INTRODUCTION TO ARTIFICIAL INTELLIGENCE PROJECT

Analysis and Visualization of Pathfinding Algorithms

University of Petroleum and Energy Studies, Dehradun
B.Tech (CSE) Spl. Artificial Intell & ML, B3

INTRODUCTION TO ARTIFICIAL INTELLIGENCE PROJECT

Analysis and Visualization of Pathfinding Algorithms

STUDENT DETAILS:

Name	SAP ID
Desh Iyer	500081889
Bhavy Kharbanda	500082531
Devashish Agarwal	500082411

Faculty: Mr. Virender Kadyan

Batch: B.Tech (CSE) Spl. Artificial Intell & ML, Semester 3, B3

PATHFINDING

Pathfinding is closely related to the shortest path problem - the path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized - within graph theory, which researches how to identify the path that best meets some criteria (shortest, cheapest, fastest, and so on) between two points in a large network.

Pathing is a more practical variation of maze solving, and it is mainly based on Dijkstra's algorithm for finding the shortest path on a weighted graph.

PATHFINDING ALGORITHMS

Schools of thought with Pathfinding Algorithms

At its core, a pathfinding method searches a graph by starting at one vertex and exploring adjacent nodes until the destination node is reached, generally with the intent of finding the cheapest route. Although graph searching methods such as a breadth-first search would find a route if given enough time, other methods, which "explore" the graph, would tend to reach the destination sooner. An analogy would be a person walking across a room; rather than examining every possible route in advance, the person would generally walk in the direction of the destination and only deviate from the path to avoid an obstruction, and make deviations as minor as possible.

What Pathfinding Algorithms aim to solve

Two primary problems of pathfinding are

1. To find a path between two nodes in a graph; and
2. The shortest path problem—to find the optimal shortest path.

Types of Pathfinding Algorithms

BASIC ALGORITHMS

Basic algorithms such as breadth-first and depth-first search address the first problem by exhausting all possibilities; starting from the given node, they iterate over all potential paths until they reach the destination node. These algorithms run in

$$O(|V| + |E|)$$

or linear time, where V is the number of vertices, and E is the number of edges between vertices.

OPTIMAL PATH ALGORITHMS

The more complicated problem is finding the optimal path. The exhaustive approach in this case is known as the Bellman–Ford algorithm, which yields a time complexity of

$$O(|V||E|)$$

or quadratic time.

HEURISTIC OPTIMAL PATH ALGORITHMS

However, it is not necessary to examine all possible paths to find the optimal one. Algorithms such as A* and Dijkstra's algorithm strategically eliminate paths, either through heuristics or through dynamic programming. By eliminating impossible paths, these algorithms can achieve time complexities as low as

$$O(|E|\,log(|V|))$$

CONCLUSION

The third type of pathfinding algorithms which use heuristics or dynamic programming, are among the best general algorithms which operate on a graph without pre-processing. However, in practical travel-routing systems, even better time complexities can be attained by algorithms which can pre-process the graph to attain better performance. One such algorithm is contraction hierarchies but we won’t be touching on those algorithms in this project.

Thus, we can observe that pathfinding algorithms which do not using pre-processing are classified on their time complexities as taking linear time, quadratic time, or better using heuristics or dynamic programming. There are categories higher which offer better runtimes but those require the graph to be pre-processed.

A* PATHFINDING ALGORITHM

Aforementioned are the different types of pathfinding algorithms classified on the basis of their runtimes. Among those algorithms that run without the use of pre-processing of the graph, A* falls into the category that can achieve better runtime because it makes use of heuristics.

Heuristics

In mathematical optimization and computer science, heuristic ("I find, discover" in Greek) is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.

A heuristic function, also simply called a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow. For example, it may approximate the exact solution.

The objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. This solution may not be the best of all the solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time.

EXAMPLE OF USING HEURISTICS TO SOLVE PROBLEMS QUICKER

Travelling salesman problem

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

An optimal solution for TSP for even a moderate size problem is difficult to solve. The mainstream method of solving it is using the greedy algorithm to give a good but not optimal solution (it is an approximation to the optimal answer) in a reasonably short amount of time.

However, the greedy algorithm heuristic says to pick whatever is currently the best next step regardless of whether that prevents good steps later. It is a heuristic in that practice says it is a good enough solution, theory says there are better solutions.

Working of A* Pathfinding Algorithm

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

WHAT HAPPENS EACH ITERATION

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal using heuristics. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

USE OF PRIOIRTY QUEUE

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbours are updated accordingly, and these neighbours are added to the queue. The algorithm continues until a removed node (thus the node with the lowest f value out of all fringe nodes) is a goal node. The f value of that goal is then also the cost of the shortest path, since h at the goal is zero in an admissible heuristic.

HOW THE ALGORITHM KEEPS TRACK OF SHORTEST PATH

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

Pseudocode

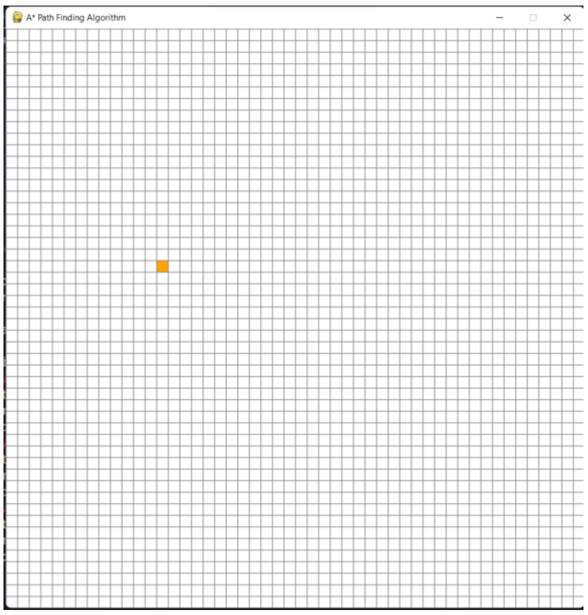
```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

# A* finds a path from start to goal.
# h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    # The set of discovered nodes that may need to be (re-)expanded.
    # Initially, only the start node is known.
    # This is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    # For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
    # to n currently known.
    cameFrom := an empty map

    # For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    # For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
    # how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        # This operation can occur in O(1) 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
```

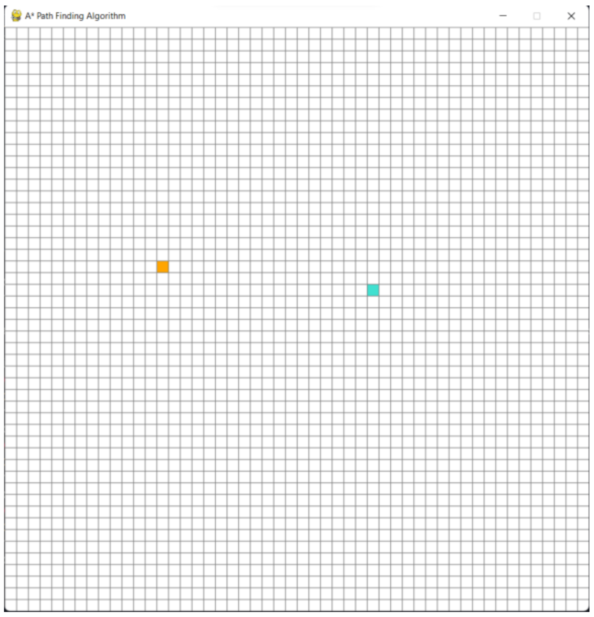
VISUALIZATION OF A* PATHFINDING ALGORITHM IN PYTHON

Program Flow

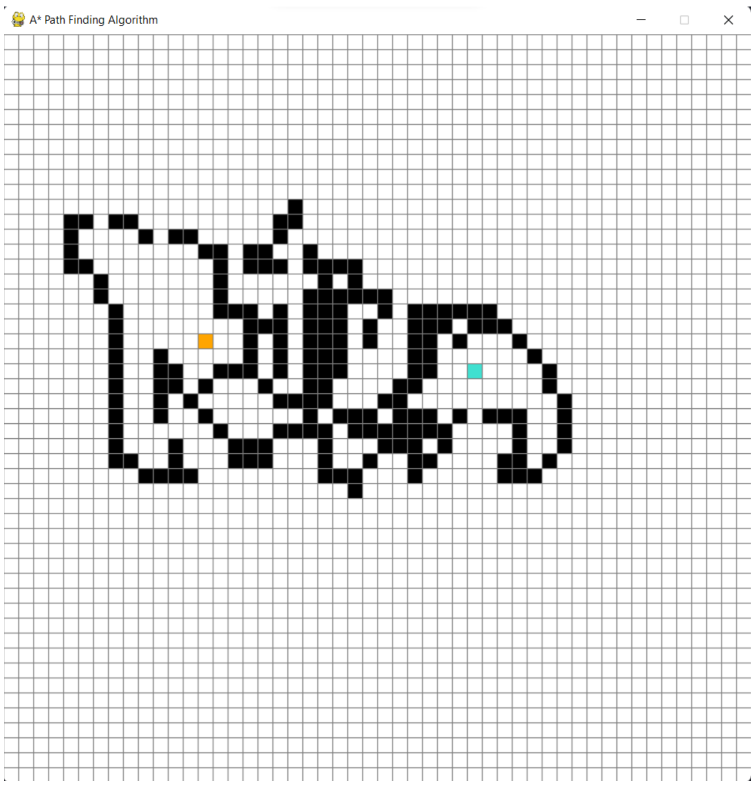
The program initially displays a grid of set size. Then prompts the user to use the mouse to click any of the squares in the grid to select the starting node then the ending node. The starting node is displayed in orange and the final node is displayed in cyan. After selecting the starting and the final nodes, the user is then prompted to draw a maze in between and around the two nodes. Upon finishing the maze, the user then is prompted to press spacebar at which time the program starts the algorithm visualizing the algorithm. When the algorithm runs, at any given point, the nodes in red are those nodes which are considered and the ones in green are the nodes to be considered next. When the final node is reached, the program then backtracks and reveals the shortest path in purple.

Screenshots

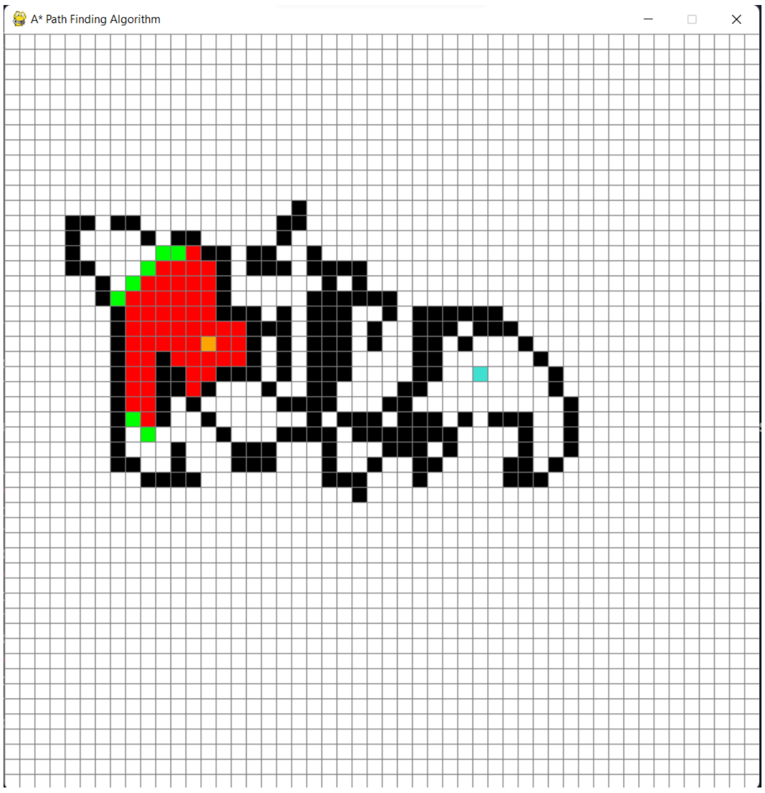




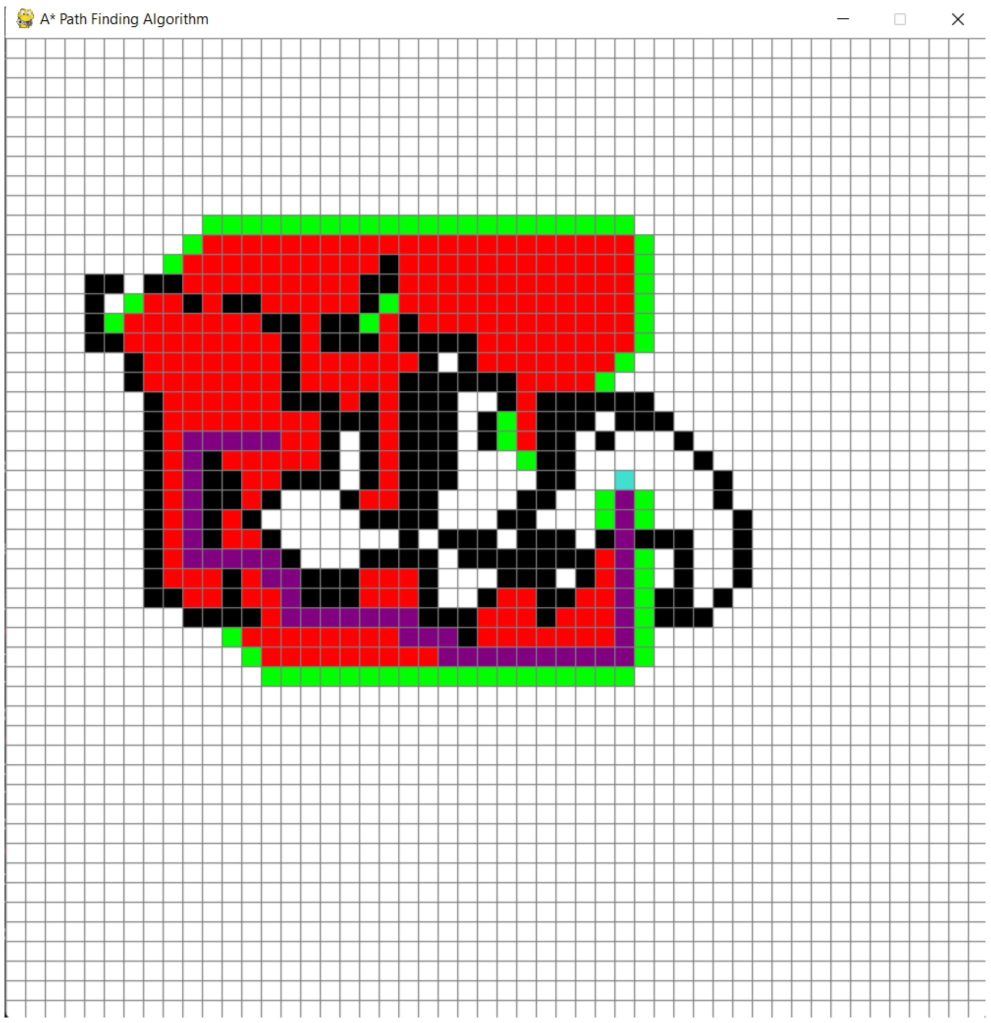
Steps 1 & 2 – Choosing of Starting and Ending Nodes



Step 3 – Drawing Maze (Obstacle Nodes)



Step 4 – Intermediate Step showing the algorithm at work



Step 5 – Final Step when the algorithm is complete and the shortest path between two nodes is displayed in purple

Source code

```python
import pygame
import math
from queue import PriorityQueue

WIDTH = 800
WIN = pygame.display.set_mode((WIDTH, WIDTH))
pygame.display.set_caption("A* Path Finding Algorithm")

RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 255, 0)
YELLOW = (255, 255, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
PURPLE = (128, 0, 128)
ORANGE = (255, 165 ,0)
GREY = (128, 128, 128)
TURQUOISE = (64, 224, 208)

class Spot:
    def __init__(self, row, col, width, total_rows):
        self.row = row
        self.col = col
        self.x = row * width
        self.y = col * width
        self.color = WHITE
        self.neighbors = []
        self.width = width
        self.total_rows = total_rows

    def get_pos(self):
        return self.row, self.col

    def is_closed(self):
        return self.color == RED

    def is_open(self):
        return self.color == GREEN

    def is_barrier(self):
        return self.color == BLACK

    def is_start(self):
        return self.color == ORANGE

    def is_end(self):
        return self.color == TURQUOISE

    def reset(self):
        self.color = WHITE

    def make_start(self):
        self.color = ORANGE

    def make_closed(self):
        self.color = RED

    def make_open(self):
        self.color = GREEN

    def make_barrier(self):
        self.color = BLACK

    def make_end(self):
        self.color = TURQUOISE

    def make_path(self):
        self.color = PURPLE

    def draw(self, win):
        pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.width))

    def update_neighbors(self, grid):
        self.neighbors = []
        if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier(): # DOWN
            self.neighbors.append(grid[self.row + 1][self.col])

        if self.row > 0 and not grid[self.row - 1][self.col].is_barrier(): # UP
            self.neighbors.append(grid[self.row - 1][self.col])

        if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier(): # RIGHT
            self.neighbors.append(grid[self.row][self.col + 1])

        if self.col > 0 and not grid[self.row][self.col - 1].is_barrier(): # LEFT
            self.neighbors.append(grid[self.row][self.col - 1])

    def __lt__(self, other):
        return False


def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)


def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        current.make_path()
        draw()


def algorithm(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(), end.get_pos())

    open_set_hash = {start}

    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from, end, draw)
            end.make_end()
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1

            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()

        draw()

        if current != start:
            current.make_closed()

    return False


def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot = Spot(i, j, gap, rows)
            grid[i].append(spot)

    return grid


def draw_grid(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, GREY, (0, i * gap), (width, i * gap))
        for j in range(rows):
            pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, width))


def draw(win, grid, rows, width):
    win.fill(WHITE)

    for row in grid:
        for spot in row:
            spot.draw(win)

    draw_grid(win, rows, width)
    pygame.display.update()


def get_clicked_pos(pos, rows, width):
    gap = width // rows
    y, x = pos

    row = y // gap
    col = x // gap

    return row, col


def main(win, width):
    ROWS = 50
    grid = make_grid(ROWS, width)

    start = None
    end = None

    run = True
    while run:
        draw(win, grid, ROWS, width)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            if pygame.mouse.get_pressed()[0]: # LEFT
                pos = pygame.mouse.get_pos()
                row, col = get_clicked_pos(pos, ROWS, width)
                spot = grid[row][col]
                if not start and spot != end:
                    start = spot
                    start.make_start()

                elif not end and spot != start:
                    end = spot
                    end.make_end()

                elif spot != end and spot != start:
                    spot.make_barrier()

            elif pygame.mouse.get_pressed()[2]: # RIGHT
                pos = pygame.mouse.get_pos()
                row, col = get_clicked_pos(pos, ROWS, width)
                spot = grid[row][col]
                spot.reset()
                if spot == start:
                    start = None
                elif spot == end:
                    end = None

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE and start and end:
                    for row in grid:
                        for spot in row:
                            spot.update_neighbors(grid)

                    algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)

                if event.key == pygame.K_c:
                    start = None
                    end = None
                    grid = make_grid(ROWS, width)

    pygame.quit()

main(WIN, WIDTH)
```

APPLICATIONS AND DRAWBACK OF A*

This project wouldn’t be complete if we didn’t talk about the applications the A* pathfinding algorithm and its drawbacks.

Some common variants of Dijkstra's algorithm can be viewed as a special case of A* where the heuristic $h(n) = 0$ for all nodes; in turn, both Dijkstra and A* are special cases of dynamic programming. A* itself is a special case of a generalization of branch and bound.

As a result, A* is often used for the common pathfinding problem in applications such as video games, but was originally designed as a general graph traversal algorithm. It finds applications in diverse problems, including the problem of parsing using stochastic grammars in NLP. Other cases include an Informational search with online learning.

One major practical drawback is its $O(b\text{^}d)$ space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, as well as memory-bounded approaches; however, A* is still the best solution in many cases.