CS 573: Algorithms, Fall 2014

Dynamic Programming

Lecture 6

September 11, 2014

Part I

Total Recall on DAGs

DAGs

Definition

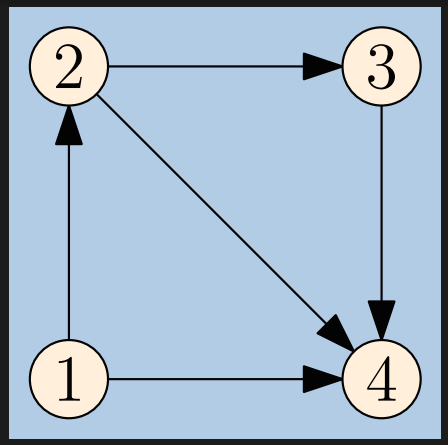
A ***DAG*** is a directed acyclic graph. That is a directed graph with no cycles.

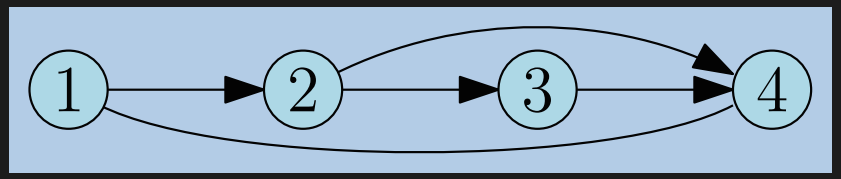
Topological Ordering/Sorting

You should already know that...

Definition

A ***topological ordering*** / ***topological sorting*** of $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.





Lemma

A directed graph **G** *can be topologically ordered iff it is a* DAG.

Lemma

More things you should already know

You should already know that...

Lemma

The strong connected components of a directed graph, can be computed in $O(n+m)$ time.

Part II

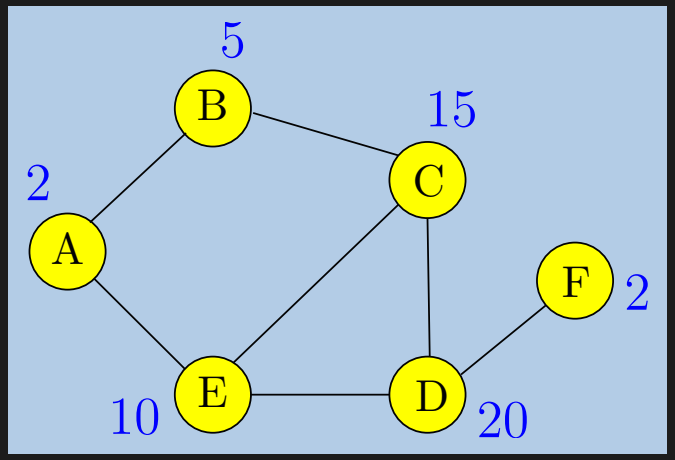
Maximum Weighted Independent Set in Trees

Maximum Weight Independent Set Problem

Problem (Max Weight Independent Set)

Input: *Graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *and weights* $w(v) \geq 0$ *for each* $v \in \mathbf{V}$.

Goal: *Find maximum weight independent set in* $\mathbf{G}$.



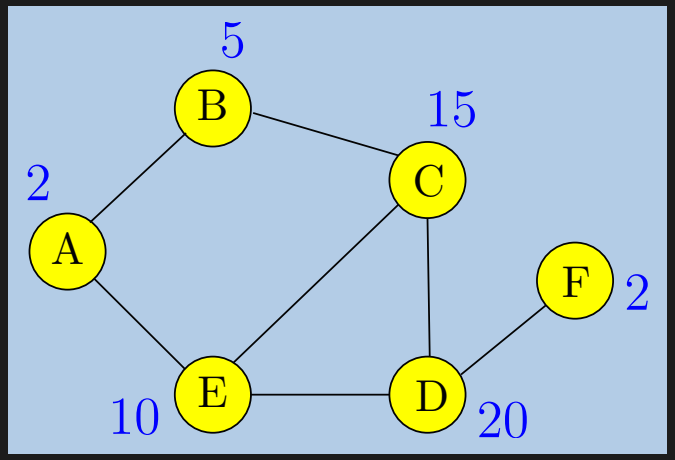
Maximum weight independent set in above graph: $\{B, D\}$.

Maximum Weight Independent Set Problem

Problem (Max Weight Independent Set)

Input: *Graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *and weights* $w(v) \geq 0$ *for each* $v \in \mathbf{V}$.

Goal: *Find maximum weight independent set in* $\mathbf{G}$.



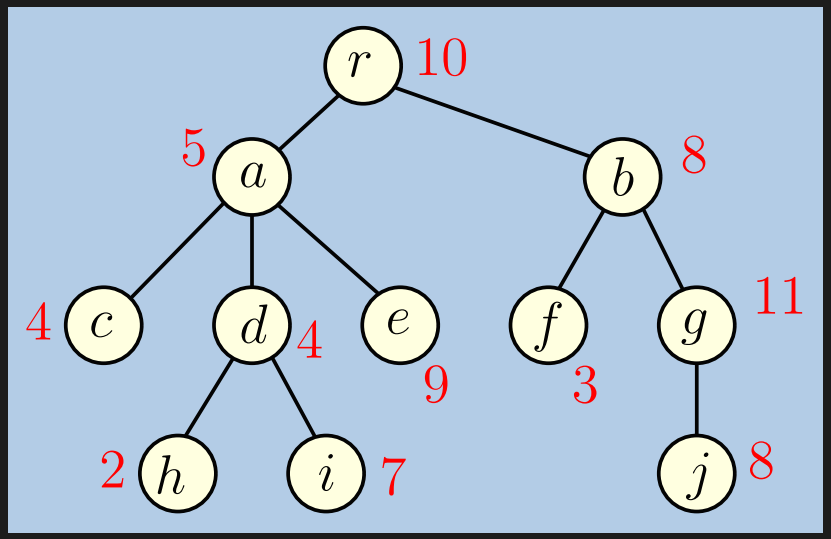
Maximum weight independent set in above graph: $\{B, D\}$.

Maximum Weight Independent Set in a Tree

Problem (Max W. Independent Set in Tree)

Input: *Tree* $\mathbf{T} = (\mathbf{V}, \mathbf{E})$ *and weights* $w(v) \geq 0$ *for each* $v \in \mathbf{V}$.

Goal: *Find maximum weight independent set in* $\mathbf{T}$.



Maximum weight independent set in above tree: ??

Towards a Recursive Solution

1. For an arbitrary graph **G**:
   1. Number vertices as $v_1, v_2, \ldots, v_n$
   2. Find recursively optimum solutions without v_n (recurse on $\mathbf{G} - v_n$) and with v_n (recurse on $\mathbf{G} - v_n - N(v_n)$ & include v_n).
   3. Saw that if graph **G** is arbitrary there was no good ordering that resulted in a small number of subproblems.
2. What about a tree?
3. Natural candidate for v_n is root r of **T**?

Towards a Recursive Solution

1. For an arbitrary graph **G**:
   1. Number vertices as $v_1, v_2, \ldots, v_n$
   2. Find recursively optimum solutions without v_n (recurse on $\mathbf{G} - v_n$) and with v_n (recurse on $\mathbf{G} - v_n - N(v_n)$ & include v_n).
   3. Saw that if graph **G** is arbitrary there was no good ordering that resulted in a small number of subproblems.
2. What about a tree?
3. Natural candidate for v_n is root r of **T**?

Towards a Recursive Solution

1. For an arbitrary graph $\mathbf{G}$:
   1. Number vertices as $v_1, v_2, \ldots, v_n$
   2. Find recursively optimum solutions without v_n (recurse on $\mathbf{G} - v_n$) and with v_n (recurse on $\mathbf{G} - v_n - N(v_n)$ & include v_n).
   3. Saw that if graph $\mathbf{G}$ is arbitrary there was no good ordering that resulted in a small number of subproblems.
2. What about a tree?
3. Natural candidate for v_n is root r of $\mathbf{T}$?

Towards a Recursive Solution

1. For an arbitrary graph $\mathbf{G}$:
   1. Number vertices as $v_1, v_2, \ldots, v_n$
   2. Find recursively optimum solutions without v_n (recurse on $\mathbf{G} - v_n$) and with v_n (recurse on $\mathbf{G} - v_n - N(v_n)$ & include v_n).
   3. Saw that if graph $\mathbf{G}$ is arbitrary there was no good ordering that resulted in a small number of subproblems.
2. What about a tree?
3. Natural candidate for v_n is root r of $\mathbf{T}$?

Towards a Recursive Solution

1. For an arbitrary graph $\mathbf{G}$:
   1. Number vertices as $v_1, v_2, \ldots, v_n$
   2. Find recursively optimum solutions without v_n (recurse on $\mathbf{G} - v_n$) and with v_n (recurse on $\mathbf{G} - v_n - N(v_n)$ & include v_n).
   3. Saw that if graph $\mathbf{G}$ is arbitrary there was no good ordering that resulted in a small number of subproblems.
2. What about a tree?
3. Natural candidate for v_n is root r of $\mathbf{T}$?

Towards a Recursive Solution

1. For an arbitrary graph **G**:
   1. Number vertices as $v_1, v_2, \ldots, v_n$
   2. Find recursively optimum solutions without v_n (recurse on $\mathbf{G} - v_n$) and with v_n (recurse on $\mathbf{G} - v_n - N(v_n)$ & include v_n).
   3. Saw that if graph **G** is arbitrary there was no good ordering that resulted in a small number of subproblems.
2. What about a tree?
3. Natural candidate for v_n is root r of **T**?

Towards a Recursive Solution

Maximum Weight Independent Set in a Tree

- Natural candidate for v_n is root r of $\mathbf{T}$?
- Let $\mathcal{O}$ be an optimum solution to the whole problem.

  Case $r \notin \mathcal{O}$: $\mathcal{O}$ contains optimum solution for each subtree hanging from a child of r.

  Case $r \in \mathcal{O}$: None of children of r are in $\mathcal{O}$. $\mathcal{O} \setminus \{r\}$ contains an optimum solution for each subtree hanging at a grandchild of r.
- Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

Towards a Recursive Solution

Maximum Weight Independent Set in a Tree

- Natural candidate for v_n is root r of $\mathbf{T}$?
- Let $\mathcal{O}$ be an optimum solution to the whole problem.

  Case $r \notin \mathcal{O}$: $\mathcal{O}$ contains optimum solution for each subtree hanging from a child of r.

  Case $r \in \mathcal{O}$: None of children of r are in $\mathcal{O}$. $\mathcal{O} \setminus \{r\}$ contains an optimum solution for each subtree hanging at a grandchild of r.
- Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

Towards a Recursive Solution

Maximum Weight Independent Set in a Tree

- Natural candidate for v_n is root r of $\mathbf{T}$?
- Let $\mathcal{O}$ be an optimum solution to the whole problem.

  Case $r \notin \mathcal{O}$: $\mathcal{O}$ contains optimum solution for each subtree hanging from a child of r.

  Case $r \in \mathcal{O}$: None of children of r are in $\mathcal{O}$. $\mathcal{O} \setminus \{r\}$ contains an optimum solution for each subtree hanging at a grandchild of r.
- Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

Towards a Recursive Solution

Maximum Weight Independent Set in a Tree

- Natural candidate for v_n is root r of $\mathbf{T}$?
- Let $\mathcal{O}$ be an optimum solution to the whole problem.

  Case $r \notin \mathcal{O}$: $\mathcal{O}$ contains optimum solution for each subtree hanging from a child of r.

  Case $r \in \mathcal{O}$: None of children of r are in $\mathcal{O}$. $\mathcal{O} \setminus \{r\}$ contains an optimum solution for each subtree hanging at a grandchild of r.
- Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

Towards a Recursive Solution

Maximum Weight Independent Set in a Tree

1. Natural candidate for v_n is root r of $\mathbf{T}$?
2. Let $\mathcal{O}$ be an optimum solution to the whole problem.

   Case $r \notin \mathcal{O}$: $\mathcal{O}$ contains optimum solution for each subtree hanging from a child of r.

   Case $r \in \mathcal{O}$: None of children of r are in $\mathcal{O}$. $\mathcal{O} \setminus \{r\}$ contains an optimum solution for each subtree hanging at a grandchild of r.
3. Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

A Recursive Solution

- $T(u)$: subtree of **T** hanging at node u.
- $OPT(u)$: max weighted independent set value in $T(u)$.
- $OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$

A Recursive Solution

- $T(u)$: subtree of **T** hanging at node u.
- $OPT(u)$: max weighted independent set value in $T(u)$.
- $OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$

A Recursive Solution

1. $T(u)$: subtree of **T** hanging at node u.
2. $OPT(u)$: max weighted independent set value in $T(u)$.
3. $OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$

A Recursive Solution

1. $T(u)$: subtree of **T** hanging at node u.
2. $OPT(u)$: max weighted independent set value in $T(u)$.
3. $OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$

Iterative Algorithm

- Compute $OPT(u)$ bottom up.
- To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u
- What is an ordering of nodes of a tree $\mathbf{T}$ to achieve above?
- Post-order traversal of a tree.

Iterative Algorithm

- Compute $OPT(u)$ bottom up.
- To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u
- What is an ordering of nodes of a tree $\mathbf{T}$ to achieve above?
- Post-order traversal of a tree.

Iterative Algorithm

1. Compute $OPT(u)$ bottom up.
2. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u
3. What is an ordering of nodes of a tree **T** to achieve above?
4. Post-order traversal of a tree.

Iterative Algorithm

1. Compute $OPT(u)$ bottom up.
2. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u
3. What is an ordering of nodes of a tree $\mathbf{T}$ to achieve above?
4. Post-order traversal of a tree.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

return $M[v_n]$ // Note: v_n is the root of **T**

Space: $O(n)$ to store the value at each node of **T**

Running time:

- Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
- Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ // Note: v_n is the root of **T**

Space: $O(n)$ to store the value at each node of **T**

Running time:

- Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
- Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ // Note: v_n is the root of **T**

Space: $O(n)$ to store the value at each node of **T**

Running time:

- Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
- Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ // Note: v_n is the root of **T**

Space: $O(n)$ to store the value at each node of **T**

Running time:

- Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
- Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ // Note: v_n is the root of **T**

Space: $O(n)$ to store the value at each node of **T**

Running time:

- Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
- Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

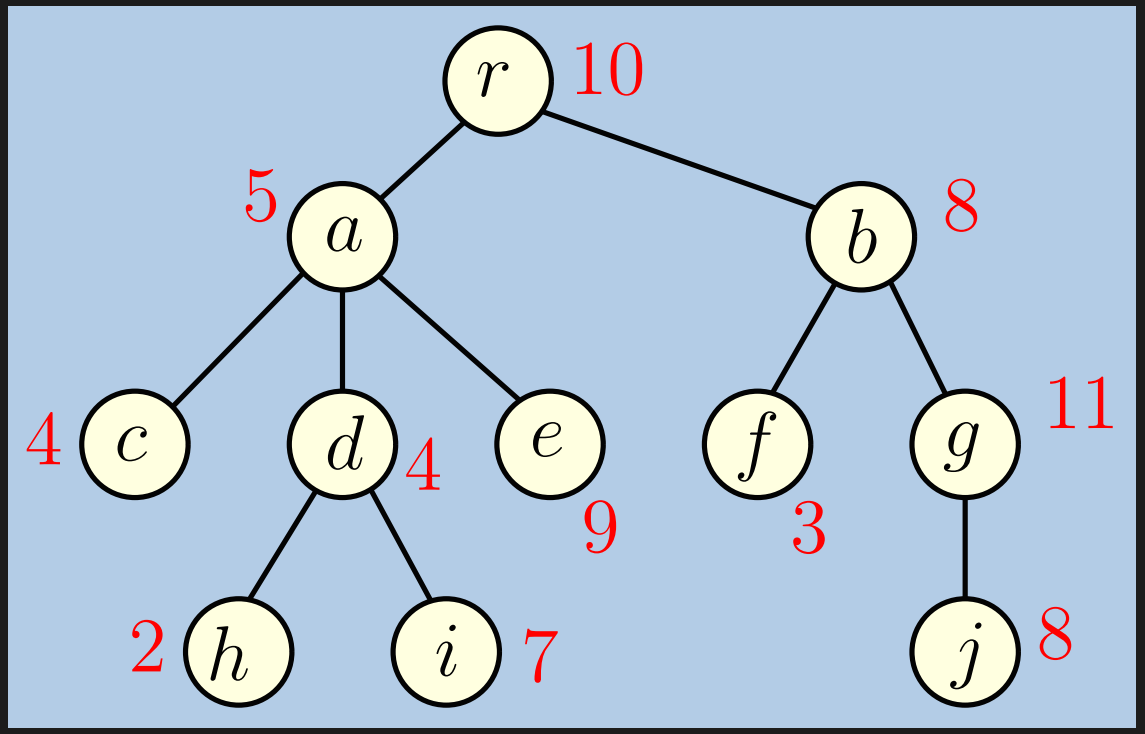
return $M[v_n]$ // Note: v_n is the root of **T**

Space: $O(n)$ to store the value at each node of **T**

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

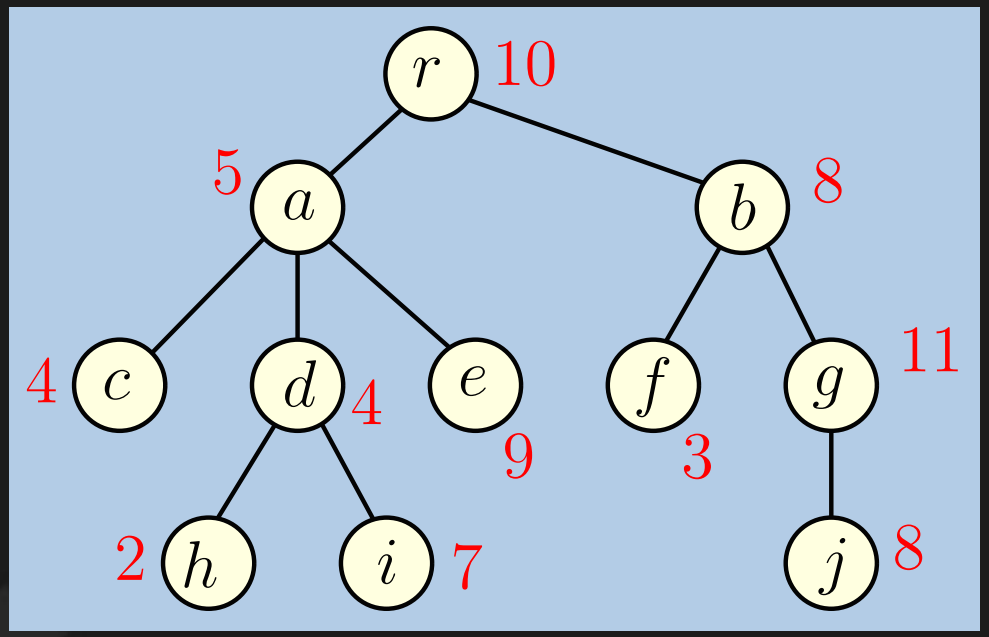
Example



Dominating set

Definition

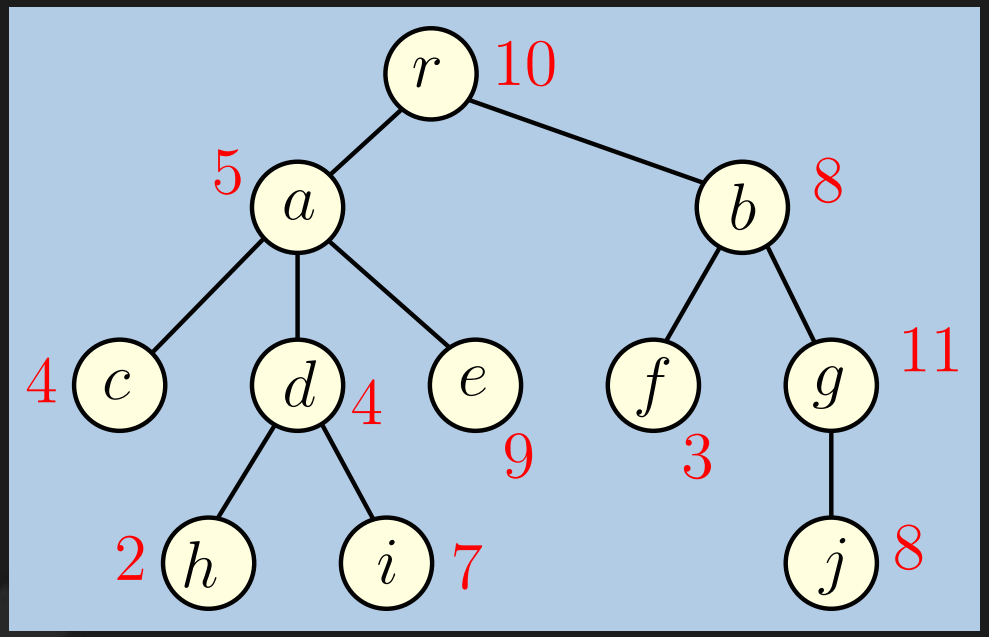
$\mathbf{G} = (\mathbf{V}, \mathbf{E})$. The set $X \subseteq \mathbf{V}$ is a ***dominating set***, if any vertex $v \in \mathbf{V}$ is either in X or is adjacent to a vertex in X.



Dominating set

Definition

$\mathbf{G} = (\mathbf{V}, \mathbf{E})$. The set $X \subseteq \mathbf{V}$ is a ***dominating set***, if any vertex $v \in \mathbf{V}$ is either in X or is adjacent to a vertex in X.



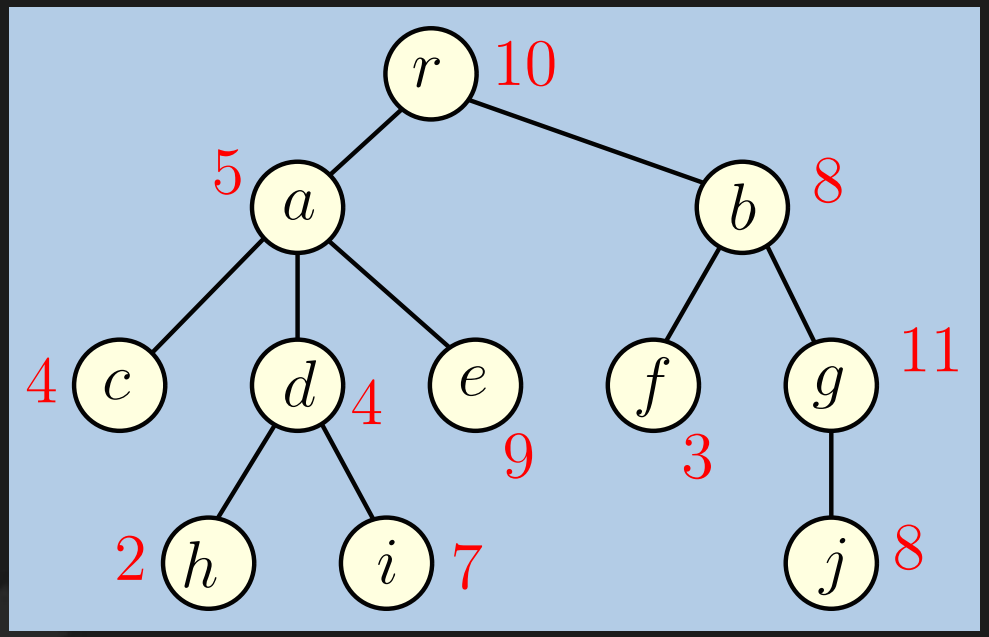
Problem

Given weights on vertices, compute the ***minimum*** *weight dominating set in* $\mathbf{G}$.

Dominating set

Definition

$\mathbf{G} = (\mathbf{V}, \mathbf{E})$. The set $X \subseteq \mathbf{V}$ is a ***dominating set***, if any vertex $v \in \mathbf{V}$ is either in X or is adjacent to a vertex in X.



Problem

Given weights on vertices, compute the ***minimum*** *weight dominating set in* $\mathbf{G}$.

Dominating Set is **NP-Hard**!

Part III

DAGs and Dynamic Programming

Recursion and DAGs

Observation

A: recursive algorithm for problem Π.

For each instance I of Π there is an associated DAG $\mathbf{G}(I)$.

- Create directed graph $\mathbf{G}(I)$ as follows...
- For each sub-problem in the execution of A on I create a node.
- If sub-problem v *depends* on or *recursively calls* sub-problem u add *directed* edge (u, v) to graph.
- $\mathbf{G}(I)$ is a DAG. Why? If $\mathbf{G}(I)$ has a cycle then A will not terminate on I.

Recursion and DAGs

Observation

A: recursive algorithm for problem Π.
For each instance I of Π there is an associated DAG $\mathbf{G}(I)$.

- Create directed graph $\mathbf{G}(I)$ as follows...
- For each sub-problem in the execution of A on I create a node.
- If sub-problem v *depends* on or *recursively calls* sub-problem u add *directed* edge (u, v) to graph.
- $\mathbf{G}(I)$ is a DAG. Why? If $\mathbf{G}(I)$ has a cycle then A will not terminate on I.

Recursion and DAGs

Observation

A: recursive algorithm for problem Π.
For each instance I of Π there is an associated DAG $\mathbf{G}(I)$.

1. Create directed graph $\mathbf{G}(I)$ as follows...
2. For each sub-problem in the execution of A on I create a node.
3. If sub-problem v *depends* on or *recursively calls* sub-problem u add *directed* edge (u, v) to graph.
4. $\mathbf{G}(I)$ is a DAG. Why? If $\mathbf{G}(I)$ has a cycle then A will not terminate on I.

Recursion and DAGs

Observation

A: recursive algorithm for problem Π.
For each instance I of Π there is an associated DAG $\mathbf{G}(I)$.

1. Create directed graph $\mathbf{G}(I)$ as follows...
2. For each sub-problem in the execution of A on I create a node.
3. If sub-problem v *depends* on or *recursively calls* sub-problem u add *directed* edge (u, v) to graph.
4. $\mathbf{G}(I)$ is a DAG. Why? If $\mathbf{G}(I)$ has a cycle then A will not terminate on I.

Recursion and DAGs

Observation

A: recursive algorithm for problem Π.
For each instance I of Π there is an associated DAG $\mathbf{G}(I)$.

1. Create directed graph $\mathbf{G}(I)$ as follows...
2. For each sub-problem in the execution of A on I create a node.
3. If sub-problem v *depends* on or *recursively calls* sub-problem u add *directed* edge (u, v) to graph.
4. $\mathbf{G}(I)$ is a DAG. Why? If $\mathbf{G}(I)$ has a cycle then A will not terminate on I.

Recursion and DAGs

Observation

A: recursive algorithm for problem Π.
For each instance I of Π there is an associated DAG $\mathbf{G}(I)$.

1. Create directed graph $\mathbf{G}(I)$ as follows...
2. For each sub-problem in the execution of A on I create a node.
3. If sub-problem v *depends* on or *recursively calls* sub-problem u add *directed* edge (u, v) to graph.
4. $\mathbf{G}(I)$ is a DAG. Why? If $\mathbf{G}(I)$ has a cycle then A will not terminate on I.

Iterative Algorithm for...

Dynamic Programming and DAGs

Observation

An iterative algorithm B obtained from a recursive algorithm A for a problem Π does the following:

> *For each instance I of Π, it computes a topological sort of $\mathbf{G}(I)$ and evaluates sub-problems according to the topological ordering.*

- Sometimes the DAG $\mathbf{G}(I)$ can be obtained directly without thinking about the recursive algorithm A
- In some cases (not all) the computation of an optimal solution reduces to a shortest/longest path in DAG $\mathbf{G}(I)$
- Topological sort based shortest/longest path computation is dynamic programming!

Iterative Algorithm for...

Dynamic Programming and DAGs

Observation

An iterative algorithm B obtained from a recursive algorithm A for a problem Π does the following:

For each instance I of Π, it computes a topological sort of $\mathbf{G}(I)$ and evaluates sub-problems according to the topological ordering.

- Sometimes the DAG $\mathbf{G}(I)$ can be obtained directly without thinking about the recursive algorithm A
- In some cases (not all) the computation of an optimal solution reduces to a shortest/longest path in DAG $\mathbf{G}(I)$
- Topological sort based shortest/longest path computation is dynamic programming!

Iterative Algorithm for...

Dynamic Programming and DAGs

Observation

An iterative algorithm B obtained from a recursive algorithm A for a problem Π does the following:

For each instance I of Π, it computes a topological sort of $\mathbf{G}(I)$ and evaluates sub-problems according to the topological ordering.

1. Sometimes the DAG $\mathbf{G}(I)$ can be obtained directly without thinking about the recursive algorithm A
2. In some cases (not all) the computation of an optimal solution reduces to a shortest/longest path in DAG $\mathbf{G}(I)$
3. Topological sort based shortest/longest path computation is dynamic programming!

Iterative Algorithm for...

Dynamic Programming and DAGs

Observation

An iterative algorithm B obtained from a recursive algorithm A for a problem Π does the following:

For each instance I of Π, it computes a topological sort of $\mathbf{G}(I)$ and evaluates sub-problems according to the topological ordering.

1. Sometimes the DAG $\mathbf{G}(I)$ can be obtained directly without thinking about the recursive algorithm A
2. In some cases (not all) the computation of an optimal solution reduces to a shortest/longest path in DAG $\mathbf{G}(I)$
3. Topological sort based shortest/longest path computation is dynamic programming!

A quick reminder...

A Recursive Algorithm for weighted interval scheduling

Let O_i be value of an optimal schedule for the first i jobs.

```
Schedule(n):
        if n = 0 then return 0
        if n = 1 then return w(v_1)
        O_{p(n)} ← Schedule(p(n))
        O_{n-1} ← Schedule(n - 1)
        if (O_{p(n)} + w(v_n) < O_{n-1}) then
            O_n = O_{n-1}
        else
            O_n = O_{p(n)} + w(v_n)
        return O_n
```

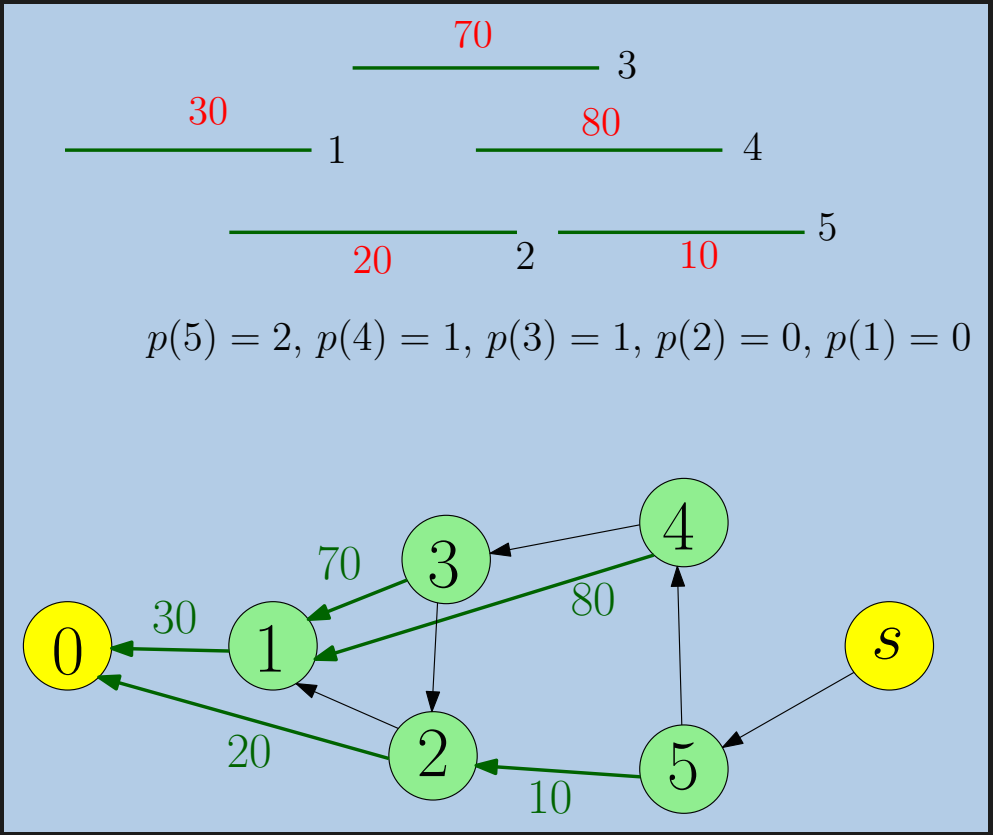
Weighted Interval Scheduling via...

Longest Path in a DAG

Given intervals, create a DAG as follows:

1. Create one node for each interval, plus a dummy sink node 0 for interval 0, plus a dummy source node s.
2. For each interval i add edge $(i, p(i))$ of the length/weight of v_i.
3. Add an edge from s to n of length 0.
4. For each interval i add edge $(i, i-1)$ of length 0.

Example



Relating Optimum Solution

1. Given interval problem instance I let $\mathbf{G}(I)$ denote the DAG constructed as described.

Claim

2. *Optimum solution to weighted interval scheduling instance I is given by* longest *path from s to 0 in $\mathbf{G}(I)$.*

3. Assuming claim is true,
   1. If I has n intervals, DAG $\mathbf{G}(I)$ has $n+2$ nodes and $O(n)$ edges. Creating $\mathbf{G}(I)$ takes $O(n \log n)$ time: to find $p(i)$ for each i. How?
   2. Longest path can be computed in $O(n)$ time — recall $O(m+n)$ algorithm for shortest/longest paths in DAGs.

Relating Optimum Solution

1. Given interval problem instance I let $\mathbf{G}(I)$ denote the DAG constructed as described.

Claim

2. *Optimum solution to weighted interval scheduling instance I is given by* longest *path from s to 0 in $\mathbf{G}(I)$.*

3. Assuming claim is true,
   1. If I has n intervals, DAG $\mathbf{G}(I)$ has $n+2$ nodes and $O(n)$ edges. Creating $\mathbf{G}(I)$ takes $O(n \log n)$ time: to find $p(i)$ for each i. How?
   2. Longest path can be computed in $O(n)$ time — recall $O(m+n)$ algorithm for shortest/longest paths in DAGs.

Relating Optimum Solution

1. Given interval problem instance I let $\mathbf{G}(I)$ denote the DAG constructed as described.

Claim

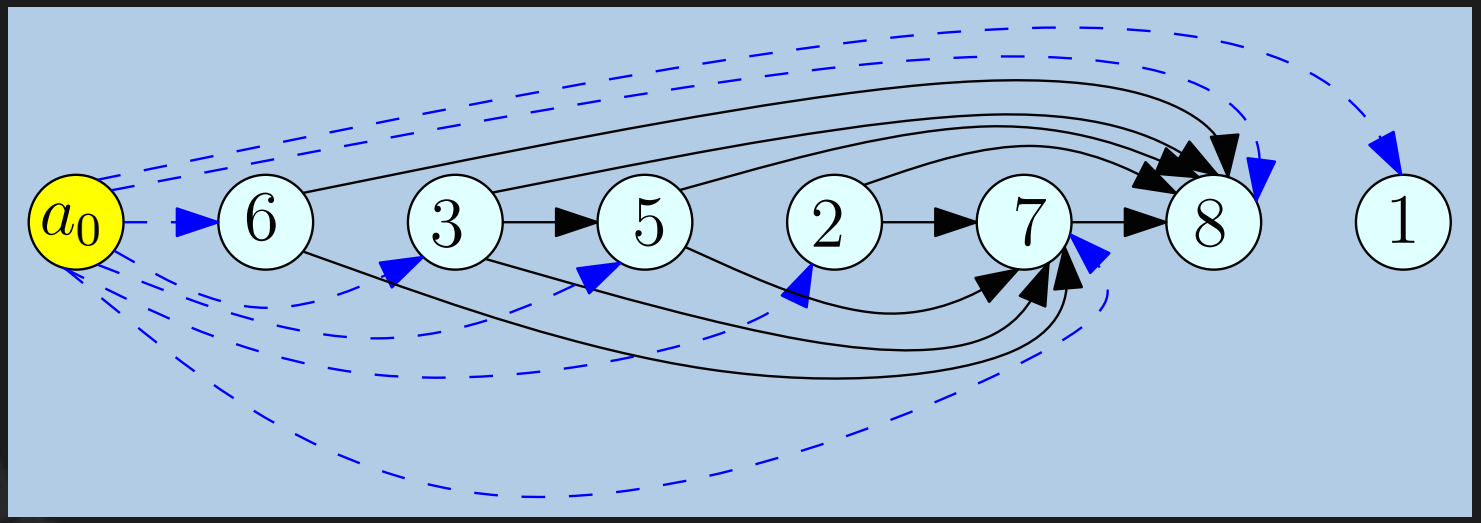
2. *Optimum solution to weighted interval scheduling instance I is given by* longest *path from s to 0 in $\mathbf{G}(I)$.*

3. Assuming claim is true,
   1. If I has n intervals, DAG $\mathbf{G}(I)$ has $n+2$ nodes and $O(n)$ edges. Creating $\mathbf{G}(I)$ takes $O(n \log n)$ time: to find $p(i)$ for each i. How?
   2. Longest path can be computed in $O(n)$ time — recall $O(m+n)$ algorithm for shortest/longest paths in DAGs.

DAG for Longest Increasing Sequence

Given sequence $a_1, a_2, \ldots, a_n$ create DAG as follows:

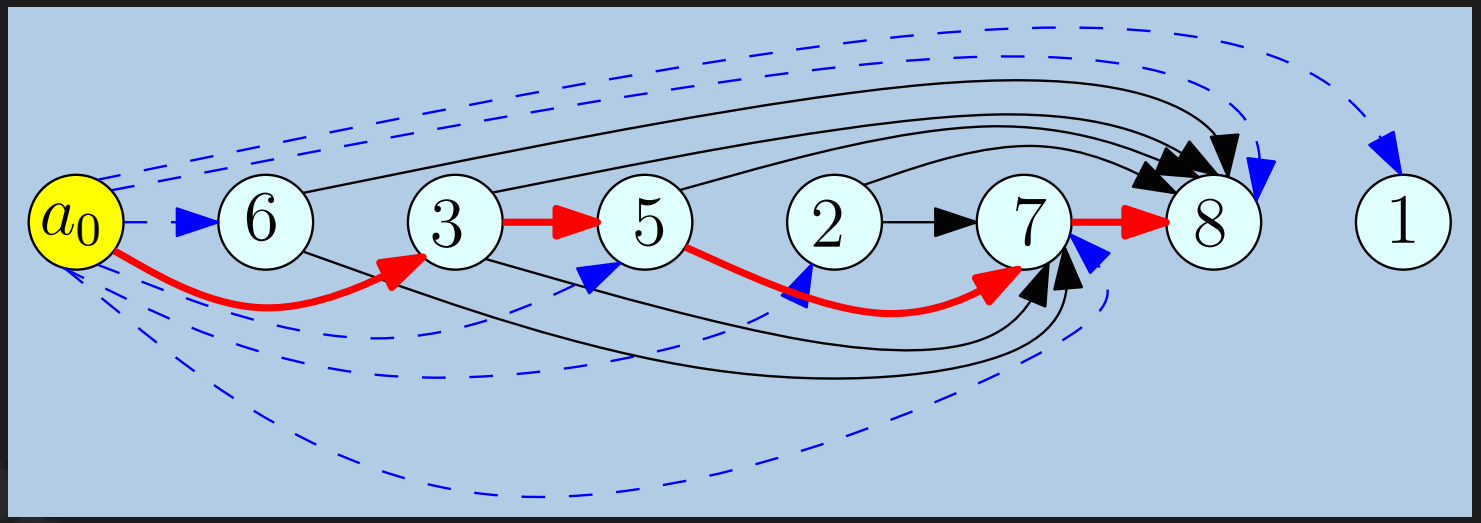
1. add sentinel a_0 to sequence where a_0 is less than smallest element in sequence
2. for each i there is a node v_i
3. if $i < j$ and $a_i < a_j$ add an edge (v_i, v_j)
4. find longest path from v_0



DAG for Longest Increasing Sequence

Given sequence $a_1, a_2, \ldots, a_n$ create DAG as follows:

1. add sentinel a_0 to sequence where a_0 is less than smallest element in sequence
2. for each i there is a node v_i
3. if $i < j$ and $a_i < a_j$ add an edge (v_i, v_j)
4. find longest path from v_0



Part IV

Edit Distance and Sequence Alignment

Spell Checking Problem

- Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?
- What does nearness mean?
- Question: Given two strings $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a *distance* between them?
- Edit Distance: minimum number of "edits" to transform x into y.

Spell Checking Problem

1. Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?
2. What does nearness mean?
3. Question: Given two strings $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a *distance* between them?
4. Edit Distance: minimum number of "edits" to transform x into y.

Spell Checking Problem

1. Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?
2. What does nearness mean?
3. Question: Given two strings $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a *distance* between them?
4. Edit Distance: minimum number of "edits" to transform x into y.

Spell Checking Problem

1. Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?
2. What does nearness mean?
3. Question: Given two strings $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a *distance* between them?
4. Edit Distance: minimum number of "edits" to transform x into y.

Edit Distance

Definition

Edit distance between two words X and Y is the number of letter insertions, letter deletions and letter substitutions required to obtain Y from X.

Example

The edit distance between FOOD and MONEY is at most 4:

$$\underline{\mathrm{F}}\mathrm{OOD} \to \mathrm{MO}\underline{\mathrm{O}}\mathrm{D} \to \mathrm{MON}\underline{\mathrm{O}}\mathrm{D} \to \mathrm{MONE}\underline{\mathrm{D}} \to \mathrm{MONEY}$$

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1,1), (2,2), (3,3), (4,5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1,1), (2,2), (3,3), (4,5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1,1),(2,2),(3,3),(4,5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

Applications

1. Spell-checkers and Dictionaries
2. Unix `diff`
3. DNA sequence alignment ... but, we need a new metric

Similarity Metric

Definition

For two strings X and Y, the cost of alignment M is

1. [Gap penalty] For each gap in the alignment, we incur a cost δ.
2. [Mismatch cost] For each pair p and q that have been matched in M, we incur cost α_{pq}; typically $\alpha_{pp} = 0$.

Edit distance is special case when $\delta = \alpha_{pq} = 1$.

Similarity Metric

Definition

For two strings X and Y, the cost of alignment M is

1. [Gap penalty] For each gap in the alignment, we incur a cost δ.
2. [Mismatch cost] For each pair p and q that have been matched in M, we incur cost α_{pq}; typically $\alpha_{pp} = 0$.

Edit distance is special case when $\delta = \alpha_{pq} = 1$.

An Example

Example

$$\begin{array}{c|c|c|c|c|c|c|c|c|c|}
o & & c & u & r & r & a & n & c & e \\
o & c & c & u & r & r & e & n & c & e
\end{array} \qquad \text{Cost } = \delta + \alpha_{ae}$$

Alternative:

$$\begin{array}{c|c|c|c|c|c|c|c|c|c|}
o & & c & u & r & r & & a & n & c & e \\
o & c & c & u & r & r & e & & n & c & e
\end{array} \qquad \text{Cost } = 3\delta$$

Or a really stupid solution (delete string, insert other string):

$$\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
o & c & u & r & r & a & n & c & e & & o & c & c & u & r & r & e & n & c & e
\end{array}$$

Cost $= 19\delta$.

Sequence Alignment

Problem (Sequence Alignment)

Input: *Given two words* X *and* Y*, and gap penalty* δ *and mismatch costs* α_{pq}*.*

Goal: *Find alignment of minimum cost.*

Edit distance

Basic observation

Let $X = \alpha x$ and $Y = \beta y$
α, β: strings.
x and y single characters.
Think about optimal edit distance between X and Y as alignment, and consider last column of alignment of the two strings:

α	x
β	y

or

α	x
βy	

or

αx	
β	y

Observation

Prefixes must have optimal alignment!

Problem Structure

Observation

Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$. If (m, n) are not matched then either the mth position of X remains unmatched or the nth position of Y remains unmatched.

1. Case x_m and y_n are matched.
   1. Pay mismatch cost $\alpha_{x_m y_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$
2. Case x_m is unmatched.
   1. Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$
3. Case y_n is unmatched.
   1. Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$

Subproblems and Recurrence

Optimal Costs

Let $\mathrm{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\mathrm{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1, j-1), \\ \delta + \mathrm{Opt}(i-1, j), \\ \delta + \mathrm{Opt}(i, j-1) \end{cases}$$

Base Cases: $\mathrm{Opt}(i, 0) = \delta \cdot i$ and $\mathrm{Opt}(0, j) = \delta \cdot j$

Subproblems and Recurrence

Optimal Costs

Let $\mathrm{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\mathrm{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1, j-1), \\ \delta + \mathrm{Opt}(i-1, j), \\ \delta + \mathrm{Opt}(i, j-1) \end{cases}$$

Base Cases: $\mathrm{Opt}(i, 0) = \delta \cdot i$ and $\mathrm{Opt}(0, j) = \delta \cdot j$

Dynamic Programming Solution

for `all` i **do** $M[i,0] = i\delta$
for `all` j **do** $M[0,j] = j\delta$

for $i = 1$ `to` m **do**
  for $j = 1$ `to` n **do**

$$M[i,j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

- Running time is $O(mn)$.

Dynamic Programming Solution

for `all` i **do** $M[i,0] = i\delta$
for `all` j **do** $M[0,j] = j\delta$

for $i = 1$ `to` m **do**
  for $j = 1$ `to` n **do**

$$M[i,j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.

Dynamic Programming Solution

for `all` i **do** $M[i,0] = i\delta$
for `all` j **do** $M[0,j] = j\delta$

for $i = 1$ `to` m **do**
  for $j = 1$ `to` n **do**

$$M[i,j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.
2. Space used is $O(mn)$.

Matrix and DAG of Computation

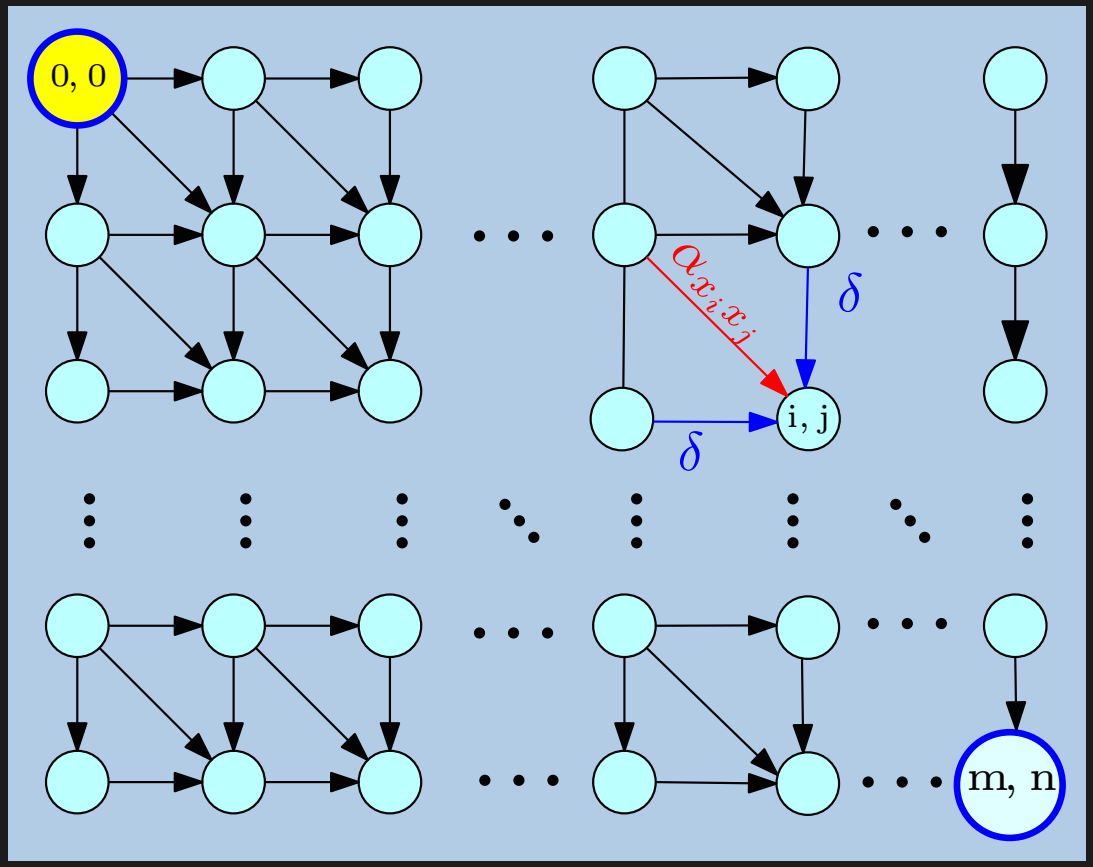


Figure: Iterative algorithm in previous slide computes values in row order. Optimal value is a shortest path from $(0, 0)$ to (m, n) in DAG.

Sequence Alignment in Practice

1. Typically the DNA sequences that are aligned are about 10^5 letters long!
2. So about 10^{10} operations and 10^{10} bytes needed
3. The killer is the 10GB storage
4. Can we reduce space requirements?

Optimizing Space

1. Recall

$$M(i,j) = \min \begin{cases} \alpha_{x_i y_j} + M(i-1, j-1), \\ \delta + M(i-1, j), \\ \delta + M(i, j-1) \end{cases}$$

2. Entries in jth column only depend on $(j-1)$st column and earlier entries in jth column
3. Only store the current column and the previous column reusing space; $N(i,0)$ stores $M(i,j-1)$ and $N(i,1)$ stores $M(i,j)$

Computing in column order to save space

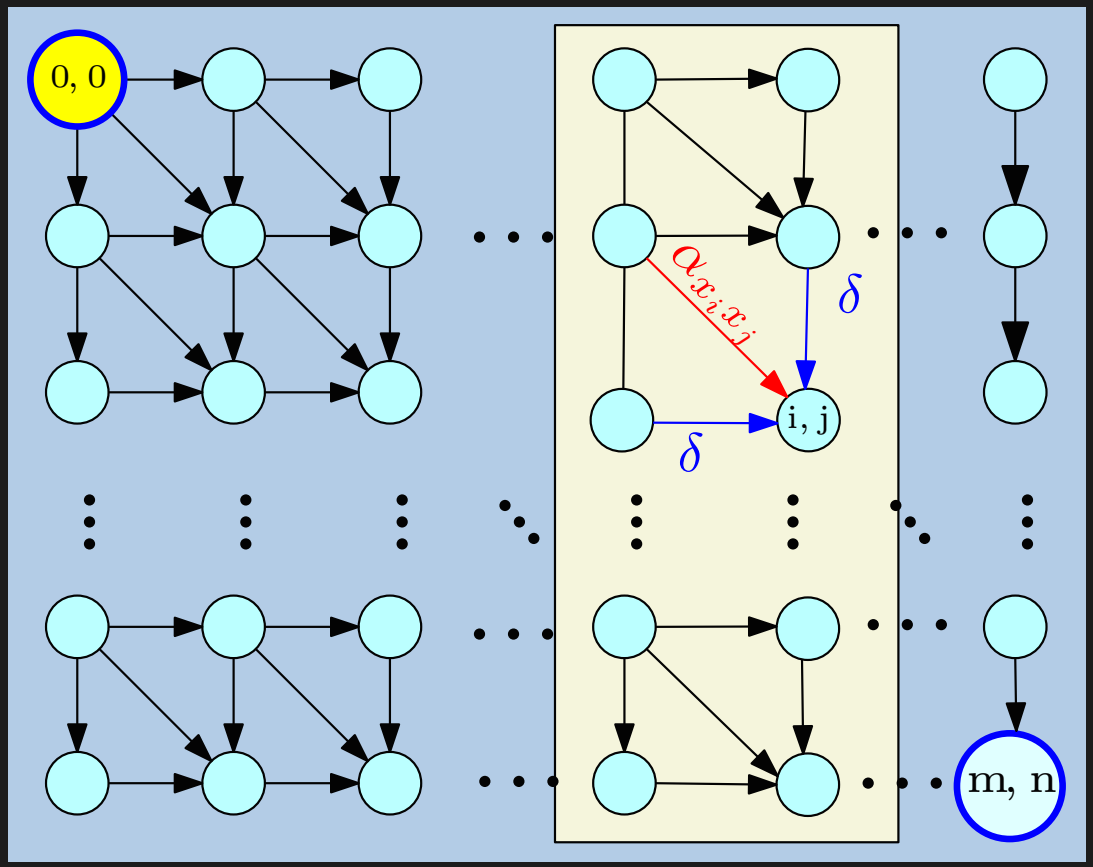


Figure: $M(i, j)$ only depends on previous column values. Keep only two columns and compute in column order.

Space Efficient Algorithm

```
for all i do N[i,0] = iδ
for j = 1 to n do
    N[0,1] = jδ (* corresponds to M(0,j) *)
    for i = 1 to m do
        N[i,1] = min { α_{x_i y_j} + N[i-1,0]
                     { δ + N[i-1,1]
                     { δ + N[i,0]
    for i = 1 to m do
        Copy N[i,0] = N[i,1]
```

Analysis

Running time is $O(mn)$ and space used is $O(2m) = O(m)$

Analyzing Space Efficiency

1. From the $m \times n$ matrix M we can construct the actual alignment (exercise)
2. Matrix N computes cost of optimal alignment but no way to construct the actual alignment
3. Space efficient computation of alignment? More complicated algorithm — see text book.

Takeaway Points

1. Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.
2. Given a recursive algorithm there is a natural DAG associated with the subproblems that are generated for given instance; this is the dependency graph. An iterative algorithm simply evaluates the subproblems in some topological sort of this DAG.
3. The space required to evaluate the answer can be reduced in some cases by a careful examination of that dependency DAG of the subproblems and keeping only a subset of the DAG at any time.

Part V

All Pairs Shortest Paths

Shortest Path Problems

Shortest Path Problems

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

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.
3. Find shortest paths for all pairs of nodes.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

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

1. Given nodes s, t find shortest path from s to t.
2. 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: $O(nm)$.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

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

1. Given nodes s, t find shortest path from s to t.
2. 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 Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ 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: $O(n^2 m)$. $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

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

1. Find shortest paths for all pairs of nodes.

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

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$. $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

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

1. Find shortest paths for all pairs of nodes.

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

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$. $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

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

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

Sub-problem idea: paths of fewer hops/edges

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

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

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

Sub-problem idea: paths of fewer hops/edges

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

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

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

Sub-problem idea: paths of fewer hops/edges

Hop-based Recur': 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 $OPT(v, k)$:

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

- Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
- Leads to Bellman-Ford algorithm — see text book.
- $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

Hop-based Recur': Single-Source Shortest Paths

1. Single-source problem: fix source s.
2. $OPT(v, k)$: shortest path dist. from s to v using at most k edges.
3. Note: $dist(s, v) = OPT(v, n-1)$.
4. Recursion for $OPT(v, k)$:

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

5. Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
6. Leads to Bellman-Ford algorithm — see text book.
7. $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

Hop-based Recur': Single-Source Shortest Paths

1. Single-source problem: fix source s.
2. $OPT(v, k)$: shortest path dist. from s to v using at most k edges.
3. Note: $dist(s, v) = OPT(v, n-1)$.
4. Recursion for $OPT(v, k)$:

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

5. Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
6. Leads to Bellman-Ford algorithm — see text book.
7. $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

Hop-based Recur': Single-Source Shortest Paths

1. Single-source problem: fix source s.
2. $OPT(v, k)$: shortest path dist. from s to v using at most k edges.
3. Note: $dist(s, v) = OPT(v, n-1)$.
4. Recursion for $OPT(v, k)$:

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

5. Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
6. Leads to Bellman-Ford algorithm — see text book.
7. $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

Hop-based Recur': Single-Source Shortest Paths

1. Single-source problem: fix source s.
2. $OPT(v, k)$: shortest path dist. from s to v using at most k edges.
3. Note: $dist(s, v) = OPT(v, n-1)$.
4. Recursion for $OPT(v, k)$:

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

5. Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
6. Leads to Bellman-Ford algorithm — see text book.
7. $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

Hop-based Recur': Single-Source Shortest Paths

1. Single-source problem: fix source s.
2. $OPT(v, k)$: shortest path dist. from s to v using at most k edges.
3. Note: $dist(s, v) = OPT(v, n-1)$.
4. Recursion for $OPT(v, k)$:

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

5. Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
6. Leads to Bellman-Ford algorithm — see text book.
7. $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

Hop-based Recur': Single-Source Shortest Paths

1. Single-source problem: fix source s.
2. $OPT(v, k)$: shortest path dist. from s to v using at most k edges.
3. Note: $dist(s, v) = OPT(v, n-1)$.
4. Recursion for $OPT(v, k)$:

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

5. Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
6. Leads to Bellman-Ford algorithm — see text book.
7. $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

Hop-based Recur': Single-Source Shortest Paths

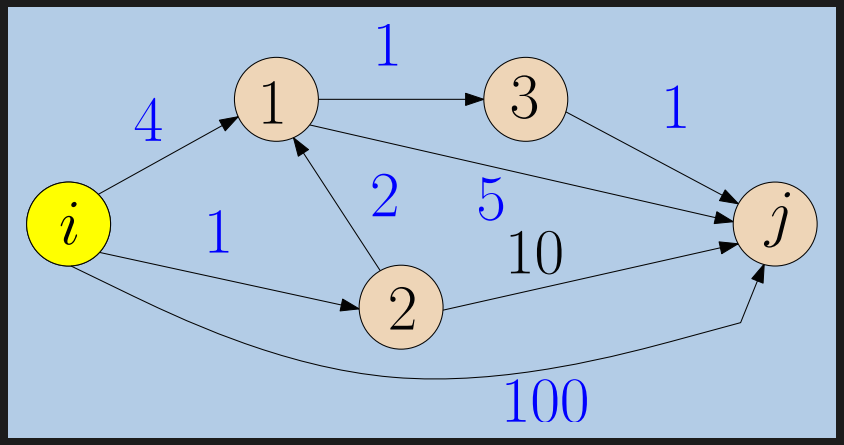
- Single-source problem: fix source s.
- $OPT(v, k)$: shortest path dist. from s to v using at most k edges.
- Note: $dist(s, v) = OPT(v, n-1)$.
- Recursion for $OPT(v, k)$:

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

- Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
- 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 index of intermediate nodes

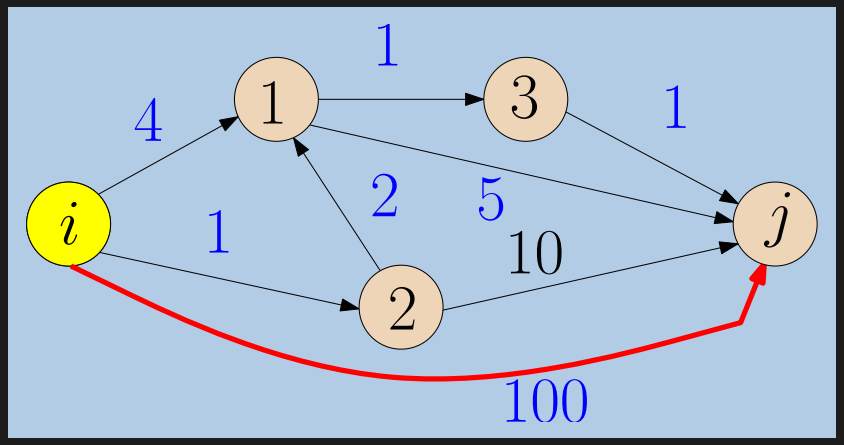
- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $dist(i, j, k)$: shortest path distance between v_i and v_j among all paths in which the largest index of an *intermediate node* is at most k



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= 8 \\
dist(i, j, 3) &= 5
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

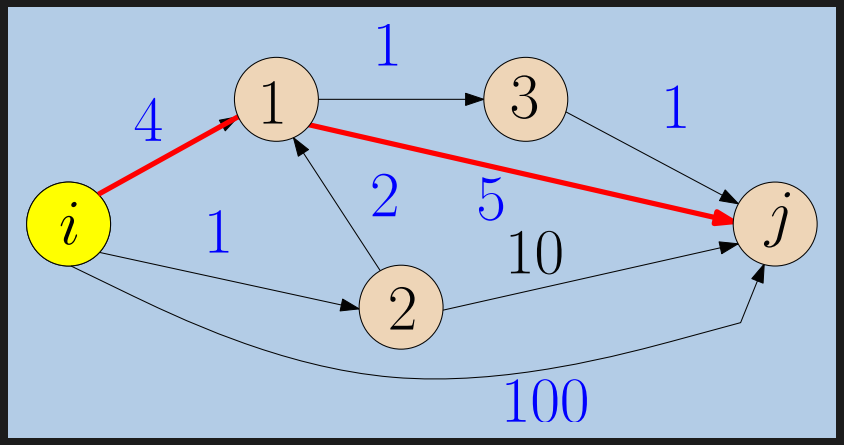
- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $dist(i, j, k)$: shortest path distance between v_i and v_j among all paths in which the largest index of an *intermediate node* is at most k



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= 8 \\
dist(i, j, 3) &= 5
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

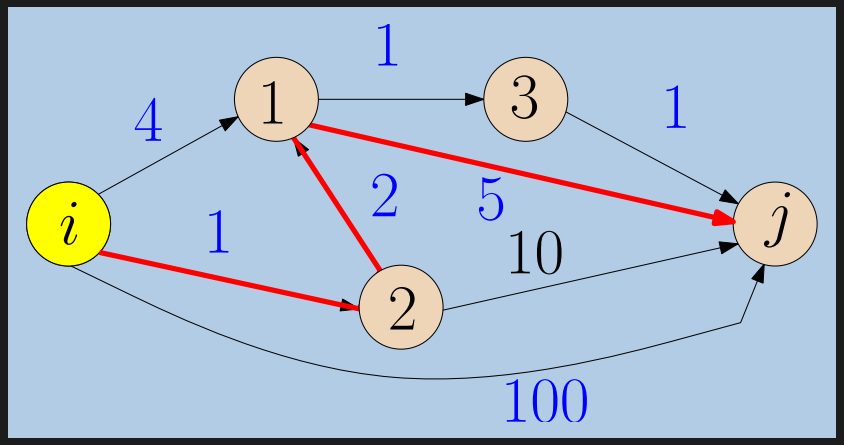
- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $dist(i, j, k)$: shortest path distance between v_i and v_j among all paths in which the largest index of an *intermediate node* is at most k



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= 8 \\
dist(i, j, 3) &= 5
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

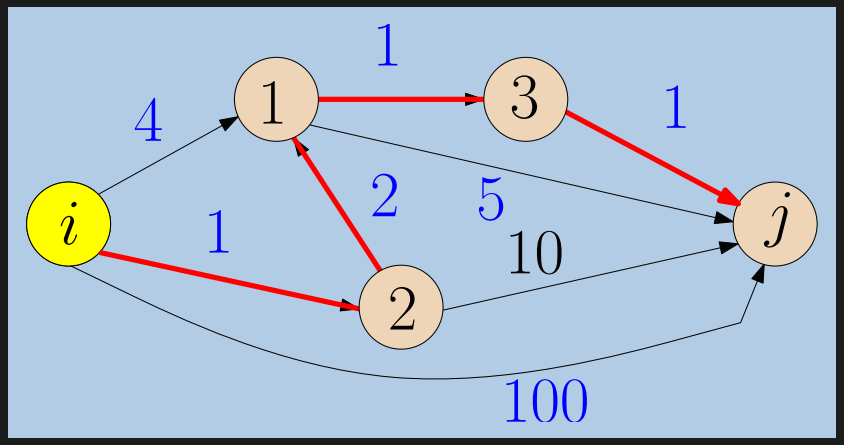
- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
- $dist(i, j, k)$: shortest path distance between v_i and v_j among all paths in which the largest index of an *intermediate node* is at most k



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= 8 \\
dist(i, j, 3) &= 5
\end{aligned}
$$

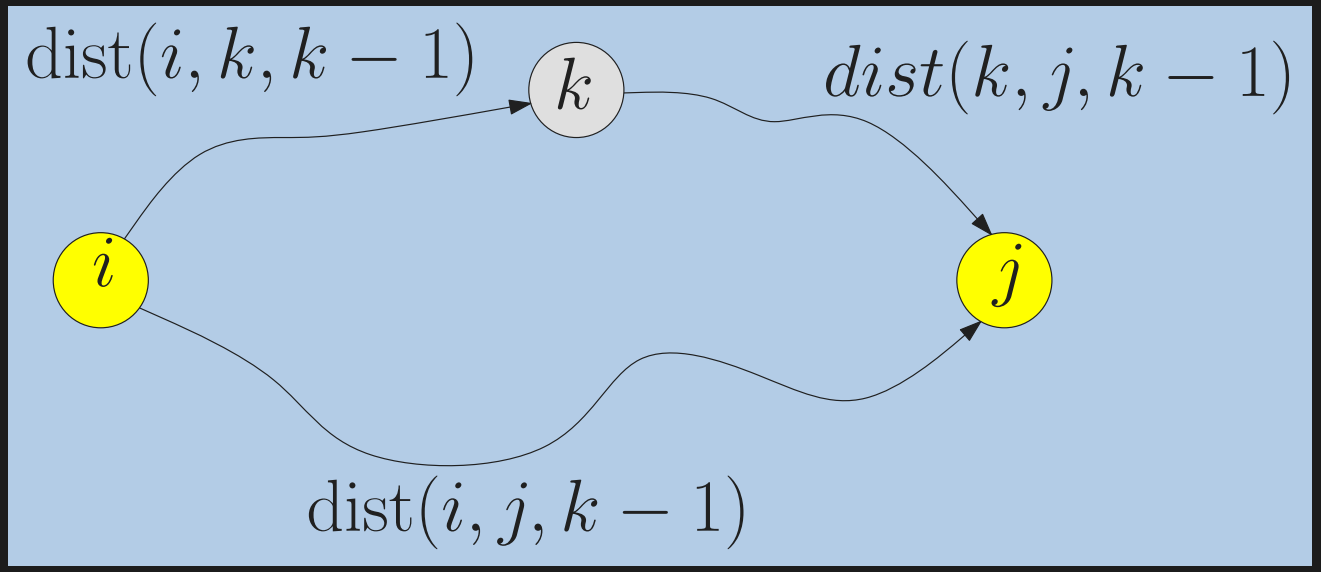
All-Pairs: Recursion on index of intermediate nodes

- 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



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= 8 \\
dist(i, j, 3) &= 5
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes



$$dist(i,j,k) = \min \begin{cases} dist(i,j,k-1) \\ dist(i,k,k-1) + dist(k,j,k-1) \end{cases}$$

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

Correctness: If $i \to j$ 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 // Bellman-Ford: $O(mn)$ time
if there is a negative cycle **then return** ``Negative cycle''

for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    $dist(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

for $k = 1$ to n **do**
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**

$$dist(i, j, k) = \min \begin{cases} dist(i, j, k-1), \\ dist(i, k, k-1) + dist(k, j, k-1) \end{cases}$$

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

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

Check if **G** has a negative cycle // Bellman-Ford: $O(mn)$ time
if there is a negative cycle **then return** ``Negative cycle''

for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    $dist(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

for $k = 1$ to n **do**
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**

$$dist(i, j, k) = \min \begin{cases} dist(i, j, k-1), \\ dist(i, k, k-1) + dist(k, j, k-1) \end{cases}$$

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

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
Check if G has a negative cycle // Bellman-Ford: O(mn) time
if there is a negative cycle then return ``Negative cycle''

for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = c(i,j) (* c(i,j) = ∞ if (i,j) ∉ E, 0 if 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: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    $dist(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, 0 if $i =$
not edge, 0 if $i = j$ *)

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

for $i = 1$ to n **do**
  if $(dist(i, i, n) < 0)$ **then**
    Output that there is a negative length cycle in **G**

Correctness: exercise

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    $dist(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, 0 if $i =$
not edge, 0 if $i = j$ *)

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

for $i = 1$ to n **do**
  if $(dist(i, i, n) < 0)$ **then**
    Output that there is a negative length cycle in **G**

Correctness: exercise

Floyd-Warshall Algorithm: Finding the Paths

1. **Question:** Can we find the paths in addition to the distances?
2. Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices
3. With array Next, for any pair of given vertices i, j can compute a shortest path in $O(n)$ time.

Floyd-Warshall Algorithm: Finding the Paths

1. **Question:** Can we find the paths in addition to the distances?
2. Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices
3. With array Next, for any pair of given vertices i, j can compute a shortest path in $O(n)$ time.

Floyd-Warshall Algorithm: Finding the Paths

1. **Question:** Can we find the paths in addition to the distances?
2. Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices
3. With array Next, for any pair of given vertices i, j can compute a shortest path in $O(n)$ time.

Floyd-Warshall Algorithm

Finding the Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = c(i,j) (* c(i,j) = ∞ if (i,j) not edge, 0 if
        Next(i,j) = -1
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if (dist(i,j,k-1) > dist(i,k,k-1) + dist(k,j,k-1))
                dist(i,j,k) = dist(i,k,k-1) + dist(k,j,k-1)
                Next(i,j) = k

for i = 1 to n do
    if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Exercise: Given $Next$ array and any two vertices i, j describe an $O(n)$ algorithm to find a i-j shortest path.

Summary of results on shortest paths

Single vertex		
No negative edges	Dijkstra	$O(n \log n + m$
Edges cost might be negative But no negative cycles	Bellman Ford	$O(nm)$

All Pairs Shortest Paths

No negative edges	n * Dijkstra	$O(n^2 \log n + nm)$
No negative cycles	n * Bellman Ford	$O(n^2 m) = O(n^4)$
No negative cycles	Floyd-Warshall	$O(n^3)$

Part VI

Knapsack

Knapsack Problem

Problem (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.*

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

Knapsack Problem

Problem (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.*

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

Knapsack Example

Example

Item	I_1	I_2	I_3	I_4	I_5
Value	1	6	18	22	28
Weight	1	2	5	6	7

If $W = 11$, the best is $\{I_3, I_4\}$ giving value 40.

Special Case

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

Greedy Approach

1. Pick objects with greatest value
   1. 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 $\{1, 2\}$
2. Pick objects with smallest weight
   1. 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 $\{2\}$
3. Pick objects with largest v_i/w_i ratio
   1. 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 $\{1, 2\}$
   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: $\mathbf{Opt}(i)$ 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 $\mathcal{O} - \{i\}$ 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 $\mathbf{Opt}(1), \ldots, \mathbf{Opt}(i-1)$.*

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

Towards a Recursive Solution

First guess: $\mathbf{Opt}(i)$ 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 $\mathcal{O} - \{i\}$ 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 $\mathbf{Opt}(1), \ldots, \mathbf{Opt}(i-1)$.

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

Towards a Recursive Solution

First guess: $\mathbf{Opt}(i)$ 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 $\mathcal{O} - \{i\}$ 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 $\mathbf{Opt}(1), \ldots, \mathbf{Opt}(i-1)$.

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

Dynamic Programming Solution

Definition

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

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

An Iterative Algorithm

```
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            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 $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but "pseudo-polynomial"!

An Iterative Algorithm

```
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            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 $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but "pseudo-polynomial"!

An Iterative Algorithm

```
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            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

1. Time taken is $O(nW)$
2. Input has size $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but "pseudo-polynomial"!

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. 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.
5. Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. 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.
5. Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. 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.
5. Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. 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.
5. Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. 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.
5. Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. 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.
5. Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. 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.
5. Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Part VII

Traveling Salesman Problem

Traveling Salesman Problem

Problem (TSP)

Input: *A graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *with non-negative edge costs/lengths. Cost* $c(e)$ *for each edge* $e \in \mathbf{E}$.
Goal: *Find a tour of minimum cost that visits each node.*

- No polynomial time algorithm known. Problem is **NP-Hard**.

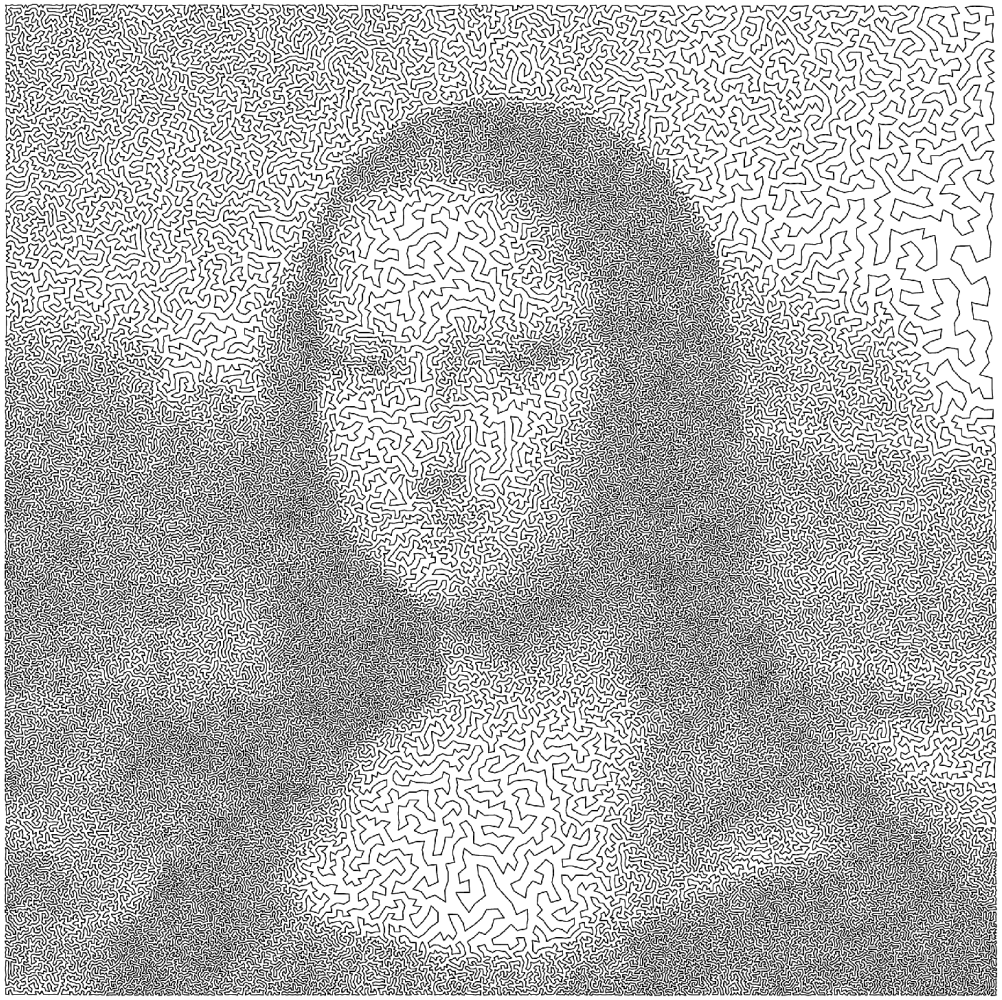
Traveling Salesman Problem

Problem (TSP)

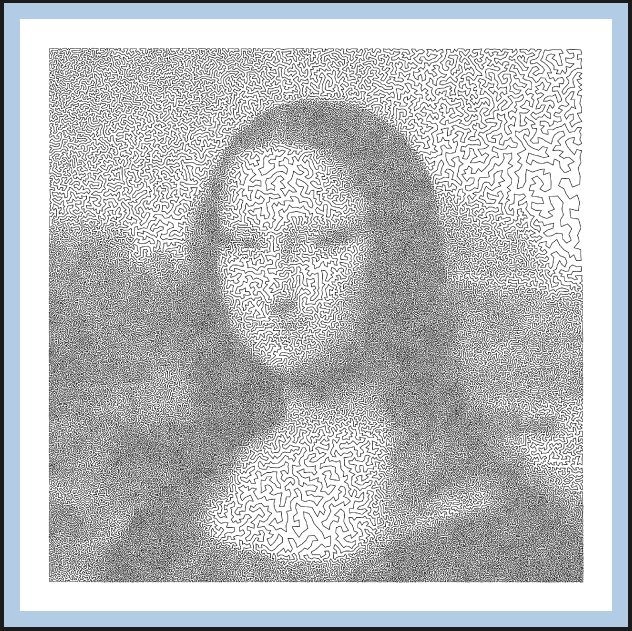
Input: *A graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *with non-negative edge costs/lengths. Cost* $c(e)$ *for each edge* $e \in \mathbf{E}$.
Goal: *Find a tour of minimum cost that visits each node.*

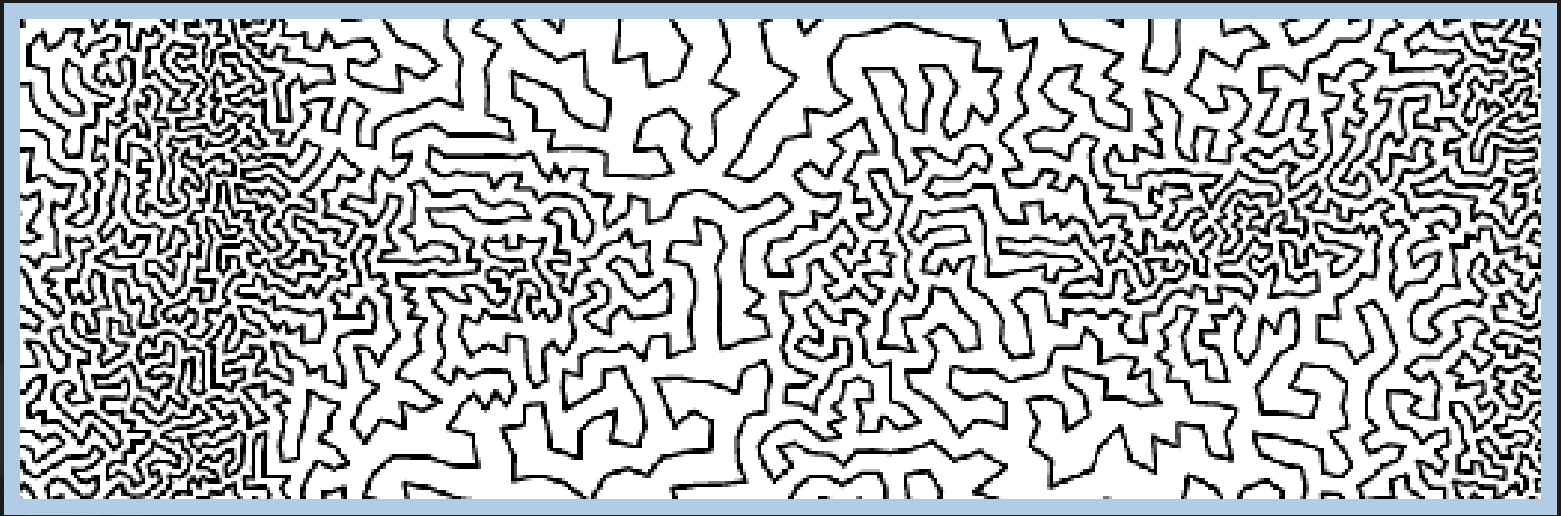
- No polynomial time algorithm known. Problem is **NP-Hard**.

Drawings using TSP

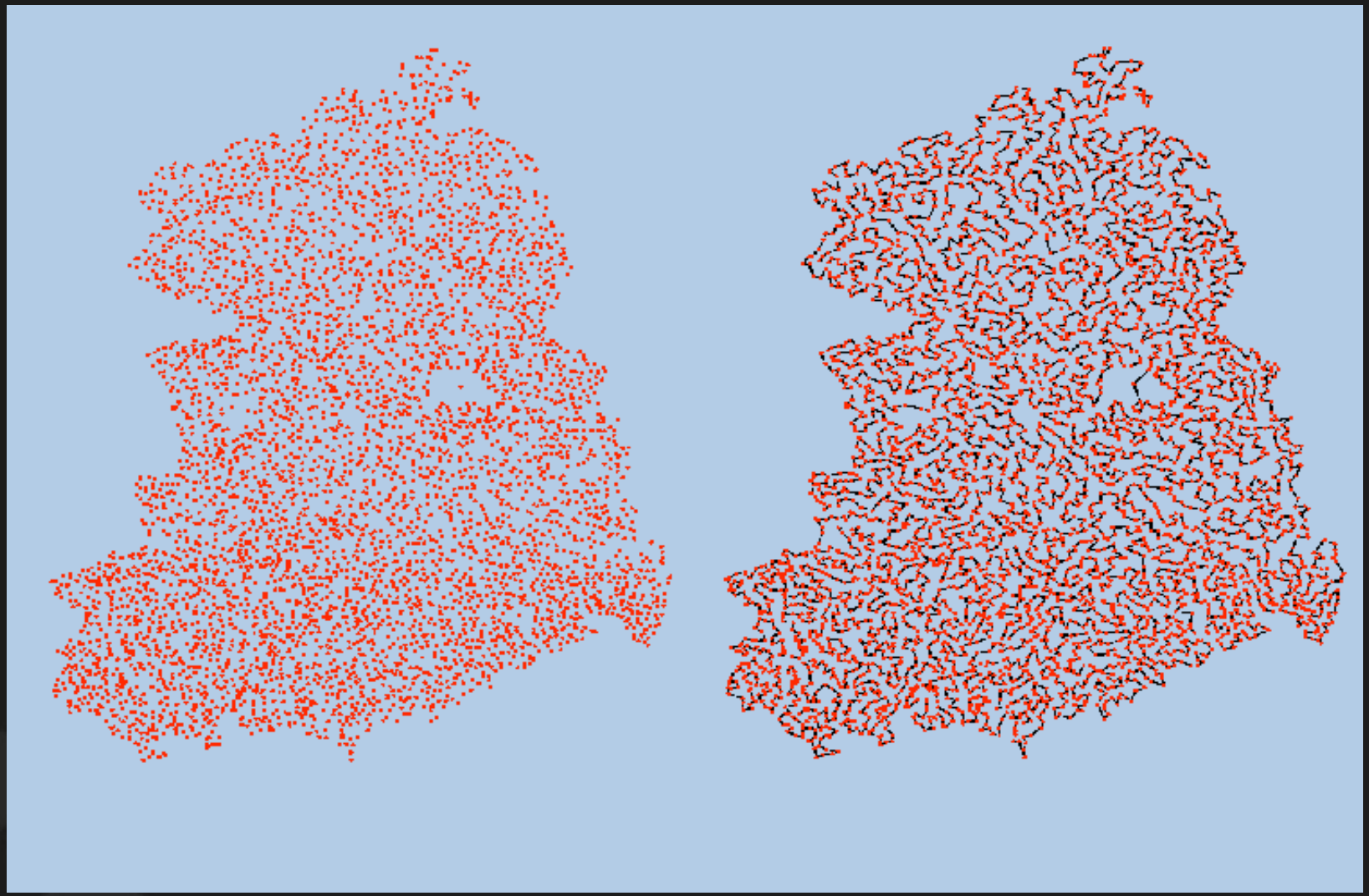


Drawings using TSP





Example: optimal tour for cities of a country (which one?)



An Exponential Time Algorithm

1. How many different tours are there? $n!$
2. Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$
3. Can we do better? Can we get a $2^{O(n)}$ time algorithm?

An Exponential Time Algorithm

1. How many different tours are there? $n!$
2. Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$
3. Can we do better? Can we get a $2^{O(n)}$ time algorithm?

An Exponential Time Algorithm

1. How many different tours are there? $n!$
2. Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$
3. Can we do better? Can we get a $2^{O(n)}$ time algorithm?

An Exponential Time Algorithm

- How many different tours are there? $n!$
- Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ 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?

An Exponential Time Algorithm

1. How many different tours are there? $n!$
2. Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$
3. 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?

- 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.

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.

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

Problem (TSP Path)

1. **Input:** *A graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *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

Problem (TSP Path)

1. **Input:** *A graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *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.*

2. Can solve TSP using above. Do you see how?
3. Recursion for optimum TSP Path problem:
   1. $OPT(u, v, S)$: optimum TSP Path from u to v in the graph restricted to S (here $u, v \in S$).

A More General Problem: TSP Path

Problem (TSP Path)

1. **Input:** *A graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *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.*

2. Can solve TSP using above. Do you see how?
3. Recursion for optimum TSP Path problem:
   1. $OPT(u, v, S)$: optimum TSP Path from u to v in the graph restricted to S (here $u, v \in S$).

A More General Problem: TSP Path

Problem (TSP Path)

1. **Input:** *A graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ *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.*

2. Can solve TSP using above. Do you see how?
3. Recursion for optimum TSP Path problem:
   1. $OPT(u, v, S)$: optimum TSP Path from u to v in the graph restricted to S (here $u, v \in S$).

A More General Problem: TSP Path

Continued...

- 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 More General Problem: TSP Path

Continued...

1. What is the next node in the optimum path from u to v?
2. Suppose it is w. Then what is $OPT(u, v, S)$?
3. $OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$
4. We do not know w! So try all possibilities for w.

A More General Problem: TSP Path

Continued...

1. What is the next node in the optimum path from u to v?
2. Suppose it is w. Then what is $OPT(u, v, S)$?
3. $OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$
4. We do not know w! So try all possibilities for w.

A More General Problem: TSP Path

Continued...

1. What is the next node in the optimum path from u to v?
2. Suppose it is w. Then what is $OPT(u, v, S)$?
3. $OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$
4. We do not know w! So try all possibilities for w.

A Recursive Solution

1. $OPT(u, v, S) = \min_{w \in S, w \neq u, v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$
2. What are the subproblems for the original problem $OPT(s, t, V)$?
   $OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.
3. How many subproblems?
   1. number of distinct subsets S of V is at most 2^n
   2. number of pairs of nodes in a set S is at most n^2
   3. hence number of subproblems is $O(n^2 2^n)$
4. **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!

A Recursive Solution

1. $OPT(u, v, S) = \min_{w \in S, w \neq u, v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$
2. What are the subproblems for the original problem $OPT(s, t, V)$?
   $OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.
3. How many subproblems?
   1. number of distinct subsets S of V is at most 2^n
   2. number of pairs of nodes in a set S is at most n^2
   3. hence number of subproblems is $O(n^2 2^n)$
4. **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!

A Recursive Solution

1. $OPT(u, v, S) = \min_{w \in S, w \neq u, v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$
2. What are the subproblems for the original problem $OPT(s, t, V)$?
   $OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.
3. How many subproblems?
   1. number of distinct subsets S of V is at most 2^n
   2. number of pairs of nodes in a set S is at most n^2
   3. hence number of subproblems is $O(n^2 2^n)$
4. **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!

A Recursive Solution

1. $OPT(u, v, S) = \min_{w \in S, w \neq u, v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$
2. What are the subproblems for the original problem $OPT(s, t, V)$?
   $OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.
3. How many subproblems?
   1. number of distinct subsets S of V is at most 2^n
   2. number of pairs of nodes in a set S is at most n^2
   3. hence number of subproblems is $O(n^2 2^n)$
4. **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!

A Recursive Solution

1. $OPT(u, v, S) = \min_{w \in S, w \neq u, v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$
2. What are the subproblems for the original problem $OPT(s, t, V)$?
   $OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.
3. How many subproblems?
   1. number of distinct subsets S of V is at most 2^n
   2. number of pairs of nodes in a set S is at most n^2
   3. hence number of subproblems is $O(n^2 2^n)$
4. **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!

A Recursive Solution

1. $OPT(u, v, S) = \min_{w \in S, w \neq u, v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$
2. What are the subproblems for the original problem $OPT(s, t, V)$?
   $OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.
3. How many subproblems?
   1. number of distinct subsets S of V is at most 2^n
   2. number of pairs of nodes in a set S is at most n^2
   3. hence number of subproblems is $O(n^2 2^n)$
4. **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!

A Recursive Solution

1. $OPT(u, v, S) = \min_{w \in S, w \neq u,v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$
2. What are the subproblems for the original problem $OPT(s, t, V)$?
   $OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.
3. How many subproblems?
   1. number of distinct subsets S of V is at most 2^n
   2. number of pairs of nodes in a set S is at most n^2
   3. hence number of subproblems is $O(n^2 2^n)$
4. **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: Postscript

Dynamic Programming = Smart Recursion + Memoization

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

Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?

Some Tips

1. 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.
2. Problems involving trees: recursion based on subtrees.
3. More generally:
   1. Problem admits a natural recursive divide and conquer
   2. If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
   3. If optimal solution depends on all pieces then can apply dynamic programming if *interface/interaction* between pieces is *limited*. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.

Examples

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

Notes

Notes

Notes

Notes