Summary of the course so far:

I. Algorithmic Paradigms
- reductions
- divide and conquer
- greedy algorithms
- dynamic programming

II. Graph Algorithms

You have seen many *efficient* algorithms = run time is polynomial in input size, e.g., $O(n)$, $O(n \log n)$, $O(n^3)$, etc.

But there are many practical problems where no efficient algorithm is known e.g., 0-1 knapsack, Travelling Salesman, shortest path in a graph with negative weights

Options for these "hard" problems:

- heuristics — run quickly but no guarantee on run time or quality of solution
- approximation algorithms — guarantee quality of solution
- exact solutions that take exponential time — today's topic

We sometimes need exact solutions, e.g., to test the quality of heuristics

Backtracking

- a systematic way to try all possible solutions
- like searching in an **implicit graph** of partial solutions
- used for **decision** problems (we'll deal with optimization problems later)

Example. Subset Sum (a decision version of Knapsack with value = weight)
Given elements 1, 2, . . . n, with weights $w_1, w_2, \ldots w_n$ and target weight W, is there a subset $S \subseteq \{1, 2, \ldots n\}$ such that $$\sum_{i \in S} w_i = W$$

example

weights = {2, 2, 3, 5, 7}, W = 13

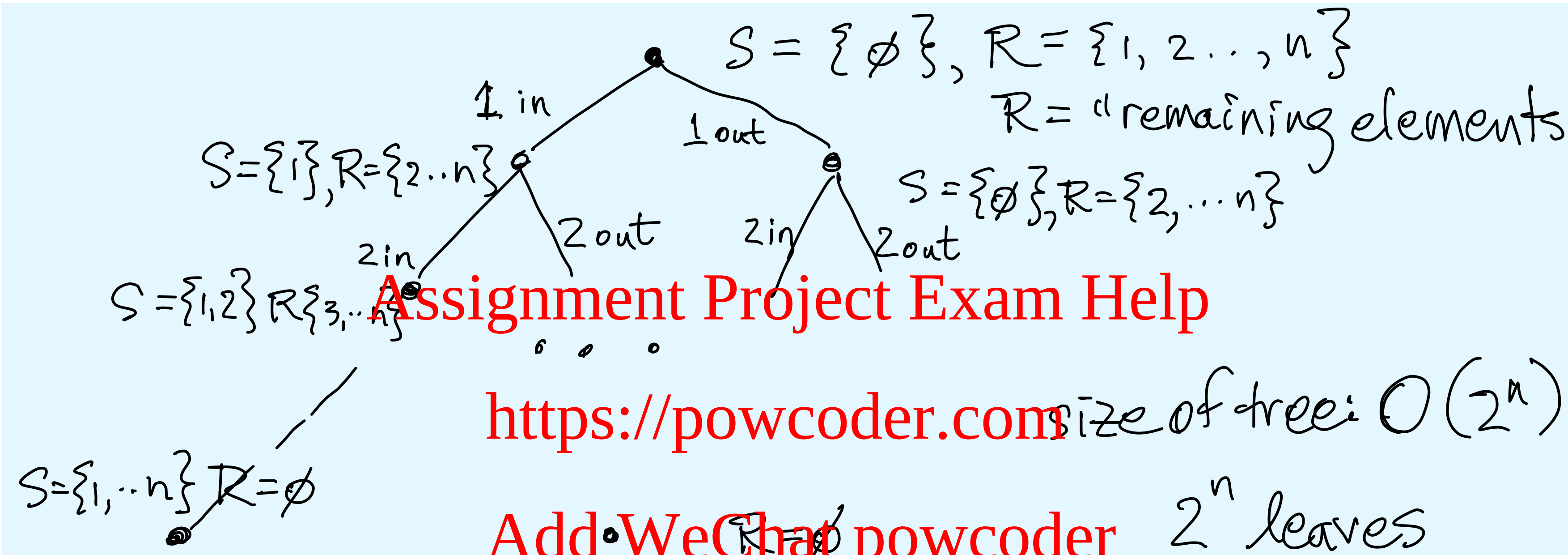
Is there a solution? No.

Fact. This problem is NP-complete (proof later). No one knows a polynomial time algorithm.

The best we can do is explore all subsets.

How many subsets are there? 2^n

Backtracking to explore all subsets of {1, 2, . . . , n}



Each node corresponds to a ***configuration***

C = (S,R) where S ⊆ {1, 2, . . . , i-1}, R = {i, . . . , n}

and has two children — put i **in** or **out** of S.

Next: how to explore a backtracking tree in general.

General Backtracking Algorithm

A = set of active configurations. Initially A has just one configuration.

e.g., for subsets of {1, 2, . . . n} the initial configuration is S = $\varnothing$, R = {1, . . . , n}.

while A $\neq \varnothing$
  C := remove a configuration from A
  # explore configuration C
  if C solves the problem then DONE
  if C is a dead-end then discard it
  else expand C to child configurations $C_1, \ldots, C_t$ by making additional choices,
    and add each C_i to A
end

Options:
- store A as a stack. DFS of configuration space. Size of A = height of tree.
- store A as a queue. BFS of configuration space. Size of A = width of tree.

To reduce space, store A as a stack.
e.g., for Subset Sum, width is 2^n, height is n.

Note: we might also explore the "most promising" configuration first. Then store A as a priority queue.

Applying the Backtracking Algorithm to Subset Sum

```
while A ≠ ∅
   C := remove a configuration from A
   # explore configuration C
   if C solves the problem then DONE
   if C is a dead-end then discard it
   else expand C to child configurations C_1, . . . , C_t by making additional choices,
      and add each C_i to A
end
```


How to explore configuration C = (S,R) for Subset Sum
(recall S = set so far, R = remaining elements)

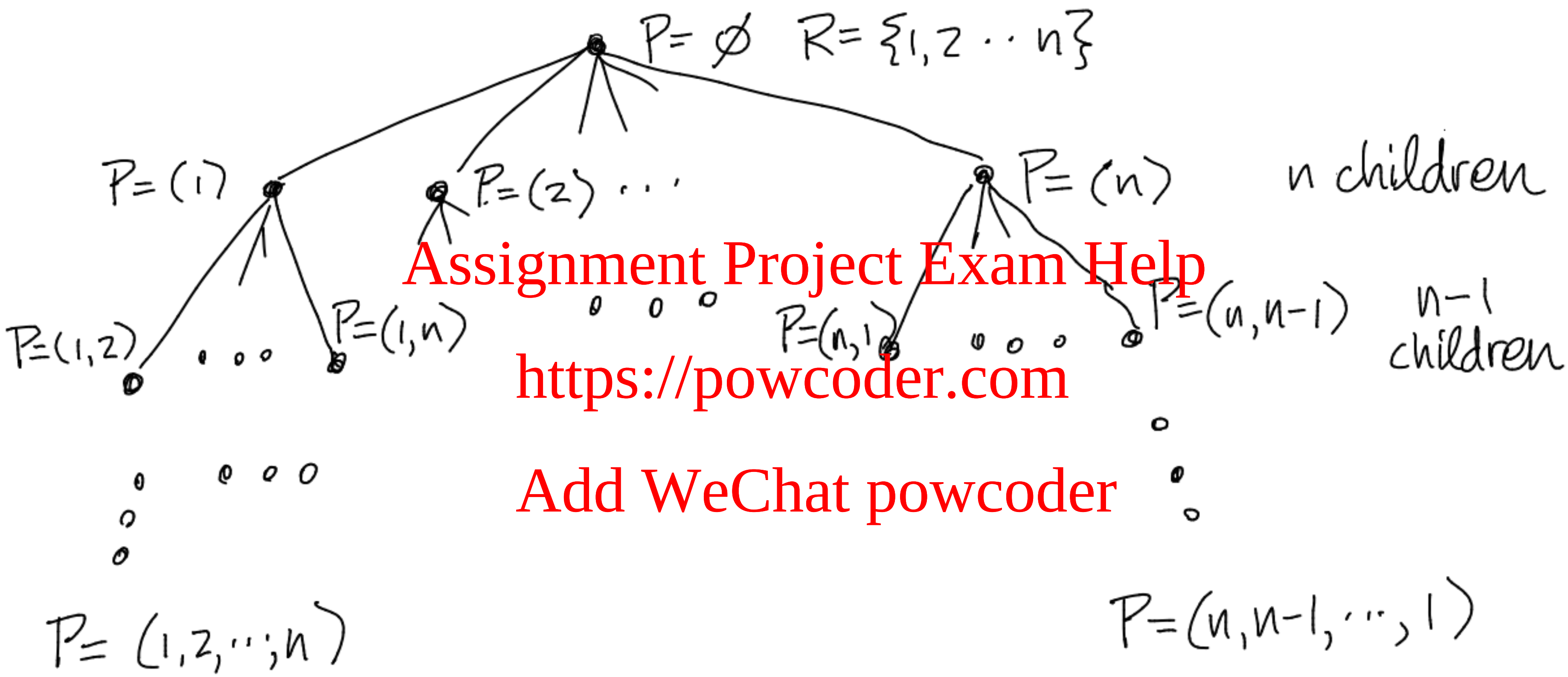
Keep: $w = \sum_{i \in S} w_i$ $\quad r = \sum_{i \in R} w_i$

Then:
- if w = W — SUCCESS (solved problem)
- if w > W — dead end (don't expand this configuration)
- if r+w < W — dead end

Run time: $O(2^n)$

There is also a dynamic programming algorithm for Subset Sum (like for knapsack) with runtime O(nW). Which is better? It depends! If W is small, O(nW) is better. If W has n bits then backtracking is better.

Backtracking to explore all permutations of {1, 2, . . . , n}



There are n! leaves.

configuration C = (P, R), P = permutation so far
R = remaining elements (not drawn above)

Summary of Lecture 17, Part 1

- backtracking to try all possibilites
- examples: explore all subsets (Subset Sum), explore all permutations

What you should know from Lecture 17, Part 1:

- how backtracking works

Next:

- branch-and-bound for optimization problems

Optimization versus Decision problems

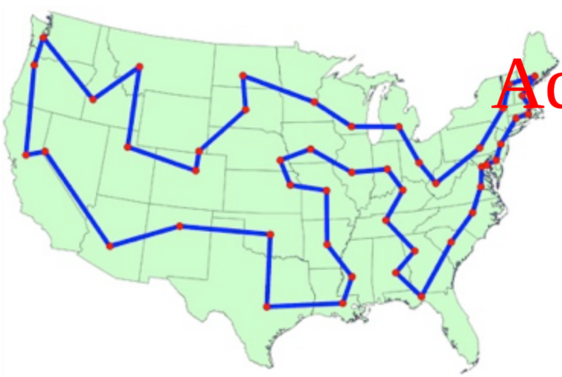
Sometimes we want **a** solution and sometimes we want **the best** solution according to some objective function.

Example 1. Subset Sum versus 0-1 Knapsack.

Example 2. Hamiltonian cycle versus Travelling Salesman Problem (TSP).

Hamiltonian cycle: Given a graph, find a Hamiltonian cycle — a cycle that goes through every vertex exactly once.

Travelling Salesman: Given a graph with weights on the edges, find a Hamiltonian cycle such that the sum of its edge weights is minimum.



A min TSP of US state capitals (the Canadian version is boring).

There is a whole book about TSP, and there are contests to solve bigger and bigger instances. Waterloo has a world's expert:
http://www.math.uwaterloo.ca/tsp/index.html

24,978 cities in Sweden.

To solve Hamiltonian cycle, we could use backtracking to try all n! vertex orderings.

Exercise: go through the problems we've covered in the course — which were decision problems? optimization problems? neither? (e.g. sorting)

Branch and Bound

- exhaustive search for **optimization** problems.
- rather than DFS order, explore the “most promising” configuration first
- keep the best (minimum/maximum) found so far
- “branch” — generate children
- “bound” — compute a lower bound on the objective function for a configuration (= the best we might get from this configuration) and discard the configuration if its lower bound is greater than best so far

General Branch and Bound Algorithm

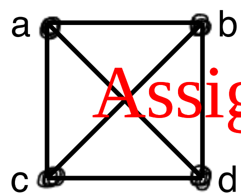
A = set of active configurations. Initially A has just one configuration.
best-cost := ∞

while $A \neq \emptyset$
  C := remove “most promising” configuration from A
  expand C to $C_1, \ldots, C_t$ by making additional choices BRANCH
  for i = 1 . . t
    if C_i solves the problem then if cost(C_i) < best-cost then update best-cost
    else if C_i is a dead-end then discard it
    else if lower-bound(C_i) < best-cost then add C_i to A BOUND
end

Branch-and-Bound for the Travelling Salesman Problem

- based on enumerating all subsets of edges (not all vertex orderings!)
- configuration $C = (N, X)$, where $N \subseteq E$ is the iNcluded edges,
  and $X \subseteq E$ is the eXcluded edges (with $N \cap X = \varnothing$).

Example.

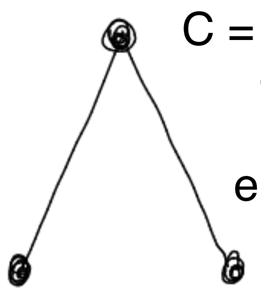


If $X = \{(a,b)\}$ then the only TSP tour is acbd
If $X = \{(a,b)\}$ and $N = \{(c,d)\}$ then there is no solution

Necessary conditions (used to detect dead ends)

- $E - X$ is connected (actually, biconnected)
- N has ≤ 2 edges incident to each vertex
- N contains no cycle (except on all the vertices)

How to branch:



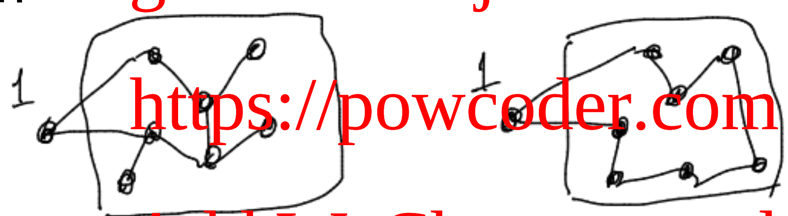
Branch-and-Bound for the Travelling Salesman Problem

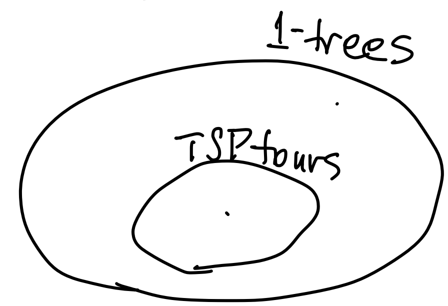
How to bound: Given a configuration (N,X) we want to **efficiently compute** a lower bound on the min cost TSP that includes N and excludes X.

A relaxed (easier) problem:

Definition. A **1-tree** is a spanning tree on vertices 2, 3, . . . n plus two edges incident to vertex 1.

examples





Claim. Any TSP tour is a 1-tree.
Thus min weight of TSP ≥ min weight of 1-tree. So this gives our lower bound.

Given configuration (N,X) we can efficiently compute the minimum weight 1-tree that includes N and excludes X:

- discard edges X
- assign (temporarily) weight 0 to edges in N
- find a Min Spanning Tree on vertices 2 . . . n
- add the two min. weight edges incident to vertex 1

then compute weight of 1-tree (add up true weights of edges in 1-tree)

Branch-and-Bound for the Travelling Salesman Problem

Plugging this into the general branch and bound algorithm:

A = set of active configurations. Initially A has just one configuration, C = (∅,∅)
min-weight := ∞
while A ≠ ∅
  C =(N,X) := remove "most promising" configuration from A
  choose e ∈ E – (N ∪ X)
  expand C to C_1, C_2 by choosing e in or e out   BRANCH
  for i = 1, 2
    if C_i solves the problem then if weight(C_i) < min-weight then update min-weight
    else if C_i is a dead-end then discard it
    else if min-weight-1-tree(C_i) < min-weight then add C_i to A   BOUND
end

Enhancements:

- "most promising" = min weight 1-tree
- branch wisely by choosing e depending on the min weight 1-tree

These, plus further enhancements, lead to competitive TSP algorithms.

Don't worry about the details of 1-trees — the point is to have some idea of the "bound" step of branch and bound

Summary of Lecture 17

- backtracking to try all possibilites
- branch-and-bound for optimization problems

What you should know from Lecture 17:

- assignment/programming may ask you to do backtracking/branch-and-bound

Next:

- NP-completeness — the “hard” problems where we should resort to exhaustive search