Lecture 10

Greedy Algorithm

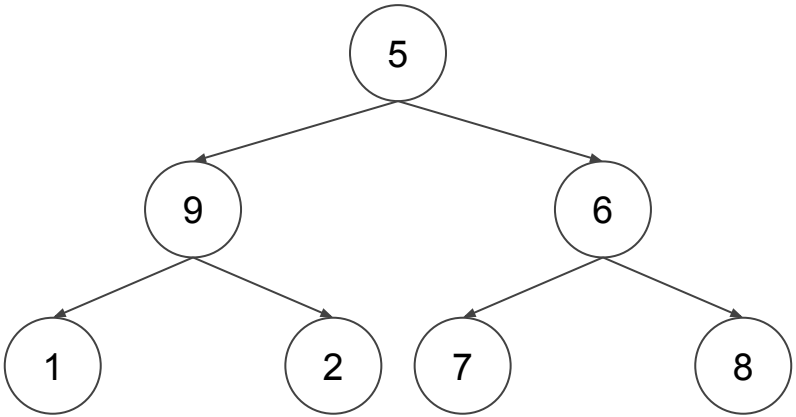
Greedy Algorithm

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time. A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems.

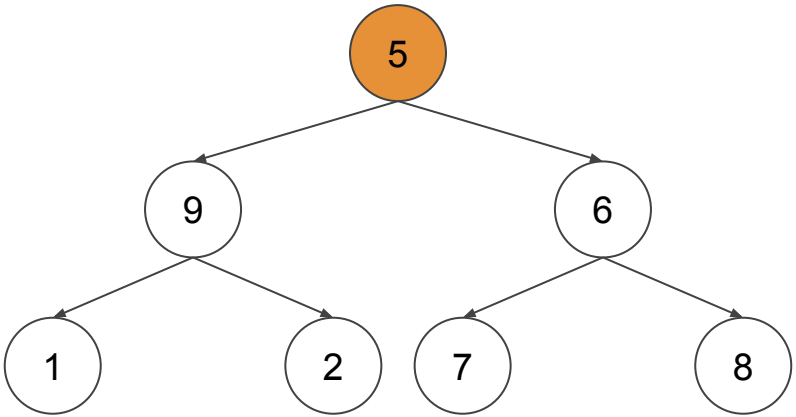
Greedy Algorithm Failing to Give Optimal Solution

In many problems, a greedy strategy does not produce an optimal solution. For example, in the example ahead, the greedy algorithm seeks to find the path with the largest sum. It does this by selecting the largest available number at each step. The greedy algorithm fails to find the largest sum, however, because it makes decisions based only on the information it has at any one step, without regard to the overall problem.

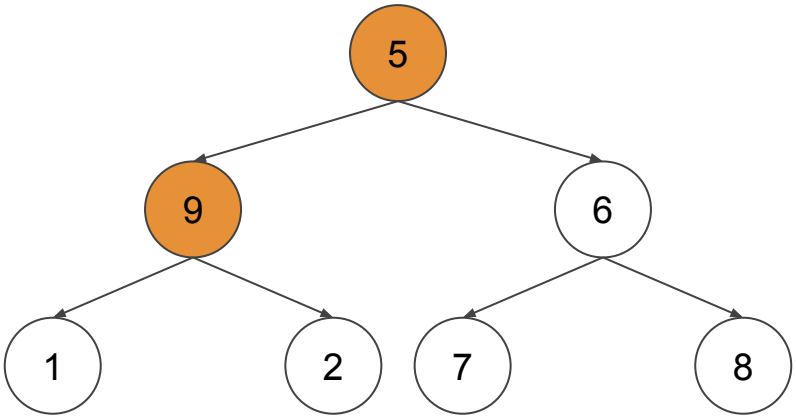
Greedy Algorithm Failing to Give Optimal Solution Continued



Greedy Algorithm Failing to Give Optimal Solution Continued

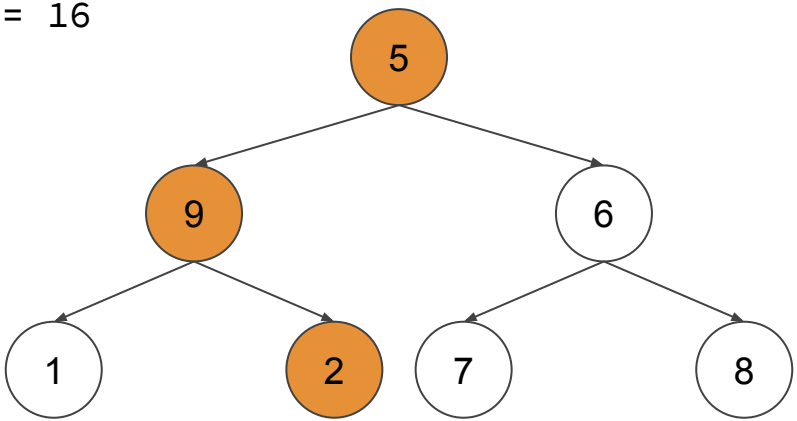


Greedy Algorithm Failing to Give Optimal Solution Continued



Greedy Algorithm Failing to Give Optimal Solution Continued

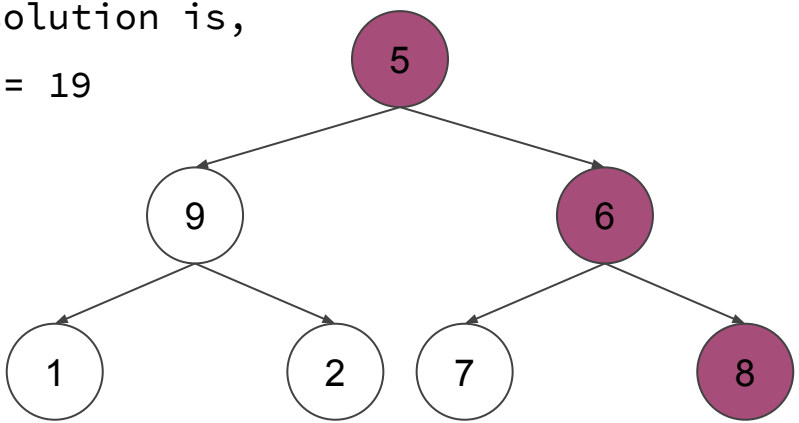
SUM = 5 + 9 + 2 = 16



Greedy Algorithm Failing to Give Optimal Solution Continued

But the actual solution is,

SUM = 5 + 6 + 8 = 19



Conditions for Greedy Algorithm

Greedy algorithms take all of the data in a particular problem, and then set a rule for which elements to add to the solution at each step of the algorithm. If both of the properties below are true, a greedy algorithm can be used to solve the problem.

1. **Greedy Choice Property**: A global (overall) optimal solution can be reached by choosing the optimal choice at each step.
2. **Optimal Substructure Property**: A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.

Conditions for Greedy Algorithm Continued

In other words, greedy algorithms work on problems for which it is true that, at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem.

Minimum Spanning Trees

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, we can use an arrangement of n - 1 wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

Minimum Spanning Trees Continued

We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v. We then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it “spans” the graph G.

Growing a Minimum Spanning Tree

We will be applying a greedy generic method, which grows the minimum spanning tree one edge at a time. The generic method manages a set of edges A, maintaining the following loop invariant:

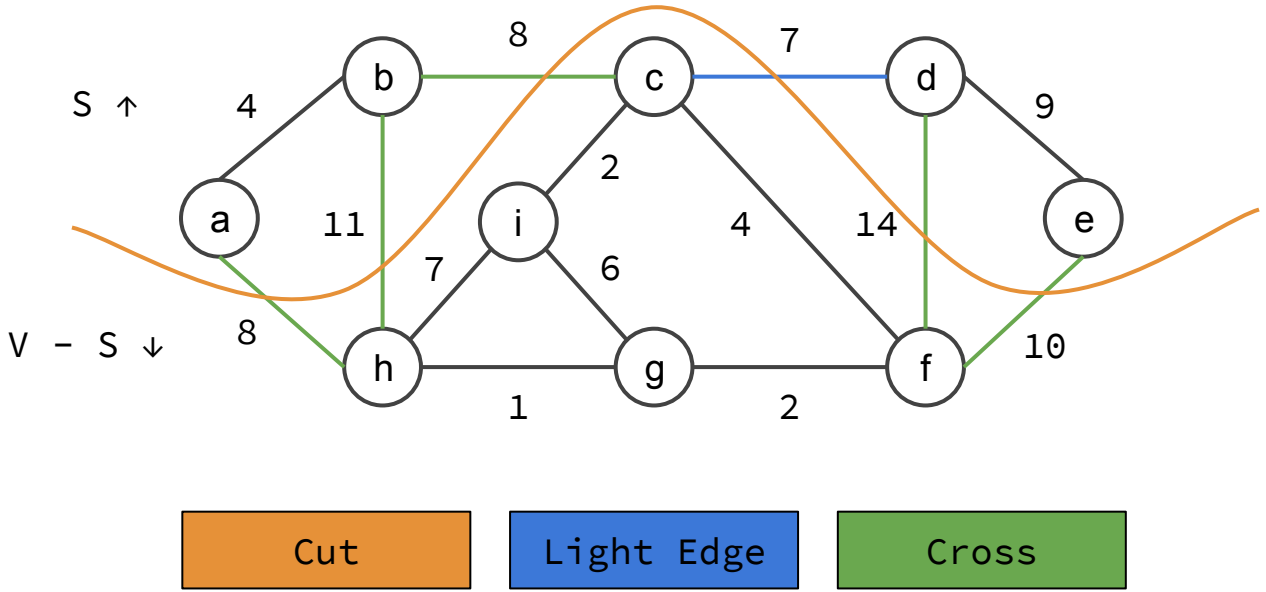
Prior to each iteration, A is a subset of some minimum spanning tree. At each step, we determine an edge (u, v) that we can add to A without violating this invariant, in the sense that A ∪ {(u, v)} is also a subset of a minimum spanning tree. We call such an edge a safe edge for A, since we can add it safely to A while maintaining the invariant.

Minimum Spanning Trees Continued

---

1. **Cut**: A cut $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V.
2. **Crosses**: We say that an edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$. We say that a cut respects a set A of edges if no edge in A crosses the cut.
3. **Light Edge:** An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut. Note that there can be more than one light edge crossing a cut in the case of ties.

Minimum Spanning Trees Continued



Growing a Minimum Spanning Tree

```
Generic_MST(G, w):
    A = ∅
    while A does not form a spanning tree:
        find an edge (u, v) that is safe for A
        A = A ∪ {(u, v)}
    return A
```

Kruskal’s Algorithm

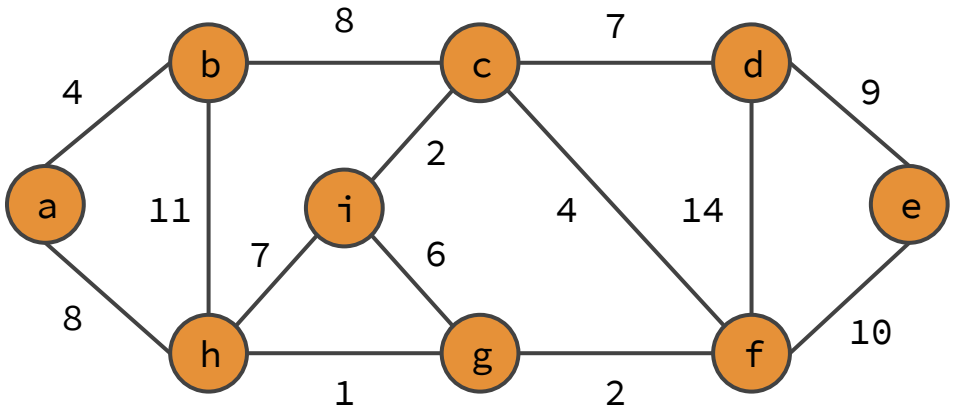
Kruskal’s algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight. Let C_1 and C_2 denote the two trees that are connected by (u, v). Since (u, v) must be a light edge connecting C_1 to some other tree, where (u, v) is a safe edge for C_1.

Kruskal’s algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight.

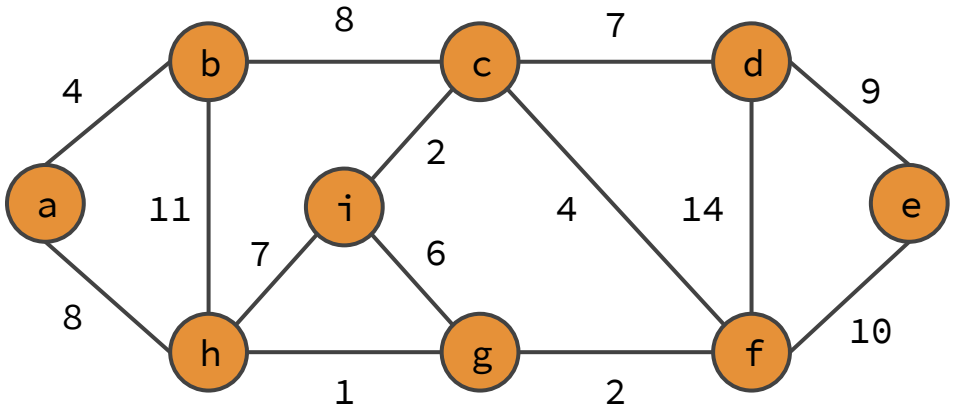
Kruskal's Algorithm Pseudocode

```
MST_Kruskal(G, w):
    A = ∅
    for each vertex v ∈ G.V:
        Make_Set(v)
    sort the edges of G.E into nondecreasing order by weight w
    for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight:
        if Find_Set(u) ≠ Find_Set(v):
            A = A ∪ {(u,v)}
            Union(u, v)
    return A
```

Simulation of Kruskal's Algorithm



Simulation of Kruskal's Algorithm Continued.

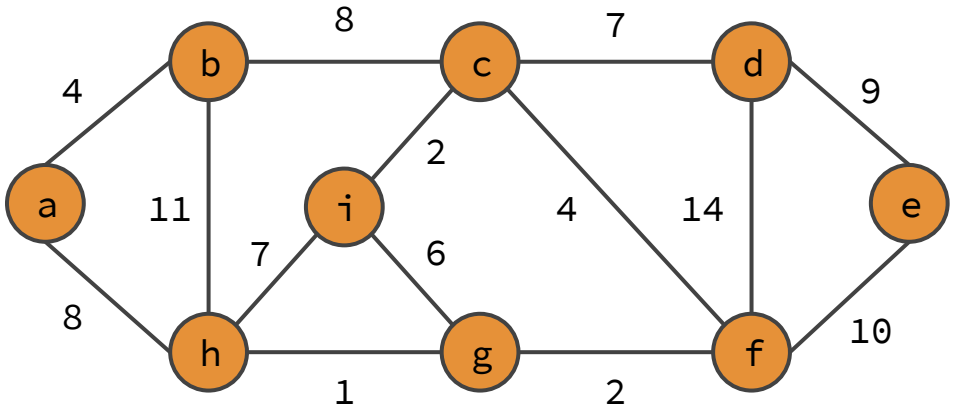


{a}
{b}
{c}
{d}
{e}

{f}
{g}
{h}
{i}

Edges	Weight	Status
(a, b)	4	
(a, h)	8	
(b, c)	8	
(b, h)	11	
(c, d)	7	
(c, f)	4	
(c, i)	2	
(d, e)	9	
(d, f)	14	
(e, f)	10	
(f, g)	2	
(g, h)	1	
(g, i)	6	
(h, i)	7	

Simulation of Kruskal's Algorithm Continued.

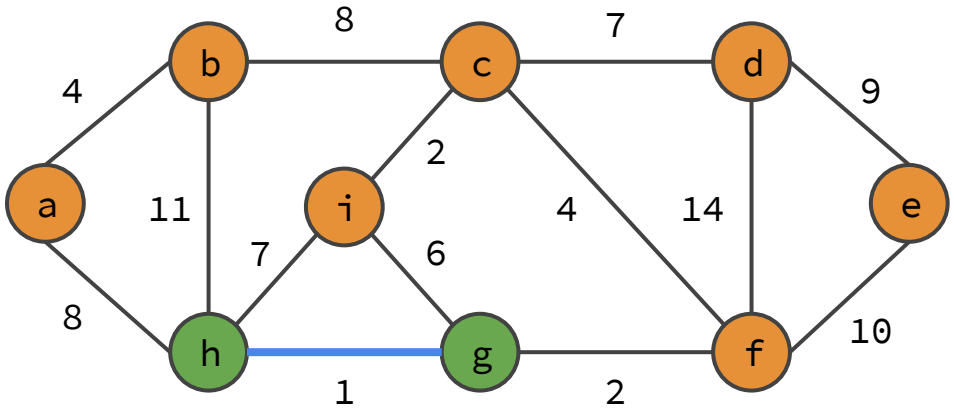


{a}
{b}
{c}
{d}
{e}

{f}
{g}
{h}
{i}

Edges	Weight	Status
(g, h)	1	
(c, i)	2	
(f, g)	2	
(a, b)	4	
(c, f)	4	
(g, i)	6	
(c, d)	7	
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

Simulation of Kruskal's Algorithm Continued.



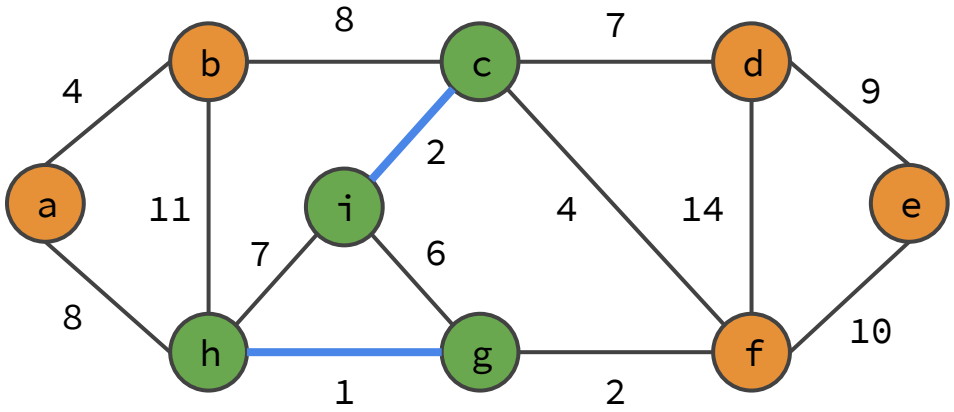
{a}
{b}
{c}
{d}
{e}

{f}
{g, h}
{i}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	
(f, g)	2	
(a, b)	4	
(c, f)	4	
(g, i)	6	
(c, d)	7	
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

Simulation of Kruskal's Algorithm Continued.

– – –

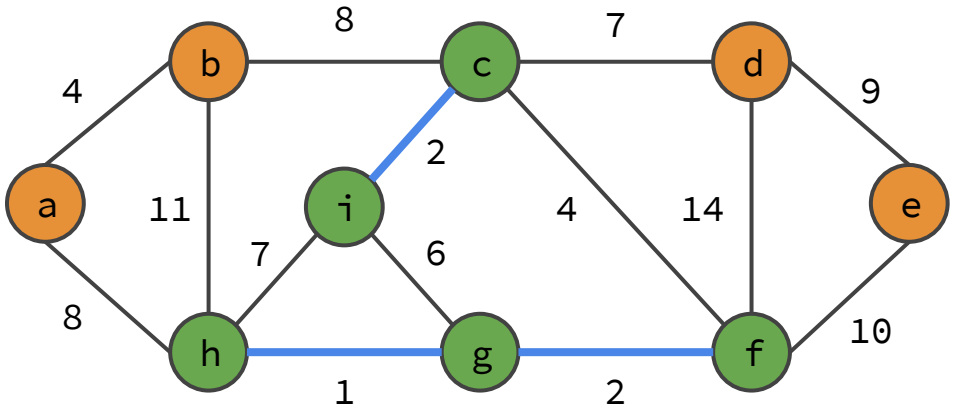


{a}
{b}
{c, i}
{d}
{e}

{f}
{g, h}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	
(a, b)	4	
(c, f)	4	
(g, i)	6	
(c, d)	7	
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

Simulation of Kruskal's Algorithm Continued.

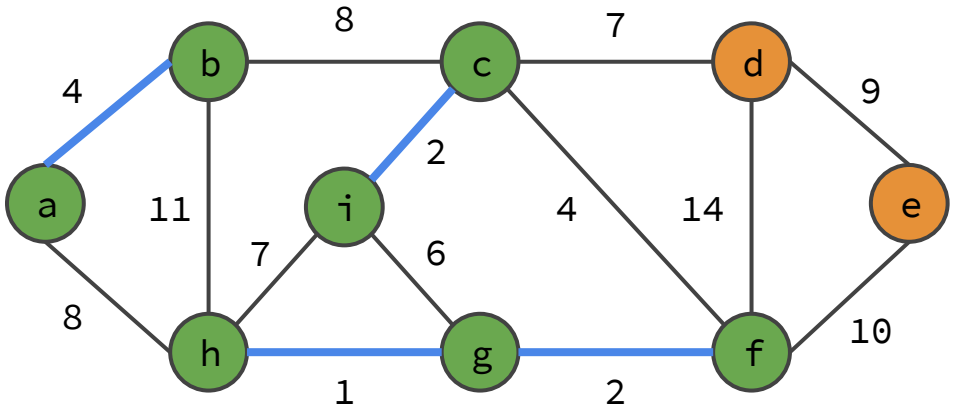


{a}
{b}
{c, i}
{d}
{e}

{f, g, h}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	
(c, f)	4	
(g, i)	6	
(c, d)	7	
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

Simulation of Kruskal's Algorithm Continued.



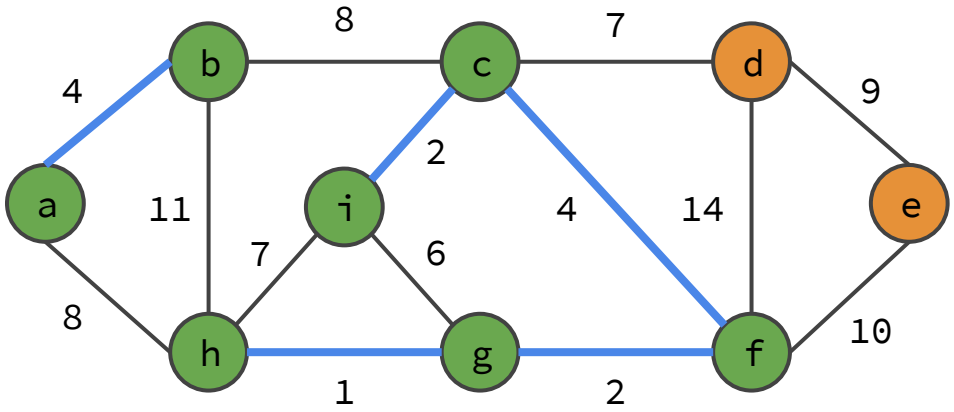
{a, b}
{c, i}
{d}
{e}

{f, g, h}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	
(g, i)	6	
(c, d)	7	
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

Simulation of Kruskal's Algorithm Continued.

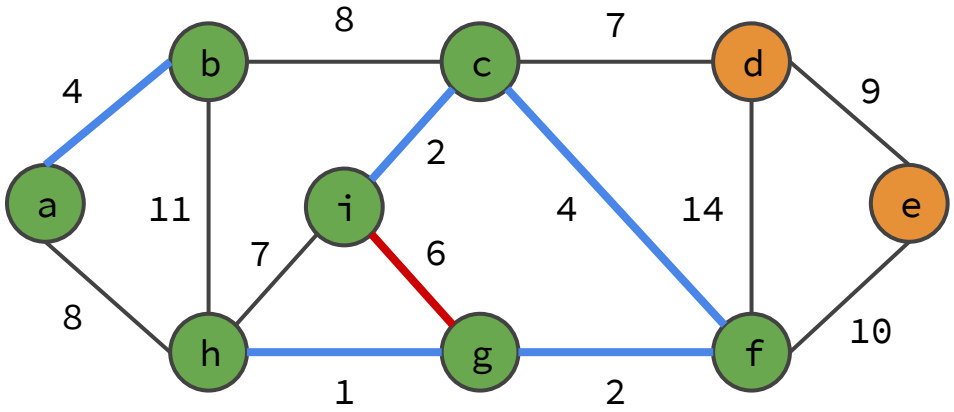
– – –



{a, b}
{c, i, f, g, h}
{d}
{e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	
(c, d)	7	
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

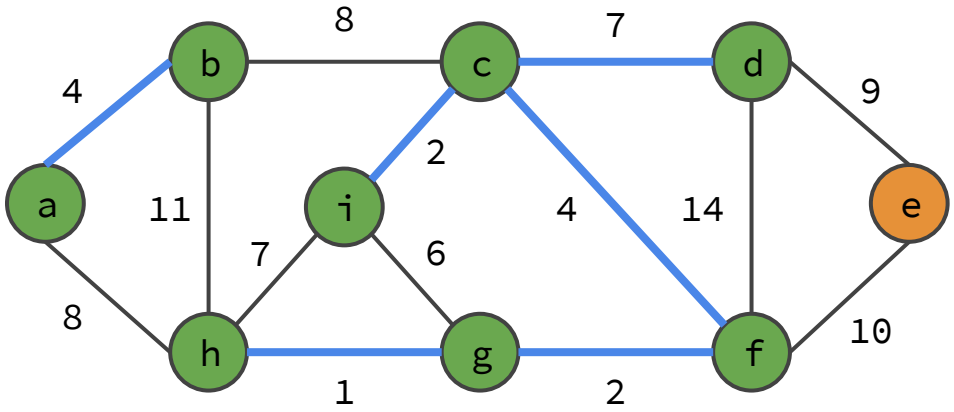
Simulation of Kruskal's Algorithm Continued.



{a, b}
{c, i, f, g, h}
{d}
{e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	×
(c, d)	7	
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

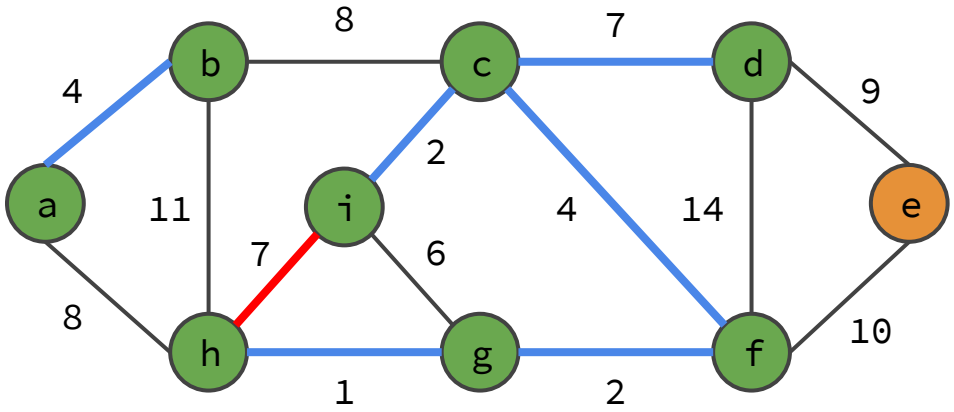
Simulation of Kruskal's Algorithm Continued.



{a, b}
{c, i, f, g, h}
{d}
{e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	×
(c, d)	7	✓
(h, i)	7	
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

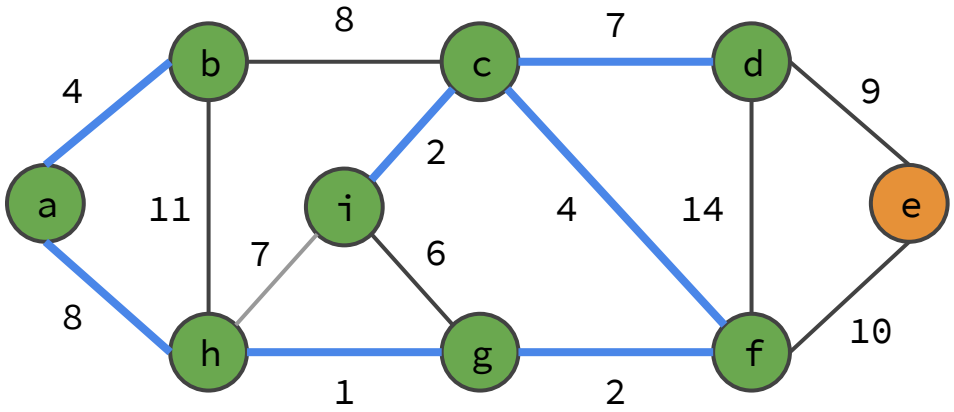
Simulation of Kruskal's Algorithm Continued.



{a, b}
{c, i, f, g, h, d}
{e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	×
(c, d)	7	✓
(h, i)	7	×
(a, h)	8	
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

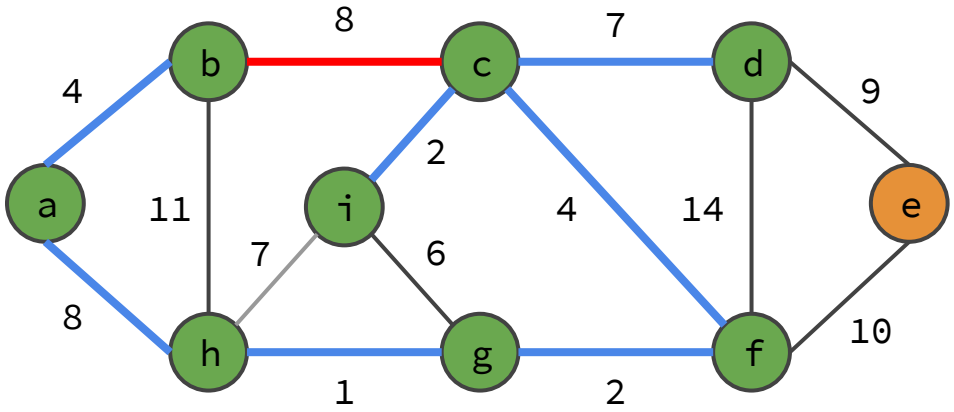
Simulation of Kruskal's Algorithm Continued.



{a, b, c, i, f, g, h, d}
{e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	×
(c, d)	7	✓
(h, i)	7	×
(a, h)	8	✓
(b, c)	8	
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

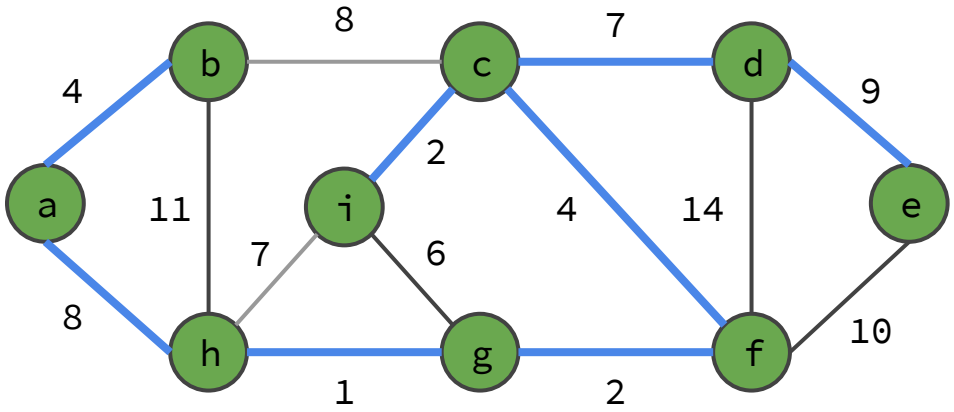
Simulation of Kruskal's Algorithm Continued.



{a, b, c, i, f, g, h, d}
{e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	×
(c, d)	7	✓
(h, i)	7	×
(a, h)	8	✓
(b, c)	8	×
(d, e)	9	
(e, f)	10	
(b, h)	11	
(d, f)	14	

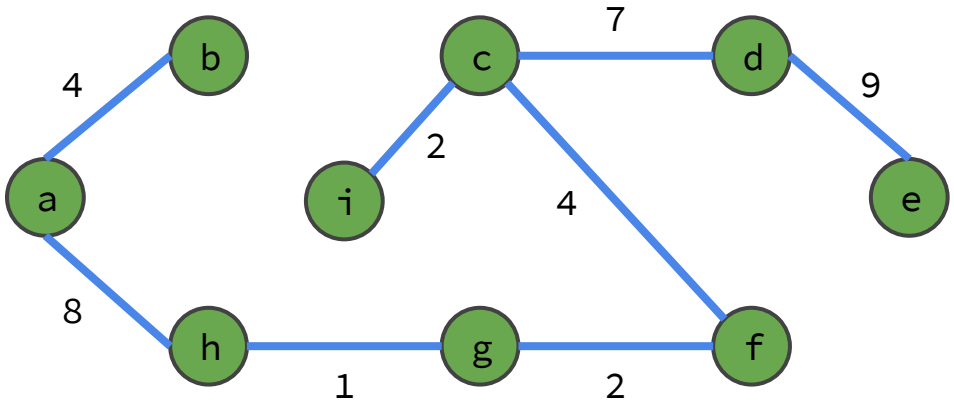
Simulation of Kruskal's Algorithm Continued.



{a, b, c, i, f, g, h, d, e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	×
(c, d)	7	✓
(h, i)	7	×
(a, h)	8	✓
(b, c)	8	×
(d, e)	9	✓
(e, f)	10	-
(b, h)	11	-
(d, f)	14	-

Simulation of Kruskal's Algorithm Continued.



{a, b, c, i, f, g, h, d, e}

Edges	Weight	Status
(g, h)	1	✓
(c, i)	2	✓
(f, g)	2	✓
(a, b)	4	✓
(c, f)	4	✓
(g, i)	6	×
(c, d)	7	✓
(h, i)	7	×
(a, h)	8	✓
(b, c)	8	×
(d, e)	9	✓
(e, f)	10	-
(b, h)	11	-
(d, f)	14	-

Complexity Analysis of Kruskal's Algorithm

The running time of Kruskal's algorithm for a graph G = (V, E) depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initializing the set A takes O(1) time, and the time to sort the edges is O(ElgE). The main for loop performs O(E) Find_Set and Union operations on the disjoint-set forest. Along with the |V| MAKE-SET operations, these take a total of O((V + E)α(V)) time, where α is the very slowly growing function. Because we assume that G is connected, we have |E| ≥ |V| - 1, and so the

Complexity Analysis of Kruskal's Algorithm Continued

---

disjoint-set operations take O(E α(V)) time. Moreover, since α|V| = O(lgV) = O(lgE), the total running time of Kruskal's algorithm is O(ElgE). Observing that |E| < $|V|^2$, we have lg|E| = O(lgV), and so we can restate the running time of Kruskal's algorithm as O(ElgV).

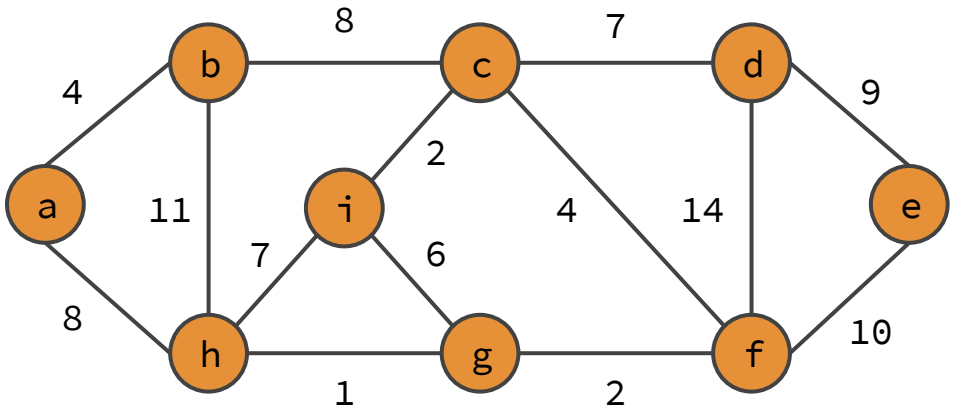
Prim’s Algorithm

Like Kruskal’s algorithm, Prim’s algorithm is a special case of the generic minimum-spanning-tree method. Prim’s algorithm has the property that the edges in the set A always form a single tree. As the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V . Each step adds to the tree A a light edge that connects A to an isolated vertex—one on which no edge of A is incident. This rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree’s weight.

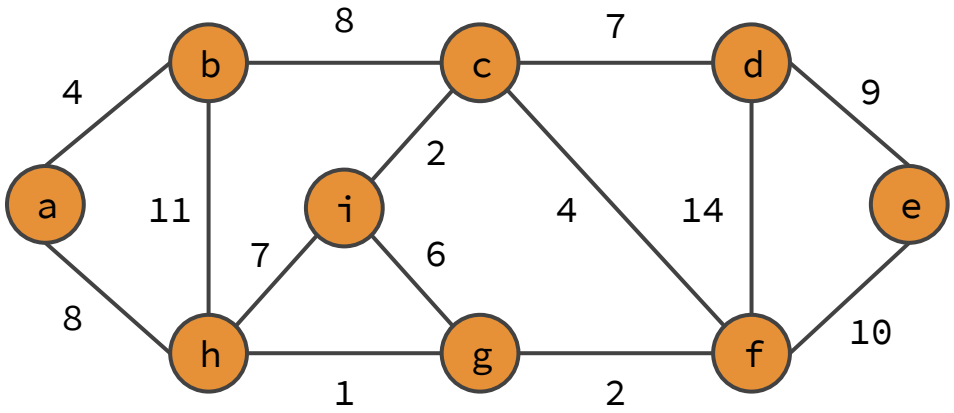
Prim’s Algorithm Pseudocode

```
MST_Prim(G, w, r):
     for each u ∈ G.V:
          u.cost = ∞
          u.π = NIL
     r.cost = 0
     Q = G.V
     while Q ≠ ∅:
          u = Extract_Min(Q)
          for each v ∈ G.Adj[u]:
               if v ∈ Q and w(u, v) < v.cost:
                    v.π = u
                    v.cost = w(u, v) [Update Key]
```

Simulation of Prim's Algorithm

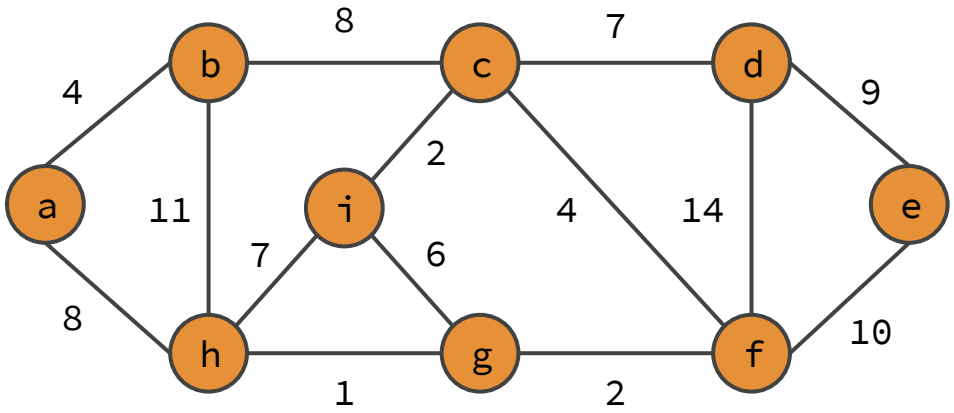


Simulation of Prim's Algorithm



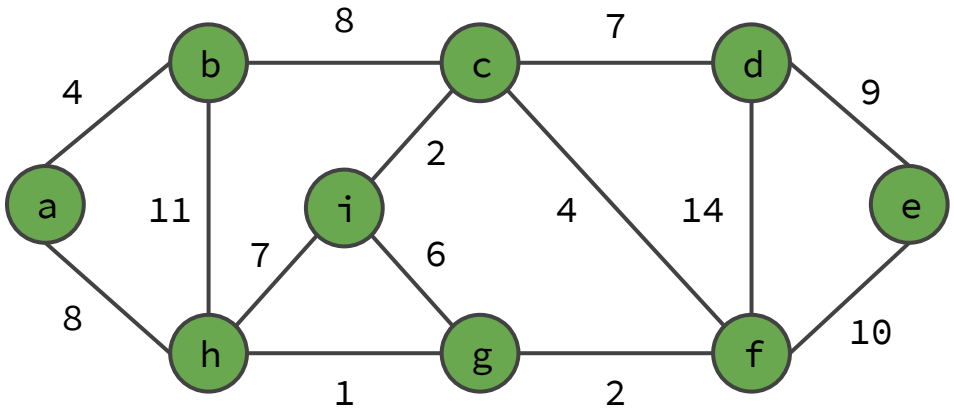
Vertex	Cost	Parent	Visited
a	-	-	-
b	-	-	-
c	-	-	-
d	-	-	-
e	-	-	-
f	-	-	-
g	-	-	-
h	-	-	-
i	-	-	-

Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	∞	NIL	-
b	∞	NIL	-
c	∞	NIL	-
d	∞	NIL	-
e	∞	NIL	-
f	∞	NIL	-
g	∞	NIL	-
h	∞	NIL	-
i	∞	NIL	-

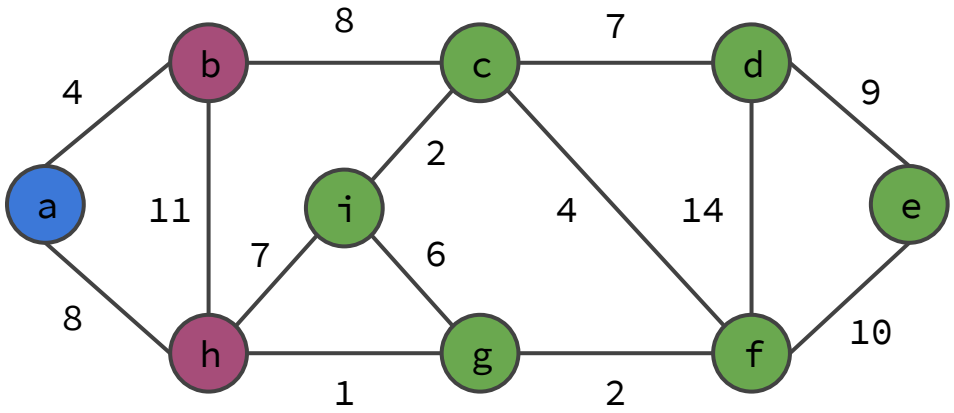
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	-
b	∞	NIL	-
c	∞	NIL	-
d	∞	NIL	-
e	∞	NIL	-
f	∞	NIL	-
g	∞	NIL	-
h	∞	NIL	-
i	∞	NIL	-

Min Priority Queue: a b c d e f g h i

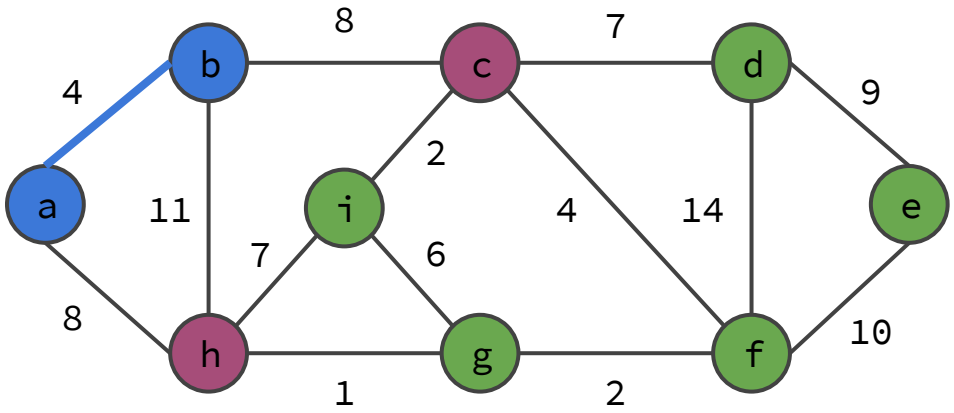
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	True
c	∞	NIL	-
d	∞	NIL	-
e	∞	NIL	-
f	∞	NIL	-
g	∞	NIL	-
h	8	a	True
i	∞	NIL	-

Min Priority Queue: b c d e f g h i

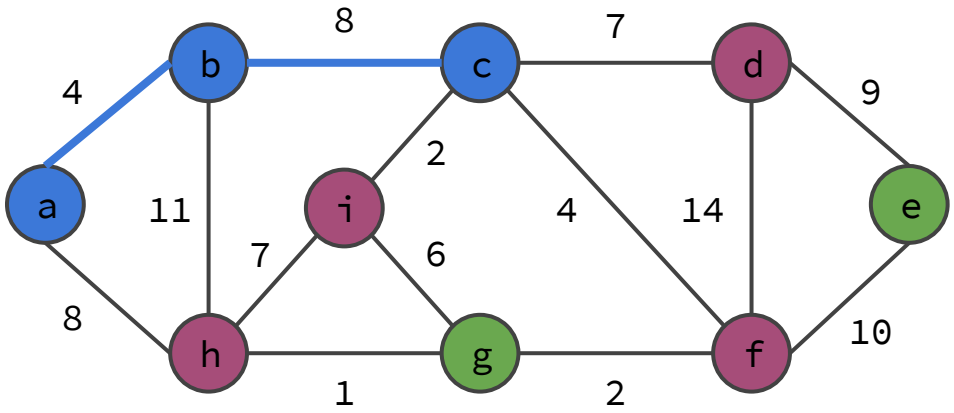
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	True
d	∞	NIL	-
e	∞	NIL	-
f	∞	NIL	-
g	∞	NIL	-
h	8	a	True
i	∞	NIL	-

Min Priority Queue: c d e f g h i

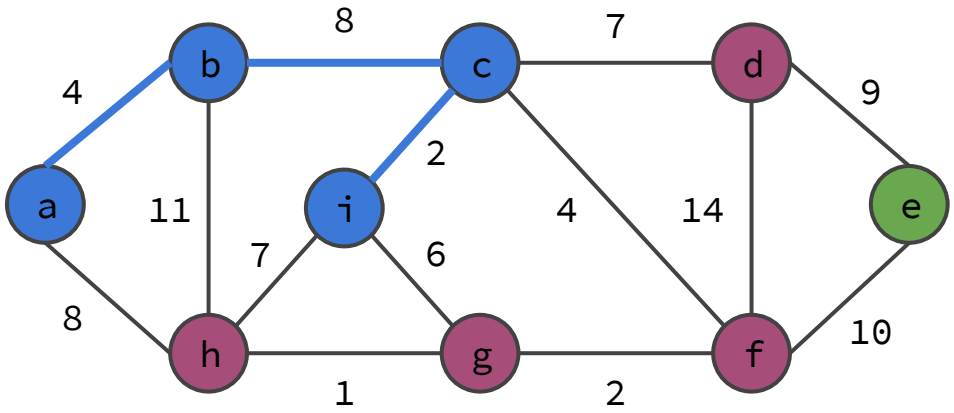
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	True
e	∞	NIL	-
f	4	c	True
g	∞	NIL	-
h	8	a	True
i	2	c	True

Min Priority Queue: d e f g h i

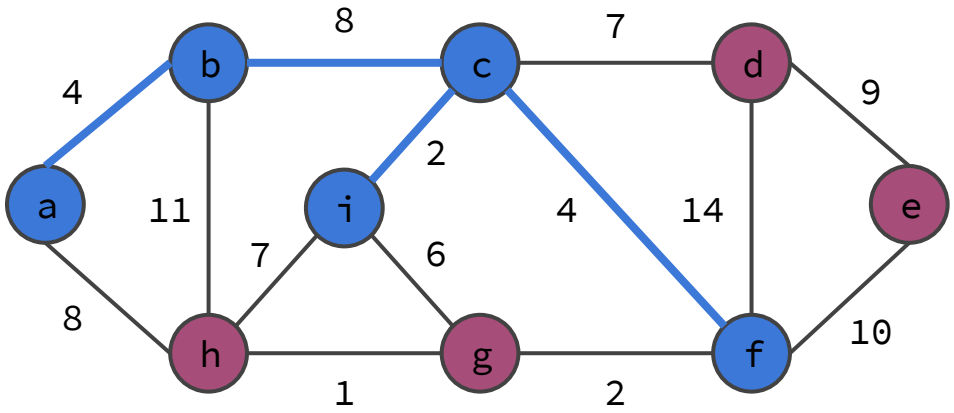
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	True
e	∞	NIL	-
f	4	c	True
g	6	i	True
h	7	i	True
i	2	c	Extracted

Min Priority Queue: d e f g h

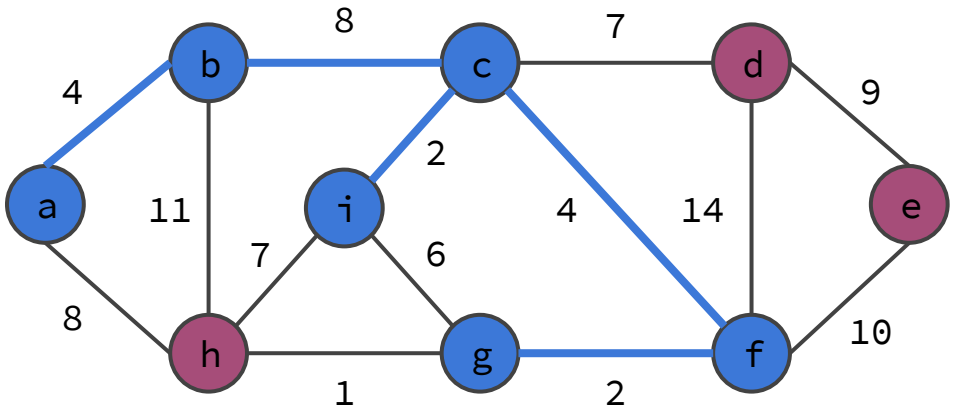
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	True
e	10	f	True
f	4	c	Extracted
g	2	f	True
h	7	i	True
i	2	c	Extracted

Min Priority Queue: d e g h

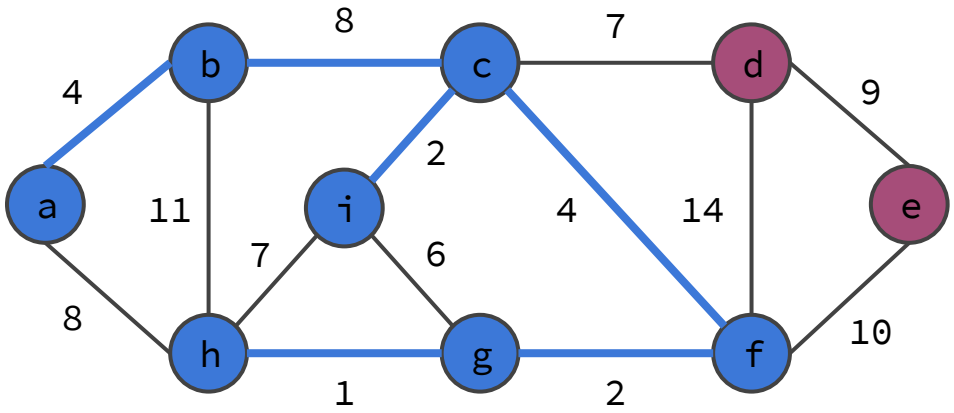
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	True
e	10	f	True
f	4	c	Extracted
g	2	f	Extracted
h	1	g	True
i	2	c	Extracted

Min Priority Queue: d e h

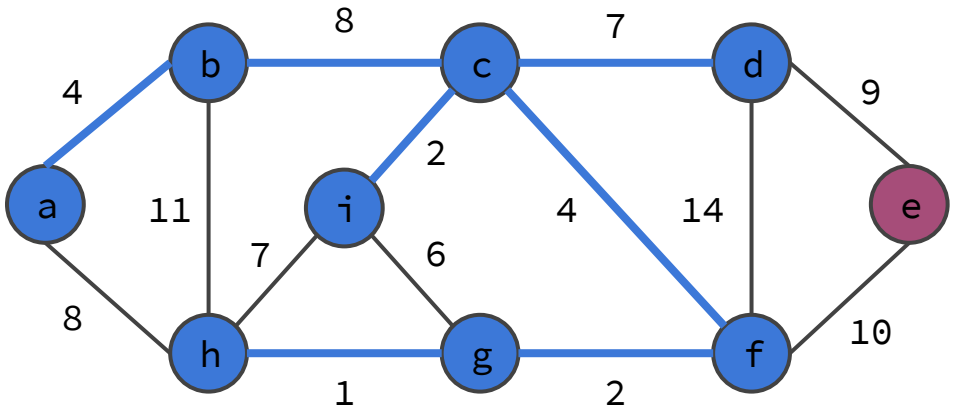
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	True
e	10	f	True
f	4	c	Extracted
g	2	f	Extracted
h	1	g	Extracted
i	2	c	Extracted

Min Priority Queue: d e

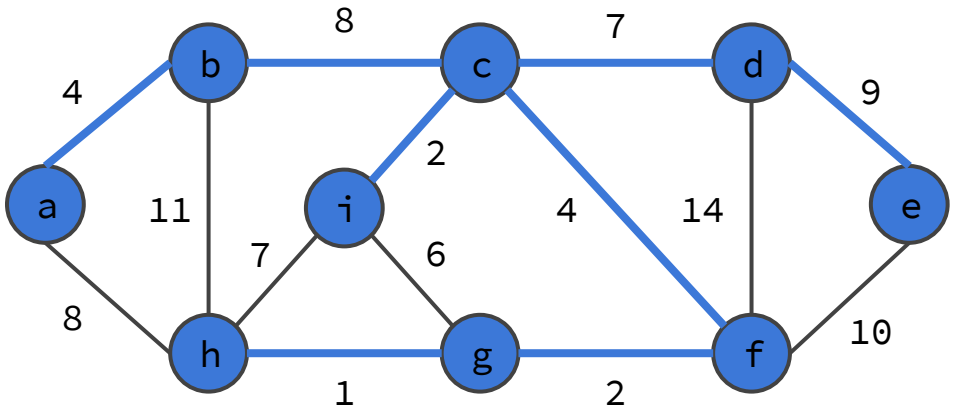
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	Extracted
e	9	d	True
f	4	c	Extracted
g	2	f	Extracted
h	1	g	Extracted
i	2	c	Extracted

Min Priority Queue: e

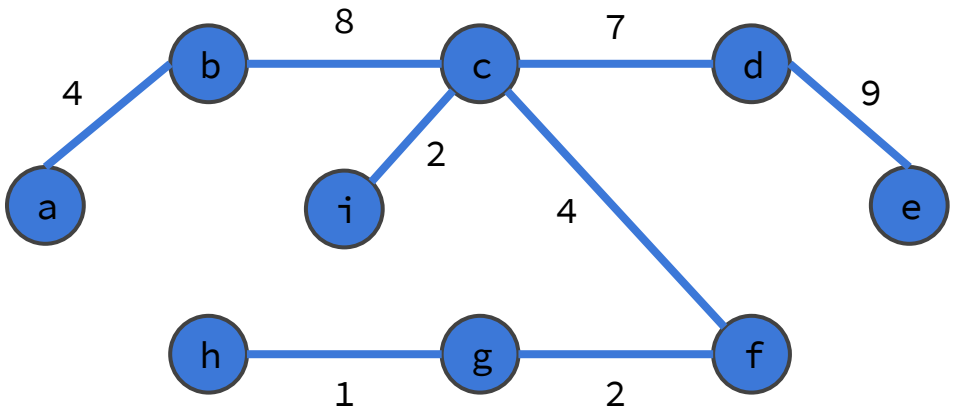
Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	Extracted
e	9	d	Extracted
f	4	c	Extracted
g	2	f	Extracted
h	1	g	Extracted
i	2	c	Extracted

Min Priority Queue:

Simulation of Prim's Algorithm



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	8	b	Extracted
d	7	c	Extracted
e	9	d	Extracted
f	4	c	Extracted
g	2	f	Extracted
h	1	g	Extracted
i	2	c	Extracted

Min Priority Queue:

Complexity Analysis of Prim's Algorithm

The running time of Prim's algorithm depends on how we implement the min priority queue Q. If we implement Q as a binary min-heap, we can use the Build_Min_Heap procedure to perform in O(V) time. The body of the while loop executes |V| times, and since each Extract_Min operation takes O(lgV) time, the total time for all calls to Extract_Min is O(VlgV). The main for loop run in O(E) times altogether, since the sum of the lengths of all adjacency lists is 2|E|. Within the for loop, we can implement the test for membership in Q in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q.

Complexity Analysis of Prim’s Algorithm Continued

The assignment involves an implicit Decrease_Key operation on the min-heap, which a binary min-heap supports in O(lgV) time. Thus, the total time for Prim’s algorithm is O(VlgV + ElgV) = O(ElgV), which is asymptotically the same as for our implementation of Kruskal’s algorithm. We can improve the asymptotic running time of Prim’s algorithm by using Fibonacci heaps. If a Fibonacci heap holds |V| elements, an Extract_Min operation takes O(lgV) amortized time and a Decrease_Key operation takes O(1) amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim’s algorithm improves to O(E + VlgV).

Fractional Knapsack

A thief breaks into a store holding a knapsack that can carry up to a maximum weight W > 0. The store contains items 1, 2, . . . , n, where item i has value v_i > 0 and weight w_i ≥ 0. The thief can steal some amount x_i of item i, where 0 ≤ x_i ≤ w_i. The value of this amount of item i is (xi/wi)*vi , i.e., the fraction of the item's weight stolen times the item's value. The thief must decide what fraction of each item to steal, so as to maximize the total value of the stolen goods, subject to the constraint that their total weight must not exceed the knapsack capacity W.

Fractional Knapsack Pseudocode

```
Fractional_Knapsack(w, v, W):
    sort the items so that v_1/w_1 ≥ v_2/w_2 ≥ . . . ≥ v_n/w_n
    S = 0 # total weight of stolen items in knapsack so far
    i = 1 # next item to be considered
    while i ≤ n and S + w_i ≤ W:
        x_i = w_i
        S = S + w_i
        i = i + 1
    if i ≤ n:
        x_i = W - S
        for j = i + 1 to n:
            x_j = 0
    return (x_1, . . . , x_n)
```

Fractional Knapsack Pseudocode

Item Number	1	2	3	4
Weight	10	30	50	70
Value	50	80	100	250
Value/Weight	5	2.67	2	3.57

Fractional Knapsack Pseudocode

Item Number	1	4	2	3
Weight	10	70	30	50
Value	50	250	80	100
Value/Weight	5	3.57	2.67	2

```
S = 0
W = 100
i = 1
n = 4
```

x	-	-	-	-

Fractional Knapsack Pseudocode

Item Number	1	4	2	3
Weight	10	70	30	50
Value	50	250	80	100
Value/Weight	5	3.57	2.67	2

```
S = 10
W = 100
i = 1
n = 4
```

x	10	-	-	-

Fractional Knapsack Pseudocode

Item Number	1	4	2	3
Weight	10	70	30	50
Value	50	250	80	100
Value/Weight	5	3.57	2.67	2

```
S = 80
W = 100
i = 2
n = 4
```

x	10	70	-	-

Fractional Knapsack Pseudocode

Item Number	1	4	2	3
Weight	10	70	30	50
Value	50	250	80	100
Value/Weight	5	3.57	2.67	2

```
S = 100
W = 100
i = 3
n = 4
```

x	10	70	20	-

$x_i = W - S_{previous}$

Fractional Knapsack Pseudocode

Item Number	1	4	2	3
Weight	10	70	30	50
Value	50	250	80	100
Value/Weight	5	3.57	2.67	2

```
S = 100
W = 100
i = 4
n = 4
```

x	10	70	20	0

```
Total Value  = 50 + 250 + 20/30 * 80
                          ↑
                        2.67
```

Complexity Analysis of Fractional Knapsack

In the beginning of the algorithm we have to sort the items by the ratio value/weight. As we have to sort fractional values so it will take O(nlgn) time. The rest of the algorithm runs in linear time. So, the running time of fractional knapsack is O(nlgn).

Optimal File Merge Patterns Continued

– – –

Huffman Coding

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string. Huffman encoding is an example of an algorithm where a greedy approach is successful.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies.

Huffman Coding Continued

Characters	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Huffman Coding Continued

That is, only 6 different characters appear, and the character a occurs 45,000 times. We have many options for how to represent such a file of information. Here, we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a codeword. If we use a fixed-length code, we need 3 bits to represent 6 characters: a = 000, b = 001, ..., f = 101. This method requires 300,000 bits to code the entire file.

Can we do better?

Huffman Coding Continued

A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires

(45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5+4)*1,000 = 224,000 bits

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix Code

A prefix code is a type of code system distinguished by its possession of the "prefix property", which requires that there is no whole code word in the system that is a prefix (initial segment) of any other code word in the system. We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called prefix codes. A prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.

Huffman Coding Pseudocode

```
Huffman_Compression(C):
     n = |C|
     Q = C
     for i = 1 to n - 1:
          allocate a new node z
          z.left = x = Extract_Min(Q)
          z.right = y = Extract_Min(Q)
          z.freq = x.freq + y.freq
          Insert(Q,z)
      return Extract_Min(Q)
```

Simulation of Huffman Coding

Characters	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

Huffman Coding Pseudocode

```
// root represents the root of Huffman Tree and S refers to bit-stream to be decompressed
Huffman_Decompression(root, S):
     n = |S|
     for i = 1 to n:
          current = root
          while current.left != NULL and current.right != NULL:
               if S[i] == '0':
                    current = current.left
               else:
                    current = current.right
               i = i + 1
          print current.symbol
```

Limitations of Greedy Algorithms

Sometimes greedy algorithms fail to find the globally optimal solution because they do not consider all the data. The choice made by a greedy algorithm may depend on choices it has made so far, but it is not aware of future choices it could make.

Optimal File Merge Patterns

Suppose, you have n sorted files. We have to find the minimum computations done to reach Optimal Merge Pattern. When two or more sorted files are to be merged all together to form a single file, the minimum number of computations done to reach this file are known as Optimal Merge Pattern.

If more than 2 files need to be merged then it can be done in pairs. For example, if need to merge 4 files A, B, C, D. First Merge A with B to get X1, merge X1 with C to get X2, merge X2 with D to get X3 as the output file.