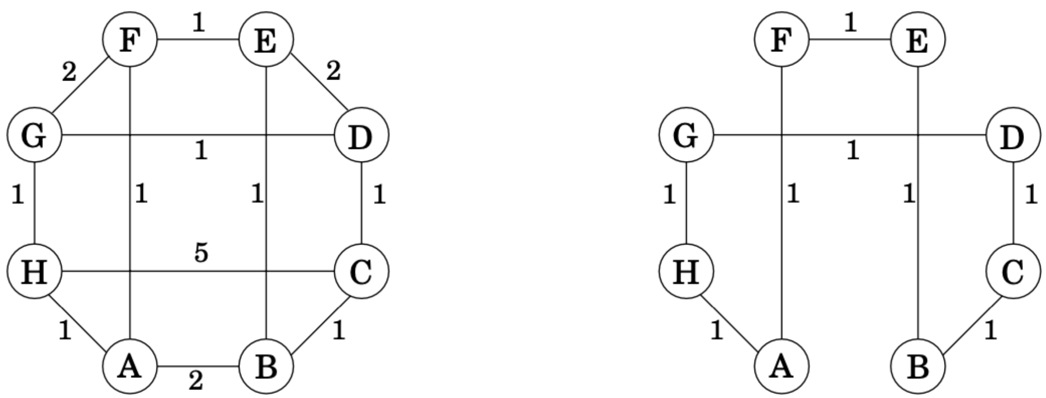
How can we lower-bound the cost of completing a partial tour $[a, S, b]$? Many sophisticated methods have been developed for this, but let's look at a rather simple one. The remainder of the tour consists of a path through $V - S$, plus edges from a and b to $V - S$. Therefore, its cost is at least the sum of the following:

1. The lightest edge from a to $V - S$.
2. The lightest edge from b to $V - S$.
3. The minimum spanning tree of $V - S$.

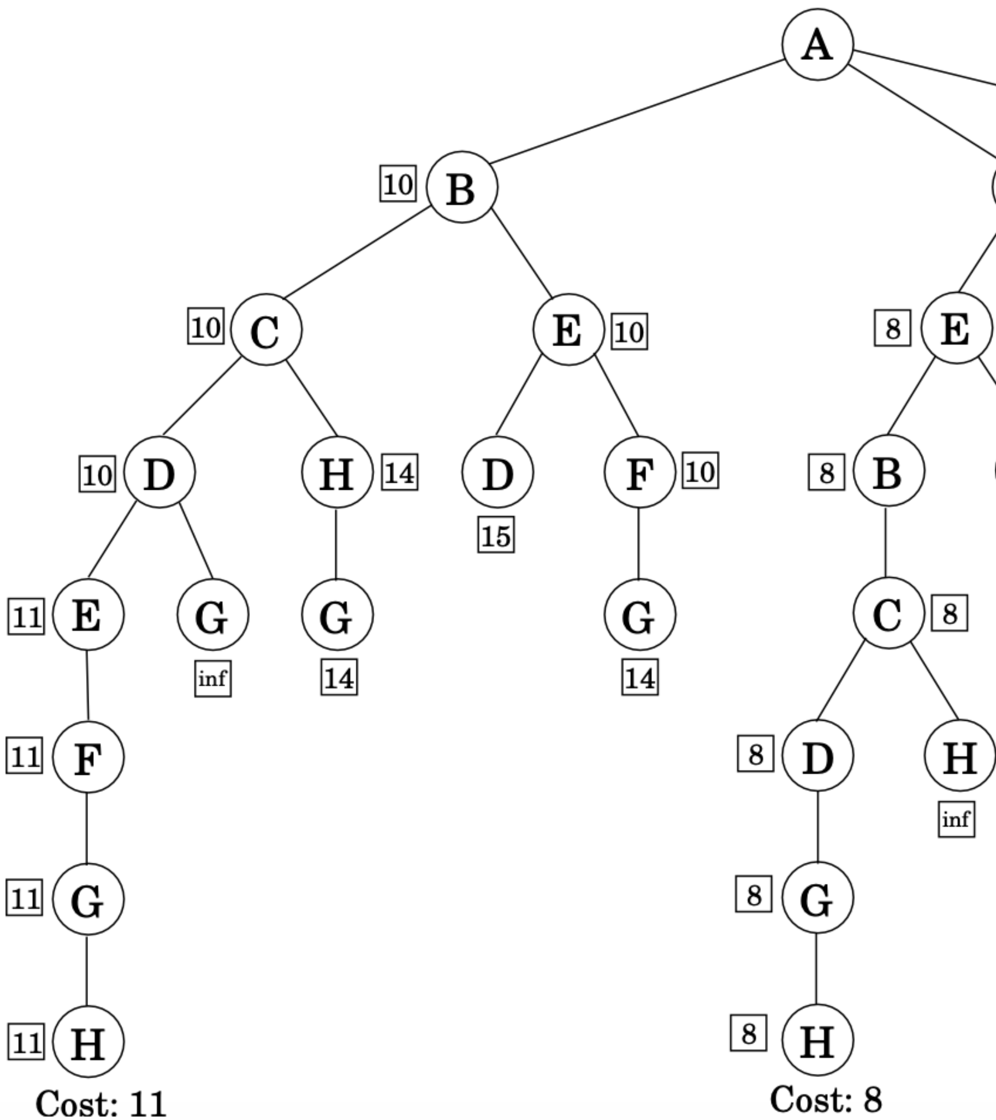
Branch and Bound: TSP

Figure 9.2 (a) A graph and its optimal traveling salesman tour. (b) The branch-and-bound search tree, explored left to right. Boxed numbers indicate lower bounds on cost.

(a)



Branch and Bound: TSP

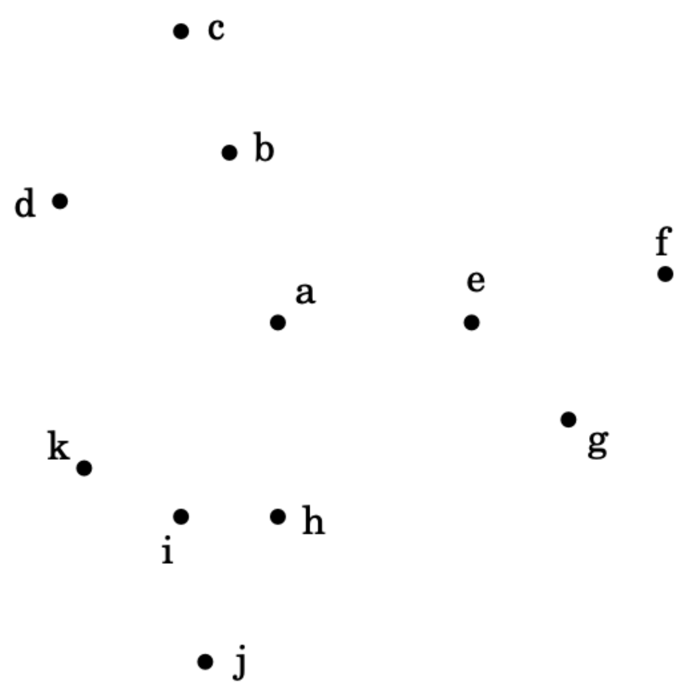


28 partial solutions considered, instead of 5,040 with brute force

Approximation Algorithms

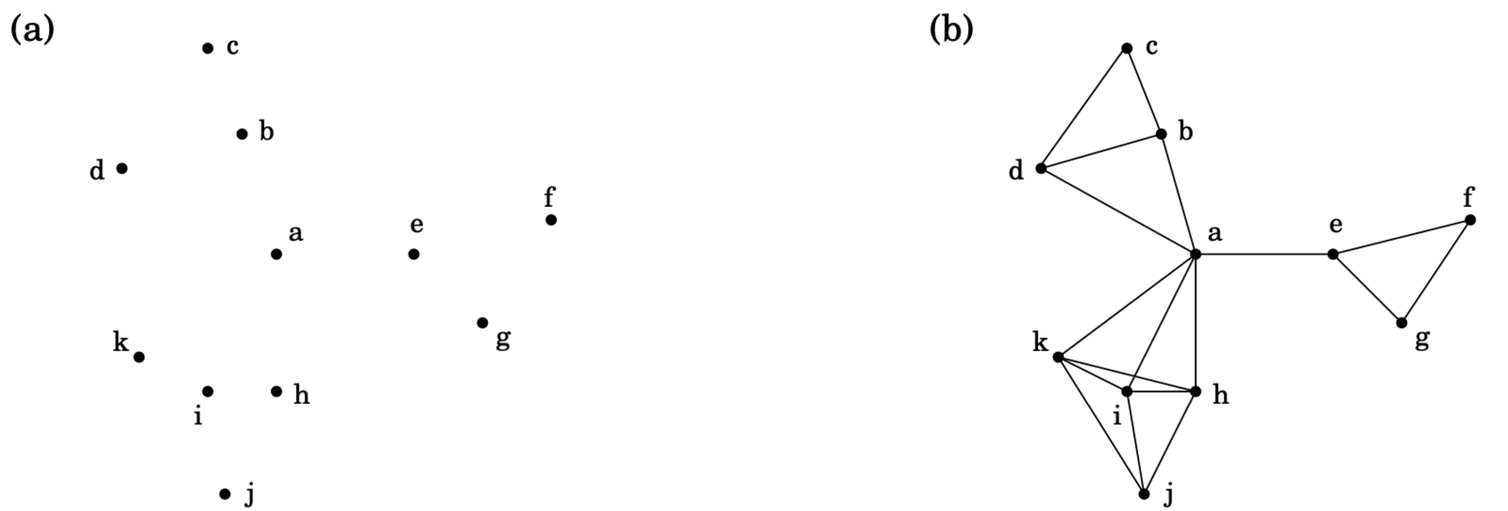
- Let's start with an example problem

The dots in Figure 5.11 represent a collection of towns. This county is in its early stages of planning and is deciding where to put schools. There are only two constraints: each school should be in a town, and no one should have to travel more than 30 miles to reach one of them. What is the minimum number of schools needed?



Approximation Algorithms

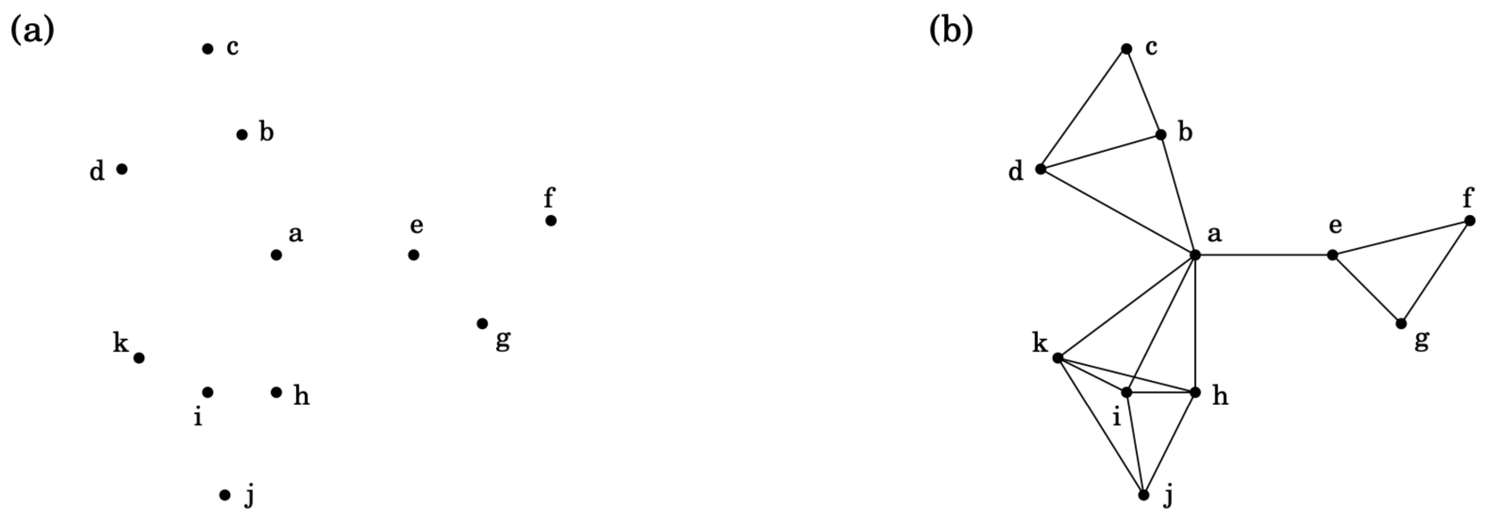
Figure 5.11 (a) Eleven towns. (b) Towns that are within 30 miles of each other.



We've already seen this problem.
Do you recognize it?

Approximation Algorithms

Figure 5.11 (a) Eleven towns. (b) Towns that are within 30 miles of each other.



This is a typical *set cover* problem. For each town x, let S_x be the set of towns within 30 miles of it. A school at x will essentially "cover" these other towns. The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

SET COVER

Input: A set of elements B; sets $S_1, \ldots, S_m \subseteq B$

Output: A selection of the S_i whose union is B.

Cost: Number of sets picked.

Approximation Algorithms

- What do we know about this problem?

This is a typical *set cover* problem. For each town x, let S_x be the set of towns within 30 miles of it. A school at x will essentially "cover" these other towns. The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

SET COVER

Input: A set of elements B; sets $S_1, \ldots, S_m \subseteq B$

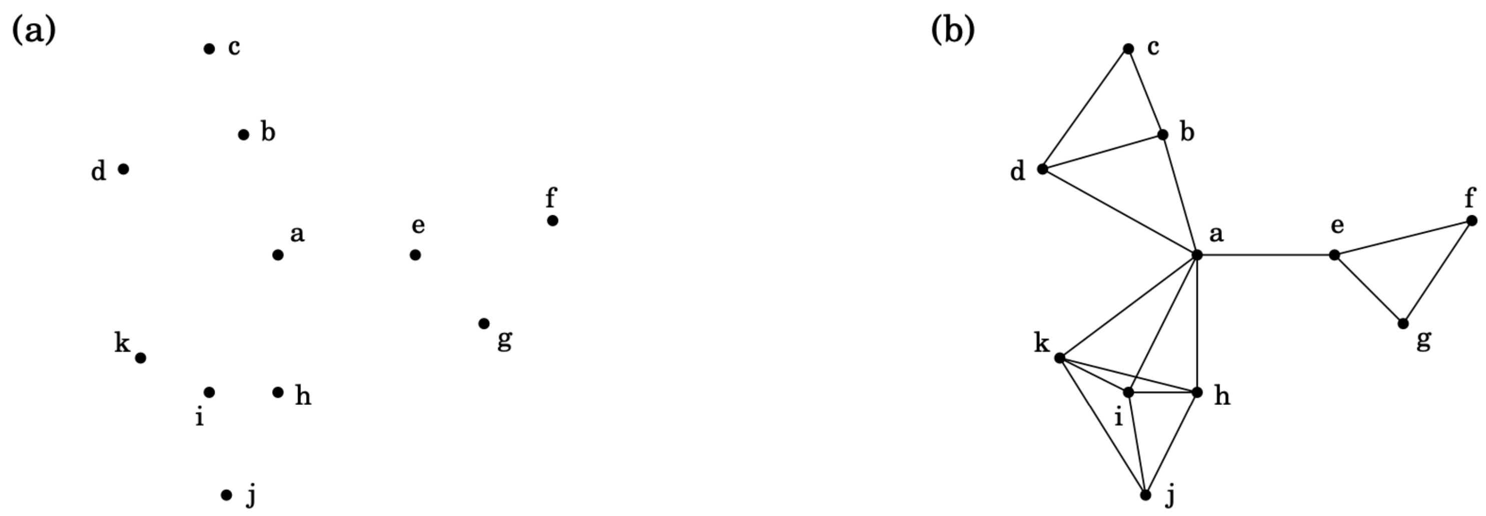
Output: A selection of the S_i whose union is B.

Cost: Number of sets picked.

Approximation Algorithms

- Assuming you don't know what we know, what might you try?

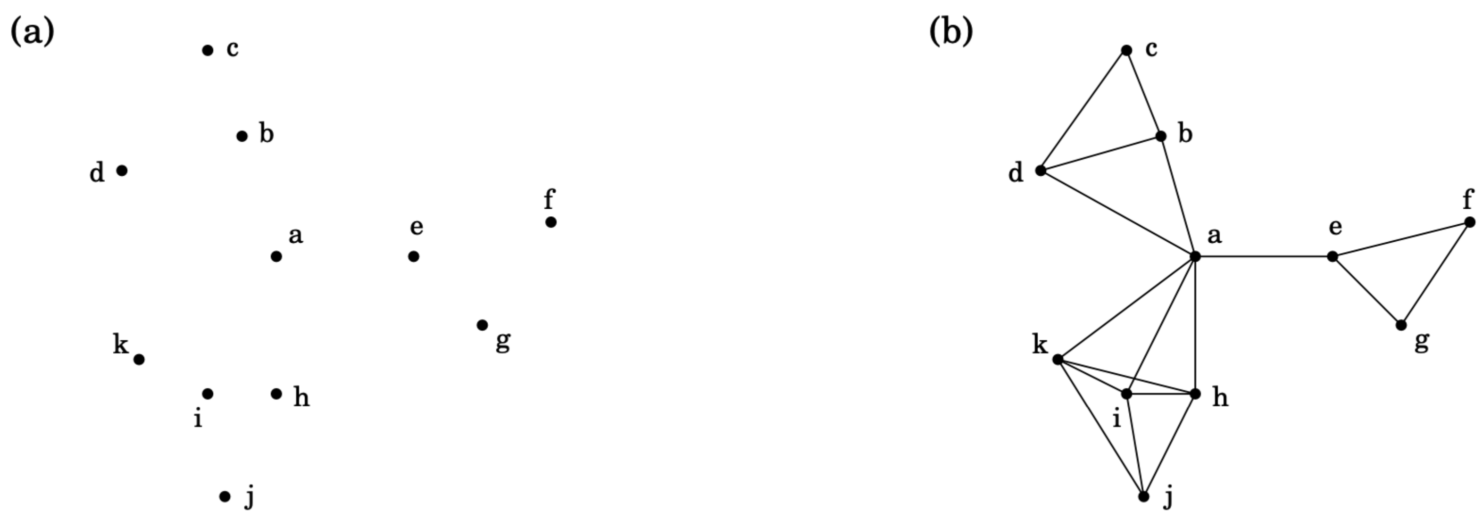
Figure 5.11 (a) Eleven towns. (b) Towns that are within 30 miles of each other.



Approximation Algorithms

- Assuming you don't know what we know, what might you try?
- Greedy gives: a, c, j, and f or g
- Optimal is b,e,i

Figure 5.11 (a) Eleven towns. (b) Towns that are within 30 miles of each other.



Approximation Algorithms

- But…

Claim *Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

Approximation Algorithms

- But…

Claim *Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

- Let n_t be the number of elements still not covered after t iterations of the greedy algorithm
  - So $n_0 = n$
- Claim?

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t\left(1 - \frac{1}{k}\right),$$

Approximation Algorithms

- But…

Claim *Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

- Let n_t be the number of elements still not covered after t iterations of the greedy algorithm
  - So $n_0 = n$
- Claim:

$$n_t \leq n_0(1 - 1/k)^t$$

Approximation Algorithms

- But…

Claim *Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

- So what happens when t = k ln n?

$$n_t \;\leq\; n_0\left(1-\frac{1}{k}\right)^t \;<\; n_0(e^{-1/k})^t \;=\; ne^{-t/k}.$$

Approximation Algorithms

- But…

Claim *Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

- Thus we have an approximation algorithm with an *approximation factor* of ln n

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t < n_0 (e^{-1/k})^t = n e^{-t/k}.$$

Approximation Algorithms

- Let's look at some more of these
- Notation: for a problem instance I, let OPT(I) denote the value of the optimum solution to problem I
- And one simplifying assumption: OPT(I) is always a positive integer
  - Which is generally the case

Approximation Algorithms

More generally, consider any minimization problem. Suppose now that we have an algorithm $\mathcal{A}$ for our problem which, given an instance I, returns a solution with value $\mathcal{A}(I)$. The *approximation ratio* of algorithm $\mathcal{A}$ is defined to be

$$\alpha_{\mathcal{A}} = \max_I \frac{\mathcal{A}(I)}{\text{OPT}(I)}.$$

What do we do if it's a maximization problem?

Approximate: Vertex Cover

VERTEX COVER

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

Output: A subset of the vertices $S \subseteq V$ that touches every edge.

Goal: Minimize $|S|$.

Does this look similar to anything already considered?

Approximate: VERTEX COVER

VERTEX COVER

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

Output: **A subset of the vertices** $S \subseteq V$ **that touches every edge.**

Goal: **Minimize** $|S|$.

Does this look similar to anything already considered?

It should. It's a special case of SET COVER
Which we can approximate using a greedy approach
with approximation ratio log n

Better Appx: VERTEX COVER

- We can do better
- Given a graph G, a *matching* is a subset of edges that have no vertices in common
- A matching is *maximal* if no more edges can be added
- Maximal matchings will help us find good covers
  - And they are easy to find!
  - Can you figure out how?

Better Appx: VERTEX COVER

- Claim: Any vertex cover of G must be at least as large as the number of edges in any matching of G
  - Prove it!

Better Appx: VERTEX COVER

- Claim: Any vertex cover of G must be at least as large as the number of edges in any matching of G
  - Prove it!
- Another claim: Let S be a set that contains both vertices of each edge in a maximal matching M. Then S is a vertex cover of G.
  - Prove it!

Better Appx: VERTEX COVER

- So, the algorithm
  - Find a maximal matching M of G
  - Return S = all endpoints of edges in S
- How good is this cover?

Approximation: k-CLUSTER

- The problem:
  - There is data that one wishes to divide into groups
  - There is some notion of distance
    - Can be usual distance, or some more general metric

1. $d(x, y) \geq 0$ **for all** x, y.
2. $d(x, y) = 0$ **if and only if** $x = y$.
3. $d(x, y) = d(y, x)$.
4. **(Triangle inequality)** $d(x, y) \leq d(x, z) + d(z, y)$.

Approximation: k-CLUSTER

- The problem:

We would like to partition the data points into groups that are compact in the sense of having small diameter.

k-CLUSTER

Input: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance metric $d(\cdot, \cdot)$; integer k.

Output: A partition of the points into k clusters $C_1, \ldots, C_k$.

Goal: Minimize the diameter of the clusters,

$$\max_j \max_{x_a, x_b \in C_j} d(x_a, x_b).$$

This problem is NP-hard

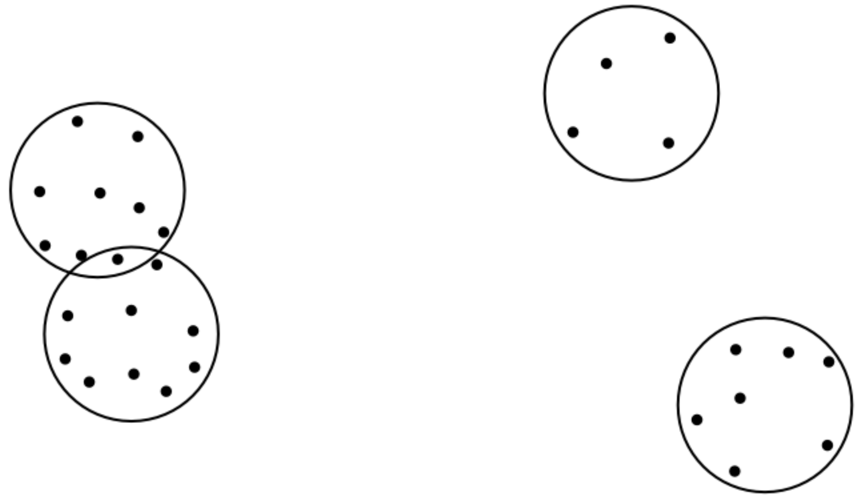
Approximation: k-CLUSTER

```
Pick any point μ₁ ∈ X as the first cluster center
for i = 2 to k:
  Let μᵢ be the point in X that is farthest from μ₁, ..., μᵢ₋₁
  (i.e., that maximizes min_{j<i} d(·, μⱼ))
Create k clusters:  Cᵢ = {all x ∈ X whose closest center is μᵢ}
```

But has a simple approximation algorithm

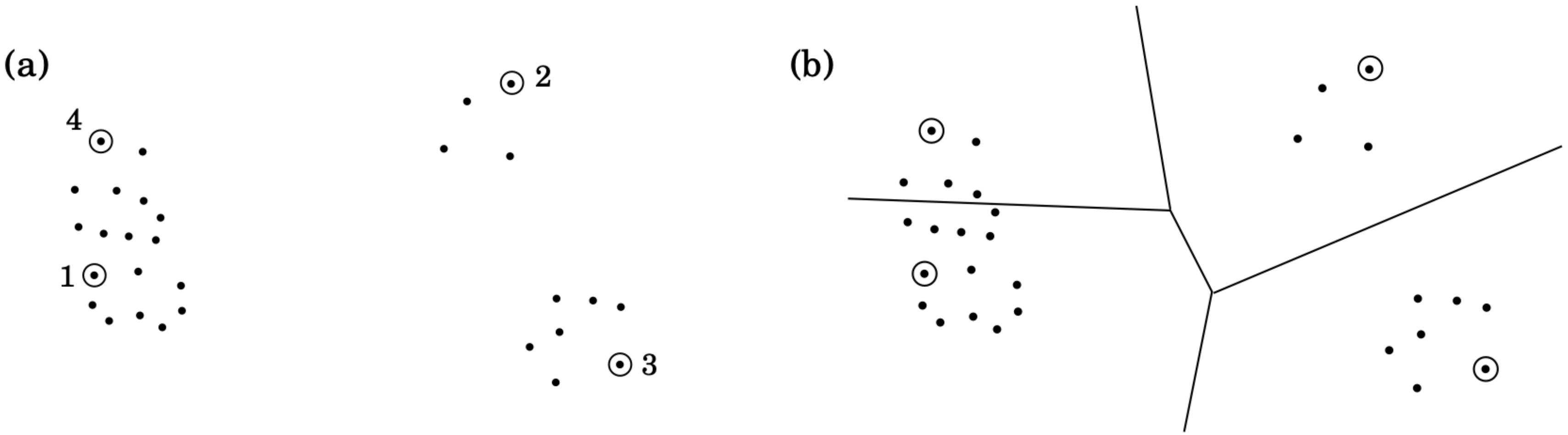
Approximation: k-CLUSTER

Figure 9.5 Some data points and the optimal $k = 4$ clusters.



Approximation: k-CLUSTER

Figure 9.6 (a) Four centers chosen by farthest-first traversal. (b) The resulting clusters.



Approximation: k-CLUSTER

Pick any point $\mu_1 \in X$ as the first cluster center
for $i = 2$ to k:
  Let μ_i be the point in X that is farthest from $\mu_1, \ldots, \mu_{i-1}$
  (i.e., that maximizes $\min_{j<i} d(\cdot, \mu_j)$)
Create k clusters: $C_i = \{$all $x \in X$ whose closest center is $\mu_i\}$

Claim: the diameter of the resulting clustering is no worse than twice the optimal

Prove it!

Hint: Consider the next cluster center x you would add if you wanted k+1 clusters, and let r be its distance to the closest of the k centers.

Approximation: k-CLUSTER

Pick any point $\mu_1 \in X$ as the first cluster center
for $i = 2$ to k:
  Let μ_i be the point in X that is farthest from $\mu_1, \ldots, \mu_{i-1}$
  (i.e., that maximizes $\min_{j<i} d(\cdot, \mu_j)$)
Create k clusters: $C_i = \{$all $x \in X$ whose closest center is $\mu_i\}$

Claim: the diameter of the resulting clustering is no worse than twice the optimal

Prove it!

Hint: Now consider the points (μ_1, μ_2, μ_3,□ μ_κ,x). At least two of them must be in the same cluster…

Approximation: k-CLUSTER

Pick any point $\mu_1 \in X$ as the first cluster center
for $i = 2$ to k:
  Let μ_i be the point in X that is farthest from $\mu_1, \ldots, \mu_{i-1}$
  (i.e., that maximizes $\min_{j<i} d(\cdot, \mu_j)$)
Create k clusters: $C_i = \{$all $x \in X$ whose closest center is $\mu_i\}$

So, the approximation factor for this is 2.
As far as I am aware, there is no known approximation algorithm for this that does better.

Approximation: TSP

- Important: we are assuming the triangle inquality is valid for our metric

Let’s start with this:
Can you compare the length of
the optimal tour with the length of
a minimum spanning tree?

Approximation: TSP

- Important: we are assuming the triangle inquality is valid for our metric

Let's start with this…

$$\text{TSP cost} \geq \text{cost of this path} \geq \text{MST cost.}$$

Approximation: TSP

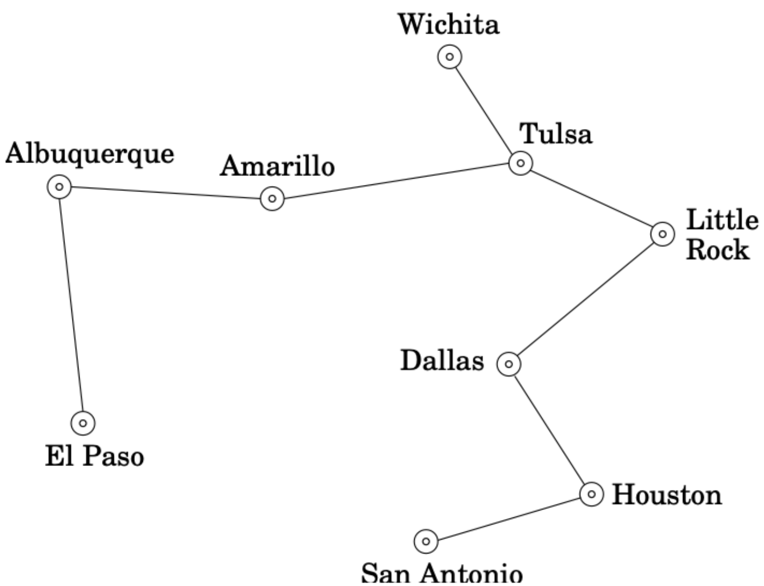
- Important: we are assuming the triangle inquality is valid for our metric

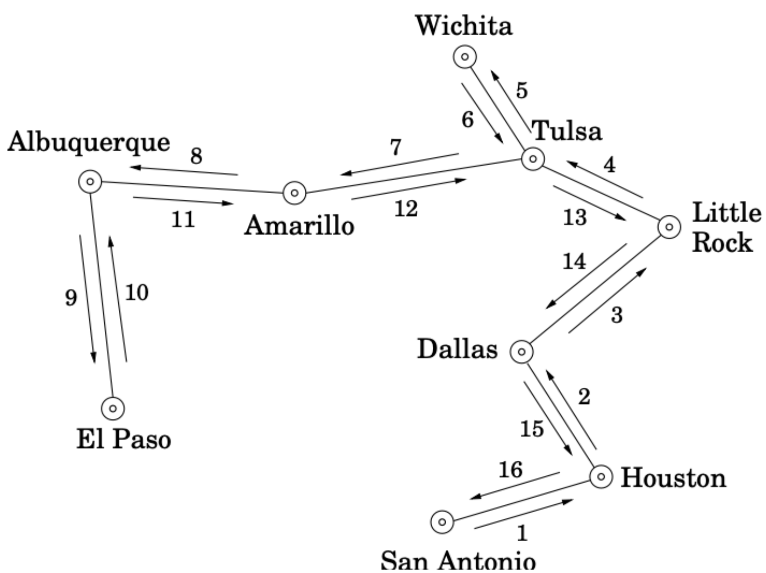
Then use the (an) MST to build a tour…

$$\text{TSP cost} \geq \text{cost of this path} \geq \text{MST cost.}$$

Approximation: TSP

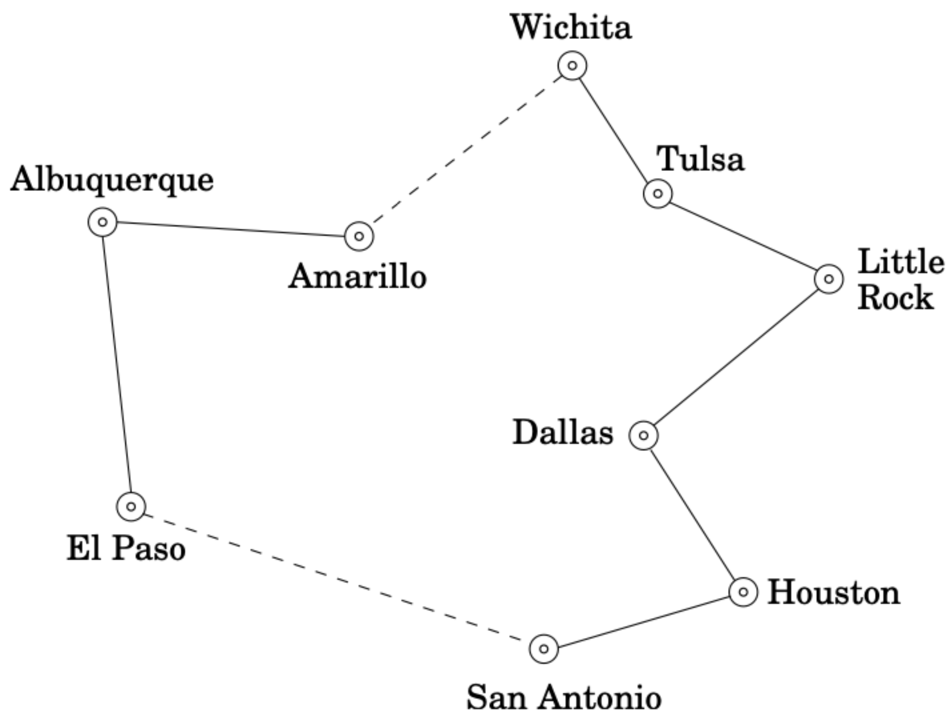
- But this is not a tour. How do we fix that?





Approximation: TSP

- But this is not a tour. How do we fix that?



Approximation: TSP

- What if we do not assume the triangle inequality?
- Well, then the existance of an efficient approximation algorithm for the problem would imply that P=NP.
  - Needless to say, we don't have one