Lecture 11: Chapter 4 Part 3

The Greedy Approach

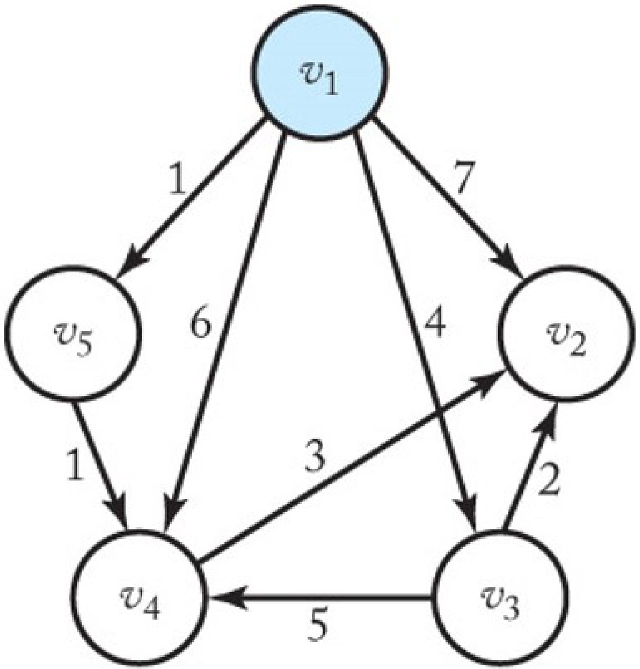
CS3310

Dijkstra's Algorithm

Prim's and Kruskal's algorithms determine a minimum spanning tree in a weighted, undirected graph.

Dijkstra's algorithm works with a weighted, **directed graph**.

- In a directed graph, arrows indicate in which direction an edge can be taken.
  - i.e. in this graph, v_1 to v_5 is a valid direction, but v_5 to v_1 is not.
- We use angle brackets to indicate that an edge is directed: $<v_1, v_5>$



Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that solves what is known as the *Single-Source Shortest Paths* problem.

- The goal is to determine the shortest paths from one particular vertex in a graph to every other vertex in that graph.
  - i.e. the shortest path from v_1 to v_2, the shortest path from v_1 to v_3 , etc
- We initialize a set Y to contain only the vertex whose shortest paths we are determining.
- An empty set F will eventually contain all the edges needed to connect v_1 to every other vertex with the least amount of weight.
- In our examples, we will always solve the paths from v_1, but in practice any vertex can be chosen.

Dijkstra's Algorithm

- After initialization, select the vertex v nearest to v_1 and add it to Y.
  - The edge $<v_1, v>$ is therefore the shortest path from v_1 to v: add it to F.

- Next, find the shortest path from v_1 to any vertex in V - Y, using only the other vertices in Y as possible intermediaries.
  - i.e. if $Y = \{v_1, v_2\}$ and $V - Y = \{v_3, v_4, v_5\}$ which paths do we need to check?

Dijkstra’s Algorithm

- After initialization, select the vertex v nearest to v_1 and add it to Y.
  - The edge $<v_1, v>$ is therefore the shortest path from v_1 to v: add it to F.

- Next, find the shortest path from v_1 to any vertex in V - Y, using only the other vertices in Y as possible intermediaries.
  - i.e. if $Y = \{v_1, v_2\}$ and $V - Y = \{v_3, v_4, v_5\}$ which paths do we need to check?
  - Any of the following paths that exist in the graph:
    $<v_1, v_3>, <v_1, v_4>, <v_1, v_5>, <v_1, v_2, v_3>, <v_1, v_2, v_4>, <v_1, v_2, v_5>$.

- Add the vertex at the end of shortest path to Y. Add the path’s final edge to F.
  - If the shortest path is $<v_1, v_3>$, add: ??
  - If the shortest path is $<v_1, v_2, v_4>$, add: ??

Dijkstra's Algorithm

- After initialization, select the vertex v nearest to v_1 and add it to Y.
  - The edge $<v_1, v>$ is therefore the shortest path from v_1 to v: add it to F.

- Next, find the shortest path from v_1 to any vertex in V - Y, using only the other vertices in Y as possible intermediaries.
  - i.e. if $Y = \{v_1, v_2\}$ and $V - Y = \{v_3, v_4, v_5\}$ which paths do we need to check?
  - Any of the following paths that exist in the graph:
    $<v_1, v_3>$, $<v_1, v_4>$, $<v_1, v_5>$, $<v_1, v_2, v_3>$, $<v_1, v_2, v_4>$, $<v_1, v_2, v_5>$.

- Add the vertex at the end of shortest path to Y. Add the path's final edge to F.
  - If the shortest path is $<v_1, v_3>$, add: $<v_1, v_3>$ to F and v_3 to Y.
  - If the shortest path is $<v_1, v_2, v_4>$, add: $<v_2, v_4>$ to F and v_4 to Y.

Dijkstra's Algorithm

Problem: Determine the shortest paths from a vertex (v_1 in this example) to all other vertices in a weighted, directed graph.

Inputs: integer $n \geq 2$, and a connected, weighted, directed graph containing n vertices. The graph is represented by a two-dimensional array W, with its rows and columns indexed from 1 to n, where W[i][j] is the weight on the edge from the ith vertex to the jth vertex.

Outputs: set of edges F containing edges in shortest paths.

Dijkstra’s Algorithm High-Level

```
Y = { v₁ }
F = {}
while (the instance is not solved)                    // selection procedure and
        select a vertex v from V - Y, with a          // feasibility check
        shortest path from v₁, using only vertices
        in Y as possible intermediaries.
        add the vertex v to Y
        add the edge (on the shortest path) that touches v to F)
        if (Y == V)
                the instance is solved                // Solution Check
```

Dijkstra's Algorithm Initialization

Suppose we want to find the shortest paths of v_1

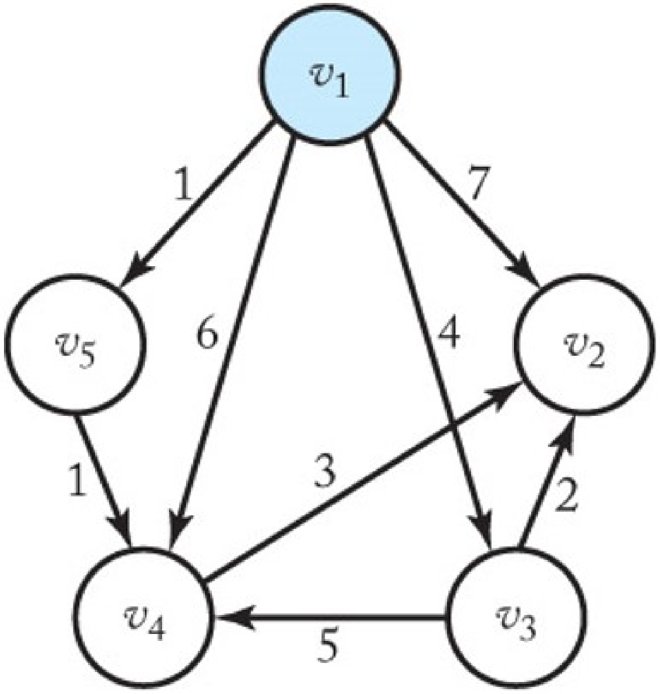
$Y = \{ v_1 \}$

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

$F = \{\}$

- Initialize Y to contain v_1
- Initialize F to the empty set.

Y only contains v_1, so there are no potential intermediaries yet. For the first step, simply find the vertex in the graph nearest to v_1



Dijkstra's Algorithm Step One

$Y = \{ v_1, v_5 \}$

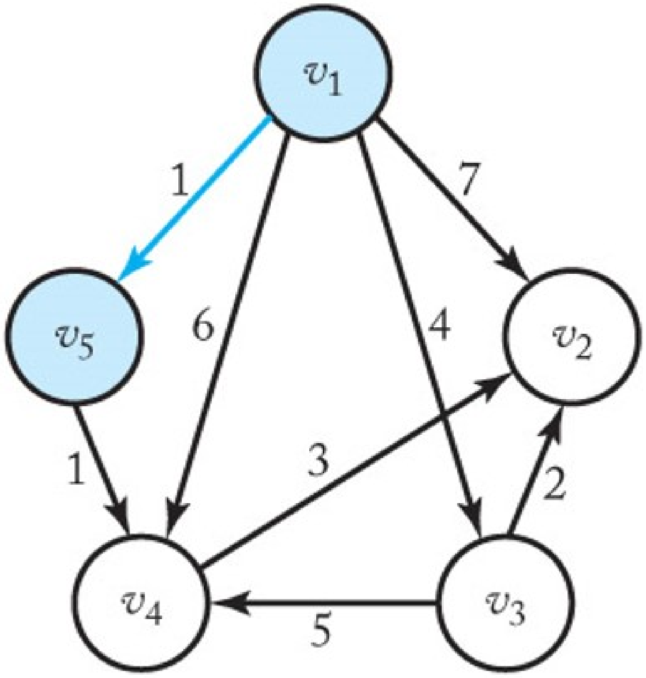
$V - Y = \{ v_2, v_3, v_4 \}$

$F = \{ <v_1, v_5> \}$

v_5 is the nearest vertex to v_1.

- v_5 is added to Y
- $<v_1, v_5>$ is added to F

We next choose the vertex in V - Y that is closest to v_1, either by direct connection or by passing through v_5 first.



Dijkstra's Algorithm Step Two

$Y = \{ v_1, v_4, v_5 \}$

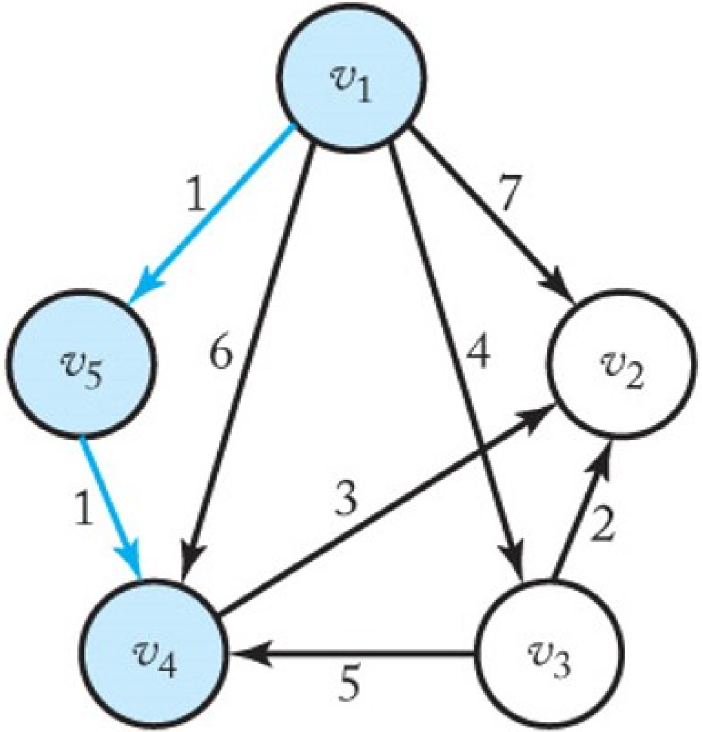
$V - Y = \{ v_2, v_3 \}$

$F = \{ \langle v_1, v_5 \rangle, \langle v_5, v_4 \rangle \}$

v_4 is the closest vertex to v_1 when passing through v_5 with a total weight of 2.

- v_4 is added to Y
- $\langle v_5, v_4 \rangle$ is added to F, since v_5 is the vertex in Y that *touches* v_4.

We next choose the vertex in V - Y that is closest to v_1, either by direct connection or by passing through v_5 or v_4 (or both)



Dijkstra's Algorithm Step Three

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

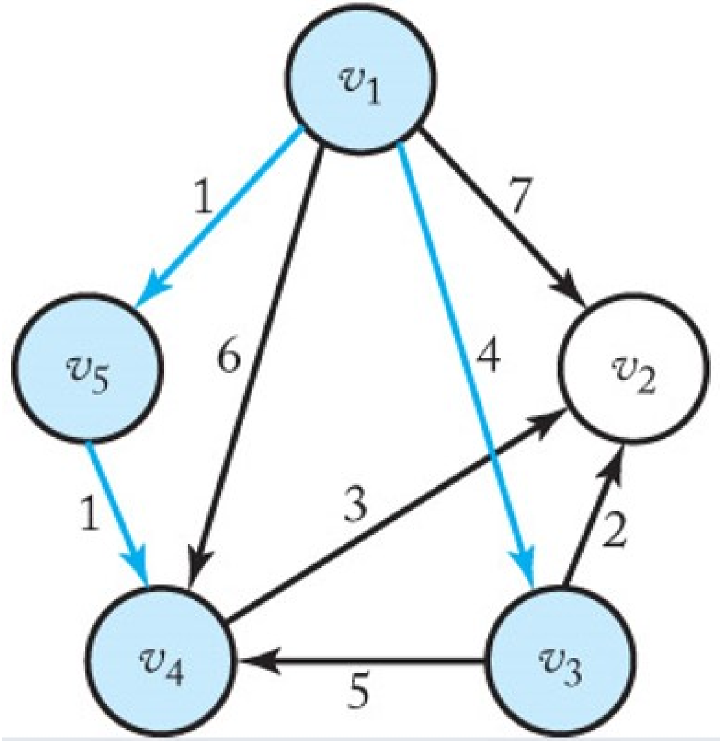
$V - Y = \{ v_2 \}$

$F = \{ <v_1, v_5>, <v_1, v_3>, <v_5, v_4> \}$

v_3 is the nearest vertex to v_1 by direct connection

- v_3 is added to Y
- $<v_1, v_3>$ is added to F, since v_1 is the vertex in Y that *touches* v_3.

We finally find the shortest path from v_1 to v_2



Dijkstra's Algorithm Step Four

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

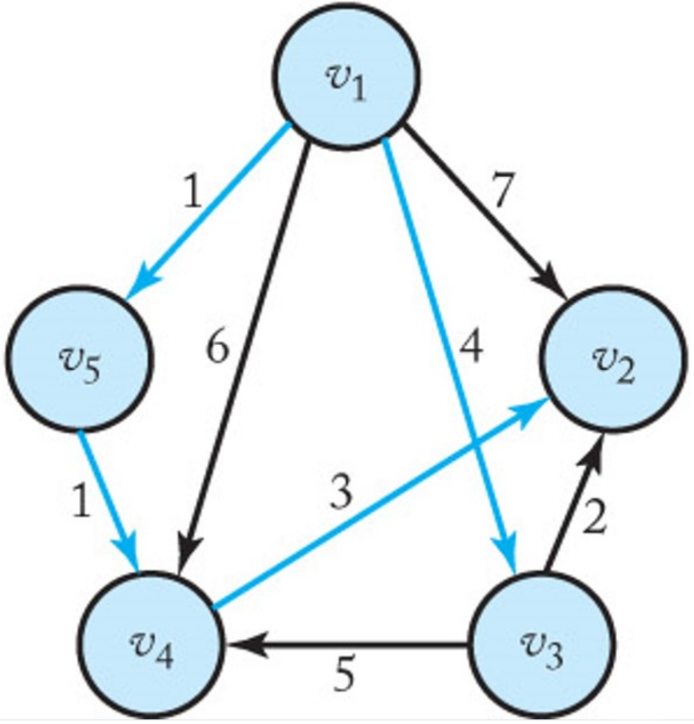
$V - Y = \{\}$

$F = \{ <v_1, v_5>, <v_1, v_3>, <v_5, v_4>, <v_4, v_2> \}$

The shortest path between v_1 and v_2 is $[v_1, v_5, v_4, v_2]$

- v_2 is added to Y
- $<v_4, v_2>$ is added to F.

We are finished!



Dijkstra's Algorithm

- This high-level algorithm only works for a human solving a small graph.

For the detailed algorithm, we keep two arrays, `touch` and `length`:

- `touch[i]` = index of the vertex v in Y such that the edge <v, v_i> is the last edge on the current shortest path from v_1 to v_i, using only vertices in Y as intermediaries.
  - i.e. If `touch[2]` = 4 then <v_4, v_2> is the last edge on the current shortest path from v_1 to v_2
- `length[i]` = length of the current shortest path from v_1 to v_i using only vertices in Y as intermediaries.

Dijkstra's Algorithm

```
void dijkstra (int n, const number W[][], set_of_edges& F)
        index i, vnear;
        edge e;
        index touch[2..n];
        index length[2..n];

        F = {}
        for( i = 2; i <= n; i++)// For each vertex vi , initialize v1 to be
                touch[i] = 1;    the final vertex on the shortest path from Y
                length[i] = W[1][i]; to vi , and initialize the length of that
                                 path to be the weight on the edge from v1 to
                                 vi
```

Dijkstra's Algorithm cont'd

```
repeat (n - 1 times)                          // Add all n - 1 vertices to Y
        min = ∞;
        for( i = 2; i <= n; i++)     // see which vertex is on current shortest
path
                if ( 0 <= length[i] < min)
                        min = length[i];
                        vnear = i;                    // vnear is closest vertex
to those in Y
        e = edge from touch[vnear] to vnear. Add to F;

    // for each vertex in V - Y, update shortest path from v1 if necessary
    for ( i = 2; i <= n; i++)
        if ( length[vnear] + W[vnear][i] < length[i])
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;
    length[vnear] = -1;                      // Add vnear to Y
```

Dijkstra's Algorithm Initialization

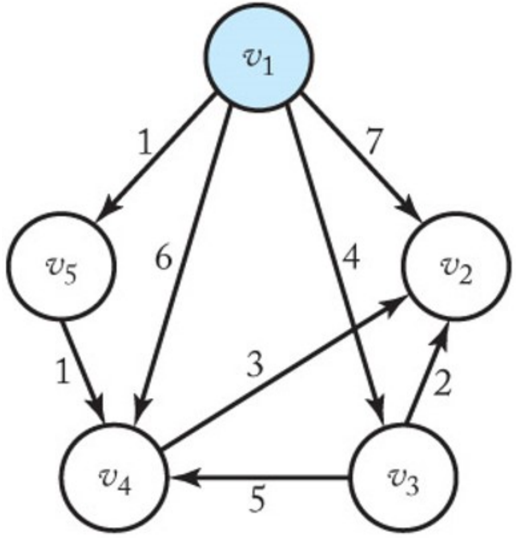
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { ? }

Y = { ? }

F = {}



`length` = { ? }	`touch` = { ? }
`vnear` =	e =

F = {}

Dijkstra's Algorithm Initialization

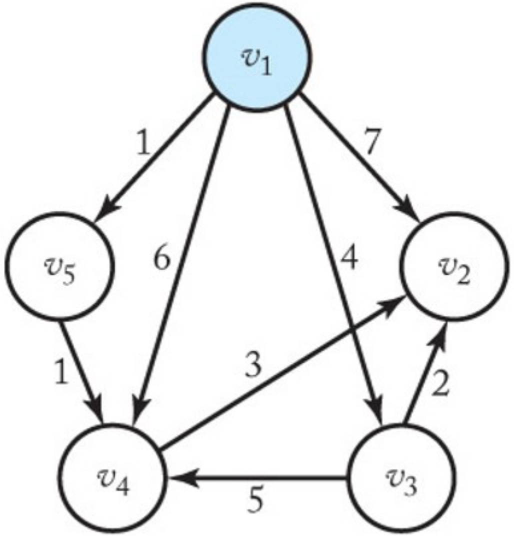
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3, 4, 5 }

Y = { 1 }

F = {}



`length` = { -1, 7, 4, 6, 1 }	`touch` = { 0, 1, 1, 1, 1 }
`vnear` =	*e* =

F = {}

Dijkstra's Algorithm Step 1

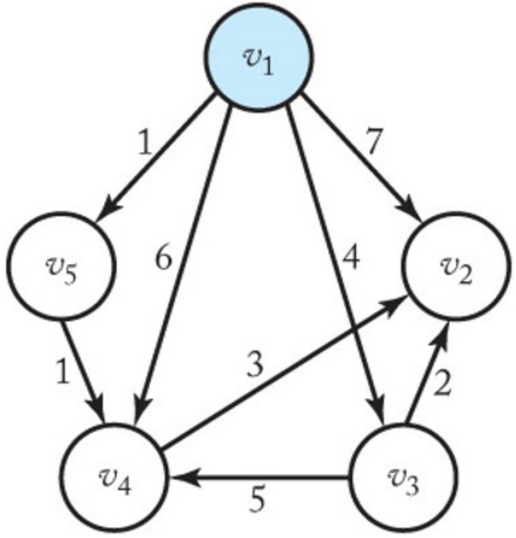
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3, 4, 5 }

Y = { 1 }

F = {}



`length` = { -1, 7, 4, 6, **1** }	`touch` = { 0, 1, 1, 1, **1** }
`vnear` = 5	e = <1, 5>

- v_5 is the closest vertex in V - Y to Y with a weight of 1.
- The vertex it touches in Y is v_1

Dijkstra’s Algorithm Step 1

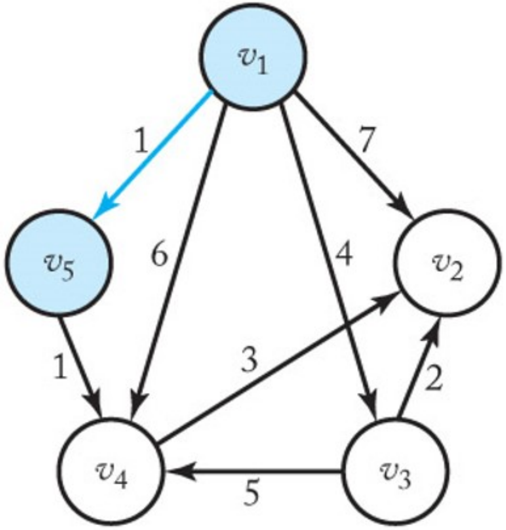
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3, 4 }

Y = { 1, 5 }

F = { <1, 5> }



`length` = { -1, 7, 4, 6, 1 }	`touch` = { 0, 1, 1, 1, 1 }
`vnear` = 5	*e* = <1, 5>

- Add *e* to F and update `length` and `touch`.
- `length[vnear] + W[vnear][4] = 1 + 1 = 2`, which is less than `length[4] = 6`.
    - i.e. v_4 is only 2 away from v_1 using v_5 as an intermediary.

Dijkstra's Algorithm Step 1

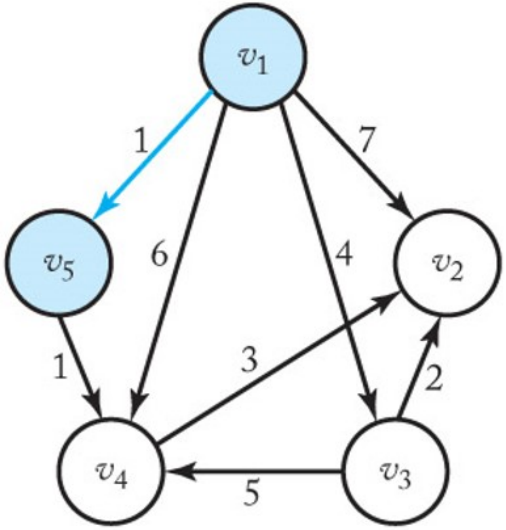
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3, 4 }

Y = { 1, 5 }

F = { <1, 5> }



`length` = { -1, 7, 4, **2**, **-1** }	`touch` = { 0, 1, 1, **5**, 1 }
`vnear` = 5	*e* = <1, 5>

- Set `length[4]` to 2 indicating the current shortest path from v_1 to v_4 has a weight of 2.
- Set `touch[4]` to 5 indicating v_5 is the vertex in Y that v_4 touches on this shortest path
- Set `length[vnear]` to -1 to add `vnear` to Y

Dijkstra's Algorithm Step 2

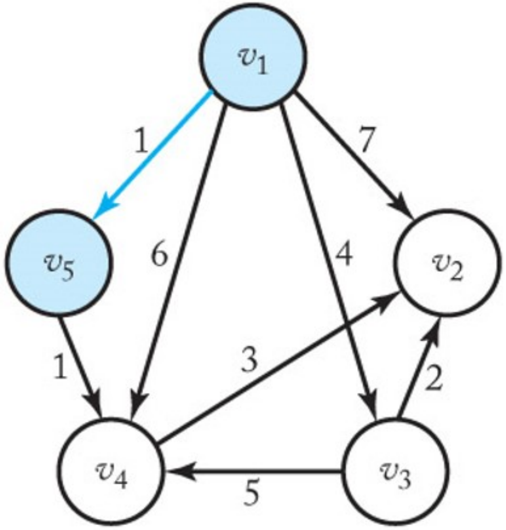
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3, 4 }

Y = { 1, 5 }

F = { <1, 5> }



`length` = { -1, 7, 4, 2, -1 }	`touch` = { 0, 1, 1, 5, 1 }
`vnear` = **4**	e = **<5, 4>**

- v_4 is the closest vertex in V - Y to Y.
- The vertex it touches in Y is v_5.

Dijkstra's Algorithm Step 2

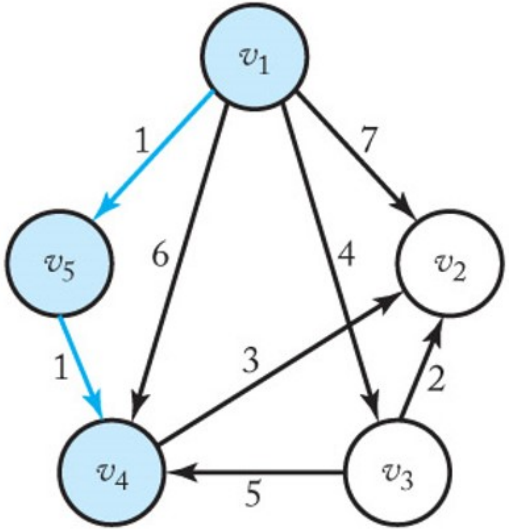
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3 }

Y = { 1, 4, 5 }

F = { <1, 5>, <5, 4> }



`length` = { -1, 7, 4, 2, -1 }	`touch` = { 0, 1, 1, 5, 1 }
`vnear` = 4	e = <5, 4>

- Add e to F and see if we need to update `length` and `touch`.
- `length[2]` = `7` and `length[vnear]` + `W[vnear][2]` = `2 + 3 = 5`.
  - i.e. v_2 is closer to v_1 via v_4 with a weight of 5 than it is to v_1 directly.

Dijkstra's Algorithm Step 2

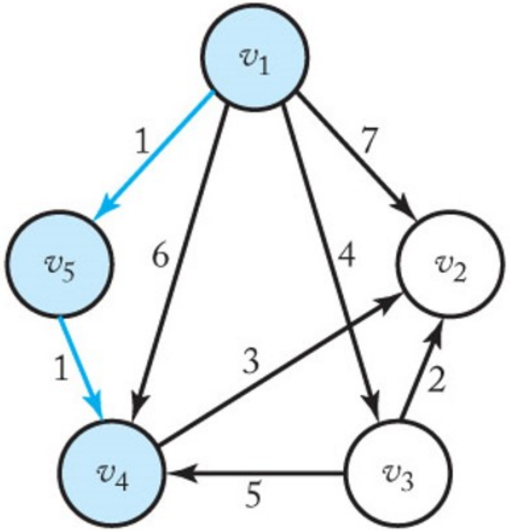
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3 }

Y = { 1, 4, 5 }

F = { <1, 5>, <5, 4> }



`length` = { -1, **5**, 4, **-1**, -1 }	`touch` = { 0, **4**, 1, 5, 1 }
`vnear` = 4	*e* = <5, 4>

- v_2 is closer to v_1 via v_4 with a weight of 5 than it is to v_1 directly.
  - Update `touch[2]` to 4 and `length[2]` to 5
  - Add `vnear` to Y by setting its `length` to -1

Dijkstra's Algorithm Step 3

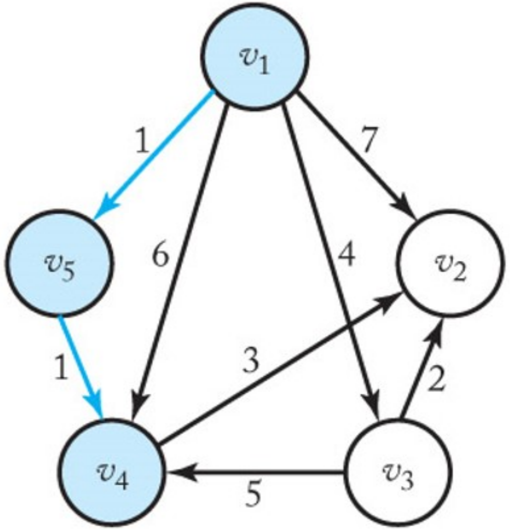
$W =$

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2, 3 }

Y = { 1, 4, 5 }

F = { <1, 5>, <5, 4> }



`length` = { -1, 5, 4, -1, -1 }	`touch` = { 0, 4, 1, 5, 1 }
`vnear` = **3**	e = **<1, 3>**

- v_3 is the closest vertex in V - Y to Y with a weight of 4.
- The vertex it touches in Y is v_1.

Dijkstra's Algorithm Step 3

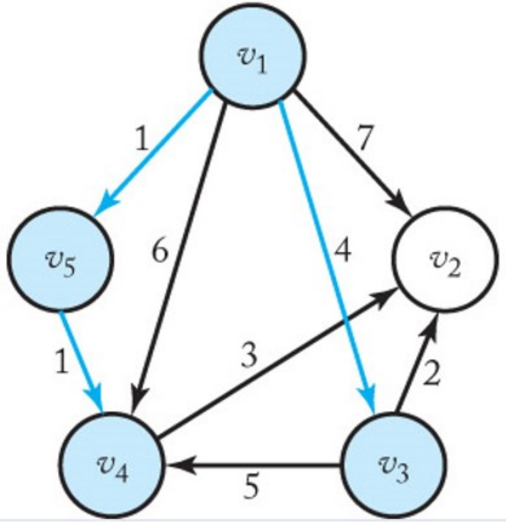
W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2 }

Y = { 1, 3, 4, 5 }

F = { <1, 5>, <5, 4>, <1, 3> }



`length` = { -1, 5, **-1**, -1, -1 }	`touch` = { 0, 4, 1, 5, 1 }
`vnear` = 3	*e* = <1, 3>

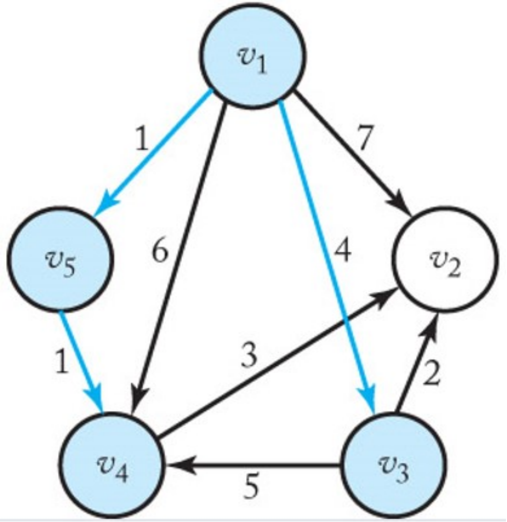
- Add *e* to F and see if we need to update `length` and `touch`.
  - No updates needed!
  - Add `vnear` to Y by setting `length` to -1

Dijkstra’s Algorithm Step 4

W =

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

V - Y = { 2 }
Y = { 1, 3, 4, 5 }
F = { <1, 5>, <5, 4>, <1, 3>, <4, 2> }



`length` = { -1, 5, -1, -1, -1 }	`touch` = { 0, 4, 1, 5, 1 }
`vnear` = **2**	*e* = **<4, 2>**

- v_2 is the closest vertex in V - Y to Y with a weight of 5.
  - The vertex it touches in Y is v_4.
  - Add *e* to F and we’re done!

0-1 Knapsack Problem

The 0-1 Knapsack problem is famous and has multiple solutions or partial solutions.

- Today, we will discuss the greedy approach to this problem. Later in the semester we will look at dynamic programming and backtracking solutions.

The Problem:

- Imagine that a thief breaks into a jewelry store with a knapsack. The knapsack will break if the total weight of the items stolen exceeds some maximum weight W.
- Each item has a value and a weight.
- How do we maximize the total value of the items taken while not breaking the bag?

0-1 Knapsack Problem

S = $\{item_1, item_2, \ldots, item_n\}$, the list of items to be stolen

w_i = weight of $item_i$

p_i = profit of $item_i$

W = maximum weight the knapsack can hold

A = bag of stolen items.

- The brute-force solution is to consider all subsets of the n items, discard the ones that exceed W, and take one of the remaining subsets with maximum total profit.
- What is the Order of this solution?

0-1 Knapsack Problem

S = {$item_1$, $item_2$, …, $item_n$ }, the list of items to be stolen

w_i = weight of $item_i$

p_i = profit of $item_i$

W = maximum weight the knapsack can hold

A = bag of stolen items.

- The brute-force solution is to consider all subsets of the n items, discard the ones that exceed W, and take one of the remaining subsets with maximum total profit.
- What is the Order of this solution? $\mathbf{\Theta(2^n)}$

0-1 Knapsack Problem

What is a greedy solution to this problem?

1st idea: steal the items with the largest profit first.

What is wrong with this solution?

0-1 Knapsack Problem

What is a greedy solution to this problem?

1st idea: steal the items with the largest profit first.

Suppose we have three items and W = 30:

Item	Weight	Profit
a	25	$10
b	10	$9
c	10	$9

We would take item *a* and not be able to take *b* or *c*, but taking *b* and *c* is the optimal solution!

0-1 Knapsack Problem

What is a greedy solution to this problem?

2nd idea: steal the items with the smallest weight first.

Suppose we have three items and W = 30:

Item	Weight	Profit
a	5	$1
b	10	$2
c	20	$10

We would take item *a* and *b* and not be able to take or *c*, but taking *b* and *c* is the optimal solution!

0-1 Knapsack Problem

What is a greedy solution to this problem?

3rd idea: Steal the items with the largest profit-per-unit weight first. We order the items in nonincreasing order according to profit-per-unit weight, and select them in sequence.

- An item is placed in the knapsack if its weight does not bring the total weight above W.

Is there a counterexample for this strategy?

0-1 Knapsack Problem

3rd idea: Steal the items with the largest profit-per-unit weight first. We order the items in nonincreasing order according to profit-per-unit weight, and select them in sequence.

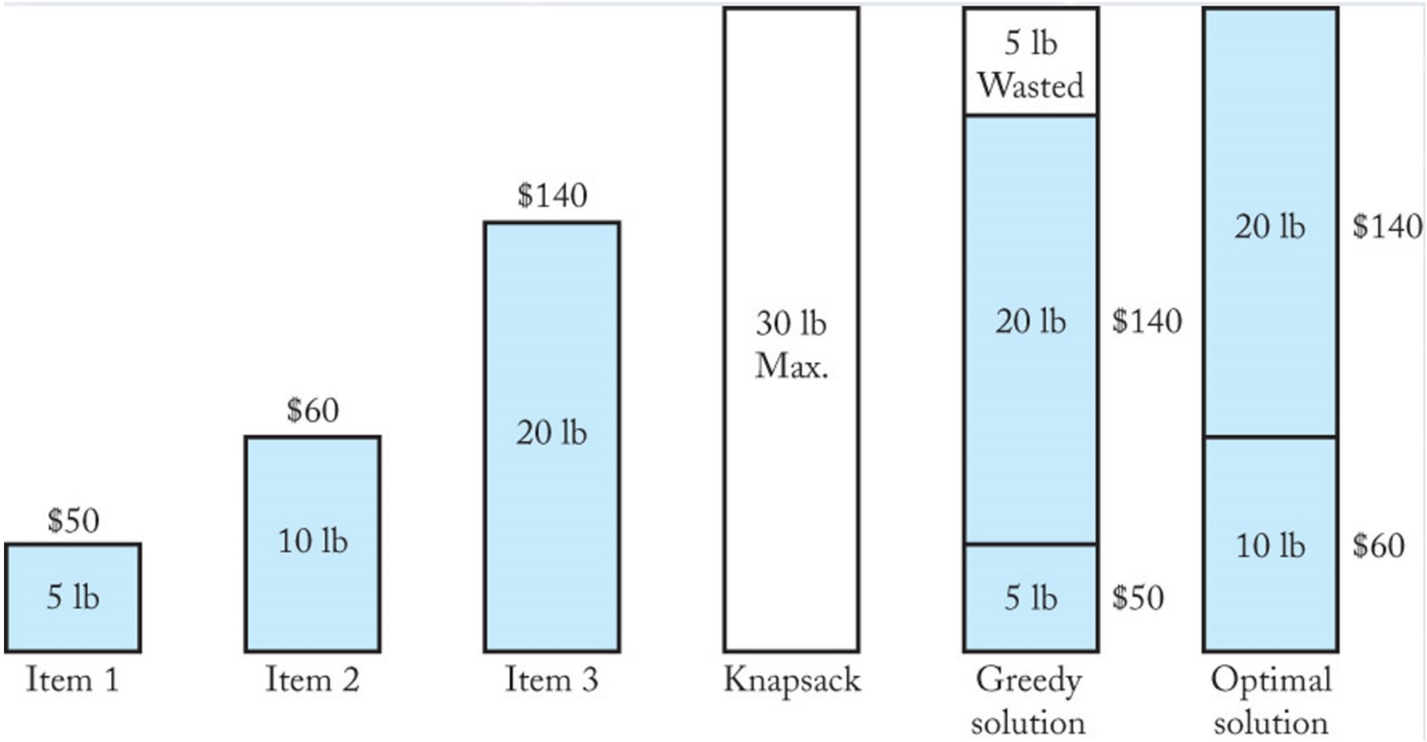
- An item is placed in the knapsack if its weight does not bring the total weight above W.

Imagine we have the following items, and W = 30:

- $item_1$: \$50 / 5 = \$10 profit-per-unit weight
- $item_2$: \$60 / 10 = \$6 profit-per-unit weight
- $item_3$: \$140 / 20 = \$7 profit-per-unit weight

0-1 Knapsack Problem

Our Greedy Solution is better, but still not optimal!



The Fractional Knapsack Problem

- Now imagine the thief does not have to steal all of an item, but can take any fraction of an item.
  - i.e. bags of gold dust
- Instead of having 5 pounds leftover in the bag, we would simply take 5/10($60) of item 2, adding $30 to the total, giving us $220!

In-Class Exercise

1. Use Dijkstra's Algorithm to find the shortest path from v_5 to all the other vertices in the following graph. Show the values in F, `length`, and `touch` after each step

	1	2	3	4	5	6
1	0	∞	1	5	9	2
2	∞	0	3	2	5	7
3	1	3	0	∞	15	9
4	5	2	∞	0	2	3
5	9	8	15	2	0	8
6	2	7	9	3	8	0