Disjoint-set data structure: Union-find

Letícia Rodrigues Bueno

Federal University of ABC (UFABC)

Disjoint-set data structure: Introduction

Disjoint-set data structure: Introduction

- Applications that require grouping of n distinct elements in a collection of disjoint sets;

Disjoint-set data structure: Introduction

- Applications that require grouping of n distinct elements in a collection of disjoint sets;
- **Main Operations:**

Disjoint-set data structure: Introduction

- Applications that require grouping of n distinct elements in a collection of disjoint sets;
- **Main Operations:**
  1. finding the set to which an element belongs (**"find"**);

Disjoint-set data structure: Introduction

- Applications that require grouping of *n* distinct elements in a collection of disjoint sets;
- **Main Operations:**
  1. finding the set to which an element belongs (**“find”**);
  2. joining two distinct sets in one (**“union”**).

Disjoint-set Operations

Disjoint-set Operations

- Collection of disjoint dynamic sets: $S = \{s_1, s_2, \ldots, s_k\}$;

Disjoint-set Operations

- Collection of disjoint dynamic sets: $S = \{s_1, s_2, \ldots, s_k\}$;
- Each set is identified by a representative member (an element in the set);

Disjoint-set Operations

- Collection of disjoint dynamic sets: $S = \{s_1, s_2, \ldots, s_k\}$;
- Each set is identified by a representative member (an element in the set);
- **Other Operations:**

Disjoint-set Operations

- Collection of disjoint dynamic sets: $S = \{s_1, s_2, \ldots, s_k\}$;
- Each set is identified by a representative member (an element in the set);
- **Other Operations:**
  1. **makeSet(x)**: create sets with a single element x (representative member is x itself);

Disjoint-set Operations

- Collection of disjoint dynamic sets: $S = \{s_1, s_2, \ldots, s_k\}$;
- Each set is identified by a representative member (an element in the set);
- **Other Operations:**
  1. **makeSet(x)**: create sets with a single element x (representative member is x itself);
  2. **union(x, y)**: join dynamic sets S_x and S_y. Representative member is chosen for the new set;

Disjoint-set Operations

- Collection of disjoint dynamic sets: $S = \{s_1, s_2, \ldots, s_k\}$;
- Each set is identified by a representative member (an element in the set);
- **Other Operations:**
  1. **makeSet(x)**: create sets with a single element x (representative member is x itself);
  2. **union(x, y)**: join dynamic sets S_x and S_y. Representative member is chosen for the new set;
  3. **findSet(x)**: return representative member of the set containing x;

Implementation using Linked Lists

Implementation using Linked Lists

- each set is represented by a singly linked list;

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:
  1. an element of the set;

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:
  1. an element of the set;
  2. a pointer to the next node;

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:
  1. an element of the set;
  2. a pointer to the next node;
  3. a pointer to the representative member;

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:
  1. an element of the set;
  2. a pointer to the next node;
  3. a pointer to the representative member;
- **makeSet(x)**: time $O(1)$;

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:
  1. an element of the set;
  2. a pointer to the next node;
  3. a pointer to the representative member;
- **makeSet(x)**: time $O(1)$;
- **findSet(x)**: time $O(1)$;

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:
  1. an element of the set;
  2. a pointer to the next node;
  3. a pointer to the representative member;
- **makeSet(x)**: time $O(1)$;
- **findSet(x)**: time $O(1)$;
- **union(x, y)**:

Implementation using Linked Lists

- each set is represented by a singly linked list;
- first element of the list is the representative member;
- each node contains:
  1. an element of the set;
  2. a pointer to the next node;
  3. a pointer to the representative member;
- **makeSet(x)**: time $O(1)$;
- **findSet(x)**: time $O(1)$;
- **union(x, y)**: **??????**

Implementation using Linked Lists

Implementation of **union**$(\mathbf{x}, \mathbf{y})$:

Implementation using Linked Lists

Implementation of **union(x, y)**:

- list of x is attached to the end of the list of y;

Implementation using Linked Lists

Implementation of **union(x, y)**:

- list of x is attached to the end of the list of y;
- representative member of the new list is y;

Implementation using Linked Lists

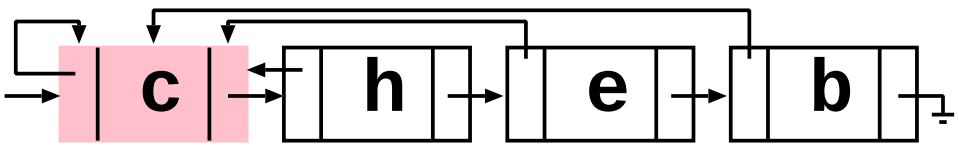
Implementation of **union(x, y)**:

- list of x is attached to the end of the list of y;
- representative member of the new list is y;
- nodes of x must point to the representative member y: cost is the length of the list of x;

Implementation using Linked Lists

Implementation of **union(x, y)**:

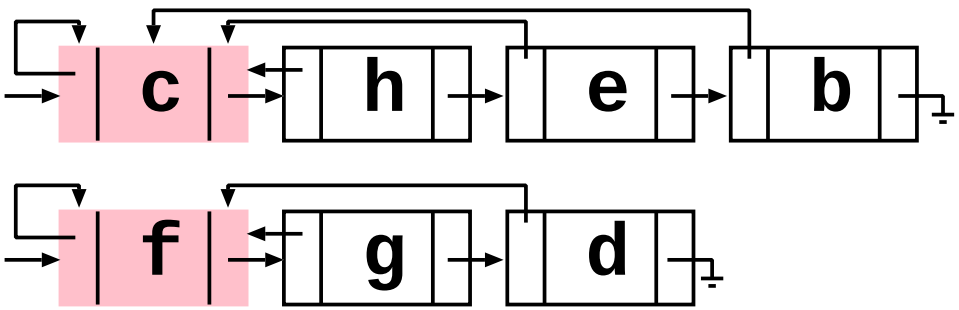
- list of *x* is attached to the end of the list of *y*;
- representative member of the new list is *y*;
- nodes of *x* must point to the representative member *y*: cost is the length of the list of *x*;



Implementation using Linked Lists

Implementation of **union(x, y)**:

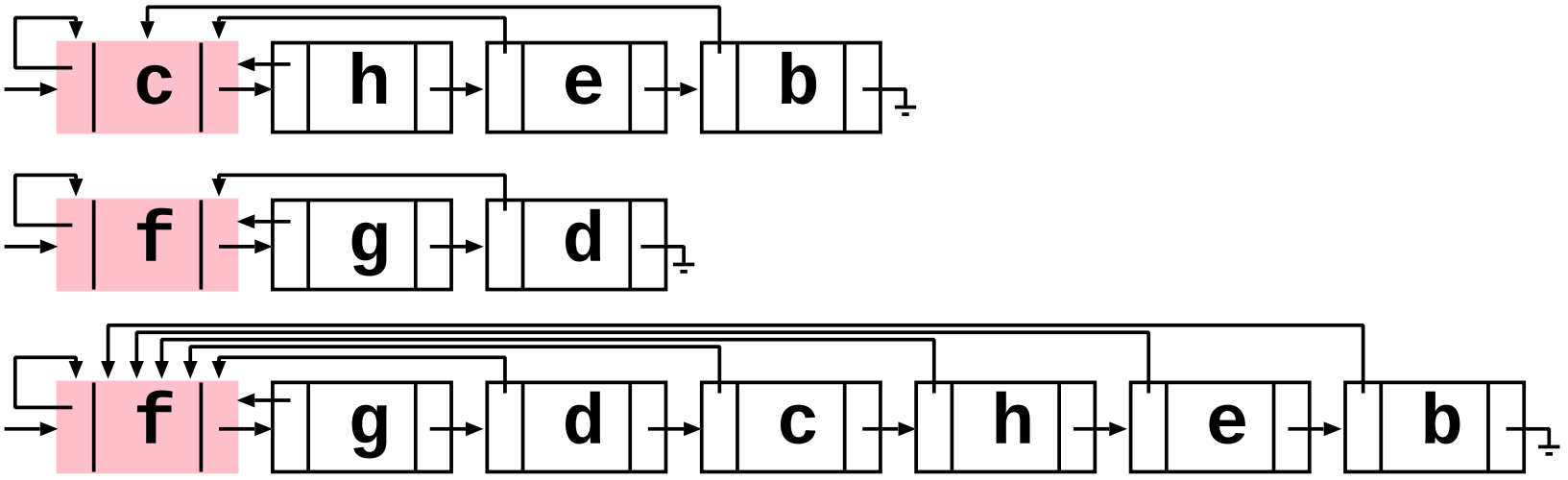
- list of *x* is attached to the end of the list of *y*;
- representative member of the new list is *y*;
- nodes of *x* must point to the representative member *y*: cost is the length of the list of *x*;



Implementation using Linked Lists

Implementation of **union(x, y)**:

- list of *x* is attached to the end of the list of *y*;
- representative member of the new list is *y*;
- nodes of *x* must point to the representative member *y*: cost is the length of the list of *x*;



Implementation using Linked Lists

Implementation of **union**$(\mathbf{x}, \mathbf{y})$:

Implementation using Linked Lists

Implementation of **union**(**x**, **y**):

- **using weighted-union heuristic:**

Implementation using Linked Lists

Implementation of **union**$(\mathbf{x}, \mathbf{y})$:

- **using weighted-union heuristic:** append the shorter list to the longer list;

Implementation using Disjoint-Sets Forests

Implementation using Disjoint-Sets Forests

- Implementation faster than if using linked lists;

Implementation using Disjoint-Sets Forests

- Implementation faster than if using linked lists;
- the asymptotically **faster** disjoint-sets data structure known so far;

Implementation using Disjoint-Sets Forests

- Implementation faster than if using linked lists;
- the asymptotically **faster** disjoint-sets data structure known so far;
- sets represented by rooted trees:

Implementation using Disjoint-Sets Forests

- Implementation faster than if using linked lists;
- the asymptotically **faster** disjoint-sets data structure known so far;
- sets represented by rooted trees:
  1. each node contains one element;

Implementation using Disjoint-Sets Forests

- Implementation faster than if using linked lists;
- the asymptotically **faster** disjoint-sets data structure known so far;
- sets represented by rooted trees:
  1. each node contains one element;
  2. each node points only to its parent;

Implementation using Disjoint-Sets Forests

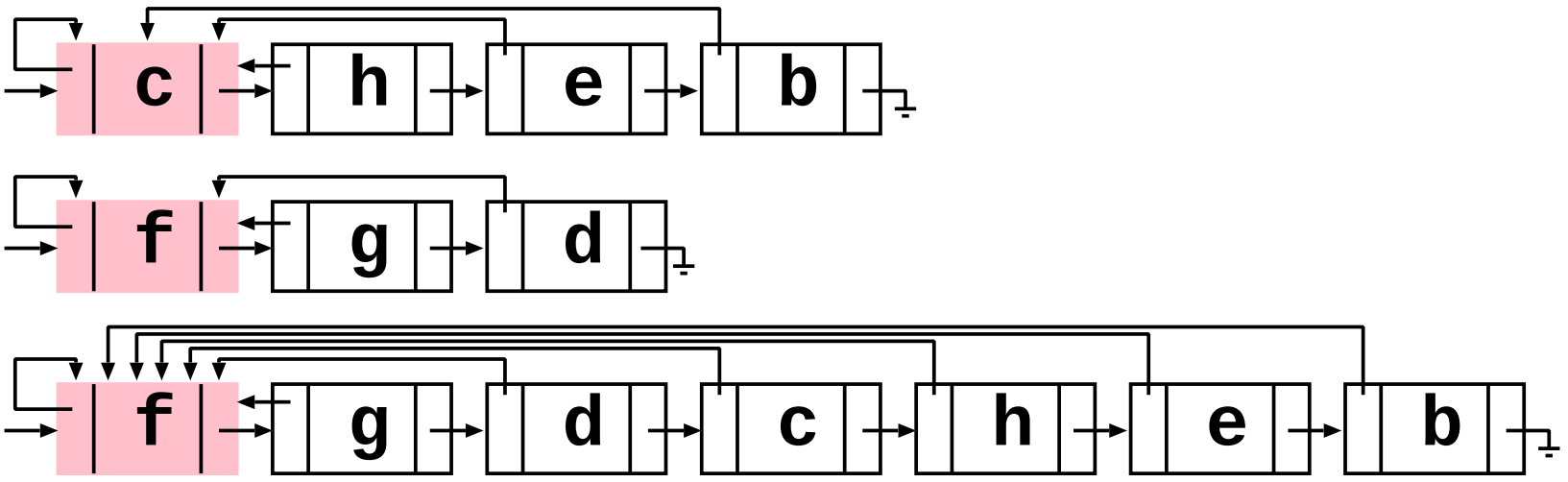
- Implementation faster than if using linked lists;
- the asymptotically **faster** disjoint-sets data structure known so far;
- sets represented by rooted trees:
  1. each node contains one element;
  2. each node points only to its parent;
  3. each tree represents a set;

Implementation using Disjoint-Sets Forests

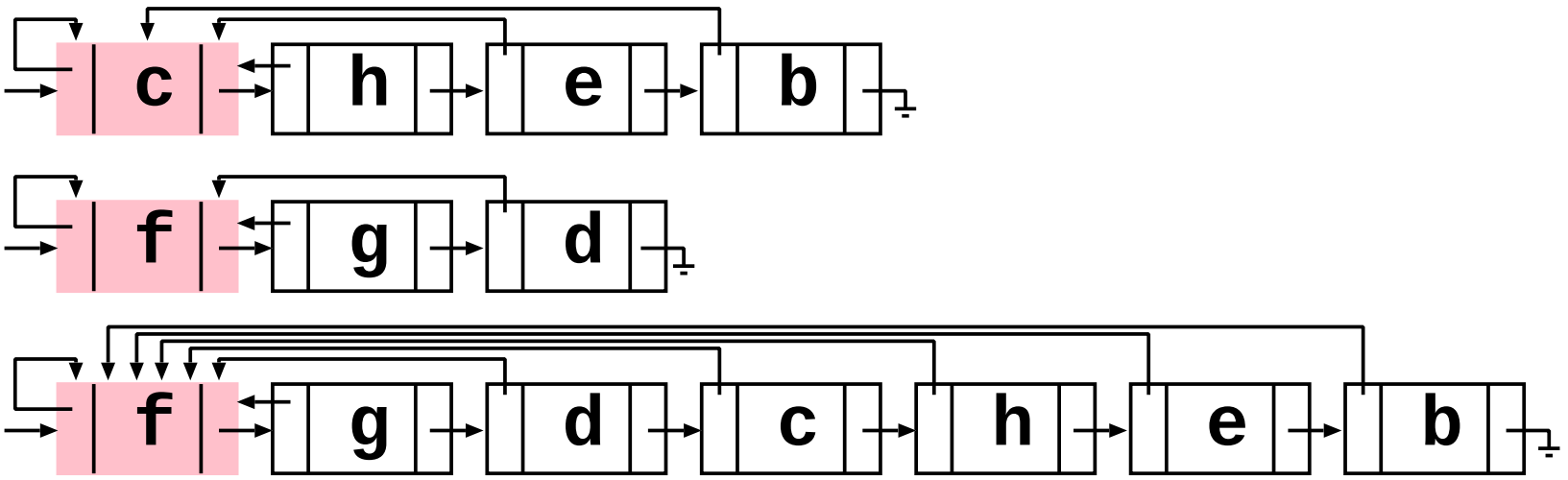
- Implementation faster than if using linked lists;
- the asymptotically **faster** disjoint-sets data structure known so far;
- sets represented by rooted trees:
  1. each node contains one element;
  2. each node points only to its parent;
  3. each tree represents a set;
  4. **representative member**: root of the tree;

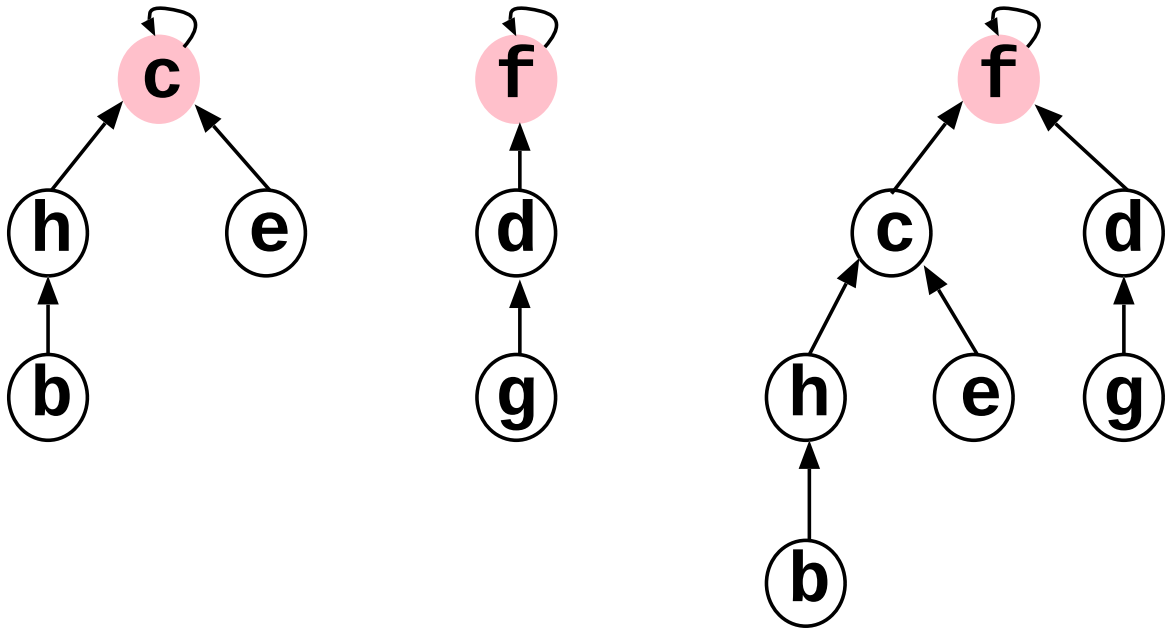
Implementation using Disjoint-Sets Forests

Implementation using Disjoint-Sets Forests



Implementation using Disjoint-Sets Forests





Implementation using Disjoint-Sets Forests

Implementation using Disjoint-Sets Forests

- **makeSet(x)**: create tree with a node containing x;

Implementation using Disjoint-Sets Forests

- **makeSet(x)**: create tree with a node containing x;
- **union(x, y)**: root of x points to the root of y;

Implementation using Disjoint-Sets Forests

- **makeSet(x)**: create tree with a node containing x;
- **union(x, y)**: root of x points to the root of y;
- **findSet(x)**: follow the pointers of parents upwards until representative element is reached at the root of the tree.

Implementation using Disjoint-Sets Forests

- **makeSet(x)**: create tree with a node containing x;
- **union(x, y)**: root of x points to the root of y;
- **findSet(x)**: follow the pointers of parents upwards until representative element is reached at the root of the tree.
- sequence of $n-1$ operations **union(x, y)** it can create tree that is a list;

Implementation using Disjoint-Sets Forests

- **makeSet(x)**: create tree with a node containing x;
- **union(x, y)**: root of x points to the root of y;
- **findSet(x)**: follow the pointers of parents upwards until representative element is reached at the root of the tree.
- sequence of $n-1$ operations **union(x, y)** it can create tree that is a list;
- two heuristics can be used to improve performance:

Implementation using Disjoint-Sets Forests

- **makeSet(x)**: create tree with a node containing x;
- **union(x, y)**: root of x points to the root of y;
- **findSet(x)**: follow the pointers of parents upwards until representative element is reached at the root of the tree.
- sequence of $n-1$ operations **union(x, y)** it can create tree that is a list;
- two heuristics can be used to improve performance:
  1. **union by rank**: root of the shorter tree points to the root of the taller tree;

Implementation using Disjoint-Sets Forests

- **makeSet(x)**: create tree with a node containing x;
- **union(x, y)**: root of x points to the root of y;
- **findSet(x)**: follow the pointers of parents upwards until representative element is reached at the root of the tree.
- sequence of $n-1$ operations **union(x, y)** it can create tree that is a list;
- two heuristics can be used to improve performance:
  1. **union by rank**: root of the shorter tree points to the root of the taller tree;
  2. **path compression**: each node points directly to the root;

Application: minimum spanning tree

Application: minimum spanning tree

- **electronic circuit design:** make component pins to be electrically equivalent by joining the wiring of all of them;

Application: minimum spanning tree

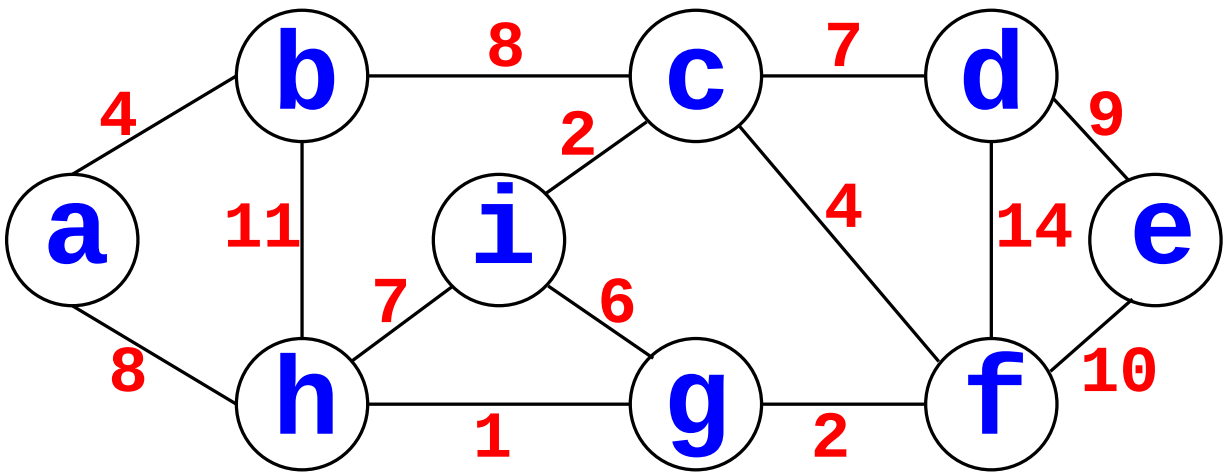
- **electronic circuit design:** make component pins to be electrically equivalent by joining the wiring of all of them;
- **problem:** optimize the number of wires;

Application: minimum spanning tree

- **electronic circuit design:** make component pins to be electrically equivalent by joining the wiring of all of them;
- **problem:** optimize the number of wires;
- **problem modeled as a graph:** minimum spanning tree;

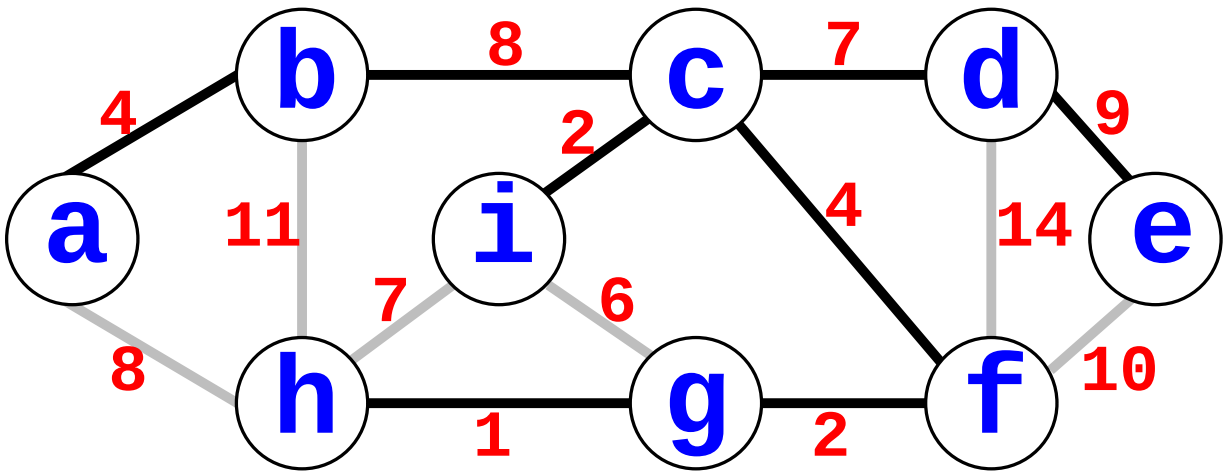
Application: minimum spanning tree

- **electronic circuit design:** make component pins to be electrically equivalent by joining the wiring of all of them;
- **problem:** optimize the number of wires;
- **problem modeled as a graph:** minimum spanning tree;



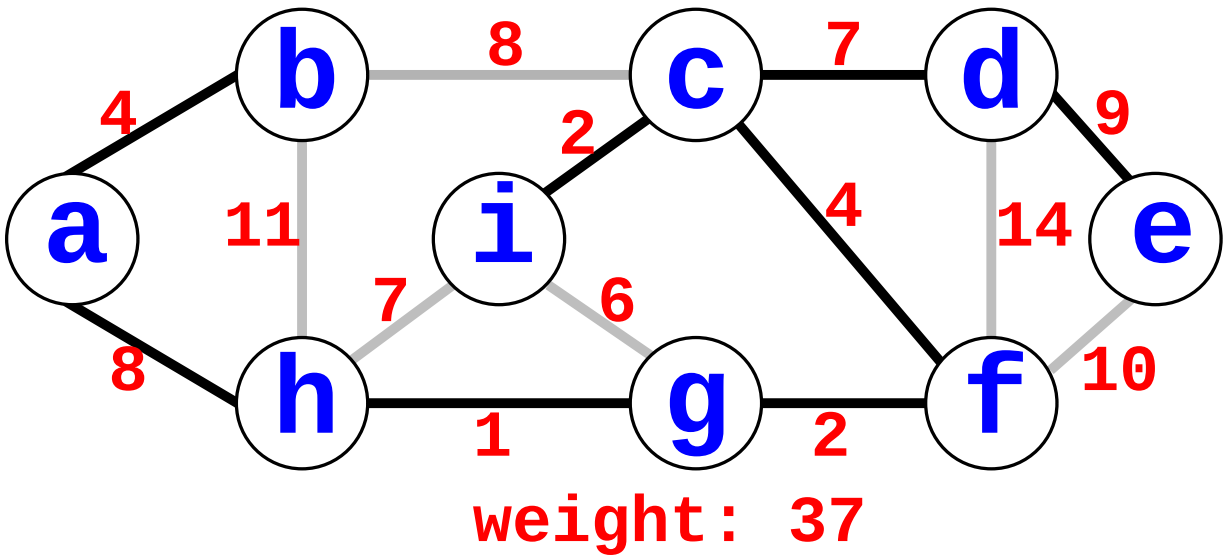
Application: minimum spanning tree

- **electronic circuit design:** make component pins to be electrically equivalent by joining the wiring of all of them;
- **problem:** optimize the number of wires;
- **problem modeled as a graph:** minimum spanning tree;



Application: minimum spanning tree

- **electronic circuit design:** make component pins to be electrically equivalent by joining the wiring of all of them;
- **problem:** optimize the number of wires;
- **problem modeled as a graph:** minimum spanning tree;



Minimum spanning tree: general algorithm

Minimum spanning tree: general algorithm

Greedy strategy:

Minimum spanning tree: general algorithm

Greedy strategy:

```
1  genericAlg(G):
2      A ← ∅
3      while A is not a spanning tree do
4          search a safe edge (u, v) for A
5          A ← A ∪ {(u, v)}
6      return A
```

Minimum spanning tree: general algorithm

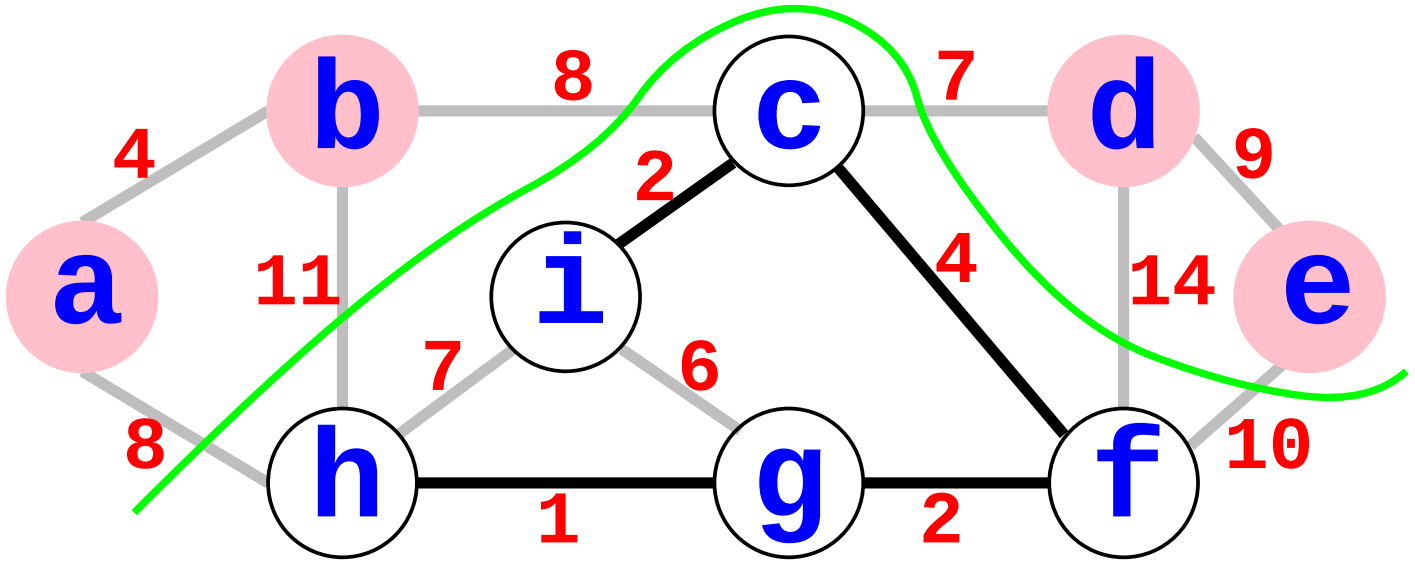
Greedy strategy:

```
1  genericAlg(G):
2      A ← ∅
3      while A is not a spanning tree do
4          search a safe edge (u, v) for A
5          A ← A ∪ {(u, v)}
6      return A
```

Safe edge: does not create cycles in A

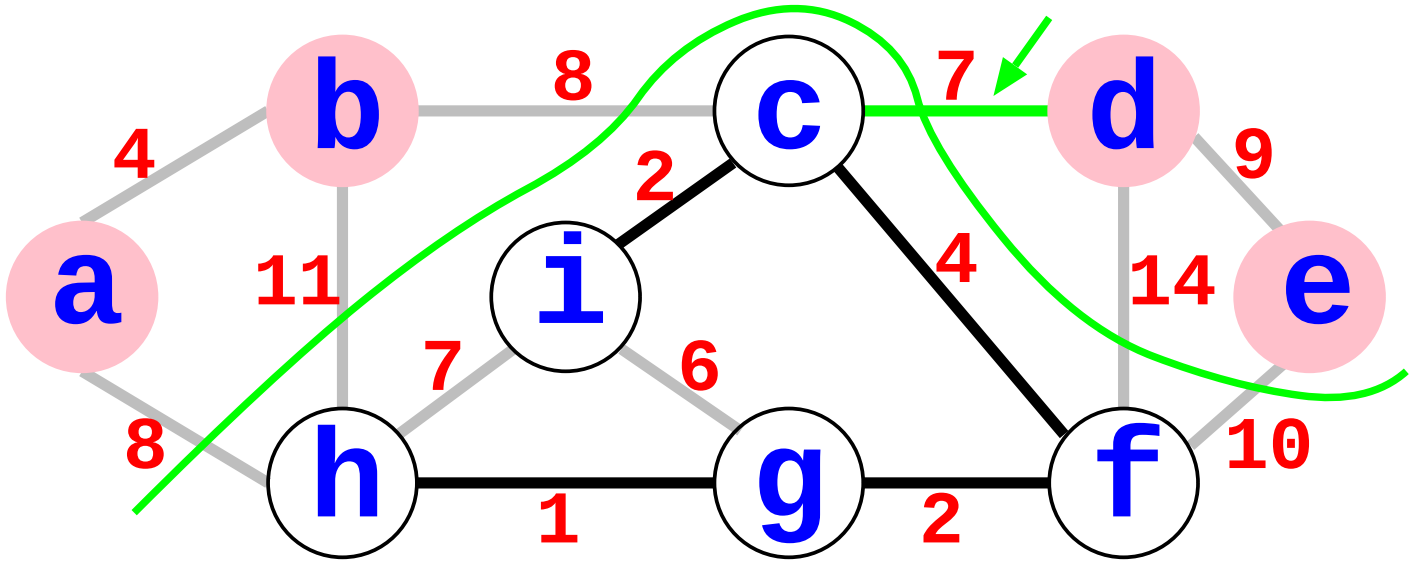
Minimum spanning tree: general algorithm

Light edge: edge with the minimum weight crossing the cut;

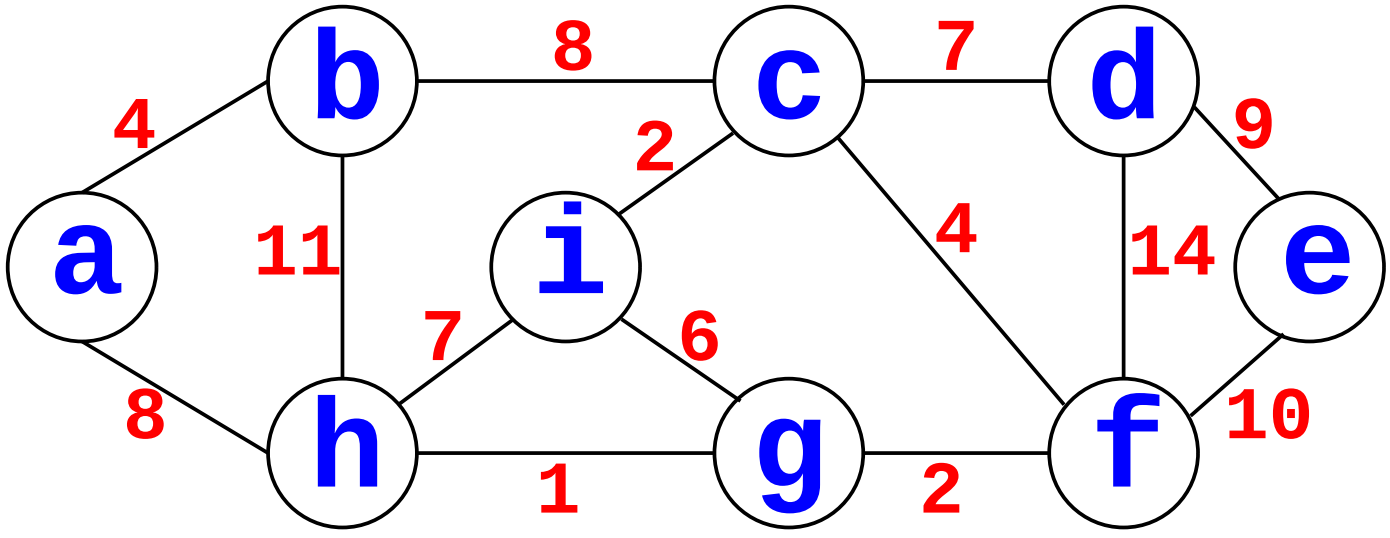


Minimum spanning tree: general algorithm

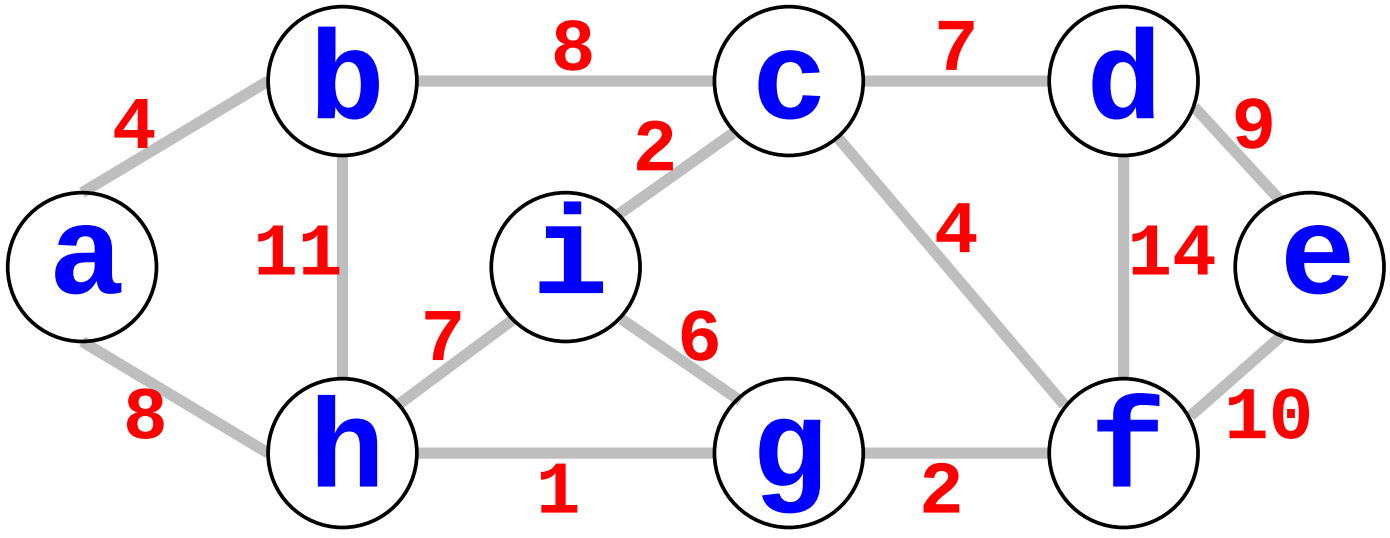
Light edge: edge with the minimum weight crossing the cut;



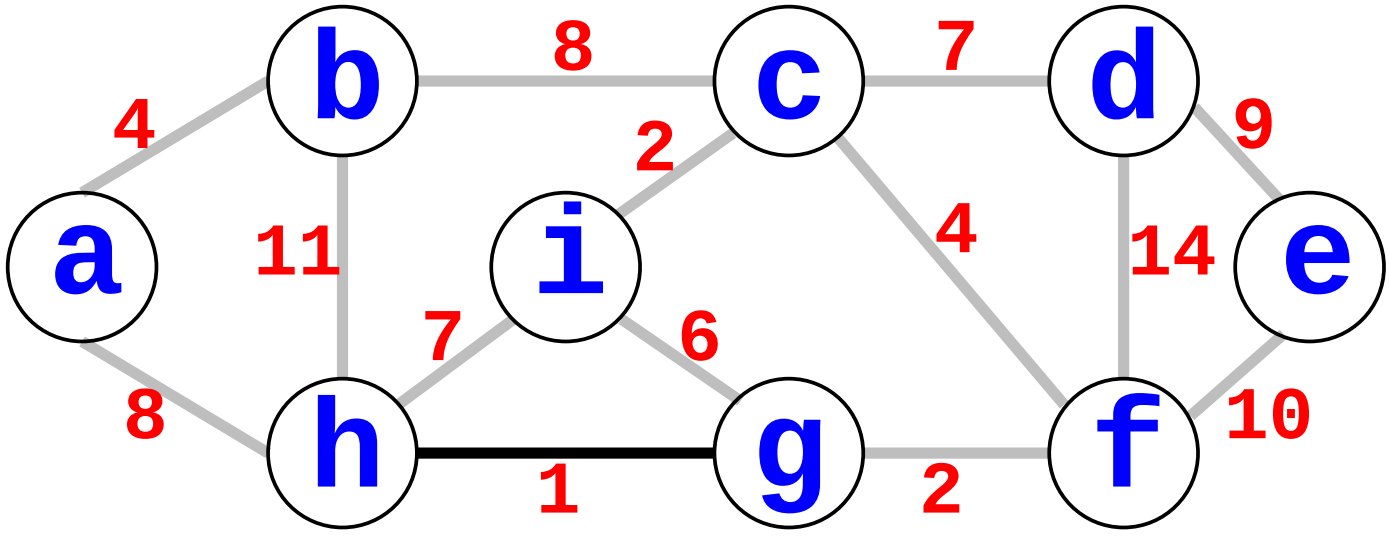
Minimum spanning tree: example



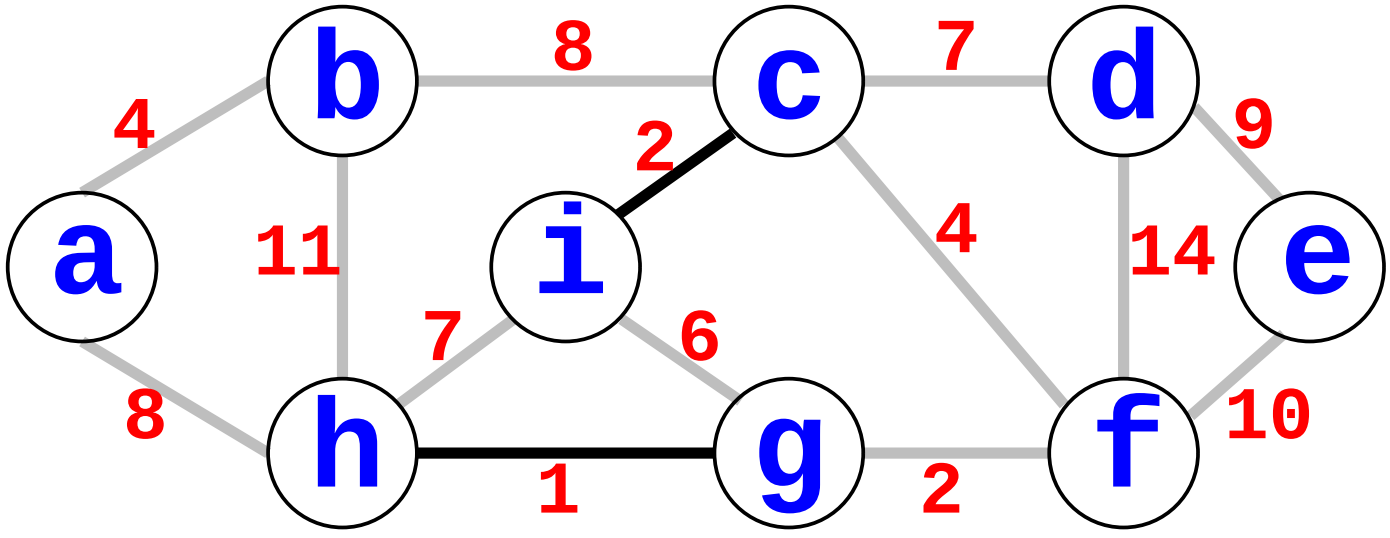
Minimum spanning tree: example



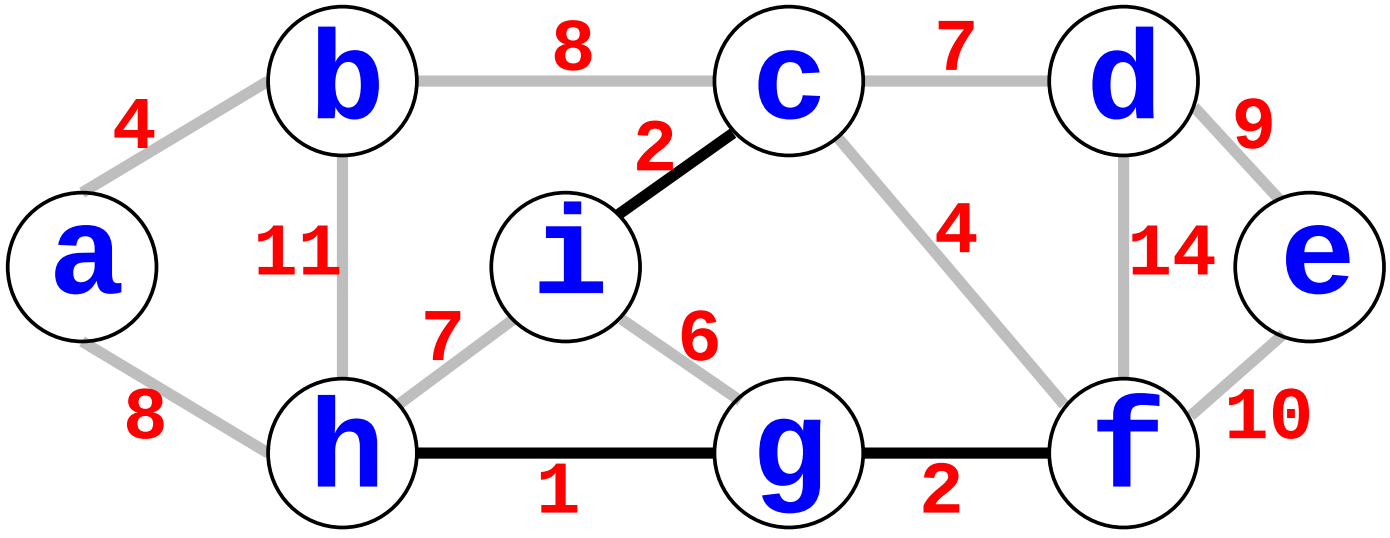
Minimum spanning tree: example



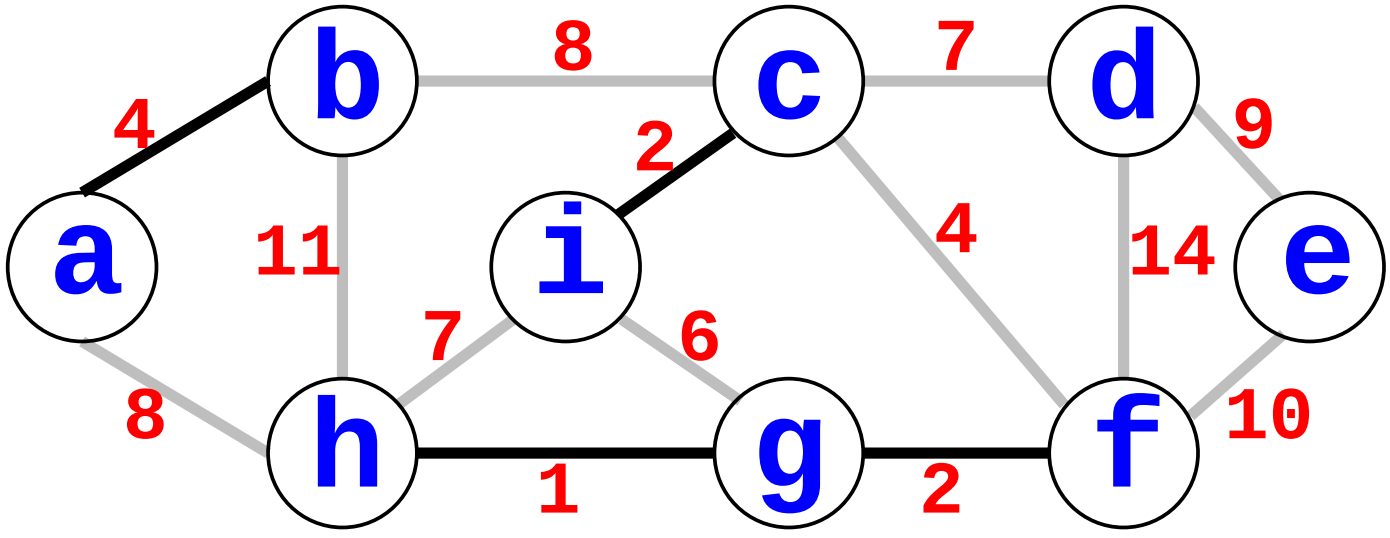
Minimum spanning tree: example



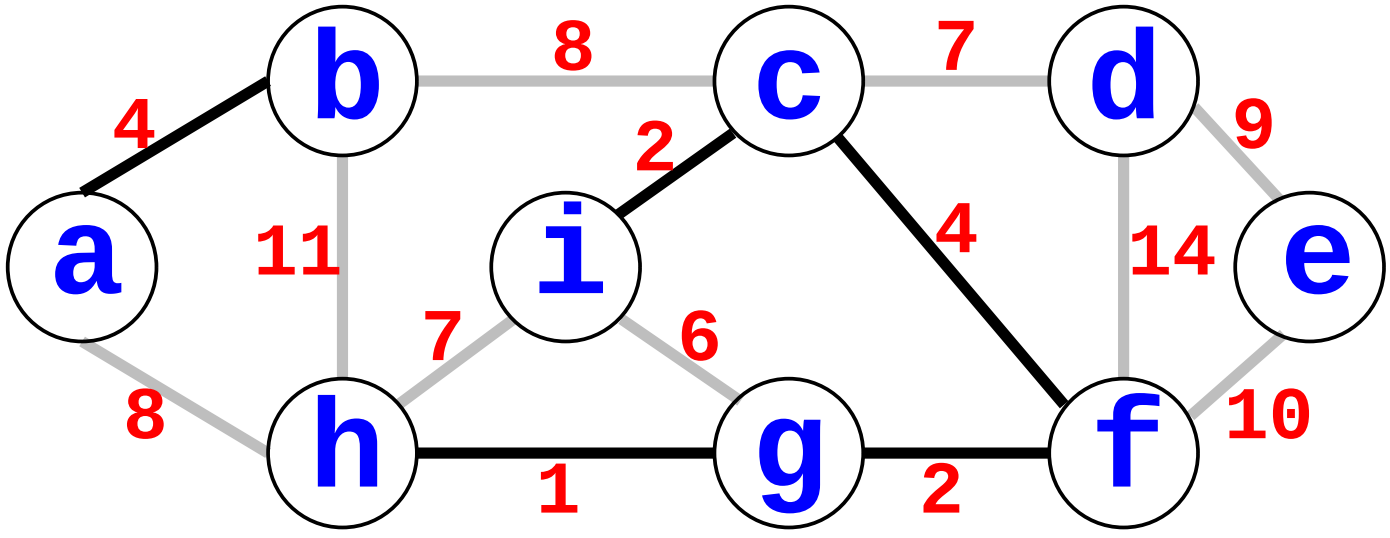
Minimum spanning tree: example



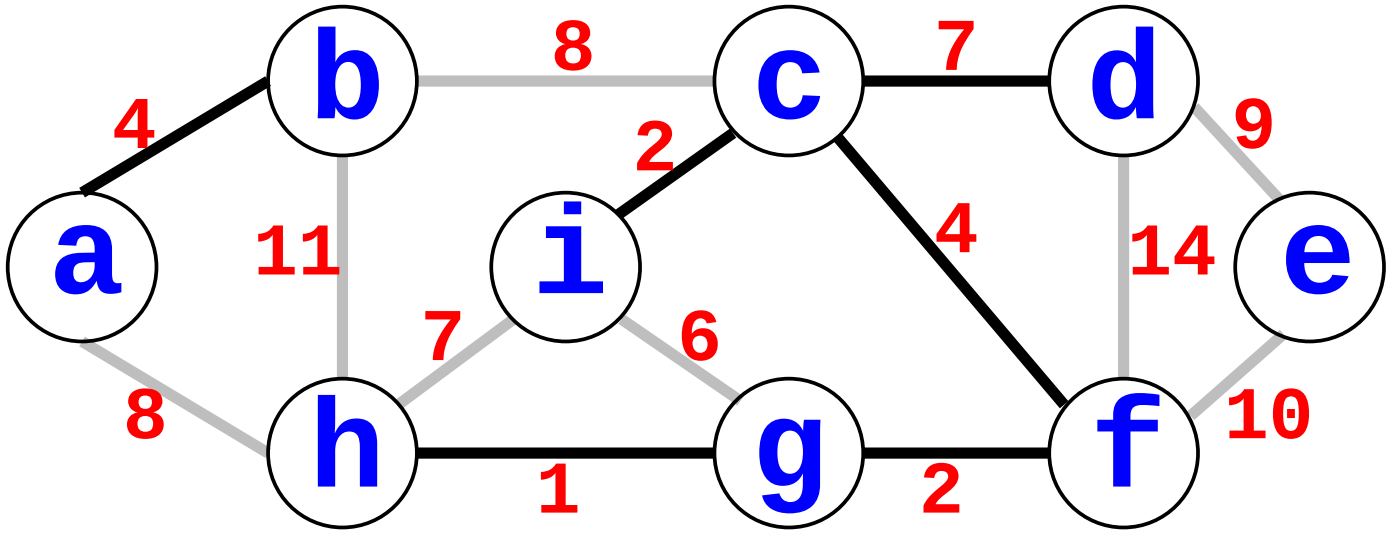
Minimum spanning tree: example



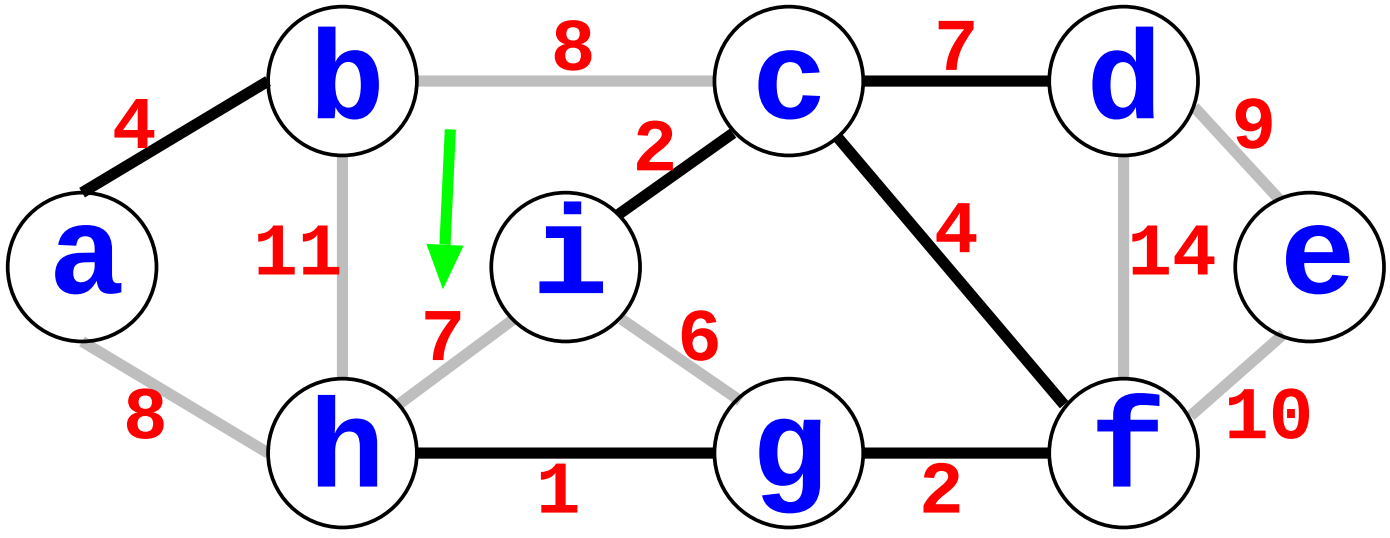
Minimum spanning tree: example



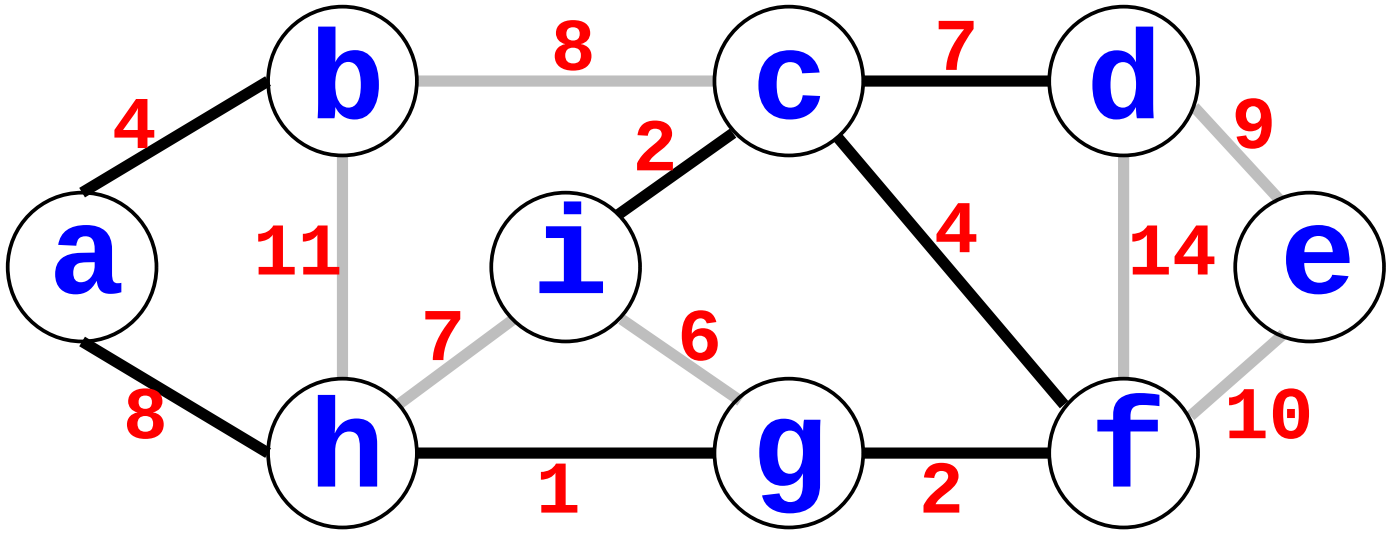
Minimum spanning tree: example



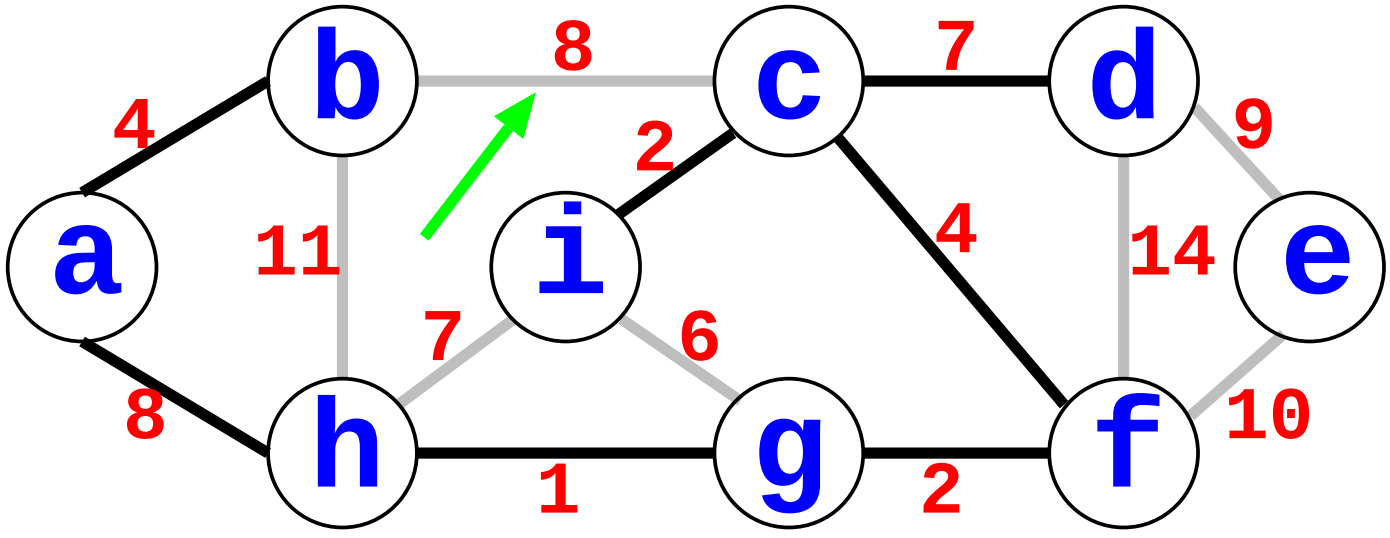
Minimum spanning tree: example



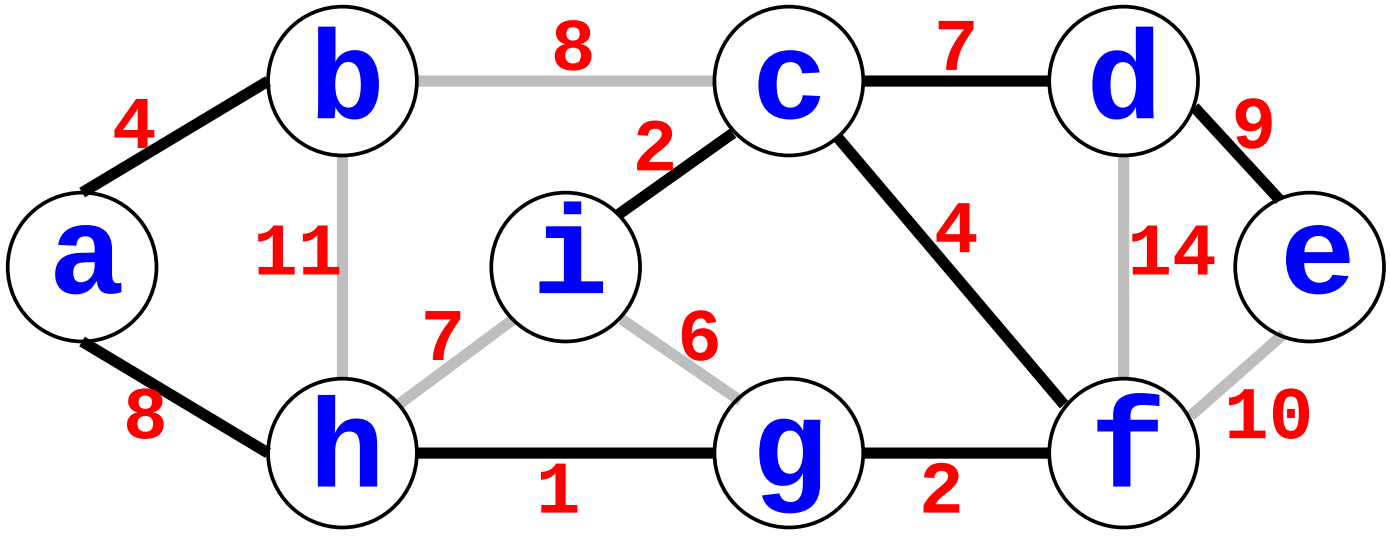
Minimum spanning tree: example



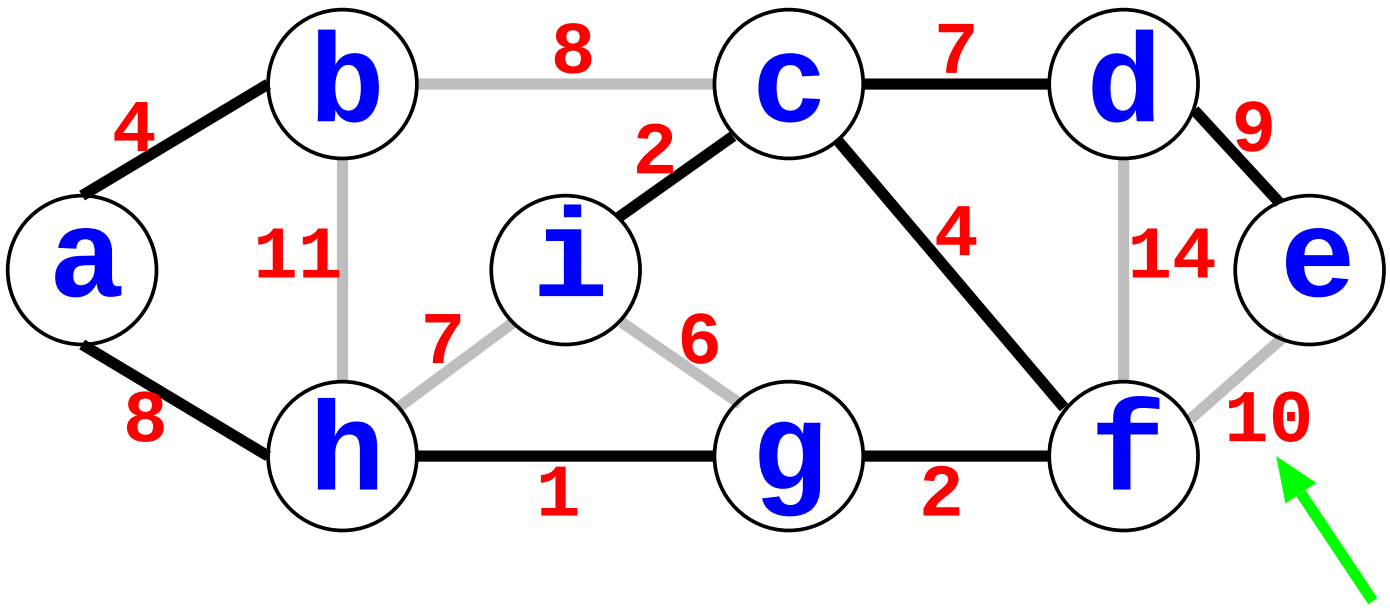
Minimum spanning tree: example



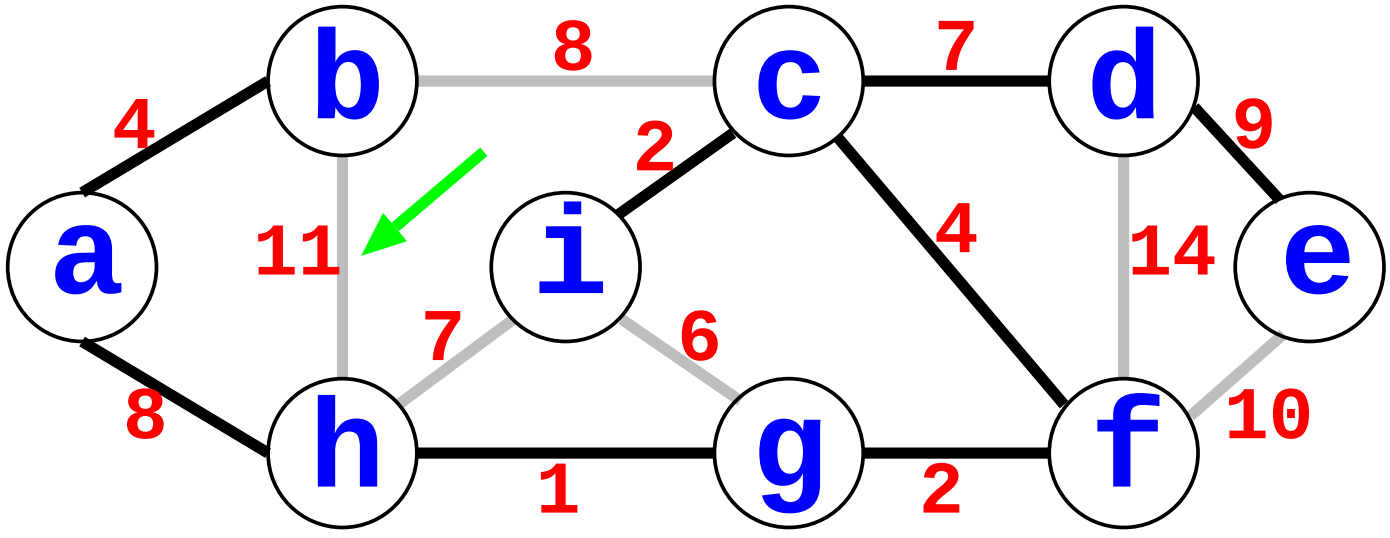
Minimum spanning tree: example



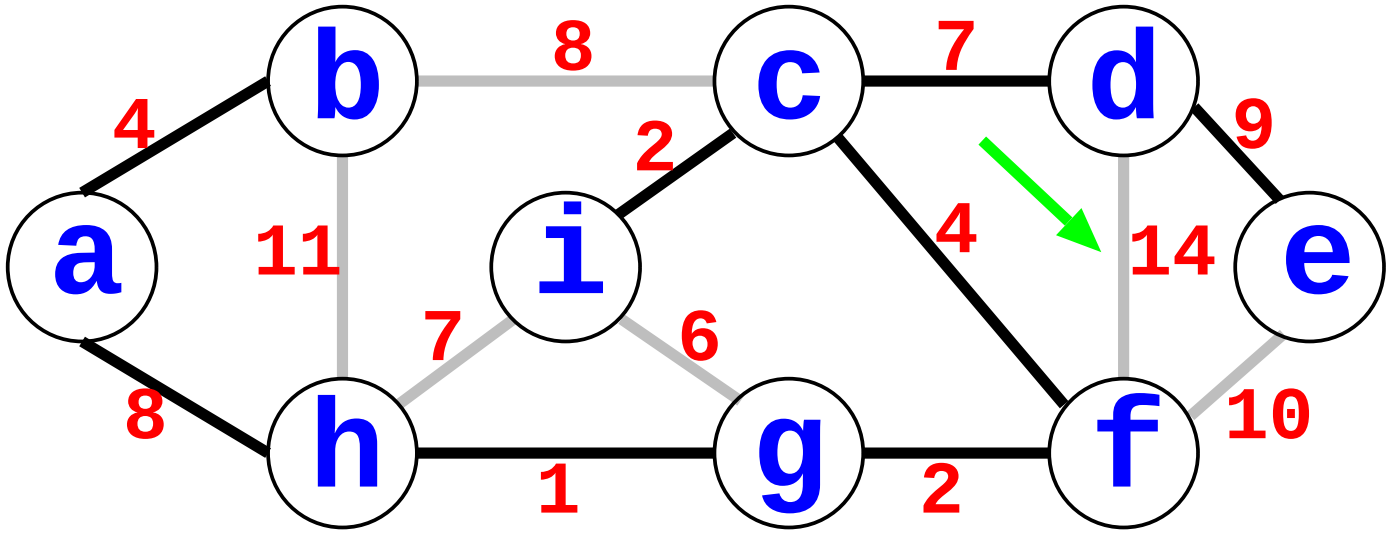
Minimum spanning tree: example



Minimum spanning tree: example



Minimum spanning tree: example



Minimum spanning tree: Kruskal's algorithm

Minimum spanning tree: Kruskal's algorithm

Implementation using union-find:

Minimum spanning tree: Kruskal's algorithm

Implementation using union-find:

```
1  kruskal(G):
2      A ← ∅
3      for each vertex v ∈ V(G) do
4          makeSet(v)
5      sort edges of E by their weight w in ascending order
6      for each edge (u, v) ∈ E(G) in ascending order do
7          if findSet(u) ≠ findSet(v) then
8              union(u, v)
9      return A
```

Exercises

1. Write a pseudocode for **makeSet**(**x**), **union**(**x**, **y**) and **findSet**(**x**) using the representation of linked lists and the weighted-union heuristic. Suppose each object x has an attribute *rep* pointing to the representative member of the set which contains x, and each set S has the attributes *begin*, *end* and *size*, where *size* is the length of the list.
2. The tree returned by Kruskal's algorithm is unique, i.e., the algorithm always returns the same minimum spanning tree? Prove or give a counterexample.

Bibliography

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. and STEIN, C. *Introduction to Algorithms*, 3rd edition, MIT Press, 2009.

ZIVIANI, N. Projeto de Algoritmos: com implementações em Java e C++, Cengage Learning, 2009. (in Portuguese)