Maze

Version: 4.1.1

User Manual

An attempt to visualize
five search algorithms

Nikos Kanargias
Hellenic Open University student
Athens 2013

Contents

A few words about the problem

The stimulus for the preparation of this project was the first assignment of PLI31 (Artificial Intelligence – Applications) module of Academic Year 2012-13, two questions of which were related to a robot motion planning problem, ie search for the path towards a target for a robot that can move on a grid which includes obstacles. The idea for this realization came from the animation that one can see in the Wikipedia article related to Dijkstra's algorithm.

(http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm).

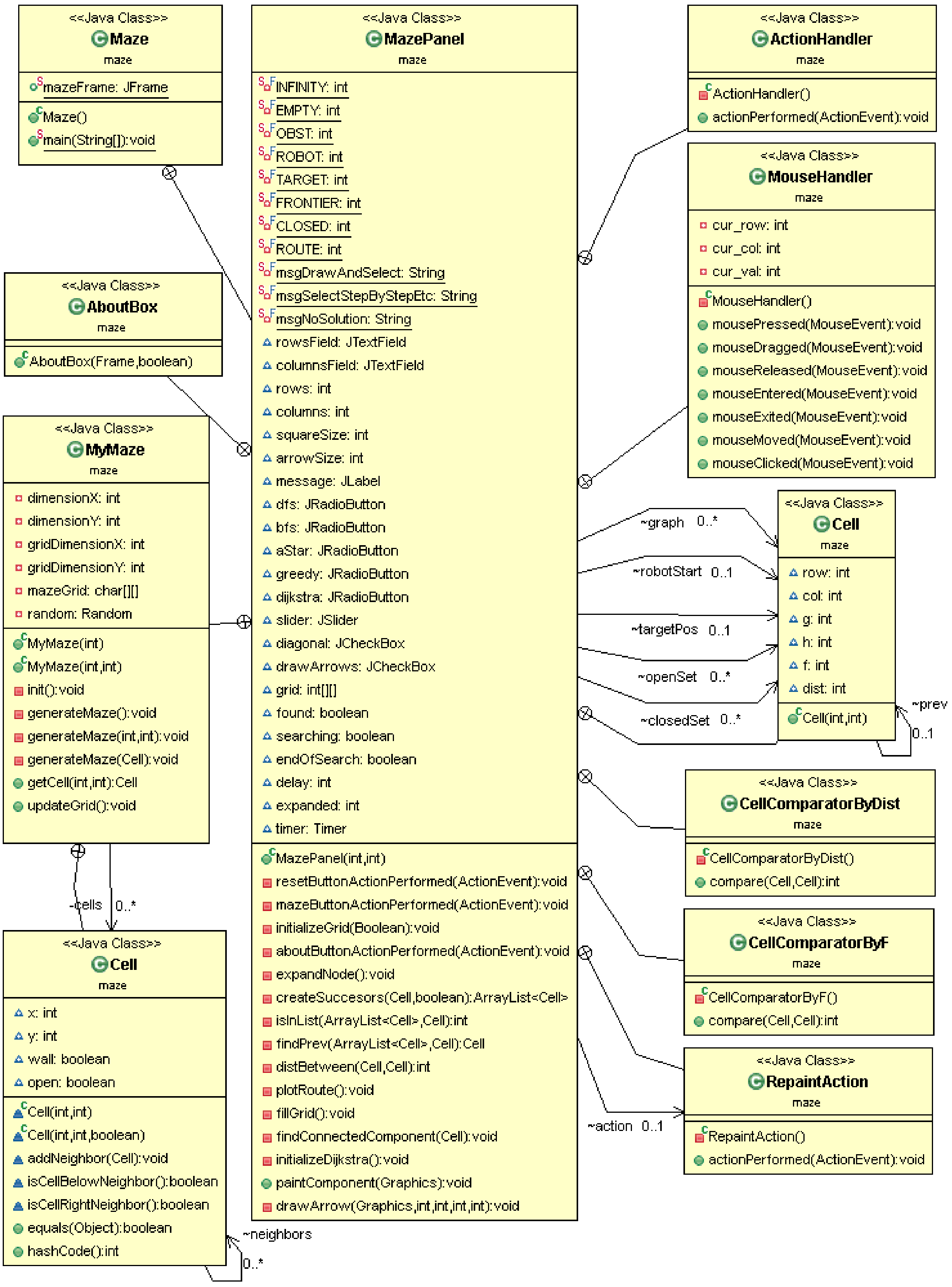
The described software solves and visualizes the aforementioned problem by implementing a variant of DFS, BFS and A* algorithms, as described by E. Keravnou in her book: "Artificial Intelligence and Expert Systems", Hellenic Open University, Patra 2000 (in Greek) and the Greedy search algorithm, as a special case of A*.

The software also implements Dijkstra's algorithm, as just described in the relevant article in Wikipedia.

I would like to express my sincere thanks to Professor Dimitrios Kalles for his encouragement and support, to Professor Panagiotis Stamatopoulos for his valuable comments and to my teacher, Professor Spyros Lykothanasis for his excellent teaching.

N. K.
Athens, November 2013

The class diagram



The above diagram was created using the ObjectAid add-on (http://www.objectaid.com) of Eclipse IDE.

Comments on the structure of classes

In addition to the extensive commentary existing in the Java code, the following can be added:

Maze class is the main class, which has the main form (mazeFrame) of the software as its field.

Nested in Maze class is the MazePanel class.

MazePanel class defines the contents of the main form and contains all the functionality of the program.

Nested in MazePanel class are the classes:

ActionHandler : Specifies the function to be executed when the user presses each of the six buttons on the main form.

MouseHandler : Handles mouse movements as the user "paints" obstacles or move the robot and/or the target.

Cell : Represents the cell of the grid.

CellComparatorByF : Specifies that the cells will be sorted according to their **f** field (used by A* and Greedy algorithms).

CellComparatorByDist : Specifies that the cells will be sorted according to their **dist** field (used by Dijkstra’s algorithm).

RepaintAction : The class that is responsible for the animation.

AboutBox : The class that raises the About Box of the software.

MyMaze : Creates a random, perfect (without cycles) maze. The code of the class is an adaptation, with the original commentary, of the answer given by user DoubleMx2 on August 25 to a question posted by user nazar_art at stackoverflow.com:

http://stackoverflow.com/questions/18396364/maze-generation-arrayindexoutofboundsexception

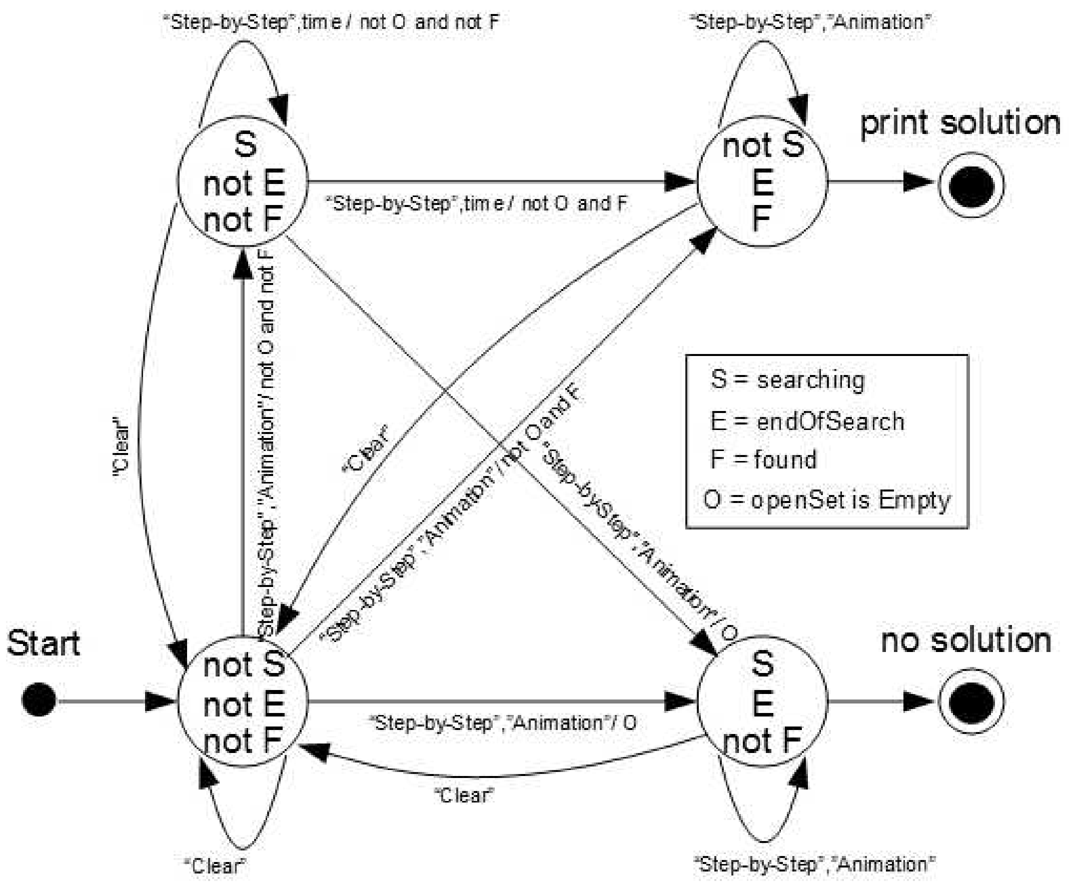
Nested in MyMaze class is **Cell** class that represents the cell for the creation of the random labyrinth.

The state transition diagram

The search state is a combination of three parameters:

- Whether the search is in progress (searching).
- Whether the search has reached the end (endOfSearch).
- Whether the target has been found (found).

There are eight combinations of values that can take the above logical variables. Of these, only four combinations correspond to possible search states and are depicted in the following state transition diagram:

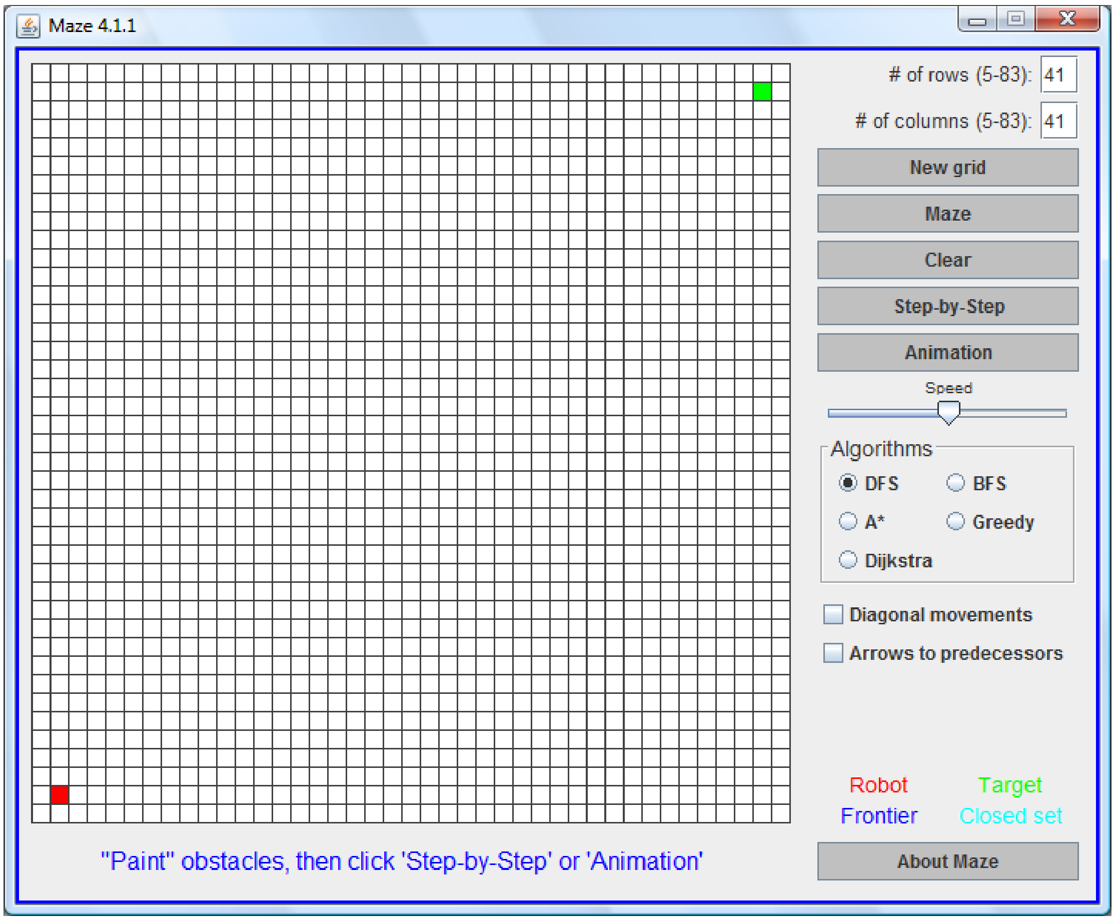

"Step-by-Step",time / not O and not F
"Step-by-Step","Animation"
S not E not F
not S E F
print solution
"Step-by-Step",time / not O and F
S = searching
E = endOfSearch
F = found
O = openSet is Empty
"Clear"
"Step-by-Step","Animation"/ not O and not F
"Clear"
"Step-by-Step","Animation"/ not O and F
"Step-by-Step","Animation"/ O
Start
not S not E not F
S E not F
no solution
"Step-by-Step","Animation"/ O
"Clear"
"Clear"
"Step-by-Step","Animation"

"Step-by-Step", "Animation" and "Clear" means: after pressing the corresponding button.

The indication ‘time’ means: after the delay time (if the search started by pressing the "Animation" button).

The operation of the program

On starting the program displays the following form:



The application uses a constant surface area, measuring 500×500 pixels, to display the grid.

The user can change the number of rows and columns of the grid by entering the desired values in the corresponding fields and then pressing the button "New grid".

The program will adjust the size of the cell for better utilization of the available area.

The user can add as many obstacles he/she wants, as he/she would "paint" free curves with a drawing program.

Individual obstacles can be removed by clicking on them.

The position of the robot and/or the target can be changed by dragging with the mouse.

The search algorithm can be selected by pressing the corresponding button.

If diagonal movements of the robot are also desired, check the corresponding box.

For each node expanded the application may draw arrows from the successors to the predecessor state. The calculation and drawing of these arrows is a time consuming process. Up to the present version of the application, for each node that expands, the arrows of the nodes that have already been expanded should be calculated and drawn again. The time required for this calculation is unfortunately an exponential function of the number of nodes expanded and, at worst, a search may take about 6 minutes (with processor Intel Core2 Duo @2.66GHz). The problem is overcome by disabling the drawing of arrows for large grids, using the checkbox “Arrows to predecessors”, so that the maximum search time becomes less than 45 seconds. Note that the arrows are not sufficiently visible when the cell size is very small.

The application considers that the robot itself has some volume. Therefore it can’t move diagonally to a free cell passing between two obstacles adjacent to an apex. In other words, diagonal movement is only allowed when at least one of the two cells adjacent to the initial one and the final position of the robot are free.

The search can be done in two ways:

1. With repetitive pressure on the “Step-by-Step” button, so that we can expand a node at a time.
2. With a single press on the “Animation” button, so that the expansion of nodes is done automatically, interpolating a pause of 0 to 1 second between two successive expansions. The length of the pause can be adjusted depending on the location of the slider “Speed”.

Jump from search in “Step-by-Step” way to “Animation” way and vice versa is done by pressing the corresponding button, even when a search is in progress.

The speed of a search can be changed, even if the search is in progress. It is sufficient to place the slider “Speed” in the new desired position and then press the “Animation” button.

Pressing the “Clear” button once a search is completed, or when it is still in progress, results in cleansing the current search, but the obstacles, the robot and the target remain in place, in order to allow experimentation with another algorithm with the same data.

Pressing the “Clear” button for the second time in a row, results in also cleansing the obstacles, and the robot and target returning to their original position.

It is not possible to change the position of the obstacles, robot and target, as well as the search algorithm, while the search is underway.

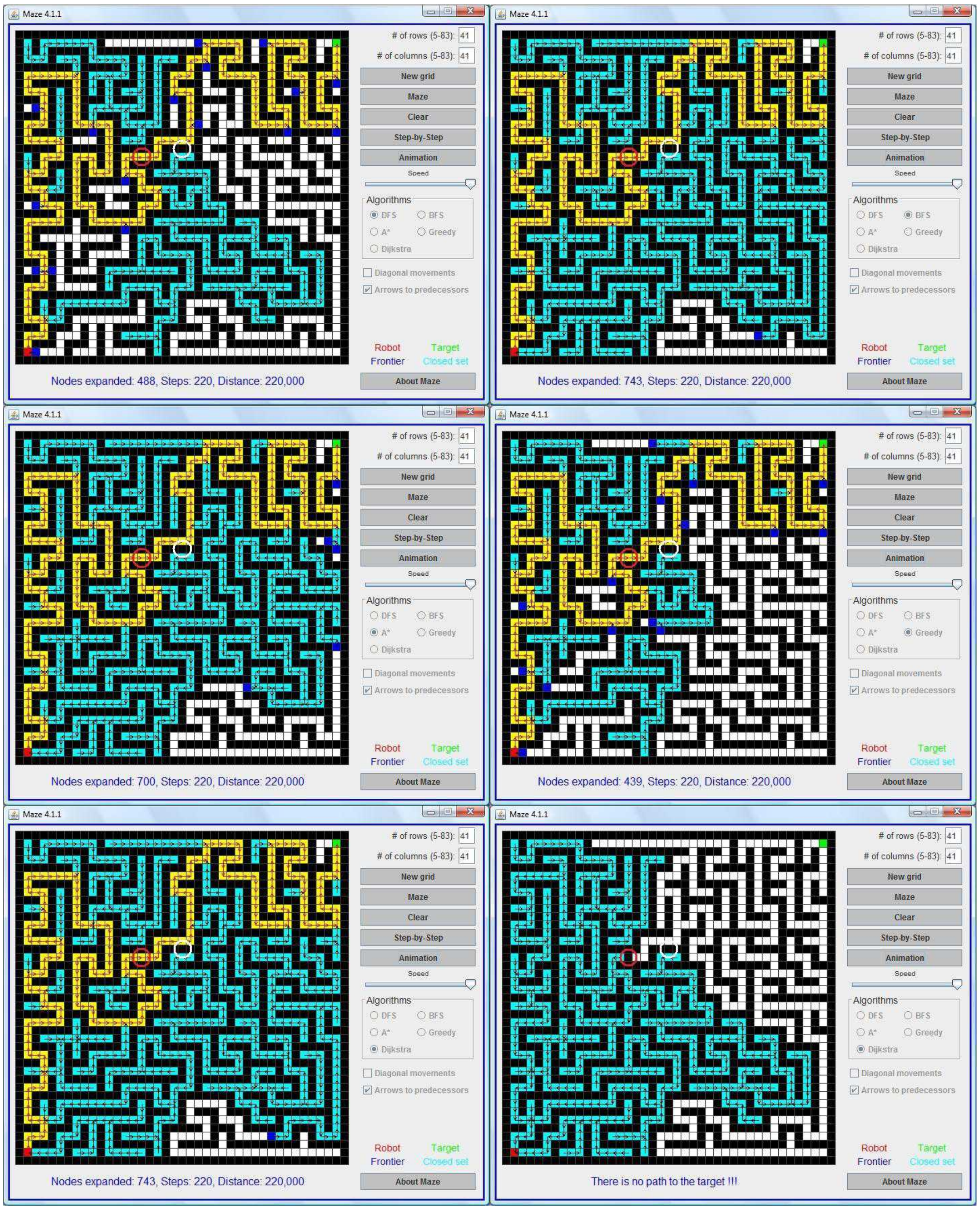
About solving the maze

(Just for remember a few elements of graph theory)

The following discussion assumes that diagonal movements are not allowed.

If instead the “New Grid” button, the “Maze” button is pressed, the application will create and place a random maze on the grid.

If we consider the graph whose vertices are the free cells of the grid, and whose edges are the line segments joining every two adjacent free cells together, then this graph is coherent and without cycles (such as cared by the algorithm that constructed it). This means it is a tree.



Maze 4.1.1 — # of rows (5-83): 41 — # of columns (5-83): 41 — Algorithms: [✓ DFS] BFS A* Greedy Dijkstra — ☐ Diagonal movements — ☑ Arrows to predecessors — Nodes expanded: 488, Steps: 220, Distance: 220,000

Maze 4.1.1 — # of rows (5-83): 41 — # of columns (5-83): 41 — Algorithms: DFS [✓ BFS] A* Greedy Dijkstra — ☐ Diagonal movements — ☑ Arrows to predecessors — Nodes expanded: 743, Steps: 220, Distance: 220,000

Maze 4.1.1 — # of rows (5-83): 41 — # of columns (5-83): 41 — Algorithms: DFS BFS [✓ A*] Greedy Dijkstra — ☐ Diagonal movements — ☑ Arrows to predecessors — Nodes expanded: 700, Steps: 220, Distance: 220,000

Maze 4.1.1 — # of rows (5-83): 41 — # of columns (5-83): 41 — Algorithms: DFS BFS A* [✓ Greedy] Dijkstra — ☐ Diagonal movements — ☑ Arrows to predecessors — Nodes expanded: 439, Steps: 220, Distance: 220,000

Maze 4.1.1 — # of rows (5-83): 41 — # of columns (5-83): 41 — Algorithms: DFS BFS A* Greedy [✓ Dijkstra] — ☐ Diagonal movements — ☑ Arrows to predecessors — Nodes expanded: 743, Steps: 220, Distance: 220,000

Maze 4.1.1 — # of rows (5-83): 41 — # of columns (5-83): 41 — Algorithms: DFS BFS A* Greedy [✓ Dijkstra] — ☐ Diagonal movements — ☑ Arrows to predecessors — There is no path to the target !!!

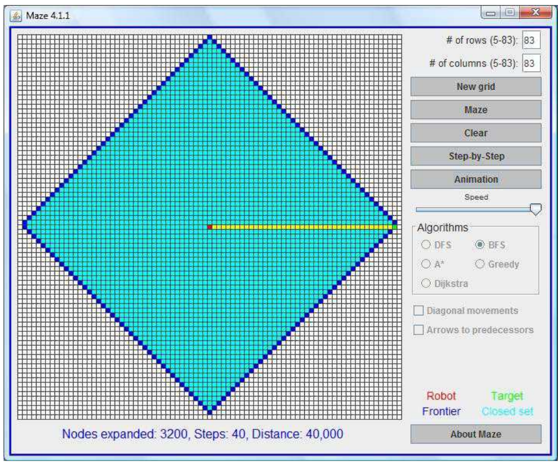
As we all know, in any coherent tree there is only one path connecting any two nodes. As seen in the previous pictures, all the search algorithms calculated the same, unique path to the target, having expanded different nodes each.

As is also known, if any internal vertex of a tree is removed, the tree is not a coherent graph any more and becomes a forest, with so many trees as the edges ending at the vertex that has been removed. If the vertex and edges which have been removed were part of the route between two specific nodes, these nodes are no longer mutually accessible, since they would belong to different connected components of the graph.

In the last of the previous figures, the obstacle which was added (inside the red circle) corresponds to removing a vertex with two edges, because the obstacle is located between two free cells. There are places on the maze where an obstacle can be placed between three, even four free cells. The placement of this particular obstacle results in the original tree to split into two subtrees: the first tree is the set of closed states, which includes the initial position of the robot, while the second tree is the set of cells which have not been explored, in which the target belongs too. This separation makes the target inaccessible by the robot. The picture shows the failure of Dijkstra's algorithm to locate the target. The remaining four algorithms have similar behavior.

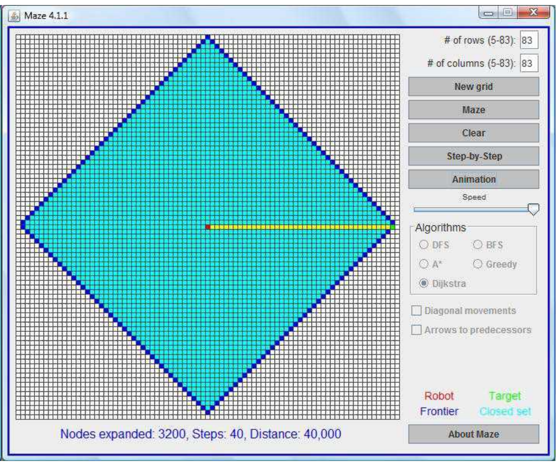
Among the five algorithms implemented by the application, only BFS and Dijkstra's explore in all directions, while open set nodes (frontier) form a wave that expands centered on the initial position of the robot.

In case that diagonal movements are not allowed, this wave for the above two algorithms has a diamond shape as shown in the pictures below, while these two algorithms have exactly the same behavior. This means that for a given problem they find the same solution, having expanded the same set of nodes.



# of rows (5-83): 83
# of columns (5-83): 83
Algorithms: DFS [✓ BFS] A* Greedy Dijkstra
Diagonal movements ☐
Arrows to predecessors ☐

Nodes expanded: 3200, Steps: 40, Distance: 40,000

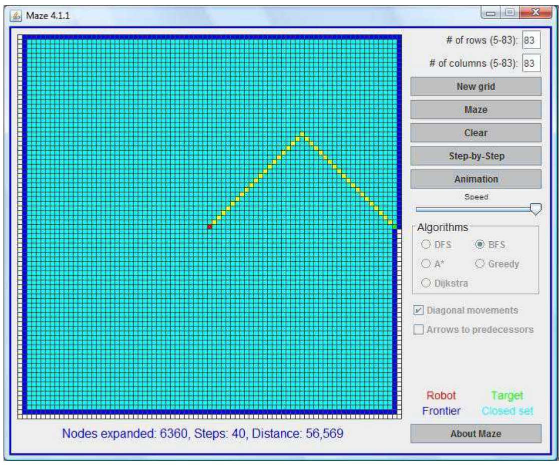


# of rows (5-83): 83
# of columns (5-83): 83
Algorithms: DFS BFS A* Greedy [✓ Dijkstra]
Diagonal movements ☐
Arrows to predecessors ☐

Nodes expanded: 3200, Steps: 40, Distance: 40,000

That is why the solutions given by the two algorithms, and depicted on the previous page, are identical.

The two algorithms behave differently when diagonal movements were allowed, as shown in the following pictures.


Maze 4.1.1

# of rows (5-83): 83

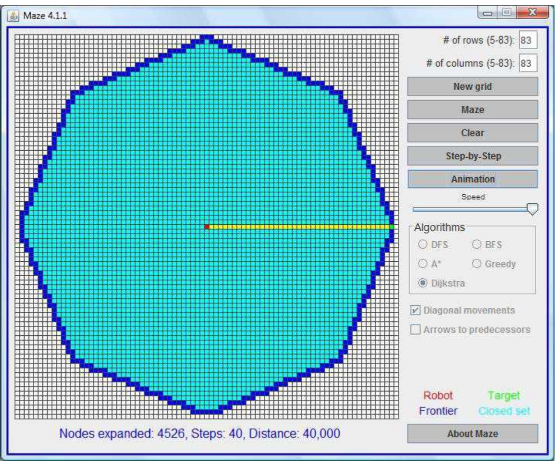
# of columns (5-83): 83

Algorithms: ○ DFS [● BFS] ○ A* ○ Greedy ○ Dijkstra

[✓] Diagonal movements

☐ Arrows to predecessors

Nodes expanded: 6360, Steps: 40, Distance: 56,569


Maze 4.1.1

# of rows (5-83): 83

# of columns (5-83): 83

Algorithms: ○ DFS ○ BFS ○ A* ○ Greedy [● Dijkstra]

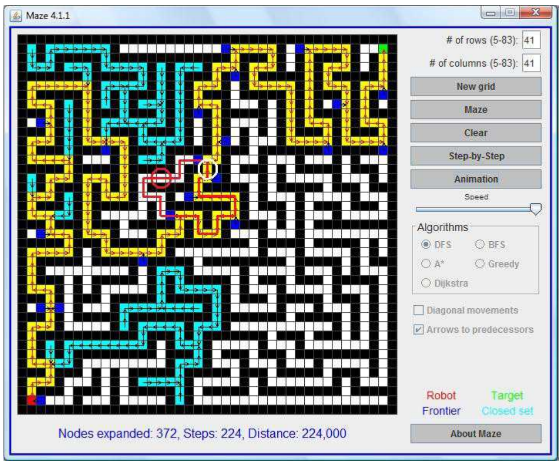
[✓] Diagonal movements

☐ Arrows to predecessors

Nodes expanded: 4526, Steps: 40, Distance: 40,000

Conversely, if we remove an obstacle (inside the white circle) from the original maze, ie we add one more vertex with degree at least two (at least two edges that result in it) in the original tree, then the graph is no longer a tree (ie a coherent graph without cycles), because one cycle is then formed. Therefore between two specific nodes, there may be more than one route.

The following pictures show the cycle (the thick red line) formed by removing the obstacle, as well as the two different routes discovered by the DFS and BFS algorithms respectively.


Maze 4.1.1

# of rows (5-83): 41

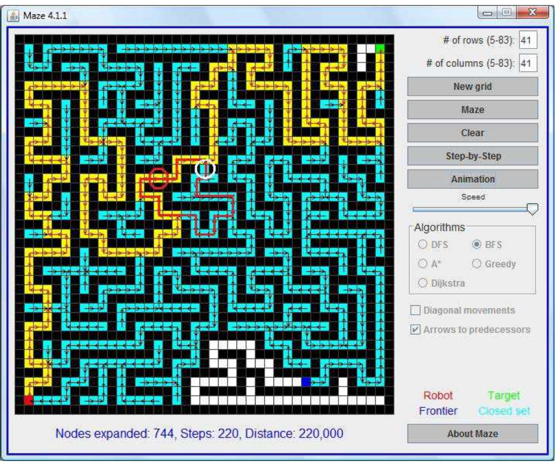
# of columns (5-83): 41

Algorithms: [● DFS] ○ BFS ○ A* ○ Greedy ○ Dijkstra

☐ Diagonal movements

[✓] Arrows to predecessors

Nodes expanded: 372, Steps: 224, Distance: 224,000


Maze 4.1.1

# of rows (5-83): 41

# of columns (5-83): 41

Algorithms: ○ DFS [● BFS] ○ A* ○ Greedy ○ Dijkstra

☐ Diagonal movements

[✓] Arrows to predecessors

Nodes expanded: 744, Steps: 220, Distance: 220,000

The search algorithms

The application implements the following algorithms:

Depth First Search algorithm:

1. OPEN SET := [s_o], CLOSED SET := [ ]
2. If OPEN SET = [ ], then terminate. There is no solution.
3. Remove the first state, s_i, from OPEN SET and add it to CLOSED SET. If s_i is the s_g (the goal), then terminate and return the route from s_g to s_o.
4. Create the successors of s_i, not already in OPEN SET or CLOSED SET, based on actions that can be implemented on s_i. Each successor has a pointer to the s_i, as its predecessor.
5. Add the successors at the **beginning** of the list OPEN SET and repeat from step 2.

Breadth First Search algorithm (BFS) is exactly the same as the Depth First Search algorithm (DFS), with the only difference that in step 5, the new open set states are added at the **end** and not at the beginning of the list.

Heuristic Search: A* Algorithm

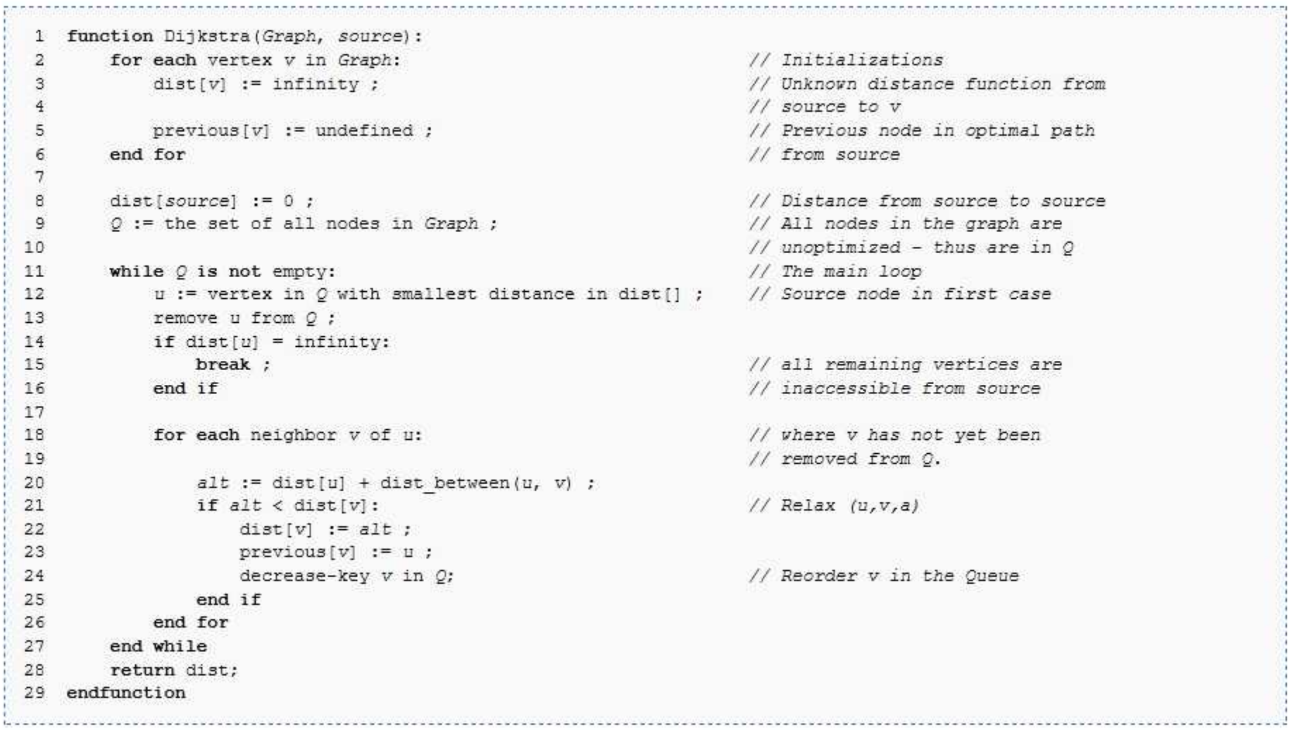
1. OPEN SET := [s_o], CLOSED SET := [ ]
2. If OPEN SET = [ ], then terminate. There is no solution.
3. Remove the first state, s_i, from OPEN SET, for which $f(s_i) \leq f(s_j)$ for all other open states s_j and add it to CLOSED SET. If s_i is the s_g (the goal), then terminate and return the route from s_g to s_o.
4. Create the successors of s_i and give each successor a pointer to the s_i, as its predecessor.
5. Repeat the following for each successor, s_j, of s_i:
   - Calculate the value $f(s_j)$.
   - If s_j is neither on OPEN SET nor on CLOSED SET states, then add the s_j in OPEN SET evaluated as $f(s_j)$.
   - If s_j already belongs to OPEN SET or CLOSED SET, then compare the new value assessment, say *new*, with the older, say *old*. If $old \leq new$, then eject the new node with state s_j. Otherwise, remove the element (s_j, *old*) from the list to which it belongs (OPEN SET or CLOSED SET) and add the item (s_j, *new*) to the OPEN SET.
6. Repeat from step 2.

The **Greedy search algorithm** is exactly the same as the A* algorithm, with the only difference that in the calculation of the value of the function $f(s_j)$ in step 5, only function h is used, ie $g(s) = 0$ for all states.

Reminder:
$f(s_i) = g(s_i) + h(s_i)$
$f(s_i)$: evaluation function
$g(s_i)$: real cost for going from s_o (initial state) to s_i.
$h(s_i)$: heuristic estimate of the cost for going from s_i to the s_g (the goal).

There are detailed comments in the code corresponding to each of the above steps with the command that implements them.

For Dijkstra's algorithm the pseudo-code from the relevant article in Wikipedia is used:



```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                                // Initializations
        dist[v] := infinity ;                                  // Unknown distance function from
                                                               // source to v
        previous[v] := undefined ;                             // Previous node in optimal path
    end for                                                    // from source

    dist[source] := 0 ;                                        // Distance from source to source
    Q := the set of all nodes in Graph ;                       // All nodes in the graph are
                                                               // unoptimized - thus are in Q
    while Q is not empty:                                      // The main loop
        u := vertex in Q with smallest distance in dist[] ;    // Source node in first case
        remove u from Q ;
        if dist[u] = infinity:
            break ;                                            // all remaining vertices are
        end if                                                 // inaccessible from source

        for each neighbor v of u:                              // where v has not yet been
                                                               // removed from Q.
            alt := dist[u] + dist_between(u, v) ;
            if alt < dist[v]:                                  // Relax (u,v,a)
                dist[v] := alt ;
                previous[v] := u ;
                decrease-key v in Q;                           // Reorder v in the Queue
            end if
        end for
    end while
    return dist;
endfunction
```

Again, there are detailed comments in the code relating the steps of the above pseudo-code to the program commands.

A part of the initialization of the algorithm, not included in the above pseudo-code, which is discussed adequately in the application code, may require some time (seconds), especially in large grids.

Immediately after this initialization, it is trivial for the application to give a negative response to a problem (it only has only to check if the target is within the set of nodes forming the coherent component to which the initial position of the robot belongs). For demonstration purposes only, it tries the search until it reaches the specified failure.

Highlights of the program

Here are two snapshots of searches with A* and Dijkstra's algorithms, compared to the solutions from the corresponding articles of Wikipedia:
http://en.wikipedia.org/wiki/A*_search_algorithm
http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

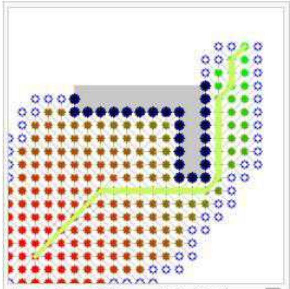
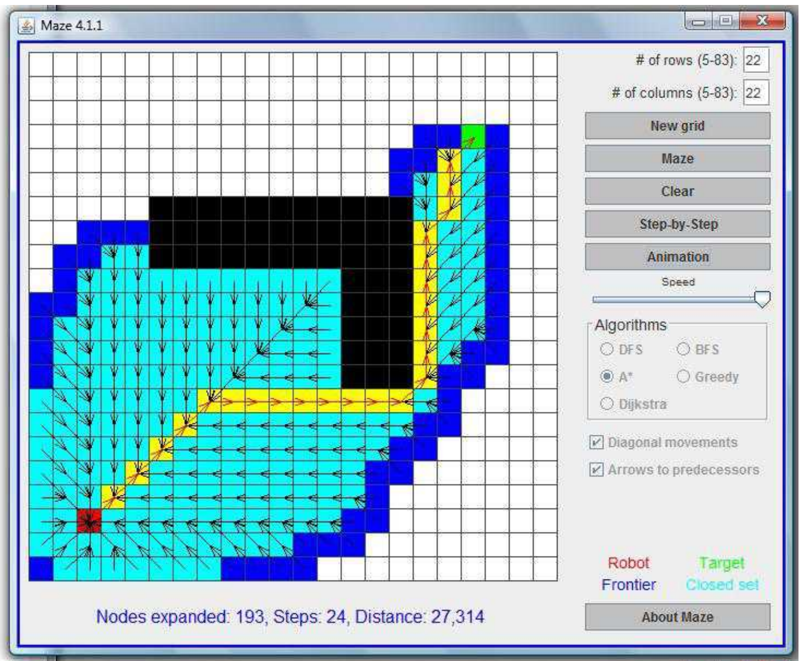


Illustration of A* search for finding path from a start node to a goal node in a robot motion planning problem. The empty circles represent the nodes in the *open set*, i.e., those that remain to be explored, and the filled ones are in the closed set. Color on each closed node indicates the distance from the start: the greener, the farther. One can first see the A* moving in a straight line in the direction of the goal, then when hitting the obstacle, it explores alternative routes through the nodes from the open set.

See also: Dijkstra's algorithm



Maze 4.1.1

# of rows (5-83): 22

# of columns (5-83): 22

Algorithms: DFS, BFS, [✓ A*], Greedy, Dijkstra

✓ Diagonal movements

✓ Arrows to predecessors

Nodes expanded: 193, Steps: 24, Distance: 27,314

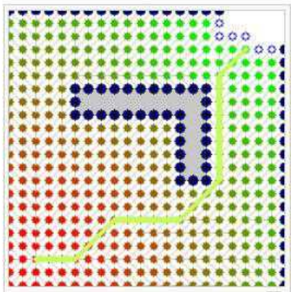
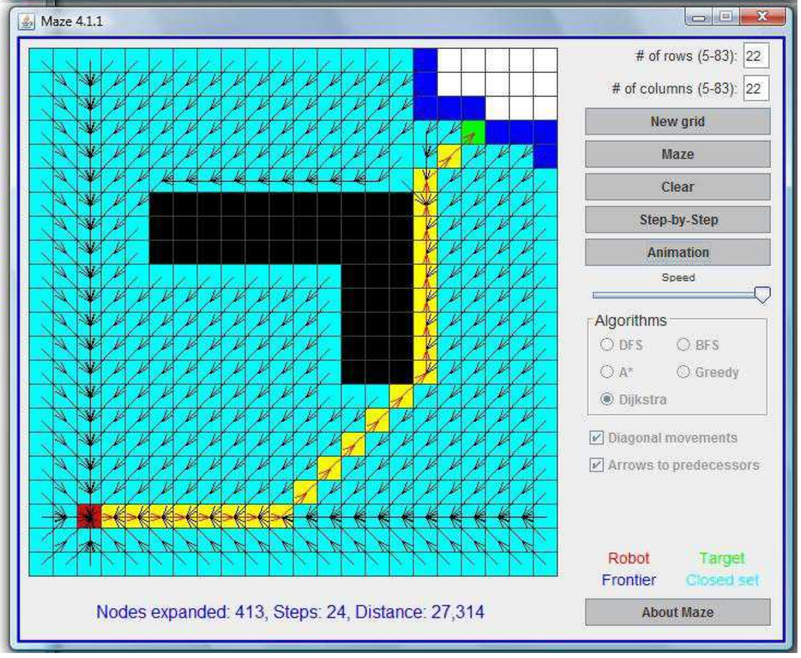


Illustration of Dijkstra's algorithm search for finding path from a start node (lower left, red) to a goal node (upper right, green) in a robot motion planning problem. Open nodes represent the "tentative" set. Filled nodes are visited ones, with color representing the distance: the greener, the farther. Nodes in all the different directions are explored uniformly, appearing as a more-or-less circular wavefront as Dijkstra's algorithm uses a heuristic identically equal to 0.



Maze 4.1.1

# of rows (5-83): 22

# of columns (5-83): 22

Algorithms: DFS, BFS, A*, Greedy, [✓ Dijkstra]

✓ Diagonal movements

✓ Arrows to predecessors

Nodes expanded: 413, Steps: 24, Distance: 27,314