Greedy Algorithms II

Summer 2017 • Lecture 07/27

A Few Notes

Homework 4

- Due tomorrow 7/28 at 11:59 p.m. on Gradescope.

Homework 5

- Released Friday 7/28.

Outline for Today

- Greedy algorithms
  - Greedy graph algorithms
    - Minimum Spanning Trees
    - Prim’s Algorithm
    - Kruskal’s Algorithm

Course Review

We’ve covered a lot so far!

Course Review

- We've covered a lot so far!
  - Techniques for algorithmic analysis
    - Asymptotics, lower-bounding functions, proofs of correctness, runtime
  - 4 algorithmic paradigms: divide and conquer, randomized, greedy, graph.
    - Randomized/graph: `karger`
    - Divide and conquer/randomized: `quicksort`, `quickselect`
    - Greedy/graph: today!

Course Review

- We’ve covered a lot so far!
  - Techniques for algorithmic analysis
    - Asymptotics, lower-bounding functions, proofs of correctness, runtime
  - 4 algorithmic paradigms: divide and conquer, randomized, greedy, graph.
    - Randomized/graph: `karger`
    - Divide and conquer/randomized: `quicksort`, `quickselect`
    - Greedy/graph: today!
  - Several problems: sorting, single-source shortest path, global minimum cut, activity scheduling, hashing, linear-time selection, SCC finding, topological sorting, bipartite finding.

Course Review

- We've covered a lot so far!
  - Techniques for algorithmic analysis
    - Asymptotics, lower-bounding functions, proofs of correctness, runtime
  - 4 algorithmic paradigms: divide and conquer, randomized, greedy, graph.
    - Randomized/graph: `karger`
    - Divide and conquer/randomized: `quicksort`, `quickselect`
    - Greedy/graph: today!
  - Several problems: sorting, single-source shortest path, global minimum cut, activity scheduling, hashing, linear-time selection, SCC finding, topological sorting, bipartite finding.
- A lot of cool stuff ahead!
  - 1 more algorithmic paradigm: dynamic programming.
  - Approximation algorithms, amortized analysis, intractability.

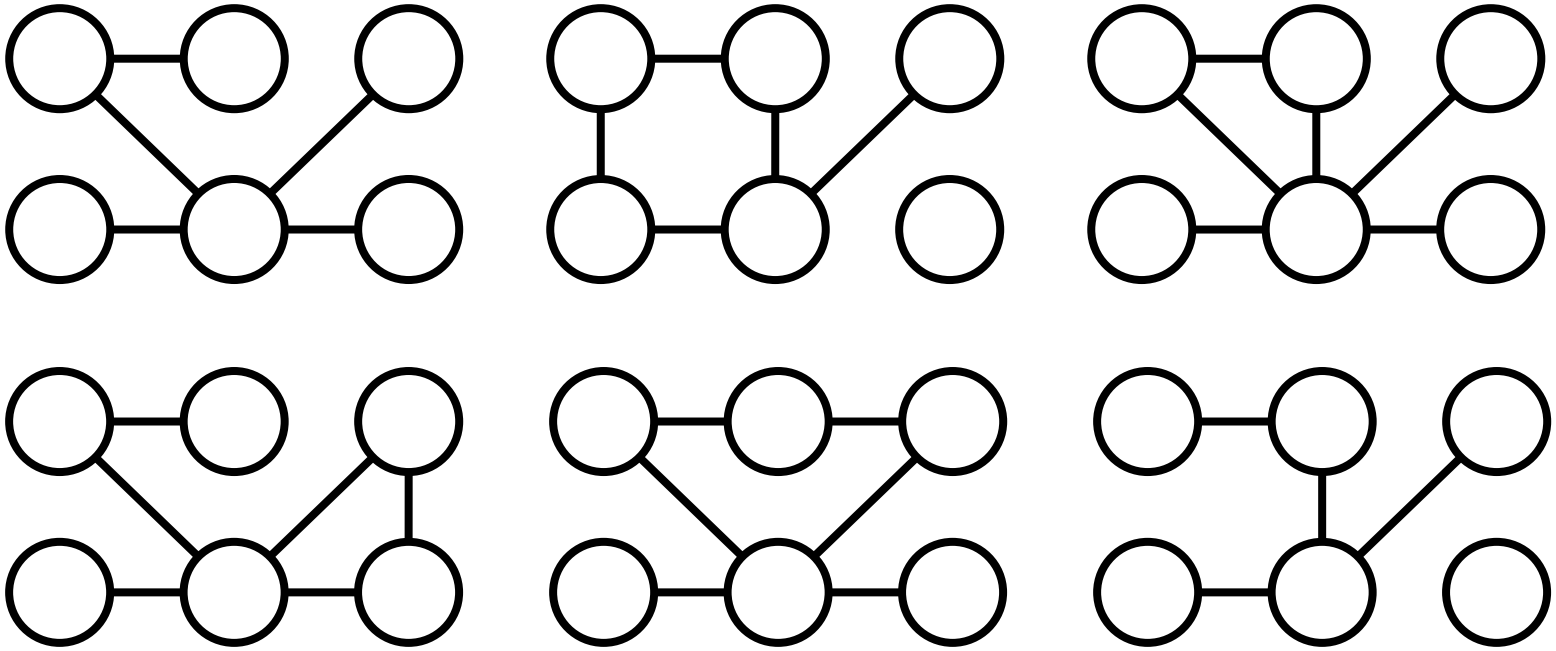
Minimum Spanning Trees

MSTs

A tree is an undirected, acyclic, connected graph.

In Lecture 3, we studied trees with directed edges from parent to children vertices. In this lecture, edges will be undirected.

Which of these graphs contain connected components which are trees?

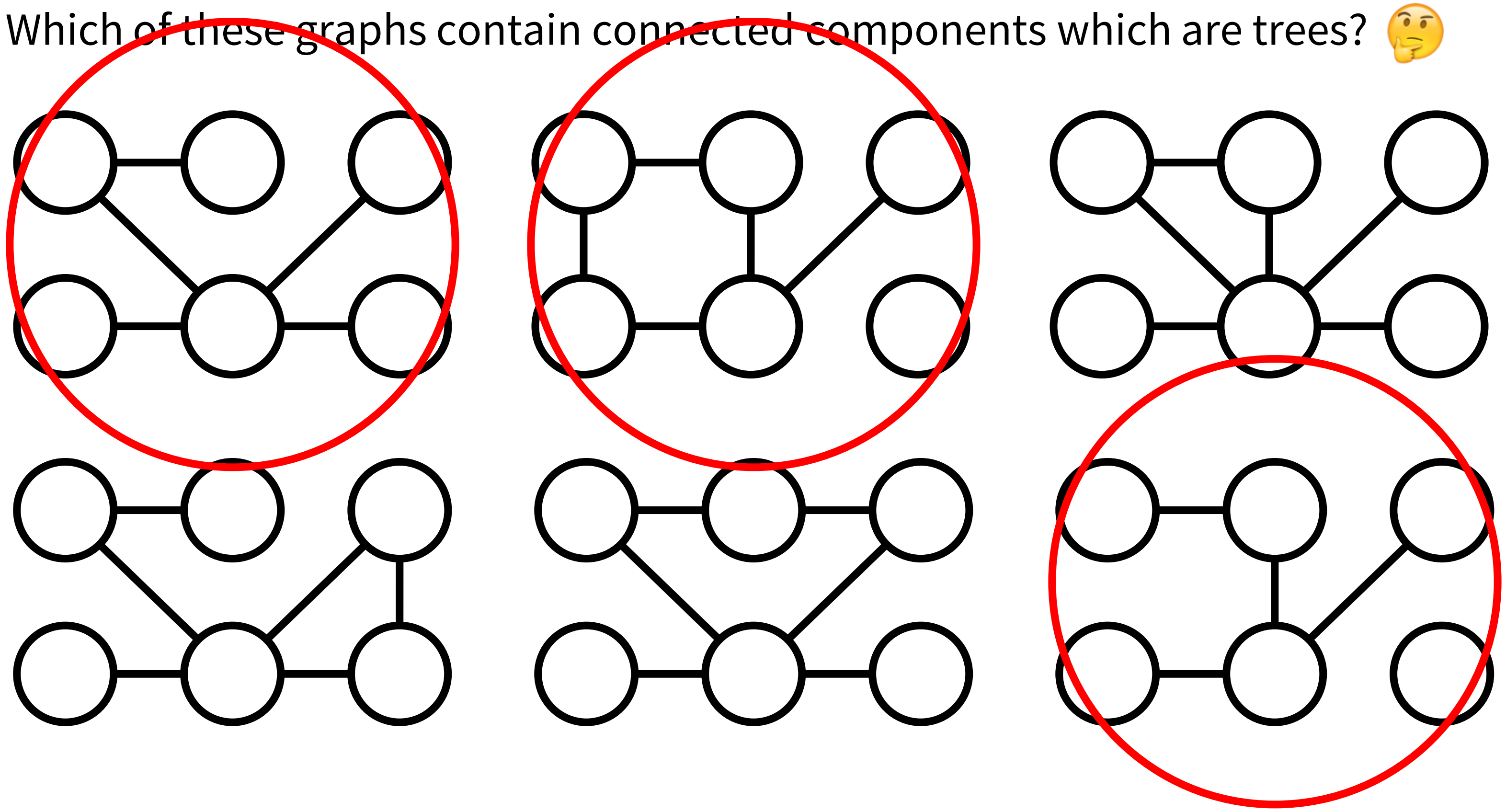


MSTs

In Lecture 3, we studied trees with directed edges from parent to children vertices. In this lecture, edges will be undirected.

A tree is an undirected, acyclic, connected graph.

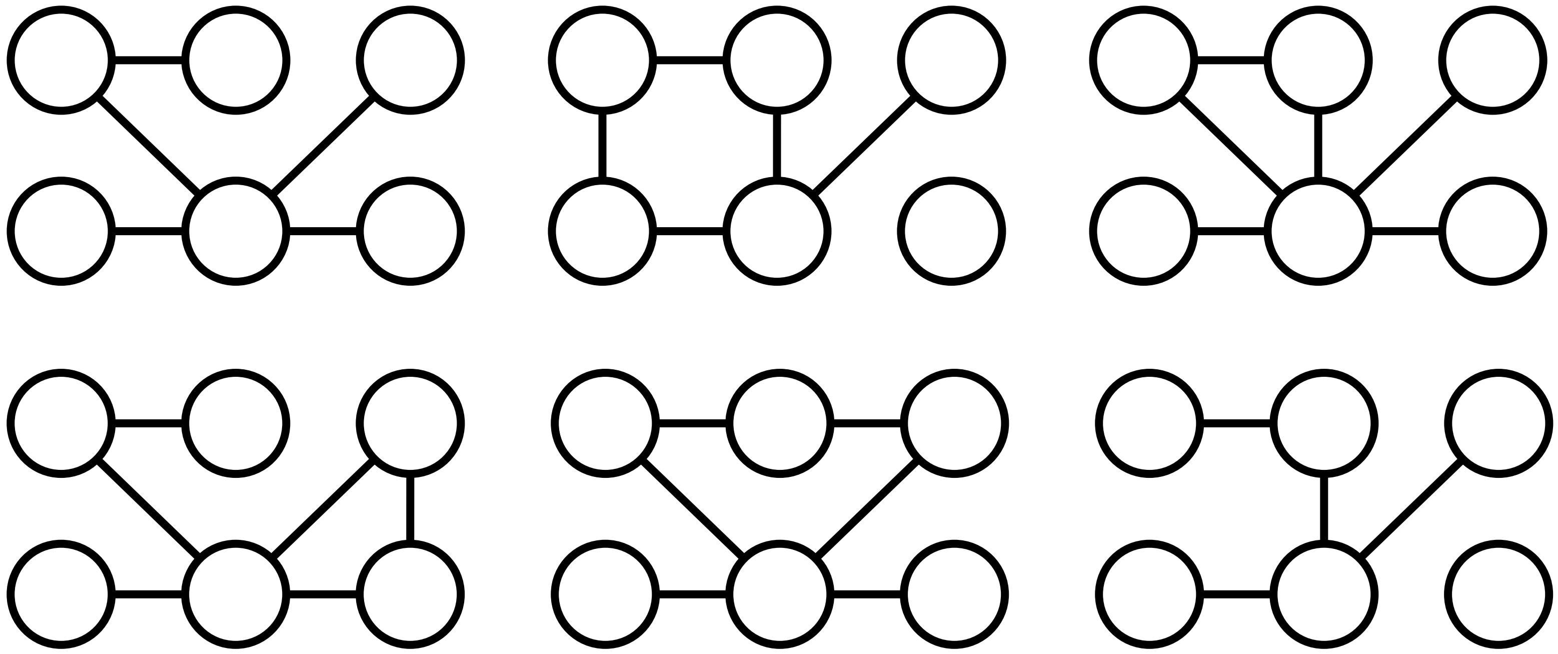
Which of these graphs contain connected components which are trees? 🤔



MSTs

A spanning tree is a tree that connects all of the vertices.

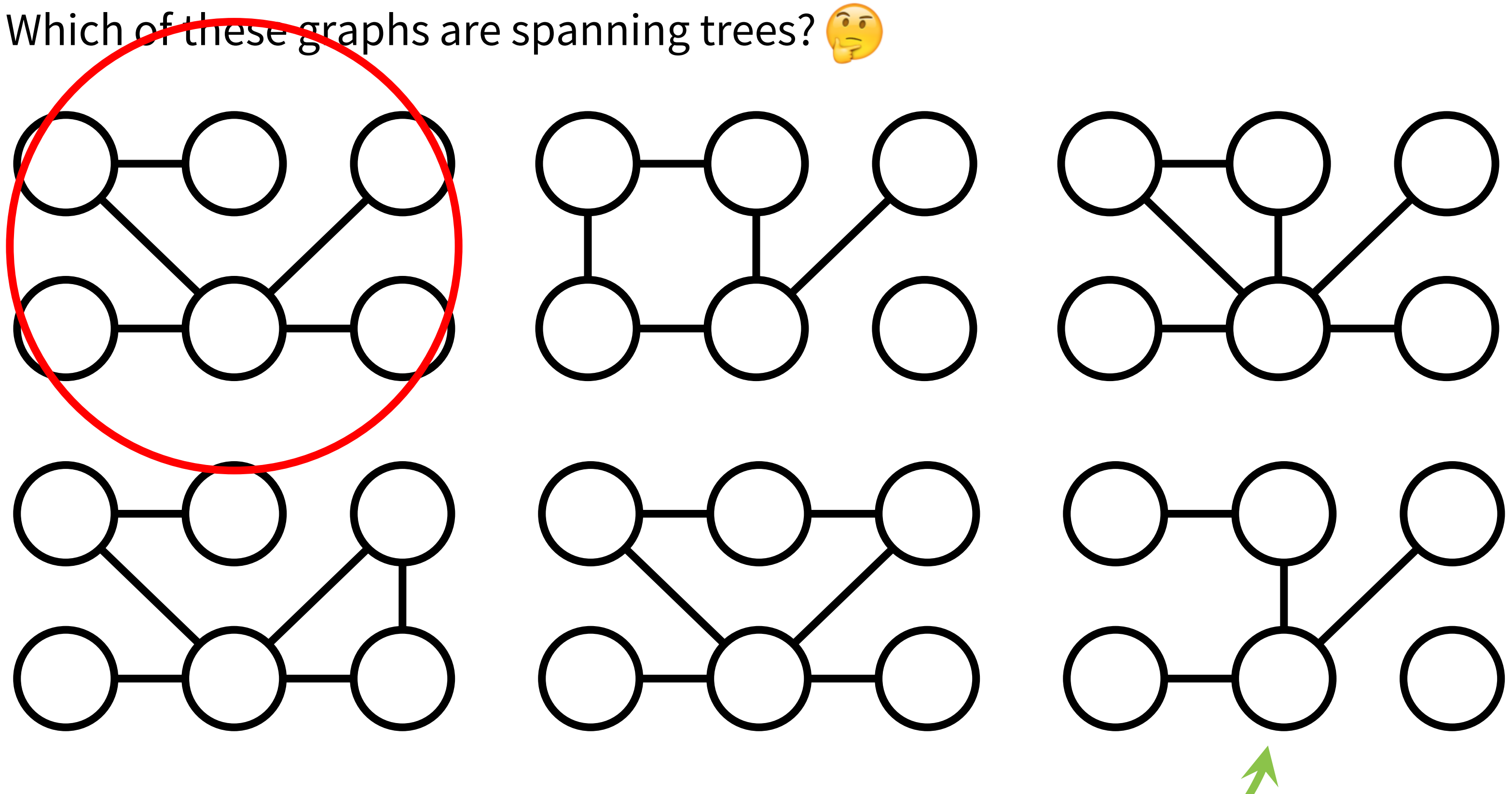
Which of these graphs are spanning trees?



MSTs

A spanning tree is a tree that connects all of the vertices.

Which of these graphs are spanning trees? 🤔

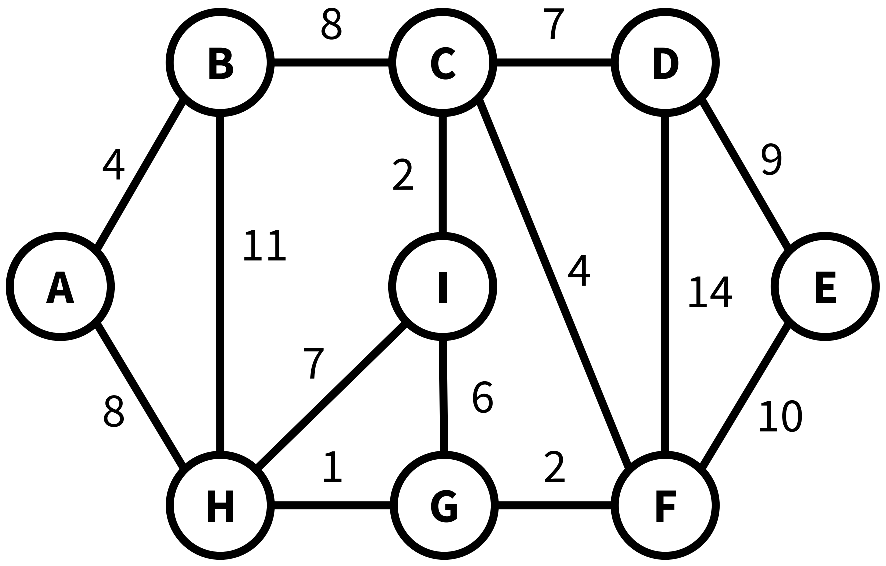


This connected component of the graph is a tree, but it doesn't include all of the vertices.

MSTs

A spanning tree is a tree that connects all of the vertices.

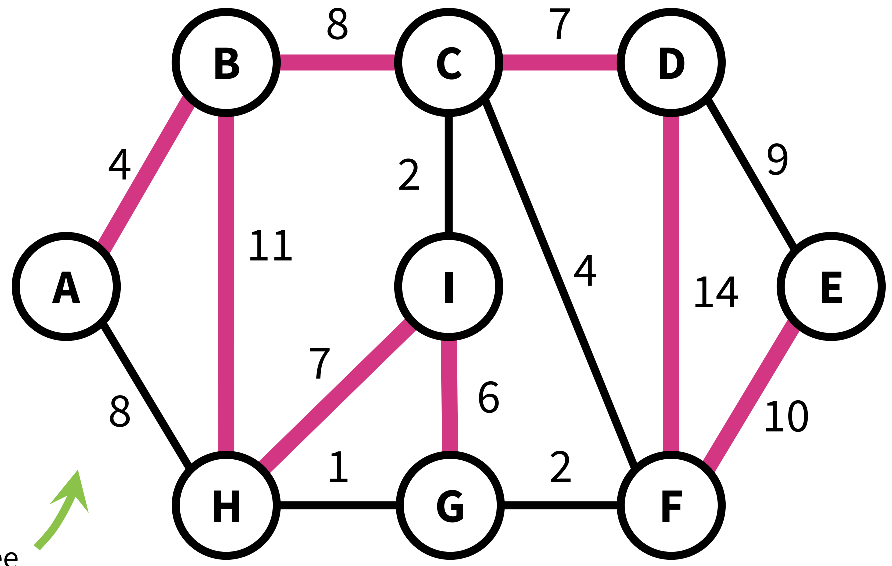
- The cost of a spanning tree is the sum of the weights on the edges.



MSTs

A spanning tree is a tree that connects all of the vertices.

The cost of a spanning tree is the sum of the weights on the edges.

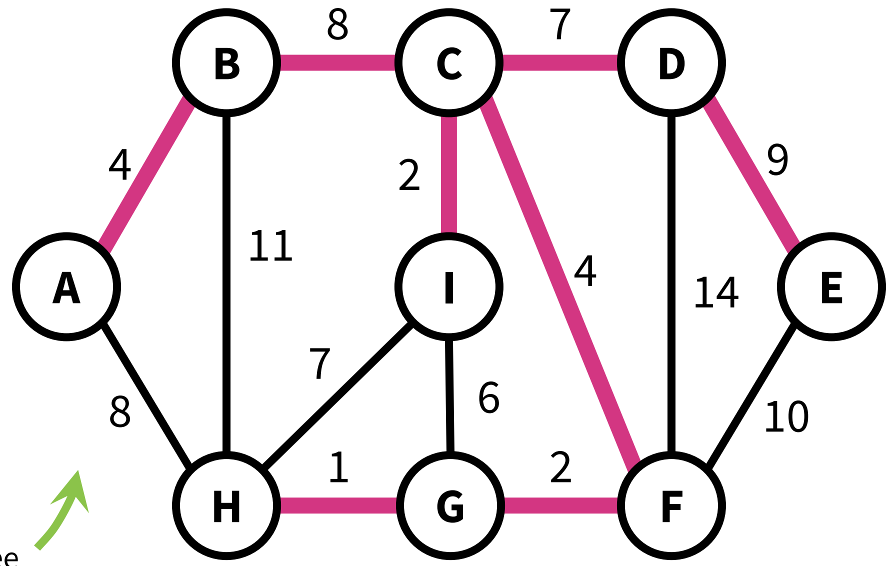


This spanning tree has a cost of 67.

MSTs

A spanning tree is a tree that connects all of the vertices.

The cost of a spanning tree is the sum of the weights on the edges.



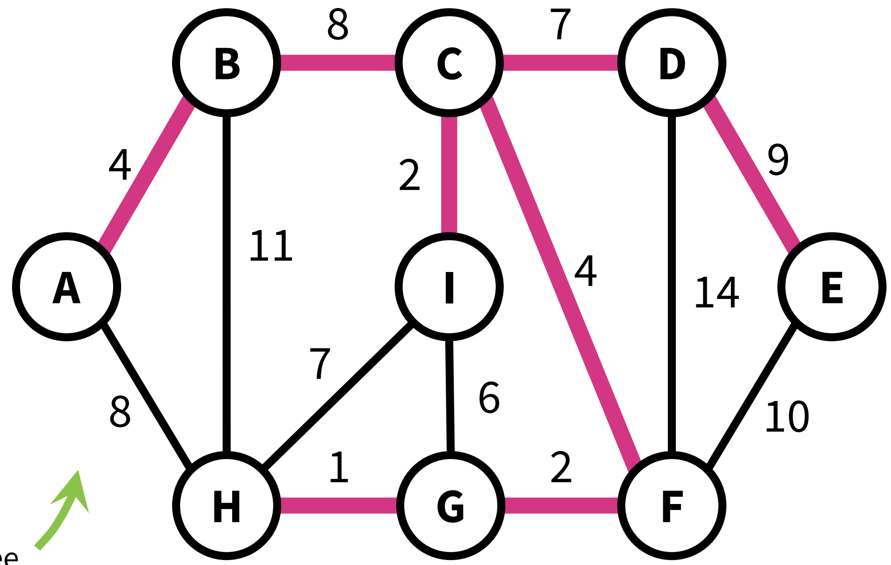
This spanning tree has a cost of 37.

MSTs

mininmum　　　　　　　　　　of minimal cost

A spanning tree is a tree that connects all of the vertices.

The cost of a spanning tree is the sum of the weights on the edges.



This spanning tree has a cost of 37. This is a minimum spanning tree.

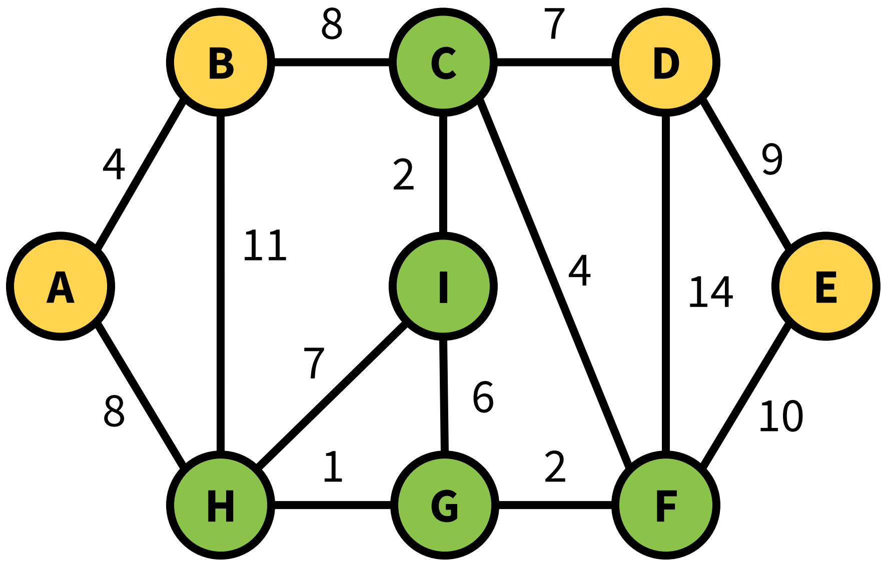
MSTs

- How might we find an MST?
  - Today, we’ll see two greedy algorithms that find an MST.

MSTs

Recall from Lecture 7, a **cut** is a partition of the vertices into two nonempty parts.

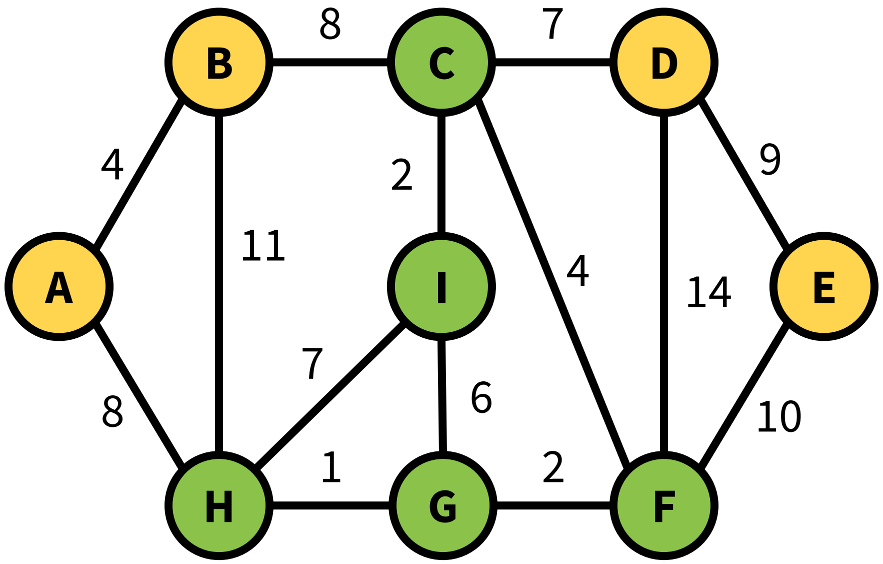
e.g. This is the cut “{A, B, D, E} and {C, I, F, G, H}”.



MSTs

Recall from Lecture 7, a **cut** is a partition of the vertices into two nonempty parts.

e.g. This is the cut “{A, B, D, E} and {C, I, F, G, H}”.



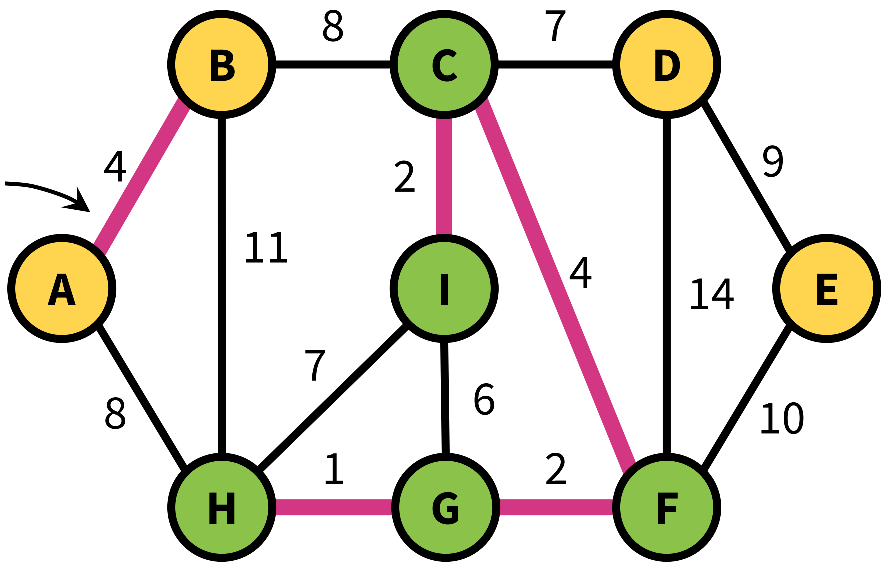
A cut respects a set of edges if no edges in the set cross the cut.

MSTs

Recall from Lecture 7, a **cut** is a partition of the vertices into two nonempty parts.

e.g. This is the cut “{A, B, D, E} and {C, I, F, G, H}”.

This cut respects this set of edges.

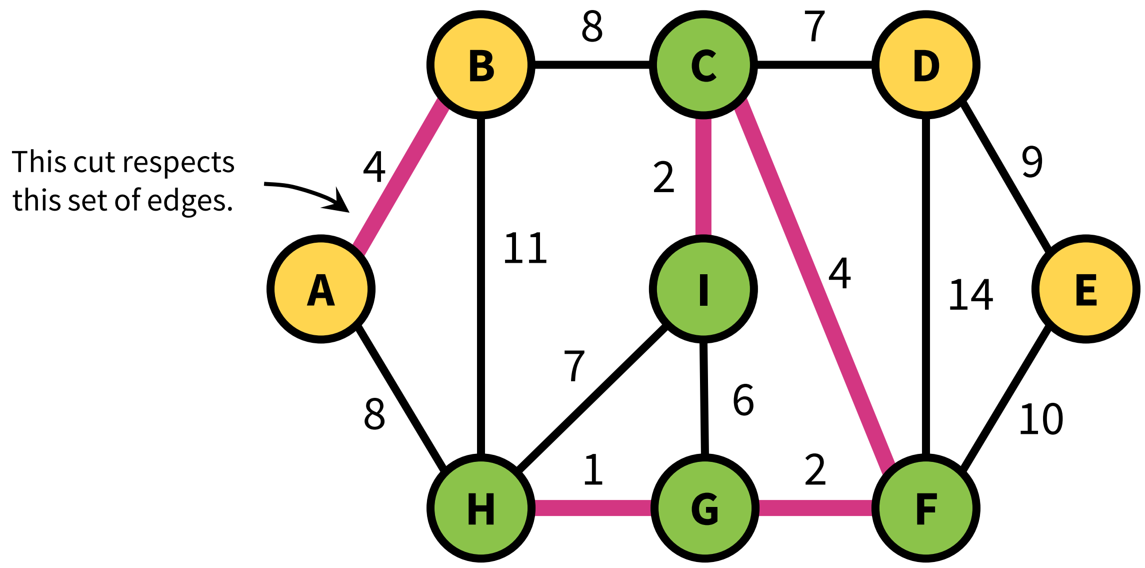


A cut respects a set of edges if no edges in the set cross the cut.

MSTs

Recall from Lecture 7, a **cut** is a partition of the vertices into two nonempty parts.

e.g. This is the cut “{A, B, D, E} and {C, I, F, G, H}”.



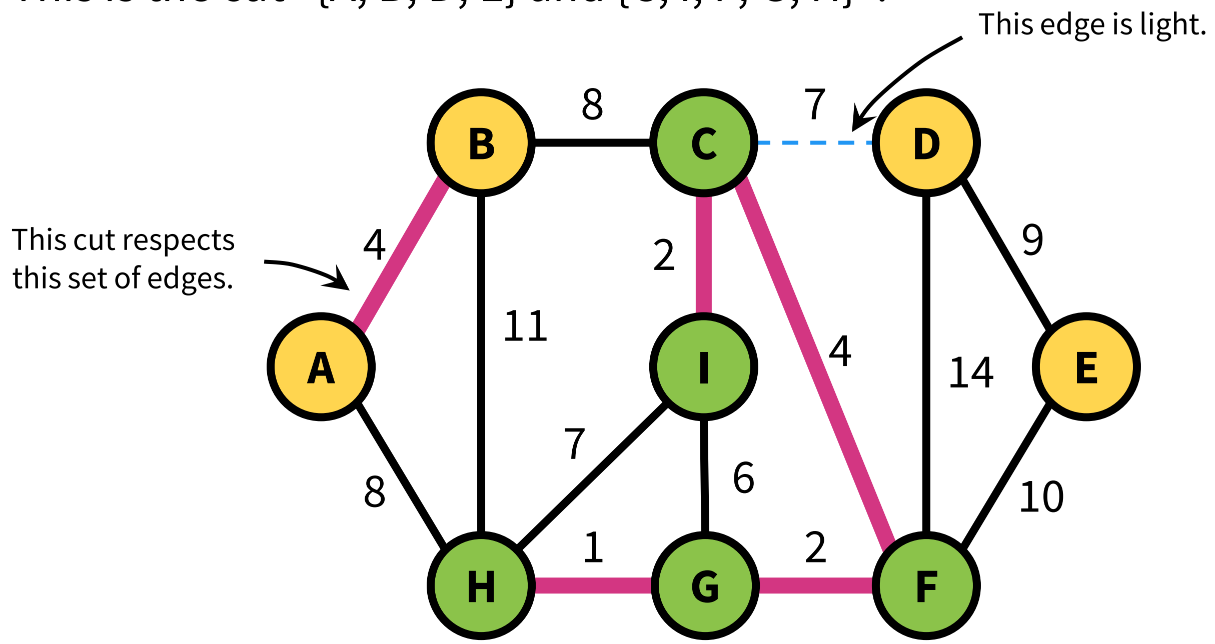
A cut respects a set of edges if no edges in the set cross the cut.

An edge is light if it has the smallest weight of any edge crossing the cut.

MSTs

Recall from Lecture 7, a **cut** is a partition of the vertices into two nonempty parts.

e.g. This is the cut “{A, B, D, E} and {C, I, F, G, H}”.



A cut respects a set of edges if no edges in the set cross the cut.

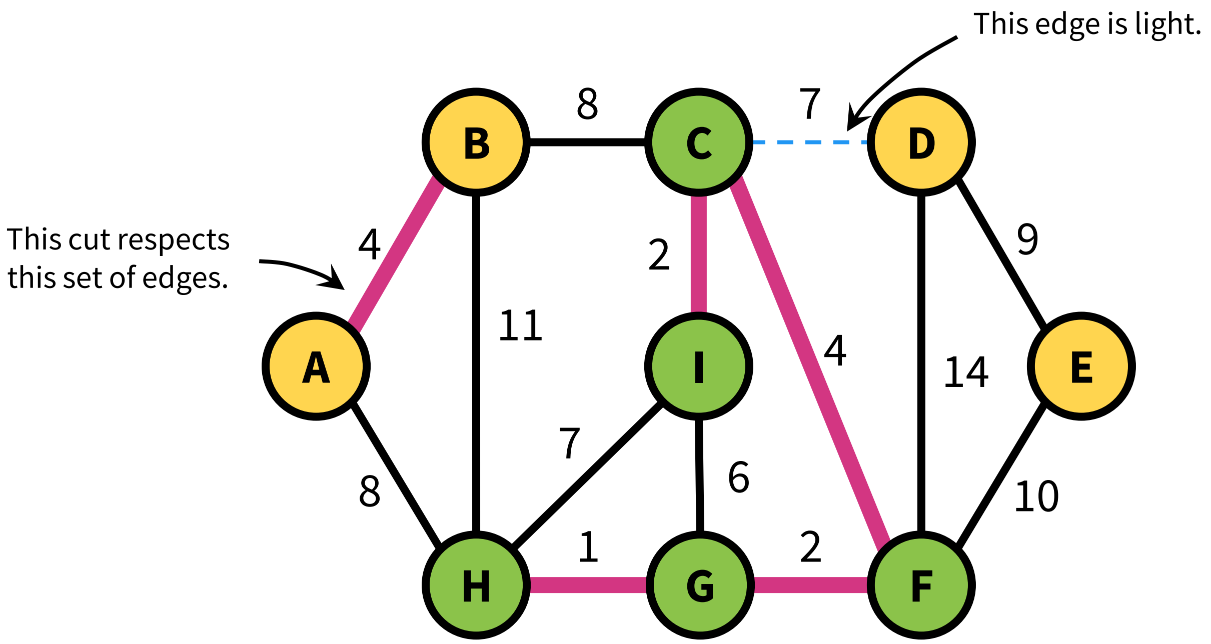
An edge is light if it has the smallest weight of any edge crossing the cut.

Lemma

Consider a cut that respects a set of edges **A**.

Suppose there exists an MST containing **A**.

Let (u, v) be a light edge.



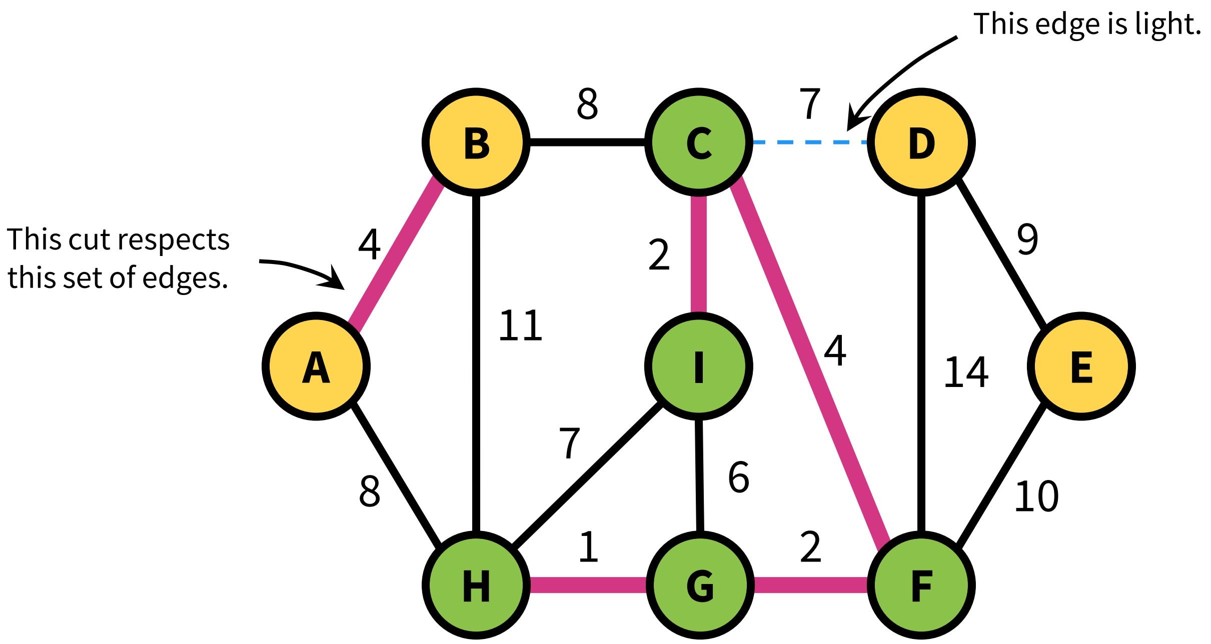
Lemma

Consider a cut that respects a set of edges **A**.

Suppose there exists an MST containing **A**.

Let (u, v) be a light edge.

Then there exists an MST containing **A** ∪ {(u, v)}.



Lemma

Consider a cut that respects a set of edges **A**.

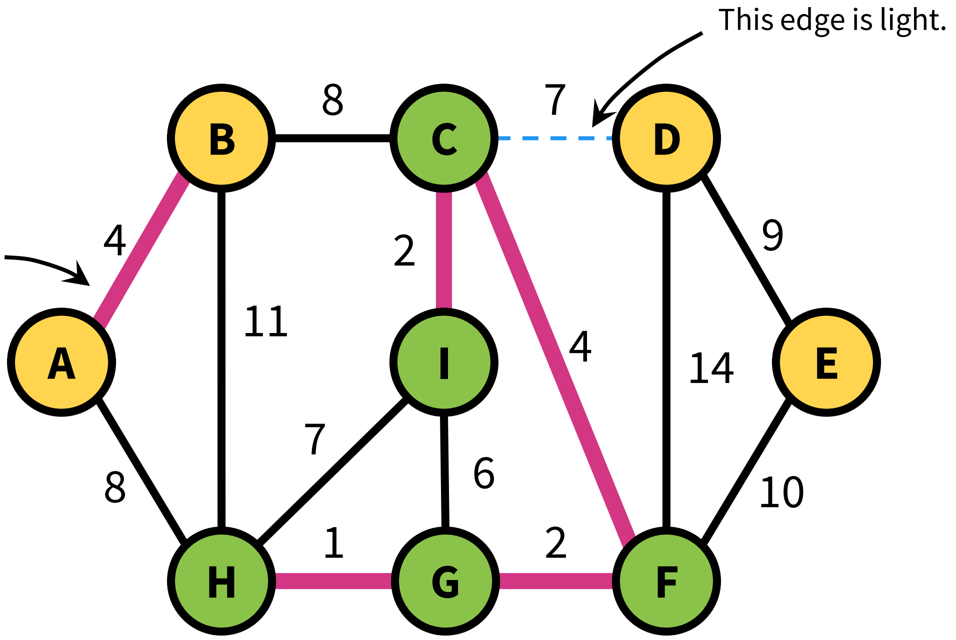
Suppose there exists an MST containing **A**.

Let (u, v) be a light edge.

Then there exists an MST containing **A** ∪ {(u, v)}.

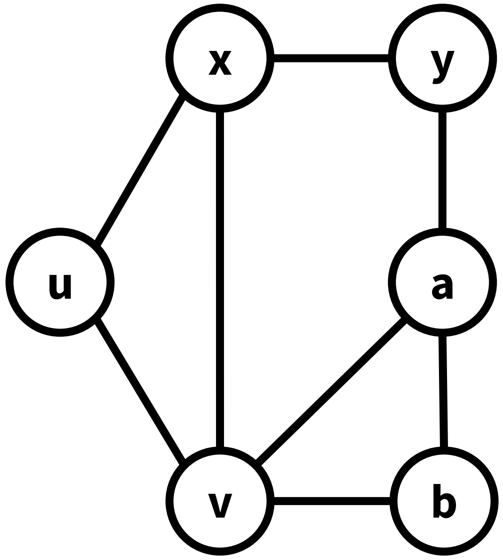
This is precisely the sort of statement we need for a greedy algorithm: If we haven't ruled out the possibility of success so far, then adding a light edge won't rule it out.

This cut respects this set of edges.



Proof of Lemma

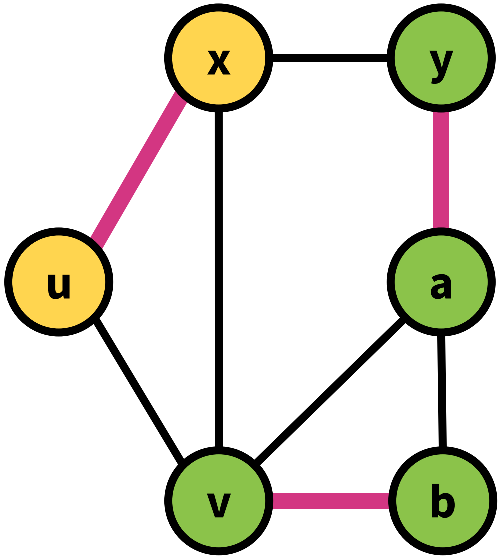
Consider a graph with . . .



Proof of Lemma

Consider a graph with …

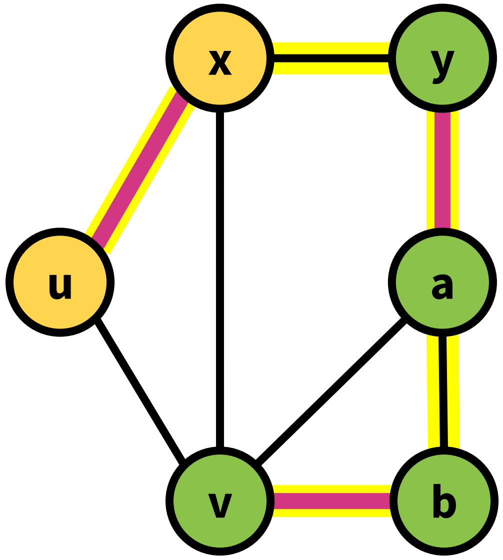
- A cut that respects a set of edges **A**



Proof of Lemma

Consider a graph with …

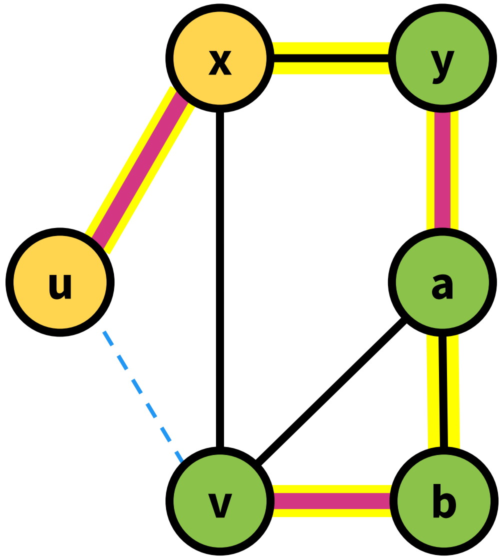
- A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**,



Proof of Lemma

Consider a graph with …

A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

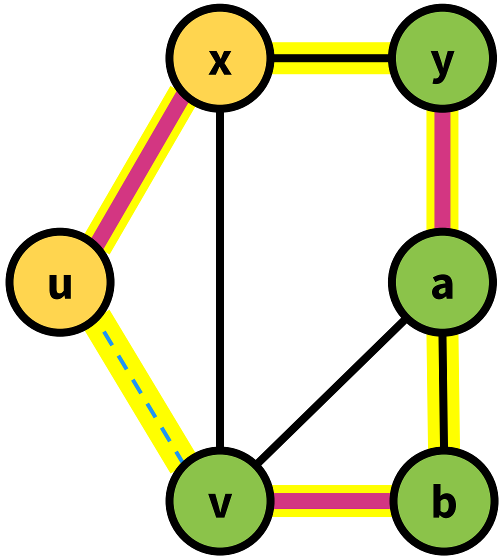


Proof of Lemma

Consider a graph with …

- A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Adding (u, v) to **T** will make a cycle.



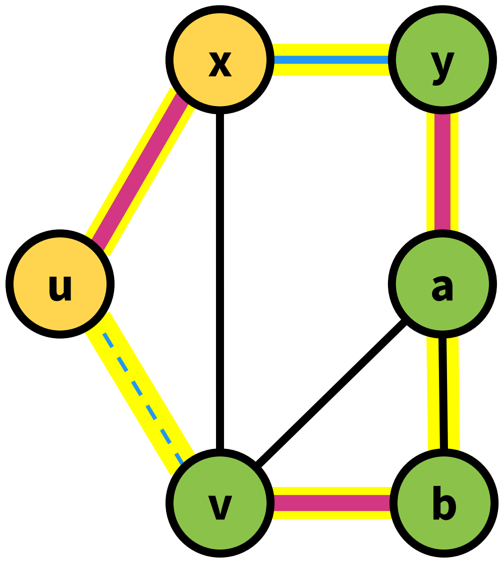
Proof of Lemma

Consider a graph with …

- A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Adding (u, v) to **T** will make a cycle.

- There must be another edge in this cycle crossing this cut. Let's call this edge (x, y).



Proof of Lemma

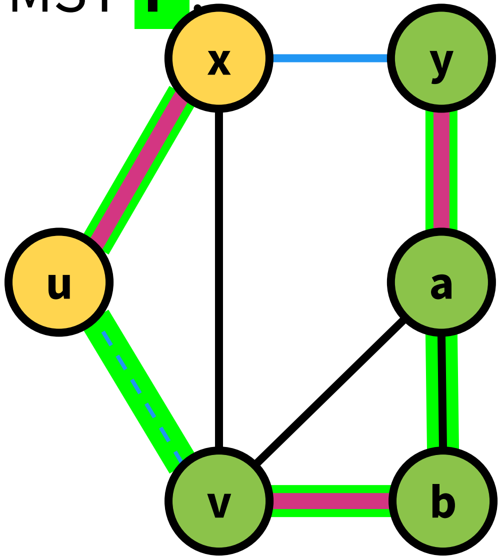
Consider a graph with …

- A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Adding (u, v) to **T** will make a cycle.

- There must be another edge in this cycle crossing this cut.
- Let's call this edge (x, y).

Exchange (u, v) for (x, y) in **T**; call the resulting MST **T'**.



Proof of Lemma

Consider a graph with …

A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Adding (u, v) to **T** will make a cycle.

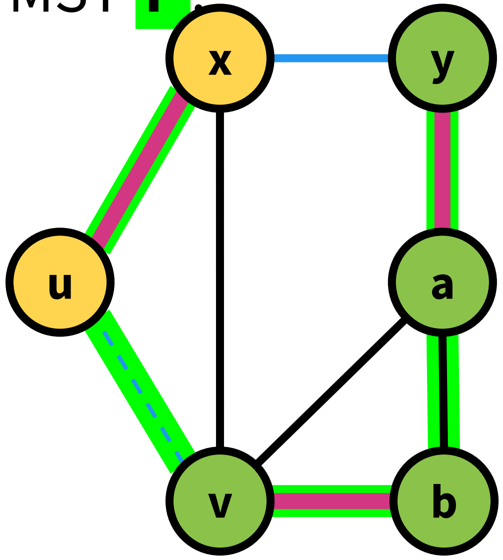
There must be another edge in this cycle crossing this cut.
Let's call this edge (x, y).

Exchange (u, v) for (x, y) in **T**; call the resulting MST **T'**.

Claim: **T'** is still an MST.

Since we deleted (x, y), **T'** is still a tree.

Since (u, v) is light, **T'** has cost at most that of **T**.



Proof of Lemma

Consider a graph with …

A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Adding (u, v) to **T** will make a cycle.

There must be another edge in this cycle crossing this cut. Let's call this edge (x, y).

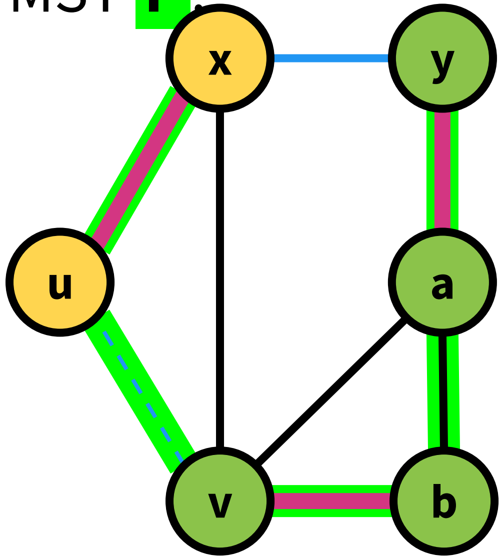
Exchange (u, v) for (x, y) in **T**; call the resulting MST **T'**.

Claim: **T'** is still an MST.

Since we deleted (x, y), **T'** is still a tree.

Since (u, v) is light, **T'** has cost at most that of **T**.

Thus, there exists an MST containing **A** ∪ {(u, v)}.



Prim's Algorithm

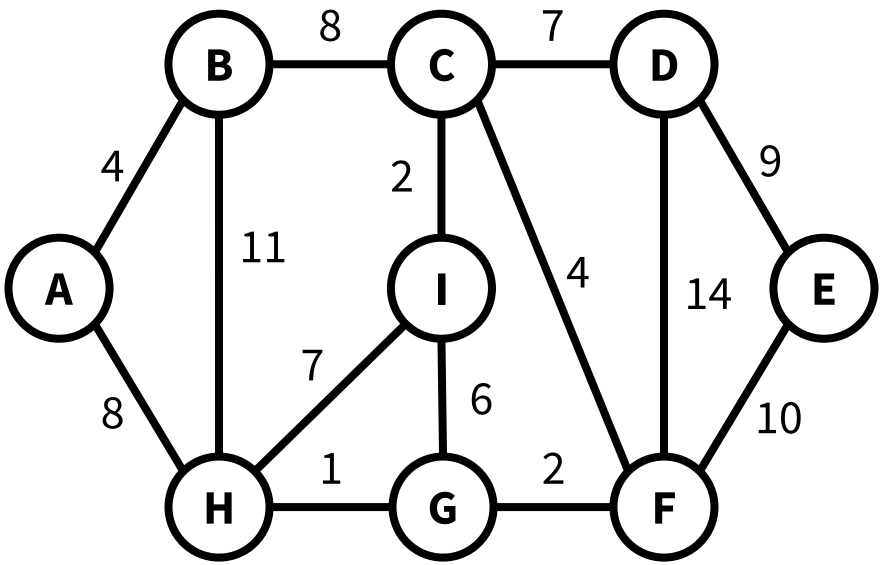
Any Ideas?

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

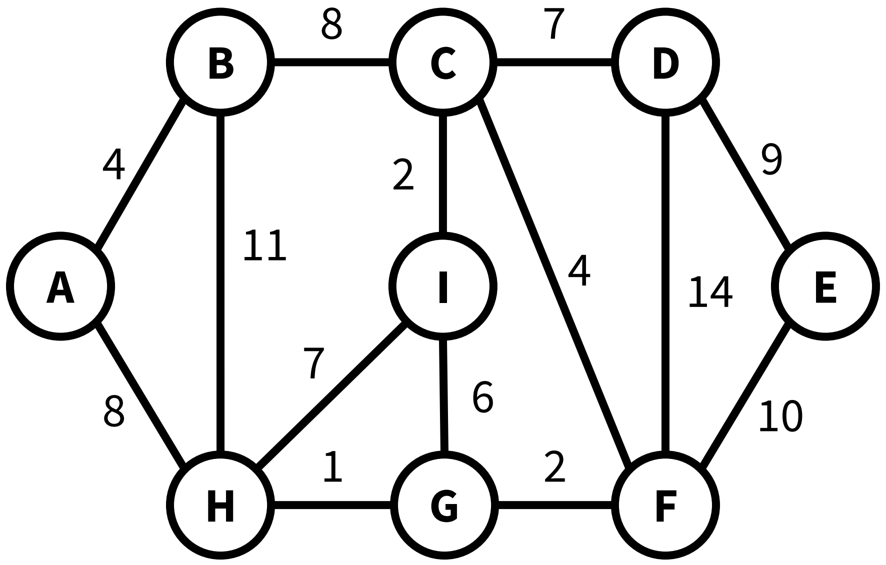
Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Any ideas about what to greedily choose?



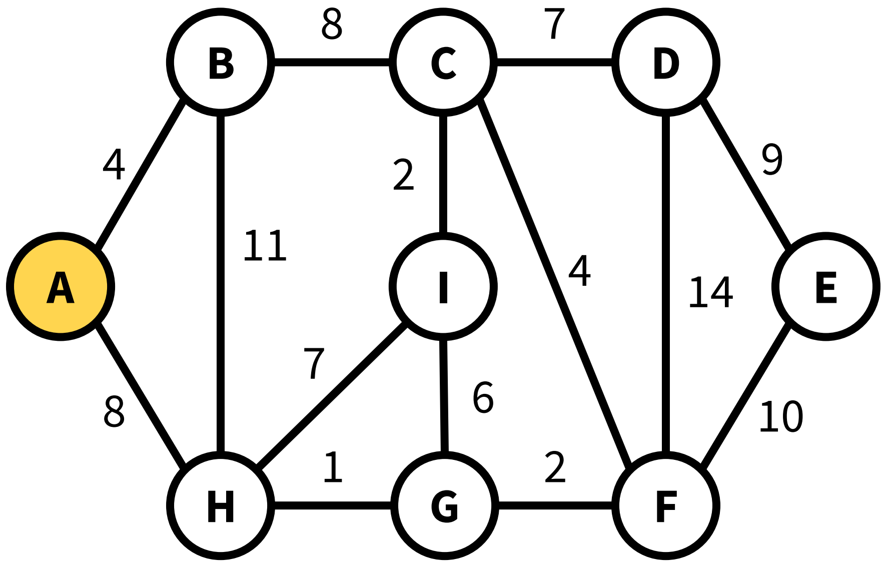
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



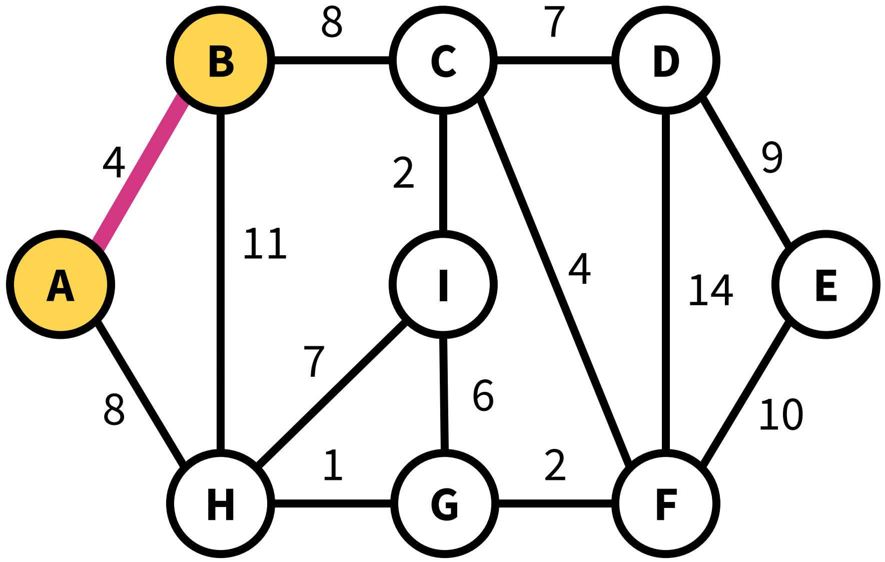
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



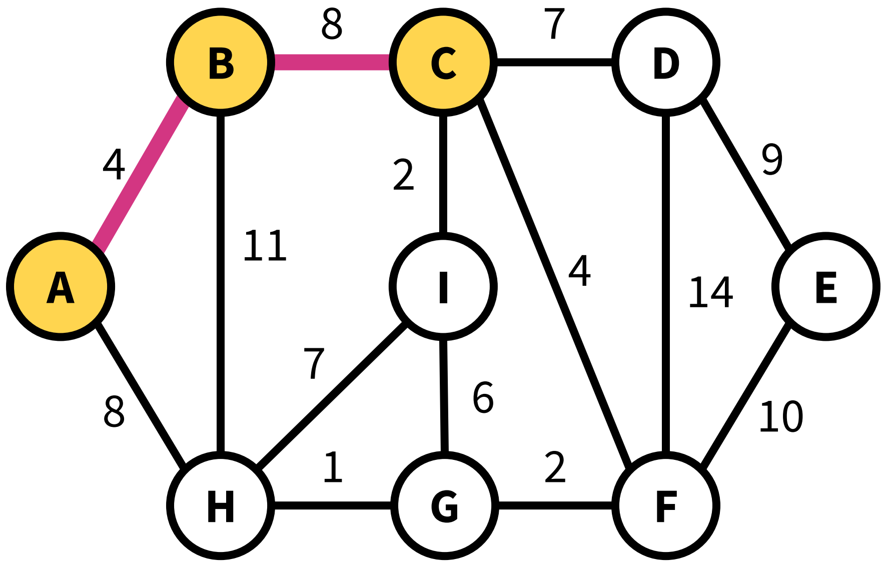
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



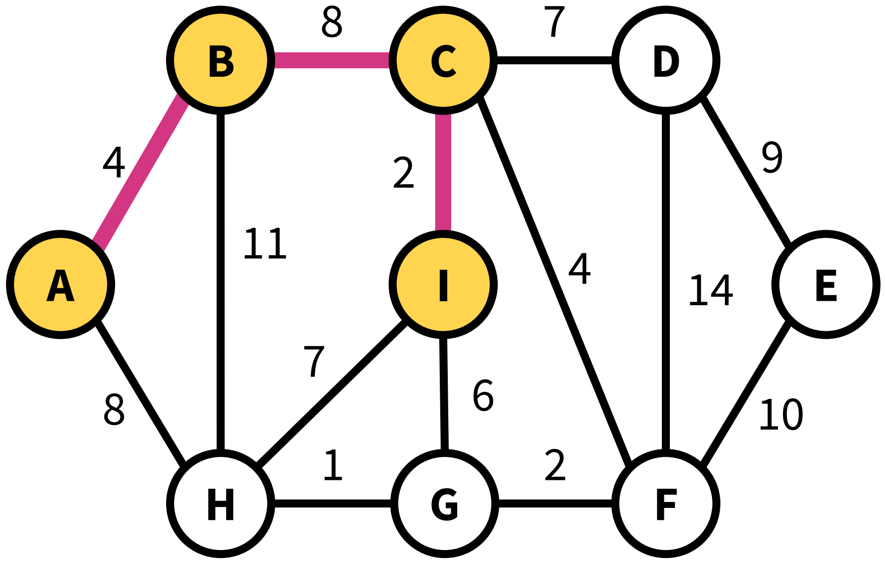
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



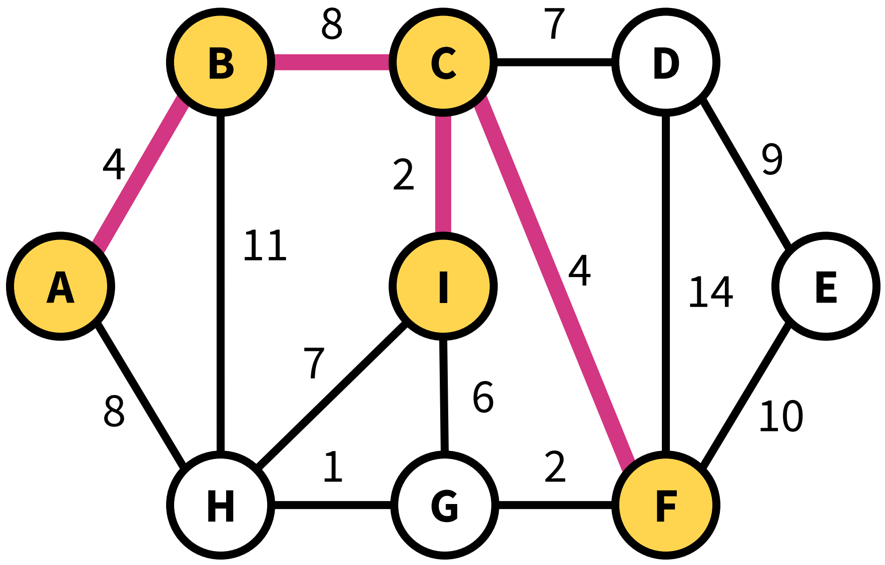
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



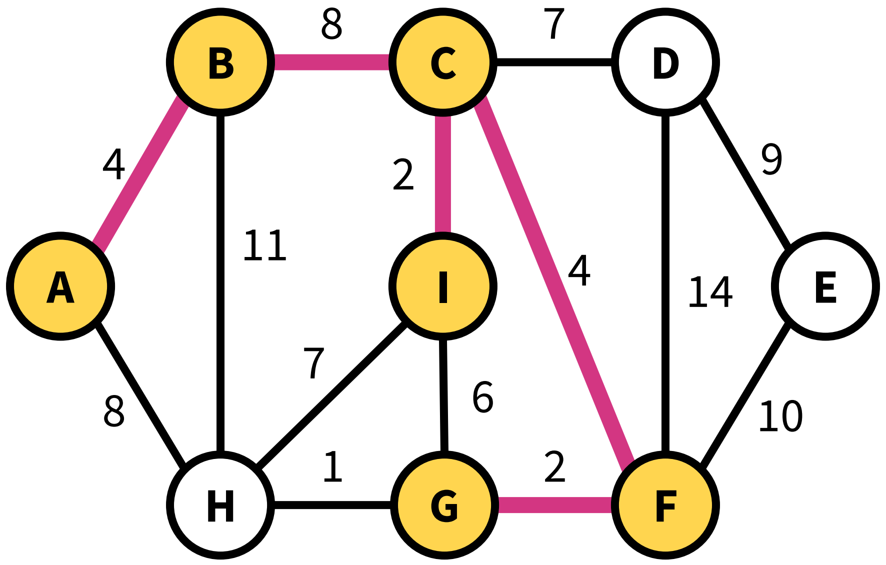
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



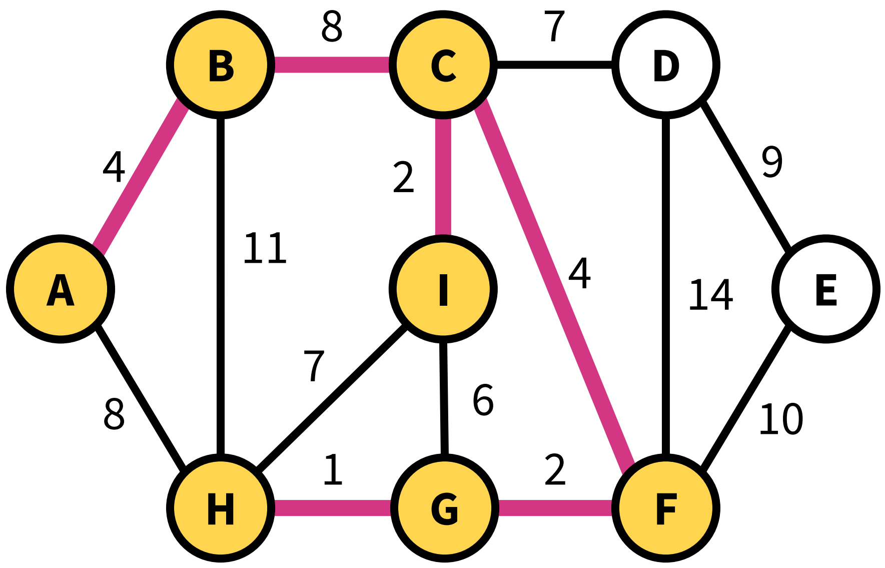
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



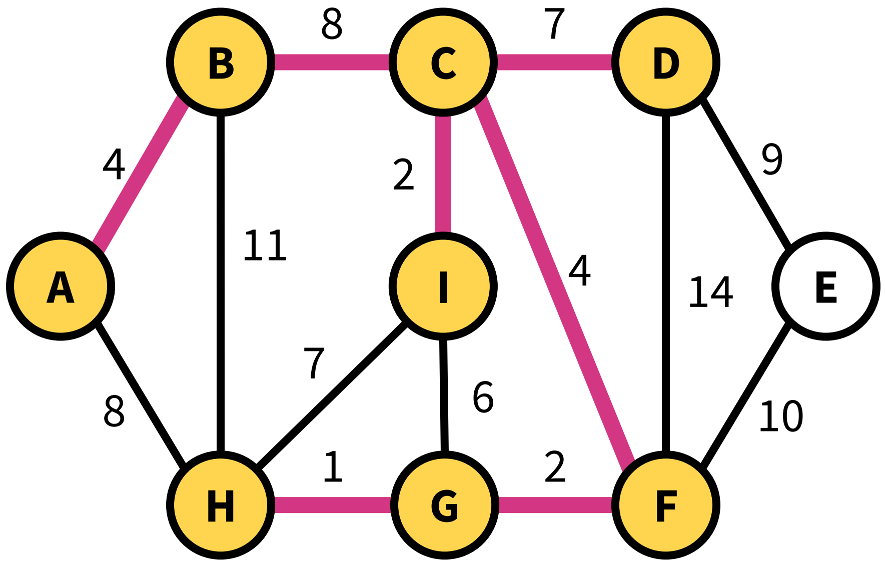
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



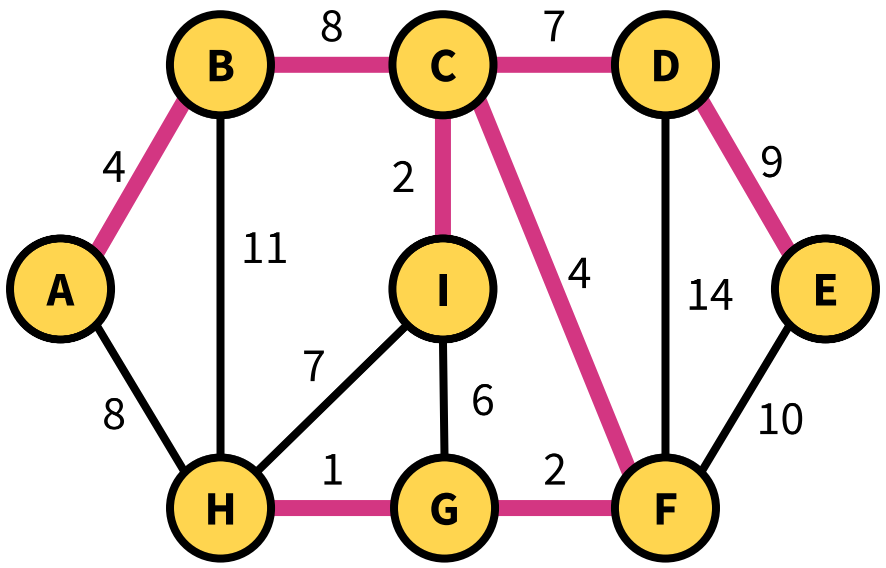
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex.



Prim's Algorithm

```
algorithm slow_prim(G):
  s = random vertex in G
  MST = {}
  visited_vertices = {s}
  while |visited_vertices| < |V|:
    (x, v) = lightest_edge(G, visited_vertices)
    MST.add((x, v))
    visited_vertices.add(v)
  return MST
```

Runtime: $O(|V| \cdot |E|)$

Prim's Algorithm

```
algorithm slow_prim(G):
  s = random vertex in G
  MST = {}
  visited_vertices = {s}
  while |visited_vertices| < |V|:
    (x, v) = lightest_edge(G, visited_vertices)
    MST.add((x, v))
    visited_vertices.add(v)
  return MST
```

aka while we haven't visited all of the vertices

Runtime: $O(|V| \cdot |E|)$

Prim's Algorithm

```
algorithm slow_prim(G):
  s = random vertex in G
  MST = {}
  visited_vertices = {s}
  while |visited_vertices| < |V|:
    (x, v) = lightest_edge(G, visited_vertices)
    MST.add((x, v))
    visited_vertices.add(v)
  return MST
```

aka while we haven't visited all of the vertices

Finds the lightest edge (x, v) in E such that x is in visited_vertices and v is not.

Runtime: O(|V|•|E|)

Prim's Algorithm

```
algorithm slow_prim(G):
  s = random vertex in G
  MST = {}
  visited_vertices = {s}
  while |visited_vertices| < |V|:
    (x, v) = lightest_edge(G, visited_vertices)
    MST.add((x, v))
    visited_vertices.add(v)
  return MST
```

aka while we haven't visited all of the vertices

Finds the lightest edge (x, v) in E such that x is in visited_vertices and v is not.

Runtime: $O(|V| \cdot |E|)$

For each of the |V| iterations of the while loop, might need to iterate through all edges.

Proving Feasibility

Theorem: `prim` finds a feasible spanning tree.

Proof:

To prove this statement, we prove the loop invariant: `MST` contains edges of a spanning tree of the vertices in `visited_vertices`.

Proving Feasibility

Theorem: `prim` finds a feasible spanning tree.

Proof:

To prove this statement, we prove the loop invariant: `MST` contains edges of a spanning tree of the vertices in `visited_vertices`.

At the start of the first iteration, `MST` contains no edges, which corresponds to a spanning tree of one vertex.

Proving Feasibility

Theorem: `prim` finds a feasible spanning tree.

Proof:

To prove this statement, we prove the loop invariant: `MST` contains edges of a spanning tree of the vertices in `visited_vertices`.

At the start of the first iteration, `MST` contains no edges, which corresponds to a spanning tree of one vertex.

Now, we prove the inductive step. Suppose that the invariant holds at the start of iteration i, so the edges in `MST` are (1) acyclic and (2) connect all vertices in `visited_vertices`. Then prim adds an edge (x, v) to `MST` and vertex v to `visited_vertices`.

Proving Feasibility

Theorem: `prim` finds a feasible spanning tree.

Proof:

To prove this statement, we prove the loop invariant: `MST` contains edges of a spanning tree of the vertices in `visited_vertices`.

At the start of the first iteration, `MST` contains no edges, which corresponds to a spanning tree of one vertex.

Now, we prove the inductive step. Suppose that the invariant holds at the start of iteration i, so the edges in `MST` are (1) acyclic and (2) connect all vertices in `visited_vertices`. Then prim adds an edge (x, v) to `MST` and vertex v to `visited_vertices`. By construction, v has not been visited yet, so the edges in `MST` must still be acyclic.

Proving Feasibility

Theorem: `prim` finds a feasible spanning tree.

Proof:

To prove this statement, we prove the loop invariant: `MST` contains edges of a spanning tree of the vertices in `visited_vertices`.

At the start of the first iteration, `MST` contains no edges, which corresponds to a spanning tree of one vertex.

Now, we prove the inductive step. Suppose that the invariant holds at the start of iteration i, so the edges in `MST` are (1) acyclic and (2) connect all vertices in `visited_vertices`. Then prim adds an edge (x, v) to `MST` and vertex v to `visited_vertices`. By construction, v has not been visited yet, so the edges in `MST` must still be acyclic. Furthermore, v connects to x, which connects to the rest of the vertices in `visited_vertices`; therefore, the edges in `MST` must still connect all vertices in `visited_vertices`, completing the induction.

Proving Feasibility

Theorem: `prim` finds a feasible spanning tree.

Proof:

To prove this statement, we prove the loop invariant: `MST` contains edges of a spanning tree of the vertices in `visited_vertices`.

At the start of the first iteration, `MST` contains no edges, which corresponds to a spanning tree of one vertex.

Now, we prove the inductive step. Suppose that the invariant holds at the start of iteration i, so the edges in `MST` are (1) acyclic and (2) connect all vertices in `visited_vertices`. Then prim adds an edge (x, v) to `MST` and vertex v to `visited_vertices`. By construction, v has not been visited yet, so the edges in `MST` must still be acyclic. Furthermore, v connects to x, which connects to the rest of the vertices in `visited_vertices`; therefore, the edges in `MST` must still connect all vertices in `visited_vertices`, completing the induction.

At the termination of the loop, `visited_vertices` contains all of the vertices, so `MST` contains a spanning tree over the entire graph. □

Proving Optimality

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Proving Optimality

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `slow_prim` returns a minimum spanning tree.

Proof:

Proving Optimality

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `slow_prim` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

Proving Optimality

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `slow_prim` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

Consider the cut of visited vertices and unvisited vertices; `MST` respects This cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(x, v)}.

Proving Optimality

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `slow_prim` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

Consider the cut of visited vertices and unvisited vertices; `MST` respects This cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(x, v)}.

Recall, we proved our lemma with an exchange argument!

Proving Optimality

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `slow_prim` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

Consider the cut of visited vertices and unvisited vertices; `MST` respects This cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(x, v)}.

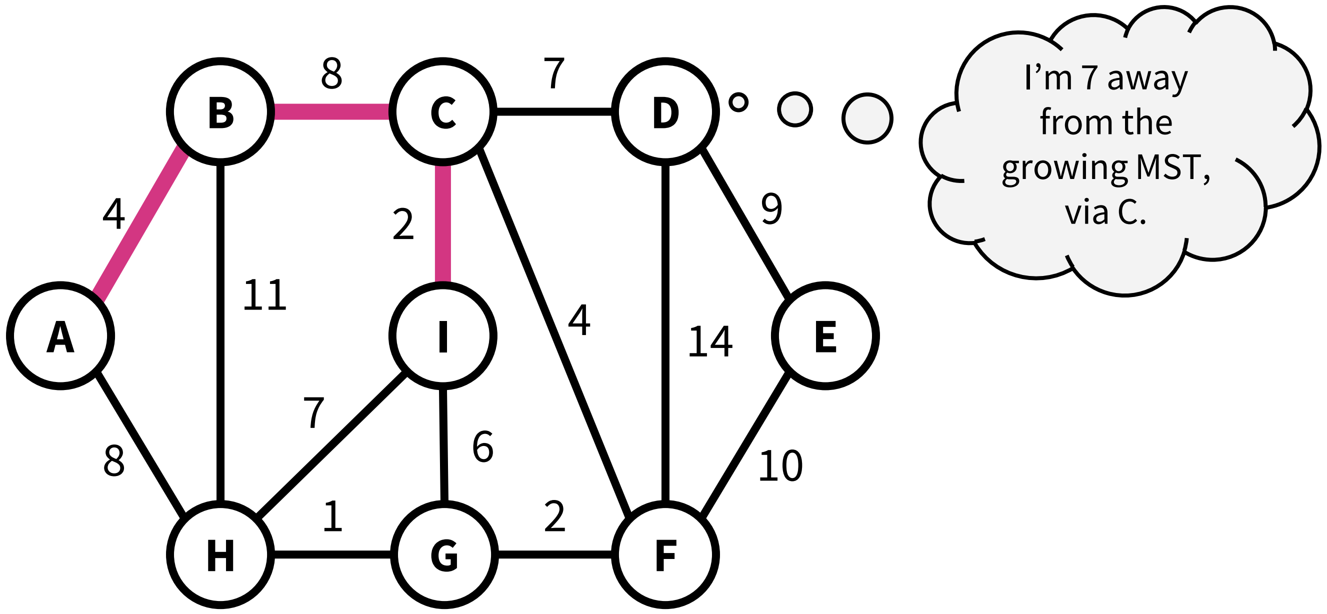
Recall, we proved our lemma with an exchange argument!

After adding the the (n-1)st edge, we have a spanning tree; therefore, `MST` contains a minimum spanning tree. ∎

Prim's Algorithm

We called the algorithm `slow_prim`. There's a more efficient implementation.

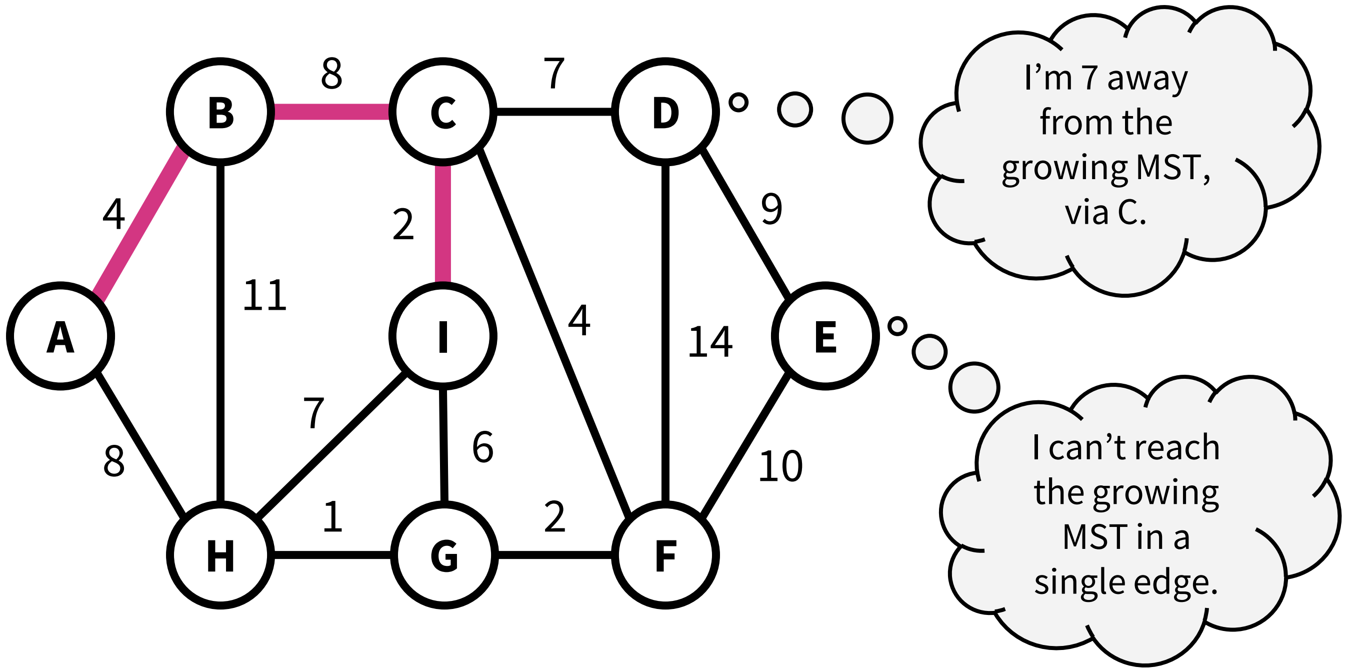
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.



Prim's Algorithm

We called the algorithm `slow_prim`. There's a more efficient implementation.

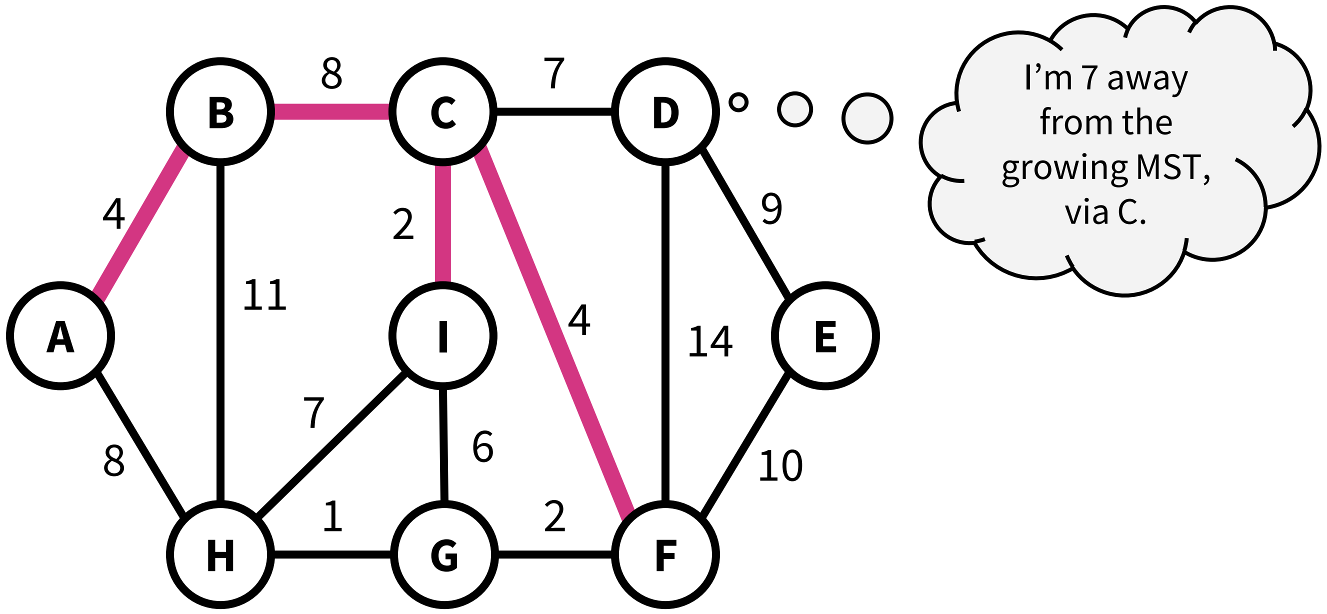
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.


A
B
C
D
E
F
G
H
I
8
7
4
2
11
4
9
14
7
6
8
1
2
10
I'm 7 away from the growing MST, via C.
I can't reach the growing MST in a single edge.

Prim's Algorithm

We called the algorithm `slow_prim`. There's a more efficient implementation.

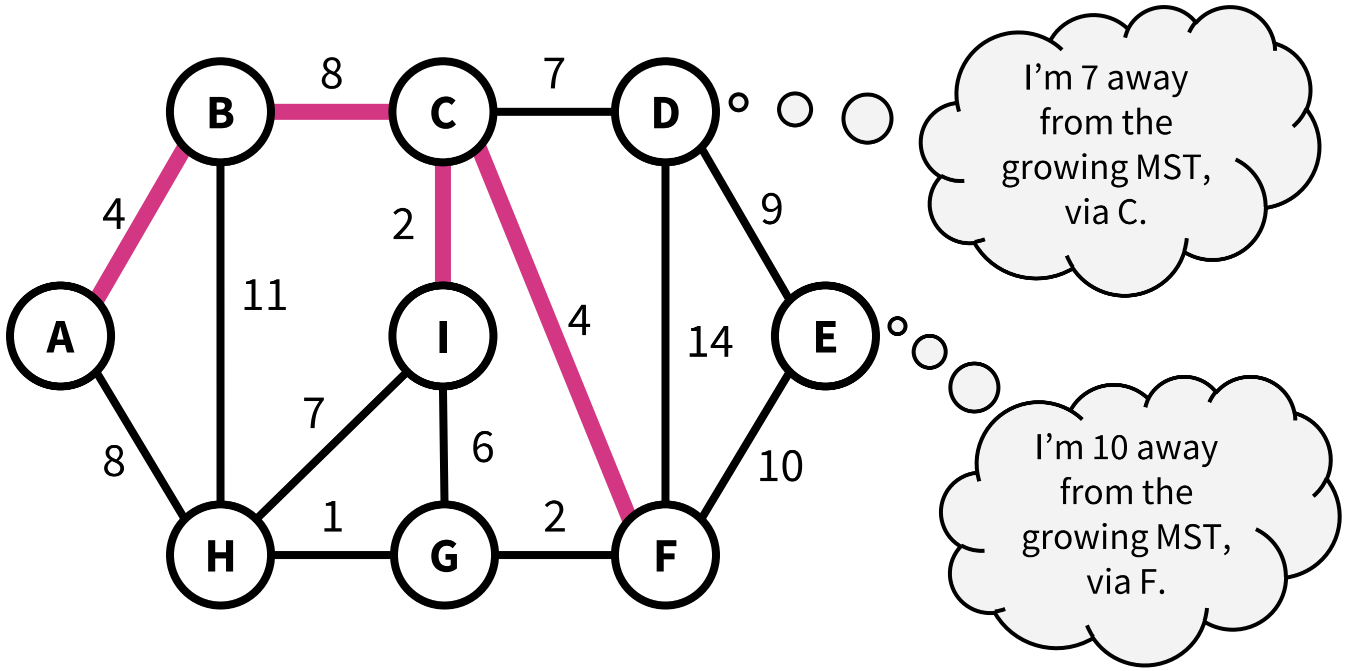
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.



Prim's Algorithm

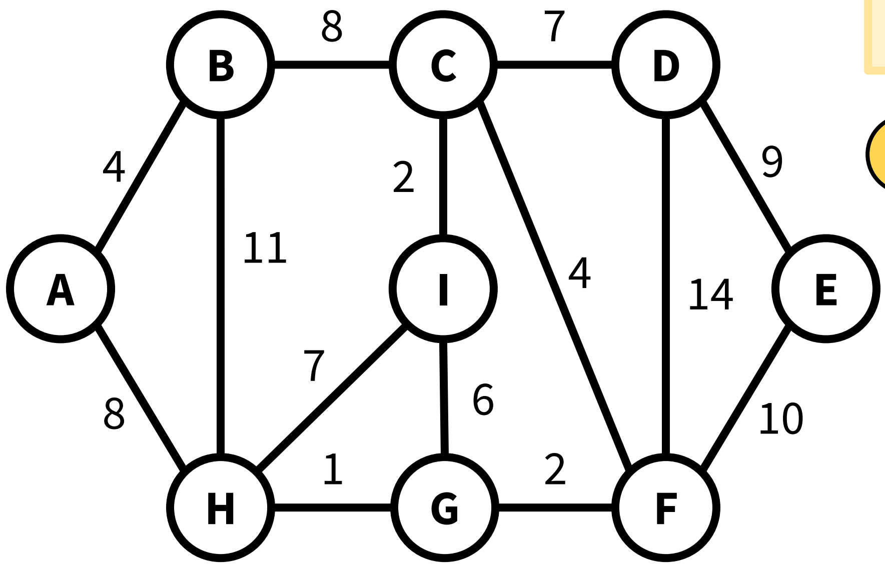
We called the algorithm `slow_prim`. There's a more efficient implementation.

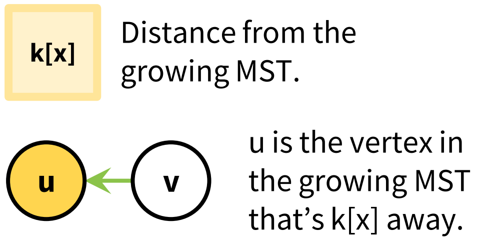
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.


A
B
C
D
E
F
G
H
I
8
7
4
11
2
4
14
9
7
6
8
1
2
10
I'm 7 away from the growing MST, via C.
I'm 10 away from the growing MST, via F.

Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.





Distance from the growing MST.

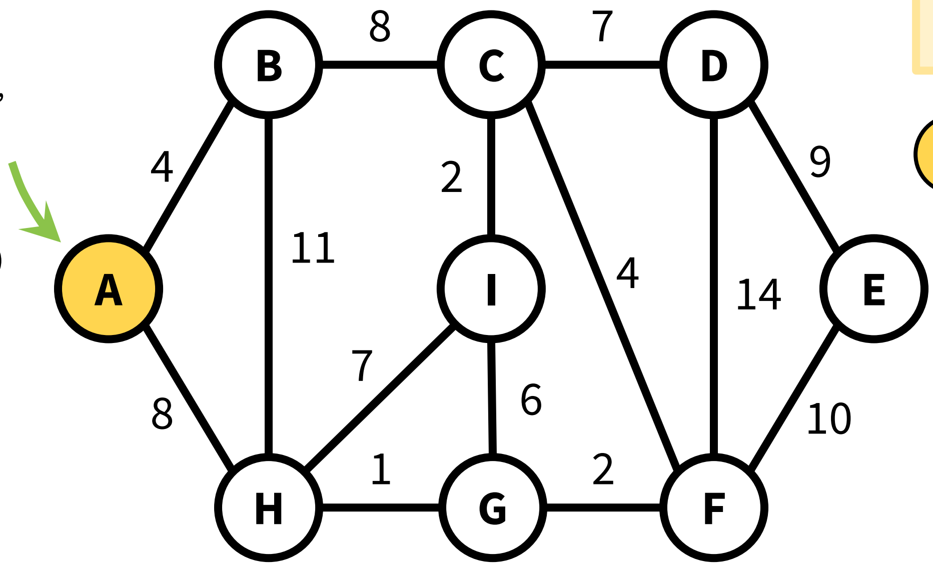
u is the vertex in the growing MST that's k[x] away.

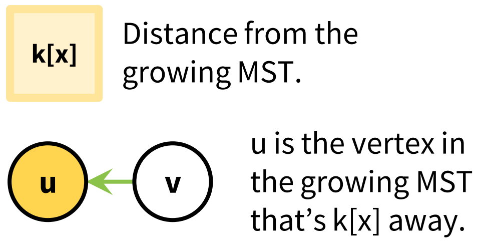
Prim’s Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.

In addition to visiting A, update information of its neighbors with:

```
for v in u.neighbors:
  k[v] = min(k[v], w(u, v))
```





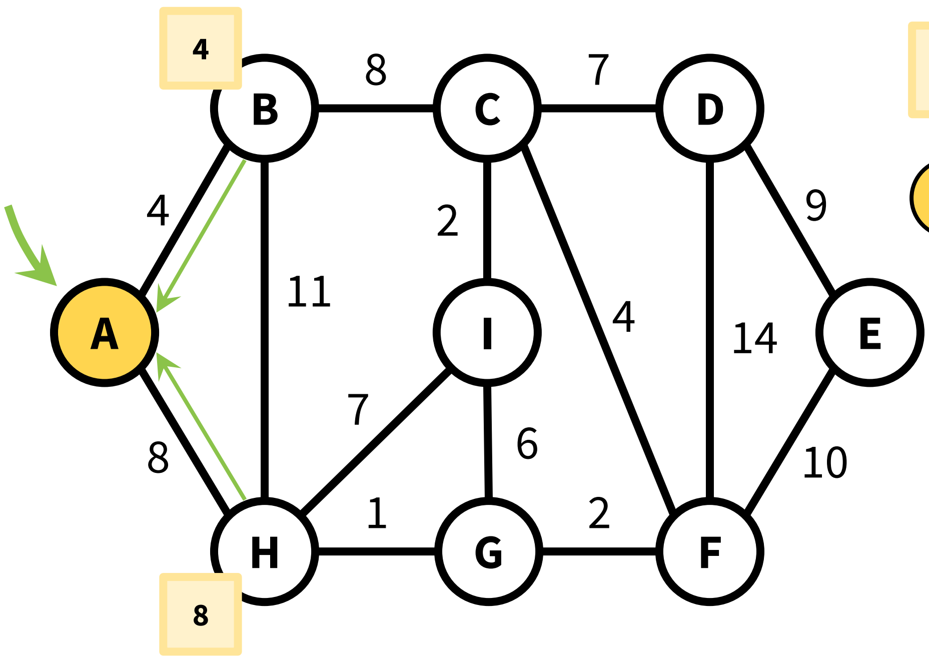
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that’s k[x] away.

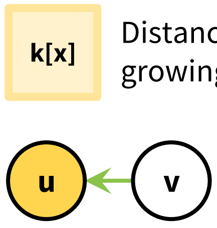
Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.

In addition to visiting A, update information of its neighbors with:

```
for v in u.neighbors:
  k[v] = min(k[v], w(u, v))
```



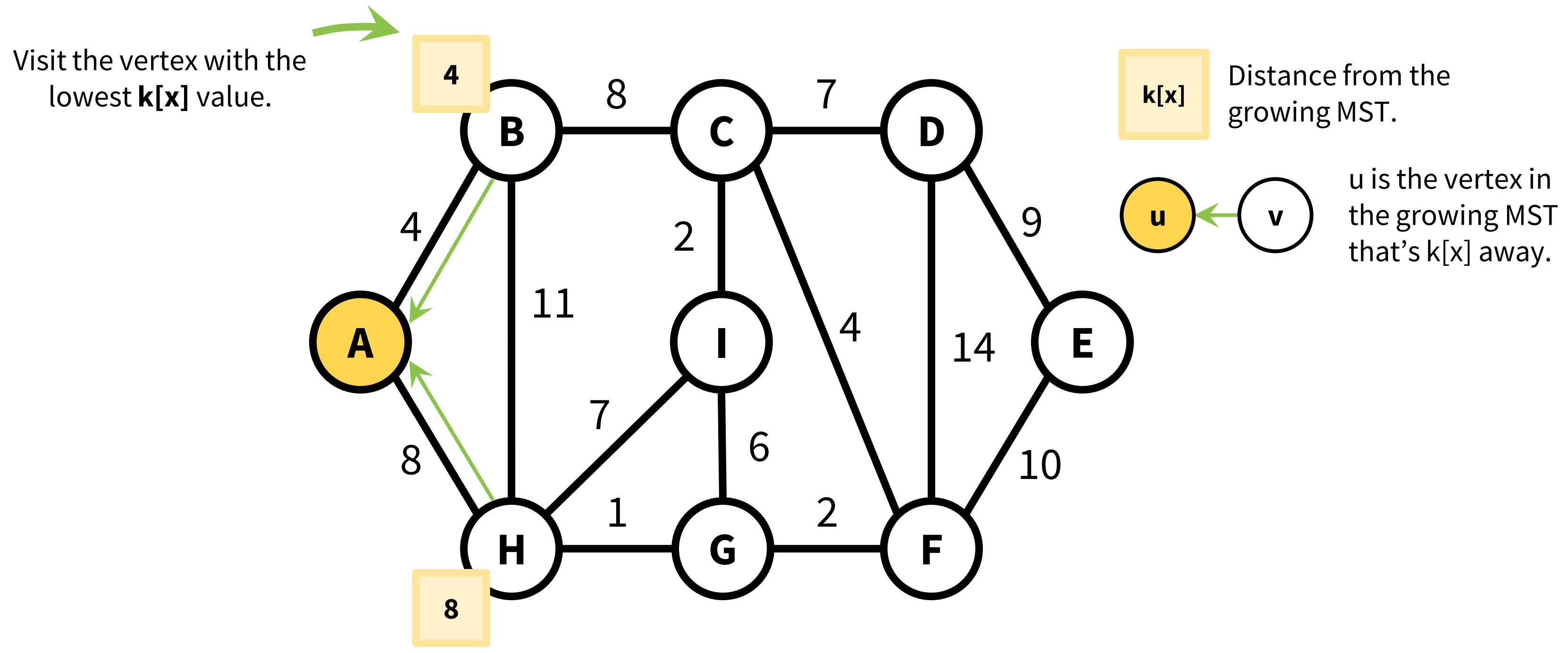


Distance from the growing MST.

u is the vertex in the growing MST that's k[x] away.

Prim's Algorithm

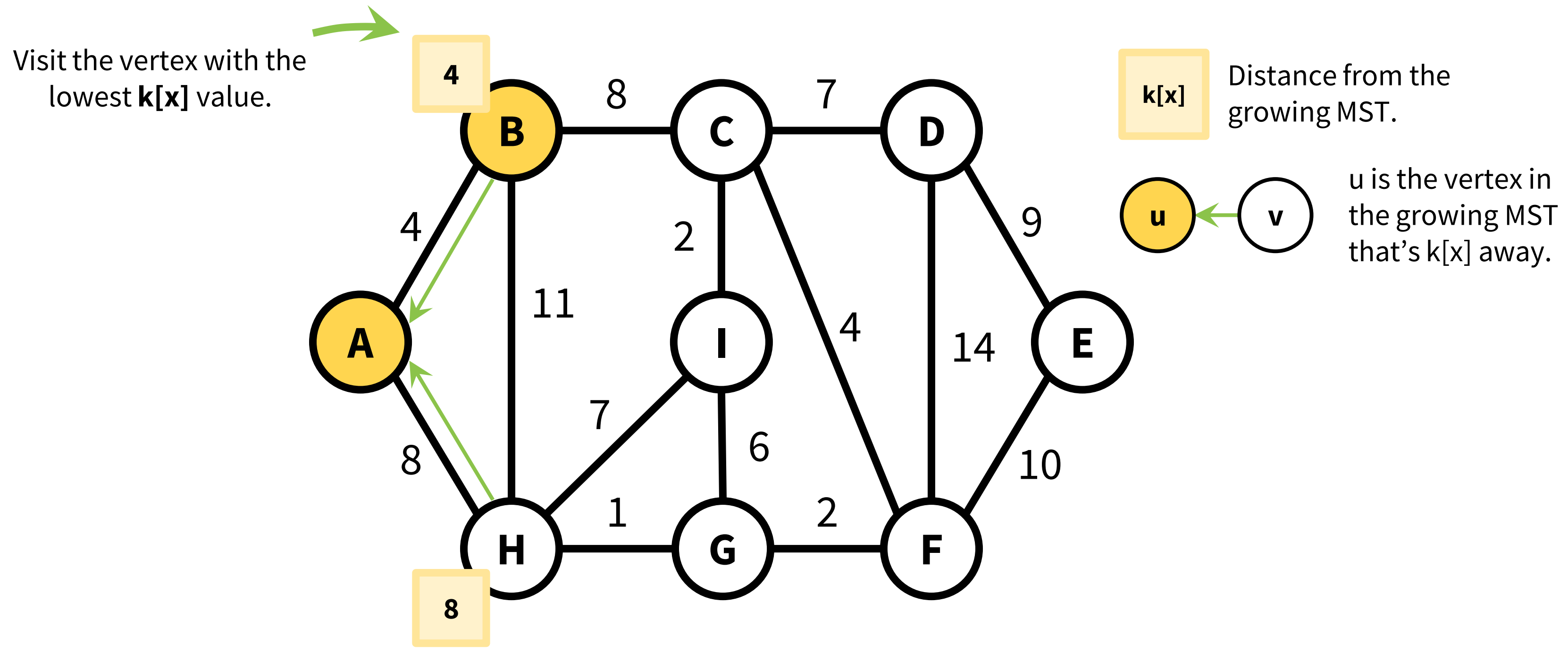
Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.



Visit the vertex with the lowest **k[x]** value.
4
8
A
B
C
D
E
F
G
H
I
8
7
4
11
2
9
4
14
7
6
8
1
2
10
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that's k[x] away.

Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.

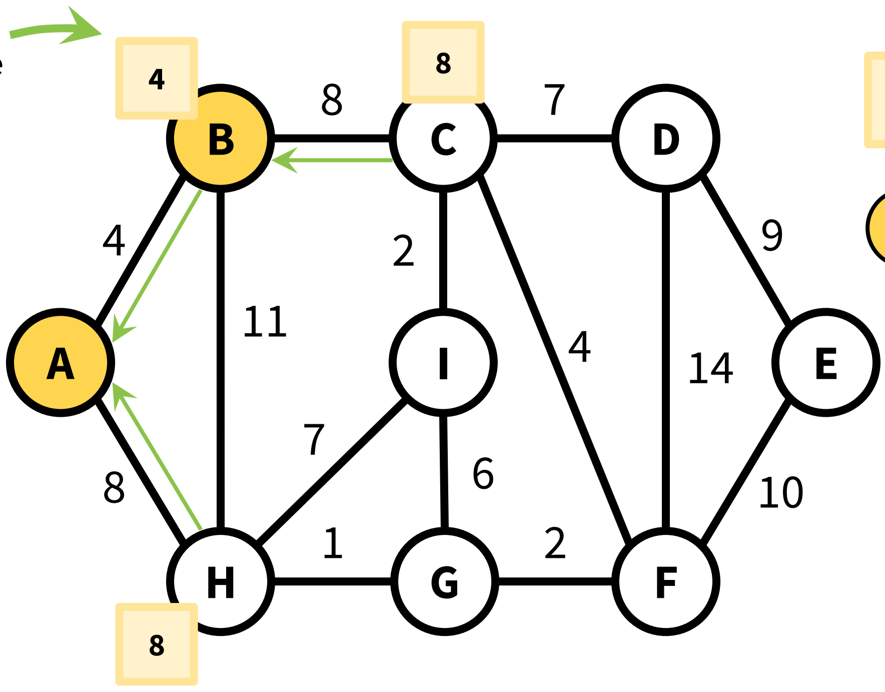


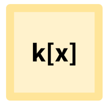
Visit the vertex with the lowest **k[x]** value.
4
8
A
B
C
D
E
F
G
H
I
8
7
4
11
2
4
9
14
8
7
6
10
1
2
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that's k[x] away.

Prim's Algorithm

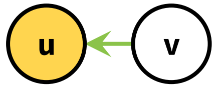
Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.

Visit the vertex with the lowest **k[x]** value.





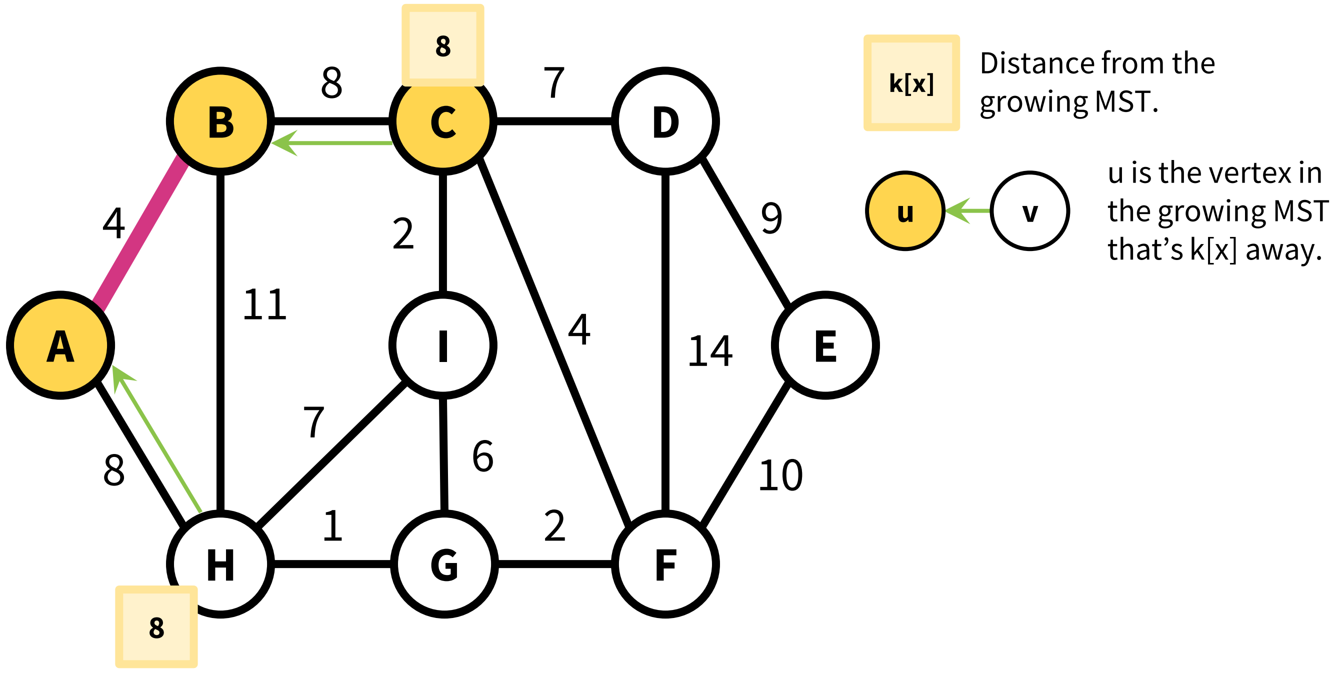
Distance from the growing MST.



u is the vertex in the growing MST that's k[x] away.

Prim's Algorithm

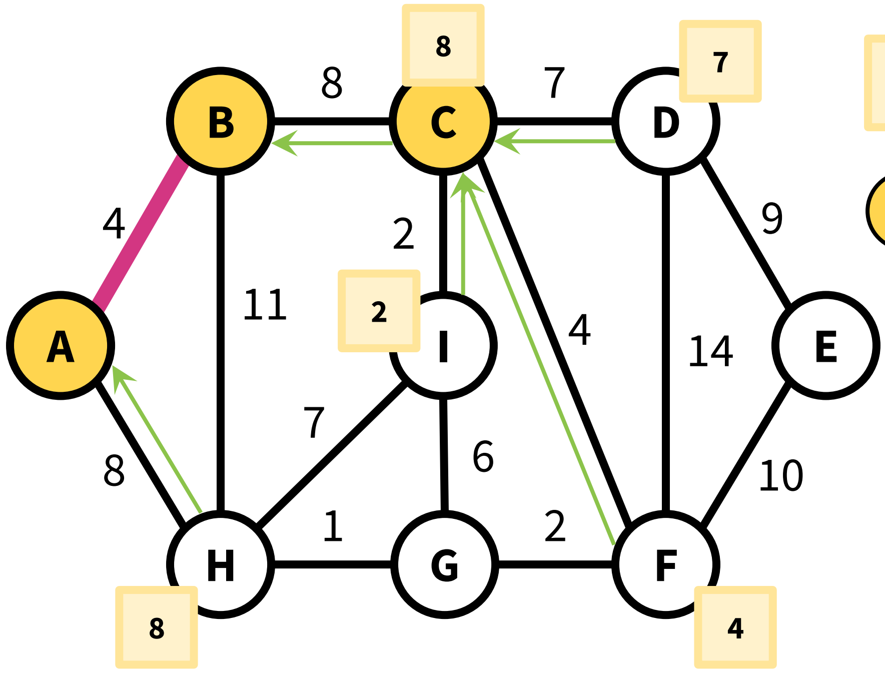
Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.

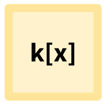


8
B
8
C
7
D
4
2
9
11
A
I
4
14
E
7
6
8
10
1
2
H
G
F
8
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that's k[x] away.

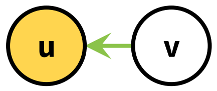
Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.





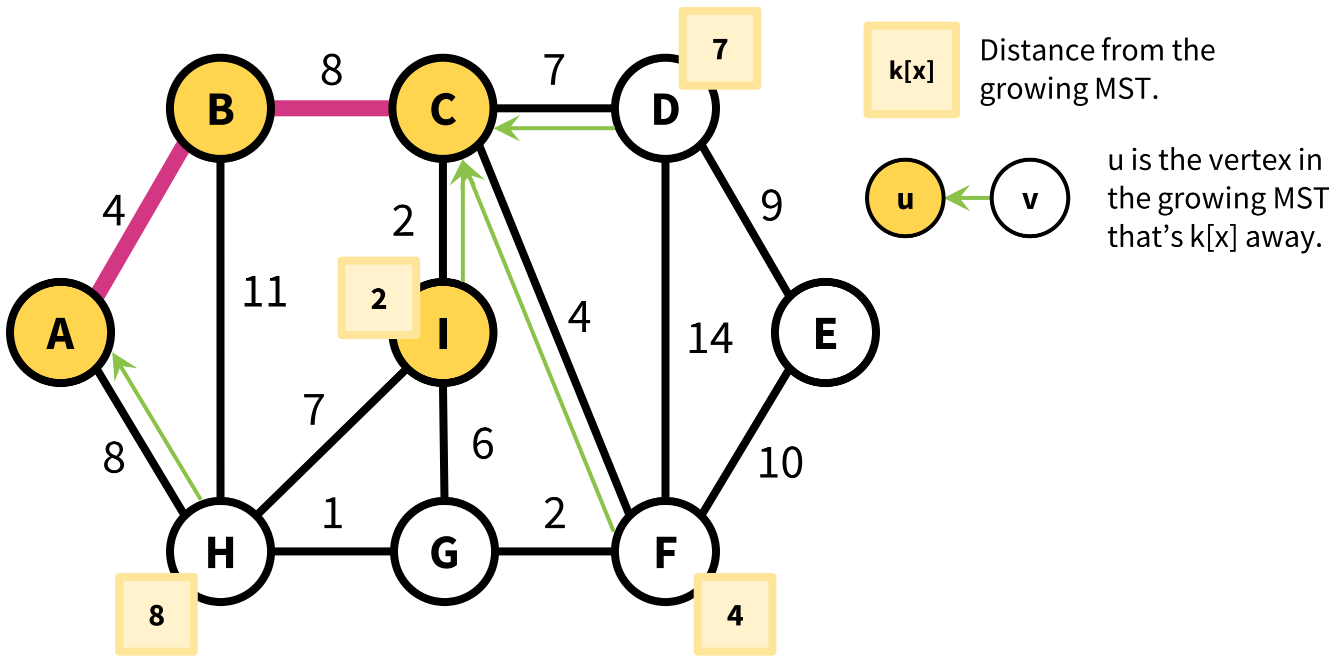
Distance from the growing MST.



u is the vertex in the growing MST that's k[x] away.

Prim’s Algorithm

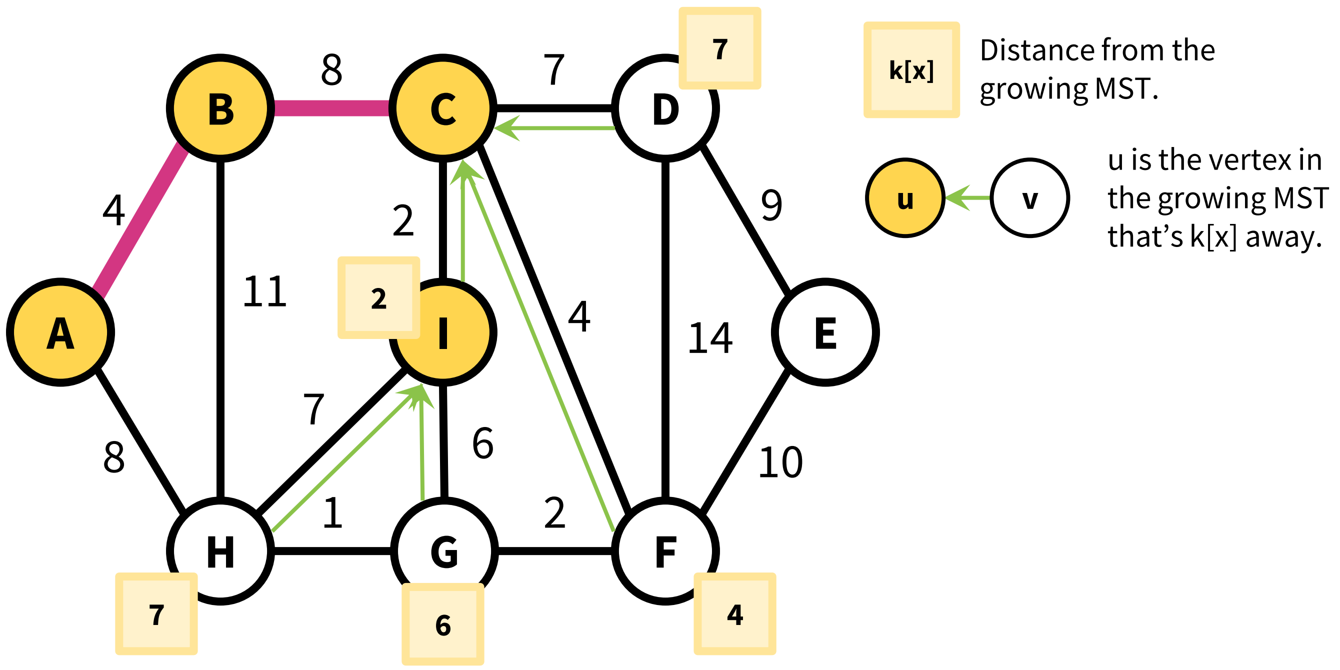
Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.



B
C
D
A
I
E
H
G
F
8
7
4
11
2
9
4
14
8
7
6
10
1
2
7
2
8
4
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that’s k[x] away.

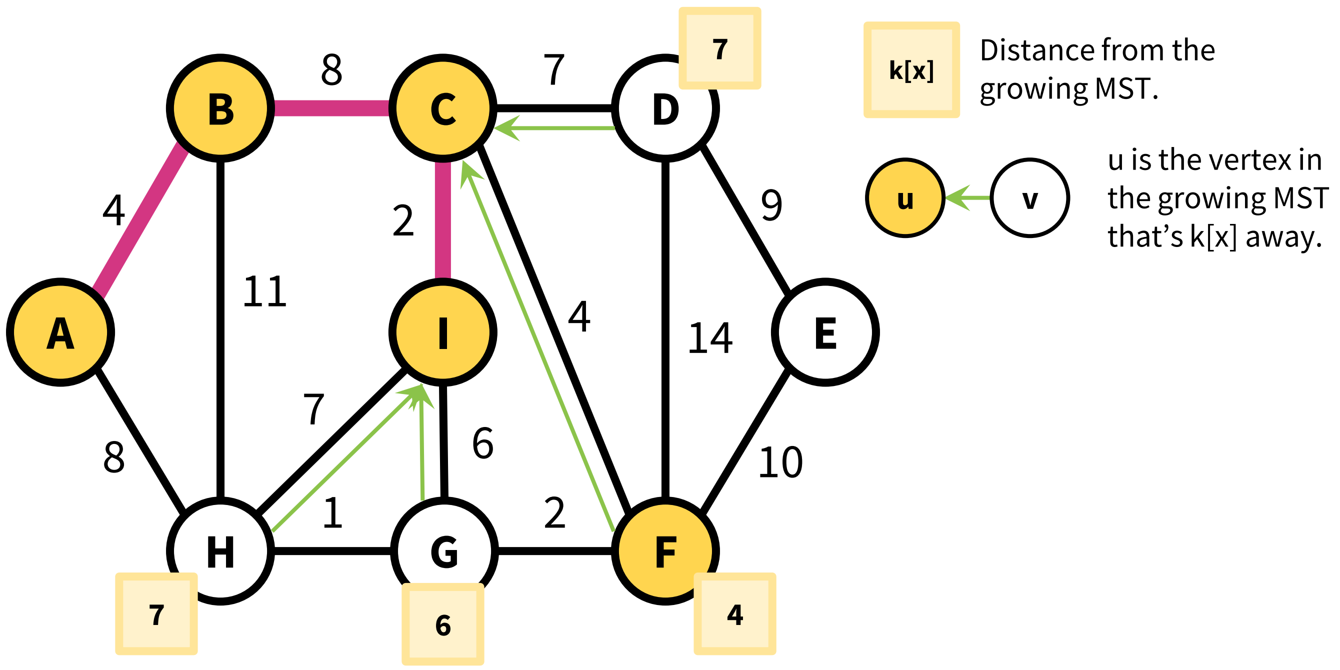
Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.


A
B
C
D
E
F
G
H
I
8
7
4
11
2
9
14
4
7
6
8
1
2
10
7
2
7
6
4
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that's k[x] away.

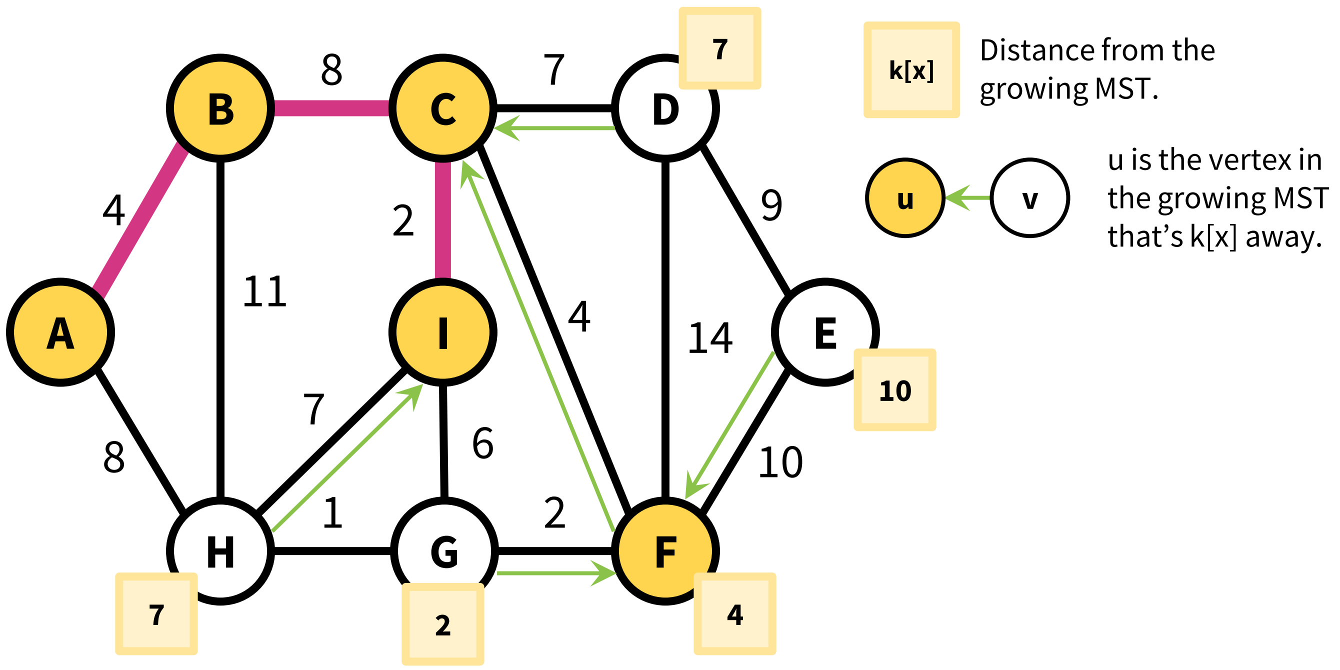
Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.


A
B
C
D
E
F
G
H
I
8
7
4
2
11
9
4
14
7
6
8
1
2
10
7
7
6
4
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that's k[x] away.

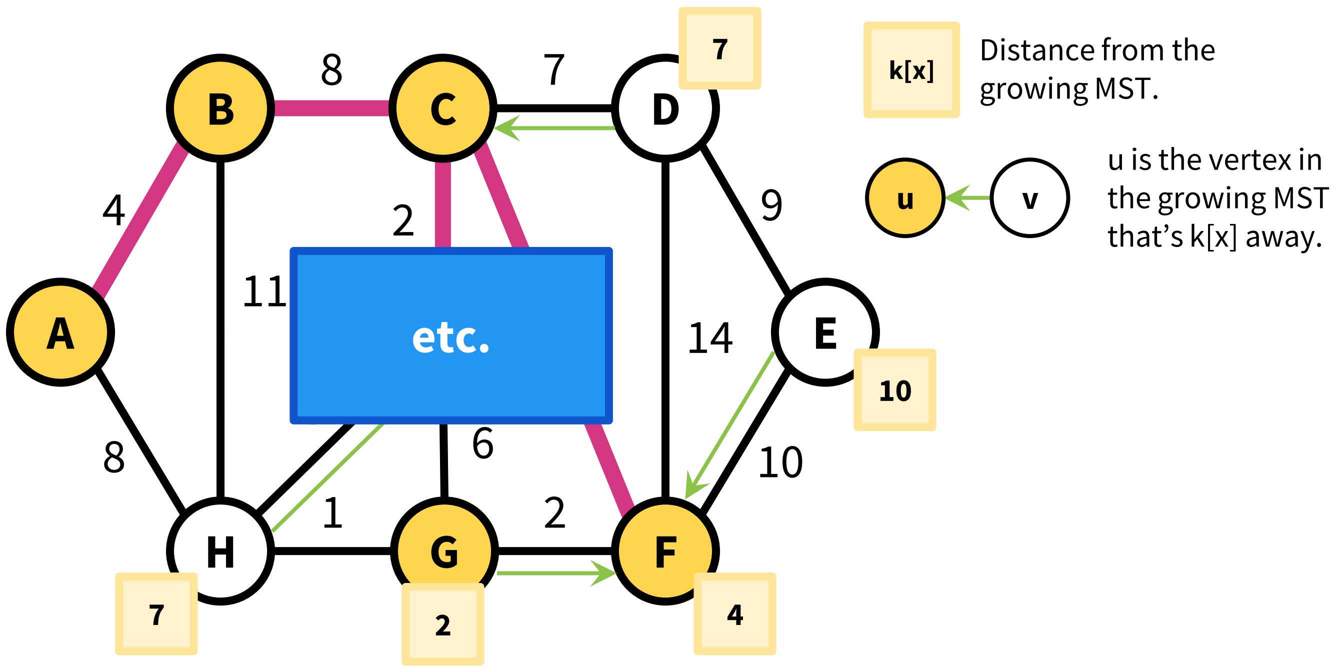
Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.


A
B
C
D
E
F
G
H
I
8
7
4
2
11
4
14
9
7
6
8
1
2
10
7
7
10
2
4
7
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that's k[x] away.

Prim's Algorithm

Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.



A
B
C
D
E
F
G
H
4
8
8
7
2
11
9
14
10
6
1
2
etc.
7
10
4
2
7
k[x]
Distance from the growing MST.
u
v
u is the vertex in the growing MST that's k[x] away.

Prim's Algorithm

```
algorithm prim(G):
  s = random vertex in G
  MST = {}
  visited_vertices = {s}
  update_info(G, s)
  while |visited_vertices| < |V|:
    (x, v) = lightest_edge(G, visited_vertices)
    MST.add((x, v))
    visited_vertices.add(v)
    update_info(G, v)
  return MST
```

Updates information about distance from the growing MST.

Runtime:

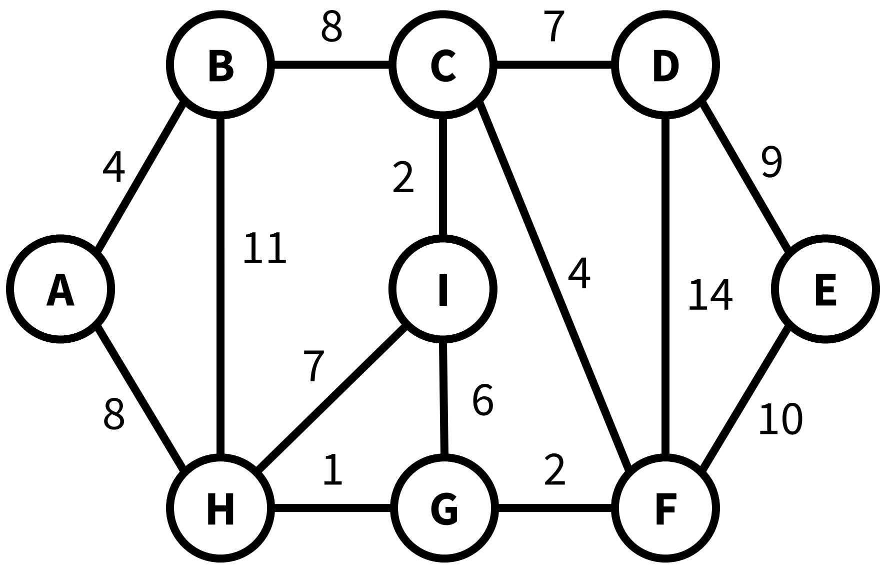
$O(|E|\log(|V|))$ Using a red-black tree as a priority queue

$O(|E|+|V|\log(|V|))$ Using a fibonacci heap

Kruskal’s Algorithm

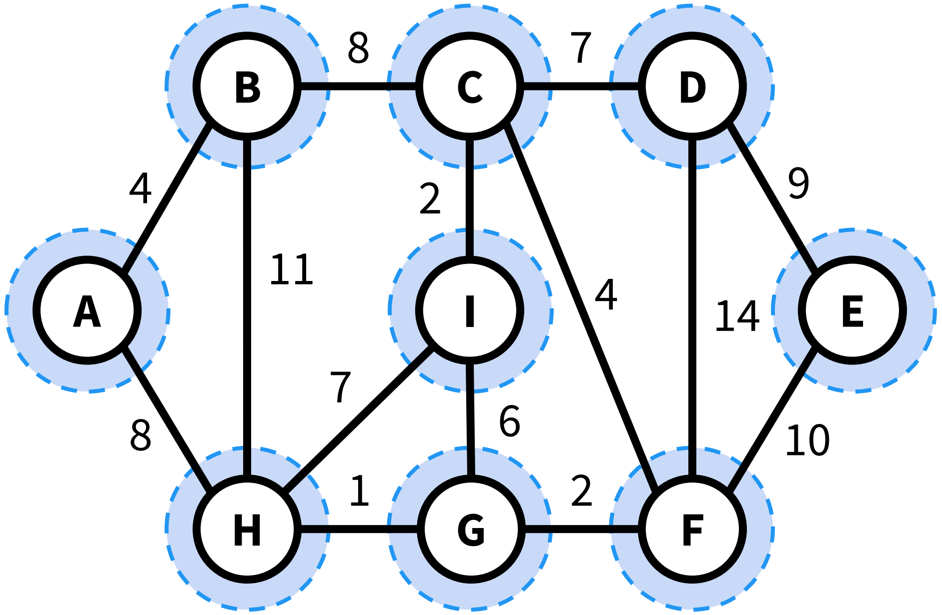
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



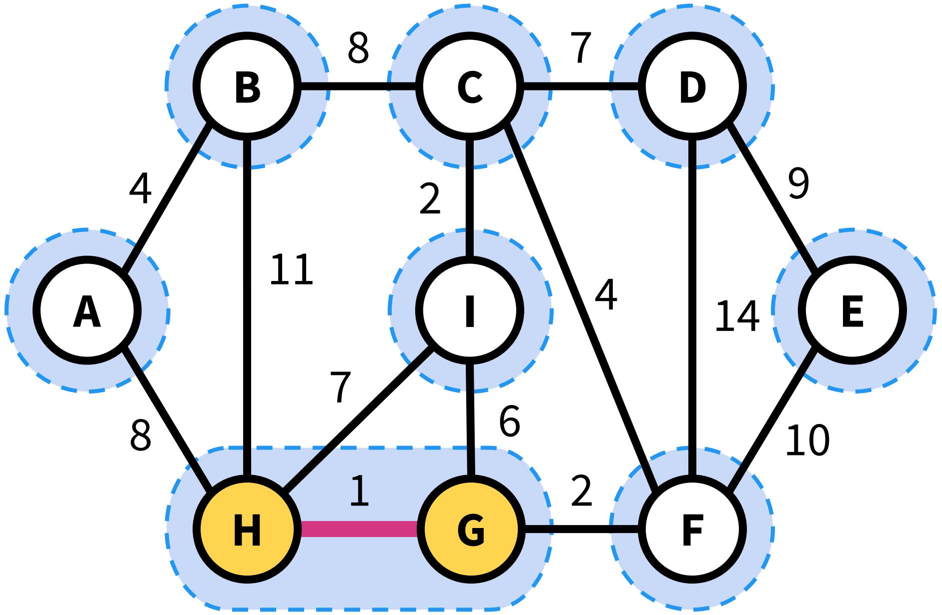
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



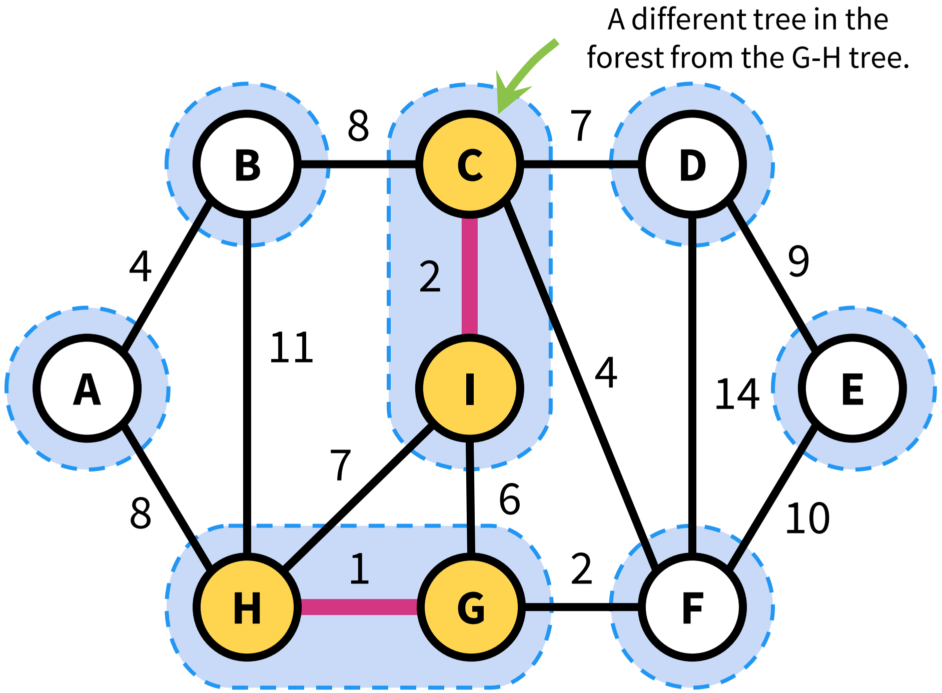
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



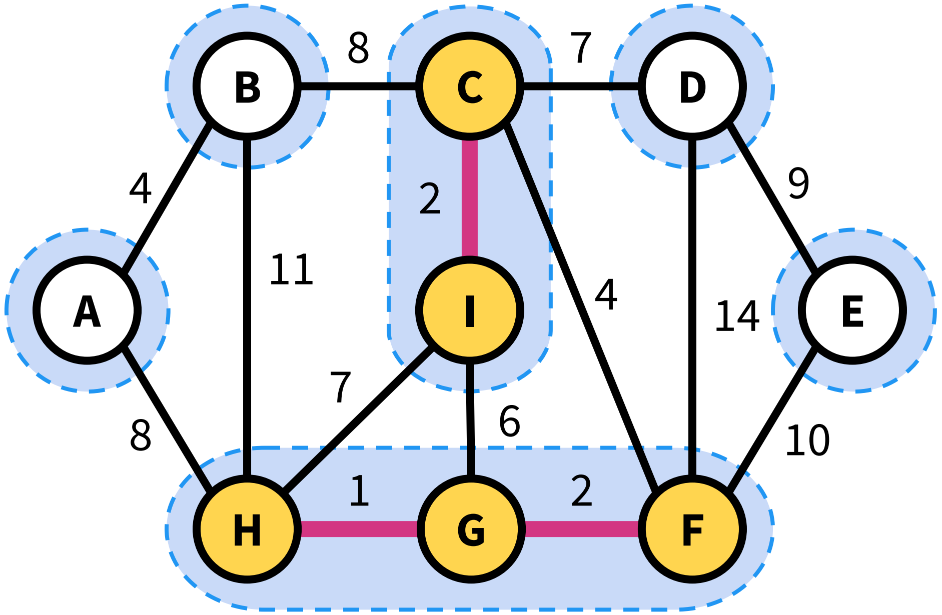
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



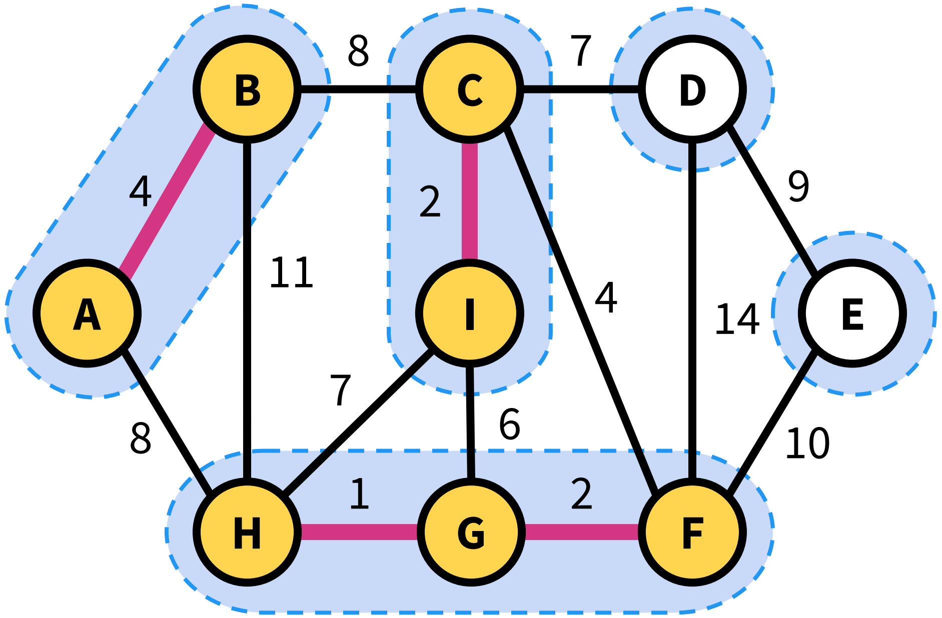
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



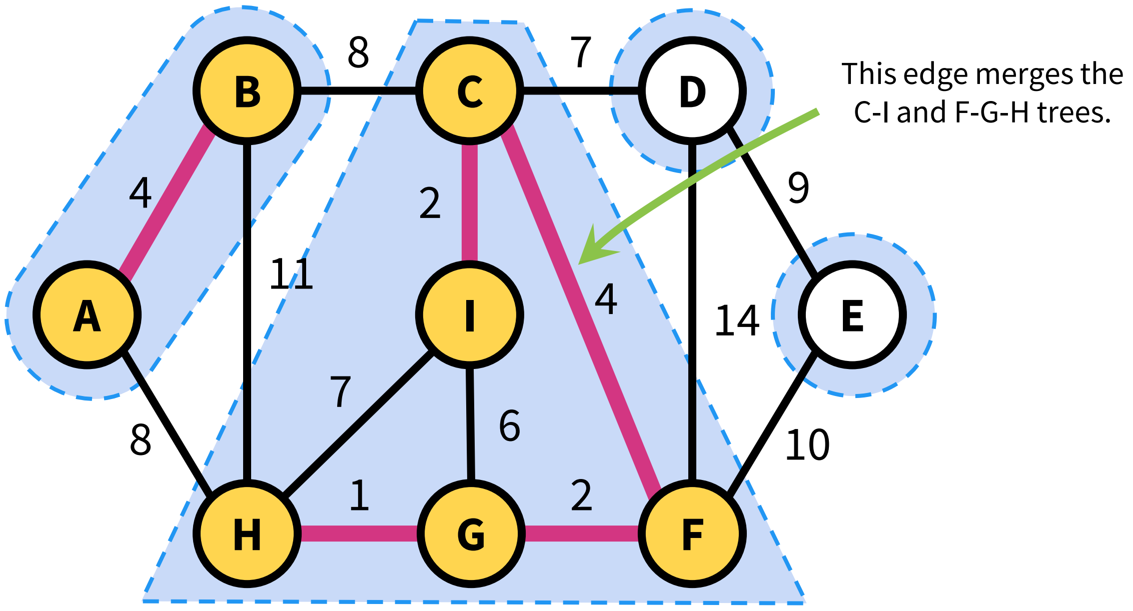
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



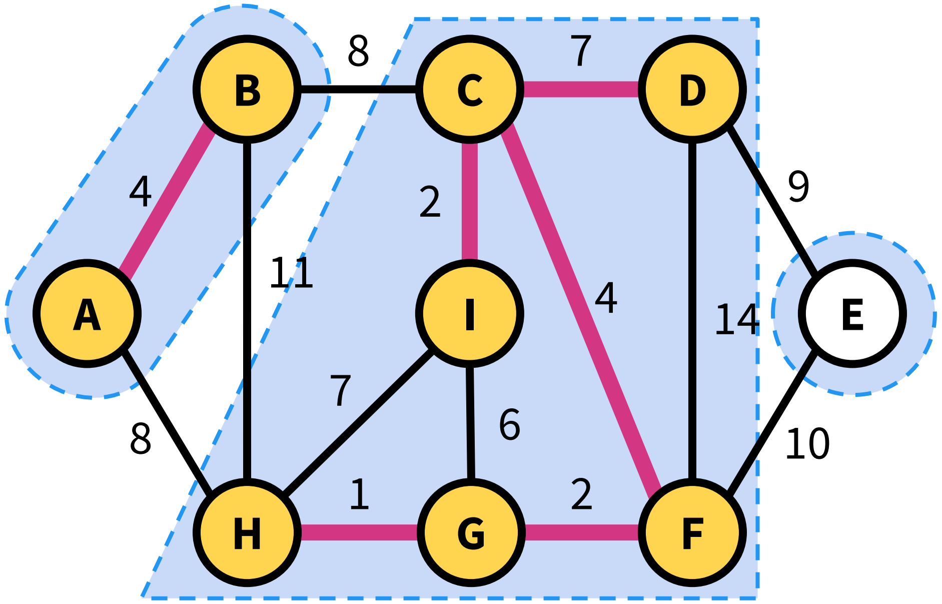
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Kruskal's Algorithm

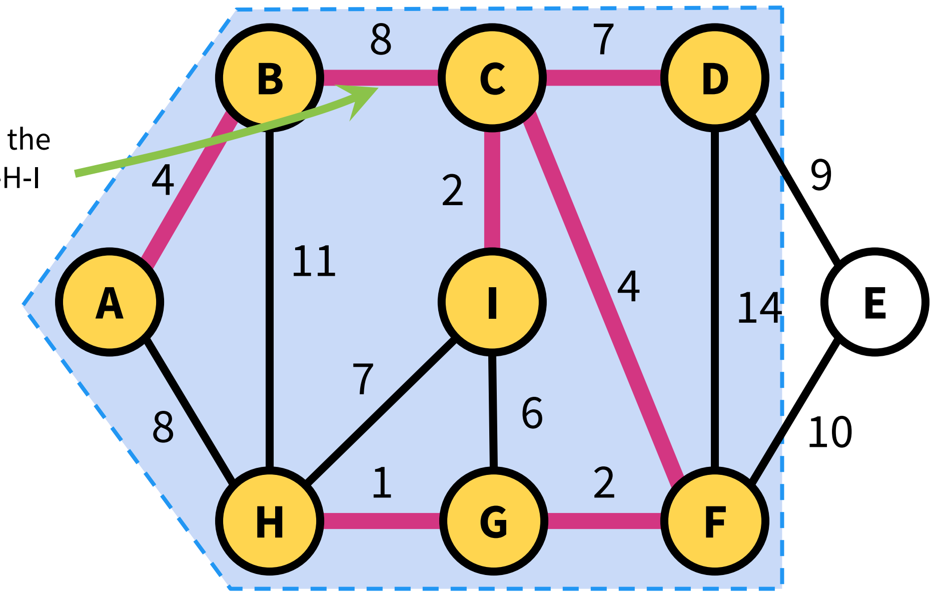
Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Kruskal's Algorithm

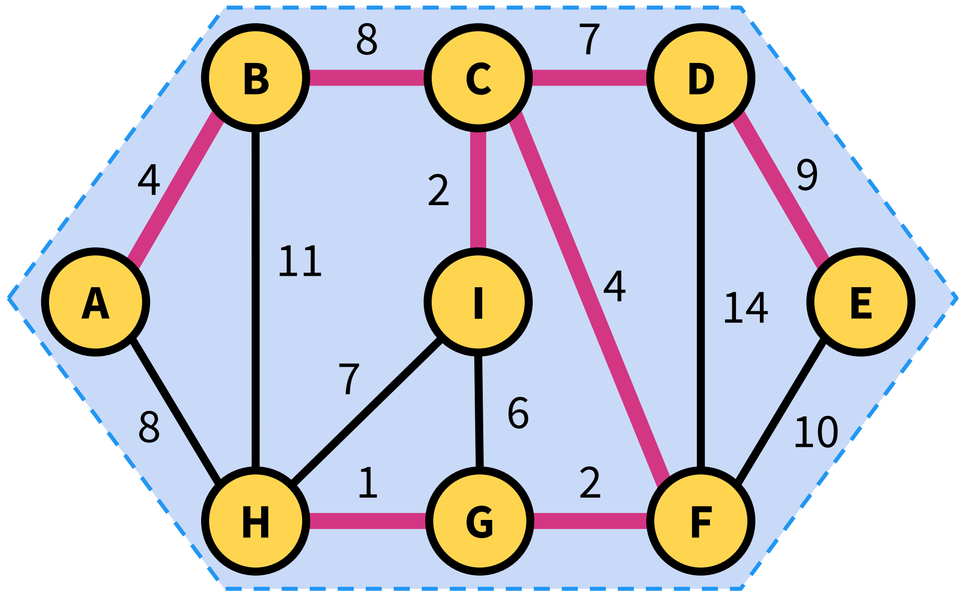
Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.

This edge merges the A-B and C-D-F-G-H-I trees.



Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Kruskal's Algorithm

`kruskal` uses union-find data structure, which supports …

- `make_set(u)`: create a set {u} in **`O(1)`**
- `find(u)`: returns the set containing u in **`O(1)`**
- `union(u,v)`: merges the sets containing u and v in **`O(1)`**

Technically, these operations all run in amortized-time **`α(|V|)`**; $\alpha(n) \leq 4$, provided $n <$ # of atoms in the universe. We will discuss amortized analysis in greater detail later this quarter.

Kruskal’s Algorithm

```
algorithm kruskal(G):
  E_sorted = sort the edges in E by non-decreasing weight
  MST = {}
  for v in V:
    make_set(v)  # put each vertex in its own tree
  for (u, v) in E_sorted:
    if find(u) != find(v):  # u and v in different trees
      MST.add((u, v))
      union(u, v)  # merge u’s tree with v’s tree
  return MST
```

Runtime:

$O(|E|\log(|V|))$ ← Using comparison-based sort. Note $|E|\log(|E|) = O(|E|\log(|V|^2)) = O(|E| \cdot 2\log(|V|) = O(|E|\log(|V|))$.

$O(|E|)$ ← Using radix sort

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

Kruskal’s Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there’s an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

`kruskal` finds an edge (u, v) that merges two trees T_1 and T_2. Consider the cut $\{T_1, V - T_1\}$; `MST` respects this cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(u, v)}.

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

`kruskal` finds an edge (u, v) that merges two trees T_1 and T_2. Consider the cut $\{T_1, V - T_1\}$; `MST` respects this cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(u, v)}.

Recall, we proved our lemma with an exchange argument!

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

`kruskal` finds an edge (u, v) that merges two trees T_1 and T_2. Consider the cut $\{T_1, V - T_1\}$; `MST` respects this cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(u, v)}.

Recall, we proved our lemma with an exchange argument!

After adding the the (n-1)st edge, we have a spanning tree; therefore, `MST` contains a minimum spanning tree. ∎

Prim’s and Kruskal’s

	Description	Runtime	Use-cases
Prim’s	Grows a tree	`O(|E|log(|V|))` **with red-black tree** `O(|E|+|V|log(|V|))` **with Fibonacci heap**	Better on dense graphs
Kruskal’s	Grows a forest	`O(|E|log(|V|))` **with union-find** `O(|E|)` **with union-find and radix sort**	Better on sparse graphs and if the edge weights can be radix sorted.

Beyond Prim's and Kruskal's

Karger-Klein-Tarjan (1995): Las Vegas randomized algorithm

- $O(|E|)$ expected, $O(\min\{|E|\log(|V|), |V|^2\})$ worst-case

Chazelle (2000): $O(|E|\alpha(|V|))$ deterministic algorithm

Inverse Ackermann function