Graphs

7

Many problems in Computer Science and Mathematics can be reduced to a set of states and a set of transitions between these states. A graph is a mathematical representation of problems like these. In the last chapter we saw that trees serve a variety of purposes in Computer Science. Trees are graphs. However, graphs are more general than trees. Abstracting away the details of a problem and studying it in its simplest form often leads to new insight. As a result, many algorithms have come out of the research in graph theory. Graph theory was first studied by mathematicians. Many of the algorithms in graph theory are named for the mathematician that developed or discovered them. Dijkstra and Kruskal are two such mathematicians and this chapter covers algorithms developed by them.

Representing a graph can be done one of several different ways. The correct way to represent a graph depends on the algorithm being implemented. Graph theory problems include graph coloring, finding a path between two states or nodes in a graph, or finding a shortest path through a graph among many others. There are many algorithms that have come from the study of graphs. To understand the formulation of these problems it is good to learn a little graph notation which is presented in this chapter as well.

7.1 Chapter Goals

This chapter covers the representation of graphs. It also covers a few graph algorithms. Depth first search of a graph is presented, along with breadth first search. Dijkstra's algorithm is famous in Computer Science and has many applications from networking to construction planning. Kruskal's algorithm is another famous algorithm used to find a minimum weighted spanning tree. By the end of the chapter you should have a basic understanding of graph theory and how many problems in Computer Science can be posed in the form of graphs.

To begin we'll study some notation and depth first search of a graph. Then we'll examine a couple of *Greedy Algorithms* that answer some interesting questions about graphs. Greedy algorithms are algorithms that never make a wrong choice in finding

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_7

a solution. We'll examine two of these algorithms called Kruskal's Algorithm and Dijkstra's Algorithm, both named for the people that formulated the algorithm to solve their respective problems.

7.2 Graph Notation

A little notation will help in the graph definitions in this chapter. A set is an unordered collection of items. For instance, $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ is the set of the first 13 natural numbers. A *subset* of a set is some collection, possibly empty, of items from its *superset*. The set $U = \{5, 8, 2\}$ is a subset of V. The *cardinality* of a set is its size or number of elements. The cardinality of the set V is written as $|V|$. The cardinality of V is 13 and U is 3, so $|V| = 13$ and $|U| = 3$.

A graph $G = (V,E)$ is defined by a set of vertices, named V, and a set of edges, named E. The set of edges are subsets of V where each member of E has *cardinality* 2. In other words, edges are denoted by pairs of vertices. Consider the simple, undirected graph given in Fig. 7.1. The sets $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ and $E = \{\{0, 1\},\{0, 3\},\{0, 10\},\{1, 10\},\{1, 4\},\{2, 3\},\{2, 8\},\{2, 6\},\{3, 9\},\{5, 4\},\{5, 12\},\{5, 7\},\{11, 12\},\{11, 10\},\{9, 10\}\}$ define this graph. Since each edge is itself a set of cardinality 2, the order of the vertices in each edge set does not matter. For instance, $\{1, 4\}$ is the same edge as $\{4, 1\}$.

Many problems can be formulated in terms of a graph. For instance, we might ask how many colors it would take to color a map so that no two countries that shared a border were colored the same. In this problem the vertices in Fig. 7.1 would represent countries and two countries that share a border would have an edge between them. The problem can then be restated as finding the minimum number of colors required to color each vertex in the graph so that no two vertices that share an edge have the same color.

A directed graph $G = (V,E)$ is defined in the same way as an undirected graph except that the set of edges, E, is a set of tuples instead of subsets. By defining $E = \{(v_i, v_j)\ \textit{where}\ v_i, v_j \in V\}$ means that edges can be traversed in one direction

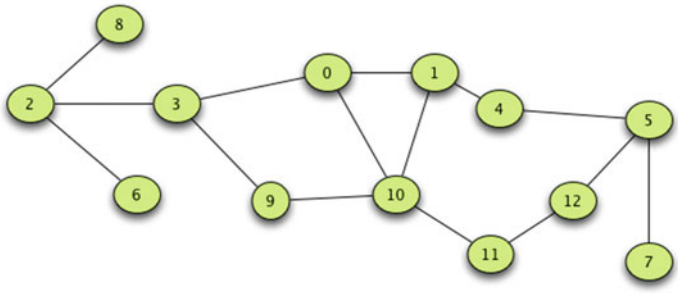

Fig. 7.1 An Undirected Graph

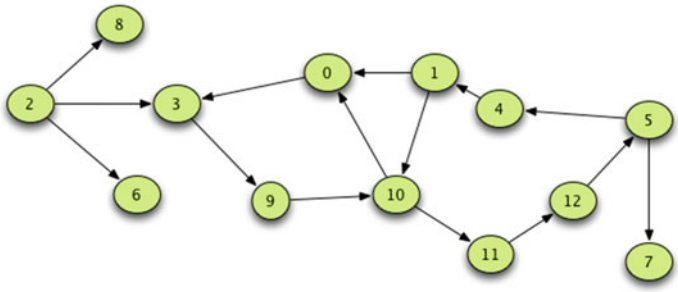

Fig. 7.2 A Directed Graph

only. In Fig. 7.2 we can move from vertex 10 to vertex 0 along the edge (10,0), but we cannot move from vertex 0 to 10, at least not without going through some other vertices, because the edge (0,10) is not in the set *E*.

A *path* in a graph is a series of edges, none repeated, that can be traversed in order to travel from one vertex to another in a graph. A *cycle* in a graph is a path which begins and ends with the same vertex. The last chapter covered trees in computer science. Now armed with some notation from graph theory we can give a formal definition of a tree. A *tree* is a *directed, connected acyclic graph*. An acyclic graph is a graph without any cycles.

Sometimes in graph theory a tree is defined as an *acyclic connected graph* dropping the requirement that it be a directed graph. In this case, a tree may be defined as a graph which is fully connected, but has only one path between any two vertices.

Both directed and undirected graphs can be used to model many different kinds of problems. The graph in Fig. 7.1 might represent register allocation in a CPU. The vertices could represent symbolically named registers and two registers that were both in use at the same time would have an edge between them. The question that might be asked is, "How many physical registers of the machine are required for the symbolic registers of this computation?".

It turns out that register allocation and map coloring represent the same problem. When we abstract away the details, the problem boils down to a graph coloring problem. An answer to "How many colors are required to color the map?" would answer "How many physical registers are required for this computation?" and vice-versa.

A weighted graph is a graph where every edge has a weight assigned to it. More formally, a weighted graph $G = (V,E,w)$ is a graph with the given set of vertices, V, and edges, E. In addition, a weighted graph has a weight function, w, that maps edges to real numbers. So the signature of w is given by $w: E \rightarrow Real$. Weighted graphs can be used to represent the state of many different problems. For instance, a weighted graph might provide information about roads and intersections. Cost/benefit analysis can sometimes be expressed in terms of a weighted graph. The weights can represent the available capacity of network connections between nodes in a network.

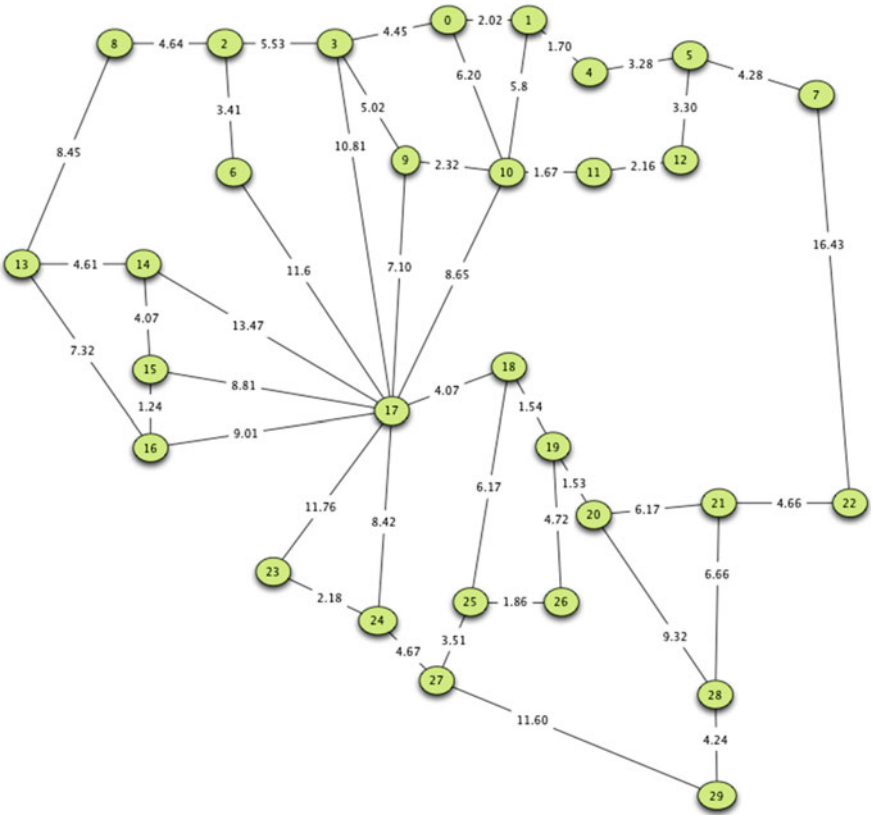

Fig. 7.3 A Weighted, Undirected Graph

A weighted graph can be used to represent the state of many different kinds of problems. Figure 7.3 depicts a weighted graph which represents roads and intersections.

7.3 Searching a Graph

Many problems have been formulated in terms of graph theory. One of the more common problems is discovering a path from one vertex to another in a graph. The question might be, does a path exist from vertex v_i to v_j and if so, what edges must you traverse to get there? Performing depth first search on a graph is similar to the algorithm first presented in Chap. 6, but we must be wary of getting stuck in a cycle within a graph.

Consider searching for a path from vertex 0 to vertex 1 in the directed graph of Fig. 7.2. The blue lines in Fig. 7.4 highlight the path between these vertices. In the

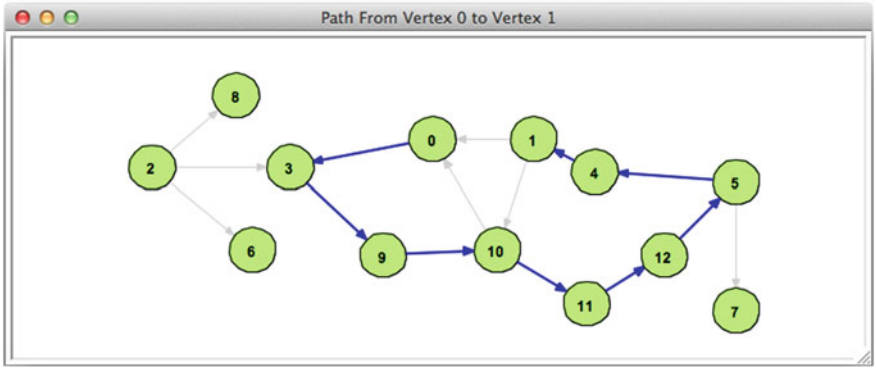

Fig. 7.4 A Path from Vertex 0 to Vertex 1

graph, there seems to be only one choice in most cases. However, when the search reaches vertex 10 we must choose between two edges. One edge takes us back to vertex 0 which we have already visited. The other edge takes us closer to the final path. Another choice is made at vertex 5. If the edge to 7 is wrongly examined, we must have a way of backing up and trying the other edge to vertex 4.

Searching a graph in this manner is also called *depth first search*, as first discussed in Chap. 6, and requires the ability to backtrack. Consider when vertex 5 is encountered. If a choice is made to go to vertex 7, we must be able to back up to fix that choice and go to vertex 4 instead. A stack data structure or recursion handles the backtracking.

Depth first search must also avoid possible cycles within the graph. The avoidance of cycles is accomplished by maintaining a set of *visited* vertices. When a vertex is visited, it is added to the *visited* set. If a vertex is in the visited set, then it is not examined again later in the search should a cycle take the search back to the same vertex.

An iterative (i.e. non-recursive) graph depth first search algorithm begins by initializing the *visited* set to the empty set and by creating a stack for backtracking. The start vertex is pushed onto the stack to begin the algorithm. Steps similar to those taken in Sect. 7.3.1 are executed to find the goal. This code is pseudo-code, but presents the necessary details.

7.3.1 Iterative Depth First Search of a Graph

```
1  def graphDFS(G, start, goal):
2    # G = (V,E) is the graph with vertices, V, and edges, E.
3    V,E = G
4    stack = Stack()
5    visited = Set()
6    stack.push(start)
7
8    while not stack.isEmpty():
9      # A vertex is popped from the stack. This is called the current vertex.
10     current = stack.pop()
```

```
    # The current vertex is added to the visited set.
    visited.add(current)

    # If the current vertex is the goal vertex, then we discontinue the
    # search reporting that we found the goal.
    if current == goal:
      return True # or return path to goal perhaps

    # Otherwise, for every adjacent vertex, v, to the current vertex
    # in the graph, v is pushed on the stack of vertices yet to search
    # unless v is already in the visited set in which case the edge
    # leading to v is ignored.
    for v in adjacent(current,E):
      if not v in visited:
        stack.push(v)

  # If we get this far, then we did not find the goal.
  return False # or return an empty path
```

If the while loop in Sect. 7.3.1 terminates the stack was empty and therefore no path to the goal exists. This algorithm implements depth first search of a graph. It can also be implemented recursively if pushing on the stack is replaced with a recursive call to *depth first search*. When implemented recursively, the *depth first search* function is passed the current vertex and a mutable *visited* set and it returns either the path to the goal or alternatively a boolean value indicating that the goal or target was found. Given the graph in Fig. 7.4 the search returned *True*.

The iterative version of *depth first search* can be modified to do a *breadth first search* of a graph if the stack is replaced with a queue. Breadth first search is an exhaustive search, meaning that it looks at all paths at the same time, but will also find the shortest path, with the least number of edges, between any two vertices in a graph. Performing breadth first search on large graphs may take too long to be of practical use.

7.4 Kruskal's Algorithm

Consider for a moment a county which is responsible for plowing roads in the winter but is running out of money due to an unexpected amount of snow. The county supervisor has been told to reduce costs by plowing only the necessary roads for the rest of the winter. The supervisor wants to find the shortest number of total miles that must be plowed so any person can travel from one point to any other point in the county, but not necessarily by the shortest route. The county supervisor wants to minimize the miles of plowed roads, while guaranteeing you can still get anywhere you need to in the county.

Joseph Kruskal was an American computer scientist and mathematician who lived from 1928 to 2010. He imagined this problem, formalized it in terms of a weighted graph, and devised an algorithm to solve this problem. His algorithm was first published in the *Proceedings of the American Mathematical Society* [5] and is commonly called *Kruskal's Algorithm*.

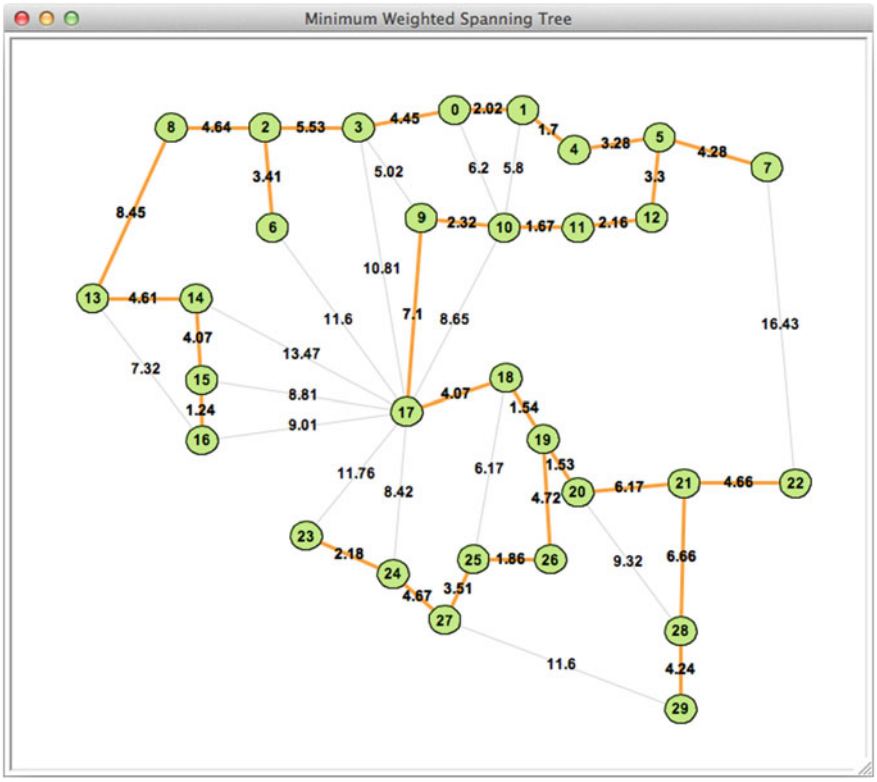

Fig. 7.5 A Minimum Weighted Spanning Tree

The last chapter introduced trees by using them in various algorithms like binary search. The definition doesn't change from the last chapter. But trees, in the context of graph theory, are a subset of the set of all possible graphs. A tree is just a graph without any cycles. In addition, it is relatively easy to prove that a tree must contain one less edge than its number of vertices. Otherwise, it would not be a tree.

Clearly the graph in Fig. 7.3 is not a tree. There are many cycles within the graph. Kruskal's paper presented an algorithm to find a minimum weighted spanning tree for such a graph. Figure 7.5 contains a minimum weighted spanning tree for the graph in Fig. 7.3 with the tree edges highlighted in orange. We don't say *the* minimum weighted spanning tree because in general there could be more than one minimum weighted spanning tree. In this case, there is likely only one possible.

Kruskal's algorithm is a *greedy algorithm*. The designation *greedy* means that the algorithm always chooses the first alternative when presented with a list of alternatives and never makes a mistake, or wrong choice, when choosing. In other words, no backtracking is required in Kruskal's algorithm.

The algorithm begins by sorting all the edges in ascending order of their weights. Assuming that the graph is fully connected, the spanning tree will contain $|V|-1$ edges. The algorithm forms sets of all the vertices in the graph, one set for each

vertex, initially containing just that vertex that corresponds to the set. In the example in Fig. 7.3 there are initially 30 sets each containing one vertex.

The algorithm proceeds as follows until $|V|-1$ edges have been added to the set of spanning tree edges.

1. The next shortest edge is examined. If the two vertex end points of the edge are in different sets, then the edge may be safely added to the set of spanning tree edges. A new set is formed from the union of the two vertex sets and the two previous sets are dropped from the list of vertex sets.
2. If the two vertex endpoints of the edge are already in the same set, the edge is ignored.

That's the entire algorithm. The algorithm is *greedy* because it always chooses the next smallest edge unless doing so would form a cycle. If a cycle would be formed by adding an edge, it is known right away without having to undo any mistake or backtrack.

Consider Fig. 7.6. In this snapshot the algorithm has already formed a forest of trees, but not a spanning tree yet. The edges in orange are part the spanning tree.

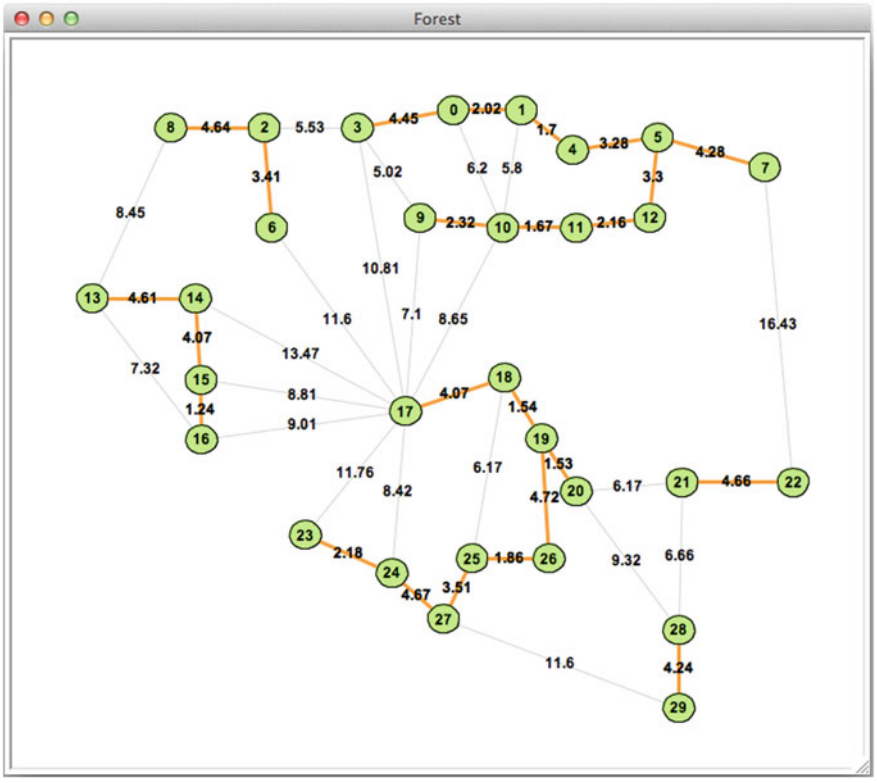

Fig. 7.6 Kruskal's: Snapshot 1

The next shortest edge, the edge from vertex 3 to vertex 9 is currently being considered. The set containing 3 and 9 is the set {3, 0, 1, 4, 5, 12, 11, 10, 9}. Adding the edge with end points 3 and 9 cannot be done because vertices 3 and 9 are already in the same set. So, this edge is skipped. It cannot be a part of the minimum weighted spanning tree.

The next shortest edge is the edge between vertices 2 and 3. Since 2 is a member of {8, 2, 6} and 3 is a member of its set in the previous paragraph, the edge {2, 3} is added to the minimum weighted spanning tree edges and the new set {8, 2, 6, 3, 0, 1, 4, 5, 12, 11, 10, 9} is formed replacing its previous two subsets as depicted in Fig. 7.7.

The next shortest edge is the edge between vertices 1 and 10. This edge again cannot be added since 1 and 10 are in the same set and therefore adding the edge would form a cycle. The next shortest edge is the edge between vertices 18 and 25, but again adding it would form a cycle so it is skipped. The algorithm proceeds in this manner until the resulting spanning tree is formed with $|V|-1$ edges (assuming the graph is fully connected).

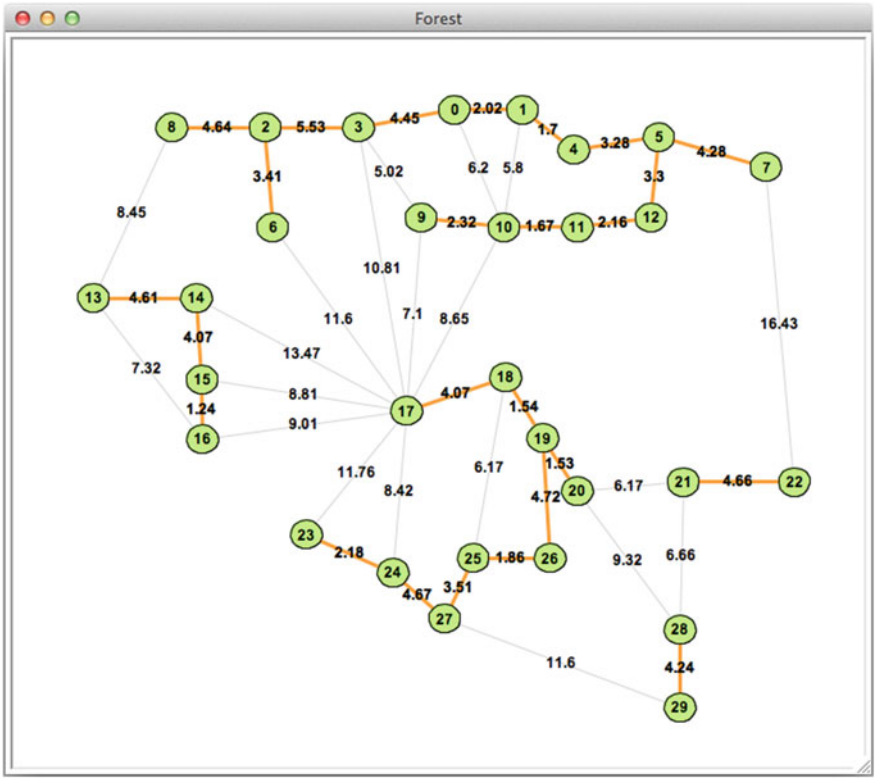

Fig. 7.7 Kruskal's: Snapshot 2

7.4.1 Proof of Correctness

Proving Kruskal's algorithm correctly finds a minimum weighted spanning tree can be done with a proof by contradiction. The proof starts by recognizing that there must be $|V|-1$ edges in the spanning tree. Then we assume that some other edge would be better to add to the spanning tree than the edges picked by the algorithm. The new edge must be a part of one and only one cycle. If adding the new edge formed two or more cycles then there would have had to be a cycle in the tree before adding the new edge. One of the edges in this newly formed cycle must be deleted from the minimum weighted spanning tree to once again make it a tree. And, the deleted edge must have weight greater than the newly added edge. This is only possible if the new edge and the deleted old edge have exactly the same weight since all the old edges in the cycle were chosen before the new edge and the new edge was skipped because choosing it would have formed a cycle. So dropping the same weighted edge of the older edges will result in a minimum weighted spanning tree with the same weight. Therefore, the new spanning tree has the same weight as the original spanning tree and that contradicts our assumption that a better edge could be found.

7.4.2 Kruskal's Complexity Analysis

The complexity of Kruskal's algorithm depends on sorting the list of edges and then forming the union of sets as the algorithm proceeds. Sorting a list, as was shown in Chap. 4 when we looked at the complexity of quicksort, is $O(|E|\log|E|)$.

Sorting the list is one half of Kruskal's algorithm. The other half is choosing the correct edges. Recall that each edge starts in a set by itself and that an edge belongs to the minimum weighted spanning tree if the two endpoint vertices are in separate sets. If so, then the union is formed for the two sets containing the endpoints and this union of two sets replaces the previous two sets going forward.

Three operations are required to implement this part of the algorithm.

1. First we must discover the set for each endpoint of the edge being considered for addition to the spanning tree.
2. Then the two sets must be compared for equality.
3. Finally, the union of the two sets must be formed and any necessary updates must be performed so the two endpoint vertices now refer to the union of the two sets instead of their original sets.

One way to implement these operations would be to create a list of sets where each position in the list corresponded to one vertex in the graph. The vertices are conveniently numbered 0–29 in the example in the text but vertices can be reassigned integer identifiers starting at 0 otherwise. The set corresponding to a vertex can be determined in O(1) time since indexed lookup in a list is a constant time operation.

If we make sure there is only one copy of each set, we can determine if two sets are the same or not in O(1) time as well. We can just compare their references to see whether they are the same set or not. The keyword *is* in Python will accomplish this. So if we want to know that x and y refer to the same set we can write x *is* y and this operation is O(1).

The third operation requires forming a new set from the previous two. This operation will be performed $|V|-1$ times. In the worst case the first time this operation occurs 1 vertex will be added to an existing set. The second time, two vertices will be added to an existing set, and so on. So in the end, the overall worst case complexity of this operation is $O(|V|^2)$ assuming once again that the graph is connected. Clearly, this is the expensive operation of this algorithm. The next section presents a data structure that improves on this considerably.

7.4.3 The Partition Data Structure

To improve on the third required operation, the merging of two sets into one set, a specialized data structure called a *Partition* may be used. The partition data structure contains a list of integers with one entry for each vertex. Initially, the list simply contains a list of integers which match their indices:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29]
```

Think of this list as a list of trees, representing the sets of connected edges in the spanning forest constructed so far. A tree's root is indicated when the value at a location within the list matches its index. Initially, each vertex within the partition is in its own set because each vertex is the root of its own tree.

Discovering the set for a vertex means tracing a tree back to its root. Consider what happens when the edge from vertex 3 to vertex 9 is considered for adding to the minimum weighted spanning tree as pictured in Fig. 7.6. The partition at that time looks like this.

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[ 4, 4, 2, 7, 5,11, 2, 7, 2,11,11, 7,11,16,16,16,16,19,19,19,19,22,22,24,26,26,19,26,29,29]
```

Vertex 3 is not the root of its own tree at this time. Since 7 is found at index 3 we next look at index 7 of the partition list. That position in the partition list matches its value. The 3 is in the set (i.e. tree) rooted at location 7. Looking at vertex 9 next, index 9 in the list contains 11. Index 11 in the list contains 7. Vertex 9 is also in the set (i.e. tree) rooted at index 7. Therefore vertex 3 and 9 are already in the same set and the edge from 3 to 9 cannot be added to the minimum spanning tree since a cycle would be formed.

The next edge to be considered is the edge between vertices 2 and 3. The root of the tree containing 2 is at index 2 of the partition. The root of the vertex containing 3 is at index 7 as we just saw. These two vertices are not in the same set so the edge from 2 to 3 is added to the minimum spanning tree edges.

The third operation that must be performed is the merging of the sets containing 2 and 3. This is where the partition comes in handy. Having found the root of the two trees, we simply make the root of one of the trees point to the root of the other tree. We end up with this partition after merging these two sets.

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[ 4, 4, 2, 7, 5,11, 2, 2, 2,11,11, 7,11,16,16,16,16,19,19,19,19,22,22,24,26,26,19,26,29,29]
```

At this point in the algorithm, the tree rooted at 7 has been altered to be rooted at 2 instead. That's all that was needed to merge the two sets containing vertex 2 and 3! The root of one tree can be made to point to the root of the other tree when two sets are merged into one.

The partition data structure combines the three required operations from Sect. 7.4.2 into one method called *sameSetAndUnion*. This method is given two vertex numbers. The method returns true if the two vertices are in the same set (i.e. have the same root). If they do not have the same root, then the root of one tree is made to point to the other and the method returns false.

The *sameSetAndUnion* method first finds the roots of the two vertices given to it. In the worst case this could take O(|V|) time leading to an overall complexity of O(|V|^2). However, in practice these *set trees* are very flat. For instance, in the example presented in this chapter, the average depth of the *set trees* is 1.7428, meaning that on average it takes 1.7428 comparisons to find the root of a *set tree* in a graph with 30 vertices and 45 edges to consider adding to the minimum weighted spanning tree. Another example containing 133 vertices and 8778 edges had an average *set tree* depth of 7.5656. The average complexity of this *sameSetAndUnion* method is much better than the solution considered in Sect. 7.4.2. The average case complexity of *sameSetAndUnion* is much closer to O(log|V|). This means that the second part of Kruskal's algorithm, using this partition data structure, exhibits O(|E|log|V|) complexity in the average case.

In a connected graph the number of edges must be no less than one less than the total number of vertices. Sorting the edges takes O(|E|log|E|) time and the second part of Kruskal's algorithm takes O(|E|log|V|) time. Since the number of edges is at least on the same order as the number of vertices in a connected graph the complexity O(|E|log|V|) $\leq$ O(|E|log|E|). So we can say that the overall average complexity of Kruskal's algorithm is O(|E|log|E|). In practice, Kruskal's algorithm is very efficient and finds a minimum weighted spanning tree quickly even for large graphs with many edges.

7.5 Dijkstra's Algorithm

Edsger Dijkstra was a Dutch computer scientist who lived from 1930 to 2002. In 1959 he published a short paper [4] that commented on Kruskal's solution to the minimum spanning tree problem and provided an alternative that might in some cases be more efficient. More importantly, he provided an algorithm for finding the

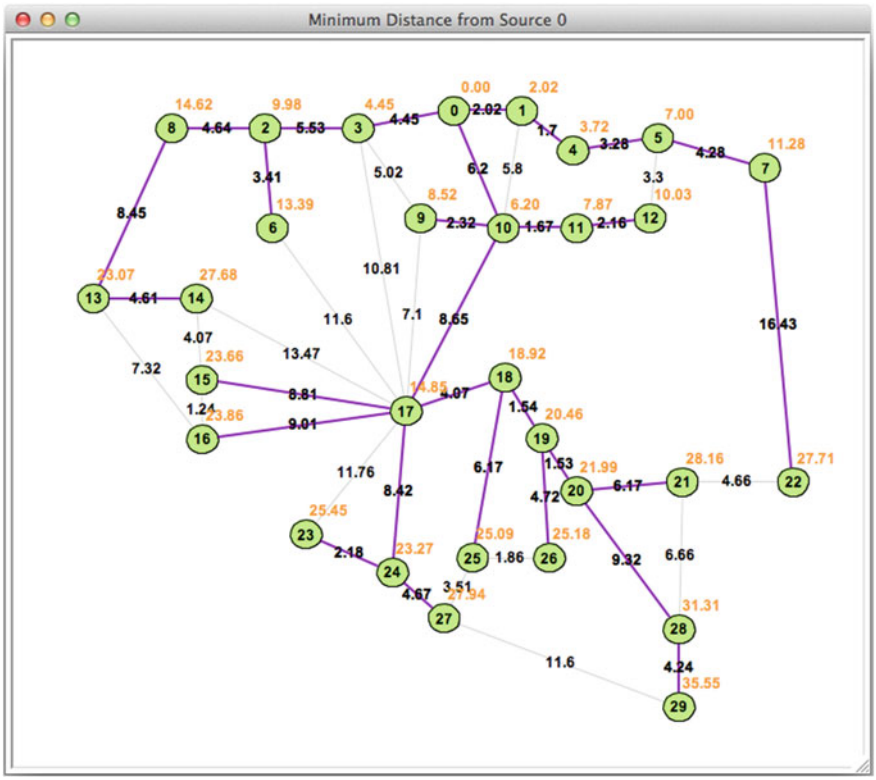

Fig. 7.8 Minimum Cost Paths and Total Cost from Source Vertex 0

minimum cost path between any two vertices in a weighted graph. This algorithm can be, and sometimes is, generalized to find the minimum cost path between a *source* vertex and all other vertices in a graph. This algorithm is known as Dijkstra's algorithm. Figure 7.8 shows the result of running Dijkstra's algorithm on the graph first presented in Fig. 7.3. The purple edges show the minimum cost paths from source vertex 0 to all other vertices in the graph. The orange values are the minimum cost of reaching each vertex from source vertex 0.

Efficiently finding a minimum cost path from one vertex to another is used in all kinds of problems including network routing, trip planning, and other planning problems where vertices represent intermediate goals and edges represent the cost of transitioning between intermediate goals. These kind of planning problems are very common.

Dijkstra's algorithm proceeds in a greedy fashion from the single *source* vertex. Each vertex, v, in the graph is assigned a cost which is the sum of the weighted edges on the path from the *source* to v. Initially the *source* vertex is assigned cost 0. All other vertices are initially assigned infinite cost. Anything greater than the sum of all weights in the graph can serve as an infinite value.

Dijkstra's algorithm shares some commonality with depth first search. The algorithm proceeds as depth first search proceeds, but starts with a single source eventually visiting every node within the graph. There are two sets that Dijkstra's algorithm maintains. The first is an *unvisited* set. This is a set of vertices that yet need to be considered while looking for minimum cost paths. The *unvisited* set serves the same purpose as the stack when performing *depth first search* on a graph. The *visited* set is the other set used by the algorithm. The *visited* set contains all vertices which already have their minimum cost and path computed. The *visited* set serves the same purpose as the *visited* set in depth first search of a graph.

To keep track of the minimum cost path from the source to a vertex, v, it is only necessary to keep track of the previous vertex on the path to v. For each vertex, v, we keep track of the previous vertex on its path from the source.

Initially the *source* vertex, with its cost of 0, is added to the *unvisited* set. Then the algorithm proceeds as follows as long as there is at least one vertex in the *unvisited* set.

1. Remove the vertex we'll call *current* from the *unvisited* set with the least cost. All other paths to this vertex must have greater cost because otherwise they would have been in the *unvisited* set with smaller cost.
2. Add *current* to the *visited* set.
3. For every vertex, *adjacent*, that is adjacent to *current*, check to see if *adjacent* is in the *visited* set or not. If *adjacent* is in the *visited* set, then we know the minimum cost of reaching this vertex from the source so don't do anything.
4. If *adjacent* is not in the *visited* set, compute a new cost for arriving at *adjacent* by traversing the edge, *e*, from *current* to *adjacent*. A new cost can be found by adding the cost of getting to *current* and *e*'s weight. If this new cost is better than the current cost of getting to *adjacent*, then update *adjacent*'s cost and remember that *current* is the previous vertex of *adjacent*. Also, add *adjacent* to the *unvisited* set.

When this algorithm terminates the cost of reaching all vertices in the graph has been computed assuming that all vertices are reachable from the source vertex. In addition, the minimum cost path to each vertex can be determined from the *previous vertex* information that was maintained as the algorithm executed.

7.5.1 Dijkstra's Complexity Analysis

In the first step of Dijkstra's algorithm, the next *current* vertex is always the unvisited vertex with smallest cost. By always picking the vertex with smallest cost so far, we can be guaranteed that no other cheaper path exists to this vertex since we always proceed by considering the next cheapest vertex on our search to find cheapest paths in the graph.

The number of edges of any vertex in a simple, undirected graph will always be less than the number of total vertices in the graph. Each vertex becomes the *current* vertex exactly once in the algorithm in step 1. Assume finding the next *current* takes

O(|V|) time. Since this happens |V| times, the complexity of the first step is O(|V|^2) over the course of running the algorithm. The rest of the steps consider those edges adjacent to *current*. Since the number of edges of any vertex in a simple, undirected graph will always be less than |V|, the rest of the algorithm runs in less than O(|V|^2) time. So, the complexity of Dijkstra's Algorithm is O(|V|^2) assuming that the first step takes O(|V|) to find the next *current* vertex.

It turns out that selecting the next *current* can be done in O(log|V|) time if we use a priority queue for our *unvisited* set. Priority queues and their implementation are discussed in Chap. 9. Using a priority queue, Dijkstra's Algorithm will run in O(|V|log|V|) time.

7.6 Graph Representations

How a graph, $G = (V,E)$, is represented within a program depends on what the program needs to do. Consider the directed graph in Fig. 7.2. The graph itself can be stored in an XML file containing vertices and edges as shown in Sect. 7.6.1. A weighted graph would include a weight attribute for each edge in the graph. In this XML file format the vertexId is used by the edges to indicate which vertices they are attached to. The labels, which appear in Fig. 7.2 are only labels and are not used within the XML file to associate edges with vertices.

7.6.1 A Graph XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<Graph width="595.80" height="229.20" directed="True" weighted="False">
  <Vertices>
    <Vertex vertexId="12" x="343.15" y="156.10" label="10"/>
    <Vertex vertexId="11" x="246.15" y="161.10" label="9"/>
    <Vertex vertexId="10" x="288.15" y="58.10" label="0"/>
    <Vertex vertexId="9" x="374.15" y="58.10" label="1"/>
    <Vertex vertexId="8" x="135.15" y="156.10" label="6"/>
    <Vertex vertexId="7" x="49.65" y="83.10" label="2"/>
    <Vertex vertexId="6" x="167.15" y="83.05" label="3"/>
    <Vertex vertexId="5" x="121.15" y="19.10" label="8"/>
    <Vertex vertexId="4" x="419.15" y="204.10" label="11"/>
    <Vertex vertexId="3" x="426.15" y="87.10" label="4"/>
    <Vertex vertexId="2" x="546.15" y="96.10" label="5"/>
    <Vertex vertexId="1" x="546.15" y="210.10" label="7"/>
    <Vertex vertexId="0" x="485.15" y="161.10" label="12"/>
  </Vertices>
  <Edges>
    <Edge tail="12" head="10"/>
    <Edge tail="10" head="6"/>
    <Edge tail="6" head="11"/>
    <Edge tail="7" head="6"/>
    <Edge tail="7" head="8"/>
    <Edge tail="7" head="5"/>
```

```
    <Edge tail="11" head="12"/>
    <Edge tail="12" head="4"/>
    <Edge tail="4" head="0"/>
    <Edge tail="0" head="2"/>
    <Edge tail="2" head="1"/>
    <Edge tail="2" head="3"/>
    <Edge tail="3" head="9"/>
    <Edge tail="9" head="10"/>
    <Edge tail="9" head="12"/>
  </Edges>
</Graph>
```

The *x* and *y* vertex attributes are not required in any graph representation, but to draw a graph it is nice to have location information for the vertices. All this information is stored in the XML file, but what about the three algorithms presented in this chapter? What information is actually needed by each algorithm.

When searching a graph by depth first search vertices are pushed onto a stack as the search proceeds. In that case the vertex information must be stored for use by the search. In this case, since edges have the *vertexId* of their edge endpoints, it would be nice to have a method to quickly lookup vertices within the graph. A map or dictionary from *vertexId* to vertices would be convenient. It makes sense to create a class to hold the vertex information like the class definition of Sect. 7.6.2.

7.6.2 A Vertex Class

```
class Vertex:
    def __init__(self,vertexId,x,y,label):
        self.vertexId = vertexId
        self.x = x
        self.y = y
        self.label = label
        self.adjacent = []
        self.previous = None
```

In this *Vertex* class definition for directed graphs it makes sense to store the edges with the vertex since edges connect vertices. The *adjacent* list can hold the list of adjacent vertices. When running *depth first search* a map of *vertexId* to *Vertex* for each of the vertices in the graph provides the needed information for the algorithm.

When implementing Kruskal's algorithm, a list of edges is the important feature of the graph. The class definition of Sect. 7.6.3 provides a *less-than* method which allows edge objects to be sorted, which is crucial for Kruskal's algorithm. The vertices themselves are not needed by the algorithm. A list of edges and the *partition* data structure suffice for running Kruskal's algorithm.

7.6.3 An Edge Class

```
class Edge:
    def __init__(self,v1,v2,weight=0):
        self.v1 = v1
        self.v2 = v2
        self.weight = weight

    def __lt__(self,other):
        return self.weight < other.weight
```

Running Dijkstra's algorithm benefits from having both the edge and vertex objects. The weight of each edge is needed by the algorithm so storing the weight in the edge and associating vertices and edges is useful.

There are other potential representations for graphs. For instance, a two-dimensional *matrix* could be used to represent edges between vertices. The rows and columns of the matrix represent the vertices. The weight of an edge from vertex v_i to vertex v_j would be recorded at *matrix[i][j]*. Such a representation is called an *adjacency matrix*. Adjacency matrices tend to be sparsely populated and are not used much in practice due to their wasted space.

The chosen graph representation depends on the work being done. Vertices with adjacency information may be enough. An edge list is enough for the Kruskal's algorithm. Vertex and edge information is required for Dijskstra's algorithm. An adjacency matrix may be required for some situations. As programmers we need to be mindful about wasted space, algorithm needs, and efficiency of our algorithms and the implications that the choice of data representation has on our programs.

7.7 Chapter Summary

Graph notation was covered in this chapter. Several terms and definitions were given for various types of graphs including weighted and directed graphs. The chapter presented three graph theory algorithms: depth first search, Kruskal's algorithm, and Dijkstra's algorithm. Through looking at those algorithms we also explored graph representations and their use in these various algorithms.

After reading this chapter you should know the following.

- A graph is composed of vertices and edges.
- A graph may be directed or undirected.
- A tree is a graph where one path exists between any two vertices.
- A spanning tree is a subset of a graph which includes all the vertices in a connected graph.
- A minimum weighted spanning tree is found by running Kruskal's algorithm.
- Dijkstra's algorithm finds the minimum cost of reaching all vertices in a graph from a given source vertex.
- Choosing a graph representation depends on the work to be done.

- Some typical graph representations are a vertex list with adjacency information, an edge list, or an adjacency matrix.

7.8 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. In the definition of a graph, $G = (V, E)$, what does the V and the E stand for?
2. What is the difference in the definition of E in directed and undirected graphs?
3. In depth first search, what is the purpose of the *visited* set?
4. How is backtracking accomplished in depth first search of a graph? Explain how the backtracking happens.
5. What is a path in a graph and how does that differ from a cycle?
6. What is a tree? For the graph in Fig. 7.2 provide three trees that include the vertices 0, 1, and 10.
7. Why does Kruskal's algorithm never make a mistake when selecting edges for the minimum weighted spanning tree?
8. Why does Dijkstra's algorithm never make a mistake when computing the cost of paths to vertices?
9. What graph representation is best for Kruskal's algorithm? Why?
10. Why is the previous vertex stored by Dijkstra's algorithm? What purpose does the previous vertex have and why is it stored?

7.9 Programming Problems

1. Write a program to find a path between vertex 9 and 29 in the graph shown in Fig. 7.9. Be sure to print the path (i.e. the sequence of vertices) that must be traversed in the path between the two vertices. An XML file describing this graph can be found on the text website.
2. Modify the first problem to find the shortest path between vertices 9 and 29 in terms of the number of edges traversed. In other words, ignore the weights in this problem. Use breadth first search to find this solution.
3. Write the code and perform Dijkstra's algorithm on the graph in Fig. 7.9 to find the minimum cost of visiting all other vertices from vertex 9 of the graph.
4. Write the code and perform Kruskal's algorithm on either the directed graph in Fig. 7.9 or the undirected example found in the chapter. XML files for both graphs can be found on the text website.
5. Not every graph must be represented explicitly. Sometimes it is just as easy to write a function that given a vertex, will compute the vertices that are adjacent to it (that have edges between them). For instance, consider the water bucket

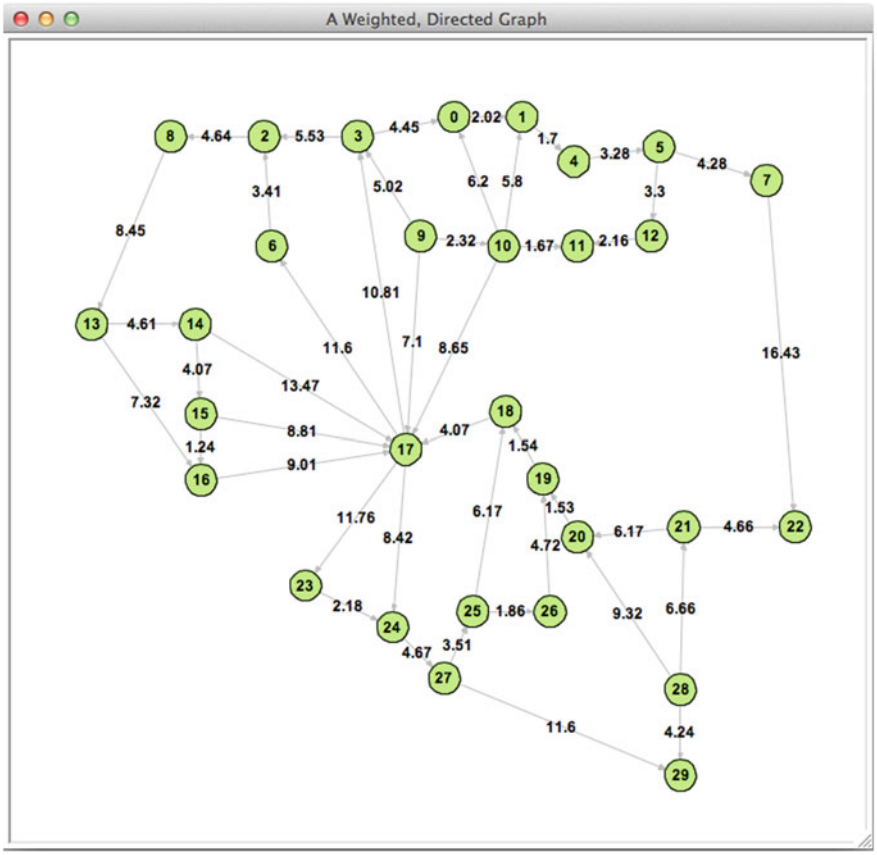

Fig. 7.9 A Sample Weighted, Directed Graph

problem. There are two buckets in this problem: a 3 gallon bucket and a 5 gallon bucket. Your job is to put exactly 4 gallons in the 5 gallon bucket. The rules of the game say that you can completely fill a bucket of water, you can pour one bucket into another, and you can completely dump a bucket out on the ground. You cannot partially fill up a bucket, but you can pour one bucket into another. You are to write a program that tells you how to start with two empty buckets and end with 4 gallons in the 5 gallon bucket.

To complete this problem you must implement depth first search of a graph. The vertices in this problem consist of the state of the problem which is given by the amount of water in each bucket. Along with the search algorithm you must also implement an adjacent function that given a vertex containing this state information will return a list of states that may be adjacent to it. It may be easier to generate some extra adjacent states and then filter out the unreasonable ones before returning the list from adjacent. For instance, it may be easier to generate a state with 6 gallons in the 5 gallon bucket and then throw that state out later

by removing states from the list which have more gallons than allowed in that bucket.

The program should print out the list of actions to take to get from no water in either bucket to four gallons in the five gallon pail. The solution may not be the absolute best solution, but it should be a valid solution that is printed when the program is completed.

6. A bipartite graph is a graph where the vertices may be divided into two sets such that no two vertices in the same set have an edge between them. All edges in the graph go between vertices that appear in different sets. A program can test to see if a graph is bipartite by doing a traversal of the graph, like a depth first search, and looking for odd cycles. A graph is bipartite if and only if it does not contain an odd cycle. Write a program that given a graph decides if it is bipartite or not. The program need only print *Yes, it is bipartite*, or *No, it is not bipartite*.
7. Extend the program from the previous exercise to print the set of vertices in each of the two bipartite sets if the graph is found to be bipartite.