بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

مسئلهٔ سکه ها

Our goal is not only to give the correct change, but to do so with as few coins as possible. Our criterion for deciding which coin is best (locally optimal) is the value of the coin.

```
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)                      // feasibility check
     reject the coin;
  else
     add the coin to the change;
  if (the total value of the change equals the
               amount owed)                           // solution check
     the instance is solved;
}
```

Again, the algorithm is called "greedy" because the selection procedure simply consists of greedily grabbing the next-largest coin without considering the potential drawbacks of making such a choice. There is no opportunity to reconsider a choice. Once a coin is accepted, it is permanently included in the solution; once a coin is rejected, it is permanently excluded from the solution. This procedure is very simple, but does it result in an optimal solution?

The widely used coin denominations in the United States are 25 cents (quarter), 10 cents (dime), 5 cents (nickel), and 1 cent (penny)

Amount owed: 36 cents

Step	Total Change
1. Grab quarter	
2. Grab first dime	
3. Reject second dime	
4. Reject nickel	
5. Grab penny	

Amount owed: 16 cents

Step	Total Change
1. Grab 12-cent coin	
2. Reject dime	
3. Reject nickel	
4. Grab four pennies	

Optimal : $16 = 10 + 5 + 1$.

☞ *As this Change problem shows, a greedy algorithm does not guarantee an optimal solution. We must always determine whether this is the case for a particular greedy algorithm.*
☞ *A selection procedure chooses the next item to add to the set. The selection is performed according to a greedy criterion that satisfies some locally optimal consideration at the time.*
☞ *A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.*

☞ *Greedy algorithms do not always yield optimal solutions, but for many problems they do.*

☞ *A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution but sometimes it does.*

☞ *We must prove that a greedy choice at each step yields a globally optimal solution.*

On each step—and this is the central point of this technique—the choice made must be:

* *feasible, i.e., it has to satisfy the problem's constraints*
* *locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step*
* *irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm*

These requirements explain the technique's name: on each step, it suggests a "greedy" grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

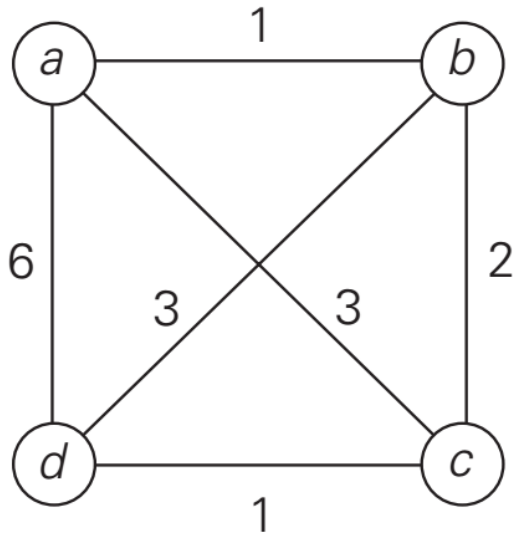
* *A selection procedure chooses the next item to add to the set. The selection is performed according to a greedy criterion that satisfies some locally optimal consideration at the time.*

* *A feasibility check determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.*

* *A solution check determines whether the new set constitutes a solution to the instance.*

Greedy Algorithms for the TSP

The following well-known greedy algorithm is based on the nearest-neighbor heuristic: always go next to the nearest unvisited city.



For the instance represented by the graph in the above figure, with a as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) $s_a : a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$ of length 10.

The optimal solution, as can be easily checked by exhaustive search, is the tour $s : a \to b \to d \to c \to a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

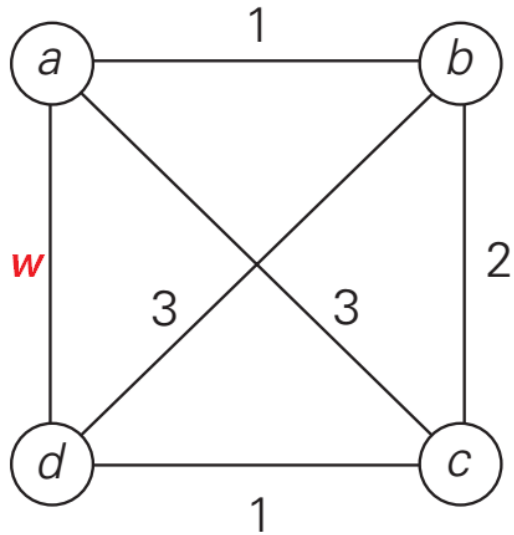
(i.e., tour s_a is 25% longer than the optimal tour s).

☞ s_a: *An approximate solution to the problem*

☞ s^*: *An exact solution to the problem*

توجه کنید که اگر مسئله کمینه سازی باشد (مثل مسئلهٔ فروشندهٔ دوره گرد)، آنگاه $r(s_a)$ (یعنی همان نسبت تقریب) برابر با $\frac{f(s_a)}{f(s^*)}$ تعریف می شود (که بزرگتر از یک است) و هرچه به ۱ نزدیکتر باشد، s_a بهتر است. اما اگر مسئله بیشینه سازی باشد (مثل مسئلهٔ کوله‌پشتی ۰ـ۱)، آنگاه $r(s_a)$ برابر با $\frac{f(s^*)}{f(s_a)}$ تعریف می شود (که باز بزرگتر از یک است) و باز هرچه به یک نزدیکتر باشد s_a بهتر است.

Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbor algorithm. In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour:



Indeed, if we change the weight of edge (a, d) from 6 to an arbitrary large number $w \geq 6$ in our example, the algorithm will still yield the tour $a \to b \to c \to d \to a$ of length $4 + w$, and the optimal solution will still be $a \to b \to d \to c \to a$ of length 8. Hence,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4 + w}{8}$$

which can be made as large as we wish by choosing an appropriately large value of w.

DEFINITION: *A polynomial-time approximation algorithm is said to be a c-approximation algorithm, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question:*

$$r(s_a) \leq c.$$

The best (i.e., the smallest) value of c for which the above inequality holds for all instances of the problem is called the performance ratio of the algorithm and denoted R_A.

مثلاً، برای یک الگوریتم 5-*approximation* برای یک مسئلهٔ کمینه سازی داریم $\frac{f(s_a)}{f(s^*)} \leq 5$؛ و برای یک مسئلهٔ 3-*approximation* برای یک مسئلهٔ بیشینه سازی داریم $\frac{f(s^*)}{f(s_a)} \leq 3$.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with R_A as close to 1 as possible. Unfortunately, some approximation algorithms have infinitely large performance ratios ($R_A = \infty$). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

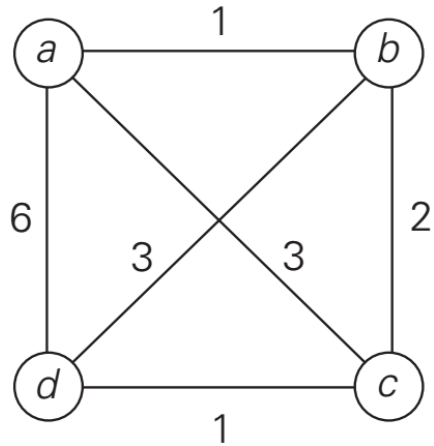
Multifragment-heuristic algorithm

Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2. An application of the greedy technique to this problem leads to the following algorithm:

* *Step 1: Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.*

* *Step 2: Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.*

* *Step 3: Return the set of tour edges.*



As an example, applying the algorithm to the above graph yields $\{(a, b), (c, d), (b, c), (a, d)\}$. *This set of edges forms the same tour as the one produced by the nearest-neighbor algorithm. In general, the multifragment-heuristic algorithm tends to produce significantly better tours than the nearest-neighbor algorithm. But the performance ratio of the multifragment-heuristic algorithm is also unbounded, of course.*

THEOREM: *If $P \neq NP$, there exists no c-approximation algorithm for the traveling salesman problem, i.e., there exists no polynomial-time approximation algorithm for this problem so that for all instances $f(s_a)/f(s^*) \leq c$. for some constant c.*

اثباتش را بعداً بیان خواهیم کرد انشاءالله.

APX *is the class of all optimization problems $\mathcal{P}$ such that, for some $r > 1$, there exists a polynomial-time r-approximate algorithm for $\mathcal{P}$. TSP is a problem for which determining approximate solutions with constant bounded performance ratio is computationally hard. Thus, TSP is not in APX.*

There is, however, a very important subset of instances, called **Euclidean**, *for which we can make a nontrivial assertion about the accuracy of both the nearest-neighbor and multifragment-heuristic algorithms. These are the instances in which intercity distances satisfy the following natural conditions:*

* ***triangle inequality*** $d[i, j] \leq d[i, k] + d[k, j]$ *for any triple of cities* i, j, *and* k *(the distance between cities* i *and* j *cannot exceed the length of a two-leg path from* i *to some intermediate city* k *to* j*)*

* ***symmetry*** $d[i, j] = d[j, i]$ *for any pair of cities* i *and* j *(the distance from* i *to* j *is the same as the distance from* j *to* i*)*

A substantial majority of practical applications of the traveling salesman problem are its Euclidean instances. They include, in particular, geometric ones, where cities correspond to points in the plane and distances are computed by the standard Euclidean formula. ***Although the performance ratios of the nearest-neighbor and multifragment-heuristic algorithms remain unbounded for Euclidean instances, their accuracy ratios satisfy the following inequality for any such instance with*** $n \geq 2$ ***cities:***

$$\frac{f(s_a)}{f(s^*)} \leq \frac{\lceil \log_2 n \rceil + 1}{2},$$

where $f(s_a)$ ***and*** $f(s^*)$ ***are the lengths of the heuristic tour and shortest tour, respectively.***

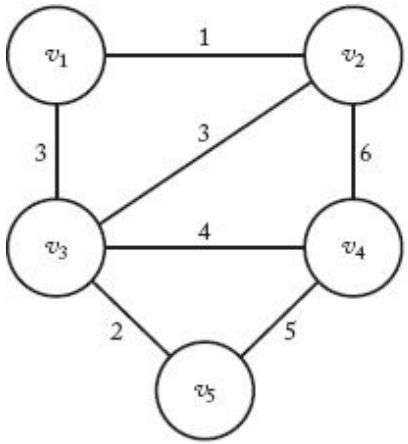
در جلسات آتی، یک الگوریتم ***2-approximation*** را برای حل نمونه های مسئلهٔ ***TSP*** اقلیدسی معرفی خواهیم کرد. این الگوریتم مبتنی بر بهره گیری از درخت فراگیر کمینه است.

Minimum Spanning Trees

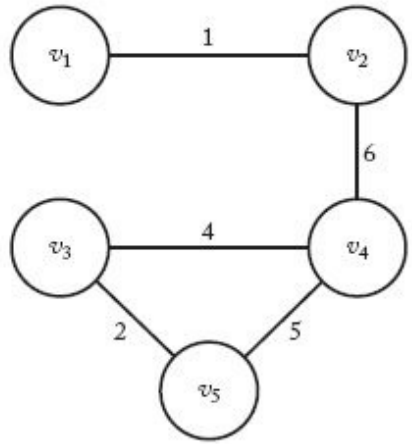
DEFINITION: *A spanning tree of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.*

A spanning tree for a connected, weighted, undirected graph G is a connected subgraph that contains all the vertices in G and is a tree. Our aim is to obtain a spanning tree of minimum weight. Such a tree is called a minimum spanning tree. A graph can have more than one minimum spanning tree.

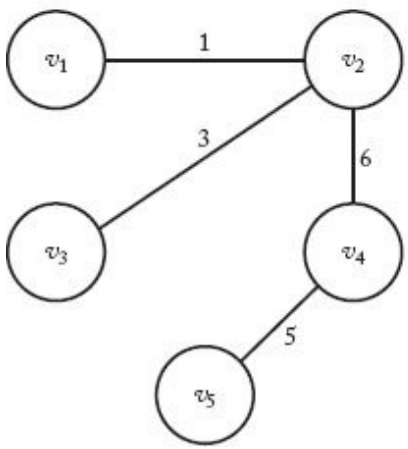
(a) A connected, weighted, undirected graph G.



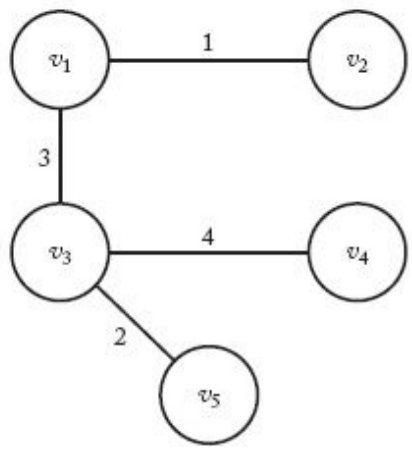
(b) If (v_4,v_5) were removed from this subgraph, the graph would remain connected.



(c) A spanning tree for G.



(d) A minimum spanning tree for G.



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. We will 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. A high-level greedy algorithm for the problem could proceed as follows:

```
F = ∅                                                    // Initialize set of
                                                         // edges to empty.
while (the instance is not solved){

  select an edge according to some locally
  optimal consideration;                                 // selection procedure

  if (adding the edge to F does not create a cycle)
    add it;                                              // feasibility check

  if (T = (V,F) is a spanning tree)                      // solution check
    the instance is solved;
}
```

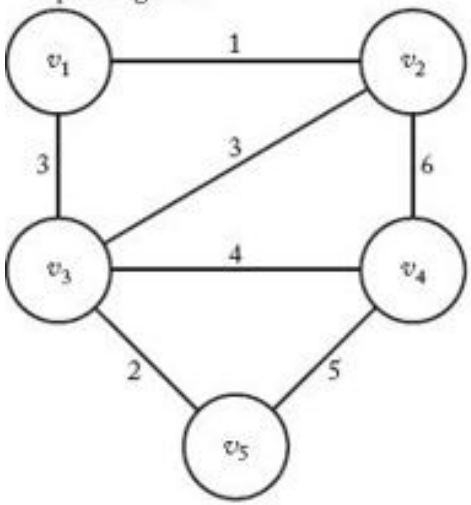
This algorithm simply says "select an edge according to some locally optimal consideration." There is no unique locally optimal property for a given problem. We will investigate two different greedy algorithms for this problem, Prim's algorithm and Kruskal's algorithm. Each uses a different locally optimal property. Recall that there is no guarantee that a given greedy algorithm always yields an optimal solution. One must prove whether or not this is the case. It can be proved that both Prim's and Kruskal's algorithms always produce minimum spanning trees.

Prim's Algorithm

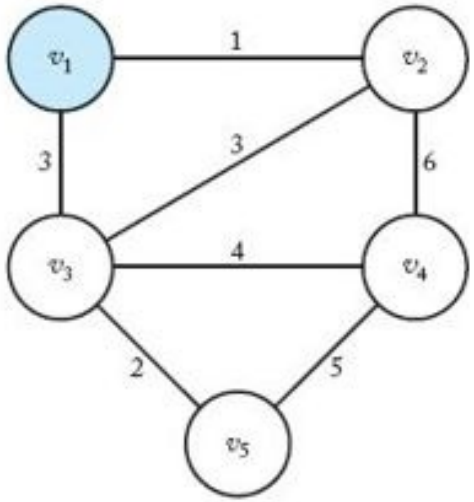
```
F = ∅;                                   // Initialize set of edges
                                         // to empty.
Y = {v1};                                // Initialize set of vertices to
                                         // contain only the first one.
while (the instance is not solved){

   select a vertex in V − Y that is      // selection procedure and
   nearest to Y;                         // feasibility check

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

   if (Y == V)                           // solution check
       the instance is solved;
}
```

The selection procedure and feasibility check are done together because taking the new vertex from $V - Y$ guarantees that a cycle is not created.

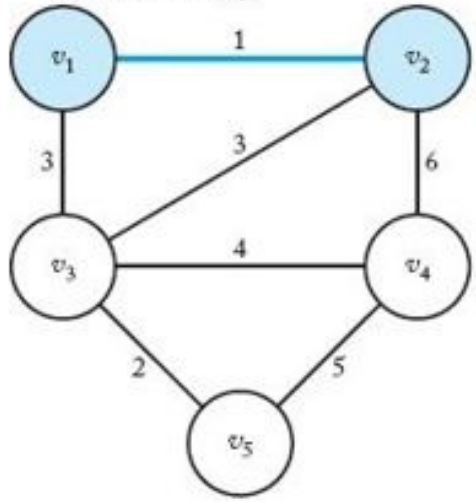
Determine a minimum spanning tree.



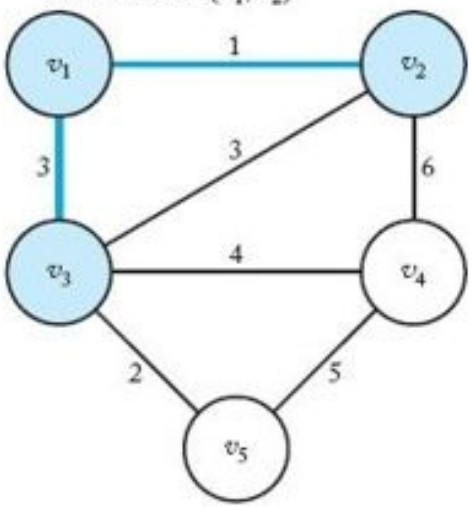
1. Vertex v_1 is selected first.



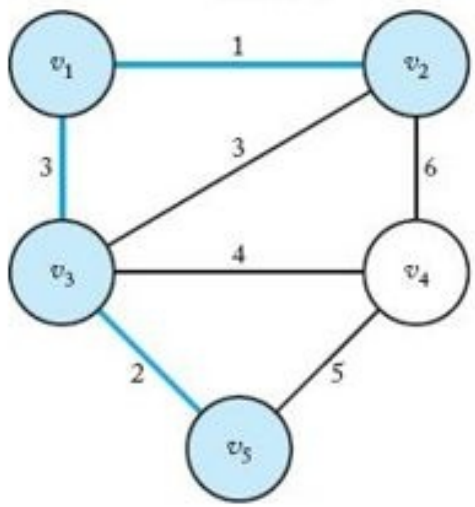
2. Vertex v_2 is selected because it is nearest to $\{v_1\}$.



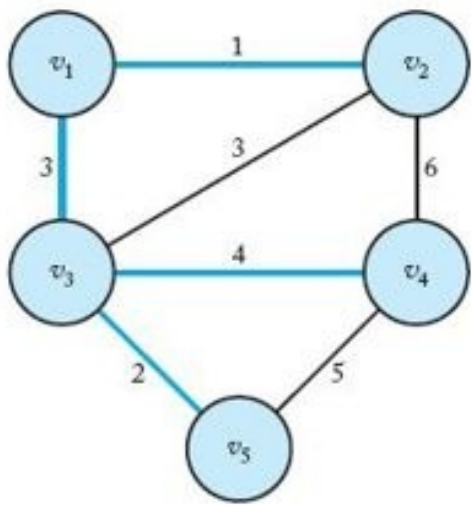
3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$.



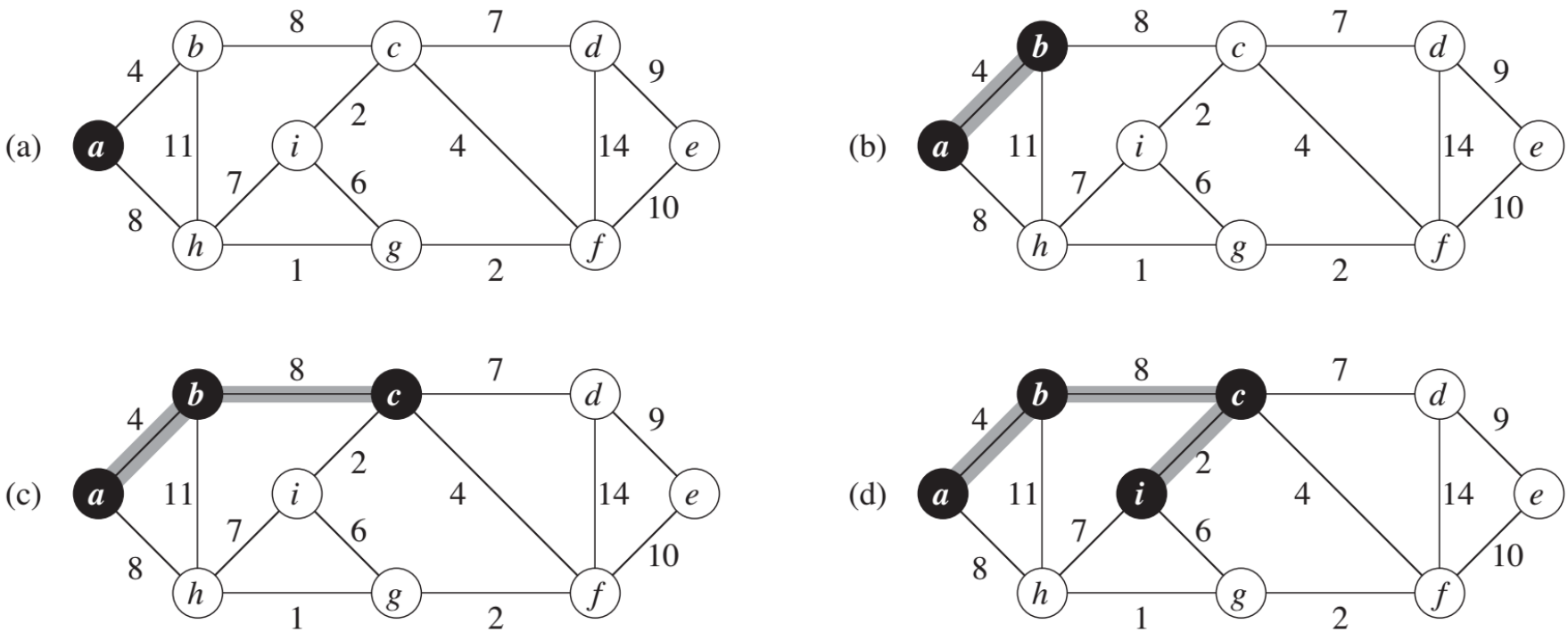
4. Vertex v_2 is selected because it is nearest to $\{v_1, v_2, v_3\}$.



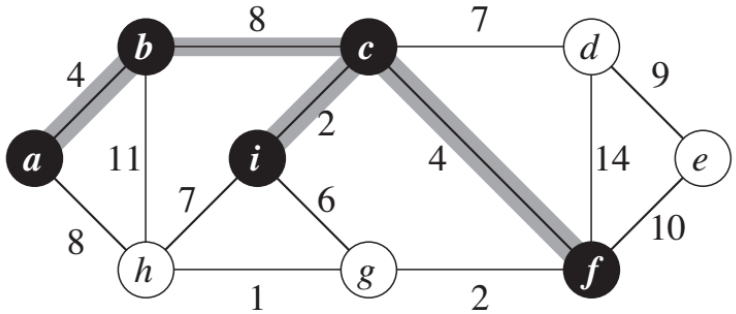
5. Vertex v_4 is selected.



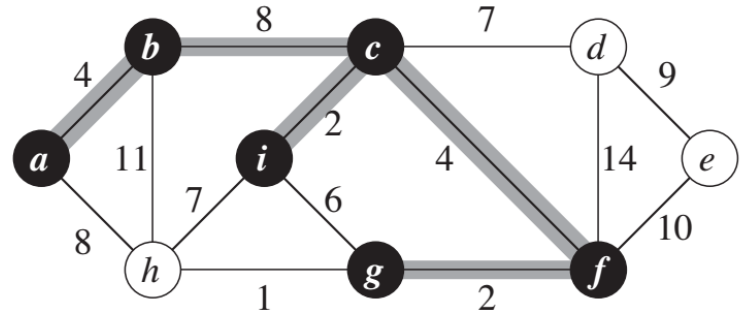
As another example



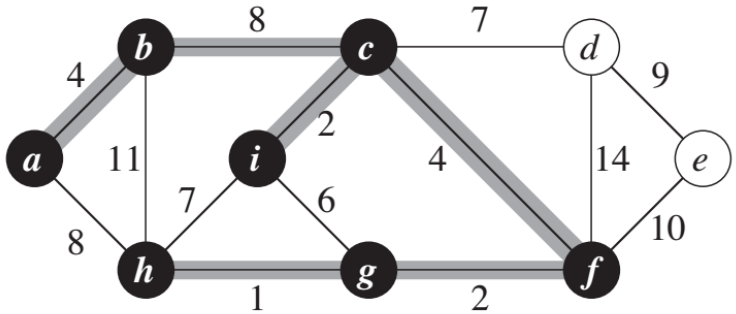
(e)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
14
9
7
6
10
1
2

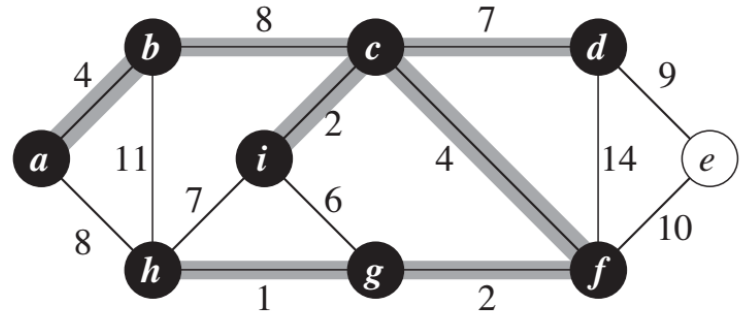
(f)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
14
9
7
6
10
1
2

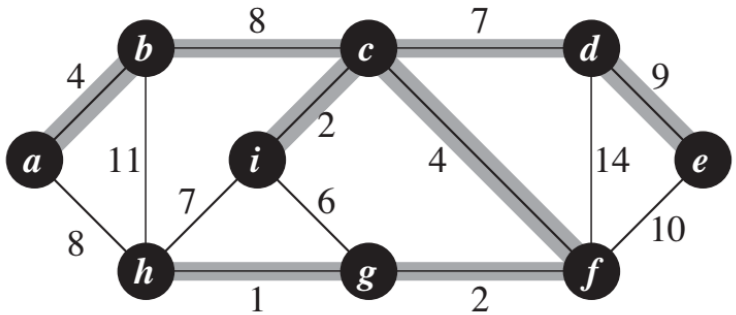
(g)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
14
9
7
6
10
1
2

(h)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
14
9
7
6
10
1
2

(i)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
14
9
7
6
10
1
2

Disjoint Sets

Some applications involve grouping n distinct elements into a collection of disjoint sets. These applications often need to perform two operations in particular: finding the unique set that contains a given element and uniting two sets.

A disjoint-set data structure maintains a collection

$$\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$$

of disjoint dynamic sets. We identify each set by a representative, which is some member of the set. In some applications, it does not matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times. Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (assuming, of course, that the elements can be ordered).

We wish to support the following operations:

☞ $makeset(x)$: ***creates a new set whose only member (and thus representative) is*** x. ***Since the sets are disjoint, we require that*** x ***not already be in some other set.***

☞ $union(x, y)$: ***unites the dynamic sets that contain*** x ***and*** y, ***say*** S_x ***and*** S_y, ***into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of*** $S_x \cup S_y$, ***although many implementations of*** $union$ ***specifically choose the representative of either*** S_x ***or*** S_y ***as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets*** S_x ***and*** S_y, ***removing them from the collection*** $\mathcal{S}$. ***In practice, we often absorb the elements of one of the sets into the other set.***

☞ $findset(x)$: ***returns a pointer to the representative of the (unique) set containing*** x.

Since the sets are disjoint, each $union$ ***operation reduces the number of sets by one. After*** $n-1$ $union$ ***operations, therefore, only one set remains. The number of*** $union$ ***operations is thus at most*** $n-1$.

***Kruskal's algorithm** is one of a number of applications that require a dynamic partition of some n element set $\mathcal{U}$ into a collection of disjoint subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$. After being initialized as a collection of n one-element subsets, each containing a different element of $\mathcal{U}$, the collection is subjected to a sequence of intermixed union and find operations.*

☞ $makeset(x)$: *creates a one-element set $\{x\}$. It is assumed that this operation can be applied to each of the elements of set $\mathcal{U}$ only once.*

☞ $find(x)$: *returns a subset containing x.*

☞ $union(x, y)$: *constructs the union of the disjoint subsets S_x and S_y containing x and y, respectively, and adds it to the collection to replace S_x and S_y, which are deleted from it.*

For example, let $\mathcal{U} = \{1, 2, 3, 4, 5, 6\}$***. Then*** $makeset(i)$ ***creates the set*** $\{i\}$ ***and applying this operation six times initializes the structure to the collection of six singleton sets:***

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}.$$

Performing $union(1, 4)$ ***and*** $union(5, 2)$ ***yields***

$$\{1, 4\}, \{5, 2\}, \{3\}, \{6\},$$

and, if followed by $union(4, 5)$ ***and then by*** $union(3, 6)$***, we end up with the disjoint subsets***

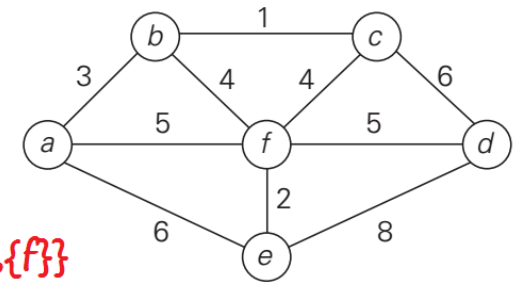
$$\{1, 4, 5, 2\}, \{3, 6\}.$$

Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Kruskal's Algorithm

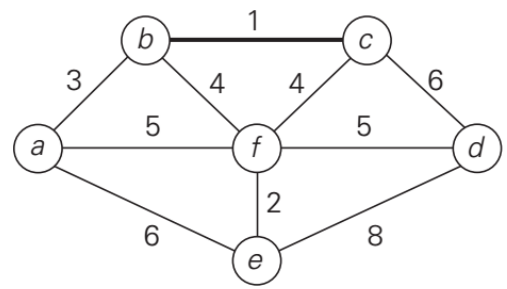
```
F = ∅;                                          // Initialize set of
                                                // edges to empty.
create disjoint subsets of V, one for each
vertex and containing only that vertex;

sort the edges in E in nondecreasing order;

while (the instance is not solved){

  select next edge;                             // selection procedure

  if (the edge connects two vertices in         // feasibility check
                    disjoint subsets){
      merge the subsets;
      add the edge to F;
  }
  if (all the subsets are merged)               // solution check
     the instance is solved;
}
```



{{a},{b},{c},{d},{e},{f}}

Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

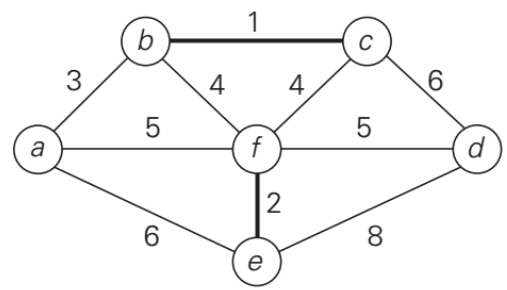
{{a},{b,c},{d},{e},{f}}



bc
1

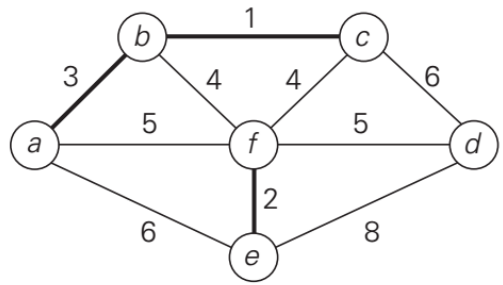
bc **ef** ab bf cf af df ae cd de
1 2 3 4 4 5 5 6 6 8

{{a},{b,c},{d},{e,f}}



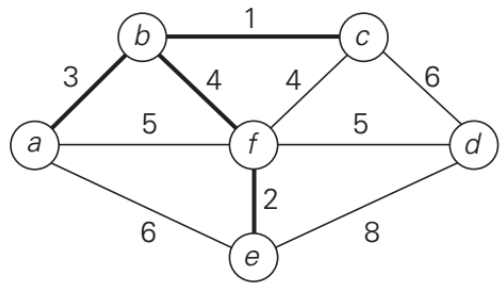
ef	bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2	1	2	3	4	4	5	5	6	6	8

{{a,b,c},{d},{e,f}}



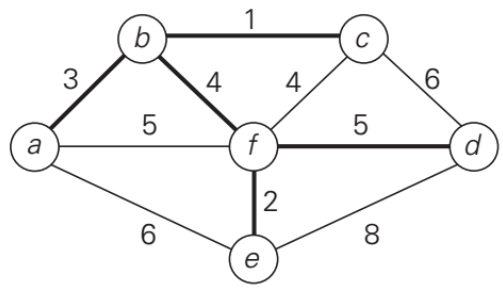
ab	bc	ef	ab	**bf**	cf	af	df	ae	cd	de
3	1	2	3	4	4	5	5	6	6	8

{{a,b,c,e,f},{d}}



bf	bc	ef	ab	bf	cf	af	**df**	ae	cd	de
4	1	2	3	4	4	5	5	6	6	8

{{a,b,c,d,e,f}}



df
5