Search Tree

Can Eldem

September 23, 2013

Abstract

Purpose of this project is find paths for obstacle navigation using uninformed and informed symbolic search.Various type of search methods are implemented such as Breadth-first implementation,Depth-first,Uniform Cost,Greedy best-first,A*.

Contents

Part I

INRODUCTION

Purpose of this practical is find paths for obstacle navigation using uninformed and informed symbolic search.

Part II

BREADTH-FIRST DESIGN DESCRIPTION

Russell and Norvig describes Breadth-first search mechanism as a simple strategy in which the root node is expanded first, then all the BFS successors of the root node are expanded next, then their successors, and so on. [1]All nodes are expanded in a depth before moving to next level as it is shown in figure-1.

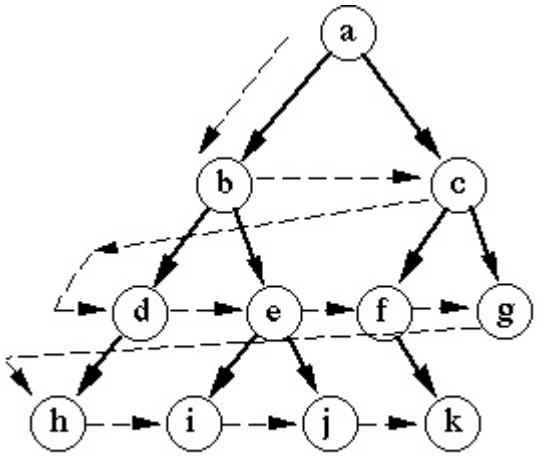

Figure 1: breadth first search

In order to implement Breadth-first FIFO (first in first out) structure should be used .This structure enables shallower nodes to expend first before moving on to the second level in the tree .In each step current state and goal state should be compared to decide whether to continue to next level or not .Breadth-first search should be complete because untill it expands all nodes until reaches its target state . Complexity of Breadth-first algorithm is O(bd) b represents nodes

and d represents depth .This shows that BFS displays the best performance with minimum depth. [1]

Graph object is designed for this project to hold all information about nodes and tree structure in arraysList. In this case, BFS will search vertices with given vertex number on graph and the goal state will be desired vertex with labeled number.BFS search is implemented as it is describe by Russell and Norvig . Queue structure used to meet requirement of FIFO and states are check with whether the current state is the same as the goal state .When the desired state is the same as the current state the algorithm terminates, otherwise the algorithm explores all other nodes and terminates.

Part III

BREADTH-FIRST IMPLEMENTATION

BFS algorithm is represented as an class in handed project. When the BFS algorithm is requires to run, it is decided by General Search method (triggered from GUI).General Search passes all required information such as tree structure from Graph object and sends to BFS class. Then , BFS runs breadthFirstTraversal() method which gets the starting vertex and the target vertex and starts to search described in previous part and labels visited nodes (is_discovered=true). When the goal state is same as the current state BFS runs recursive backtrace() algorithm and displays the path to user in GUI .In addition to backtrace, BFS calculates information about explored nodes and total cost(each step represents one cost). Meanwhile GS calculates time to represent efficiency in GUI screen .

Part IV
I/O FOR BREADTH- FIRST IMPLEMENTATION

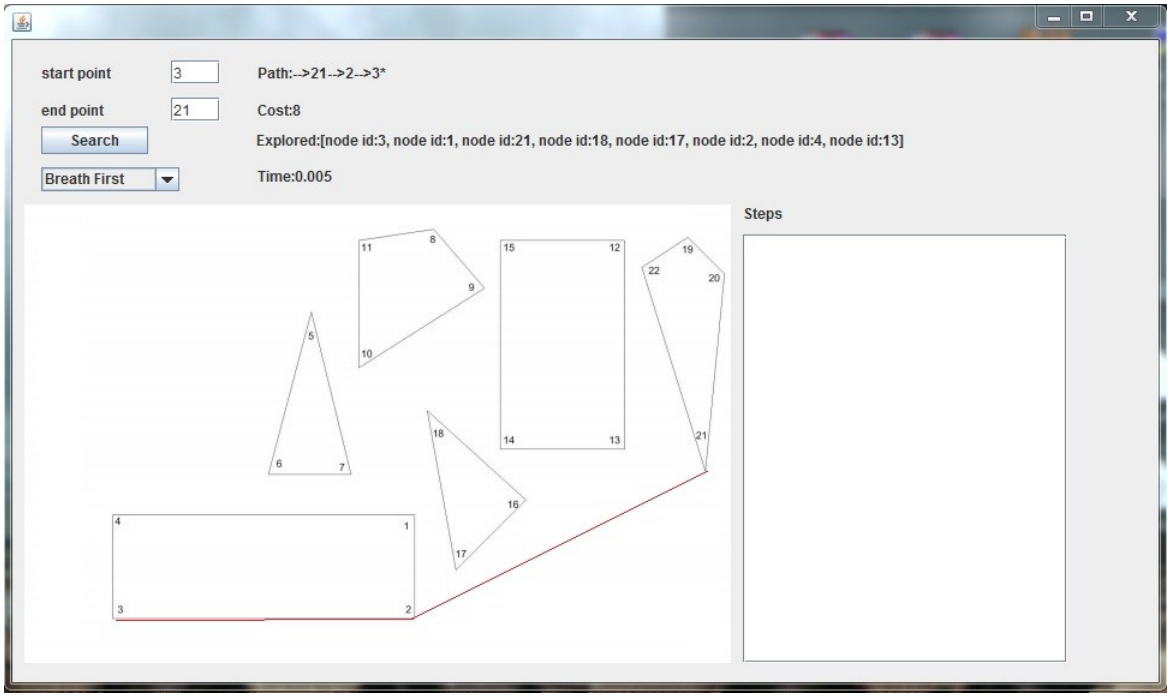

Figure 2:

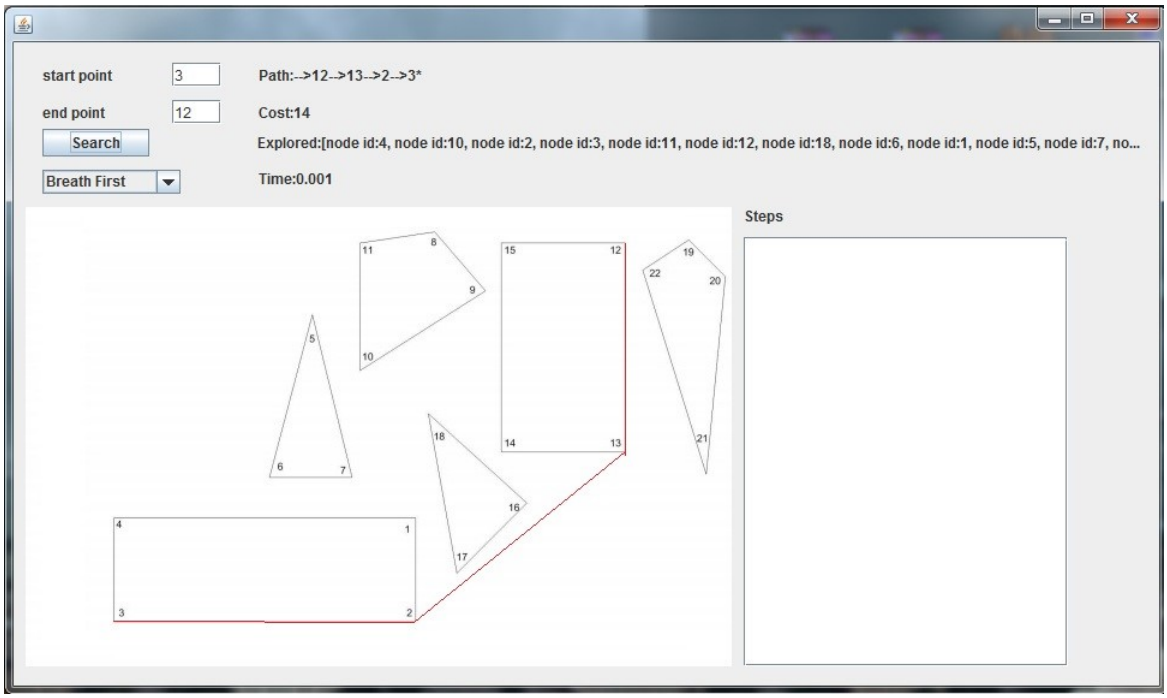
start point
3
end point
12
Search
Breath First
Path:-->12-->13-->2-->3*
Cost:14
Explored:[node id:4, node id:10, node id:2, node id:3, node id:11, node id:12, node id:18, node id:6, node id:1, node id:5, node id:7, no...
Time:0.001
Steps
11
8
9
10
5
6
7
15
12
14
13
18
16
17
4
1
3
2
19
22
20
21

Figure 3:

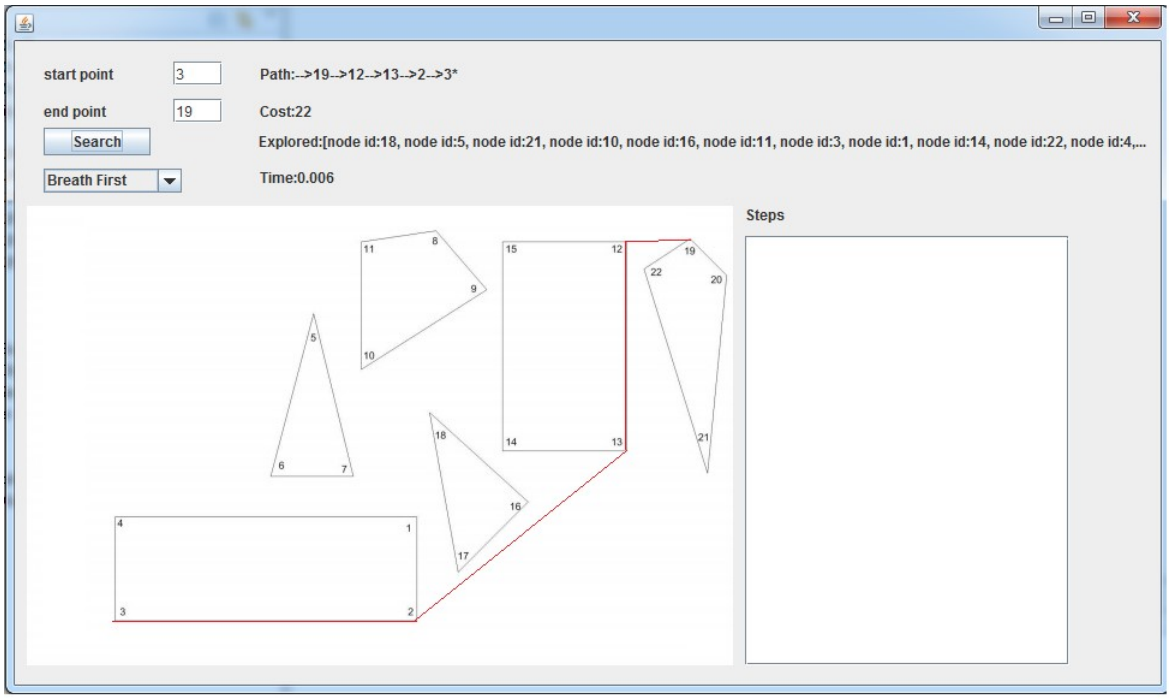

Figure 4:

Part V
PROLOG

Prolog is a logic programming language associated with Artificial Intelligence. Unlike Java , Prolog is not a procedural programming language .The Prolog program specify relationships among objects and it saves these relationships in a database file. [2] e.g "john own the book" means the relationship between john and book objects will be declared. Prolog has different capabilities than java . For instance , Prolog uses term(atom,number,variable) data type and recursion function for loops. Thus , methods written in prolog can't return objects like in Java . The Prolog language is convenient solving problems with logic because it enables user to easily create states with predicates .Therefore the user can model problems with less lines of codes to reach a solution compared with java .

```
e.g solution for depth first search with prolog
dfs(S,[S]) :- goal(S).
dfs(S,[S|Rest]) :- arc(S,S2), dfs(S2,Rest).
```

:- symbol shows true/false if true/false s is a statement , description of a program [] represents list in first line of and algorithm recursive method gets statement as an input and checks whether it is corresponds goal statement or not , if it is not corresponds this statement it links first statement with second statement and call recursive method again. Comparing with java implementation of same algorithm prolog is fairly cleaner and faster because all methods return true or false statements, iterations done recursively. Secondly, java implementation, more complicated data structures are used in order to define tree structure instead of predicates and logical definitions in prolog .Thus ,there is less recursive use in java implementation .

In my opinion, simple problems which are capable of being separated according to Kowalski's equation (algorithm = logic + control or algorithm = Prolog DB + Prolog interpreter)[3] with logical states should be implemented using Prolog programming language .Because prolog language is easy to grasp and faster than java language. However, for complex systems prolog is not an convenient language because of compatibility problems between the module systems of the major Prolog compilers. [3] Because of time constrain and lack of knowledge about Prolog.

Part VI

DFS (DEPTH FIRST SEARCH)

Depth first search expands the deepest node of current frontier of the search tree and checks, in every step whether current state fits with the solution state .If it does not DFS chooses other undiscovered nodes and repeats the same process .Instead of using FIFO like in BFS DFS uses LIFO (Last in First Out) structure because LIFO gives priority to discover first branch up to deepest level. Figure x shows how DFS works .The disadvantage of depth first search is that, if it may not terminate on an infinite tree, and it might be too costly if solution that we search is not in first branch . [1]Time complexity for DFS is $O(b^m)$ m represent max level of any node.[1]For instance in figure x imagine that left branch of root level is 100 if solution is k DFS expands all other 100 level in left side and this will increase cost dramatically.

In order to simulate LIFO condition, stack collection is used in java project and DFS is implemented as it is described in above . Method runs recursively, and program terminates when goal statement is reached or tree is discovered. However, there is not any further developments in implementation to solve problems mentioned by Russel and Norvig for infinite loops.

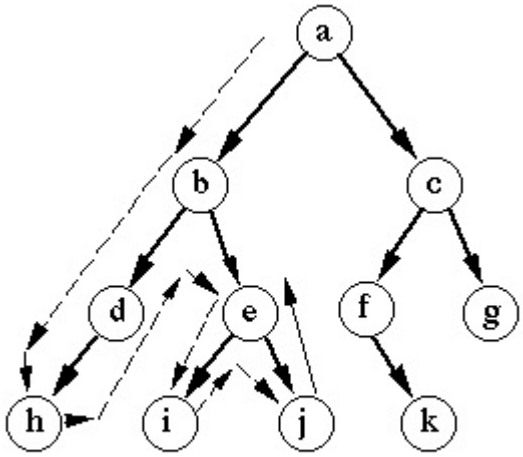

Figure 5:

Part VII

DEPTH-FIRST IMPLEMENTATION

DFS algorithm is represented as an class in handed project . When DFS algorithm is called by General Search method (triggered from GUI).General Search passes all required information such as tree structure from Graph object and sends to DFS class. Then , DFS runs depthFirst() method which gets the starting vertex and the target vertex and starts to search recursively and labels visited nodes as described in previous part . When the goal state is the same with current state DFS runs recursive backtrace() algorithm display path to user in GUI .In addition to backtrace DFS calculates information about explored nodes and total cost(each step represents one cost). Meanwhile GS calculates time to represent efficiency in GUI screen .

Part VIII
I/O FOR DFS IMPLEMENTATION

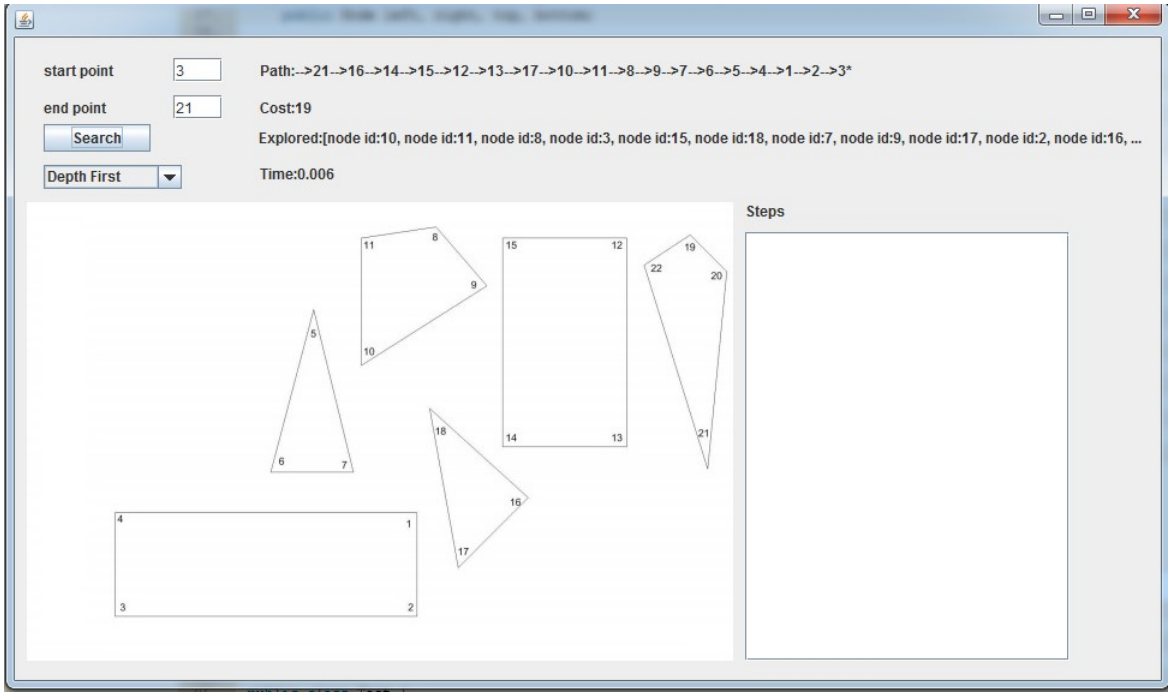

Figure 6: Difference between breath first search for same positions could be seen comparing figures

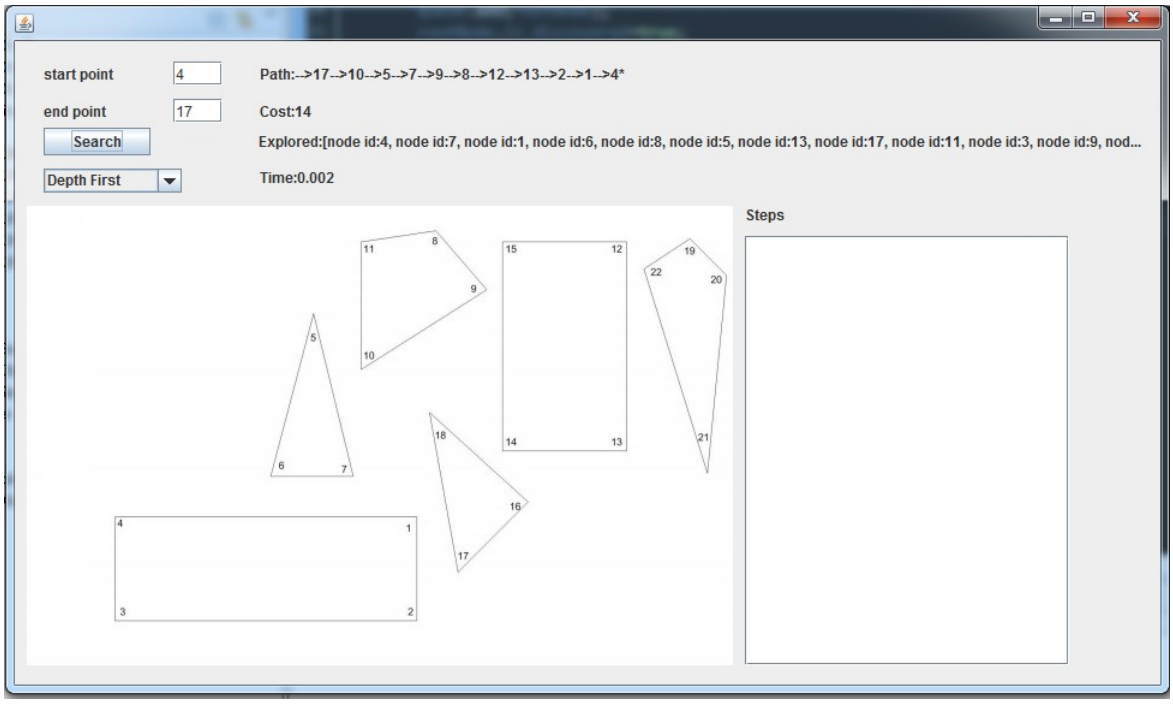
start point
4
end point
17
Search
Depth First
Path:-->17-->10-->5-->7-->9-->8-->12-->13-->2-->1-->4*
Cost:14
Explored:[node id:4, node id:7, node id:1, node id:6, node id:8, node id:5, node id:13, node id:17, node id:11, node id:3, node id:9, nod...
Time:0.002
Steps
11
8
9
10
5
6
7
15
12
14
13
19
22
20
21
18
16
17
4
1
3
2

Figure 7:

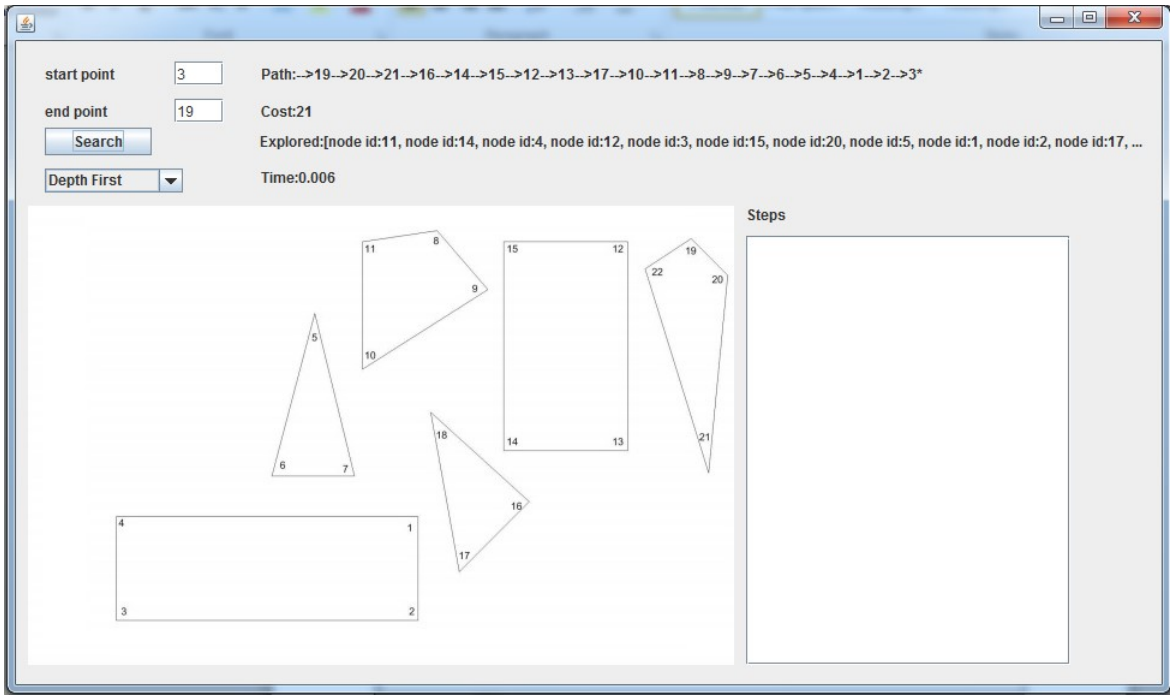

Figure 8:

Example runs shows that, since DFS explores up to the deepest point of one branch and move to another branch cost is variety depends on targeting point. If the target point is visible from first expanded branch cost is less than the if the target is in the last branch expanded.

Part IX
UNİFORM COST DESCRİPTİON

Uniform-cost might be considered as an extended version of Breath-First search .Instead of expanding the shallowest node in BFS , uniform-cost search expands the node with the lowest cost .In order to do this, uniform cost sorts frontier in every step according to cost constrain therefore, shallowest node becomes node with lowest cost . In addition, there are two other differences from BFS. First of all, Uniform costs checks current statement when selected node is expanded . Secondly, in case of better path found to discovered point, frontier is manipulated. Algorithms complexity is $O(b^{1+\lfloor c/€ \rfloor})$ [1] greater than breath first cost considering all sorting and looking alternative paths to goal .

Part X
UNIFORM COST IMPLEMENTATION

Figure-9 shows the example for how Uniform Cost algorithm is implemented. All algorithm taking takes its place in the UniformCost class .The algorithm first expands its neighbours and checks them for whether they are the goal statement or not . If they are not the goal statement, the cost is calculated (cost=current cost + cost for going that point) and put in custom priority queue. When priority queue have a new item, it checks all other items in the array and sorts them with bubble sort according to cost. The lowest cost will be the first item in array when process is complete. *insert (Node item,int type)* method gets two items, first item going to be inserted into the array and second parameter is a number tells sorting method to make bubble sorting considering uniform cost value because same method will be used for Greedy Best First method .Then the priority queue chooses the node with the the smallest cost, in this case it is going to be node A then Uniformcost algorithm checks whether this node seen before or not by checking value of *is_discovered* and *searchItem* method. If it is seen from other nodes before, and calculated cost is more than current cost, algorithm updates priority queue .The algorithm first chooses to direct A towards to C but if it does not update its priority queue, uniform cost can't be complete . When the algorithm sees Node C it calculates cost for other paths which sees node C and updates priority queue that going node C will be least costly path . In this case going C from B is least so priority queue will be updated .Once the goal statement is found , recursive *backtrace()* function called and return path is displayed .

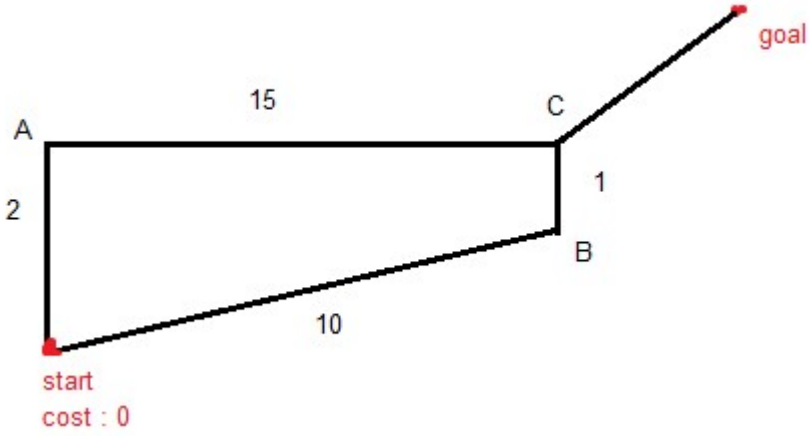

Figure 9: shows the implementation of the algorithm

Part XI
I/O FOR UNIFORM-COST IMPLEMENTATION

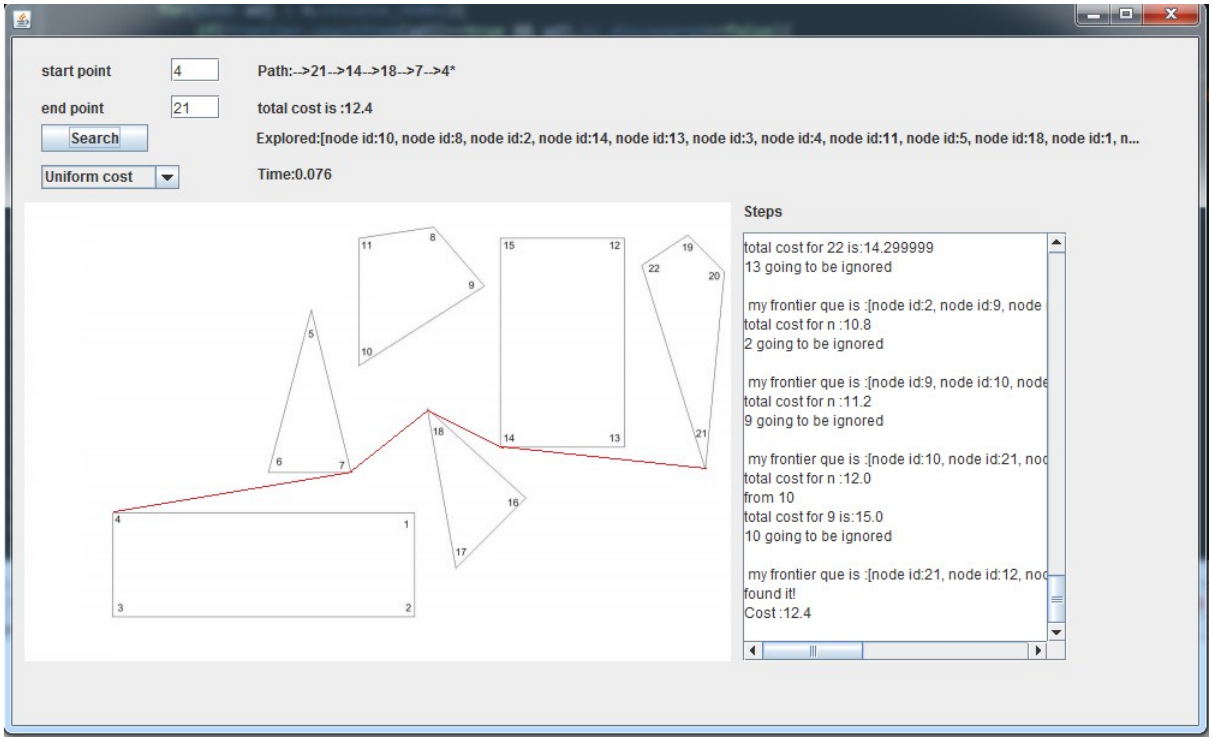

Figure 10:

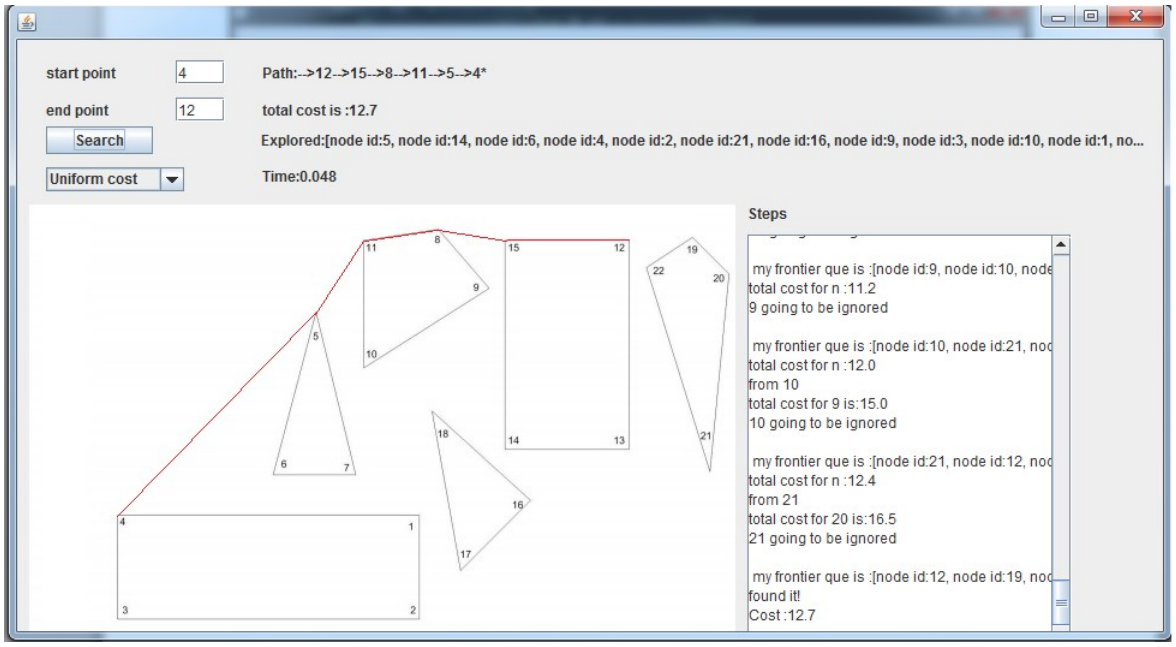
start point
4
end point
12
Search
Uniform cost
Path:-->12-->15-->8-->11-->5-->4*
total cost is :12.7
Explored:[node id:5, node id:14, node id:6, node id:4, node id:2, node id:21, node id:16, node id:9, node id:3, node id:10, node id:1, no...
Time:0.048
Steps
my frontier que is :[node id:9, node id:10, node
total cost for n :11.2
9 going to be ignored
my frontier que is :[node id:10, node id:21, noc
total cost for n :12.0
from 10
total cost for 9 is:15.0
10 going to be ignored
my frontier que is :[node id:21, node id:12, noc
total cost for n :12.4
from 21
total cost for 20 is:16.5
21 going to be ignored
my frontier que is :[node id:12, node id:19, noc
found it!
Cost :12.7

Figure 11:

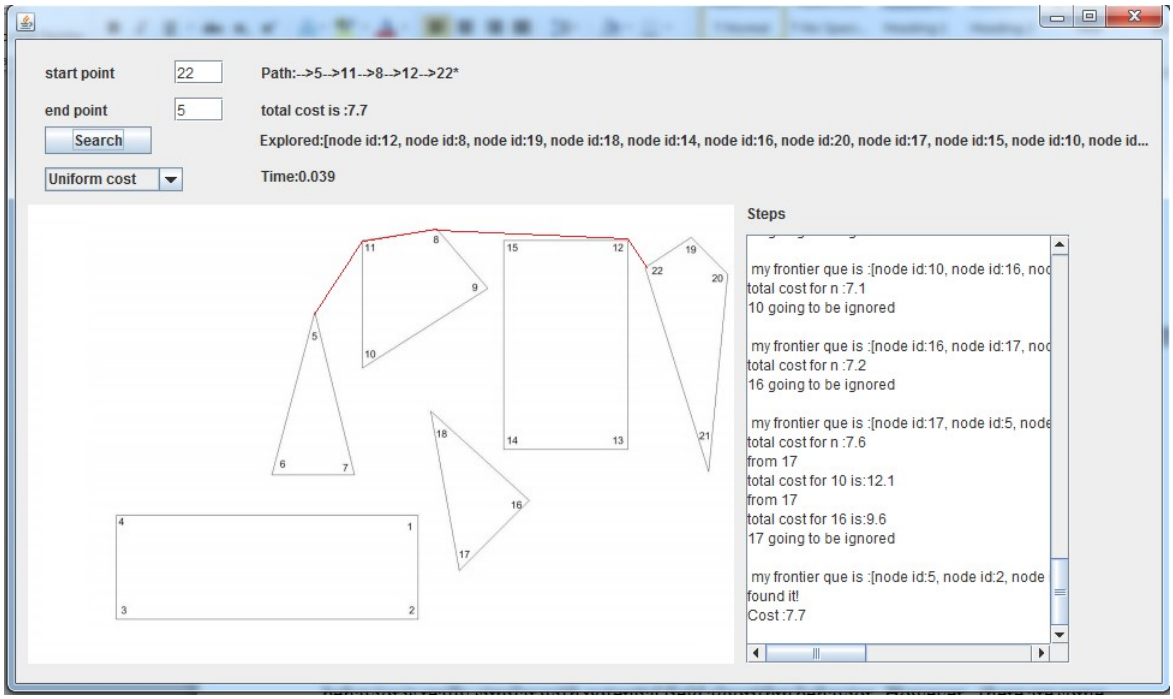
start point
22
end point
5
Search
Uniform cost
Path:-->5-->11-->8-->12-->22*
total cost is :7.7
Explored:[node id:12, node id:8, node id:19, node id:18, node id:14, node id:16, node id:20, node id:17, node id:15, node id:10, node id...
Time:0.039
Steps
my frontier que is :[node id:10, node id:16, noc
total cost for n :7.1
10 going to be ignored
my frontier que is :[node id:16, node id:17, noc
total cost for n :7.2
16 going to be ignored
my frontier que is :[node id:17, node id:5, node
total cost for n :7.6
from 17
total cost for 10 is:12.1
from 17
total cost for 16 is:9.6
17 going to be ignored
my frontier que is :[node id:5, node id:2, node
found it!
Cost :7.7

Figure 12:

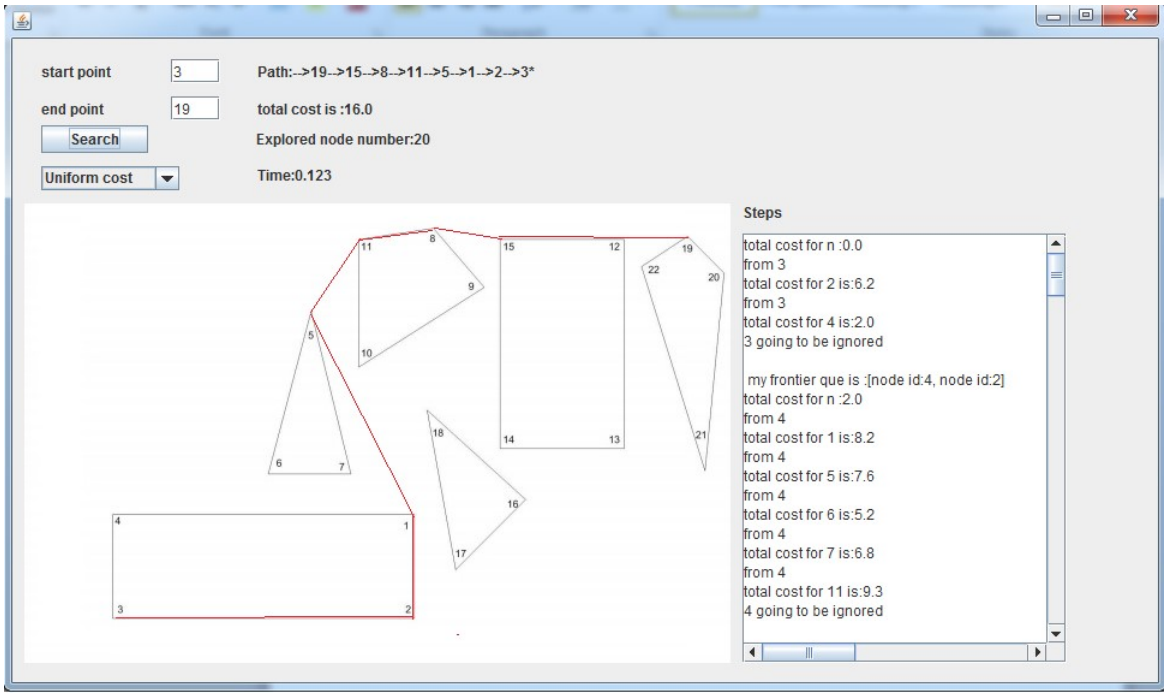

Figure 13:

Part XII
GREEDY BEST-FIRST DESCRIPTION

GBF search tries to expand the node that is closest to the goal or in other words GBF tries to expand the node which has the smallest heuristics value .In this implementation heuristics value is distance between a node and goal .Therefore, GBF tries to move towards goal. GBF behavior is really similar with potential field algorithm behavior. However , there are some issues with GBF like Amit PhD student in Stanford indicates , "The trouble is that Greedy Best-First-Search is "greedy" and tries to move towards the goal even if it's not the right path. " GBF ignores other alternative paths which are costly in beginning but less costly in total. Figure- shows the problem with GBF clearly.

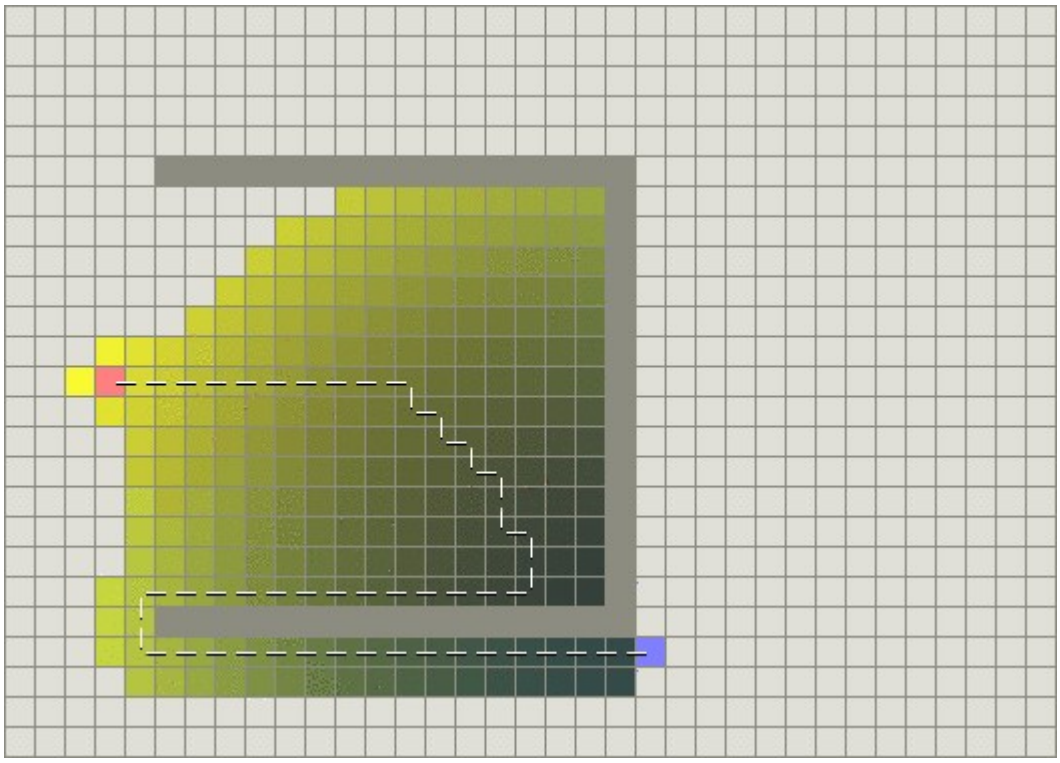

Figure 14: http://theory.stanford.edu/

Part XIII

GREEDY BEST-FIRST IMPLEMENTATION

When user wants to search a value, heuristics for all nodes are set based to target with g.setHeuristics(target) method.Heuristics for this project is distance between node and target . Then, greedy first explorers its neighbors and check whether they are goal statement or not. If current statement is goal statement, than recursive backtrace() method works and gives path too user .Otherwise, they are sorted with bubble sort algorithm and put into array with "frontier.insert(adj,2)" method .This iteration continues until end of tree or goal statement found .However , since the nature of GBF does not allow it to find best path all the time GBF in this implementation also is not able to find best path for target. Therefore, it gives different path solutions with uniform cost and A* search.

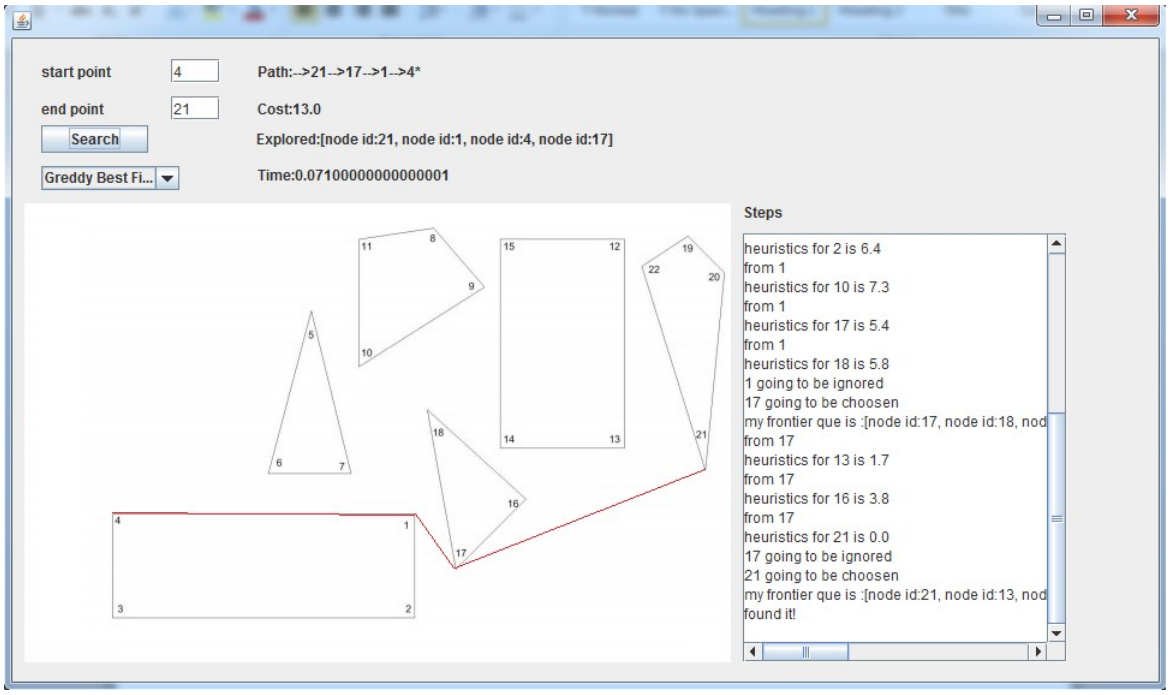
start point
4
end point
21
Search
Greddy Best Fi...
Path:-->21-->17-->1-->4*
Cost:13.0
Explored:[node id:21, node id:1, node id:4, node id:17]
Time:0.07100000000000001
Steps
heuristics for 2 is 6.4
from 1
heuristics for 10 is 7.3
from 1
heuristics for 17 is 5.4
from 1
heuristics for 18 is 5.8
1 going to be ignored
17 going to be choosen
my frontier que is :[node id:17, node id:18, nod
from 17
heuristics for 13 is 1.7
from 17
heuristics for 16 is 3.8
from 17
heuristics for 21 is 0.0
17 going to be ignored
21 going to be choosen
my frontier que is :[node id:21, node id:13, nod
found it!

Figure 15:

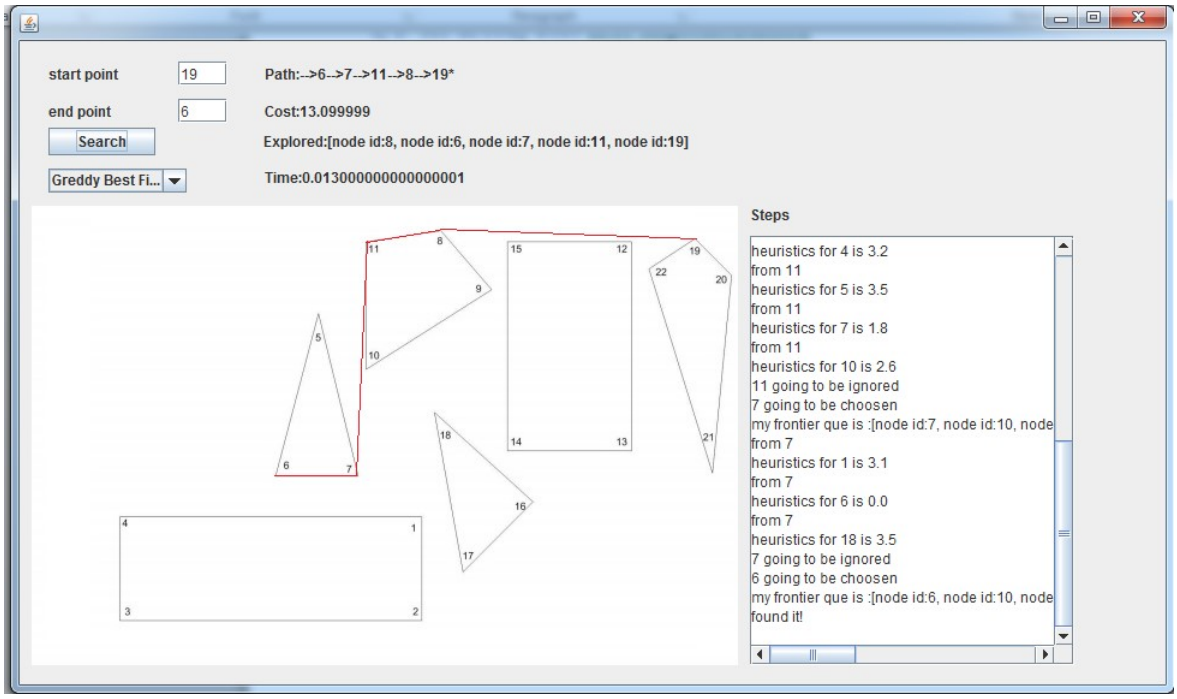
start point
19
end point
6
Search
Greddy Best Fi...
Path:-->6-->7-->11-->8-->19*
Cost:13.099999
Explored:[node id:8, node id:6, node id:7, node id:11, node id:19]
Time:0.013000000000000001
Steps
heuristics for 4 is 3.2
from 11
heuristics for 5 is 3.5
from 11
heuristics for 7 is 1.8
from 11
heuristics for 10 is 2.6
11 going to be ignored
7 going to be choosen
my frontier que is :[node id:7, node id:10, node
from 7
heuristics for 1 is 3.1
from 7
heuristics for 6 is 0.0
from 7
heuristics for 18 is 3.5
7 going to be ignored
6 going to be choosen
my frontier que is :[node id:6, node id:10, node
found it!

Figure 16:

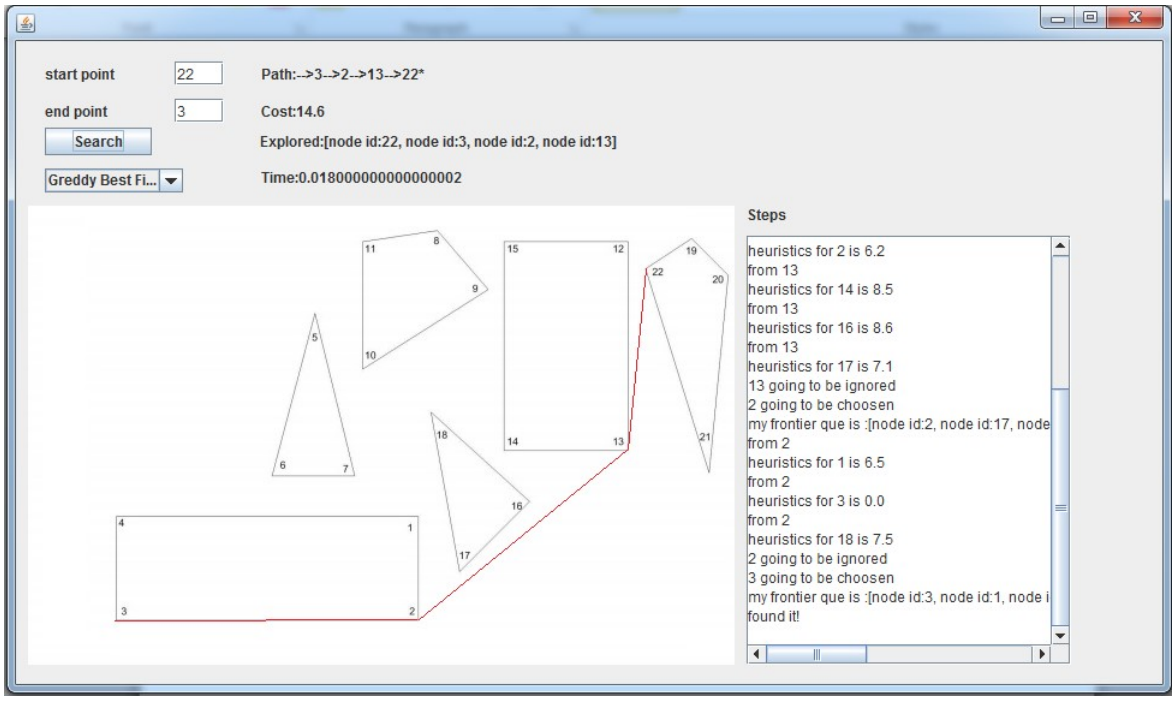
start point
22
end point
3
Search
Greddy Best Fi...
Path:-->3-->2-->13-->22*
Cost:14.6
Explored:[node id:22, node id:3, node id:2, node id:13]
Time:0.018000000000000002
Steps
heuristics for 2 is 6.2
from 13
heuristics for 14 is 8.5
from 13
heuristics for 16 is 8.6
from 13
heuristics for 17 is 7.1
13 going to be ignored
2 going to be choosen
my frontier que is :[node id:2, node id:17, node
from 2
heuristics for 1 is 6.5
from 2
heuristics for 3 is 0.0
from 2
heuristics for 18 is 7.5
2 going to be ignored
3 going to be choosen
my frontier que is :[node id:3, node id:1, node i
found it!

Figure 17:

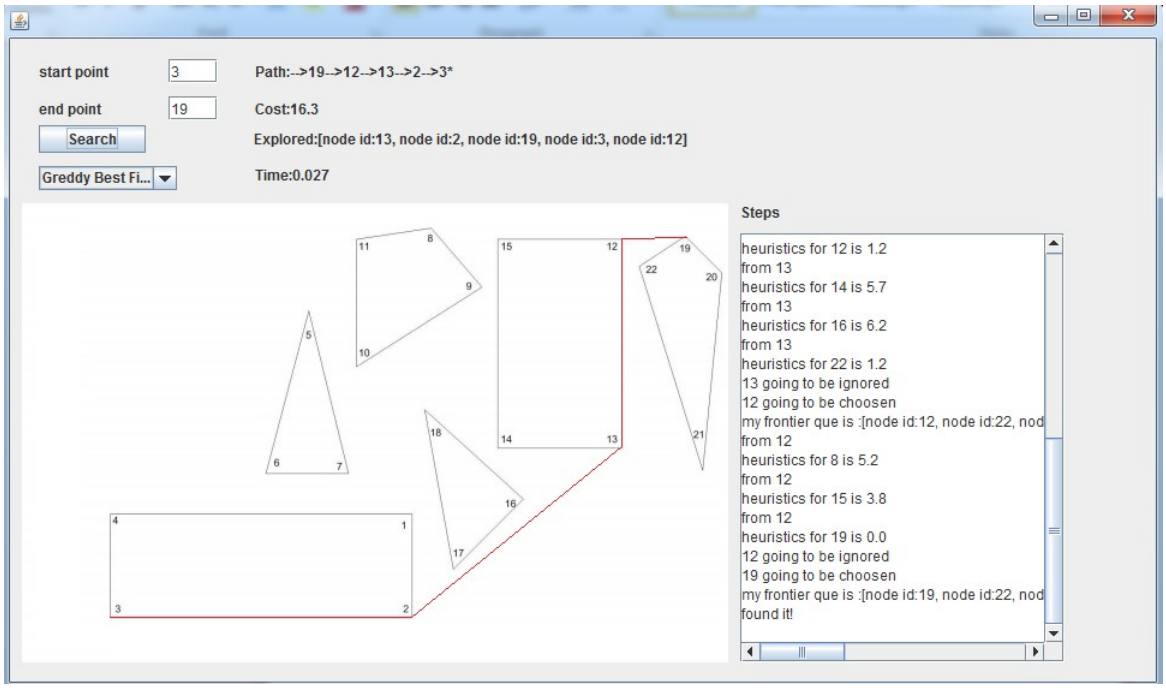

Figure 18:

Part XIV

A* (A START SEARCH) DESCRIPTION

A* algorithm is very similar to uniform cost algorithm except it calculates cost like $f(n) = g(n)+h(n)$ [1] where $g(n)$ represents past cost from start to n(current) and h(n) gives the cost to get from the node to the goal (admissible heuristics). This new way of calculating cost provides robust design and makes A* both complete and optimal. The complexity of A* algorithm depends on the heuristic.

Part XV

A* İMPLEMENTATİON

A* implementation is almost same as uniform cost implementation and related code takinf places in AStar class .Only difference is, when A*called it calculates heuristics for each node based on target node and uses $f(n) = g(n)+h(n)$[1]

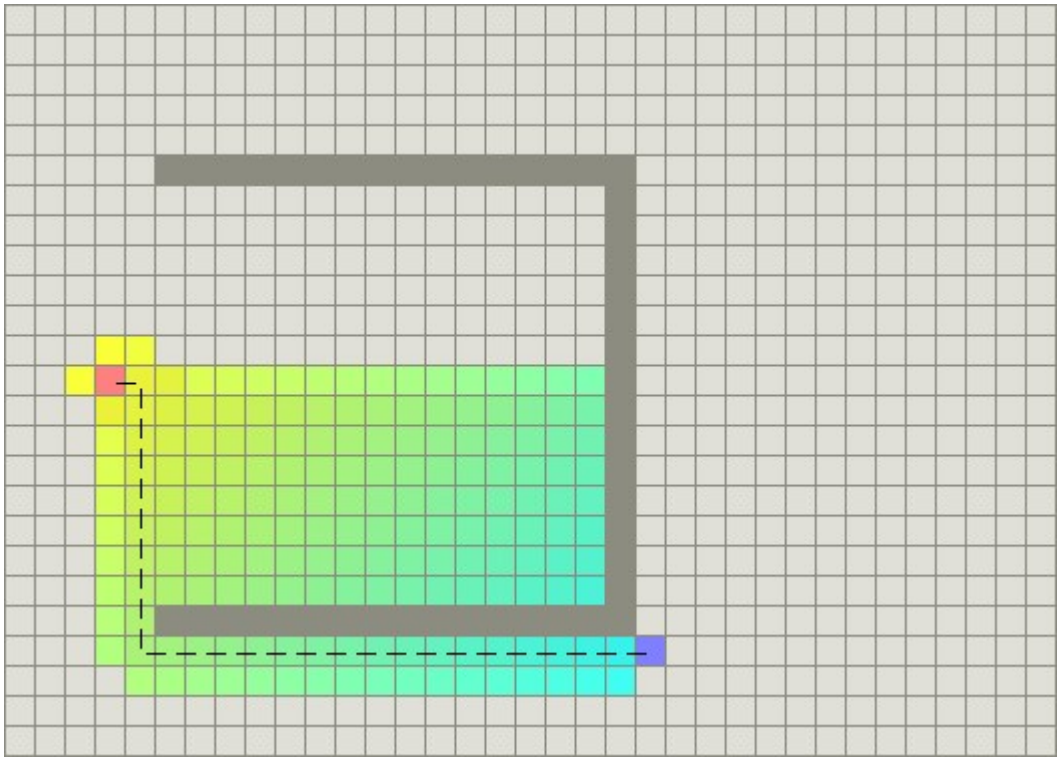

Figure 19: This figure shows the way A* to finds the target.Compared to figure-14, performance has improved remarkably

algorithm to calculate cost and min value with “astartcost” is chosen to move.

Part XVI
I/O FOR A* IMPLEMENTATION

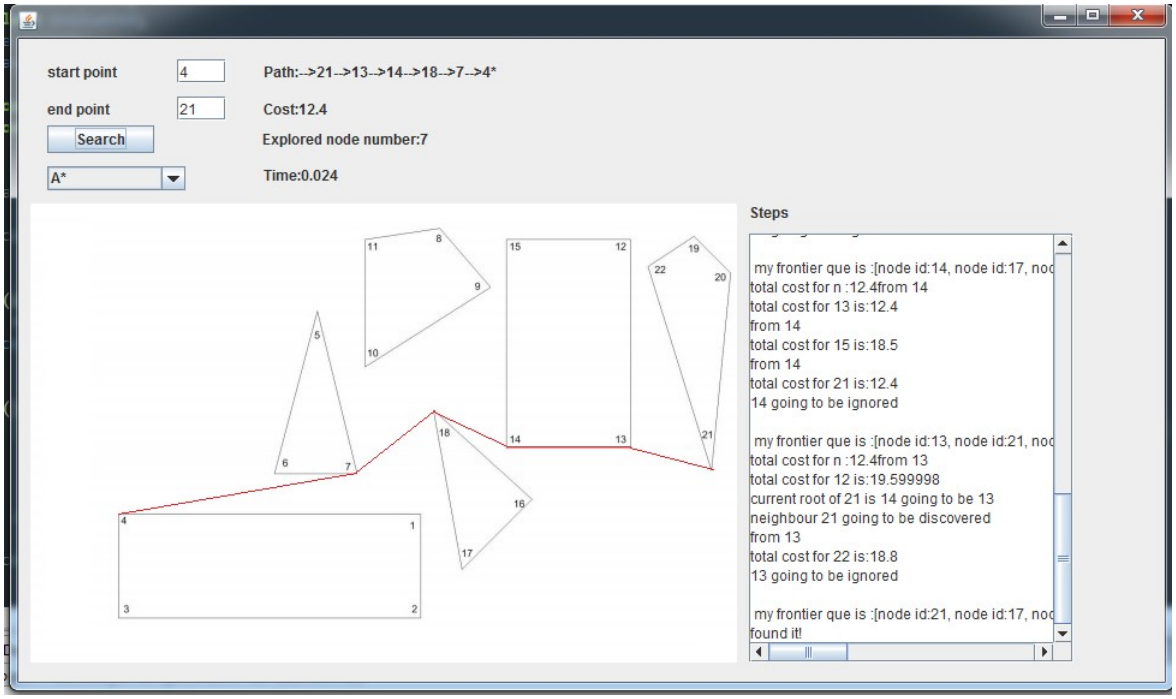

Figure 20: has same result with uniform cost since there is one optimum solution for that path with less explored nodes .(given distance table have not altered)

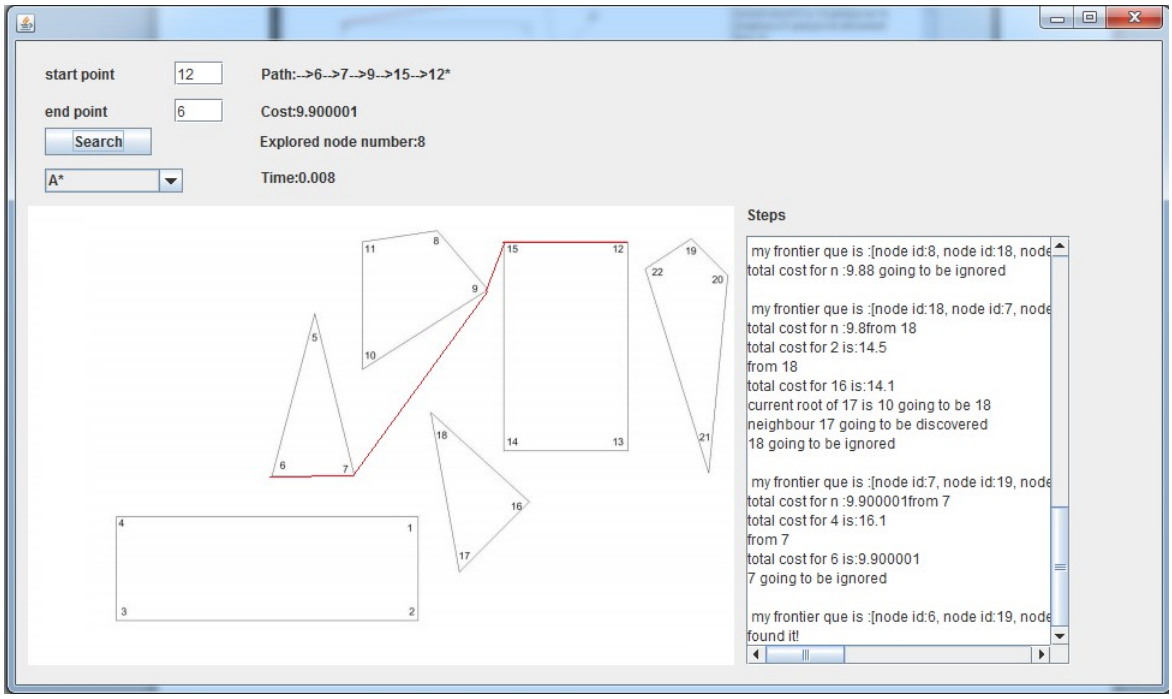
start point
12
end point
6
Search
A*
Path:-->6-->7-->9-->15-->12*
Cost:9.900001
Explored node number:8
Time:0.008
Steps
my frontier que is :[node id:8, node id:18, node
total cost for n :9.88 going to be ignored
my frontier que is :[node id:18, node id:7, node
total cost for n :9.8from 18
total cost for 2 is:14.5
from 18
total cost for 16 is:14.1
current root of 17 is 10 going to be 18
neighbour 17 going to be discovered
18 going to be ignored
my frontier que is :[node id:7, node id:19, node
total cost for n :9.900001from 7
total cost for 4 is:16.1
from 7
total cost for 6 is:9.900001
7 going to be ignored
my frontier que is :[node id:6, node id:19, node
found it!

Figure 21:

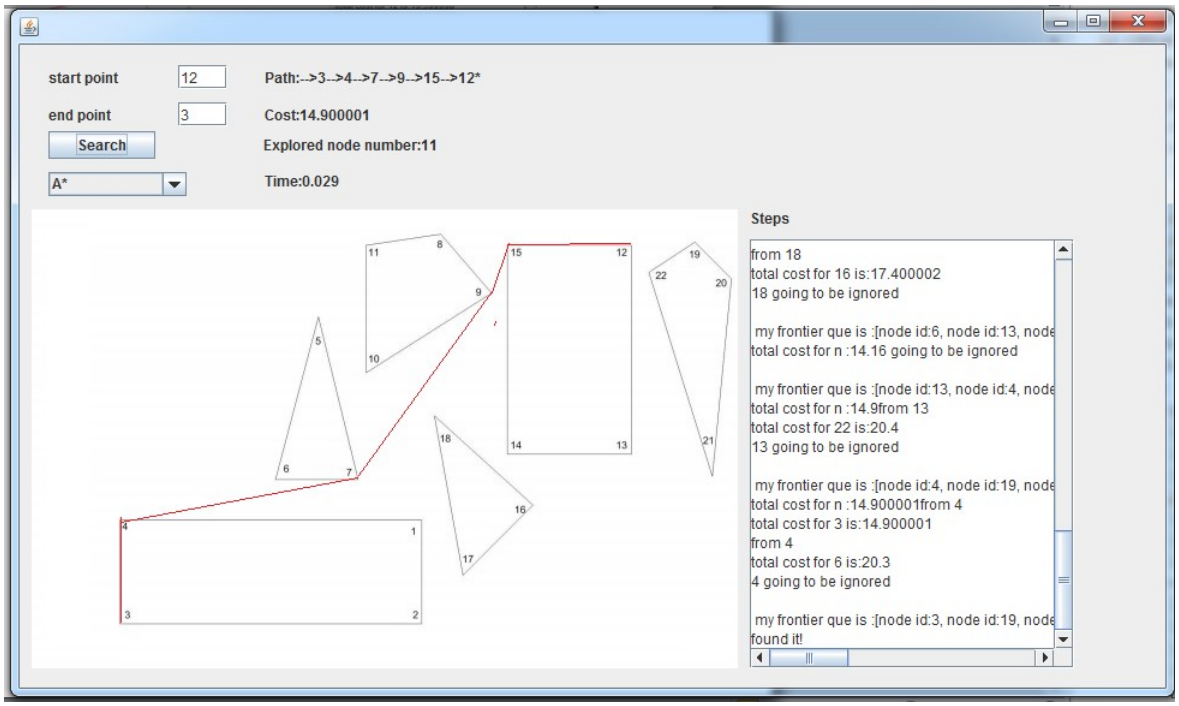
start point
12
end point
3
Search
A*
Path:-->3-->4-->7-->9-->15-->12*
Cost:14.900001
Explored node number:11
Time:0.029
Steps
from 18
total cost for 16 is:17.400002
18 going to be ignored
my frontier que is :[node id:6, node id:13, node
total cost for n :14.16 going to be ignored
my frontier que is :[node id:13, node id:4, node
total cost for n :14.9from 13
total cost for 22 is:20.4
13 going to be ignored
my frontier que is :[node id:4, node id:19, node
total cost for n :14.900001from 4
total cost for 3 is:14.900001
from 4
total cost for 6 is:20.3
4 going to be ignored
my frontier que is :[node id:3, node id:19, node
found it!

Figure 22:

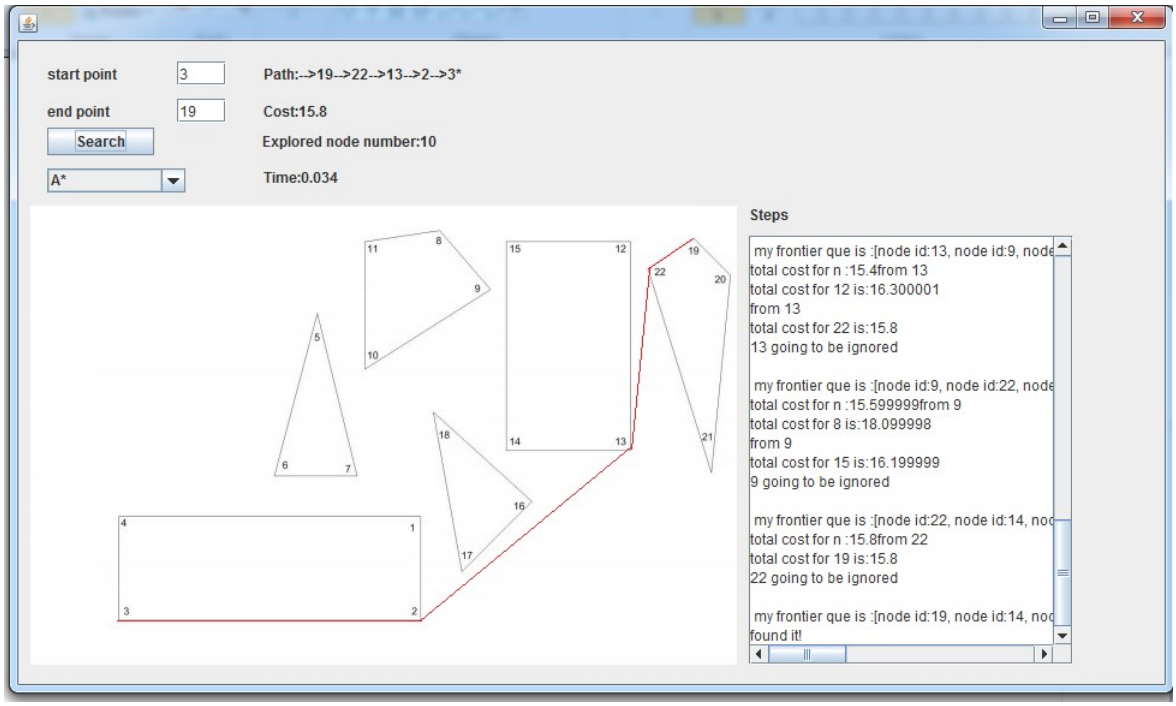

Figure 23:

Part XVII
COMPARİSON OF ALGORİTHMS

The practical and research shows that every algorithm performs differently in terms of the way of exploring nodes and number of nodes involved . BFS and DFS do not consider cost constraint, meanwhile uniform-cost, GBF and A* uses cost information to reach target node and minimum cost is most attractive for these algorithms to explore. This gives cost-based algorithms to advantage of reaching target point with less explored number of nodes and better path to reach target. However, these cost-based algorithms have different performance even though they take cost into consideration .They differ in the way of calculating cost is given in above sections for each algorithm.GBF looks for heuristics(future cost) and uniform cost considers past cost .This way of calculating cost allows uniform cost to optimise its path after iterations while GBF just looks to future costs and does not optimize its path .On the other hand since GBF moves towards the target for each iteration it visits less node com-

paring with uniform-cost (output for same paths figure x-y shows clearly about number of expanded nodes different from each other) .At the same time A* uses, combination of GBF and uniform-cost for calculating costs it gives the best result (minimum cost,optimum path) with less explored nodes. (output can be seen in figures or GUI) Minimum cost is method of node exploration for uniform cost, GBF and A* meanwhile BFS and DFS have different methods for exploring nodes .BFS algorithm looks for all nodes in level on the other hand, DFS tries to explore up to the deepest level of an branch . It is hard to say that one is better to another . In my opinion these algorithms should be implemented by taking into consideration the structure of the tree .

References

[1] N. Russell, Artificial Intelligence A Modern Approach, 2010.

[2] S. Luger, AI Algorithms,Data Structures and Idioms in Prolog , Lisp and Java.

[3] Lucas, Introduction to Prolog, University f Aberdeen .