Dijkstra Algorithm Heuristic Approach for Large Graph

1Sahar Idwan and 2Wael Etaifi
1Department Computer Science and Application, Hashemite University, P.O. Box 150459, Zarqa 13115, Jordan
2Department of Computer Science and Application, Al-Balqa' Applied University, Al-Salt, Jordan

Abstract: In this study, we developed an algorithm that improves a Dijkstra implementation in a large and dense graph based on hMetis partitioning. hMetis is a hyper graph partitioning algorithm that divides the massive graph into sub-graphs. The algorithm is called h-Dijkstra consists of pre-processing the flat graph by partitioning it onto a sub graph then building a two-level hierarchal graph. The shortest path algorithm computes on the top-level graph. Our experimental results show the dominance of our algorithm over the traditional Dijkstra algorithm and other alternative solutions based on the time and the number of Input/Output (I/O) operations.

Key words: h-Dijkstra, Dijkstra, hMetis, graph, hierarchal graph, RAM

INTRODUCTION

Dijkstra algorithm computes a Single Source Shortest Path (SSSP) in weighted graphs with n nodes and m edges from a node to all others (Comen et al., 2001). Goldberg (2001a, b) presented the algorithms for the single-source shortest path problem. Their labeling algorithm based on multi-level bucket shortest path algorithm that used to determine the length of the edges. We also can use the Breadth First search algorithm to compute the shortest paths for unweighted graphs. It is easy to apply the Dijkstra on the small graph since it fits in the memory, as for large graph, Dijkstra poses many challenges related with its storage in the memory and the I/O operations. Different algorithms are introduced to overcome the difficulties related with computing the shortest path using Dijkstra on the massive graph based on partitioning the graph into a set of fragments that fit in the memory.

This study introduces an acceleration algorithm for enhancing Dijkstra computation (h-Dijkstra) on a large dense graph based on the RAM model. The main idea is dividing the large graph into the set of sub-graphs that fit in the memory by using hMetis. The partition techniques reduce the number of crossing edges between the sub-graphs, which reduce the number of I/O operations. Then we build the two-level hierarchal graph that used to compute the shortest path.

Chan and Zhang (2001), Jing et al. (1998), Jing et al. (1996), Huang et al. (1996) and Shekhar et al. (1997) presented a disk based shortest path algorithms by partitioning the graph into a set of square-like blocks depends on spatial proximity. Jing et al. (1998) proposed building the hierarchal graph from the flat graph based on fragmentation, they pushed all border nodes, that nodes belong to more than one fragment to generate the next level. Another approach proposed by Huang et al. (1996) is to sort all edges by the x coordinates of their source nodes and then applies a plane sweep technique to sweep all x-sorted edges from left to right. The sweeping process stops periodically to sort the edges swept since the last storage by the y-coordinates of their origin nodes. This technique is applicable only if coordinates are available for the vertices of the graph and usually yield partitions that are worse than those partitioned by other partitioning techniques.

Huang et al. (1995), Houtsma et al. (1991), Jung and Pramanik (1996) and Shekhar et al. (1993) proposed hierarchal graph models but they are not guaranteeing the optimality of the retrieved path.

Gutman (2004), Meyer (2001) and Schulz et al. (2002) are presented the hierarchical decomposition based on the geometric information. Gutman (2004) computed the shortest path by pruning the nodes that are far a way from the possible shortest path based on the "reach" information. This information computed from the original
graph and it represents the importance of the node. The reach value and the Euclidean information associated with each node. His work is time consuming in the preprocessing phase and need more assumptions about the input.

Meyer (2001) presented a complicated algorithm to compute the shortest path algorithm for edges drawn from a uniform distribution with a linear complexity.

Schulz et al. (2002) built a multi-level graphs by adding one level of edges to the original graph based on different criteria for including vertices on the subsets such as the selection of the importance of the station, highest degrees, or random choice. Their work need a lot of investigation about the best choice of subsets and the size to determine the best values for the number of levels which could lead sometimes to the bad choice.

Our proposed algorithm guarantees the optimality in computing the shortest path using the Dijkstra algorithm on a large graph. It consists of two phases: the pre-processing phase and the computation phase. The main idea is to divide the massive graph into smaller sub-graphs by using hMetis that guarantee the number of nodes in each partitioning is equal and minimizes the number of crossing edges. We build the two-level hierarchical graph that contains the border nodes and the source and the target nodes. Then apply the shortest path algorithm on the top-level graph.

We run our algorithm using a grid data with different number of nodes and compared it with the Dijkstra's algorithm. We also compared our results with the HEPV (Hierarchal Encoded Path View) and FEVP (Flat Encoded Path View) presented by Jing et al. (1996) that use different approaches of partitioning.

**IMPROVED DIJKSTRA ALGORITHM IMPLEMENTATION IN LARGE GRAPH (h-Dijkstra)**

Here, we address the acceleration algorithm (h-Dijkstra) for finding the shortest path of a weighted massive graph. Our approach consists of pre-processing the original (large) graph by partitioning it into sub-graphs using hMetis. Then we build a two-level hierarchical graph.

**Graph partitioning:** We used the hMetis software package that was developed for partitioning the VLSI circuits by Karypis and Kumar (1998) and Karypis et al. (1997) to partition the original graph. Graph partitioning is NP-complete and consequently the VLSI design automation community has explored several approaches for practical graph partitioning. In a traditional graph, each edge connects two vertices. hMetis permits to specify a parameter K, which represents the desired number of fragments. It partitions the graph into K fragments such that each fragment contains approximately the same number of vertices, while minimizing the number of edges that cross the fragments. Figure 1 illustrates the graph that has been partitioned. There are other methods to partition the graph that depends on spatial proximity.

**Compute the shortest path by using h-Dijkstra:** Our algorithm contains two steps: The first step is the pre-processing step that uses hMetis. Then we build a two-level hierarchical graph that contains the source and the target nodes. The hMetis algorithm partitions the vertices of a graph into fragments. The edges of the graph are of two types: local and border. A local edge is one whose vertices belong to the same fragment, whereas a border edge is one both vertices of which belong to different fragments. Two fragments are said to be adjacent if they have edges with endpoints belonging to different fragments. The endpoints of the border edges are border nodes other nodes are local nodes. The pre-processing step considers specifying the type of each node in the fragment.

We build the two-level hierarchical graph from the flat graph by inserting a border node at the half point of each border edge. Figure 2 illustrates these concepts. Two fragments are said to be adjacent if they have at least one
common border node. For example, P1 and P2 are adjacent, but P1 and P3 are not.

We moved up the border nodes to the top-level, where they form the vertices of the top-level graph Fig. 3. Edges added to the top-level graph so if the shortest path distance between a pair of border nodes in the original flat graph is some quantity Z, then there is a shortest path of length Z at the top-level graph. The algorithm of building the hierarchal graph is presented on Algorithm 1.

**Algorithm 1: Building the hierarchal graph**

1. `Generate Hierarchical Graph(G)`
2. For each fragment F in fragment list do
   - For each border node e F
     - sPath=compute the shortest path by using the edges in the flat graph

Having created a hierarchal graph, we are ready to solve the point-to-point shortest path computation. We compute the shortest path between the source and the target by using the top-level graph. An edge is added from the source to the border nodes adjacent to the fragment that contains the source node. The length of the edge is the length of the shortest path between the source and the border nodes in the flat graph. Similarly, an edge is added between the border nodes adjacent to the fragment that contains the target and the target node. The shortest path between the source and target is presented in Fig. 4. The algorithm is presented in Algorithm 2.

**Algorithm 2: Shortest Path Algorithm**

1. Given source and target nodes
2. Compute shortest path from source to target

Fig. 4: Illustration of computing the shortest path in the top-level graph. Vertices A and K denote the source and the target nodes, respectively. Edges (A,T) (A,S) (A,R) (Q,A) (X,K) (W,K) (U,K) and (K,V) are added to the top-level graph. The lengths of these edges are the lengths of the shortest path between the vertices in the original graph. The shortest path from A to K computed by using the Dijkstra's algorithm is A-S-W-K.
Table 1: Results on Grid with different partitioning

<table>
<thead>
<tr>
<th>Rows×Columns</th>
<th>Dijkstra</th>
<th>h-Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>40×40</td>
<td>0.23</td>
<td>0.031</td>
</tr>
<tr>
<td>100×100</td>
<td>3.80</td>
<td>0.098</td>
</tr>
<tr>
<td>150×150</td>
<td>13.20</td>
<td>4.945</td>
</tr>
<tr>
<td>200×200</td>
<td>56.10</td>
<td>15.319</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#/I/O</th>
<th>4 Partitions</th>
<th>#/I/O</th>
<th>8 Partitions</th>
<th>#/I/O</th>
<th>16 Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td></td>
<td>Time (sec)</td>
<td></td>
<td>Time (sec)</td>
</tr>
<tr>
<td></td>
<td>236</td>
<td></td>
<td>497</td>
<td></td>
<td>987</td>
</tr>
<tr>
<td></td>
<td>744</td>
<td></td>
<td>1474</td>
<td></td>
<td>2234</td>
</tr>
<tr>
<td></td>
<td>1552</td>
<td></td>
<td>3189</td>
<td></td>
<td>4135</td>
</tr>
<tr>
<td></td>
<td>2812</td>
<td></td>
<td>4008</td>
<td></td>
<td>5210</td>
</tr>
</tbody>
</table>

#/I/O = No. of input/output operations

Fig. 5: Computing the shortest path by using h-Dijkstra vs HEPV and FPEV

Algorithm 2: Computing the shortest path by using h-Dijkstra

H Dijkstra(source, destination)
- Add edge from the source to the border node belong to source fragment: Length of the edge = Shortest path (source, border nodes)
- Add edge from the border node belong to the target fragment to the target node: Length of the edge = Shortest path (border nodes, target)
- $\text{Spa}\text{th} = \text{Shortest path(HEP, source, destination)} \cap \text{HEP Hierarchical Graph}$

IMPLEMENTATION AND EXPERIMENTAL RESULTS

Our algorithm was developed in Visual C++® 6.0 Enterprise Edition in (Malik, 2006) and (Deitel and Deitel, 2005) were implemented on an Intel® Pentium® M with speed 1.73 GHz processor and 504 MB RAM, running on the Microsoft Windows XP Professional Service Pack 2. In all experiments, the results contain the following:
- Running time
- No. of I/O operations
- No. of partitioning

We ran our experiments on 40×40, 100×100, 150×150 and 200×200 directed weighted grids graph (this models the road networks structure in urban areas). Table 1 shows these results. We make the following observations about the results of our experiments:
- Table 1 show that our h-Dijkstra are significantly superior to the Dijkstra under either measure the run time and number of I/O operations. The Dijkstra takes longer time due to the large number of I/O operations for the massive graph that doesn’t fit in the memory
- The number of I/O operations increase as we increase the number of partitions but still less than the number of I/O of Dijkstra

Figure 5 compared h-Dijkstra with the HEPV (Hierarchical Encoded Path View) and FPEV (Flat Encoded Path View) presented (Jing et al., 1996) that use different approaches of partitioning the massive graph. We found that superiority of h-Dijkstra approach over the other alternative solutions.

CONCLUSION

The contribution of this study is to show that when we partitioned the large graph into fragments by using hMetis where the number of nodes in each fragment is equal and reduces the number of the crossing edges and computed point-to-point shortest path computation at the top level, h-Dijkstra has several advantages over Dijkstra, HEPV, and FPEV. The experimental results show that we enhance the running time and reduce the number of I/O operations. Our future work includes the deployment of the h-Dijkstra on a sparse graph and introduces a new parallel version of h-Dijkstra.

REFERENCES


