CS 473: Algorithms

Chandra Chekuri
chekuri@cs.illinois.edu
3228 Siebel Center
University of Illinois, Urbana-Champaign

Fall 2009
Part I

All Pairs Shortest Paths
Shortest Path Problems

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.
- Find shortest paths for all pairs of nodes.
Single-Source Shortest Path Problems

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.
Single-Source Shortest Paths

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.

**Dijkstra’s algorithm** for non-negative edge lengths. Running time:
$O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

**Bellman-Ford algorithm** for arbitrary edge lengths. Running time: $O(nm)$. 

---
All-Pairs Shortest Path Problem

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Find shortest paths for all pairs of nodes.
All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.

- Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
- Arbitrary edge lengths: $O(n^2 m)$. 
## All-Pairs Shortest Path Problem

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.

- Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.

- Arbitrary edge lengths: $O(n^2m)$. Can we do better?
Can we compute the shortest path distance from $s$ to $t$ recursively?

What are the smaller sub-problems?
Shortest Paths and Recursion

- Can we compute the shortest path distance from $s$ to $t$ recursively?
- What are the smaller sub-problems?

**Lemma**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

- $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$. 

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

- $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$. 

Chekuri

CS473ug
Can we compute the shortest path distance from $s$ to $t$ recursively?

What are the smaller sub-problems?

**Lemma**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

- $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges
Hop-based Recursion for Single-Source Shortest Paths

Single-source problem: fix source $s$.

$OPT(v,k)$: shortest path dist from $s$ to $v$ using at most $k$ edges.
Hop-based Recursion for Single-Source Shortest Paths

Single-source problem: fix source $s$.

$OPT(v, k)$: shortest path dist from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = OPT(v, n - 1)$
Single-source problem: fix source $s$.

$OPT(v, k)$: shortest path dist from $s$ to $v$ using at most $k$ edges.

Note: $dist(s, v) = OPT(v, n - 1)$

Recursion for $OPT(v, k)$:
Hop-based Recursion for Single-Source Shortest Paths

Single-source problem: fix source $s$.

$OPT(v, k)$: shortest path dist from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = OPT(v, n - 1)$

Recursion for $OPT(v, k)$:

$$OPT(v, k) = \min_{u \in V} (OPT(u, k - 1) + c(u, v)).$$

Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$
Hop-based Recursion for Single-Source Shortest Paths

Single-source problem: fix source $s$.

$OPT(v, k)$: shortest path dist from $s$ to $v$ using at most $k$ edges.

Note: $dist(s, v) = OPT(v, n - 1)$

Recursion for $OPT(v, k)$:

$$OPT(v, k) = \min_{u \in V} (OPT(u, k - 1) + c(u, v)).$$

Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$

Leads to Bellman-Ford algorithm — see text book.
Single-source problem: fix source $s$.

$OPT(v, k)$: shortest path dist from $s$ to $v$ using at most $k$ edges.

Note: $dist(s, v) = OPT(v, n - 1)$

Recursion for $OPT(v, k)$:

$$OPT(v, k) = \min_{u \in V} (OPT(u, k - 1) + c(u, v)).$$

Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$

Leads to Bellman-Ford algorithm — see text book.

$OPT(v, k)$ values are also of independent interest: shortest paths with at most $k$ hops
All-Pairs: recursion on the index of intermediate nodes

- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $\text{dist}(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an *intermediate node* is at most $k$

\[
\begin{align*}
\text{dist}(i, j, 0) & = \\
\text{dist}(i, j, 1) & = \\
\text{dist}(i, j, 2) & = \\
\text{dist}(i, j, 3) & = 
\end{align*}
\]
All-Pairs: recursion on the index of intermediate nodes

- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $dist(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an intermediate node is at most $k$

```
| $dist(i, j, 0)$ | $= 100$
| $dist(i, j, 1)$ | $=$
| $dist(i, j, 2)$ | $=$
| $dist(i, j, 3)$ | $=$
```
All-Pairs: recursion on the index of intermediate nodes

- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $\text{dist}(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an intermediate node is at most $k$

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= \\
\text{dist}(i, j, 3) &= 
\end{align*}
\]
All-Pairs: recursion on the index of intermediate nodes

- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $\text{dist}(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an intermediate node is at most $k$

![Graph with labels](image)

<table>
<thead>
<tr>
<th>$\text{dist}(i, j, k)$</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{dist}(i, j, 0)$</td>
<td>100</td>
</tr>
<tr>
<td>$\text{dist}(i, j, 1)$</td>
<td>9</td>
</tr>
<tr>
<td>$\text{dist}(i, j, 2)$</td>
<td>8</td>
</tr>
<tr>
<td>$\text{dist}(i, j, 3)$</td>
<td></td>
</tr>
</tbody>
</table>
All-Pairs: recursion on the index of intermediate nodes

- Number vertices arbitrarily as \( v_1, v_2, \ldots, v_n \)
- \( \text{dist}(i, j, k) \): shortest path distance between \( v_i \) and \( v_j \) among all paths in which the largest index of an intermediate node is at most \( k \)

\[
\begin{align*}
\text{dist}(i, j, 0) & = 100 \\
\text{dist}(i, j, 1) & = 9 \\
\text{dist}(i, j, 2) & = 8 \\
\text{dist}(i, j, 3) & = 5 
\end{align*}
\]

What is \( \text{dist}(i, j) \)?

\[ \text{dist}(i, j) = \text{dist}(i, j, n) \]
All-Pairs: recursion on the index of intermediate nodes

\[
dist(i, j, k) = \min(dist(i, j, k - 1), dist(i, k, k - 1) + dist(k, j, k - 1))
\]

Base case: \( dist(i, j, 0) = c(i, j) \) if \((i, j) \in E\), otherwise \( \infty \)

Correctness: If \( i \rightarrow j \) shortest path goes through \( k \) then \( k \) occurs only once on the path — otherwise there is a negative length cycle.
Floyd-Warshall Algorithm for All-Pairs Shortest Paths

Check if G has a negative cycle using Bellman-Ford in $O(mn)$ time
If there is a negative cycle return

for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = c(i, j)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min(dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1))

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

Running Time: $\Theta(n^3)$
Floyd-Warshall Algorithm for All-Pairs Shortest Paths

Check if G has a negative cycle using Bellman-Ford in $O(mn)$ time
If there is a negative cycle return

for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = c(i, j)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min(dist(i, j, k-1), dist(i, k, k-1)+dist(k, j, k-1))

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).
Running Time:
Floyd-Warshall Algorithm for All-Pairs Shortest Paths

Check if G has a negative cycle using Bellman-Ford in $O(mn)$ time
If there is a negative cycle return

for i=1 to n do
    for j=1 to n do
        dist(i,j,0) = c(i,j)

for k = 1 to n do
    for i=1 to n do
        for j=1 to n do
            dist(i,j,k) = min(dist(i,j,k-1), dist(i,k,k-1)+dist(k,j,k-1))

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).
Running Time: $\Theta(n^3)$
Part II

Knapsack
Input  Given a Knapsack of capacity $W$ lbs. and $n$ objects with $i$th object having weight $w_i$ and value $v_i$; assume $W$, $w_i$, $v_i$ are all positive integers

Goal  Fill the Knapsack without exceeding weight limit while maximizing value.
Knapsack Problem

Input  Given a Knapsack of capacity $W$ lbs. and $n$ objects with $i$th object having weight $w_i$ and value $v_i$; assume $W, w_i, v_i$ are all positive integers

Goal  Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.
Knapsack Example

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

If $W = 11$, the best is $\{3, 4\}$ giving value 40.
Knapsack Example

Example

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

If $W = 11$, the best is \{3, 4\} giving value 40.

Special Case

When $v_i = w_i$, the Knapsack problem is called the Subset Sum Problem.
Greedy Approach

- Pick objects with greatest value
Greedy Approach

- Pick objects with greatest value
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick \{3\}, but the optimal is \{1, 2\}
Greedy Approach

- Pick objects with greatest value
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$
- Pick objects with smallest weight

Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to $W$.
Greedy Approach

- Pick objects with greatest value
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick \{3\}, but the optimal is \{1, 2\}
- Pick objects with smallest weight
  - Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick \{1\}, but the optimal is \{2\}

Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to $W$. 
Greedy Approach

- Pick objects with greatest value
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$

- Pick objects with smallest weight
  - Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick $\{1\}$, but the optimal is $\{2\}$

- Pick objects with largest $v_i/w_i$ ratio
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$

Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to $W$. 
Greedy Approach

- Pick objects with greatest value
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick \{3\}, but the optimal is \{1, 2\}

- Pick objects with smallest weight
  - Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick \{1\}, but the optimal is \{2\}

- Pick objects with largest $v_i/w_i$ ratio
  - Let $W = 4$, $w_1 = w_2 = 2$, $w_3 = 3$, $v_1 = v_2 = 3$ and $v_3 = 5$; greedy strategy will pick \{3\}, but the optimal is \{1, 2\}
Greedy Approach

- Pick objects with greatest value
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$

- Pick objects with smallest weight
  - Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick $\{1\}$, but the optimal is $\{2\}$

- Pick objects with largest $v_i/w_i$ ratio
  - Let $W = 4$, $w_1 = w_2 = 2$, $w_3 = 3$, $v_1 = v_2 = 3$ and $v_3 = 5$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$
  - Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to $W$. 
Towards a Recursive Solution

First guess: $\text{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$.

**Observation**

*Consider an optimal solution $O$ for $1, \ldots, i$*

- **Case item** $i \notin O$ \quad $O$ is an optimal solution to items $1$ to $i - 1$
- **Case item** $i \in O$ \quad Then $O - \{i\}$ is an optimum solution for items $1$ to $n - 1$ in knapsack of capacity $W - w_i$. 

Chekuri CS473ug
First guess: $\text{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$.

**Observation**

Consider an optimal solution $O$ for $1, \ldots, i$

Case item $i \notin O$  $O$ is an optimal solution to items $1$ to $i - 1$

Case item $i \in O$  Then $O - \{i\}$ is an optimum solution for items $1$ to $n - 1$ in knapsack of capacity $W - w_i$.

*Subproblems depend also on remaining capacity.*

*Cannot write subproblem only in terms of $\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.*
Towards a Recursive Solution

First guess: $\text{Opt}(i)$ is the optimum solution value for items 1, \ldots, $i$.

**Observation**

Consider an optimal solution $\mathcal{O}$ for 1, \ldots, $i$

Case item $i \notin \mathcal{O}$ $\mathcal{O}$ is an optimal solution to items 1 to $i - 1$

Case item $i \in \mathcal{O}$ Then $\mathcal{O} \setminus \{i\}$ is an optimum solution for items 1 to $n - 1$ in knapsack of capacity $W - w_i$.

Subproblems depend also on remaining capacity. Cannot write subproblem only in terms of $\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.

$\text{Opt}(i, w)$: optimum profit for items 1 to $i$ in knapsack of size $w$

Goal: compute $\text{Opt}(n, W)$
Definition

Let $\text{Opt}(i, w)$ be the optimal way of picking items from 1 to $i$, with total weight not exceeding $w$

$$\text{Opt}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{Opt}(i - 1, w) & \text{if } w_i > w \\
\max\{\text{Opt}(i - 1, w), \text{Opt}(i - 1, w - w_i) + v_i\} & \text{otherwise}
\end{cases}$$
An Iterative Algorithm

for $w = 0$ to $W$
    $M[0,w] = 0$
for $i = 1$ to $n$
    for $w = 1$ to $W$
        if ($w_i > w$)
            $M[i,w] = M[i-1,w]$
        else
            $M[i,w] = \max(M[i-1,w], M[i-1,w-w_i] + v_i)$

Running Time

Time taken is $O(nW)$

Input has size $O(n + \log W)$; so running time not polynomial but “pseudo-polynomial”!
An Iterative Algorithm

for w = 0 to W
    M[0,w] = 0
for i = 1 to n
    for w = 1 to W
        if (w_i > w)
            M[i,w] = M[i-1,w]
        else
            M[i,w] = max(M[i-1,w], M[i-1,w-w_i] + v_i)

Running Time

- Time taken is $O(nW)$
An Iterative Algorithm

for w = 0 to W
    M[0,w] = 0
for i = 1 to n
    for w = 1 to W
        if (w_i > w)
            M[i,w] = M[i-1,w]
        else
            M[i,w] = max(M[i-1,w], M[i-1,w-w_i] + v_i)

Running Time

- Time taken is $O(nW)$
- Input has size $O(n \cdot \log W)$; so running time not polynomial but “pseudo-polynomial”!
Knapsack Algorithm and Polynomial time

Input size for Knapsack:

\[ O(\sum_{i=1}^{n} (\log w_i + \log v_i)) \]

Running time of dynamic programming algorithm:

\[ O(nW) \]

Not a polynomial time algorithm.

Example:

\[ W = 2^n \]

Input size is \( O(n^2) \), running time is \( O(n^2n) \) arithmetic/comparisons.

Algorithm is called a pseudo-polynomial time algorithm because running time is polynomial if numbers in input are of size polynomial in combinatorial size of problem.

Knapsack is NP-hard if numbers are not polynomial in \( n \).
Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i)$
Knapsack Algorithm and Polynomial time

Input size for Knapsack: \( O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i) \)

Running time of dynamic programming algorithm: \( O(nW) \)

Not a polynomial time algorithm.
Example: \( W = 2^n \) and \( w_i, v_i \in [1..2^n] \).
Input size is \( O(n^2) \), running time is \( O(n2^n) \) arithmetic/comparisons.

Algorithm is called a *pseudo-polynomial* time algorithm because running time is polynomial if *numbers* in input are of size polynomial in *combinatorial* size of problem.

Knapsack is NP-hard if numbers are not polynomial in \( n \).
\[ V_1 = 10, \; \omega_1 = 3000 \]
\[ V_2 = 5, \; \omega_2 = 1000000 \]
\[ V_n = 10, \; \omega_n = 50,000 \]
\[ W = 200,000,000 \]

\( H > 0 \) 

(3) approx in time \( \frac{1}{\varepsilon} n \log n \)
Part III

Traveling Salesman Problem
Traveling Salesman Problem

**Input**  A graph $G = (V, E)$ with non-negative edge costs/lengths. $c(e)$ for edge $e$

**Goal**  Find a tour of minimum cost that visits each node.
Traveling Salesman Problem

**Input**  A graph $G = (V, E)$ with non-negative edge costs/lengths. $c(e)$ for edge $e$

**Goal**  Find a tour of minimum cost that visits each node.

No polynomial time algorithm known. Problem is NP-Hard.
Example: optimal tour for cities of a country (which one?)
How many different tours are there?
An Exponential Time Algorithm

How many different tours are there? $n!$
An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling’s formula: $n! \sim \sqrt{n}(n/e)^n$
An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling’s formula: $n! \approx \sqrt{n} (n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$
How many different tours are there? $n!$

Stirling’s formula: $n! \approx \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?
Towards a Recursive Solution

- Order vertices as $v_1, v_2, \ldots, v_n$
- $OPT(S)$: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to $S$. Want $OPT(V)$.

Can we compute $OPT(S)$ recursively?
Towards a Recursive Solution

- Order vertices as $v_1, v_2, \ldots, v_n$
- $OPT(S)$: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to $S$. Want $OPT(V)$.

Can we compute $OPT(S)$ recursively?
- Say $v \in S$. What are the two neighbors of $v$ in optimum tour in $S$?
- If $u, w$ are neighbors of $v$ in an optimum tour of $S$ then removing $v$ gives an optimum path from $u$ to $w$ visiting all nodes in $S - \{v\}$. 

Towards a Recursive Solution

- Order vertices as \( v_1, v_2, \ldots, v_n \)
- \( OPT(S) \): optimum TSP tour for the vertices \( S \subseteq V \) in the graph restricted to \( S \). Want \( OPT(V) \).

Can we compute \( OPT(S) \) recursively?

- Say \( v \in S \). What are the two neighbors of \( v \) in optimum tour in \( S \)?
- If \( u, w \) are neighbors of \( v \) in an optimum tour of \( S \) then removing \( v \) gives an optimum path from \( u \) to \( w \) visiting all nodes in \( S - \{v\} \).

Path from \( u \) to \( w \) is not a recursive subproblem! Need to find a more general problem to allow recursion.
A More General Problem: TSP Path

**Input** A graph $G = (V, E)$ with non-negative edge costs/lengths ($c(e)$ for edge $e$) and two nodes $s, t$

**Goal** Find a path from $s$ to $t$ of minimum cost that visits each node exactly once.
A More General Problem: TSP Path

**Input** A graph \( G = (V, E) \) with non-negative edge costs/lengths \( c(e) \) for edge \( e \) and two nodes \( s, t \)

**Goal** Find a path from \( s \) to \( t \) of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?
A More General Problem: TSP Path

**Input** A graph $G = (V, E)$ with non-negative edge costs/lengths ($c(e)$ for edge $e$) and two nodes $s, t$

**Goal** Find a path from $s$ to $t$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

- $OPT(u, v, S)$: optimum TSP Path from $u$ to $v$ in the graph restricted to $S$ (here $u, v \in S$).
A More General Problem: TSP Path

**Input**
A graph $G = (V, E)$ with non-negative edge costs/lengths ($c(e)$ for edge $e$) and two nodes $s, t$

**Goal**
Find a path from $s$ to $t$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

- $OPT(u, v, S)$: optimum TSP Path from $u$ to $v$ in the graph restricted to $S$ (here $u, v \in S$).

What is the next node in the optimum path from $u$ to $v$? Suppose it is $w$. Then what is $OPT(u, v, S)$?
A More General Problem: TSP Path

**Input** A graph $G = (V, E)$ with non-negative edge costs/lengths ($c(e)$ for edge $e$) and two nodes $s, t$

**Goal** Find a path from $s$ to $t$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

- $OPT(u, v, S)$: optimum TSP Path from $u$ to $v$ in the graph restricted to $S$ (here $u, v \in S$).

What is the next node in the optimum path from $u$ to $v$? Suppose it is $w$. Then what is $OPT(u, v, S)$?

$$OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$$

We do not know $w$! So try all possibilities for $w$. 
A Recursive Solution

\[ OPT(u, v, S) = \min_{w \in S, w \neq u, v} (c(u, w) + OPT(w, v, S - \{u\})) \]
A Recursive Solution

\[ \text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + \text{OPT}(w, v, S - \{u\}) \right) \]

What are the subproblems for the original problem OPT(s, t, V)?

Exercise: Show that one can compute TSP using above dynamic program in \( O(n^3 \times n) \) time and \( O(n^2 \times n) \) space.

Disadvantage of dynamic programming solution: memory!
A Recursive Solution

\[ OPT(u, v, S) = \min_{w \in S, w \neq u, v} (c(u, w) + OPT(w, v, S - \{u\})) \]

What are the subproblems for the original problem \( OPT(s, t, V) \)?

\( OPT(u, v, S) \) for \( u, v \in S, S \subseteq V \).

How many subproblems?

Exercise: Show that one can compute TSP using above dynamic program in \( O(n^3) \) time and \( O(n^2) \) space.

Disadvantage of dynamic programming solution: memory!
A Recursive Solution

\[ OPT(u, v, S) = \min_{w \in S, w \neq u, v} (c(u, w) + OPT(w, v, S - \{u\})) \]

What are the subproblems for the original problem \( OPT(s, t, V) \)?
\( OPT(u, v, S) \) for \( u, v \in S, S \subseteq V \).

How many subproblems?
- number of distinct subsets \( S \) of \( V \) is at most \( 2^n \)
- number of pairs of nodes in a set \( S \) is at most \( n^2 \)
- hence number of subproblems is \( O(n^22^n) \)
A Recursive Solution

\[ \text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} (c(u, w) + \text{OPT}(w, v, S - \{u\})) \]

What are the subproblems for the original problem \( \text{OPT}(s, t, V) \)? \( \text{OPT}(u, v, S) \) for \( u, v \in S, S \subseteq V \).

How many subproblems?
- number of distinct subsets \( S \) of \( V \) is at most \( 2^n \)
- number of pairs of nodes in a set \( S \) is at most \( n^2 \)
- hence number of subproblems is \( O(n^22^n) \)

**Exercise:** Show that one can compute TSP using above dynamic program in \( O(n^32^n) \) time and \( O(n^22^n) \) space.

Disadvantage of dynamic programming solution:
A Recursive Solution

\[
OPT(u, v, S) = \min_{w \in S, w \neq u, v} (c(u, w) + OPT(w, v, S - \{u\}))
\]

What are the subproblems for the original problem \(OPT(s, t, V)\)? \(OPT(u, v, S)\) for \(u, v \in S, S \subseteq V\).

How many subproblems?

- number of distinct subsets \(S\) of \(V\) is at most \(2^n\)
- number of pairs of nodes in a set \(S\) is at most \(n^2\)
- hence number of subproblems is \(O(n^2 2^n)\)

Exercise: Show that one can compute TSP using above dynamic program in \(O(n^3 2^n)\) time and \(O(n^2 2^n)\) space.

Disadvantage of dynamic programming solution: memory!
Dynamic Programming = Smart Recursion + Memoization
Dynamic Programming = Smart Recursion + Memoization

- How to come up with the recursion?
- How to recognize that dynamic programming may apply?
Some Tips

- Problems where there is a natural linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
- Problems involving trees: recursion based on subtrees.
- More generally:
  - Problem admits a natural recursive divide and conquer
  - If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
  - If optimal solution depends on all pieces then can apply dynamic programming if interface/interaction between pieces is limited. Augument recursion to not simply find an optimum solution but also an optimum soluton for each possible way to interact with the other pieces.
Examples

- Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?
- Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?
- Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?
- Independent Set in an graph: break graph into two graphs. What is the interaction? Very high!
- Knapsack: Split items into two sets of half each. What is the interaction?