# Least Cost Path Discovery over graphs defined for large volumes of data satisfying node and link constraints

Shom C. Abraham
Bachelor of Engineering
Manipal Institute of technology
Manipal-576104, Karnataka

## ABSTRACT
Computing shortest paths in graphs is one of the most fundamental and well-studied problems in combinatorial optimization. Numerous real-world applications have stimulated research investigations in this area. Several applications include large graphs involving thousands of nodes, which we cannot assume to be fully loaded into memory. The problem is of much interest, when the nodes and edges have several constraints to be satisfied apart from being large, in the computation of shortest path. Conventional Dijkstra's algorithm does not serve the purpose. There has not been much research done in this area, although some papers investigate the problem of large graphs.

The problem of finding an efficient point-to-point shortest path algorithm for graphs of larger sizes, satisfying node as well as link constraints is solved using two optimization strategies. First, we implement bi-directional Dijkstra's algorithm with priority queue implementation using the heuristic value, in the path finding. The bi-directional strategy reduces the search space. Second, we introduce index of the graph table to preserve the local shortest segments, and exploit the table to further improve the performance. The final experimental results illustrates that this novel approach with the optimization strategies achieves high scalability and performance.

## General Terms
Graph Theory, Algorithms

## Keywords
Bidirectional Dijkstra's algorithm; Point-to-point shortest path algorithm satisfying node and link constraints; Combinatorial Optimization

## 1. INTRODUCTION
In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. A variation of the shortest path problem evolves out of the following example which forms the base research of this paper.

A network has nodes and links. Packets are to be transported from the origin to the destination through the route incurring minimum cost. However, not all existing paths are useful as it might violate node or edge constraints or both. A typical example of a node constraint is 'load'. It signifies that a node cannot bear the load greater than a given value. Similarly, an example of link constraint is 'congestion' which means that a particular route cannot be used for travel as congestion is present.

Thus, the requirement is to design and implement an efficient point-to-point shortest path algorithm satisfying node and link constraints taking into consideration the fact that graph can be large with thousands of nodes and millions of links.

## 2. BACKGROUND
### 2.1 Preprocessing
An offline stage where we pre compute all shortest paths using well known algorithm and store it in a distance matrix. Preprocessing may take hours for large graphs but further query on this matrix should only take milliseconds. This is not feasible for dynamic graphs.

### 2.2 Min Heaps
A min heap is a left complete binary tree which satisfies the property key (parent) ≤ key (child) for all the nodes. Due to this property, the node with the lowest key will always be present at the root of the tree. Hence, extraction of minimum key node is an O (1) operation.

In order to maintain the heap property after the extraction of root node, the non-leaf nodes would have to undergo percolate down operation (as shown below). This operation is an O (n) operation.

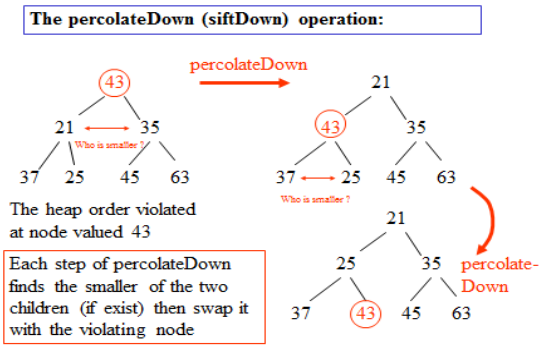


**Fig 1 : percolateDown operation in a heap**

### 2.3 Dijkstra's algorithm
It is a solution to the single-source shortest path problem in graph theory. Works on both directed and undirected graphs. However, all edges must have nonnegative weights. Input is a weighted graph G= {E, V} and source vertex vεV, such that all edge weights are nonnegative. Output is the lengths of shortest paths (or the shortest paths themselves) from a given source vertex vεV to all other vertices.

dist[s] ←0 (distance to source vertex is zero)
for allvεV–{s}
do dist[v] ←∞ (set all other distances to infinity)
S←ϕ (S, the set of visited vertices is initially empty)
Q←V (Q, the queue initially contains all vertices)
while Q ≠ ϕ (while the queue is not empty)

do u ← mindist(Q,dist)(select the element with the min. dist)
S←Sυ{u} (add u to list of visited vertices)
for all v ε neighbors[u]
do if dist[v] > dist[u] + w(u, v) (if new shortest path found)
then d[v] ←d[u] + w(u, v) (set new value of shortest path)

return dist

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of O (|V|^2).

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of O (|E| log |V|).

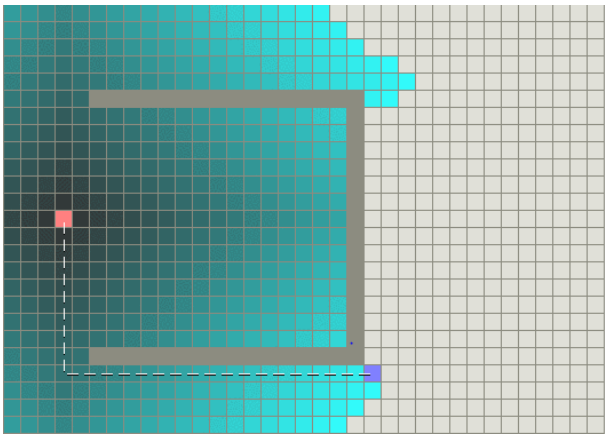

**Fig 2: Explored vertices in Dijkstra's algorithm**

## 2.4 A* Algorithm

1. Create a set S that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a cost value (heuristic + distance) to all vertices in the input graph. Initialize all distance values as INFINITE. Assign cost value as 0 for the source vertex so that it is picked first. This is the open set O.
3. While lowest rank in S is not the GOAL
   a. Pick a vertex u from O that has minimum cost value.
   b. Include u to S. (if u is already there update the value).
   c. Update cost value of all adjacent vertices of u.

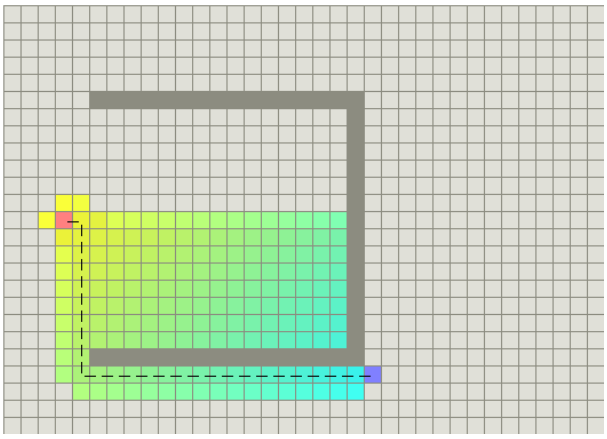

**Fig 3: Explored vertices in A* algorithm**

## 2.5 Comparison

Fig 2 and Fig 3 shows the explored vertices of Dijkstra's and A* algorithm respectively. Since the graph in consideration is huge, we would like the explored vertices to be as minimal as possible. So we use a modified version of A* algorithm for our problem.

# 3. PROPOSED ALGORITHM

WEIGHT (U, V)

{

return (actual weight (U,V) + Heuristic value)

}

NEW_DIJ (P, Q) {

P←ϕ (P, the set of visited vertices is initially empty)

u ← mindist(Q,dist)(select the element with the min. dist)
P←Pυ{u} (add u to list of visited vertices)

if u doesnot satisy node constraint RETURN
for all v ε neighbors[u]

if v doesnot satisy node constraint CONTINUE

if u-v link doesnot satisfy link constraint CONTINUE
do if dist[v] > dist[u] + WEIGHT(u, v)
then dist[v] ←dist[u] + WEIGHT(u, v)

return dist

}

MAIN_ALGO

{

1. Fill vertices in min-heap S sorted by distance from s (source).

   dist[s] ←0 (distance to source vertex is zero)
   for allvεV−{s}
   do dist[v] ←∞ (set all other distances to infinity)

2. Similarly, fill vertices in min-heap T sorted by distance from t (target).

3. Let S' and T' be two sets.

4. REPEAT

{

NEW_DIJ (S', S)

NEW_DIJ (T', T)

} UNTIL (∃v AND vεS' AND v εT')

5. FOR ALL xεS'

FOR ALL yεT'

L= dist[x] in S + WEIGHT(x, y) + dist[y] in T.

If L<(dist[v] in S + dist[v] in T) return

}

The above algorithm works well for graphs of larger sizes when we need to compute shortest path satisfying node and link constraints. Due to the large size of the graph the indices are maintained for fast retrieval of node details from the database. An index can be considered to be a copy of a database table reduced to certain fields. The data is stored in sorted form in this copy. This sorting permits fast access to the records of the table (for example using a binary search). Not all of the fields of the table are contained in the index. The index also contains a pointer from the index entry to the corresponding table entry to permit all the field contents to be read.

Another approach to deal with the larger size of the graph is to run the shortest path algorithm (NEW_DIJ) from both the directions (i.e. source and target). We maintain two min-heaps, S and T for the two directions sorted by distances from source and target respectively. The nodes extracted from the min-heaps during the run of algorithm are stored in sets S' and T' respectively. We stop this process until a common vertex v is found in both sets. The shortest path from s to t does not necessarily run through the vertex v. Hence we apply step 5 of the algorithm to search for the shortest path.

The NEW_DIJ function extracts the root node of the min-heap and stores it in a set. If the node doesn't satisfy the constraint we return from the function. Else we look for the neighbors of the extracted node. If they satisfy node and link constraints, we update their distance if the sum of distance of extracted node and weight of the link is less than the present distance.

Since the use of heuristic can reduce the explored vertices the WEIGHT function is modified to include the heuristic value.

## 4. IMPLEMENTATION AND RESULTS

An example of node constraint can be that the port Jakarta should allow the transportation of packets less than 101.2 million tonnes.

An example of link constraint is that the link between Chennai and Jakarta does not contain congestion.

For testing the algorithm a few nodes are taken on the Google maps and links between them are marked with red lines as shown below.

**Fig 4: Initial Configuration**

Following snapshot shows the shortest path shown in blue when the user clicks the source and destination node. The constraints are not provided.

***Fig 5: Shortest path calculation* without constraints**

Following figure shows the results of the application when the source and target node are chosen and node and link constraints are specified. The shortest path between New York and Tokyo is to be determined. The node constraint is given as 'Load' with value '100' and link constraint is specified as 'Congestion' with value Boolean 'N'. Now the algorithm computes the shortest path from New York to Tokyo such that only those ports who allow load of 100 million tonnes and only those routes where congestion is not present appear in the result.

**Fig 6: Shortest path calculation with constraints**

## 5. CONCLUSION

This paper proposes an efficient point to point shortest path algorithm for large graphs involving various node and link constraints. . The problem requirements have been met using modified version of Dijkstra's algorithm with satisfactory results. The implemented algorithm satisfies node and link constraints as required per the problem statement. The speed

factor is taken into account by using heap for the shortest path algorithm implementation. Experimental results ascertains that the proposed technique outperform commonly used Dijkstra's algorithm with adjacency matrix implementation qualitatively and quantitatively.

## 6. REFERENCES

[1] Ahuja, R. K., Magnanti, T. L. and Orlin, J. B. (1993), Network Flows: Theory, Algorithms and Applications, Prentice Hall, Englewood Cliffs, NJ .

[2] Bellman, R. (1958), On a Routing Problem, Quart. Appl. Math. 16, 87-90.

[3] Cai, X., Klocks, T. and Wong, C.K. (1997), Time-Varying Shortest Path Problems with Constraints, Networks, 29 , 141-149

[4] Cooke, K. L. and Halsey, E. (1966), The shortest route through a network with time-dependent internodal transit times, Journal of Mathematical Analysis and Applications, 14, 493-498.

[5] Dijkstra, E. W. (1959), A Note on Two Problems in Connexion with Graphs, Numerishe Mathematic 1 , 269-271

[6] Dreyfus, S. E. (1969), An Appraisal of Some Shortest-Path Algorithms, Operations Research, 17, 395-412

[7] Floyd, R.W. (1962), Algorithm 97: shortest path. Comm. ACM 5345.

[8] Klein, P.N. and Subramanian, S. (1997), A Randomized Parallel Algorithm for Single-Source Shortest Paths, Journal of Algorithms, Vol. 25, No. 2, pp. 205-220

[9] Henzinger, M. R., P. Klein, P., Rao, S. and Sairam Subramanian (1997), Faster Shortest-Path Algorithms for Planar Graphs, Journal of Computer and System Science 55, 3-23.

[10] Cherkassky, B. V., Goldberg, A. V. and Radzik, T. (1996), Shortest path algorithms: Theory and experimental evaluation, Mathematical Programming 73, 129-174.