

# Homework#1: Implementing Uniform Cost Search and A\* Search Algorithms

Submission date: 03/21/2023  
2018312164  
김석진

## ◆Research and Study

### Uniform Cost Search

-definition

uniform-cost search expands the node  $n$  with the lowest path cost  $g(n)$ . This is done by storing the frontier as a priority queue ordered by  $g$ .

-algorithm

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
    explored  $\leftarrow$  an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child  $\leftarrow$  CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier  $\leftarrow$  INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

<essential code review>

1. frontier must be saved in a priority queue which is ordered by PATH-COST
2. *problem*.Goal-Test(*node*.STATE) is applied just after expansion
3. check a queue for the same node with higher cost in order to replace with a lower cost

### 2. A\* Search

-definition

A\* search is an informed search algorithm which evaluates nodes by combining  $g(n)$ , the cost to reach the node, and  $h(n)$ , the cost to get from the node to the goal:  $f(n) = g(n) + h(n)$ . When heuristic function  $h(n)$  satisfies certain conditions, A\* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A\* uses  $g + h$  instead of  $g$

-algorithm

Almost same with uniform cost search but the priority is calculated in order of cost + distance

<essential code>

```

while openSet is not empty
    // This operation can occur in  $O(\log(N))$  time if openSet is a min-heap or a priority
    queue
    current := the node in openSet having the lowest fScore[] value
    if current = goal
        return reconstruct_path(cameFrom, current)

    openSet.Remove(current)
    for each neighbor of current
        // d(current, neighbor) is the weight of the edge from current to neighbor
        // tentative_gScore is the distance from start to the neighbor through current
        tentative_gScore := gScore[current] + d(current, neighbor)
        if tentative_gScore < gScore[neighbor]
            // This path to neighbor is better than any previous one. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := tentative_gScore + h(neighbor)
            if neighbor not in openSet
                openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure

```

### 3. Strength and weaknesses of each algorithm

Algorithm	Strengths	Weaknesses
Uniform cost search	<ul style="list-style-type: none"> <li>- uniform cost search guarantees to find the optimal path in terms of path cost.</li> <li>- uniform cost search explores the search space systematically, expanding the least cost nodes first, which makes it more efficient than uninformed search algorithms like Breadth-First Search.</li> <li>- uniform cost search is complete, meaning that it will always find a solution if one exists.</li> </ul>	<ul style="list-style-type: none"> <li>- uniform cost search can be slow when the cost of each step is high, or when the search space is very large.</li> <li>- uniform cost search can also be memory-intensive if there are many paths to explore, which can make it impractical for very large search problems.</li> <li>- in uniform cost search if the cost of the goal state is much higher than the cost of other states, UCS may explore many unnecessary nodes before finding the goal.</li> </ul>
A* search	<ul style="list-style-type: none"> <li>- A* search is generally faster than UCS, as it uses heuristics to guide the search towards the goal state.</li> <li>- A* search is also more memory-efficient than UCS, as it uses an evaluation function that considers both the cost of the path so far and an</li> </ul>	<ul style="list-style-type: none"> <li>- A* search requires a heuristic function, which can be difficult to design and can affect the performance of the search.</li> <li>- in A* search The quality of the heuristic can greatly affect the performance of A*, and it may</li> </ul>

	estimate of the remaining cost to the goal. - A* search is optimal, meaning that it guarantees to find the shortest path to the goal, as long as the heuristic is admissible (never overestimates the true cost to the goal).	not always be easy to find a good heuristic for a given problem. - A* search can expand many unnecessary nodes if the heuristic is not admissible or consistent, which can slow down the search significantly.
--	--	---

#### 4. Performance evaluation

- Completeness: A search algorithm is complete if it guarantees to find a solution if one exists. Completeness is a desirable property for any search algorithm.
- Optimality: A search algorithm is optimal if it finds the optimal solution, i.e., the solution with the minimum cost. This is important when finding the best solution is critical.
- Time Complexity: This metric measures the amount of time the search algorithm takes to find the solution. This is important when there are time constraints on finding the solution.
- Space Complexity: This metric measures the amount of memory the search algorithm requires to execute. This is important when memory constraints exist.
- Path Quality: This metric measures the quality of the solution found by the search algorithm. A good search algorithm should find solutions that are close to optimal and meet the requirements of the problem.
- Nodes Expanded: This metric measures the number of nodes that the search algorithm expands during the search. This is important when evaluating the efficiency of the algorithm.
- Branching Factor: This metric measures the average number of children a node has in the search tree. A lower branching factor indicates a more efficient search algorithm.
- Heuristic Quality: This metric measures the quality of the heuristic function used by the search algorithm. A good heuristic should be admissible, consistent, and informative.

## ◆ Uniform cost search code review

```
def uniform_cost_search(self):
    '''use priority queue to sort path by cost
    as we use only cost for sorting the structure inside a queue is a form of (cost, path)'''
    queue = PriorityQueue()
    queue.put((0, self.maze.entry_coor))
    #initial queue must be (cost = 0, path=entry)

    start = time.perf_counter()
    print("\nSolving the maze with uniform_cost_search...")
    #starting time of searching

    '''algorithm starts with a two infinity loop until solution is found or search failier loop cotinues'''
    while True:
        while queue:
            # infinity loop until solution is found and break
            # infinity loop until no possible way of expansion

            '''when expansion cost and postion must be saved in a queue'''
            cost, (x, y) = queue.get()
            self.maze.grid[x][y].visited = True
            self.path.append((x, y), False)
            # Search one cell on the current level
            # check that the position is visited
            # add current position to total search path

            '''when current position is exit return solution'''
            if (x, y) == self.maze.exit_coor:
                end = time.perf_counter()
                self.total_spead = end - start
                self.total_cost = len(self.path)
                # save the total cost
                # save the total time
                print("Number of moves performed: {}".format(self.total_cost))
                print("Execution time for algorithm: {:.4f}".format(self.total_spead))
                return self.total_cost, self.total_spead, self.path
                #return length of the path found and cost

            '''from current position search neighbour position and find if there is a new position to expand'''
            exapand_xy = self.maze.find_neighbours(x, y)
            exapand_xy = self.maze.validate_neighbours_solve(exapand_xy, x, y, self.maze.exit_coor[0],
                                                            self.maze.exit_coor[1], "brute-force")

            '''where there is a room to exapansion expand and add it to the queue with cost+1'''
            if exapand_xy is not None:
                for xy in exapand_xy:
                    queue.put((cost+1, xy))
```

<code review>

```
queue = PriorityQueue()
queue.put((0, self.maze.entry_coor))
```

in order to perform uniform cost search the queue must be priority queue, which has a data structure of (cost, coordinate).

```
while True:
    while queue:
```

algorithm starts with a two infinity loop until solution is found or search failier loop continues. First loop is entered when starting the search and exits only if the solution is found. The second loop is entered when the queue is expanded. Unless there is no possible node to expand. Then queue will return Boolean false and the while loop will break.

```
cost, (x, y) = queue.get()
self.maze.grid[x][y].visited = True
self.path.append((x, y), False))
```

Inside the loop first we need to extract an element from the queue. In this priority queue element with the smallest cost will be popped.

```
if exapand_xy is not None:
    for xy in exapand_xy:
        queue.put((cost+1,xy))
```

Then we must evaluate the possibility of expansion. If the node is expandable new path needed to be appended in a queue.

```
'''when current position is exit return solution'''
if (x, y) == self.maze.exit_coor:
    end = time.perf_counter()
    self.total_spead = end - start          # save the total cost
    self.total_cost = len(self.path)        # save the total time
    print("Number of moves performed: {}".format(self.total_cost))
    print("Execution time for algorithm: {:.4f}".format(self.total_spead))
    return self.total_cost, self.total_spead, self.path
```

Inside the expansion we must exit the loop when next node contains an exit position. Before exiting the loop save the total cost and time spent for performance check.

## ◆ A\* search code review

```
def a_star_search(self):
    '''use priority queue to sort path by cost
    as we use only cost for sorting the structure inside a queue is a form of (distance, cost, path)'''
    queue = PriorityQueue()
    #using priority queue

    '''in a star search g(x) and h(x) is define as cost of the path and distance between current position to goal position'''
    cost = 0
    distance2goal = numpy.linalg.norm(numpy.array(self.maze.entry_coor)
    - numpy.array(self.maze.exit_coor)) #use distance between goal and current position as heuristic
    queue.put((cost + distance2goal, cost, self.maze.entry_coor)) #initial queue must be set (distance from here to goal, cost, path=entry)

    start = time.perf_counter() #starting time of searching
    print("\nSolving the maze with A_star_search...")

    '''algorithm starts with a two infinity loop until solution is found or search failier loop cotinues'''
    while True: # infinity loop until solution is found and break
        while queue: # infinity loop until no possible way of expansion
            '''when expansion cost and postion must be saved in a queue'''
            cost, (x, y) = queue.get()[1:] # Search one cell on the current level
            self.maze.grid[x][y].visited = True # check that the position is visited
            self.path.append((x, y), False) # add current position to total search path

            '''when current position is exit return solution'''
            if (x, y) == self.maze.exit_coor:
                end = time.perf_counter()
                self.total_spead = end - start # save the total cost
                self.total_cost = len(self.path) # save the total time
                print("Number of moves performed: {}".format(self.total_cost))
                print("Execution time for algorithm: {:.4f}".format(self.total_spead))
                return self.total_cost, self.total_spead, self.path #return length of the path found and cost
```

```
    '''from current position search neighbour position and find if there is a new position to expand'''
    exapand_xy = self.maze.find_neighbours(x, y)
    exapand_xy = self.maze.validate_neighbours_solve(exapand_xy, x,
    y, self.maze.exit_coor[0],
    self.maze.exit_coor[1], "brute-force")

    '''where there is a room to exapansion expand then calculate the distance again and add it to the queue with cost+1'''
    if exapand_xy is not None:
        for xy in exapand_xy:
            distance2goal = numpy.linalg.norm(numpy.array(xy) - numpy.array(self.maze.exit_coor)) #recalculate distance
            queue.put((cost + distance2goal, cost+1, xy)) #add it to the queue
```

<code review>

```
queue = PriorityQueue()
#using priority queue

'''in a star search g(x) and h(x) is define as cost of the path and
cost = 0
distance2goal = numpy.linalg.norm(numpy.array(self.maze.entry_coor)
- numpy.array(self.maze.exit_coor))
queue.put((cost + distance2goal, cost, self.maze.entry_coor))
```

in order to perform uniform cost search the queue must be priority queue, which has a data structure of (distance + cost, cost, coordinate). As the queue is sorted by  $f = g + h$ .  $g(x)$  is the cost of path and  $h(x)$  is heruistic function of Euclidean distance.

```
distance2goal = numpy.linalg.norm(numpy.array(self.maze.entry_coor)
- numpy.array(self.maze.exit_coor))
```

This part is a heuristic function of calculating the of Euclidean distance between the current position and exit position.

```
while True:
    while queue:
```

algorithm starts with a two infinity loop until solution is found or search failier loop continues. First loop is entered when starting the search and exits only if the solution is found. The second loop is entered when the queue is expanded. Unless there is no possible node to expand. Then queue will return Boolean false and the while loop will break.

```
cost, (x, y) = queue.get()
self.maze.grid[x][y].visited = True
self.path.append((x, y), False))
```

Inside the loop first we need to extract an element from the queue. In this priority queue element with the smallest cost will be popped.

```
if expand_xy is not None:
    for xy in expand_xy:
        distance2goal = numpy.linalg.norm(numpy.array(xy) - numpy.array(self.maze.exit_coor))
        queue.put((cost + distance2goal, cost+1, xy))
```

Then we must evaluate the possibility of expansion. If the node has a room to exapansion expand then calculate the distance again and add it to the queue with cost+1

```
if (x, y) == self.maze.exit_coor:
    end = time.perf_counter()
    self.total_spead = end - start # save the total cost
    self.total_cost = len(self.path) # save the total time
    print("Number of moves performed: {}".format(self.total_cost))
    print("Execution time for algorithm: {:.4f}".format(self.total_spead))
    return self.total_cost, self.total_spead, self.path
```

Inside the expansion we must exit the loop when next node contains an exit position. Before exiting the loop save the total cost and time spent for performance check.

## ◆ Critical code difference between uniform cost search and A\* search

### 1. Priority Queue Data structure

<Uniform Cost Search>

```
queue = PriorityQueue()
queue.put((0, self.maze.entry_coor))
```

<A\* Search>

```
queue = PriorityQueue()
| | | | | | | #using priority queue

'''in a star search g(x) and h(x) is define as cost of the path and d
cost = 0
distance2goal = numpy.linalg.norm(numpy.array(self.maze.entry_coor)
| | | | | | | - numpy.array(self.maze.exit_coor))
queue.put((cost + distance2goal, cost, self.maze.entry_coor))
```

In Uniform Cost Search sorts the queue in order of total cost of path, but in A\* search queue is sorted by  $f = g + h$ .  $g(x)$  (cost of path + Euclidean distance)

### 2. expanding the node

<Uniform Cost Search>

```
if exapand_xy is not None:
    for xy in exapand_xy:
        queue.put((cost+1,xy))
```

<A\* Search>



```
if exapand_xy is not None:
    for xy in exapand_xy:
        distance2goal = numpy.linalg.norm(numpy.array(xy) - numpy.array(self.maze.exit_coor))
        queue.put((cost + distance2goal, cost+1, xy))
```

In Uniform Cost Search when expanding and adding the frontier in a queue only cost + 1 is updated but when in A\* search Euclidean distance must be calculated every time.

## ◆ Result

```
# two manager for two algorithm
uniform_cost_manager= MazeManager()
a_star_manager = MazeManager()

performance = []

for i in range(3):

    # for copying the maze I used copy.deepcopy() method in .add_existing_maze
    maze = uniform_cost_manager.add_maze(20, 20)          # add new maze in
    copy_maze = a_star_manager.add_existing_maze(maze)     # copy the same m

    # solve the maze in uniform cost search
    uniform_cost_manager.solve_maze(maze.id, "perfrom_uniform_cost_search")
    uniform_cost_manager.show_solution(maze.id)
    performance.append((i+1, "uniform_cost", maze.cost, maze.time))

    # solve the maze in a star search
    a_star_manager.solve_maze(copy_maze.id, "perfrom_a_star_search")
    a_star_manager.show_solution(copy_maze.id)
    performance.append((" ", "a_star", copy_maze.cost, copy_maze.time))

print("\n")
print("Maze\tAlgorithm\tCost\tTime")
for item in performance:
    print("{}\t{}\t{}\t{}".format(item[0], item[1], item[2], item[3]))
print("\n")
```

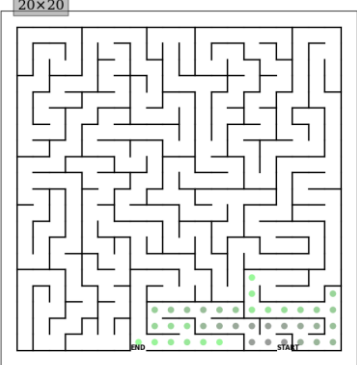
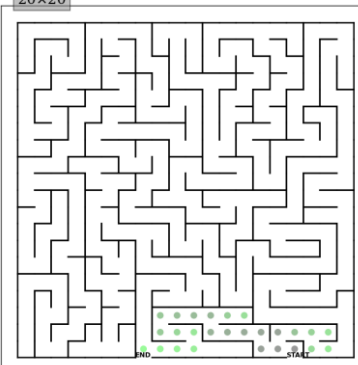
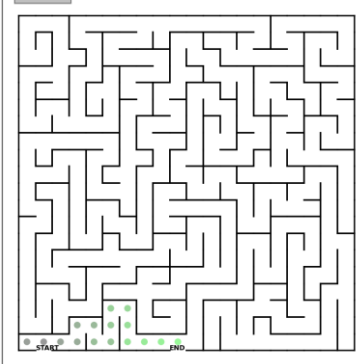
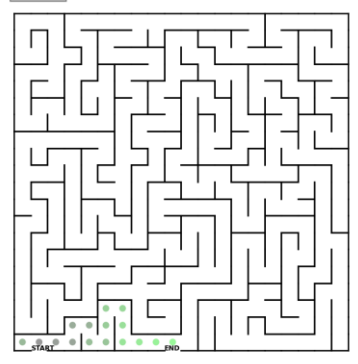
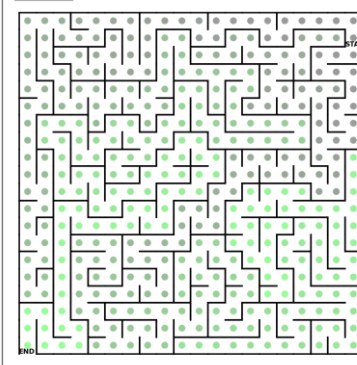
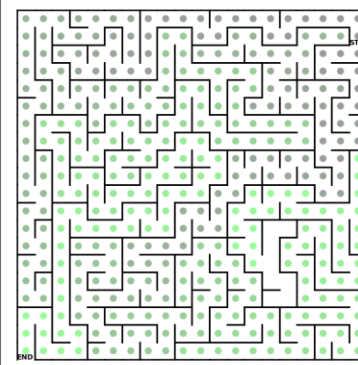
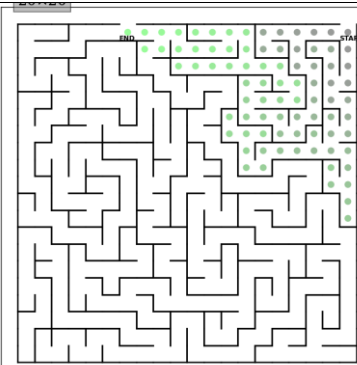
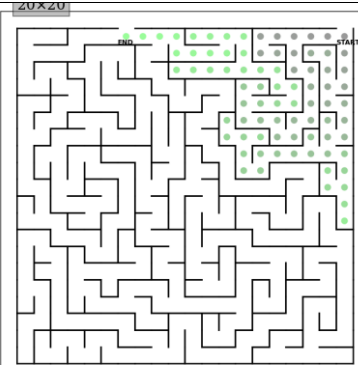
In main.py I made two Maze manager and make these two manager have the same 10 20x20 Maze using copy.deepcopy(). And solving the same maze using two search algorithm I tried to evaluate the performance.

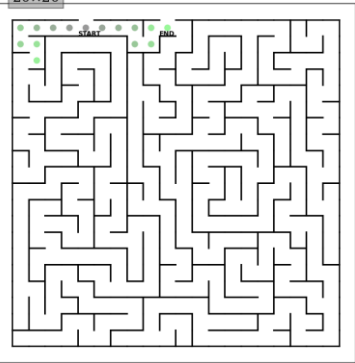
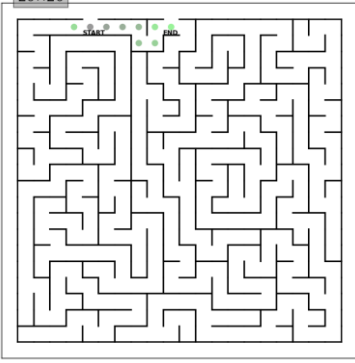
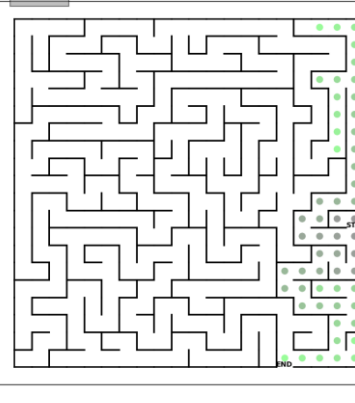
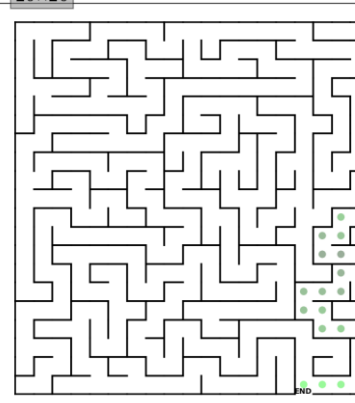
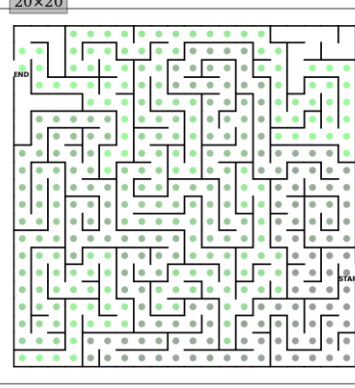
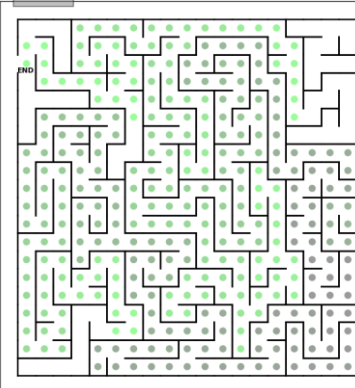
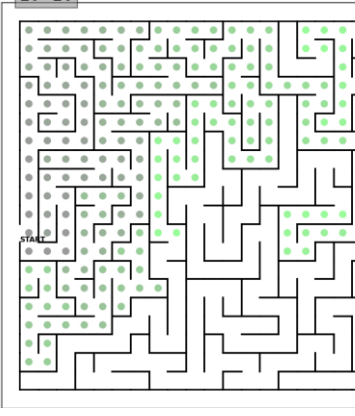
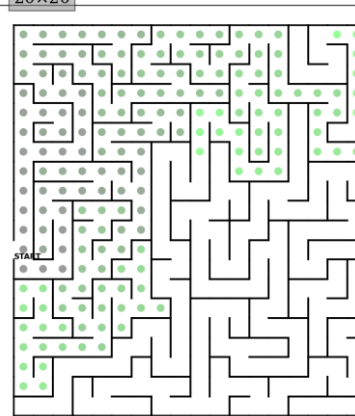
## Visualization of cost and time of algorithm

2	uniform_cost	16	0.0013738000000103057
	a_star	16	0.0017320000000040636
3	uniform_cost	400	0.007729299999994055
	a_star	392	0.030315000000001646
4	uniform_cost	83	0.00309960000000129698
	a_star	81	0.008766100000002552
5	uniform_cost	15	0.00039079999999103165
	a_star	9	0.0012485000000031072
6	uniform_cost	55	0.0036096000000043205
	a_star	35	0.0033038000000026058
7	uniform_cost	377	0.014763700000003155
	a_star	344	0.03250789999999846
8	uniform_cost	232	0.0077715000000046672
	a_star	204	0.0137798999999974532
9	uniform_cost	87	0.011889900000002832
	a_star	82	0.007231799999999566
10	uniform_cost	114	0.00474250000000203
	a_star	101	0.0068235999999961516

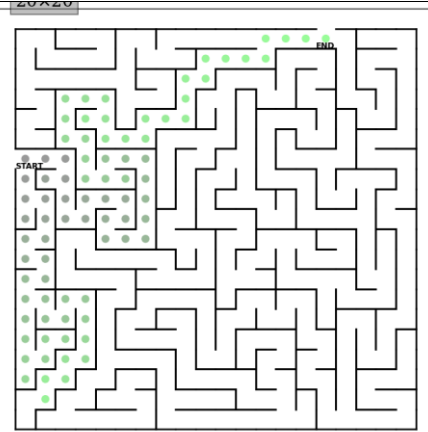
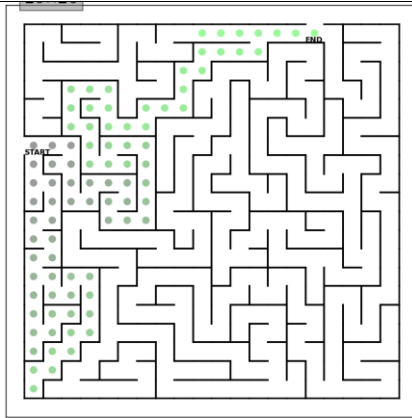
(the result of maze 1 was cut off due to terminal screen size)

# Visualization of solution path

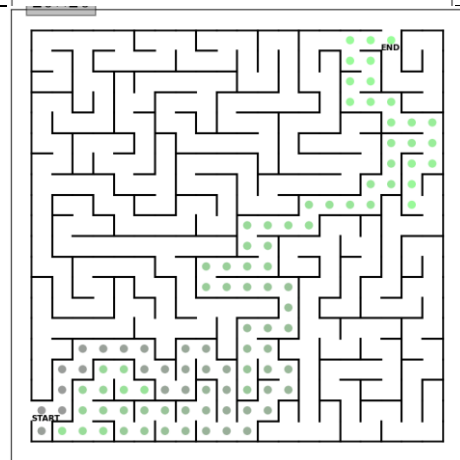
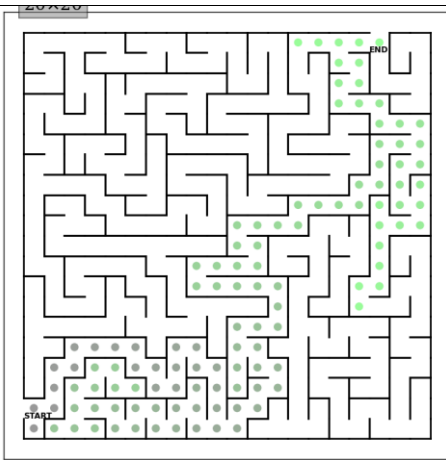
Maze number	<Uniform Cost Search>	<A* Search>
1		
2		
3		
4		

5		
6		
7		
8		

9



10



## ◆ Performance

1. Path length: The goal is to find a route that is shortest. So the algorithm with shorter path as a solution has better performance.

Maze number	<Uniform Cost Search>	<A* Search>	Ratio (uniform/a*)	Winner
2	16	16	1	Draw
3	400	392	1.02	A*
4	83	81	1.02	A*
5	15	9	1.67	A*
6	55	35	1.57	A*
7	377	344	1.09	A*
8	232	204	1.14	A*
9	87	82	1.06	A*
10	114	101	1.13	A*

As in the table beside maze 2, in 8 mazes a\* search has shorter solution path. And to clarify how better performance a\* search has in compare with uniform cost search I calculate the ratio of path length. On average a\* search was about 1.25 times better than uniform cost search

**a\* search is about 1.25 times better than uniform cost search**

2. Total time of searching: Time algorithm takes to find a optimal route is also important. Assume an algorithm that finds the best route but takes a day to find the route. It might be better to get a less optimal route if it takes less time to find.

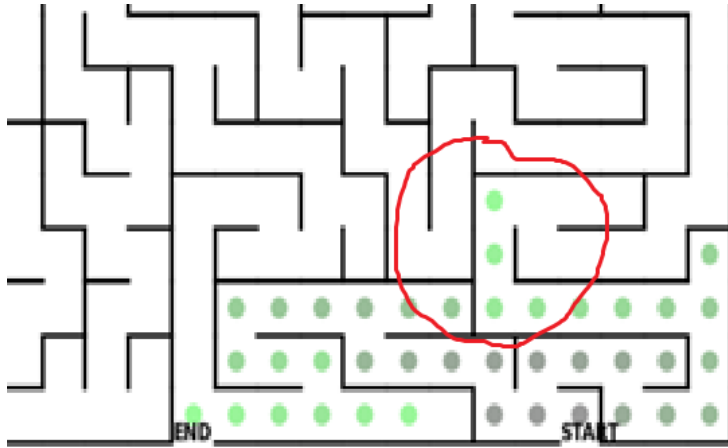
Maze number	<Uniform Cost Search>	<A* Search>	Ratio (a* / uniform)	Winner
2	137	172	1.25	Uniform Cost
3	772	3031	3.92	Uniform Cost
4	309	876	2.83	Uniform Cost
5	390	1248	3.2	Uniform Cost
6	3609	3303	0.91	A*
7	1476	3250	2.20	Uniform Cost
8	7771	13779	1.77	Uniform Cost
9	11889	7231	0.60	A*
10	4742	6823	1.44	Uniform Cost

As in table beside two maze this time uniform cost is the winner. Uniform cost search calculate the solution much faster. And to clarify how much faster uniform cost search calculate the solution I calculate the ratio of time. On average a\* search was about 2 times better than uniform cost search.

uniform cost search is about 2 times faster than A\* search

### 3. Detail comparison

#### <Uniform Cost Search>



#### <A\* Search>



This picture is a part of Maze 1. As we can see in these picture the solution path of A\* search has less backtracking, so having a shorter solution path.

### 4. Overall performance

A\* search has a better performance than Uniform cost search. A\* search find a 1.25 times shorter path. Uniform cost search find the answer 2 times faster then A\* search but both algorithm finds the solution in less then a millisecond so this ratio has less effect in overall performance



## ◆ Performance improvement attempt

### 1. using a faster function for Euclidean distance

In a A\* search algorithm Euclidean distance has to be calculated in every expansion.

```
queue = PriorityQueue()
| | | | | | | #using priority queue

'''in a star search g(x) and h(x) is define as cost of the path and d
cost = 0
distance2goal = numpy.linalg.norm(numpy.array(self.maze.entry_coor)
| | | | | | | - numpy.array(self.maze.exit_coor))
queue.put((cost + distance2goal, cost, self.maze.entry_coor))
```

As we can see in the code in every expansion from the frontier has to calculate the Euclidean distance from the current position to exit position so I use a faster calculation model.

<slow calculation>

```
distance2goal = math.sqrt((self.maze.entry_coor)** 2 + (self.maze.exit_coor) ** 2)
```

<faster calculation>

```
distance2goal = numpy.linalg.norm(numpy.array(xy) - numpy.array(self.maze.exit_coor))
```

While searching for a faster calculation I found an information about uinpy model which calculate distance from two point faster than math model so I implement the code.

## 2. optimizing the heuristic function

When I tried to find a way to optimize a heuristic function, google gives a multiple ways to optimize a heuristic function. These are the ways.

- Typically, using heuristic functions such as Euclidean distance, Manhattan distance, etc. are standard methods for validity checks. However, adjusting the heuristic function to fit specific problems can further improve search efficiency.
- Pre-computing and storing the shortest distance between the destination point and all other points is recommended. This can speed up A\* search as there is no need to recalculate the heuristic function each time.

I tried the first way adjusting the heuristic function to fit specific problems can further improve search efficiency.

The way I tried is to change the ratio of Euclidean distance in  $f(x)$ . The formula I used is like below.

$$f(x) = \alpha g(x) + (1-\alpha) h(x)$$

By alpha the power of distance affects  $f(x)$  change. And I tried different alpha value.

```
def a_star_search(self):
    '''use priority queue to sort path by cost
    as we use only cost for sorting the structure inside a queue is a form of (distance, cost)
    queue = PriorityQueue() #using priority queue

    alpha = 0.2 # value for optimization

    '''in a star search g(x) and h(x) is define as cost of the path and distance between
    cost = 0
    distance2goal = numpy.linalg.norm(numpy.array(self.maze.entry_coors)
    - numpy.array(self.maze.exit_coors)) #use distance
    queue.put((cost*alpha + distance2goal*(1-alpha), cost, self.maze.entry_coors))
    here to goal, cost, path=entry)
```

### 3. expand the neighbors more smartly

As we expand the path from frontier nodes there are two methods in the file given one is "brute-force" and the other is "fancy" method. 'brute-force' chooses neighbors randomly. And "fancy" next cell is chosen based on which neighbor that gives the shortest distance to the final destination. So I tried to use the smart way expecting a faster solution.

```
def validate_neighbours_solve(self, neighbour_indices, k, l, k_end, l_end, method):
    """Function that validates whether a neighbour is unvisited or not and discards the
    neighbours that are inaccessible due to walls between them and the current cell. The
    function implements two methods for choosing next cell; one is 'brute-force' where one
    of the neighbours are chosen randomly. The other is 'fancy' where the next cell is ch
    based on which neighbour that gives the shortest distance to the final destination.

    Args:
        neighbour_indices
        k
        l
        k_end
        l_end
        method

    Return:

    """
    neigh_list = list()

    if method == "fancy":
        neigh_list = list()
        min_dist_to_target = 100000

        for k_n, l_n in neighbour_indices:
            if (not self.grid[k_n][l_n].visited
                and not self.grid[k][l].is_walls_between(self.grid[k_n][l_n])):
                dist_to_target = math.sqrt((k_n - k_end) ** 2 + (l_n - l_end) ** 2)

                if (dist_to_target < min_dist_to_target):
                    min_dist_to_target = dist_to_target
                    min_neigh = (k_n, l_n)

            if "min_neigh" in locals():
                neigh_list.append(min_neigh)

        elif method == "brute-force":
            neigh_list = [n for n in neighbour_indices if not self.grid[n[0]][n[1]].visited
                          and not self.grid[k][l].is_walls_between(self.grid[n[0]][n[1]])]

    if len(neigh_list) > 0:
        return neigh_list
    else:
        return None
```

<expanding in random direction>

```
exapand_xy = self.maze.find_neighbours(x, y)
exapand_xy = self.maze.validate_neighbours_solve(exapand_xy, x,
```

<expanding toward the exit point>

```
exapand_xy = self.maze.find_neighbours(x, y)
exapand_xy = self.maze.validate_neighbours_solve(exapand_xy, x,
```

## ◆ Performance improvement result

### 1. using a faster function for Euclidean distance

Using a faster function has helped in calculation speed performance. But not a stark difference.

<slow calculation result >

Maze	Algorithm	Cost	Time
1	uniform_cost	186	0.0036399000000000292
	a_star	144	0.004041299999999914
2	uniform_cost	313	0.005052700000000021
	a_star	292	0.007199600000000084
3	uniform_cost	273	0.004280600000000019
	a_star	264	0.009259700000000093

<faster calculation result>

Maze	Algorithm	Cost	Time
1	uniform_cost	10	0.00037960000000003546
	a_star	6	0.000410900000000010285
2	uniform_cost	133	0.0012095999999999218
	a_star	115	0.0018910999999999234
3	uniform_cost	344	0.007321899999999992
	a_star	344	0.015989299999999984

## 2. optimizing the heuristic function

I tried 4 different variable for alpha. 0.2, 0.4, 0.6, 0.8. And the result is below.

<alpha = 0.2>

Maze	Algorithm	Cost	Time
1	uniform_cost	246	0.0019575999999998928
	a_star	229	0.004023899999999969
2	uniform_cost	269	0.006298899999999996
	a_star	229	0.009932999999999997
3	uniform_cost	261	0.004897099999999988
	a_star	218	0.009056100000000011

<alpha = 0.4>

Maze	Algorithm	Cost	Time
1	uniform_cost	137	0.0022693000000000296
	a_star	106	0.004747500000000127
2	uniform_cost	246	0.003346500000000141
	a_star	224	0.008688699999999994
3	uniform_cost	244	0.0028516999999999015
	a_star	235	0.010600600000000071

<alpha = 0.6>

Maze	Algorithm	Cost	Time
1	uniform_cost	137	0.0022693000000000296
	a_star	106	0.004747500000000127
2	uniform_cost	246	0.003346500000000141
	a_star	224	0.008688699999999994
3	uniform_cost	244	0.0028516999999999015
	a_star	235	0.010600600000000071

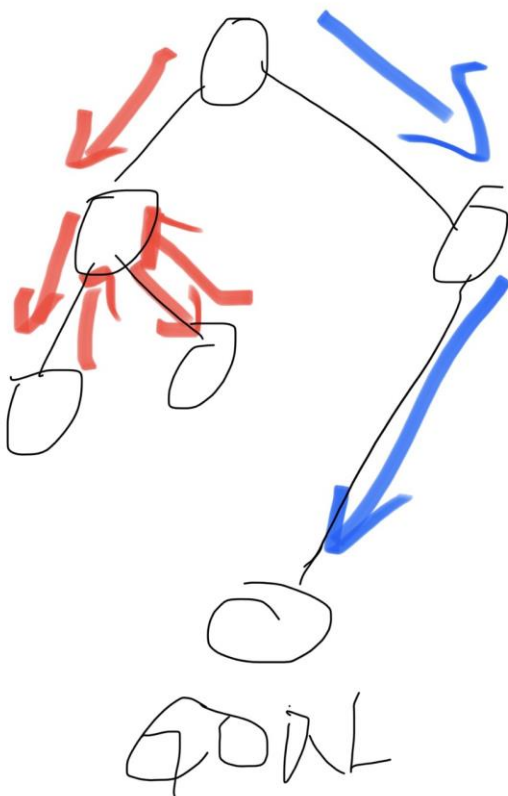
<alpha = 0.8>

Maze	Algorithm	Cost	Time
1	uniform_cost	283	0.006071300000000113
	a_star	277	0.007464099999999735
2	uniform_cost	399	0.0054090999999998335
	a_star	399	0.014279799999999954
3	uniform_cost	212	0.007673100000000099
	a_star	203	0.009572799999999937

And by comparing these four result I found that the most dramatic difference was when alpha is 0.2. As you can see the cost difference between uniform cost search is a lot difference from other alpha value so the best optimization for heuristic fuction is when alpha is 0.2.

Not like the expectation "fancy" way of expanding the node actually fails to give an answer. When the algorithm has succeeded in getting the solution it calculate faster but many time it fails to give an answer.

Greedy search tries to expand the node that is closest to the goal. It is similar to "fancy" method. it is likely to lead to a solution quickly. But there is a problem. It can be leaded into a infinite loop.



As it is seen in the picture when the node is expanded in a red direction it stays in a infinite loop. Or gives a much longer solution path and a slower solution calculation speed.