# CSCE 629 Analysis of Algorithms
# Course Project Report

## Introduction

The purpose of this project is to implement a network routing protocol using certain data structures and algorithms explored throughout the course. The MAX-BANDWIDTH-PATH problem is considered an important network optimization problem and is the focus of this project. Finding a path that contains the maximum bandwidth between the source and destination can be achieved in multiple ways, but the performance difference can vary based on the type of structures used and how the algorithms are implemented. For this project, we used a graph network in order to find the optimal path two vertices. Additionally, we used two well-known algorithms to achieve finding a path: Dijkstra's and Kruskal's. However, they were each modified and tuned to either use a heap structure, heap sort for edges, or not use the heap structure at all. With these three different versions of implementations, we measured the performance on random weighted undirected graphs that mathematically represented a sparse or dense graph.

## Implementation Details

### *1. Random Graph Generation*

The first aspect of this network was to generate two types of graphs: sparse and dense. Each of the graphs contains 5000 vertices and are implemented using adjacency lists, where the key will represent the name of the vertex and the value contains a list of vertex nodes. In our implementation of the *Vertex class* we keep track of the vertex name and the respective bandwidth value of the edge that connects those two vertices together. The *Graph class* is able to create the sparse and dense graphs. The sparse graph must have a vertex degree of 6, which means that on average the number of edges of a vertex is 6 and they must be randomly assigned. The respective edge weight was also assigned a value from 1 to 1000 at random. On the other hand, the dense graph must ensure that each vertex is adjacent to about 20% of the other vertices, which means that the possibility of connecting a pair of vertices is based upon a vertex degree of 1000. The respective edge weight is also assigned a value from 1 to 1000 at random.

### *2. Heap Structure*

The maximum heap structure that will be used for Dijkstra's algorithm was implemented in the *Heap* class, supporting the functions of *Maximum, Insert,* and *Delete.* The implementation was closely based on the solution found from our assignment #2, but to further improve the performance of the *Delete* operation, we used a list to the store the index of the vertices in the heap as we described in class. This *position* list is updated each time there is an *Insert* or *Delete* operation, where if the index of the vertex changes in the heap, then it is updated in the *position* list as well. This allows us to avoid inefficiently searching

for a specific vertex in the heap considering we will know the exact position with this additional list.

## 3. *Routing Algorithms*

The input for each of these algorithms is graph *G*, a source vertex *s*, and a destination vertex *t*. Each of these algorithms will then return the maximum bandwidth path from *s* to *t*.

### a. *Dijkstra's Algorithm without Heap Structure*

Dijkstra's algorithm was implemented based on the same pseudo code that was provided in class. In general, three arrays are initialized to store the parent, bandwidth, and status of each node. The status value was implemented using the numerical value of 0 for when a node is in-tree, 1 for when a node is a fringe, and 2 for when a node is still unseen. Considering we are using an array to keep track of the fringes, we will need to iterate through the entire array in order to find the maximum fringe, causing the overall Dijkstra algorithm to run in $O(n^2)$. The implementation of Dijkstra's algorithm with the use of an array to find the maximum fringe is shown below:

```
def dijkstra(G, s, t):
    bw = [0]*V
    parent = [-1]*V
    status = [2]*V

    status[s] = 0
    bw[s] = sys.maxsize
    for v in G.findNeighbor(s):
        bw[v.dst] = v.bw
        parent[v.dst] = s
        status[v.dst] = 1

    while status[t] != 0:
        v = maxfringe(bw, status)
        status[v] = 0
        n = G.findNeighbor(v)
        for i in range(len(n)):
            w = n[i].dst
            if (status[w] == 2):
                bw[w] = min(bw[v], n[i].bw)
                parent[w] = v
                status[w] = 1
            elif (status[w] == 1 and bw[w] < min(bw[v], n[i].bw)):
                bw[w] = min(bw[v], n[i].bw)
                parent[w] = v
    return bw[t]
```

### *b. Dijkstra's Algorithm with Heap Structure*

Similarly to the above algorithm, we will use the same pseudo code to implement Dijkstra's algorithm, but with a slight modification on how we keep track of the fringes. Instead of the array we used above, we will use the maximum heap structure we previously implemented. This will require a few modifications to our original implementation of Dijkstra. In addition to the three initialized arrays, we will also need to initialize a heap *hp* created from our *Heap* class. In addition, we will insert all the nodes and respective bandwidths that are neighbors to *s* into *hp*. By using this heap, we are able to return the maximum fringe in *O(1)* time considering it is the first value of the heap. In doing so, we basically will use the *Maximum* function from the heap and the *Delete* function on the maximum node, which has been proven to take *O(log n)* based on the height of the tree. Lastly, our two conditions consist of an unseen status and a fringe status. If a neighbor node is still unseen, then we will insert the fringe into the heap. However, if the second condition is true, then we will delete this vertex from the heap, update the bandwidth and parent arrays and insert this new vertex back into the heap. The operations for deleting and inserting into a heap take *O(log n)* as mentioned previously and finding the maximum *O(1)*, which will result in overall complexity of this version of Dijkstra to be *O(m log n)*, where *m* represents the edges of the graph and *n* the number of vertices. The implementation of this modified version of Dijkstra's algorithm with the use of heap structure to find the maximum fringe is shown below:

```
def dijkstraHeap(G, s, t):
    hp = Heap()
    bw = [0]*V
    parent = [-1]*V
    status = [2]*V

    status[s] = 0
    bw[s] = sys.maxsize
    for v in G.findNeighbor(s):
        bw[v.dst] = v.bw
        parent[v.dst] = s
        status[v.dst] = 1
        hp.insert(v.dst, bw[v.dst])

    while status[t] != 0:
        v = hp.maximum()
        hp.delete(hp.maximum())
        status[v] = 0
        n = G.findNeighbor(v)
        for i in range(len(n)):
            w = n[i].dst
            if (status[w] == 2):
                bw[w] = min(bw[v], n[i].bw)
                parent[w] = v
                status[w] = 1
                hp.insert(w, bw[w])
            elif (status[w] == 1 and bw[w] < min(bw[v], n[i].bw)):
                hp.delete(w)
                bw[w] = min(bw[v], n[i].bw)
                parent[w] = v
                hp.insert(w,bw[w])
    return bw[t]
```

### *c. Kruskal's Algorithm with HeapSort*

Kruskal's algorithm was implemented using similar pseudo code presented in class with a few modifications in order to take advantage of the *heapSort* function. The general implementation of using Kruskal to find the maximum bandwidth path is to build a maximum spanning tree and then use the Depth First Search (DFS) algorithm to find a path between *s* and *t*. However, for our implementation we added a few modifications. We first create a list of all the edges of the graph and use *heapSort* to sort the edges in non-increasing order of their edge weights. Additionally, we implement the *MakeUnionFind* class in order to have the ability of using the operations *Make, Find,* and *Union* as was presented in the course. This class keeps track of the parent, size, and ranking for each vertex allowing us to use an iterative approach for the *Find* operation. These functions allow us to build a new graph that contains only the edges that pertain to the maximum spanning tree. This graph is then passed to the *path* function which uses *DFS* to find the path along *s* to *t.* The returned path will be the maximum bandwidth path from *s* to *t.* The final implementation of this modified version of Kruskal's algorithm with the use of heap sort is shown below:

```
def kruskal(G, s, t):
    edges = list()
    bw = [0]*V
    status = [2]*V
    heapSort(G, edges)
    muf = MakeUnionFind(V)
    newG = Graph()
    for i in range(len(edges)-1, -1, -1):
        e = edges[i]
        source = muf.Find(e.src)
        destiny = muf.Find(e.dst)
        if source != destiny:
            newG.addEdge(e.src, e.dst, e.bw)
            newG.addEdge(e.dst, e.src, e.bw)
            muf.Union(source, destiny)

    return path(newG, s, t)
```

## Evaluation Setup

The implementation of this network was created using *Python 3.7.0* and consisted of the following files:

- *graph.py*: Contains the class of *Graph, Vertex,* and *Edge*
- *krsukalAlgo.py:* Contains the implementation of Kruskal's algorithm with the use of functions such as *heapSort, DFS,* and *path.*
    - *MakeUnionFind.py:* Contains the functions of *Make, Set,* and *Union* which help the Kruskal implementation in path compression
- *dijkstraAlgo.py:* Contains the implementation of Dijkstra's algorithm with and without the use of the heap structure.
    - *heapStruct.py:* Contains the functions *Maximum, Insert,* and *Delete* in order to implement the maximum heap structure
- *main.py:* Returns the running time of all three routing algorithms based on a sparse or dense graph and a specified number of trials.

To test the performance of our algorithms, we generated five random sparse graphs and five random dense graphs. For each of these generated graphs, we picked five random pairs of source-destination pairs. We then ran all three algorithms on each of these pairs for the respective graph. We measured the running time of our algorithm by returning the number of seconds it takes to finish executing the specific algorithm, ignoring the time initialization of generating the graph itself.

## Performance Analysis

The results of our evaluation tests for each of these algorithms on sparse and dense graphs are shown in the tables below with their overall average running time.

| Sparse Graph Results | | | | | |
|---|---|---|---|---|---|
| | | | Running Time (s) | | |
| | Source | Destination | Dijkstra w/o Heap | Dijkstra w/ Heap | Kruskal w/ Heap Sort |
| **Sparse Graph 1** | 4084 | 2672 | 0.40640 | 0.04687 | 0.17188 |
| | 1582 | 3362 | 1.31253 | 0.09374 | 0.15622 |
| | 4138 | 2674 | 0.76584 | 0.01562 | 0.17188 |
| | 4118 | 1118 | 1.42201 | 0.09373 | 0.15624 |
| | 4191 | 1360 | 0.60944 | 0.06250 | 0.17186 |
| **Sparse Graph 2** | 1702 | 1249 | 0.34374 | 0.04685 | 0.18749 |
| | 3683 | 3816 | 1.40656 | 0.12497 | 0.17186 |
| | 4332 | 4448 | 1.68768 | 0.03124 | 0.18748 |
| | 1367 | 1829 | 0.18750 | 0.03125 | 0.18747 |
| | 1421 | 3457 | 1.58710 | 0.10938 | 0.18749 |
| **Sparse Graph 3** | 4684 | 1265 | 0.11395 | 0.10492 | 0.17590 |
| | 3258 | 3249 | 1.67908 | 0.15891 | 0.17288 |
| | 2773 | 4175 | 0.97344 | 0.09994 | 0.17491 |
| | 4725 | 685 | 0.57069 | 0.05898 | 0.17188 |
| | 3143 | 2743 | 0.73088 | 0.06398 | 0.17509 |
| **Sparse Graph 4** | 4528 | 4565 | 1.78315 | 0.06296 | 0.17988 |
| | 301 | 2955 | 1.09737 | 0.10795 | 0.17790 |
| | 1283 | 4081 | 0.76657 | 0.06098 | 0.17690 |
| | 4787 | 1270 | 0.11195 | 0.04696 | 0.19189 |
| | 2495 | 664 | 0.09595 | 0.02501 | 0.17247 |
| **Sparse Graph 5** | 1741 | 3486 | 1.85638 | 0.14249 | 0.29683 |
| | 2176 | 0 | 0.75566 | 0.10394 | 0.17090 |
| | 1689 | 2602 | 1.88405 | 0.14793 | 0.18219 |
| | 1476 | 1548 | 0.72669 | 0.07936 | 0.17592 |
| | 4587 | 2649 | 1.41512 | 0.11292 | 0.18289 |
| **Average** | | | **0.97159** | **0.08134** | **0.18113** |

| Dense Graph Results | | | | | |
|---|---|---|---|---|---|
| | | | Running Time (s) | | |
| | Source | Destination | Dijkstra w/o Heap | Dijkstra w/ Heap | Kruskal w/ Heap Sort |
| **Dense Graph 1** | 3481 | 4643 | 4.18804 | 2.20328 | 47.22265 |
| | 1596 | 4419 | 3.58098 | 0.16276 | 47.74224 |
| | 4745 | 2579 | 1.50017 | 0.89078 | 48.87095 |
| | 1150 | 3677 | 1.78939 | 0.64859 | 48.05643 |
| | 2752 | 4995 | 3.14093 | 0.35949 | 47.97892 |
| **Dense Graph 2** | 100 | 2938 | 2.48458 | 0.85945 | 45.69069 |
| | 4770 | 977 | 0.64858 | 1.43517 | 46.07365 |
| | 3130 | 4762 | 3.71886 | 1.75013 | 64.09112 |
| | 1121 | 2956 | 3.95339 | 2.06262 | 46.73823 |
| | 1525 | 3633 | 3.45135 | 2.17200 | 46.37837 |
| **Dense Graph 3** | 4727 | 564 | 1.75999 | 0.31226 | 47.82317 |
| | 4267 | 1461 | 3.09108 | 1.00492 | 48.09912 |
| | 902 | 3797 | 3.33222 | 1.44881 | 48.78855 |
| | 1824 | 3227 | 4.36550 | 2.33966 | 50.05552 |
| | 2563 | 973 | 0.09494 | 0.31682 | 47.97605 |
| **Dense Graph 4** | 3079 | 2951 | 4.18361 | 2.35367 | 50.17676 |
| | 1542 | 4360 | 4.30654 | 2.44862 | 53.00689 |
| | 3246 | 3931 | 4.28824 | 2.36563 | 49.77307 |
| | 4422 | 3512 | 3.20328 | 1.47348 | 52.60056 |
| | 3504 | 4400 | 3.48201 | 1.83595 | 51.57521 |
| **Dense Graph 5** | 1547 | 2016 | 3.30173 | 0.77428 | 51.30508 |
| | 2982 | 4519 | 4.34102 | 1.99986 | 48.88656 |
| | 4387 | 3348 | 3.35012 | 3.20753 | 51.85922 |
| | 3087 | 1678 | 1.76701 | 1.54510 | 48.80716 |
| | 1656 | 3985 | 3.19619 | 2.12477 | 47.75720 |
| **Average** | | | **3.06079** | **1.52383** | **49.49333** |

Based on the overall average running time, we can observe the following performance relation for sparse graphs:

*(Faster) Dijkstra with Heap > Kruskal > Dijkstra without Heap (Slower)*

The most time-consuming algorithm for sparse graphs is Dijkstra without the use of the heap structure, which as explained in the implementation, the overall complexity would be O(n2). This is due to having to iterate through the entire array of vertices every time it needs to find the maximum fringe. Therefore, it performs as expected. As was proven in class, Kruskal's algorithm with the use of Union-Find operations and Dijkstra's algorithm with the help of a heap structure would both take an overall complexity time of O(m log n). This is based on the notion that the Find operation would take O(log n) time and the Insert and Delete operations would also

take O(log n). However, one additional modification that was added to Kruskal's algorithm was the *HeapSort* function, which requires to sort through all the edges of the graph. In addition, after building the maximum spanning tree, the algorithm has to run a DFS algorithm to provide the path between the two vertices. Therefore, the performance difference would be due to the constants in the Big-O complexity. Although there could be a number of insertion and deletion operations, the amount of time it will take to sort through all edges in the graph and also perform DFS on all the edges in the new graph increase the constant value of overall time complexity, and therefore explain the reason it performs slower than Dijkstra's implementation with a heap.

Based on the overall average running time, we can observe the following performance relation for dense graphs:

*(Faster) Dijkstra with Heap > Dijkstra without Heap > Kruskal (Slower)*

Similar to the performance in sparse graphs, Dijkstra's algorithm with the use of the heap structure takes the least amount of time to return the maximum bandwidth path. This demonstrates the usefulness of this algorithm in both types of graphs and is able to hold the overall complexity time of O(m log n) despite the number of edges being greater than the number of vertices. On the other end of the spectrum, Kruskal's algorithm obtains the worst performance in dense graphs as it performs 16 times slower than Dijkstra, which is not a surprising observation. In our implementation, we had to use *HeapSort* to sort through the edges and therefore on a dense graph, this will be highly inefficient and dependent on the number of edges. In a dense graph, the number of edges will be greater than the number of vertices and the probability for two vertices to be connected to 20% of its neighbors is similar to having a vertex degree of 1000. As a result, the total number of edges would be at least over a million edges and having to sort through all these edges would be time consuming. This explains the terrible performance of Kruskal and a possible flaw in the implementation that could be improved.

## Conclusion and Future Improvements

As was noted in our performance analysis, the Kruskal's algorithm has the worst performance in dense graphs and *HeapSort* is the cause of most of the time consumption. This aspect of the implementation could be possibly improved with another type of sorting technique. One of the main issues is the dependency Kruskal has on the number of edges, meaning that the greater the number of edges, the slower this implementation runs due to sorting through all the edges of the graph. If another sorting technique could more quickly sort through the edges, then the implementation could have a better performance. However, the amount of improvement that could be accomplished is questionable, but even reducing it enough to be comparable to Dijkstra's various implementations would be beneficial.

Overall, we were able to successfully implement three different implementations to solve the network optimization issue of finding the maximum bandwidth path. In doing so, we were able to test the performance of each of these implementations on sparse and dense graphs. The results demonstrated that Dijkstra's implementation with a heap took the least amount of time to execute and therefore had the best performance for both sparse and dense graphs.