Networking routing optimization algorithms, implementation and analysis.

Project Report

CPSC 629 – Analysis of Algorithms Fall 2014

1 Abstract:

Network optimization has been an important area in the current research in computer science and computer engineering. In this course project, I implemented a network routing protocol using the data structures and algorithms I have learnt in the class. This provided me with an opportunity to translate my theoretical understanding into a real-world practical computer program. Translating algorithmic ideas at a "higher level" of abstraction into real implementations in a particular programming language is not at all always trivial. The implementations forced me to work on more details of the algorithms, which sometimes led to a much better understanding.

2 Algorithms:

2.1 Maximum capacity problem

In graph algorithms, the maximum capacity path problem, or the widest path problem, also known as the bottleneck shortest path problem or, is the problem of finding a path between two designated vertices in a weighted graph, maximizing the weight of the minimum-weight edge in the path.

2.2 Dijkstra's Algorithm

The Dijkstra's algorithm is proposed for the shortest path problem. In this project it is modified for finding the maximum capacity and maximum capacity path. The pseudo code can be found as follows.

```
function Dijkstra(Graph, source):
2      for each vertex v in Graph:                                  // Initializations
3          width[v] := -infinity  ;                                 // Unknown width function from
4                                                                   // source to v
5          previous[v] := undefined ;                               // Previous node in optimal path
6      end for                                                      // from source
7
8      width[source] := infinity ;                                  // Width from source to source
9      Q := the set of all nodes in Graph ;                         // All nodes in the graph are
10                                                                  // unoptimized – thus are in Q
11      while Q is not empty:                                       // The main loop
12          u := vertex in Q with largest width in width[] ;        // Source node in first case
13          remove u from Q ;
14          if width[u] = -infinity:
15              break ;                                             // all remaining vertices are
16          end if                                                  // inaccessible from source
17
18          for each neighbor v of u:                               // where v has not yet been
19                                                                  // removed from Q.
20              alt := max(width[v], min(width[u], width_between(u, v))) ;
21              if alt > width[v]:                                  // Relax (u,v,a)
22                  width[v] := alt ;
23                  previous[v] := u ;
24                  decrease-key v in Q;                            // Reorder v in the Queue
25              end if
26          end for
27      end while
```

```
28      return width;
29  endfunction
```

2.2.1 Using array data structure

The algo can be implemented using arrays for maintaining the priority and data structure, in that case finding minimum would take $O(N)$.Insertion and deletion will take $O(1)$. Overall the algo takes time of $O(n^2)$

2.2.2 Using heap structures

The algo can be implemented using self-balancing max heap structure for maintaining the priority and data structure, in that case extracting maximum would take $O(1)$.Insertion and deletion will take $O(\log N)$. Overall the algo takes time of $O(n\ logn)$

2.3 Kruskal's Maximum Spanning tree algorithm

This algorithm is used for constructing the maximum spanning tree. The algorithm is simple in its form. Initially sorting all edges in decreasing order and maximum spanning tree is constructed by taking maximum weight edge and excluding it if it makes cycle with already formed maximum spanning tree.

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) ordered by weight(u, v), decreasing: // Such that maximum weight is taken always
5    if FIND-SET(u) ≠ FIND-SET(v):
6       A = A ∪ {(u, v)}
7       UNION(u, v)
8 return A
```

This algorithm is proved to provide the maximum capacity path between any two vertices given (source and destination).

2.3.1 Optimization

In the implemented algorithm an optimization is possible my maintaining count of number of edges added. Since the number of edges in the maximum spanning tree is V-1, this can be applied in the kruskal code and as the count of edges in MST reaches V-1 the loop can be exited. This gives optimization of the code, instead of running for all edges we can run for limited vertices wherever possible. This would not change worst case complexity but it will definitely improve the average time taken. This optimization is implemented in the code.

2.3.2 Depth First search

Given that maximum spanning tree gives the path from source to destination, the task reduces to traversing in that path and finding the bottleneck capacity in the path. The DFS algorithm is used for this purpose to traverse to each node starting from source until we find the destination node and finding the bottleneck capacity during the traversal.

Pseudo code for DFS

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5            v ← S.pop()
6            if v is not labeled as discovered:
```

```
7                label v as discovered
8                for all edges from v to w in G.adjacentEdges(v) do
9                    S.push(w)
procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
.
```

During DFS once the destination is reached the path while returning to source vertex in recursion is the place the maximum capacity is noted and the path is recorded for theoutput.

2.3.3 Disjoint Set data structure

This data structure is used for maintaining the tree property and to find if an edge would make a cycle for the current spanning tree.

```
function MakeSet(x)
     x.parent := x
     x.rank   := 0
 function Union(x, y)
     xRoot := Find(x)
     yRoot := Find(y)
     if xRoot == yRoot
         return
     // x and y are not already in same set. Merge them.
     if xRoot.rank < yRoot.rank
         xRoot.parent := yRoot
     else if xRoot.rank > yRoot.rank
         yRoot.parent := xRoot
     else
         yRoot.parent := xRoot
         xRoot.rank := xRoot.rank + 1
```

Union by rank and path compression during the finding the root are used for the optimization. This reduces the algorithm complexity from $O(E\ logV)\ to\ O(E\ log^{*}V)$

3 Graph Generation

The subroutines are written for generating two kinds of “random” graphs of 5000 vertices.Positive weights are assigned randomly for the edges.

In the first graph G1 **sparse**, every vertex has **degree exactly 6**.
In the second graph G2 dense, every vertex has degree about 20% of other the vertices i.e., degree about 1000.

3.1.1 Implementation

Adjacency list is used for this purpose. An array of linked lists to represent the list. Using STL vector and list structures for maintaining the data structure.

- Graph is initialized to all vertices named 0 to 5000, No edges is added yet, so it only represents vertices.
- In a loop, for every vertex the outgoing edges are added until the degree constraint is reached
    - While outgoing edge is added, list in the destination is updated
    - Check not to add edge from source vertex to source vertex
    - Check not to add edge to a destination where already degree reached
    - Check not to add edge whose edge is already been added.
- To maintain the connectivity in the graph(to make sure that all vertices are connected in the graph)
    - Initially the graph is generated for with exactly the degree of 2, to help in connecting all nodes. This is similar to DFS algorithm where an unexplored node is explored and added an edge. Explored edge already has two edges so the other unexplored edge is considered for new edge to be connected
    - The above connectivity makes sure that graph randomness is preserved at the same time graph is connected.
- Graph generation is optimized in few areas. In general random number generation creates trouble in converging the generation faster at many constraints in hand. For this purpose list of vertices is used to find out to generate random number only in those list. This is not adding any extra complexity and improves convergence and in fact reduces the time.

4 Further Improvements

- Optimizing the data structures of adjacency list using STL can be improved using simple arrays.
- For denser graphs the adjacency list traversal and algorithm running takes more time this can be improved by maintaining adjacency matrix instead of adjacency list
- Graph generation can be improved with much more optimization by removing redundancies in the random generation.
- Graph generation can be improved using DFS, to generate maze life structures, instead of plain loop implementations. This can improve the algorithm to complexity of DFS with some degree. This is more relevant to sparse graph generation. For denser graphs we may have to work around it.

5 Results

By running a single instance of application, results are generated for

1. 5 Dense graphs
2. 5 Sparse graphs
3. 5 pairs of source and destination for each graph
4. 3 Algorithms

The below tables the time taken for the 3 algorithms and 5 graphs for each pair

For Sparse graphs

Graph no	Algorithm	Pair-1	Pair- 2	Pair- 3	Pair- 4	Pair- 5	Average time (ms)
1	Dijkstra's Without Heap	0.000068	0.001030	0.000152	0.000370	0.000124	0.348800
	Dijkstra's With Heap	0.000023	0.000175	0.000066	0.000163	0.000065	0.098400
	Kruskal	0.001190	0.001109	0.001018	0.001092	0.001011	1.084000
2	Dijkstra's Without Heap	0.000351	0.000089	0.000440	0.000626	0.001041	0.509400
	Dijkstra's With Heap	0.000191	0.000016	0.000068	0.000276	0.000479	0.206000
	Kruskal	0.001054	0.000826	0.000842	0.000911	0.000951	0.916800
3	Dijkstra's Without Heap	0.000901	0.000110	0.000671	0.000992	0.000415	0.617800
	Dijkstra's With Heap	0.000404	0.000042	0.000258	0.000477	0.000178	0.271800
	Kruskal	0.000825	0.000789	0.000831	0.000809	0.000762	0.803200
4	Dijkstra's Without Heap	0.000831	0.000104	0.000135	0.000451	0.000423	0.388800
	Dijkstra's With Heap	0.000412	0.000037	0.000056	0.000202	0.000124	0.166200
	Kruskal	0.000763	0.000678	0.000708	0.000708	0.000696	0.710600
5	Dijkstra's Without Heap	0.000739	0.000799	0.000808	0.000093	0.000147	0.517200
	Dijkstra's With Heap	0.000368	0.000384	0.000337	0.000019	0.000045	0.230600
	Kruskal	0.000681	0.000725	0.000709	0.000648	0.000659	0.684400

The Bar diagrams are placed in the average values summary for understanding times taken by algorithms in visualized manner.

Graph no	Algorithm	Pair-1	Pair- 2	Pair- 3	Pair- 4	Pair- 5	Average time (ms)
1	Dijkstra's Without Heap	0.009004	0.000689	0.008993	0.002738	0.012217	6.728200
	Dijkstra's With Heap	0.008825	0.000557	0.003430	0.002330	0.012106	5.449600
	Kruskal	0.035275	0.032995	0.031961	0.032006	0.032258	32.899000
2	Dijkstra's Without Heap	0.011354	0.000986	0.003298	0.006761	0.009924	6.464600
	Dijkstra's With Heap	0.011188	0.000766	0.003513	0.003403	0.012169	6.207800
	Kruskal	0.032776	0.033696	0.032907	0.033138	0.033114	33.126200
3	Dijkstra's Without Heap	0.009116	0.001897	0.012998	0.006059	0.008142	7.642400
	Dijkstra's With Heap	0.011771	0.004045	0.012587	0.006119	0.008008	8.506000
	Kruskal	0.032506	0.032266	0.032016	0.032143	0.032011	32.188400
4	Dijkstra's Without Heap	0.001936	0.012166	0.013461	0.009816	0.011693	9.814400
	Dijkstra's With Heap	0.005829	0.011759	0.012913	0.009530	0.011020	10.210200
	Kruskal	0.034921	0.032170	0.031918	0.031989	0.031995	32.598600
5	Dijkstra's Without Heap	0.003176	0.009874	0.007297	0.009085	0.011603	8.207000
	Dijkstra's With Heap	0.006302	0.009787	0.007268	0.007772	0.011285	8.482800
	Kruskal	0.032638	0.032776	0.032819	0.032743	0.033249	32.845000

Total Summary time are shown as below, this is obtained by averaging many test pairs and many sparse and dense graphs.

Algo	Sparse Graph	Dense Graph
Dijkstra's Without Heap	0.4764	7.77132
Dijkstra's using Heap	0.1946	7.77128
Kruskal	0.8398	32.73144

6 Observations and reasoning:

1. Dijkstra's without heap always takes higher time than with heap in case most of the times, where as in case of sparse graphs the difference is significant and dense graphs the difference is less. For a graph of V vertices and E edges the algorithm with simple array structure takes complexity of

$$O(|E| + |V|^2) = O(|V|^2)$$

Whereas with heap structure or some simpler priority queue will take complexity of

$$\Theta((|E| + |V|) \log |V|)$$

So in case of sparse graphs $E \ll V$ and in case of dense graphs $E \sim V^2$ Though the optimization of $V \log V$ happens for sparse and dense graphs.The difference in dense graphs $O(E)$ becomes dominant over the other term- $O(V^2) or\ V\ logV$, and it will cover up the time difference, and for smaller graphs difference between $O(V^2)\ and\ V\ logV$ is very clear.

2. The Kruskal maximum spanning tree algorithm time taken is varied a lot across dense and sparse graphs. Relative to the dijkstra's algorithm as well it takes very high amount of time.
This can be explained as following
    a. Sorting of edges takes, using heap sort $E\ O(\log E)$
    b. The main disjoint sets data structure and maximum spanning tree building takes $E \log V$.
    c. The DFS to find the final path takes $O(V + E)$

For all the above case as the graph becomes denser edged approach $E \to V^2$
So the time taken increases dominated by the edges E factor and sorting those edge takes huge time more than any other module in the algorithm.
The Kruskal algorithm takes more time than dijkstra's algorithm because of all the steps mentioned above.

7 Running and testing code

1. For running the code, the exe/out can be run directly in the terminal. For simplicity of the project no command line arguments are not taken.
2. For running the algorithm makefile made for maintaining the compiler optimizations and generating and running the results faster.

Command*: make clean all*
./routing.out

3. Testing is done recursively for multiple test vectors. *srand(time(NULL))* is used for seeding differently every time for random number generation. I am printing the seed value in the starting of the log. This is to help in debugging for a certain stream/seed of test vector. Helps in debugging and regenerating the random numbers with exact seed.
4. The code is well tested for all the cases. The output prints the path from the destination vertex from source vertex, for all the three algorithms. This can be compared against the other algorithms and check if the capacity obtained is minimum bottleneck capacity in the path.
5. I have profiled/recording time using time.h library utilities. If the prints are enabled in the graph. It would take disturb the readings of timings. Specific care has been taken while noting down the reading for this purpose.

SAMPLE OUTPUT:

```
/*****************************************************************/
/******************                GRAPH 0                **********/
/*****************************************************************/
TOTAL number of edges =          1500
For Graph generation     0.013556 seconds.
=========        CASE 0  =========
Finding max capacity path          from 621-------to--------243          printed in reverse order
ALGORITHM : Dijkstra's without Heap
243, 124, 213, 393, 419, 154, 971, 621

ALGORITHM : Dijkstra's using Heap
243, 124, 213, 393, 419, 154, 971, 621

ALGORITHM : Kruskal Max Spanning Tree
243, 675, 495, 866, 540, 104, 988, 224, 425, 851, 635, 979, 73, 236, 974, 762, 332, 388, 487, 146,
978, 700, 28, 476, 330, 274, 273, 706, 176, 846, 51, 260, 452, 226, 500, 938, 391, 271, 693, 167,
716, 31, 246, 175, 188, 345, 26, 320, 902, 445, 154, 971, 621.
Summary :
Dijksra' W/out heap       took 6.8e-05 seconds.   &Found max capacity =  2178
Dijksra' WITH heap        took 2.3e-05 seconds.   &Found max capacity =  2178
Kruskal  MST algo         took 0.00119 seconds.  &Found max capacity =  2178
```