Laura Battle 2 December 2020

Project Report - COT 5405

After implementing Breadth-First Search, Depth-First Search, Dijkstra, Prim's for Minimum Spanning Tree, and Ford-Fulkerson/Edmonds-Karp (using BFS) for Minimum Cut between two vertices, below are my results. Each result had twenty random graphs generated for it, so the average in milliseconds and standard deviation for each is given below. Graphs follow on the final 7 pages.

Run Time (ms) based on increasing density for n = 10000 (up to 6 significant digits)

Edges	measure	BFS.exe	DFS.exe	Dijkstra.exe	Prims.exe	MinCut.exe
n	$average(ms)$	0.1877	0.8407	0.7273	0.2707	3.1843
	σ	0.00565	0.22985	0.06824	0.09041	0.09007
	$edges$	10021	10028	10021	10028	10016
$2n$	$average(ms)$	1.8558	1.1999	147.505	25.235	4491.68
	σ	0.46997	0.00793	17.7897	6.15615	1787.17
	$edges$	20075	20030	20075	20030	19984
$5n$	$average(ms)$	2.8469	2.11884	271.745	33.5859	13573.8
	σ	0.48761	0.40185	32.4739	8.29876	3189.28
	$edges$	50183	50208	50183	50208	50270
$10n$	$average(ms)$	3.45847	2.6862	291.095	32.0642	19650.8
	σ	0.29960	0.04625	25.5403	3.15837	3399.19
	$edges$	100285	100520	100285	100520	100212
$20n$	$average(ms)$	3.89665	3.8183	284.202	37.0099	32422.9
	σ	0.09950	0.07148	26.5423	0.65872	4490.99
	$edges$	200188	200398	200188	200398	200545
$50n$	$average(ms)$	7.4248	6.95015	352.252	54.2012	88282.6
	σ	0.30477	0.28450	33.3318	1.94030	9349.33
	$edges$	500275	500530	500275	500530	499367

*MinCut.exe returned that the two given vertices, 1 and 2, were not connected in all twenty trials.

Edges	measure	MinCut.exe
n*	$average$	0
	σ	0
	$edges$	10016
$2n$	$average$	9.45
	σ	3.84537
	$edges$	19984
$5n$	$average$	26.8
	σ	4.25008
	$edges$	50270
$10n$	$average$	44.7
	σ	8.02693
	$edges$	100212
$20n$	$average$	86.05
	σ	10.4301
	$edges$	200545
$50n$	$average$	265.95
	σ	18.1933
	$edges$	499367

In the graphs that follow you see a sharp uptick at the beginning of Dijkstra and Prim's which I assume has to do with how connected the graph is. Afterward, at different points, they both experience a slight downtick. Perhaps this shows a defect in the way that I implemented the priority queue which is discussed on the page discussing descriptions of utility functions, as I implemented the priority queue in essentially the same way

Description of Utility Functions and Data Structures

All algorithm files have utility functions called Print Answer and Dump. The Dump function operates if the algorithm is run in verbose mode. It outputs what each vertex looks like at the end of the algorithm. It was also used for development. Print Answer takes care of user output to either console or file.

DataStruc.cpp/h:

DataStruc( ) This function reads in an edge list that is in sorted order (sorted both by starting and finishing vertex) and builds a corresponding adjacency list data structure. The data structure is a vector of vectors, the list is populated with the struct "edge" which contains the second vertex and weight, both as int values. The starting vertex is the index of the first dimension of the vector. Vertices that have no edges pointing outward are empty vectors.

SortMe/QuickSortF/QuickSortS/PartitionF/PartitionS/Swap - these functions are for the Directed.cpp and Flow.cpp graph generators. They sort the list of edges before writing them to an output file. Since since quicksort is unstable, I had to write two different sorts, the first sorts the starting vertices, the second sorts the finishing vertex by being called on each starting vertex. It is a rather inefficient way of doing things, but I built Directed.cpp and Flow.cpp after everything else was built so I was limited in my choices.

Start Timer/End Timer: this starts the timer right before the official start of the algorithm and the end timer function stops the function and then appends the number of vertices, number of edges, and time in seconds to a file called dfs.time/bfs.time/etc.

Dijkstra.cpp:

RestoreMinHeap/ExtractMin/Parent are functions used for the minimum priority queue for Dijkstra. The queue is implemented as a standard deque template in C++. The queue's data type are pointers to vertices, so while the Graph[ ] data structure operates as it does in other algorithms, as a vector of vertices that correspond to their index number with Graph[0] being essentially empty, the deque of pointers can have a heap data structure. The vertices only get pushed onto the queue when their weights are changing. This is different from the algorithm in the book where all vertices get pushed onto the queue initially, but I think it works because anything that isn't going to be considered will be unreachable. While running it on examples, it returned correct values.
ExtractMin( ) pops first value off of the deque and returns an int that serves as this vertex's index value in the Graph[ ] data structure. RestoreMinHeap essentially inserts the vertex whose distance has just changed into the queue and then uses the Parent( ) function to restore the MinHeap property.

Prims.cpp

Uses the same function as above for Dijkstra. I think this works too since it runs on undirected graphs, anything that isn't put onto the queue will be unreachable.

Trace( ) uses the level attribute in the vertex and starts at vertex to print out the tree in a recursive manner.

MinCut.cpp

This has the function Ford -Fulkerson( ) which is the meat of the method and it has a function BFS ( ) which implements the Edmonds-Karp variation of the Ford - Fulkerson method.

Flow. cpp
Directed. cpp
Undirected. cpp

These programs build random Edge Lists. Flow and Undirected are not particularly efficient, but they do build lists compatible with the back end of the algorithms. As they aren't central to the purpose of the project, I am not going into details about them.

If I were going to do this project again, I would build a data structure on the back end that didn't depend on the edge list being in order so that these graph generators would be faster.
Instructions for compiling and running code are contained in the README file as well as on the next page.

README for COT 5405 Project by Laura Battle

Instructions for each program are given in detail at the top of each .cpp file.

QuickStart

Step 1: Type "make"

Step 2: Type "make clean"

Step 3: Generate random graphs

To generate a random edge list for a directed graph, an undirected graph, and a directed graph with no reverse edges in the files "directed.in", "undirected.in" and "flow.in" with p values of 0.1, 20 nodes, and max edge weights of 100, type:

Directed.exe
Undirected.exe
Flow.exe

Step 4 (optional): Generate random graphs with different parameters

Alter the p-value to 0.15, nodes to 50, and max weight to 5000 for each graph while giving filenames for the outputs:

Directed.exe 0.15 50 5000 directedTest.in
Undirected.exe 0.15 50 5000 undirectedTest.in
Flow.exe 0.15 50 5000 flowTest.in

Please note the error checking on this is not robust, so enter these correctly
Programs will accept no arguments or all arguments, nothing in between

Step 5 (optional): There will be a file called "AdjListDS.exe" you can run this to see what the data structure of the Adjacency list looks like. This is the data structure used for the algorithms. To use it type:

AdjListDS.exe file.in (optional third argument)

the second argument is the file you want read into the data structure. If you type in any argument after the file, it will also create the transpose of the graph and dump it. This additional functionality was built for something that ultimately was not done, but I haven't deleted it in case I want to use it in the future.

Below are instructions for BFS, Prims, Dijkstra, DFS and MinCut

---

DFS and Prims - UNDIRECTED GRAPHS

These programs are built to work with undirected graphs with edge weights. The edge weights are ignored in DFS and used in Prims. Each program has two modes, regular and verbose.

REGULAR MODE

To run a program in regular mode type the name of the executable with the file containing the graph like so:

DFS.exe file.in
Prims.exe file.in

For Prims, this means the root is set to its default value of 1. If you want to use a different root for Prims, enter it as a third argument. If you enter a root value larger than the largest node value, it will default back to one. An example for how to do this would be:

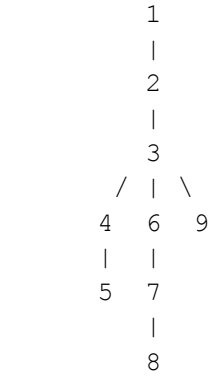
```
Prims.exe file.in 4
```

DFS.exe will output the number of connected components contained in the graph. Prims will output the minimum spanning tree of the graph to the console and it will also contain the weights. If the number of nodes to be evaluated is over 100, the output will be redirected to a file called "prims.out" and the console will inform you of this.

The Prims spanning tree will look something like this:

```
1
--2 (4)
----3 (8)
------4 (7)
--------5 (9)
------6 (4)
--------7 (2)
----------8 (1)
------9 (2)
```

This is interprted as



and the value in the parentheses are the edge weights. I got the idea for how to represent this from the following website:
https://www.geeksforgeeks.org/print-n-ary-tree-graphically/
and added a "level" member of my vertex struct to keep track of the level each vertex should be in on the tree.
The example given above is from pg 635 in the book and is provided in the folder as "testprim.in"

Lastly, once you start to run these programs the files "prims.time" and "dfs.time" will appear and start to record runtimes. Each new runtime will be appended to the file in the form: nodes edges runtime.

```
----------
VERBOSE MODE
----------
```

Verbose mode will output the final state of the Graph with its values of discovery times, predecessors, etc. depending on the algorithm. This mode is disabled when there are over 100 nodes. It is invoked by simply adding any argument. So if you want to run it with Prims, you will need to provide a root node. You can run it like this:

```
DFS.exe file.in y
Prims.exe file.in 1 y
```

The last argument can be any character input, it doesn't have to be y.

BFS and Dijkstra - DIRECTED GRAPHS

These programs are built to work with directed graphs with edge weights. The edge weights are ignored in BFS and used in Dijkstra. Each program has two modes, regular and verbose.

REGULAR MODE

To run a program in regular mode type the name of the executable with the file containing the graph like so:

```
BFS.exe file.in
Dijkstra.exe file.in
```

This means the source is set to its default value of 1. If you want to use a different source, enter it as a third argument. If you enter a root value larger than the largest node value, it will default back to one. An example for how to do this would be:

```
BFS.exe file.in 6
Dijkstra.exe file.in 6
```

BFS.exe will output the number of hops from each vertex to the source outputting infinity if the vertex is unreachable. Dijkstra will do the same except that it will output the distance according to edge weight. For both of these, if the number of nodes to be evaluated is over 100, the output will be redirected to a file called "bfs.out" or "dijkstra.out" and the console will inform you of this.

Lastly, once you start to run these programs the files "bfs.time" and "dijkstra.time" will appear and start to record runtimes. Each new runtime will be appended to the file in the form: nodes edges runtime.

VERBOSE MODE

Verbose mode will output the final state of the Graph with its values of hop distances, predecessors, etc. depending on the algorithm. This mode is disabled when there are over 100 nodes. It is invoked by simply adding any argument. So if you want to run it with these algorithms you will need to provide a source node. You can run it like this:

```
BFS.exe file.in 1 y
Dijkstra.exe file.in 1 y
```

This last argument can be any character input, it doesn't have to be y.

MinCut - FLOW GRAPHS

This program is built to work with directed graphs with edge weights, no self loops and no reverse edges. The program has two modes, regular and verbose.

REGULAR MODE

To run the program in regular mode type the name of the executable with the file containing the graph like so:

```
MinCut.exe file.in
```

This means the source and sink are set to default values of 1 and 2. This program will crash if you try to run it on a graph with fewer than two nodes. If you want to use a different source and sink, enter them as 3rd and 4th arguments. If you enter source or sink values larger than the largest node value, they will default back t 1 and 2 respectively. An example for how to do this would be:

```
MinCut.exe file.in 1 6
```

The file exflow.in is provided in the repository and is the example from CLRS of the hockey pucks. The MinCut.cpp file explains this in detail in its comments at the top. This program will output the maximum flow value between these two vertices.

Lastly, once you start to run this program the files "minCut.time" will appear and start to record runtimes. Each new runtime will be appended to the file in the form: nodes edges runtime.

VERBOSE MODE

Verbose mode will output each residual graph created. This mode is disabled when there are over 50 nodes. It is invoked by simply adding any argument. So if you want to run it with these algorithms you will need to provide a source and a sink. You can run it like this:

```
MinCut.exe file.in 1 6 y
```

This last argument can be any character input, it doesn't have to be y.

FILE STRUCTURE

---

makefile targets everything. All files except AdjListDS.cpp depend on DataStruc.cpp which has a separate header file DataStruc.h which is included at the top of all programs except for AdjListDS.cpp.

All algorithm files will require that you give an appropriate file for them to operate on.

KNOWN ISSUES

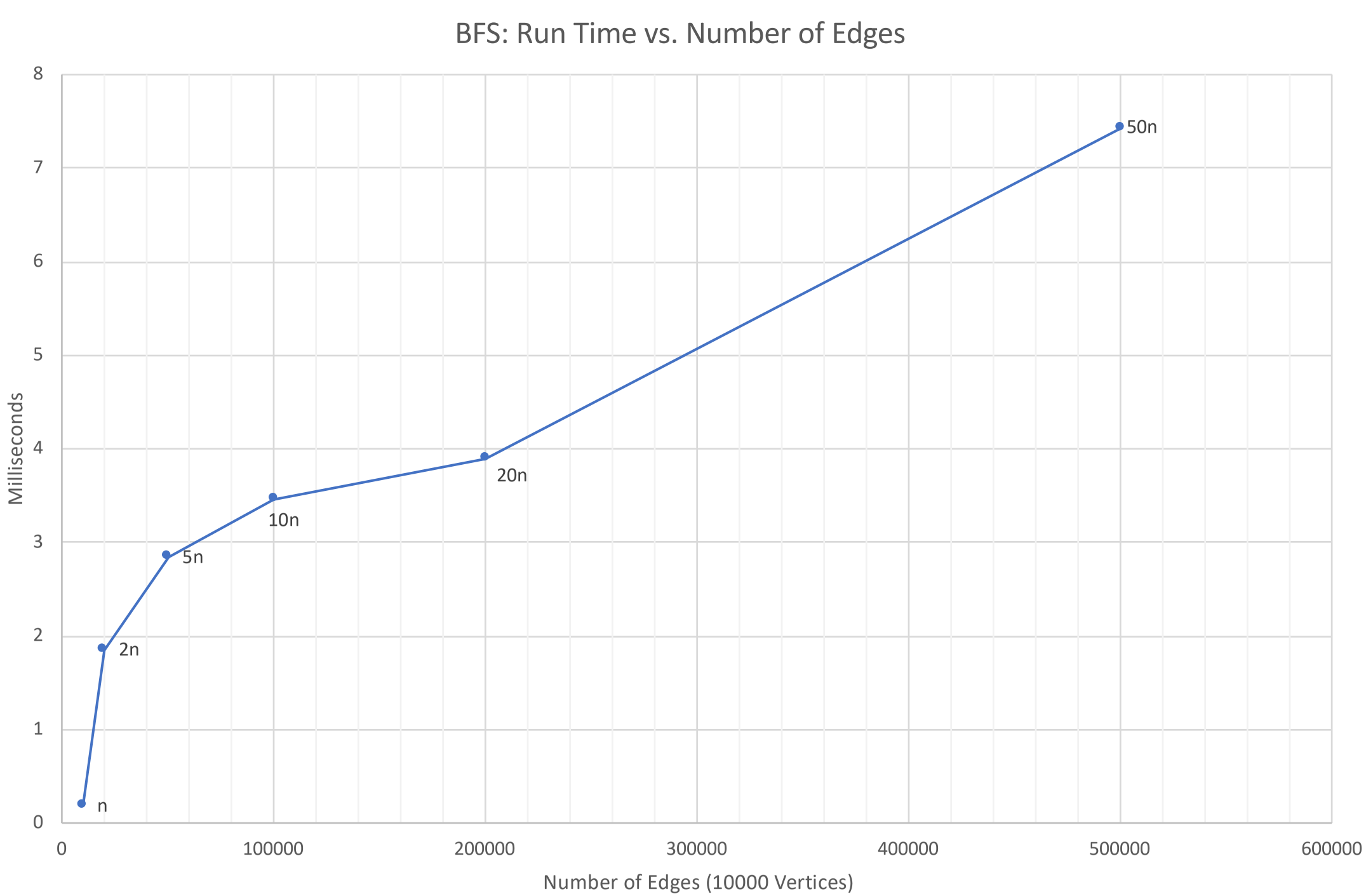
Error checking on user input is not necessarily robust.

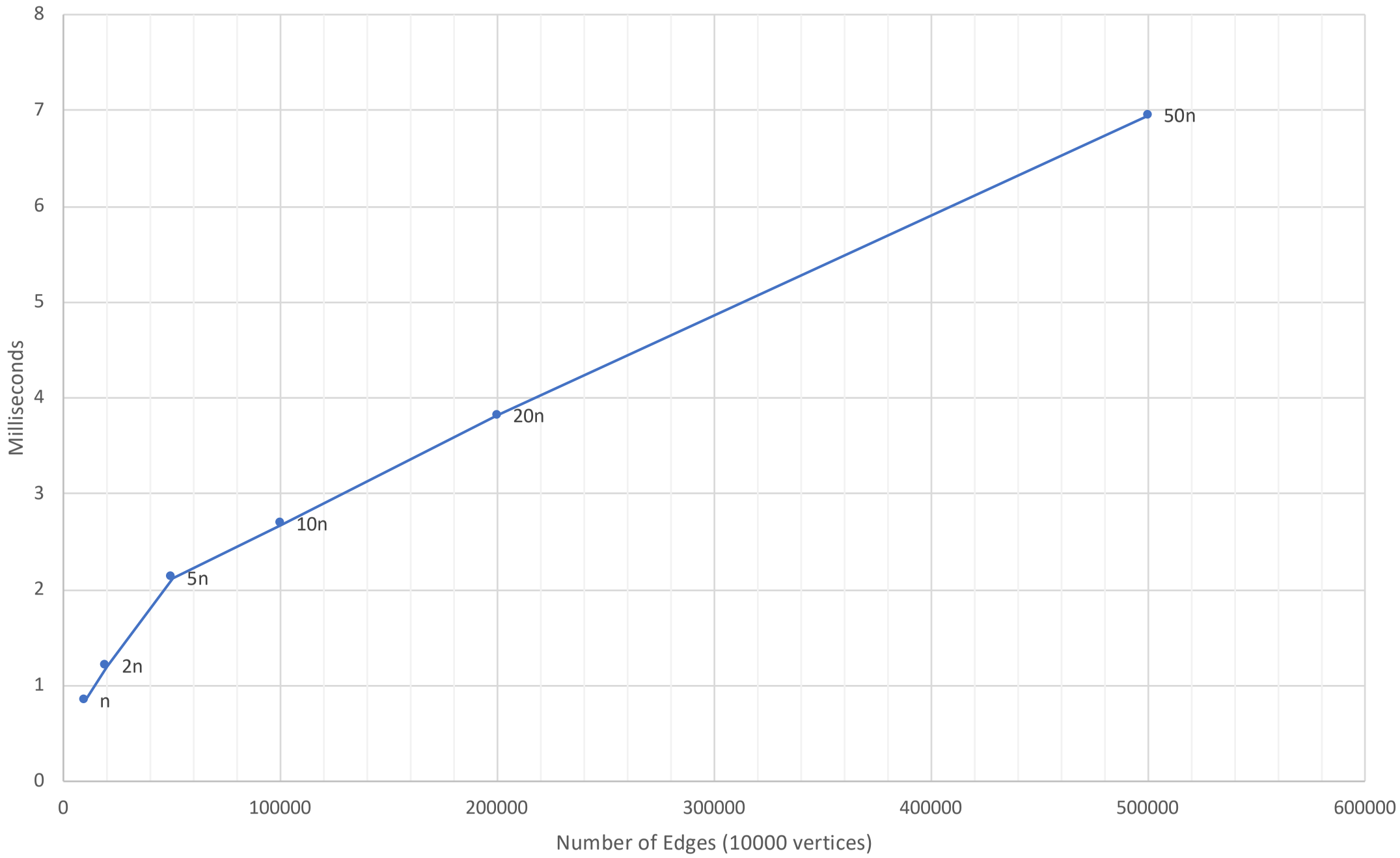
Infinity is represented by "TINY_INFINTIY" which is 2,147,483,647 which obviously has limitations.

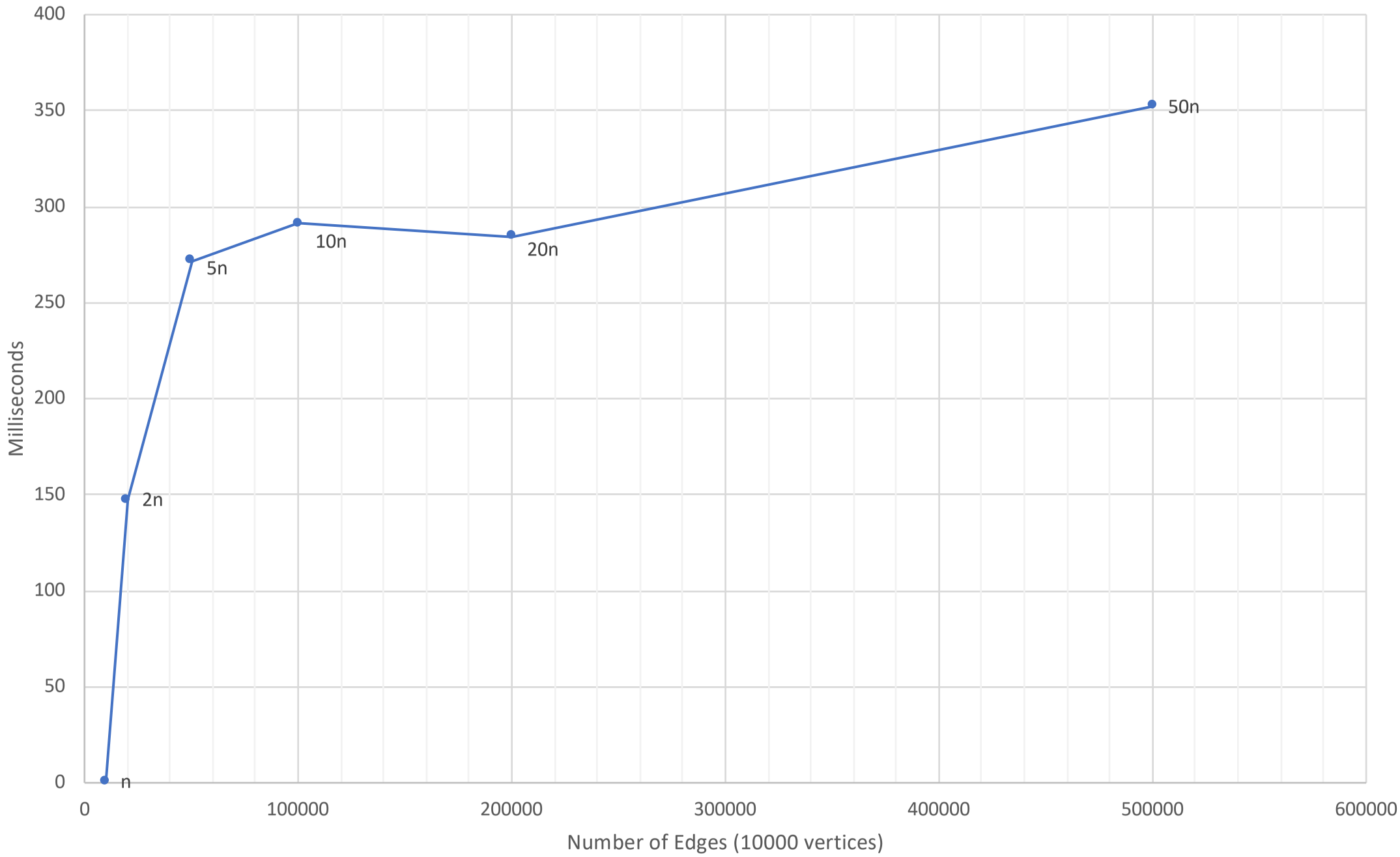
My use of Minimum Priority Queues may be problematic. See discussion on page 2.

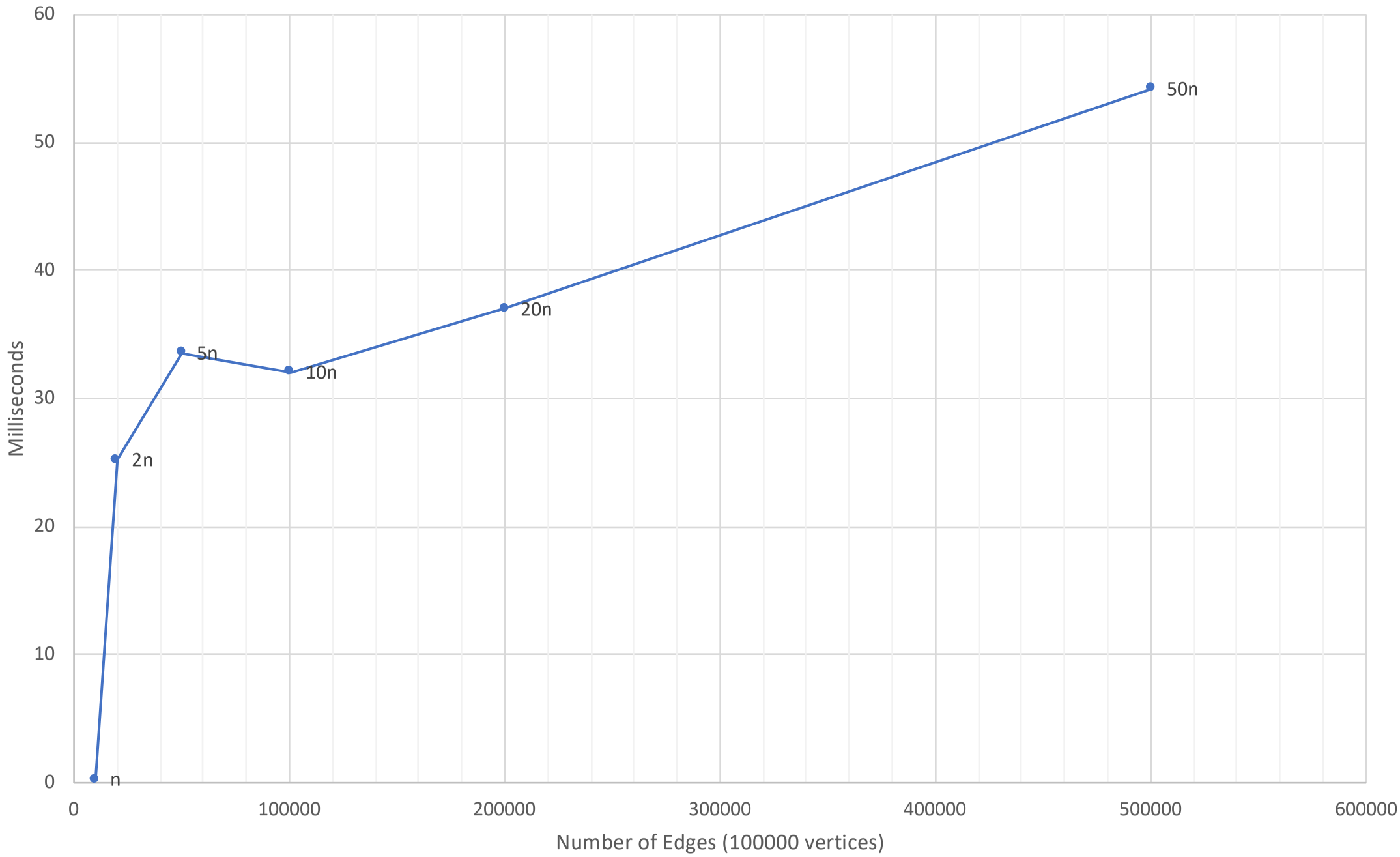
Flow graphs and Undirected graphs take substantially longer to generate due to sorting (done with quicksort).

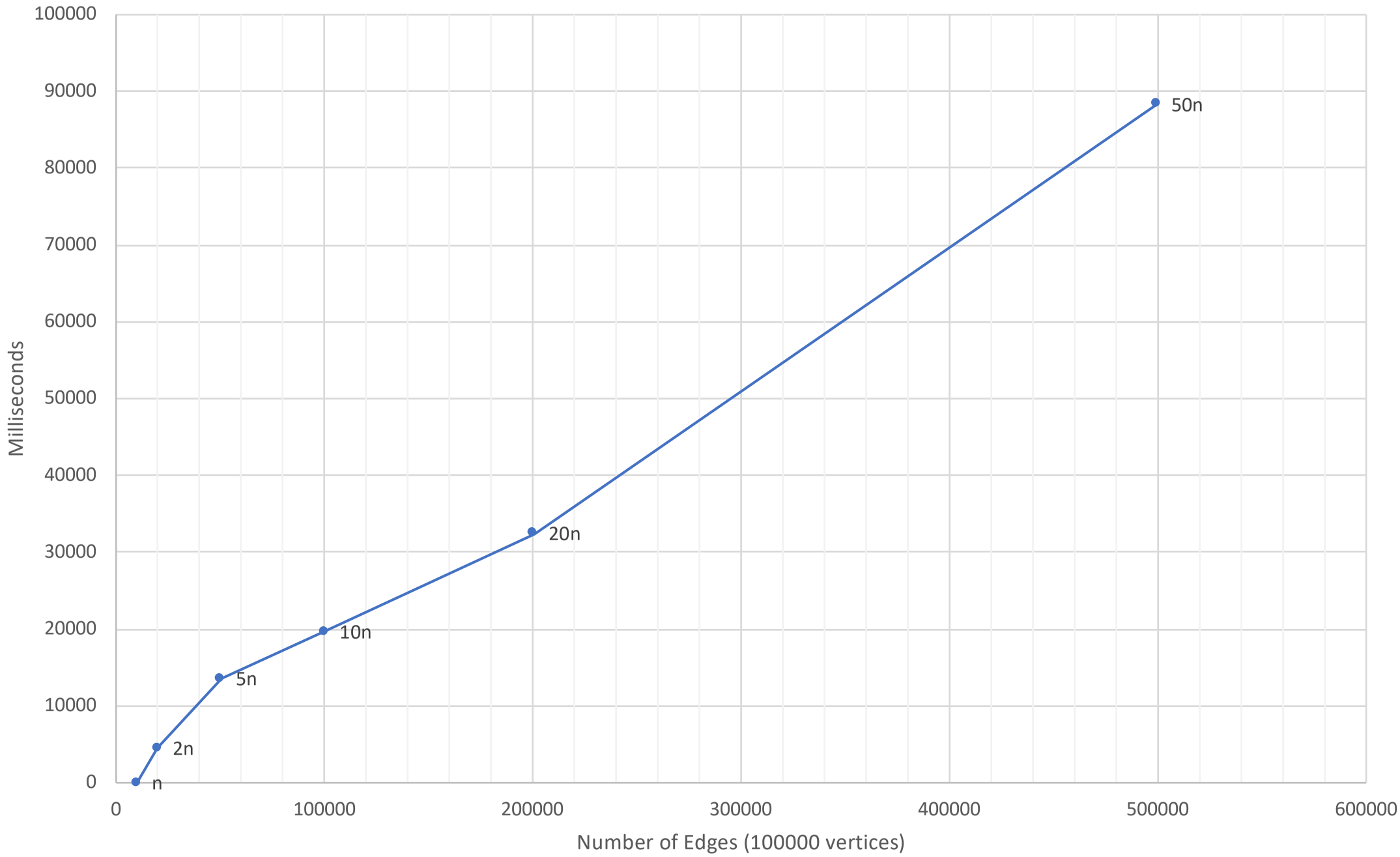
Graph are on the next 7 pages.

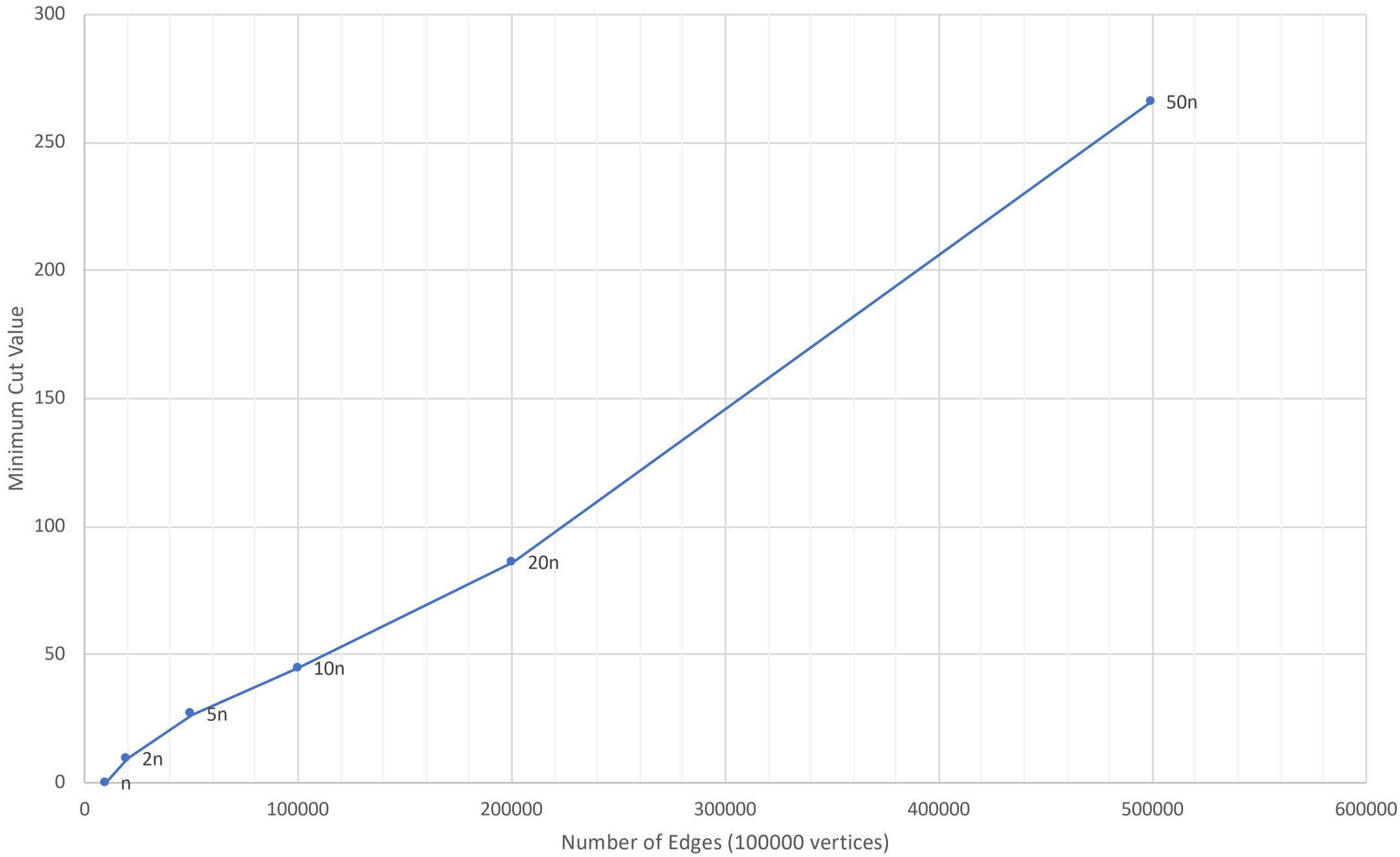

BFS: Run Time vs. Number of Edges
Milliseconds
Number of Edges (10000 Vertices)
0
1
2
3
4
5
6
7
8
0
100000
200000
300000
400000
500000
600000
n
2n
5n
10n
20n
50n


DFS: Run Time vs. Number of Edges
Milliseconds
Number of Edges (10000 vertices)
0
1
2
3
4
5
6
7
8
0
100000
200000
300000
400000
500000
600000
n
2n
5n
10n
20n
50n


Dijkstra: Run Time vs. Number of Edges
Milliseconds
Number of Edges (10000 vertices)
0
50
100
150
200
250
300
350
400
0
100000
200000
300000
400000
500000
600000
n
2n
5n
10n
20n
50n


Prims: Run Time vs. Number of Edges
Milliseconds
Number of Edges (100000 vertices)
0
10
20
30
40
50
60
0
100000
200000
300000
400000
500000
600000
n
2n
5n
10n
20n
50n


MinCut: Run Time vs. Number of Edges
Milliseconds
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
100000
200000
300000
400000
500000
600000
Number of Edges (100000 vertices)
n
2n
5n
10n
20n
50n


Minimum Cut vs. Number of Edges
Minimum Cut Value
Number of Edges (100000 vertices)
300
250
200
150
100
50
0
0
100000
200000
300000
400000
500000
600000
n
2n
5n
10n
20n
50n