Name: Costiug Mihai

Group: 912

**DATA STRUCTURES AND ALGORITHMS**

**Project documentation**

1. **Problem statement**

Path in a maze (labyrinth)

A robot is asked to navigate a maze. It is placed at a certain position (the starting position: S) in the maze and is asked to try to reach another position (the goal position: G). Maze has rectangular shape. Positions are identified by (x, y) coordinates.

Positions in the maze will either be open (\*) or blocked with an obstacle (X).

At any given moment, the robot can only move 1 step in one of 4 directions. Valid moves are:

* Go North: (x, y)
* Go East: (x, y)
* Go South: (x, y)
* Go West: (x, y)

The robot can only move to positions without obstacles and must stay within the maze.

The robot should search for a path from the starting position (S) to the goal position (G) until it finds one or until it exhausts all possibilities.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| S | \* | X | X | \* | \* | \* |
| \* | X | \* | \* | X | \* | \* |
| \* | \* | \* | \* | \* | \* | \* |
| \* | X | \* | X | \* | \* | X |
| \* | X | \* | \* | \* | \* | X |
| \* | \* | \* | \* | X | \* | G |

Find optimum solution (shortest path).

1. **ADT: priority queue (domain & operations specification)**

Domain:

DTElem = { v| v – general element }

DPriority = { k | v – general element }

D<TElem, Priority>= { (k, v) | keys not necessarily unique} k ∊ DPriority, v ∊ DTElem

Operations:

**Subalg initEmpty()**

descr: initialize the empty queue

data: - prec: -

result:- post:-

**Subalg enqueue(Telem e)**

descr: adds a new element in the queue

data: e prec: e ∊ DTElem

result:- post:-

**Function Telem dequeue()**

descr: removes the element with the biggest priority from the queue and returns it

data: - prec: -

result: e, e ∊ DTElem post:- NULL. If prioriy queue is empty, print a message

**Subalg isEmpty() : bool**

descr: verifies if the queue is empty or not

data: - prec: -

result:bool post:-

**Function getSize()**

descr: returns the number of elements in the queue

data: - prec: -

result: integer post:-

1. **DS: dynamic linked list**

Here will be described only the nontrivial sub-algorithms. The others can be found in the folder PQOverDynamicList under the modules DynamicList.h

The opertions of the DS: dynamic linked list:

**Subalg initEmpty**()

complexity : O(1)

Desc: initialize the empty list

Pre: none

Post: none

{

q^.head = NULL;

q.nr = 0;

}

End \_iniEmpty

**Subalg isEmpty**()

complexity : O(1)

Desc: checks if the list is empty or not

Pre: none

Post: true if the list is not empty, false otherwise

{

**if** head = NIL then

**return** **true**;

end if

**return** **false**;

}

End \_isEmpty

**Subalg isFull**()

complexity : O(1)

Desc: checks whether the list is full or not

Pre: none

Post: true if the list is full, false otherwise

{

**if** q.nr > cap then

**return** **true**;

end if

**return** **false**;

}

End \_isFull

**Subalg enqueue**(**TE** el)

complexity : O(1)

Desc: adds a new element into the list

Pre: e is of type TE

the list cannot be empty

Post: none

{

**struct** node<**TE**> \*n;

n = **new** node<**TE**>;

n^.data = el;

n^.next = NULL;

**if** q^.head != NIL then

n^.next = q^.head;

end if

q^.head = n;

q.nr++;

}

End \_enqueue

**TE** **dequeue**()

complexity : O(1)

Desc: removes the element with the biggest priority and returns it

Pre: list cannot be empty

Post: e of type TE

{

**TE** a;

a = 0;

**if** isEmpty() == **true then**

{

cout << "Empty. Nothing to remove! " << **endl**;

**return** a;

}

end if

**struct** node<**TE**> \*temp;

temp = q^.head;

a = top();

q^.head = temp^.next;

**delete** temp;

nr--;

**return** a;

}

End \_dequeue

**TE** **top**()

complexity : O(1)

Desc: returns the element at the top of the list (the head)

Pre: list cannot be empty

Post: e of type TE

{

**TE** a;

a = q^.head^.data;

**return** a;

}

End \_top

**Subalg printEls**()

complexity : O(n)

Desc: displays the list of elements

Pre: none

Post: none

{

**struct** node<**TE**> \*ptr = q^.head;

**while**(ptr != NULL)

{

cout << ptr->data<<" -> ";

ptr = ptr->next;

}

end while

cout << " NIL " << **endl**;

}

End \_printELs

**Subalg destroyList**()

Desc: removes all the elements from the list

Pre: none

Post: none

{

**while** head != NIL then

{

**struct** node<**TE**>\* temp;

temp = q^.head;

q^.head = temp^.next;

**delete** temp;

}

end while

**delete** q^.head;

}

End \_destroy

The implementation of the priority queue over the list:

**Subalg initEmpty**()

Desc: initialize the empty queue

Pre: none

Post: none

{

l.initEmpty();

}

End \_initEmpty

**Subalg isEmpty**()

Desc: checks if the queue is empty or not

Pre: none

Post: true if the list is not empty, false otherwise

{

**return** l.isEmpty();

}

End \_isEmpty

**Subalg isFull**()

Desc: checks whether the queue is full or not

Pre: none

Post: true if the list is full, false otherwise

{

**return** l.isFull();

}

End \_isFull

**Subalg enqueue**(**TE** el)

Desc: adds a new element into the queue

Pre: e is of type TE

Post: none

{

**if**(l.isFull() == **true**) then

cout << "Cannot add into queue. The queue is full. "<< **endl**;

**else**

{

l.enqueue(el);

}

End if

}

End \_enqueue

**TE dequeue**()

Desc: removes the element with the biggest priority and returns it

Pre: none

Post: e of type TE

{

**return** l.dequeue();

}

End \_dequeue

**TE peek**()

{

**return** l.top();

}

End \_peek

**Subalg** **destroy**()

Desc: removes all the elements from the queue

Pre: none

Post: none

{

l.destroyList();

}

End \_destroy

1. **DS: heap**

I will use heap to build a priority queue. My implementation is based on the fact that integer having higher value gets preference. I will also use vector container to store items of the heap to ensure that memory goes dynamically. I am using a vector of integer as items in the heap. I will define the following functions in the priority queue:   
1. Inserting into the queue. The strategy is to insert after the rightmost element. This might break the heap property and so will re-heap if the inserted element is not on the correct position.  
2. Getting the max element in the heap. This will create a hole. We are going to put rightmost element of the heap to fill the hole and re-heap it again to ensure that heap properties are met.

**Subalg enqueue**(**int** item)

complexity : O(logn)

Desc: adds a new element into the queue

Pre: item is of type TE (here of type int)

Post: none

{

pq\_keys.push\_back(item);

shiftLeft(0, pq\_keys.size() - 1);

**return**;

}

End \_enqueue

**Function dequeue**()

complexity : O(logn)

Desc: removes the element with the biggest priority from the queue

Pre: none

Post: returns tmp of type TE (here of type int)

{

assert(pq\_keys.size() != 0);

**int** last = pq\_keys.size() - 1;

**int** tmp = pq\_keys[0];

pq\_keys[0] = pq\_keys[last];

pq\_keys[last] = tmp;

pq\_keys.pop\_back();

shiftRight(0, last-1);

**return** tmp;

}

End \_dequeue

**Subalg shiftLeft**(**int** low, **int** high)

complexity : O(logn)

Desc: shifts the elements to the left if the nodes break the heap’s properties

Pre: none

Post: none

{

**int** childIdx = high;

**while** (childIdx < low)

{

**int** parentIdx = (childIdx-1)/2;

/\*if child is bigger than parent we need to swap\*/

**if** (pq\_keys[childIdx] > pq\_keys[parentIdx])

{

**int** tmp = pq\_keys[childIdx];

pq\_keys[childIdx] = pq\_keys[parentIdx];

pq\_keys[parentIdx] = tmp;

/\*Make parent index the child and shift towards left\*/

childIdx = parentIdx;

}

**else**

{

**break**;

}

}

**return**;

}

End \_shiftLeft

**Subalg shiftRight**(**int** low, **int** high)

complexity : O(logn)

Desc: shifts the elements to the right if the nodes break the heap’s properties

Pre: none

Post: none

{

**int** root = low;

**while** ((root\*2)+1 >= high)

{

**int** leftChild = (root \* 2) + 1;

**int** rightChild = leftChild + 1;

**int** swapIdx = root;

/\*Check if root is less than left child\*/

**if** (pq\_keys[swapIdx] < pq\_keys[leftChild])

{

swapIdx = leftChild;

}

/\*If right child exists check if it is less than current root\*/

**if** ((rightChild <= high) && (pq\_keys[swapIdx] < pq\_keys[rightChild]))

{

swapIdx = rightChild;

}

/\*Make the biggest element of root, left and right child the root\*/

**if** (swapIdx != root)

{

**int** tmp = pq\_keys[root];

pq\_keys[root] = pq\_keys[swapIdx];

pq\_keys[swapIdx] = tmp;

/\*Keep shifting right and ensure that swapIdx satisfies

heap property aka left and right child of it is smaller than

itself\*/

root = swapIdx;

}

**else**

{

**break**;

}

}

**return**;

}

End \_shiftRight

**Subalg buildHeap**()

complexity : O(logn)

Desc: acts like a constructor for the heap

Pre: none

Post: none

{

/\*Start with middle element. Middle element is chosen in

such a way that the last element of array is either its

left child or right child\*/

**int** size = pq\_keys.size();

**int** midIdx = (size -2)/2;

**while** (midIdx >= 0)

{

shiftRight(midIdx, size-1);

--midIdx;

}

**return**;

}

End \_buildHeap

1. **Interface**

![](data:None;base64,)

**Present how you place code into the files (short indication of the content of each file):**

The program is structured in the files:

* “DynamicList.h” which contains the class linkedList, its operations and also the implementation of the node;
* “PQOverList.h” which contains the class PQueue with the specific operations of a priority queue implemented over a dymanic linked list;
* In “PQOverHeap” I defined a class: PriorityQueue where all specific operations of a priority queue are implemented such that they respect the structure of heap (which is like a binary tree);
* In “Maze.h” we have the implementation of the solution of the problem using the Backtracking Method
* In “app.cpp” we have the main program with the interface of the menu and is the file where we include all the other files from our program

1. **Test program**

While writing the source code, the program has been tested aagain and again for working properly.

**void** **testAll**()

{

cout<<" Testing the list "<<**endl**;

linkedList<**int**> l;

l.enqueue(4);

l.enqueue(5);

assert(l.isEmpty() == **false**);

assert(l.dequeue() == 5);

cout<<" Testing the heap "<<**endl**;

PriorityQueue q;

q.enqueue(5);

q.enqueue(4);

q.enqueue(1);

q.enqueue(3);

assert(q.getSize() == 4);

assert(q.dequeue() == 5);

}

We will give a few examples:

If the inserted numbers are 4, 5 and 6 then the enqueue method will work as following:

- first inserts 4

- insert 5

- 5 will be inserted first

- insert 6

- 6 will be inserted in front of 5

When we remove numbers, the first one in is the first one out. In the example above, the first element removed will be 4, followed by 5 and then 6.

1. **Solution of the problem**

**1) - present the idea;**

**2) - justify why the program should work correctly; apply justification to some appropriate examples;**

**3) -if appropriate: present (maybe not standard) pseudocode for main algorithm**

First we read from the text file “maze.in”, the number of rows and columns of the maze (having the structure of matrix), then we construct the maze by reading form the file each character and place it in the matrix in the right place. Finally we read the initial and the end position of the robot. With all this data we can search for the shortest path in the maze. The algorithm works like this: if the position is a good one(there is no wall), save it and when we get at the end point we verify if it’s indeed the best one (the shortest); if not we keep searching for a suitable solution.

Main algorithm for solving the problem:

/\*

Checking whether the robot is still in the maze.

Returns true if it is, false otherwise.

\*/

**bool** **Maze::isInside**(**int** i,**int** j)

{

**if**(i >= 1 && i <= r && j >= 1 && j <= c)

**return** **true**;

**return** **false**;

}

/\*

Stores solutions into the queus q

The solutions will be considered as pairs of 2 integers, the 2 coordinates.

\*/

**void** **Maze::keep**(**int** steps)

{

**if**(steps < min)

{

min = steps;

**for**(i = 1; i <= steps; i++)

{

// Keeping the coordinates in the queue

q.enqueue(coord[i][1]);

q.enqueue(coord[i][2]);

}

}

}

**void** **Maze::back**(**int** currentx,**int** currenty, **int** steps)

{

**int** i, j,k;

**if**(currentx == endi && currenty == endj)

keep(steps-1);

**else**

**for**(k = 0; k <= 3; k++)

{

i = currentx + movei[k];

j = currenty + movej[k];

**if**((isInside(i,j)==**true**) && ((matrix[i][j] == '\*') || (matrix[i][j] == 'S') || (matrix[i][j] == 's') || (matrix[i][j] == 'G') || (matrix[i][j] == 'g')))

{

matrix[i][j] = steps;

coord[steps][1] = i;

coord[steps][2] = j;

back(i,j,steps+1);

matrix[i][j] = 0;

}

}

}

1. **Suitable DS**

The suitable data structure is the heap because using a linked list we can keep the list sorted (which makes insert O(N)) or unsorted (which makes removeMax O(N)). Note that if the list is sorted we must use linear search to find the place to insert the new value, but there is no need to move old values over. Also, unless we maintain a "tail" pointer, we should keep the list in order from high to low, since removing from the front of the list can be done in O(1) time.

1. **Execution Time**

Execution time for this algorithm is 0.01 seconds. This depends on the machine running it.