

CS 170

Efficient Algorithms and Intractable Problems

Lecture 23

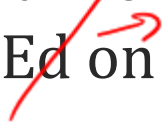
Coping with NP-Completeness II

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Nika's OH as usual on Tuesdays (today) after class
→ Meet at the podium or the lecture hall entrance.

End-of-semester course evaluations are open now
→ You can receive an additional homework drop if you fill it out (see "End-of-Semester Feedback Form" on Ed  on how to receive HW drop)

Midterm 2 regrade requests will be resolved by end of the week

Strategies for coping with NP-Completeness!

1. Approximation Algorithms:

→ You've seen 2-approx. for Vertex Cover

→ You've seen 2-approx. for Metric TSP

→ You've seen $\log(n)$ -approx. for Set-Cover

→ Today: $1 - \epsilon$ -approx. for Knapsack problems.

} Last lecture!

→ Lecture 10 on greedy algs!

2. Exact algorithms that are fast in practice (even if exponential time in theory!)

○ Intelligent exhaustive search

→ Backtracking (for decision problems)

→ Branch-and-bound (for optimization problems)

3. Heuristics

○ Local search algorithms, etc.

} We won't cover in class

Knapsack

Recall Knapsack (no repetition)

Input: A weight capacity W , and n items $(w_1, v_1), \dots, (w_n, v_n)$. All integers.

Output: Most valuable subset of items (no repetition), whose total weight is $\leq W$.

Step1: Subproblems: For all $c \leq W$ and all $j \leq n$

$K(j, c)$ = best value achievable for knapsack of **capacity c** using only **items $1, \dots, j$**

$$K(j, c) = \max \left\{ \underbrace{K(j-1, c)}_{\text{not use } j} + \underbrace{v_j + K(j-1, c-w_j)}_{\text{use } j\text{th item}} \right\} \leftarrow$$

Algorithm:

- Fill out the array of size nW ,
- each cell take max over 2 values

Runtime: Pseudo-polynomial time $O(nW)$

- The input size is $O(n \log(W))$ so this is not truly a polynomial time algorithm.

Is there a truly polynomial time algorithm for knapsack?

For any $0 < \epsilon < 1$, we will give an **approximation algorithm** that runs in time $O(n^3/\epsilon)$ and finds a subset of items of value **Val** where

$$\text{Val} \geq \text{OPT} (1 - \epsilon),$$

Where **OPT** is the value of the optimal solution to the knapsack.

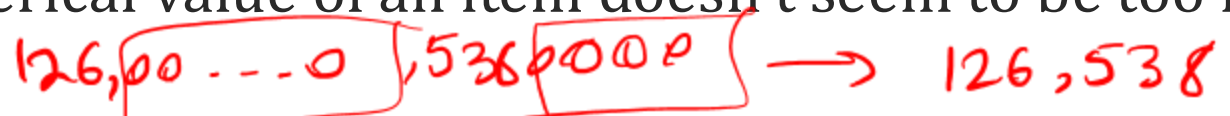
An Approach: Rounding Down Item Values

Input: A weight capacity W , and n items $(w_1, v_1), \dots, (w_n, v_n)$. All integers.

Output: Most valuable subset of items (no repetition), whose total weight is $\leq W$.

Idea: Solve for the optimal solution of a slightly approximated instance.

→ Using the exact numerical value of an item doesn't seem to be too important.

→ Round them down 

Values: 126,378,210 and 538,943,121, ... become: 126 and 538

→ For this to work, we need an algorithm for exact knapsack that would run faster than $O(nW)$ when the item values are small.

Alternative exact Alg for Knapsack

When item values (v_i s are integers), there is a DP algorithm (call it **Knapsack-II**) that finds the optimal knapsack solution in $O(nV)$, where $V = \sum_{i=1}^n v_i$.

Knapsack Approximation Algorithm

$m=10$ 2.2
 $v=22$
 $\lfloor \frac{v}{m} \rfloor = 2$

Knapsack-approx($W, (w_1, v_1), \dots, (w_n, v_n), \epsilon$)

Assume no item with $w_n > W$ (or just remove them before you start)

→ $m \leftarrow ?$

// We'll discuss the choice of m later

For $i = 1, \dots, n, \hat{v}_i \leftarrow \lfloor v_i/m \rfloor$

$w_i \leq W$

Run **Knapsack-II** with values \hat{v}_i (not changing the capacity or the weights), output the resulting choice of items.

Plan for our analysis:

1. (Feasibility) Verify that Knapsack-approx. returns a feasible solution, i.e., its total weight is less than W . ✓ Knapsack-II with $w_1, \dots, w_n, W \leftarrow$
- ? 2. (Approximation factor) Verify that for a good choice of m , Knapsack-approx. has a value that's at least $(1 - \epsilon) \times$ value of the optimal solution in the original instance.
- ? 3. (Runtime) Verify that for the choice of m discussed in step 2, the runtime of Knapsack-approx is polynomial time.

Analysis of the Approximation Factor

Knapsack-approx($W, (w_1, v_1), \dots, (w_n, v_n), \epsilon$)

Assume no item with $w_n > W$ (or just remove them before you start)

$m \leftarrow$ // We'll discuss the choice of m

For $i = 1, \dots, n, \hat{v}_i \leftarrow \lfloor v_i/m \rfloor$

Run **Knapsack-II** with values \hat{v}_i (not changing the capacity or the weights), output the resulting choice of items.

Discuss: How does the value of m affect the approximate values?

Choose the correct answer:

1. $\hat{v}_i \in [v_i - m, v_i]$
2. $\hat{v}_i \in [v_i, v_i + m]$

3. $m\hat{v}_i \in [v_i - m, v_i]$
4. $m\hat{v}_i \in [v_i, v_i + m]$

$\lfloor \frac{v_i}{m} \rfloor$ at most $\frac{1}{m}$ shares. $\frac{v_i}{m} - \frac{1}{m} \leq \hat{v}_i \leq \frac{v_i}{m}$

$$v_i - m \leq \hat{v}_i \leq v_i \quad \leftarrow$$

Step 2: Analysis of the Approximation Factor

Let \hat{S} be the outcome of Knapsack-approx Alg, and let S^* be the optimal knapsack solution to the original problem. We will prove that

$$\underbrace{\sum_{i \in \hat{S}} v_i}_{\text{Val}} \geq \underbrace{\sum_{i \in S^*} v_i}_{\text{OPT}} - nm$$

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

By discussion. By discussion. $\leq \text{OPT} - nm$

\hat{S} is optimal for \hat{v}_i OPT \hat{S} has at most n items.

$$\text{Val} \geq \text{OPT} - nm$$

Step 2: Analysis of the Approximation Factor

We proved that $\sum_{i \in \hat{S}} v_i \geq \sum_{i \in S^*} v_i - nm$ already. How should we set m so that this gives $\text{Val} \geq (1 - \epsilon) \text{OPT}$?

$\text{Val} \geq \text{OPT} - nm$

$\text{Val} \geq \text{OPT} - \epsilon \cdot \text{OPT}$

I need: $nm \leq \epsilon \cdot \text{OPT} \Rightarrow n \cdot m \leq \epsilon \cdot \text{OPT}$

$n \cdot m \leq \epsilon \cdot \text{OPT}$

$\frac{\epsilon \cdot v_{\max}}{n}$

$\text{OPT} \geq v_{\max}$

$v_{\max} = \max_{i \in [n]} v_i$

$m := \frac{\epsilon \cdot v_{\max}}{n}$

(Recall all items $w_i \leq W$)

Step 3: Analysis of the Runtime

Knapsack-approx($W, (w_1, v_1), \dots, (w_n, v_n), \epsilon$)

Assume no item with $w_n > W$ (or just remove them before you start)

$m \leftarrow \frac{\epsilon \cdot v_{\max}}{n}$ // We'll discuss the choice of m

For $i = 1, \dots, n, \hat{v}_i \leftarrow \lfloor v_i/m \rfloor$

Run **Knapsack-II** with values \hat{v}_i (not changing the capacity or the weights), output the resulting choice of items.

What is the runtime of Knapsack-approx.?

Knapsack-II $O(n \hat{V})$

$$\hat{V} = \sum_{i \in I} \hat{v}_i$$

$$\hat{V} = \sum_{i=1}^n \hat{v}_i \leq \sum_{i=1}^n \frac{v_i}{m} \leq n \cdot \frac{v_{\max}}{m} = \frac{n v_{\max} n}{\epsilon \cdot v_{\max}} = \frac{n^2}{\epsilon}$$

Total Runtime $O(n^3/\epsilon)$

The making of Knapsack-II Algorithm

So far we assumed that when item values (v_i s are integers), there is a DP algorithm (call it **Knapsack-II**) 1 that finds the optimal knapsack solution in $O(nV)$, where $V = \sum_{i=1}^n v_i$.

Let's see this algorithm

Lec 13 Alg

$A(j, v)$ = lightest weight
achievable by a set of items $1, \dots, j$
whose value is **at least value v** .
(∞ if no such subset exists)

$$A(j, v) = \min \left\{ \begin{array}{l} A(j-1, v) \\ w_j + A(j-1, v - v_j) \end{array} \right\}$$

not use item j use item j

Return: highest value achievable by $\leq W$
using n items.
 $\max \{ K, A(n, K) \leq W \}$

$K(j, c)$ = best value achievable using
subset of only **items $1, \dots, j$** with total
weights at most c .
(0 if no such subset)

$$K(j, c) = \max \left\{ \begin{array}{l} K(j-1, c) \\ v_j + K(j-1, c - w_j) \end{array} \right\}$$

Return $K(n, W)$

Knapsack-II Pseudo-code

Input: A weight capacity W , and n items $(w_1, v_1), \dots, (w_n, v_n)$. All integers.

Output: Most valuable subset of items (no repetition), whose total weight is $\leq W$.

$A(j, v)$ = lightest weight
achievable by a set of items $1, \dots, j$
whose value is **at least value v** .

Maximum v to consider is $V = \sum_{i=1}^n v_i$ (can't get a higher value)

Runtime: $O(nV)$
→ # subproblems: $O(nV)$
→ Runtime per subproblem: $O(1)$

Knapsack-II($W, (w_1, v_1), \dots, (w_n, v_n)$)

An array A of size $(n + 1) \times V$ initialized to ∞

$A[0, 0] \leftarrow 0$

For $j = 1, \dots, n$:

For $v = 1, \dots, V$,

IF $v_i > v$, **then** $A[j, v] = A[j - 1, v]$

ELSE $A[j, v] = \min \left\{ \begin{array}{l} A(j - 1, v), \\ w_j + K(j - 1, v - v_j) \end{array} \right\}$

return $\max \{ K \mid A(n, K) \leq W \}$

Can also track $S[j, v]$ the corresponding item set

Overview of Approximation Algorithms so far!

Constant approx. is pretty good! Sometimes that's the best you can do (unless $P=NP$)

→ E.g., It is hard to approximate Vertex Cover to better than 2-approximation and set-cover to better than $O(\log(n))$ -approximation.

Getting $(1 - \epsilon)$ -approximation with runtime $poly\left(\frac{1}{\epsilon}\right)$ is really great!

→ This is called a Fully Polynomial Time Approximation Algorithm (FPTAS). *Schen*

→ If the runtime is not $poly\left(\frac{1}{\epsilon}\right)$, this is still quite nice. It is called having a Polynomial Time Approximation Algorithm (PTAS). *Schen*

→ There is a PTAS for metric TSP!

What if we are not satisfied with the approximation guarantees of polynomial time algorithms? Let's see after the break!

Intelligent Exhaustive Search; Backtracking

We can do an exhaustive search but in a smart way.

→ It is often possible to reject one style of solutions by looking only at a small part of it.

Backtracking for SAT.

→ Start with the entire formula (root of the tree)

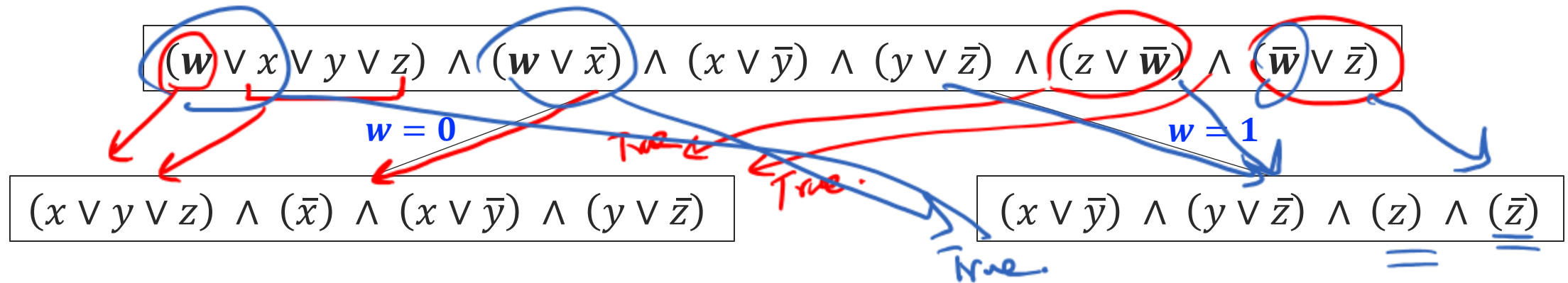
→ Grow the tree: In each node, branch out on the value of one variable.

→ If you can verify quickly that there is no way to satisfy the formula with the current partial solution, don't expand.

Backtracking on SAT Instances

$$(w \vee x \vee y \vee z) \wedge (w \vee \bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z \vee \bar{w}) \wedge (\bar{w} \vee \bar{z})$$

Backtracking on SAT Instances

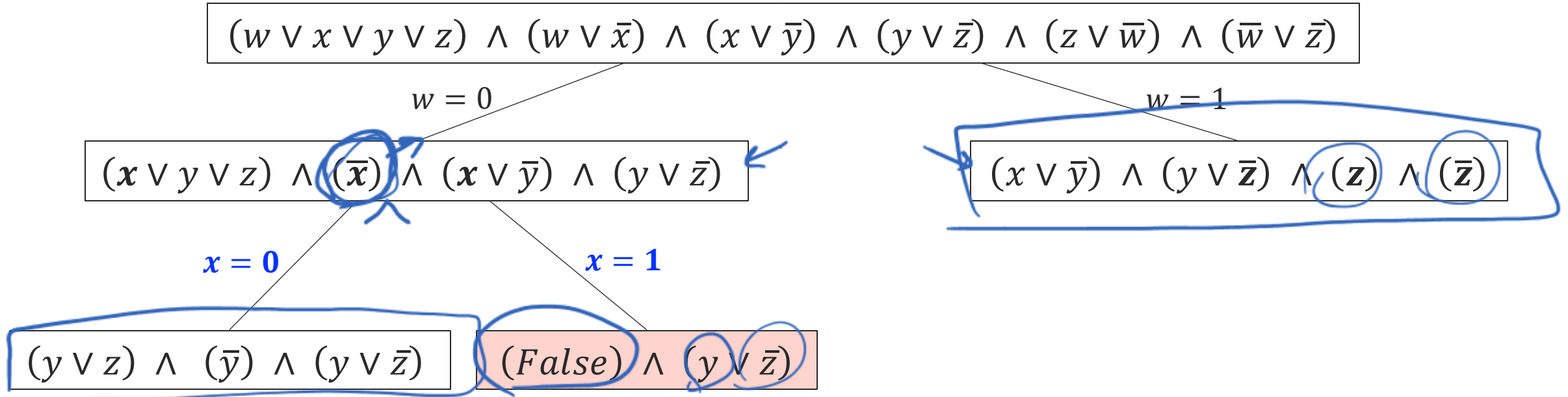


If a clause is satisfied, just remove it.

Eg. When $w = 0$, clauses $(z \vee \bar{w})$ and $(\bar{w} \vee \bar{z})$ are already satisfied!

So they are removed from the left branch

Backtracking on SAT Instances



Heuristically:

We chose subproblem that includes the smallest clause and expanded on the variable in the smallest clause

Backtracking on SAT Instances

$$(w \vee x \vee y \vee z) \wedge (w \vee \bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z \vee \bar{w}) \wedge (\bar{w} \vee \bar{z})$$

$w = 0$

$w = 1$

$$(x \vee y \vee z) \wedge (\bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z})$$

$$(x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z) \wedge (\bar{z})$$

$x = 0$

$x = 1$

$$(y \vee z) \wedge (\bar{y}) \wedge (y \vee \bar{z})$$

$$(False) \wedge (y \vee \bar{z})$$

$y = 0$

$y = 1$

$$(z) \wedge (\bar{z})$$

$$(False)$$

Heuristically:

We chose subproblem that includes the smallest clause and expanded on the variable in the smallest clause

Backtracking on SAT Instances

$$(w \vee x \vee y \vee z) \wedge (w \vee \bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z \vee \bar{w}) \wedge (\bar{w} \vee \bar{z})$$

$w = 0$

$w = 1$

$$(x \vee y \vee z) \wedge (\bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z})$$

$$(x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z) \wedge (\bar{z})$$

$x = 0$

$x = 1$

$$(y \vee z) \wedge (\bar{y}) \wedge (y \vee \bar{z})$$

$$(False) \wedge (y \vee \bar{z})$$

$y = 0$

$y = 1$

$$(z) \wedge (\bar{z})$$

$$(False)$$

$z = 0$

$z = 1$

$$(False)$$

$$(False)$$

Heuristically:

We chose subproblem that includes the smallest clause and expanded on the variable in the smallest clause

Backtracking on SAT Instances

$$(w \vee x \vee y \vee z) \wedge (w \vee \bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z \vee \bar{w}) \wedge (\bar{w} \vee \bar{z})$$

$w = 0$

$w = 1$

$$(x \vee y \vee z) \wedge (\bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z})$$

$$(x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z) \wedge (\bar{z})$$

$x = 0$

$x = 1$

$z = 0$

$z = 1$

$$(y \vee z) \wedge (\bar{y}) \wedge (y \vee \bar{z})$$

$$(False) \wedge (y \vee \bar{z})$$

$$(x \vee \bar{y}) \wedge (False)$$

$$(x \vee \bar{y}) \wedge (y) \wedge (False)$$

$y = 0$

$y = 1$

$$(z) \wedge (\bar{z})$$

$$(False)$$

$z = 0$

$z = 1$

$$(False)$$

$$(False)$$

Heuristically:

We chose subproblem that includes the smallest clause and expanded on the variable in the smallest clause

Backtracking on SAT Instances

$$(w \vee x \vee y \vee z) \wedge (w \vee \bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z \vee \bar{w}) \wedge (\bar{w} \vee \bar{z})$$

$w = 0$

$w = 1$

$$(x \vee y \vee z) \wedge (\bar{x}) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z})$$

$$(x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z) \wedge (\bar{z})$$

$x = 0$

$x = 1$

$z = 0$

$z = 1$

$$(y \vee z) \wedge (\bar{y}) \wedge (y \vee \bar{z})$$

$$(False) \wedge (y \vee \bar{z})$$

$$(x \vee \bar{y}) \wedge (False)$$

$$(x \vee \bar{y}) \wedge (y) \wedge (False)$$

$y = 0$

$y = 1$

$$(z) \wedge (\bar{z})$$

$$(False)$$

$z = 0$

$z = 1$

$$(False)$$

$$(False)$$

We verified that the instance is not satisfiable, without looking at all the 2^4 assignments of the variable.

Backtracking more Generally

Backtracking requires a “test” algorithm that runs fast and state one of 3 outcomes:

1. Failure: The subproblem has no solution
2. Success: The solution is found!
3. Uncertain

Example heuristic:

Choose subproblem that includes the smallest clause.

Expand on one of the variables in the smallest clause

Start with problem P_0 and let $S = \{P_0\}$, the set of active subproblems

Repeat while $S \neq \emptyset$:

Choose a subproblem $P \in S$ and remove it from S

Expand the problem into smaller subproblems P_1, P_2, \dots, P_k

For each P_i :

If $test(P_i) = \mathbf{success}$: halt and **return** the solution of P_i

If $test(P_i) = \mathbf{uncertain}$: Add P_i to S

Return that no solution exists. // Means no subproblem was successful

Intelligent Search in Optimization Problems

Backtracking works quite well for decision problems, like SAT. How about optimization problems? Say, if you want to minimize or maximize solution value.

Branch-and-bound, uses backtracking on optimization problems.

We need to eliminate a branch of subproblems quickly. How?

- **Idea 0:** Keep track of the value of the best solution so far
- ~~**Idea 1:** Exactly compute the optimal solution of the branch and see if it's worse?~~
- **Idea 2:** Give an approximate bound on the optimal solution of the branch,
 - Discard the branch if the approximate bound rules out optimality
 - Keep expanding the problem if the bound is uncertain.

Branch and Bound

Rule out optimality for minimization problem:

- We need a function $lowerbound(P_i)$ that looks at a partial solution P_i and quickly gives us a lower bound on the value of any possible completion of P_i .
- If $lowerbound(P_i) > best\text{-}so\text{-}far$, the entire branch under P_i can be eliminated.

opt value
in P_i →

Branch and Bound

Rule out optimality for minimization problem:

- We need a function *lowerbound*(P_i) that looks at a partial solution P_i and quickly gives us a lower bound on the value of any possible completion of P_i .
- If *lowerbound*(P_i) > *best-so-far*, the entire branch under P_i can be eliminated.

Branch-an-bound for a minimization problem

Start with problem P_0 and let $S = \{P_0\}$, the set of active subproblems

best-so-far = ∞

Repeat while $S \neq \emptyset$:

Choose a subproblem (partial solution) $P \in S$ and remove it from S

Expand the problem into smaller subproblems P_1, P_2, \dots, P_k

For each P_i :

If P_i is a complete solution, update *best-so-far* if it's the best value so far

Else if *lowerbound*(P_i) < *best-so-far*, add P_i to S .

Return *best-so-far*

Branch and Bound for TSP

Recall: TSP(graph $G = ([n], E)$ and edge lengths $d_e > 0$ for all $e \in E$, returns a tour (a cycle passing through all nodes) of the smallest length.

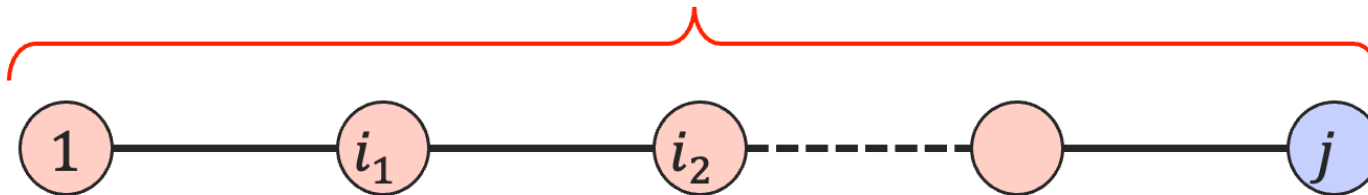
Partial Solutions: Same subproblems as in our DP algorithm for TSP.

Lecture 13

Think of subproblems as partial tour!

→ It starts from city 1, ends in city j , and passing through all cities in a set S (which includes 1 and j).

Set S of cities (including 1 and j)



Subproblems: For all $j \leq n$ and $S \subseteq \{1, \dots, n\}$, s.t. S includes 1 and j .

$T[S, j]$ = length of the shortest path visiting all cities in S exactly once, starting from 1 and ending at j .

Lower-Bounding Value of Partial TSP

Subproblems: For all $j \leq n$ and $S \subseteq \{1, \dots, n\}$, s.t. S includes 1 and j .

$T[S, j]$ = the shortest path visiting **all cities in S exactly once**, starting from 1 and **ending at j** .

lowerbound($T[S, j]$) needs to lower bound the completion of this tour.

