CITS2200 Project Editorial

Graph Representation

Consider a graph containing V vertices and E edges. There are two common representations of graphs: The adjacency list, and the adjacency matrix. An adjacency list is a list, for each vertex of all the vertices that are adjacent to this vertex (that is, all the edges leaving this vertex). An adjacency matrix is a $V \times V$ matrix where the element at row i, column j represents the edge between vertex i and vertex j. An adjacency list uses $O(E)$ memory, and has the benefit of allowing us to enumerate the edges leaving a vertex in linear time, whereas an adjacency matrix would require us to check every possible edge, taking $O(V)$ time. An adjacency matrix uses $O(V^2)$ memory and has the benefit of allowing us to check the existence of an edge in constant time, whereas to perform the same check in an adjacency list is linear in the number of edges leaving the vertex the edge originates from, which is worst case $O(V)$. As you will see, the algorithms used in this project do not require us to ever test for the existence of an edge, but often require us to enumerate all edges incident to a vertex. For this reason this project uses an adjacency list to represent the graph.

In this project, our vertices are represented by strings, the URL of the Wikipedia page. Indexing by these strings can be messy and inefficient, so we instead assign each vertex a unique integer ID between in the range $[0, V)$. This ID will serve as an index into the adjacency list, allowing us to find a vertex's list of neighbours in constant time. To allow us to convert back and forth between the string and integer representations of our vertices, we introduce a list of strings that can be indexed efficiently by vertex ID, and a map from the vertex URL to its ID.

Adding an edge to the graph requires us to first make sure both vertices exist in the graph. To this end, we introduce the `addVert()` function that checks if a vertex exists using our map from URL to ID, and adds it to the graph if it does not. We then need simply add an entry to the adjacency list to represent the new edge.

Question 1: Shortest Path

In this project, our vertices represent Wikipedia pages, and our edges represent links from one page to another. This means that our graph is directed, but unweighted. In an unweighted graph the length of the shortest path from one vertex to another is the minimum number of edges that must be traversed in order to get from our source to our destination. We can find the lengths of these shortest paths by performing a Breadth First Search (BFS), which enumerates vertices according to the number of edges they are away from our starting vertex. That is, a BFS will visit our starting vertex, then all vertices that are one edge away, then two, and so on. By maintaining an array of distances from our starting vertex, we can fill this array in as we perform our BFS. When the BFS has finished, our array will hold the length of the shortest path from our source to each vertex it can reach, or the original value of the array if no such path exists. A BFS visits each vertex at most once, and for each vertex iterates over every edge leaving that vertex. This means every vertex and every edge will be considered at most once, giving our BFS an expected and worst-case time complexity of $O(V+E)$, which is optimal.

Question 2: Hamiltonian Path

A Hamiltonian path is any path through a graph that visits every vertex exactly once. Finding such a path is a well-known computationally intensive problem, which is why this question specified it would be run for at most 20 vertices. There is no known polynomial time solution to this problem. A common algorithm for this problem is a backtracking Depth First Search (DFS), which works much like a regular DFS, but if it runs into a dead end without having found a Hamiltonian path, it backs out the way it came, marking vertices it is leaving as unvisited. This causes it to explore every path originating from its starting vertex. Repeating this process for each starting vertex allows us to explore every simple path in the graph. If this algorithm fails to find a Hamiltonian path, then there is none. In the worst case, there are as many simple paths as there are permutations of vertices, and at every step in the backtracking DFS we must consider every potential next vertex, giving a worst-case time complexity for this algorithm of $O(V\ V!)$. There exists a better solution, however.

The best known solution to this problem uses dynamic programming and is known as the Held–Karp or Bellman–Held–Karp algorithm. We can think about this program as answering the question "Is there a path ending at vertex i that visits every vertex in the subset P exactly once?". The answer to this question is a boolean, and this question can be answered in terms of smaller questions of the same format, which gives us a recurrence relation: Such a path exists if and only if we can remove the vertex i from P, and there still exist some vertex in P for which the question is true and that has an edge going to i. This breaks the question down until we end up with a trivial base case of the form "Is there a path ending at the vertex i that visits i exactly once?" which is plainly true. There are $O(2^V)$ subsets of vertices in the graph, meaning there are $O(V\ 2^V)$ questions of the form above. If we represent these subsets as a bitset, using each binary digit in an integer to represent whether the corresponding vertex is in the set or not, we can store the answer to each question as it is computed, meaning we will never have to recompute an answer. The answer to each question is defined in terms of $O(V)$ other answers. The code given in the reference solution computes every answer in an order such that the answers needed to compute the answer to a question are always computed before that question itself. We then use a similar method to perform a traceback; that is, to use the answers we have computed to reconstruct a Hamiltonian path, if one exists. This algorithm has a worst-case time complexity of $O(V^2\ 2^V)$, which is the best known.

Question 3: Strongly Connected Components

A Strongly Connected Component (SCC) is a set of vertices that can all reach each other. This implies that a vertex belongs to exactly one SCC, which may even be just that vertex. This question asks us to partition the vertices of the graph into the SCCs that contain them. There are a several known optimal algorithms, including, in chronological order of publication, Tarjan's algorithm, Dijkstra's Path-Based SCC algorithm, and Kosaraju's algorithm. In the reference solution, we have chosen to implement Kosaraju's algorithm as it is the simplest to implement. Kosaraju's algorithm is based on the observation that the SCCs in the original graph are the same as those in the transpose graph (that is, the graph with all edges reversed). This is because for the vertices i and j to be in the same SCC, there must be a path from i to j and from j to i, and so in the transpose graph there must be therefore be a path from j to i and from i to j, meaning they are in the same SCC. We can easily modify `addEdge()` in order to maintain a copy of the transpose graph as well as the original. We can compute the set of all vertices a vertex can reach by performing a DFS starting at that vertex. Doing this is in both the original graph and the transpose graph is sufficient to compute the SCC to which this vertex belongs, but requires computing the intersection of these two sets, which can be computationally expensive. Kosaraju's algorithm uses a property of the DFS order through the original graph in order to ensure that the DFS through the transpose graph only explores this intersection. The observation is that any vertex whose subtree was explored before another in the DFS order (this is called post-order) either must not have a path to that other vertex, or is a descendant of it in the DFS tree. We can therefore DFS through the transpose graph starting from the last vertex in post-order, and any vertices it visits must be part of its SCC, as any vertex that is earlier in the post-order that our starting vertex can reach must therefore have been able to reach and be reached from this starting vertex in the original graph. We can then DFS again from the next highest vertex in the post-order that is not yet visited in order to find all its SCC, and so on until we have found all the SCCs. This algorithm requires just two full-graph DFSs, each of which visit every vertex exactly once and consider each edge leaving a vertex exactly once, giving a time complexity for Kosaraju's algorithm of $O(V+E)$, which is optimal.

Question 4: Graph Centers

The center of a graph can be thought about in the same terms as the center of a circle: It is the point from which the most distant other point is as close as possible. In a graph we can think about this as the vertex from which the most distant other vertex (by shortest path distance) is as close as possible; that is, the vertex with minimum eccentricity. Unlike circles, however, the definition is not wholly unambiguous: It is possible for a graph to have multiple centers, and for there to be vertices that are unable to reach another vertex. This question permits us to make our own decision as to how to define centers in the case of a disconnected graph, and so for the reference implementation we have chosen to ignore nonexistent paths when computing the eccentricity of a vertex. We did not test for ambiguous cases, and any reasonable way of handling them was accepted. To find the set of vertices with minimum eccentricity, we need simply compute the eccentricity of each vertex, which is the length of the maximum length shortest path from this vertex to any other. There are many ways of computing this, but we conveniently already have an optimal method for computing the lengths of the shortest paths to each vertex: Our BFS from question 1. To compute the eccentricity of a vertex we need simply perform a BFS from that vertex and take the maximum value from the distances array produced. Doing this once for each starting vertex and maintaining a list of vertices with minimum eccentricity gives us a list of graph centers. This method requires us to perform one BFS per vertex, giving an overall time complexity of $O(V \times (V + E))$, which reduces to $O(V^2 + V\ E)$, which is optimal.

Common Mistakes

We observed a number of common mistakes while marking this project. Here we have compiled a list of some of them, with explanations where appropriate:

Using and misanalysing an adjacency matrix: An adjacency matrix takes $O(V^2)$ time to build and requires $O(V)$ time to find all neighbours of a vertex. This means that BFS and DFS are no longer $O(V + E)$, but rather $O(V^2)$. This made many people's solutions to all problems slower than expected.

Transposing an adjacency matrix: There is no reason to transpose an adjacency matrix; It takes $O(V^2)$ time to do so. Just swap the indices: `adjMat[i][j] = transpose[j][i]`

Using Dijkstra's algorithm: Many people solved question 1 using Dijkstra's algorithm, which in an unweighted graph has the exact same function as a BFS, but is slower. Many people also mistook Dijkstra's algorithm for BFS and vice versa.

Misanalysing Dijkstra's algorithm: Dijkstra's algorithm was also commonly analysed incorrectly. The $O(E + V\ log\ V)$ worst-case complexity given on Wikipedia is only when using a special data structure called a Fibonacci heap as the basis for the priority queue. Java does not include Fibonacci heap, and nor did any project submissions. Using a more typical binary heap based priority queue, such as the one in the Java standard library gives $O(E\ log\ V + V\ log\ V)$. A linear time priority queue gives $O(E\ V\ +\ V^2)$.

Using `Math.pow(2, n)` instead of `(1 << n)`: Java's `Math.pow()` function is not constant time, but is rather logarithmic in the power. This introduced an $O(log\ n)$ factor that is not present when using bit-shifts.

Not trying every start vertex for Hamiltonian Path: If you are using the backtracking DFS to solve Hamiltonian path, you must remember to try every vertex as a potential starting vertex. This is likely due to people adapting Hamiltonian cycle algorithms.

Incorrectly claiming to use Held–Karp: Many people reported using the Held–Karp DP algorithm described above, while actually using backtracking or another factorial algorithm.

Misanalysing calling `getShortestPath()` for all pairs of vertices: Many people computed vertex eccentricities for `getCenters()` by first calling their `getShortestPath()` method for all pairs of vertices using a pair of nested for loops, and then analyzed this incorrectly as being $O(V^2)$ when it is actually $O(V^2 \times (V + E))$ or worse depending on the shortest path algorithm used.

Using Floyd-Warshall: Floyd-Warshall is a good All Pairs Shortest Path algorithm for weighted graphs. Our graph is unweighted, however, meaning repeated BFS is better.

Returning the wrong answer for edge cases: Many people forgot to follow the spec in several cases. These included not returning -1 when no shortest path exists, not returning 0 for the shortest path from a vertex to itself, not returning an empty array when no Hamiltonian path exists, and returning arrays containing nulls. In Java a 2D array is just an array of arrays, and so can be jagged rather than square.

Not implementing the interface: An interface was provided to ensure that your code was compatible with our testing systems. Many people either failed to implement the interface correctly or modified the interface. Be glad we bothered to mark your submissions.

Not compiling or missing code: Many submissions did not compile, often due to missing code. Where possible we fixed them. Be glad we bothered to mark your submissions.