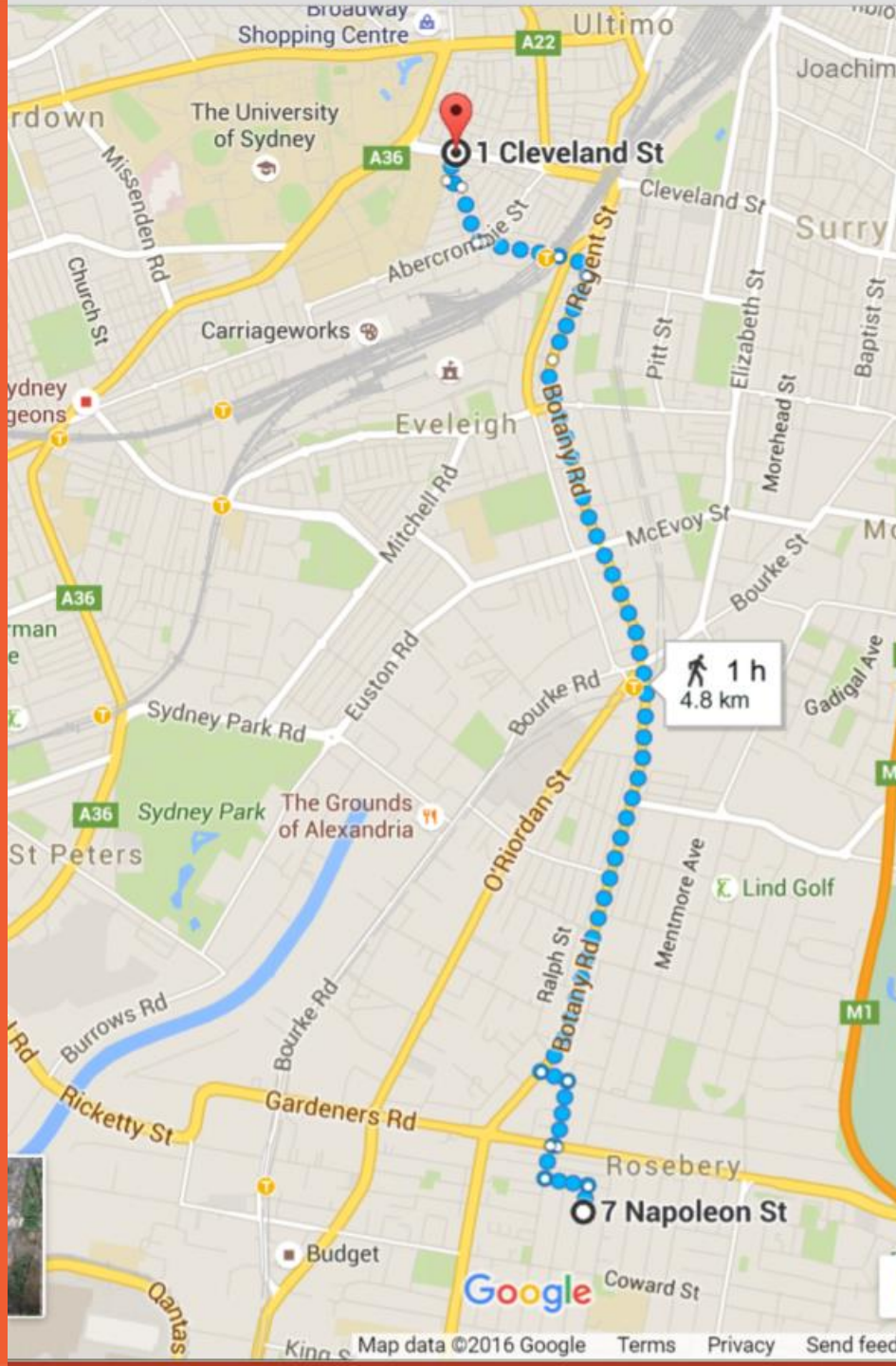
Lecture 3:
Greedy algorithms (Adv.)



THE UNIVERSITY OF SYDNEY

Greedy algorithms

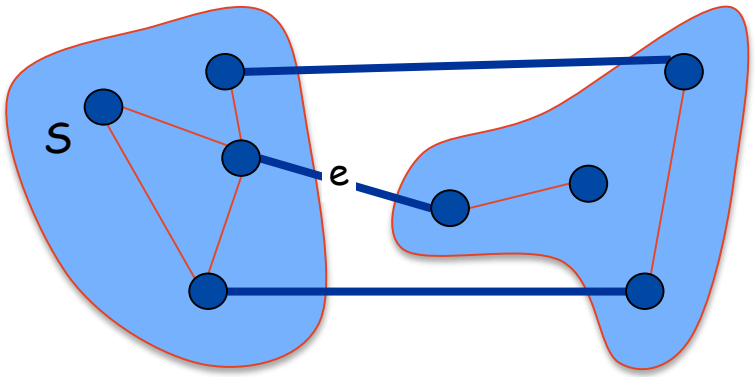
A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

MST properties

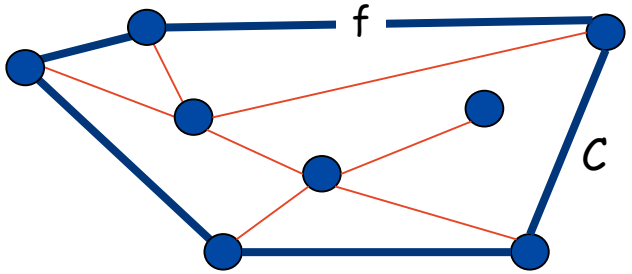
- **Simplifying assumption.** All edge costs c_e are distinct.

- **Cut property.** Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

- **Cycle property.** Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



e is in the MST



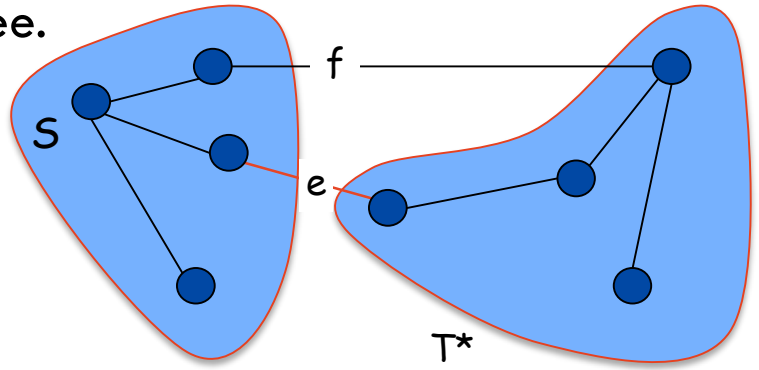
f is not in the MST

Greedy Algorithms

- **Simplifying assumption.** All edge costs c_e are **distinct**.

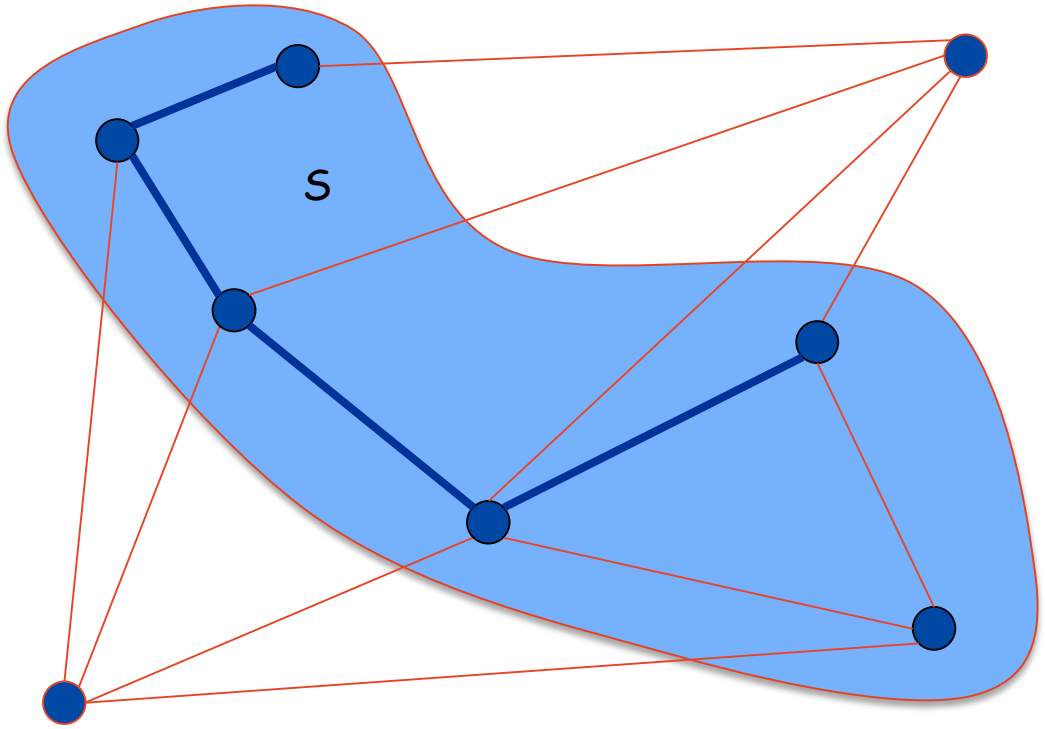
- **Cut property.** Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

- **Proof:** (exchange argument)
  - Suppose e does not belong to T*, and let's see what happens.
  - Adding e to T* creates a cycle C in T*.
  - Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D.
  - $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
  - Since $c_e < c_f$, $cost(T') < cost(T^*)$.
  - This is a contradiction. ■



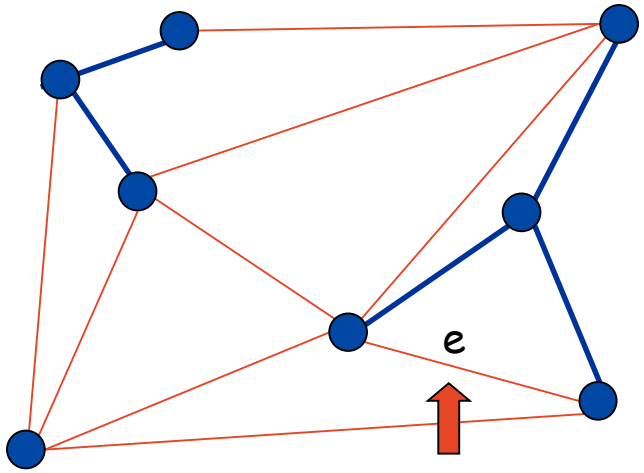
Prim's Algorithm

- Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]
  - Initialize S = any node.
  - Apply cut property to S.
  - Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.

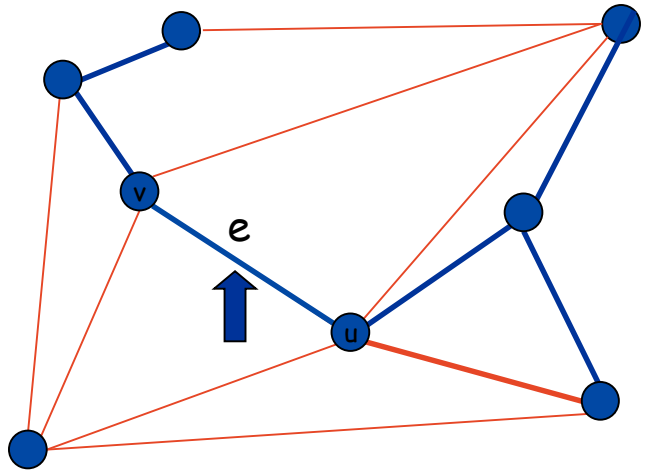


Kruskal's Algorithm

- Kruskal's algorithm. [Kruskal, 1956]
  - Consider edges in ascending order of weight.
  - **Case 1**: If adding e to T creates a cycle, discard e according to cycle property.
  - **Case 2:** Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

Implementation: Kruskal's Algorithm

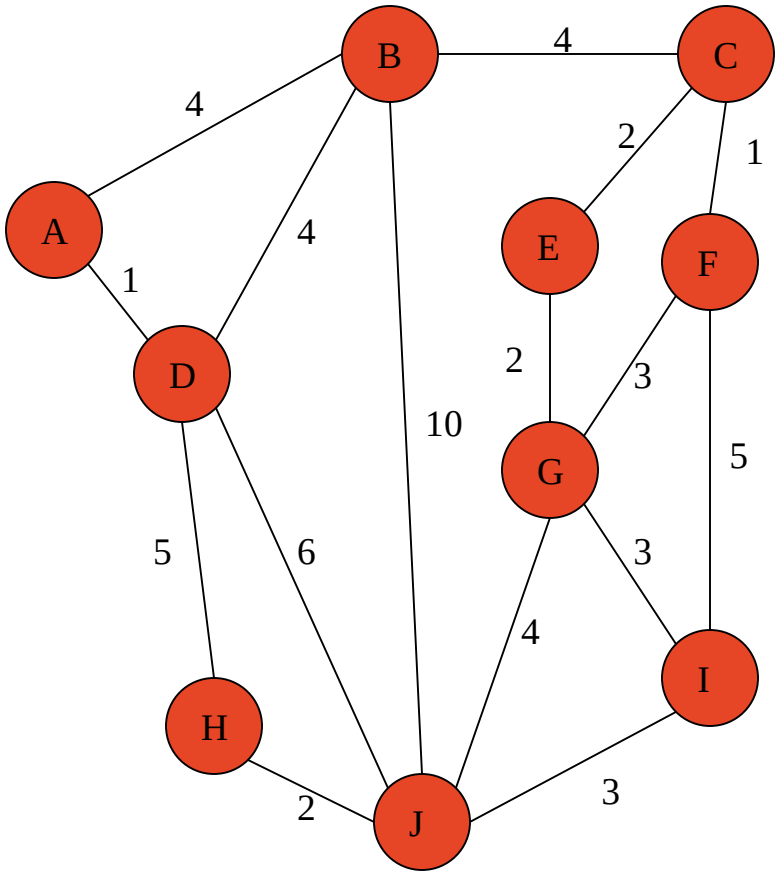
- Implementation.
  - Build set T of edges in the MST.
  - Maintain set for each connected component (= set of vertices).
  - **Time:** O(m log n) for sorting and m·(FindSet(u)+FindSet(v)+MergeSet).

    $m \le n^2 \Rightarrow \log m$ is $O(\log n)$

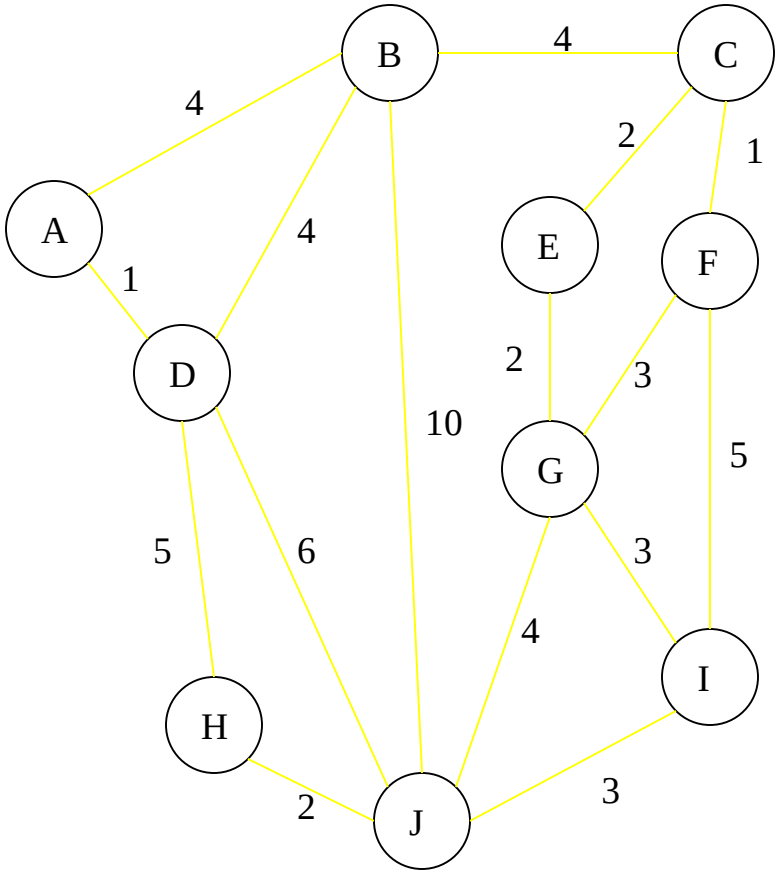
```
Kruskal(G, c) {
    Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
    T ← ∅
    foreach (u ∈ V) make a set containing singleton u
    for i = 1 to m  (in order of increasing weight)
        (u,v) = e_i
        if (u and v are in different sets) {
            T ← T ∪ {e_i}
            merge the sets containing u and v}
    return T
}
```

merge two connected components

Complete Graph



Thanks to Jonathan Davis
for animation



A 4 B

A 1 D

B 4 C

B 4 D

B 10 J

C 2 E

C 1 F

D 5 H

D 6 J

E 2 G

F 3 G

F 5 I

G 3 I

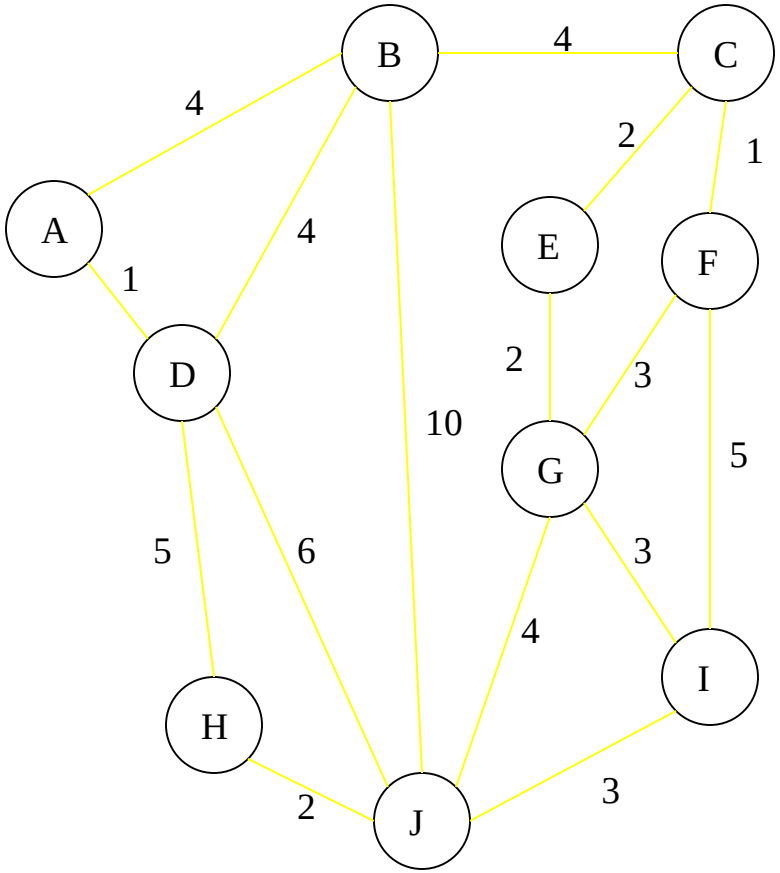
G 4 J

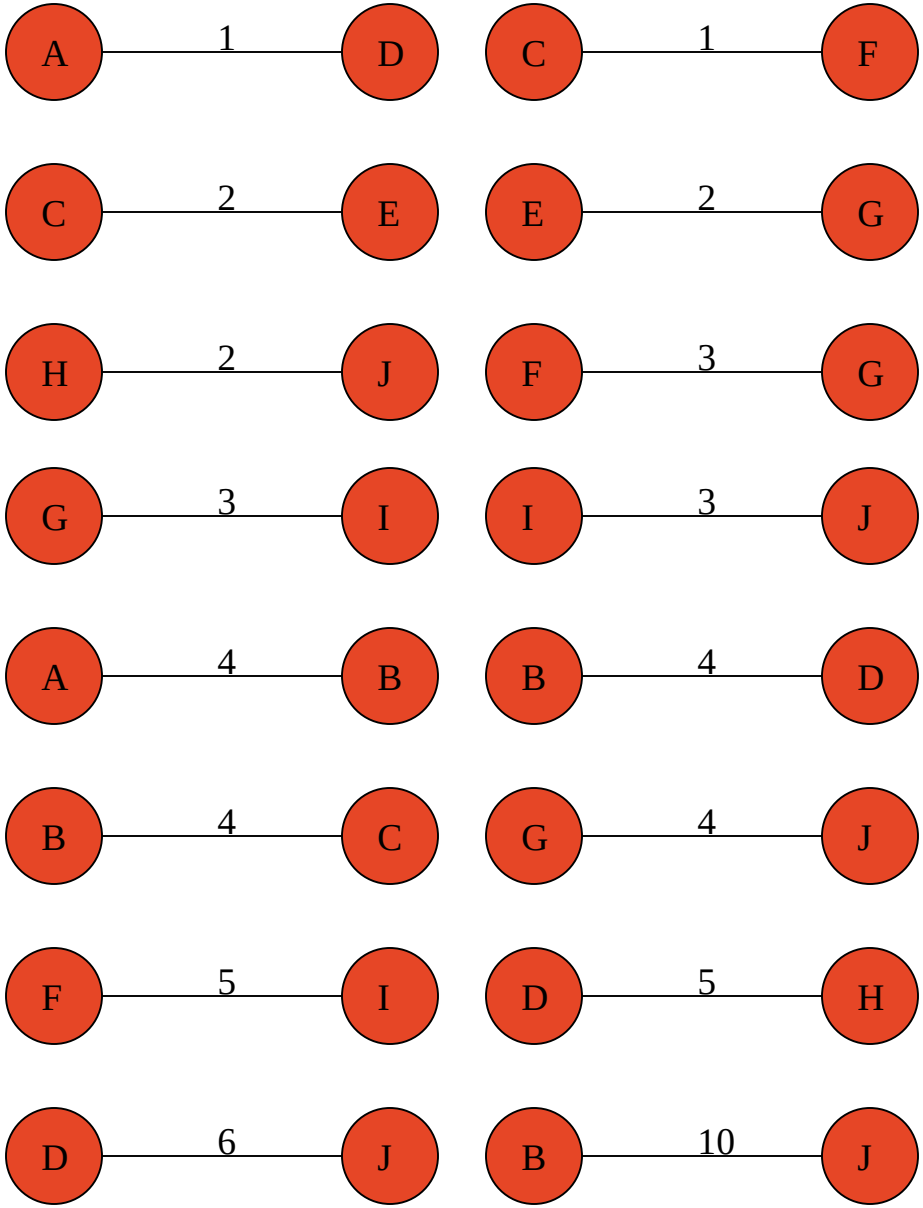
H 2 J

I 3 J

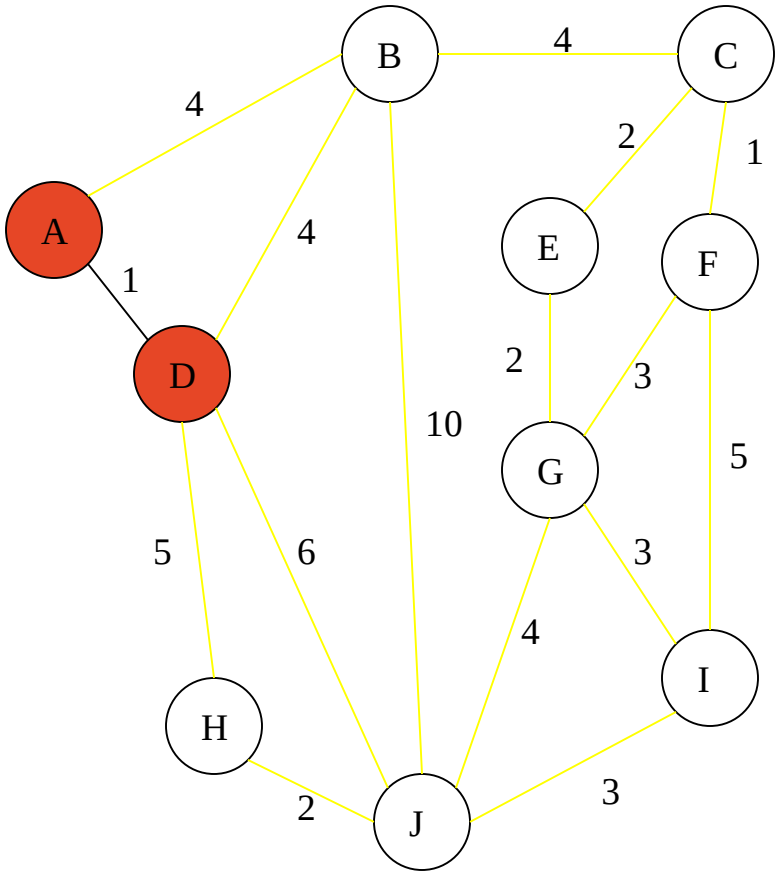
Sort Edges

(in reality they are placed in a priority queue)





Add Edge



A — 1 — D

C — 1 — F

C — 2 — E

E — 2 — G

H — 2 — J

F — 3 — G

G — 3 — I

I — 3 — J

A — 4 — B

B — 4 — D

B — 4 — C

G — 4 — J

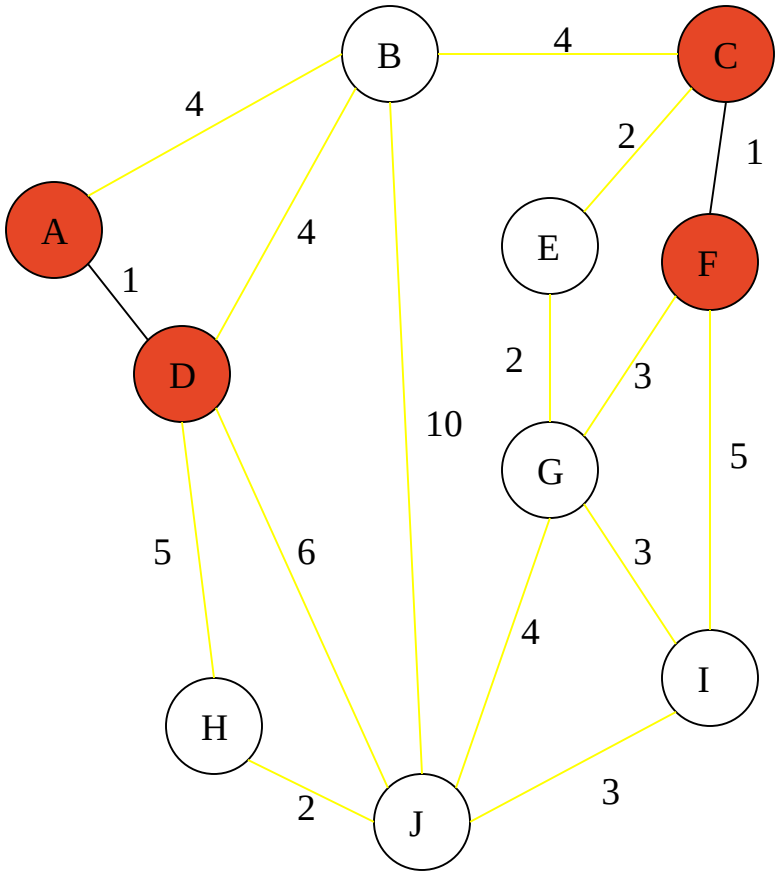
F — 5 — I

D — 5 — H

D — 6 — J

B — 10 — J

Add Edge



A 1 D

C 1 F

C 2 E

E 2 G

H 2 J

F 3 G

G 3 I

I 3 J

A 4 B

B 4 D

B 4 C

G 4 J

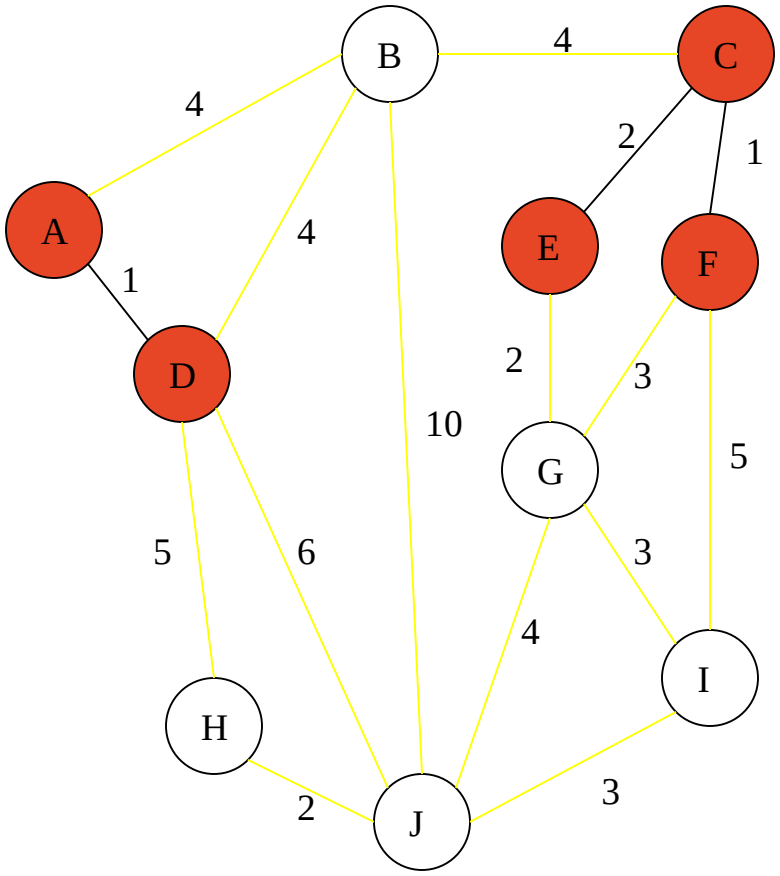
F 5 I

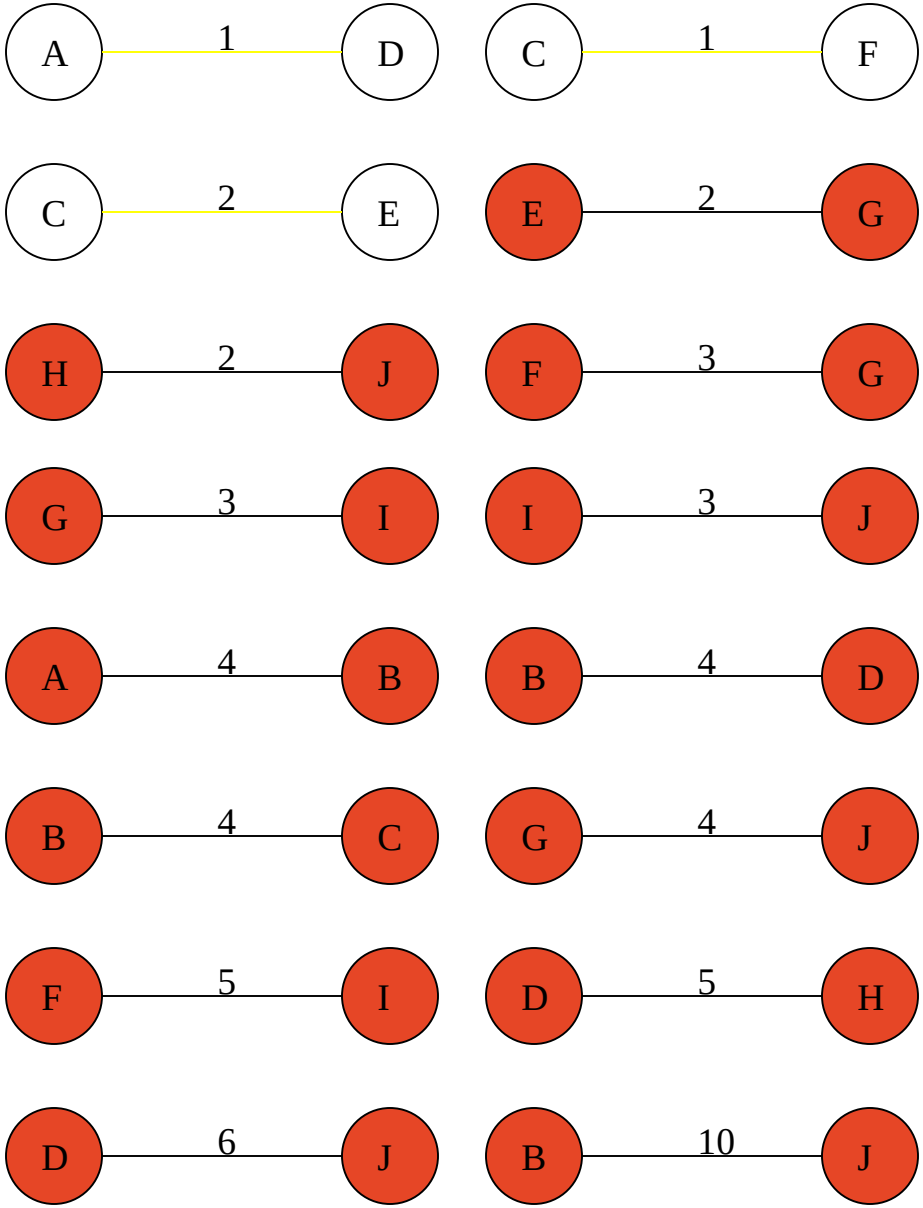
D 5 H

D 6 J

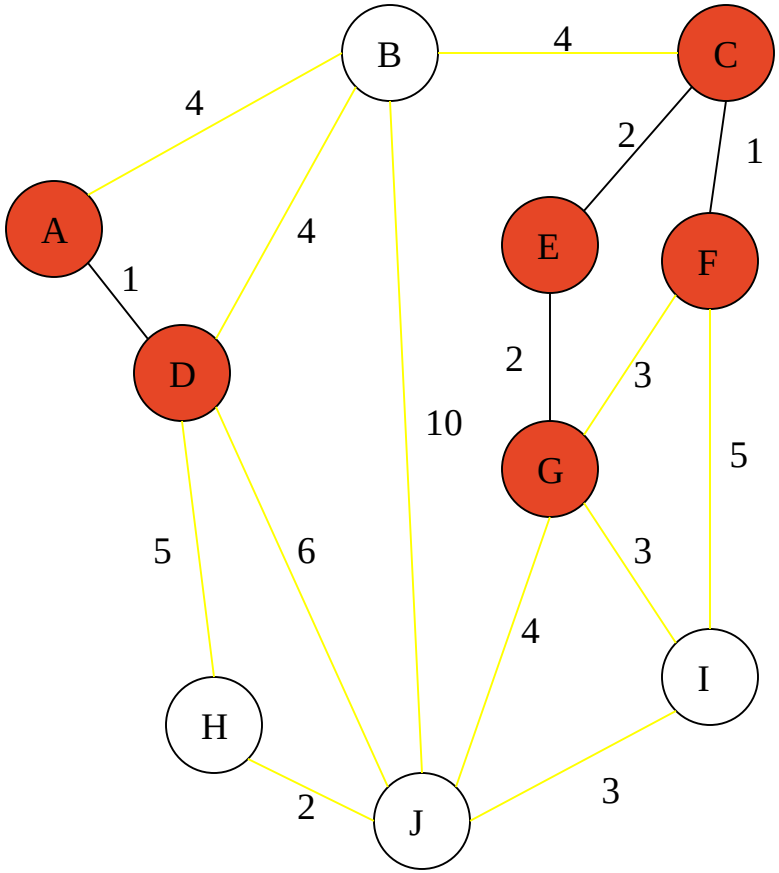
B 10 J

Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
3
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge



A 1 D

C 1 F

C 2 E

E 2 G

H 2 J

F 3 G

G 3 I

I 3 J

A 4 B

B 4 D

B 4 C

G 4 J

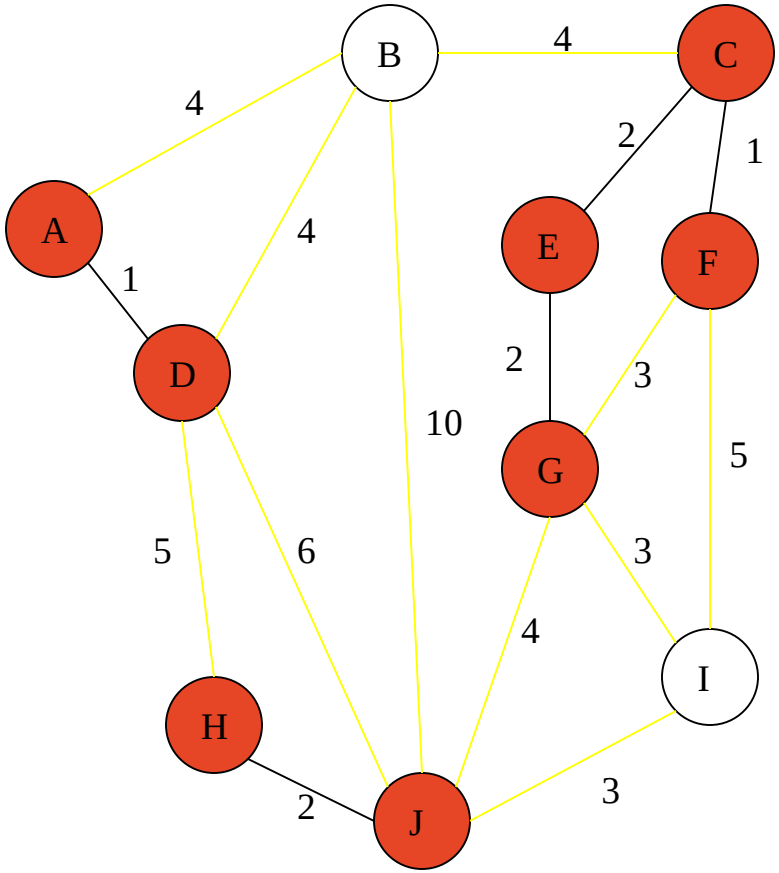
F 5 I

D 5 H

D 6 J

B 10 J

Add Edge



A 1 D

C 1 F

C 2 E

E 2 G

H 2 J

F 3 G

G 3 I

I 3 J

A 4 B

B 4 D

B 4 C

G 4 J

F 5 I

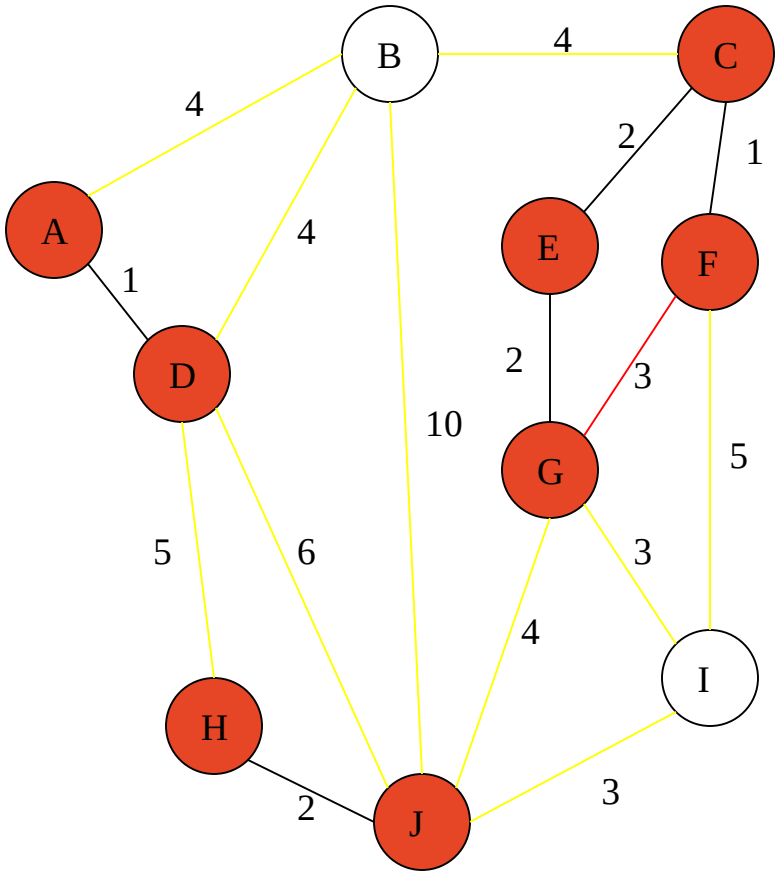
D 5 H

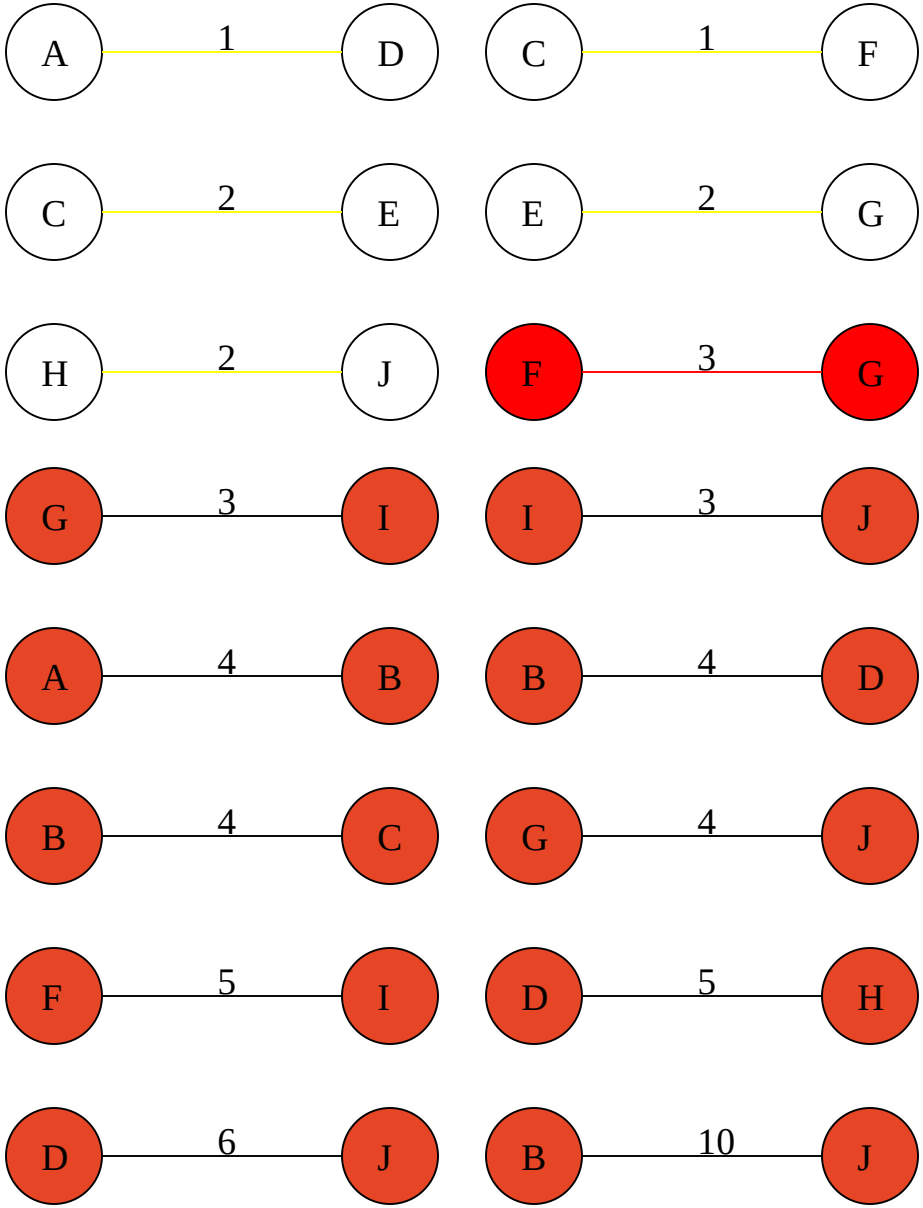
D 6 J

B 10 J

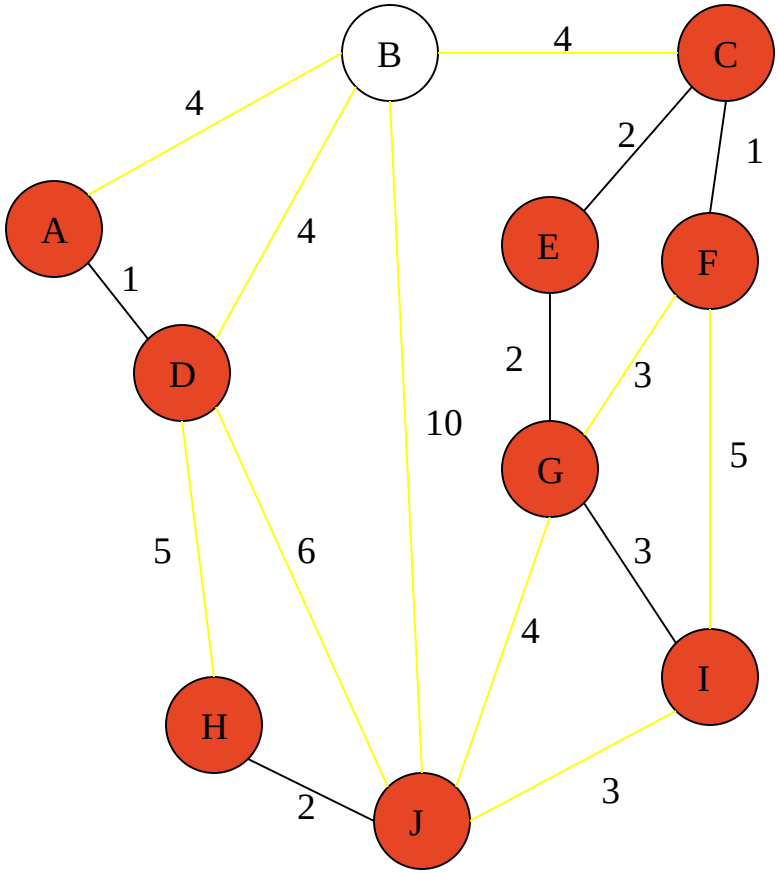
Cycle

Don’t Add Edge





Add Edge



A 1 D

C 1 F

C 2 E

E 2 G

H 2 J

F 3 G

G 3 I

I 3 J

A 4 B

B 4 D

B 4 C

G 4 J

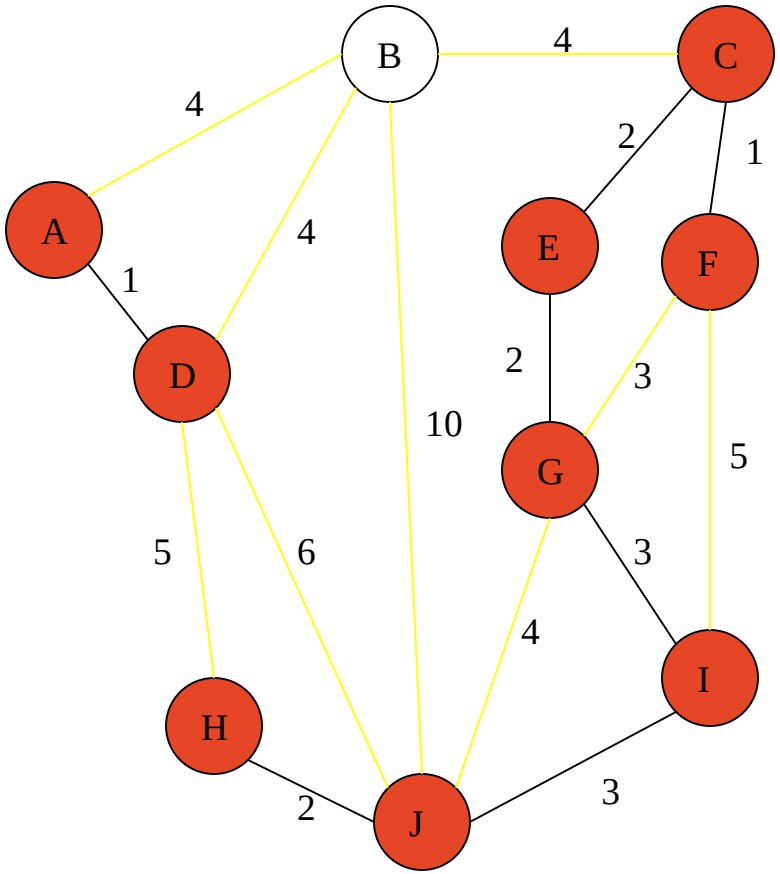
F 5 I

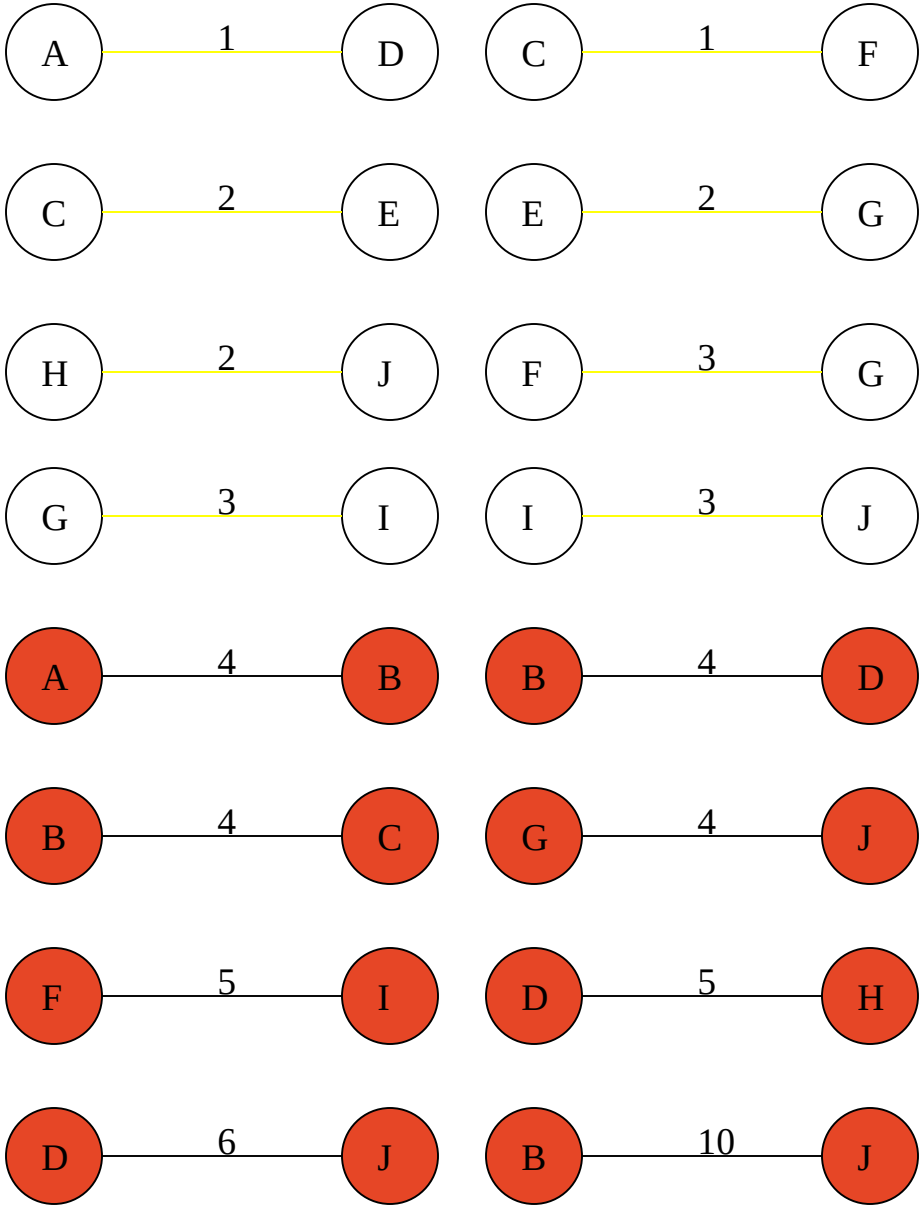
D 5 H

D 6 J

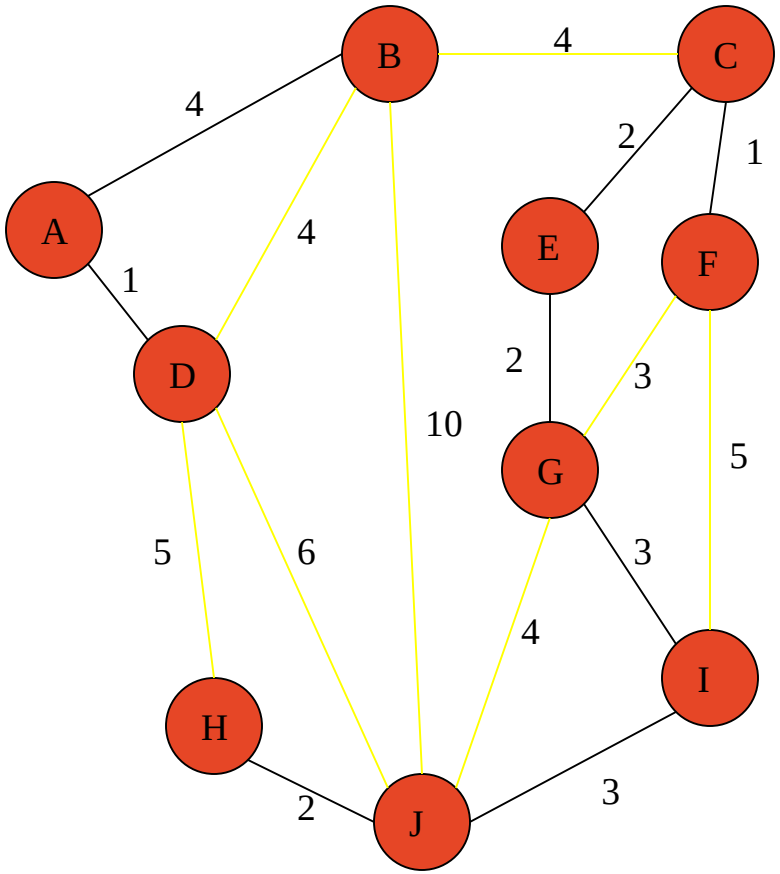
B 10 J

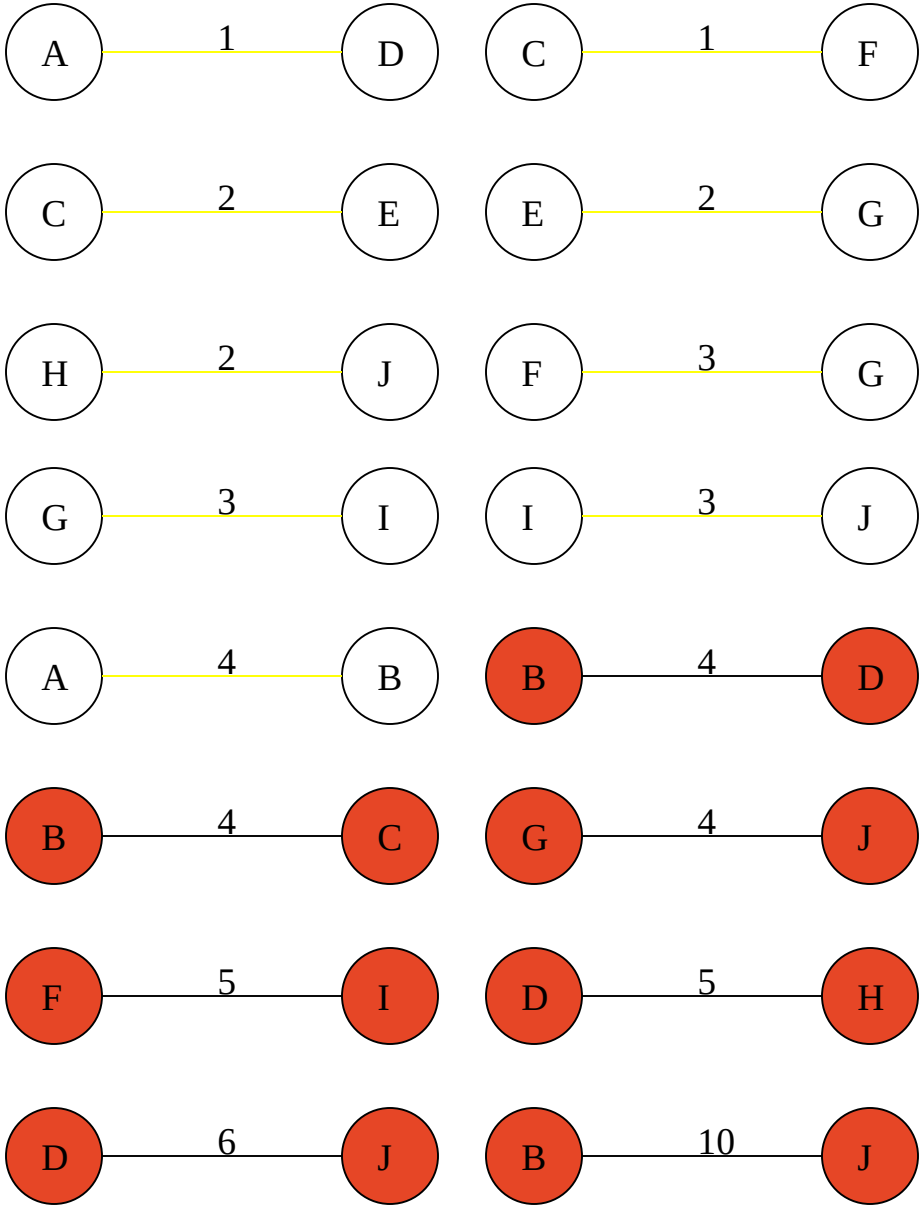
Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
3
2


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

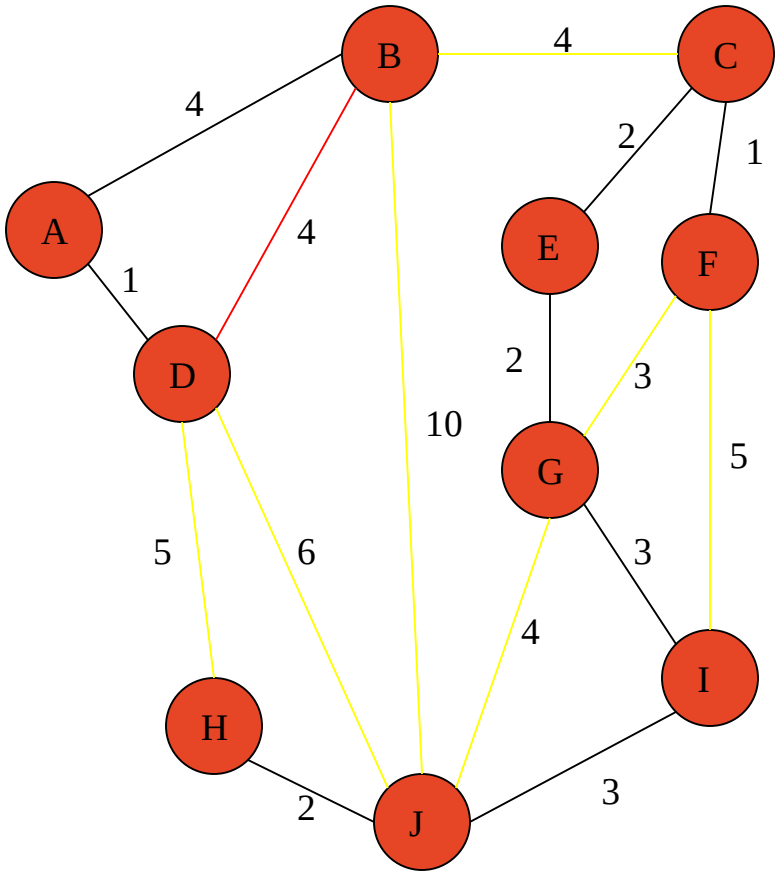
Add Edge

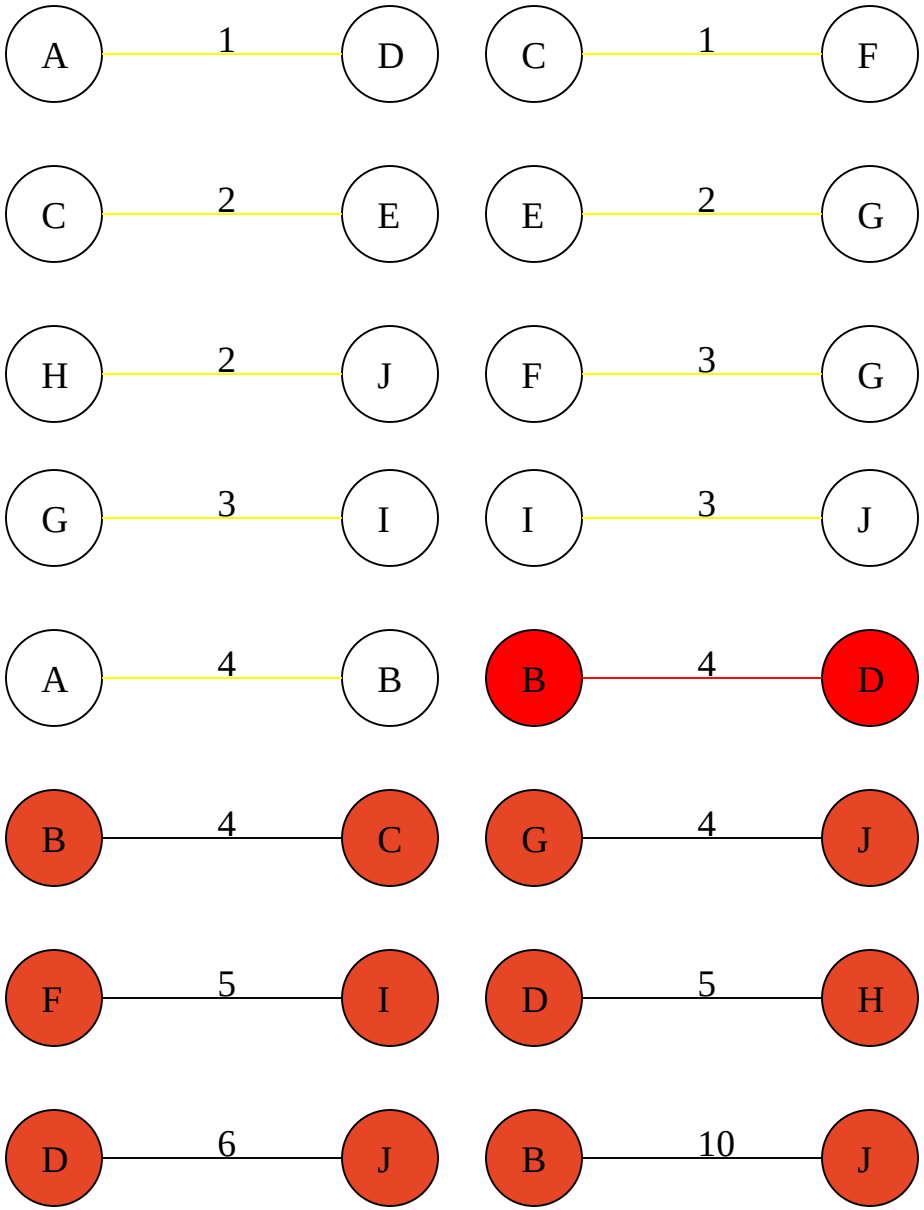

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
3
2


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

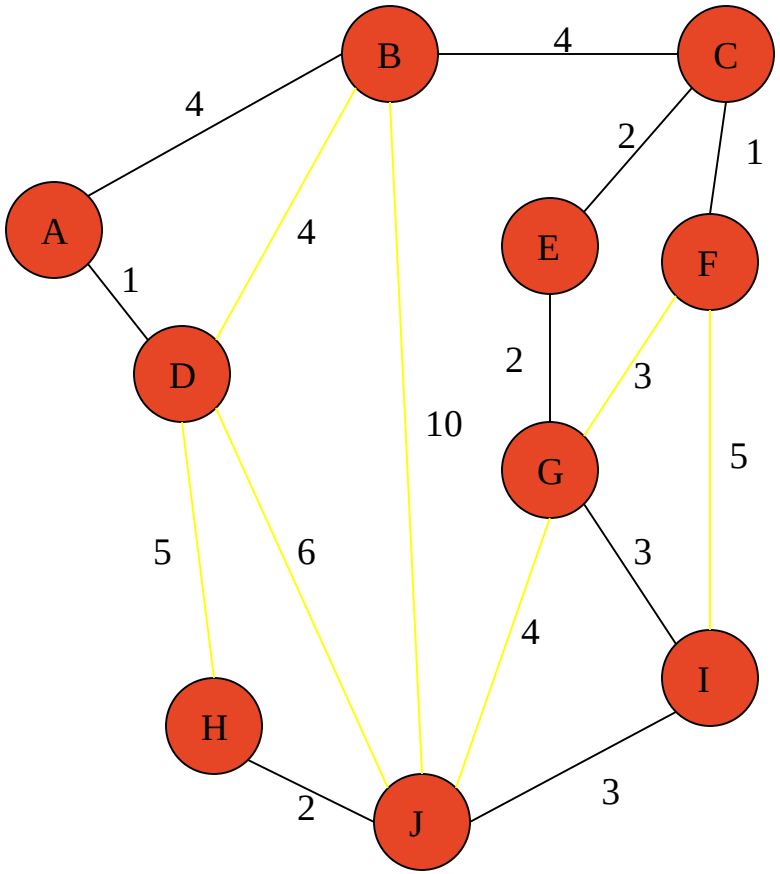
Cycle

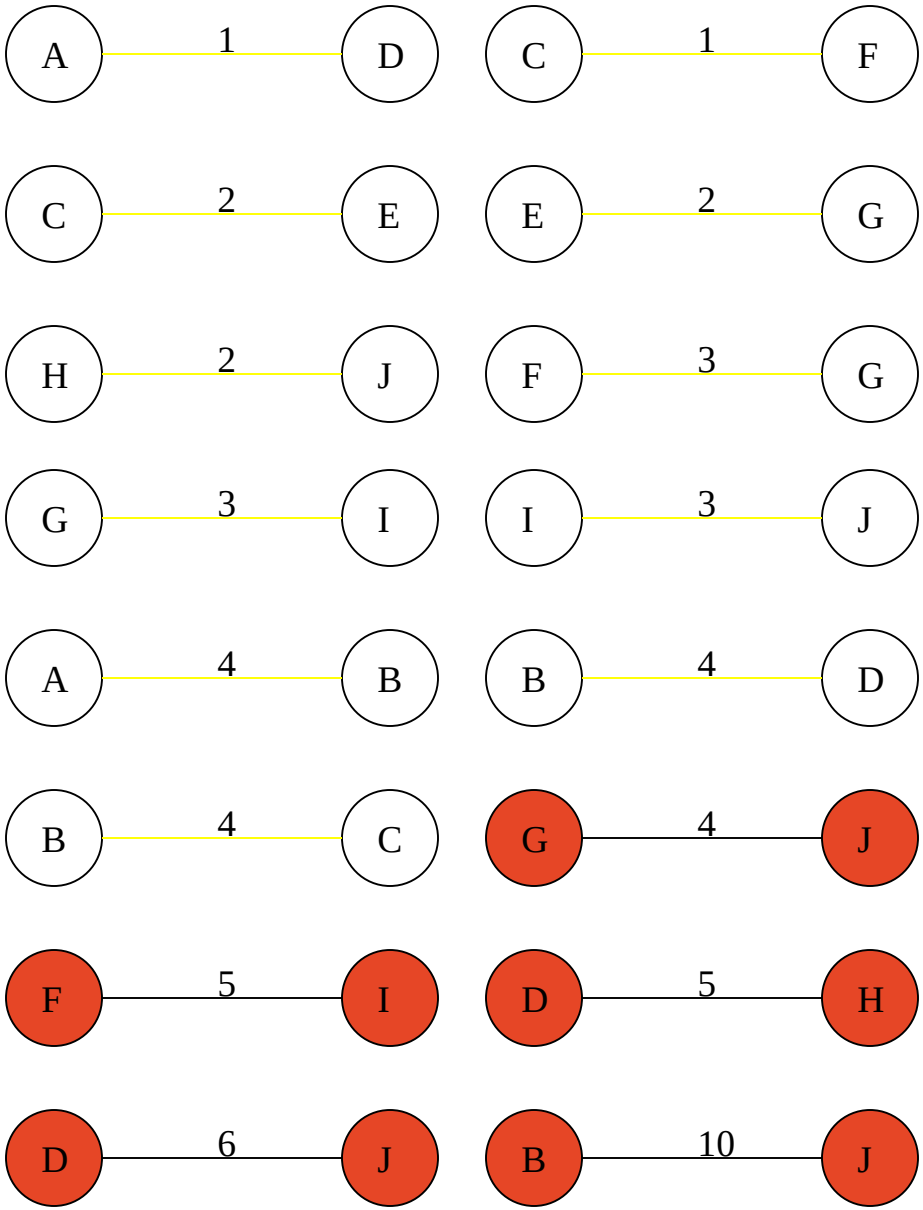
Don’t Add Edge



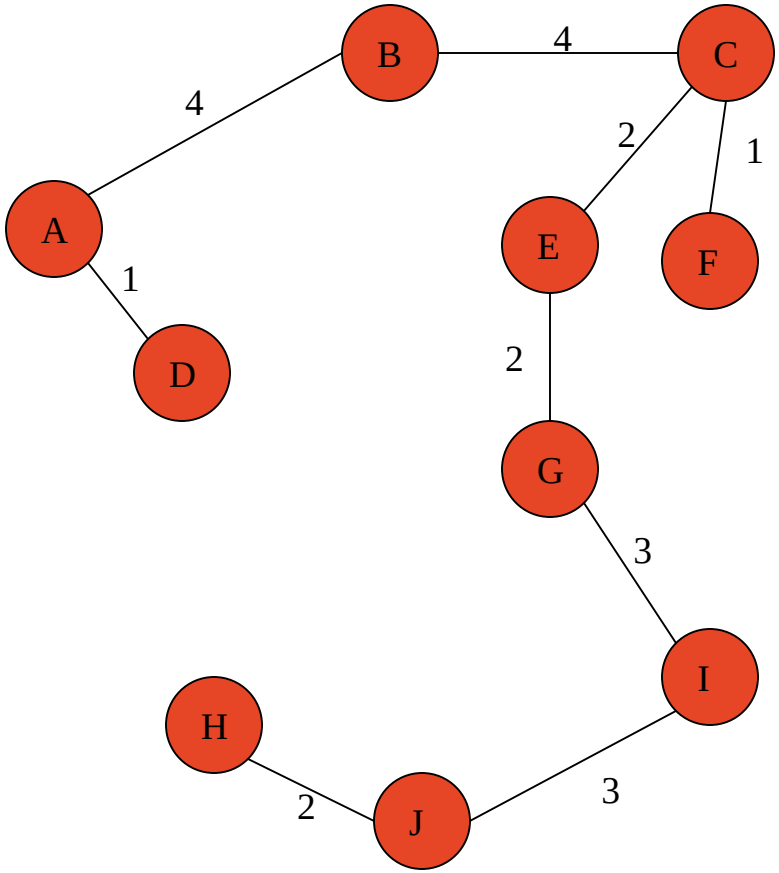


Add Edge

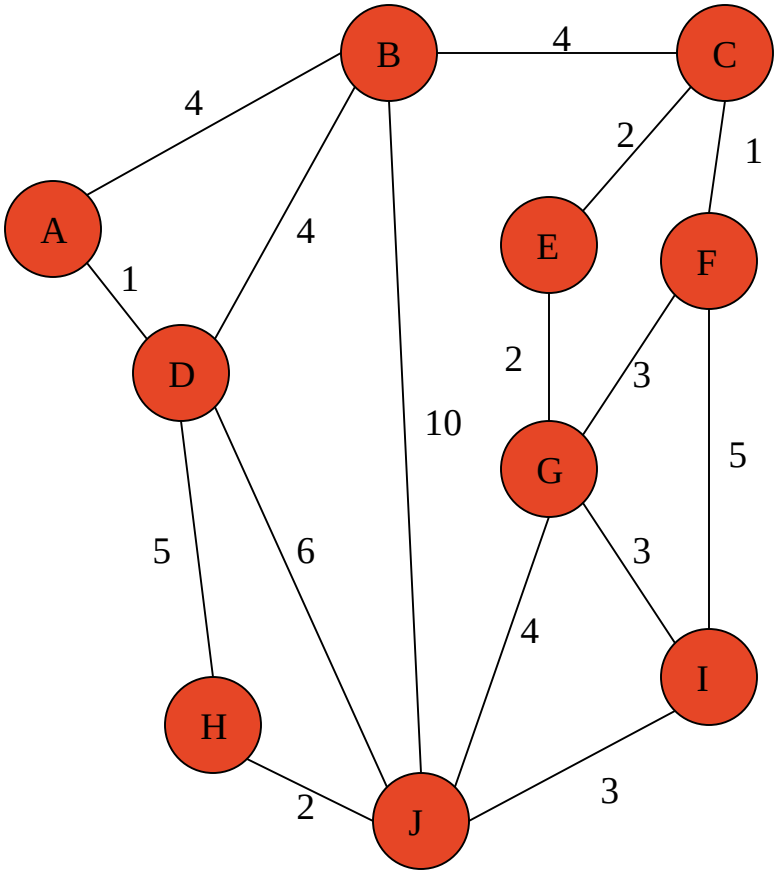

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
3
2


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Minimum Spanning Tree



Complete Graph



Union-Find data structure

To efficiently implement Kruskal's algorithm we need a data structure that supports:

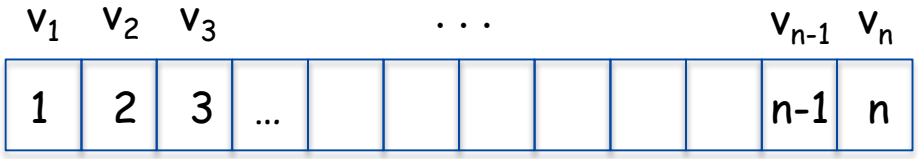
- MakeUnionFind(S): returns a data structure where each element is a set (connected component).
- Find(u): for $u \in S$ return the set (connected component) that u belongs to.
- Union(A,B): merges sets A and B into a single set.

Union-Find data structure

First attempt:

Component – array of length n

Component[s] – the name of the set $s \in S$ belongs to



- MakeUnionFind(S): O(n)
- Find(u): O(1)
- Union(A,B): O(n) [go through the array and change all B's to A's]

Union-Find data structure

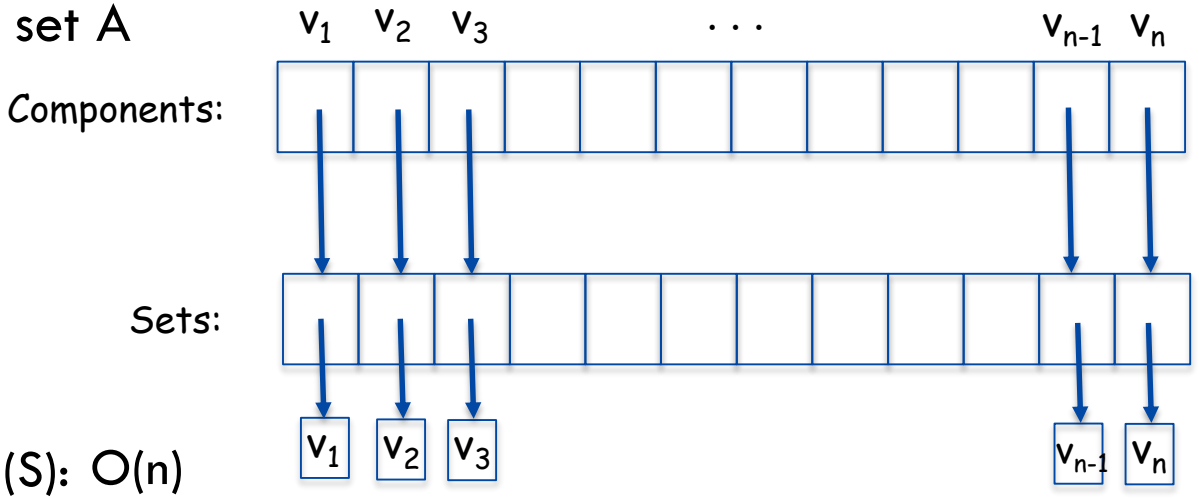
Second attempt:

Component[s] – a pointer to the set it belongs to

Set – array of length n

Set[A] – linked list of the elements in the set

Size[A] – size of set A



- MakeUnionFind(S): O(n)
- Find(u): O(1)
- Union(A,B): O(n) [go through the set and change the pointers of the smallest set]

Union-Find data structure

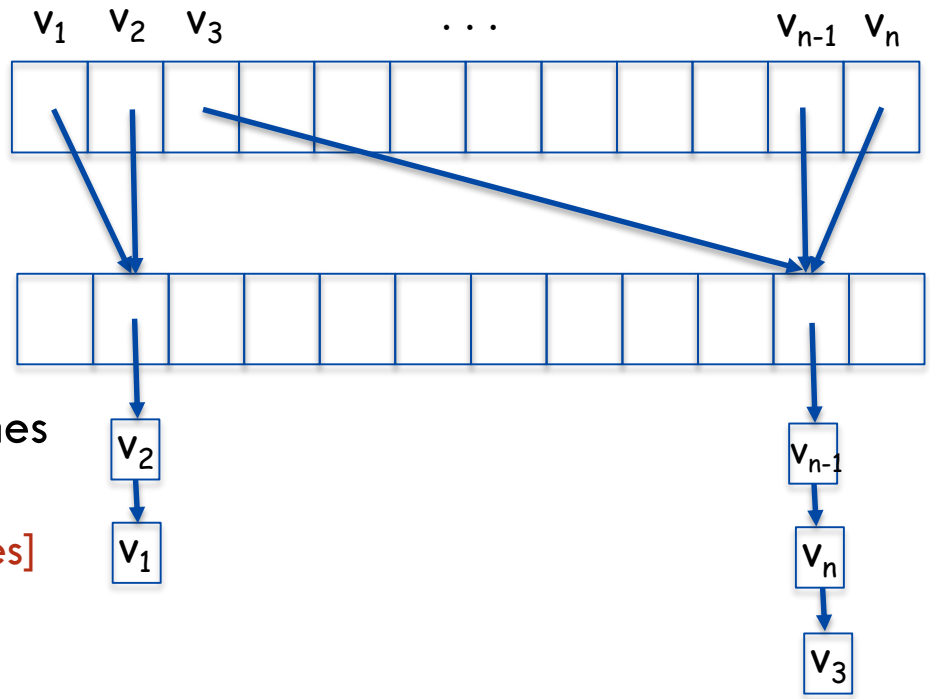
Second attempt: An amortized analysis

Consider Q Union operations.

Only step that takes more than O(1) time is updating `Component`.

Consider an element $s \in S$. How many times will `Component[s]` be updated?

O(log n) [each time the set size doubles]



- MakeUnionFind(S): O(n)
- Find(u): O(1)
- Q Union(A,B) operations: O(n log n) [go through the set and change the pointers of the smallest set]

(Can be strengthened to O(Q log Q))

Implementation: Kruskal's Algorithm

- Implementation.
  - Build set T of edges in the MST.
  - Maintain set for each connected component.
  - **Time:** O(m log n) for sorting and m·(FindSet(u)+FindSet(v)+MergeSet).

    $m \le n^2 \Rightarrow \log m$ is $O(\log n)$

```
Kruskal(G, c) {
    Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
    T ← ∅

    foreach (u ∈ V) make a set containing singleton u

    for i = 1 to m  (in order of increasing weight)
        (u,v) = e_i
        if (u and v are in different sets) {
            T ← T ∪ {e_i}
            merge the sets containing u and v}
    return T
}
```

merge two connected components

Implementation: Kruskal's Algorithm

- Implementation.
  - Build set T of edges in the MST.
  - Maintain set for each connected component.
  - **Time:** O(m log n) for sorting and O(m+n log n) for merging.

    $m \le n^2 \Rightarrow \log m$ is $O(\log n)$

```
Kruskal(G, c) {
    Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
    T ← ∅

    foreach (u ∈ V) make a set containing singleton u

    for i = 1 to m   (in order of increasing weight)
        (u,v) = e_i
        if (u and v are in different sets) {
            T ← T ∪ {e_i}
            merge the sets containing u and v}
    return T
}
```

merge two connected components

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

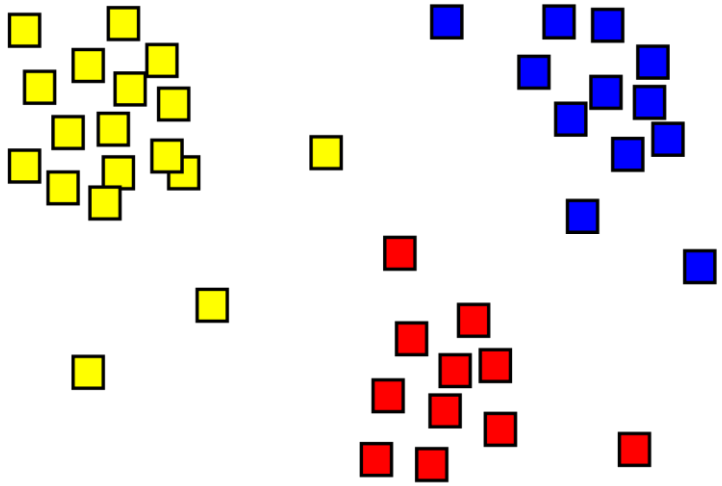
Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

e.g., if all edge costs are integers, perturbing cost of edge e_i by i / n^2

```
boolean less(i, j) {
   if      (cost(e_i) < cost(e_j)) return true
   else if (cost(e_i) > cost(e_j)) return false
   else if (i < j)                 return true
   else                            return false
}
```

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

Clustering



Clustering

- **Clustering.** Given a set U of n objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

  photos, documents. micro-organisms

- **Distance function.** Numeric value specifying "closeness" (distance) of two objects.

  number of corresponding pixels whose intensities differ by some threshold

- **Fundamental problem.** Divide into clusters so that points in different clusters are far apart.
  - Routing in mobile ad hoc networks.
  - Identify patterns in gene expression.
  - Document categorization for web search.
  - Similarity searching in medical image databases
  - Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

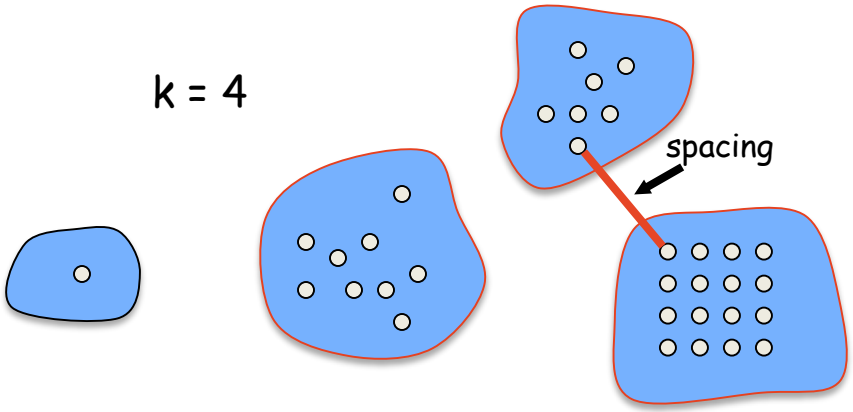
Clustering of Maximum Spacing

- **k-clustering.** Divide objects into k non-empty groups.

- **Distance function.** Assume it satisfies some properties.
  - $d(p_i, p_j) = 0$ if and only if $p_i = p_j$
  - $d(p_i, p_j) \geq 0$ (nonnegativity)
  - $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

- **Spacing.** Min distance between any pair of points in different clusters.

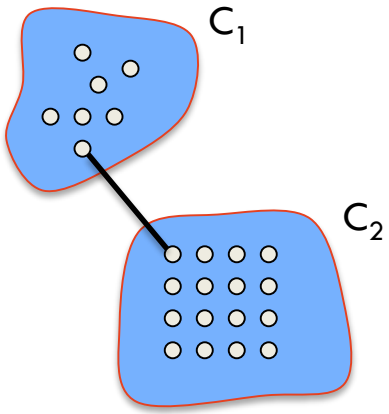
- **Clustering of maximum spacing.** Given an integer k, find a k-clustering of maximum spacing.



Clustering of Maximum Spacing

k=2

Consider two clusters C_1 and C_2.



What is the distance between them?

⇒ Weight of the cheapest edge between them

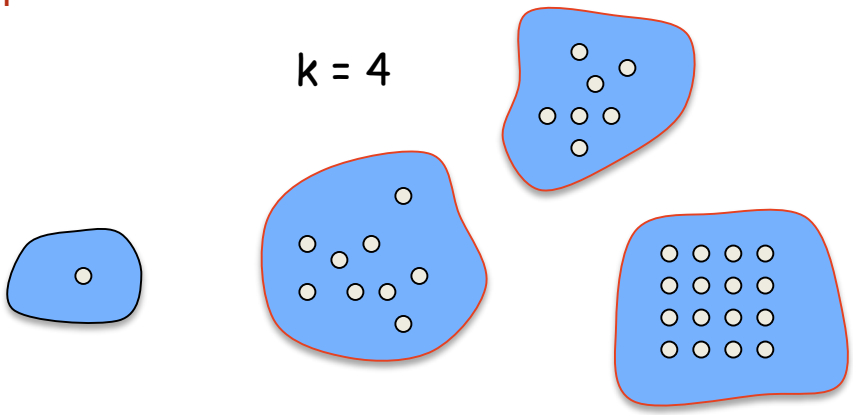
This must be an edge in the MST!

Can there be two MST edges between C_1 and C_2? No

Vertices in a cluster must be a connected component in MST.

Similar argument for k>2.

(Distance defined by an MST edge)



Greedy Clustering Algorithm

- **Single-link k-clustering algorithm.**
  - Form a graph on the vertex set V, corresponding to n clusters.
  - Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
  - Repeat n-k times until there are exactly k clusters.

- **Key observation.** This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

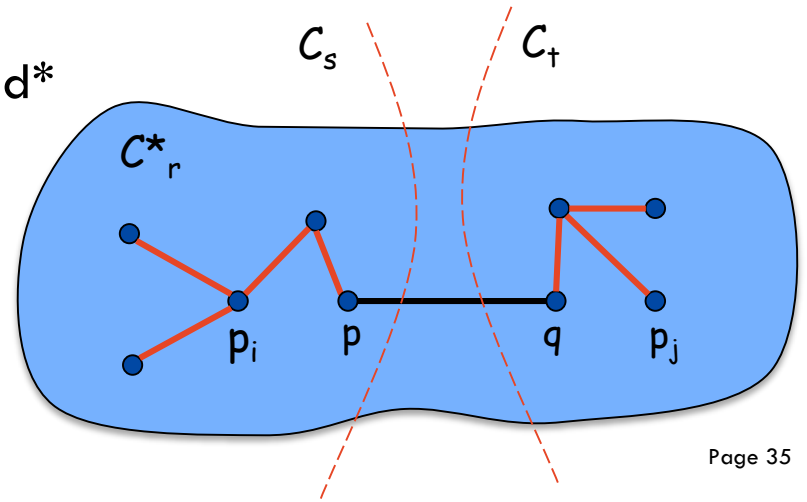
- **Remark.** Equivalent to finding an MST and deleting the k-1 most expensive edges.

Greedy Clustering Algorithm: Analysis

Theorem: Let C* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C* is a k-clustering of max spacing.

Proof: Let C denote some other clustering $C_1, \ldots, C_k$.

- The spacing of C* is the length d* of the $(k-1)^{st}$ most expensive edge.
- Let p_i, p_j be in the same cluster in C*, say C^*_r, but different clusters in C, say C_s and C_t.
- Some edge (p, q) on p_i-p_j path in C^*_r spans two different clusters in C.
- All edges on p_i-p_j path have length $\leq d^*$ since Kruskal chose them.
- Spacing of C is $\leq d^*$ since p and q are in different clusters. ■



Euclidean Travelling Salesman Problem (TSP)

- Vertices are points in some Euclidean space (assume 2D)
- Distance between vertices is the usual Euclidean distance.
- A set of n points (described by their (x,y) coordinates in the plane), can be considered as a **complete** weighted undirected graph on n vertices.
- **TSP problem:** Given a set of n locations (coordinates in 2D) find a travelling salesperson tour of minimum length.
- Algorithm for this problem?

Note: Metric or distance functions

A cost function d(u,v) is a metric if it satisfies the following properties:

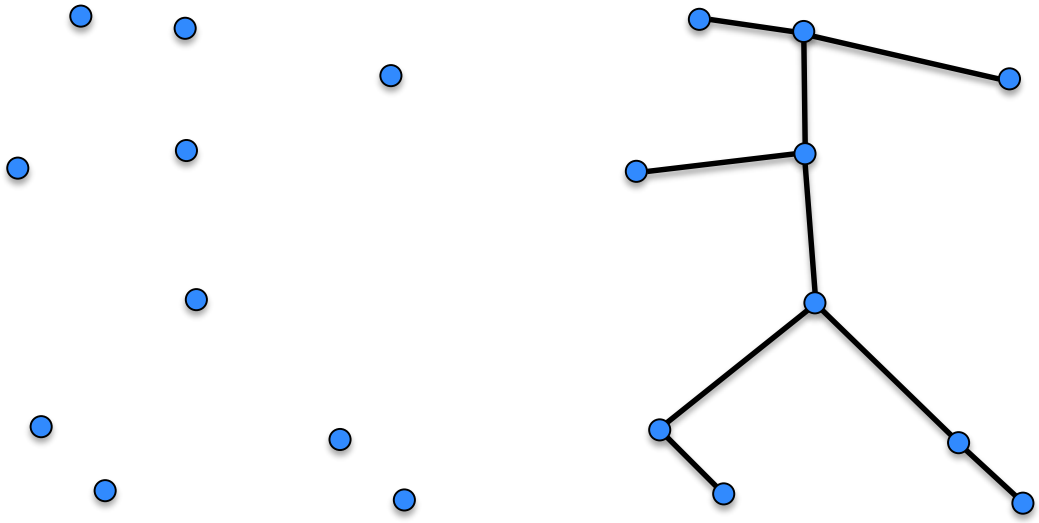
- $d(u,v) \geq 0$
- $d(u,u) = 0$
- $d(u,v) = d(v,u)$
- $d(u,v) \leq d(u,x) + d(x,v)$

Metric TSP

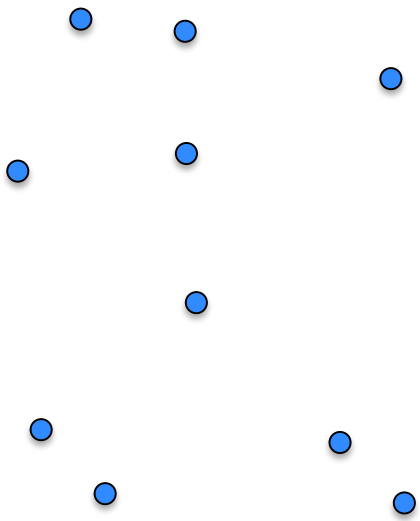
Approximate TSP(G)

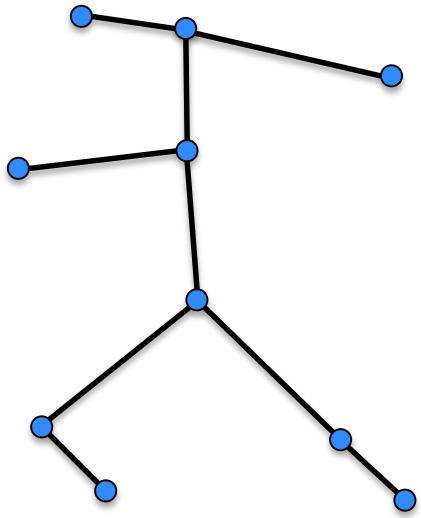
1. T := MST(G)
2. “Double” the tree T to make it into a tour H
3. Return H as the TSP tour

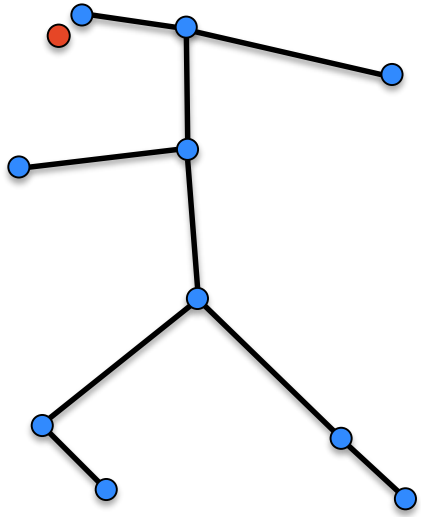
The input



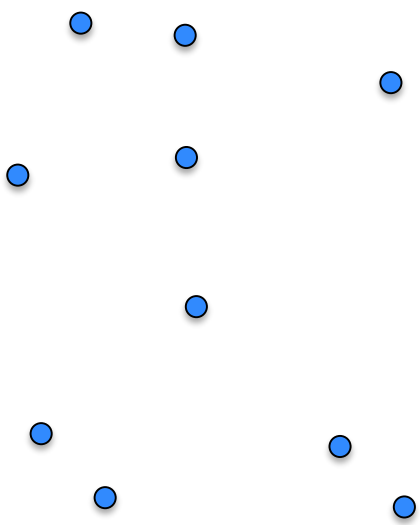
The tour

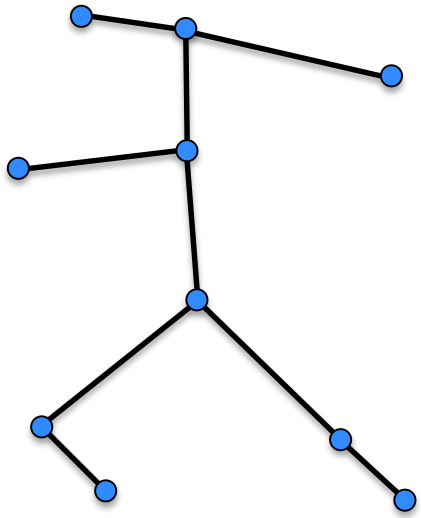


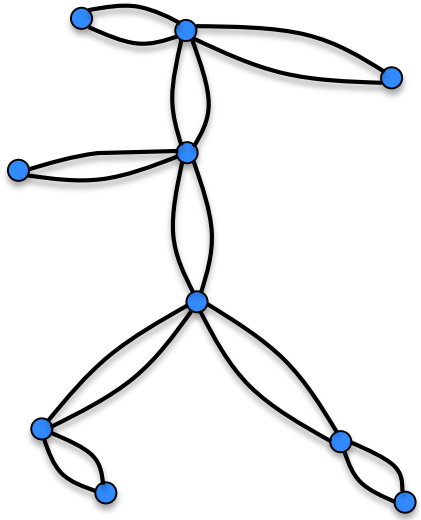




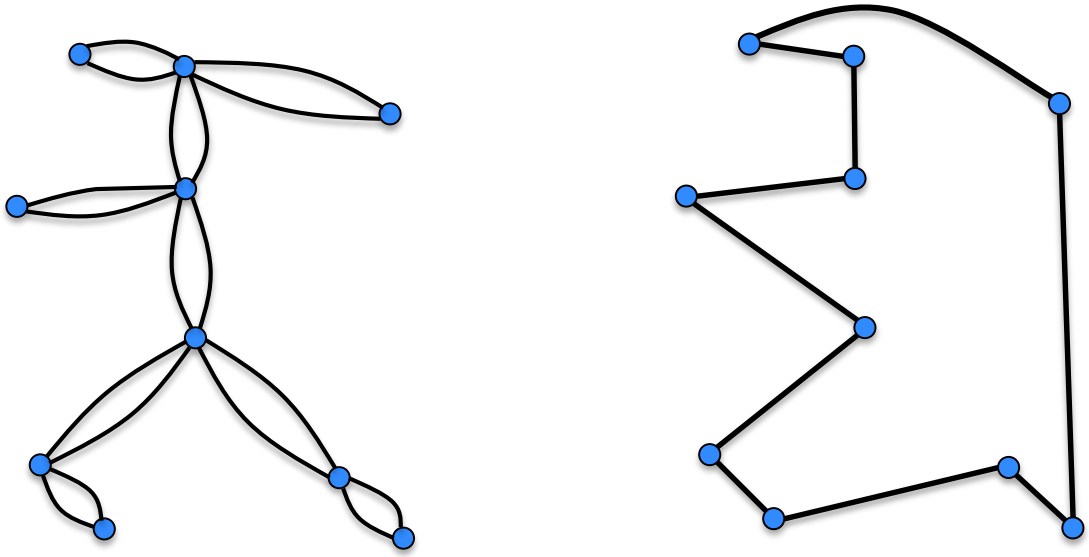
The tour







The tour with shortcuts



Is this a good tour?

- The tour may not be optimal (shortest possible).
- How do we prove that statement?
- We only need provide a counterexample:

  A graph (input instance) for which the algorithm will return a sub-optimal tour.

How far off can the tour be from optimal?

Is there an approximation guarantee?

$$\text{Approximation ratio} = \frac{\text{Cost of apx solution}}{\text{Cost of optimal solutions}}$$

An approximation algorithm for a minimization problem requires and approximation guarantee:

- Approximation ratio $\leq c$
- Approximation solution $\leq c \cdot$ value of optimal solution

2-approximation

Theorem: The MST based approximation algorithm described on the previous slides is a 2-approximation algorithm for the Metric TSP.

Proof:

Assume H_{opt} is the optimal tour and that H_A is the tour returned by the approximation algorithm.

We want to prove that $cost(H_A) \leq cost(H_{opt})$.

Let T denote the MST and let W denote the walk around the MST

$$\Rightarrow \quad cost(T) \leq cost(H_{opt})$$
$$cost(W) \leq 2 \cdot cost(T) \leq 2 \cdot cost(H_{opt})$$

2-approximation

- Is this bound tight?
- We know that the solution is at most 2 times the optimal.
- Are there input instances where the solution given by the approximation algorithm is indeed equal 2 times the optimal?

Notes for the metric and Euclidean TSP

- The previous algorithm is a simplified version of an algorithm due to Christofides from 1976
- The Christofides algorithm is a 3/2 approximation.
- For the Euclidean TSP, there is a polynomial time approximation scheme (Arora 1998, Mitchell 1998)

SET-COVER

[Slides by D Moshkovitz]

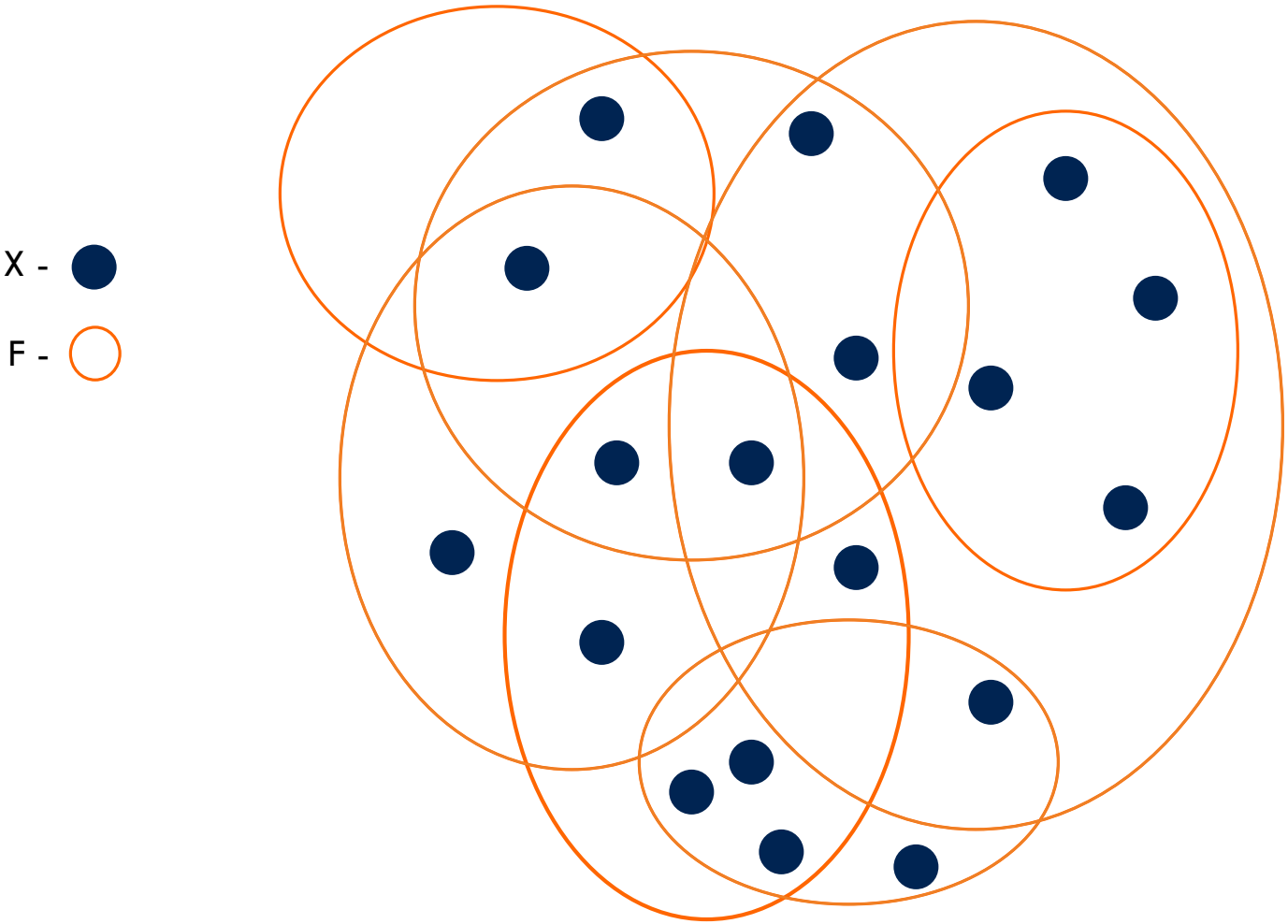
Instance: a finite set X and a family F of subsets of X, such that

$$X = \bigcup_{S \in F} S$$

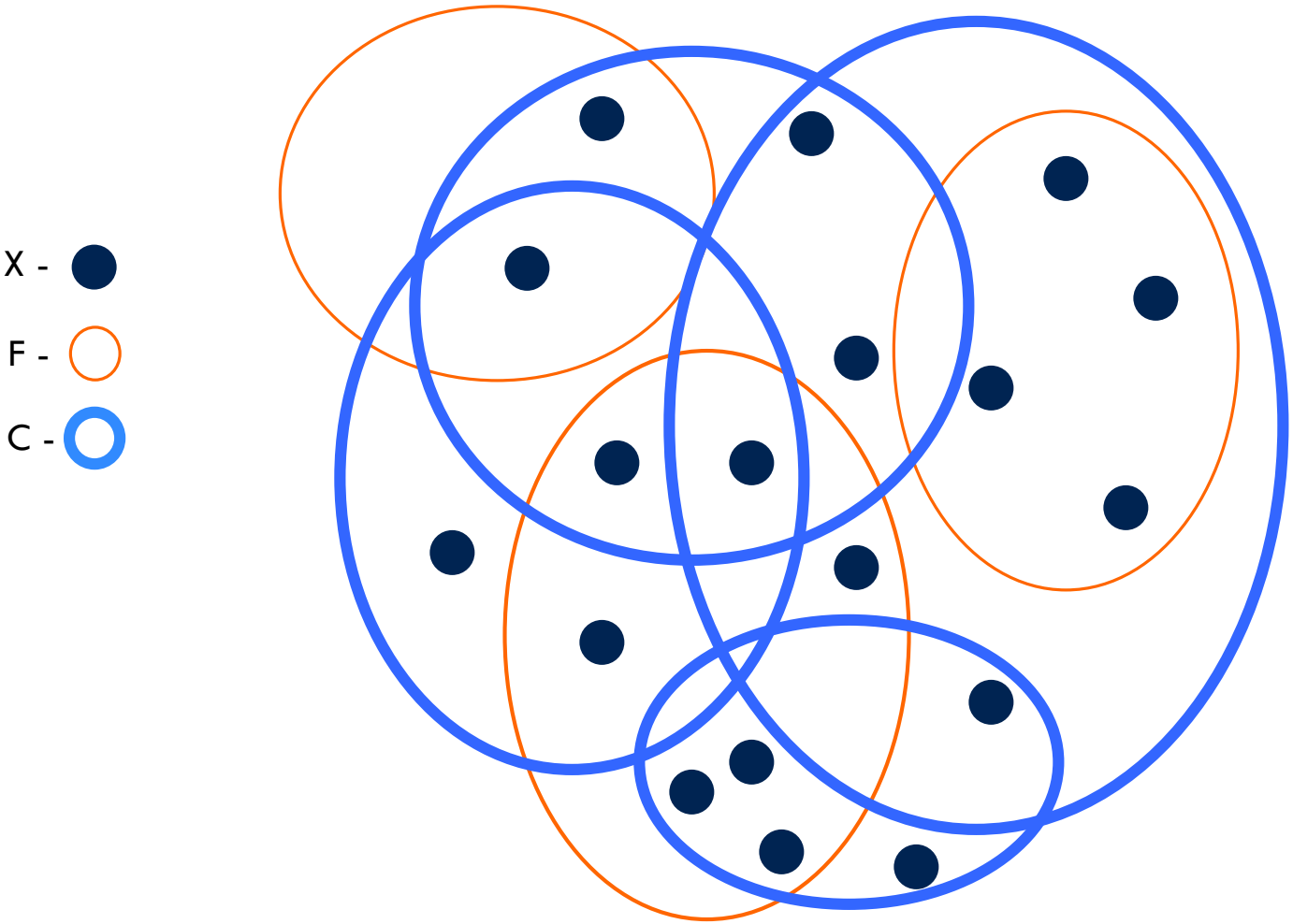
Problem: find a set $C \subseteq F$ of minimal size which *covers* X, i.e.

$$X = \bigcup_{S \in C} S$$

SET-COVER: Example



SET-COVER: Example



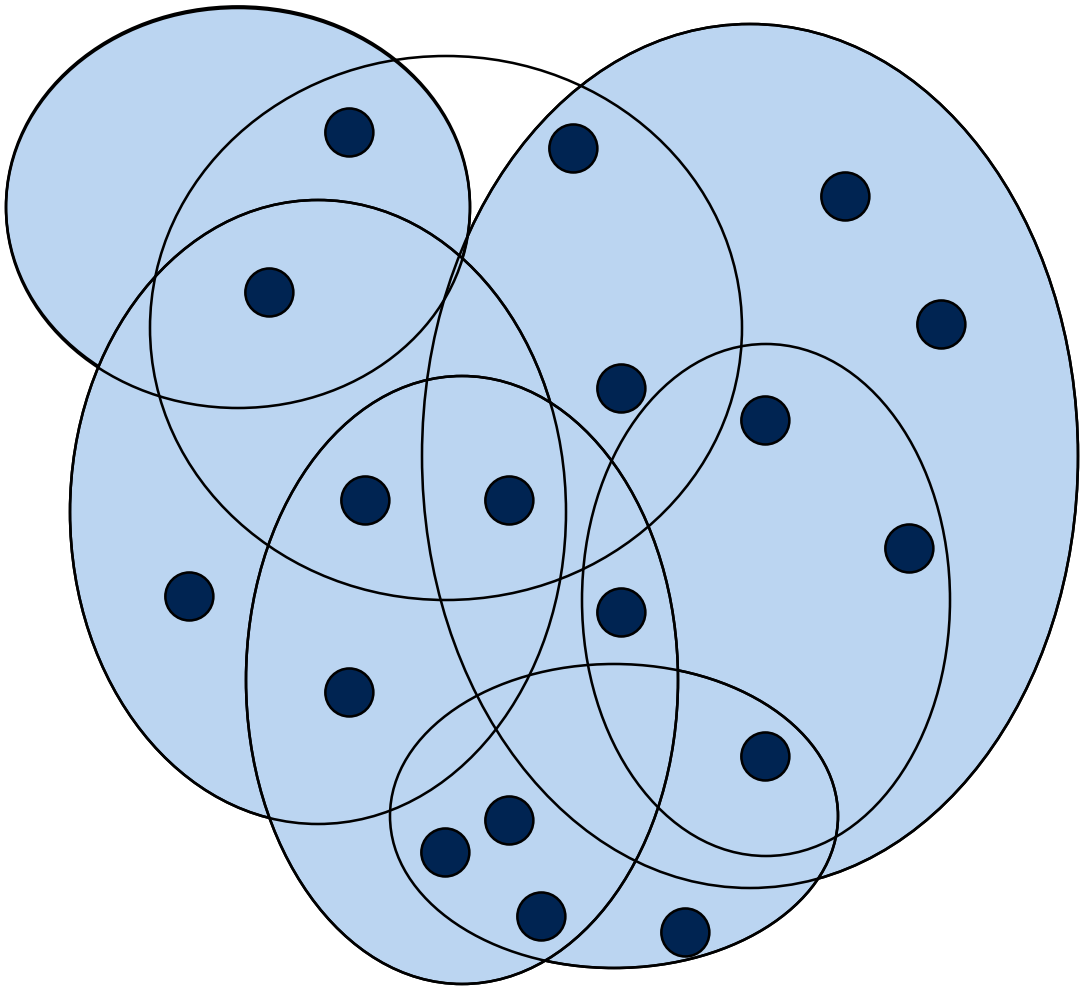
The Greedy Algorithm

- $C \leftarrow \emptyset$
- $U \leftarrow X$
- **while** $U \neq \emptyset$ **do**
  - select $S \in F$ that maximizes $|S \cap U|$
  - $C \leftarrow C \cup \{S\}$
  - $U \leftarrow U - S$
- return C

$O(|F| \cdot |X|)$

$\min\{|X|, |F|\}$

Example



5

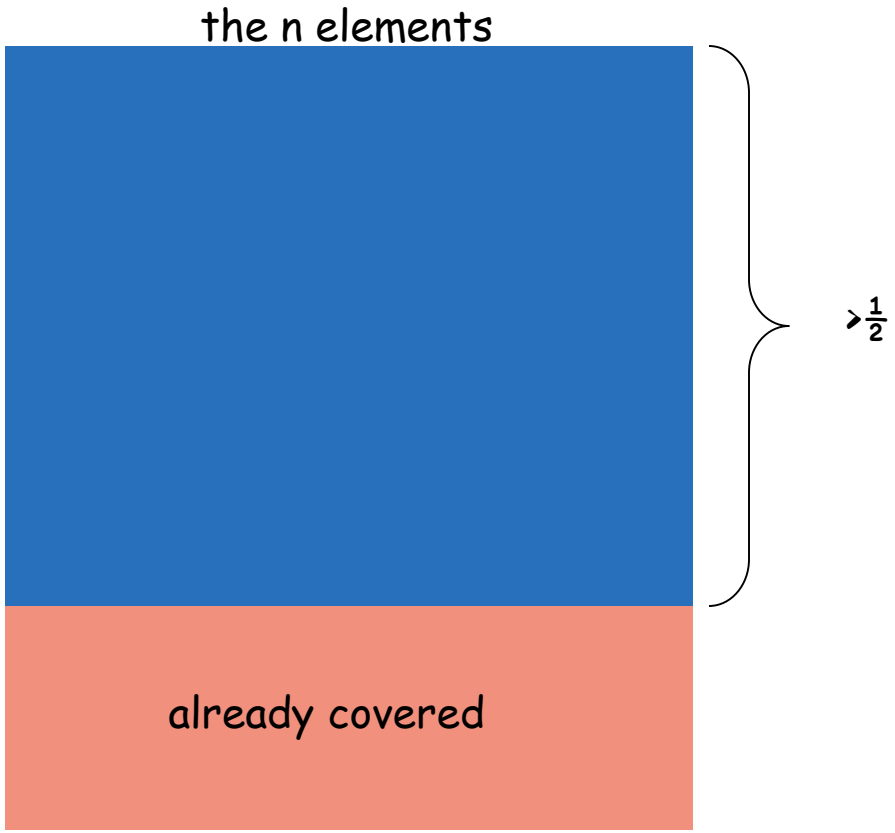
The Trick

- We'd like to compare the number of subsets returned by the greedy algorithm to the optimal
- The optimal is unknown, however, if it consists of k subsets, then any part of the universe can be covered by k subsets!

Loose Ratio-Bound

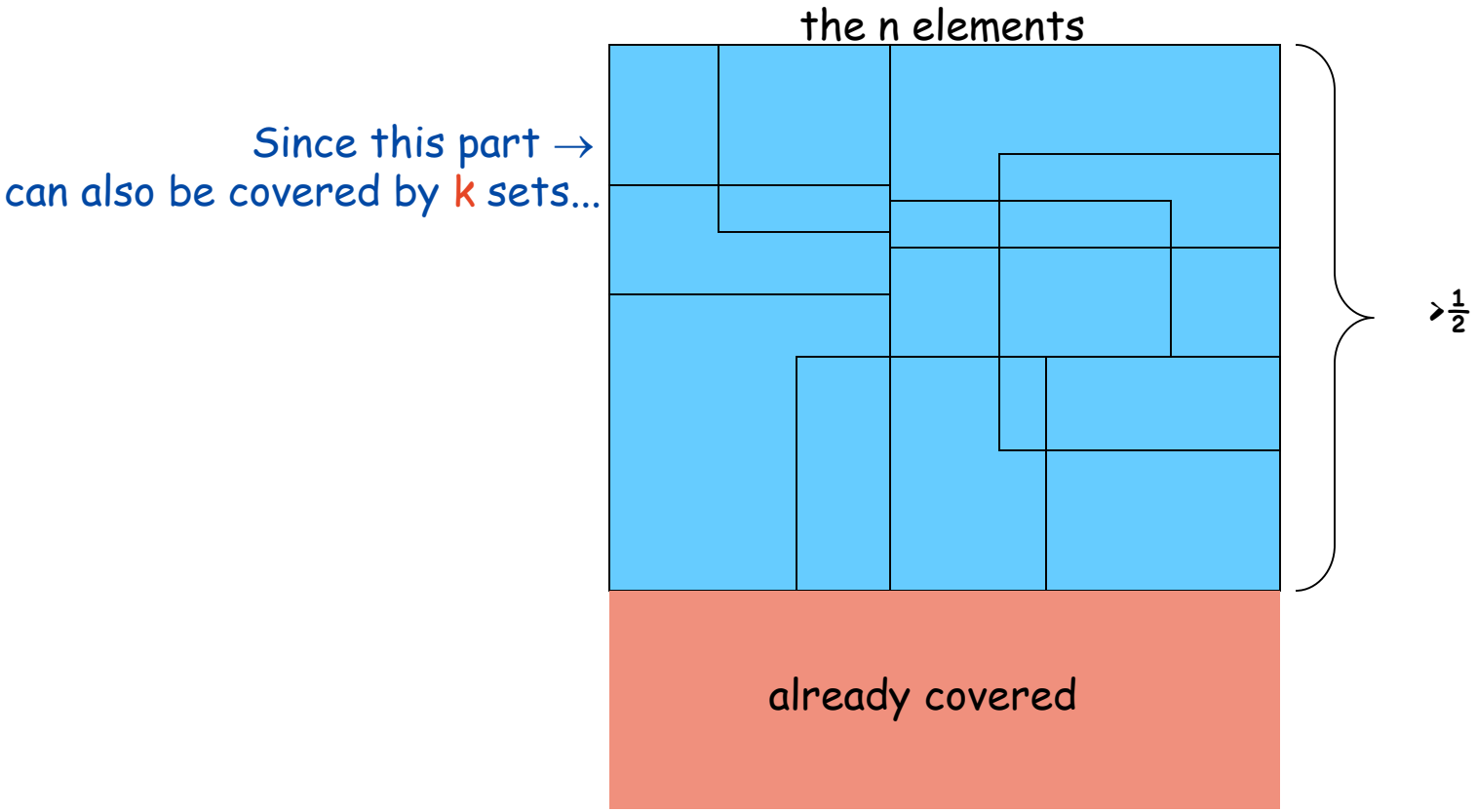
Claim: If $\exists$ cover of size k, then after k iterations the algorithm has covered at least $\frac{1}{2}$ of the elements

Assume the opposite and observe the situation after k iterations:



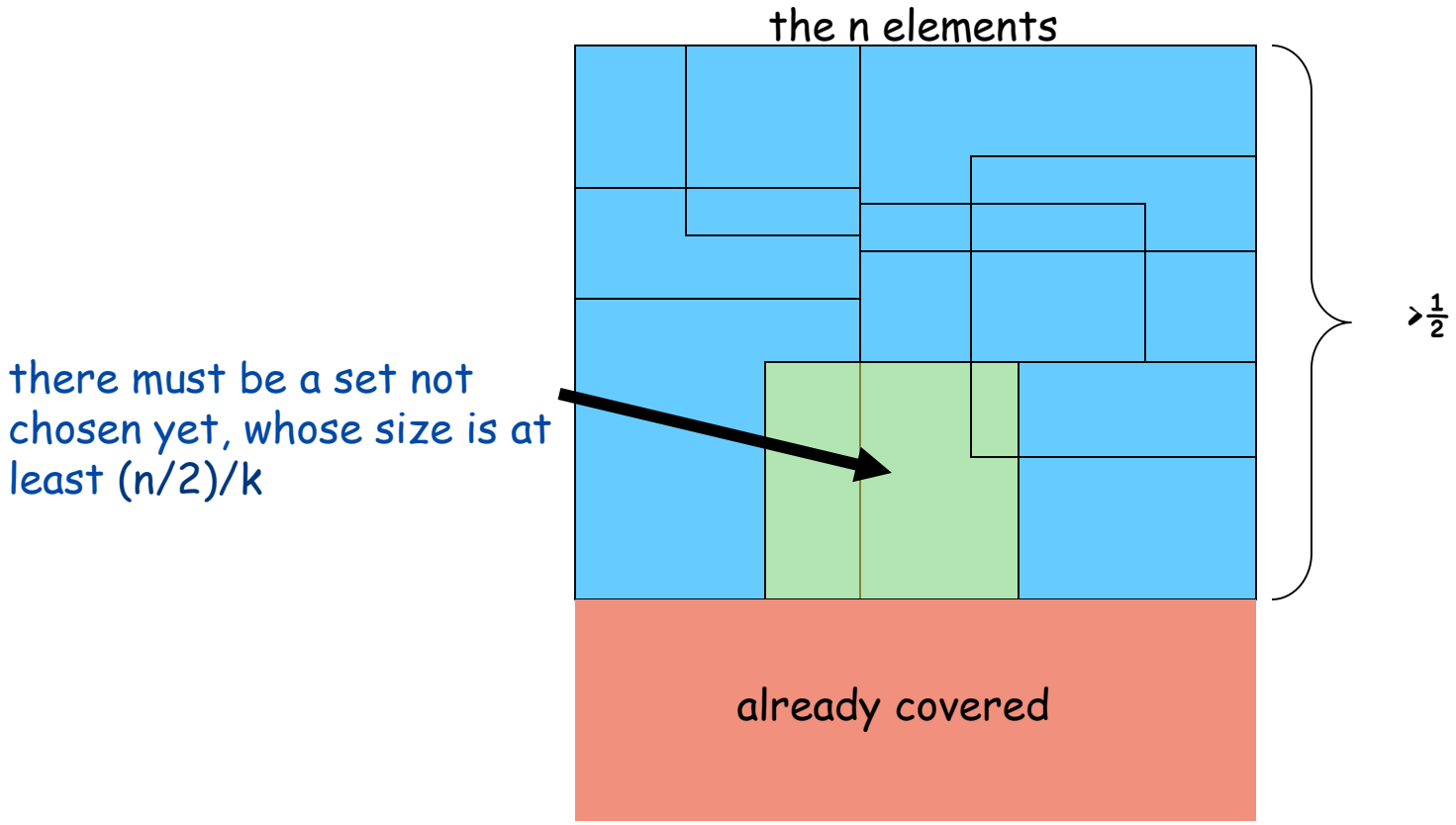
Loose Ratio-Bound

Claim: If ∃ cover of size k, then after k iterations the algorithm has covered at least ½ of the elements



Loose Ratio-Bound

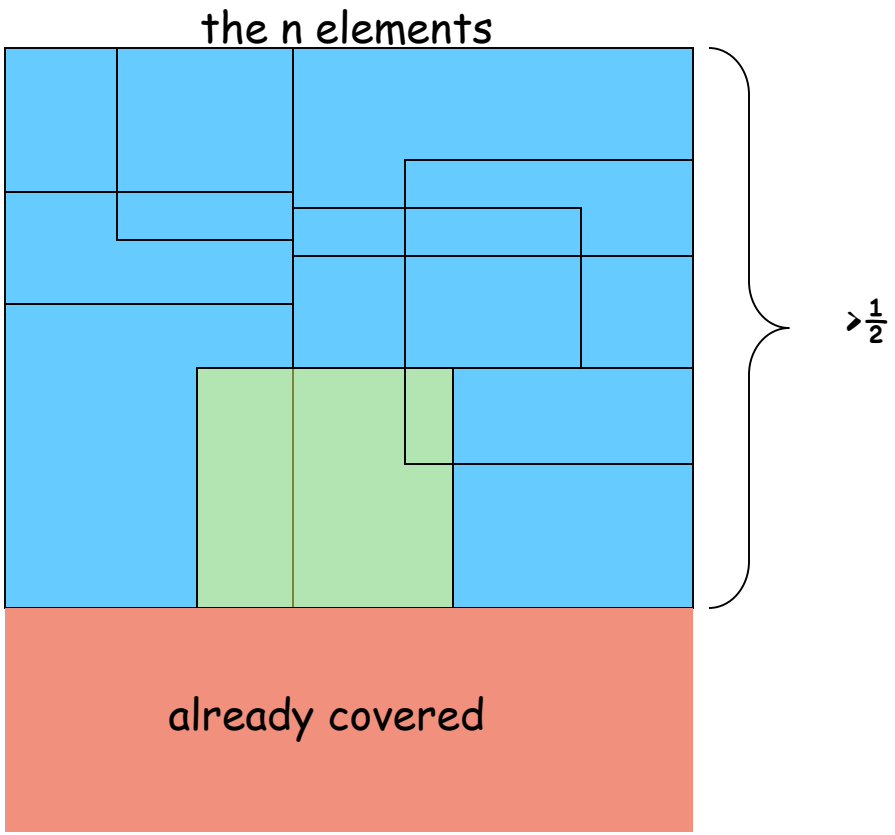
Claim: If ∃ cover of size k, then after k iterations the algorithm has covered at least ½ of the elements



Loose Ratio-Bound

Claim: If ∃ cover of size k, then after k iterations the algorithm has covered at least ½ of the elements

and the claim is proven!



Thus in each of the k iterations we've covered at least (n/2)/k new elements

Loose Ratio-Bound

Claim: If $\exists$ cover of size k, then after k iterations the algorithm covered at least $\frac{1}{2}$ of the elements.

How many times can we half the set? O(log n) times!

Each time we perform at most k iterations.

Therefore after k log n iterations (i.e - after choosing k log n sets) all the n elements must be covered, and the bound is proved.

■

Summary: Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Problems

- Euclidean TSP
- Set Cover
- ...