Greedy Algorithms I

Summer 2018 • Lecture 08/02

A Few Notes

Homework 4

Homework 5

- Original soft deadline was 8/9. I’m extending it to 8/10 at 5 p.m. due to the lecture schedule.

Course Overview

- Algorithmic Analysis
- Divide and Conquer
- Randomized Algorithms
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- **Greedy Algorithms**
- Advanced Algorithms

Outline for Today

Greedy algorithms

- Frog Hopping
- Minimum Spanning Trees

Frog Hopping

Greedy Algorithms

Greedy algorithms construct solutions one step at a time, at each step choosing the locally best option.

Advantages: simple to design, often efficient

Disadvantages: difficult to verify correctness or optimality

Freddie the Frog

Freddie the Frog starts at position 0 along a river. His goal is to reach position n.

There are lilypads at various positions, including at position 0 and position n.

Freddie can hop at most r units at a time.

Task: Find the path Freddie should take to minimize hops, assuming such a path exists.

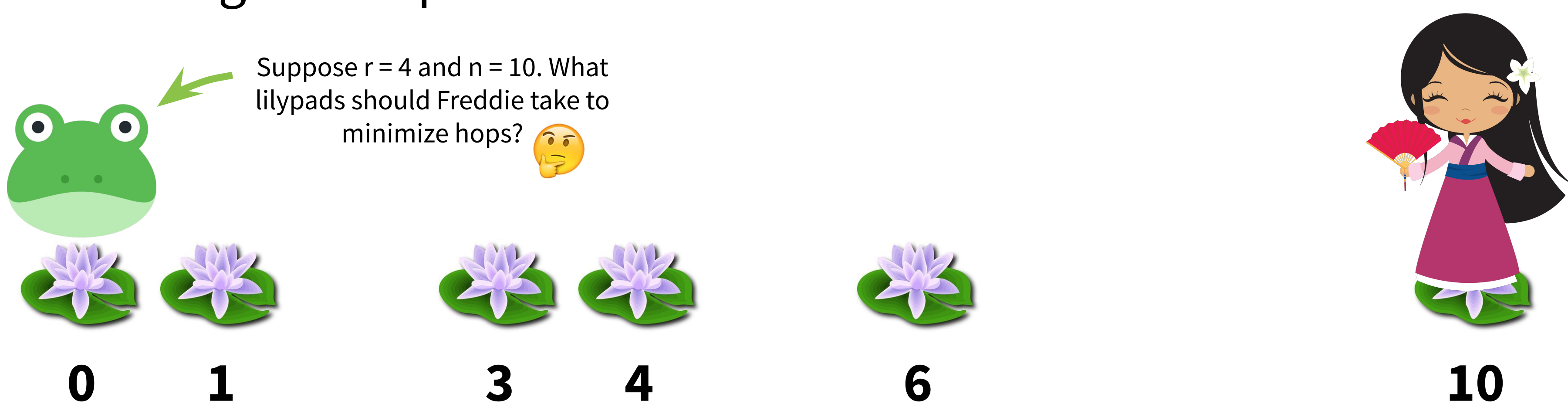
Freddie the Frog

Freddie the Frog starts at position 0 along a river. His goal is to reach position n (with Princess Paula).

There are lilypads at various positions, including at position 0 and position n.

Freddie can hop at most r units at a time.

Task: Find the path Freddie should take to minimize hops, assuming such a path exists.



Frog Hopping

```python
def frog_hopping(lilys, r, n):
  # lilys = [0, 1, 3, 4, 6, 10] in the previous example
  H = [0]  # contains hops
  cur_lily = {"index": 0, "position": 0}
  while cur_lily["position"] < n:
    next_lily = furthest_reachable_lily(
      cur_lily, lilys, r
    )  # finds the furthest lilypad still reachable
       # from cur_lily
    H.append(next_lily["position"])
    cur_lily = next_lily
  return H
```

You should be able to implement this function yourself.

Runtime: `O(n)`

Frog Hopping

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules).

(2) **Optimality.** The algorithm finds an optimal series of hops (i.e. there isn't a better path available).

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Suppose it did not. A path might be infeasible for one of three reasons:

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Suppose it did not. A path might be infeasible for one of three reasons:
(1) H.first ≠ 0, (2) H[k] + r < H[k+1] for some k, or (3) H.last ≠ n.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first ≠ 0, (2) H[k] + r < H[k+1] for some k, or (3) H.last ≠ n.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first ≠ 0, (2) H[k] + r < H[k+1] for some k, or (3) H.last ≠ n.

Since the algorithm initializes H to [0], (1) is impossible.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first ≠ 0, (2) H[k] + r < H[k+1] for some k, or (3) H.last ≠ n.

Since the algorithm initializes H to [0], (1) is impossible.

By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first $\neq$ 0, (2) H[k] + r < H[k+1] for some k, or (3) H.last $\neq$ n.

Since the algorithm initializes H to [0], (1) is impossible.

By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

By construction of the function `furthest_reachable_lily`, H.last $\not>$ n. To prove that H.last $\not<$ n, we proceed by contradiction.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first ≠ 0, (2) H[k] + r < H[k+1] for some k, or (3) H.last ≠ n.

Since the algorithm initializes H to [0], (1) is impossible.

By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

By construction of the function `furthest_reachable_lily`, H.last $\not>$ n. To prove that H.last $\not<$ n, we proceed by contradiction. Assume that H.last < n. This happens if our algorithm fails to halt, and `furthest_reachable_lily` must have been unable to reach lilypad k+1 from some lilypad k.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first $\neq$ 0, (2) H[k] + r < H[k+1] for some k, or (3) H.last $\neq$ n.

Since the algorithm initializes H to [0], (1) is impossible.

By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

By construction of the function `furthest_reachable_lily`, H.last $\ngtr$ n. To prove that H.last $\nless$ n, we proceed by contradiction. Assume that H.last < n. This happens if our algorithm fails to halt, and `furthest_reachable_lily` must have been unable to reach lilypad k+1 from some lilypad k.

aka “gets stuck”

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first $\neq 0$, (2) $H[k] + r < H[k+1]$ for some k, or (3) H.last $\neq n$.

Since the algorithm initializes H to [0], (1) is impossible.

By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

aka "gets stuck"

By construction of the function `furthest_reachable_lily`, H.last $\not> n$. To prove that H.last $\not< n$, we proceed by contradiction. Assume that H.last $< n$. This happens if our algorithm fails to halt, and `furthest_reachable_lily` must have been unable to reach lilypad k+1 from some lilypad k. Since there exists a path from the first lilypad to the last lilypad, there **must** be some hop in that path that starts at lilypad $s < k$ and ends at or after lilypad k+1.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first $\neq 0$, (2) $H[k] + r < H[k+1]$ for some k, or (3) H.last $\neq n$.

Since the algorithm initializes H to [0], (1) is impossible.

By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

aka "gets stuck"

i.e. the path from the first lilypad to the last one

By construction of the function `furthest_reachable_lily`, H.last $\ngtr$ n. To prove that H.last $\nless$ n, we proceed by contradiction. Assume that H.last $<$ n. This happens if our algorithm fails to halt, and `furthest_reachable_lily` must have been unable to reach lilypad k+1 from some lilypad k. Since there exists a path from the first lilypad to the last lilypad, there **must** be some hop in that path that starts at lilypad $s < k$ and ends at or after lilypad k+1.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first $\neq 0$, (2) $H[k] + r < H[k+1]$ for some k, or (3) H.last $\neq n$.

Since the algorithm initializes H to [0], (1) is impossible.

By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

aka "gets stuck"

i.e. the path from the first lilypad to the last one

By construction of the function `furthest_reachable_lily`, H.last $\not>$ n. To prove that H.last $\not<$ n, we proceed by contradiction. Assume that H.last < n. This happens if our algorithm fails to halt, and `furthest_reachable_lily` must have been unable to reach lilypad k+1 from some lilypad k. Since there exists a path from the first lilypad to the last lilypad, there **must** be some hop in that path that starts at lilypad $s < k$ and ends at or after lilypad k+1. But then we have $lilys[s] + r < lilys[k] + r < lilys[k+1]$, so this hop is illegal.

Frog Hopping

Lemma 1: `frog_hopping` always finds a feasible series of hops for Freddie.

Proof: We proceed by contradiction.

Notation for the first element in list H.

Suppose it did not. A path might be infeasible for one of three reasons: (1) H.first $\neq 0$, (2) $H[k] + r < H[k+1]$ for some k, or (3) H.last $\neq n$.

Since the algorithm initializes H to [0], (1) is impossible.

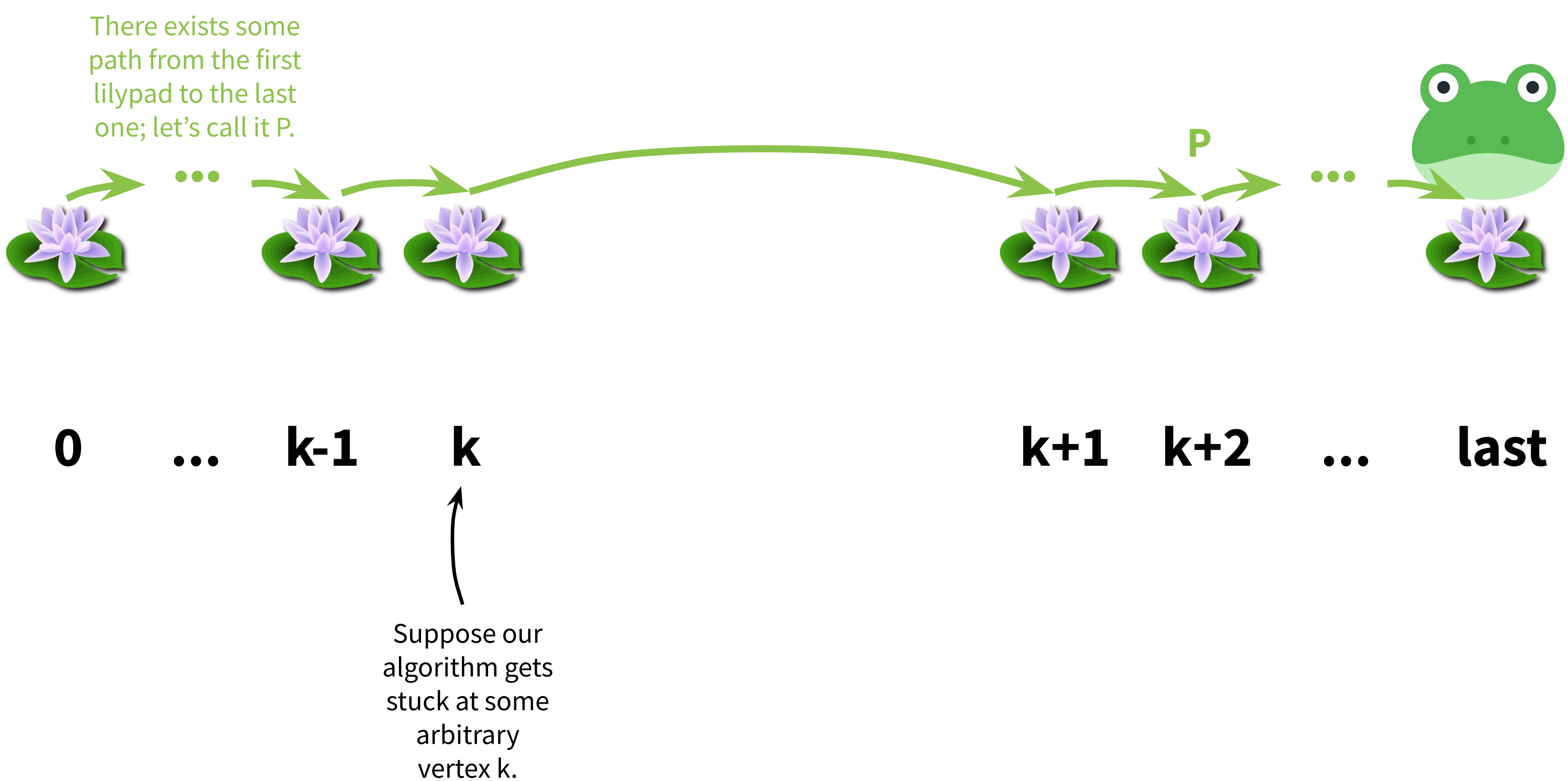
By construction of the function `furthest_reachable_lily`, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

aka "gets stuck"

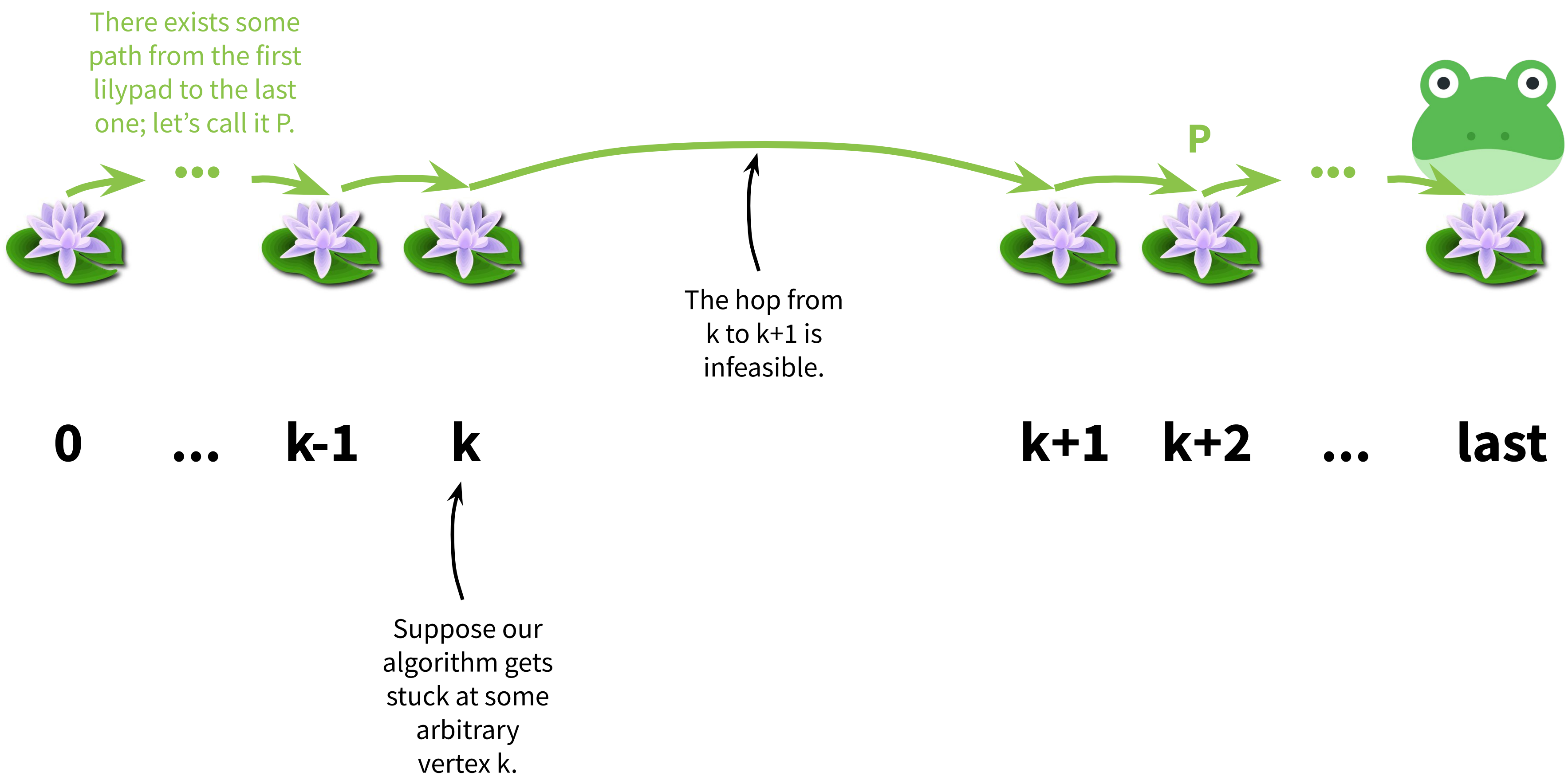
i.e. the path from the first lilypad to the last one

By construction of the function `furthest_reachable_lily`, H.last $\ngtr n$. To prove that H.last $\nless n$, we proceed by contradiction. Assume that H.last $< n$. This happens if our algorithm fails to halt, and `furthest_reachable_lily` must have been unable to reach lilypad k+1 from some lilypad k. Since there exists a path from the first lilypad to the last lilypad, there **must** be some hop in that path that starts at lilypad $s < k$ and ends at or after lilypad k+1. But then we have $lilys[s] + r < lilys[k] + r < lilys[k+1]$, so this hop is illegal. We have reached a contradiction, so our assumption must have been incorrect; therefore, (3) is impossible. ■

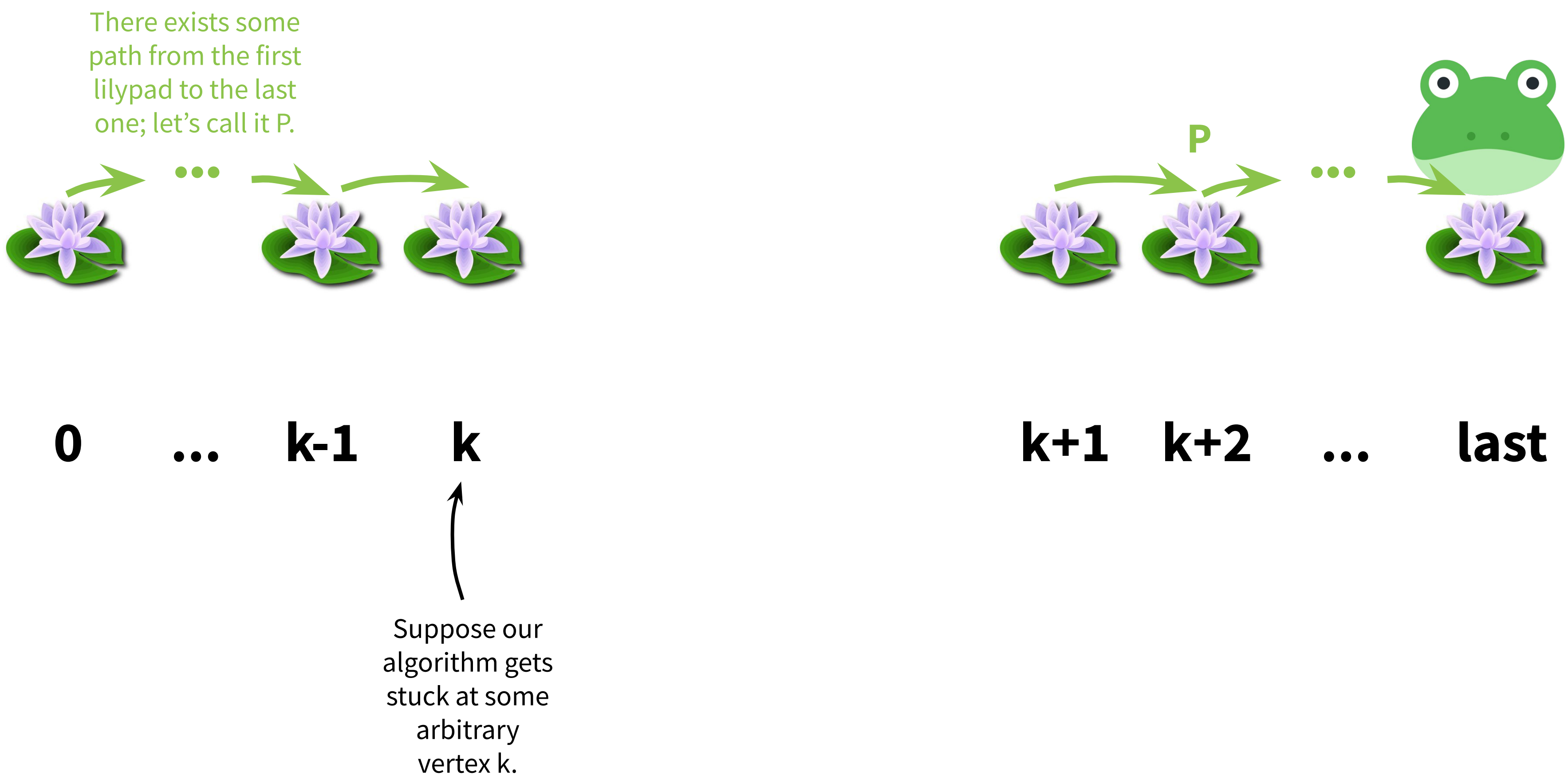
Frog Hopping



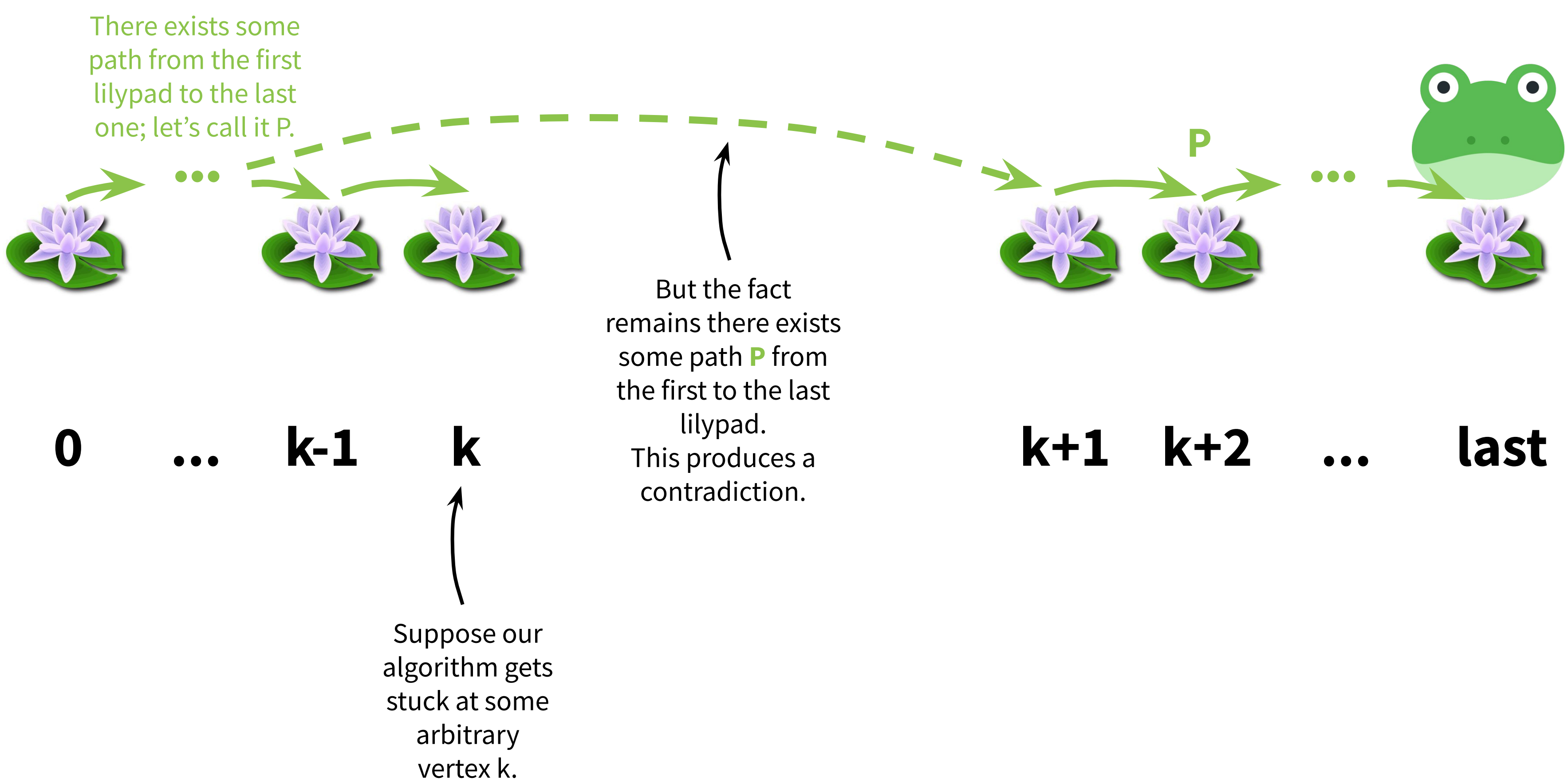
Frog Hopping



Frog Hopping



Frog Hopping



Frog Hopping

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules).

(2) **Optimality.** The algorithm finds an optimal series of hops (i.e. there isn't a better path available).

Frog Hopping

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules). 👌

(2) **Optimality.** The algorithm finds an optimal series of hops (i.e. there isn't a better path available).

Frog Hopping

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules). 👌

(2) **Optimality.** The algorithm finds an optimal series of hops (i.e. there isn't a better path available). 🤔

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` always finds an optimal series of hops?

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` always finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision …

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` always finds an optimal series of hops?

- Let's introduce notation to talk about the algorithm with greater precision …
- Let H be the series of hops produced by our algorithm and H* be **an arbitrary** (not necessarily **the only**) optimal series of hops. Then |H| and |H*| denote the number of hops in H and H*, respectively.

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` always finds an optimal series of hops?

- Let's introduce notation to talk about the algorithm with greater precision …
- Let H be the series of hops produced by our algorithm and H* be **an arbitrary** (not necessarily **the only**) optimal series of hops. Then |H| and |H*| denote the number of hops in H and H*, respectively.
- Note that |H| ≥ |H*|. Why? 🤔

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` always finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision …

Let H be the series of hops produced by our algorithm and H* be **an arbitrary** (not necessarily **the only**) optimal series of hops. Then |H| and |H*| denote the number of hops in H and H*, respectively.

Note that |H| ≥ |H*|. Why? 🤔 Otherwise, H* wouldn't be optimal.

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` always finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision …

Let H be the series of hops produced by our algorithm and H^* be **an arbitrary** (not necessarily **the only**) optimal series of hops. Then $|H|$ and $|H^*|$ denote the number of hops in H and H^*, respectively.

Note that $|H| \geq |H^*|$. Why? 🤔 Otherwise, H^* wouldn't be optimal.

We want to prove that $|H| = |H^*|$. How?

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` always finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision …

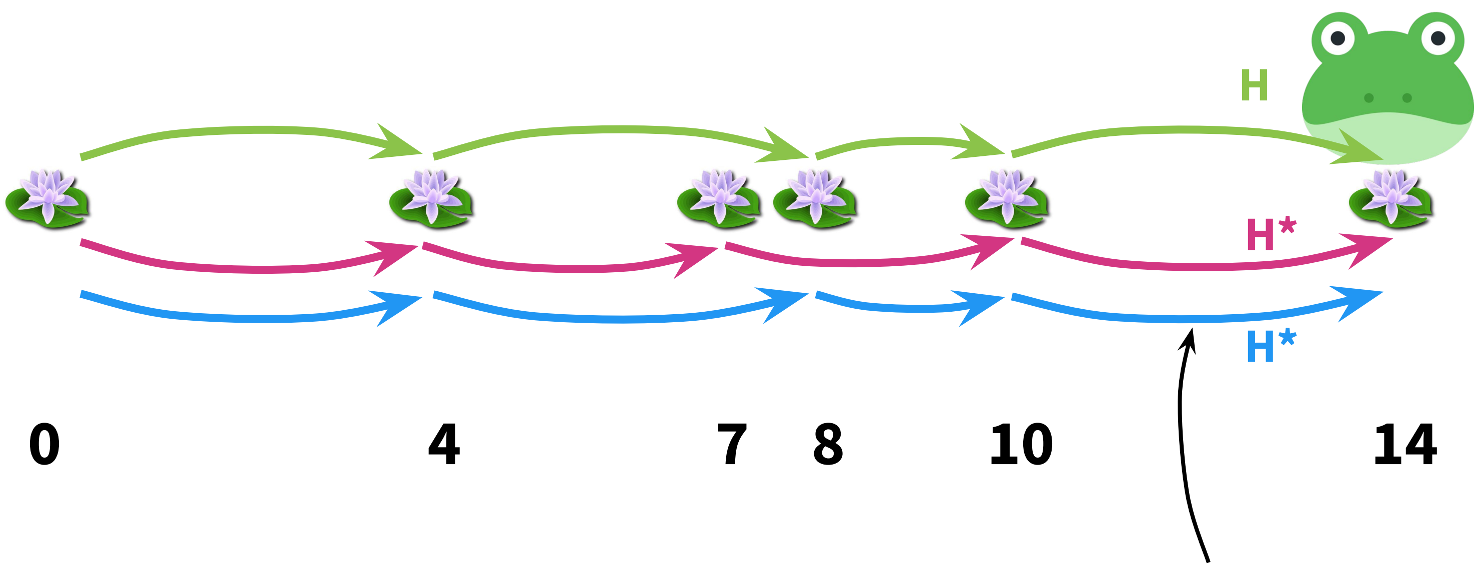
Let H be the series of hops produced by our algorithm and H* be **an arbitrary** (not necessarily **the only**) optimal series of hops. Then |H| and |H*| denote the number of hops in H and H*, respectively.

Note that |H| ≥ |H*|. Why? 🤔 Otherwise, H* wouldn't be optimal.

We want to prove that |H| = |H*|. How?

Intuition: Consider an arbitrary optimal series of hops H*, then show that our greedy algorithm produces a series of hops H no worse than H*.

What Does Arbitrary H* Mean?



There could be many optimal H* (this series of lilypads has 2); this proof relies on an arbitrary choice from among this H*.

Suppose we choose **H***.

Frog Hopping

Let $p(i, H)$ denote the frog's position after taking the first i hops from series H.

Frog Hopping

Let $p(i, H)$ denote the frog's position after taking the first i hops from series H.

Lemma: For any i in $0 \le i \le |H^*|$, we have $p(i, H) \ge p(i, H^*)$, constructing H from `frog_hopping`.

Frog Hopping

Let $p(i, H)$ denote the frog's position after taking the first i hops from series H.

Lemma: For any i in $0 \le i \le |H^*|$, we have $p(i, H) \ge p(i, H^*)$, constructing H from `frog_hopping`.

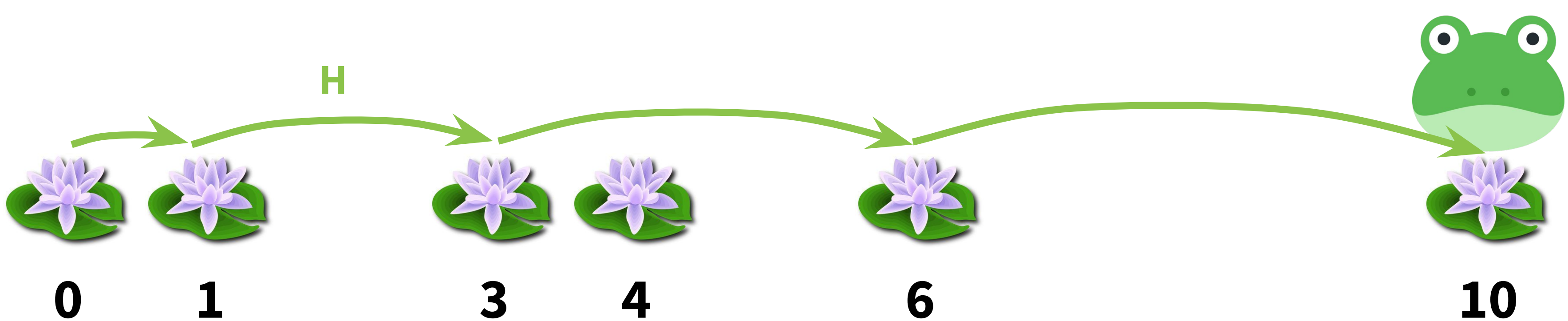
- i.e. After taking i hops according to our greedy algorithm, Freddie will be at least as far forward as if it took i jumps according to an optimal solution.

Frog Hopping

Let p(i, H) denote the frog's position after taking the first i hops from series H.

Lemma: For any i in $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

i.e. After taking i hops according to our greedy algorithm, Freddie will be at least as far forward as if it took i jumps according to an optimal solution.



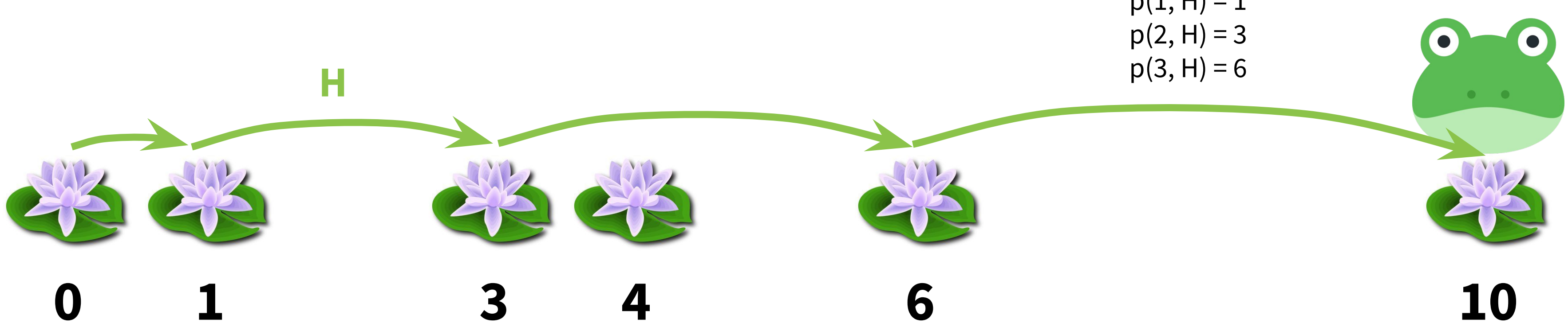
Frog Hopping

Let p(i, H) denote the frog's position after taking the first i hops from series H.

Lemma: For any i in $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

> i.e. After taking i hops according to our greedy algorithm, Freddie will be at least as far forward as if it took i jumps according to an optimal solution.

For this arbitrary **H**:
(unrelated to our alg)
p(0, H) = 0
p(1, H) = 1
p(2, H) = 3
p(3, H) = 6



Frog Hopping

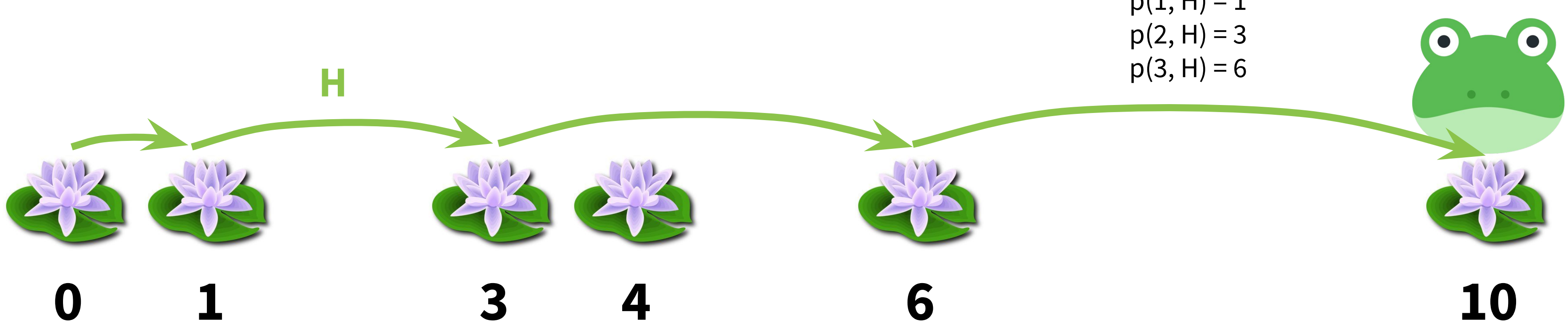
Let p(i, H) denote the frog's position after taking the first i hops from series H.

Lemma: For any i in $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

> i.e. After taking i hops according to our greedy algorithm, Freddie will be at least as far forward as if it took i jumps according to an optimal solution.

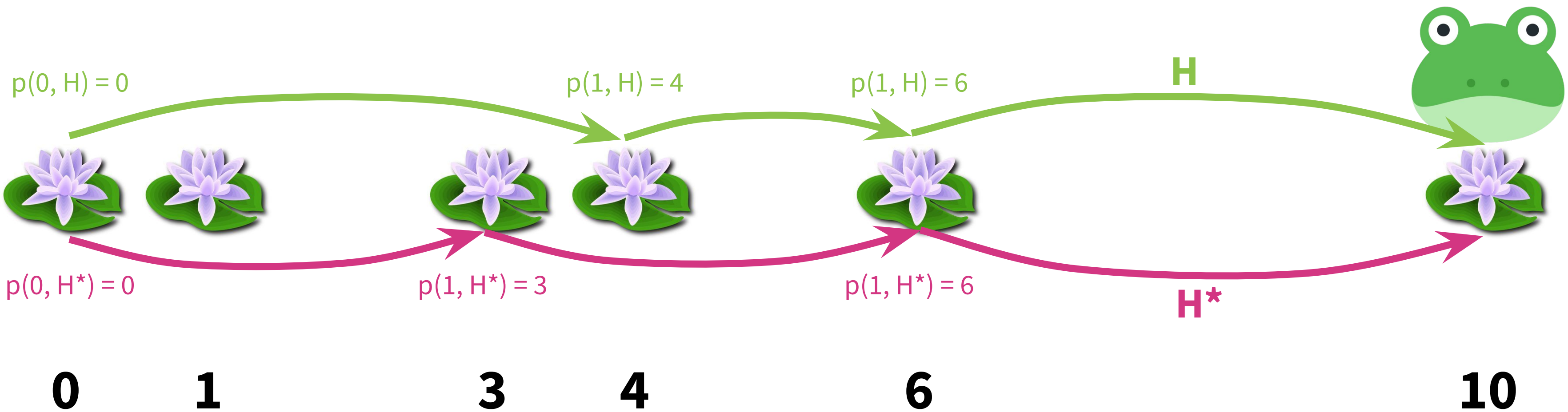
Let's formalize this using induction.

For this arbitrary **H**:
(unrelated to our alg)
p(0, H) = 0
p(1, H) = 1
p(2, H) = 3
p(3, H) = 6



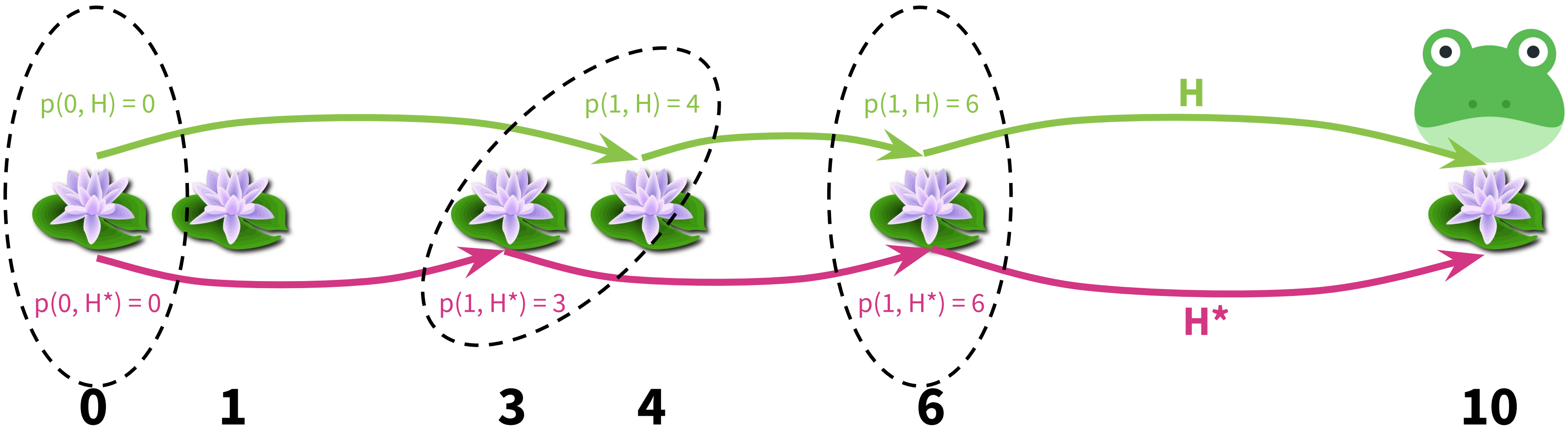
Frog Hopping

Lemma: For any i in $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.



Frog Hopping

Lemma: For any i in $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing **H** from `frog_hopping`.



Frog Hopping

Lemma 2: For all $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, if $i = 0$, then $p(0, H) = 0 \geq 0 = p(0, H^*)$ since the frog hasn't moved.

For the inductive step, assume that the claim holds for some $0 \leq i < |H^*|$. We'll prove the claim holds for $i + 1$ by considering two cases:

Frog Hopping

Lemma 2: For all $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, if $i = 0$, then $p(0, H) = 0 \geq 0 = p(0, H^*)$ since the frog hasn't moved.

For the inductive step, assume that the claim holds for some $0 \leq i < |H^*|$. We'll prove the claim holds for $i + 1$ by considering two cases:

Case 1: $p(i, H) \geq p(i+1, H^*)$. Since each hop moves forward, we have $p(i+1, H) \geq p(i, H)$, so we have $p(i+1, H) \geq p(i+1, H^*)$.

Frog Hopping

Lemma 2: For all $0 \le i \le |H^*|$, we have $p(i, H) \ge p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, if $i = 0$, then $p(0, H) = 0 \ge 0 = p(0, H^*)$ since the frog hasn't moved.

For the inductive step, assume that the claim holds for some $0 \le i < |H^*|$. We'll prove the claim holds for $i + 1$ by considering two cases:

Case 1: $p(i, H) \ge p(i+1, H^*)$. Since each hop moves forward, we have $p(i+1, H) \ge p(i, H)$, so we have $p(i+1, H) \ge p(i+1, H^*)$.

Case 2: $p(i, H) < p(i+1, H^*)$.

Frog Hopping

Lemma 2: For all $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, if $i = 0$, then $p(0, H) = 0 \geq 0 = p(0, H^*)$ since the frog hasn't moved.

For the inductive step, assume that the claim holds for some $0 \leq i < |H^*|$. We'll prove the claim holds for $i + 1$ by considering two cases:

Case 1: $p(i, H) \geq p(i+1, H^*)$. Since each hop moves forward, we have $p(i+1, H) \geq p(i, H)$, so we have $p(i+1, H) \geq p(i+1, H^*)$.

Case 2: $p(i, H) < p(i+1, H^*)$. Each hop is of size at most r, so $p(i+1, H^*) \leq p(i, H^*) + r$.

Frog Hopping

Lemma 2: For all $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, if $i = 0$, then $p(0, H) = 0 \geq 0 = p(0, H^*)$ since the frog hasn't moved.

For the inductive step, assume that the claim holds for some $0 \leq i < |H^*|$. We'll prove the claim holds for $i + 1$ by considering two cases:

Case 1: $p(i, H) \geq p(i+1, H^*)$. Since each hop moves forward, we have $p(i+1, H) \geq p(i, H)$, so we have $p(i+1, H) \geq p(i+1, H^*)$.

Case 2: $p(i, H) < p(i+1, H^*)$. Each hop is of size at most r, so $p(i+1, H^*) \leq p(i, H^*) + r$. By our inductive hypothesis, we know $p(i, H) \geq p(i, H^*)$, so $p(i+1, H^*) \leq p(i, H) + r$; i.e. position $p(i+1, H^*)$ is reachable from position $p(i, H)$.

Frog Hopping

Lemma 2: For all $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, if $i = 0$, then $p(0, H) = 0 \geq 0 = p(0, H^*)$ since the frog hasn't moved.

For the inductive step, assume that the claim holds for some $0 \leq i < |H^*|$. We'll prove the claim holds for $i + 1$ by considering two cases:

Case 1: $p(i, H) \geq p(i+1, H^*)$. Since each hop moves forward, we have $p(i+1, H) \geq p(i, H)$, so we have $p(i+1, H) \geq p(i+1, H^*)$.

Case 2: $p(i, H) < p(i+1, H^*)$. Each hop is of size at most r, so $p(i+1, H^*) \leq p(i, H^*) + r$. By our inductive hypothesis, we know $p(i, H) \geq p(i, H^*)$, so $p(i+1, H^*) \leq p(i, H) + r$; i.e. position $p(i+1, H^*)$ is reachable from position $p(i, H)$. Since the greedy algorithm hops to the furthest lilypad still reachable from position $p(i, H)$, it hops to at least position $p(i+1, H^*)$. Therefore, $p(i+1, H) \geq p(i+1, H^*)$.

Frog Hopping

Lemma 2: For all $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, if $i = 0$, then $p(0, H) = 0 \geq 0 = p(0, H^*)$ since the frog hasn't moved.

For the inductive step, assume that the claim holds for some $0 \leq i < |H^*|$. We'll prove the claim holds for $i + 1$ by considering two cases:

Case 1: $p(i, H) \geq p(i+1, H^*)$. Since each hop moves forward, we have $p(i+1, H) \geq p(i, H)$, so we have $p(i+1, H) \geq p(i+1, H^*)$.

Case 2: $p(i, H) < p(i+1, H^*)$. Each hop is of size at most r, so $p(i+1, H^*) \leq p(i, H^*) + r$. By our inductive hypothesis, we know $p(i, H) \geq p(i, H^*)$, so $p(i+1, H^*) \leq p(i, H) + r$; i.e. position $p(i+1, H^*)$ is reachable from position $p(i, H)$. Since the greedy algorithm hops to the furthest lilypad still reachable from position $p(i, H)$, it hops to at least position $p(i+1, H^*)$. Therefore, $p(i+1, H) \geq p(i+1, H^*)$.

So $p(i+1, H) \geq p(i+1, H^*)$, completing the induction. ■

Frog Hopping

Now for the theorem: `frog_hopping` produces an optimal solution for Freddie.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \le |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \ge p(k, H^*)$.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H^*, we know that $p(k, H) \geq p(k, H^*) = n$.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H^*, we know that $p(k, H) \geq p(k, H^*) = n$. Because the greedy algorithm never hops past position n, we know $p(k, H) \leq n$.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H*, we know that $p(k, H) \geq p(k, H^*) = n$. Because the greedy algorithm never hops past position n, we know $p(k, H) \leq n$. Since $n \leq p(k, H) \leq n$, then $p(k, H) = n$.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H^*, we know that $p(k, H) \geq p(k, H^*) = n$. Because the greedy algorithm never hops past position n, we know $p(k, H) \leq n$. Since $n \leq p(k, H) \leq n$, then $p(k, H) = n$.

The greedy algorithm arrives at position n after k hops, so $|H| = k$.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H^*, we know that $p(k, H) \geq p(k, H^*) = n$. Because the greedy algorithm never hops past position n, we know $p(k, H) \leq n$. Since $n \leq p(k, H) \leq n$, then $p(k, H) = n$.

The greedy algorithm arrives at position n after k hops, so $|H| = k$. Importantly, it's impossible to reach position n in fewer than k hops since doing so would contradict the optimality of H^*.

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H^*, we know that $p(k, H) \geq p(k, H^*) = n$. Because the greedy algorithm never hops past position n, we know $p(k, H) \leq n$. Since $n \leq p(k, H) \leq n$, then $p(k, H) = n$.

The greedy algorithm arrives at position n after k hops, so $|H| = k$. Importantly, it's impossible to reach position n in fewer than k hops since doing so would contradict the optimality of H^*. Thus, $|H| = k = |H^*|$, so `frog_hopping` produces an optimal solution. ■

Frog Hopping

Theorem: `frog_hopping` produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.

Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H^*, we know that $p(k, H) \geq p(k, H^*) = n$. Because the greedy algorithm never hops past position n, we know $p(k, H) \leq n$. Since $n \leq p(k, H) \leq n$, then $p(k, H) = n$.

The greedy algorithm arrives at position n after k hops, so $|H| = k$. Importantly, it's impossible to reach position n in fewer than k hops since doing so would contradict the optimality of H^*. Thus, $|H| = k = |H^*|$, so `frog_hopping` produces an optimal solution. ■

Here, we proved this step using a direct proof. You should be able to structure the proof by contradiction here too.

Frog Hopping

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules). 👌

(2) **Optimality.** The algorithm finds an optimal series of hops (i.e. there isn't a better path available).

Frog Hopping

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules). 👌

(2) **Optimality.** The algorithm finds an optimal series of hops (i.e. there isn't a better path available). 👌

Greedy Stays Ahead

The style of proof we just wrote is an example of a **greedy stays ahead** proof.

(1) Find intermediate values that evaluate the solution produced by any algorithm, including the greedy one.

What's our values in `frog_hopping`? 🤔 The position after i hops.

(2) Show the greedy algorithm produces values at least as good as any solution's (using induction).

(3) Prove that since the greedy algorithm produces values at least as good as any solution's, it must be optimal (using direct proof or proof by contradiction).

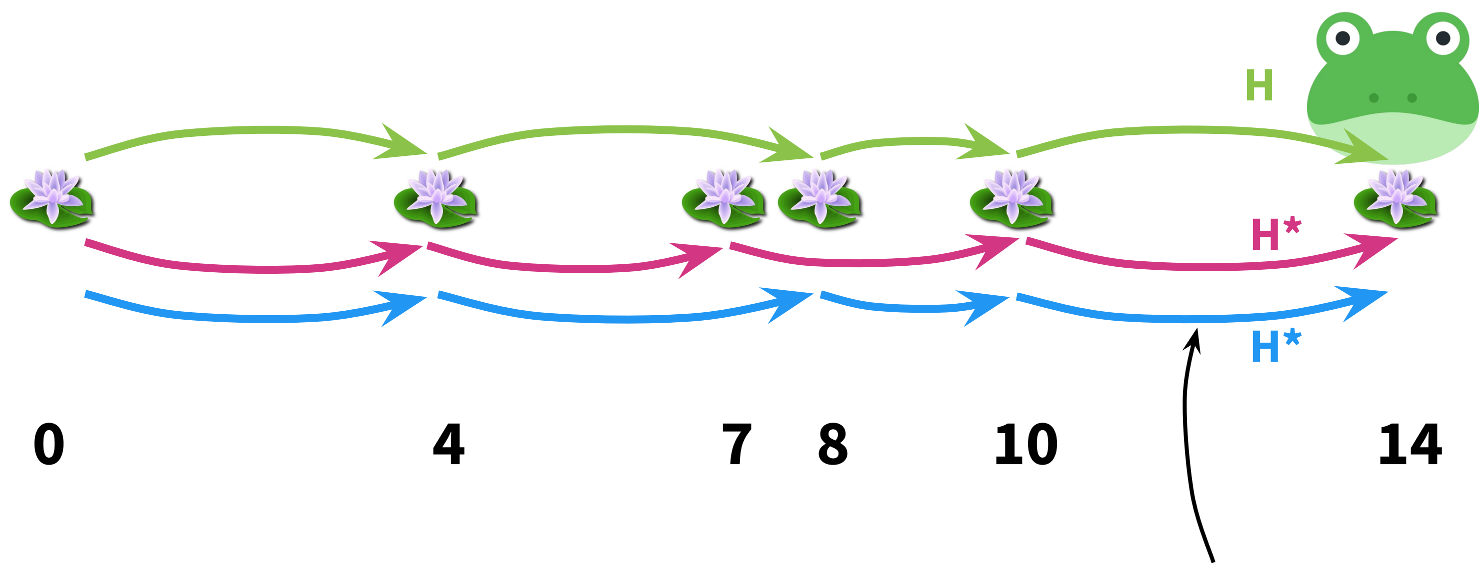
Greedy Exchange Argument

There's another style of proof that uses **greedy exchange argument**.

- If we swap an optimal solution out for the greedy solution, argue that we're still optimal.

Greedy Exchange Argument

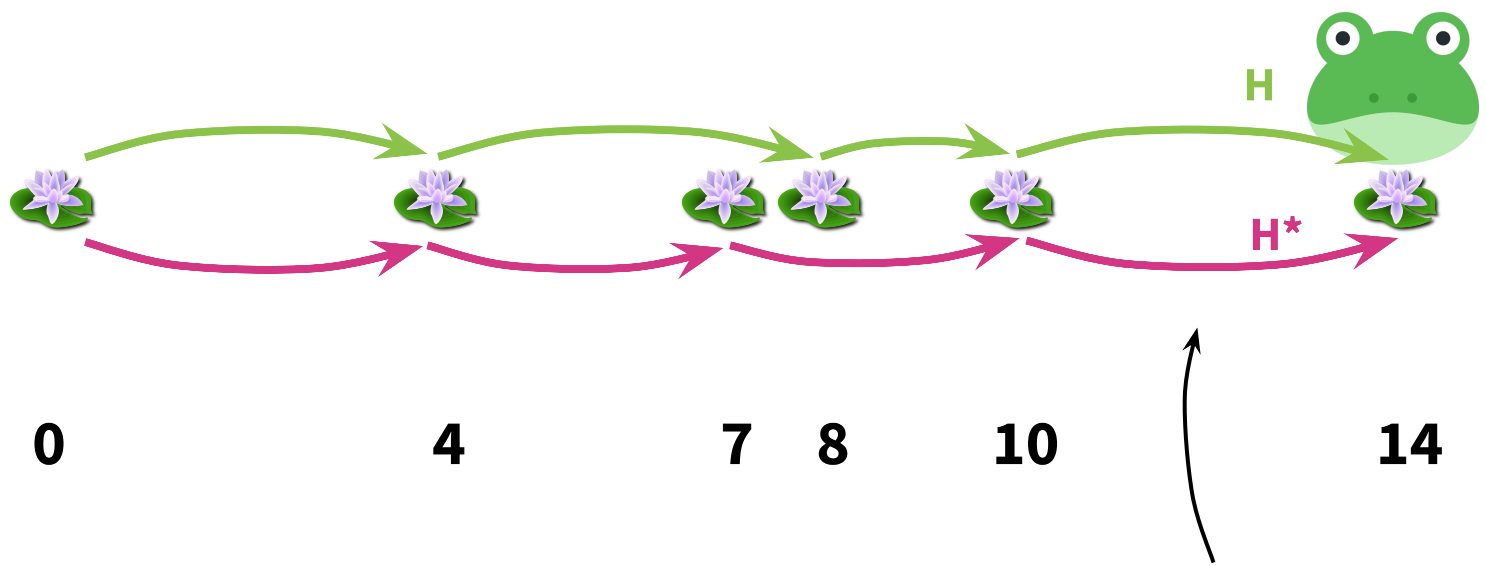
Again, this proof will rely on an arbitrary choice of H*.



There could be many optimal H* (this series of lilypads has 2); this proof relies on an arbitrary choice from among this H*.

Suppose we choose **H***.

Greedy Exchange Argument



There could be many optimal H* (this series of lilypads has 2); this proof relies on an arbitrary choice from among this H*.

Suppose we choose **H***.

Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j].

Greedy Exchange Argument

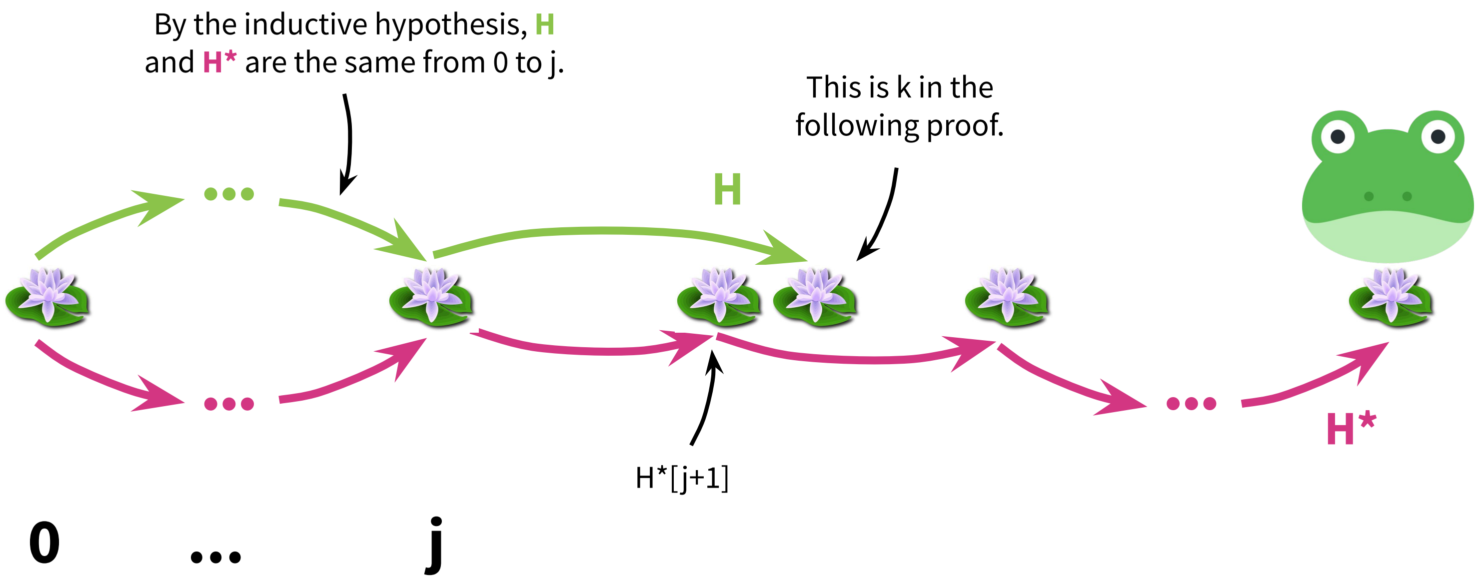
Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

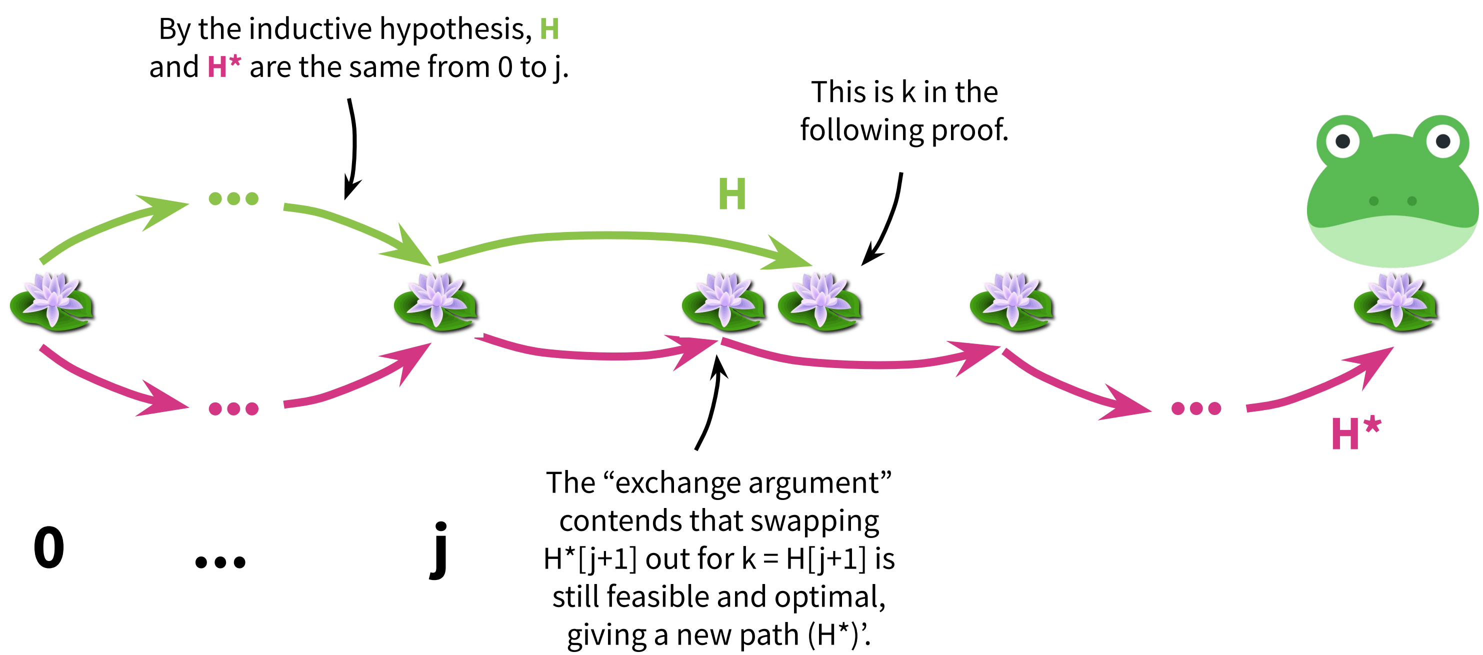
As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

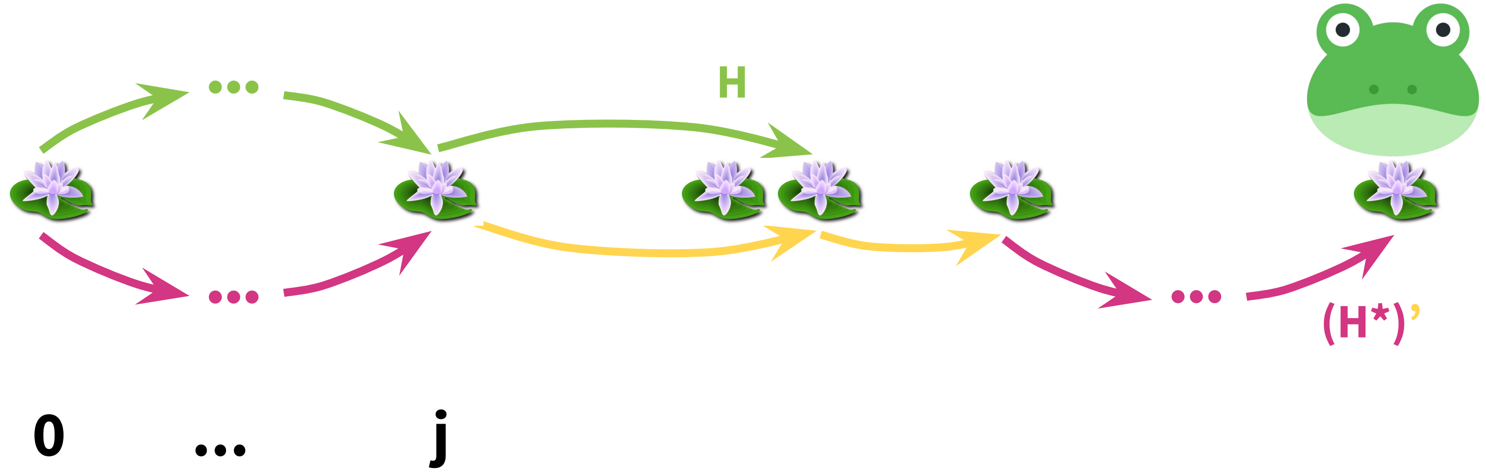
Greedy Exchange Argument



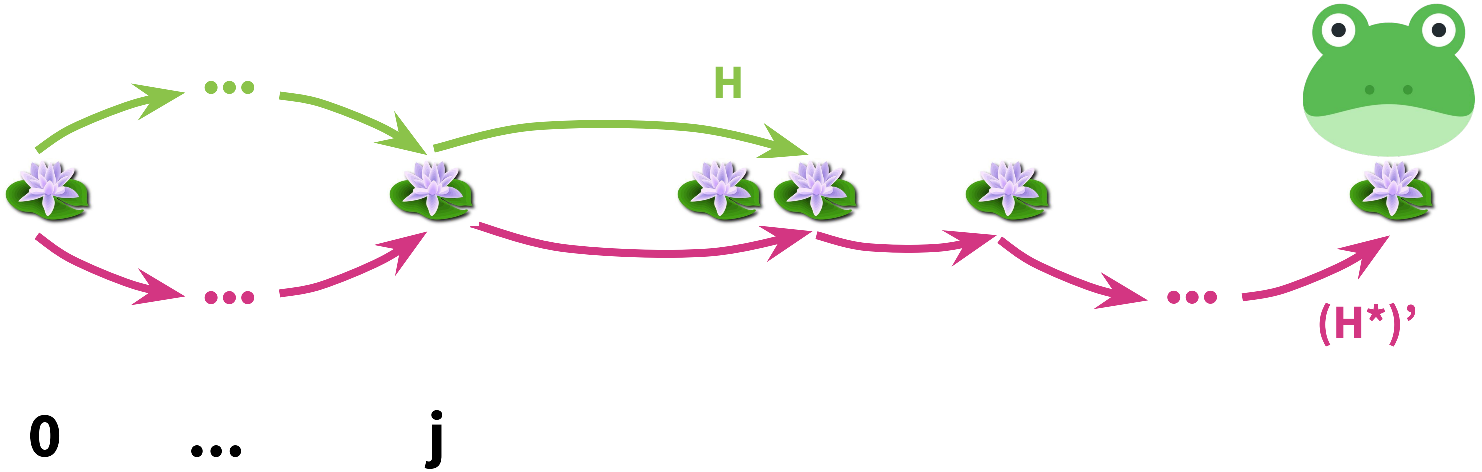
Greedy Exchange Argument



Greedy Exchange Argument



Greedy Exchange Argument



Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

Let H* be an optimal series of hops such that H*[0..j] = H[0..j]. Suppose we add k as H[j+1].

Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

Let H* be an optimal series of hops such that H*[0..j] = H[0..j]. Suppose we add k as H[j+1]. Then k ≥ H*[j+1] since H*[j+1] ≤ r + H*[j] = r + H[j] and, by construction, k is the furthest lilypad such that k ≤ r + H[j].

Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

Let H* be an optimal series of hops such that H*[0..j] = H[0..j]. Suppose we add k as H[j+1]. Then k ≥ H*[j+1] since H*[j+1] ≤ r + H*[j] = r + H[j] and, by construction, k is the furthest lilypad such that k ≤ r + H[j].

Consider (H*)' obtained from H* and setting H*[j+1] = k.

Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

Let H* be an optimal series of hops such that H*[0..j] = H[0..j]. Suppose we add k as H[j+1]. Then k ≥ H*[j+1] since H*[j+1] ≤ r + H*[j] = r + H[j] and, by construction, k is the furthest lilypad such that k ≤ r + H[j].

Consider (H*)' obtained from H* and setting H*[j+1] = k.

This is still feasible since (H*)'[j+1] = k ≤ r + (H*)'[j] and (H*)'[j+2] = H*[j+2] ≤ r + H*[j+1] ≤ r + k = r + (H*)'[j].

Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

Let H* be an optimal series of hops such that H*[0..j] = H[0..j]. Suppose we add k as H[j+1]. Then k ≥ H*[j+1] since H*[j+1] ≤ r + H*[j] = r + H[j] and, by construction, k is the furthest lilypad such that k ≤ r + H[j].

Consider (H*)' obtained from H* and setting H*[j+1] = k.

This is still feasible since (H*)'[j+1] = k ≤ r + (H*)'[j] and (H*)'[j+2] = H*[j+2] ≤ r + H*[j+1] ≤ r + k = r + (H*)'[j].

Since H* and (H*)' are the same except at position j+1.

Greedy Exchange Argument

Theorem: `frog_hopping` produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to `[0]` and all feasible hops H* must have H*[0] = 0.

For the inductive step, assume that after hop j has been added to H, there exists an optimal feasible series of hops H* such that H*[0..j] = H[0..j]. We'll prove that after hop j+1 has been added to H, there still exists an optimal series of hops H_{new}* such that H_{new}*[0..j+1] = H[0..j+1].

Let H* be an optimal series of hops such that H*[0..j] = H[0..j]. Suppose we add k as H[j+1]. Then k ≥ H*[j+1] since H*[j+1] ≤ r + H*[j] = r + H[j] and, by construction, k is the furthest lilypad such that k ≤ r + H[j].

Consider (H*)' obtained from H* and setting H*[j+1] = k.

This is still feasible since (H*)'[j+1] = k ≤ r + (H*)'[j] and (H*)'[j+2] = H*[j+2] ≤ r + H*[j+1] ≤ r + k = r + (H*)'[j].

Since H* and (H*)' are the same except at position j+1.

This is still optimal since (H*)' has the same number of hops as H*.

3 min break

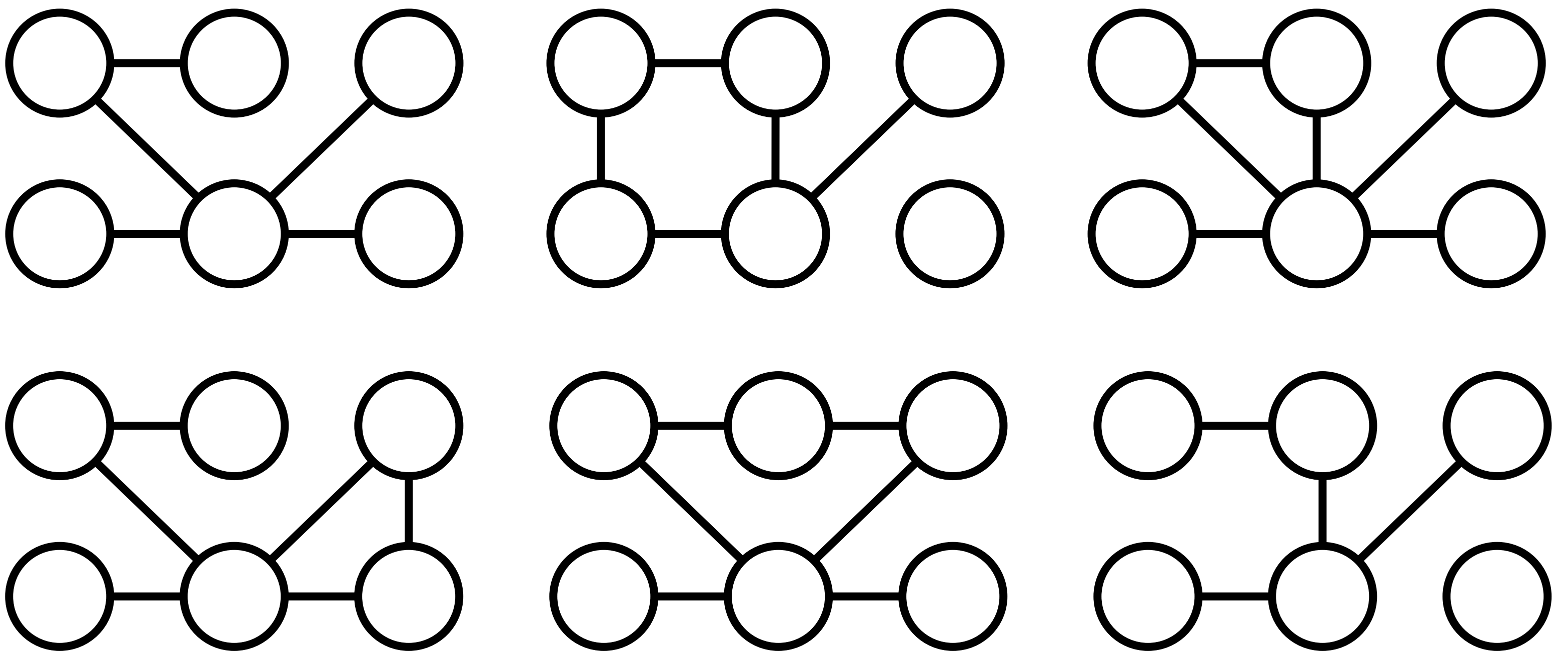
Minimum Spanning Trees

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? 🤔

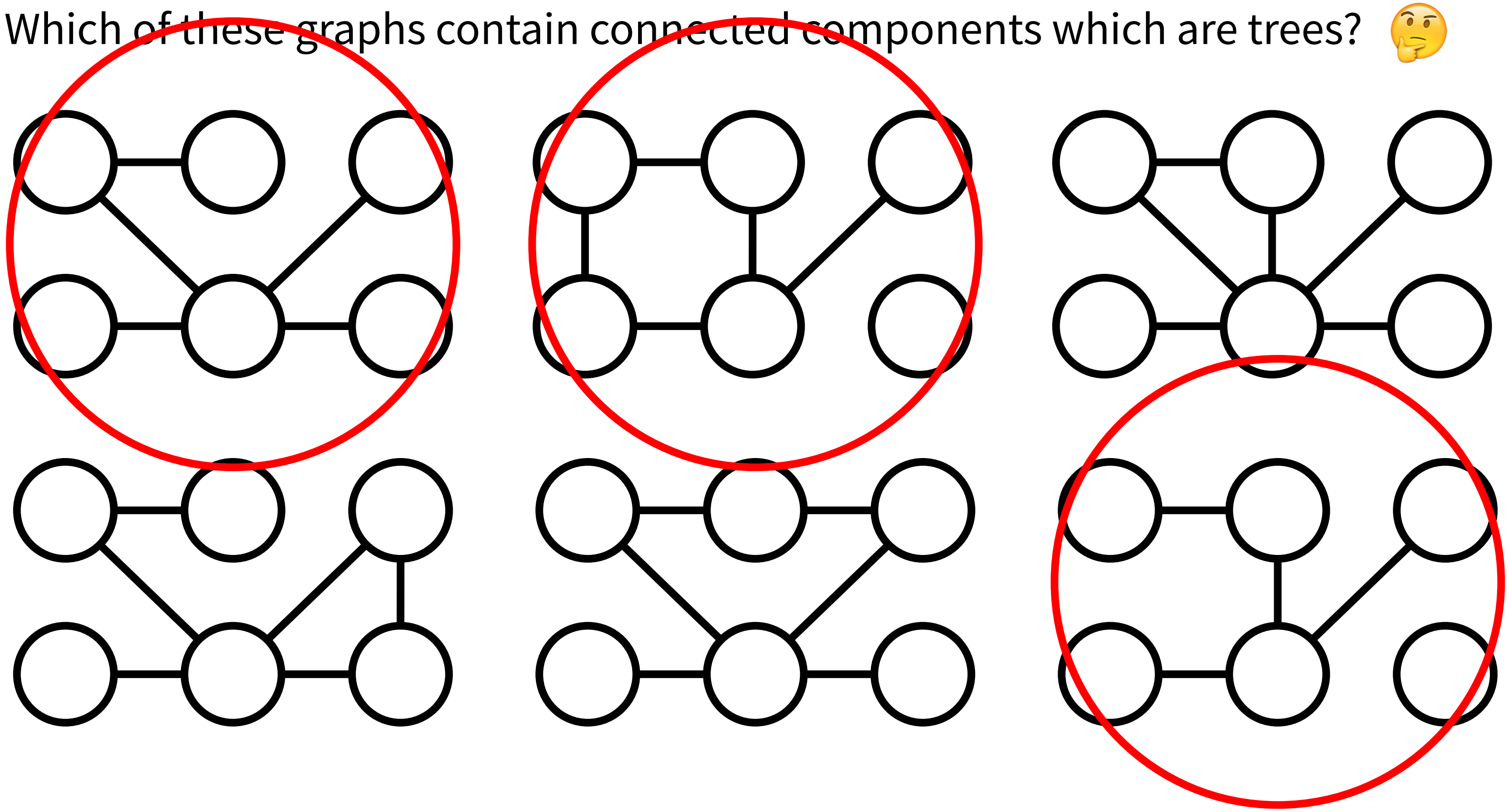


MSTs

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.

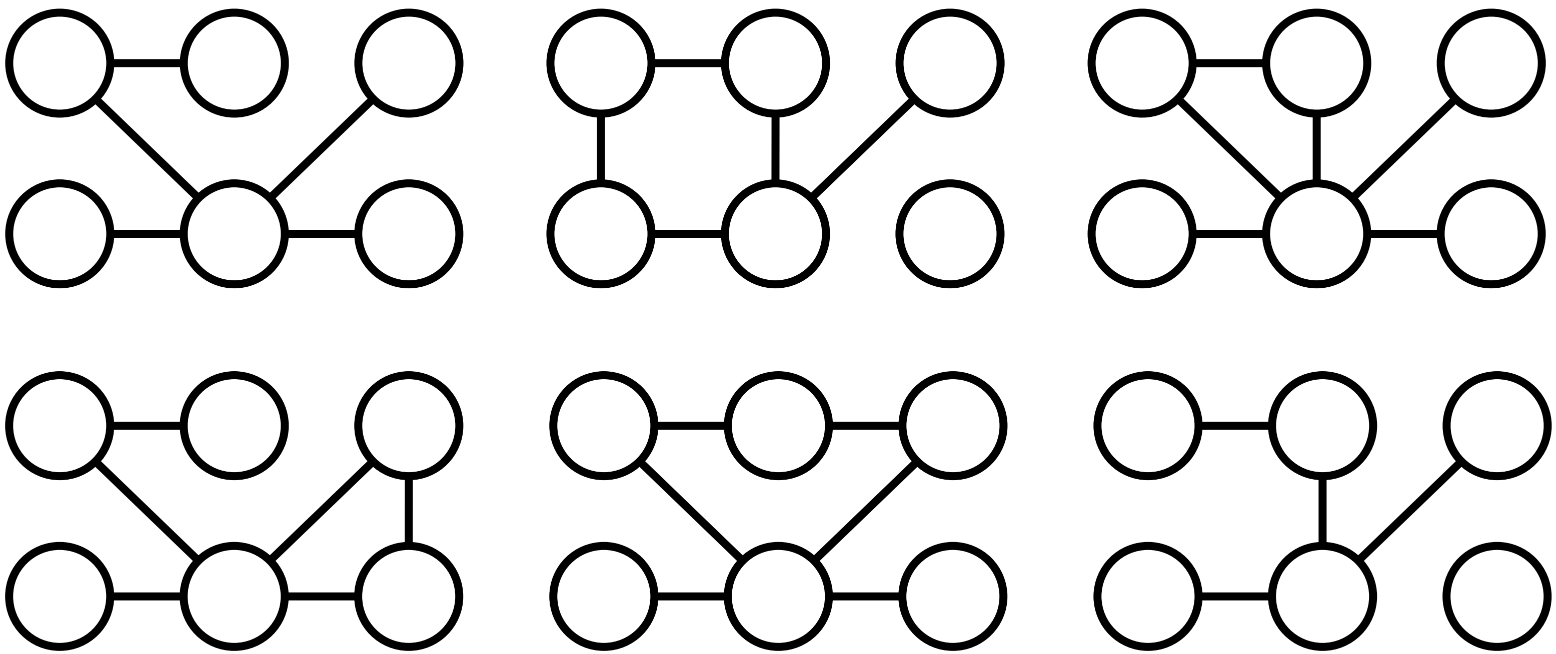
Which of these graphs contain connected components which are trees? 🤔



MSTs

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

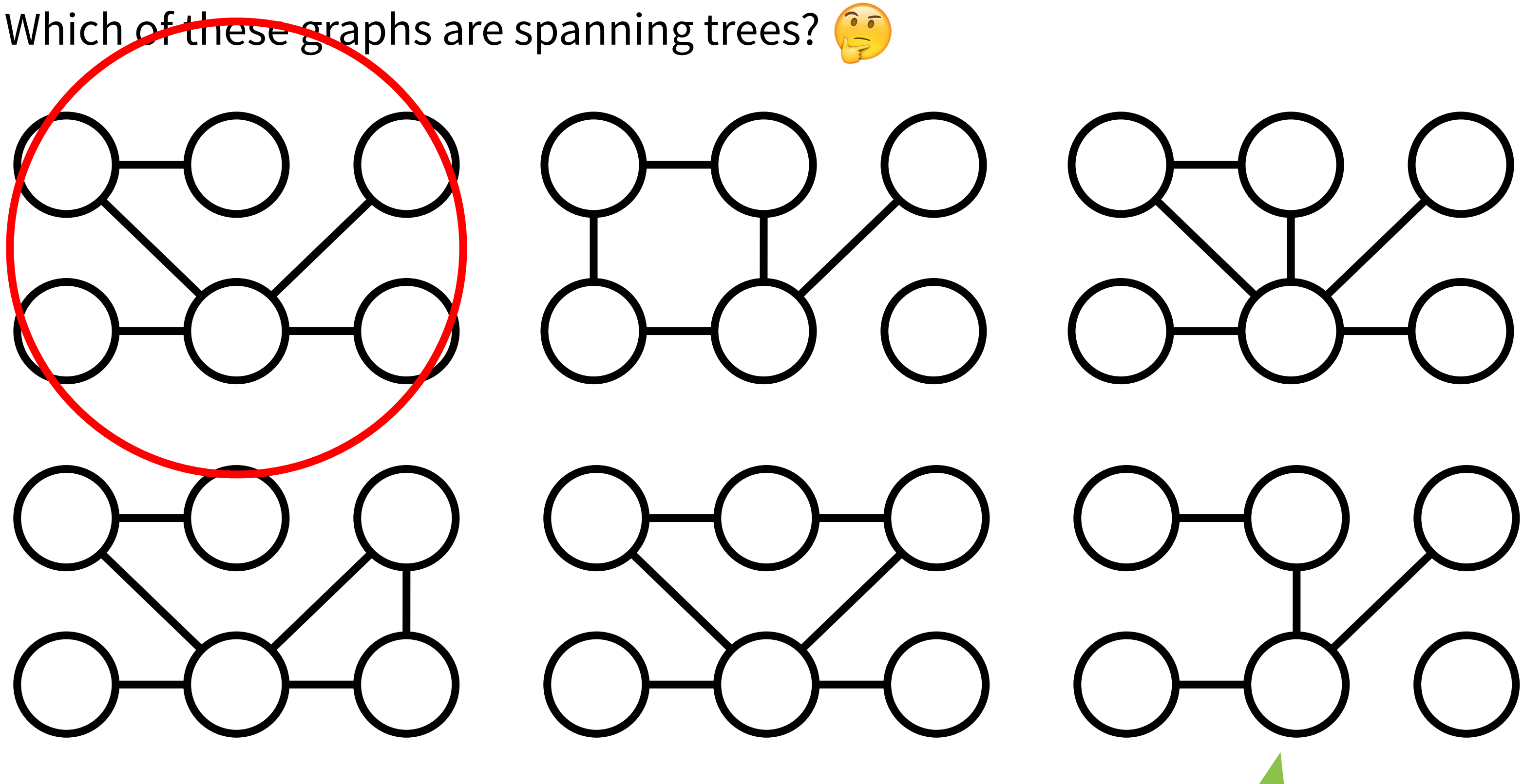
Which of these graphs are spanning trees? 🤔



MSTs

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

Which of these graphs are spanning trees? 🤔

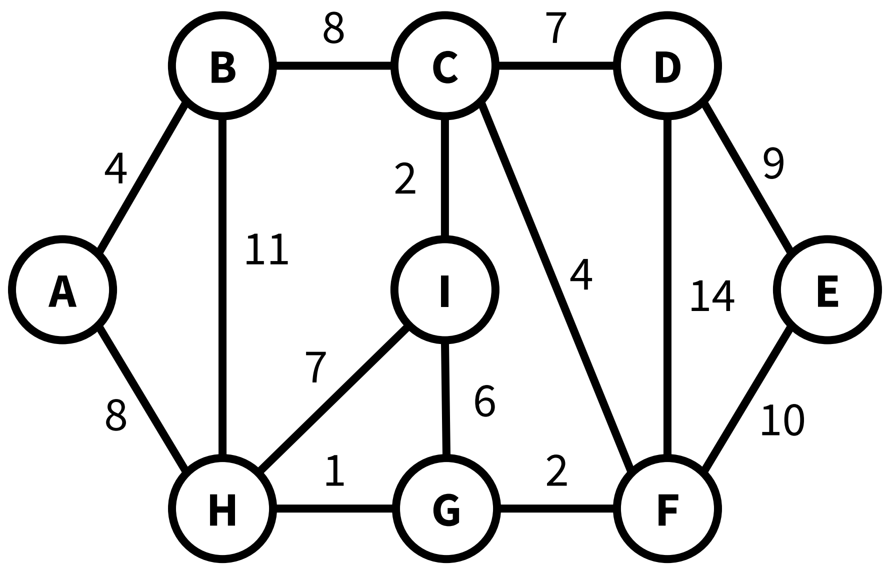


This connected component of the graph is a tree, but it doesn't include all of the vertices.

MSTs

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

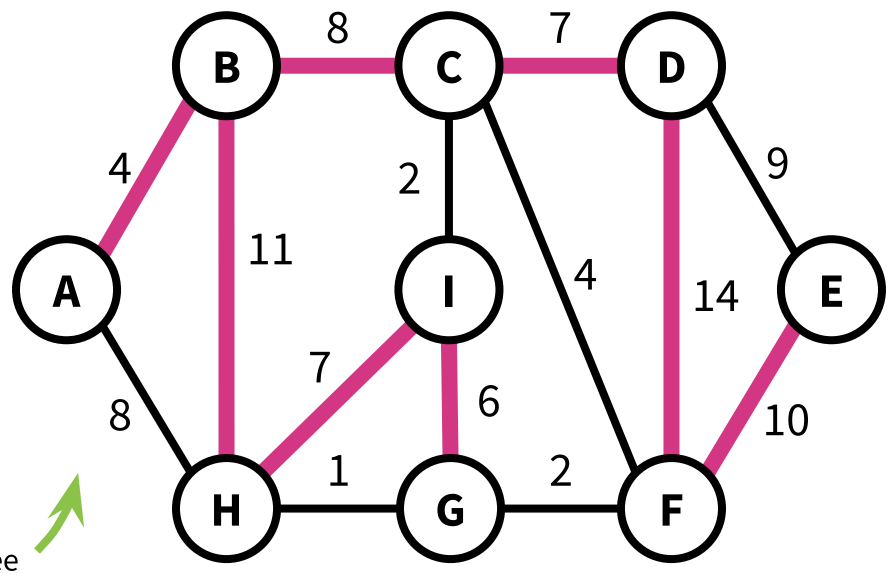
The cost of a spanning tree is the sum of the weights on the edges.



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.

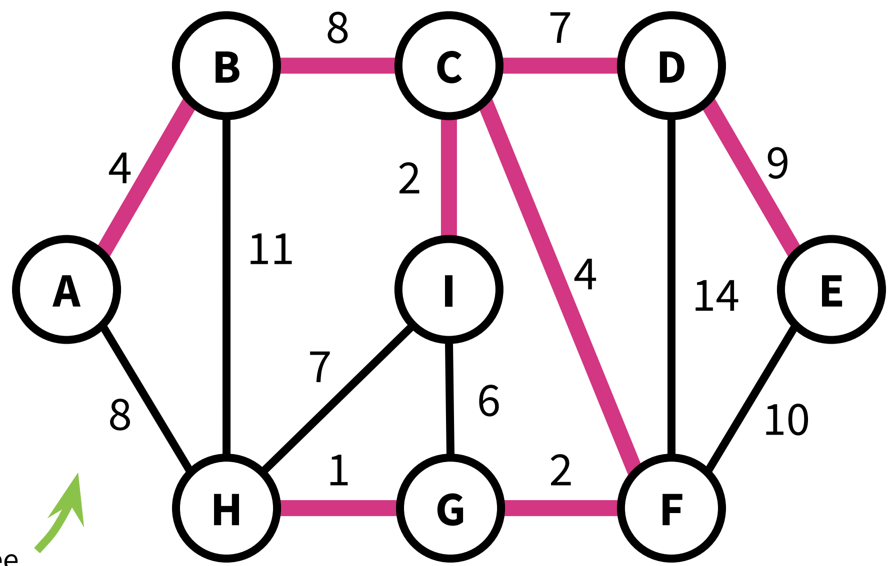


This spanning tree has a cost of 67.

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.

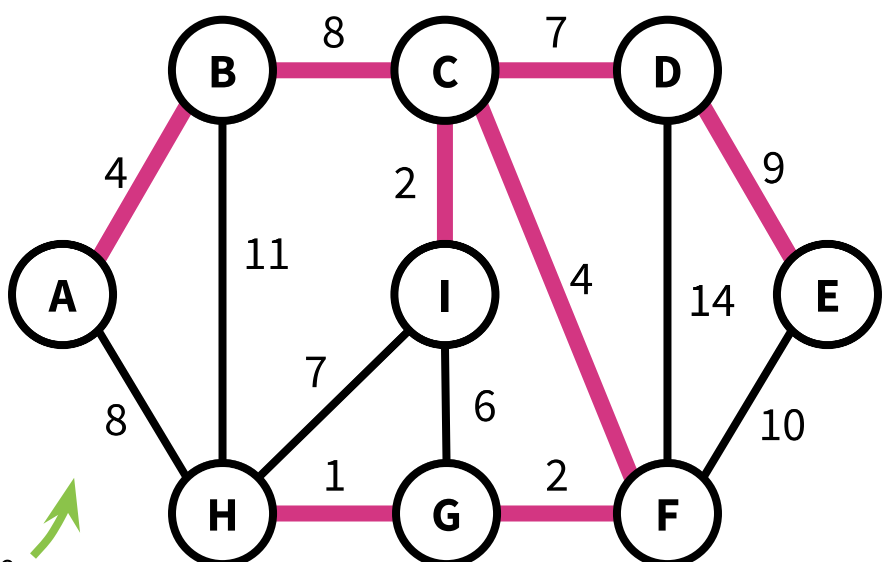
MSTs

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

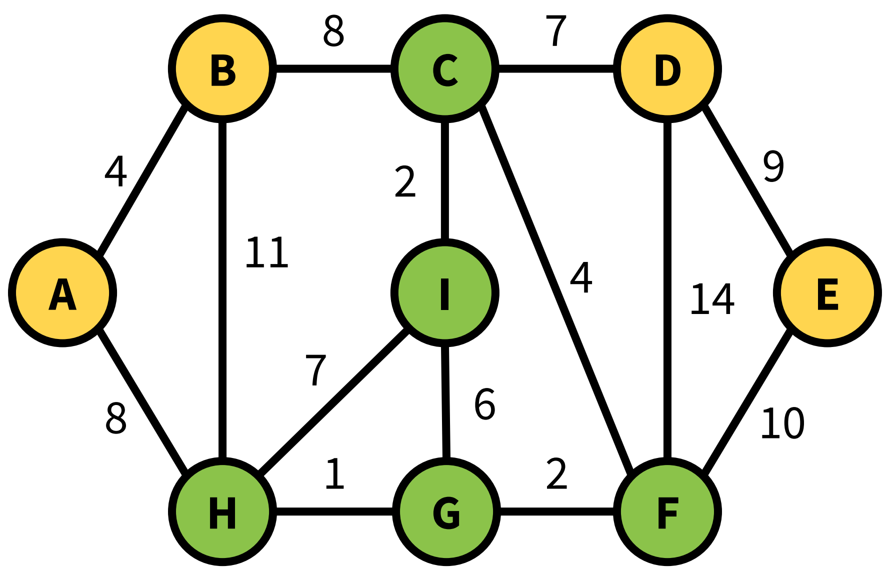
How might we find an MST?

- Today, we'll see two greedy algorithms that find an MST.

MSTs

A **cut** is a partition of the vertices into two nonempty parts.

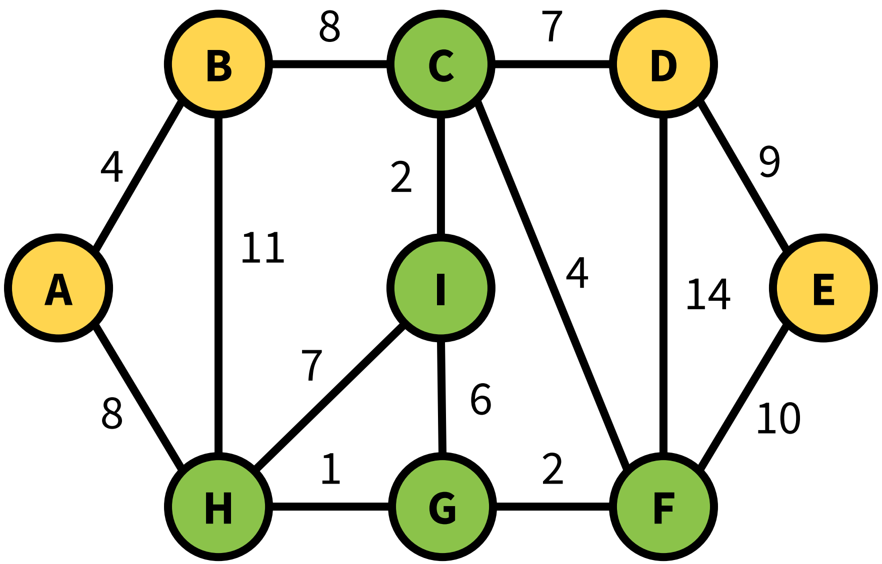
e.g. This is the cut "{A, B, D, E} and {C, I, F, G, H}".



MSTs

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}".



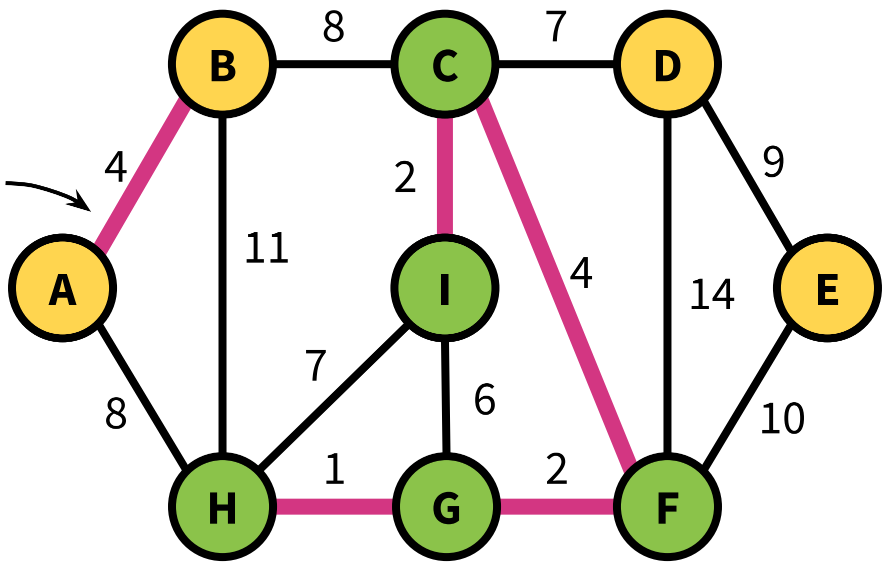
A cut respects a set of edges if no edges in the set cross the cut.

MSTs

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.



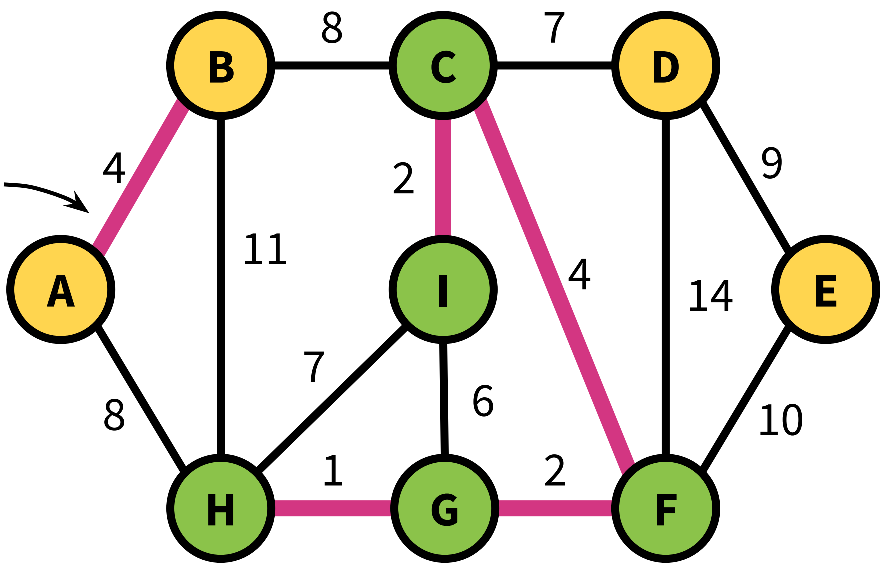
A cut respects a set of edges if no edges in the set cross the cut.

MSTs

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.



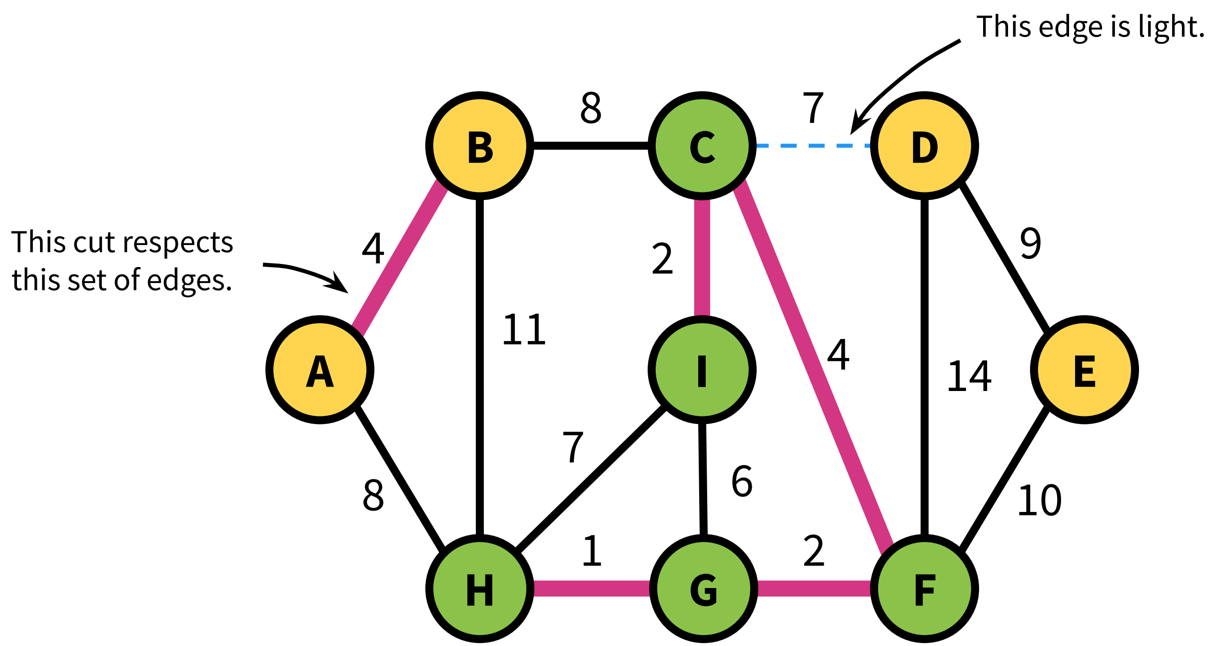
A cut respects a set of edges if no edges in the set cross the cut.

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

MSTs

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

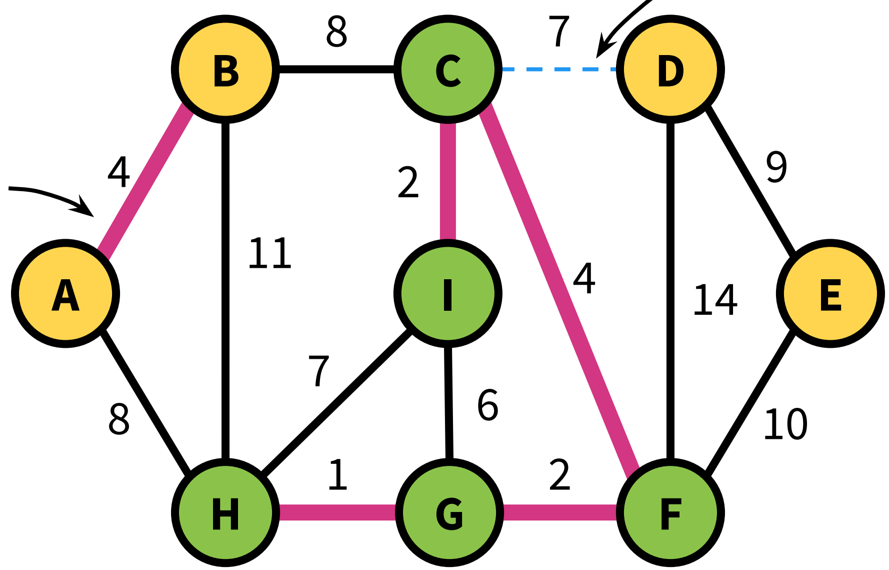
Consider a cut that respects a set of edges **A**.

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

Let (u, v) be a light edge.

This edge is light.

This cut respects this set of edges.



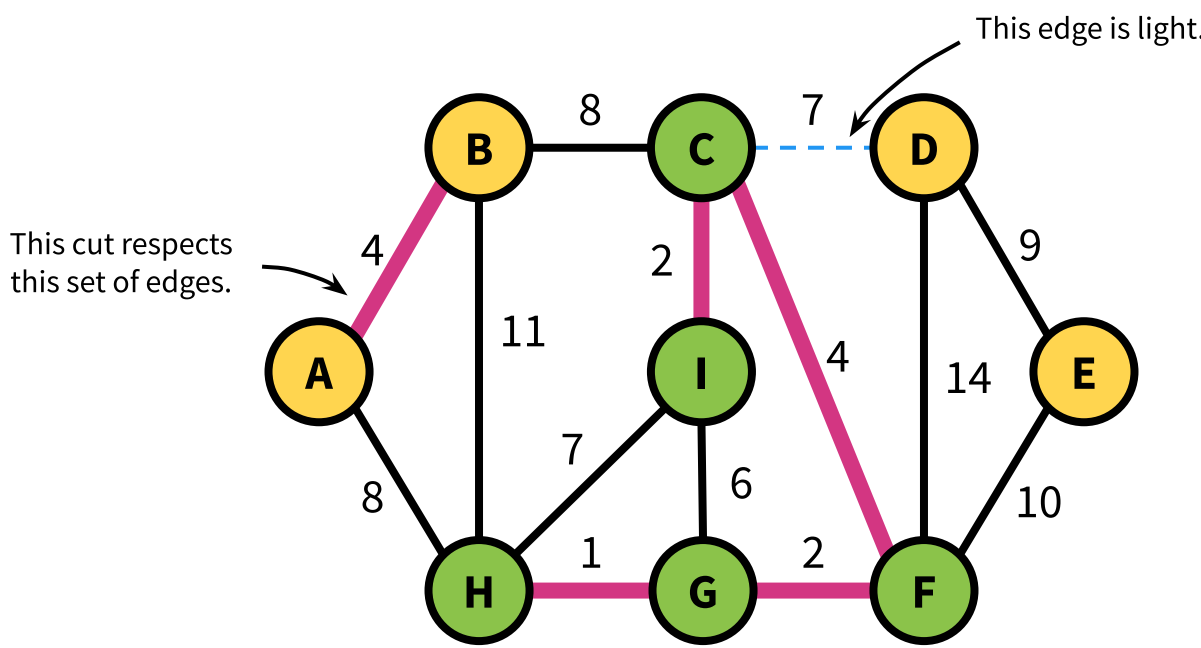
Lemma

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**.

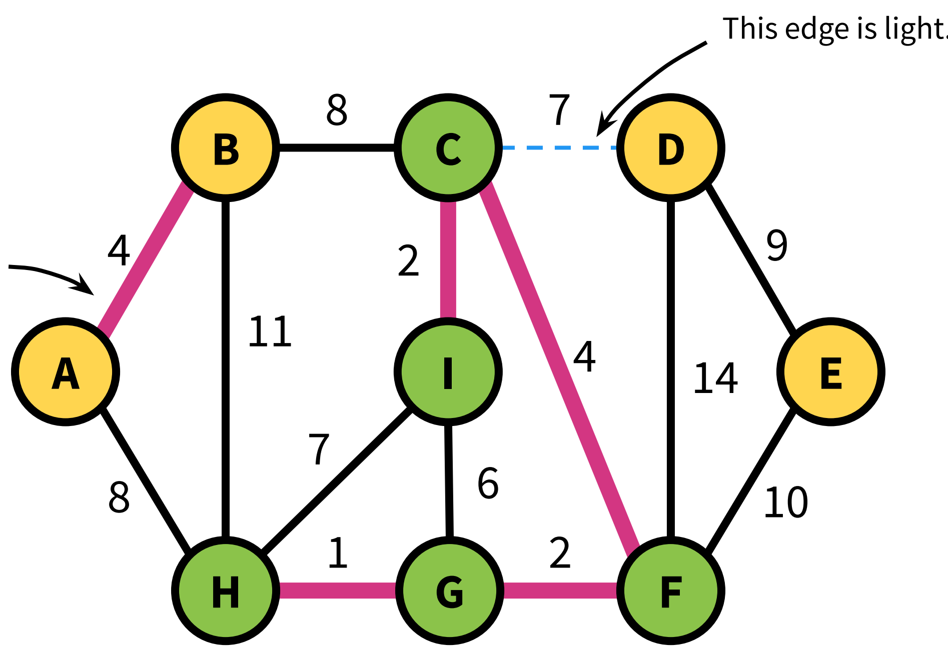
Suppose there exists an MST containing **A**.

Let (u, v) be a light edge.

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

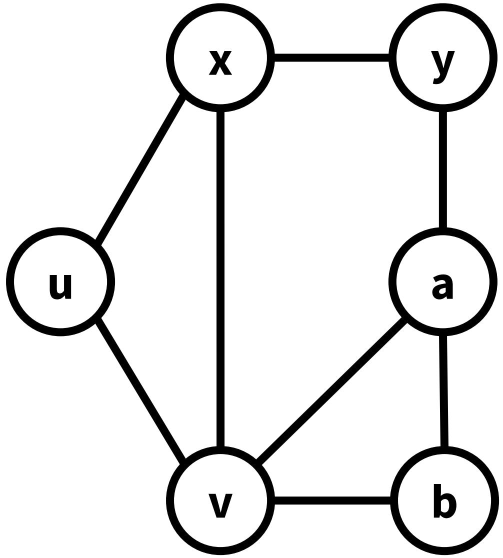
This is precisely the sort of statement we need for a greedy algorithm: If we haven't ruled out the possibility of success so far, then adding a light edge won't rule it out.

This cut respects this set of edges.



Proof of Lemma

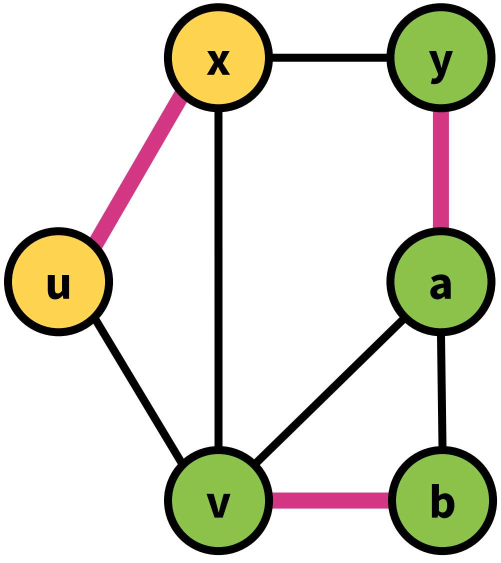
Consider a graph with …



Proof of Lemma

Consider a graph with …

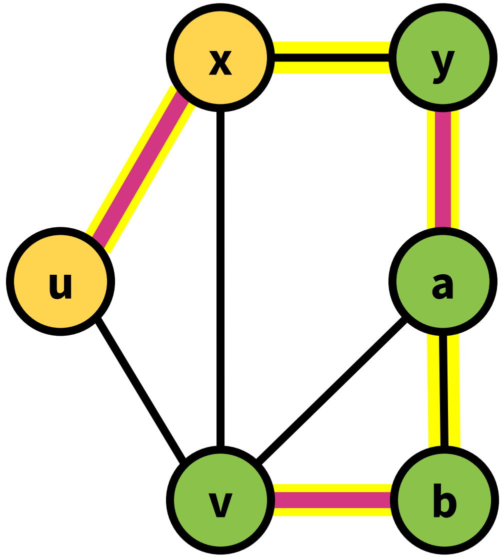
- A cut that respects a set of edges **A**



Proof of Lemma

Consider a graph with …

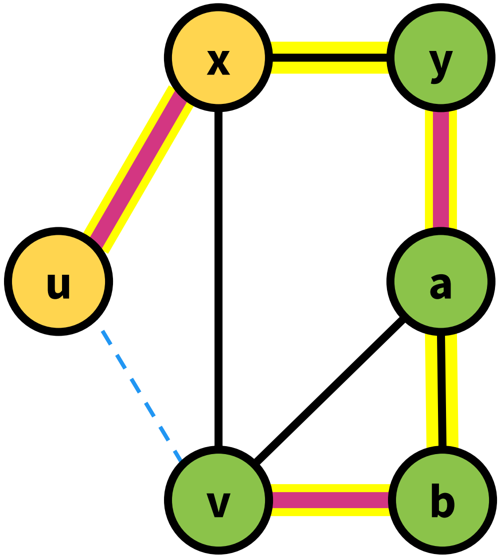
A cut that respects a set of edges **A**, such that there's an MST **T** containing **A**,



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**.

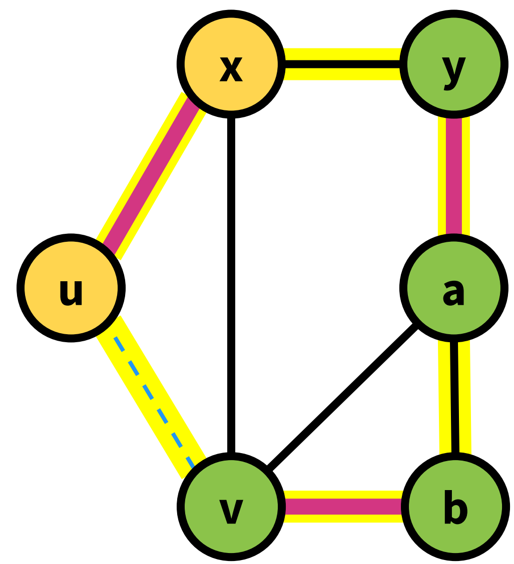


Proof of Lemma

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.



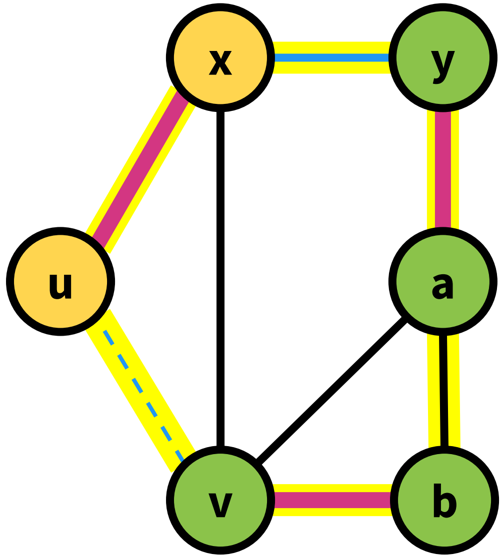
Proof of Lemma

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

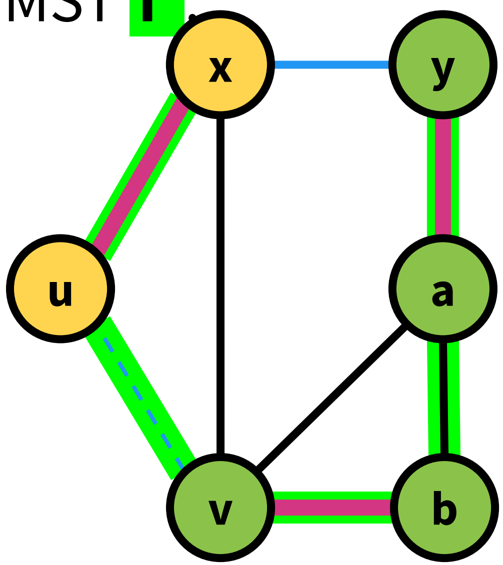
Consider a graph with …

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.

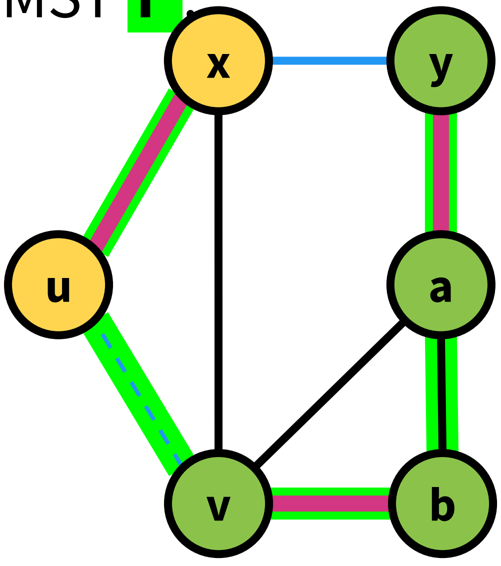
There must be another edge in this cycle crossing this cut.
Let's call this edge (x, y).

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).

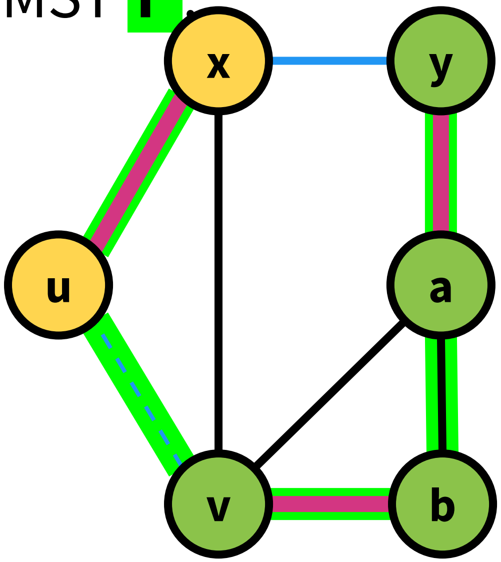
Exchange (u, v) for (x, y) in **T**; call the resulting MST **T'**.

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

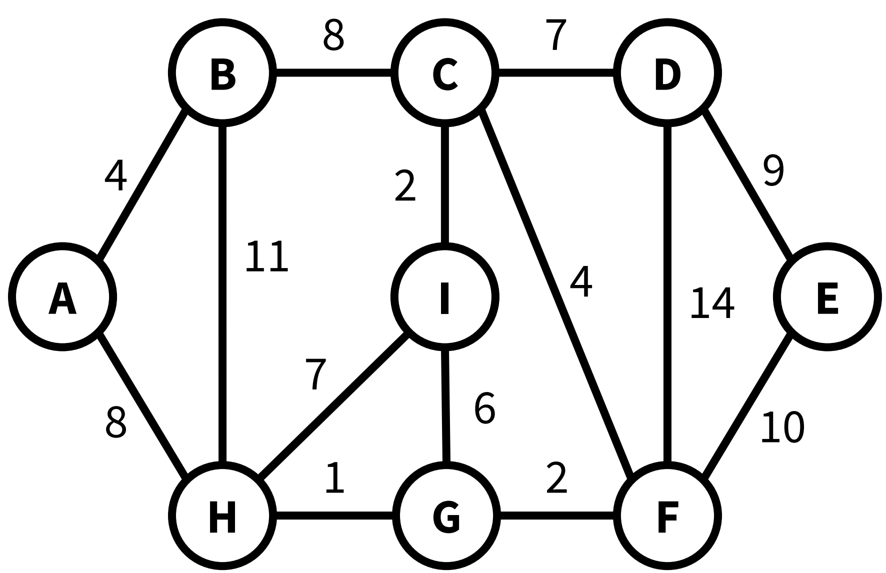
Any Ideas?

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**.

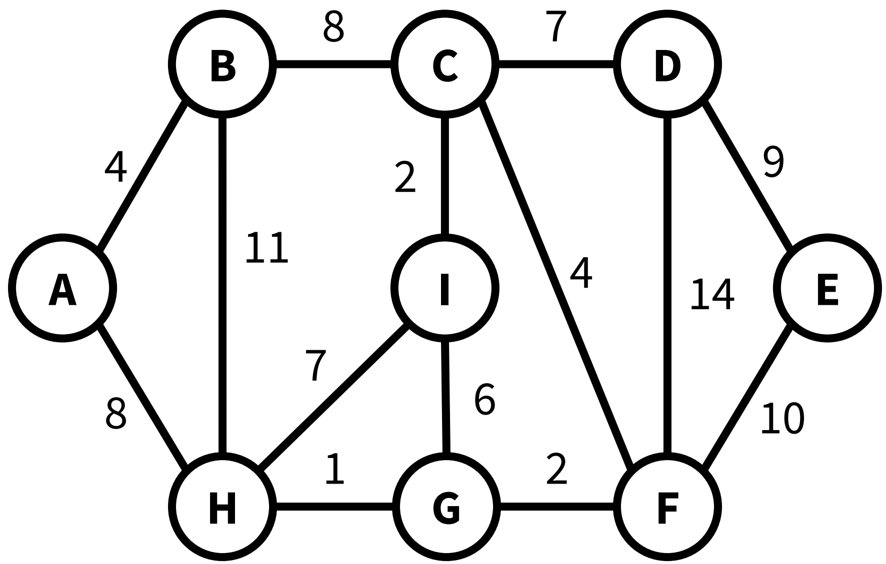
Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Any ideas about what to greedily choose?



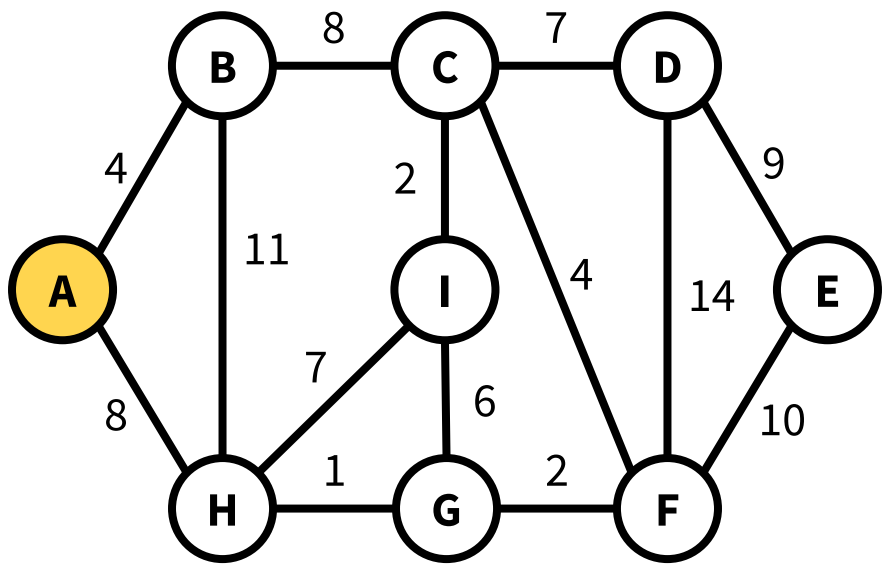
Prim's Algorithm

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



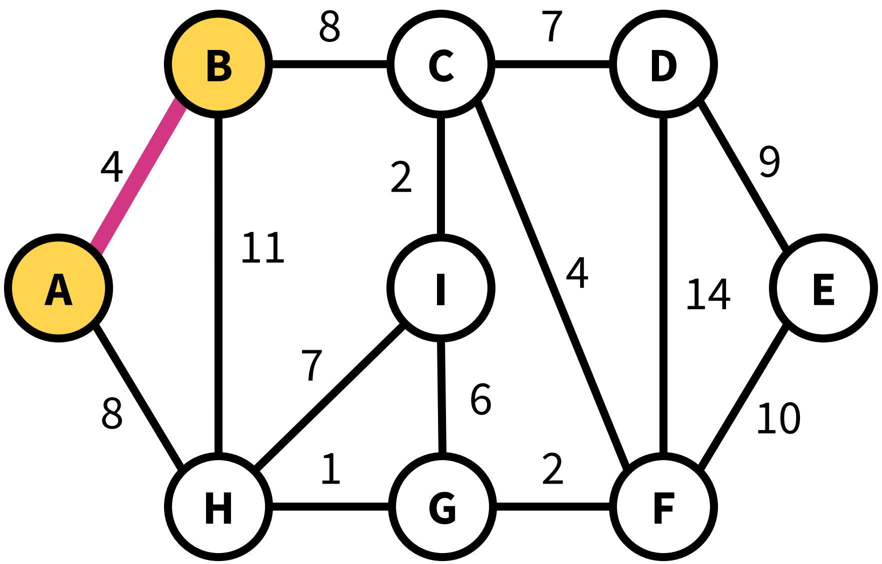
Prim's Algorithm

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



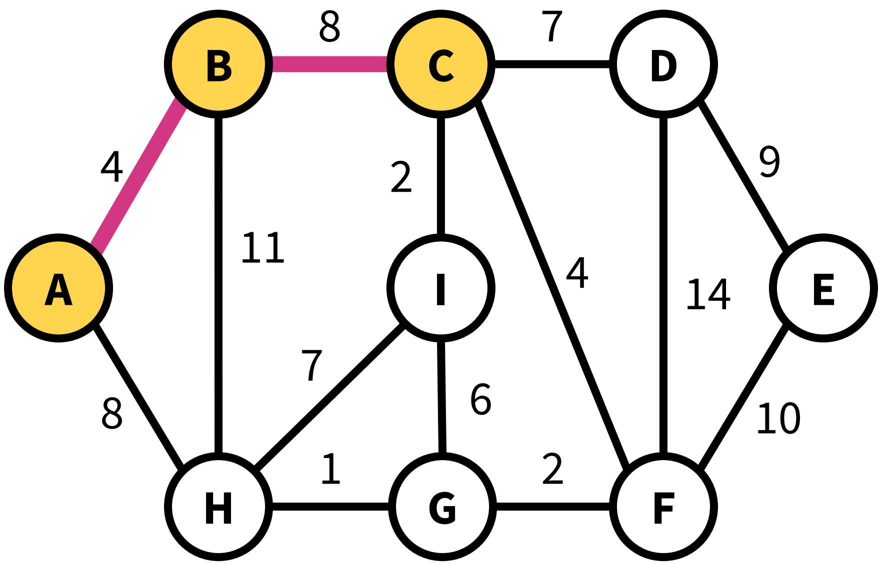
Prim's Algorithm

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



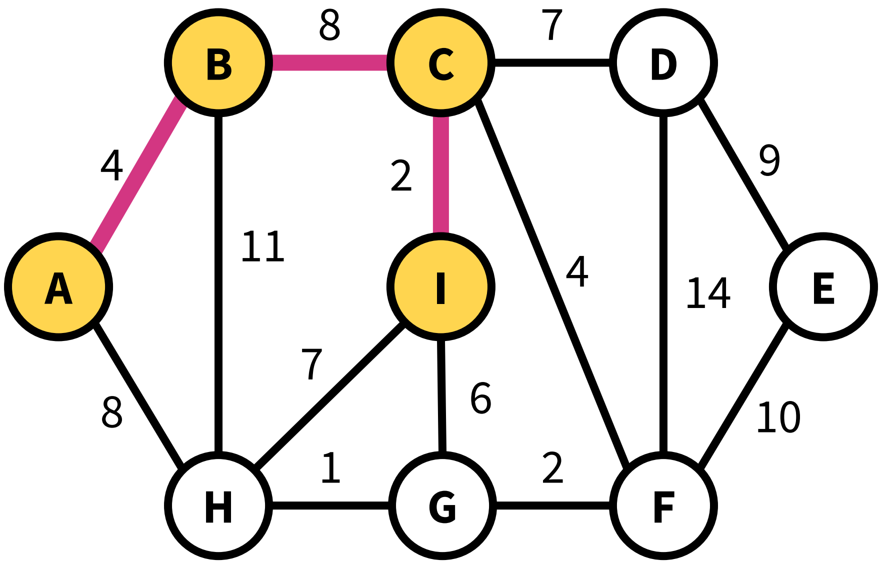
Prim's Algorithm

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



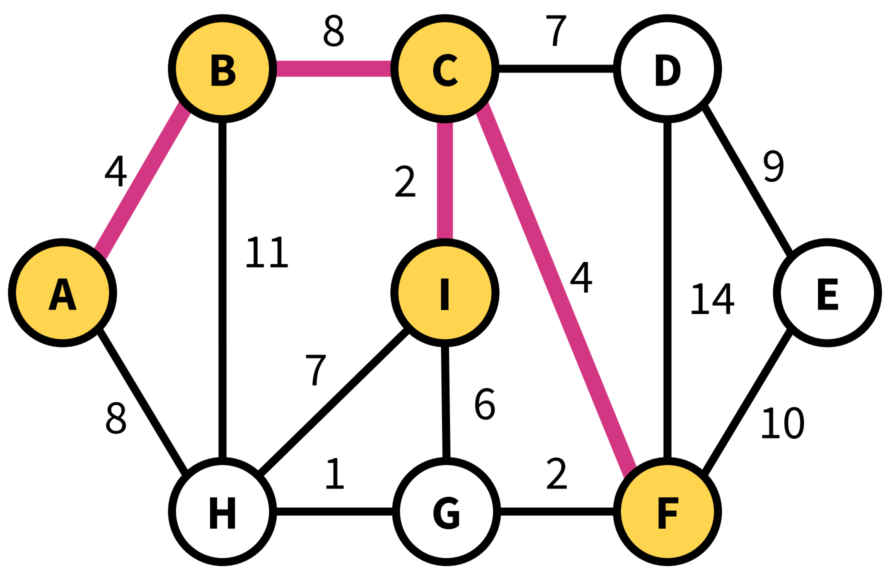
Prim's Algorithm

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



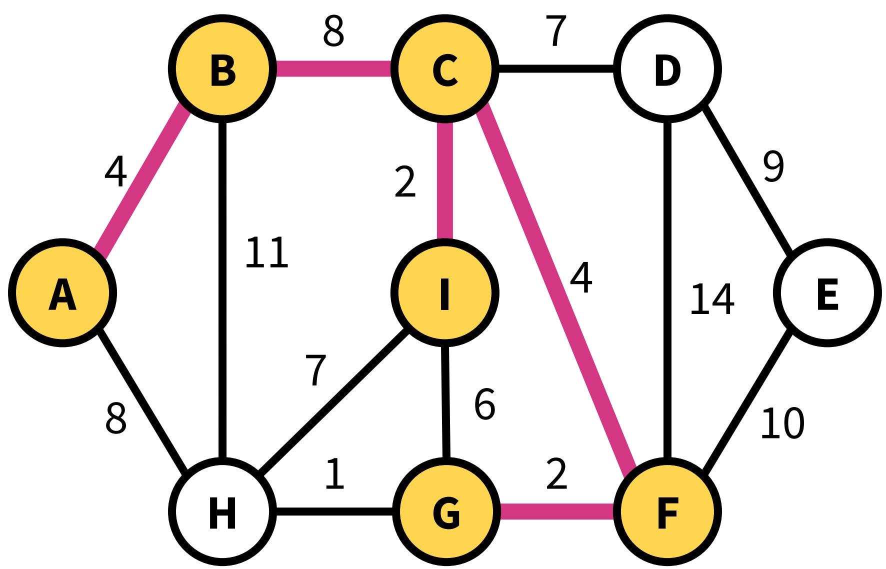
Prim's Algorithm

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



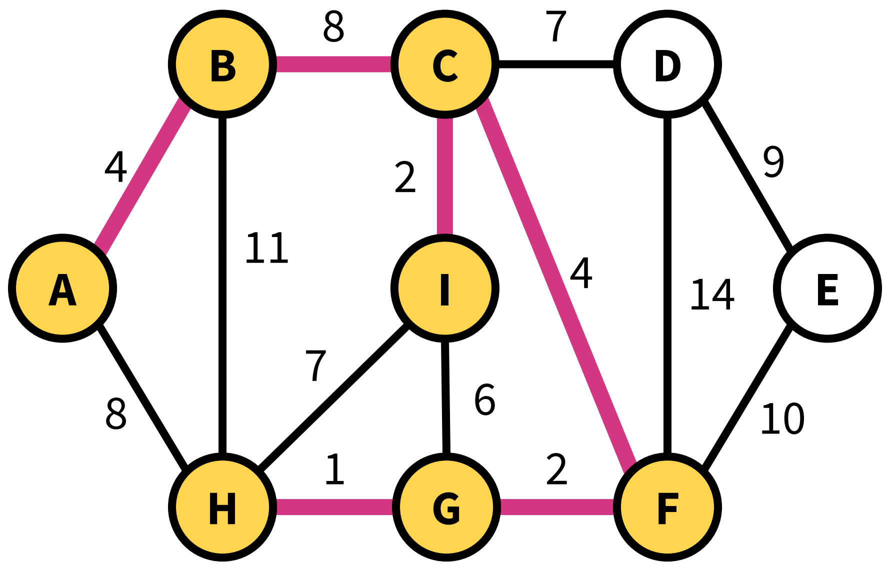
Prim's Algorithm

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



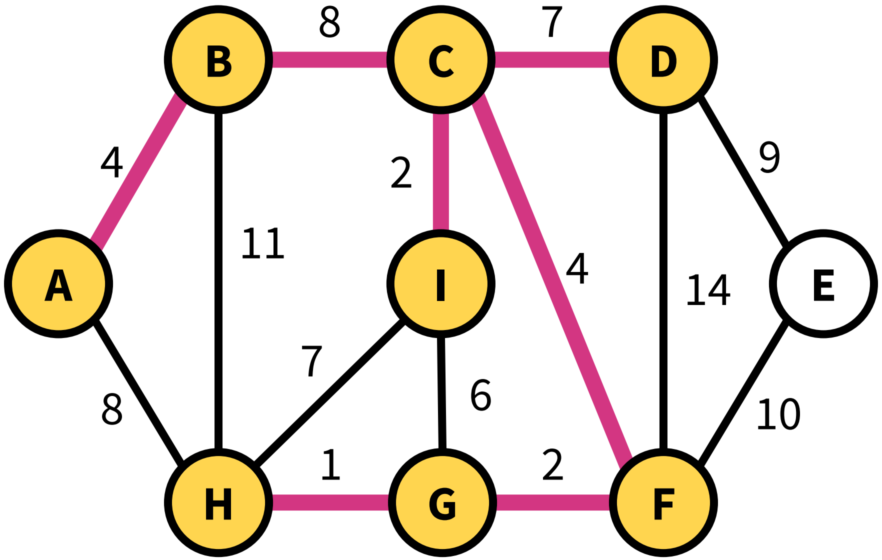
Prim's Algorithm

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



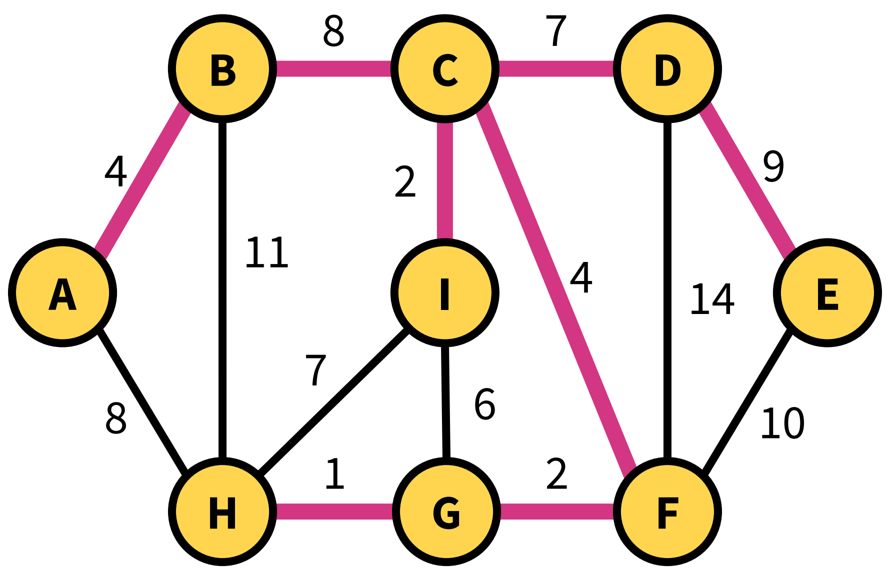
Prim's Algorithm

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

```
def 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

```
def 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

```
def 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: $O(|V| \cdot |E|)$

Prim's Algorithm

```
def 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)}.

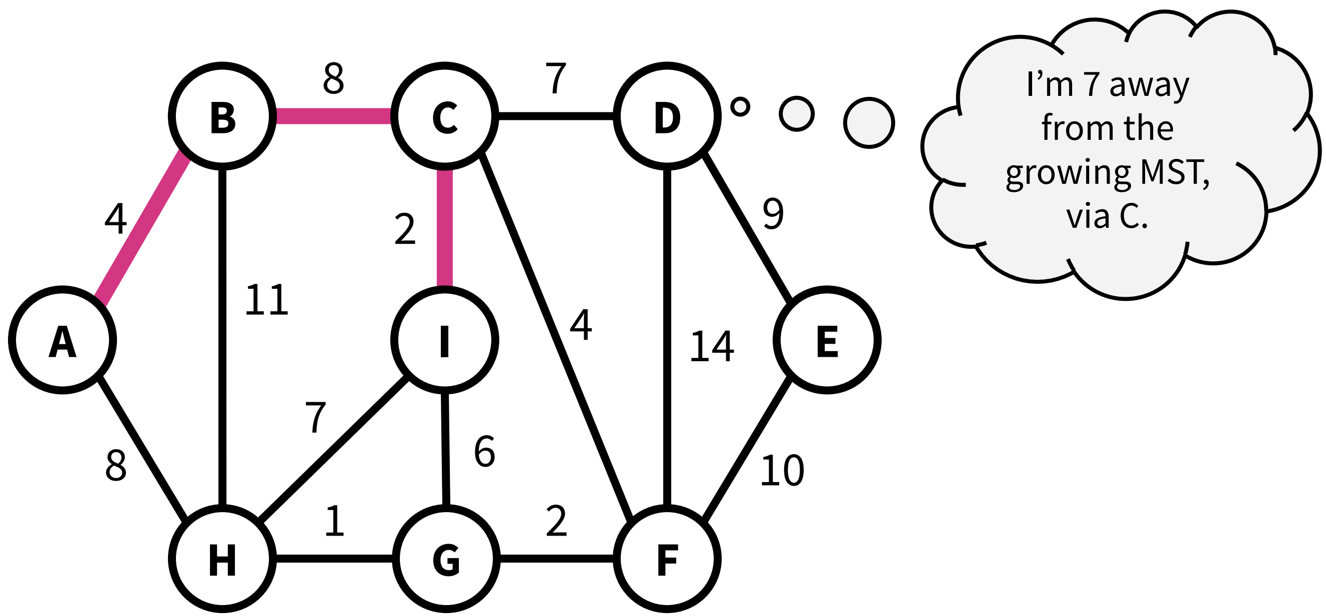
After adding the the (n-1)st edge, we have a spanning tree; therefore, `MST` contains a minimum spanning tree. ■

Recall, we proved our lemma with an exchange argument!

Prim's Algorithm

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

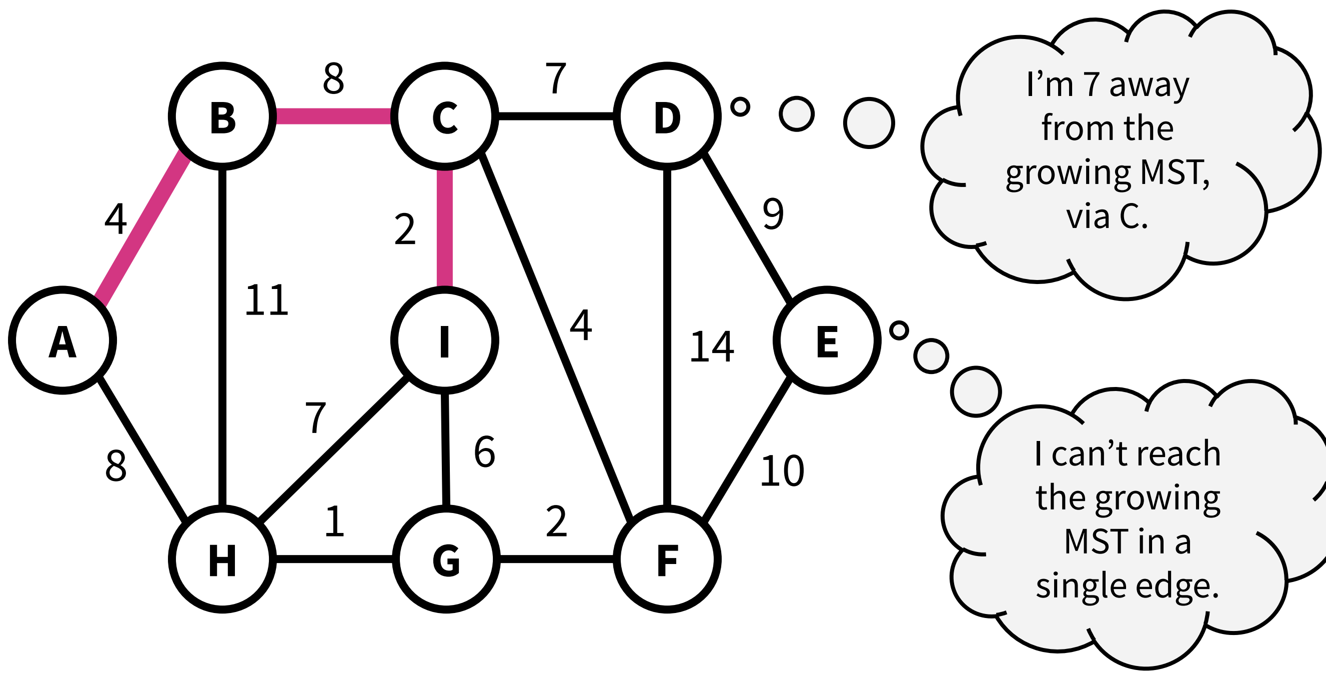
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.



Prim's Algorithm

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

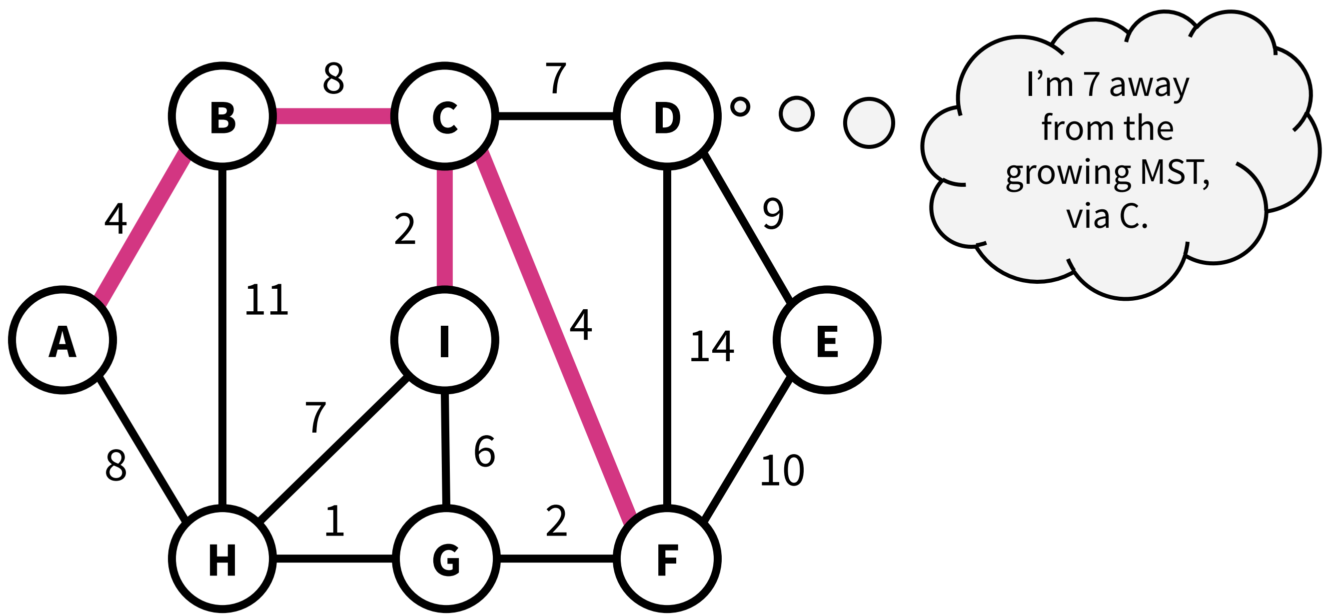
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.



Prim's Algorithm

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

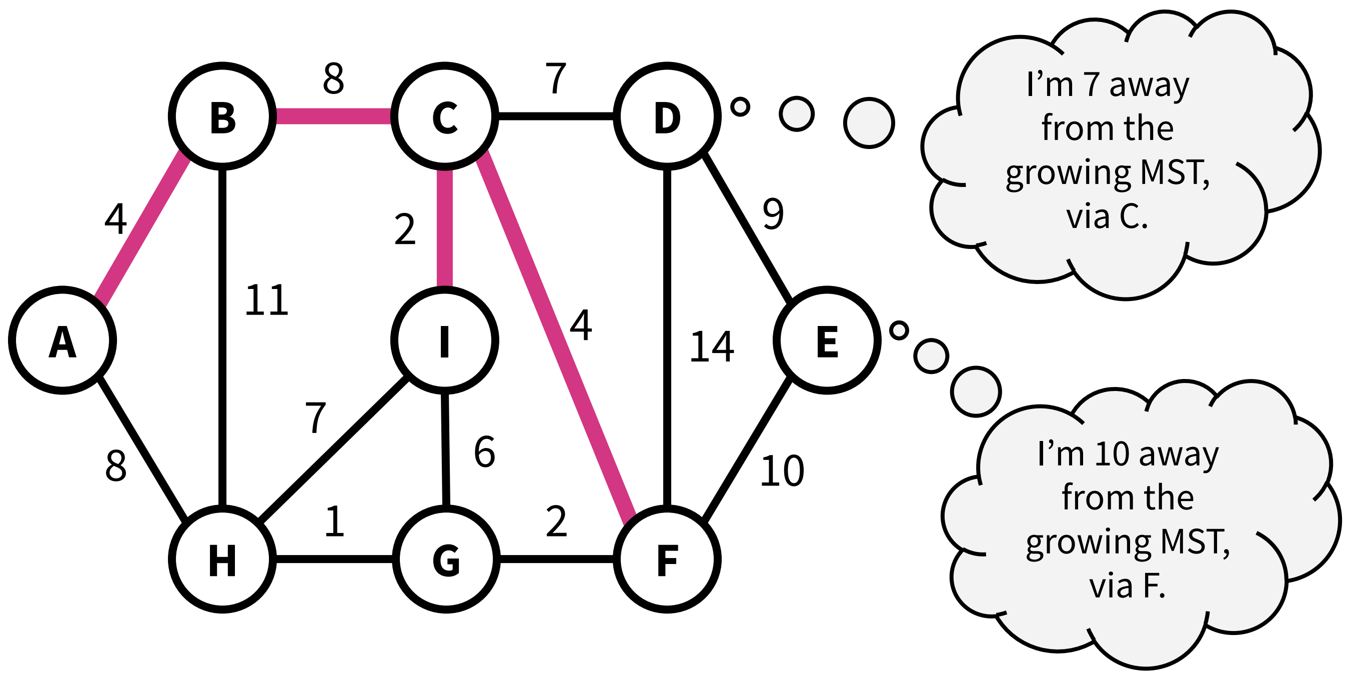
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.



Prim's Algorithm

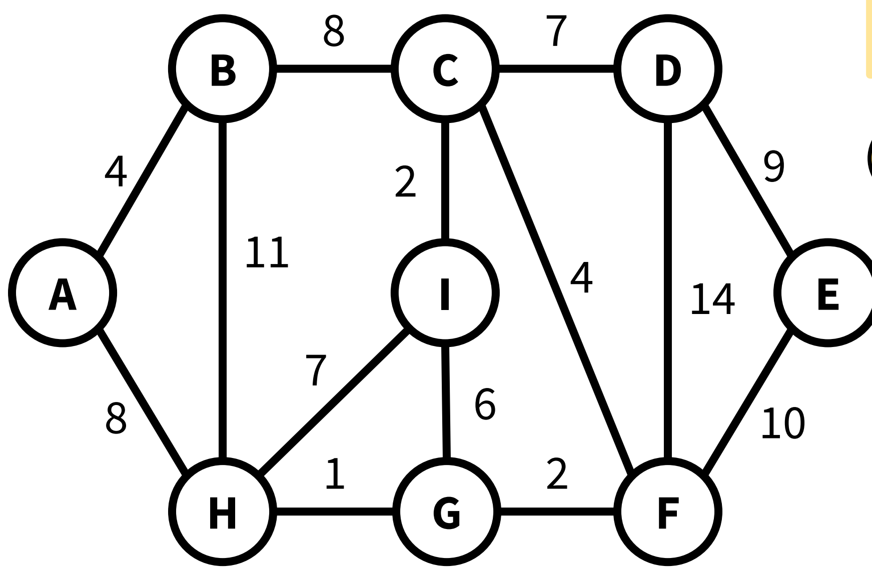
We called the algorithm `slow_prim`. There's a more efficient implementation.

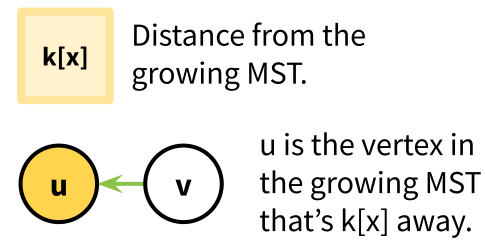
Main idea: vertices maintain information about the distance from itself to the growing spanning tree (if one edge away) and how to get there.



Prim's Algorithm

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



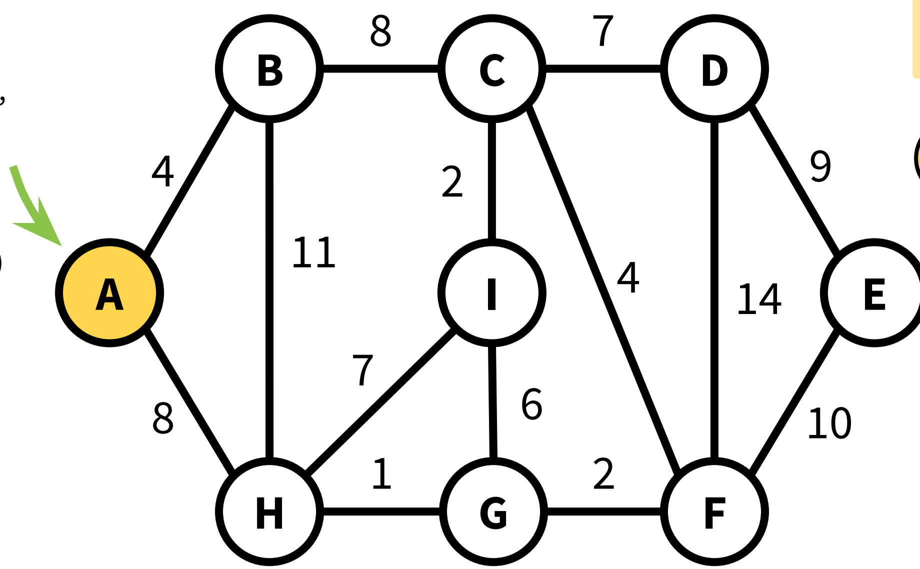


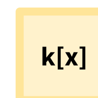
Prim's Algorithm

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

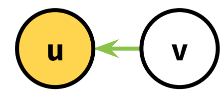
In addition to visiting A, update information of its neighbors with:

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





Distance from the growing MST.



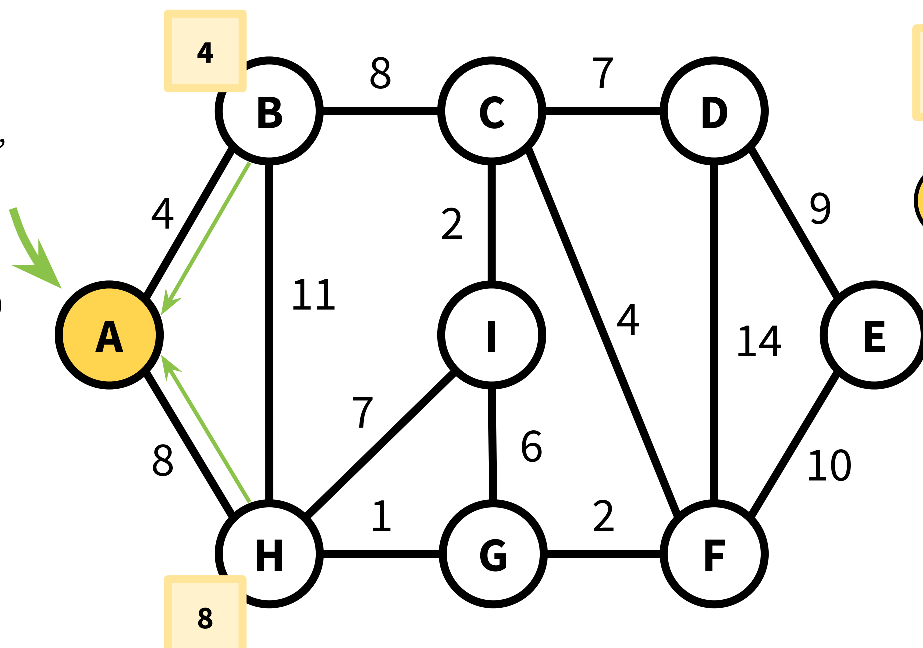
u is the vertex in the growing MST that's k[x] away.

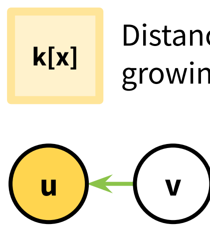
Prim's Algorithm

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

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

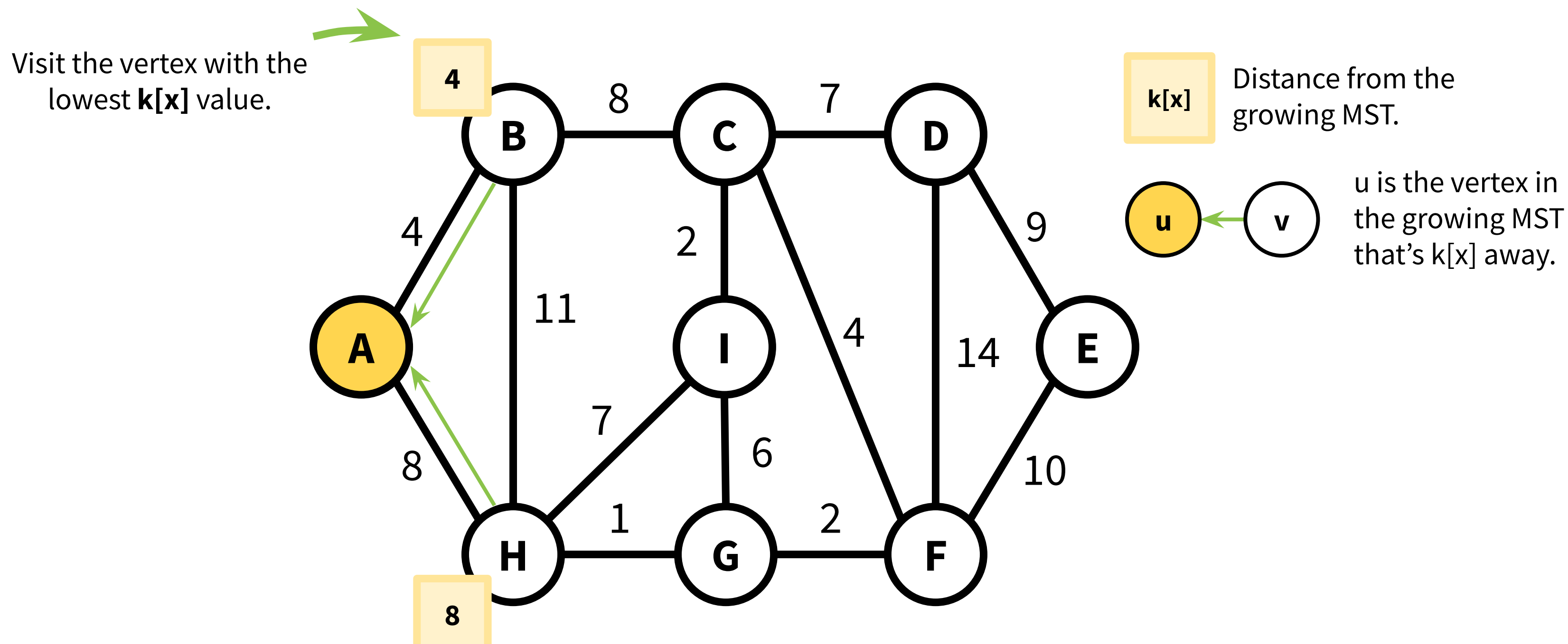
for v in u.neighbors:
  k[v] = min(k[v], w(u, v))





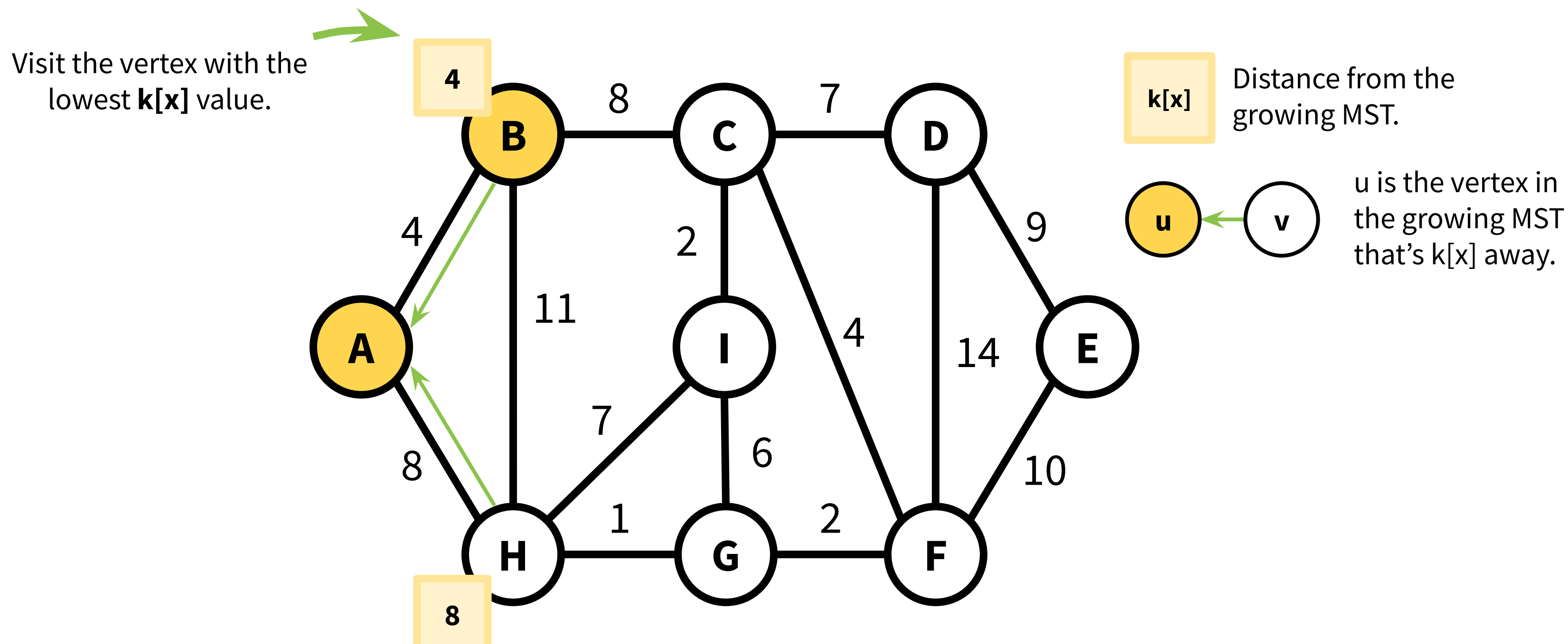
Prim's Algorithm

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



Prim's Algorithm

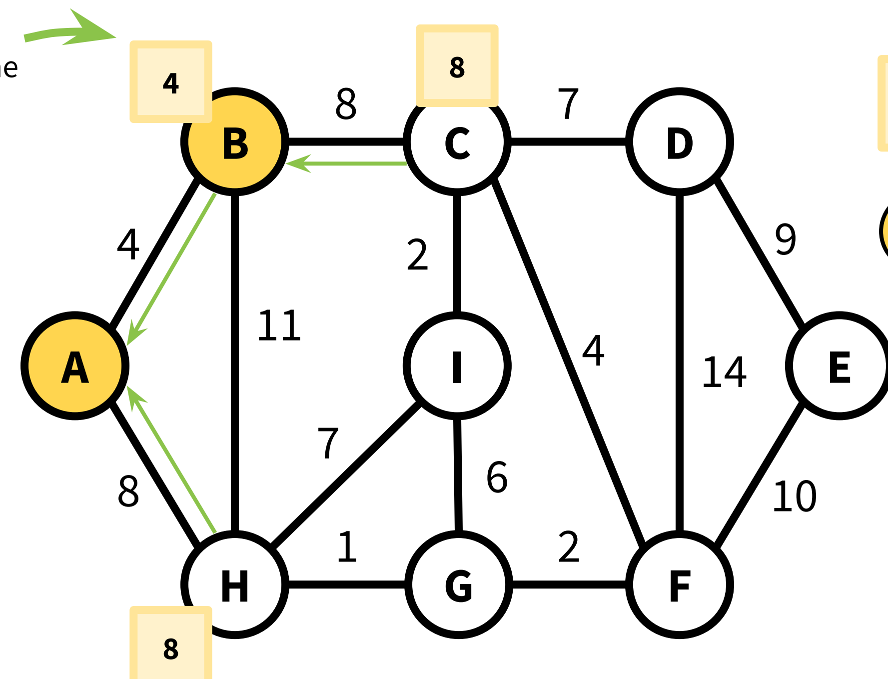
Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.

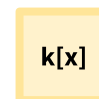


Prim's Algorithm

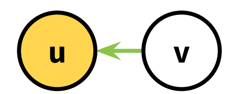
Main idea: vertices maintain information about the distance from the growing spanning tree and how to get there.

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





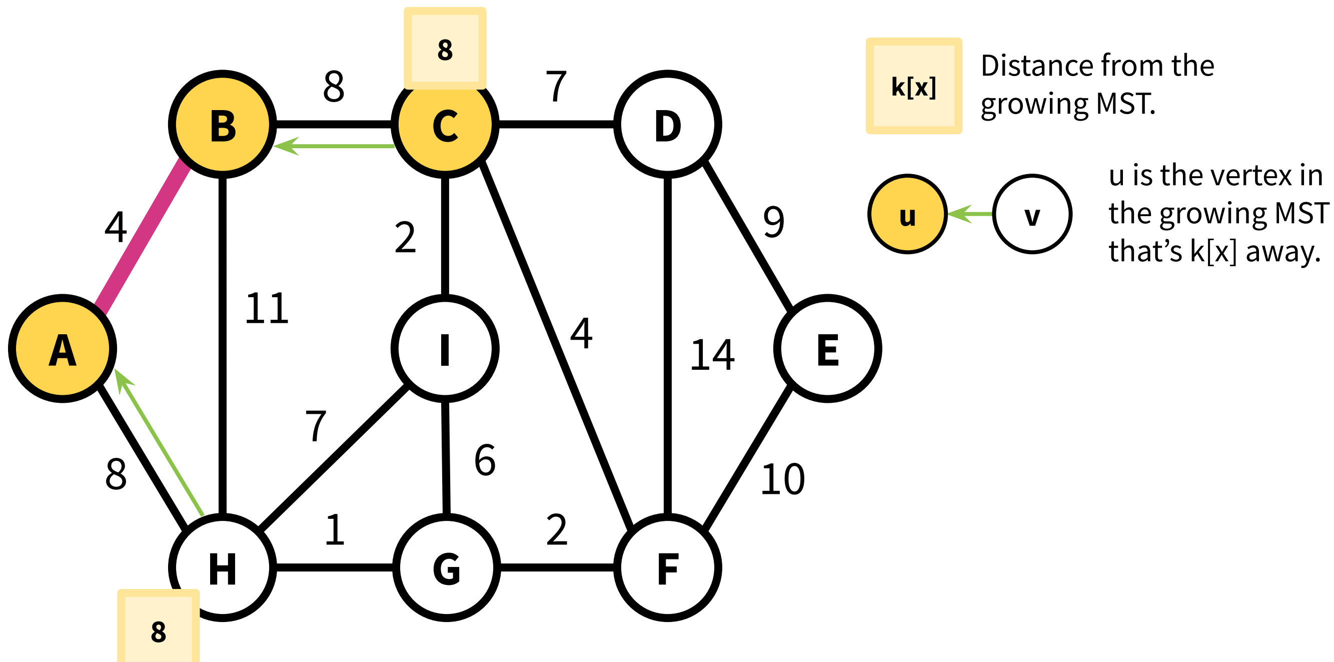
Distance from the growing MST.



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

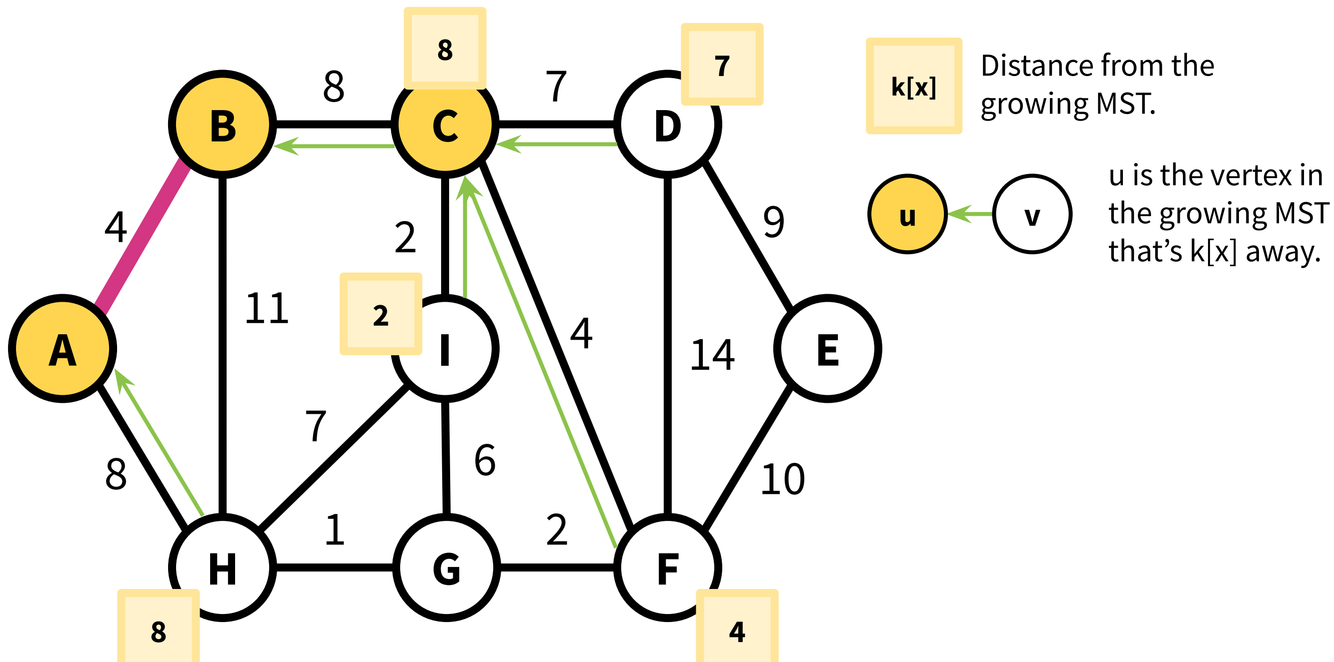
Prim's Algorithm

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



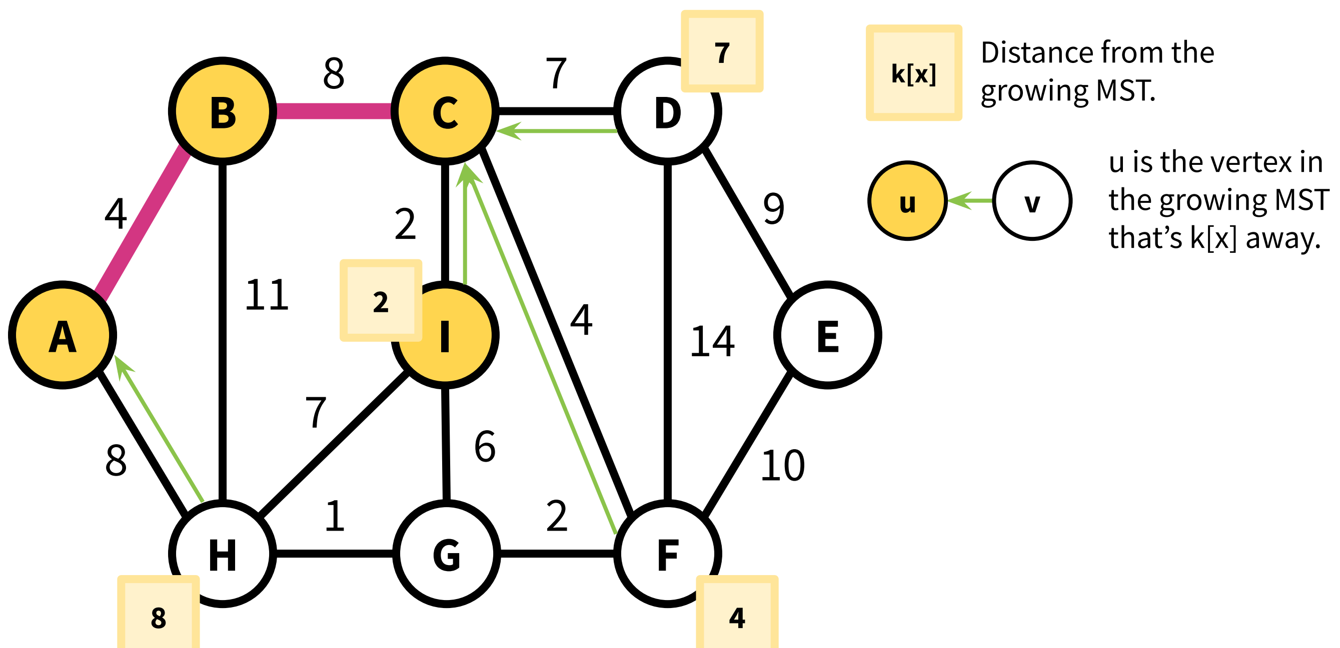
Prim's Algorithm

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



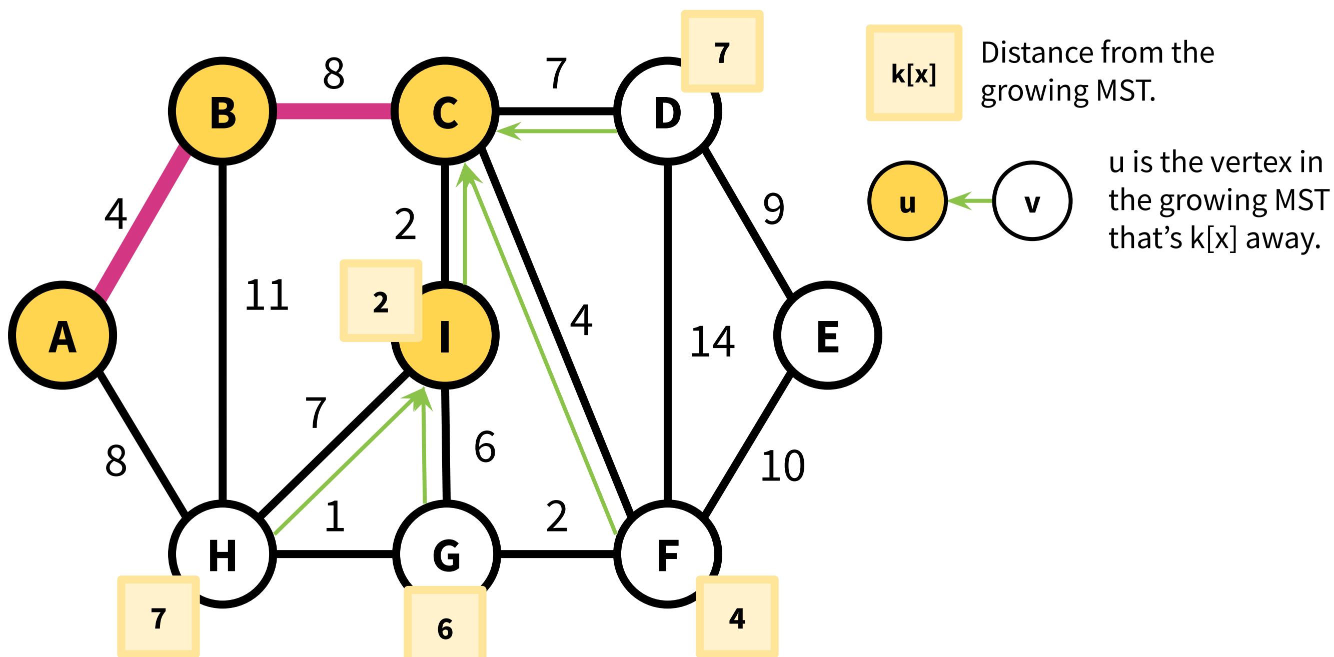
Prim's Algorithm

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



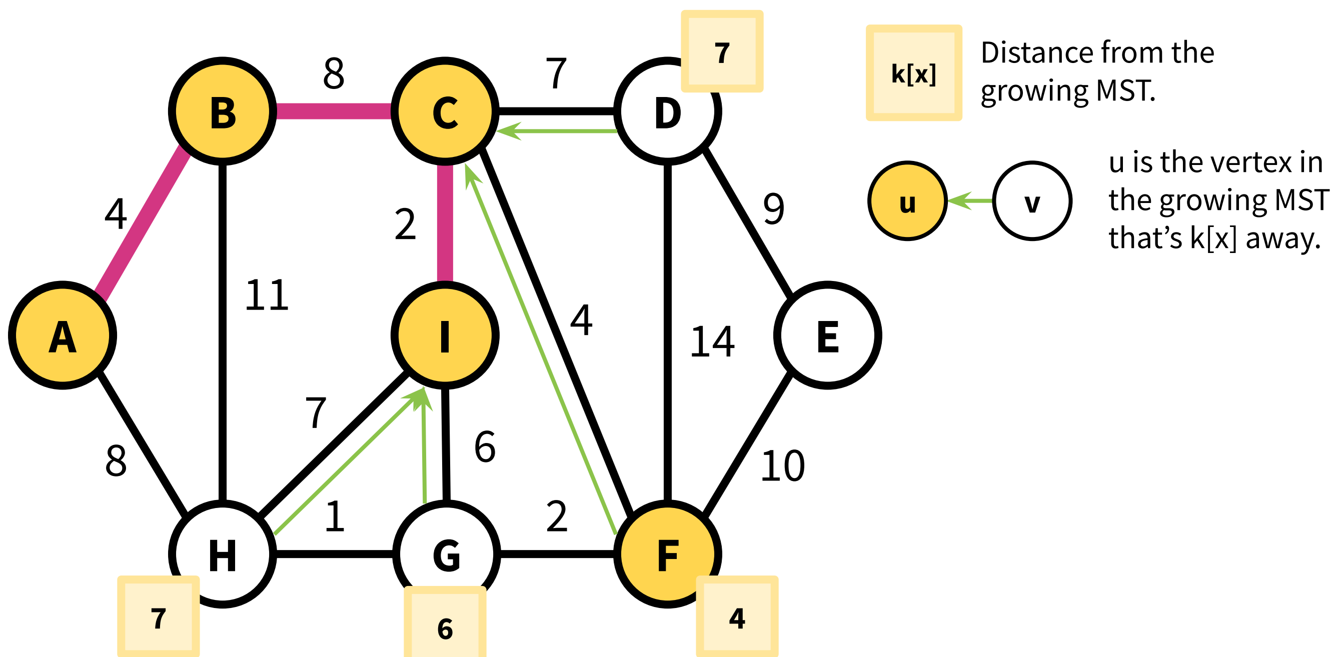
Prim's Algorithm

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



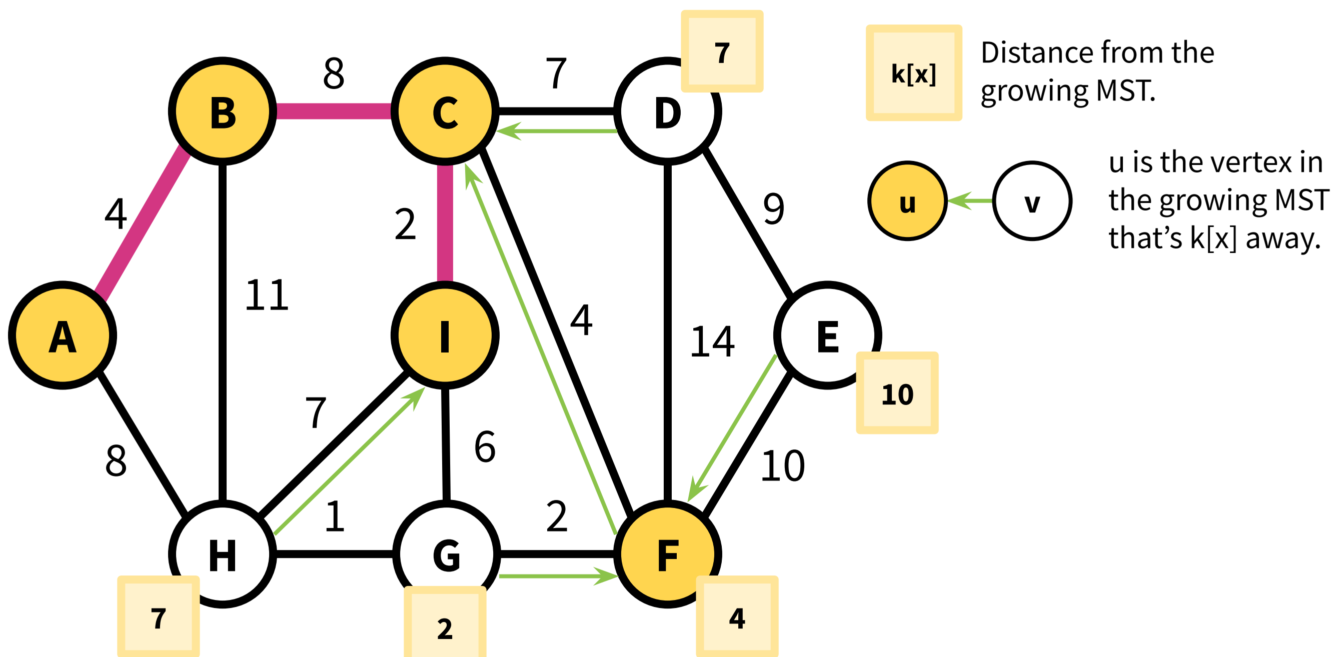
Prim's Algorithm

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



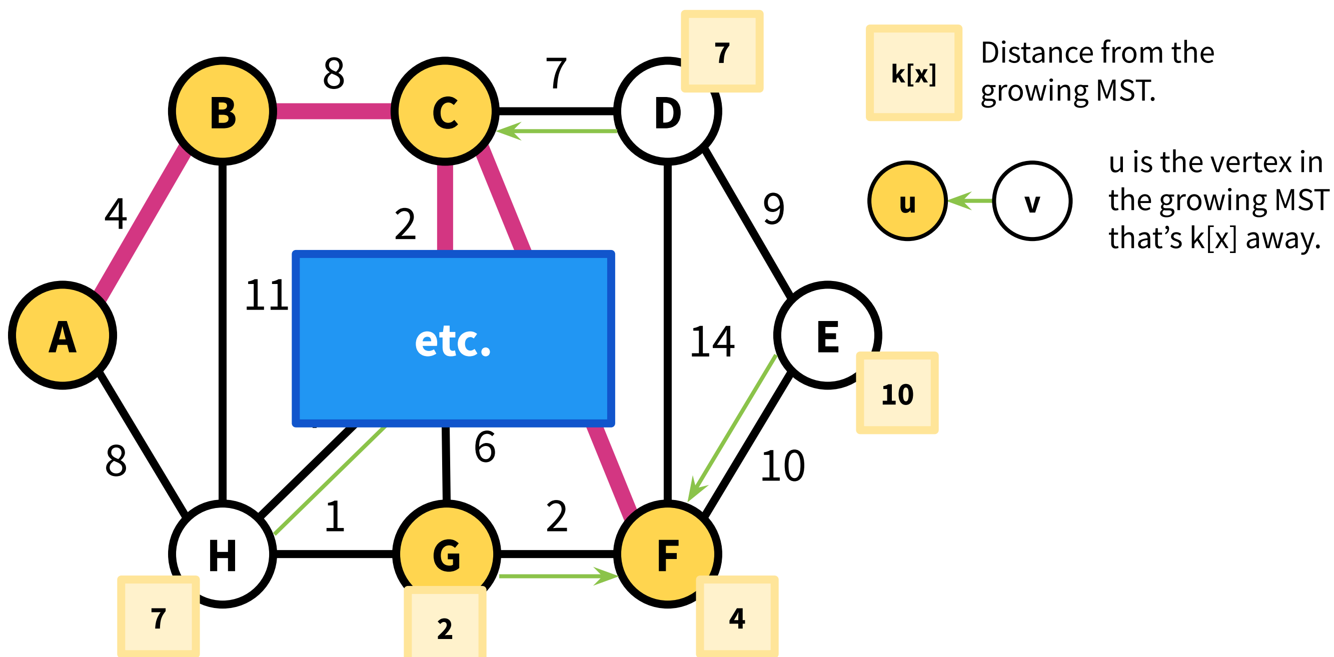
Prim's Algorithm

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



Prim's Algorithm

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



Prim's Algorithm

```
def 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