Homework 2 – Dijkstra's All Pairs Shortest Path

Advanced Data Structures

COP 5536 – Fall 2011

Aakash Shah
UFID: 5249-4301
aakash@cise.ufl.edu

Contents

Compilation

- ✓ Execute './compile.sh' on the prompt
- ✓ Make sure you have permissions to execute 'compile.sh'.
- ✓ Alternatively, you can execute 'gcc –o ssp main.c random.c utility.c adjlist.c simple.c bino.c fibo.c' on the prompt.

Note: The program was written in C, built on a machine running Ubuntu 11.10 and gcc (version 4.6.1) was used to compile and link the files in the program.

Execution

- ✓ You are ready to run the program using the following options:

Command	Meaning
./ssp –r	Random Mode
./ssp –is	Calculations using array– Manual Input
./ssp –ib	Calculations using binomial heap – Manual Input
./ssp –if	Calculations using fibonacci heap – Manual Input
./ssp –i[s|b|f] ***filename***	Calculations using the method specified – Input taken from the file '*filename*'.

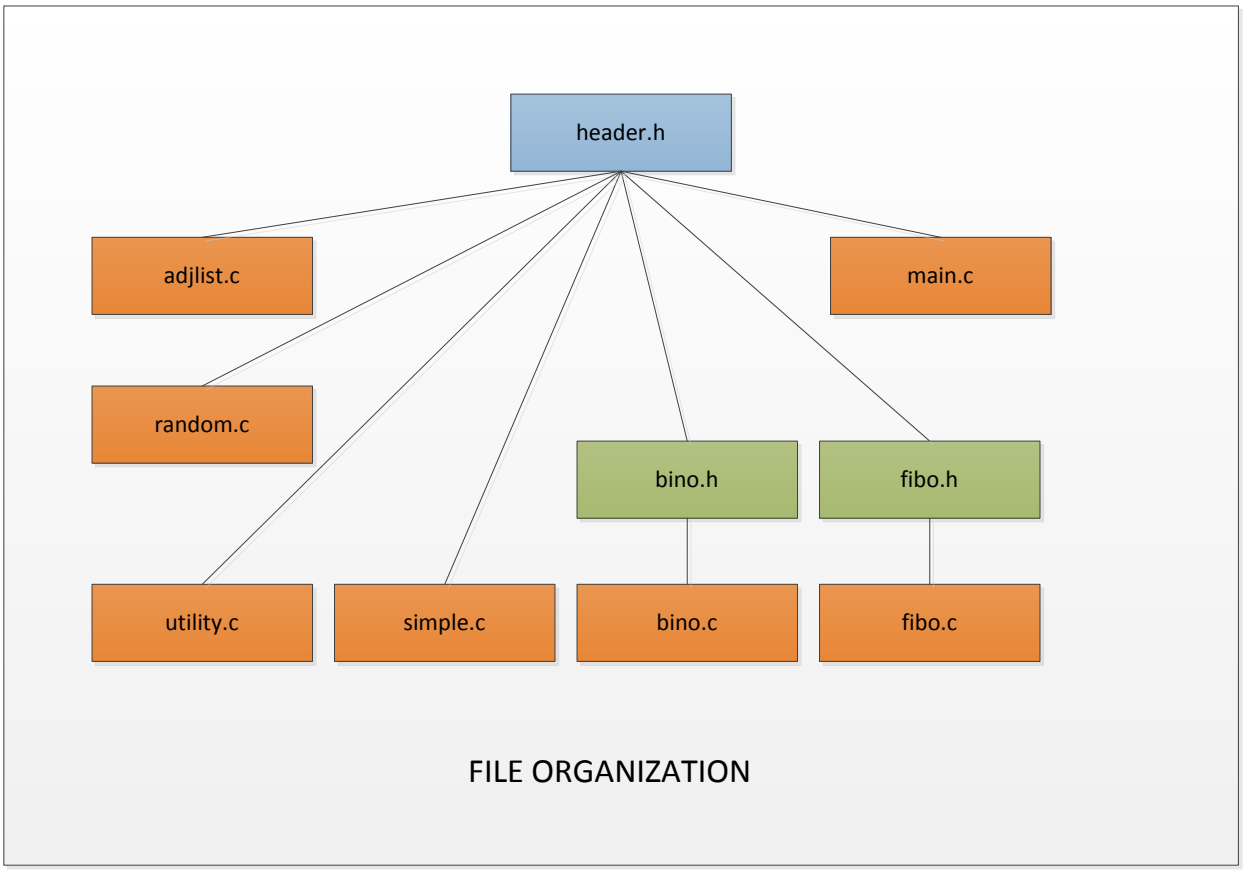
- ✓ If you choose to manually input the graph data, you should to STRICTLY in '<V1> <V2> <Cost>' format. When you are done, enter only '*' and the program will start executing and will display the results if successful. If not, it should throw a reasonable error, which would guide you to enter the input in a proper format.
- ✓ If the input is taken from the file, the data in the file should STRICTLY adhere to the format '<V1> <V2> <Cost>'. If there are multiple entries, they should be mentioned in subsequent lines, each entry taking up only one line. '*' is not needed to mark the end of data.
- ✓ The output will be in a tabular form, providing the minimum cost from every vertex to every other vertex. If it is not possible to reach a vertex (in case of a disconnected graph), it should display INF. A sample screenshot is attached here to give you an idea.

```
aakash@ubuntu:~/ADS$ ./ssp -ib small.txt

Nodes   0       1       2       3       4       5       6       7       8       9
        ------  ------  ------  ------  ------  ------  ------  ------  ------  ------
0   |   0       1170    714     1446    914     1777    1431    453     1020    1169
1   |   276     0       990     910     21      884     538     729     127     276
2   |   2202    2615    0       1476    1915    2742    945     1898    1505    2134
3   |   726     1160    1440    0       439     1266    920     1179    29      658
4   |   812     757     1526    889     0       863     517     855     106     255
5   |   160     1330    874     1033    1074    0       502     613     1062    1329
6   |   1257    1670    1841    531     970     1797    0       953     560     1189
7   |   993     717     888     993     461     1324    978     0       567     716
8   |   1186    1131    1900    783     410     1237    891     1265    0       629
9   |   557     502     1271    793     523     608     262     1010    629     0
```

File Organization

Following is the diagrammatic representation of how the files are organized in the program.



FILE ORGANIZATION

File Contents

main.c

The is, as the name suggests, the file containing main(). This file has the logic to detect the mode specified by the user and perform invoke the correct function based on it.

- ✓ main() scans the mode specified by the user and invokes the function accordingly.

adjlist.c

This file contains functions related to the management of adjacency list.

- ✓ initAdjList() creates an adjacency list of size 'gNoOfVertex', a global variable. This variable can hold a value of maximum MAX_VERTEX_NUM (500).
- ✓ addToAdjList() adds the given pair of vertices and the cost associated with them into the adjacency list.
- ✓ doesPairExistsInAdjList() checks if the given pair of vertices exists in the adjacency list. If they exist, returns the cost associated with them.

- ✓ buildRandomAdjList() builds a random graph represented via an adjacency list. The information about the graph density and vertices are given by two global variables gGraphDensity and gNoOfVertex. The minimum value of gGraphDensity is 10% while the maximum value is 100%. The number of edges are calculated by gNoOfVertex and gGraphDensity with the formulae: noOfEdges = (gGraphDensity * gNoOfVertex * (gNoOfVertex – 1))/ 100. The cost of each edge is decided randomly. It is restricted in the interval [1, MAX_EDGE_COST]. MAX_EDGE_COST (1000).
- ✓ connectGraph() connects the graph if it is disconnected. Useful in random mode when the randomly generated graph is disconnected. It checks the same and adds edges into the graph until it is connected.
- ✓ destroyAdjList() deallocates the memory used up by the adjacency list.

utility.c

This file has all the helper functions needed to run the program. The list of important functions are mentioned below:

- ✓ myLog() prints the given log on screen with the filename, the line number and the log level. Three defined log levels are DEBUG, INFORMATIONAL and ERROR.
- ✓ copyFromAdjListToDist() copies the data from an adjacency list to the dist[][] matrix.
- ✓ printDistMatrix() shows the output to the user in a tabular format with the cost associated between vertex pairs in the cells of the table.
- ✓ printTimeMatrix() shows the output to the user in a tabular format with the time taken by a scheme for computing the dist[][]. All times shown in the table are in milliseconds.

simple.c

This file contains all functions related to the simple scheme. Simple scheme uses array of size nxn = n^2 to perform all source shortest path calculations; where n is the maximum vertices in the graph.

- ✓ simpleProc() is the entry function which takes the input from the user and generates the adjacency list. From this adjacency list, it calculateDistanceMatrixSimpleScheme() and then displays the matrix to the user via printDistMatrix().
- ✓ simpleFileProc() is the entry function which reads the input from the file name given by the user and using the same method as simpleProc(), displays the dist[][] matrix to the user via printDistMatrix().
- ✓ simpleSchemeGetClosestVertex() behaves as the typical removeMin() operation.
- ✓ simpleSchemeRelax() behaves as the typical decreaseKey() operation.

bino.c

This file contains all the functions which are needed while calculating the dist[][] using the binomial scheme. Binomial scheme uses Binomial heaps to perform all source shortest path calculations.

- ✓ binoProc() and binoFileProc() behave exactly the same as simpleProc() and simpleFileProc() but instead of calculateDistanceMatrixSimpleScheme () they call runBinoScheme() to compute the dist[][].
- ✓ initBinomialHeap() initializes the binomial heap, dist[][] and copies the details in adjList to dist[][]. The values will be further relaxed. It returns a blank binomial heap.
- ✓ getBlankBinoNode() allocates memory for a new binomial node and returns it back to the caller function.
- ✓ binoInsert() inserts the given BinoNode into the Binomial Heap given by initBinomialHeap(). It updates the root if the newly inserted node's key is lesser than the existing root.
- ✓ binoMeld() melds the two binomial trees/heaps and returns the new root of the melded tree/heap.
- ✓ binoCombine() combines two binomial trees of the same degree and returns the combined binomial tree pointing at the root of the combined tree.
- ✓ binoRemoveMin() performs the removeMin operation on the binomial heap. It also frees the memory of the root node which was popped from the binomial heap.
- ✓ binoDecreaseKey() performs the decrease key operation. If the key after the decrease of a node is less than its parent, parent and the node are swapped until either the node does not have a parent or the key of the node is not lesser than its parent.
- ✓ isBinoHeapEmpty() checks if the heap is empty or not.
- ✓ destroyBinoHeap() frees the memory used up by the binomial nodes when called. It might be useful when computations fail due to unforeseen reasons and the binomial heap is not empty.
- ✓ runBinoScheme() takes a adjacency list as an input and calculates the dist[][].

fibo.c

This file contains all the functions which are needed while calculating the dist[][] using the fibonacci scheme. Binomial scheme uses Fibonacci heaps to perform all source shortest path calculations.

- ✓ fiboProc() and fiboFileProc()behave exactly the same as simpleProc() and simpleFileProc() but instead of calculateDistanceMatrixSimpleScheme () they call runFiboScheme () to compute the dist[][].
- ✓ initFibonacciHeap() initializes the binomial heap, dist[][] and copies the details in adjList to dist[][]. The values will be further relaxed. It returns a blank fibonacci heap.
- ✓ getBlankFiboNode() allocates memory for a new fibonacci node and returns it back to the caller function.
- ✓ fiboInsert() inserts the given FiboNode into the Fibonacci Heap given by initFibonacciHeap(). It updates the root if the newly inserted node's key is lesser than the existing root.
- ✓ fiboMeld() melds the two fibonacci trees/heaps and returns the new root of the melded tree/heap.
- ✓ fiboCombine() combines two fibonacci trees of the same degree and returns the combined fibonacci tree pointing at the root of the combined tree.
- ✓ fiboRemoveMin() performs the removeMin operation on the fibonacci heap. It also frees the memory of the root node which was popped from the binomial heap.

- ✓ fiboDecreaseKey() performs the decrease key operation. If the key after the decrease of a node is less than its parent, the node is isolated from the fibonacci heap and then fiboMeld()'d to the heap. The childCut value of the parent is made to be TRUE, if it was FALSE. It already TRUE, the parent is also isolated from the fibonacci heap. This process is repeated until we find a parent whose childCut is FALSE or there are no more parents.
- ✓ isFiboHeapEmpty() checks if the heap is empty or not.
- ✓ destroyFiboHeap() frees the memory used up by the fibonacci nodes when called. It might be useful when computations fail due to unforeseen reasons and the fibonacci heap is not empty.
- ✓ runFiboScheme() takes a adjacency list as an input and calculates the dist[][].

random.c

This invokes all schemes one after the other. Each scheme is run with 100, 200, 300 and 500 vertices with 10%, 20%, 30%, 40%, …, 100% graph density. For each vertex and density pair, each scheme is run five times and then the average time is computed.

- ✓ randomProc() invokes the above mentioned procedure. At the end, it printTimeMatrix() for each scheme individually.

*.h

All header files contain the macros and node structures required by the program. They are trivial and can be easily understood by looking inside each header file.

Program Evaluation

User Input Mode

The user provides the input in the format discussed in the 'Execution' Section. The graph may be connected or disconnected. Dijkstra's algorithm is ran by using the scheme specified by the user and the dist[][] matrix is displayed on screen to the user. User is not allowed to enter more than 500 vertices. No time evaluations would be done in this mode.

User Input File Mode

The user provides the file name from which the data is to be read instead of providing the input manually. Rest things do not differ from the user input non-file mode.

Random Mode

Following steps would be executed in the random mode:

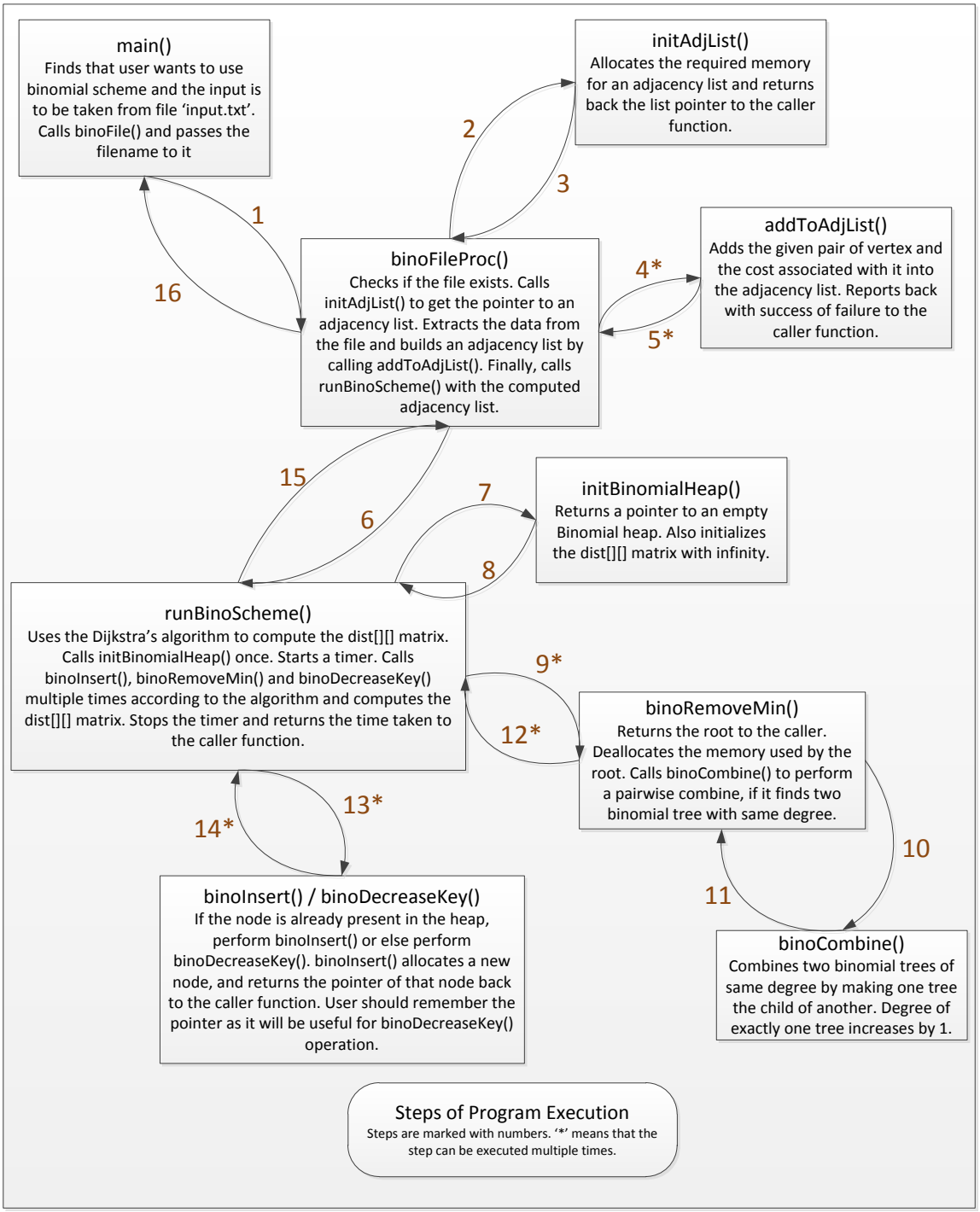
- ✓ Generate a graph randomly and check if the graph is disconnected by running simple scheme on that graph. If disconnected, add an edge until the graph becomes connected.
- ✓ Start the clock. Run simple scheme. Stop clock.
- ✓ Repeat the above step for binomial scheme and fibonacci scheme.
- ✓ Display time matrix.

Sample Program Flow

Let's say that the user wants to compute the dist[][] using binomial scheme. The user wants to enter the data manually. He enters the following at the prompt:

$ ssp –ib input.txt

The following is the sequence of function execution



Comparison of Schemes

Theoretical Complexities

There are two main aspects to be taken into consideration for calculating the time complexity of computing the dist[][] via Dijkstra's algorithm:

1. **Remove Min:**
   RemoveMin() operation is done n times, where n are the number of vertices in the graph.
2. **Decrease Key:**
   DecreaseKey() operation is done e times, where e are the number of edges in the graph.

The following table shows the amortized complexity of the operations of the given scheme:

	Simple Scheme	Binomial Scheme	Fibonacci Scheme
Dependencies	n	n and e	n and e
Remove Min	O(n)	O(log n)	O(log n)
Decrease Key	O(1)	O(log n)	O(1)
Time Complexities (Single Source)	$O(n^2)$	O(n log n + e log n)	O(n log n + e)
Time Complexities (All Source)	$O(n^3)$	$O(n^2 \log n + ne \log n)$	$O(n^2 \log n + ne)$

Expectations

Keeping in mind the theoretical time complexities shown above,

- ✓ Simple Scheme should not depend on the density of graph.
- ✓ Simple Scheme should depend only on the number of vertices in graph by a cubic factor.
- ✓ Binomial Scheme and Fibonacci Scheme should linearly depend on the density of graph.
- ✓ Binomial Scheme and Fibonacci Scheme for a given density should depend on the number of vertices in graph by a factor of $n^2 \log n$.
- ✓ For sparse graphs, fibonacci scheme should be the fastest followed by binomial scheme and simple scheme.
- ✓ For dense graphs, simple scheme should be the fastest followed by fibonacci scheme and binomial scheme.

Practical Revelations

The following is the result captured by running all three schemes on the same connected directed graph with vertices 100, 200, 300 and 500; and with densities ranging from 10% to 100%. Each value shown is in milliseconds. For accuracy, each scheme was run 5 times and then an average time was taken.

--- SIMPLE SCHEME ----

	10	20	30	40	50	60	70	80	90	100
100	[22]	[28]	[32]	[32]	[32]	[32]	[34]	[32]	[32]	[32]
200	[236]	[238]	[244]	[250]	[254]	[256]	[258]	[258]	[250]	[252]
300	[782]	[814]	[828]	[834]	[876]	[920]	[926]	[912]	[928]	[970]
500	[4538]	[4602]	[4846]	[5168]	[5256]	[5532]	[5800]	[5540]	[5488]	[5694]

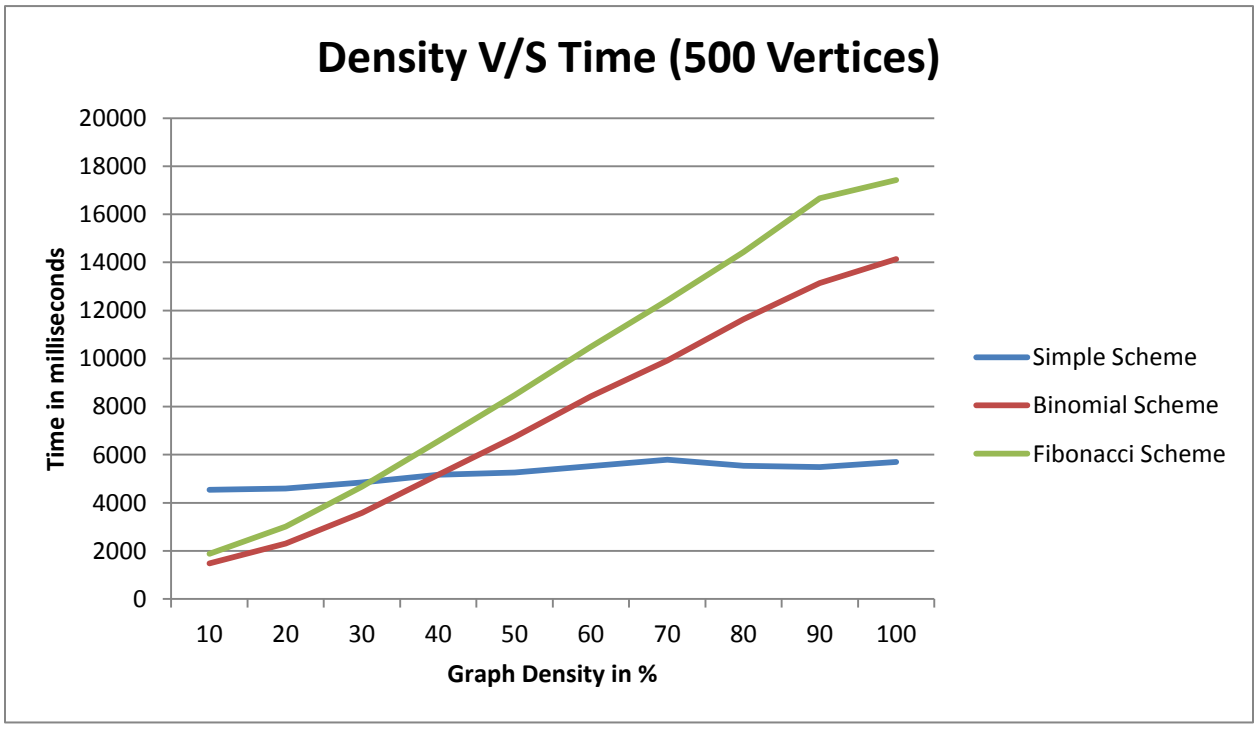
--- BINOMIAL SCHEME ----

	10	20	30	40	50	60	70	80	90	100
100	[14]	[16]	[20]	[20]	[22]	[24]	[30]	[26]	[34]	[34]
200	[96]	[114]	[132]	[152]	[168]	[202]	[210]	[232]	[264]	[312]
300	[288]	[352]	[428]	[522]	[700]	[890]	[1136]	[1408]	[1728]	[2072]
500	[1486]	[2306]	[3580]	[5172]	[6740]	[8434]	[9926]	[11632]	[13140]	[14134]

--- FIBONACCI SCHEME ----

	10	20	30	40	50	60	70	80	90	100
100	[20]	[18]	[26]	[28]	[34]	[38]	[42]	[42]	[50]	[52]
200	[114]	[150]	[180]	[210]	[246]	[274]	[310]	[346]	[392]	[442]
300	[350]	[464]	[584]	[728]	[926]	[1158]	[1512]	[1834]	[2186]	[2560]
500	[1878]	[3016]	[4674]	[6572]	[8476]	[10492]	[12422]	[14432]	[16670]	[17424]

For 500 vertices, following is the comparison of different schemes with the increase in density (edges) of graph:

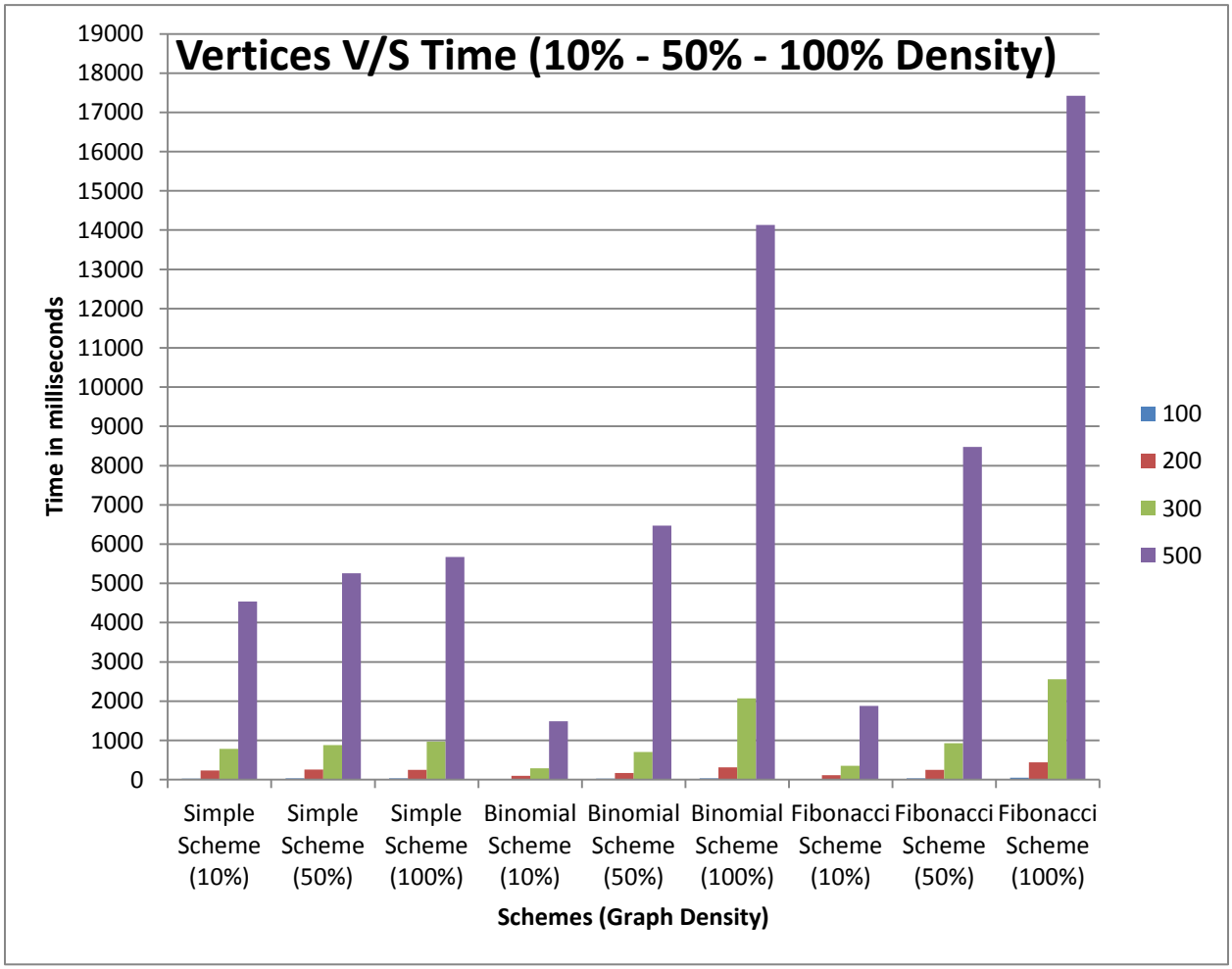


Analysis:

- ✓ The graph for simple scheme remains almost constant for all densities and hence it does not depend much on the number of edges in the graph. This is in line with its overall complexity, $O(n^3)$, which does not depend on edges.

- ✓ The graph for binomial scheme and fibonacci scheme depend on the density of graph and increase by a constant factor (straight line). This is in line with it their overall complexity proportional to e. Binomial Scheme: $O(n^2 \log n + ne \log n)$ and Fibonacci Scheme: $O(n^2 \log n + ne)$. As n = 500 (constant) here, time depends on e.
- ✓ For sparse graphs, simple scheme is slower than the other two, but binomial scheme is surprisingly faster than fibonacci scheme.
- ✓ For dense graphs, simple scheme is faster than the other two, but again, binomial heap is faster than fibonacci scheme.

For 10%, 50% and 100% graph density, following is the comparison of different schemes with the increase in number of vertices:



Analysis:

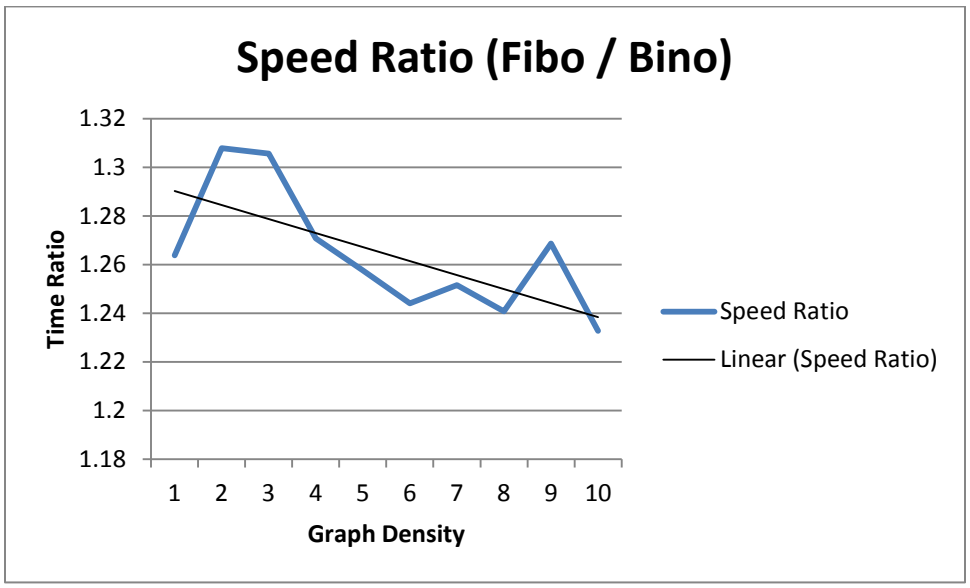
- ✓ Time taken for each scheme depends on the number of vertices in the graph.
- ✓ For Simple Scheme, at 50% density D, with 100 vertices V, the time taken T is 32ms. Taking fractional value of $T/V^3 = 32/100^3 = 0.000032$. For D = 50%, V = 300, T = 876. Fractional value of $T/V^3 = 876/300^3 = 0.000032$. For D = 50%, V = 500, T = 5256. Fractional value of $T/V^3 = 5256/500^3$

= 0.00004 = 0.4×10^{-5}. As the fractional values are close to each other, time complexity is indeed $O(n^3)$.

- ✓ Similar fractional calculations ($T/(n^2 \log n + ne \log n) = 0.000012 = 1.2 \times 10^{-5}$) for binomial scheme and fractional calculations ($T/(n^2 \log n + ne) = 0.00014 = 1.4 \times 10^{-4}$) also reveal that the time complexity is as per expectations.

Conclusion

- ✓ Everything until now is as per expectations except one thing i.e. fibonacci scheme is practically slower as compared to binomial scheme. Let us see the speed ratio of fibonacci scheme / binomial scheme.



- ✓ The graph shows a clear decrease in the speed ratio of binomial over fibonacci scheme. If we look at the derived constants in the previous analysis, we can see that fibonacci has a bigger constant than that of binomial (10 times bigger). This is same as insertion sort ($O(n^2)$) outperforming quick sort ($O(n \log n)$) for small value of n because quick sort has a bigger constant as compared to the constant of insertion sort.
- ✓ Also fibonacci will be faster than binomial where there are frequent decreaseKey operations as the complexity of removeMin is the same for both. Dense graphs have higher probability of decreaseKey operation as compared to sparse graph.
- ✓ Thus, fibonacci scheme should be faster than binomial scheme for very large value of n and when the density of the graphs is very high.
- ✓ For our given limited comparatively smaller data set, binomial scheme out performs fibonacci scheme. The other expectations are still valid.