

Implementation documentation

This document studies the *data structures* and *algorithms* used in the program, reports their *time* and *memory requirements* as for their current *implementations*. *Flaws* of the implementation are also mentioned.

Queue

Implemented methods: *enqueue*, *dequeue*, *peek*, *clear*, *isEmpty*, *size*, *toLinkedList*

All methods for this data structure only perform *simple value assignments*, some of them with *if-checks*. That way all Queue's actions are done in *constant time* and with *constant memory*. *enqueue* creates a new object, which still requires constant memory. *toLinkedList* method is an exception, as its time and memory requirement is $O(n)$, where n is amount of *elements* in the Queue. Besides the elements, Queue only needs to store a constant amount of variables. So, overall, memory requirement for the Queue is $O(n)$.

LinkedList

Implemented methods: *add*, *clear*, *isEmpty*, *reset*, *hasNext*, *getNext*, *size*

All methods for this data structure only perform simple value assignments, some of them with if-checks. That way all LinkedList's actions are done in constant time and with constant memory. *add* creates a new object, which still requires constant memory. Besides the elements, LinkedList only needs to store a constant amount of variables. So, overall, memory requirement for the LinkedList is $O(n)$ where n is amount of elements in the LinkedList. *remove* method was not implemented, as the use of LinkedList in this project didn't need to remove specific elements on the LinkedList. Because the method would require $O(n)$ time to find the element and its predecessor as elements are *linked* in one way, it would be better to use a Tree if removing elements is required.

Tree

Implemented methods: *add*, *remove*, *getMin*, *clear*, *isEmpty*, *contains*, *size*, *toLinkedList*

Implemented Tree is *self-balancing* using the *red-black principle*. Implementation mimics the guidelines presented in the Wikipedia article: http://en.wikipedia.org/wiki/Red-black_tree Custom *comparator* class can be used to compare elements in a desired way.

add: First, tree finds a suitable *placement* for the new element. In worst case scenario, the placement in question may be found at the bottom of the tree. Because tree's height is below $2 * \log(n)$ (n = amount of elements), finding the placement requires $O(\log(n))$ time. After that several tree *balancing techniques* are used depending on a specific *scenario*. Most techniques require constant time, however one scenario requires to iterate the balancing techniques for added *node's parent*. Balancing on addition requires $O(\log(n))$ time. Overall, time requirement is $O(\log(n))$. Memory requirement is constant.

remove: Deleted element first must be *found*, which requires $O(\log(n))$ time. Some balancing techniques are performed in constant time, others *iterate upwards*, requiring $O(\log(n))$ time. Overall, time requirement is $O(\log(n))$. Memory requirement is constant.

contains: Required element may be a *leaf* of the Tree, in which case finding it requires $O(\log(n))$ time. Same time is required, if Tree doesn't contain given element. Overall, time requirement is $O(\log(n))$. Memory requirement is constant.

toLinkedList: All elements are traversed *breadth first* using the Queue. Each element is processed once, so required time is $O(n)$. Elements need to be stored in Queue and the LinkedList, which the method will return. Overall memory requirement is $O(n)$

getMin, *clear*, *isEmpty* and *size* perform at constant time and memory. Besides the elements, Tree only needs to store a constant amount of variables, making overall memory requirement $O(n)$.

Tree can be extended to act as a *map* that holds *key-value pairs*. Overall memory requirement is not changed in *O-notation*.

get: finds the *key* element and returns its *value*. Time requirement $O(\log(n))$, memory requirement constant.

put: Calls *contains* method. If element is contained, elements value is changed. If not, *add* is called to add a new element to the tree. Time requirement $O(\log(n))$, memory requirement constant.

There is a flaw in the implementation. Suppose we have elements *a* and *b*. If Tree's comparator's values for both comparisons are on the same side from zero ($\text{compare}(a,b) * \text{compare}(b,a) \geq 0$), these elements cannot be *differentiated*. This may lead to many unexpected results. If comparator classes have to be customly implemented, compare method shouldn't return zero, unless programmer is willing to assume that if $\text{compare}(a,b) = 0$, then *a* and *b* are considered to be the *same* element.

Graph

While Graph is not explicitly coded in the program, Graph is implemented with Vertex objects and VertexContainer, which contains all created vertices. Vertex keeps its adjacent vertices in a Tree, and VertexContainer keeps vertices in a Tree. VertexContainer stores *v* amount of memory and sum of all stored adjacent vertices for every vertex is *e*, where *v* is amount of vertices and *e* is amount of edges. Overall memory requirement $O(v+e)$.

Heap

Implemented methods: *insert*, *pop*, *peek*, *clear*, *isEmpty*, *size*, *valueChanged*. Heap in question is a *binary tree*, which stores elements in an *array*. Custom comparator class can be used to compare elements in a desired way. Heap can be paired with a TreeMap, which returns given element's *index value* in heap's array.

insert: Added element's *location* in the Heap is found, which takes $O(\log(n))$ time. Element is placed to the end of the array, after which it is *swapped* with the *parent* if Heap rules are violated. On each swap, TreeMap's value is changed, which also takes $O(\log(n))$ time. When placement is found, TreeMap's value is changed one more time. Overall, time requirement with a TreeMap is $O(\log^2(n))$, without it it's $O(\log(n))$. Memory requirement is $O(1)$. However, because Heap stores its data in a *finite* array, if array is *overflowed*, elements must be copied into a bigger array (this implementation doubles the size of the array). Increasing array takes $O(n)$ time and memory. However, the array is increased at exceptional situations. So, overall, time

requirement is $O(\log^2(n))$ and memory requirement is $O(1)$.

pop: First element in the Heap is replaced by the last element, which is then swapped with one of the childs if heap rules are violated. This takes $O(\log(n))$ time without TreeMap, with it it's $O(\log^2(n))$ as TreeMap is updated on every swap. Memory requirement is $O(1)$.

valueChanged: First, element's index is searched from the TreeMap, which takes $O(\log(n))$ time. Then, element is moved up or down on the heap at a correct position in the heap. On each position swap, TreeMap is updated. Overall, time requirement is $O(\log^2(n))$ and memory requirement is $O(1)$.

peek, *clear*, *isEmpty* and *size* perform at constant time and memory. Heap allocates memory for elements in the array and TreeMap. Heap also has constant amount of variables. Overall, memory requirement for the Heap is $O(n)$.

Heap has same flaws with comparators as with the Tree. When pairing Heap with TreeMap, each data structure should use a different *comparator class*. The values that Heap's comparator *compares* can be assumed to change, while TreeMap's compared values must stay *fixed* throughout the use of the Heap.

Path finding with Dijkstra's algorithm

Algorithm's rough progression:

1. Algorithm places all vertices in a Heap and a TreeMap of *distance estimates*, where every estimate is MAXVALUE except for the *starting vertex*, whose estimate is 0. Heap's comparator accesses the TreeMap with distance estimates, that way vertex with smallest distance estimation is *on top*.
2. Vertex is *popped* from the Heap, all of its *adjacent* vertices are iterated, and their distances are re-estimated. Because comparator's *compare* method uses Treemap's *get* method, this increases Heap's *poll* method's time requirement to $O(\log^3(n))$. Now every adjacent vertex stores popped vertex as its previous point in the path.
3. Repeat 2 until Heap is empty. Now we have found all shortest paths from a given starting vertex. Overall time requirement is $O((v + e) * \log^3(v))$, where v is amount of vertices and e is amount of edges in the graph. Memory requirement is $O(v)$.

Implemented algorithm mimics A* by using a simple *heuristic* to check less edges in the graph. Heuristic in question is *chebyshev distance*. Vertices with smaller chebyshev distance are preferred to vertices with bigger chebyshev distance. Algorithm *terminates* itself when end vertex is popped from the heap.

Implemented algorithm requires more than $O((v + e) * \log(v))$ time, which is widely considered to be Dijkstra's algorithm's requirement. To achieve that time, TreeMap should be replaced with HashMap for Heap's index retrieval and keeping distance estimations. HashMap's *get* and *put* methods perform at constant time instead of logarithmic time.

Graph building with tracing

Tracing algorithm was largely underestimated in the requirements documentation. Algorithm's rough progression:

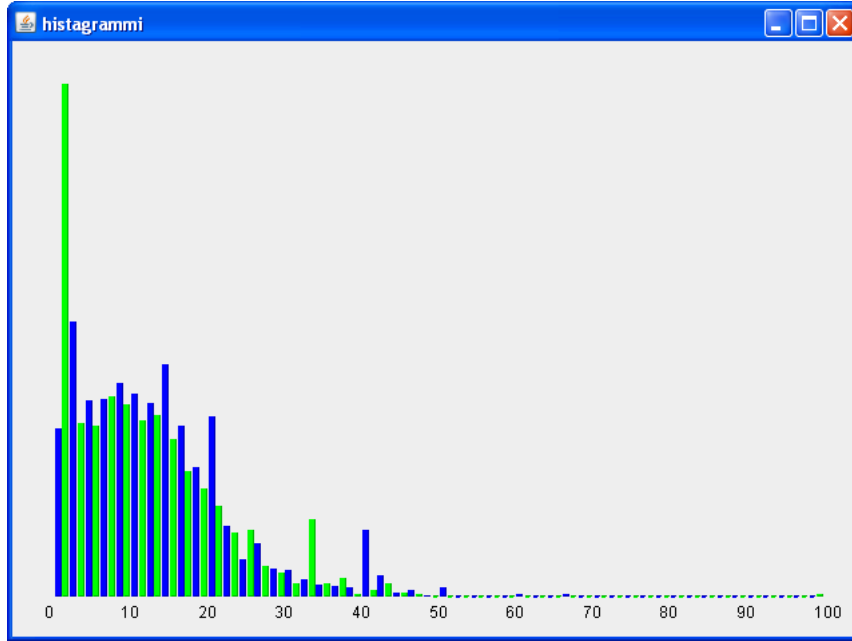
1. Choose a vertex v from which point of view other vertices are traced.
2. Choose another point p . Store the *direction* and *distance* from v to p . Also store direction and distance from v to $p.right$, assuming that p is a part of a *polygon* and thus has a connection with a *left and right neighbour*, all of which are a part of the same polygon. This data can be represented as a *sector* of two directions.
3. Repeat 2 for every *point* in the field. Sectors are stored in a Heap, where a sector with the smallest *left direction value* is placed on top of the Heap. Time requirement: $O(p * \log(p))$ Memory requirement: $O(p)$, where p is amount of geometry points.
4. Go through the sectors and remove *redundant and overlapping* sectors. Each stored sector is pulled from the Heap (Time: $O(p * \log(p))$) and stored in a Tree (Time: $O(p * \log(p))$), where sectors closest to the v are stored as a *left child*. Using this technique, amount of stored sectors is reduced significantly, where sector amount is s and $s \leq p$. Memory requirement: $O(p)$.
5. All vertices are checked whether they are *obstructed* by any sector. Time requirement: $O(r * s)$, where r is amount of points with *reflex angles* (which is less than or equals p) and s is amount of usable sectors (also less than p). Memory requirement: $O(q)$, where q is amount of points that are unobstructed to the v . That way $q \leq p$.
6. Repeat this for every v in the field.

Overall time requirement: $O(v * ((p * \log(p)) + (r * s)))$, where v is amount

of all vertices in the field, regardless if they are part of some polygon. That way $p \leq v$, also we have established that $r \leq p$ and $s \leq p$. So, in worst case scenario, we could argue that the time to rebuild the graph is:
 $O(v * (p * \log(p) + p^2)) = O(v^2 * \log(v) + v^3) = O(v^3)$.

In current implementation, it is fair to assume that $v = p$, because with currently implemented tools user can only place two vertices, which are not a part of any polygon. To assume that $r = p$ is a bit of a stretch, however that will happen if user decides to have all of the polygons to be *reflex*. The interesting bit is studying the relation between s and p .

Because $s \leq p$, we can represent the relation as: $s = c * p$, where $c \geq 0$ and $c \leq 1$. This histogram shows the value of c retrieved by *empirical* means.



This data was collected from 55672 occurrences. X-axis represents the value of c (in percents), and height of a column shows how many occurrences fit the particular *value interval*. We can see that c hardly reaches above 0.5 and has a distinct peak at $[0.1, 0.2]$. This suggests that equating s with p isn't entirely fair. Now time requirement looks like: $O(p^2 * (\log(p) + c))$

Memory requirement: $O(p)$. At any stage of the algorithm no more than p amount of memory allocation is required.

Built graph is redundant in many places, hurting the performance. However, the means to detect these redundancies may hurt the performance as badly. Perhaps a different approach to tracing would take less time in expense of more memory requirement. If field was divided into a *grid*, where each rect-

angle stores all lines that *intersect* it (allowing the same line to be stored in several rectangles), each trace would check appropriate areas of the field for intersections with other lines. Line candidates would be reduced, taking less time for tracing, however memory load would increase.