A graph data structure is similar to a binary tree except that it can have any number of connections from a node to other nodes. The set of highways between towns and cities in Saskatchewan form a graph. A binary tree is just a restricted graph in that we can have only two outgoing references to children and one reference (optional) to the parent. In graph we could have references to siblings, and multiple parents, as well as friends, co-workers, neighbors, etc.

Conceptually, this is what a graph looks like:

A picture containing text, watch

Description automatically generatedDiagram, schematic

Description automatically generated

We can identify nodes (usually called vertices in a graph) as the the points where data are stored. The links between are known as edges. Binary trees and linked lists have edges as well; we just don't call them that. The graph is more robust and complex in that edges can be traveled both ways unless they are specifically directed meaning travel is allowed only in one direction (like the binary tree). The edges can also be weighted, meaning there is a cost to traveling over the edge. For example, there are multiple edges (routes) between Saskatoon and Edmonton. You could attach distance as a weight then choose the path with the shortest distance to travel. But perhaps cost or road condition may be the deciding factors. The shortest distance may in fact be a poor road in which case you will take longer to travel it than a super highway that may in fact be longer in distance but affords a higher travel speed. In this case the weighting may be travel time, not distance.

The following illustrations demonstrate the various types of graphs:

Diagram

Description automatically generated with medium confidenceDiagram

Description automatically generated with medium confidenceDiagram

Description automatically generatedChart, diagram

Description automatically generated with medium confidence

Undirected graphs contain edges which can be traversed in both directions. Directed graphs have one-way edges only. The directed graph nodes (green and red) have two edges: one in each direction. Cyclic graphs allow cycles: a path that loops back on itself. Acyclic graphs are not allowed to have loops. If a graph is undirected acyclic, then there can not be any shapes such as the green-orange-purple in the left most graph shown here.

How we represent the graph in memory will determine our implementation characteristics. Obviously we can't use the binary tree approach by adding extra references to children because we don't know, ahead of time, how many children a given vertices will have. There are typically two representations for graphs that support graphs needs well.

**Adjacency Matrix**

An adjacency matrix is a two dimensional array with the vertices providing the index into the dimensions and the array value a 0 or 1, indicating whether or not an edge exists between the start vertex and the end vertex. Here are the adjacency matrices for the above graphs:

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generatedA picture containing text, keyboard

Description automatically generated

Given the arrays shown, we can make a number of observations:

* From any vertex, we can determine if a path exists to any other vertex. For example, in the undirected graph, can we get from cyan (C) to blue (B)? Beginning with the row C, we look into column B --> 0: there is no direct path. However, from C we can go to purple (P). From P is no direct path to B, but P can go to green (G), orange (O) and C. But we came from C so that is no good. Let's say we go to G. From G we see an edge to P, O, yellow (Y) and red (R). Let's suppose we follow edge to R. From R we can go to G and B. We have arrived, so our path exists: C-->P-->G-->R-->B. Of course this is determined in part by how we write the path-finding algorithm since another path exists: C-->P-->O-->G-->R-->B. Also, C-->P-->G-->O-->P-->G-->R-->B is valid although some edges are crossed multiple times.
* The number of outgoing edges is determined by summing all the 1's in the row of the vertex.
* The number of incoming edges is determined by summing all the 1's in the column of the vertex.
* In an undirected graph, the outgoing edges = the incoming edges.

The biggest issue with this representation is the large number of 0's relative to 1's. This is very space inefficient if the graph is large (imagine thousands or millions of vertices common in graph databases). The next approach to graph representation uses a sparce matrix.

**Sparse Matrix**

In a sparse matrix, the node is stored in a one-dimensional array or even hash table. The key of the node identifies the location of a list of nodes reachable directly from that node. Consider the undirected graph. We can implement the adjacency in the following manner:

Table

Description automatically generatedTable

Description automatically generatedTable

Description automatically generatedTable

Description automatically generated

In the above sparce matrix, the linked list to the right of each node is the list of nodes directly reachable from the given node. For example, from G, we can reach each of the nodes Y, O, P and R in one edge. The list does NOT represent paths between nodes. So based on the sparce matrix, we can again answer the question, can we get from cyan (C) to blue (B)? Here is how we can answer that question:

* Search the list for C to see if it contains B. If it does we are finished.
* If it doesn't contain B, add C-->P (first adjacent vertex from C) to a possible path and check P next.
* If P contains B we are finished. If not, check O, C-->P-->O.
* If O contains B, we are done. If not, the first adjacent vertex from O is G.
* If G contains B, we are done with C-->P-->O-->G. If not the first adjacent vertex is Y. Y goes back to G only so this is a deadend. The next vertex is O but O is already on our possible path so that would be backtracking. The next vertex is P but it is also on our possible path. The last vertex is R. our possible path is now C-->P-->O-->G-->R.
* We check R and find B in its list, so our final path is C-->P-->O-->G-->R-->B.

Notice that there are other paths that may be shorter or longer. It really depends on the order in which the lists are created. Since we don't know when we begin what the graph is going to look like, it really doesn't matter how we create the lists. Ordering the entries will help because looking for a vertex in the list is a standard search. Therefor, another approach to storing the vertex adjacency data would be to use an AVLtree. We computer the key for the vertex we are interested in (here, the letters in the boxes), then maintain AVLtrees of the vertices that are directly reachable from that vertex. The ordering of the adjacent vertices is strictly to support searching - it provides no other benefit to the graph.