Amirkabir University of Technology
(Tehran Polytechnic)

Department of
Computer Engineering

Data Structure & Algorithms

Greedy Algorithms

Greedy Algorithms

- Like Dynamic Programming, greedy algorithms are used for optimization.
- The basic concept is that when we have a choice to make, make the one that looks best right now.
- That is, make a locally optimal choice in hope of getting a globally optimal solution.
- Greedy algorithms won't always yield an optimal solution, but sometimes, as in the activity selection problem, they do.

Common Situation for Greedy Problems

- a set (or a list) of candidates
- the set of candidates that have already been used
- a function that checks whether a particular set of candidates provides a solution to the problem
- a function to check feasibility
- a selection function that indicates the most promising candidate not yet used
- an objective function that gives the value of the solution

Basic Greedy Algorithm

function greedy (C: set) :	set	
S ←∅ while not solution (S) and	C ≠ ∅ do	

x ←	an	element	of	C	maximizing	select(x)
C ←	C	- {x}				

if	feasible	(S U {x})
if	then S ← S U {x} solution (S)	

then	return	S	
else	return	"no	solutions"

Idea: Kruskal's Algorithm

Central Idea:

- Grow a forest out of edges that do not grow a cycle, just like for the spanning tree problem.
- But now consider the edges in order by weight

Basic implementation:

- Sort edges by weight → O(|E| **`log`** |E|) = O(|E| **`log`** |V|)
- Iterate through edges using DSUF for cycle detection
  → O(|E| **`log`** |V|)

Somewhat better implementation:

- The algorithm to build min-heap with edges → O(|E|)
- Iterate through edges for cycle detection and deleteMin to get next edge → O(|E| **`log`** |V|)
- Not better worst-case asymptotically, but often stop long before considering all edges

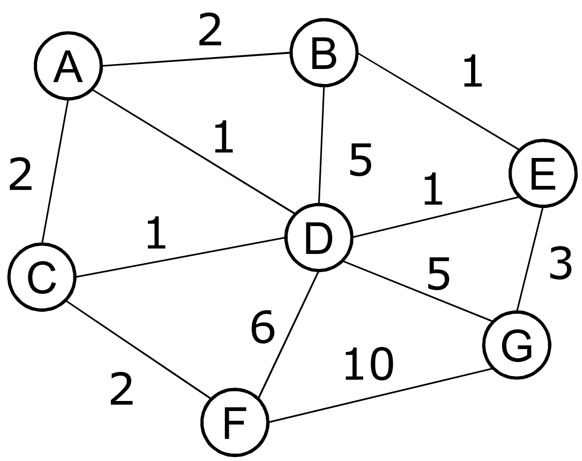
Pseudocode: Kruskal's Algorithm

1. Put edges in min-heap using edge weights
2. While output size < |V|-1
   a) Consider next smallest edge (u,v)
   b) if find(u,v) indicates u and v are in different sets
      - output (u,v)
      - union(u,v)

Recall invariant:

u and v in same set if and only if connected in output-so-far

Example: Kruskal's Algorithm



Edges in sorted order:

1: (A,D) (C,D) (B,E) (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

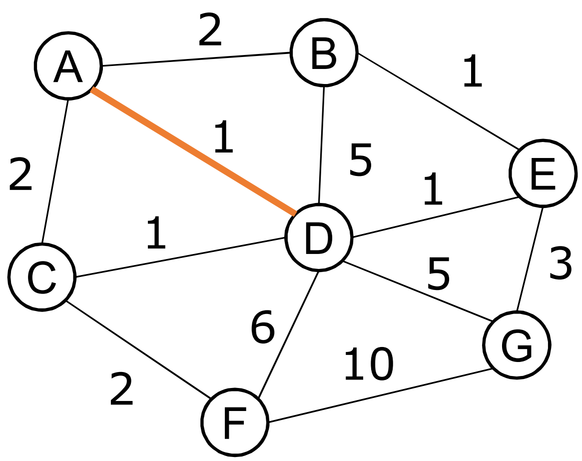
10: (F,G)

Sets: (A) (B) (C) (D) (E) (F) (G)

Output:

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D)~~ (C,D) (B,E) (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

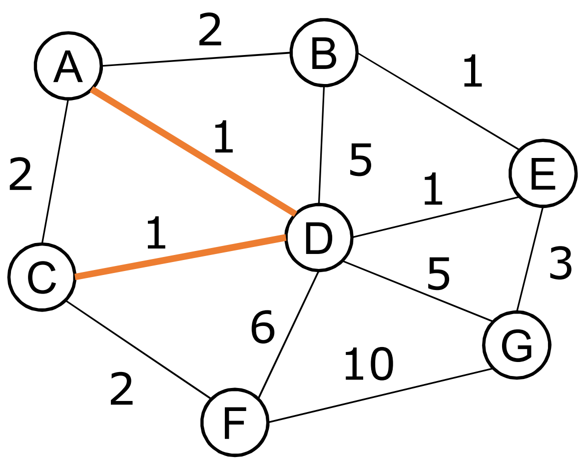
10: (F,G)

Sets: (A,D) (B) (C) (E) (F) (G)

Output: (A,D)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ (B,E) (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

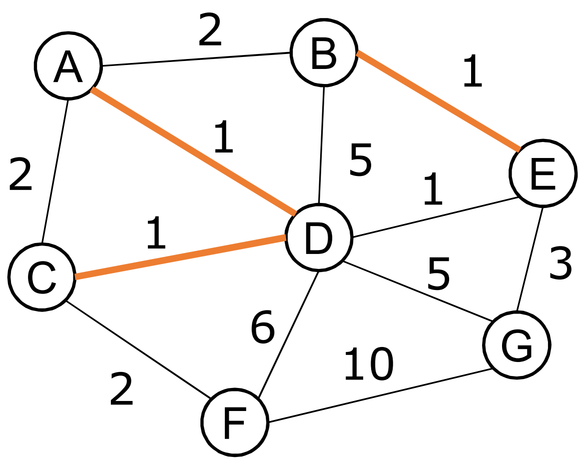
10: (F,G)

Sets: (A,C,D) (B) (E) (F) (G)

Output: (A,D) (C,D)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

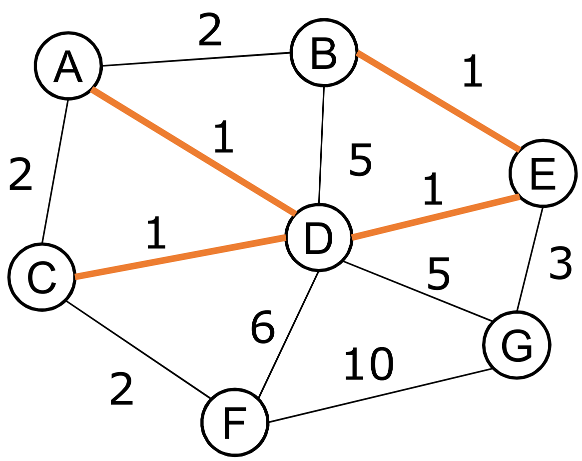
10: (F,G)

Sets: (A,C,D) (B,E) (F) (G)

Output: (A,D) (C,D) (B,E)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D) (C,D) (B,E) (D,E)~~

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

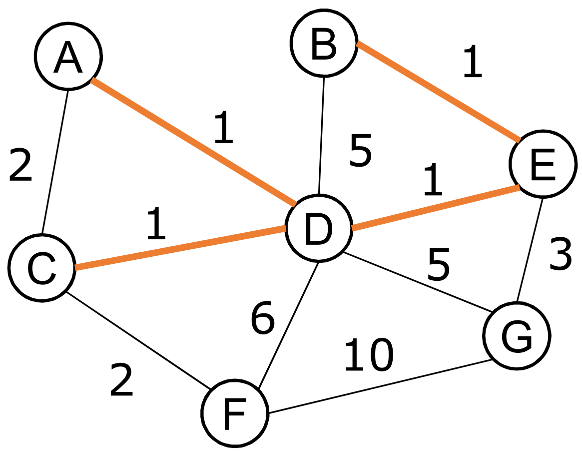
10: (F,G)

Sets: (A,B,C,D,E) (F) (G)

Output: (A,D) (C,D) (B,E) (D,E)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

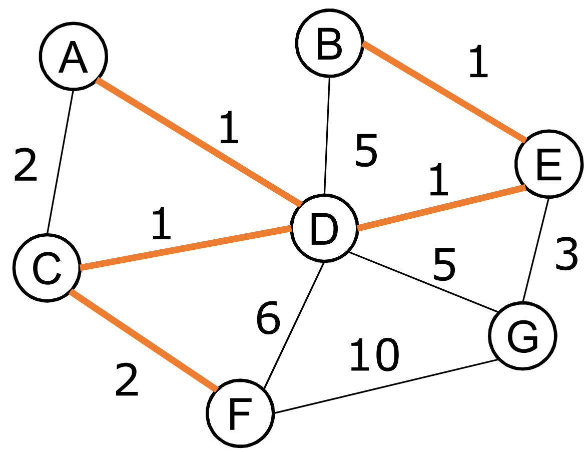
10: (F,G)

Sets: (A,B,C,D,E) (F) (G)

Output: (A,D) (C,D) (B,E) (D,E)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ ~~(C,F)~~ (A,C)

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

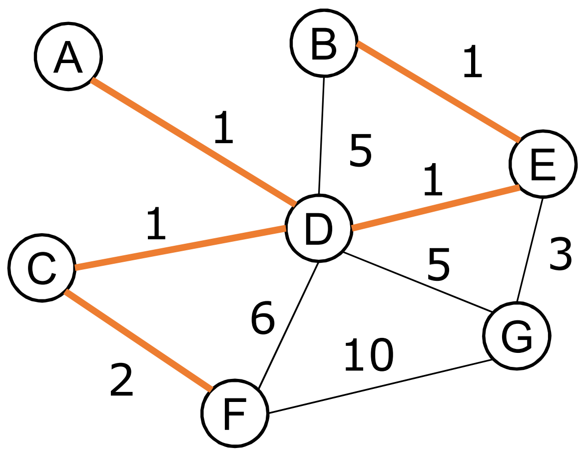
10: (F,G)

Sets: (A,B,C,D,E,F) (G)

Output: (A,D) (C,D) (B,E) (D,E) (C,F)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ ~~(C,F)~~ ~~(A,C)~~

3: (E,G)

5: (D,G) (B,D)

6: (D,F)

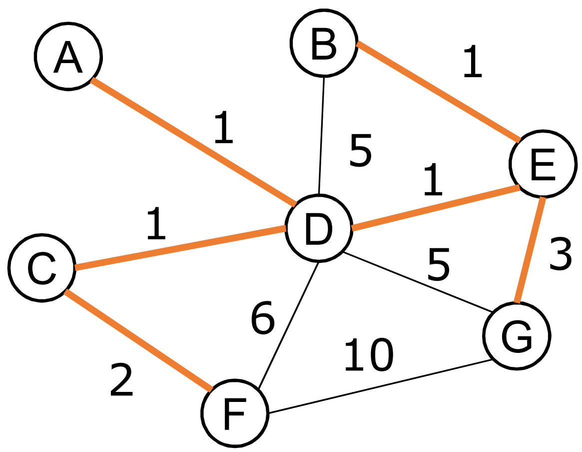
10: (F,G)

Sets: (A,B,C,D,E,F) (G)

Output: (A,D) (C,D) (B,E) (D,E) (C,F)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ ~~(C,F)~~ ~~(A,C)~~

3: ~~(E,G)~~

5: (D,G) (B,D)

6: (D,F)

10: (F,G)

Sets: (A,B,C,D,E,F,G)

Output: (A,D) (C,D) (B,E) (D,E) (C,F) (E,G)

At each step, the union/find sets are the trees in the forest

Analysis: Kruskal's Algorithm

Correctness: It is a spanning tree

- When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
- The graph is connected, we consider all edges

Correctness: That it is minimum weight

- Can be shown by induction
- At every step, the output is a subset of a minimum tree

Run-time

- $O(|E| \log |V|)$

Elements of the Greedy Strategy

- A greedy strategy results in an optimal solution for some problems, but for other problems it does not.
- There is no general way to tell if the greedy strategy will result in an optimal solution
- Two ingredients are usually necessary
  - greedy-choice property
  - optimal substructure

Greedy-choice Property

- *Greedy-choice property:* A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Unlike dynamic programming, we solve the problem in a top down manner.
- Must prove that the greedy choices result in a globally optimal solution.

Optimal Substructure

- *Optimal substructure:*
  A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

- Dynamic programming also requires that a problem have the property of optimal substructure.

Conclusion

- A greedy algorithm works by always making the best choice at the moment - a "locally optimal" choice.
- This means that we don't have to consider any previous choices, or worry about any of the choices ahead; we just pick the one that seems best at the moment.
- A greedy algorithm works well on a specific problem if the problem exhibits optimal substructure and has the *greedy-choice* property.
- If a problem does not have the *greedy-choice* property the greedy algorithm is not guaranteed to produce optimal results, but usually enables you to find a solution <u>faster</u> than algorithms that guarantee an optimum solution.