Exhaustive Search
Summary of the course so far:

- **Algorithmic Paradigms**
  - Reductions
  - divide and conquer
  - greedy algorithms
  - dynamic programming

- **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.
Introduction

- **Tractable** problems: problems that are solvable by polynomial-time algorithms
- **Intractable** problems or **NP-Complete** problems
  - No polynomial-time algorithm has yet been discovered to solve these problems
  - The runtime are exponential in the size of the input to the problem
  - Why does it matter?
  - Examples:
    - 0-1 knapsack, Travelling Salesman, longest weight path
    - Many intractable problems seems not to be harder than sorting or graph searching
      - Euler tour: Finding a cycle that traverses each edge of G exactly once can be done in polynomial time
      - Hamiltonian cycle: Finding a cycle that visits each vertex only once is intractable (NP-complete)
      - Shortest path can be solved in polynomial time but longest path is NP-complete
  - How to prove a problem in NP-complete? The main tool is reduction
Dealing with intractable problems

- Heuristics — run quickly but no guarantee on run time or quality of solution
- Approximation algorithms — guarantee quality of solution
  - Finding an approximate (sub-optimal) solution in polynomial time
- Exact solutions that take exponential time
  - Exhaustive search (brute force)
  - **Backtracking** and **Branch and Bound**:
    - An improvement over exhaustive search approach
    - Does not guarantee to find a polynomial time solution
  - We sometimes need exact solutions, e.g., to test the quality of heuristics
Decision problems vs. Optimization problems

- **Optimization problems**: the answer is an optimal (minimum or maximum) value
  - Shortest-path problem: given G, u, v (G a directed graph G) find the shortest path (minimum-weight path) from u to v?
  - Knapsack problem
  - Traveling Salesman Problem: Given a weighted graph, find a Hamiltonian cycle such that the sum of its edge weights is minimum.
  - Finds the best solution

- **Decision problem**: answers a given question (problem statement) as yes or no
  - Input=(G, u, v, k), where u and v are vertices in G and k is an integer, is there a path of weight k or shorter from u to v?
  - Subset-sum problem
  - Hamiltonian cycle: Given a graph, find a Hamiltonian cycle — a cycle that goes through every vertex exactly once.
  - Finds a solution
Backtracking

● **Brute-force:**
  ○ Generating all candidate solutions and then identifying the one with a desired property
  ○ works for small instances of the problem

● **Backtracking**
  ○ A more intelligent variation of brute-force technique
  ○ a **systematic way** to try and **eliminate** possibilities
  ○ Construct candidate solutions one component at a time and evaluate the partially constructed solutions
  ○ If a partial solution cannot lead to a solution, the remaining components are not generated at all
    ▪ The algorithm backtracks to replace the last component of the partial solution with its next option
  ○ Used for decision problems
Backtracking

● The idea is implemented using an **implicit** state-space tree
  ○ Node: a configuration representing a partial solution
  ○ Edge: choices in extending partial solution
  ○ Leaf: a candidate solution
● Explore the state-space tree using DFS/BFS
● Prune non-promising nodes
  ○ Stop exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node’s parent to continue the search
Backtracking: 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: Example

to explore all subsets of \{1, 2, \ldots, n\}

- Each node corresponds to a configuration
  - \( \mathbf{C} = (S,R) \) where \( S \subseteq \{1, 2, \ldots, i-1\} \), \( R = \{i, \ldots, n\} \)
  - for subsets of \{1, 2, \ldots n\} the initial configuration is \( S = \emptyset, R = \{1, \ldots, n\} \).

- Each node has two children — put \( i \) in or out of \( S \).

- How to explore a backtracking tree in general
  - DFS
  - BFS
General Backtracking Algorithm: Iterative

A = set of active configurations. Initially A has just one configuration.
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 C₁, . . . , Cₜ by making additional choices, and add each Cᵢ 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.

- How to explore configuration C=(S, R) for subset sum
  - S=set so far, R=remaining elements
  - if w = W — SUCCESS (solved problem)
  - if w > W — dead end (don’t expand this configuration)
  - if r+w < W — dead end

- Runtime: O(2ⁿ)
General Backtracking Algorithm: Recursive

Backtrack (C)
    if IsSolution(C)
        process(C)
        print("find a solution")
    if isDeadEnd(C)
        return
    else
        for each possible $C_i$ generated from C
            if $C_i$ does not lead to a dead end
                Backtrack($C_i$)
    end
Backtrack(initial configuration)
N-queens problem

- **Input:** n x n board
- **Output:** return the number of possible placement of n queens on the board so that none of them attack each other: no two of them are in the same row, column or diagonal
- If n=1 → solution is trivial
- If n=2 or n=3 → there is no solution
- Let’s solve for n=4
N-queens problem

```
1 | 2 | 3 | 4
-|-|--|--
1 |
2 |
3 |
4 |
```

- queen 1
- queen 2
- queen 3
- queen 4
N-Queens problem
Naive solution

```
def nqueen(col):
    if col >= N:
        count += 1
        for i in range(n):
            if isSafe(i, col):
                board[i][col] = 1
                queen(col+1)
                board[i][col] = 0
    return count
```
Hamiltonian Cycle Problem

- Decision problem
- State-space tree
Branch and Bound
Branch and Bound

- Exhaustive search for optimization problems.
- Applicable to **optimization problems**
- Strengthen the same idea in backtracking
  - Cut off a branch of the problem’s state-space tree as soon as we find out it cannot lead to a solution
- Keep the best (min/max) found so far
- **Branch**: generate children
- **Bound**:
  - A bound on the part that is going to complete the partial solution (similar to backtracking)
  - A bound on all extensions of the partial solution
    - compute a lower bound (upper bound) on the objective function for a configuration (= the best we might get from this configuration) and discard the configuration if its lower bound (upper bound) is greater (smaller) than the best found so far
Branch and Bound: Iterative version

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

\[
\text{best-cost := } \infty
\]

while A ≠ ∅

\[
C = \text{remove a configuration from A}
\]

expand C to child configurations \( C_1, \ldots, C_t \) by making additional choices Branch

for i=1 to t

\# explore configuration C

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: 0-1 Knapsack problem

The knapsack capacity $W$ is 10


<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
<th>value/weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>$40</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>$42</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>$25</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>$12</td>
<td>4</td>
</tr>
</tbody>
</table>
0/1 Knapsack problem

- First optimization:
  - If adding an items leads to knapsack weight to exceed the capacity do not include that item.
0/1 Knapsack problem

- First optimization:
  - If adding an item leads to knapsack weight to exceed the capacity do not include that item.

- Second optimization:
  - Calculate the maximum that you can get by including the rest of the items.
  - Upperbound = $v + (W-w) \left( \frac{v_{i+1}}{w_{i+1}} \right)$
  - Maximum value per weight among the remaining items
  - The remaining capacity of the knapsack
  - If partial_solution + upperbound < current_max then prune this branch and do not continue
Upperbound = \( v + (W-w) \left( \frac{v_{i+1}}{w_{i+1}} \right) \)

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
<th>value/weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>$40</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>$42</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>$25</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>$12</td>
<td>4</td>
</tr>
</tbody>
</table>
Traveling Salesman Problem

**Input:** A weighted complete graph with non-negative edge weight

**Output:** A tour of minimum weight that visits every vertex only once
Traveling Salesman Problem

- Naive Solution:
  - Try all possible Hamiltonian cycles in the graph.
  - How many Hamiltonian cycles are there?
- Answer: \((n - 1)! / 2\)
- Spend \(O(n)\) time processing each cycle
- Total time: \(\Theta(n!)\)
Traveling Salesman Problem

- Branch-and-Bound:
- First optimization
  - When adding a node, make sure the cost of partial solution plus the new edge is less than the minimum found so far, otherwise there is no point trying this solution.
Traveling Salesman Problem

- Backtracking:
- Branch-and-Bound:
- First optimization
  - When adding a node, make sure the cost of partial solution plus the new edge is less than the minimum found so far, otherwise there is no point trying this solution
- Second Optimization
  - $s_i =$ sum of the two lowest-weight edges incident to vertex i
  - $s =$ sum of $s_i$ over all i’s (over all nodes)
  - Lower-bound: $\lceil s/2 \rceil$
  - When adding a node, if the lower bound calculated so far is more than the minimum found so far, we do not expand that configuration
A different Branch-and-Bound for the TSP

- based on enumerating all subsets of edges (not all vertex orderings!)
- configuration $C = (N, X)$
  - $N \subseteq E$ is the included edges
  - $X \subseteq E$ is the excluded edges (with $N \cap X = \emptyset$).
- Example.
  - If $X = \{(a,b)\}$ then the only possible 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
  - $N$ has $\leq 2$ edges incident to each vertex
  - $N$ contains no cycle (except on all the vertices)
- How to branch: $C = (N, X)$
$C = (N, X)$

choose some edge in $e \in E - (N \cup X)$ to branch on

$e$ in $e$ out

$(N \cup \{e\}, X)$  $(N, X \cup \{e\})$
TSP: Branch-and-Bound

- **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, \ldots \(n\) plus two edges incident to vertex 1.

![Diagram of TSP tours and 1-trees](image)
TSP: Branch-and-Bound

- **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 in 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 weight of edges in 1-tree)
TSP: Branch-and-Bound

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

\[
\text{min-weight} := \infty
\]

while A ≠ ∅

\[
C = (N, X) = \text{remove "most promising" configuration from A}
\]

Choose \( e \in E - (N \cup X) \)

expand \( C \) to child configurations \( C_1, \ldots, C_t \) by choosing \( e \) in or \( e \) out

Branch

for \( i=1 \) to \( t \)

- if \( C_i \) solves the problem then
  - if weight(\( C_i \)) < \( \text{min-weight} \) then update \( \text{min-weight} \)
  - else if \( C_i \) is a dead-end then discard it

Bound

else if \( \text{min-weight-1-tree}(C_i) < \text{min-weight} \) then add \( C_i \) to A