Lecture 9: Chapter 4 Part 1

The Greedy Approach
CS3310

Optimization Problems

- An **optimization problem** can have more than one possible solution for an instance.
- Sorting is <u>not</u> an optimization problem.
  - i.e. A list is either sorted or not, so there is only one solution.
- Finding a path from your home to school *is* an optimization problem, since many different routes exist.
  - However, only one of these routes is **optimal**.
    - **Note**: ties can exist (two or more different routes can be equally fast). In such a case, we arbitrarily pick one of the fastest routes.
- With *optimization problems* we are usually interested in maximizing or minimizing some value.

Greedy Algorithms

- A **greedy algorithm** calculates a solution to an optimization problem by making a sequence of choices, each of which seems best at the moment.
    - Each choice is called **locally optimal**.
- The goal is to find a solution that is **globally optimal** i.e. the best solution for the overall problem.
    - Some greedy algorithms accomplish this, others don't.

- Imagine you are picking a route to school. You must go through either points A-B-C-D or A-E-F-G to get here.
- A-B is much faster than A-E so you take that road; it is locally optimal.
- However, there is horrible traffic from C-D, so A-E-F-G would have been faster overall. This greedy algorithm didn't work because it didn't consider **global state**.

Greedy Algorithms

In the next few lectures, we'll discuss a few greedy algorithms that always find a globally optimal solution, unlike the previous example.

Greedy Algorithms repeat the following steps:

1. **Selection Procedure**: Decide what option is *locally* optimal.
2. **Feasibility Check**: Make sure the chosen value does not make us exceed our goal.
3. **Solution Check**: See if the problem is solved. If not, return to step 1.

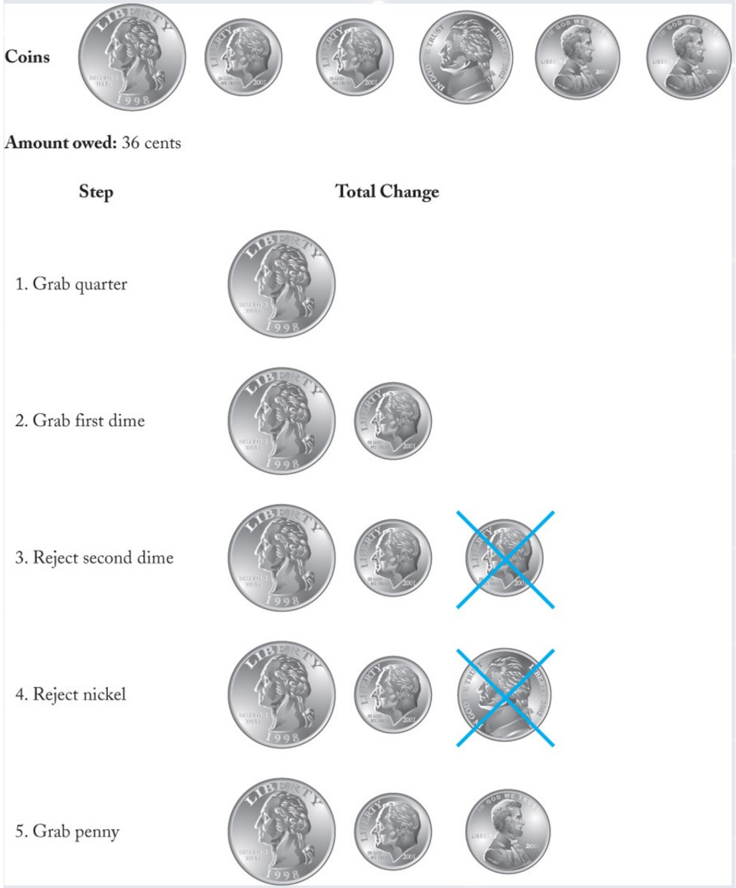
Greedy Algorithms

While using **U.S. coins**, a greedy algorithm always works for the problem of giving change.

- We want to calculate a given amount of change using as few coins as possible.

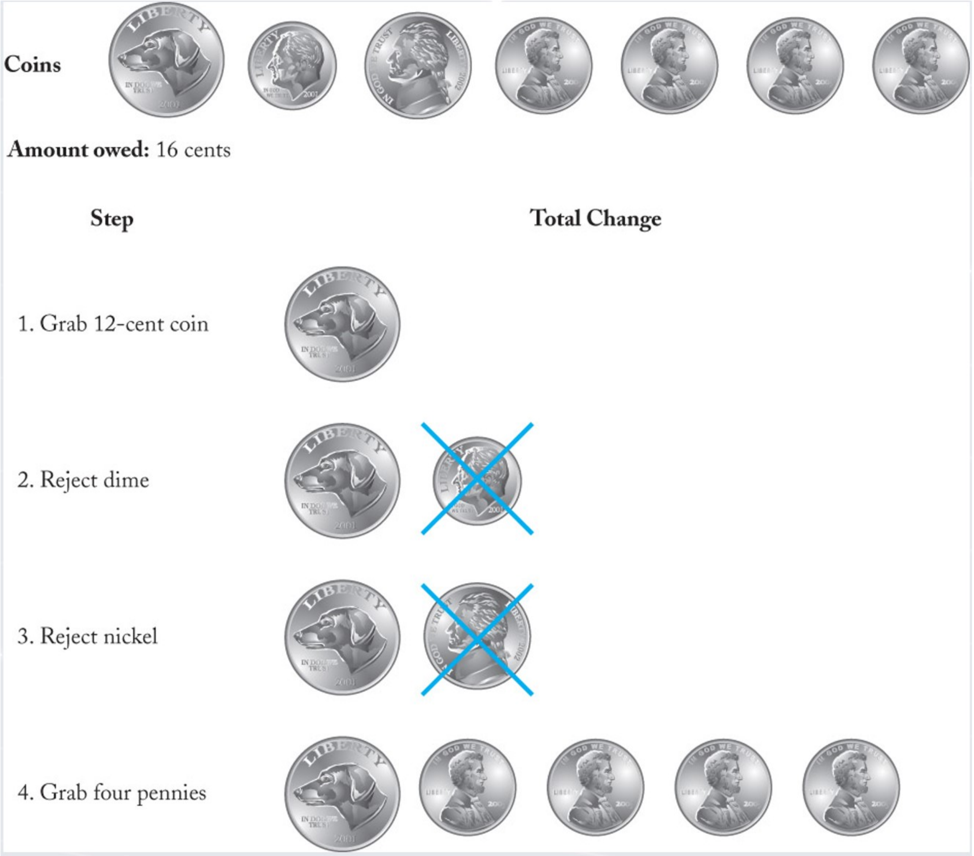
```
while (there are more coins and instance isn't solved)
        grab the largest remaining coin;                        // selection procedure
        if (adding coin makes change > amount owed)        // feasibility check
                reject the coin
        else
                add the coin to the change;
        if (total value of change = amount owed)           // solution check
                instance is solved
```

Greedy Algorithms



Greedy Algorithms

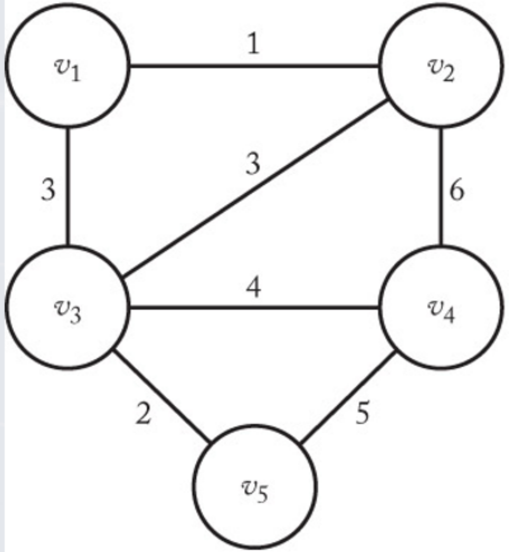
While the previous algorithm always generates a globally optimal solution with U.S. coins, it doesn't always do so if other values are added. Imagine we also have access to a 12 cent coin.



Graph Theory Refresher

- A **graph** is made up a **vertices** and **edges** which connect them.
- A graph is **undirected** if its edges have no direction
- A **path** in an undirected graph is a sequence of vertices such that there is an edge between each vertex and its successor
    - i.e. { v_1, v_2, v_4 }
- An undirected graph is **connected** if there is a path between every pair of vertices.
- A path from a vertex to itself, which contains at least three vertices and in which all intermediate vertices are distinct, is a simple **cycle**
    - i.e. { v_1, v_2, v_3, v_1 }

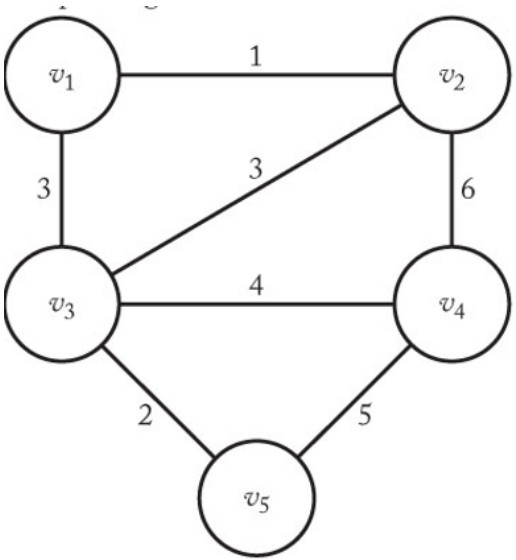
(a) A connected, weighted, undirected graph G.



Graph Theory Refresher

An **undirected** graph G consists of a finite set V whose members are the vertices of G, together with a set E of pairs of vertices in V. These pairs are the **edges** of G. We denote G by G = (V, E).

- Members of E are denoted by (v_i, v_j)



This graph is made up of the following vertices and edges:

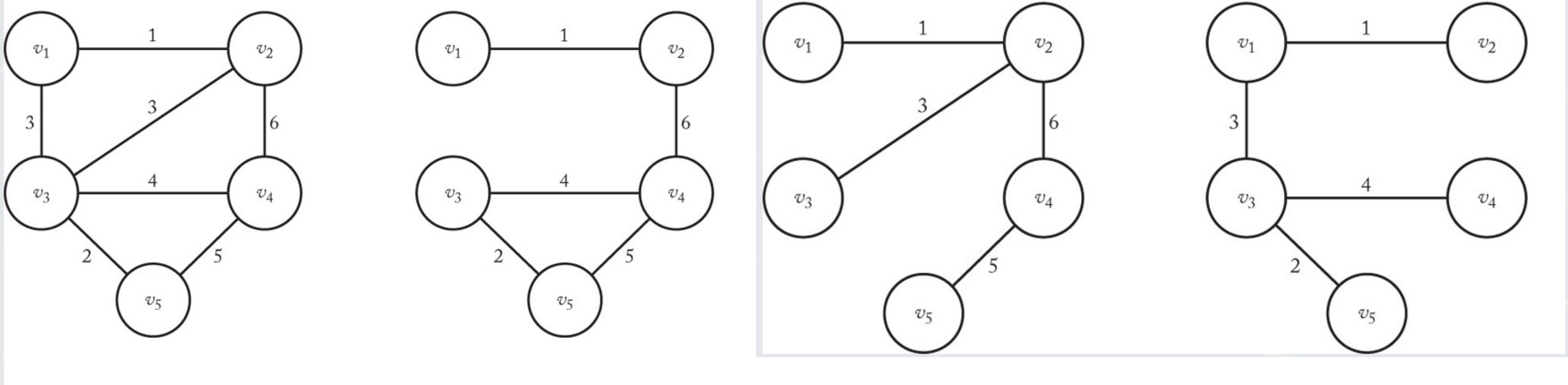
- $V = \{ v_1, v_2, v_3, v_4, v_5 \}$
- $E = \{ (v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5)$

Minimum Spanning Trees

- A **free tree** is a connected, undirected graph such that no cycles exist.
- No vertex is designated the root in a free tree.
  - This is different from the more common **rooted tree**, such as a binary tree.

- Suppose we want to connect a group of certain cities with roads, but we also want to use as little road as possible.
  - If we create a **minimum spanning tree**, each city in the graph is connected to each other city, but not necessarily directly.

Minimum Spanning Trees



(a) A connected, weighted, undirected graph G.

(b) If (v_4, v_5) were removed from this subgraph, the graph would remain connected.

(c) A spanning tree for G.

(d) A minimum spanning tree for G.

Minimum Spanning Trees

A very high-level greedy algorithm to find a minimum spanning tree.

```
F = {}
while (the instance is not solved)
{
        select an edge that is locally optimal;      // selection procedure

        if (adding edge to F doesn't create a cycle) // feasibility check
                add it;
        if (T = (Y, F) is a spanning tree)            // solution check
                the instance is solved
}
```

Prim's Algorithm

Problem: Given a graph G = (V, E), determine a minimum spanning tree T = (Y, F).

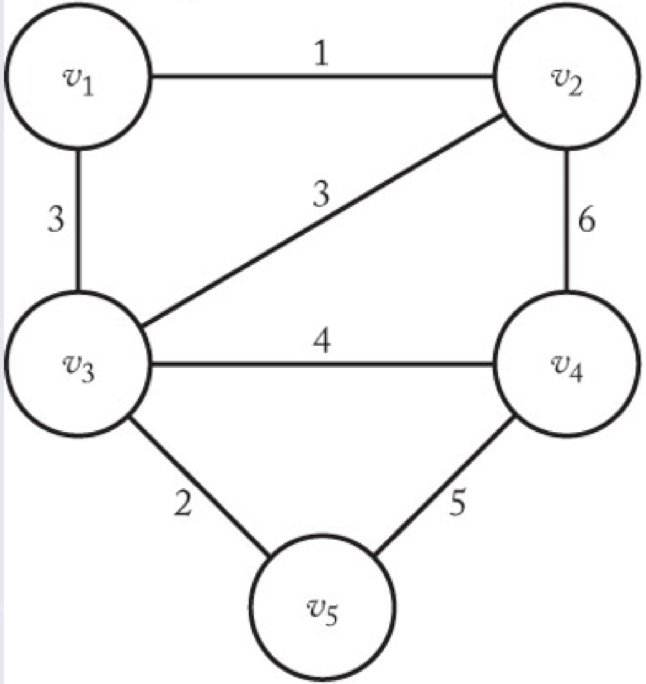
- Initialize an empty subset F to contain the edges in the minimum spanning tree.
- Initialize a subset of vertices Y with one arbitrary vertex from V.
  - Once every vertex from V is in Y, the spanning tree is complete.
- A vertex **nearest** to Y is a vertex in V - Y that is connected to a vertex in Y by an edge of minimum weight.
  - In other words, select a vertex in V that is *not* in Y that is connected to a vertex in Y with an edge of the smallest possible weight.
    - This ensures that a cycle is not created.
  - Ties are broken arbitrarily.

Prim's Algorithm

```
F = {}                    // initialize set of edges to empty
Y = { v1 }                // initialize set of vertices to contain first one

while (the instance is not solved)
          // selection procedure and feasibility check:
          select a vertex in V - Y that is nearest to Y

          add the vertex to Y;
          add the edge to F;

          // solution check:
          if (Y == V)
                    Every vertex has been added to Y: the instance is solved.
```

Prim's Algorithm

Determine a minimum spanning tree.



Prim's Algorithm

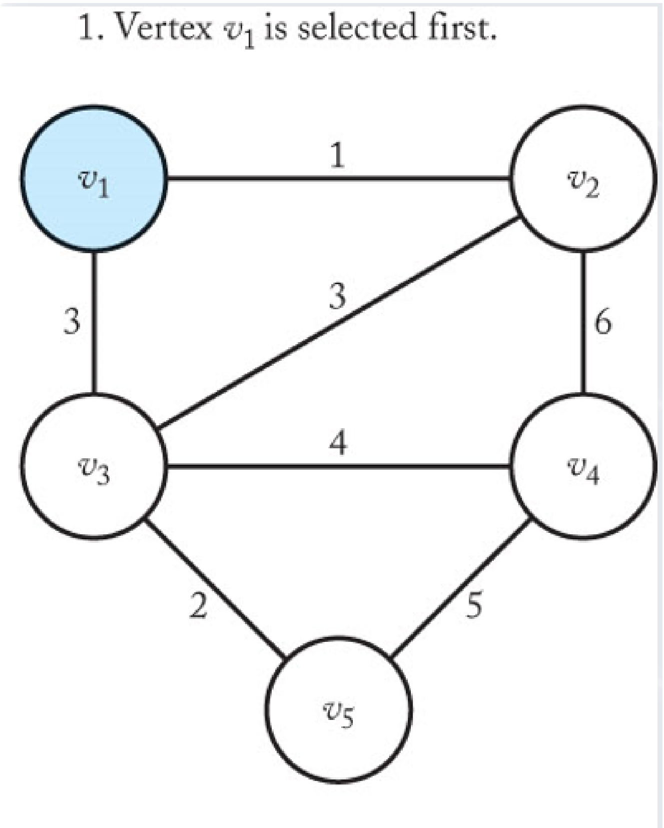
$F = \{\}$

$Y = \{v_1\}$

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{v_2, v_3, v_4, v_5\}$

Which vertex in V - Y is closest to a vertex in Y?



Prim's Algorithm

$F = \{\}$

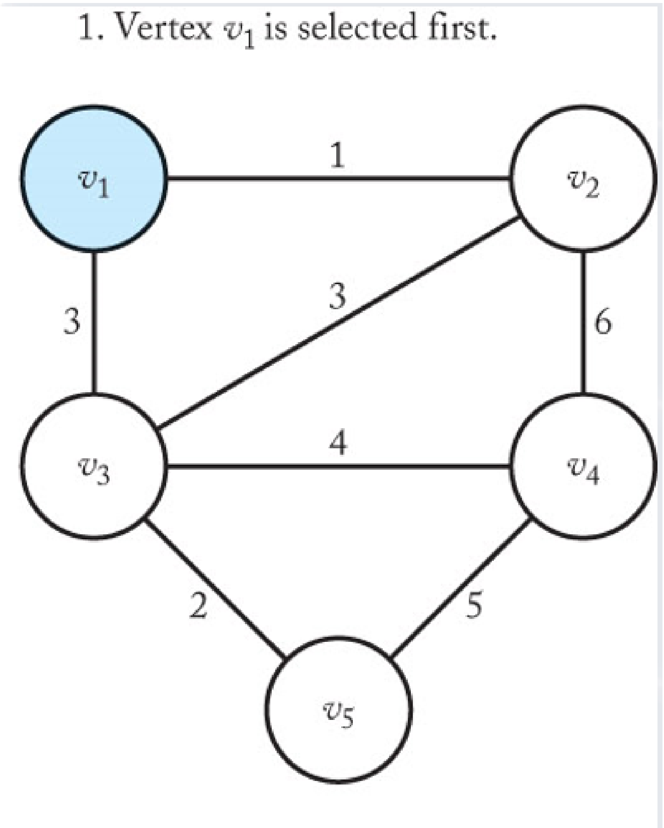
$Y = \{v_1\}$

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{v_2, v_3, v_4, v_5\}$

Which vertex in V - Y is closest to a vertex in Y?

- v_3 connects to v_1 with a weight of 3
- v_2 connects to v_1 with a weight of 1



Prim's Algorithm

$F = \{(\mathbf{v_1}, \mathbf{v_2})\}$

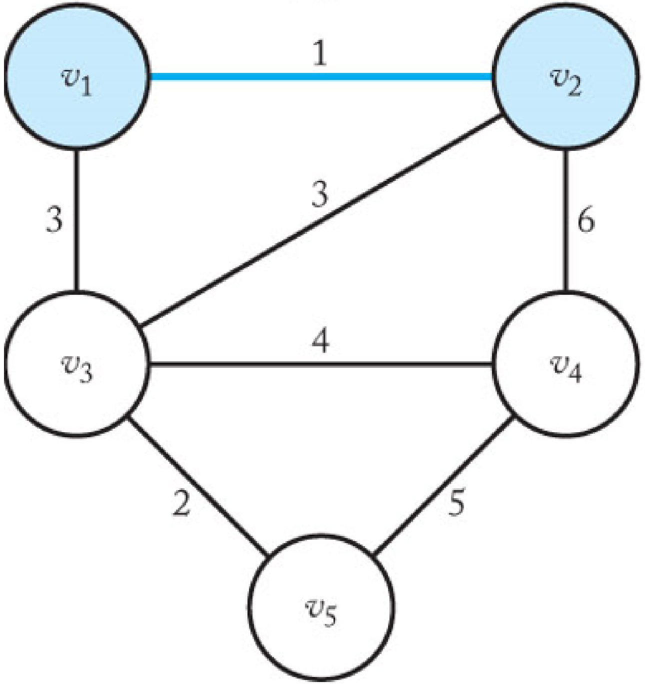
$Y = \{v_1, \mathbf{v_2}\}$

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{v_3, v_4, v_5\}$

Which vertex in V - Y is closest to a vertex in Y?

2. Vertex v_2 is selected because it is nearest to $\{v_1\}$.



Prim's Algorithm

$F = \{(v_1, v_2)\}$

$Y = \{v_1, v_2\}$

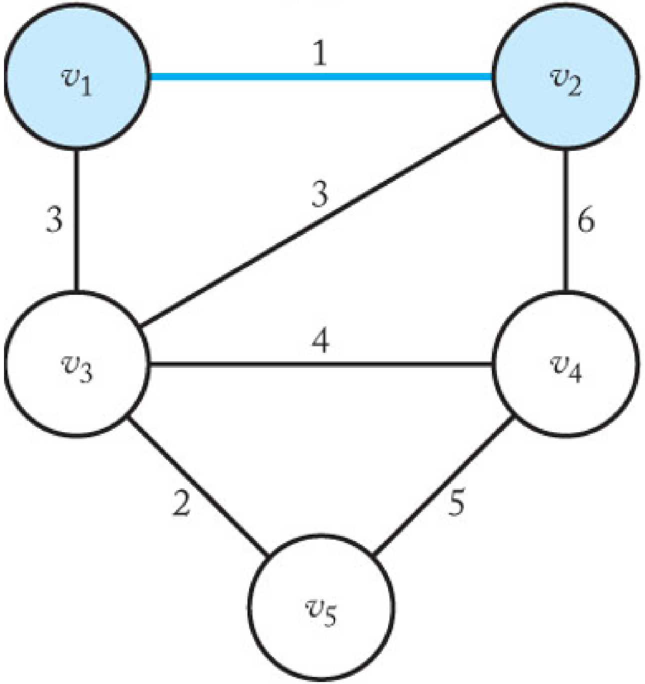
$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{v_3, v_4, v_5\}$

Which vertex in V - Y is closest to a vertex in Y?

- v_4 connects to v_2 with a weight of 6.
- v_3 connects with a weight of 3 to both v_1 and v_2
    - Tie is broken arbitrarily. We'll choose v_1

2. Vertex v_2 is selected because it is nearest to $\{v_1\}$.



Prim's Algorithm

$F = \{(v_1, v_2), (\boldsymbol{v_1}, \boldsymbol{v_3})\}$

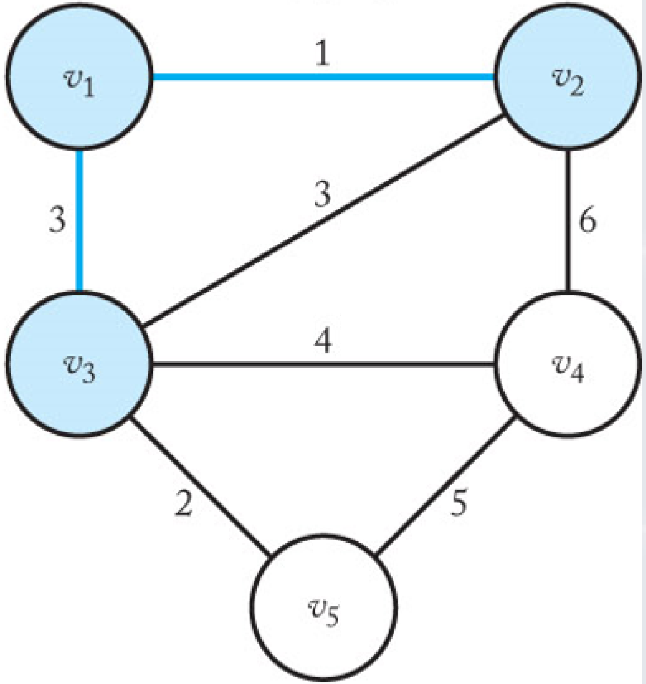
$Y = \{v_1, v_2, \boldsymbol{v_3}\}$

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{v_4, v_5\}$

Which vertex in V - Y is closest to a vertex in Y?

3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$.



Prim's Algorithm

$F = \{(v_1, v_2), (v_1, v_3)\}$

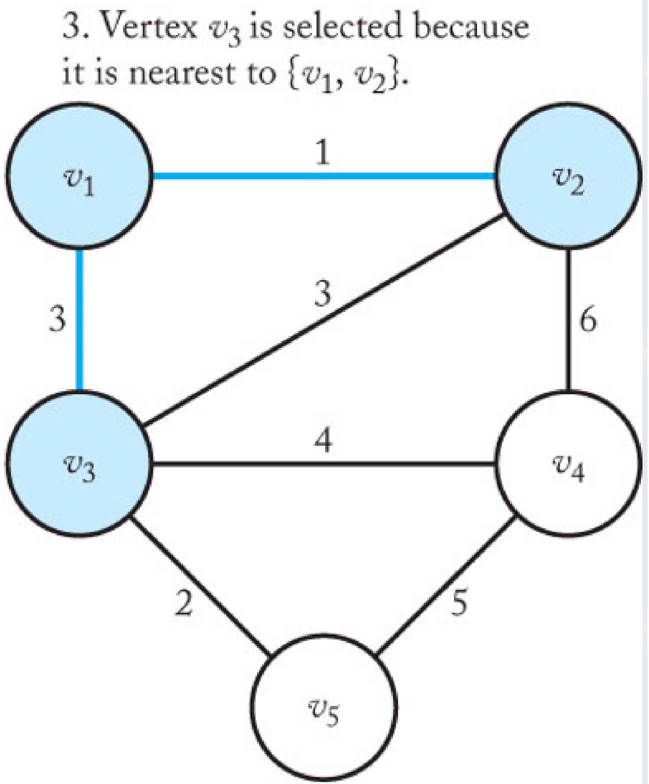
$Y = \{v_1, v_2, v_3\}$

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{v_4, v_5\}$

Which vertex in V - Y is closest to a vertex in Y?

- v_4 connects to v_2 with a weight of 6 and v_3 with a weight of 4.
- v_5 connects v_3 with a weight of 2.



Prim's Algorithm

$F = \{(v_1, v_2), (v_1, v_3), (\boldsymbol{v_3}, \boldsymbol{v_5})\}$

$Y = \{v_1, v_2, v_3, \boldsymbol{v_5}\}$

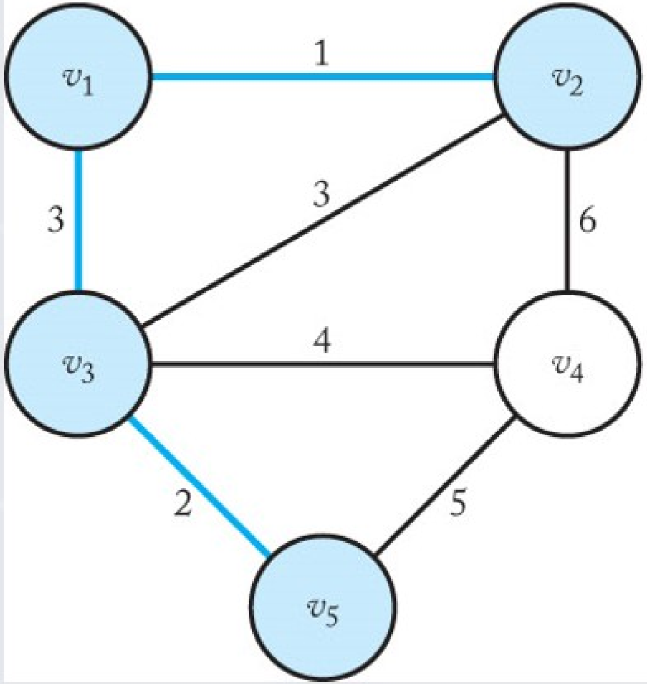
$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{v_4\}$

Which vertex in V - Y is closest to a vertex in Y?

- v_4 is all that's left! The edge with the smallest weight between it and Y is the one to v_3

4. Vertex v_2 is selected because it is nearest to $\{v_1, v_2, v_3\}$.



Prim's Algorithm

$F = \{(v_1, v_2), (v_1, v_3), (v_3, v_5), (\boldsymbol{v_3}, \boldsymbol{v_4})\}$

$Y = \{v_1, v_2, v_3, \boldsymbol{v_4}, v_5\}$

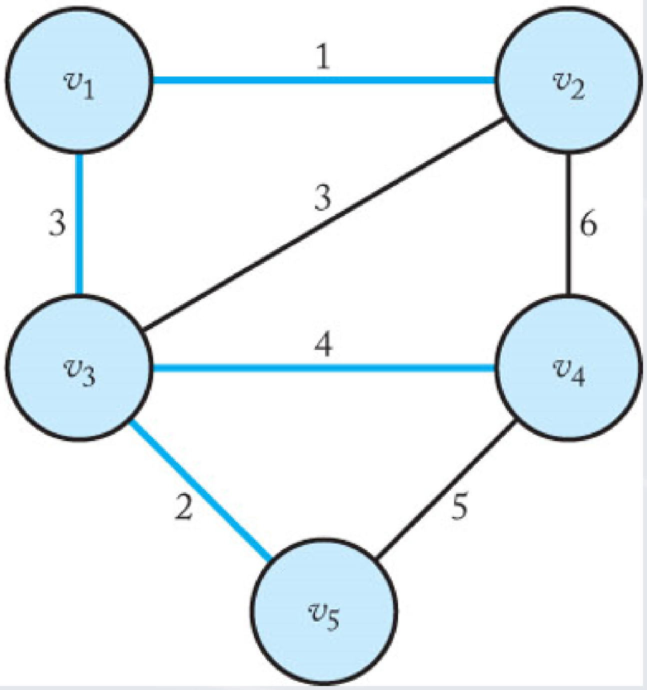
$V = \{v_1, v_2, v_3, v_4, v_5\}$

$V - Y = \{\}$

Once V - Y is empty, we have a spanning tree. Every vertex from V is in Y.

- T = (F, Y) is a minimum spanning tree with a total weight of 10.

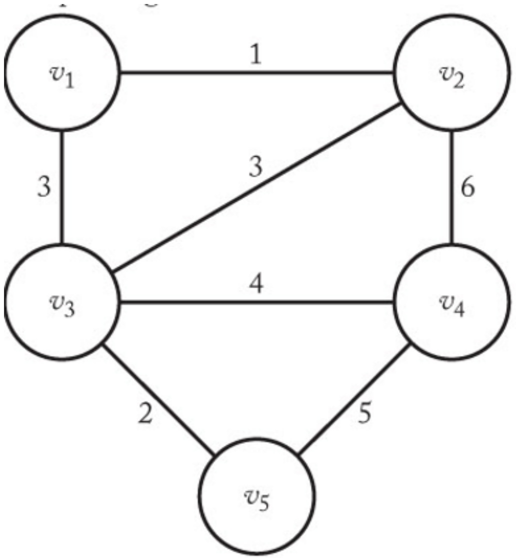
5. Vertex v_4 is selected.



Prim's Algorithm

While the previous steps are fairly straightforward for a human, we need a step-by-step procedure for a computer to follow.

Prim's Algorithm



We will represent the graph by an adjacency matrix

W[*i*][*j*] =

- 0 if $i = j$
- ∞ if there is no edge between *i* and *j*
- weight on edge if there is an edge between *i* and *j*

Prim's Algorithm

Along with the adjacency matrix, we keep two other arrays, `nearest` and `distance`.

for $i = 2, \ldots, n$:

- `nearest[i]` = vertex in Y nearest to v_i
- `distance[i]` = weight on edge between v_i and the vertex in `nearest[i]`

For example, if we have the following:

Y = {1, 3}

V - Y = {2, 4, 5}

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

- ➢ `nearest[4] = 3`. This means 3 is the vertex in Y nearest to 4
- ➢ `distance[4] = 4`. This means the edge (3, 4) has a weight of 4

Prim's Algorithm

When we begin Prim's algorithm, Y = { 1 }

- ➢ Arbitrarily start with vertex 1 in Y (the spanning tree)

Determine:

- `nearest[2]`:
  - i.e. what vertex in Y is closest to 2?
- `distance[2]`:
  - i.e. what is the distance between `nearest[2]` and 2?

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim's Algorithm

When we begin Prim's algorithm, Y = { 1 }

➢ Arbitrarily start with vertex 1 in Y (the spanning tree)

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Determine:

- `nearest[2]`: **1**, since it's the only vertex in Y
  - i.e. what vertex in Y is closest to 2?
- `distance[2]`: `W[1][2]` = **1**
  - i.e. what is the distance between `nearest[2]` and 2?

Prim Pseudocode

```
void prim (int n, const number W[][], set_of_edges& F)
        int vnear; // will contain the vertex in V - Y nearest to Y each iteration
        edge e;

        index nearest[2..n];
        number distance[2..n];

        F = {}
        for (index i = 2; i <= n; i++)
                nearest[i] = 1;
                distance[i] = W[1][i];
```

For all vertices other than 1, initialize 1 to be its nearest vertex in Y. Initialize the distance from Y to the weight on the edge from to 1 to *i*

// continued on next slide

Prim Pseudocode

```
// continued from previous slide

repeat (n - 1 times)
        int min = ∞
        for (i = 2; i <= n; i++)
                if (0 <= distance[i] < min)// if distance[i] is -1, i is already in Y
                        min = distance[i];
                        vnear = i;
    e = edge connecting vnear and nearest[vnear]
    add e to F;
    distance[vnear] = -1;
    for (i = 2; i <= n; i++)// see if any vertices in V - Y are closer to vnear
        if (W[i][vnear] < distance[i])
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
```

Prim's Algorithm Initialization

nearest:

distance:

min:

vnear:

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

$F = \{\}$

Prim's Algorithm Initialization

nearest: { -1, 1, 1, 1, 1}

distance: {-1, 1, 3, ∞, ∞}   i.e. (Y = {v_1})

min:

vnear:

- Setting a `distance` value to -1 indicates that that vertex is in the partial spanning tree, Y.
- Every vertex in V - Y is closest to v_1 in Y (because v_1 is the only vertex currently in Y!)
  - In the `distance` array, we set each index to contain the distance from v_1 to the corresponding vertex.

F = {}

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim's Algorithm First Step

nearest: { -1, 1, 1, 1, 1}

distance: {-1, 1, 3, ∞, ∞} (Y = {v_1})

min: `distance[2]` which is 1.

vnear: 2

- The smallest value in `distance` is 1 in index 2.
- `vnear` indicates the vertex in V - Y closest to Y. We set it to 2

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

F = {}

Prim's Algorithm First Step

nearest: { -1, 1, 1, 1, 1}
distance: {-1, 1, 3, ∞, ∞} $(Y = \{v_1\})$

min: `distance[2]` which is 1.
vnear: 2

- Add the edge (nearest[vnear], vnear) to F, which connects the following vertices:
  - nearest[vnear] = a vertex in Y
  - vnear = a vertex in V - Y
- set distance[vnear] to -1 to indicate vnear is now in Y

F = {}

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim’s Algorithm First Step

nearest: { -1, 1, 1, 1, 1}
distance: {-1, **-1**, 3, ∞, ∞} (Y = {v_1, v_2})

min: `distance[2]` which is 1.
vnear: 2

- Add the edge (nearest[vnear], vnear) to F, which connects the following vertices:
  - nearest[vnear] = a vertex in Y
  - vnear = a vertex in V - Y
- set distance[vnear] to -1 to indicate vnear is now in Y

F = { **(1, 2)** }

Anything else we need to do?

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim's Algorithm First Step

nearest: { -1, 1, 1, **2**, 1}
distance: {-1, -1, 3, **6**, ∞} ($Y = \{v_1, v_2\}$)

min: `distance[2]` which is 1.
vnear: 2

- See if any vertices *i* in V - Y are closer to vnear than nearest[*i*].
- i.e. are any vertices *i* not in the partial spanning tree closer to the vertex we just added than distance[*i*]?

4 is 6 away from 2. Update nearest[4] to 2 and distance[4] to 6

F = { (1, 2) }

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim's Algorithm Second Step

nearest: { -1, 1, 1, 2, 1}

distance: {-1, -1, 3, 6, ∞} ($Y = \{v_1, v_2\}$)

min:

vnear:

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

F = { (1, 2) }

Prim's Algorithm Second Step

nearest: { -1, 1, 1, 2, 1}

distance: {-1, -1, 3, 6, ∞} (Y = {v_1, v_2})

min: distance[3] which is 3

vnear: 3

- Add (nearest[vnear], vnear) to F
- Set distance[vnear] to -1

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

F = { (1, 2) }

Prim's Algorithm Second Step

nearest: { -1, 1, 1, 2, 1}

distance: {-1, -1, **-1**, 6, ∞}   (Y = $\{v_1, v_2, v_3\}$)

min: distance[3] which is 3

vnear: 3

- Add (nearest[vnear], vnear) to F
- Set distance[vnear] to -1

Any remaining vertices *i* in V - Y closer to *vnear* than nearest[*i*]?

F = { (1, 2), **(1, 3)** }

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim's Algorithm Second Step

nearest: { -1, 1, 1, **3**, **3**}

distance: {-1, -1, -1, **4**, **2**} ($Y = \{v_1, v_2, v_3\}$)

min: distance[3] which is 3

vnear: 3

- 4 is 4 away from 3
- 5 is 2 away from 3

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

F = { (1, 2), (1, 3) }

Prim's Algorithm Third Step

nearest: { -1, 1, 1, 3, 3}

distance: {-1, -1, -1, 4, 2} (Y = {v_1, v_2, v_3})

min:

vnear:

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

F = { (1, 2), (1, 3) }

Prim's Algorithm Third Step

nearest: { -1, 1, 1, 3, 3}

distance: {-1, -1, -1, 4, 2} (Y = {v_1, v_2, v_3})

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

min: distance[5] which is 2

vnear: 5

- Add (nearest[vnear], vnear) to F
- Set distance[vnear] to -1

F = { (1, 2), (1, 3) }

Prim's Algorithm Third Step

nearest: { -1, 1, 1, 3, 3}
distance: {-1, -1, -1, 4, **-1**} $(Y = \{v_1, v_2, v_3, v_5\})$

min: distance[5] which is 2
vnear: 5

- Add (nearest[vnear], vnear) to F
- Set distance[vnear] to -1

Any remaining vertices *i* in V - Y closer to *vnear* than nearest[*i*]?

F = { (1, 2), (1, 3), **(3, 5)** }

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim's Algorithm Third Step

nearest: { -1, 1, 1, 3, 3}
distance: {-1, -1, -1, 4, -1}

($Y = \{v_1, v_2, v_3, v_5\}$)

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

min: distance[5] which is 2
vnear: 5

- Add (nearest[vnear], vnear) to F
- Set distance[vnear] to -1

Any remaining vertices *i* in V - Y closer to *vnear* than nearest[*i*]? **No!**

F = { (1, 2), (1, 3), (3, 5) }

Prim's Algorithm Fourth Step

nearest: { -1, 1, 1, 3, 3}

distance: {-1, -1, -1, 4, -1}

(Y = {v_1, v_2, v_3, v_5})

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

min:

vnear:

F = { (1, 2), (1, 3), (3, 5) }

Prim's Algorithm Fourth Step

nearest: { -1, 1, 1, 3, 3}

distance: {-1, -1, -1, 4, -1}

(Y = $\{v_1, v_2, v_3, v_5\}$)

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

min: distance[4] which is 4

vnear: 4

- Add (nearest[vnear], vnear) to F.
- Set distance[vnear] to -1

F = { (1, 2), (1, 3), (3, 5) }

Prim's Algorithm Fourth Step

nearest: { -1, 1, 1, 3, 3}

distance: {-1, -1, -1, **-1**, -1}

($Y = \{v_1, v_2, v_3, v_5\}$)

min: distance[4] which is 4

vnear: 4

- Add (nearest[vnear], vnear) to F.
- Set distance[vnear] to -1

Done!

F = { (1, 2), (1, 3), (3, 5), **(3, 4)** }

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

Prim's Algorithm Analysis

```
repeat (n - 1 times)
          int min = ∞
          for (i = 2; i <= n; i++)
                    if (0 <= distance[i] < min)   // if distance[i] is -1, i is already in
Y
                              min = distance[i];
                              vnear = i;
     e = edge connecting vnear and nearest[vnear]
     add e to F;
     distance[vnear] = -1;
     for (i = 2; i <= n; i++) // see if any vertices in V - Y are closer to vnear
          if (W[i][vnear] < distance[i])
                    distance[i] = W[i][vnear];
                    nearest[i] = vnear;
```

What is the basic operation?

Prim’s Algorithm Analysis

```
repeat (n - 1 times)
          int min = ∞
          for (i = 2; i <= n; i++)
                    if (0 <= distance[i] < min)   // if distance[i] is -1, i is already in
Y
                              min = distance[i];
                              vnear = i;
     e = edge connecting vnear and nearest[vnear]
     add e to F;
     distance[vnear] = -1;
     for (i = 2; i <= n; i++) // see if any vertices in V - Y are closer to vnear
          if (W[i][vnear] < distance[i])
                    distance[i] = W[i][vnear];
                    nearest[i] = vnear;
```

What is the basic operation? There are two: the **if** statements in each loop.

How many times do they occur?

Prim’s Algorithm Analysis

```
repeat (n - 1 times)
          int min = ∞
          for (i = 2; i <= n; i++)
                    if (0 <= distance[i] < min)   // if distance[i] is -1, i is already in Y
                              min = distance[i];
                              vnear = i;
     e = edge connecting vnear and nearest[vnear]
     add e to F;
     distance[vnear] = -1;
     for (i = 2; i <= n; i++) // see if any vertices in V - Y are closer to vnear
          if (W[i][vnear] < distance[i])
                    distance[i] = W[i][vnear];
                    nearest[i] = vnear;
```

What is the basic operation? There are two: the **if** statements in each loop.

How many times do they occur? $(n - 1)$ times each, plus $(n - 1)$ for the outer loop:

$2(n - 1)(n - 1) \in \Theta(n^2)$

In-Class Exercise

1. Use Prim's Algorithm to find a minimum spanning tree for the following graph. Show the values in `nearest`, `distance`, and `F` for each step.

$$\begin{array}{c|c} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{bmatrix} 0 & 10 & \infty & 30 & 45 & \infty \\ 10 & 0 & 50 & \infty & 40 & 25 \\ \infty & 50 & 0 & \infty & 35 & 15 \\ 30 & \infty & \infty & 0 & \infty & 20 \\ 45 & 40 & 35 & \infty & 0 & 55 \\ \infty & 25 & 15 & 20 & 55 & 0 \end{bmatrix} \end{array}$$