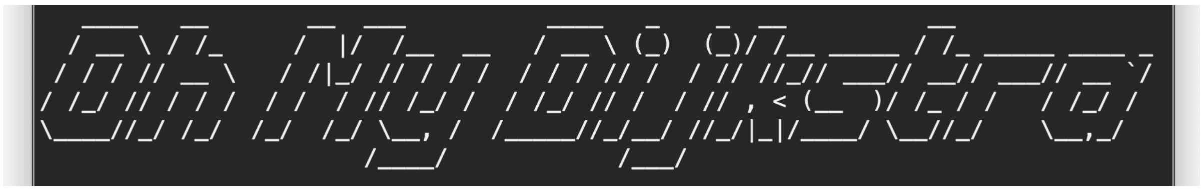


Report on ADS Project 2

Shortest Path Algorithm with Heaps

GROUP 7

程政淋 米博宇 黄亦霄

27 March 2022

1. Programming

Problem Landing

Dijkstra Algorithm for shortest path in weighted graphs is known to be essential in many aspects. As `unknown minimal` vertices are always needy, the data structure enabling speedy search for them, say **min-heap**, will surely make a difference. In this project, we tried three different data structures to install Dijkstra Algorithm, namely **Ordinary, Binomial Heap, Fibonacci Heap**. Additionally, `STL` data structure `map` and a hash table `unordered_map` storing direct address for each vertex are used to **further reduce time complexity** though causing little extra memory usage. Details will be given in **Chapter 2**.

STL `map` & `unordered_map`

`map`

We use `map` to store our graph because some test cases are not connected, their vertices are not continuous, to simplify index visiting and consider to reduce the waste of memory due to a fixed size of array, i.e. `int g[100000000]`, We choose `map`.

Based on **Red-Black Tree**, the time complexity for some operations are as follows:

```
+------------------+--------------------------+
|    Expression    |        Complexity        |
+------------------+--------------------------+
| a_uniq.insert(t) | logarithmic              |
| a.erase(k)       | log(a.size())+a.count(k) |
| b.find(k)        | logarithmic              |
+------------------+--------------------------+
```

The time complexity of `map` is guaranteed to be `O(log n)`.

`unordered_map`

We find that when `DecreaseKey()`, it's hard to offer the address of the heap node. Because fibonacci heap does not support search operations well. if we search for the node, it would be linear search, which would make heap optimization useless. So we consider space-for-time practice, we use a hash table to supply direct entry for heap node.

Based on **hash table**, the time complexity for some operations are as follows:

```
+------------------+-----------------------------------------------------+
|    Expression    |                      Complexity                     |
+------------------+-----------------------------------------------------+
| a_uniq.insert(t) | Average case O(1), worst case O(a_uniq.size()).     |
| a.erase(k)       | Average case O(a.count(k)), worst case O(a.size()). |
| b.find(k)        | Average case O(1), worst case O(b.size()).          |
+------------------+-----------------------------------------------------+
```

Algorithm Illustration

Dijkstra Algorithm takes the same thoughts as unweighted shortest problems. Rooted in the source vertex, and for each step begins withthe unvisited vertex with least distance from source. The basic idea is updating diatance for all adjacent vertices If one node is not yet found connected with source (denotes as INF) or had some larger distance. When all vertices are marked visited, the denoted distance should be shortest.

Fibonacci Heap

DEFINITION

In a fibonacci heap, a node can have more than two children or no children at all. Also, it has more efficient heap operations than that supported by the binomial and binary heaps.

The fibonacci heap is called a fibonacci heap because the trees are constructed in a way such that a tree of order n has at least $F(n+2)$, the $(n+2)^{th}$ Fibonacci number, nodes in it.

ADVANTAGES

by using circular doubly linked list to store all nodes with same "seniority", we have

- `O(1)` time for node deleting;
- `O(1)` time for concatenation of two such lists takes.

OPERATIONS

- Insert: if empty, insert new node; else, insert and update min.

```
procedure fib::insert():
  new singleton tree.
  Add to root list;
  update min pointer (if necessary).
```

- FindMin: though not sorted, `min` the pointer always points at min element.
- Union: connect the roots of both the heaps, update min by selecting a minimum key from the new root lists. Delete the min node.
- DecreaseKey: mark it by setting a min value for the key to delete. If a marked node loses another child, then cut it from its parent. This node now becomes the root of a separate tree and is no longer marked. This is called a cascading cut, because several of these could occur in one decrease_key operation.

```
procedure fib::decreasekey():
  if heap-order is not violated:
    decrease the key of x;
  else
    cut tree rooted at x;
    meld into root list;
  if node has its second child cut:
    cut it off;
    meld into root list (and unmark it);
```

Key Class Specification and File Arrangments

- main.cpp

  Two entries either for systematic testing or user interface.

- Graph.h

  `class Graph` with functions `ReadGraph` and `ReadGraphTest`.

  Storing directed graphs as **adjacency list**. Both functions are highly customed for testing files given by 9th DIMACS Implementation Challenge - Shortest Paths, where all files consists of lines of text beginning with some certain identifier. Say `a` for arc, in format of

  ```
  a <(int)vertex_id_from> <(int)vertex_id_to> <(int)distance>
  ```

  while `p` gives total number of arcs and vertices.

  `class ProgBar` with functions `ResetProgress`, `PrintProgress` and `ProgressBar`.

  Relative to progress-bar component, indicating the percentage accomplished in console with

- Graph.cpp
- FibonacciHeap.h

  `class Fibonacci` with functions `Insert`, `Union` , `DeleteMin` and `DecreaseKey`.

- FibonacciHeap.cpp
- Dijkstra.h

  `class Dijkstra` with 3 main entries of all implements.

- Dijkstra.cpp
- CMakeLists.txt

  dependancies in the project.

- BinHeap.

  `class BinHeap`

  `class BinNode`

- BinHeap.cpp

2. Testing

Environment Specification

For neat horizontal comparison, all tests taken into consideration were done on one single device whose configuration is as follows. Project is built with `cmake` in the release mode.

```
Apple clang version 13.0.0 (clang-1300.0.18.6)
Target: arm64-apple-darwin21.4.0
Thread model: posix
Operation system: macOS Monterey 12.3 (21E230)
Processor: Apple M1 @ 3.2GHz
```

Since our testing program is single-threaded, the average CPU occupancy is 100%, i.e. exactly one core is taken up. This ensures running time for all test cases being approaxmacitly proportional to time complexity at some certain scope.

Testing Stratagies

Testing files are full and regional USA road network with distance. Links for all 12 files are given below.

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.USA.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CTR.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.W.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.E.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.LKS.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CAL.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NE.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NW.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.FLA.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.COL.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.BAY.gr.gz

http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NY.gr.gz

The size of road networks ranges from 733846 arcs connecting 264346 vertices, to 58333344 arcs connecting 23947347 vertices. Details are as follows, and case 1, 4, 10 are used most often since these are relatively small.

file no.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
arcs	733846	800172	1057066	2712798	2840208	3897636	4657742	6885658	8778114	15248146	34292496	58333344
vertice	264346	321270	435666	1070376	1207945	1524453	1890815	2758119	3598623	6262104	14081816	23947347

For all 12 files, randomly choose valid src to run Dijkstra of both implymentations, and have a valid random terminal to print the shortest path. Pseudocode of test program is given here.

```
procedure test():
  for testfile in filelist:
    graph = read graph from testfile;
    for testcount in range(test iterations):
      src = random valid source;             // generate random test pair
      ter = valid random terminal;

      resetdij();
      dijkstra_bin(graph, src);              // binomial heap
      printpath(src, ter) and generate log;

      resetdij();
      dijkstra_fib(graph, src);              // fibonacci heap
      printpath(src, ter) and generate log;
      if file is small enough to perform oridinary method:
        resetdij();
        dijkstra_ord(graph, src);           // ordinary implementation
        printpath(src, ter) and generate log;
```

Larger files runs fewer iterations, while smaller more. For the two smallest cases, ordinary approach is also tested. Details are given in the table below, and 3 journal files `Ord.txt`, `Bin.txt`, `Fib.txt` are in the submitted folder.

File ID	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Ordinary Dijkstra	1	0	0	0	0	0	0	0	0	1	0	0
Binominal Heap Dijkstra	810	10	10	810	10	10	10	10	10	810	10	10
Fibonacci Heap Dijkstra	810	10	10	810	10	10	10	10	10	810	10	10

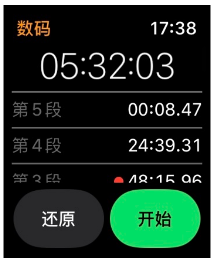
Totally, 2522 pair of src-ter were tested on 12 standardized files.

The log file `Log.txt` contains 1200 pair of testing results each in the format below.

```
TEST CASE 1 BIN 321270 vertices, 354.827ms
The pathLen is 1773710, path from 16807 to 78919 is:
78919 <- 78914 <- 78912 <- 78894 <- 78174 <- 78176 <- 78177 <- 77634 <- ... <- 16807
```

For all pairs the two methods gives same output, making result more convincing.

Moreover, before turning to standardized data set, we also tested on some extreme files, say a subset of American Road Network with over 20 million vertices. The result is just for FUN. However, the steep contrast also gave us a shocking alert of the significance of using proper data structures.

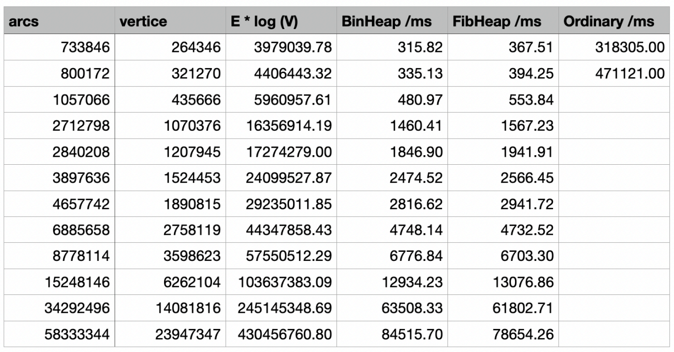


(*Ran over 5.5 hours while Fibonacci Heap implement costs only 4 secs.*)

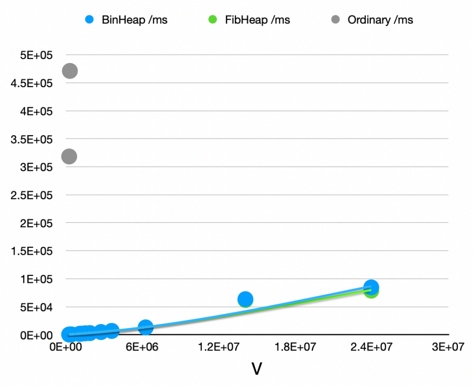
Heap vs. Non-Heap

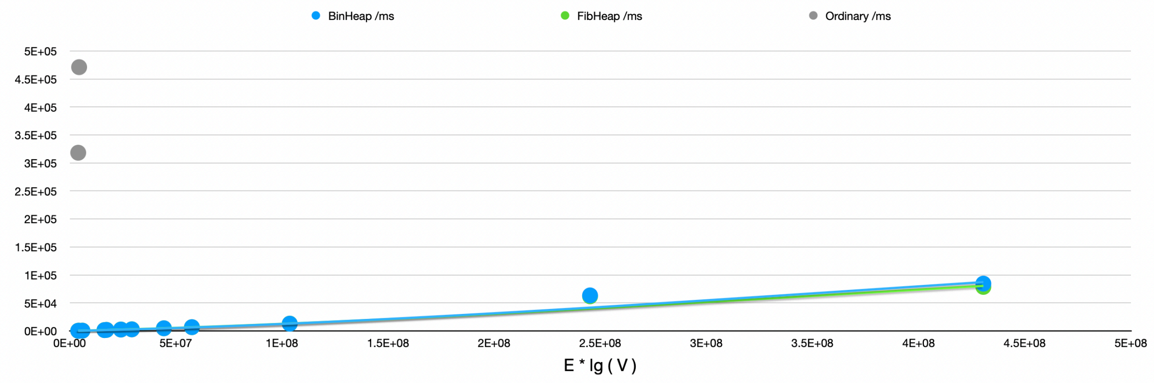
For one thing, heap is highly optimized for `delete_min`, which Dijkstra made full use of. For another, all heap implementations take an extra mapping table to directly plot node to memory address.

From the diagram below, we find it ridiculously slow when using ordinary method, since all delete-min's are linear search, resulting in square time complexity, $O(|E|+|V|^2)$ for instance.



arcs	vertice	E * log (V)	BinHeap /ms	FibHeap /ms	Ordinary /ms
733846	264346	3979039.78	315.82	367.51	318305.00
800172	321270	4406443.32	335.13	394.25	471121.00
1057066	435666	5960957.61	480.97	553.84	
2712798	1070376	16356914.19	1460.41	1567.23	
2840208	1207945	17274279.00	1846.90	1941.91	
3897636	1524453	24099527.87	2474.52	2566.45	
4657742	1890815	29235011.85	2816.62	2941.72	
6885658	2758119	44347858.43	4748.14	4732.52	
8778114	3598623	57550512.29	6776.84	6703.30	
15248146	6262104	103637383.09	12934.23	13076.86	
34292496	14081816	245145348.69	63508.33	61802.71	
58333344	23947347	430456760.80	84515.70	78654.26	



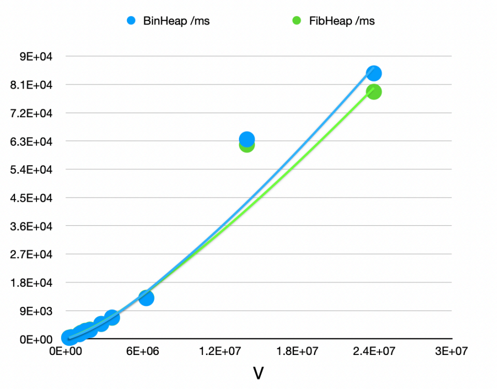


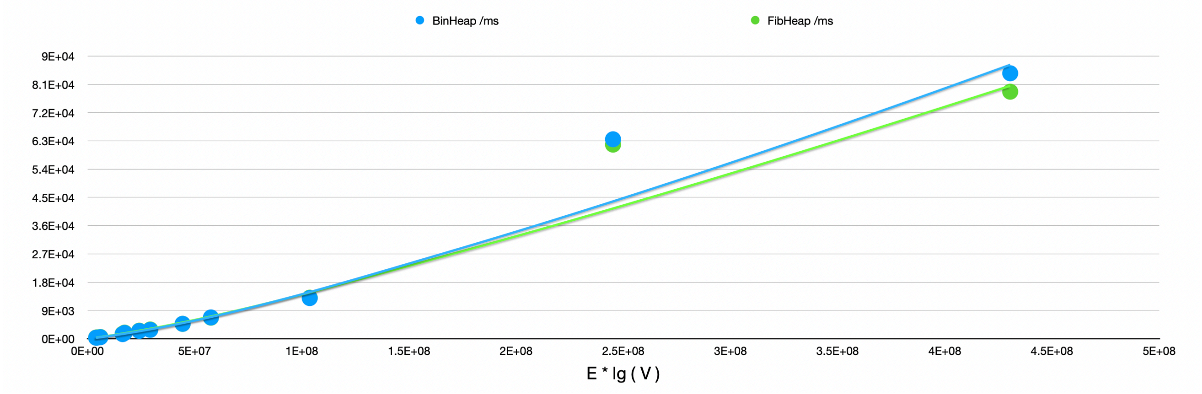
Fibonacci Heap vs. Binomial Heap

Running both implementations on all 12 files gives. In all diagrams below, **green** stands for **Fibonacci** Heap, while **blue** stands for **Binominal** Heap.

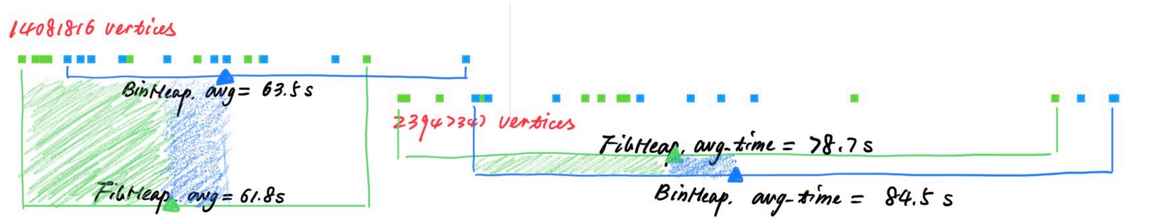
- General table, showing all 12 files. Basically, BinHeap runs faster when $V < 10000000$, and slower when not. Specially, the 14081816-vertex case is somewhat tough to explain.

arcs	vertice	E * log (V)	BinHeap /ms	FibHeap /ms	Ordinary /ms
733846	264346	3979039.78	315.82	367.51	318305.00
800172	321270	4406443.32	335.13	394.25	471121.00
1057066	435666	5960957.61	480.97	553.84	
2712798	1070376	16356914.19	1460.41	1567.23	
2840208	1207945	17274279.00	1846.90	1941.91	
3897636	1524453	24099527.87	2474.52	2566.45	
4657742	1890815	29235011.85	2816.62	2941.72	
6885658	2758119	44347858.43	4748.14	4732.52	
8778114	3598623	57550512.29	6776.84	6703.30	
15248146	6262104	103637383.09	12934.23	13076.86	
34292496	14081816	245145348.69	63508.33	61802.71	
58333344	23947347	430456760.80	84515.70	78654.26	

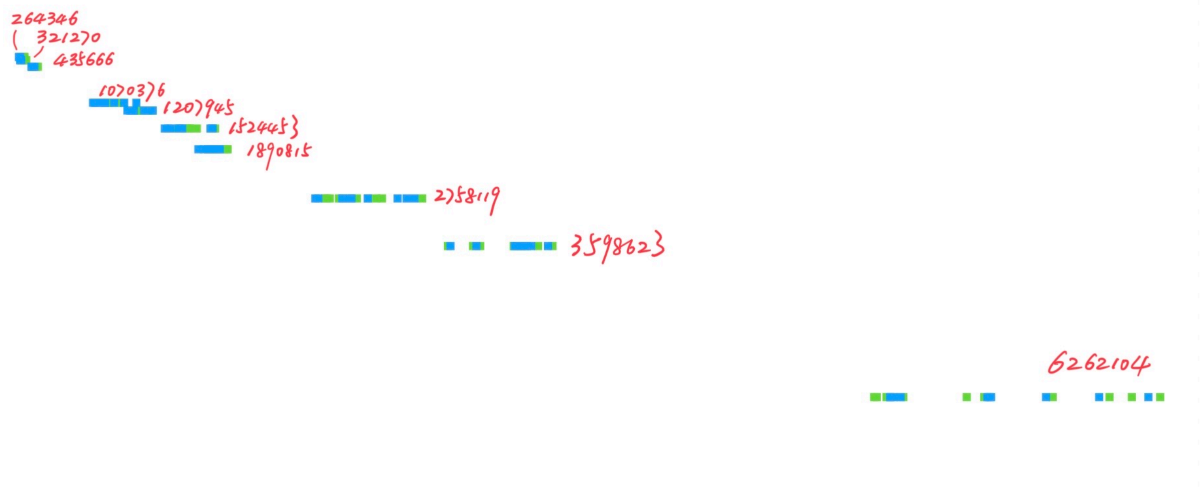




- The two largest graphs. In these cases, say over 10 million vertices, Fibonacci heap runs significantly faster.

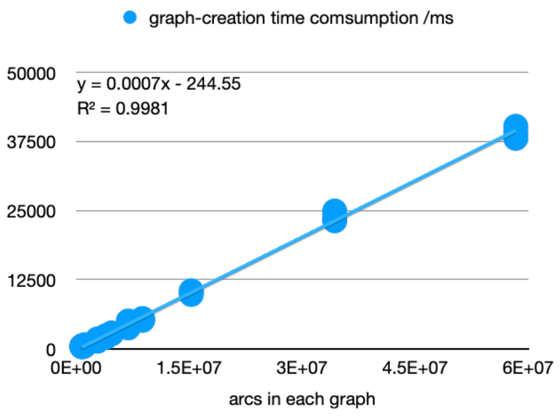


- All smaller cases. Binomial heaps runs faster.



Complexity

Graph Reading



Always **linear**, since with each edge added in, both space and time.

Ordinary Approach

1. All arcs are visited exactly once, resulting in $O(|E|)$;
2. Considering each vertex should be found as minimal-distance unvisited vertex, though each search is `std::map` based, still linear searching, thus needs $O(|V|)$, resulting in $O(|V|^2)$;
3. Since the diagram is quite scattered, $|V|$ and $|E|$ are in the same scale, so $O(|E|)$ will be eliminated, making $O(|V|^2)$ time complexity.
4. As for space, using adjacency list and a map, resulting in $O(|E|+|V|)$

Binomial Heap Approach

1. All arcs are visited exactly once, resulting in $O(|E|)$;
2. Considering each vertex should be found as minimal-distance unvisited vertex, each search is `std::map` based, needs $O(log|V|)$, resulting in $O(|V|log|V|)$;
3. DecreaseKey costs $O(log|V|)$ each, making $O(|E|log|V|)$;
4. Making $O(|V|log|V|+|E|log|V|)$ time complexity.
5. As for space, using adjacency list and a map, resulting in $O(|E|+|V|)$

Fibonacci Heap Approach

1. All arcs are visited exactly once, resulting in $O(|E|)$;
2. Considering each vertex should be found as minimal-distance unvisited vertex, each search is `std::unordered_map` based, needs average time complexity of $O(1)$, and deleteMin costs $O(log|V|)$ on average, decreaseKey nees $O(1)$, resulting in $O(|V|log|V|)$;
3. Time complexity is $O(|E|+|V|log|V|)$;
4. As for space, using adjacency list and a map, resulting in $O(|E|+|V|)$.