Combinatorial Optimization Report

Shortest Path algorithms comparison and Max Flow optimizations

Maurina Gabriele

March 5, 2021

Abstract

This document contains a report on the project created by Gabriele Maurina for the exam Combinatorial Optimization. The report is divided in two parts. The first is about the Shortest Path Problem. It contains an explanation of the problem, an explanation of the algorithms considered in the project and their implementations, and an evaluation of their performance with comparison between them. The algorithms implemented and compared in the first part are Dijkstra, Bellman-Ford and Moore. The results obtained show that Moore's algorithm is consistently the fastest, followed by a closed second Dijkstra's algorithm and Bellman-Ford's algorithm is consistently far behind. The second part is about the Max Flow problem. It contains an explanation of the problem, an explanation of the algorithms considered in the project and their implementations, and an evaluation of their performance with comparison between them. Particularly, this part of the project tried to use bidirectional extension of labels to improve the Ford-Fulkerson algorithm in its Breadth First Search (BFS) and Dijkstra variations. Overall the algorithms implemented and compared in the second part are Ford-Fulkerson with BFS for Shortest Augmenting Path, Ford-Fulkerson with bidirectional BFS for Shortest Augmenting Path, Ford-Fulkerson with Dijkstra for Maximum Capacity Augmenting Path and Ford-Fulkerson with bidirectional Dijkstra for Maximum Capacity Augmenting Path. The results obtained show that the bidirectional Dijkstra variation is consistently the fastest, with its monodirectional counterpart following closely. The BFS variation is consistently the slowest, whereas it bidirectional counterpart shows mixed results, being as fast as Dijkstra's variation when the average path length is short, i.e. with less than ten edges between source and destination, and being as slow as it monodirectional counterpart when the average path length is long, i.e. with hundreds of edges between source and destination.

Contents

1 Shortest Path

1.1 Problem

Given a directed graph $G = \{V, E\}$, where V is the set of vertices, E is the set of edges, $n = ||V|$ and $m = |E|$. Let $s \in V$ be the source and let $c : E \to \mathbb{R}$ be a cost function.

Shortest Path Problem: find all minimum cost paths from s to all vertices in V.

1.2 Implementation

There exist several algorithms to solve this problem. The algorithms implemented and tested are Dijkstra's algorithm, Bellman-Ford's algorithm and Moore's algorithm. The programming language used is C++ because it offers great control over it's data structures and has a rich standard library.

1.3 Graph representation

In order to execute any algorithm for the Shortest Path Problem it is necessary to have a data structure to represent a graph. For simplicity of implementation and to save space, an out-star representation is used. Three structures are used: Graph, Vertex and Edge. Where Graph contains an array of Vertex and an array of Edge, each Vertex contains an array of the out going Edge and each Edge contains a reference to the Vertex it comes from, a reference to the Vertex it goes to and its cost. For simplicity the cost is stored as an 8 bit unsigned integer, i.e. only positive costs are allowed. Both Vertex and Edge contain a field id, which is their position in the array in the graph. This representation saves space when the graph is sparse, compared to an adjacency matrix.

The structures used are:

```
struct Vertex{
        ui id;
        vector<Edge*> edges;
};

struct Edge{
        ui id;
        sui cost;
        Vertex* from;
        Vertex* to;
        Edge(ui _id,sui _cost,Vertex* _from,Vertex* _to);
};

struct Graph{
        vector<Vertex> vertices;
```

```
        vector<Edge> edges;
        Graph(ui N,f density,ui delta=0);
};
```

Where s is always the first Vertex.

The graph is generated randomly given size, density and delta, according to the following pseudocode:

```
create array of vertices of size size;
for each vertex v1:
    for each vertex v2:
        if delta = 0 or abs(v1.id-v2.id) <= delta:
            if rand_float(0,1) < density:
                create an edge with random cost between v1 and v2;
```

Delta represents the maximum difference in id between vertices. When delta is zero it is discarded. Delta is necessary if we want to create a certain kind of graph, i.e. one whose shortest path between first and last vertex has many edges. Density is the density of the graph, only when delta is zero.

1.4 Dijkstra's algorithm

Dijkstra's algorithm is implemented in its simplest form with an array, (no priority queue). Hence its execution time is $O(n^2)$. An array *cost* is used to store the distance from s to every vertex. An array *pred* is used to store the predecessor of each vertex in a path with minimum cost from s to itself. An array $flag$ is used to store whether a path of minimum cost to a vertex has been found.

The implementation is:

```
void dijkstra(const Graph& g,const ui s){
        const ui N = g.vertices.size();
        vector<ui> cost(N,INF);
        vector<ui> pred(N,s);
        vector<bool> flag(N,false);
        cost[s] = 0;
        for(ui i=0;i<N;i++){
                ui cmin = INF;
                ui idmin = 0;
                for(ui j=0;j<N;j++)
                if(!flag[j]&&cost[j]<cmin){
                        idmin = j;
                        cmin = cost[j];
                }
                flag[idmin] = true;
                for(auto e:g.vertices[idmin].edges)
                if(!flag[e->to->id])
```

```
                if(cost[idmin]+e->cost<cost[e->to->id]){
                        cost[e->to->id] = cost[idmin]+e->cost;
                        pred[e->to->id] = idmin;
                }
        }
}
```

1.5 Bellman-Ford's algorithm

Bellman-Ford's algorithm loops $|V|$ times over the entire Edge array. Hence its execution time is $O(nm)$. An array *cost* is used to store the distance from s to every vertex. An array *pred* is used to store the predecessor of each vertex in a path with minimum cost from s to itself.

The implementation is:

```
void bellman_ford(const Graph& g,const ui s){
        const ui N = g.vertices.size();
        vector<ui> cost(N,INF);
        vector<ui> pred(N,s);
        cost[s] = 0;
        for(ui i=1;i<N;i++)
        for(auto e:g.edges)
        if(cost[e.from->id]+e.cost<cost[e.to->id]){
                cost[e.to->id] = cost[e.from->id]+e.cost;
                pred[e.to->id] = e.from->id;
        }
}
```

1.6 Moore's algorithm

Moore's algorithm is implemented with a FIFO queue. Its execution time is $O(nm)$. The FIFO queue is implemented combinig a queue and a set for $O(1)$ lookup. Its structure is:

```
struct MooreFifoQueue{
        queue<ui> q;
        unordered_set<ui> us;
        void push(const ui value);
        ui pop();
        bool empty();
        bool contains(const ui value);
};
```

An array *cost* is used to store the distance from s to every vertex. An array *pred* is used to store the predecessor of each vertex in a path with minimum cost from s to itself.

The implementation of Moore's algorithm is:

```
void moore(const Graph& g,const ui s){
        const ui N = g.vertices.size();
        vector<ui> cost(N,INF);
        vector<ui> pred(N,s);
        cost[s] = 0;
        MooreFifoQueue q;
        q.push(s);
        while(!q.empty()){
                ui i = q.pop();
                for(auto e:g.vertices[i].edges)
                if(cost[i]+e->cost<cost[e->to->id]){
                        cost[e->to->id] = cost[i]+e->cost;
                        pred[e->to->id] = i;
                        if(!q.contains(e->to->id))
                        q.push(e->to->id);
                }
        }
}
```

1.7 Results

The three previously discussed algorithms are evaluated in order to test their efficiency w.r.t execution time. All experiments are conducted on a laptop running Fedora Workstation 32 with an Intel® Core™ i7-2670QM CPU @ 2.20GHz×8 and 6 GB of ram. The evaluation is performed on large graphs with up to 4thousands vertices and 1.6 million edges. Different levels of density and delta are used. Table 1 summarizes the results obtained.

Figures 1,2 and 3 shows the execution time of the algorithms with graphs of delta=0 and increasing size and density. Moore's algorithm is fastest, followed by Dijkstra's algorithm. Bellman-Ford's algorithm is by far the slowest.

Figures 4,5 and 6 shows the execution time of the algorithms with graphs of density=0.8 and increasing size and delta. Moore's algorithm is fastest, followed by Dijkstra's algorithm. Bellman-Ford's algorithm is by far the slowest.

The results obtained are consitent across all kinds of graphs tested.

2 Max Flow

2.1 Problem

Given a directed graph $G = \{V, E\}$, where V is the set of vertices, E is the set of edges, $n = ||V|$ and $m = |E|$. Let $s \in V$ be the source, let $t \in V$ be the sink and let $u : E \to \mathbb{R}_+$ be a capacity function. Let z be the flow, i.e. a quantity that can traverse each edge from a vertex to another starting in s and finishing in t such that it never exceeds the capacity of an edge, it is not negative and the flow entering a vertex is equal to the flow leaving it, unless the vertex is s or t.

delta	density	$\|V\|$	$\|E\|$	dijkstra	bellman_ford	moore
0	0.01	1000	9934	36	207	2
0	0.01	2000	39924	140	1656	7
0	0.01	3000	90086	317	5630	17
0	0.01	4000	160679	562	13479	27
0	0.05	1000	50087	37	1028	4
0	0.05	2000	200945	148	8395	22
0	0.05	3000	449742	336	28859	60
0	0.05	4000	798221	582	68454	91
0	0.1	1000	100254	39	2066	11
0	0.1	2000	399959	155	17008	38
0	0.1	3000	900183	338	57717	66
0	0.1	4000	1600469	595	137566	102
10	0.8	1000	15848	31	323	10
10	0.8	2000	31954	125	1306	55
10	0.8	3000	47744	279	2923	139
10	0.8	4000	63882	496	5223	288
50	0.8	1000	77943	35	1611	6
50	0.8	2000	157870	133	6470	34
50	0.8	3000	237856	293	14795	69
50	0.8	4000	318184	514	26621	139
100	0.8	1000	151970	41	3116	13
100	0.8	2000	311629	151	13078	42
100	0.8	3000	471814	329	29715	71
100	0.8	4000	631519	573	53171	144

Table 1: Shortest Path Problem algorithms comparison

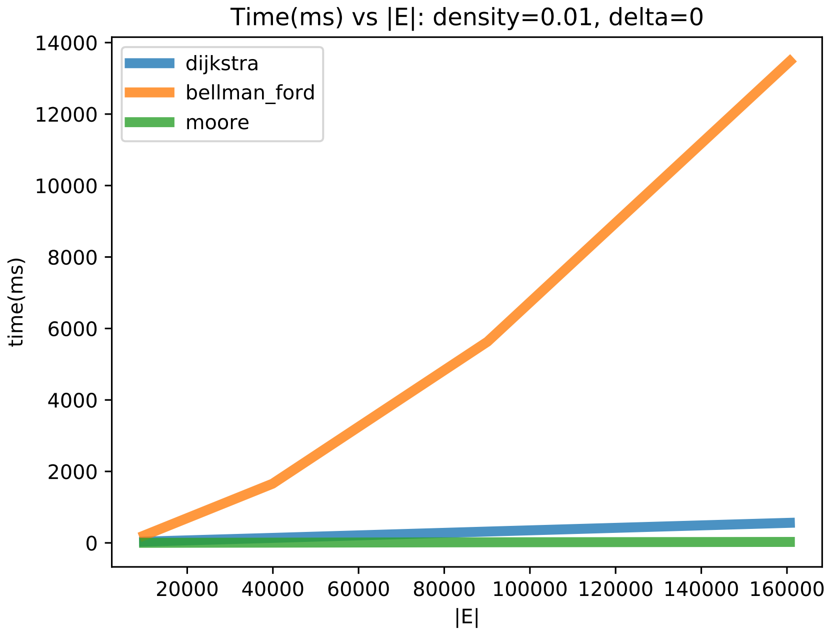

Figure 1: density=0.01

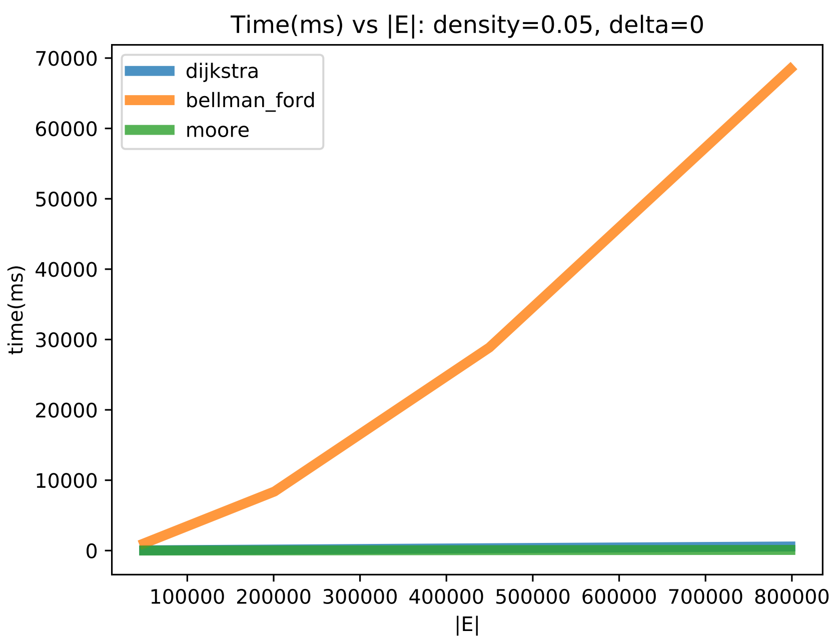

Figure 2: density=0.05

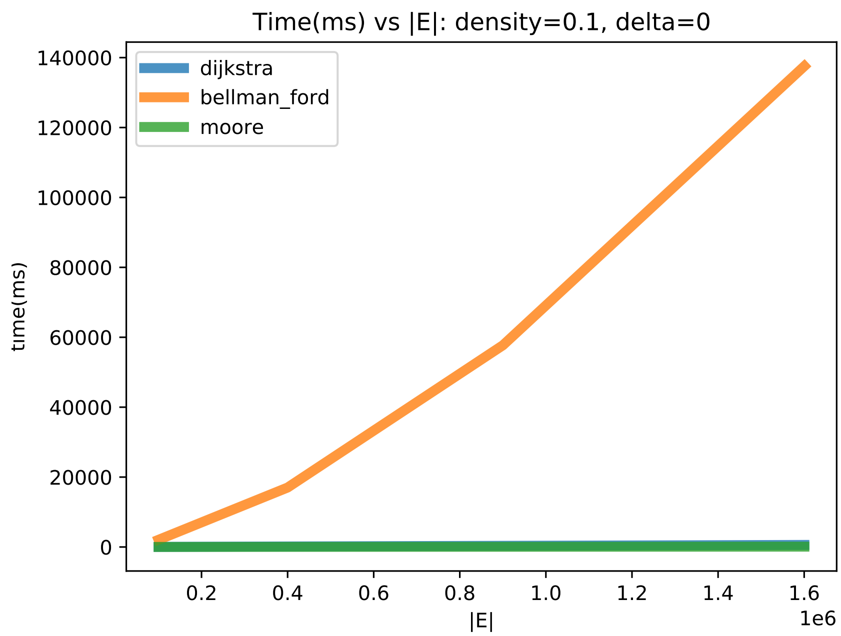

Figure 3: density=0.1

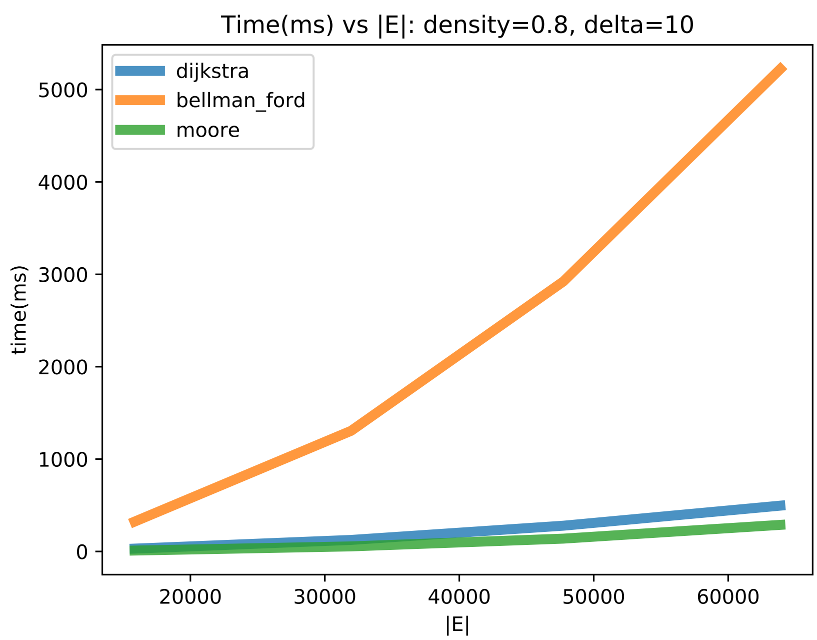

Figure 4: delta=10

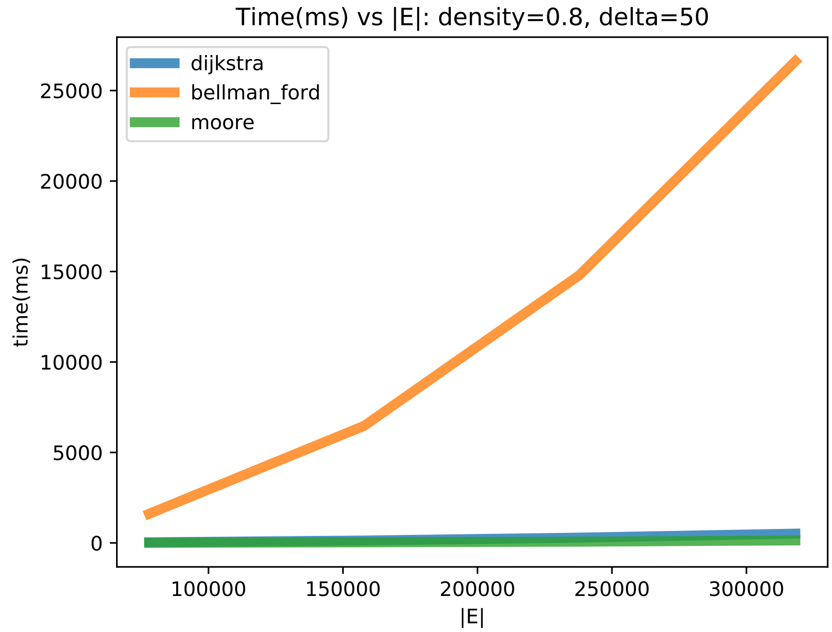

Figure 5: delta=50

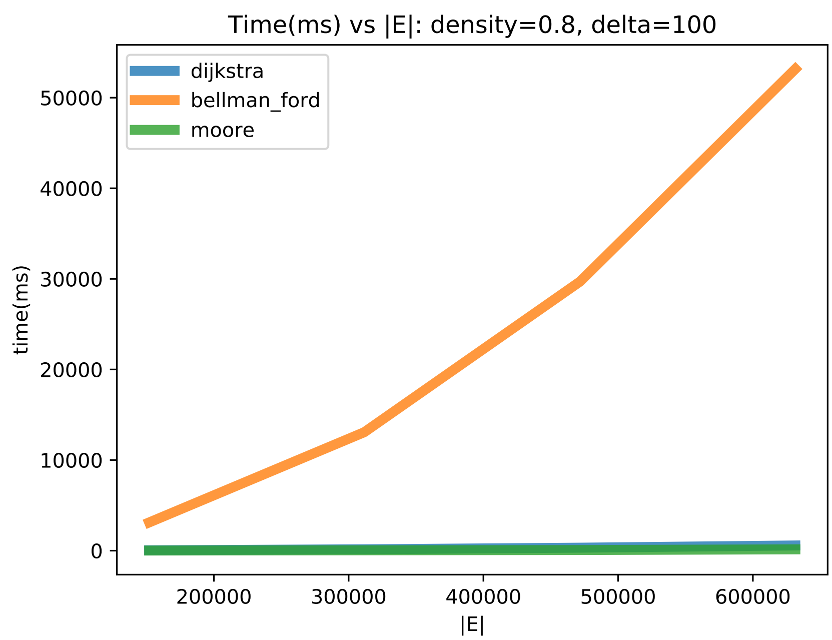

Figure 6: delta=100

delta	density	$\|V\|$	$\|E\|$	bfs	bfs_bi	dijkstra	dijkstra_bi
0	0.01	1000	9904	17	1	14	2
0	0.01	2000	39951	192	8	46	10
0	0.01	3000	89629	679	23	149	20
0	0.01	4000	160320	2279	52	862	53
0	0.05	1000	50371	471	25	97	23
0	0.05	2000	199272	5341	172	1769	98
0	0.05	3000	450017	22678	425	3101	248
0	0.05	4000	799392	60093	923	18611	629
0	0.1	1000	99926	2038	92	797	71
0	0.1	2000	400840	25488	443	6703	309
0	0.1	3000	897988	101777	1076	14265	747
0	0.1	4000	1598115	273573	2202	66067	1711
10	0.8	1000	15894	421	424	26	13
10	0.8	2000	32085	1023	1070	17	20
10	0.8	3000	47694	1128	1130	25	30
10	0.8	4000	63956	2136	2183	165	87
50	0.8	1000	78045	3331	3234	70	42
50	0.8	2000	157841	11794	11701	63	69
50	0.8	3000	238410	30368	29734	471	166
50	0.8	4000	318378	42911	44485	1396	237
100	0.8	1000	151990	9369	7343	578	125
100	0.8	2000	311963	36788	34126	2021	174
100	0.8	3000	472190	70093	64804	3023	274
100	0.8	4000	632013	122896	115424	4036	359

Table 2: Max Flow Problem algorithms comparison

Furthermore the flow leaving s is equal to the flow entering t. The non-negative continuous variable x_{ij} indicates the flow on the edge $(i,j) \in E$.

Max Flow Problem: maximize z such that:

$$\sum_{i \in V:(i,j) \in E} x_{ij} - \sum_{i \in V:(j,i) \in E} x_{ji} = \begin{cases} -z & \text{if } i = s \\ 0 & \text{if } i \neq s,t \\ z & \text{if } i = t \end{cases}$$

2.2 Implementation

2.3 Results

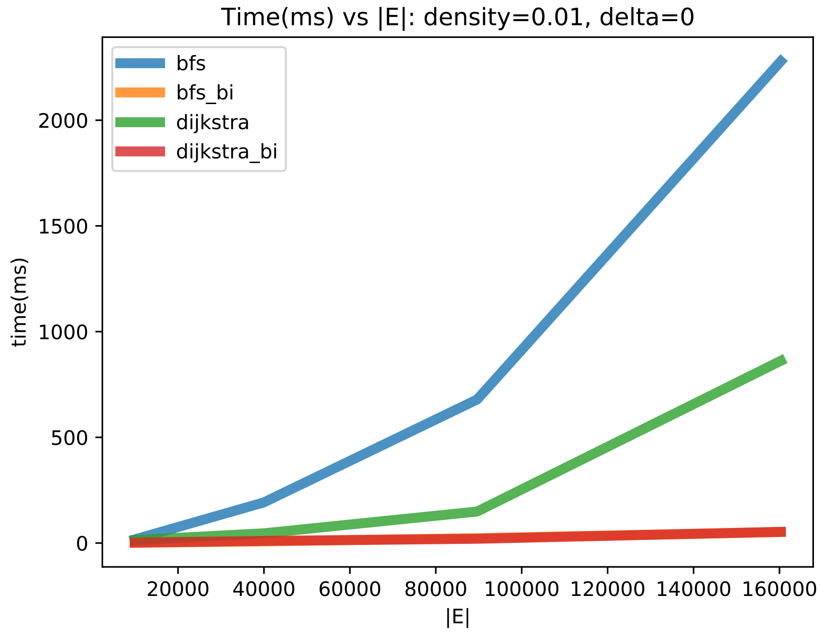

Figure 7: density=0.01

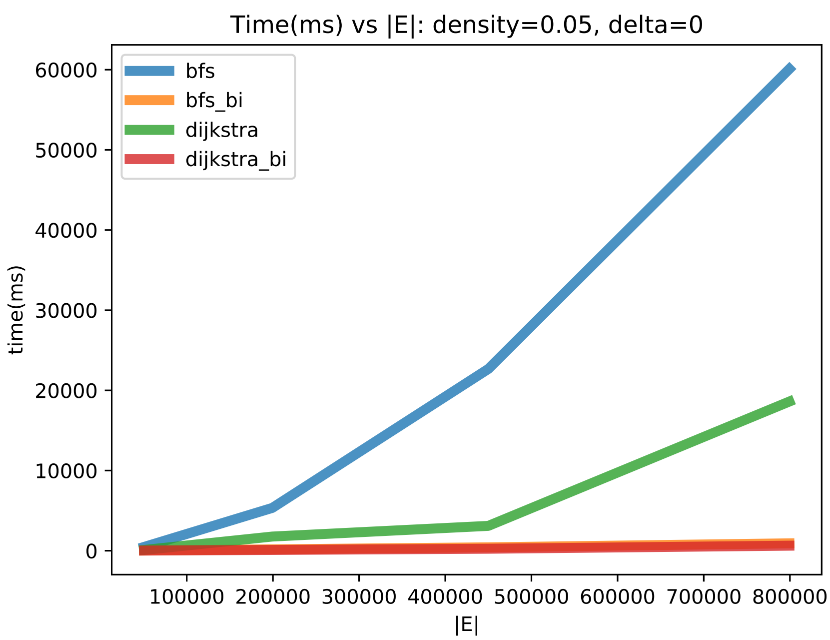

Figure 8: density=0.05

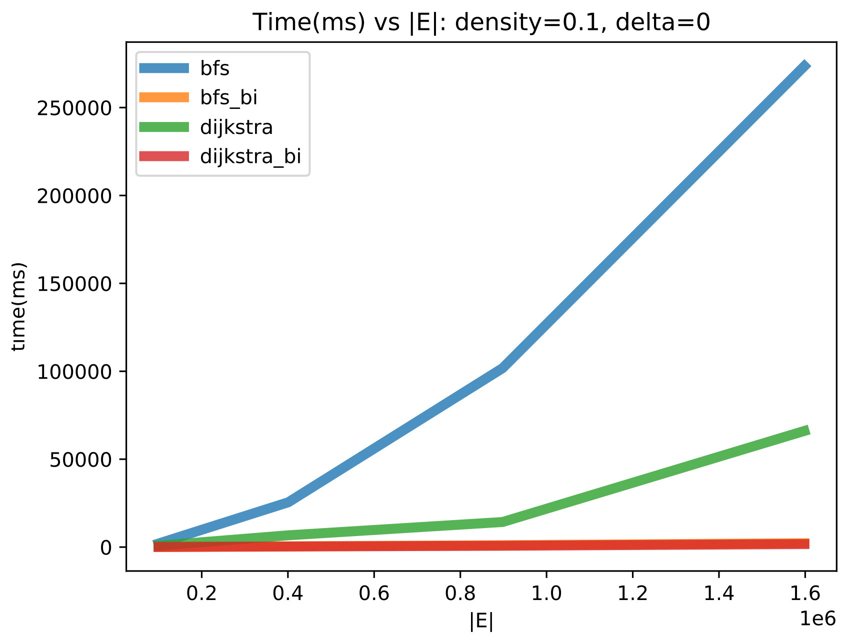

Figure 9: density=0.1

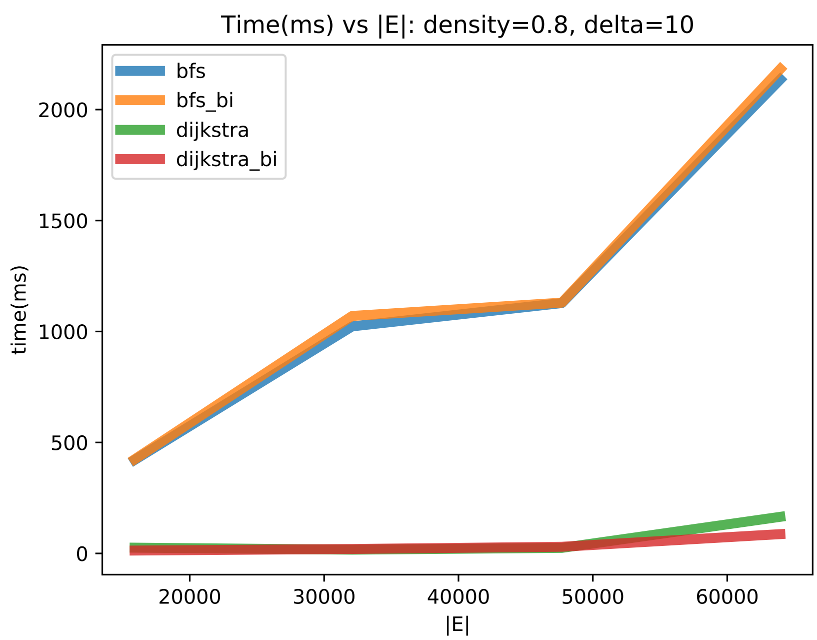

Figure 10: delta=10

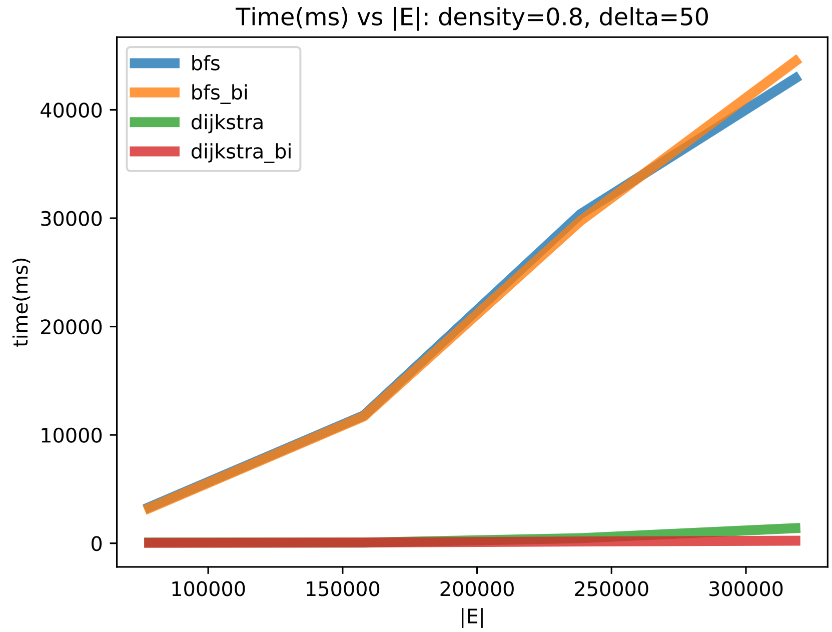

Figure 11: delta=50

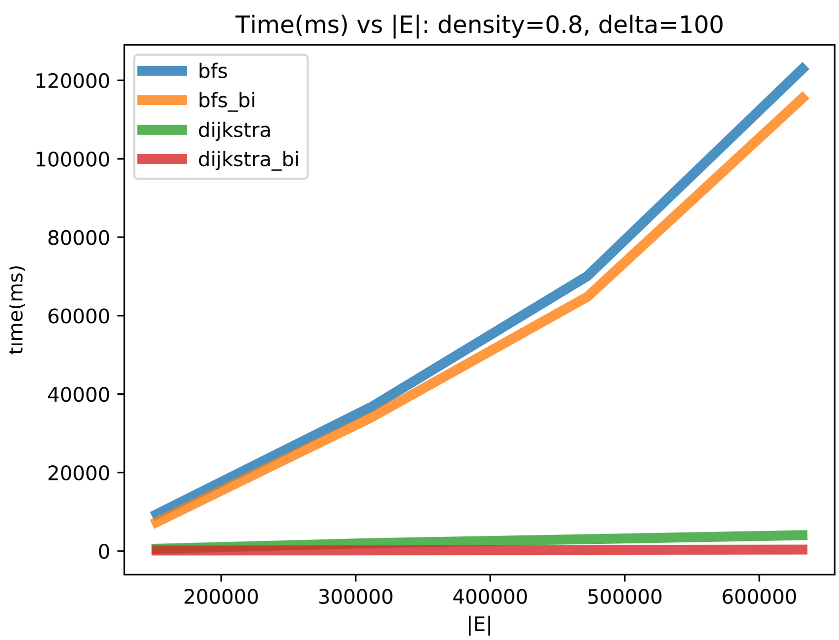

Figure 12: delta=100