Chapter 4. (Part 1)

The Greedy Approach

경북대학교 배준현 교수

(joonion@knu.ac.kr)

Contents

The Greedy Approach

- Greedy Algorithm
  - arrives at a solution by making *a sequence of choices*,
    - each of which simply *looks* *the best at the moment*.
  - That is, each choice is *locally optimal*.
  - The hope is that a *globally optimal* solution will be obtained,
    - but this is ***not always*** the case.
  - For a given (greedy) algorithm,
    - we *must determine* whether the (greedy) solution is *always optimal*.

The Greedy Approach

- The problem of *giving change* for a purchase:
  - Our goal is to give the correct change with *as few coins as possible*.
  - A ***greedy approach*** to the problem:
    - Initially, there are no coins in the change.
    - *(selection procedure)* Look for the largest coin (in value) you can find.
    - *(feasibility check)* If the total change does not exceed the amount owed,
      - add the coin to the change
    - *(solution check)* Check if the change is now equal to the amount owed.
    - If the values are not equal, *repeat the process until*
      - the value of the change equals the amount owed,
      - or there is no coins left.

The Greedy Approach

- High-level algorithm for the greedy approach:

```
while (there are more coins and the instance is not solved) {
    grab the largest remaining coin;    // selection procedure
    if (adding the coin makes the change exceed the amount owed)
        reject the coin;                // feasibility check
    else
        add the coin to the change;
    if (the total value of the change equals the amount owed)
        the instance is solved;         // solution check
}
```

- An example:
  - coins = [*quarter*, *dime*, *dime*, *nickel*, *penny*, *penny*] = [25, 10, 10, 5, 1, 1]
  - amount owed = 36 cents.
  - A greedy algorithm for giving change.
    - change = [25] < 36. Grab.
    - change = [25, 10] < 36. Grab.
    - change = [25, 10, ~~10~~] > 36. Reject.
    - change = [25, 10, ~~5~~] > 36. Reject.
    - change = [25, 10, 1] = 36. Grab and terminate.

The Greedy Approach

- Does it *always* result in an *optimal* solution?
  - Notice here that if we include a 12-cent coin with the U.S. coins,
    - the greedy algorithm *does not always* give an *optimal* solution.
  - coins = [12, 10, 5, 1, 1, 1, 1]
  - amount owed = 16 cents.
  - A greedy algorithm for giving change.
    - change = [12] < 16. Grab.
    - change = [12, ~~10~~] > 16. Reject.
    - change = [12, ~~5~~] > 16. Reject.
    - change = [12, 1, 1, 1, 1] = 16. Grab and terminate.
    - optimal change = [10, 5, 1]

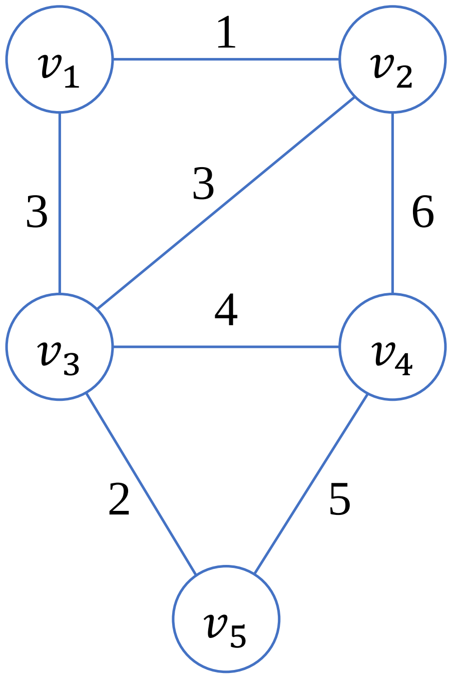
The Greedy Approach

- The ***Greedy Algorithm***
  - starts with an *empty set* and adds items to the set *in sequence*
    - until the set represents a solution to an instance of a problem.
  - Each iteration consists of three steps:
    1. *Selection Procedure*:
        - chooses the next item to add to the set.
    2. *Feasibility Check*:
        - determines if the new set if feasible.
    3. *Solution Check*:
        - determines whether the new set constitutes a solution.

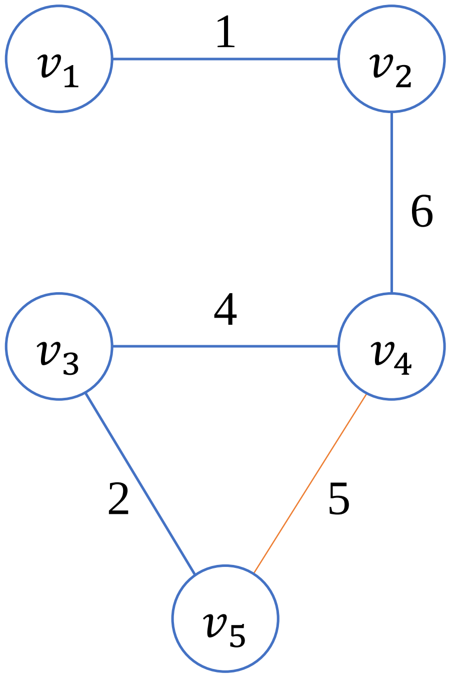
4.1 Minimum Spanning Trees

- *Minimum Spanning Tree* Problem:
  - The problem of removing edges
    - from a *connected*, *weighted*, *undirected* graph G
    - to form a *subgraph* such that *all the vertices* remains *connected*, and
    - the *sum of the weights* on the remaining edges is *as small as possible*.
  - A *spanning tree* for G is a connected subgraph
    - that contains all the vertices in G and is a tree.
  - A ***minimum spanning tree*** (MST) is
    - a spanning tree of *minimum weight*.

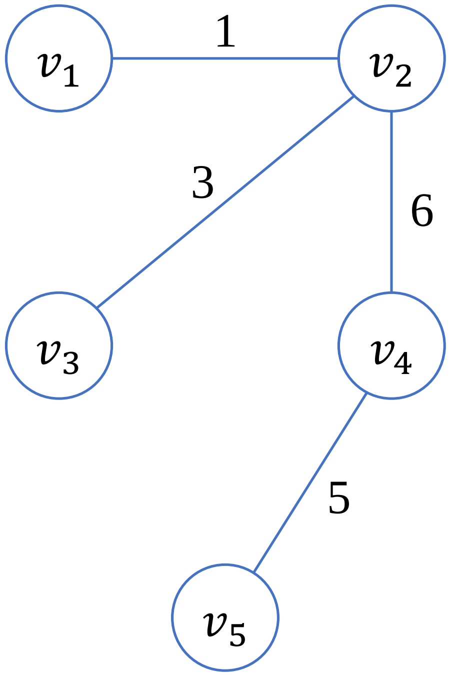
4.1 Minimum Spanning Trees



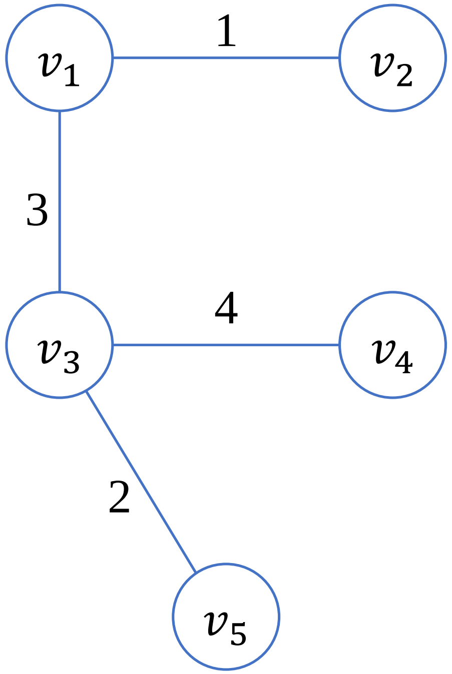
- A connected, weighted, undirected graph G.



- If (v_4, v_5) were removed, the graph would remain connected.



- A spanning tree for G.



- A minimum spanning tree for G.

4.1 Minimum Spanning Trees

- Formal Definition of the MST Problem:
  - Given a *connected*, *weighted*, *undirected* graph $G = (V, E)$.
  - A spanning tree T for G has the same vertices V as G,
    - but the set of edges of T is a subset F of E.
  - Denote a spanning tree by $T = (V, F)$.
  - Our problem is to find a subset F of E
    - such that $T = (V, F)$ is a *minimum spanning tree* for G.

The Greedy Approach

- High-level greedy algorithm for the MST problem

```
F = ∅;
while (the instance is not solved) {
    select an edge according to some locally optimal consideration;
    if (adding the edge to F does not create a cycle)
        add the edge to the solution;
    else
        discard the edge;
    if (T = (V, F) is a spanning tree)
        the instance is solved;
}
```

- ***Prim's Algorithm***
  - starts with an *empty set* of edges F
    - and a *subset of vertices* Y initialized to contain an *arbitrary* vertex (v_1).
  - 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*.
  - The *vertex* that is *nearest* to Y is added to Y
    - and the *edge* is added to F. (Ties are broken arbitrarily)
  - This process of adding nearest vertices is
    - repeated until $Y = V$.

- High-level pseudo-code for the Prim's algorithm

$F = \emptyset;$

$Y = \{v_1\};$

while (*the instance is not solved*) {

  select a vertex in $V - Y$ that is *nearest to* Y;

  add the vertex to Y;

  add the edge to F;

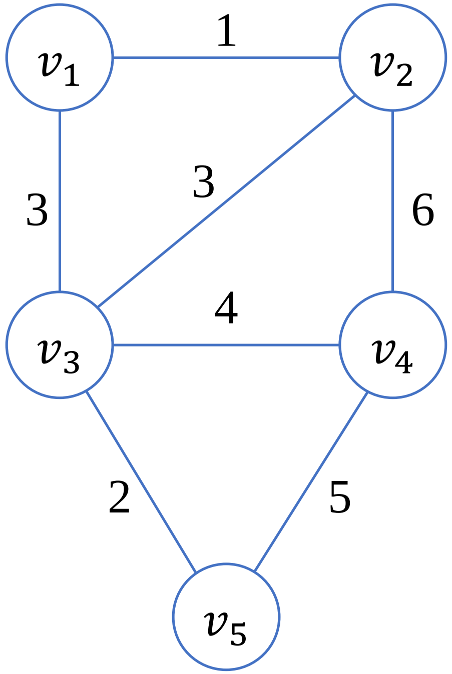
  if ($Y = V$)

    the instance is solved;

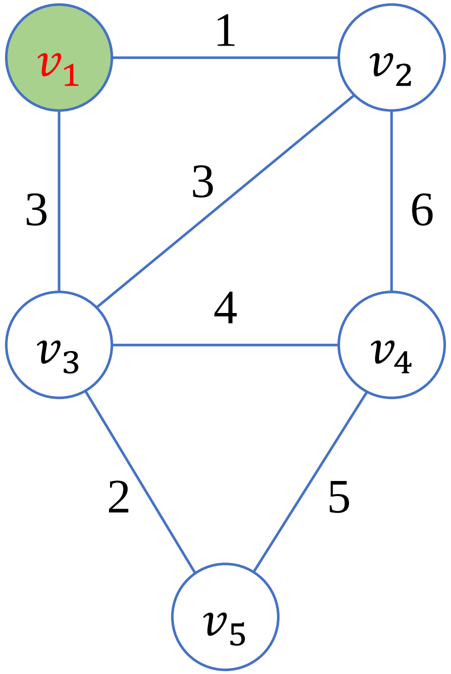
}

4.1 Minimum Spanning Trees

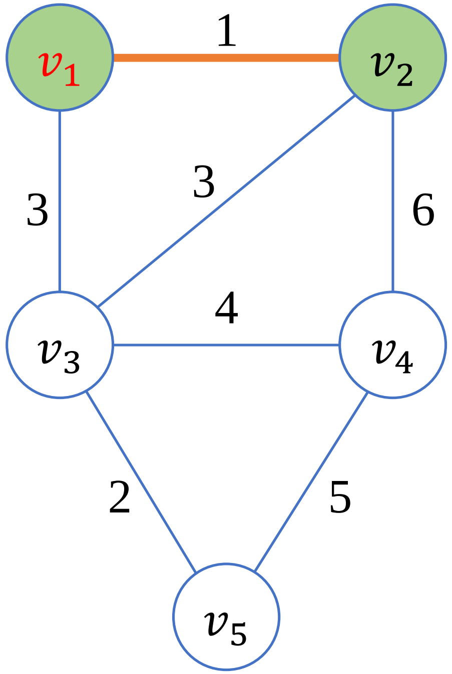
- Determine a minimum spanning tree



1. Vertex v_1 is selected first

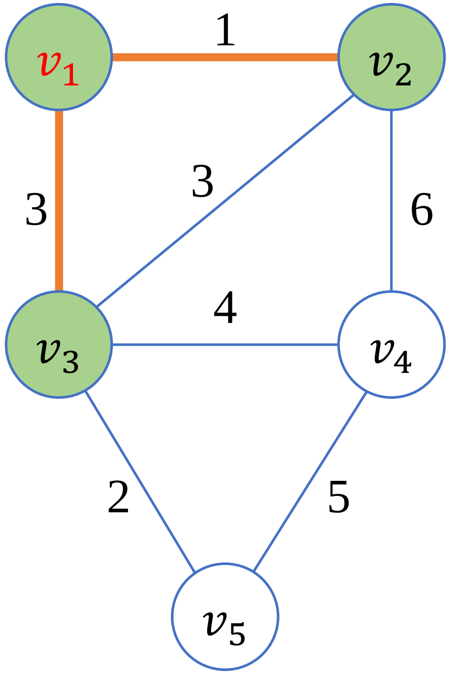


2. Vertex v_2 is selected because it is nearest to $\{v_1\}$

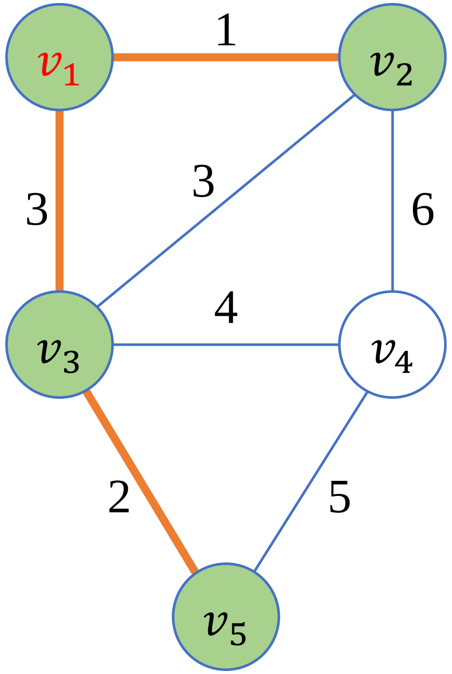


4.1 Minimum Spanning Trees

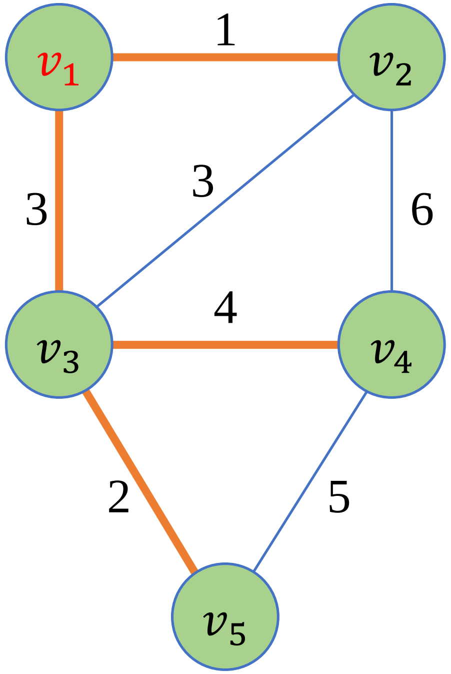
3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$



4. Vertex v_5 is selected because it is nearest to $\{v_1, v_2, v_3\}$



5. Vertex v_4 is selected because it is nearest to $\{v_1, v_2, v_3, v_5\}$



4.1 Minimum Spanning Trees

- Implementing the Prim's algorithm:
  - Represent a *weighted* graph by its *adjacency matrix*.
    - $W[i][j] = \begin{cases} weight\ on\ edge & \text{if there is an edge between } v_i \text{ and } v_j \\ \infty & \text{if there is no edge between } v_i \text{ and } v_j \\ 0 & \text{if } i = j \end{cases}$
  - We maintain two arrays, *nearest* and *distance*, where, for $i = 2, \ldots, n$,
    - $nearest[i] =$ index of the vertex in Y *nearest* to v_i
    - $distance[i] =$ weight on edge between v_i and the vertex *indexed* by $nearest[i]$

4.1 Minimum Spanning Trees

ALGORITHM 4.1: Prim's Algorithm

```
void prim(int n, matrix_t& W, set_of_edges& F)
{
    int vnear, min;
    vector<int> nearest(n + 1), distance(n + 1);

    F.clear(); // F = Ø;
    for (int i = 2; i <= n; i++) {
        nearest[i] = 1;
        distance[i] = W[1][i];
    }
```

ALGORITHM 4.1: Prim's Algorithm (continued)

```
    repeat (n - 1 times) {
        min = ∞;
        for (int i = 2; i <= n; i++)
            if (0 <= distance[i] && distance[i] < min) {
                min = distance[i];
                vnear = i;
            }
        e = edge connecting vertices indexed by vnear and nearest[vnear];
        add e to F;
        distance[vnear] = -1;
        for (int i = 2; i <= n; i++)
            if (distance[i] > W[i][vnear]) {
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
            }
    }
}
```

4.1 Minimum Spanning Trees

```
#define INF 0xffff

typedef vector<vector<int>> matrix_t;
typedef vector<pair<int, int>> set_of_edges;
typedef pair<int, int> edge_t;


// e = edge connecting vertices indexed by vnear and nearest[vnear];
// add e to F;
F.push_back(make_pair(vnear, nearest[vnear]));


set_of_edges F;
prim(n, W, F);
for (edge_t e: F) {
    u = e.first; v = e.second;
    cout << u << " " << v << " " << W[u][v] << endl;
}
```

W	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

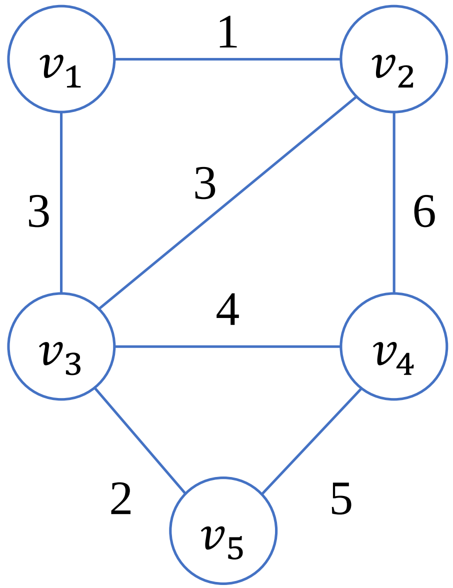
	i	2	3	4	5	*e*
init:	nearest[i]	1	1	1	1	
	distance[i]	1	3	∞	∞	
step 1:	nearest[i]	1	1	2	1	(2, 1, 1)
	distance[i]	-1	3	6	∞	
step 2:	nearest[i]	1	1	3	3	(3, 1, 3)
	distance[i]	-1	-1	4	2	
step 3:	nearest[i]	1	1	3	3	(5, 3, 2)
	distance[i]	-1	-1	4	-1	
step 4:	nearest[i]	1	1	3	3	(4, 3, 4)
	distance[i]	-1	-1	-1	-1	

4.1 Minimum Spanning Trees

- Time Complexity of Algorithm 4.1:
  - Basic Operation: the *instructions* inside each of two loops.
  - Input Size: n, the *number of vertices*.
  - Note that there are two (nested) loops,
    - and the *repeat* loop has n – 1 iterations.
  - Therefore,
    - $T(n) = 2(n-1)(n-1) \in \Theta(n^2)$

4.1 Minimum Spanning Trees

- Does it *always* produce an *optimal solution*?
  - We need to prove that
    - Prim's algorithm *always* produces a minimum spanning tree.
  - Given an undirected graph $G = (V, E)$,
    - A subset F and E is called *promising*
      - if edges can be added to it so as to *form* a *minimum spanning tree*.

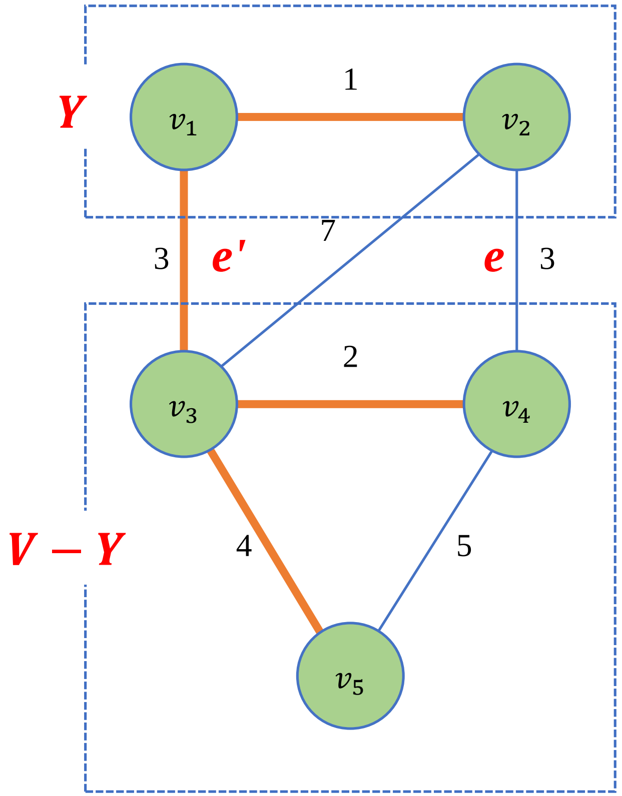


- The subset $\{(v_1, v_2), (v_1, v_3)\}$ is *promising*.
- The subset $\{(v_2, v_4)\}$ is *not promising*.

4.1 Minimum Spanning Trees

- Lemma:
  - If F is a *promising* subset of E
    - then $F \cup \{e\}$ is *promising*,
    - where e is *an edge of minimum weight* that
    - connects a vertex in Y and a vertex in $V - Y$.
  - Proof:
    - Let F' be a set edges in an MST s.t. $F \subseteq F'$.
    - If $e \in F'$, then $F \cup \{e\} \subseteq F'$.
    - If $e \notin F'$, then $F' \cup \{e\}$ must have a cycle.
    - There is an edge $e' \notin F'$ in the cycle
      - Remove e', then the cycle disappears.
      - Hence, $F' \cup \{e\} - \{e'\}$ is an MST.
    - Hence, $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$.

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



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

$F' \cup \{e\}$ has a cycle: $[v_1, v_2, v_4, v_3]$

- **Theorem:**
  - Prim's algorithm *always produces* a minimum spanning tree.
  - Proof:
    - Clearly, the *empty set* $\emptyset$ is *promising*.
    - Assume that, after a given iteration,
      - the selected edges F is *promising*.
    - The set $F \cup \{e\}$ is *promising*,
      - where e is the edge selected in the next iteration.
    - Because the e is an edge of minimum weight that
      - connects a vertex in Y to a vertex in $V - Y$. (by the Lemma)

▪ *Kruskal's Algorithm*

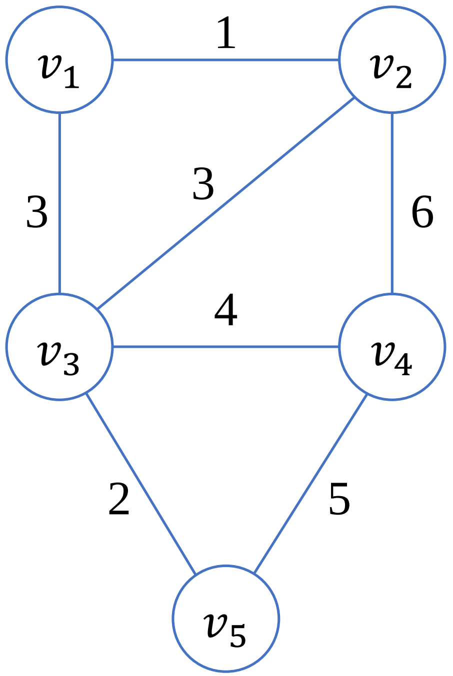
- starts by creating *disjoint subsets* of V,
  - one for *each vertex* and containing *only that vertex*.
- If then, inspects the edge according to nondecreasing weight
  - ties are broken arbitrarily.
- If an edge *connects* two vertices in *disjoint subsets*,
  - the edge is *added* and the subsets are *merged into one set*.
- This process is repeated
  - until *all the subsets* are *merged into one set*.

- High-level pseudo-code for the Kruskal's algorithm

```
F = ∅;
create disjoint subsets of V, one for each vertex and containing only that vertex;
sort the edges in E in nondecreasing order;
while (the instance is not solved) {
   select next edge;
   if (the edge connects two vertices in disjoint subsets) {
      merge the subsets;
      add the edge to F;
   }
   if (all the subsets are merged)
      the instance is solved;
}
```

4.1 Minimum Spanning Trees

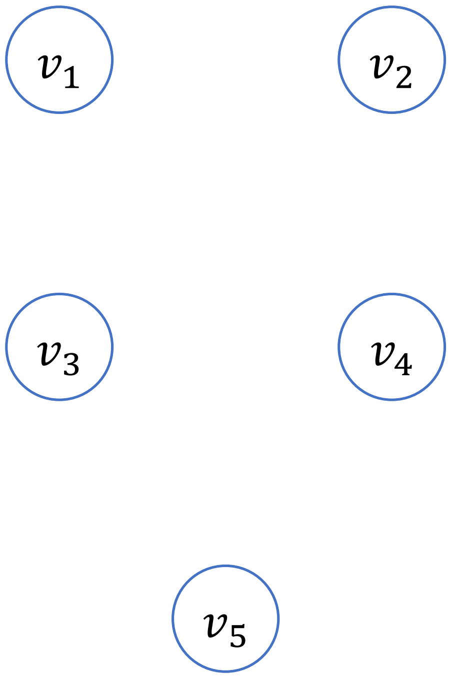
- Determine a minimum spanning tree.



1. Edges are sorted by their weights.

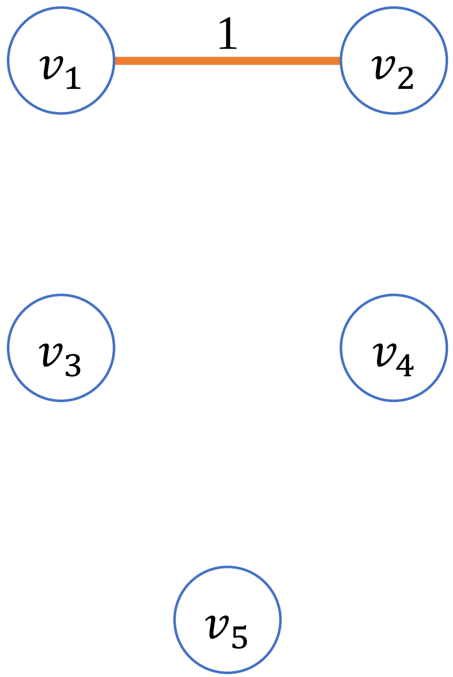
edges	*weight*
(v_1, v_2)	1
(v_3, v_5)	2
(v_1, v_3)	3
(v_2, v_3)	3
(v_3, v_4)	4
(v_4, v_5)	5
(v_2, v_4)	6

2. Disjoint sets are created.

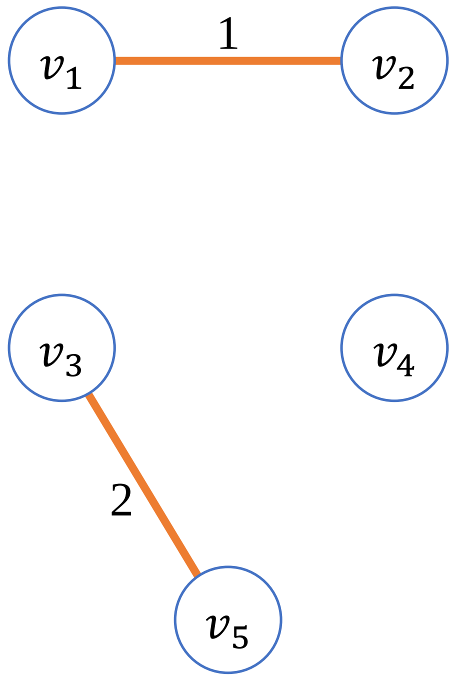


4.1 Minimum Spanning Trees

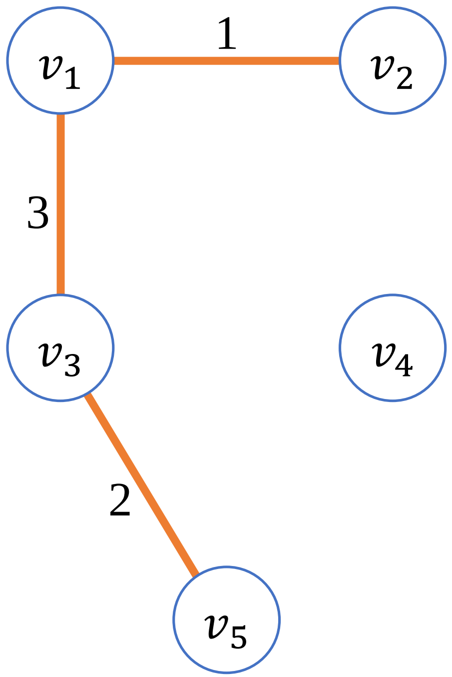
3. The first edge (v_1, v_2) is selected



4. Next edge (v_3, v_5) is selected

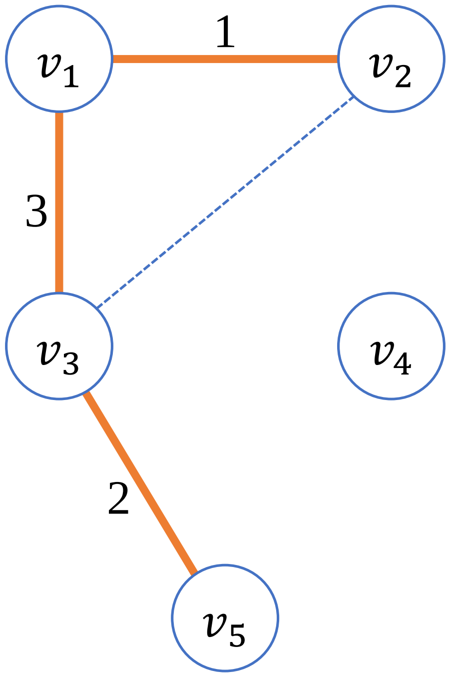


5. Next edge (v_1, v_3) is selected

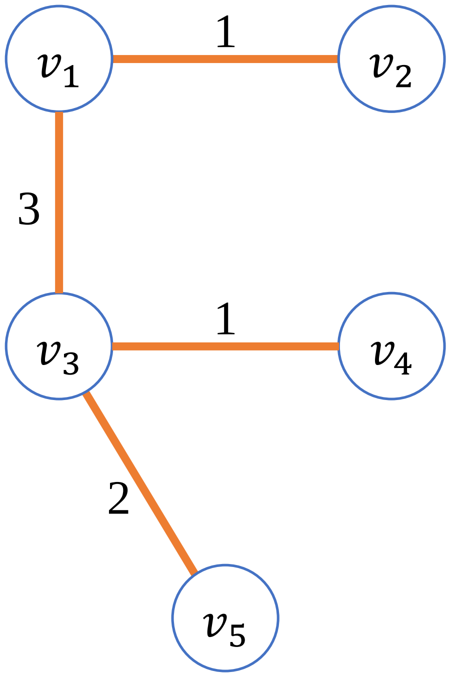


- Ties are broken *arbitrarily*

6. Next edge (v_2, v_3) is *discarded*, because it creates a cycle



7. Next edge (v_3, v_4) is selected



- (v_4, v_5) and (v_2, v_4) are *not considered*, because all the subsets are merged

- Abstract Data Type: *Disjoint Set*
  - To write a formal version of Kruskal's algorithm,
    - we need a *disjoint set* abstract data type: Refer to *Appendix C*.
  - The ADT of the disjoint set defines two data types:
    - $index\ i;$
    - $set\_pointer\ p, q;$
  - Then the routines are defined:
    - $initial(n)$: initialzes n disjoint subsets.
    - $p = find(i)$: makes p point to the set containing index i.
    - $merge(p, q)$: merges the two sets, to which p and q point, into the set.
    - $equal(p, q)$: returns true if p and q both point to the same set.

4.1 Minimum Spanning Trees

- Let $U = \{\, A, B, C, D, E \,\}$ be a universe of elements

```
for i in U:
    initial(i);
```

$\{A\}$ $\{B\}$ $\{C\}$ $\{D\}$ $\{E\}$ **(disjoint sets)**

```
p = find(B);
q = find(C);
```

$\{B\} \leftarrow p$, $\{C\} \leftarrow q$

```
merge(p, q);
```

$\{A\}$ $\{B, C\}$ $\{D\}$ $\{E\}$

```
p = find(C);
q = find(E);
```

$\{B, C\} \leftarrow p$, $\{E\} \leftarrow q$

`equal(C, E);` *returns false;*

```
merge(p, q);
```

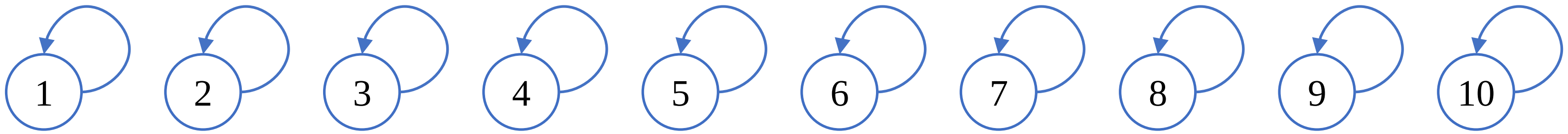
$\{A\}$ $\{B, C, E\}$ $\{D\}$

```
p = find(C);
q = find(E);
```

$\{B, C, E\} \leftarrow p$, q

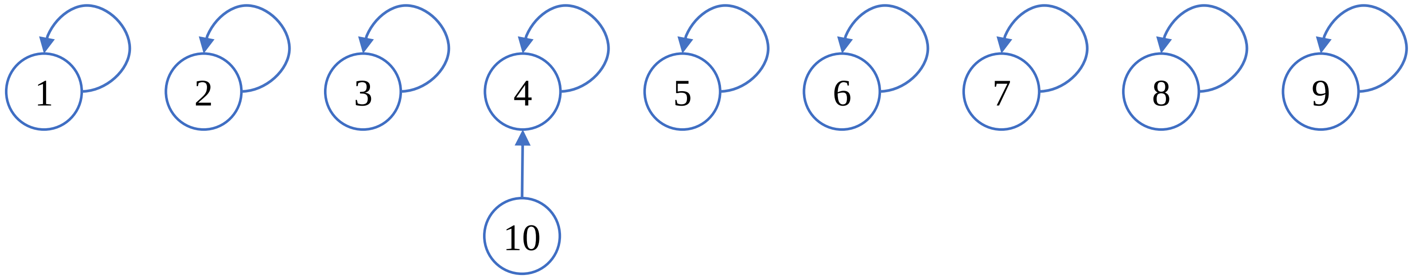
`equal(C, E);` *returns true;*

```
initial(10);
```



1	2	3	4	5	6	7	8	9	10
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

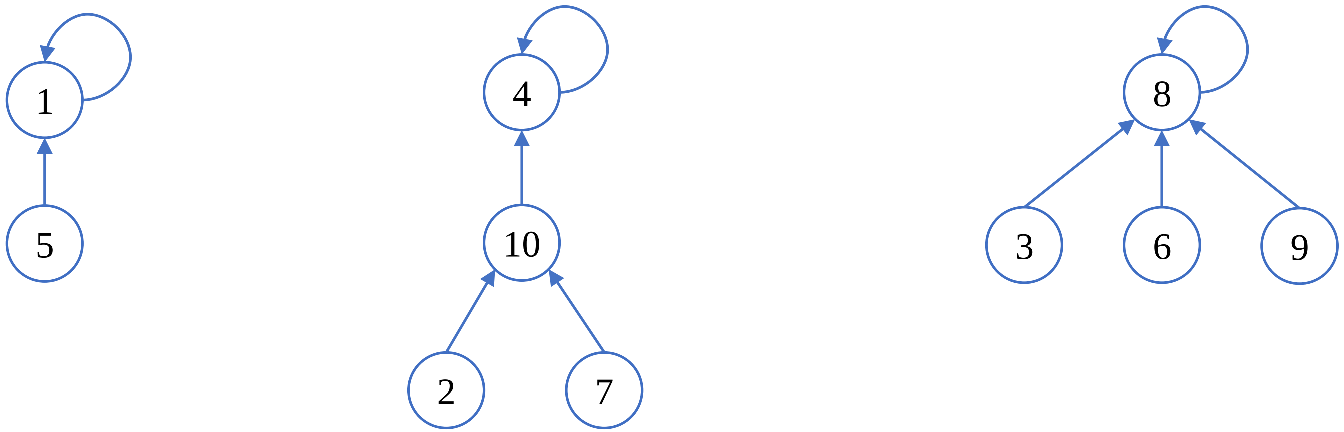
```
merge(find(4), find(10);
```



1	2	3	4	5	6	7	8	9	4
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

4.1 Minimum Spanning Trees

`After several union and find:`



$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$
1	10	8	**4**	1	8	10	**8**	8	4

- Analyze the complexity of ***union***(merge) and ***find***,
  - and improve the efficiency to $\Theta(m \lg m)$,
    - where m is the *number of passes* to call the routines($merge$ and $find$).

```
vector<int> dset;

void dset_init(int n) {
    dset.resize(n + 1);
    for (int i = 1; i <= n; i++)
        dset[i] = i;
}

int dset_find(int i) {
    while (dset[i] != i)
        i = dset[i];
    return i;
}

void dset_merge(int p, int q) {
    dset[p] = q;
}
```

4.1 Minimum Spanning Trees

ALGORITHM 4.2: Kruskal's Algorithm

```
void kruskal(int n, int m, set_of_edges& E, set_of_edges& F) {
    int p, q;
    edge_t e;
    PriorityQueue PQ;

    sort the m edges in E by weight in nondecreasing order;
    F.clear(); // F = Ø;
    dset_init(n);
    while (number of edges in F is less than n - 1) {
         e = PQ.top(); PQ.pop(); // edge with least weight not yet considered;
        p = dset_find(e.u);
        q = dset_find(e.v);
        if (p != q) {
            dset_merge(p, q);
            F.push_back(e); // add e to F
        }
    }
}
```

```
typedef struct edge {
    int u, v, w;
} edge_t;

struct edge_compare {
    bool operator()(edge_t e1, edge_t e2) {
        if (e1.w > e2.w) return true;
        else return false;
    }
};

typedef vector<edge_t> set_of_edges;
typedef priority_queue<edge_t, vector<edge_t>, edge_compare> PriorityQueue;


// sort the m edges in E by weight in nondecreasing order;
for (edge_t e: E)
    PQ.push(e);
```

4.1 Minimum Spanning Trees

- Time Complexity of Algorithm 4.2:
  - Basic Operation: a *comparison* instruction.
  - Input Size: n, the *number of vertices*, and m, the *number of edges*.
  - Three considerations in this algorithm:
    1. The time to sort the edges: $\Theta(m \lg m)$
    2. The time to initialize n disjoint sets: $\Theta(n)$.
    3. The time in the while loop: $\Theta(m \lg m)$
       - The time complexity of *Union-Find* (Appendix C)
  - Since $m \geq n-1$, $W(m, n) \in \Theta(m \lg m)$
  - In worst-case, the number of edges is $m = n(n-1)/2$
    - $w(m, n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 \lg n)$

4.1 Minimum Spanning Trees

- Comparing Prim's Algorithm with Kruskal's Algorithm:
  - The time complexity of two algorithms:
    - Prim's: $T(n) \in \Theta(n^2)$
    - Kruskal's: $W(m, n) \in \Theta(m \lg m) = \Theta(n^2 \lg n)$
  - We can show that $n - 1 \leq m \leq \frac{n(n-1)}{2}$.
  - For a *sparse graph*,
    - whose number of edges m is near the low end of these limits,
    - Kruskal's algorithm is $\Theta(n \lg n)$, which is *faster than Prim's*.
  - For a *dense graph*,
    - whose number of edges m is near the high end of those limits,
    - Kruskal's algorithm is $\Theta(n^2 \lg n)$, which is *slower than Prim's*.

4.2 Dijkstra's Algorithm for Single-Source Shortest Paths

- The Problem of *Single-Source-Shortest-Paths*
  - Find a shortest path from *one particular vertex* to *all the others*.
  - *Dijkstra's Algorithm* uses the *greedy approach*
    - to develop a $\Theta(n^2)$ algorithm for this problem.

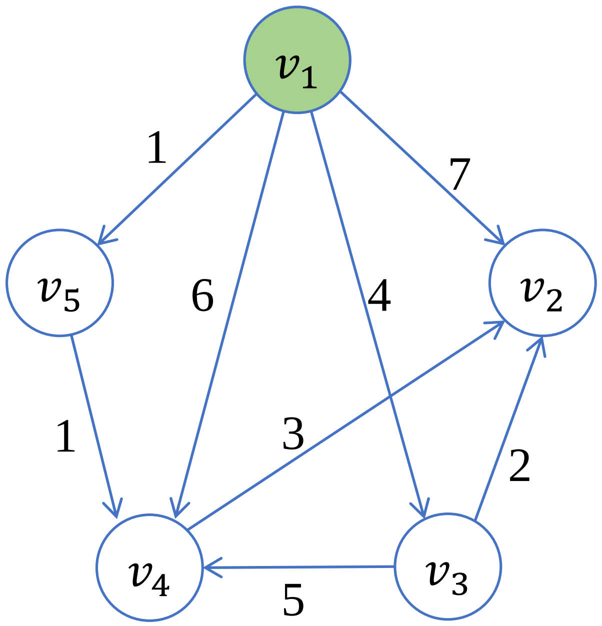
4.2 Dijkstra's Algorithm for Single-Source Shortest Paths

- Dijkstra's Algorithm
  - initializes a set Y to contain only the source vertex v_1,
    - and initializes a set F of edges to being empty.
  - First, choose a vertex v that is *nearest* to v_1,
    - add it to Y, and add the edge $\langle v_1, v \rangle$ (a *shortest edge*) to F.
  - Next, check the paths from v_1 to the vertices in $V - Y$
    - that allow only vertices in Y as intermediate vertices.
  - The vertex at the end of such a path is added to Y,
    - and the edge (on the path) that *touches* that vertex is added to F.
  - Continue this procedure, until Y equals V.
    - At this point, F contains the edges in *shortest paths*.

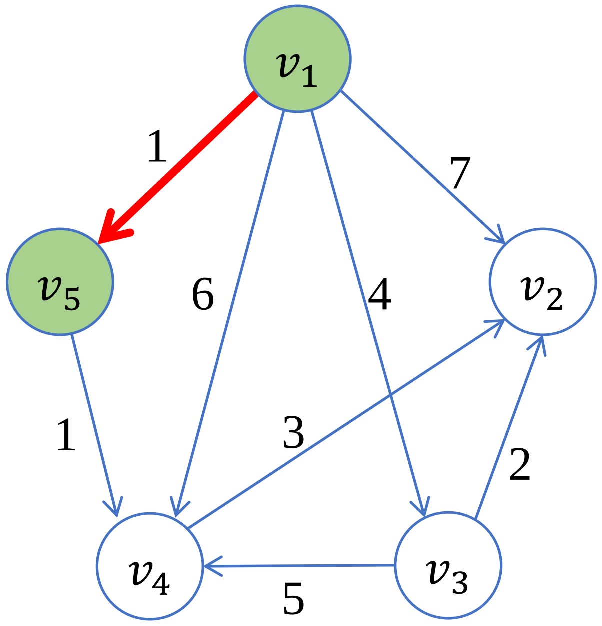
- High-level pseudo-code for the Dijkstra's algorithm:

```
Y = {v_1};
F = ∅;
while (the instance is not solved) {
    select a vertex v from V − Y that has a shortest path from v_1,
            using only vertices in Y as intermediates;
    add the new vertex v to Y;
    add the edge (on the shortest path) that touches v to F;
    if (Y = V)
        the instance is solved;
}
```

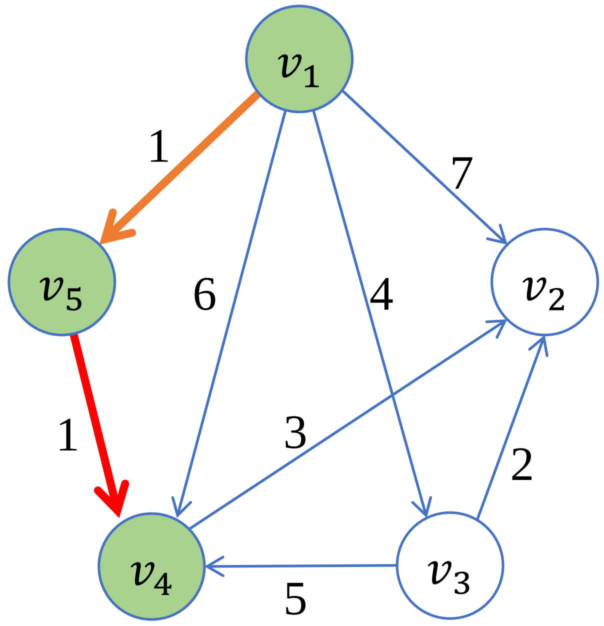
- Compute shortest paths from v_1.



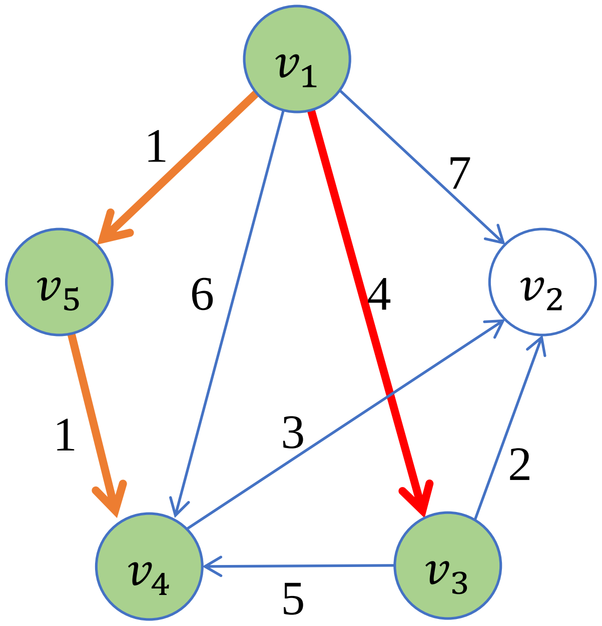
1. Vertex v_5 is selected because it is nearest to v_1.



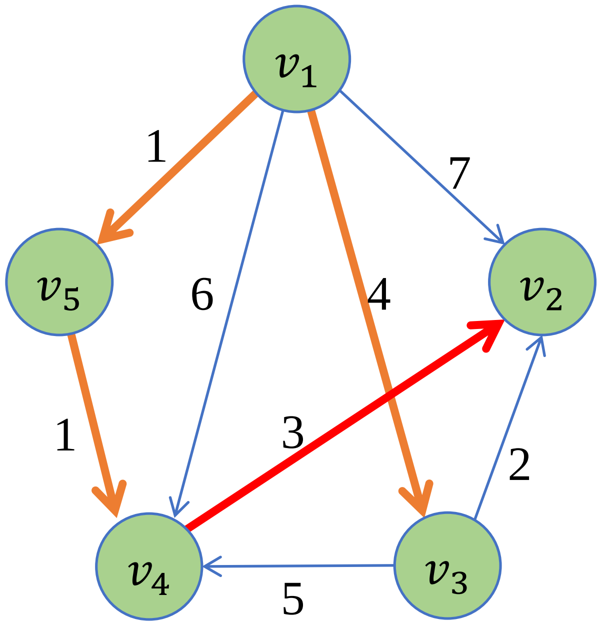
2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.



3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.



4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.



Implementation of Dijkstra's Algorithm

- It is very *similar* to *Prim's* Algorithm.
- The difference is that, instead of the arrays *nearest* and *distance*,
  - we have arrays *touch* and *length*, where for $i = 2, \cdots, n$.
- Let us define:
  - $touch[i] =$ *index* of *vertex* v in Y such that the edge $\langle v, v_i \rangle$ is the *last edge* on the *current shortest path* from v_1 to v_i using *only vertices in* Y as *intermediates*.
  - $length[i] =$ *length* of the *current shortest path* from v_1 to v_i using *only vertices in* Y as *intermediates*.

ALGORITHM 4.3: Dijkstra's Algorithm

```
void dijkstra(int n, matrix_t& W, set_of_edges& F)
{
    int vnear, min;
    vector<int> touch(n + 1), length(n + 1);

    F.clear();
    for (int i = 2; i <= n; i++) {
        touch[i] = 1;
        length[i] = W[1][i];
    }
```

ALGORITHM 4.3: Dijkstra's Algorithm (continued)

```
    repeat (n - 1 times) {
        min = INF;
        for (int i = 2; i <= n; i++)
            if (0 <= length[i] && length[i] < min) {
                min = length[i];
                vnear = i;
            }
        e = edge from vertex indexed by touch[vnear];
        add e to F;
        for (int i = 2; i <= n; i++)
            if (length[i] > length[vnear] + W[vnear][i]) {
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;
            }
        length[vnear] = -1;
    }
}
```

4.2 Dijkstra's Algorithm for Single-Source Shortest Paths

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

	i	2	3	4	5	*e*
init:	touch[i]	1	1	1	1	
	length[i]	7	4	6	1	
step 1:	touch[i]	1	1	5	1	(1, 5, 1)
	length[i]	7	4	2	-1	
step 2:	touch[i]	4	1	5	1	(5, 4, 1)
	length[i]	5	4	-1	-1	
step 3:	touch[i]	4	1	5	1	(1, 3, 4)
	length[i]	5	-1	-1	-1	
step 4:	touch[i]	4	1	5	1	(4, 2, 3)
	length[i]	-1	-1	-1	-1	

- The Lengths of Shortest Paths:
  - Algorithm 4.3 determines only the edges in the shortest paths.
    - It does not produce *the lengths of those paths*.
  - These lengths could be obtained from the edges.
    - Alternatively, they can be computed and stored in an array as well.

- Time Complexity of Algorithm 4.3
  - is the same with that of Algorithm 4.1 (Prim's Algorithm)
  - $T(n) = 2(n-1)^2 \in \Theta(n^2)$

Any Questions?