Algorithm Design and Analysis

Day 7
Greedy Algorithms

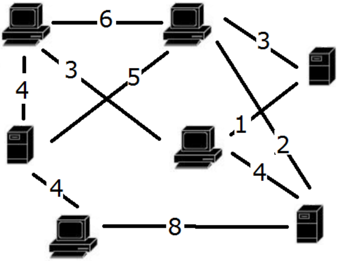
2015, AUT

Part I: Minimal Spanning Trees

A Typical Networking Problem (in 1950s)

Network a collection of computers while minimizing cost

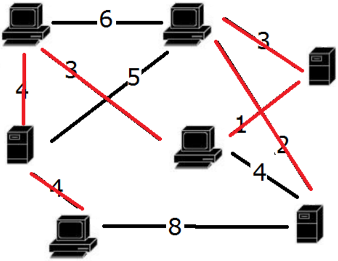
- Network is represented by an undirected graph
- Links are weighted by their set-up costs



A Typical Networking Problem (in 1950s)

Network a collection of computers while minimizing cost

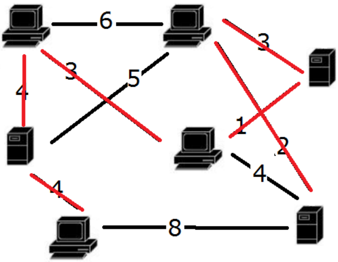
- Network is represented by an undirected graph
- Links are weighted by their set-up costs



A Typical Networking Problem (in 1950s)

Network a collection of computers while minimizing cost

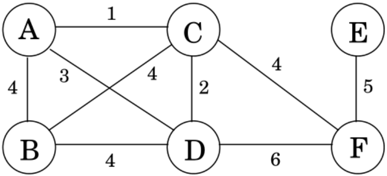
- Network is represented by an undirected graph
- Links are weighted by their set-up costs



- All computers must be connected
- We only use possible links
- Avoid cycles

Spanning Trees

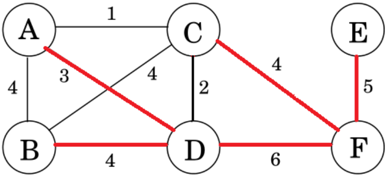
- A (undirected) graph G is connected if it has only one connected component



A connected graph

Spanning Trees

- A (undirected) graph G is connected if it has only one connected component
- A spanning tree of G is a connected subgraph that contains all nodes in V and no cycles.

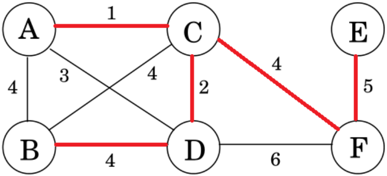


A spanning tree

(total weight = 22)

Spanning Trees

- A (undirected) graph G is connected if it has only one connected component
- A spanning tree of G is a connected subgraph that contains all nodes in V and no cycles.
- A minimal spanning tree of a weighted graph is a spanning tree whose total weight is minimal.

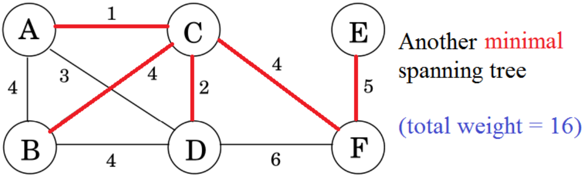


A minimal spanning tree

(total weight = 16)

Spanning Trees

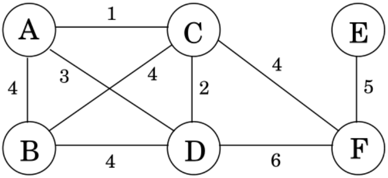
- A (undirected) graph G is connected if it has only one connected component
- A spanning tree of G is a connected subgraph that contains all nodes in V and no cycles.
- A minimal spanning tree of a weighted graph is a spanning tree whose total weight is minimal.
  Note: Minimal spanning trees may not be unique.



Minimal Spanning Tree

Minimal Spanning Tree (MST) Problem

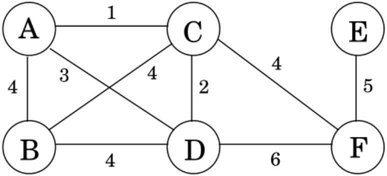
INPUT: A weighted connected undirected graph G
OUTPUT: A minimal spanning tree of G



Minimal Spanning Tree

Minimal Spanning Tree (MST) Problem

INPUT: A weighted connected undirected graph G
OUTPUT: A minimal spanning tree of G



Note: This is once again a tree construction problem, just like graph traversal and shortest path problem.

Optimisations in a Weighted Graph

Optimisation Problem

An optimisation problem contains a solution set where each solution has a value. The problem asks to find the solution with the maximal/minimal value (The optimal solution).

Optimisations in a Weighted Graph

Optimisation Problem

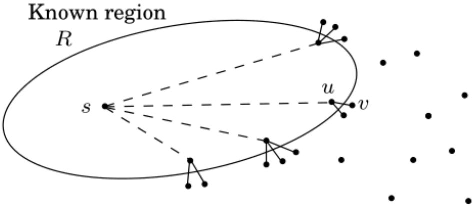
An optimisation problem contains a solution set where each solution has a value. The problem asks to find the solution with the maximal/minimal value (The optimal solution).

Optimised Tree Construction in Weighted Graphs

- **Goal:** Construct a tree in the graph that is the optimal solution
- **Optimal Substructure:** If S is an optimal solution, then any subpart of S is also an optimal solution.

Shortest Path Problem

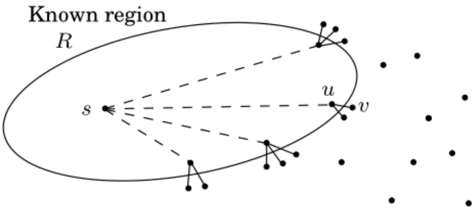
- **Goal**: Construct a tree in the graph that minimizes the distance from s to other nodes.



Optimisations in a Weighted Graph

Shortest Path Problem

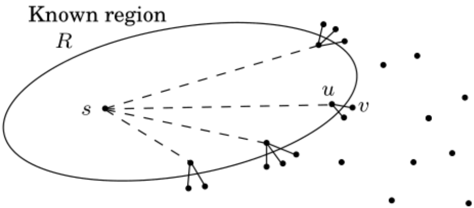
- **Goal**: Construct a tree in the graph that minimizes the distance from s to other nodes.
- **Optimal Substructure**: If $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path, then $s \rightsquigarrow u$ is a shortest path.



Optimisations in a Weighted Graph

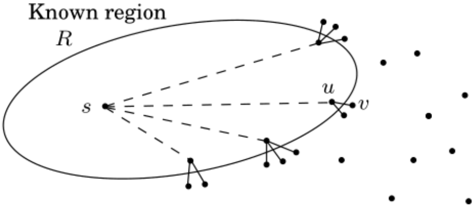
Shortest Path Problem

- **Goal**: Construct a tree in the graph that minimizes the distance from s to other nodes.
- **Optimal Substructure**: If $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path, then $s \rightsquigarrow u$ is a shortest path.
- **Greedy Choice**: Suppose R is the known region, and v is the node outside of R that minimises the current distance. Then we can add v into R in the next step.



MST Problem

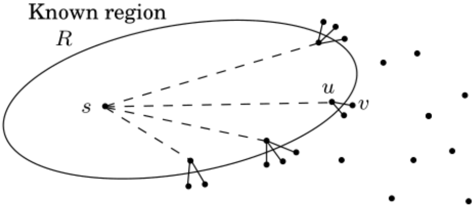
- **Goal**: Construct a tree in the graph that minimizes the overall weight of the tree.



Optimisations in a Weighted Graph

MST Problem

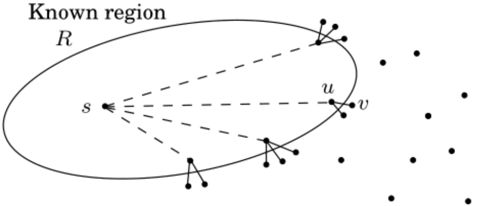
- **Goal**: Construct a tree in the graph that minimizes the overall weight of the tree.
- **Optimal Substructure**: If T is an MST of G, and v is a leaf in T, then after removing v from T, we obtain an MST in the subgraph of G with v removed.



Optimisations in a Weighted Graph

MST Problem

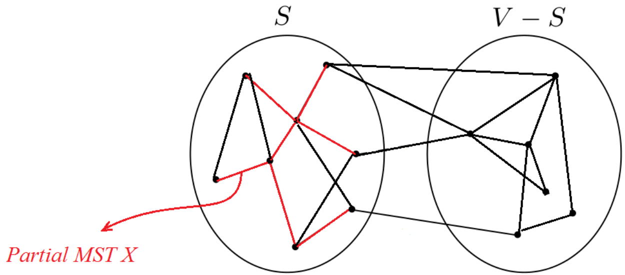
- **Goal**: Construct a tree in the graph that minimizes the overall weight of the tree.
- **Optimal Substructure**: If T is an MST of G, and v is a leaf in T, then after removing v from T, we obtain an MST in the subgraph of G with v removed.
- **Greedy Choice**: Can we obtain a similar property as for shortest path problem?



Cut Property

Cut Property (Version 1.0)

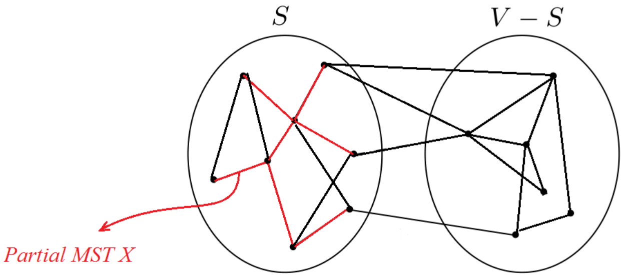
Let $G = (V, E, w)$ be a weighted graph. - We say a partial MST of G is a subtree that could lead to an MST.



Cut Property

Cut Property (Version 1.0)

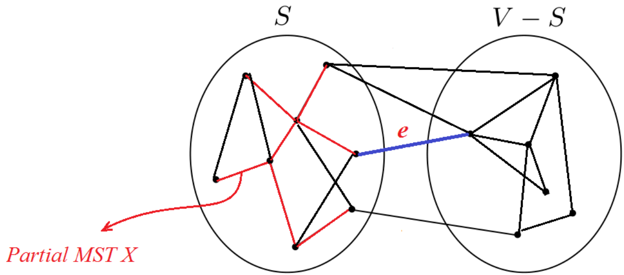
Let $G = (V, E, w)$ be a weighted graph. - We say a partial MST of G is a subtree that could lead to an MST.
- Suppose we have constructed a partial MST X on a set of nodes S.



Cut Property

Cut Property (Version 1.0)

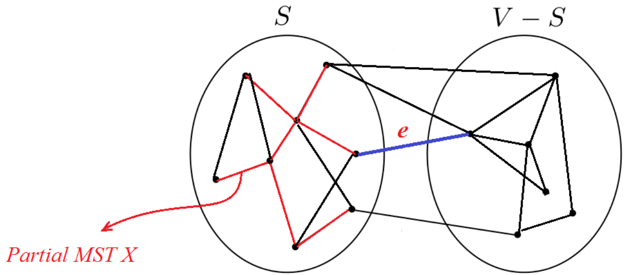
Let $G = (V, E, w)$ be a weighted graph. - We say a partial MST of G is a subtree that could lead to an MST.
- Suppose we have constructed a partial MST X on a set of nodes S.
- Let e be the lightest edge across the partition between S and $V - S$.



Cut Property

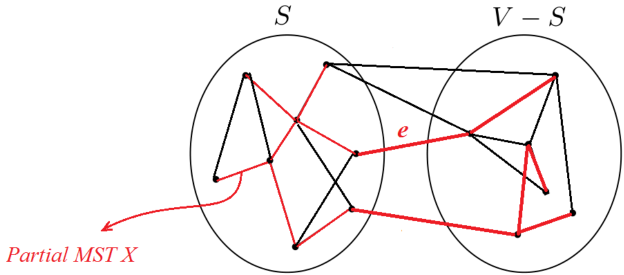
Cut Property (Version 1.0)

Let $G = (V, E, w)$ be a weighted graph. - We say a partial MST of G is a subtree that could lead to an MST.
- Suppose we have constructed a partial MST X on a set of nodes S.
- Let e be the lightest edge across the partition between S and $V - S$.
- Then $X \cup \{e\}$ is also a partial MST.



Cut Property

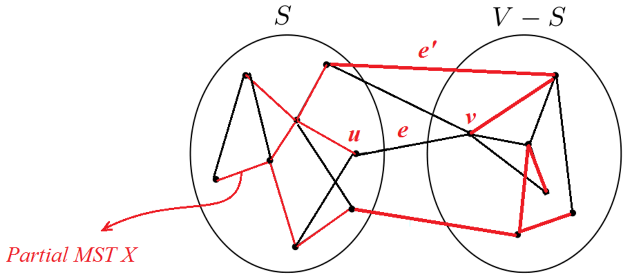
Why does cut property hold?



Since X is a partial MST, there is an MST T that contains X.
Suppose T contains e. Then we are done.

Cut Property

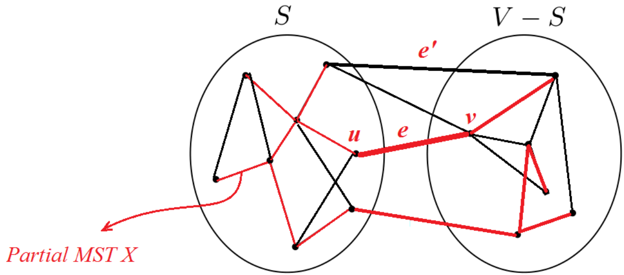
Why does cut property hold?



Since X is a partial MST, there is an MST T that contain X.
Suppose T does not contain $e = (u, v)$.
Say T uses e' in the $(S, V - S)$-partition to connect to v.

Cut Property

Why does cut property hold?



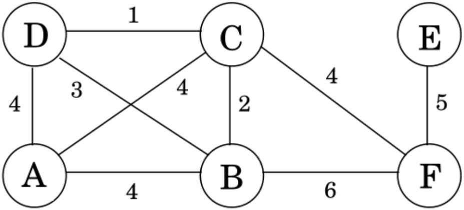
Since X is a partial MST, there is an MST T that contain X.
Suppose T does not contain $e = (u, v)$.
Say T uses e' in the $(S, V - S)$-partition to connect to v.
Then $T - \{e'\} \cup \{e\}$ is an MST.
So $X \cup \{e\}$ is a partial MST.

Prim's Algorithm

Invented by *Vojtěch Jarnik* in 1930s, then by *Robert Prim* in 1957.

Idea: Find MST in a similar way as Dijkstra's algorithm.

- Maintain a known region
- Maintain $prev(u)$ for every node u to store the tree.
- Maintain a priority queue storing the candidate edges weights.



Priority Queue P

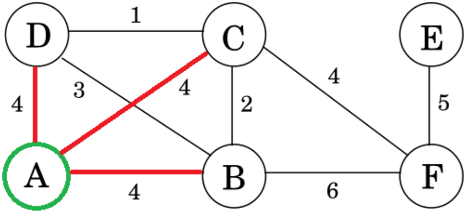
A	B	C	D	E	F
0	∞	∞	∞	∞	∞

Prim's Algorithm

Invented by *Vojtěch Jarnik* in 1930s, then by *Robert Prim* in 1957.

Idea: Find MST in a similar way as Dijkstra's algorithm.

- Maintain a known region
- Maintain $prev(u)$ for every node u to store the tree.
- Maintain a priority queue storing the candidate edges weights.



Priority Queue P

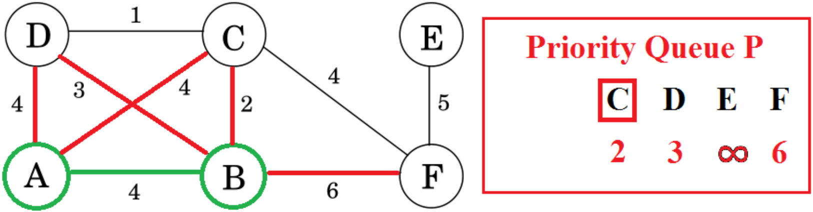
$\boxed{B}$	C	D	E	F
4	4	4	∞	∞

Prim's Algorithm

Invented by *Vojtěch Jarnik* in 1930s, then by *Robert Prim* in 1957.

Idea: Find MST in a similar way as Dijkstra's algorithm.

- Maintain a known region
- Maintain *prev*(*u*) for every node *u* to store the tree.
- Maintain a priority queue storing the candidate edges weights.

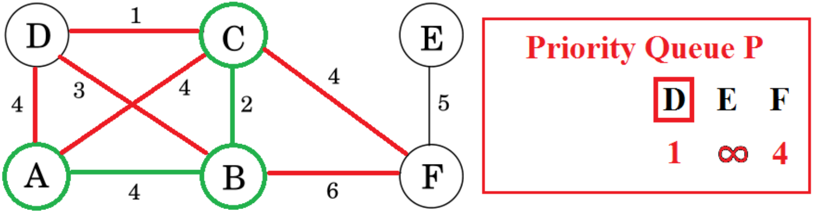


Prim's Algorithm

Invented by *Vojtěch Jarnik* in 1930s, then by *Robert Prim* in 1957.

Idea: Find MST in a similar way as Dijkstra's algorithm.

- Maintain a known region
- Maintain $prev(u)$ for every node u to store the tree.
- Maintain a priority queue storing the candidate edges weights.

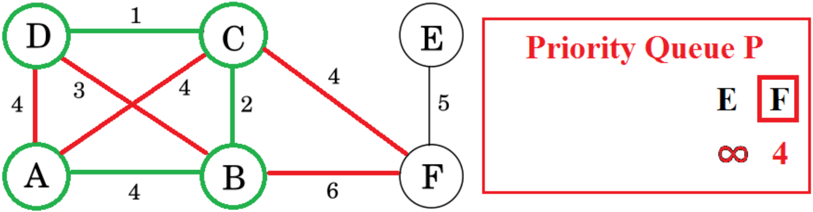


Prim's Algorithm

Invented by *Vojtěch Jarnik* in 1930s, then by *Robert Prim* in 1957.

Idea: Find MST in a similar way as Dijkstra's algorithm.

- Maintain a known region
- Maintain $prev(u)$ for every node u to store the tree.
- Maintain a priority queue storing the candidate edges weights.

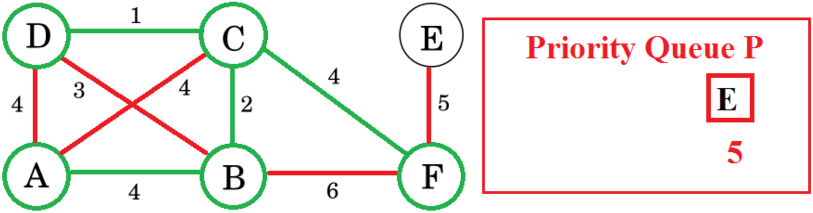


Prim's Algorithm

Invented by *Vojtěch Jarnik* in 1930s, then by *Robert Prim* in 1957.

Idea: Find MST in a similar way as Dijkstra's algorithm.

- Maintain a known region
- Maintain *prev*(u) for every node u to store the tree.
- Maintain a priority queue storing the candidate edges weights.

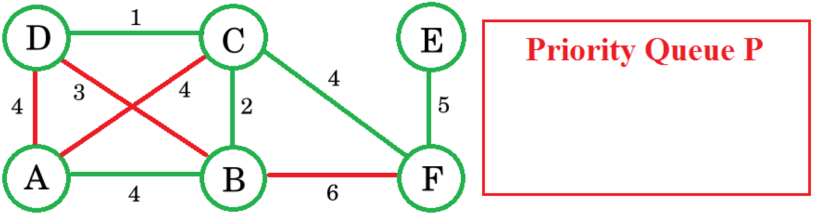


Prim's Algorithm

Invented by *Vojtěch Jarnik* in 1930s, then by *Robert Prim* in 1957.

Idea: Find MST in a similar way as Dijkstra's algorithm.

- Maintain a known region
- Maintain $prev(u)$ for every node u to store the tree.
- Maintain a priority queue storing the candidate edges weights.



Prim's Algorithm

Algorithm MST_Prim:

INPUT: A weighted graph $G = (V, E, w)$
OUTPUT: $prev(v)$ for every node $v \in V$ indicating an MST

1. Let s be the first node in V.
2. Initialize a known region $R \leftarrow \{s\}$
3. Initialize a priority queue P containing $(s, 0)$
4. `for` $u \in V, u \neq s$ `do`
    - $prev(u) \leftarrow null$
    - $value(u) \leftarrow \infty$
    - $P.Insert(u, \infty)$
5. `while` P is not empty `do`
    - $u \leftarrow P.DeleteMin()$
    - Add u to R
    - `for` $(u, v) \in E$ where $v \notin R$ `do`
        - `if` $weight(u, v) < value(v)$ `then`
            - $value(v) \leftarrow weight(u, v)$
            - $P.DecreaseKey(v, value(v))$
            - $prev(v) \leftarrow u$

Prim's Algorithm

Theorem

Let $G = (V, E, w)$ be a weighted graph. After running MST_Prim(V, E, w) the tree constructed (as represented by the *prev* pointers) is an MST.

Prim's Algorithm

Theorem

Let $G = (V, E, w)$ be a weighted graph. After running MST_Prim(V, E, w) the tree constructed (as represented by the *prev* pointers) is an MST.

Complexity Analysis

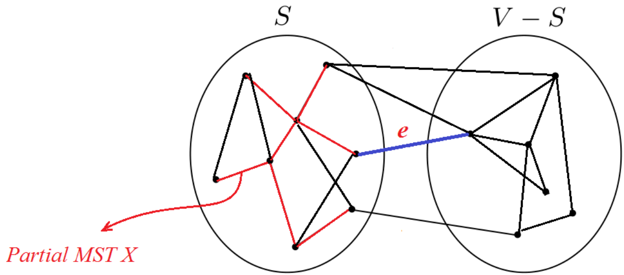
Prim's algorithm runs in exactly the same asymptotic time as Dijkstra's algorithm:
Depending on the implementations of priority queues:

- Lists: $O(n^2)$
- Binary heap/Binomial heap: $O((m + n) \log n)$
- Fibonacci heap: $O(n \log n + m)$

Cut Property Revisited

Cut Property (Version 1.0)

Let $G = (V, E, w)$ be a weighted graph. - We say a partial MST of G is a subtree that could lead to an MST.
- Suppose we have constructed a partial MST X on a subset $S \subseteq V$.
- Let e be the lightest edge across the partition between S and $V - S$.
- Then $X \cup \{e\}$ is also a partial MST.

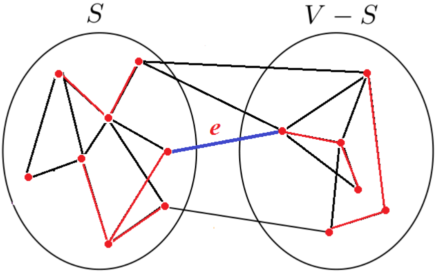


Cut Property Revisited

Cut Property (Version 2.0)

Let $G = (V, E, w)$ be a weighted graph. - We say a partial minimal spanning forest of G is a subset of edges that could lead to an MST.

Instead of Partial MST, we could allow *partial minimal spanning forest*

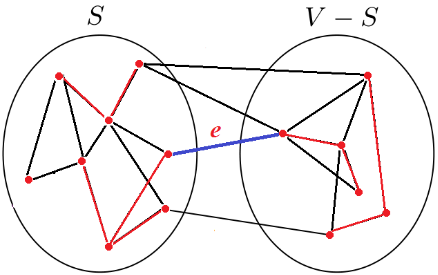


Cut Property Revisited

Cut Property (Version 2.0)

Let $G = (V, E, w)$ be a weighted graph. - We say a partial minimal spanning forest of G is a subset of edges that could lead to an MST.
- Suppose we have constructed a partial MSF X.

Instead of Partial MST, we could allow *partial minimal spanning forest*

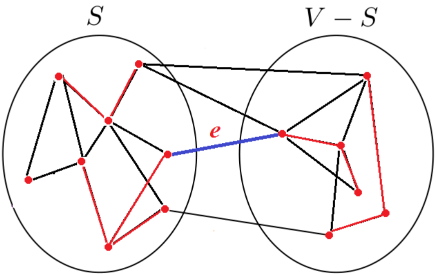


Cut Property Revisited

Cut Property (Version 2.0)

Let $G = (V, E, w)$ be a weighted graph. - We say a partial minimal spanning forest of G is a subset of edges that could lead to an MST.
- Suppose we have constructed a partial MSF X.
- Let e be the lightest edge across any two trees in this partial MSF.

Instead of Partial MST, we could allow *partial minimal spanning forest*

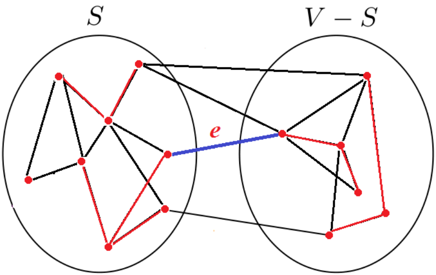


Cut Property Revisited

Cut Property (Version 2.0)

Let $G = (V, E, w)$ be a weighted graph. - We say a partial minimal spanning forest of G is a subset of edges that could lead to an MST.
- Suppose we have constructed a partial MSF X.
- Let e be the lightest edge across any two trees in this partial MSF.
- Then $X \cup \{e\}$ is also a partial MSF.

Instead of Partial MST, we could allow *partial minimal spanning forest*



In Pursuit of Elegance

The version 2.0 of the cut property allows us to conceptually simplify Prim's algorithm:

- We do not need to make a traversal.
- In other words, we do not need to keep the known region connected.
- Eventually, all the disconnected parts will link together to form an MST.

In Pursuit of Elegance

The version 2.0 of the cut property allows us to conceptually simplify Prim's algorithm:

- We do not need to make a traversal.
- In other words, we do not need to keep the known region connected.
- Eventually, all the disconnected parts will link together to form an MST.

Strategy

Add edges into the MSF one-by-one:
- In increasing order of the weights
- Make sure no cycle is created

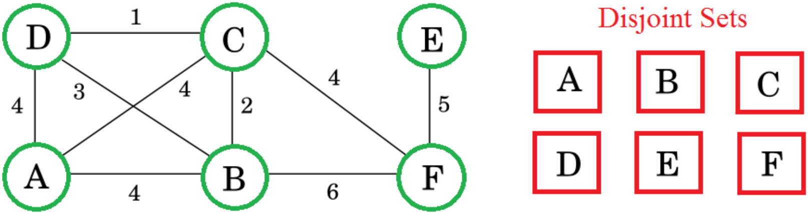
Kruskal's Algorithm

Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)



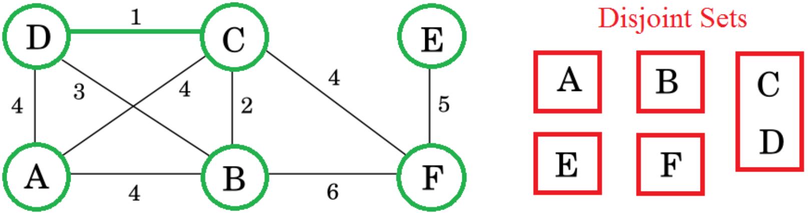
Kruskal's Algorithm

Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)



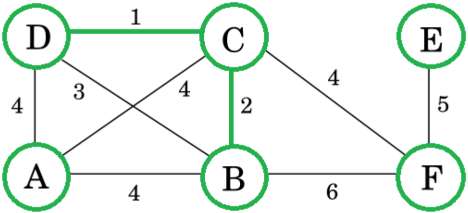
Kruskal's Algorithm

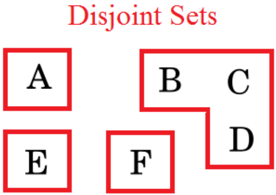
Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)





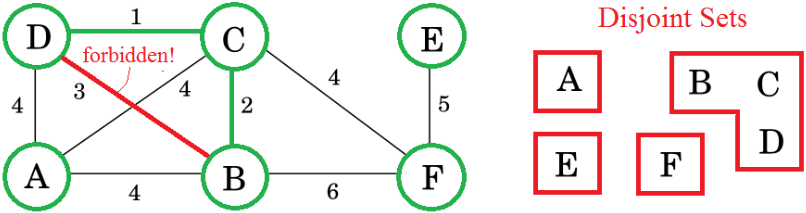
Kruskal's Algorithm

Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)



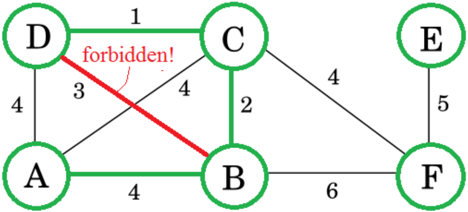
Kruskal's Algorithm

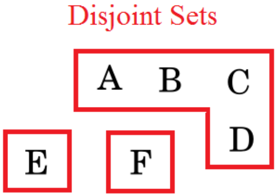
Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)





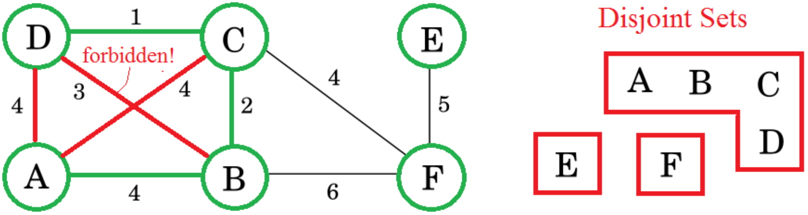
Kruskal's Algorithm

Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)



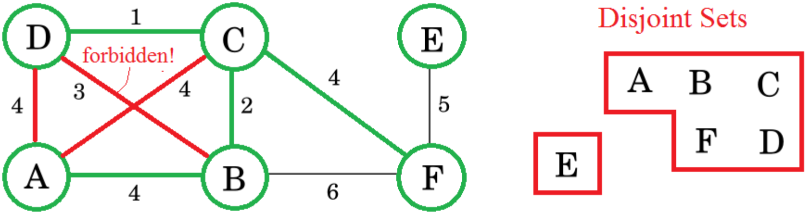
Kruskal's Algorithm

Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)



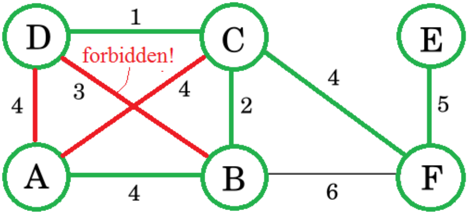
Kruskal's Algorithm

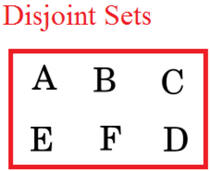
Invented by Joseph Kruskal in 1956.

Strategy

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created (using a disjoint sets data structure)





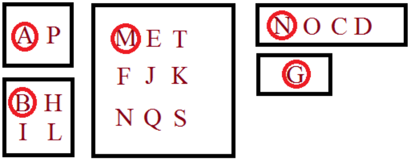
Disjoint-Sets

The disjoint-set data structure is used for identifying forbidden edges.

Disjoint-Sets

A disjoint-sets data structure maintains a collection of disjoint sets such that each set has a unique representative element and supports the following operations:

- MakeSet(u): Make a new set containing element u.
- Union(u, v): Merge the sets containing u and v.
- Find(u): Return the representative element of the set that contains u



Kruskal's Algorithm

Algorithm MST_Kruskal:

INPUT: A weighted undirected graph $G = (V, E, w)$
OUTPUT: A set X of edges representing an MST

Sort edges in E in increasing weights
Store the sorted edges in a list called SortedEdges
Initialize a disjoint-sets data structure D with each node a separate set
Initialize an empty set of edges X

```
for each {u, v} ∈ SortedEdges do
    if D.find(u) ≠ D.find(v) then
        X ← X ∪ {{u, v}}
        D.union(u, v)
return X
```

Kruskal's Algorithm: Complexity

Complexity Analysis

The running time of Kruskal's algorithm depends on

- The complexity of the sorting algorithm
- The complexity of union-find operations

Kruskal's Algorithm: Complexity

Complexity Analysis

The running time of Kruskal's algorithm depends on

- The complexity of the sorting algorithm
- The complexity of union-find operations

Define:
- $T_{sort}(x)$ = time to sort x elements
- $T_{find}(x)$ = time to find an element
- $T_{union}(x)$ = time to take the *union* of two sets

Kruskal's Algorithm: Complexity

Complexity Analysis

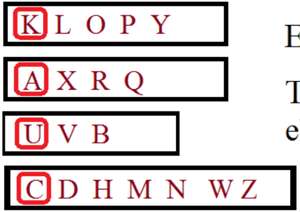
The running time of Kruskal's algorithm depends on

- The complexity of the sorting algorithm
- The complexity of union-find operations

Define:
- $T_{sort}(x)$ = time to sort x elements
- $T_{find}(x)$ = time to find an element
- $T_{union}(x)$ = time to take the *union* of two sets

Running time for MST_Kruskal: $O(T_{sort}(m) + T_{find}(x)m + T_{union}(x)n)$

Disjoint-Sets: Lists



Each set is presented by an array.

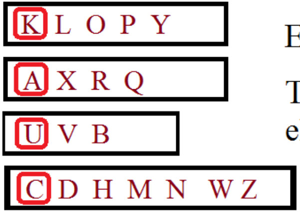
The representative is the first element

- Union:
- Find:

Therefore the running time of Kruskal's algorithm: .

Disjoint-Sets: Implementation 1

Disjoint-Sets: Lists



Each set is presented by an array.

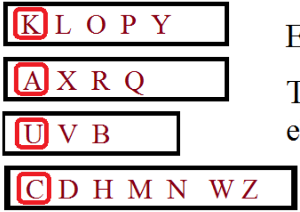
The representative is the first element

- Union: $O(1)$
- Find: Need to go through the list $O(n)$

Therefore the running time of Kruskal's algorithm: .

Disjoint-Sets: Implementation 1

Disjoint-Sets: Lists



Each set is presented by an array.

The representative is the first element

- Union: $O(1)$
- Find: Need to go through the list $O(n)$

Therefore the running time of Kruskal's algorithm: $T_{sort}(m) + O(mn)$.

Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Each set is represented by a tree. The representative is the root.
- Each node is associated with a rank, i.e., the height of its subtree.
- Union: link two trees; point the root with lower rank to the root with higher rank.
- find: follow parent pointers to find the root.

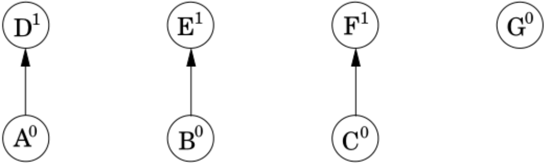
$\texttt{makeset}(A), \texttt{makeset}(B), \ldots, \texttt{makeset}(G)$:

Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Each set is represented by a tree. The representative is the root.
- Each node is associated with a rank, i.e., the height of its subtree.
- Union: link two trees; point the root with lower rank to the root with higher rank.
- find: follow parent pointers to find the root.

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

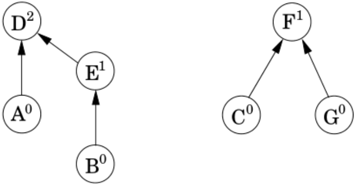


Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Each set is represented by a tree. The representative is the root.
- Each node is associated with a rank, i.e., the height of its subtree.
- Union: link two trees; point the root with lower rank to the root with higher rank.
- find: follow parent pointers to find the root.

union(C, G), union(E, A):

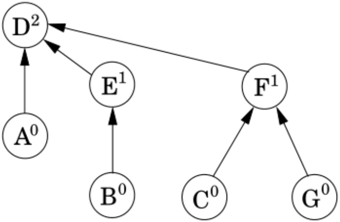


Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Each set is represented by a tree. The representative is the root.
- Each node is associated with a rank, i.e., the height of its subtree.
- Union: link two trees; point the root with lower rank to the root with higher rank.
- find: follow parent pointers to find the root.

union(B, G):



Disjoint-Set

Disjoint-Sets: Trees

- Union: link two trees; point the root with lower rank to the root with higher rank.

  Running time: $O(1)$
- find: follow parent pointers to find the root.

  Running time: Depend on the height (rank) of the tree.

Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Fact 1. A node with rank k must have at least 2^k nodes in its subtree.

Disjoint-Sets: Trees

- Fact 1. A node with rank k must have at least 2^k nodes in its subtree.

  Why? Proved by induction on k.

Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Fact 1. A node with rank k must have at least 2^k nodes in its subtree.

  Why? Proved by induction on k.

- Fact 2. Let k be the largest rank. There can be at most $n/2^k$ nodes of rank k.

Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Fact 1. A node with rank k must have at least 2^k nodes in its subtree.

  Why? Proved by induction on k.

- Fact 2. Let k be the largest rank. There can be at most $n/2^k$ nodes of rank k.

$\Rightarrow$ the largest rank k is at most $\log n$.

$\Rightarrow$ $Find(u)$ takes time $O(\log n)$.

Disjoint-Sets: Implementation 2

Disjoint-Sets: Trees

- Fact 1. A node with rank k must have at least 2^k nodes in its subtree.

  Why? Proved by induction on k.

- Fact 2. Let k be the largest rank. There can be at most $n/2^k$ nodes of rank k.

$\Rightarrow$ the largest rank k is at most $\log n$.

$\Rightarrow$ $Find(u)$ takes time $O(\log n)$.

Therefore Kruskal's algorithm takes time $T_{sort}(m) + O(m \log n)$.

Kruskal's Algorithm

Different Disjoint-Sets

Kruskal's algorithm has different running time for different disjoint-set implementations:

- Arrays:
- Trees:

Kruskal's Algorithm

Different Disjoint-Sets

Kruskal's algorithm has different running time for different disjoint-set implementations:

- Arrays: $T_{sort}(m) + O(mn)$
- Trees: $T_{sort}(m) + O(m \log n)$
- Trees with Path Compression: $T_{sort}(m) + O(m \log^* n)$, where $\log^* n$ is $O(\underbrace{\log \log \ldots \log}_{k} n)$ for any $k > 0$.

  (typically called the iterated logarithmic function.)

Part II: Being Greedy as an Algorithm Design Technique

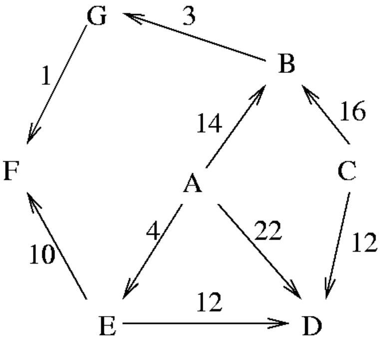
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Dijkstra's: Always choose the node with lowest estimated distance.



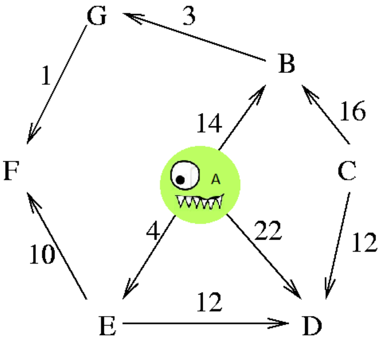
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Dijkstra's: Always choose the node with lowest estimated distance.



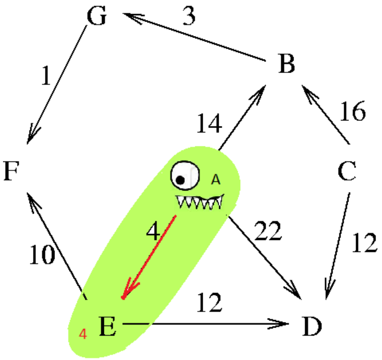
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Dijkstra's: Always choose the node with lowest estimated distance.



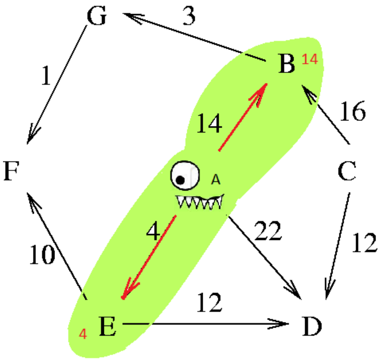
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Dijkstra's: Always choose the node with lowest estimated distance.

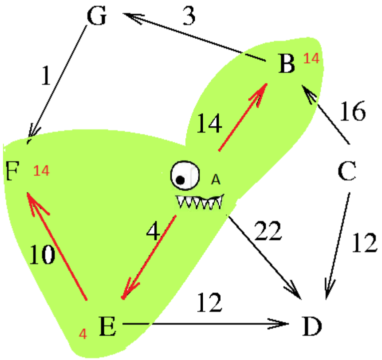


Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Dijkstra's: Always choose the node with lowest estimated distance.



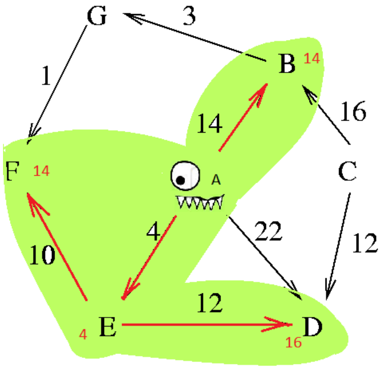
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Dijkstra's: Always choose the node with lowest estimated distance.



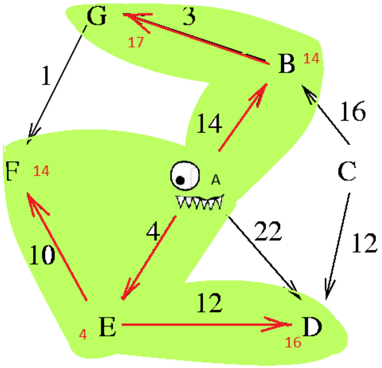
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Dijkstra's: Always choose the node with lowest estimated distance.



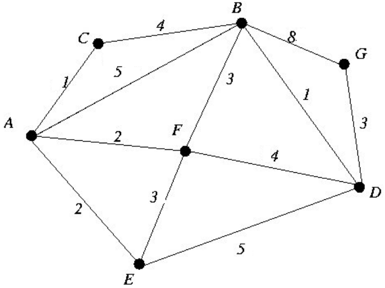
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



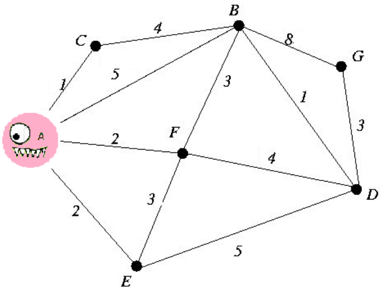
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



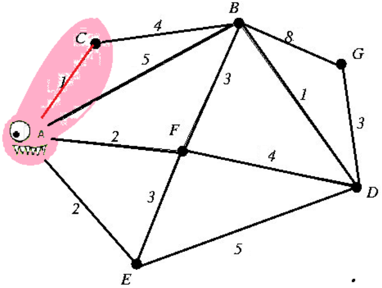
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



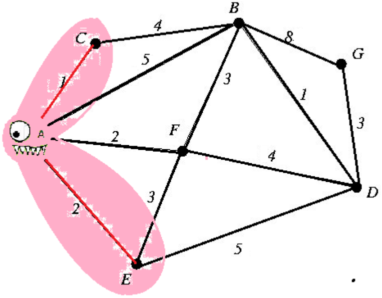
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



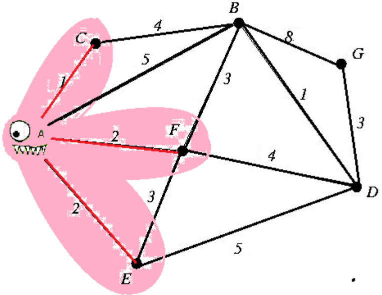
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



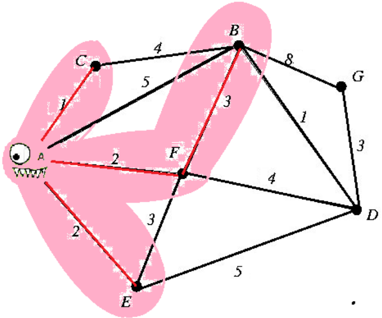
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



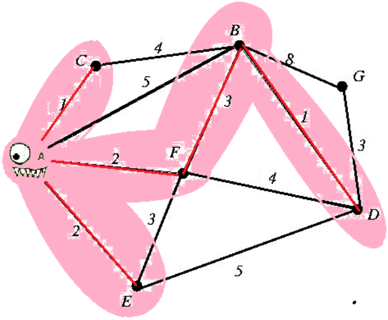
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



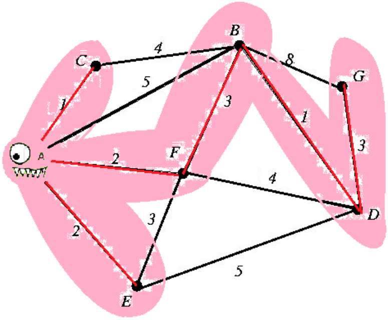
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Prim's: Always choose the edge with lowest weight and connected to the known region.



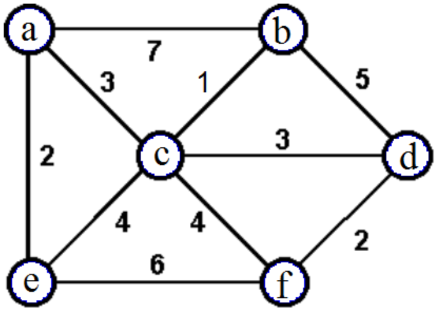
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Kruskal's: Always choose the edge with lowest weight and not form a cycle.



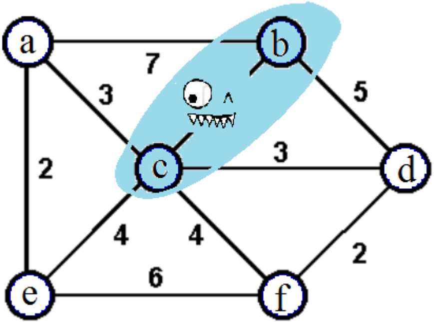
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Kruskal's: Always choose the edge with lowest weight and not form a cycle.



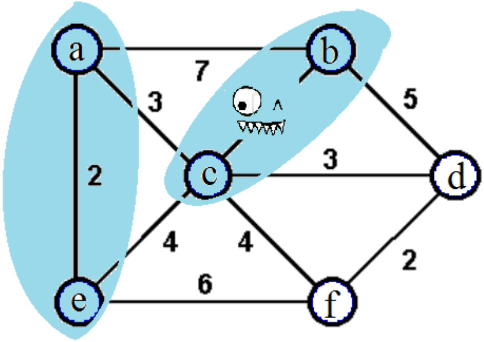
Dijkstra, Prim and Kruskal

Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Kruskal's: Always choose the edge with lowest weight and not form a cycle.

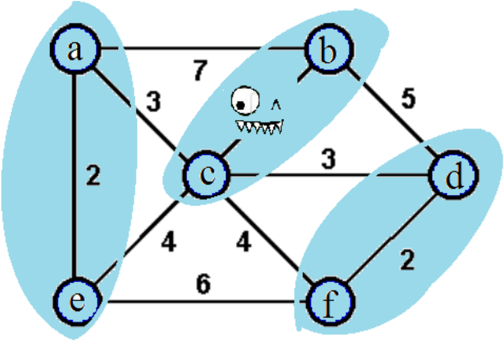


Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Kruskal's: Always choose the edge with lowest weight and not form a cycle.

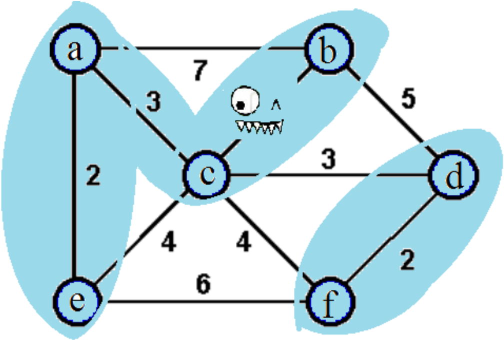


Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Kruskal's: Always choose the edge with lowest weight and not form a cycle.

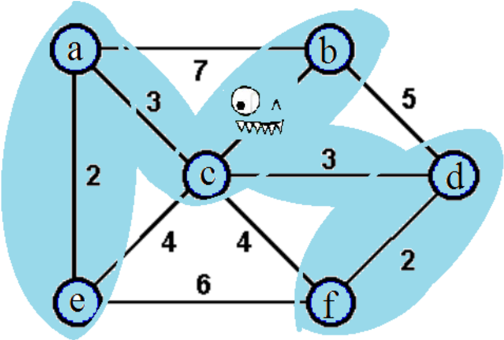


Similarity?

All these algorithm can be seen as greedy monsters:

At each iteration, make a decision that seems best at this instance.

Kruskal's: Always choose the edge with lowest weight and not form a cycle.



Similarity?

All these algorithm can be seen as greedy monsters:

:I will always eat the node that is closest to A

: I will always eat the shortest edge attaching me with an outside node

: I will always eat the shortest edge that does not create a cycle

Greedy Choice Property

- **Dijkstra's**:
  - A subtree X is a partial shortest path tree if the distance from s to any node in X is optimized
  - Suppose X is a partial shortest path tree on S and $v \notin S$ is a current closest node via edge (u, v). Then $X \cup \{(u, v)\}$ is also a partial shortest path tree.

Greedy Choice Property

- **Dijkstra's**:
  - A subtree X is a partial shortest path tree if the distance from s to any node in X is optimized
  - Suppose X is a partial shortest path tree on S and $v \notin S$ is a current closest node via edge (u, v). Then $X \cup \{(u, v)\}$ is also a partial shortest path tree.
- **Prim's**:
  - A subtree X is a partial MST if it can be extended to an MST.
  - Suppose X is a partial MST on S and $\{u, v\}$ is a minimal edge joining some $u \in S$ with $v \notin S$. Then $X \cup \{\{u, v\}\}$ is also a partial MST.

Greedy Choice Property

- **Dijkstra's**:
  - A subtree X is a partial shortest path tree if the distance from s to any node in X is optimized
  - Suppose X is a partial shortest path tree on S and $v \notin S$ is a current closest node via edge (u, v). Then $X \cup \{(u, v)\}$ is also a partial shortest path tree.
- **Prim's**:
  - A subtree X is a partial MST if it can be extended to an MST.
  - Suppose X is a partial MST on S and $\{u, v\}$ is a minimal edge joining some $u \in S$ with $v \notin S$. Then $X \cup \{\{u, v\}\}$ is also a partial MST.
- **Kruskal's**:
  - A subgraph X is a partial MSF if it can be extended to an MST.
  - Suppose X is a partial MSF and $\{u, v\}$ is a minimal edge joining two trees. Then $X \cup \{\{u, v\}\}$ is also a partial MSF.

Greedy Algorithms

Greedy Algorithms

A greedy algorithm solves an optimisation problem by making a locally optimal choice at each time.

Greedy Algorithms

A greedy algorithm solves an optimisation problem by making a locally optimal choice at each time.

Note

Being greedy is risky!
e.g. In shortest path problem that allows negative weights, Dijkstra's algorithm doesn't work.

Greedy Algorithms: Global v.s. Local Optimisation

- Global optimisation value:

  - Shortest Path: The distance
  - MST: The sum of chosen edges

  The optimal solution is said to reach the global optimum.

- Local optimisation value

  - Shortest Path: The estimated distances so far
  - MST: The length of edges

Greedy Algorithms: General Strategy

General Strategy

Starting with an empty solution, repeat the following steps:

1. Examine all ways to expand the current solution
2. Select the way that gives the best local optimisation value

The process stops when there is no way to expand the solution

Greedy Algorithms: General Strategy

General Strategy

Starting with an empty solution, repeat the following steps:

1. Examine all ways to expand the current solution
2. Select the way that gives the best local optimisation value

The process stops when there is no way to expand the solution

Correctness

We need to prove a greedy choice property: Suppose we start with a partial solution that could lead to a global optimum. If we expand the partial solution with the best local optimisation value, the resulting partial solution can also lead to a global optimum.

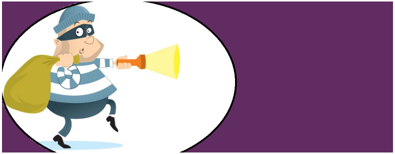
Example 1: Fractional Knapsack Problem

Scenario

A burglar enters a store and finds n items: the ith item is worth v_i dollars and weights $w_i$kg.
Constraint: The thief's knapsack can only carry Wkg in total.
Goal: Maximize the total value of items in the knapsack.

Knapsack Problem

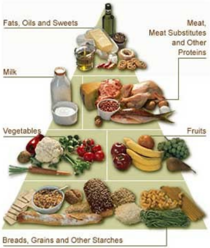
INPUT: Values $v_1, \ldots, v_n$, weights $w_1, \ldots, w_n$, and capacity W
OUTPUT: A selection of S_i amount of item i for $i = 1, \ldots, n$ that maximises total value, but keep total weight within W



Example 1: Fractional Knapsack Problem

Fractional Knapsack Problem

Suppose the burglar enters a food store.
Items are such things as milk, rice, flour, beans, etc.
You may take a fraction of any items.



Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Greedy Choice Property

Suppose S is a partial optimal solution, we have W'kg left, and each item i has w_i'kg left.

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value*/*weight* ratio of items.

Greedy Choice Property

Suppose S is a partial optimal solution, we have W'kg left, and each item i has w_i'kg left.
We make another greedy choice:

- Take the remaining item with highest *value*/*weight* ratio, say j
- Add $\min\{W', w_j'\}$kg item j

Then the resulting solution S' is also a partial optimal solution.

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Greedy Choice Property

Suppose S is a partial optimal solution, we have W'kg left, and each item i has w'_ikg left.
We make another greedy choice:

- Take the remaining item with highest *value/weight* ratio, say j
- Add $\min\{W', w'_j\}$kg item j

Then the resulting solution S' is also a partial optimal solution.

Why?
If a solution contains S but does not contain $\min\{W', w'_j\}$kg item j, then we can increase its value by changing some other items to item j.

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Example

$W = 100$, $w = [20, 25, 50, 40, 30, 60]$, $v = [70, 125, 80, 80, 120, 60]$.
Iteration 0:

$P = \{(1, 5), (4, 4), (0, 3.5), (3, 2), (2, 1.6), (5, 1)\}$
$S = \{\}$
$w' = [20, 25, 50, 40, 30, 60]$
$W' = 100$
$TotalValue = 0$

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Example

$W = 100, w = [20, 25, 50, 40, 30, 60], v = [70, 125, 80, 80, 120, 60].$
Iteration 1:

$P = \{(4, 4), (0, 3.5), (3, 2), (2, 1.6), (5, 1)\}$
$S = \{(1, 25)\}$
$w' = [20, 0, 50, 40, 30, 60]$
$W' = 75$
$TotalValue = 125$

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Example

$W = 100$, $w = [20, 25, 50, 40, 30, 60]$, $v = [70, 125, 80, 80, 120, 60]$.
Iteration 2:

$P = \{(0, 3.5), (3, 2), (2, 1.6), (5, 1)\}$
$S = \{(1, 25), (4, 30)\}$
$w' = [20, 0, 50, 40, 0, 60]$
$W' = 45$
$TotalValue = 125 + 120 = 245$

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Example

$W = 100, w = [20, 25, 50, 40, 30, 60], v = [70, 125, 80, 80, 120, 60]$.
Iteration 3:

$P = \{(3, 2), (2, 1.6), (5, 1)\}$

$S = \{(1, 25), (4, 30), (0, 20)\}$

$w' = [0, 0, 50, 40, 0, 60]$

$W' = 25$

$TotalValue = 245 + 70 = 315$

Example 1: Fractional Knapsack Problem

Local Optimisation Value

At each step, we optimise the *value/weight* ratio of items.

Example

$W = 100$, $w = [20, 25, 50, 40, 30, 60]$, $v = [70, 125, 80, 80, 120, 60]$.
Iteration 4:

$P = \{(2, 1.6), (5, 1)\}$
$S = \{(1, 25), (4, 30), (0, 20), (3, 25)\}$
$w' = [0, 0, 50, 15, 0, 60]$
$W' = 0$
$TotalValue = 315 + 50 = 365$

Example 1: Fractional Knapsack Problem

Algorithm FracKnapsack($v[1..n], w[1..n], W$)

INPUT: $v[1..n]$,$w[1..n]$, and capacity W
OUTPUT: A set S of (i, S_i) pairs indicating amount of item i to take

Create an empty priority queue P (for storing item-ratio pairs)
Create a partial solution set S (for storing item-weight pairs)
`for` $i = 1..n$ `do`
    P.add$(i, v[i]/w[i])$
Create an empty set S
`while` $W > 0$ `do`
    $(i, r) \leftarrow P.RemoveMin()$.
    $S \leftarrow S \cup \{(i, \min\{W, w[i]\})\}$.
    $W \leftarrow W - \min\{W, w[i]\}$.

Example 2: Activity Selection Problem

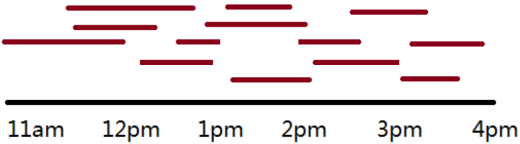
Scenario

Select the activities so that we maximize the number of activities to attend,
under the constraint that we do not attend two activities at the same time and we always attend an entire activity.

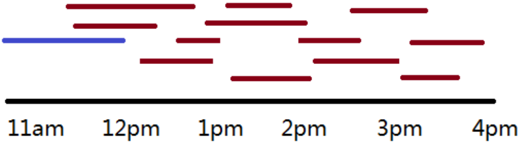
Example 2: Activity Selection Problem

Activity Selection Problem

INPUT: Activities specified by (s_i, f_i) for $i = 1, \ldots, n$, where s_i is the starting time and f_i the finishing time.
OUTPUT: Set S of activities that are not overlapping



Example 2: Activity Selection Problem

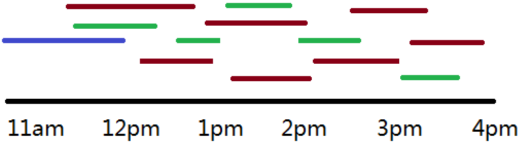


Local Optimal Value

- The finishing time of activities
- At each step, we choose the activity that finishes the earliest

Greedy-choice Property

Example 2: Activity Selection Problem



Local Optimal Value

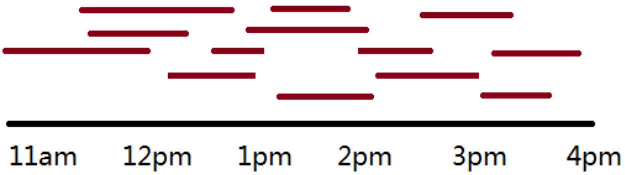
- The finishing time of activities
- At each step, we choose the activity that finishes the earliest

Greedy Choice Property

Suppose there is an optimal selection S that doesn't consist of activity i.
Then replacing the activity that finishes first in S by i, we still have an optimal selection.

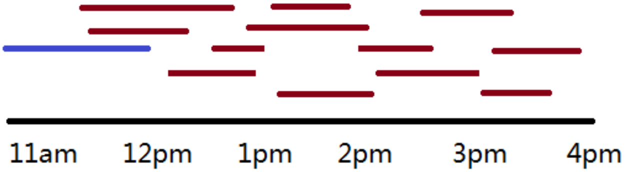
Example 2: Activity Selection Problem

Therefore we can solve the problem by making a greedy choice on the finishing times.



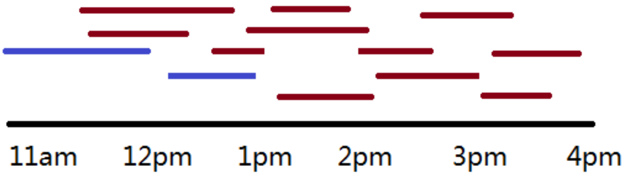
Example 2: Activity Selection Problem

Therefore we can solve the problem by making a greedy choice on the finishing times.



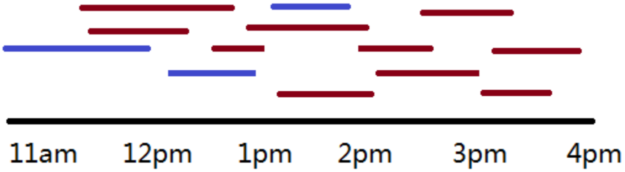
Example 2: Activity Selection Problem

Therefore we can solve the problem by making a greedy choice on the finishing times.



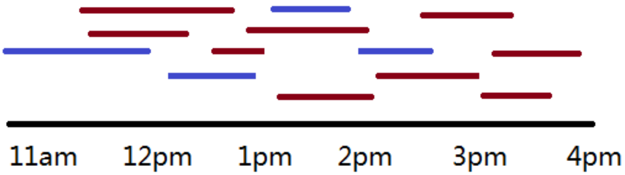
Example 2: Activity Selection Problem

Therefore we can solve the problem by making a greedy choice on the finishing times.



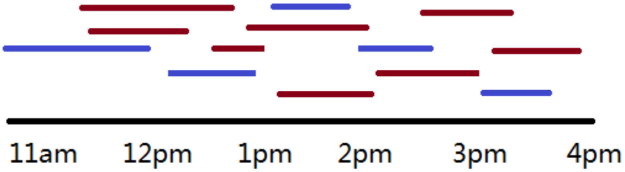
Example 2: Activity Selection Problem

Therefore we can solve the problem by making a greedy choice on the finishing times.



Example 2: Activity Selection Problem

Therefore we can solve the problem by making a greedy choice on the finishing times.



Example 2: Activity Selection Problem

Therefore we can solve the problem by making a greedy choice on the finishing times.

Algorithm ActivitySelect($begin[1..n], end[1..n]$)

INPUT: starting times $s[1..n]$, finishing times $f[1..n]$
OUTPUT: A set S of activities from $\{1, \ldots, n\}$
Maintain a set $I = \{1, \ldots, n\}$
Create an empty priority queue P (to store finishing times)
`for` $i = 1..n$ `do`
    $P.Insert(i, f[i])$
$S \leftarrow \emptyset$
`while` $C \neq \emptyset$ `do`
    $(x, e) \leftarrow P.RemoveMin()$
    $S \leftarrow S \cup \{x\}$
    Delete in C all activities overlapping with interval x.
`return` S