Hamiltonian Path Search Using Dijkstra's Algorithm

John Dodzweit
Florida Institute of Technology, Orlando FL
Computational Complexity, CSE 5610

ABSTRACT

Finding the shortest Hamiltonian Path in an undirected weighted graph can be found using Dijkstra's algorithm. The problem is NP-Complete because 3-CNF, a known problem in the NP-Complete class, is reducible to the Hamiltonian Path and a solution can be verified in polynomial time. This paper discusses the results of computing Hamiltonian paths using Dijkstra's shortest-path algorithm on randomly generated graphs.

Index Terms

Hamiltonian path, Dijkstra's Algorithm, NP-Complete problems, computational complexity theory

1. INTRODUCTION

A Hamiltonian path is a path that visits every node in a connected graph without traveling over any edges in the graph more than once[1]. Dijkstra's algorithm is a method for finding the shortest path between any two nodes in a graph. Using Dijkstra's algorithm to find a Hamiltonian path allows one to find the path that is most likely to be the shortest path required to visit every node in a graph.

The mathematical derivation for finding the worst case asymptotic curve for solving Hamiltonian path searches in a weighted undirected graph is extremely complex and remains beyond the scope of this paper. The complexity class can be determined empirically by implementing Dijkstra's algorithm and measuring the run-time required to solve several graphs which vary in node count and branch-factor. This paper presents a brief overview of the methodology of finding Hamiltonian paths using Dijkstra's algorithm, and then provides a summary of the results obtained from implementing and testing the algorithm.

2. DIJKSTRA's ALGORITHM

Dijkstra's algorithm was invented by Edsger Dijkstra in 1956. The algorithm works by always choosing the shortest edge leading out to a neighboring node. This is repeated until the end node is reached or until there are no neighboring nodes that have not been visited. If all neighboring nodes have already been visited, the algorithm back-tracks by one node and tries the next shortest length edge [2].

Branch-factor, also referred to as fan-out, in a connected graph, is defined as the number of edges that each vertex is connected to. Using Dijkstra's algorithm, the search for a Hamiltonian path is the fastest when the fan-out is very high, such as one less than node count, or when the fan-out is extremely low, such as a fan-out of one or two. When the fan-out is close to the node count, almost every node is connected to every other node and therefore the probably that Dijkstra's algorithm will find a Hamiltonian path on the first try, without backtracking, is very high. As the fan-out approaches a single edge per node, the probability is also very high that a Hamiltonian path will be found on the first try.

The run-time estimates for Dijkstra's algorithm approaches O(n) at very large and very small branch-factors, however, a worst-case of $O(2^n)$ or greater applies when there are a significant number of nodes and the fan-out is around 10 to 50 percent of the number of nodes. This is because there are only a few working Hamiltonian paths compared to the extremely high number of possible dead-end paths. The total number of paths, including dead-end paths where not every node is reached, is related to the number of permutations of the number of nodes in the graph. It is exponentially related to fan-out, but bounded on each end to O(n) where fan-out is either one, or one less than the number of nodes. As the number of nodes increases to several dozen and the fan-out is approximately 10 to 50 percent of the number of nodes, the number of possible paths that can be taken from any node become extremely large.

3. COMPLEXITY CLASS

Finding the shortest Hamiltonian path using Dijkstra's algorithm falls into the NP-Complete complexity class. A problem is NP-Complete if a solution can be verified quickly in polynomial time and the problem is also in the NP-Hard category[3].

Once a Hamiltonian path has been determined in a connected graph, it can be verified in O(n) time because the path only has to be traversed once to prove the solution. Finding the path, however, is difficult and very time consuming, especially for large node count graphs having medium to low branch factors.

Hamiltonian path searches are NP-Hard because the time and space required grow exponentially as the node count increases and fan-out is kept to a small percentage of node count. As each path is attempted, it must be stored in memory so that the program will not retry the same path again. This means that the program's memory usage will grow in size during the entire length of execution. The space required to hold the program's path history will grow in exact proportion to the amount of time that the program has been running.

The worst case run-time for finding a Hamiltonian path using Dijkstra's algorithm would occur in the case in which there is only one possible Hamiltonian path and it were the last path attempted after all other possible paths had been tried. Determining the number of possible paths in an undirected graph can be very complex. In 1995, McKay and Robinson derived the following equation for determining the total number of Eulerian circuits in an undirected graph[4]:

$$ec(K_n) = 2^{(n+1)/2}\pi^{1/2}e^{-n^2/2+11/12}n^{(n-2)(n+1)/2}\left(1+O(n^{-1/2+\epsilon})\right)$$

This has been provided to demonstrate how complex the the mathematical analysis is when analyzing the number of ways a

graph can be traversed. For the problem described in this paper, the above equation could be used to find the total number of Hamiltonian cycles. That number would then be subtracted from the total number of possible paths, including sub-paths that dead-end before all nodes are visited. The derivation is lengthy and is outside the scope of this paper. It is clear, however, that this type of problem falls into the NP-Hard category due to the 2^n nature of the equations governing this family of search algorithms.

4. IMPLEMENTATION AND TEST

A Microsoft .NET console application was written in C# to generate random undirected weighted graphs and then search each one to find a Hamiltonian path. Figure 4-1 shows an example of a random 10-node graph with a fan-out of 4. The graphs are represented as a table with node numbers listed in the first column and first row of the table. The values in the table represent weights or distances of the edges connecting the nodes represented by the row and column of each cell.

The following method is the main algorithm used to perform Dijkstra's search. All console input/output calls have been removed, in this case, for clarity.

```
static void DijkstraSearch(ref int[] graph, int nodes) {
  while (true)  {
    if (currentPath.Count >= nodes)  break;
    ArrayList edges = GetEdges(ref graph, nodes,
        (int)currentPath[currentPath.Count - 1]);
    bool foundNode = false;
    while (edges.Count > 0)  {
      Edge shortest = GetShortestEdge(edges);
      bool cameFrom = currentPath.Contains(shortest.node);
      currentPath.Add(shortest.node);
      String currPathStr = PathToString(currentPath);
      currentPath.RemoveAt(currentPath.Count - 1);
      bool pathTried = triedPaths.Contains(currPathStr);
      if (pathTried || cameFrom) edges.Remove(shortest);
      else  {
        currentPath.Add(shortest.node);
        lastNode = (int)currentPath[currentPath.Count - 1];
        foundNode = true;
        break;
      }//end else
    }//end while edges count > 0
    if (foundNode) continue;
    if (currentPath.Count == 1) return; //no Hamiltonian path exits
    triedPaths.Add(PathToString(currentPath));
    currentPath.RemoveAt(currentPath.Count - 1);
  }//end while (true)
}
```

For each node, the shortest edge that has not already been traveled, is chosen. If there are no connected nodes that have not already been visited, the algorithm backtracks by one node and tries the next shortest edge on that node. Each path attempt is stored as a string of node values and added to an array called 'triedPaths'. This array is searched each time a node is about to be visited to ensure that the path has not already been tried.

```
Enter node count: 10

Enter edge fan out per node: 4

Graph Generated:

     0    1    2    3    4    5    6    7    8    9
 0   0    0   11   10   24    0    0    0   98    0
 1   0    0    0    0   63    0   81    0   53   60
 2  11    0    0   20    0    0   55   53    0    0
 3  10    0   20    0    0   96   61    0    0    0
 4  24   63    0    0    0    0    0   67    0   56
 5   0    0    0   96    0    0   56   86    0   61
 6   0   81   55   61    0   56    0    0    0    0
 7   0    0   53    0   67   86    0    0   29    0
 8  98   53    0    0    0    0    0   29    0   57
 9   0   60    0    0   56   61    0    0   57    0

Performing Dijkstra's Algorithm starting at node 0....

Moving to node 3
Moving to node 2
Moving to node 7
Moving to node 8
Moving to node 1
Moving to node 9
Moving to node 4
Backing up to node 9
Moving to node 5
Moving to node 6
Backing up to node 5
Backing up to node 9
Backing up to node 1
Moving to node 4
Moving to node 9
Moving to node 5
Moving to node 6
All nodes have been visited!

The following path was successful:
0 -> 3 -> 2 -> 7 -> 8 -> 1 -> 4 -> 9 -> 5 -> 6
```

Figure 4-1 Console application that generates random undirected weighted graphs and searches for a Hamiltonian path using Dijkstra's algorithm.

In Figure 4-1 the search begins at node 0. Allowing the edge weights to be unitless distances, node 0 is 11 units away from node 2, 10 units away from node 3, 24 units away from node 4, and 98 units away from node 8. Since node 3 is the closest, at only 10 units away, node 3 is visited first. From node 3, the search visits node 2, which is only 20 units away. From node 2, the nearest unvisited node is node 7. The search continues until it arrives at node 4 and finds that every neighboring node has already been visited. It then backs up to node 9 and continues the search starting at the second nearest node, since node 4 did not succeed. When the search finally completes, the successful Hamiltonian path is printed to the console.

The console application's "Run multiple graphs & record run times" feature allows the user to enter a starting node count, ending node count, node count increment value, and number of fan-out divisions per node count. As an example, an input of "100,1000,50,10" will starting creating graphs and running the Hamiltonian path search at a node count of 100 and will end at 1000 nodes. It will increment the node count by 50 for each run.

For a node count of 100 and a fan-out division of 10, graphs with branch-factors of 90, 80, 70, 60, 50, 40, 30, 20, and 10 will be created and searched in that order.

In a typical scenario, once the branch factor decreases to about 20 to 30 percent of the node count, the run-time becomes too long to complete in a reasonable time frame.

5. RUN-TIME RESULTS

For each node count, the run-time increased exponentially as fan-out is reduced. The run-time curves are not smooth, but have a great amount of variation from one fan-out to the next. This is due to the fact that each graph is randomly created and finding a Hamiltonian path could occur within a very few tries, coincidentally, or it could take an extremely long time.

From the graphs in Figure 4-1 it is evident that run-times rise exponentially as fan-outs decrease to a small percentages of the node count. It should be noted that the y-axis scale is logarithmic, and even with each scale division increasing by a factor of 10, the curves appear to rise extremely fast. Each of the curves in Figure 5-1 were performed until a fan-out was reached that took an indeterminate amount of time. The next data point on the left-most end of each curve, if measured, would on average, be very far above the range of the chart.

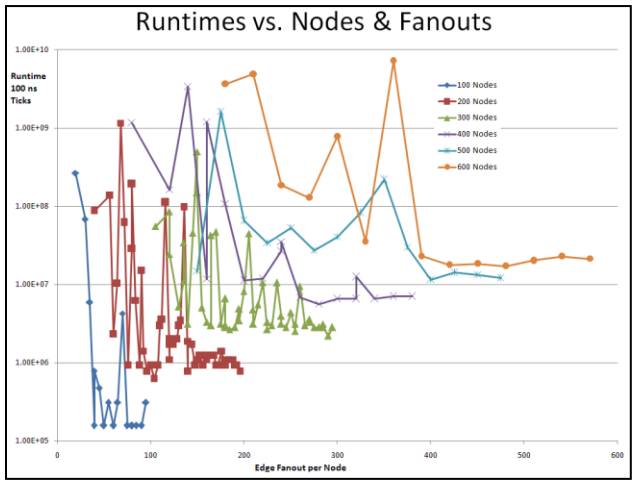

Figure 5-1 Run-time graphs for several node counts. The x-axis is the fan-out per node and y-axis is the run-time in increments of 100ns ticks. Each curve contains data for several fan-out values for a given node count.

The graphs in Figure 5-1 demonstrate both the NP-Hard characteristic and the polynomial time verification ability of the Hamiltonian path search. Many solutions are found very quickly even in high node-count graphs with a low branch-factor ratio. Under similar node counts and fan-outs, the solution may also take very long, revealing the NP-Hard complexity of the problem.

6. CONCLUSION

Dijkstra's shortest path algorithm can be used to find the shortest Hamiltonian path in a weighted undirected graph. The problem falls into the complexity class of NP-Complete because it is NP-Hard and a solution can be verified in polynomial time. The time and space required to find a Hamiltonian path grows exponentially fast as the fan-out count per node is reduced to about 20 to 30 percent of the node count.

7. REFERENCES

[1] Wikipedia, November 21, 2010, "Hamiltonian Path" http://en.wikipedia.org/wiki/Hamiltonian_path

[2] Wikipedia, November 23, 2010, "Dijkstra's Algorithm" http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[3] Wikipedia, November 18, 2010, "NP-Complete" http://en.wikipedia.org/wiki/NP-Complete

[4] Brendan McKay and Robert W. Robinson, Asymptotic enumeration of eulerian circuits in the complete graph, *Combinatorica*, 10 (1995), no. 4, 367–377.

John Dodzweit is a Staff Software Engineer at Lockheed Martin, Information Systems in Orlando, FL. He received a BSEE degree from the University of Central Florida in 1994, a MSEE degree from Florida Institute of Technology in 2002, and is currently completing the MSCS degree from Florida Institute of Technology in 2010. In 2005 he was awarded the patent for a 20 Gbps network transceiver which extends the bandwidth of multimode optical-fiber based infrastructure