Chapter B3: Paths, Graphs, and Search

B3.1 Breadth-first Search

A search is a systematic way of searching through the nodes for a specific node. The two standard searches are breadth-first search and depth-first search, which both run in time proportional to the number of edges of the graph.

The idea behind breadth-first search is to:

Visit the source; then all its neighbors; then all their neighbors; and so on.

If the graph is a tree and one starts at the root, then one visits the root, then the root’s children, then the nodes at depth 2, and so on. That is, one level at a time. This is sometimes called level ordering. BFS uses a queue: each time a node is visited, one adds its (not yet visited) out-neighbors to the queue of nodes to be visited. The next node to be visited is extracted from the front of the queue.

```
BFS (start):
  enqueue start
  while queue not empty do {
    v = dequeue
    for all out-neighbors w of v do
      if ( w not visited ) {
        visit w
        enqueue w
      }
  }
```

B3.2 Depth-First Search

The idea for depth-first search (DFS) is “labyrinth wandering”:

keep exploring new nodes from current node; when get stuck, backtrack to most recent node with unexplored neighbors

In DFS, the search continues going deeper into the graph whenever possible. When the search reaches a dead end, it backtracks to the last (visited) node that has un-visited
neighbors, and continues searching from there. A DFS uses a **stack**: each time a node is visited, its unvisited neighbors are pushed onto the stack for later use, while one of its children is explored next. When one reaches a dead end, one pops off the stack. The edges/arcs used to discover new nodes form a tree.

**Example.** Here is graph and a DFS-tree from node $A$:

![Graph and DFS-tree](image)

$$\text{DFS}(v):$$

$$\text{for all edges } e \text{ outgoing from } v \text{ do }$$

$$\quad w = \text{other end of } e$$

$$\quad \text{if } w \text{ unvisited then }$$

$$\quad \quad \text{label } e \text{ as tree-edge}$$

$$\quad \quad \text{recursively call DFS}(w)$$

$$\}$$

**B3.3 Test for Strong Connectivity**

Note that both BFS and DFS visit all nodes that are reachable. Thus they provide a linear-time test for connectivity in undirected graphs.

A directed graph is **strongly connected** if one can get from every node to every other node. Here is an algorithm to test whether a directed graph is strongly connected or not:

**Strong Connectivity**

1. Do a DFS from arbitrary node $v$ and check that all nodes are reached
2. Reverse all arcs and repeat

Why does this work? Think of node $v$ as the hub...
B3.4 Dijkstra’s Algorithm

The following question comes up often. What is the quickest way to get from $A$ to $B$? This is known as the shortest-path problem. The underlying structure is a graph. The graph need not be explicitly precalculated. It could be the state graph of a finite automaton, the search graph of an AI problem, or the position graph of a game.

If we are just interested in finding the shortest path from one node to another node in a graph, then the famous algorithm is due to Dijkstra. It essentially finds a breadth-first search tree.

We grow the tree one node at a time. We define the auxiliary function $\text{currDis}(v)$ for a node $v$ as the length of the shortest path to $v$ subject to the restriction that the penultimate node is in the current tree. At each stage we add to the current tree that node which has the smallest value of $\text{currDis}(v)$. We then update the value of $\text{currDis}(v)$ for the remaining nodes.

The following produces the distance from node $a$ to all other nodes.

\begin{verbatim}
ShortestPath (G:graph, a:node)
    for all nodes v do currDis(v) ← infinity
    currDis(a) ← 0
    remainder ← [ all nodes ]
    while remainder nonempty do {
        let w be node with minimum value of currDis
        remainder ← remainder \{ w \}
        for all nodes v in remainder do
            currDis (v) ← min ( currDis(v), currDis(w)+length(w,v) )
    }
\end{verbatim}

\textbf{Example.} For the following graph, the table gives the value of currDis at each stage.

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
  & A & B & C & D & E & F \\
\hline
 A & 0 & 5 & \infty & 11 & \infty & 13 \\
 B & & \infty & \infty & \infty & 8 & \infty \\
 C & & & \infty & 8 & \infty & 8 \\
 D & & & & \infty & \infty & 15 \\
 E & & & & & \infty & \infty \\
 F & & & & & & \infty \\
\hline
\end{tabular}
\end{center}
Complexity: quadratic. We go through the while loop about \( n \) times and the update loop takes \( O(n) \) work. To potentially speed up, use a priority queue that supports \texttt{decreaseKey}. See section on Fibonacci heap later.

### B3.5 The All Pairs Shortest Path Problem

Suppose we wanted instead to calculate the shortest path between every pair of nodes. One idea would be to run Dijkstra with every node as source node.

Another algorithm is the following dynamic programming algorithm known as Floyd–Warshall. Suppose the nodes are ordered 1 up to \( n \). Then we define

\[
d_m(u, v) \text{ as the length of the shortest path between } u \text{ and } v \text{ that uses only the nodes numbered 1 up to } m \text{ as intermediates.}
\]

The answer we want is \( d_n(u, v) \) for all \( u \) and \( v \). (Why?)

There is a formula for \( d_m \) in terms of \( d_{m-1} \). Consider the shortest \( u \) to \( v \) path that uses only nodes labeled up to \( m \)—call it \( P \). There are two possibilities. Either the path \( P \) uses node \( m \) or it doesn’t. If it doesn’t, then \( P \) is the shortest \( u \) to \( v \) path that uses only nodes up to \( m - 1 \). If it does use \( m \), then the segment of \( P \) from \( u \) to \( m \) is the shortest path from \( u \) to \( m \) using only nodes up to \( m - 1 \), and the segment of \( P \) from \( m \) to \( v \) is the shortest path from \( m \) to \( v \) using only nodes up to \( m - 1 \).

Hence we obtain:

\[
d_m(u, v) = \min \left\{ d_{m-1}(u, v), d_{m-1}(u, m) + d_{m-1}(m, v) \right\}
\]

The resultant program iterates \( m \) from \( m = 0 \) to \( m = n - 1 \). Each time there are \( O(n^2) \) values of \( d_m(u, v) \) to be calculated, and each calculation takes \( O(1) \) time. Hence, we have an \( O(n^3) \) algorithm. (Same as Dijkstra but runs faster.) Note the storage requirements.

### Exercises

1. Implement Floyd–Warshall using your favorite programming language.
   The program should take the input graph in the form of a text file provided by the user. The first line is the number of nodes. Each remaining line is an edge: three integers in order provide the number of each node and then the weight. The end of the input is signified by the triple 0 0 0.

2. Suggest ways in which the efficiency of Floyd–Warshall might be improved.
3. Illustrate the steps of Dijkstra, using node $A$ as source, on the graph in Exercise 2 of Chapter B1.

4. Sometimes graphs have edges with negative weights. Does the concept of distance still make sense? Do the above algorithms still work? Where might one find such graphs? Discuss carefully.