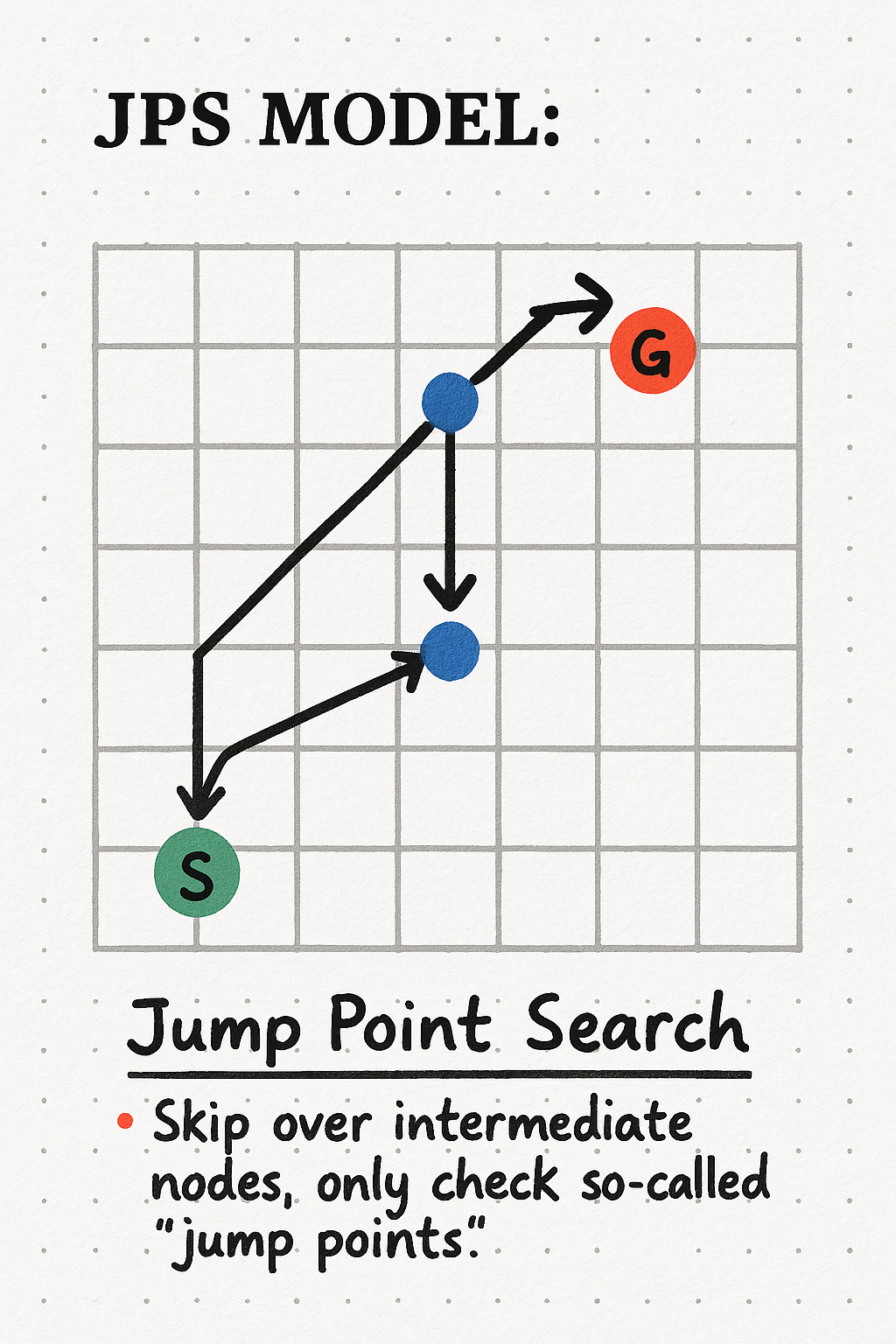
ASSIGNMENT 1:

**JUMP – POINT SEARCH (JPS):**

The **Jump Point Search (JPS)** algorithm is an optimization of the **A**\* search algorithm, designed for pathfinding on uniform-cost grids. It works by reducing the number of nodes that need to be explored during the search process. Instead of examining every single node, JPS identifies "jump points"—key nodes that are necessary for finding the shortest path—and skips over unnecessary nodes.

This approach significantly speeds up the pathfinding process while maintaining the optimality of the solution. It's commonly used in applications like robotics and video games where efficient pathfinding is crucial.

**JPS MODEL:**



**GAME CODE LOGIC:**

START

1. Initialize constants (screen size, grid size, colors, etc.) and set up the game window.

2. Create an empty grid where each cell can represent:

- Empty space

- Walls

- Path

- Start point

- End point

3. Define functions:

a. draw\_grid:

- Draw each cell based on its type (empty, wall, path, etc.).

- Highlight start and end points with specific colors.

b. clear\_path:

- Reset all path cells (used to update the grid dynamically).

c. jump\_point\_search:

- Start from the initial node (start point).

- Explore neighbors in all directions (horizontal, vertical, diagonal).

- Skip unnecessary nodes and only move to valid cells.

- Stop if the end point is found and reconstruct the shortest path.

d. reconstruct\_path:

- Trace back from the end node to the start using the path information.

- Mark the cells in the grid to display the resulting path.

4. Handle user input:

- Left-click to place walls.

- Right-click to remove walls.

- Use specific keys to set the start point, end point, or trigger pathfinding (e.g., the SPACE key).

5. Main game loop:

a. Clear the screen.

b. Handle user input and updates (start point, end point, walls, etc.).

c. Call the pathfinding function when triggered.

d. Draw the grid with updated states (walls, path, etc.).

e. Refresh the display at a consistent frame rate.

6. Exit the game if the user closes the window.

END

**SOURCE-CODE:**

import pygame

import sys

*# Constants*

WIDTH, HEIGHT = 600, 600

GRID\_SIZE = 30

CELL\_SIZE = WIDTH // GRID\_SIZE

WHITE, BLACK, RED, BLUE, GREEN, YELLOW = (255, 255, 255), (0, 0, 0), (255, 0, 0), (0, 0, 255), (0, 255, 0), (255, 255, 0)

*# Pygame Initialization*

pygame.init()

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption("JPS Visual Implementation")

clock = pygame.time.Clock()

*# Grid Initialization*

grid = [[0 for \_ in range(GRID\_SIZE)] for \_ in range(GRID\_SIZE)]

*# Start/Goal Points*

start\_point = None

end\_point = None

path = []

*# Functions to Draw the Grid*

def draw\_grid():

    for x in range(GRID\_SIZE):

        for y in range(GRID\_SIZE):

            rect = pygame.Rect(x \* CELL\_SIZE, y \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

            if grid[y][x] == 1:  *# Wall*

                pygame.draw.rect(screen, BLACK, rect)

            elif grid[y][x] == 2:  *# Path*

                pygame.draw.rect(screen, GREEN, rect)

            else:

                pygame.draw.rect(screen, WHITE, rect)

            pygame.draw.rect(screen, BLACK, rect, 1)  *# Draw cell borders*

    if start\_point:

        pygame.draw.rect(screen, BLUE, (start\_point[0] \* CELL\_SIZE, start\_point[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

    if end\_point:

        pygame.draw.rect(screen, RED, (end\_point[0] \* CELL\_SIZE, end\_point[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

*# Reset the Path*

def clear\_path():

    for y in range(GRID\_SIZE):

        for x in range(GRID\_SIZE):

            if grid[y][x] == 2:

                grid[y][x] = 0

*# Improved Jump Point Search with Diagonal Movement*

def jump\_point\_search(start, end):

    clear\_path()

    queue = [start]

    visited = set()

    came\_from = {}  *# To track the path*

    while queue:

        x, y = queue.pop(0)

        visited.add((x, y))

        if (x, y) == end:

*# Goal is reached*

            reconstruct\_path(came\_from, end)

            return True

*# Explore neighbors (8 directions including diagonals)*

        for dx, dy in [

            (0, 1), (1, 0), (0, -1), (-1, 0),  *# Cardinal directions*

            (1, 1), (-1, -1), (-1, 1), (1, -1)  *# Diagonal directions*

        ]:

            nx, ny = x + dx, y + dy

*# Check bounds and obstacles*

            if 0 <= nx < GRID\_SIZE and 0 <= ny < GRID\_SIZE and (nx, ny) not in visited and grid[ny][nx] != 1:

*# For diagonal moves, ensure no blocking obstacles*

                if dx != 0 and dy != 0:  *# Diagonal move*

                    if grid[y][nx] == 1 or grid[ny][x] == 1:

                        continue  *# Skip diagonal if blocked*

                queue.append((nx, ny))

                visited.add((nx, ny))

                came\_from[(nx, ny)] = (x, y)

*# If we exit the loop, no path was found*

    print("No path found!")  *# Console notification*

    return False

*# Reconstruct the Path*

def reconstruct\_path(came\_from, current):

    while current in came\_from:

        x, y = current

        grid[y][x] = 2

        path.append(current)

        current = came\_from[current]

*# Event Handling*

def handle\_events():

    global start\_point, end\_point

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()

            sys.exit()

        elif event.type == pygame.MOUSEBUTTONDOWN:

            x, y = pygame.mouse.get\_pos()

            grid\_x, grid\_y = x // CELL\_SIZE, y // CELL\_SIZE

            if pygame.key.get\_pressed()[pygame.K\_s]:  *# Place Start Point*

                start\_point = (grid\_x, grid\_y)

            elif pygame.key.get\_pressed()[pygame.K\_g]:  *# Place End Point*

                end\_point = (grid\_x, grid\_y)

            elif event.button == 1:  *# Left Click to Add Wall*

                grid[grid\_y][grid\_x] = 1

            elif event.button == 3:  *# Right Click to Remove Wall*

                grid[grid\_y][grid\_x] = 0

        elif event.type == pygame.KEYDOWN:

            if event.key == pygame.K\_SPACE and start\_point and end\_point:

                clear\_path()

                path\_found = jump\_point\_search(start\_point, end\_point)

                if not path\_found:

                    print("Unable to find a valid path to the goal!")

*# Main Loop*

while True:

    screen.fill(WHITE)

    handle\_events()

    draw\_grid()

    pygame.display.flip()

    clock.tick(30)

**CODE OUTPUT:**

