



Design and analysis of algorithms

Lecture 23& 24

Edyta Szymańska

edyta@cc.gatech.edu

Set cover

yet another application of the greedy approach

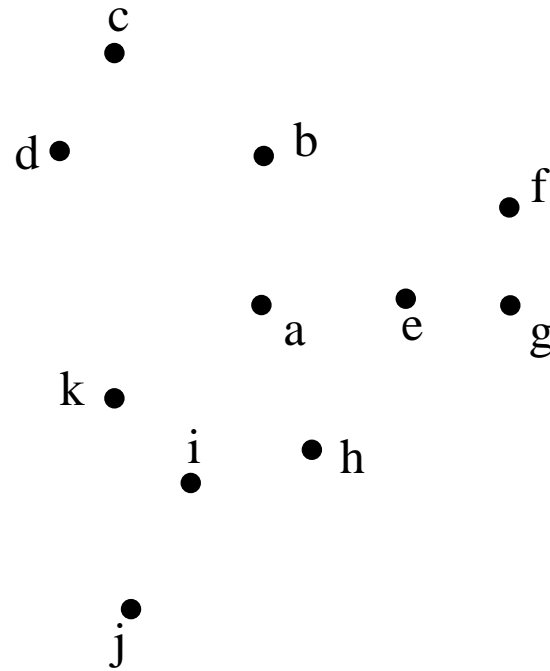
Set cover



Example: County map with a collection of towns

Set cover

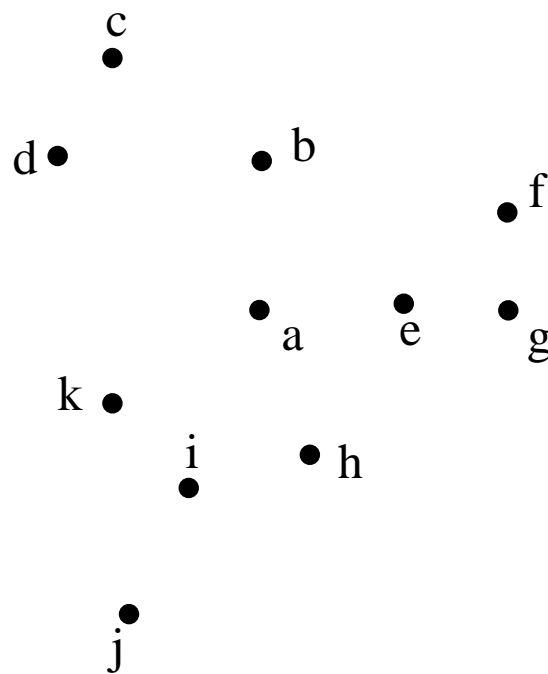
Example: County map with a collection of towns



Set cover

Example: County map with a collection of towns

Question: where to put schools?



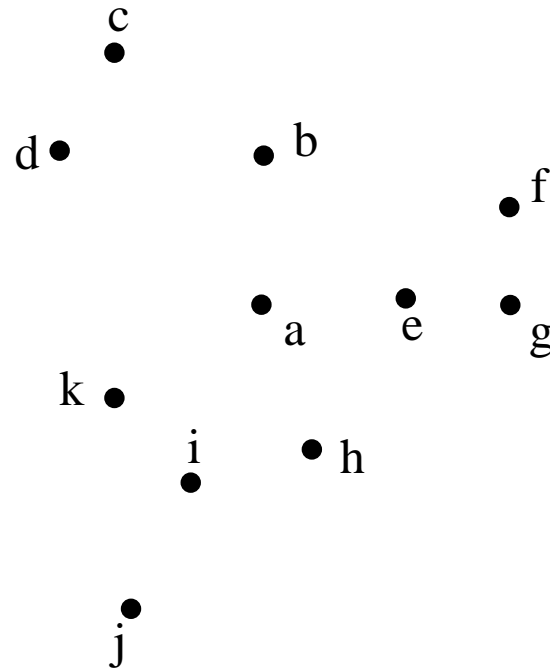
Set cover

Example: County map with a collection of towns

Question: where to put schools?

Constraints:

- ⑥ *each school should be in a town*
- ⑥ *no one should have to travel more than 30 miles to school*
- ⑥ *model: graph with an edge (u, v) if $d(u, v) \leq 30$*



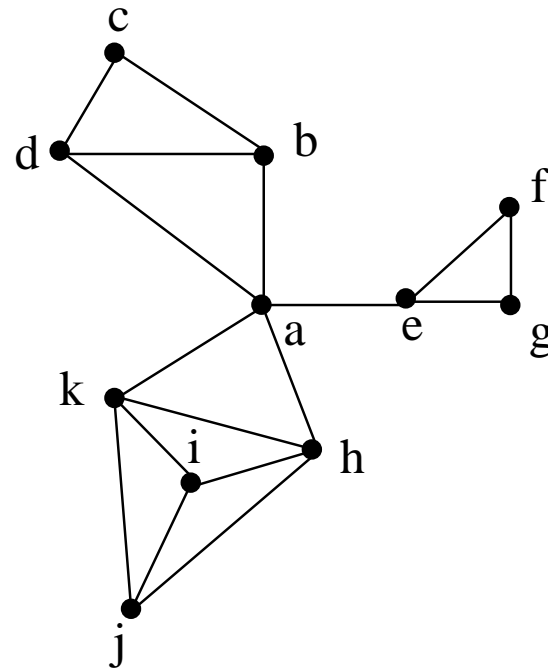
Set cover

Example: County map with a collection of towns

Question: where to put schools?

Constraints:

- ⑥ *each school should be in a town*
- ⑥ *no one should have to travel more than 30 miles to school*
- ⑥ *model: graph with an edge (u, v) if $d(u, v) \leq 30$*



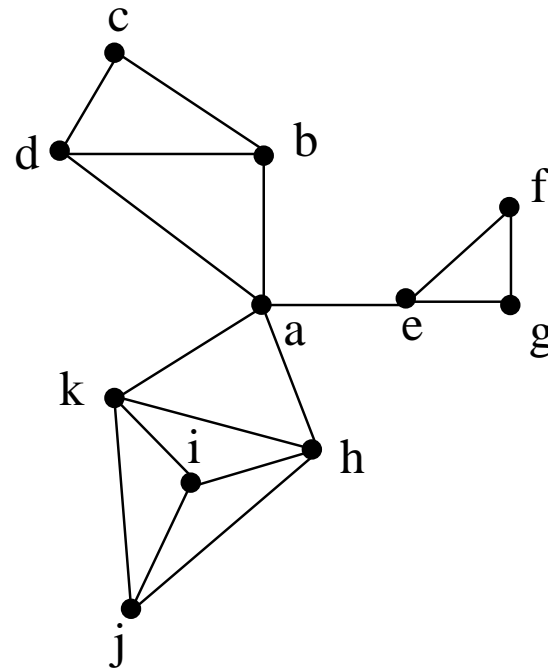
Set cover

Example: County map with a collection of towns

Question: where to put schools?

Constraints:

- ⑥ *each school should be in a town*
- ⑥ *no one should have to travel more than 30 miles to school*
- ⑥ *model: graph with an edge (u, v) if $d(u, v) \leq 30$*



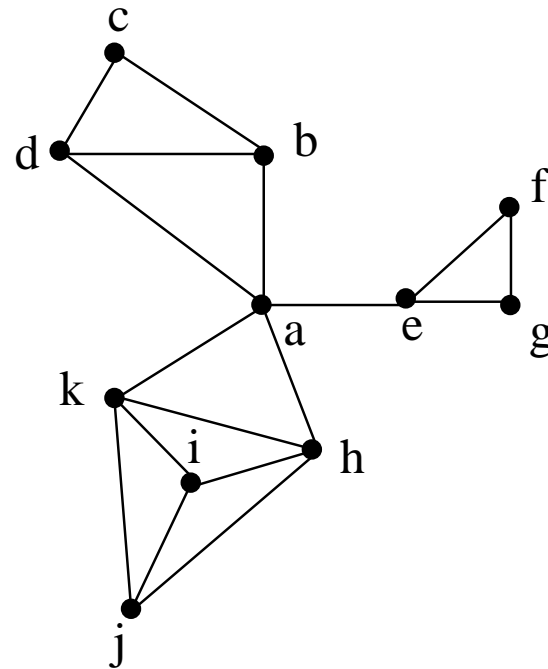
Set cover

Example: County map with a collection of towns

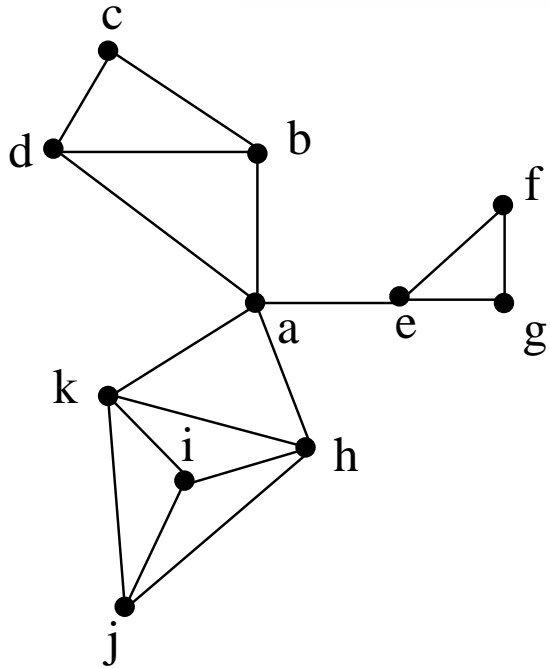
Question: where to put schools?

Constraints:

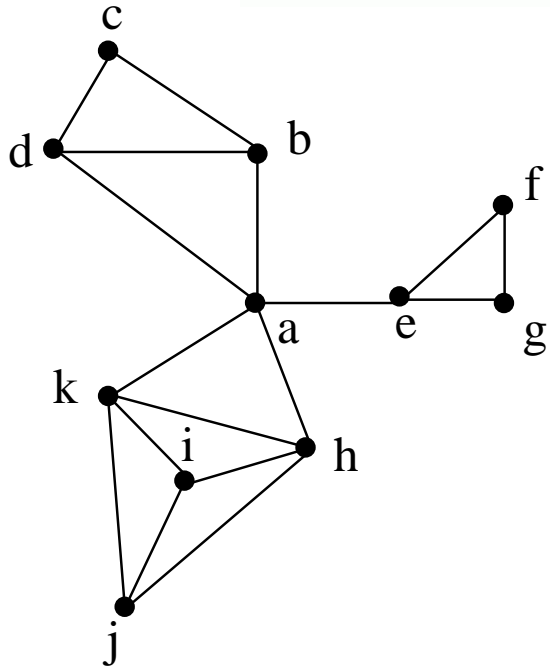
- ⑥ *each school should be in a town*
- ⑥ *no one should have to travel more than 30 miles to school*
- ⑥ *model: graph with an edge (u, v) if $d(u, v) \leq 30$*



Set cover

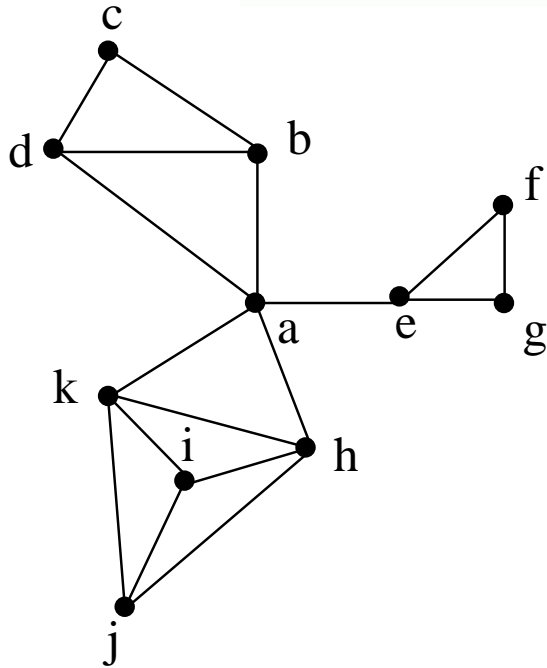


Set cover



Problem: what is the minimum number of schools needed ?

Set cover



Problem: what is the minimum number of schools needed ?
For every town x , let S_x be the set of towns within 30 miles of x . (a school in x "covers" every town in S_x .)
How many sets S_x are needed to cover all towns ?

Set Cover Problem - formal statement

INPUT: a set B of
 n elements, sets
 $S_1, S_2, \dots, S_m \subseteq B$

Set Cover Problem - formal statement

INPUT: a set B of
 n elements, sets
 $S_1, S_2, \dots, S_m \subseteq B$
OUTPUT: a collection of sets
 S_i , whose union is B .

Set Cover Problem - formal statement

INPUT: a set B of
 n elements, sets
 $S_1, S_2, \dots, S_m \subseteq B$
OUTPUT: a collection of sets
 S_i , whose union is B .
COST: number of sets
picked

Set Cover Problem - formal statement

INPUT: a set B of
 n elements, sets

$S_1, S_2, \dots, S_m \subseteq B$

OUTPUT: a collection of sets
 S_i , whose union is B .

COST: number of sets
picked

Natural approach: GREEDY
METHOD

Set Cover Problem - formal statement

INPUT: a set B of n elements, sets

$S_1, S_2, \dots, S_m \subseteq B$

OUTPUT: a collection of sets S_i , whose union is B .

COST: number of sets picked

Natural approach: GREEDY METHOD

Pick the set S_i with the largest number of uncovered elements

Set Cover Problem - formal statement

INPUT: a set B of n elements, sets

$S_1, S_2, \dots, S_m \subseteq B$

OUTPUT: a collection of sets S_i , whose union is B .

COST: number of sets picked

Natural approach: GREEDY METHOD

Repeat until all elements of B are covered

Pick the set S_i with the largest number of uncovered elements

Set Cover Problem - formal statement

INPUT: a set B of n elements, sets $S_1, S_2, \dots, S_m \subseteq B$

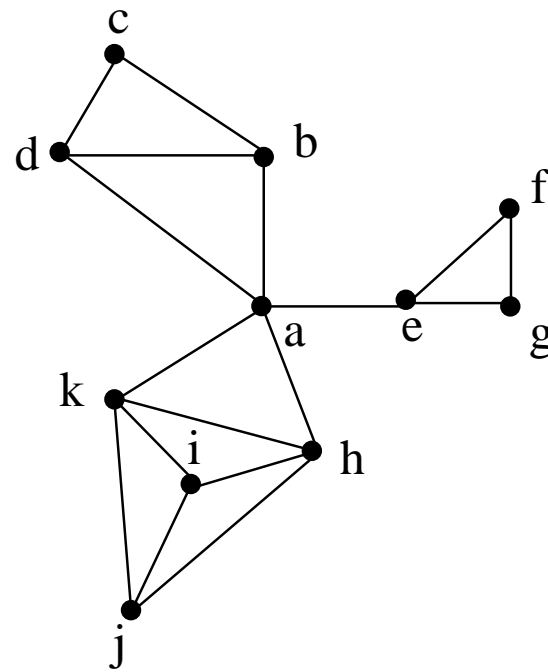
OUTPUT: a collection of sets S_i , whose union is B .

COST: number of sets picked

Natural approach: GREEDY METHOD

Repeat until all elements of B are covered

Pick the set S_i with the largest number of uncovered elements



Set Cover Problem - formal statement

INPUT: a set B of n elements, sets $S_1, S_2, \dots, S_m \subseteq B$

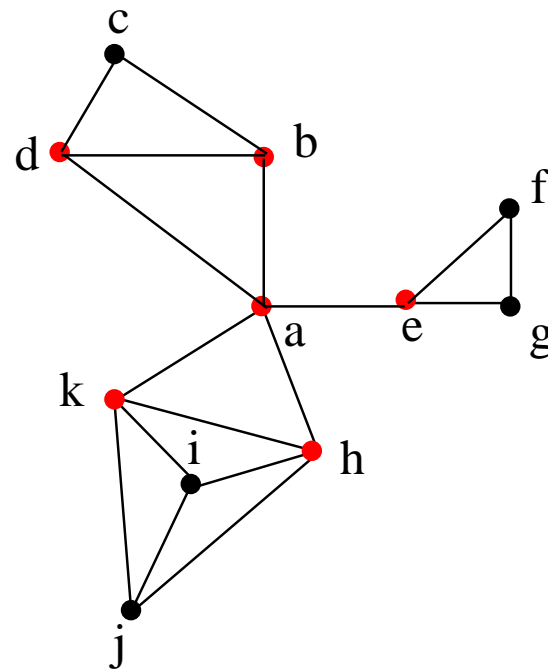
OUTPUT: a collection of sets S_i , whose union is B .

COST: number of sets picked

Natural approach: GREEDY METHOD

Repeat until all elements of B are covered

Pick the set S_i with the largest number of uncovered elements



Set Cover Problem - formal statement

INPUT: a set B of n elements, sets $S_1, S_2, \dots, S_m \subseteq B$

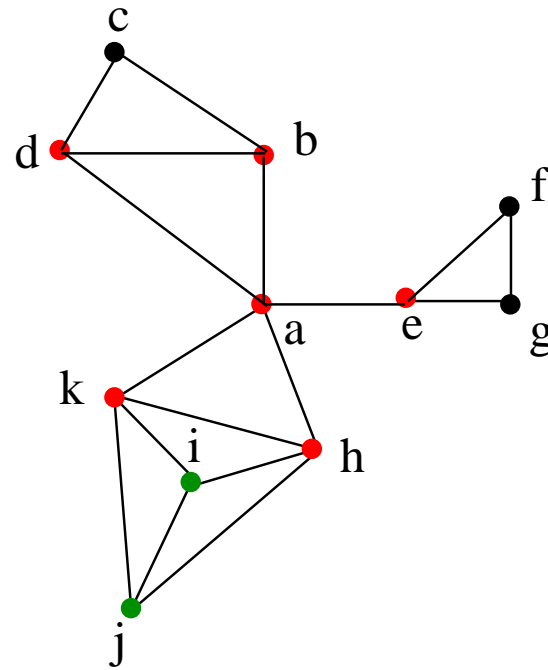
OUTPUT: a collection of sets S_i , whose union is B .

COST: number of sets picked

Natural approach: GREEDY METHOD

Repeat until all elements of B are covered

Pick the set S_i with the largest number of uncovered elements



Set Cover Problem - formal statement

INPUT: a set B of n elements, sets $S_1, S_2, \dots, S_m \subseteq B$

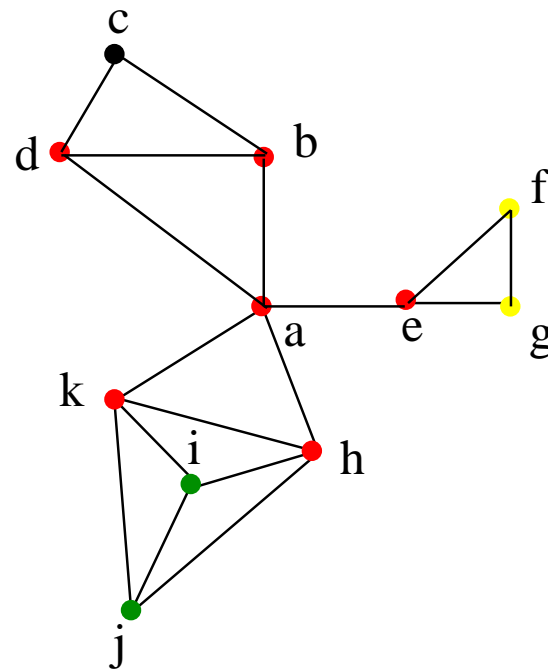
OUTPUT: a collection of sets S_i , whose union is B .

COST: number of sets picked

Natural approach: GREEDY METHOD

Repeat until all elements of B are covered

Pick the set S_i with the largest number of uncovered elements



Set Cover Problem - formal statement

INPUT: a set B of n elements, sets $S_1, S_2, \dots, S_m \subseteq B$

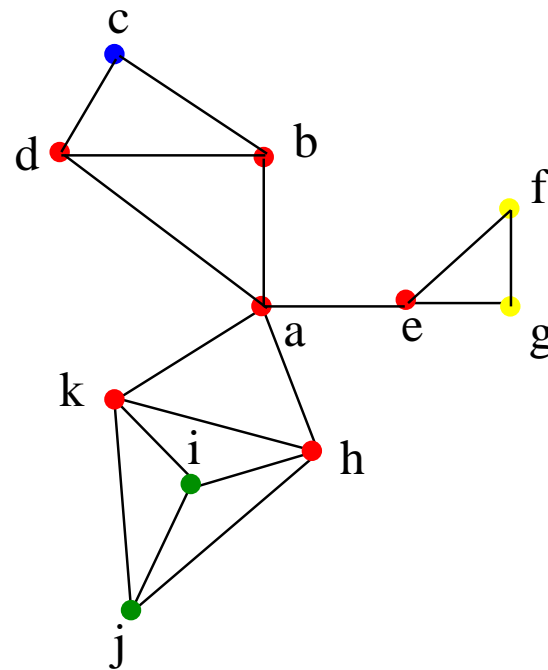
OUTPUT: a collection of sets S_i , whose union is B .

COST: number of sets picked

Natural approach: GREEDY METHOD

Repeat until all elements of B are covered

Pick the set S_i with the largest number of uncovered elements

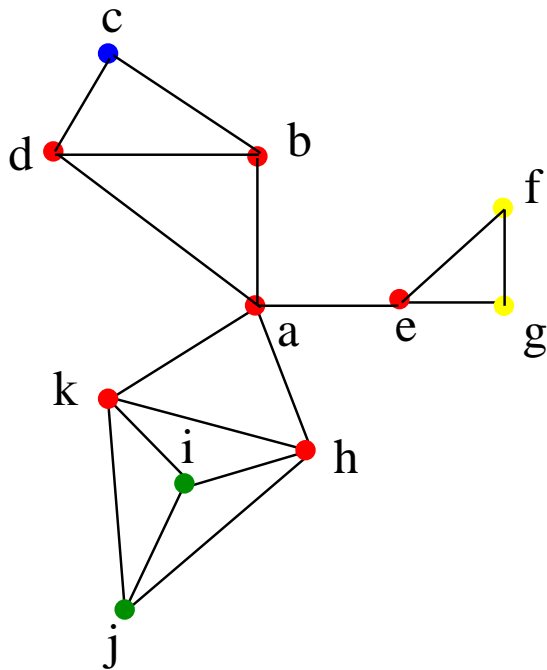


Set Cover Problem - greedy fails

In our example greedy does NOT find the optimal solution.

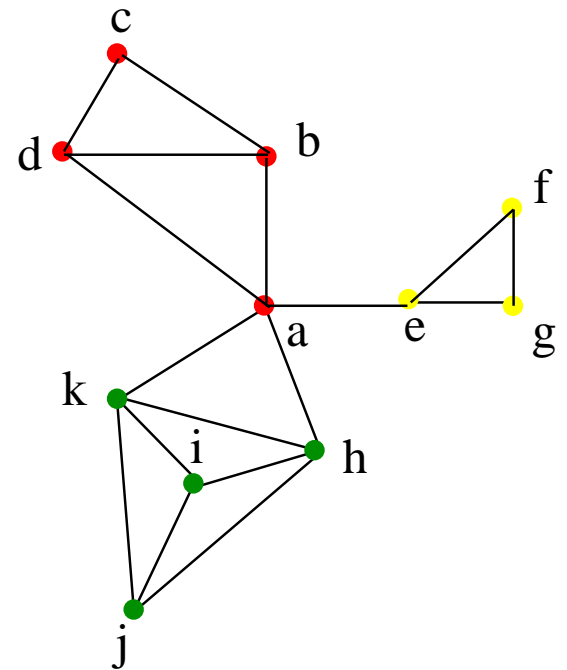
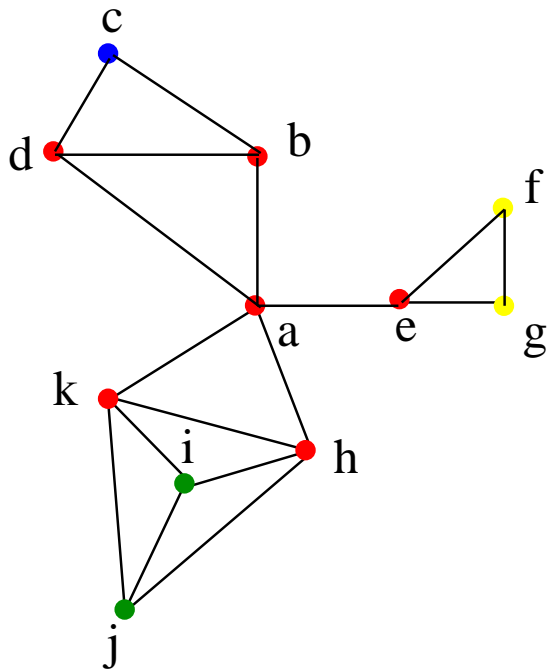
Set Cover Problem - greedy fails

In our example greedy does NOT find the optimal solution.



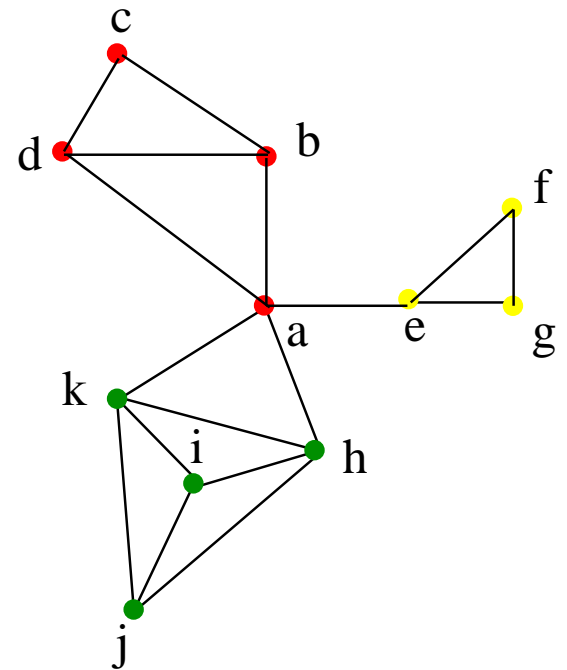
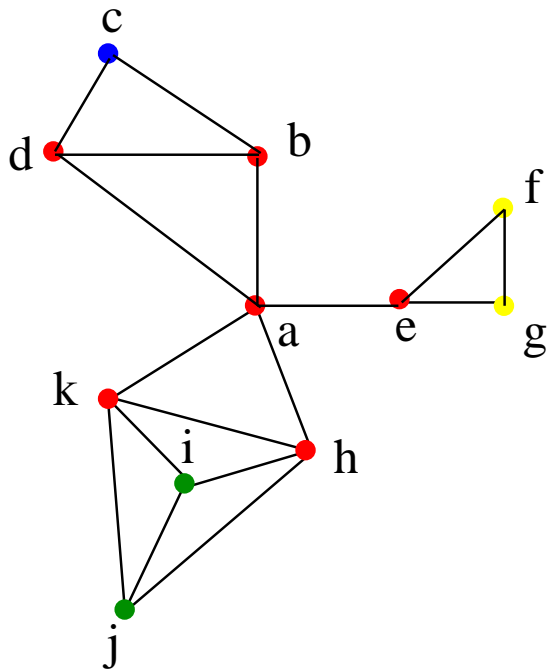
Set Cover Problem - greedy fails

In our example greedy does NOT find the optimal solution.



Set Cover Problem - greedy fails

In our example greedy does NOT find the optimal solution.



We do not eliminate the greedy method completely - it gives a solutions which is pretty close to the optimum in this case.

Set Cover Problem - greedy approximation

Claim *Let B contain n elements and the optimal cover consist of k sets. Then the greedy approach will use at most $k \ln n$ sets.*

Set Cover Problem - greedy approximation

Claim *Let B contain n elements and the optimal cover consist of k sets. Then the greedy approach will use at most $k \ln n$ sets.*

Proof: Let n_t be the number of elements still not covered after t iterations of the greedy algorithm. Then $n_0 = n$ and $n_{t+1} \leq n_t - \frac{n_t}{k}$.

This is because the optimal solution covers the n_t uncovered elements with k sets and (by averaging, pigeon-hole principle) there must exist a set which covers at least $\frac{n_t}{k}$ of them.

Set Cover Problem - greedy approximation

Claim *Let B contain n elements and the optimal cover consist of k sets. Then the greedy approach will use at most $k \ln n$ sets.*

Proof: Let n_t be the number of elements still not covered after t iterations of the greedy algorithm. Then $n_0 = n$ and $n_{t+1} \leq n_t - \frac{n_t}{k}$.

This is because the optimal solution covers the n_t uncovered elements with k sets and (by averaging, pigeon-hole principle) there must exist a set which covers at least $\frac{n_t}{k}$ of them. After unrolling the recurrence we get:

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t.$$

Set Cover Problem - greedy approximation

Claim Let B contain n elements and the optimal cover consist of k sets. Then the greedy approach will use at most $k \ln n$ sets.

Proof: Let n_t be the number of elements still not covered after t iterations of the greedy algorithm. Then $n_0 = n$ and $n_{t+1} \leq n_t - \frac{n_t}{k}$.

This is because the optimal solution covers the n_t uncovered elements with k sets and (by averaging, pigeon-hole principle) there must exist a set which covers at least $\frac{n_t}{k}$ of them. After unrolling the recurrence we get:

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t.$$

Now, applying $1 - x \leq e^{-x}$, $x > 0$ gives $n_t \leq n e^{-\frac{t}{k}}$ which is less than 1 when $t := k \ln n$.

Set Cover Problem - greedy approximation

Claim Let B contain n elements and the optimal cover consist of k sets. Then the greedy approach will use at most $k \ln n$ sets.

Proof: Let n_t be the number of elements still not covered after t iterations of the greedy algorithm. Then $n_0 = n$ and $n_{t+1} \leq n_t - \frac{n_t}{k}$.

This is because the optimal solution covers the n_t uncovered elements with k sets and (by averaging, pigeon-hole principle) there must exist a set which covers at least $\frac{n_t}{k}$ of them. After unrolling the recurrence we get:

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t.$$

Now, applying $1 - x \leq e^{-x}$, $x > 0$ gives $n_t \leq n e^{-\frac{t}{k}}$ which is less than 1 when $t := k \ln n$.

Greedy approximation of Set Cover

We say that the greedy algorithm has an **approximation ratio of $\ln n$** , the maximum ratio between its solution and the optimal solution.

Greedy approximation of Set Cover

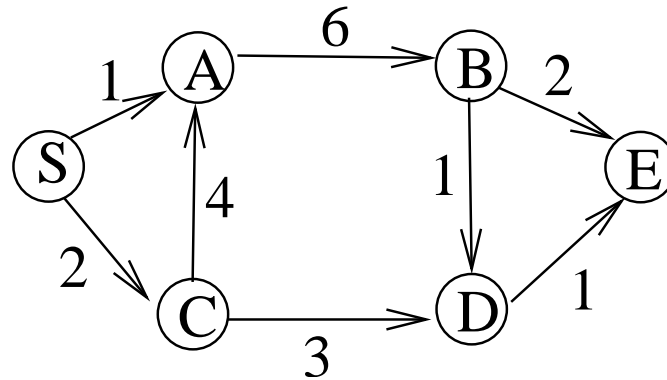
We say that the greedy algorithm has an **approximation ratio of $\ln n$** , the maximum ratio between its solution and the optimal solution.
Apparently, it is the best possible algorithm for this problem.

Dynamic programming - introduction

1. Shortest paths revisited

Dynamic programming - introduction

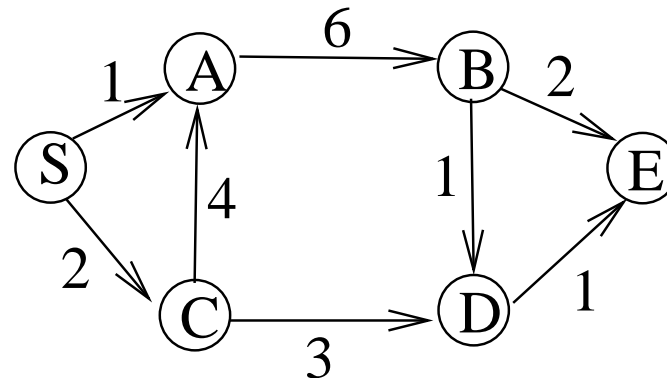
1. Shortest paths revisited



Shortest paths in a DAG

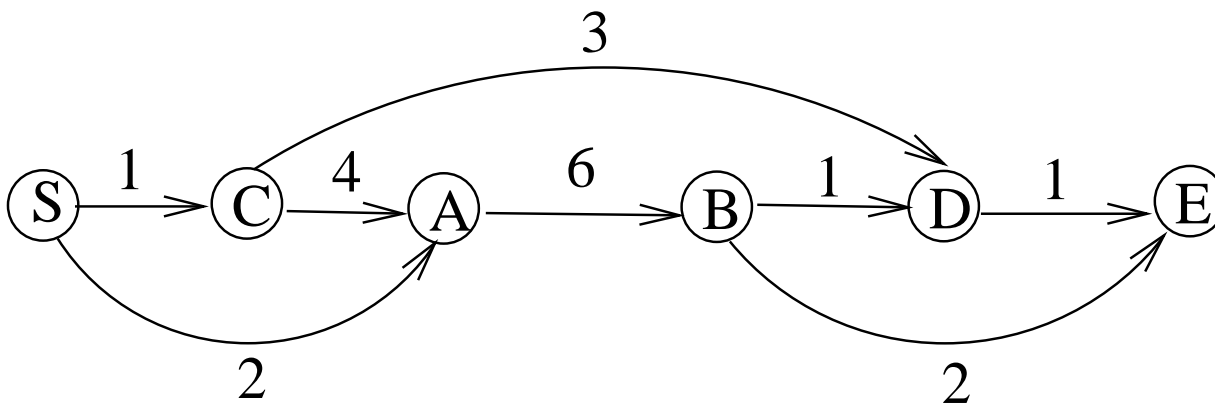
Dynamic programming - introduction

1. Shortest paths revisited



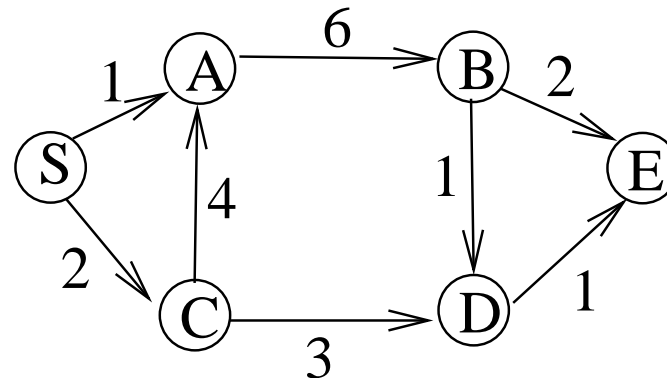
Shortest paths in a DAG

- *every dag can be topologically sorted*



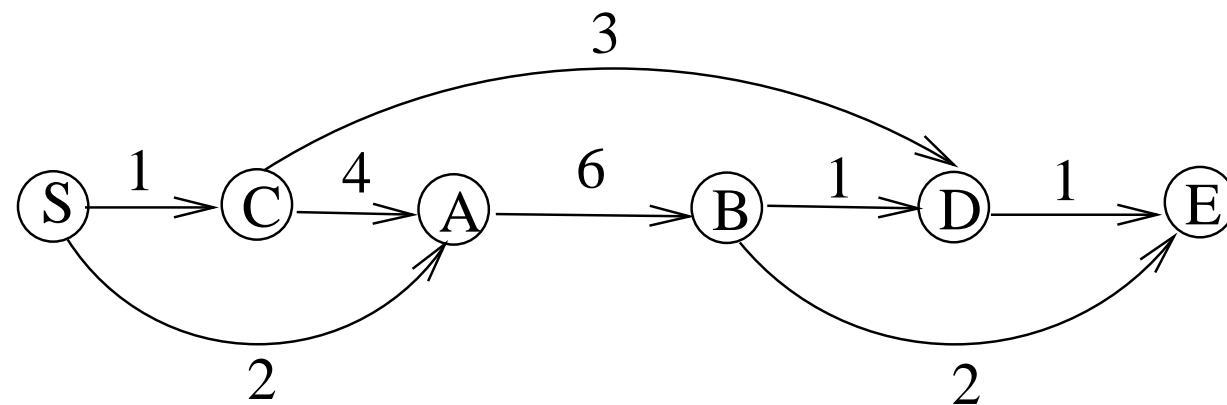
Dynamic programming - introduction

1. Shortest paths revisited



Shortest paths in a DAG

- every dag can be topologically sorted



Observation 1: $d(S, D) = \min\{d(S, C) + 3, d(S, B) + 1\}$,
where B, C are predecessors of D in the above order

Dynamic programming - introduction

If we consider the vertices in the topological order then all distances can be computed in a single pass:

Dynamic programming - introduction

If we consider the vertices in the topological order then all distances can be computed in a single pass:

$\text{PATHS_DAG}(G, w, s)$

for all $v \in V$ do $d[v] := \infty$

$d[s] := 0$

for each $v \in V \setminus \{s\}$, in topological order do

$d[v] := \min_{(u,v) \in E} \{d[u] + w(u, v)\}$

Dynamic programming - introduction

If we consider the vertices in the topological order then all distances can be computed in a single pass:

$\text{PATHS_DAG}(G, w, s)$

for all $v \in V$ do $d[v] := \infty$

$d[s] := 0$

for each $v \in V \setminus \{s\}$, in topological order do

$d[v] := \min_{(u,v) \in E} \{d[u] + w(u, v)\}$

Observation 2: the algorithm is solving subproblems $\{d[u]; u \in V\}$.

Dynamic programming - introduction



Dynamic programming - introduction

This is a general strategy which lies at the heart of the so called **dynamic programming method**

Dynamic programming - introduction

This is a general strategy which lies at the heart of the so called **dynamic programming method**

Dynamic programming

- ⑥ is an algorithmic paradigm
- ⑥ identify a collection of subproblems
- ⑥ solve the subproblems one-by one, smallest first and using the solutions to small problems to compute the larger ones
- ⑥ requires a lot of practise

Dynamic programming - introduction

This is a general strategy which lies at the heart of the so called **dynamic programming method**

Dynamic programming

- ⑥ is an algorithmic paradigm
- ⑥ identify a collection of subproblems
- ⑥ solve the subproblems one-by one, smallest first and using the solutions to small problems to compute the larger ones
- ⑥ requires a lot of practise

see example

Dynamic programming - introduction

S	A	B	C	D	E
0	∞	∞	∞	∞	∞
	$\min\{2, 4\} = 2$	8	1	$\min\{9, 4\} = 4$	$\min\{10, 5\} = 5$
0	2	8	1	4	5

Dynamic programming - introduction

S	A	B	C	D	E
0	∞	∞	∞	∞	∞
	$\min\{2, 4\} = 2$	8	1	$\min\{9, 4\} = 4$	$\min\{10, 5\} = 5$
0	2	8	1	4	5

How to reconstruct the shortest paths ?

Longest increasing subsequences



INPUT: a sequence a of n numbers a_1, a_2, \dots, a_n

Longest increasing subsequences

INPUT: a sequence a of n numbers a_1, a_2, \dots, a_n

OUTPUT: a subsequence of a of the form $a_{i_1}, a_{i_2}, \dots, a_{i_k}$,
where $1 \leq i_1 < i_2 < \dots < i_k \leq n$, the numbers are strictly
increasing and k is the largest possible

Longest increasing subsequences

INPUT: a sequence a of n numbers a_1, a_2, \dots, a_n

OUTPUT: a subsequence of a of the form $a_{i_1}, a_{i_2}, \dots, a_{i_k}$,
where $1 \leq i_1 < i_2 < \dots < i_k \leq n$, the numbers are strictly
increasing and k is the largest possible

Example:

given sequence: 5, 2, 8, 6, 3, 6, 9, 7

the longest increasing subsequence: 2, 3, 6, 9

Longest increasing subsequences

INPUT: a sequence a of n numbers a_1, a_2, \dots, a_n

OUTPUT: a subsequence of a of the form $a_{i_1}, a_{i_2}, \dots, a_{i_k}$,
where $1 \leq i_1 < i_2 < \dots < i_k \leq n$, the numbers are strictly
increasing and k is the largest possible

Example:

given sequence: 5, 2, 8, 6, 3, 6, 9, 7

the longest increasing subsequence: 2, 3, 6, 9

Graph representation of the problem:

$$G = (V, E), V = \{i, a_i\}, E = \{(i, j) : i < j \ \& \ a_i < a_j\}$$

Longest increasing subsequences

INPUT: a sequence a of n numbers a_1, a_2, \dots, a_n

OUTPUT: a subsequence of a of the form $a_{i_1}, a_{i_2}, \dots, a_{i_k}$, where $1 \leq i_1 < i_2 < \dots < i_k \leq n$, the numbers are strictly increasing and k is the largest possible

Example:

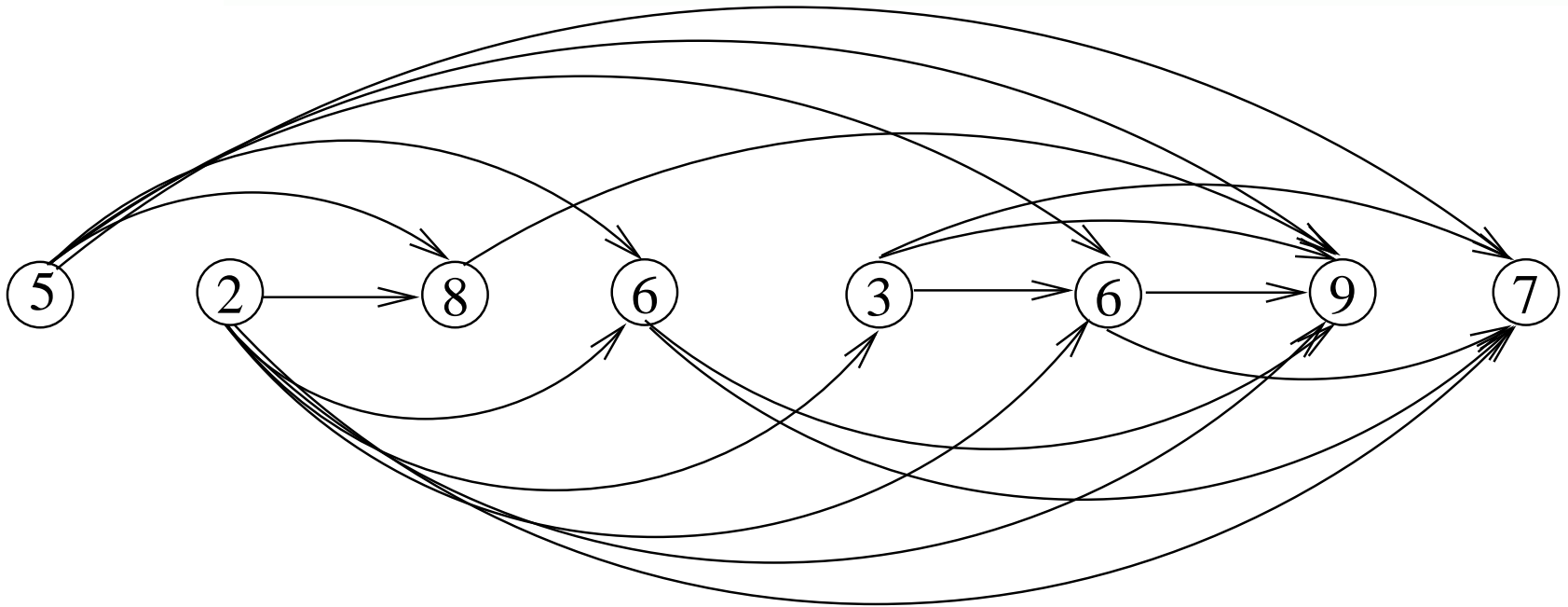
given sequence: 5, 2, 8, 6, 3, 6, 9, 7

the longest increasing subsequence: 2, 3, 6, 9

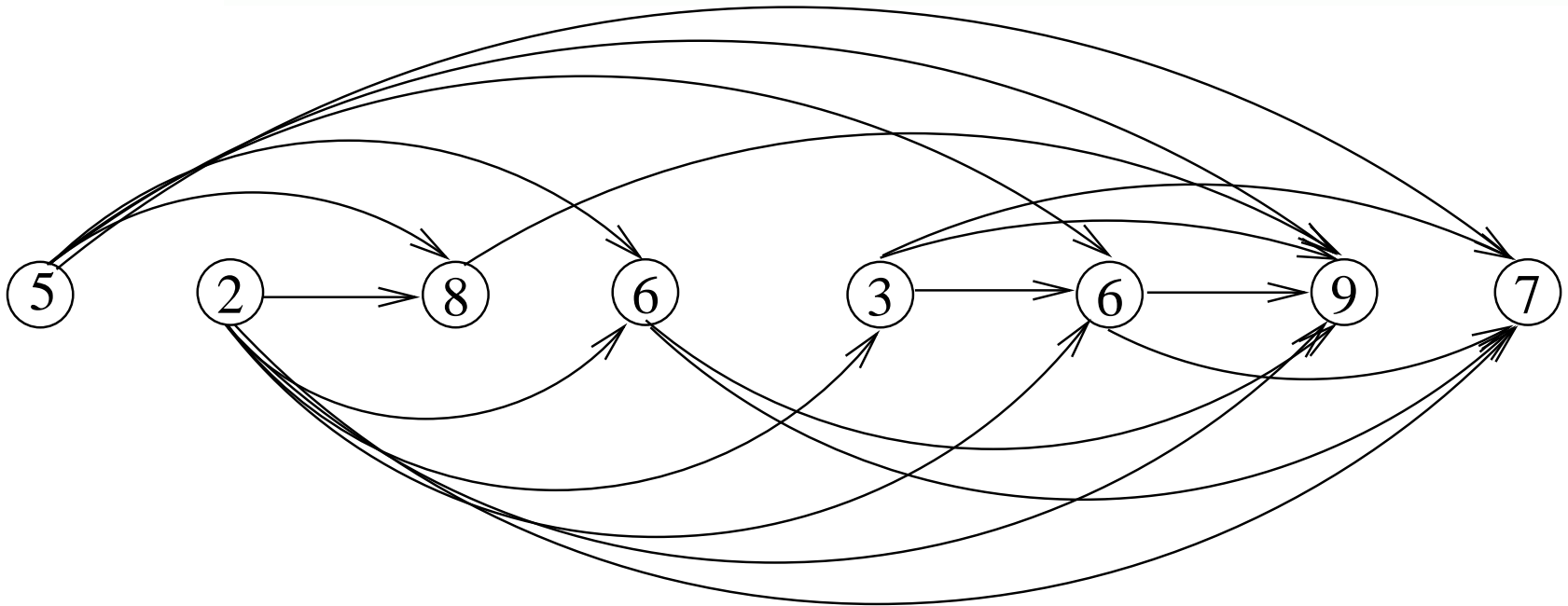
Graph representation of the problem:

$$G = (V, E), V = \{i, a_i\}, E = \{(i, j) : i < j \ \& \ a_i < a_j\}$$

Longest increasing subsequences



Longest increasing subsequences



- ⑥ $G = (V, E)$ is a DAG
- ⑥ there is one-to-one correspondence between increasing subsequences and paths in this dag
- ⑥ Goal: find the longest path in the dag

A dynamic programming solution

```
for  $j = 1$  to  $n$  do  
     $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$   
return  $\max_j L(j)$ 
```

A dynamic programming solution

```
for  $j = 1$  to  $n$  do
```

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$

```
return  $\max_j L(j)$ 
```

Time:

To compute $L(j)$, the predecessors of j are needed (we can get it from G^R in linear time.)

Total time $O(|E|)$.

Recursion? No, thanks.

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$

Recursion? No, thanks.

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$

What if we use a recursion algorithm for computing $L(j)$?

Recursion? No, thanks.

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$

What if we use a recursion algorithm for computing $L(j)$?

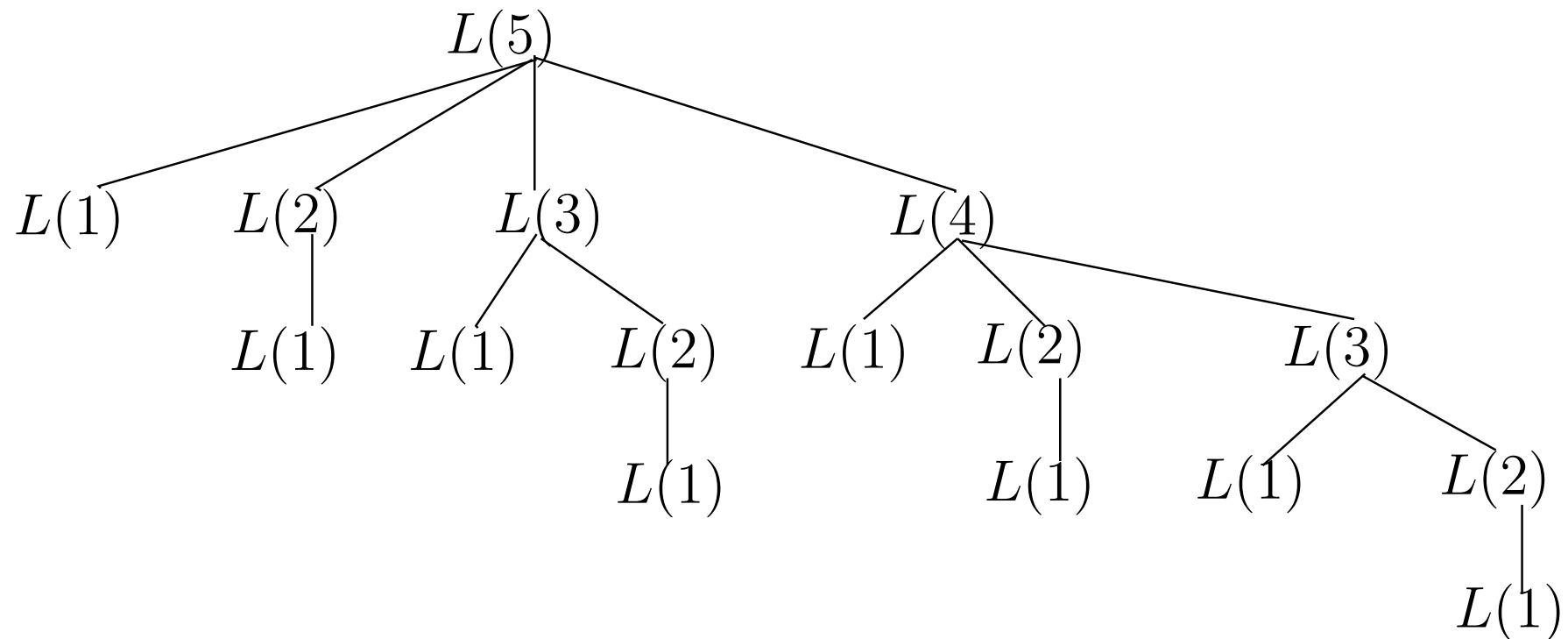
The resulting procedure would require exponential time!

Recursion? No, thanks.

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$

What if we use a recursion algorithm for computing $L(j)$?

The resulting procedure would require exponential time!

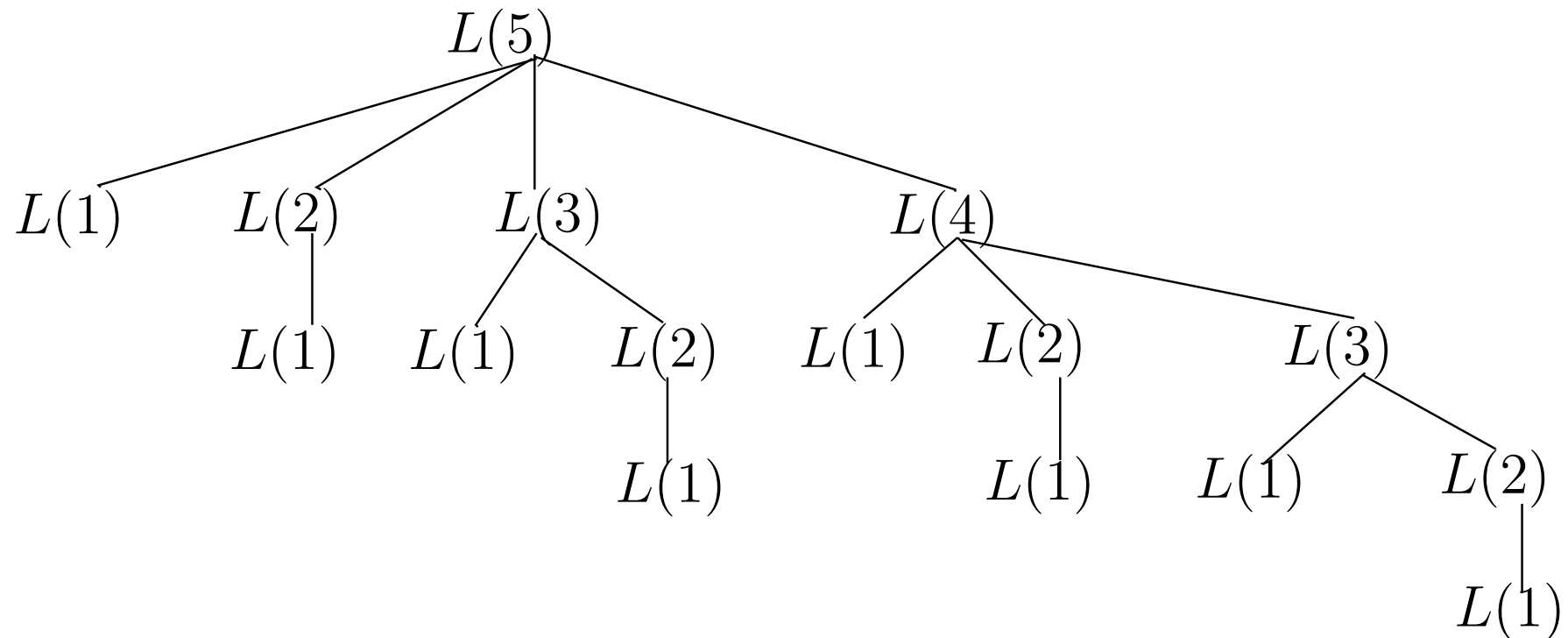


Recursion? No, thanks.

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$

What if we use a recursion algorithm for computing $L(j)$?

The resulting procedure would require exponential time!



Many subproblems get solved over and over again!