Outline

Today:

- Review of P/NP and 2-approximate solution to TSP
- Dynamic Programming:
  - 0/1 Knapsack
  - Travelling Salesperson
P, NP, and NP-Complete

If there’s an algorithm to solve a problem that runs in polynomial time, the problem is said to be in the set $P$.

If the outcome of an algorithm to solve a problem can be verified in polynomial time, the problem is said to be in the set $NP$ (non-deterministic polynomial, the “non-determinism” refers to the outcome of the algorithm, not the verification).

There is a set of problems in NP for which if there’s a polynomial solution to one there will be a polynomial solution to all. The set is called $NP$-Complete.

Sugih Jamin (jamin@eecs.umich.edu)
NP-Complete, NP-Hard

If you can show that a problem is equivalent (can be reduced) to a known NP-Complete problem, you may as well not try to find an efficient solution for it (unless you’re convinced you’re a genius)

If such a polynomial solution exists, $P = NP$
It is not known whether $P \subset NP$ or $P = NP$

**NP-hard** problems are at least as hard as an NP-complete problem, but NP-complete technically refers only to decision problems, whereas NP-hard is used to refer to optimization problems
Examples of NP-Complete Problems

Hamiltonian Cycle Problem

Traveling Salesman Problem

0/1 Knapsack Problem

Graph Coloring Problem: can you color a graph using $k \geq 3$ colors such that no adjacent vertices have the same color?
A 2-Approximation for a Special Case of TSP

If distances satisfy the **triangle inequality**, i.e., for edges 
\((u, v), (v, w), (u, w)\) in \(G: C(u, v) + C(v, w) \geq C(u, w)\), and \(G\) is a **complete graph**, we can build a 2-approximate TSP

\(k\)-approximate means that the result is off by a factor of \(k\) from the optimal

2-approximate TSP:

1. construct an MST of the graph (using Prim’s for example)
2. construct an Euler-cycle of the MST
3. replace all edges \((u, v)\) and \((v, w)\) for which \(v\) is visited more than once with edge \((u, w)\)

Running time:
\(O(|E| \log |V|)\) for MST + \(O(|V|)\) for Euler = \(O(|E| \log |V|)\)
as opposed to \(O(|V| \times |V|!)\) by brute force

Sugih Jamin (jamin@eecs.umich.edu)
Euler Cycle

An Euler cycle in a graph $G$ is a path from $v$ to $v$ that contains all of the edges and all of the vertices of $G$, but no repeated edges.

The Königsberg-bridge problem: the first graph theoretic problem!

A graph $G$ has an Euler cycle iff $G$ is connected and the degree of every vertex is even.

Given an MST, to construct the Euler cycle:

- consider each edge as 2 edges in separate directions, so each vertex has an even number of edges
- do a pre-order traversal
- running time: there’s one edge per node, a tree has at most $|V| - 1$ edges, each non-leaf node is visited 3 times: $O(|V|)$
2-Approximate Proof

Let:

- $T_{\text{opt}}$ the optimal TSP minus the last edge, which must be a spanning tree
- $M$ be an MST
- $E$ the Euler tour, visit every edge in $M$ twice (once per direction)
- $T$ the non-optimal TSP obtained from $E$

Claim: $T$ is at most 2 times longer than $T_{\text{opt}}$

Proof:

- $C(T) \leq C(E)$ by triangle inequality
  
  $C(u, w) \leq C(u, v) + C(v, w)$

- $C(E) = 2C(M)$

- $C(M) \leq C(T_{\text{opt}})$ by definition of MST

So $C(T) \leq 2C(T_{\text{opt}})$
Dynamic Programming

- used when a problem can be divided into subproblems that overlap
- solve each subproblem once and store the solution in a table
- if run across the subproblem again, simply look up its solution in the table
- reconstruct the solution to the original problem from the solutions to the subproblems
- the more overlap the better, as this reduces the number of subproblems

Origin of name (Bellman 1957):
programming: planning, decision making by a tabular method
dynamic: multi-stage, time-varying process
Devide et impera

Divide-and-conquer:

- for base case(s), solve problem directly
- do recursively until base case(s) reached:
  - divide problem into 2 or more subproblems
  - solve each subproblem independently
- solutions to subproblems combined into a solution to the original problem

Works fine when subproblems are non-overlapping, otherwise overlapping subproblems must be solved more than once (as with the Fibonacci sequence)
Computing the Fibonacci Sequence (contd)

What is a Fibonacci sequence?
\[ f_0 = 0; f_1 = 1; f_n = f_{n-1} + f_{n-2}, n \geq 2 \]

Recursive implementation:
\[
\text{int rfib(int n)} \\
\{ // assume n >= 0 \\
\text{\quad return (n <= 1 ? n : rfib(n-1)+rfib(n-2));} \\
\}
\]
Running time: \( \Omega((\frac{3}{2})^n) \)
[Preiss 3.4.3]

Iterative version:
\[
\text{int ifib(int n)} \\
\{ // assume n >= 2 \\
\text{\quad int i, f[n];} \\
\text{\quad f[0] = 0; f[1] = 1;} \\
\text{\quad for (i = 2 to n) \{} \\
\text{\quad \quad f[i] = f[i-1]+f[i-2];} \\
\text{\quad \} \\
\text{\quad return f[n];} \\
\}
\]
Running time: \( \Theta(n) \)
[Preiss 14.3.2, 14.4.1]
Computing the Fibonacci Sequence (contd)

Why is the recursive version so slow?
The number of computations grows exponentially!
Each \( r_{fib}(i), i < n - 1 \) computed more than once
Tree size grows almost \( 2^n \)
Actually the number of base case computations in computing \( f_n \) is \( f_n \)
Since \( f_n > (\frac{3}{2})^{n-1} \) (see Preiss Thm. 3.9), complexity is \( \Omega((\frac{3}{2})^n) \)

Why is the iterative version so fast?
Instead of recomputing duplicated subproblems, it saves their results in an array and simply looks them up as needed

Can we design a recursive algorithm that similarly look up results of duplicated subproblem?

Sugih Jamin (jamin@eecs.umich.edu)
Memoized Fibonacci Computation

```c
int fib_memo[n] = {0, 1, -1, . . . , -1};
int
mfib(int n, *fib_memo)
{ // assume n >= 0 and left to right evaluation
  if (fib_memo[n] < 0)
    fib_memo[n] = mfib(n-2, fib_memo) + mfib(n-1, fib_memo);
  return fib_memo[n];
}
```

**Memoization** (or tabulation): use a result table with an otherwise inefficient recursive algorithm

Record in table values that have been previously computed

Memoize only the last two terms:

```c
int
rfib2(int fn2, fn1, n)
{ // assume n >= 0
  return (n <= 1 ? n:
    rfib2(fn1, fn2+fn1, --n);
}
main() { return rfib2(0, 1, n); }
```
Example of Problems Solved Using Dynamic Programming

- All-Pair Shortest Path (Floyd's algorithm)
- 0/1 Knapsack
- Travelling Salesperson
0/1 Knapsack Problem (Preiss 14.1.2, 14.2.5, handouts)

You are a poor Mayan trader, carrying wares on your back, in a tattered knapsack, back and forth between Chichen Itza and Palenque.

Each item has both a weight and a profit.

The tattered knapsack cannot carry more than its capacity or it will rip.

Choose a set of items that doesn’t rip the knapsack while maximizing your profit.
0/1 Knapsack Problem

More formally:

- \( n \) items
- \( w_i \): weight of item \( i \)
- \( p_i \): profit for item \( i \)
- \( C \): capacity of knapsack, in total weight

Let
\[
S = \{x_1, x_2, x_3, \ldots, x_n\}
\]
be the selection set:

\( x_i = 1 \) if item \( i \) is selected, 0 otherwise (hence 0/1)

Given \( \{w_1, w_2, w_3, \ldots, w_n\} \) and \( \{p_1, p_2, p_3, \ldots, p_n\} \), our objective is to maximize

\[
\sum_{i=1}^{n} p_i x_i
\]
subject to constraint

\[
\sum_{i=1}^{n} w_i x_i \leq C
\]
How to Solve It?

Brute-Force:

Branch-and-Bound: what are the bounding factor(s)?

Greedy:

Dynamic-Programming:

• can the problem be decomposed into overlapping subproblems?
• is optimal solution to problem guaranteed to consist only of optimal solutions to subproblems?
0/1 Knapsack Brute Force Solution

Exhaustively enumerate all feasible solution and pick one with the highest total profit

Running time:
0/1 Knapsack BnB Solution

Let $S_k = \{x_1, x_2, x_3, \ldots, x_k\}$, $k < n$ be a partial solution

Bounding factors:

- $S_k$ is feasible only if $\sum_{i=1}^{k} w_i x_i \leq C$

  If $S_k$ is infeasible so are all solutions that include $S_k$

- total potential profit from a solution that includes $S_k$ is $\sum_{i=1}^{k} p_i x_i + \text{MAX}(\sum_{i=k+1}^{n} p_i)$, prune branch if maximum potential profit for branch is less than current maximum profit
0/1 Knapsack Greedy Solution

Put item into knapsack one by one, no removal of item already chosen

Possible greedy criterion:

- by weight: lowest first
- by profit: highest first
- by value (profit/weight): highest first

Optimal solution not guaranteed (see Preiss Tbl 14.1)
0/1 Knapsack Dynamic Programming Solution

Let $n = 3$, $w = \{5, 10, 20\}$, and $p = \{50, 60, 140\}$, $C = 30$

Solution table $P[i][c_j]$, $c_j$ step increments to $C$:

<table>
<thead>
<tr>
<th>$p_i$</th>
<th>$w_i$</th>
<th>$i$</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>50</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>20</td>
<td>3</td>
<td>0</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can the problem be decomposed into overlapping subproblems?

Is optimal solution to problem guaranteed to consist only of optimal solutions to subproblems?
0/1 Knapsack Dynamic Programming Solution

Let \( n = 3 \), \( w = \{5, 10, 20\} \), and \( p = \{50, 60, 140\} \), \( C=30 \)

Solution table \( P[i, c_j] \):

<table>
<thead>
<tr>
<th>( p_i )</th>
<th>( w_i )</th>
<th>( i )</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>50</td>
<td>60</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>140</td>
<td>20</td>
<td>3</td>
<td>0</td>
<td>50</td>
<td>60</td>
<td>110</td>
<td>140</td>
<td>190</td>
<td>200</td>
</tr>
</tbody>
</table>

Subproblems of \( P[i, c_j] \):

- if \( w_i > c_j \), cannot add item \( i \), so \( P[i, c_j] = P[i - 1, c_j] \)
- if \( w_i \leq c_j \), consider both adding and not adding item \( i \)
  - if item \( i \) not added, \( P[i, c_j] = P[i - 1, c_j] \)
  - if item \( i \) added, \( P[i, c_j] = p_i + P[i - 1, c_j - w_i] \)
  
and take the maximal of the two options (diff from greedy)
Running Time of 0/1 Knapsack Dynamic Programming Solution

Running time to compute table: $O(nC)$

Polynomial?

Worse than BF?
Refined 0/1 Knapsack Dynamic Programming Solution

Observation: do not need to compute every entry of the table

\[ P[i, c_j] \] only needs \( P[i - 1, c_j] \) and \( P[i - 1, c_j - w_i] \), recursively

Running time: \( 1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n \)

Running time of algorithm: \( O(\text{MIN}(nC, 2^n)) \)
Dynamic Programming for TSP

Observation on TSP:

Let the optimal (shortest) TSP path from node $v_1$ be $[v_1, v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_1]$

The subpath $[v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_1]$ must be the shortest path from $v_k$ to $v_1$ that visits each of the other vertices exactly once

Similarly for $[v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_1], [v_{k+2}, \ldots, v_{k+n}, v_1], \text{ etc.},$
down to $[v_{k+n}, v_1]$

So, what's the problem?
DP for TSP

Problem is we don’t know which permutation of 
\( \{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}\} \)
gives the shortest path

We have to try them all out:

\[
\begin{align*}
&\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
&\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n-1}, v_{k+n-2}, v_{k+n}\} \\
&\vdots \\
&\{v_k, v_{k+2}, v_{k+1}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
&\{v_{k+1}, v_k, v_{k+2}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
&\{v_{k+2}, v_k, v_{k+1}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
&\vdots \\
&\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n-2}, v_{k+n}, v_{k+n-1}\} \\
&\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_{k+n-2}, v_{k+n-1}\} \\
&\vdots
\end{align*}
\]

and all other permutations thereof

Note the amount of duplicated subproblems!

Sugih Jamin (jamin@eecs.umich.edu)
Computing TSP using DP (Example)

Given the following graph:

Store the edge weights in an adjacency matrix with $\infty$ as the weight of a non-existent edge:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$b$</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>$c$</td>
<td>$\infty$</td>
<td>7</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>$d$</td>
<td>6</td>
<td>3</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

Let $C(v, w)$ the weight of edge $(v, w)$ and $D[u, v]$ the length of a path from $u$ to $v$
Computing TSP using DP (1 hop)

Consider all alternatives, not greedy:

\[
D[b, a] \quad = \quad C(b, a) = 1 \\
D[c, a] \quad = \quad C(c, a) = \infty \\
D[d, a] \quad = \quad C(d, a) = 6
\]
Computing TSP using DP (2 hops)

Consider all permutations, not greedy:

\[ D[c, b, a] = C(c, b) + D[b, a] = 7 + 1 = 8 \]
\[ D[d, b, a] = C(d, b) + D[b, a] = 3 + 1 = 4 \]

\[ D[b, c, a] = C(b, c) + D[c, a] = 6 + \infty = \infty \]
\[ D[d, c, a] = C(d, c) + D[c, a] = \infty + \infty = \infty \]

\[ D[b, d, a] = C(b, d) + D[d, a] = 4 + 6 = 10 \]
\[ D[c, d, a] = C(c, d) + D[d, a] = 8 + 6 = 14 \]
Computing TSP using DP (3 hops)

Consider all permutations when adding new vertex:

$D[d, c, b, a] = C(d, c) + D[c, b, a] = \infty + 8 = \infty$

$D[c, d, b, a] = C(c, d) + D[d, b, a] = 8 + 4 = 12$

$D[d, b, c, a] = C(d, b) + D[b, c, a] = 3 + \infty = \infty$

$D[b, d, c, a] = C(b, d) + D[d, c, a] = 4 + \infty = \infty$

$D[c, b, d, a] = C(c, b) + D[b, d, a] = 7 + 10 = 17$

$D[b, c, d, a] = C(b, c) + D[c, d, a] = 6 + 14 = 20$
Computing TSP using DP (full path)

\[
D[a, \ldots, a] = \min(C(a, c) + D[c, d, b, a],
C(a, b) + D[b, c, d, a],
C(a, c) + D[c, b, d, a])
\]
\[
= \min(9 + 12, 2 + 20, 9 + 17)
\]
\[
= 21
\]

**Extracted path (TSP):**
\[
[a, [c, \ldots, a]] = [a, [c, [d, \ldots, a]]] = [a, [c, [d, [b, a]]]]
\]
Complexity of DP for TSP

From Neapolitan & Naimipour pp. 130-131:
Time complexity: $\Theta(N^22^N)$
Space complexity: $\Theta(N2^N)$

Still exponential, not polynomial! What good is that?
Compared against the brute force algorithm with time complexity $O(N \times N!)$, this is a big improvement

Assuming 1 $\mu$sec per “basic computation” in each algorithm, to compute a TSP of 20 cities:

- brute force takes 3,857 years
- DP takes 45 seconds, using about 40 MB of memory
- for 60 cities, DP will also take many years . . . .