19-0: Dealing w/ NP-Completeness

- You have a problem that you need to solve, which is NP-Complete. What can you do?
  - Problem that is NP-Complete in general may not be for a specific instance
  - For small enough problems, exponential running time is OK
  - Find an approximate solution

19-1: Dealing w/ NP-Completeness

- Problem that is NP-Complete in general may not be for a specific instance
  - 3-SAT is NP-Complete
  - 2-SAT (special case of 3-SAT) is not
  - Typically, large clauses are “easier” to solve

19-2: Dealing w/ NP-Completeness

- Problem that is NP-Complete in general may not be for a specific instance
  - Independent set:
    - Undirected Graph $G$, integer $k \geq 2$
    - Is there a subset of the vertices in $G$: $C \subseteq V$ such that no two elements in $C$ are connected by a direct edge, and $|C| \geq k$?

19-3: Independent Set

19-4: Independent Set
19-5: **Independent Set**

- Independent Set is NP Complete
  - Reduction from 3-SAT
  - Special case of Satisfiability, each clause has exactly 3 members
  - We can transform any Satisfiability problem into a 3-SAT problem

19-6: **Independent Set**

- Independent Set is NP Complete
  - Given any instance of 3-SAT, we will create a graph $G$ and limit $k$, such that there is an independent set of vertices in $G$ with size $k$ if and only if the instance of 3-SAT is satisfiable

19-7: **Independent Set**

- For each clause in the formula, we will create a triangle, with each vertex representing a variable in the clause

\[
(x_1 \lor x_2 \lor \overline{x}_3) \quad (x_1 \lor \overline{x}_2 \lor \overline{x}_3) \quad (\overline{x}_1 \lor \overline{x}_2 \lor x_3)
\]

19-8: **Independent Set**

- Next, add an edge between every vertex and its negation:
19-9: Independent Set

- Finally, we set the limit \( k \) equal to the number of clauses in the formula:

\[
(x_1 \lor x_2 \lor \overline{x}_3) \quad (x_1 \lor \overline{x}_2 \lor x_3) \quad (\overline{x}_1 \lor \overline{x}_2 \lor x_3)
\]

- \( k = 3 \)

19-10: Independent Set

- If there is an independent set of size 3, then one node from each triangle must be picked.
- Cannot have both a node and its negation in the independent set (link between each node and its negation)
- Picking a node to be in the independent set = setting that variable to true
- There is an independent set of size 3, if we can pick one variable from each clause and mark it as “true”

19-11: Independent Set

- Independent Set is NP-Complete, in general
- Independent Set is in \( P \), if the graph is a tree
  - How can we find a maximal Independent Set in a tree?

19-12: Node Cover

- Node Cover:
  - Undirected Graph \( G \), integer \( b \geq 2 \)
• Is there a subset of the vertices in $G$: $C \subseteq V$ such that all edges in $G$ connect to at least one element of $C$, and $|C| \leq b$?

19-13: **Node Cover**

19-14: **Node Cover**

19-15: **Node Cover**

• Node Cover is NP-Complete, in general
• Node Cover is in $P$, *if the graph is a tree*
  • How could we find a minimal node cover, in a tree?

19-16: **Dealing w/ NP-Completeness**

• For small enough problems, exponential running time is OK
• TSP for 10 cities is very doable
• Only 10! = 3628800 permutations to check
• Run a straightforward backtracking algorithm (allows you to trim the search space a little bit)

19-17: **Dealing w/ NP-Completeness**

\[ X = x_1, x_2, \ldots, x_n, C = \text{set of clauses}, k = 1 \text{ initially} \]

\[
\text{SolveSAT}(X, C, k) \\
\quad \text{if } (k > n) \text{ return true} \\
\quad \text{set } x_k = \text{true} \\
\quad \text{if } ((\text{truth assignment of } x_1 \ldots x_k \text{ does not} \\
\quad \text{cause a contradiction}) \&\& (\text{SolveSAT}(X, C, k + 1))) \\
\quad \text{return true} \\
\quad \text{set } x_k = \text{false} \\
\quad \text{if } ((\text{truth assignment of } x_1 \ldots x_k \text{ does not} \\
\quad \text{cause a contradiction}) \&\& (\text{SolveSAT}(X,C,k+1))) \\
\quad \text{return true} \\
\quad \text{return false}
\]

19-18: **Dealing w/ NP-Completeness**

\[ X = x_1, x_2, x_3, \ldots, x_n, \text{ Weights of each object} \]
\[ L = \text{limit} \]
\[ k = 1 \text{ on first call to solveKNAPSACK} \]

\[
\text{SolveKNAPSACK}(X, L, k) \\
\quad \text{if } (L = 0) \\
\quad \text{return true} \\
\quad \text{if } (k > n) \\
\quad \text{return false} \\
\quad \text{if } (\text{SolveKNAPSACK}(X, L - x_k, k + 1)) \\
\quad \text{return true} \\
\quad \text{return SolveKNAPSACK}(X, L, k + 1)
\]

19-19: **Optimization Problems**

• In an optimization problem, we can often find a solution quickly
  • The difficulty is in finding an optimal solution
• Consider strategies that find sub-optimal solutions in polynomial time

19-20: **Optimization Problems**

• Depth-First Branch and Bound
  • Technically, still requires exponential time
  • Use Heuristics to speed up the search
  • Can stop at any time, use the best solution seen so far

19-21: **Optimization Problems**
\[ P = \text{partial solution} \]
maxcost = best cost so far (initially infinity)

\[
\text{BranchAndBound}(P, \text{maxcost})
\]
if ((\(P\) is a complete solution) \\
&& ((cost of \(P\) < maxcost))
\)
    save \(P\) in aux. data structure
    return (cost of \(P\))
\]
if ((cost of \(P\) + (estimated cost to complete \(P\))) \geq \text{maxcost}
\)
return maxcost

for each solution \(S\) to the next subproblem
    Modify \(P\) to \(P'\) by applying \(S\)
    maxcost = BranchAndBound(\(P'\), maxcost)
    Undo \(S\) to get \(P\) again

19-22: Optimization Problems

- Heuristics
- The “estimated cost to complete \(P\)” is a Heuristic
- For Branch and Bound to find the optimal value, Heuristic must never overestimate
  - Can underestimate as much as you want
- Heuristic is always 0, never prune
- Heuristic is huge, we prune everything (overprune)

19-23: Optimization Problems

- Example TSP:
  - Start with initial node
  - Pick an unvisited city to visit next
  - Calculate the estimated cost for the entire cycle, based on the path we’ve already started
  - If estimated cost < best solution so far
    - Continue creating the cycle
  - Undo last choice, pick new city, repeat until you’ve tried all of them

19-24: TSP Heuristic

- Once the first few vertices have been chosen, what is a good guess for the rest of the problem?
  - We can use the cost of a Minimum Spanning Tree
    - Guaranteed to be no larger than the cost to finish the cycle
    - Reasonable estimate of what we need
    - Cheap (well, relatively) to compute

19-25: TSP Heuristic
19-26: TSP Heuristic

19-27: TSP Heuristic
19-28: **Local Improvement**

- Pick any solution (doesn’t have to be good)
  - TSP: Pick an arbitrary permutation of the vertices
- Make local changes
  - swap two cities in the permutation
  - move some city to a different part of the permutation
  - etc.

19-29: **Local Improvement**

- Hill Climbing:
  - Examine all local changes that can be made
  - Pick change with greatest improvement
  - Repeat until all changes make things worse
- Problems with Hill Climbing?

19-30: **Local Improvement**

- Hill Climbing:
  - Examine all local changes that can be made
  - Pick change with greatest improvement
  - Repeat until all changes make things worse
- Problems with Hill Climbing
  - Could get stuck in a local minimum (or maximum)
  - All local changes may make things worse, several changes strung together may make things better

19-31: **Local Improvement**

- Hill Climbing:
• Never make a choice that results in a worse solution
• Can get stuck in local minimum/maximum
• What to do?
  • Occasionally make a “non-optimal” improvement
  • That is, occasionally take a step that makes solution worse, with hope of making the solution better in the long run

19-32: **Local Improvement**

• Annealing (metallurgy)
  • Heat metal sufficiently to change internal structure
  • Cool metal slowly, maintain this new structure
• Simulated Annealing
  • “Temperature” is probability of taking wrong step
  • Initially, wrong steps are very likely
  • As time goes on, take wrong steps less often (transforms into hill climbing)

19-33: **Local Improvement**

• Simulated Annealing:

\[
S' = \text{initial solution} \\
T = \text{initial temperature}
\]

while \((T > 0)\)
  Take a *random* step to get from \(S\) to \(S'\)
  \(\Delta = \text{cost}(S') - \text{cost}(S)\)
  if \((\Delta \leq 0)\)
    \[S = S'\]
  else
    \[S = S' \text{ with probability } e^{-\Delta/T}\]

19-34: **Approximation Ratio**

• An algorithm has an *approximation ratio* of \(\rho(b)\) if, for any input size \(n\), the cost of the solution produced by the algorithm is within a factor of \(\rho(n)\) of an optimal solution

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n)
\]

• For a maximization problem, \(0 < C \leq C^*\)
• For a minimization problem, \(0 < C^* \leq C\)

19-35: **Approximation Ratio**

• Some problems have a polynomial solution, with \(\rho(n) = c\) for a small constant \(c\).
• For other problems, best-known polynomial solutions have an approximation ration that is a function of \(n\)
- Bigger problems ⇒ worse approximation ratios

19-36: **Approximation Scheme**

- Some approximation algorithm takes as input both the problem, and a value $\epsilon > 0$
  - For any fixed $\epsilon$, $(1 + \epsilon)$-approximation algorithm
  - $\rho(n) = 1 + \epsilon$
- Running time increases as $\epsilon$ decreases

19-37: **Vertex Cover**

- Problem: Given an undirected graph $G = (V, E)$, find a $V' \subseteq V$ such that
  - For each edge $(u, v) \in E$, $u \in V'$ or $v \in V'$
  - $|V'|$ is as small as possible
- Vertex Cover is NP-Complete, optimal solutions will require exponential time
- Can you come up with an algorithm that will give a (possibly non-optimal) solution for the problem?

19-38: **Vertex Cover**

Approx-Vertex-Cover(V,E)

$C \leftarrow \{\}$
$E' \leftarrow E$

while $E' \neq \{\}$
  let $(u, v)$ be any edge in $E'$
  $C \leftarrow C \cup \{u, v\}$
  remove all edges from $E'$ that contain $u$ or $v$

19-39: **Vertex Cover**

19-40: **Vertex Cover**
19-41: Vertex Cover

\[ C = \{a, c\} \]

19-42: Vertex Cover

\[ C = \{a, c, d, e\} \]

19-43: Vertex Cover

\[ C = \{a, c, d, e, f, h\} \]
\[ C = \{a, c, d, e, f, h\} \]

19-44: Vertex Cover

19-45: Vertex Cover

Optimal

\[ C = \{a, d, e, h\} \]

19-46: Vertex Cover

- Approx. Vertex-Cover is a polynomial-time 2-approximation algorithm
\[ \rho(n) = 2 \]

- Let \( C \) be the set of vertices found by approx. algorithm
- Let \( C^* \) be the optimal set of vertices
- \(|C| \leq 2 \cdot |C^*|\)

19-47: 
**Vertex Cover**

- Let \( A \) be the set of edges selected by Approx. Vertex Cover
- Optimal vertex cover must pick at least one of the vertices for each edge in \( A \)
  - \(|C^*| \geq |A|\)
- Approx. vertex cover picked both vertices for each edge in \( A \):
  - \(|C| = 2 \cdot |A|\)
- Putting pieces together: \(|C^*| \geq |A| = |C|/2, |C| \leq 2 \cdot |C^*|\)

19-48: 
**TSP**

- Travelling Salesman problem
  - Complete, undirected graph \( G = (V, E) \)
  - Cost for each edge
  - Find a cycle that includes vertices, that minimizes total cost

19-49: 
**TSP w/ triangle inequality**

- TSP on plane
  - Each node has an x,y location
  - Cost between nodes is the distance between nodes
  - Slightly more general: TSP with triangle inequality
    - For any three vertices \( v_1, v_2, v_3 \in V, c(v_1, v_2) + c(v_2, v_3) \geq c(v_1, v_3)\)

19-50: 
**Approximate TSP**

\[ \text{Approx-TSP}(V, E, c) \]

- select any vertex \( r \in V \) as root vertex
- Compute MST \( T \) of graph from root \( r \) using Prim
- \( L \leftarrow \text{list of vertices visited in preorder tree walk of } T \)
- return \( L \)
19-51: Approximate TSP

Edges between all pairs of vertices
\[ \text{cost} = \text{distance between vertices} \]

19-52: Approximate TSP

Start with vertex a
create MST

19-53: Approximate TSP

Preoder Traversal of MST
\[ a,e,g,f,b,d,c \]

19-54: Approximate TSP
Traversal => Tour
a, e, g, f, b, d, c

19-55: Approximate TSP

Best TSP tour
a, e, g, f, d, c, b

19-56: Approximate TSP

- Approximate-TSP finds a tour whose cost is at most twice the cost of the optimal TSP
- $\rho(n) \leq 2$
- Why?

19-57: Approximate TSP

- Cost of TSP Tour $\geq$ cost of MST
- Consider a “full walk” of MST (revisit vertices)

19-58: Approximate TSP
"full walk" of MST
a, e, g, e, f, e, a, b, d, b, c, b, a

19-59: **Approximate TSP**
- Cost of “full walk” = 2 * cost MST
- Since we are following each edge twice
- Not a valid tour
- Repeated vertices
- Remove repeated vertices, get preorder walk
  - Cost of preorder walk ≤ cost of full walk – triangle inequality

19-60: **Approximate TSP**
- Cost of approximate TSP tour ≤ cost of full walk
- Cost of full walk ≤ 2 * cost of MST
- Cost of MST ≤ cost of optimal TSP tour

Cost of approximate TSP tour ≤ 2 * cost of optimal tour

19-61: **General TSP**
- Alas, our algorithm does not generalize to all TSP
  - Relied on the triangle inequality
- No good approximate tours can be found in polynomial time for TSP, unless NP = P

19-62: **Subset-Sum Problem**
- Subset-Sum Decision Problem
- Given:
  - A set \( S = \{x_1, x_2, x_3, \ldots, x_n\} \) of positive integers
  - A target \( t \)
- Is there a subset of \( S \) that sums exactly to \( t \)?
19-63: **Subset-Sum Problem**

- Subset-Sum Optimization Problem
- Given:
  - A set $S = \{x_1, x_2, x_3, \ldots, x_n\}$ of positive integers
  - A target $t$
- Find a subset of $S$ with the largest possible sum less than or equal to $t$

19-64: **Subset-Sum Problem**

Exact-Subset-Sum($S, t$)

$\begin{align*}
n &\leftarrow |S| \\
L &\leftarrow \{0\} \\
\text{for } i &\leftarrow 1 \text{ to } n \\
L &\leftarrow \text{MergeLists}(L, L + L[i]) \\
\text{Remove all elements larger than } t &\text{ from } L \\
\text{return largest element in } L
\end{align*}$

- $L + L[i]$ means add $L[i]$ to each element in $L$
- MergeLists: Merge two sorted lists, removing duplicates

19-65: **Subset-Sum Problem**

$S = \{1, 3, 5\}$

- $L = \{0\}$
- $L = \{0, 1\}$
- $L = \{0, 1, 3, 4\}$
- $L = \{0, 1, 3, 4, 5, 6, 8, 9\}$

19-66: **Subset-Sum Problem**

$S = \{1, 2, 3\}$

- $L = \{0\}$
- $L = \{0, 1\}$
- $L = \{0, 1, 2, 3\}$
- $L = \{0, 1, 2, 3, 4, 5, 6\}$

19-67: **Subset-Sum Problem**

- What is the worst-case running time?

19-68: **Subset-Sum Problem**

- What is the worst-case running time?
  - List $L$ could be as large as $2^n$
Running time is $O(2^n)$
- (Polynomial if sum of all elements in $L$ is bound by a polynomial in $|S|$)

19-69: **Subset-Sum Problem**
- Algorithm is exponential because $L$ can grow exponentially large
- So, if we wanted an approximation in polynomial time, what could we do?

19-70: **Subset-Sum Problem**
- Algorithm is exponential because $L$ can grow exponentially large
- So, if we wanted an approximation in polynomial time, what could we do?
  - Prune $L$ to prevent it from getting too large
  - Removing the wrong element could prevent us from finding an optimal solution
  - How can we prune $L$ to minimize / bound the error?

19-71: **Subset-Sum Problem**
- Basic idea:
  - After creating the list $L$, “trim” it by removing elements
  - If we have two elements that are close to each other, we remove the larger of them
  - Sum can be off by the difference of the elements

19-72: **Subset-Sum Problem**
- Function TRIM, takes as input a list and a $\delta$, and trims all elements that are within $\delta \%$ of the previous element in the list:

\[
\text{TRIM}(L, \delta) \\
m \leftarrow |L| \\
L' \leftarrow L[1] \\
\text{last} \leftarrow L[1] \\
\text{for } i \leftarrow 2 \text{ to } m \\
\quad \text{if } L[i] > \text{last} \times (1 + \delta) \\
\quad \text{append } L[i] \text{ to } L' \\
\text{return } L'
\]

19-73: **Subset-Sum Problem**

Approx-Subset-Sum($S, t, \epsilon$)
\[
n \leftarrow |L| \\
L \leftarrow \{0\} \\
\text{for } i \leftarrow 1 \text{ to } n \\
\quad L \leftarrow \text{MergeLists}(L, L + L[i]) \\
\quad L \leftarrow \text{TRIM}(L, \epsilon/2n) \\
\quad \text{remove elements greater than } t \text{ from } L \\
\text{return largest element in } L
• Returns an element within \((1 + \epsilon)\) of optimal

19-74: Subset-Sum Problem

\[ S = \{104, 102, 201, 101\}, t = 308, \epsilon = .4, \delta = 0.05 \]

• \(L = \{0\}\)
• \(L = \{0, 104\}\) (no trimming)
• \(L = \{0, 102, 104, 206\}\)
• \(104 < 102 + 1.05\)
• \(L = \{0, 102, 206\}\)
• \(L = \{0, 102, 201, 206, 303, 407\}\)
• \(206 < 201 + 1.05\)
• \(407 > t\)
• \(L = \{0, 102, 201, 303\}\)

19-75: Subset-Sum Problem

\[ S = \{104, 102, 201, 101\}, t = 308, \epsilon = .4, \delta = 0.05 \]

• \(L = \{0, 102, 201, 303\}\)
• \(L = \{0, 101, 102, 201, 203, 302, 303, 404\}\)
• \(102 < 101 + 1.05\)
• \(203 < 201 + 1.05\)
• \(303 < 302 + 1.05\)
• \(404 > \epsilon\)
• \(L = \{0, 101, 201, 302\}\)

• Result: 302
• Optimal: 307 \((104 + 102 + 101)\)
• Within 0.40 of optimal

19-76: Subset-Sum Problem

• Approx-Subset-Sum \((S, t, \epsilon)\)
  • Always returns a result within \((1 + \epsilon)\) of the true optimal
  • Runs in time polynomial in length of input and \(1/\epsilon\)

19-77: Subset-Sum Problem

• Runs in time polynomial in length of input and \(1/\epsilon\):
  • First, we’ll find a bound on how long each list \(L_i\) can be
  • After each trimming, consider successive elements \(z, z'\)
  • \(z'/z > 1 + \epsilon/2n\)
  • Largest that \(L_i\) could be:
    • \(0, 1, \epsilon/2n, 2\epsilon/2n, 3\epsilon/2n \ldots\)
    • size of \(L_i < \log_{1+\epsilon/2n} t + 2\)
• size of $L_i < \log_{1+\epsilon/2n} t$

\[
\log_{1+\epsilon/2n} t = \frac{\ln t}{\ln(1 + \epsilon/2n)} + 2 \\
\leq \frac{2n(1 + \epsilon/2n) \ln t}{\epsilon} + 2 \\
\leq \frac{4n \ln t}{\epsilon} + 2
\]

• Bound is clearly polynomial in size of input and $\frac{1}{\epsilon}$

\[
\frac{x}{1+x} \leq \ln(1+x) \leq x, \ 0 < \epsilon < 1
\]

19-79: **Subset-Sum Problem**

• Always returns a result within $(1+\epsilon)$ of the true optimal