SELF-BALANCING SEARCH TREES
Chapter Objectives

- To understand the impact that balance has on the performance of binary search trees
- To learn about the AVL tree for storing and maintaining a binary search tree in balance
- To learn about the Red-Black tree for storing and maintaining a binary search tree in balance
- If got time, will learn about 2-3 trees, 2-3-4 trees, and B-trees and how they achieve balance
- To understand the process of search and insertion in each of these trees and to be introduced to removal
Self-Balancing Search Trees

- The performance of a binary search tree is proportional to the height of the tree or the maximum number of nodes along a path from the root to a leaf.
- A full binary tree of height $k$ can hold $2^k - 1$ items.
- If a binary search tree is full and contains $n$ items, the expected performance is $O(\log n)$.
- However, if a binary tree is not full, the actual performance is worse than expected.
- To solve this problem, we introduce self-balancing trees to achieve a balance so that the heights of the right and left subtrees are equal or nearly equal.
- If we got time, we will also look non-binary search trees: the B-tree and its specializations, the 2-3 and 2-3-4 trees.
Tree Balance and Rotation

Section 11.1
Why Balance is Important

- Searches into this unbalanced search tree are $O(n)$, not $O(\log n)$

- A realistic example of an unbalanced tree
Rotation

- We need an operation on a binary tree that changes the relative heights of left and right subtrees, but preserves the binary search tree property.
AVL Trees

Section 11.2
AVL Trees

- In 1962 G.M. Adel'son-Vel'skiï and E.M. Landis developed a self-balancing tree. The tree is known by their initials: AVL.
- The AVL tree algorithm keeps track of the difference in height of each subtree.
- As items are added to or removed from a tree, the balance of each subtree from the insertion or removal point up to the root is updated.
- If the balance gets out of the range -1 to +1, the tree is rotated to bring it back into range.
Balancing a Left-Left Tree

Each light purple triangle represents a tree of height k
Balancing a Left-Left Tree (cont.)

The dark purple trapezoid represents an insertion into this tree, making its height $k + 1$. 
The heights of the left and right subtrees are unimportant; only the relative difference matters when balancing.

The formula $h_R - h_L$ is used to calculate the balance of each node.
Balancing a Left-Left Tree (cont.)

When the root and left subtree are both left-heavy, the tree is called a Left-Left tree.
Balancing a Left-Left Tree (cont.)

A Left-Left tree can be balanced by a rotation right.
Balancing a Left-Left Tree (cont.)
Balancing a Left-Right Tree

\[ k - (k + 2) \]

\[ (k + 1) - k \]

\[ +1 \]

\[ -2 \]
Balancing a Left-Right Tree (cont.)

A Left-Right tree cannot be balanced by a simple rotation right.
Balancing a Left-Right Tree (cont.)

Subtree b needs to be expanded into its subtrees $b_L$ and $b_R$. 
Balancing a Left-Right Tree (cont.)

40 is left-heavy. The left subtree now can be rotated left.
Balancing a Left-Right Tree (cont.)

The overall tree is now Left-Left and a rotation right will balance it.
Balancing a Left-Right Tree (cont.)
Balancing a Left-Right Tree (cont.)

In the previous example, an item was inserted in $b_L$. We now show the steps if an item was inserted into $b_R$ instead.
Balancing a Left-Right Tree (cont.)

Rotate the left subtree left.
Balancing a Left-Right Tree (cont.)

Rotate the tree right

Diagram:
- Node 50
- Node 40
- Node 25
- Node a
- Node b_L
- Node b_R
- Node c

Node values and rotations are indicated in the diagram.
Balancing a Left-Right Tree (cont.)
Four Kinds of Critically Unbalanced Trees

- Left-Left (parent balance is -2, left child balance is -1)
  - Rotate right around parent

- Left-Right (parent balance -2, left child balance +1)
  - Rotate left around child
  - Rotate right around parent

- Right-Right (parent balance +2, right child balance +1)
  - Rotate left around parent

- Right-Left (parent balance +2, right child balance -1)
  - Rotate right around child
  - Rotate left around parent
AVL Tree Example

- Build an AVL tree from the words in "The quick brown fox jumps over the lazy dog"
The overall tree is right-heavy (Right-Left)
parent balance = +2
right child balance = -1
1. Rotate right around the child
AVL Tree Example (cont.)

1. Rotate right around the child

The +2
brown +1
quick 0
AVL Tree Example (cont.)

The +2
brown +1
quick 0

1. Rotate right around the child
2. Rotate left around the parent
AVL Tree Example (cont.)

brown 0

The 0 quick 0

1. Rotate right around the child
2. Rotate left around the parent
AVL Tree Example (cont.)
AVL Tree Example (cont.)

```
Insert fox
```

![AVL Tree Diagram]

- **brown** +1
- **The** 0
- **quick** -1
- **fox** 0
AVL Tree Example (cont.)

The 0
  brown +1
  quick -1
  fox 0

Insert jumps
AVL Tree Example (cont.)

```
        brown +2
         /       \
        /         \
The 0        quick -2
         /       \
        /         \
   fox +1          \
            /      \
           /       \
      jumps 0   
```

Insert jumps
AVL Tree Example (cont.)

The tree is now left-heavy about quick (Left-Right case)
1. Rotate left around the child
AVL Tree Example (cont.)

1. Rotate left around the child
AVL Tree Example (cont.)

1. Rotate left around the child
2. Rotate right around the parent
AVL Tree Example (cont.)

1. Rotate left around the child
2. Rotate right around the parent
AVL Tree Example (cont.)

```
Insert over

brown +1

The 0 jumps 0

fox 0 quick 0
```
AVL Tree Example (cont.)

The 0 jumps +1

brown +2

fox 0 quick -1

over 0

Insert over
We now have a Right-Right imbalance
AVL Tree Example (cont.)

1. Rotate left around the parent
AVL Tree Example (cont.)

1. Rotate left around the parent
AVL Tree Example (cont.)

The fox jumps over the brown quick.
AVL Tree Example (cont.)

Insert the

```
         jumps 0
        /     \
brown 0   quick 0
 /     \        /     \
The 0  fox 0  over 0  the 0
```
AVL Tree Example (cont.)

The quick brown fox jumps over the lazy dog.

Insert lazy
AVL Tree Example (cont.)

The fox jumps over the lazy dog.
AVL Tree Example (cont.)

```
  jumps +1
     /    
brown 0   quick -1
     /    
The 0   fox 0   over -1   the 0
     |     /     |
 lazy 0 the 0
```

Insert dog

The fox jumps over the lazy dog.

0 0 0 -1 0
AVL Tree Example (cont.)

jumps 0

brown  +1  quick  -1

The  0  fox  -1  over  -1  the  0

dog  0  lazy  0

Insert dog
Section 11.3

Red-Black Trees
Red-Black Trees

- Rudolf Bayer developed the *Red-Black tree* as a special case of his B-tree.
- Leo Guibas and Robert Sedgewick refined the concept and introduced the color convention.
A Red-Black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Trees (cont.)

- Height is determined by counting only black nodes.
- A Red-Black tree is always balanced because the root node’s left and right subtrees must be the same height.
- By the standards of the AVL tree this tree is out of balance and would be considered a Left-Right tree.
- However, by the standards of the Red-Black tree it is balanced, because there are two black nodes (counting the root) in any path from the root to a leaf.
Insertion into a Red-Black Tree

- The algorithm follows the same recursive search process used for all binary search trees to reach the insertion point.
- When a leaf position is found, the new item is inserted and initially given the color red.
- If the parent is black, we are done; otherwise there is some rearranging to do.
- We introduce three situations ("cases") that may occur when a node is inserted; more than one can occur after an insertion.
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red
Insertion into a Red-Black Tree (cont.)

CASE 1

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

The root can be changed to black and still maintain invariant 4
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

The root can be changed to black and still maintain invariant 4
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Balanced tree
Insertion into a Red-Black Tree (cont.)

CASE 2

20

30

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
CASE 2

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red
Insertion into a Red-Black Tree (cont.)

Case 2

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red
There is one black node on the right and none on the left, which violates invariant 4

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 2

Rotate left around the grandparent to correct this
Insertion into a Red-Black Tree (cont.)

CASE 2

30

20  35

Rotate left around the grandparent to correct this

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

CASE 2

Balanced tree

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

CASE 3

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red.

CASE 3
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red
Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

A rotation left does not fix the violation of #4
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

A rotation left does not fix the violation of #4
Insertion into a Red-Black Tree (cont.)

Case 3

Insertion into a Red-Black Tree

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Back-up to the beginning (don't perform rotation or change colors)
Insertion into a Red-Black Tree (cont.)

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Rotate right about the parent so that the red child is on the same side of the parent as the parent is to the grandparent.
Insertion into a Red-Black Tree (cont.)

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate right about the parent so that the red child is on the same side of the parent as the parent is to the grandparent
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

NOW, change colors
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Now, change colors
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

and rotate left . . .
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

and rotate left...
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

The problem has now shifted up the tree
Insertion into a Red-Black Tree (cont.)

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

We cannot change 2 to black because its sibling 14 is already black (both siblings have to be red (unless there is no sibling) to do the color change
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

We need to rotate left around 2 so that the red child is on the same side of the parent as the parent is to the grandparent

CASE 3
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

We need to rotate left around 2 so that the red child is on the same side of the parent as the parent is to the grandparent.
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate right around 11 to restore the balance
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate right around 11 to restore the balance
Insertion into a Red-Black Tree (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Balanced tree
Build a Red-Black tree for the words in "The quick brown fox jumps over the lazy dog"
The quick Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
The quick brown

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate so that the child is on the same side of its parent as its parent is to the grandparent

CASE 3
The quick brown

Case 3

Change colors

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Rotate left
Red-Black Tree Example (cont.)

The

quick

brown

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `NULL` pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

The quick brown fox's parent and its parent's sibling are both red. Change colors.
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

We can change brown's color to black and not violate #4
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

We can change brown's color to black and not violate #4
Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate so that red child is on same side of its parent as its parent is to the grandparent

CASE 3
Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Change fox’s parent and grandparent colors
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Change fox’s parent and grandparent colors
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Rotate right about quick
Red-Black Tree Example (cont.)

The brown jumps fox quick

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate right about quick

CASE 3
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

Change colors of parent, parent's sibling and grandparent
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

Change colors of parent, parent's sibling and grandparent
Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
**Red-Black Tree Example (cont.)**

**Invariants:**
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a **null** pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

---

**CASE 1**

Because over and the are both red, change parent, parent's sibling and grandparent colors.
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 2
Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 2
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

The fox jumps over the lazy dog
Red-Black Tree Example (cont.)

Invariants:
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a NULL pointer is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same
Chapter Objectives

- To become familiar with graph terminology and the different types of graphs
- To study the Graph ADT and different implementations of the Graph ADT
- To learn the breadth-first and depth-first search traversal algorithms
- To learn some algorithms involving weighted graphs
- To study some applications of graphs and graph algorithms
Graphs

- Trees are limited in that a data structure can have only one parent
- Graphs overcome this limitation
- Graphs were studied long before computers were invented, but associated algorithms were impractical before the advent of computers
- Graphs algorithms facilitate
  - large communication networks
  - the software that makes the Internet function
  - programs to determine optimal placement of components on a silicon chip
- Graphs describe
  - roads maps
  - airline routes
  - course prerequisites
Graph Terminology

Section 12.1
Graph Terminology

- A graph is a data structure that consists of a set of vertices (or nodes) and a set of edges (relations) between pairs of vertices.
- Edges represent paths or connections between vertices.
- Both the set of vertices and the set of edges must be finite.
- Either set may be empty (if the set of vertices is empty, the set of edges also must be empty).
- We restrict our discussion to simple graphs in which there is at least one edge from a given vertex to another vertex.
Visual Representation of Graphs

- Vertices are represented as points or labeled circles and edges are represented as line segments joining the vertices.

\[ V = \{A, B, C, D, E\} \]
\[ E = \{\{A, B\}, \{A, D\}, \{C, E\}, \{D, E\}\} \]
Visual Representation of Graphs (cont.)

- Each edge is represented by the two vertices it connects.
- If there is an edge between vertices $x$ and $y$, there is a path from $x$ to $y$ and vice versa.
The physical layout of the vertices and their labeling is not relevant.

\[ V = \{0, 1, 2, 3, 4, 5, 6\} \]
\[ E = \{\{0, 1\}, \{0, 2\}, \{0, 5\}, \{0, 6\}, \{3, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\} \]
Directed and Undirected Graphs

- The edges of a graph are directed if the existence of an edge from A to B does not necessarily guarantee that there is an edge from B to A.
- A graph with directed edges is called a directed graph or digraph.
- A graph with undirected edges is an undirected graph, or simply a graph.
Directed and Undirected Graphs (cont.)

- A directed edge is like a one-way street; you can travel in only one direction.
- Directed edges are represented as lines with an arrowhead on one end (undirected edges do not have an arrowhead at either end).
- Directed edges are represented by ordered pairs of vertices \(\{\text{source}, \text{destination}\}\); the edges for the digraph on this slide are:

\[
E = \{\{A, B\}, \{B, A\}, \{B, E\}, \{D, A\}, \{E, A\}, \{E, C\}, \{E, D\}\}
\]
Directed and Undirected Graphs (cont.)

- The edges in a graph may have associated values known as weights.
- A graph with weighted edges is known as a weighted graph.
Paths and Cycles

• The following definitions describe pathways between vertices
A vertex is adjacent to another vertex if there is an edge to it from that other vertex.
A vertex is adjacent to another vertex if there is an edge to it from that other vertex.
A vertex is adjacent to another vertex if there is an edge to it from that other vertex.
A vertex is adjacent to another vertex if there is an edge to it from that other vertex.

A is adjacent to D, but D is NOT adjacent to A.
A path is a sequence of vertices in which each successive vertex is adjacent to its predecessor.
A path is a sequence of vertices in which each successive vertex is adjacent to its predecessor.
In a simple path, the vertices and edges are distinct except that the first and last vertex may be the same.
In a simple path, the vertices and edges are distinct except that the first and last vertex may be the same.
In a simple path, the vertices and edges are distinct except that the first and last vertex may be the same.

This path is NOT a simple path.
A cycle is a simple path in which only the first and final vertices are the same.
A cycle is a simple path in which only the first and final vertices are the same.

In an undirected graph a cycle must contain at least three distinct vertices; Pittsburgh → Columbus → Pittsburgh is not a cycle.
An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.
An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.
Paths and Cycles (cont.)

An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.
An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.
An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.
An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.

This graph is NOT a connected graph.
If a graph is not connected, it is considered *unconnected*, but still consists of connected components.
If a graph is not connected, it is considered unconnected, but will still consist of connected components.

{4, 5} are connected components.
If a graph is not connected, it is considered unconnected, but will still consist of connected components.

{6, 7, 8, 9} are connected components.
If a graph is not connected, it is considered *unconnected*, but will still consist of connected components.

A single vertex with no edge is considered a connected component.
A tree is a special case of a graph.

Any graph that is:
- connected
- contains no cycles

can be viewed as a tree by making one of the vertices the root.
Graph Applications

- Graphs can be used to:
  - determine if one node in a network is connected to all the others
  - map out multiple course prerequisites (a solution exists if the graph is a directed graph with no cycles)
  - find the shortest route from one city to another (least cost or shortest path in a weighted graph)
Implementing the Graph ADT

Section 12.3
Implementing the Graph ADT

- Many of the original publications of graph algorithms and their implementations did not use an object-oriented approach or even abstract data types.

- Two representations of graphs are most common:
  - Edges are represented by an array of lists called adjacency lists, where each list stores the vertices adjacent to a particular vertex.
  - Edges are represented by a two-dimensional array, called an adjacency matrix, with $|V|$ rows and $|V|$ columns.
Adjacency List

- An adjacency list representation of a graph uses an array of lists - one list for each vertex
- The vertices are in no particular order
- In each node, only the destination vertex as shown as the value field
  - In the actual implementation, the entire Edge is stored
- Symmetric entries are required for an undirected graph
  - If $[u,v]$ is an edge, then $v$ appears in the adjacency list for $u$ and $u$ appears in the adjacency list for $v$
Adjacency List – Directed Graph

Example
Adjacency List – Undirected Graph

Example
Adjacency Matrix

- Use a two-dimensional array to represent the graph.
- For an unweighted graph, the entries can be `bool` or `int` values:
  - `true` or `1`, if the edge exists
  - `false` or `0`, if the edge does not exist
- Integer values have benefits over boolean values for some graph algorithms that use matrix multiplication.
For a weighted graph, the matrix would contain the weights

- Since 0 is a valid weight, `numeric_limits<double>::infinity()` (a special double value in C++ that approximates the mathematical behavior of infinity) can represent the absence of an edge

- An unweighted graph would contain the value 1.0 to indicate the presence of an edge

- In an undirected graph, the matrix is symmetric, so only the lower diagonal of the matrix needs to be saved (an example is on the following slide)
Adjacency Matrix (cont.)
Traversals of Graphs

Section 12.4
Traversals of Graphs

- Most graph algorithms involve visiting each vertex in a systematic order.
- As with trees, there are different ways to do this.
- The two most common traversal algorithms are the breadth-first search and the depth-first search.
In a breadth-first search,
- visit the start node first,
- then visit all nodes that are adjacent to it,
- then visit all nodes that can be reached by a path from the start node containing two edges,
- then visit all nodes that can be reached by a path from the start node containing three edges,
- and so on

We must visit all nodes for which the shortest path from the start node is length $k$ before we visit any node for which the shortest path from the start node is length $k+1$
Breadth-First Search, (cont.)

- Visualize a breadth-first traversal by “picking up” the graph at the vertex that is the start node, with the remaining nodes suspended beneath it, connected by their edges.
  - Nodes that are “higher” are visited before nodes that are “lower”.
- There is no special start vertex – we arbitrarily choose the vertex with label 0.
If the graph is not connected, the process is repeated for the unconnected component(s) by selecting an unidentified vertex.
Example of a Breadth-First Search
Example of a Breadth-First Search (cont.)

Select the start node

0 unvisited  0 visited  0 identified
Example of a Breadth-First Search (cont.)

While visiting it, identify its adjacent nodes.
Example of a Breadth-First Search (cont.)

Identify its adjacent nodes and add them to a queue of identified nodes.

Visit sequence:
0

Diagram:
- Node 0
  - Adjacent nodes: 1, 2
- Node 1
  - Adjacent nodes: 0, 2, 4
- Node 2
  - Adjacent nodes: 1, 3
- Node 3
  - Adjacent nodes: 2
- Node 4
  - Adjacent nodes: 1, 5
- Node 5
  - Adjacent nodes: 4
- Node 6
  - Adjacent nodes: 5, 7
- Node 7
  - Adjacent nodes: 6
- Node 8
  - Adjacent nodes: 4
- Node 9
  - Adjacent nodes: 2
Example of a Breadth-First Search (cont.)

Queue:
1, 3

Visit sequence:
0

Identify its adjacent nodes and add them to a queue of identified nodes.
Example of a Breadth-First Search (cont.)

Queue:
1, 3

Visit sequence:
0

Color the node as visited
Example of a Breadth-First Search (cont.)

The queue determines which nodes to visit next.

Queue: 1, 3

Visit sequence: 0

0  unvisited  0  visited  0  identified
Visit the first node in the queue, 1

Queue: 1, 3
Visit sequence: 0

Example of a Breadth-First Search (cont.)
Visit the first node in the queue, 1

Queue:
3

Visit sequence:
0, 1
Example of a Breadth-First Search (cont.)

Select all its adjacent nodes that have not been visited or identified

Queue: 3

Visit sequence: 0, 1
Example of a Breadth-First Search (cont.)

Select all its adjacent nodes that have not been visited or identified.

Queue: 3, 2, 4, 6, 7

Visit sequence: 0, 1
Example of a Breadth-First Search (cont.)

Now that we are done with 1, we color it as visited

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1
Example of a Breadth-First Search (cont.)

and then visit the next node in the queue, 3 (which was identified in the first selection)

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1
Example of a Breadth-First Search (cont.)

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3

and then visit the next node in the queue, 3 (which was identified in the first selection)

0 unvisited  0 visited  0 identified
Example of a Breadth-First Search (cont.)

3 has two adjacent vertices. 0 has been visited already and 2 has been identified already. We are done with 3.

Queue: 2, 4, 6, 7

Visit sequence: 0, 1, 3
Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3
Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue:
4, 6, 7

Visit sequence:
0, 1, 3, 2
Example of a Breadth-First Search (cont.)

8 and 9 are the only adjacent vertices not already visited or identified

Queue: 4, 6, 7, 8, 9

Visit sequence: 0, 1, 3, 2
Example of a Breadth-First Search (cont.)

Queue: 6, 7, 8, 9

Visit sequence: 0, 1, 3, 2, 4

4 is next
Example of a Breadth-First Search (cont.)

5 is the only vertex not already visited or identified

Queue:
6, 7, 8, 9, 5

Visit sequence:
0, 1, 3, 2, 4
Example of a Breadth-First Search (cont.)

Queue: 7, 8, 9, 5
Visit sequence: 0, 1, 3, 2, 4, 6
Example of a Breadth-First Search (cont.)

Queue:
7, 8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6
Example of a Breadth-First Search (cont.)

7 has no vertices not already visited or identified

Queue:
8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7
Example of a Breadth-First Search (cont.)

Queue:
8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7

7 has no vertices not already visited or identified
Example of a Breadth-First Search (cont.)

Go back to the vertices of 2 and visit them

Queue:
8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7
Example of a Breadth-First Search (cont.)

8 has no vertices not already visited or identified

Queue: 9, 5

Visit sequence: 0, 1, 3, 2, 4, 6, 7, 8
Example of a Breadth-First Search (cont.)

9 has no vertices not already visited or identified

Queue:
5

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9
Finally, visit 5

Queue:
5

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9
Example of a Breadth-First Search (cont.)

which has no vertices not already visited or identified

Queue:
empty

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9, 5
Example of a Breadth-First Search (cont.)

The queue is empty; all vertices have been visited

Queue:
empty

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9, 5
Depth-First Search

- In a depth-first search,
  - start at a vertex,
  - visit it,
  - choose one adjacent vertex to visit;
  - then, choose a vertex adjacent to that vertex to visit,
  - and so on until you can go no further;
  - then back up and see whether a new vertex can be found
Example of a Depth-First Search

Diagram showing a graph with nodes 0, 2, 3, 4, 5, and 6. The nodes are labeled as 'unvisited', 'visited', and 'being visited'. Node 0 is marked as 'visited'.
Example of a Depth-First Search (cont.)

Mark 0 as being visited

Discovery (Visit) order:
0

Finish order:
Example of a Depth-First Search (cont.)

Choose an adjacent vertex that is not being visited

Discovery (Visit) order:
0

Finish order:

0 unvisited   0 visited   0 being visited
Choose an adjacent vertex that is not being visited

Discovery (Visit) order: 0, 1

Finish order:
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

Discovery (Visit) order: 0, 1, 3

Finish order:
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

Discovery (Visit) order: 0, 1, 3

Finish order:
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

Discovery (Visit) order:
0, 1, 3, 4

Finish order:
Example of a Depth-First Search (cont.)

There are no vertices adjacent to 4 that are not being visited

Discovery (Visit) order: 0, 1, 3, 4

Finish order:
Example of a Depth-First Search (cont.)

Mark 4 as visited

Discovery (Visit) order:
0, 1, 3, 4

Finish order:
4
Example of a Depth-First Search (cont.)

Return from the recursion to 3; all adjacent nodes to 3 are being visited

Finish order:
4
Example of a Depth-First Search (cont.)

Mark 3 as visited

Finish order: 4, 3
Example of a Depth-First Search (cont.)

Finish order: 
4, 3

Return from the recursion to 1
All vertices adjacent to 1 are being visited

Finish order: 4, 3
Example of a Depth-First Search (cont.)

Mark 1 as visited

Finish order: 4, 3, 1
Example of a Depth-First Search (cont.)

Return from the recursion to 0

Finish order: 4, 3, 1
Example of a Depth-First Search (cont.)

Finish order: 4, 3, 1

2 is adjacent to 1 and is not being visited
Example of a Depth-First Search (cont.)

Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1

2 is adjacent to 1 and is not being visited
Example of a Depth-First Search (cont.)

5 is adjacent to 2 and is not being visited

Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1
Example of a Depth-First Search (cont.)

Discovery (Visit) order:
0, 1, 3, 4, 2, 5

Finish order:
4, 3, 1

5 is adjacent to 2 and is not being visited
Example of a Depth-First Search (cont.)

Discovery (Visit) order:
0, 1, 3, 4, 2, 5

Finish order:
4, 3, 1

6 is adjacent to 5 and is not being visited
Example of a Depth-First Search (cont.)

Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1

6 is adjacent to 5 and is not being visited
Example of a Depth-First Search (cont.)

There are no vertices adjacent to 6 not being visited; mark 6 as visited

Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1
Example of a Depth-First Search (cont.)

There are no vertices adjacent to 6 not being visited; mark 6 as visited

Discovery (Visit) order: 0, 1, 3, 4, 2, 5, 6

Finish order: 4, 3, 1, 6
Example of a Depth-First Search (cont.)

Finish order: 4, 3, 1, 6

Return from the recursion to 5
Example of a Depth-First Search (cont.)

Mark 5 as visited

Finish order: 4, 3, 1, 6
Example of a Depth-First Search (cont.)

Mark 5 as visited

Finish order: 4, 3, 1, 6, 5
Example of a Depth-First Search (cont.)

Return from the recursion to 2

Finish order: 4, 3, 1, 6, 5

0 unvisited  0 visited  0 being visited
Example of a Depth-First Search (cont.)

Mark 2 as visited

Finish order: 4, 3, 1, 6, 5
Example of a Depth-First Search (cont.)

Mark 2 as visited

Finish order:
4, 3, 1, 6, 5, 2
Example of a Depth-First Search (cont.)

Return from the recursion to 0

Finish order:
4, 3, 1, 6, 5, 2
Example of a Depth-First Search (cont.)

There are no nodes adjacent to 0 not being visited

Finish order: 4, 3, 1, 6, 5, 2
Example of a Depth-First Search (cont.)

Mark 0 as visited

Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6, 0

Finish order:
4, 3, 1, 6, 5, 2, 0
The discovery order is the order in which the vertices are discovered
- 0, 1, 3, 4, 2, 5, 6 in this example

The finish order is the order in which the vertices are finished
- 4, 3, 1, 6, 5, 2, 0 in this example

Back edges connect a vertex with its ancestors in a depth-first search tree
Search Terms (cont.)

Diagram with nodes 0, 1, 2, 3, 4, 5, 6 connected in a graph.
Algorithm for Depth-First Search

1. Mark the current vertex, \( u \), visited (color it light blue), and enter it in the discovery order list.
2. for each vertex, \( v \), adjacent to the current vertex, \( u \)
3. \hspace{1cm} if \( v \) has not been visited
4. \hspace{2cm} Set parent of \( v \) to \( u \).
5. \hspace{1cm} Recursively apply this algorithm starting at \( v \).
6. Mark \( u \) finished (color it dark blue) and enter \( u \) into the finish order list.
Algorithms Using Weighted Graphs

Section 12.6
The breadth-first search found the shortest path from the start vertex to all other vertices, assuming that the length or weight of each edge was the same.

Dijkstra's algorithm finds the shortest path in a weighted directed graph.
Finding the Shortest Path from a Vertex to All Other Vertices (cont.)

- We need 2 sets and 2 arrays
  - Set $S$ will contain the vertices for which we have computed the shortest distance
    - Initialize $S$ by placing the start vertex $s$ into it
  - Set $V-S$ will contain the vertices we still need to process
    - Initialize $V-S$ by placing the remaining vertices into it
  - $d[v]$ will contain the shortest distance from $s$ to $v$
    - For each $v$ in $V-S$, set $d[v]$ to the weight of the edge $w(s,v)$ for each vertex $v$ adjacent to $s$ and to $\infty$ for each vertex not adjacent to $s$
  - $p[v]$ will contain the predecessor of $v$ in the path from $s$ to $v$
    - Initialize $p[v]$ to $s$ for each $v$ in $V-S$
Dijkstra's Algorithm

\[ S = \{ \} \]

\[ V - S = \{ \} \]

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm (cont.)

\[ S = \{ \} \]

\[ V-S = \{ \} \]

\[
\begin{array}{|c|c|c|}
\hline
v & d[v] & p[v] \\
\hline
1 & \_ & \_ \\
2 & \_ & \_ \\
3 & \_ & \_ \\
4 & \_ & \_ \\
\hline
\end{array}
\]

\( s \) is the start vertex

\[ S = \{ \} \]

\[ V-S = \{ \} \]
Dijkstra's Algorithm (cont.)

\[ S = \{ \} \]

\[ V - S = \{ \} \]

Set \( S \) will contain the vertices for which we have computed the shortest distance.
Dijkstra's Algorithm (cont.)

\[ S = \{ \} \]
\[ V-S = \{ \} \]

Set \( V-S \) will contain the vertices that we still need to process.
Dijkstra's Algorithm (cont.)

\[ S = \{ \} \]
\[ V-S = \{ \} \]

\[
\begin{array}{|c|c|c|}
\hline
\text{v} & d[v] & p[v] \\
\hline
1 & \text{---} & \text{---} \\
2 & \text{---} & \text{---} \\
3 & \text{---} & \text{---} \\
4 & \text{---} & \text{---} \\
\hline
\end{array}
\]

\[ v \] will contain the list of vertices not including \( s \)
Dijkstra's Algorithm (cont.)

\[ S = \{ \} \]

\[ V-S = \{ \} \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( d[v] \) will contain the shortest distance from \( s \) to \( v \)
Dijkstra's Algorithm (cont.)

\[ S = \{ \} \]
\[ V-S = \{ \} \]

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

p[v] will contain the predecessor of v in the path from s to v.
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4 \} \]

At initialization, the start vertex \( s \) is placed in \( S \), and the remaining vertices into \( V-S \).
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4 \} \]

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each \( v \) in \( V-S \), we initialize \( d \) by setting \( d[v] \) equal to the weight of the edge \( w(s, v) \) for each vertex \( v \), adjacent to \( s \).
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4 \} \]

For each \( v \) in \( V-S \), we initialize \( d \) by setting \( d[v] \) equal to the weight of the edge \( w(s, v) \) for each vertex \( v \), adjacent to \( s \).
Dijkstra's Algorithm (cont.)

$S = \{0\}$

$V-S = \{1, 2, 3, 4\}$

For each $v$ in $V-S$, we initialize $d$ by setting $d[v]$ equal to the weight of the edge $w(s, v)$ for each vertex $v$, adjacent to $s$
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]
\[ V-S = \{ 1, 2, 3, 4 \} \]

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

For each \( v \) in \( V-S \), we initialize \( d \) by setting \( d[v] \) equal to the weight of the edge \( w(s, v) \) for each vertex \( v \), adjacent to \( s \).
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]
\[ V-S = \{ 1, 2, 3, 4 \} \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

For each \( v \) not adjacent to \( s \), we set \( d[v] \) equal to \( \infty \).
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]
\[ \mathcal{V} - S = \{ 1, 2, 3, 4 \} \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( \infty )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

For each \( v \) not adjacent to \( s \), we set \( d[v] \) equal to \( \infty \)
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]
\[ V-S = \{ 1, 2, 3, 4 \} \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( \infty )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

We initialize each \( p[v] \) to \( s \)
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V - S = \{ 1, 2, 3, 4 \} \]

We initialize each \( p[v] \) to \( s \)
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4 \} \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

We now find the vertex \( u \) in \( V-S \) that has the smallest value of \( d[u] \).
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]
\[ V-S = \{ 1, 2, 3, 4 \} \]
\[ u = 1 \]

\[
\begin{array}{c|c|c}
\text{v} & \text{d[v]} & \text{p[v]} \\
1 & 10 & 0 \\
2 & \infty & 0 \\
3 & 30 & 0 \\
4 & 100 & 0 \\
\end{array}
\]

We now find the vertex \( u \) in \( V-S \) that has the smallest value of \( d[u] \)
Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V-S = \{ 1, 2, 3, 4 \}$

$u = 1$

Consider the vertices $v$ that are adjacent to $u$

<table>
<thead>
<tr>
<th>$v$</th>
<th>$d[v]$</th>
<th>$p[v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4 \} \]

\[ u = 1 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

If the distance from \( s \) to \( u \) (\( d[u] \)) plus the distance from \( u \) to \( v \) is smaller than \( d[v] \) we update \( d[v] \) to that value.
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4 \} \]

\[ u = 1 \]

<table>
<thead>
<tr>
<th>v</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

If the distance from \( s \) to \( u \) (\( d[u] \)) plus the distance from \( u \) to \( v \) is smaller than \( d[v] \) we update \( d[v] \) to that value.
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]
\[ V-S = \{ 1, 2, 3, 4 \} \]
\[ u = 1 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

If the distance from \( s \) to \( u \) (\( d[u] \)) plus the distance from \( u \) to \( v \) is smaller than \( d[v] \) we update \( d[v] \) to that value.
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V \setminus S = \{ 1, 2, 3, 4 \} \]

\[ u = 1 \]

<table>
<thead>
<tr>
<th>v</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

and set \( p[v] \) to \( u \)
Dijkstra's Algorithm (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4 \} \]

\[ u = 1 \]

<table>
<thead>
<tr>
<th>v</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Remove \( u \) from \( V-S \) and place it in \( S \).
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]

\[ V-S = \{ 2, 3, 4 \} \]

\[ u = 1 \]

\[ \begin{array}{c|c|c|c} 
 v & d[v] & p[v] \\
--- & --- & --- \\
1 & 10 & 0 \\
2 & 60 & 1 \\
3 & 30 & 0 \\
4 & 100 & 0 \\
\end{array} \]

Repeat until \( V-S \) is empty
Dijkstra's Algorithm (cont.)

\( S = \{ 0, 1 \} \)

\( V-S = \{ 2, 3, 4 \} \)

\( u = 3 \)

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

The smallest \( d[v] \) in \( V-S \) is vertex 3
Dijkstra's Algorithm (cont.)

\( S = \{ 0, 1 \} \)

\( V-S = \{ 2, 3, 4 \} \)

\( u = 3 \)

\[
\begin{array}{|c|c|c|}
\hline
v & d[v] & p[v] \\
\hline
1 & 10 & 0 \\
2 & 60 & 1 \\
3 & 30 & 0 \\
4 & 100 & 0 \\
\hline
\end{array}
\]

The distance from \( s \) to \( u \) plus the distance from \( u \) to \( v \) is 50
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]
\[ V-S = \{ 2, 3, 4 \} \]
\[ u = 3 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

50 < \( d[2] \) (which is 60)
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]

\[ V-S = \{ 2, 3, 4 \} \]

\[ u = 3 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Set \( d[2] \) to 50 and \( p[2] \) to \( u \)
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]

\[ V-S = \{ 2, 3, 4 \} \]

\[ u = 3 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Continue to the next adjacent vertex
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]

\[ V - S = \{ 2, 3, 4 \} \]

\[ u = 3 \]

\[
\begin{array}{c|c|c|c}
  v & d[v] & p[v] \\
  \hline
  1 & 10 & 0 \\
  2 & 50 & 3 \\
  3 & 30 & 0 \\
  4 & 100 & 0 \\
\end{array}
\]
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]

\[ V - S = \{ 2, 3, 4 \} \]

\[ u = 3 \]

\[ \begin{array}{c|c|c|c|c|}
   v & d[v] & p[v] \\
   \hline
   1 & 10 & 0 \\
   2 & 50 & 3 \\
   3 & 30 & 0 \\
   4 & 100 & 0 \\
\end{array} \]

90 < 100
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]

\[ V-S = \{ 2, 3, 4 \} \]

\[ u = 3 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ 90 < 100 \]
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1 \} \]
\[ V-S = \{ 2, 3, 4 \} \]

\[ u = 3 \]

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>3</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm (cont.)

\[ S = \{ 0, 1, 3 \} \]

\[ V-S = \{ 2, 4 \} \]

\[ u = 3 \]

\[
\begin{array}{|c|c|c|}
\hline
v & d[v] & p[v] \\
\hline
1 & 10 & 0 \\
2 & 50 & 3 \\
3 & 30 & 0 \\
4 & 90 & 3 \\
\hline
\end{array}
\]
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3 \} \]

\[ V-S = \{ 2, 4 \} \]

\[ u = 2 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>3</td>
</tr>
</tbody>
</table>

Select vertex 2 from V-S
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3 \} \]
\[ V-S = \{ 2, 4 \} \]
\[ u = 2 \]

\[
\begin{array}{|c|c|c|}
\hline
v & d[v] & p[v] \\
\hline
1 & 10 & 0 \\
2 & 50 & 3 \\
3 & 30 & 0 \\
4 & 90 & 3 \\
\hline
\end{array}
\]

\[ d[2] + w(2,4) = 50 + 10 = 60 \]
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3 \} \]

\[ V - S = \{ 2, 4 \} \]

\[ u = 2 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ 60 < 90 \text{ (} d[4] \text{)} \]
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3 \} \]

\[ V - S = \{ 2, 4 \} \]

\[ u = 2 \]

\[
\begin{array}{c|c|c}
 v & d[v] & p[v] \\
1 & 10 & 0 \\
2 & 50 & 3 \\
3 & 30 & 0 \\
4 & 90 & 3 \\
\end{array}
\]

update \( d[4] \) to 60 and \( p[4] \) to 2
Dijkstra's Algorithm (cont.)

\( S = \{ 0, 1, 3 \} \)

\( V-S = \{ 2, 4 \} \)

\( u = 2 \)

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>2</td>
</tr>
</tbody>
</table>

update \( d[4] \) to 60 and \( p[4] \) to 2
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3 \} \]

\[ V-S = \{ 2, 4 \} \]

\[ u = 2 \]

\[
\begin{array}{|c|c|c|}
\hline
v & d[v] & p[v] \\
\hline
1 & 10 & 0 \\
2 & 50 & 3 \\
3 & 30 & 0 \\
4 & 60 & 2 \\
\hline
\end{array}
\]

Remove 2 from \( V-S \)
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3, 2 \} \]

\[ V-S = \{ 4 \} \]

\( u = 2 \)

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>2</td>
</tr>
</tbody>
</table>

Remove 2 from \( V-S \)
Dijkstra's Algorithm (cont.)

$S = \{ 0, 1, 3, 2 \}$

$V-S = \{ 4 \}$

$u = 2$

<table>
<thead>
<tr>
<th>$v$</th>
<th>$d[v]$</th>
<th>$p[v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>2</td>
</tr>
</tbody>
</table>

The final vertex in $V-S$ is 4
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3, 2 \} \]

\[ V-S = \{ 4 \} \]

\[ u = 2 \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( p[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>2</td>
</tr>
</tbody>
</table>

The final vertex in \( V-S \) is 4
Dijkstra's Algorithm (cont.)

$S = \{ 0, 1, 3, 2 \}$

$V-S = \{ 4 \}$

$u = 2$

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>2</td>
</tr>
</tbody>
</table>

4 has no adjacent vertices; we move 4 into $S$
Dijkstra's Algorithm (cont.)

\[ S = \{ 0, 1, 3, 2, 4 \} \]
\[ V - S = \{ \} \]
\[ u = 2 \]

<table>
<thead>
<tr>
<th>v</th>
<th>d[v]</th>
<th>p[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>2</td>
</tr>
</tbody>
</table>

4 has no adjacent vertices; we move 4 into S
Dijkstra's Algorithm (cont.)

$S = \{ 0, 1, 3, 2, 4 \}$

$V-S = \{ \}$

$u = 2$

<table>
<thead>
<tr>
<th>$v$</th>
<th>$d[v]$</th>
<th>$p[v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>2</td>
</tr>
</tbody>
</table>

We are finished
Minimum Spanning Trees

- A spanning tree is a subset of the edges of a graph such that there is only one edge between any two vertices, and all of the vertices are connected.
- If we have a spanning tree for a graph, then we can access all the vertices of the graph from the start node.
- The cost of a spanning tree is the sum of the weights of the edges.
- We want to find the minimum spanning tree or the spanning tree with the smallest cost.
If we want to start up our own long-distance phone company and need to connect the cities shown below, finding the minimum spanning tree would allow us to build the cheapest network.

The solution to this problem was formulated by R.C. Prim and is very similar to Dijkstra’s algorithm.
Prim's Algorithm Example

\[ S = \{ 0 \} \]
\[ V-S = \{ 1, 2, 3, 4, 5 \} \]
Prim's Algorithm Example (cont.)

\[ S = \{ 0 \} \]

\[ V-S = \{ 1, 2, 3, 4, 5 \} \]

The smallest edge from \( u \) to \( v \) where \( u \) is in \( S \) and \( v \) is in \( V-S \) is the edge \((0,2)\)
Prim's Algorithm Example (cont.)

\( S = \{ 0 \} \)

\( V-S = \{ 1, 2, 3, 4, 5 \} \)

Add this edge to the spanning tree
Prim's Algorithm Example (cont.)

\[ S = \{ 0 \} \]
\[ V-S = \{ 1, 2, 3, 4, 5 \} \]

and move 2 to \( S \)
Prim's Algorithm Example (cont.)

\[ S = \{0, 2\} \]

\[ V - S = \{1, 3, 4, 5\} \]

and move 2 to \( S \)
Prim's Algorithm Example (cont.)

\[ S = \{ 0, 2 \} \]

\[ V-S = \{ 1, 3, 4, 5 \} \]

Consider all edges \((u, v)\) where \(u\) is in \(S\) and \(v\) is in \(V-S\) (there are 8 possible edges)
Prim's Algorithm Example (cont.)

\[S = \{0, 2\}\]

\[V - S = \{1, 3, 4, 5\}\]

The smallest is \((2, 5)\)
Prim's Algorithm Example (cont.)

$S = \{ 0, 2 \}$

$V-S = \{ 1, 3, 4, 5 \}$

Add (2,5) to the spanning tree
Prim's Algorithm Example (cont.)

$S = \{ 0, 2 \}$

$V-S = \{ 1, 3, 4, 5 \}$

Move 5 from $V-S$ to $S$
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }

Move 5 from V-S to S
Prim's Algorithm Example (cont.)

$S = \{ 0, 2, 5 \}$

$V-S = \{ 1, 3, 4 \}$

Find the smallest edge $(u, v)$ where $u$ is in $S$ and $v$ is in $V-S$
Prim's Algorithm Example (cont.)

$S = \{ 0, 2, 5 \}$

$V-S = \{ 1, 3, 4 \}$

The smallest edge is $(5, 3)$
Prim's Algorithm Example (cont.)

$S = \{0, 2, 5\}$

$V-S = \{1, 3, 4\}$

The smallest edge is (5, 3)
Prim's Algorithm Example (cont.)

\[ S = \{ 0, 2, 5 \} \]
\[ V-S = \{ 1, 3, 4 \} \]

Move 3 to \( S \)
Prim's Algorithm Example (cont.)

$S = \{ 0, 2, 5, 3 \}$

$V - S = \{ 1, 4 \}$

Move 3 to $S$
Prim's Algorithm Example (cont.)

\[ S = \{ 0, 2, 5, 3 \} \]

\[ V - S = \{ 1, 4 \} \]

The next smallest edge is (2, 1)
Prim's Algorithm Example (cont.)

\[ S = \{ 0, 2, 5, 3 \} \]

\[ V-S = \{ 1, 4 \} \]

The next smallest edge is (2, 1)
Prim's Algorithm Example (cont.)

$S = \{ 0, 2, 5, 3 \}$

$V-S = \{ 1, 4 \}$

Move 1 to $S$
Prim's Algorithm Example (cont.)

$S = \{ 0, 2, 5, 3, 1 \}$

$V-S = \{ 4 \}$

Move 1 to $S$
Prim's Algorithm Example (cont.)

\( S = \{ 0, 2, 5, 3, 1 \} \)

\( V-S = \{ 4 \} \)

The smallest edge to 4 is \((1, 4)\)
Prim's Algorithm Example (cont.)

\[ S = \{ 0, 2, 5, 3, 1 \} \]

\[ V-S = \{ 4 \} \]

The smallest edge to 4 is (1, 4)
Prim's Algorithm Example (cont.)

\[ S = \{ 0, 2, 5, 3, 1, 4 \} \]

\[ V - S = \{ \} \]

The spanning tree is complete.