CSCE 629 Analysis of Algorithms, Fall 2022

Course Project Report

Cheng-Yun Cheng

Introduction

Graph algorithms, including the shortest path, maximum bandwidth path, and maximum flow, have been critical areas in computer science. The purpose of this project is to implement Dijkstra's algorithm and Kruskal's algorithm for the maximum bandwidth path problem. Furthermore, we analyze the performance of each algorithm and the effect of distinct types of data structures. The report has four sections, a review of Dijkstra's and Kruskal's algorithm, implementation details, performance analysis, and conclusions and future improvement.

Algorithms

1. Dijkstra's Algorithm for Maximum Bandwidth Path

Dijkstra's algorithm is designed to solve the single-source shortest-path problem on a weighted graph in which all edge weights are non-negative. Given a graph *G*, a source vertex *s*, and a target vertex *t*, Dijkstra's algorithm will grow a tree *T* by repeatedly adding a new vertex into the tree until all connected vertices are in *T*. In each iteration, the algorithm will choose a vertex that is not in *T* and its bandwidth value is maximum. And then, all its neighbors would be updated. The pseudocode is shown as follows.

```
Dijkstra − BW(G, s, t)
1: for (v = 0; v ≤ n; i + +) do
2:     state[v] = unseen; b − width[v] = 0; dad[v] = 0;
3: end for
4: state[s] = in − tree; b − width[s] = ∞; dad[s] = −1;
5: for each edge [s, w] do
6:     state[w] = fringer; b − width[w] = bw(s, w); dad[w] = s;
7: end for
8: while there are friners do
9:     pick fringer v with the largest b-width value
10:    status[v] = in − tree
11:    for each edge [v,w] do
12:        if status[w] == unseen then
13:            status[w] = fringer; dad[w] = v;
14:            b − width[w] = minb − width[v], bw(v, w)
15:        else if status[w] == fringer then
16:            if (b − width[w] < min(b − width[v], bw(v, w)) then
17:                dad[w] = v; b − width[w] = min(b − width[v], bw(v, w);
18:            end if
19:        end if
20:    end for
21: end while
```

Figure 1. The pseudocode of Dijkstra's algorithm for maximum bandwidth path

2. Kruskal's Algorithm for Maximum Bandwidth Path

Although Kruskal's algorithm is designed to construct a maximum spanning tree from a graph, a tree path is also a maximum bandwidth path in the original graph. Given a graph *G*, Kruskal's algorithm will repeatedly pick the edge with the maximum weight, and then add two ends of the edge into the tree. After constructing the maximum spanning tree, DFS or BFS could be used to find a path from vertex *s* to vertex *t* in the tree. The pseudocode is shown as follows.

```
Kruskal − BW(G, s, t)
1: sort the edges of G in non-increasing order by their edge weights
2: T = the vertices of G (without any edges);
3: for v = 0; v < n; v++ do
4:     MakeSet(v);
5: end for
6: for (i = 1; i ≤ m; i++) do
7:     let e_i = [u_i, v_i]
8:     r_u = Find(u_i); r_v = Find(v_i)
9:     if r_u ≠ r_v then
10:        T = T + e_i; Union(r_u, r_v);
11:    end if
12: end for
13: Use DFS or BFS to construct the path P in T from s to t;
```

Figure 2. The pseudocode of Kruskal's algorithm for maximum bandwidth path

Implementation Details

1. Random Graph Generation

To analyze the performance of algorithms, generating graphs randomly is essential. In this project, we use a sparse graph in which the average vertex degree is 6 and a dense graph in which the average vertex degree is 1000, and both graphs contain 5000 vertices. The graph is implemented by an adjacency list. Each node represents an edge, and it is comprised of a head vertex, a tail vertex, and a value of the bandwidth. Subsequently, a random graph generation function is written which takes the average vertex degree as the input. To make sure that the graph is connected, we construct a cycle to connect all vertices in the beginning. After that, for each pair of vertices, a random variable and a threshold will decide whether there is an edge between them or not. Also, another random variable is used to generate a positive bandwidth for each edge. An example graph generated by our function is shown below.

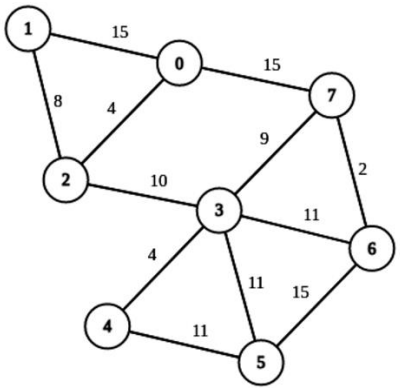

Figure 3. A graph generated by our random graph generator with 10 vertices and 3 average edge degree

2. Heap Structure

In this project, we implemented a maximum heap for Dijkstra's algorithm to find the fringer. The data structure is modified from the normal maximum heap. We used three arrays, *heap*, *pos*, and *bdwidth*, to implement it. The *heap* array is used to store vertex names (an integer number), and the bandwidth value of a vertex is stored in *bdwidth* array. To easily delete a certain vertex from the heap, the *pos* array is used to record vertex's position in the heap.

3. Routing Algorithms

All these three algorithms take a graph *G*, a start vertex *s*, and a target vertex *t* as inputs and output a maximum bandwidth path from *s* to *t* and its bandwidth value.

a. Dijkstra's Algorithm without using a heap structure

Dijkstra's algorithm was implemented based on the pseudocode. We defined a *Vertex class*, which has three arrays as its data members. They are used to record the parent, bandwidth, and status for each vertex, respectively. A slight difference between our implementation and pseudocode is that the status of a vertex is either in-tree or unseen. This is because the initial bandwidth value of each vertex is 0 and the minimum weight of each edge is 1. Therefore, we can distinguish whether the status of a vertex is unseen or seen by its bandwidth value. In each iteration, we have to traverse all the vertices to find the fringer whose bandwidth value is maximum, because the vertex status is recorded by an array. Consequently, the total running time of this algorithm is in $O(n^2)$, where *n* is the number of vertices. The maximum bandwidth path of Figure 3 generated by Dijkstra's algorithm is shown below.

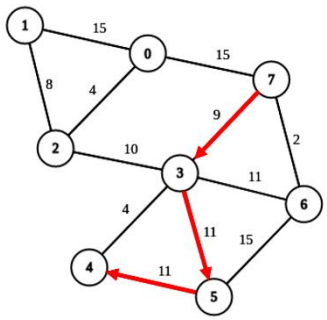

Figure 4. The result of Dijkstra's algorithm (input graph: Figure 3, s: 7, t: 4)

b. Dijkstra's Algorithm with using a heap structure

Most part of the algorithm is similar to the above one, but a maximum heap is added to manage fringers. Instead of traversing the whole array to find the vertex with the maximum bandwidth value, we can easily get this vertex in $O(1)$ through the heap and delete it in $O(\log n)$ based on the length of the heap. At the update step, there are two situations. First, if the status of the neighbor is unseen, we just update and insert it into the heap. This operation would take time $O(\log n)$. On the other hand, if the neighbor's status is fringer, remove it from the heap first. After updating, insert it back into the heap. Both the delete and insert operations would take time $O(\log n)$. The total time of this algorithm is $O(m \log n)$.

c. Kruskal's Algorithm with HeapSort

The algorithm was implemented based on the pseudocode with some modifications. First, heap sort is used to sort all the edges in non-increasing order by their edge weight. Second, we implemented *Make*, *Union*, and *Find* functions following the content presented in the class. The *Make* function is used to initialize the root and the rank for each vertex; The *Union* function merges two trees based on their rank; the *Find* function returns the root of a tree and it involves a compression process. After constructing the maximum spanning tree, DFS algorithm is used to find a path from *s* to *t*. The overall time complexity of the algorithm is $O(m \log n)$. The maximum spanning tree and the maximum bandwidth path of Figure 3 generated by Kruskal's algorithm is shown below.

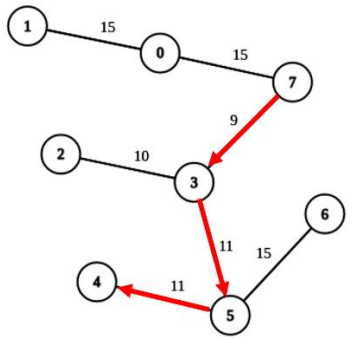

Figure 5. The result of Kruskal's algorithm (input graph: Figure 3, s: 7, t: 4)

Performance Analysis

1. Sparse Graph (number of vertices: 5000, average vertex degree: 6)

The result shows that the performance of Dijkstra's algorithm with a maximum heap is the best and Dijkstra's algorithm without a heap is the worst. The bad performance of Dijkstra without a heap meets expectations because it must traverse all vertices to find the maximum fringer. Therefore, the overall time complexity is $O(n^2)$. For the other two algorithms, their time complexity is both $O(m \log n)$, but the running time of Dijkstra with a heap is much less than Kruskal's algorithm. The reason is that there are lots of operations when constructing the maximum spanning tree, such as Find and Union, and DFS would take extra $O(n + m)$ time. Therefore, the running time of Kruskal's algorithm is greater.

Graph		Running Time(*ms*)		
		Dijkstra w/o heap	Dijkstra w/ heap	Kruskal w/ Heap Sort
Sparse Graph 1	Pair 1	34151	1208	9901
	Pair 2	36065	1117	9585
	Pair 3	32873	1171	9656
	Pair 4	31456	1145	9535
	Pair 5	25463	1117	10125
Sparse Graph 2	Pair 1	33389	1236	9923
	Pair 2	34271	1098	9486
	Pair 3	25056	841	9373
	Pair 4	30880	1141	9533
	Pair 5	35859	1192	9611
Sparse Graph 3	Pair 1	34134	1164	9852
	Pair 2	32883	1100	9452
	Pair 3	29924	1123	9551
	Pair 4	22442	840	10067
	Pair 5	30719	1158	9745
Sparse Graph 4	Pair 1	35029	1189	9854
	Pair 2	31402	1075	9916
	Pair 3	24774	934	9843
	Pair 4	31432	1024	9863
	Pair 5	30059	1180	9429
Sparse Graph 5	Pair 1	31876	1144	10138
	Pair 2	29596	1108	10108
	Pair 3	29052	997	9652
	Pair 4	33268	1138	10488
	Pair 5	31348	1127	9804
Average Time(*ms*)		**31096.04**	**1102.68**	**9779.6**

Table1. The performance results of 3 algorithms on 5 sparse graphs

2. Dense Graph (number of vertices: 5000, average vertex degree: 1000)

For dense graphs, the performance of Dijkstra with a heap is still the best, but Kruskal's algorithm becomes the worst. The reason for the terrible performance of Kruskal's algorithm is the time complexity of heap sort is $O(m \log m)$. Therefore, when the number of edges becomes larger and larger, it would take lots of time to sort them. For the two types of Dijkstra's algorithms, although the one with a heap still performs better, their performance is closed. This result demonstrates that Dijkstra with a heap is not suitable for a large number of edges, because the *Insert* and *Delete* operation would be used tons of times. This also matches the theoretical result that the complexity time of Dijkstra with a heap is $O(m \log n)$.

Graph		Running Time(ms)		
		Dijkstra w/o heap	Dijkstra w/ heap	Kruskal w/ Heap Sort
Dense Graph 1	Pair 1	292338	269740	4279480
	Pair 2	283229	271571	4328762
	Pair 3	263533	269591	4357856
	Pair 4	301126	278843	4418782
	Pair 5	301380	275720	4265326
Dense Graph 2	Pair 1	295239	271202	4196796
	Pair 2	294801	272569	4184325
	Pair 3	288260	270478	4406765
	Pair 4	323109	280975	4614852
	Pair 5	307133	275130	4524455
Dense Graph 3	Pair 1	294054	276058	4213967
	Pair 2	289599	275608	4231501
	Pair 3	262110	267449	4697190
	Pair 4	297529	277083	4271695
	Pair 5	290179	266921	4365130
Dense Graph 4	Pair 1	285263	266435	4356652
	Pair 2	287460	267416	4180019
	Pair 3	267720	266658	4404865
	Pair 4	307372	313217	4676060
	Pair 5	328638	296828	4573851
Dense Graph 5	Pair 1	292838	274730	4397787
	Pair 2	296167	274965	4233495
	Pair 3	257816	283443	4382780
	Pair 4	285638	282182	4528082
	Pair 5	271458	285259	4463700
Average Time(ms)		**290559.56**	**276402.84**	**4382166.92**

Table2. The performance results of 3 algorithms on 5 dense graphs

Conclusions

In this project, we implemented three types of algorithms to solve the maximum bandwidth path problem. Furthermore, a performance analysis was done on sparse and dense graphs. The test results show that Dijkstra with a heap and Kruskal's algorithm perform well on sparse graphs. However, they are not suitable for dense graphs, especially Kruskal's algorithm.

To conclude, algorithms and data structures are crucial for computer science. We should take both of them into account when solving problems.

Future Improvements

1. For Kruskal's algorithm, I used a recursive DFS to find the path. However, when the number of vertices become larger, this function would consume lots of stack memory. Using BFS might be better.