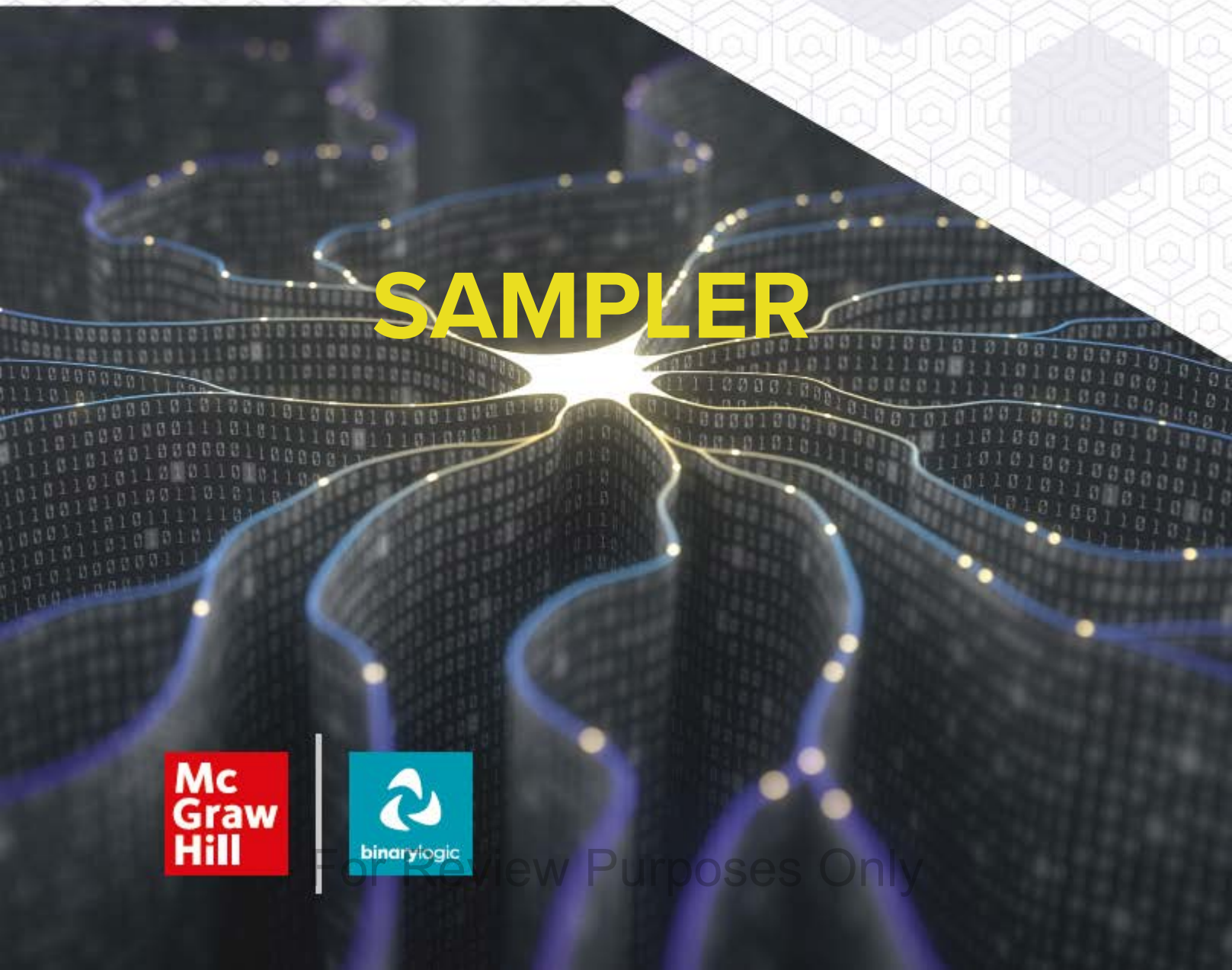
Foundations of AI

Artificial Intelligence 2



SAMPLER

Mc Graw Hill

binarylogic

For Review Purposes Only

Foundations of AI

Artificial Intelligence 2

For Review Purposes Only

Foundations of AI:
Artificial Intelligence 2

Printed and distributed by McGraw Hill in association with Binary Logic SA.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without prior written permission from the publishers. No part of this work may be used or reproduced in any manner for the purpose of training artificial intelligence technologies or systems.

Disclaimer: McGraw Hill is an independent entity from Microsoft® Corporation and is not affiliated with Microsoft Corporation in any manner. Any Microsoft trademarks referenced herein are owned by Microsoft and are used solely for editorial purposes. This work is in no way authorized, prepared, approved, or endorsed by, or affiliated with, Microsoft.

Please note: This book contains links to websites that are not maintained by the publishers. Although we make every effort to ensure these links are accurate, up-to-date, and appropriate, the publishers cannot take responsibility for the content, persistence, or accuracy of any external or third-party websites referred to in this book, nor do they guarantee that any content on such websites is or will remain accurate or appropriate.

Trademark notice: Product or corporate names mentioned herein may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe. The publishers disclaim any affiliation, sponsorship, or endorsement by the respective trademark owners.

Windows is a registered trademark of Microsoft Corporation. Tinkercad is a registered trademark of Autodesk Inc. “Python” and the Python logos are registered trademarks of Python Software Foundation. Jupyter is a registered trademark of Project Jupyter. CupCarbon is a registered trademark of CupCarbon. OpenCV is a registered trademark of Open Source Vision Foundation. Ultimaker Cura is a trademark of PIT Ultimaker Holding B.V. FreeCAD is a trademark of FreeCAD Project Association. The above companies or organizations do not sponsor, authorize, or endorse this book, nor is this book affiliated with them in any way.

Cover Credit: © ktsdesign/123rf

Copyright © 2026 Binary Logic SA MHID: 1265871787 ISBN: 9781265871789

mheducation.com binarylogic.net

For Review Purposes Only

Contents

For Review Purposes Only

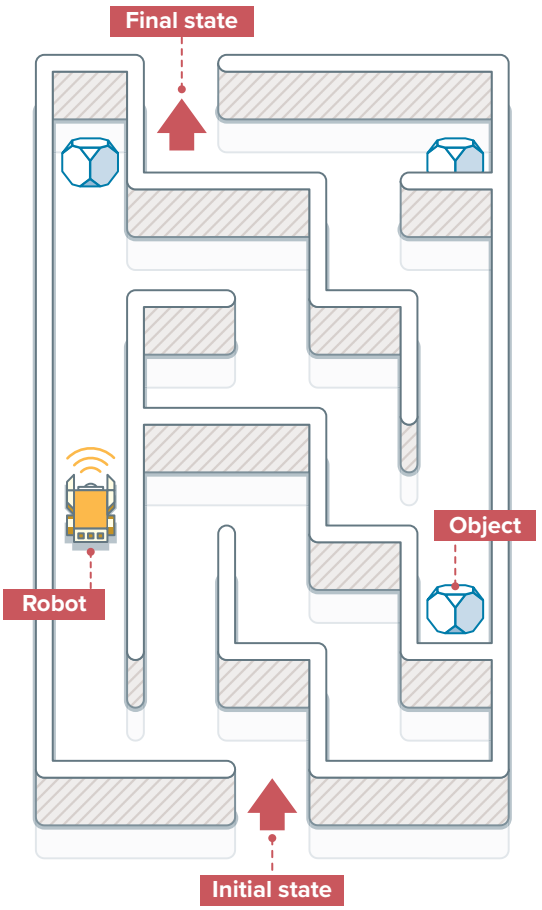
LESSON 3

Informed Search Algorithms

Applications of Search Algorithms

Search algorithms are key components of AI systems, enabling the exploration of various possibilities to find solutions to complex problems, with numerous mainstream applications. Some examples of their applications include:

- **Robotics**: A robot might use a search algorithm to find its way through a maze or to locate an object in its environment.
- **E-commerce websites**: Online shopping websites use search algorithms to match customers' queries with available products, filter results based on criteria such as price, brand, and ratings, and suggest related products.
- **Social media platforms**: Social media platforms use search algorithms to present users the most relevant posts, people, and groups based on keywords and user interests.
- **Enabling a machine to play games at a high level of skill**: A chess or Go-playing AI might use a search algorithm to evaluate different moves and choose the one that is most likely to lead to a win.
- **GPS navigation systems**: GPS navigation systems use search algorithms to find the shortest and fastest route between two locations, taking into account real-time traffic data.
- **File management systems**: Search algorithms are used in file management systems to quickly locate specific files based on their names, contents, or other attributes.



Types and examples of search algorithms

There are two main types of search algorithms: **uninformed** and **informed**.

Uninformed search algorithms

Uninformed search algorithms, also known as blind search algorithms, have no additional information about the states of a problem beyond those provided in the problem definition and perform an exhaustive search of the search space by following a predetermined set of rules. The Breadth-first search (BFS) and Depth-first search (DFS) techniques are examples of uninformed search algorithms.

For Review Purposes Only

For example, DFS begins at the root node of a tree or graph and always expands to the deepest unvisited node. It proceeds in this manner until it has exhausted the entire search space by visiting all available nodes. It then reports the best solution that was found during the search. The fact that DFS always follows these rules and does not adjust its strategy regardless of what it discovers during its search makes it an uninformed algorithm.

Another notable example in this family is Iterative Deepening Depth-first Search (IDDFS), which can be viewed as a combination of the DFS and BFS algorithms, as it uses a depth-first strategy to iteratively explore the full breadth of options up to a certain node.

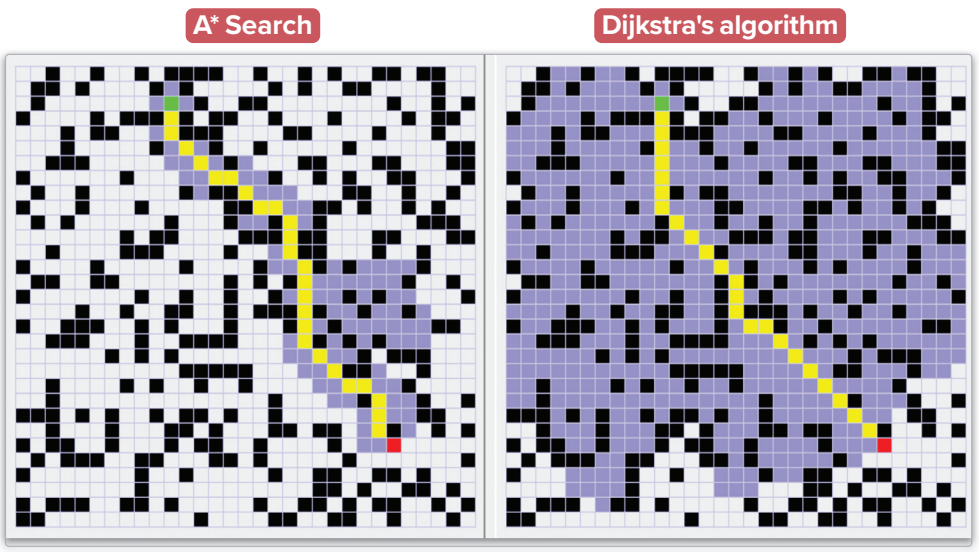
Informed search algorithms

In contrast to the uninformed search algorithms, **informed search** algorithms use information about the problem and the search space to guide their search. Examples of such algorithms include:

- **A* Search**, which uses a **heuristic function** to estimate the distance between each of the candidate nodes and the goal node. It then expands the candidate node with the lowest estimate. The A* Search algorithm is as good as its heuristic function. For example, if the heuristic is guaranteed never to overestimate the actual distance to the goal, then the algorithm is guaranteed to find the optimal solution. Otherwise, the returned solution might not be the best possible one.
- **Dijkstra's algorithm**, which expands the node with the actual lowest distance to the goal in every step. Therefore, contrary to A* Search, Dijkstra actually computes the real distance and does not use heuristic estimates. While this makes Dijkstra slower than A* Search, it also means that it is always guaranteed to find the optimal solution (the shortest path from the start to the goal).
- **Hill climbing**, which starts by generating a random solution. It then tries to iteratively improve this solution by making small changes that increase a specific **heuristic function**. Even though this approach is not guaranteed to find the optimal solution, it is easy to implement and can be very efficient for certain types of problems.

Heuristic function

A function that ranks alternatives in search algorithms at each branching stage depending on available data to choose which branch to pursue.



The purple cells are the visited cells, the green cell is the start location, the red cell is the finish location and the yellow cells represent the found route.

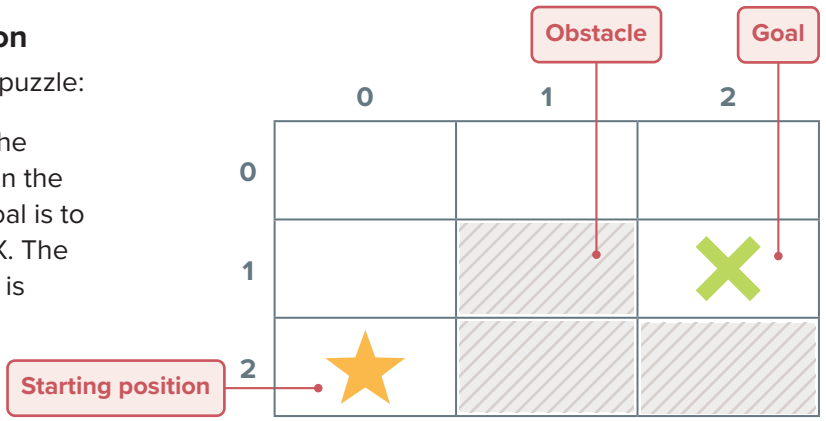
For Review Purposes Only

In this lesson, you will explore some visual examples and Python implementations of BFS and A* Search to demonstrate the differences between informed and uninformed search algorithms.

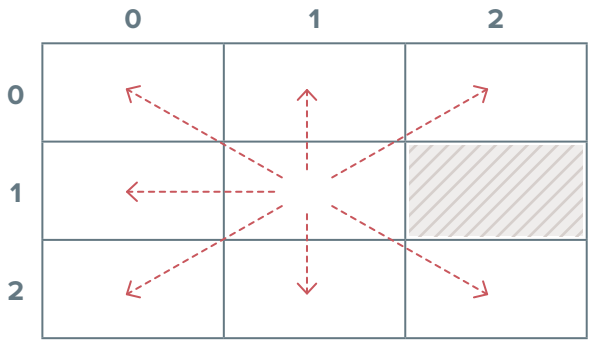
Creating maze puzzles in Python

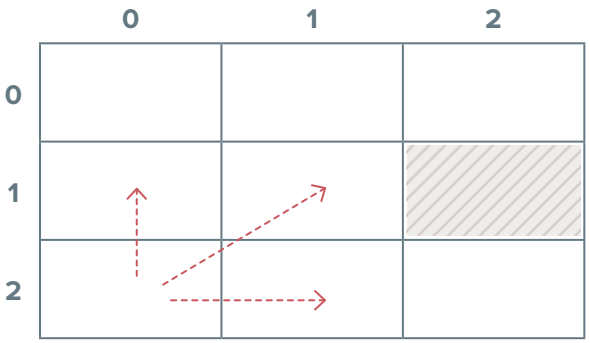
Consider the following simple maze puzzle:

The maze is defined as a 3×3 grid. The starting position is marked by a star in the lower left corner of the maze. The goal is to reach the target cell marked by the X. The player can move to any free cell that is adjacent to their current position.



A cell is considered free unless it is already occupied by a block. For example, the above maze example has three cells occupied by blocks. These blocks are colored dark gray and form an obstacle that the player has to circumnavigate to get to the X. The player can move to any horizontally, vertically, or diagonally adjacent free cell. For example:





The objective is to find the shortest possible path and find it with the smallest possible number of cell visits. Even though a small 3×3 maze might seem trivial to a human player, any intelligent algorithmic solution has to work for arbitrarily large and complex mazes. For example, consider a massive 10,000×10,000 maze with millions of blocks scattered in various complex shapes.

The following Python code can be used to create a dataset that represents the example of the 3×3 grid.

```
import numpy as np

# create a numeric 3×3 matrix full of zeros.
small_maze=np.zeros((3,3))

# coordinates of the cells occupied by blocks
blocks=[(1, 1), (2, 1), (2, 2)]

for block in blocks:
    # set the value of block-occupied cells to be equal to 1
    small_maze[block]=1

small_maze
```

```
array([[0., 0., 0.],
       [0., 1., 0.],
       [0., 1., 1.]])
```

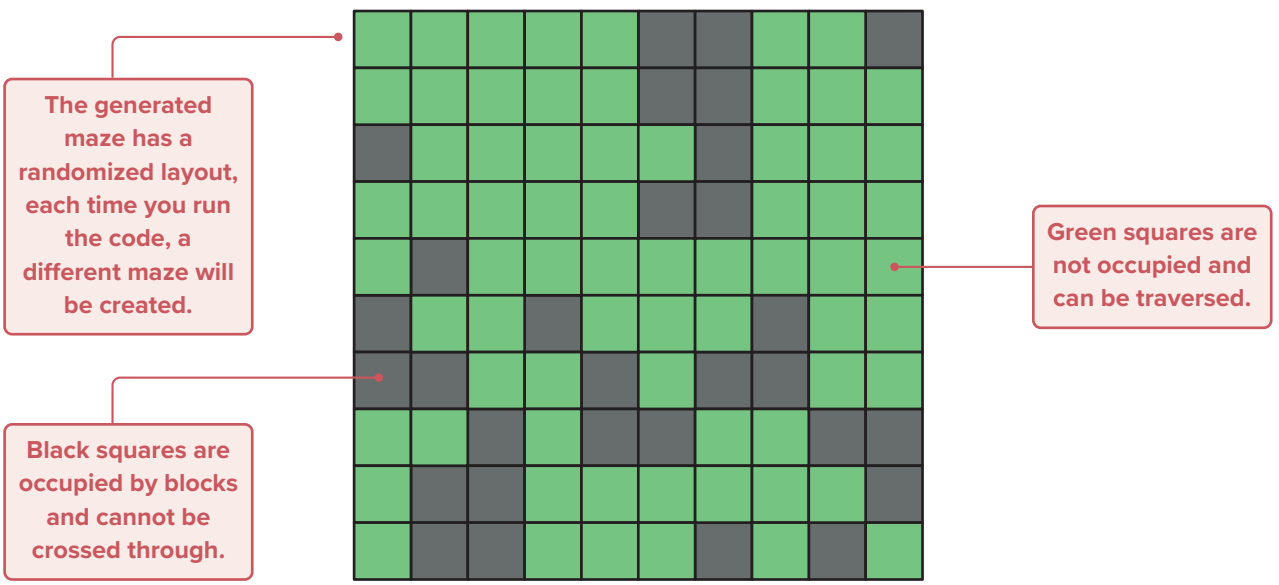
For Review Purposes Only

In this numeric representation of a maze, free and occupied cells are represented by zeros and ones, respectively. The same code can also be easily updated to create arbitrarily large and complex mazes. For example:

```
import random

random_maze=np.zeros((10,10))

# coordinates of 30 random cells occupied by blocks
blocks=[(random.randint(0,9),random.randint(0,9)) for i in range(30)]

for block in blocks:
    random_maze[block]=1
```

The following function can be used to visualize a maze:

```
import matplotlib.pyplot as plt # library used for visualization

def plot_maze(maze):
    ax = plt.gca()          # create a new figure
    ax.invert_yaxis()       # invert the y-axis to match the matrix
    ax.axis('off')          # hide the axis labels
    ax.set_aspect('equal') # make sure the cells are square

    plt.pcolormesh(maze, edgecolors='black', linewidth=2,cmap='Accent')
    plt.show()

plot_maze(random_maze)
```



For Review Purposes Only

Given any such maze, the following function can be used to return a list with all the adjacent accessible, empties and neighbors of a specific cell:

```
def get_accessible_neighbors(maze:np.ndarray, cell:tuple):

 # list of accessible neighbors, initialized to empty
    neighbors=[]

    x,y=cell

    # for each adjacent cell position
    for i,j in [(x-1,y-1),(x-1,y),(x-1,y+1),(x,y-1),(x,y+1),(x+1,y-1),(x+1,y),
(x+1,y+1)]:

        # if the adjacent cell is within the bounds of the grid and is not occupied by a block
        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and maze[(i,j)]==0:

            neighbors.append(((i,j),1))

    return neighbors
```

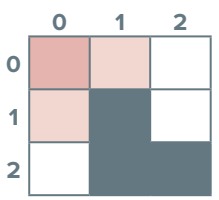
x-1, y-1	x-1, y	x-1, y+1
x, y-1	**x, y**	x, y+1
x+1, y-1	x+1, y	x+1, y+1

This implementation assumes that all possible transitions from a cell to any horizontally, vertically, or diagonally adjacent neighbor have the same cost of 1. This assumption will be revisited later in this lesson, to allow for more complex scenarios with variable transition costs.

The **get_accessible_neighbors()** function is required by any search algorithm that attempts to solve the maze. The following examples use the small 3×3 maze created above to verify that the function indeed returns the correct neighbors for a given cell.

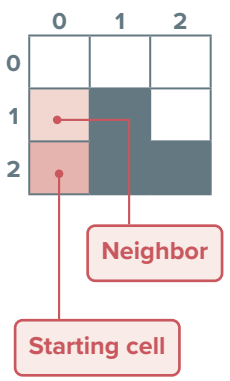
```
# this cell is the northwest corner of the grid and has only 2 accessible
neighbors
get_accessible_neighbors(small_maze, (0,0))
```

```
[((0, 1), 1), ((1, 0), 1)]
```




```
# the starting cell (in the southwest corner) has only 1 accessible neighbor
get_accessible_neighbors(small_maze, (2,0))
```

```
[((1, 0), 1)]
```



Given the ability to create mazes and to also retrieve the neighbors of any cell in a maze, the next step is to implement search algorithms that can solve a maze by finding the shortest path from a given start cell to a given target cell.

For Review Purposes Only

Using BFS to Solve Maze Puzzles

The **bfs_maze_solver()** function described in this section uses Breadth-first-search to solve maze puzzles with a start and target cell. This implementation utilizes the **get_accessible_neighbors()** function defined above to retrieve the neighboring cells that can be visited at any point during the search.

Once BFS has found the target cell, the **reconstruct_shortest_path()** function in the following code, is used to reconstruct and return the shortest path, working backward from target to start:

```
def reconstruct_shortest_path(parent:dict, start_cell:tuple, target_cell:tuple):

    shortest_path = []

    my_parent=target_cell # start with the target_cell

    # keep going from parent to parent until the search cell has been reached
    while my_parent!=start_cell:

        shortest_path.append(my_parent) # append the parent

        my_parent=parent[my_parent] # get the parent of the current parent

    shortest_path.append(start_cell) # append the start cell to complete the path

    shortest_path.reverse() # reverse the shortest path

    return shortest_path
```

The same **reconstruct_shortest_path()** function will be used to reconstruct the solution for the A* Search algorithm described later in this lesson. Given the definitions of the **get_accessible_neighbors()** and **reconstruct_shortest_path()** helper functions, the **bfs_maze_solver()** function can be implemented as follows:

```
from typing import Callable # used to call a function as an argument of another function

def bfs_maze_solver(start_cell:tuple,
                    target_cell:tuple,
                    maze:np.ndarray,
                    get_neighbors: Callable,
                    verbose:bool=False): # by default, suppresses descriptive output text

    cell_visits=0 # keeps track of the number of cells that were visited during the search
    visited = set() # keeps track of the cells that have already been visited
    to_expand = [] # keeps track of the cells that have to be expanded

    # add the start cell to the two lists
    visited.add(start_cell)
    to_expand.append(start_cell)
    # remembers the shortest distance from the start cell to each other cell
    shortest_distance = {}
```

For Review Purposes Only

```
    # the shortest distance from the start cell to itself, zero
    shortest_distance[start_cell] = 0

    # remembers the direct parent of each cell on the shortest path from the start_cell to the cell
    parent = {}
    # the parent of the start cell is itself
    parent[start_cell] = start_cell

    while len(to_expand)>0:

        next_cell = to_expand.pop(0) # get the next cell and remove it from the expansion list

        if verbose:
            print('\nExpanding cell', next_cell)

        # for each neighbor of this cell
        for neighbor,cost in get_neighbors(maze, next_cell):

            if verbose:
                print('\tVisiting neighbor cell',neighbor)

            cell_visits+=1

            if neighbor not in visited: # if this is the first time this neighbor is visited

               visited.add(neighbor)
               to_expand.append(neighbor)
               parent[neighbor]= next_cell
               shortest_distance[neighbor]=shortest_distance[next_cell] + cost

                # target reached
             if neighbor==target_cell:

                # get the shortest path to the target cell, reconstructed in reverse.
                shortest_path = reconstruct_shortest_path(parent,
                start_cell, target_cell)

                 return shortest_path, shortest_distance[target_cell],cell_visits

            else: # this neighbor has been visited before

                # if the current shortest distance to the neighbor is longer than the shortest distance
to next_cell plus the cost of transitioning from next_cell to this neighbor
                if shortest_distance[neighbor]>shortest_distance[next_cell] + cost:

                   parent[neighbor]=next_cell
                   shortest_distance[neighbor]=shortest_distance[next_cell]+cost

    # search complete but the target was never reached, no path exists
    return None,None,None
```

For Review Purposes Only

The function follows the standard BFS approach of exploring all options at the current depth prior to moving to the next depth level. This implementation uses a set called **visited** and a list **called to_expand**.

The first includes all cells that have been visited at least once by the algorithm. The second list includes all the cells that have not yet been expanded, which means that their neighbors have not been visited yet. The algorithm also uses two dictionaries **shortest_distance** and **parent**. The first one maintains the length of the shortest path from the start cell to each other cell, while the second one remembers the parent of the cell on this shortest path.

Once the target cell has been reached and the search is complete, **shortest_distance[target_cell]** will include the length of the solution: the length of the shortest path from start to target.

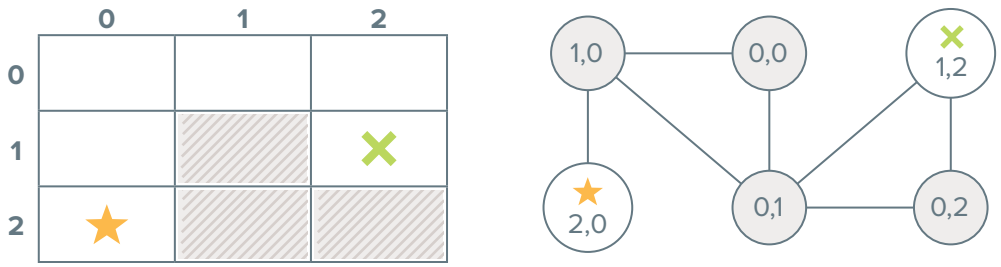
The following code uses the **bfs_maze_solver()** function to solve the small 3×3 maze defined above:

```
start_cell=(2,0)  # start cell, marked by a star in the 3×3 maze
target_cell=(1,2)  # target cell, marked by an "X" in the 3×3 maze

solution, distance, cell_visits=bfs_maze_solver(start_cell,
                                        target_cell,
                                        small_maze,
                                        get_accessible_neighbors,
                                        verbose=True)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Expanding cell (2, 0)
      Visiting neighbor cell (1, 0)

Expanding cell (1, 0)
      Visiting neighbor cell (0, 0)
      Visiting neighbor cell (0, 1)
      Visiting neighbor cell (2, 0)

Expanding cell (0, 0)
      Visiting neighbor cell (0, 1)
      Visiting neighbor cell (1, 0)

Expanding cell (0, 1)
      Visiting neighbor cell (0, 0)
      Visiting neighbor cell (0, 2)
      Visiting neighbor cell (1, 0)
      Visiting neighbor cell (1, 2)

Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 3
Number of cell visits: 10
```

For Review Purposes Only

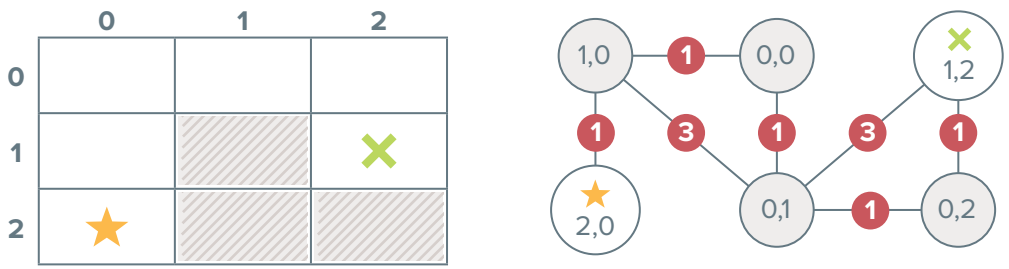
BFS successfully finds the shortest path after 10 cells visits. The search process followed by BFS can be more easily visualized if one considers a graph-based representation of the maze. Consider the following example of a simple 3×3 maze and its graph representation:



The graph representation includes one node for every non-occupied cell. The label on the nodes includes the coordinates of the corresponding matrix cell. There is an undirected edge from one node to another if their corresponding cells are accessible from each other.

One important observation about BFS is that, for **unweighted graphs**, the first path that it finds between the start cell and any other cell is guaranteed to be the one that includes the smallest number of visited cells. This means that, as long as all edges on the graph have the same weight (or, equivalently, that all transitions from one cell to another have the same cost), then the first path found to a specific node is guaranteed to be the shortest path to that node. This is why the **bfs_maze_solver()** stops the search and returns the result the first time it visits the target node.

However, this approach does not work for **weighted graphs**. Consider the following weighted version of the graph representation for the 3×3 maze:



In this example, all edges that correspond to vertical or horizontal moves (south, north, west, east) have a weight equal to 1. However, all edges that correspond to diagonal moves (southwest, southeast, northwest, northeast), have a weight equal to 3. In this weighted case, the shortest path is clearly [(2,0), (1,0), (0,0), (0,1), (0,2), (1,2)], which has a total distance of 1+1+1+1+1=5.

This more complex scenario can be encoded via the weighted version of the **get_accessible_neighbors()** function that is described below.

```
def get_accessible_neighbors_weighted(maze:np.ndarray,
                                      cell:tuple,
                                      horizontal_vertical_weight:float,
                                      diagonal_weight:float):
```

For Review Purposes Only

```
    neighbors=[]
    x,y=cell

    for i,j in [(x-1,y-1), (x-1,y+1), (x+1,y-1), (x+1,y+1)]: # for diagonal neighbors

        # if the cell is within the bounds of the grid and it is not occupied by a block
        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and maze[(i,j)]==0:

            neighbors.append(((i,j), diagonal_weight))

    for i,j in [(x-1,y), (x,y-1), (x,y+1), (x+1,y)]: # for horizontal and vertical neighbors

        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and maze[(i,j)]==0:

            neighbors.append(((i,j), horizontal_vertical_weight))

    return neighbors
```

This function allows the user to assign a custom weight for horizontal and vertical moves, and a different custom weight for diagonal moves. If this weighted version is then used by the BFS solver, the results are as follows:

```
from functools import partial

start_cell=(2,0)
target_cell=(1,2)
horz_vert_w=1  # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution, distance, cell_visits=bfs_maze_solver(start_cell,
                                        target_cell,
                                        small_maze,
                                        partial(get_accessible_neighbors_weighted,
                                              horizontal_vertical_weight=horz_vert_w,
                                               diagonal_weight=diag_w),
                                        verbose=False)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 7
Number of cell visits: 6
```

For Review Purposes Only

As expected, the BFS solver mistakenly reports the exact same path as before, even though it has a distance of 7 and is clearly not the shortest path. This is due to the uninformed nature of the BFS algorithm, in which BFS does not take the weights into account when deciding which cell to expand next. It simply applies the same breadth-first approach, which leads to the exact same solution that the algorithm found for the unweighted version of the maze.

The next section describes how this weakness can be addressed via A* Search, an informed and more intelligent search algorithm that adjusts its behavior based on the specified weights, and can therefore solve mazes with both weighted and unweighted transitions.

Using A* Search to Solve Maze Puzzles

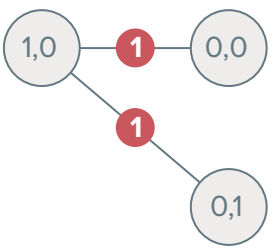
Similar to BFS, A* Search expands one cell at a time, by visiting each of its accessible neighbors. However, while BFS uses a blind breadth-first approach to decide which cell to expand next, A* Search expands the cell with the smallest estimated distance to the target cell, as computed by a heuristic function.

The exact definition of the heuristic function depends on the application. For maze puzzles, a good heuristic would provide an accurate estimate of how close a candidate cell is to the target. As long as the employed heuristic is guaranteed to never overestimate the actual distance to the target (for example, provide an estimate that is higher than the actual distance to the target), then the algorithm is guaranteed to find the shortest possible path for both weighted and unweighted graphs. If a heuristic sometimes overestimates distances, then A* Search will still return a solution, but it might not be the best one possible.

The simplest possible heuristic that is guaranteed to never lead to overestimation is a simple function that always produces an estimated distance of 1:

```
def constant_heuristic(candidate_cell:tuple, target_cell:tuple):

    return 1
```

While this is clearly an overly optimistic heuristic, it will never produce an estimate that is higher than the actual distance, and will therefore lead to the best possible solution. A more sophisticated heuristic that finds the best solution much faster will be introduced later in this section.



The following function uses a given heuristic function to find the cell that should be expanded next:

```
def get_best_candidate(expansion_candidates:set,
                       shortest_distance:dict,
                       heuristic:Callable):

    winner = None
    # best (lowest) distance estimate found so far. Initialized to a very large number
    best_estimate= sys.maxsize
    for candidate in expansion_candidates:

        # distance estimate from start to target, if this candidate is expanded next
```

For Review Purposes Only

```
        candidate_estimate=shortest_distance[candidate]
+heuristic(candidate,target_cell)
        if candidate_estimate < best_estimate:

            winner = candidate
            best_estimate=candidate_estimate

    return winner
```

The above implementation utilizes a **for** loop to iterate over all the candidates in the set and find the best one. A more efficient implementation could use a priority queue that can produce the best candidate without having to iterate over all candidates.

The **get_best_candidate()** function is used as a helper module by the **astar_maze_solver()** function presented next. In addition to the heuristic function, this implementation also uses the **get_accessible_neighbors_weighted()** and **reconstruct_shortest_path()** helper functions defined in the previous section.

```
import sys

def astar_maze_solver(start_cell:tuple,
                      target_cell:tuple,
                      maze:np.ndarray,
                      get_neighbors: Callable,
                      heuristic:Callable,
                      verbose:bool=False):

    cell_visits=0

    shortest_distance = {}
    shortest_distance[start_cell] = 0

    parent = {}
    parent[start_cell] = start_cell

    expansion_candidates = set([start_cell])

    fully_expanded = set()

    # while there are still cells to be expanded
    while len(expansion_candidates) > 0:

        best_cell = get_best_candidate(expansion_candidates,shortest_distance,
heuristic)

        if best_cell == None:  break

        if verbose: print('\nExpanding cell', best_cell)

        # if the target cell has been reached, reconstruct the shortest path and exit
        if best_cell == target_cell:
```

For Review Purposes Only

```
            shortest_path=reconstruct_shortest_path(parent,start_cell,target_cell)

            return shortest_path, shortest_distance[target_cell],cell_visits

        for neighbor,cost in get_neighbors(maze, best_cell):

            if verbose: print('\nVisiting neighbor cell', neighbor)

            cell_visits+=1

           # first time this neighbor is reached
            if neighbor not in expansion_candidates and neighbor not in fully_
expanded:

                expansion_candidates.add(neighbor)

                parent[neighbor] = best_cell # mark the best_cell as this neighbor's parent

                # update the shortest distance for this neighbor
                shortest_distance[neighbor] = shortest_distance[best_cell] + cost

           # this neighbor has been visited before, but a better (shorter) path to it has just been
found
            elif shortest_distance[neighbor] > shortest_distance[best_cell] + cost:

                shortest_distance[neighbor] = shortest_distance[best_cell] + cost

                parent[neighbor] = best_cell

                if neighbor in fully_expanded:

                    fully_expanded.remove(neighbor)

                    expansion_candidates.add(neighbor)

        # all neighbors of best_cell have been inspected at this point
        expansion_candidates.remove(best_cell)

        fully_expanded.add(best_cell)

    return None, None, None  # no solution was found
```

Similar to **bfs_maze_solver()**, the above function also uses the same two dictionaries **shortest_distance** and **parent** to keep the length of the shortest path from the start cell to each other cell and the parent of the cell on this shortest path.

However, **astar_maze_solver()** follows a different approach to visiting and expanding cells. It uses the expansion_candidates to keep track of all cells that could be expanded next. In every iteration, it uses the **get_best_candidate()** function to select which of these candidates should be expanded next.

After the **best_cell candidate** has been selected, a for loop is used to visit all its neighbors. If a neighbor is visited for the first time, then best_cell becomes the neighbor's parent on the shortest path.

For Review Purposes Only

The same happens if the neighbor has been visited before, but **best_cell** offers a shorter path than the one previously found. If such a better path is indeed found, then the neighbor has to go back to the expansion_candidates set, to reevaluate the shortest path to its own neighbors.

The code below utilizes **astar_maze_solver()** to solve the **unweighted case** of the 3×3 maze puzzle:

```
start_cell=(2,0)
target_cell=(1,2)

solution, distance, cell_visits=astar_maze_solver(start_cell,
                                    target_cell,
                                    small_maze,
                                    get_accessible_neighbors,
                                    constant_heuristic,
                                    verbose=False)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 3
Number of cell visits: 12
```

The A* Search solver finds the best possible shortest path after 12 cell visits. This is slightly higher than the BFS solver, which managed to find the solution in only 10 visits. This is due to the simplicity of the constant heuristic that was used to inform **astar_maze_solver()**. A superior heuristic can be used to help the algorithm find the solution faster.

The next step is to evaluate whether A* Search can indeed solve the weighted maze, which BFS failed to find the shortest path for:

```
start_cell=(2,0)
target_cell=(1,2)

horz_vert_w=1  # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution, distance, cell_visits=astar_maze_solver(start_cell,
                                    target_cell,
                                    small_maze,
                                    partial(get_accessible_neighbors_weighted,
                                         horizontal_vertical_weight=horz_vert_w,
                                         diagonal_weight=diag_w),
                                    constant_heuristic,
                                    verbose=False)

print('\nShortest Path:', solution)
```

For Review Purposes Only

```
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

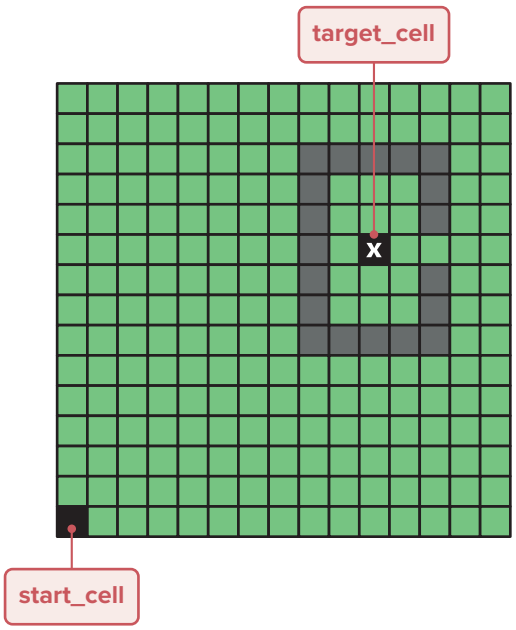
```
Shortest Path: [(2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (1, 2)]
Cells on the Shortest Path: 6
Shortest Path Distance: 5
Number of cell visits: 12
```

The results reveal that **astar_maze_solver()** manages to solve the weighted case by finding the shortest possible path [(2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (1, 2)], with a total cost of 5. This demonstrates the advantage of using an informed search algorithm, which manages to get the optimal solution even when using the simplest possible heuristic.

Algorithm Comparison

The next step is to compare BFS and A* Search on a larger and more complex maze. The following Python code can be used to create a numeric representation of such a maze:

```
big_maze=np.zeros((15,15))

# coordinates of the cells occupied by blocks
blocks=[(2,8), (2,9), (2,10), (2,11), (2,12),
        (8,8), (8,9), (8,10), (8,11), (8,12),
        (3,8), (4,8), (5,8), (6,8), (7,8),
        (3,12), (4,12), (6,12), (7,12)]

for block in blocks:
    # set the value of block-occupied cells to be equal to 1
    big_maze[block]=1
```



This 15×15 maze has a C-shaped section of blocks that the player has to circumnavigate to reach the target marked by the "X". Next, the BFS and A* Search solvers are used to solve both the weighted and unweighted versions of this larger maze:

unweighted version

```
start_cell=(14,0)
target_cell=(5,10)

solution_bfs_unw, distance_bfs_unw, cell_visits_bfs_unw=bfs_maze_solver(start_
cell,
                                        target_cell,
                                        big_maze,
                                        get_accessible_neighbors,
                                        verbose=False)
```

For Review Purposes Only

```
print('\nBFS unweighted.')
print('\nShortest Path:', solution_bfs_unw)
print('Cells on the Shortest Path:', len(solution_bfs_unw))
print('Shortest Path Distance:', distance_bfs_unw)
print('Number of cell visits:', cell_visits_bfs_unw)

solution_astar_unw, distance_astar_unw, cell_visits_astar_unw=astar_maze_solver(
                                        start_cell,
                                        target_cell,
                                        big_maze,
                                        get_accessible_neighbors,
                                        constant_heuristic,
                                        verbose=False)

print('\nA* Search unweighted with a constant heuristic.')
print('\nShortest Path:', solution_astar_unw)
print('Cells on the Shortest Path:', len(solution_astar_unw))
print('Shortest Path Distance:', distance_astar_unw)
print('Number of cell visits:', cell_visits_astar_unw)
```

```
BFS unweighted.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (9, 5), (8, 6),
(8, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13), (6, 13),
(5, 12), (4, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 1237

A* Search unweighted with a constant heuristic.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (10, 5), (10, 6),
(9, 7), (9, 8), (10, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13), (6, 13),
(5, 12), (6, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 1272
```

weighted version

```
start_cell=(14,0)
target_cell=(5,10)

horz_vert_w=1
diag_w=3

solution_bfs_w, distance_bfs_w, cell_visits_bfs_w=bfs_maze_solver(start_cell,
                                        target_cell,
                                        big_maze,
                                        partial(get_accessible_neighbors_weighted,
                                        horizontal_vertical_weight=horz_vert_w,
```

For Review Purposes Only

```
                                        diagonal_weight=diag_w),
                                        verbose=False)

print('\nBFS weighted.')
print('\nShortest Path:', solution_bfs_w)
print('Cells on the Shortest Path:', len(solution_bfs_w))
print('Shortest Path Distance:', distance_bfs_w)
print('Number of cell visits:', cell_visits_bfs_w)

solution_astar_w, distance_astar_w, cell_visits_astar_w=astar_maze_solver(start_
cell,
                                        target_cell,
                                        big_maze,
                                        partial(get_accessible_neighbors_weighted,
                                        horizontal_vertical_weight=horz_vert_w,
                                        diagonal_weight=diag_w),
                                        constant_heuristic,
                                        verbose=False)

print('\nA* Search weighted with constant heuristic.')
print('\nShortest Path:', solution_astar_w)
print('Cells on the Shortest Path:', len(solution_astar_w))
print('Shortest Path Distance:', distance_astar_w)
print('Number of cell visits:', cell_visits_astar_w)
```

```
BFS weighted.

Shortest Path: [(14, 0), (14, 1), (14, 2), (13, 2), (13, 3), (12, 3), (12, 4),
(11, 4), (11, 5), (10, 5), (10, 6), (9, 6), (9, 7), (9, 8), (9, 9), (9, 10),
(9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5, 12), (4,
11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 30
Number of cell visits: 1235

A* Search weighted with constant heuristic.

Shortest Path: [(14, 0), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (9, 1),
(9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9), (9, 10), (9,
11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5, 12), (5, 11),
(5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 25
Number of cell visits: 1245
```

For Review Purposes Only

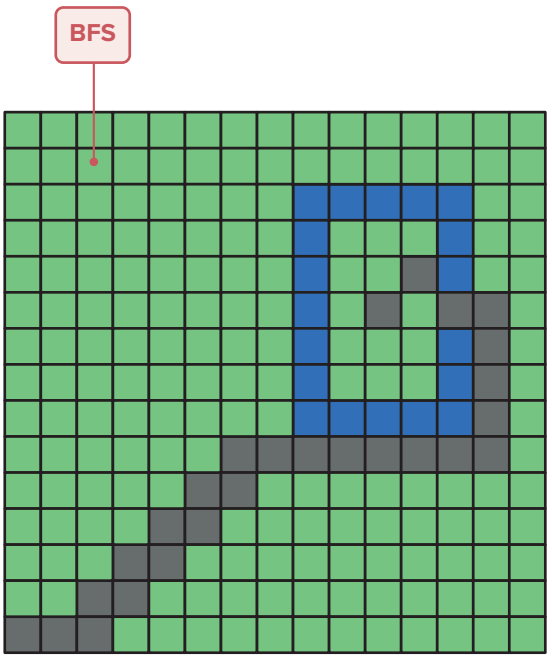
The results are consistent with the ones reported for the small maze:

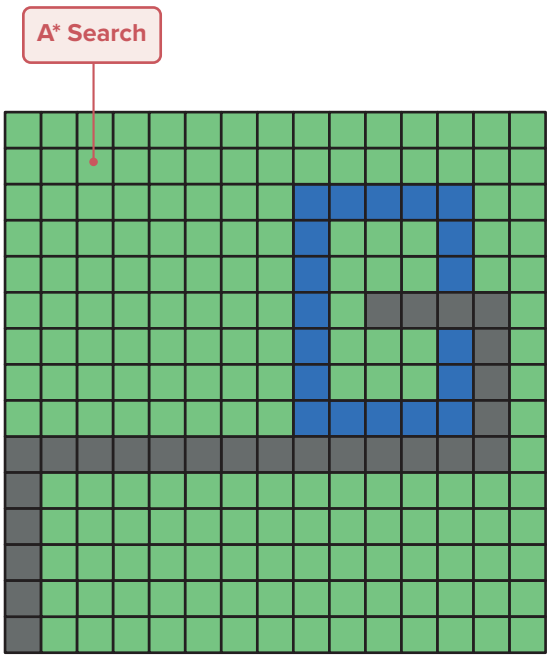
- Both BFS and A* Search find the shortest path for the unweighted version.
- BFS finds the solution in fewer visits (1,237 vs. 1,272 for A* Search).
- BFS fails to find the shortest path for the weighted version, as it reports a path with a distance of 30.
- A* Search finds the shortest path for the weighted version, reporting a path with a distance of 25.

The following code can be used to visualize the shortest path found by the BFS and A* Search algorithms on the weighted version:

```
maze_bfs_w=big_maze.copy()

for cell in solution_bfs_w:
    maze_bfs_w[cell]=2

plot_maze(maze_bfs_w)
```

```
maze_astar_w=big_maze.copy()

for cell in solution_astar_w:
    maze_astar_w[cell]=2

plot_maze(maze_astar_w)
```





The visualizations verify that the informed nature of A* Search allows it to avoid diagonal moves, as they have a higher cost than horizontal and vertical ones. On the other hand, the uninformed BFS ignores the cost of each move and reports a much more expensive solution. A general comparison of uninformed and informed algorithms is presented in the table.

For Review Purposes Only

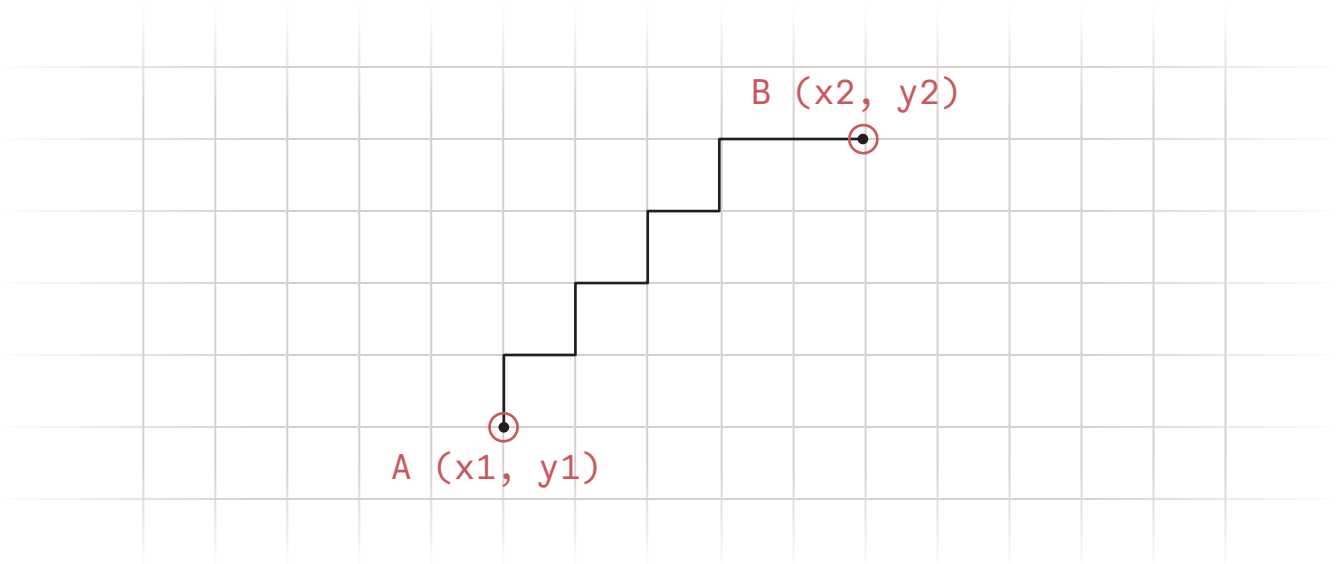
Comparison of uninformed and informed algorithms

Comparison criteria	Uninformed	Informed
Computational complexity	They are more computationally complex.	Their computational cost is lower.
Efficiency	They are slower than informed algorithms.	They perform searches quicker.
Performance	Impractical for solving large-scale search problems.	Better at handling large-scale search problems.
Effectiveness	The optimal solution is achieved.	Generally, adequate solutions are accepted.

Still, the results indicated that BFS could find the optimal solution faster (with fewer cell visits) in the unweighted case. This can be addressed by providing A* Search with a smarter heuristic. A popular heuristic in distance-based applications is the **Manhattan distance**, defined as the sum of the absolute differences between the coordinates of the two given points. An example is presented in the image below:

Manhattan distance

```
Manhattan (A, B) = |x1-x2| + |y1-y2|
```



For Review Purposes Only

This can be easily implemented as a Python function as follows:

```
def manhattan_heuristic(candidate_cell:tuple,target_cell:tuple):

    x1,y1=candidate_cell
    x2,y2=target_cell
    return  abs(x1 - x2) + abs(y1 - y2)
```

The following code can be used to test if this smarter heuristic can be used to help **astar_maze_solver()** search the space much faster for both weighted and unweighted scenarios:

```
start_cell=(14,0)
target_cell=(5,10)

solution_astar_unw_mn, distance_astar_unw_mn, cell_visits_astar_unw_mn=astar_
maze_solver(start_cell,
            target_cell,
            big_maze,
            get_accessible_neighbors,
            manhattan_heuristic,
            verbose=False)

print('\nA* Search unweighted with the Manhattan heuristic.')
print('\nShortest Path:', solution_astar_unw_mn)
print('Cells on the Shortest Path:', len(solution_astar_unw_mn))
print('Shortest Path Distance:', distance_astar_unw_mn)
print('Number of cell visits:', cell_visits_astar_unw_mn)

horz_vert_w=1  # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution_astar_w_mn, distance_astar_w_mn, cell_visits_astar_w_mn=astar_maze_
solver(start_cell,
       target_cell,
       big_maze,
       partial(get_accessible_neighbors_weighted,
               horizontal_vertical_weight=horz_vert_w,
               diagonal_weight=diag_w),
       manhattan_heuristic,
       verbose=False)

print('\nA* Search weighted with the Manhattan heuristic.')
print('\nShortest Path:', solution_astar_w_mn)
print('Cells on the Shortest Path:', len(solution_astar_w_mn))
print('Shortest Path Distance:', distance_astar_w_mn)
print('Number of cell visits:', cell_visits_astar_w_mn)
```

For Review Purposes Only

```
A* Search unweighted with the Manhattan heuristic.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (9, 5), (8, 6),
(8, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13), (6, 13),
(5, 12), (5, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 865

A* Search weighted with the Manhattan heuristic.

Shortest Path: [(14, 0), (14, 1), (13, 1), (12, 1), (12, 2), (12, 3), (12, 4),
(12, 5), (12, 6), (12, 7), (11, 7), (11, 8), (10, 8), (9, 8), (9, 9), (9, 10),
(9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5, 12), (5,
11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 25
Number of cell visits: 1033
```

The results verify that the Manhattan distance heuristic can indeed help A* Search find the shortest possible paths with a significantly lower number of cell visits for both weighted and unweighted scenarios. In fact, the use of this more intelligent heuristic led to a significantly lower visit number than the one required for the BFS algorithm.

The table below summarizes the results for the different algorithm variants on the big maze:

Comparison of algorithms performance

Graph type	BFS	A* Search with constant heuristic	A* Search with Manhattan heuristic
Weighted	dist=30, 1235 visits	dist=25, 1245 visits	dist=25, 1033 visits
Unweighted	dist=18, 1237 visits	dist=18, 1272 visits	dist=18, 865 visits

The table demonstrates the advantages of using increasingly more intelligent algorithms to solve search-based problems like the one presented in this lesson:

- Switching from an uninformed search algorithm, such as BFS, to an informed one, like A* Search, delivered better results and enabled the solution of more complex problems.
- The intelligence of informed search algorithms can be further increased by using better heuristics that allow them to find the optimal solution significantly faster.

For Review Purposes Only

EXERCISES

1. Identify two applications of search algorithms.

2. Identify a difference between uninformed and informed search algorithms and mention an example of each algorithm.

3. Explain briefly how the A* algorithm works.

4. Modify your code by changing the diagonal weight from 3 to 1.5. What do you observe? Does the shortest path change for the cases of BFS and A* Search?

5. Modify your code by swapping the starting cell with the target cell coordinates. What do you observe? Is the path the same as before for the weighted cases of BFS and A* Search?

For Review Purposes Only

PROJECT

Comparing BFS and A* Search Algorithms

Smart healthcare is one of the most important sectors that IoT technologies improve. A variety of devices and systems are interconnected and exchange large amounts of data. Medical and biological data are considered the most private personal data and must be protected by companies and governments.

1. Modify the code of the weighted BFS and A* Search algorithms by changing the horizontal and vertical weights to 3 and the diagonal weights to 5. Also change the starting point to (7, 2).

2. What is the new shortest distance path and the number of cell visits of the unweighted versions of the BFS and A* Search algorithms with the constant heuristic function? Find these values and present your observations.

3. Follow the same steps for the weighted versions of the BFS and A* Search algorithms with the constant heuristic function.

4. Repeat the process for the unweighted and weighted versions of the A* Search algorithms with the Manhattan heuristic function.

Copyright © Binary Logic SA sherryyates/123rf

For Review Purposes Only

1 WRAP UP

THIS UNIT COVERED HOW TO:

> apply advanced graph traversing algorithms.

> implement both simple and advanced rule-based systems.

> design an AI model.

> measure the effectiveness of your AI model.

> use search algorithms to solve simulations of real-life problems.

KEY TERMS

- A* Search
- Breadth-First Search (BFS)
- Confusion Matrix
- Depth-First Search (DFS)
- Gini Index
- Heuristic Function
- Informed Search
- Knowledge Base
- Maze Solving
- Model Training
- Path Finding
- Queue
- Rule-Based Systems
- Scoring Function
- Search Algorithms
- Stack
- Uninformed Search
- Unweighted Graph
- Weighted Graph

For Review Purposes Only

Foundations of AI

Artificial Intelligence 2

Build systems that think for you

Picture a world where machines can think, learn, and solve problems on their own. What if you could design algorithms that enable intelligent systems to make decisions, tackle challenges, and drive innovation—whether it's automating everyday tasks or advancing the field of robotics?

Foundations of AI: This course teaches you the basics of artificial intelligence and optimization. You'll learn about search algorithms like DFS/BFS and how to make decisions using rule-based systems. Explore optimization problems, resource management, and scheduling, while learning techniques to improve decision-making in real-world situations.

By the end of this course, you'll have the skills to design and implement powerful AI algorithms, optimize systems for efficiency, and apply your knowledge to robotics and automation. You'll be empowered to build intelligent systems that can reshape industries and solve problems across the globe.

Mc Graw Hill
binarylogic

For Review Purposes Only

ISBN-13: 978-1-2658-7178-9
ISBN-10: 1-2658-7178-7
9 781265 871789