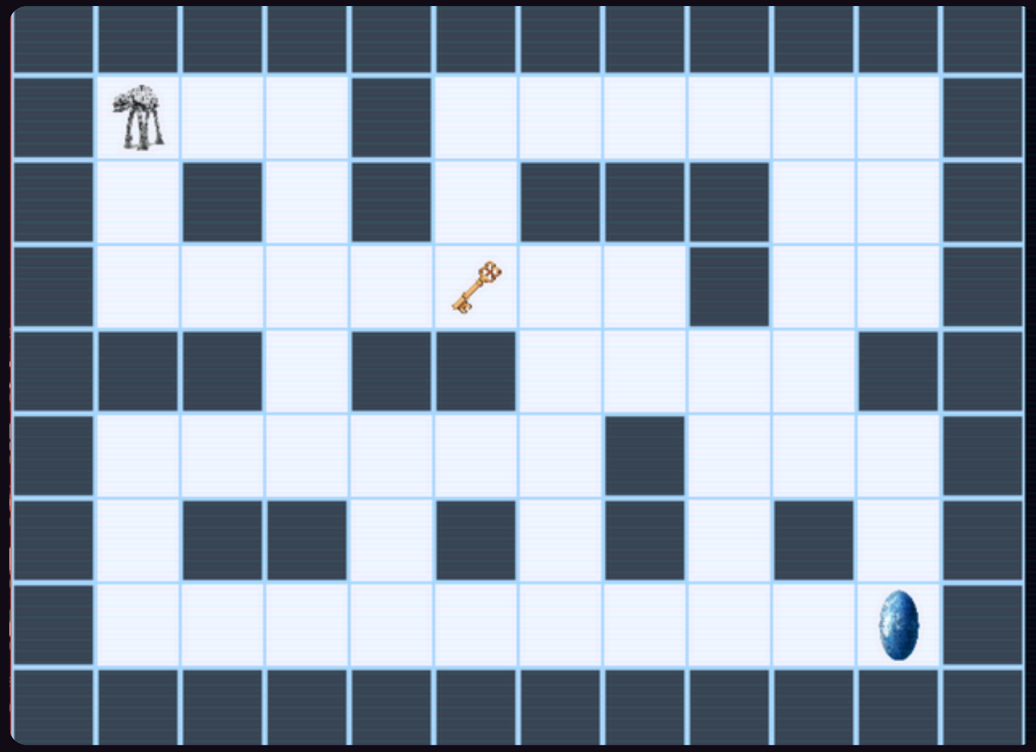

ICE MAZE
GAME

WHAT IS “ICE MAZE GAME?”

Unlike traditional maze games, Once you move in Ice maze, you are required to keep going in that direction until you hit something, as if the floor is slippery like ice.

Our game is star wars themed, since the series film features an ice planet called hoth.



CHARACTERS

AT-AT WALKER

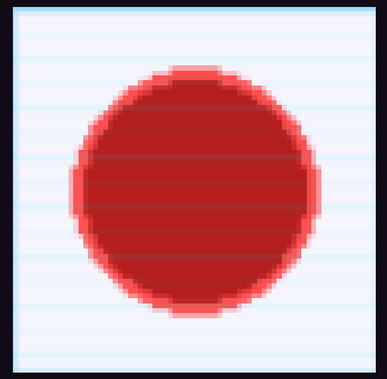
This is the “Player” that user(or AI) can control. Their goal is to reach to Echo Base.

BASE KEY

The requirement that walker must collect before entering the base, else the door won’t open.

ECHO BASE

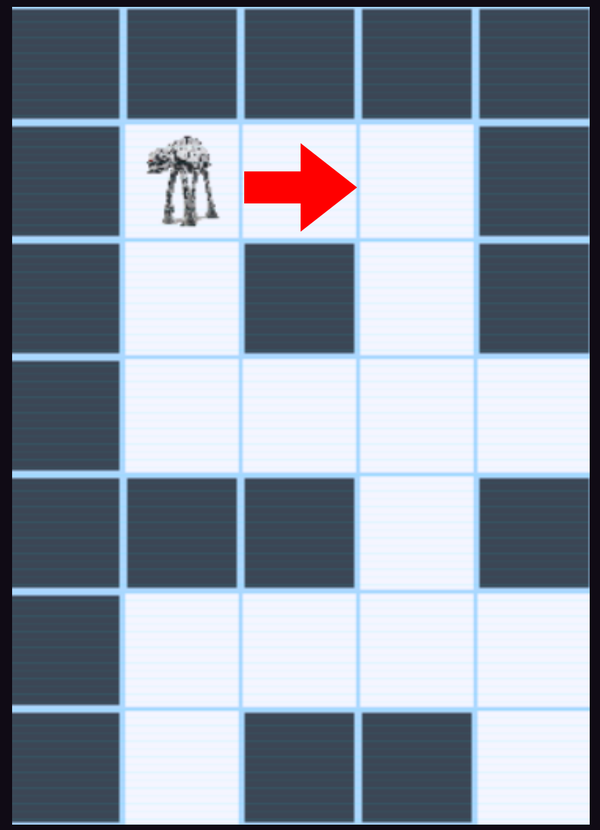
The goal of the game, the walker are to reach the echo base with the key.

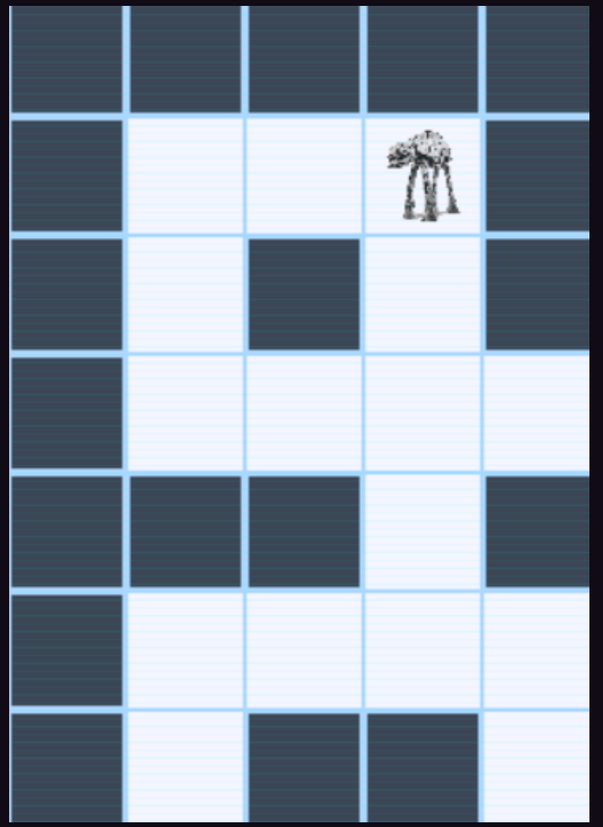


PATROLS

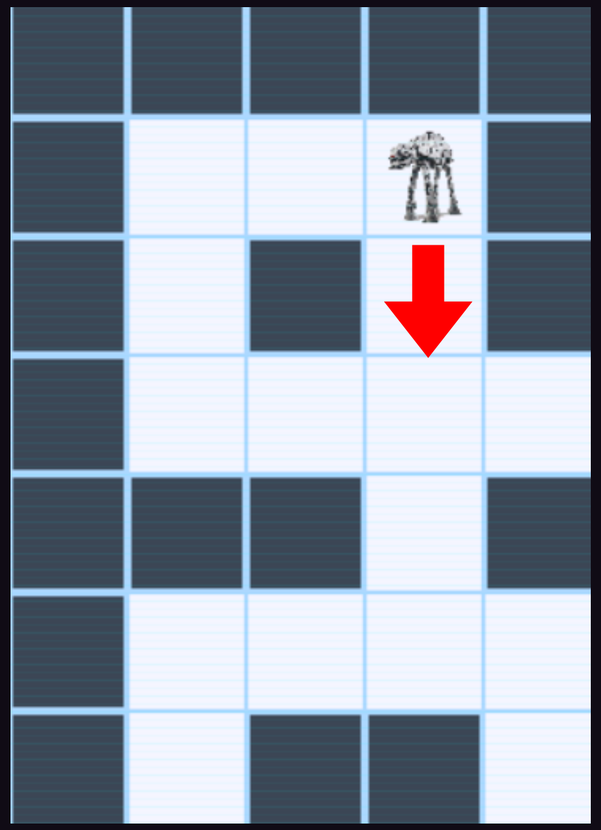
Patrols are the “enemy”, if the walker is caught in the same tile as patrol, the game is over.

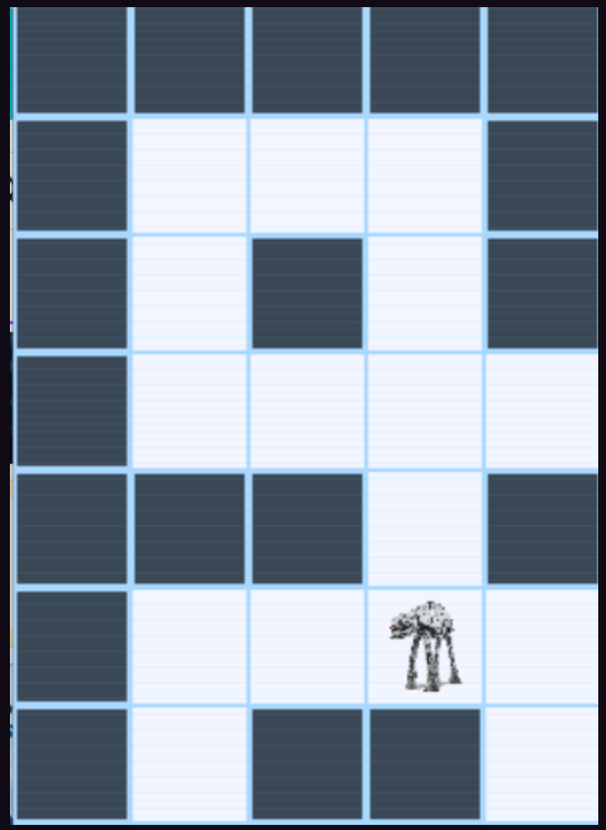
ICE MAZE TUTORIAL



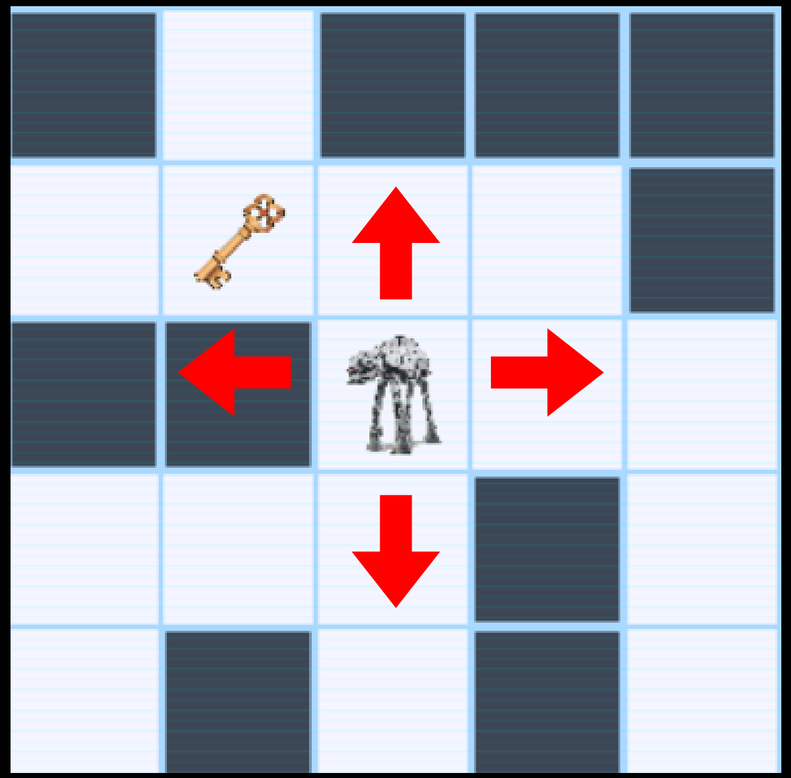


ICE MAZE TUTORIAL



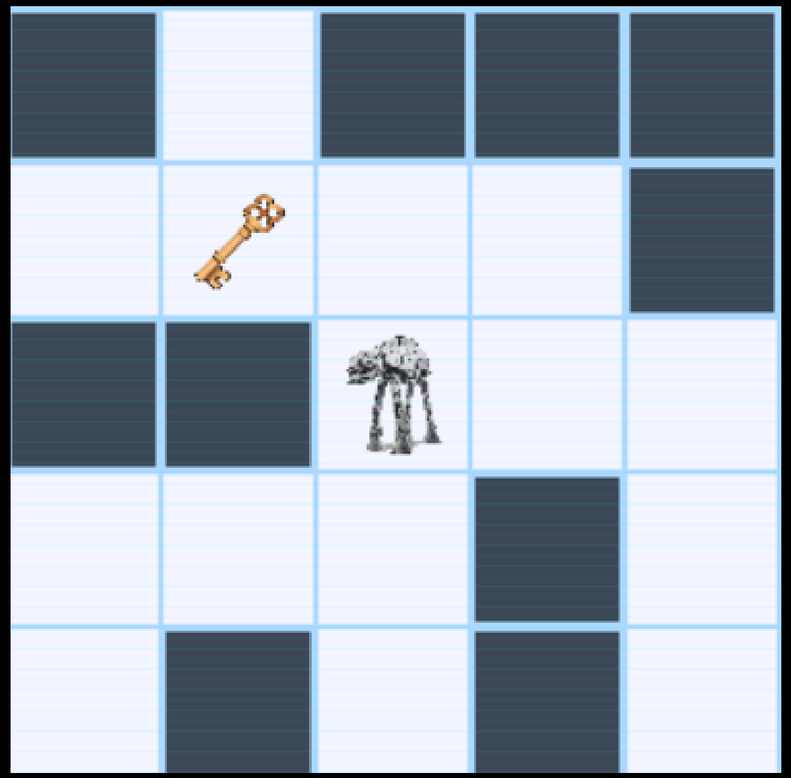


ACTION SPACE



1. Up
2. Down
3. Left
4. Right
5. Wait

Actions = {Up, down, left, right, wait}

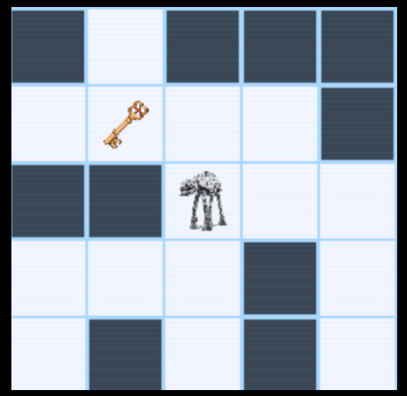


```
self.grid = [
    [1,1,1,1,1,1,1,1,1,1,1,1],
    [1,0,0,0,1,0,0,0,0,0,0,1],
    [1,0,1,0,1,0,1,1,1,0,0,1],
    [1,0,0,0,0,0,0,0,1,0,0,1],
    [1,1,1,0,1,1,0,0,0,0,1,1],
    [1,0,0,0,0,0,0,1,0,0,0,1],
    [1,0,1,1,0,1,0,1,0,0,0,1],
    [1,0,0,0,0,0,0,0,0,0,0,1],
    [1,1,1,1,1,1,1,1,1,1,1,1]
]
```

STATE SPACE

<x,y,key_collected,goal>

<2,2,0,0>

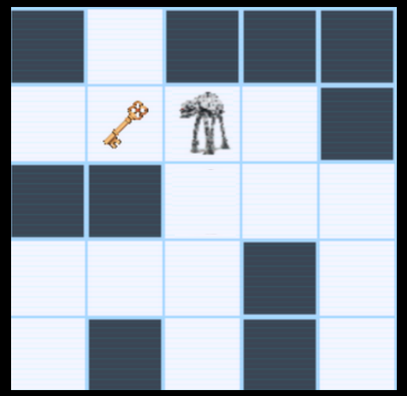


self.player_pos(2,2)

self.key_pos(1,1)

Actions = {}

<2,1,0,0>

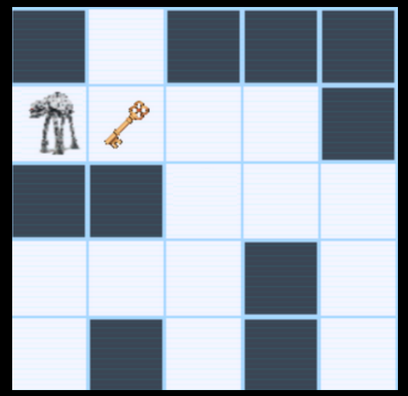


self.player_pos(2,1)

self.key_pos(1,1)

Actions = {Up}

<0,1,0,0>

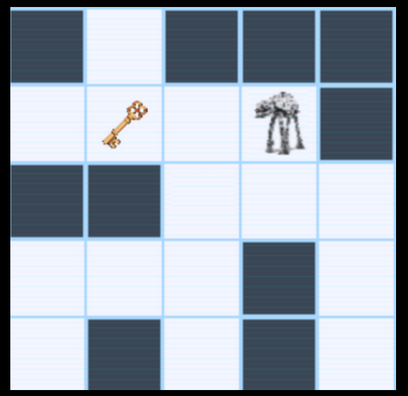


self.player_pos(0,1)

self.key_pos(1,1)

Actions = {Up, left}

<3,1,0,0>



self.player_pos(3,1)

self.key_pos(1,1)

Actions = {Up, left, right}

...

OUR GITHUB

SCAN HERE !

https://github.com/pimme05/ice-maze-blind-search

README | MIT license

Star Wars: Hoth Ice Maze

A Star Wars-themed puzzle game featuring AI pathfinding algorithms and sliding ice mechanics. Navigate through 5 challenging levels as a Rebel pilot trying to reach Echo Base while avoiding Imperial patrols.

🎮 Game Overview

Unlike traditional maze games, this ice maze requires you to keep sliding in your chosen direction until you hit an obstacle - just like moving on slippery ice! Set on the ice planet Hoth from the Star Wars universe, you must collect a golden key before reaching Echo Base to complete each level.

Key Features

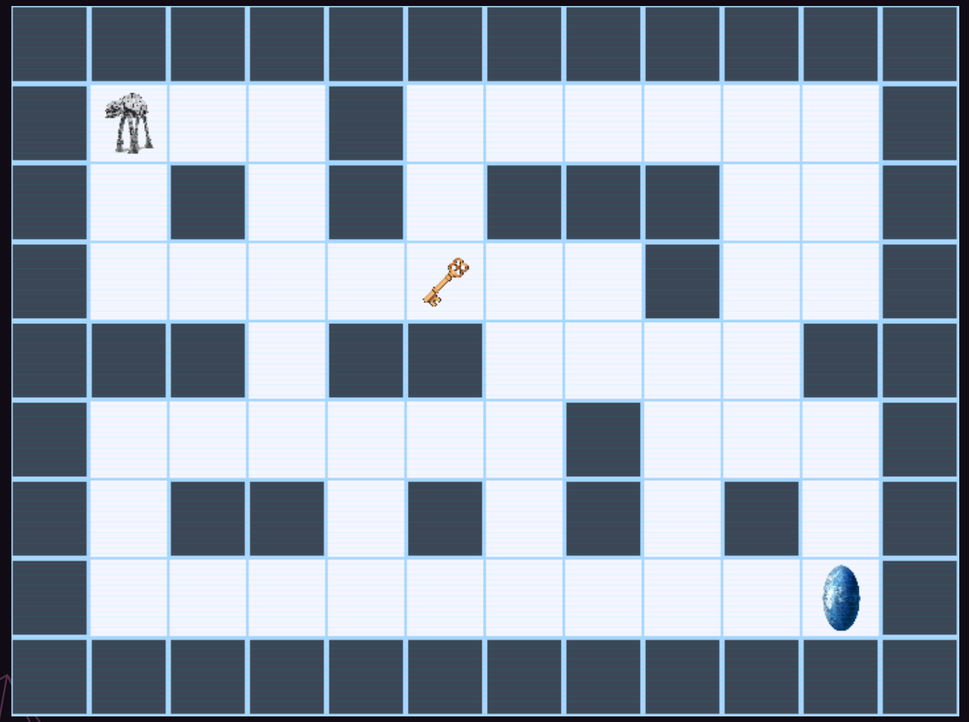
- **Unique Ice Physics**: Slide until you hit something - no stopping mid-move
- **Dual Game Modes**: Play as human or watch AI algorithms solve the maze
- **AI Pathfinding**: Implements BFS and DFS with key collection logic
- **Progressive Difficulty**: 5 levels with increasing complexity and enemy patrols
- **Star Wars Theme**: Complete with Hoth atmosphere and Rebel vs Empire elements

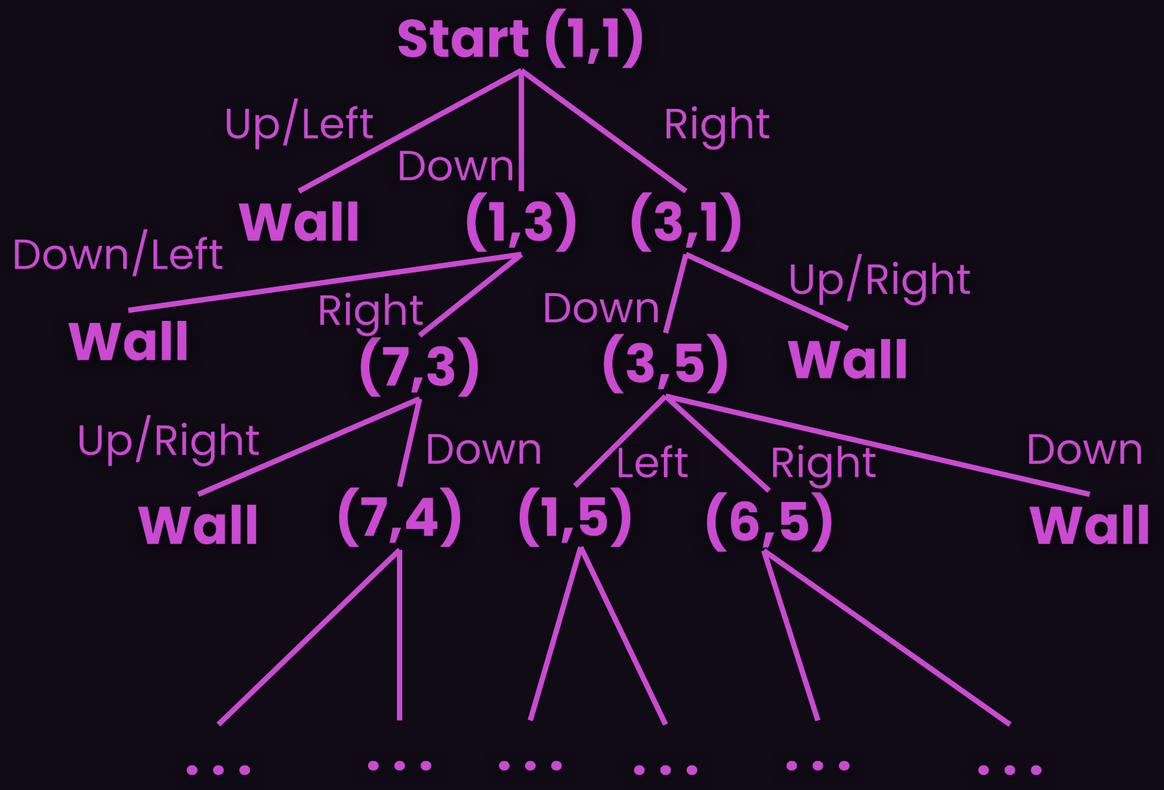
pimme05 Update README.md

- __pycache__
- .gitignore
- LICENSE
- MIRROR_OK.txt
- README.md
- key.png
- main.py
- pic1.png
- pic2.png

PARTIAL SEARCH TREE

Start State

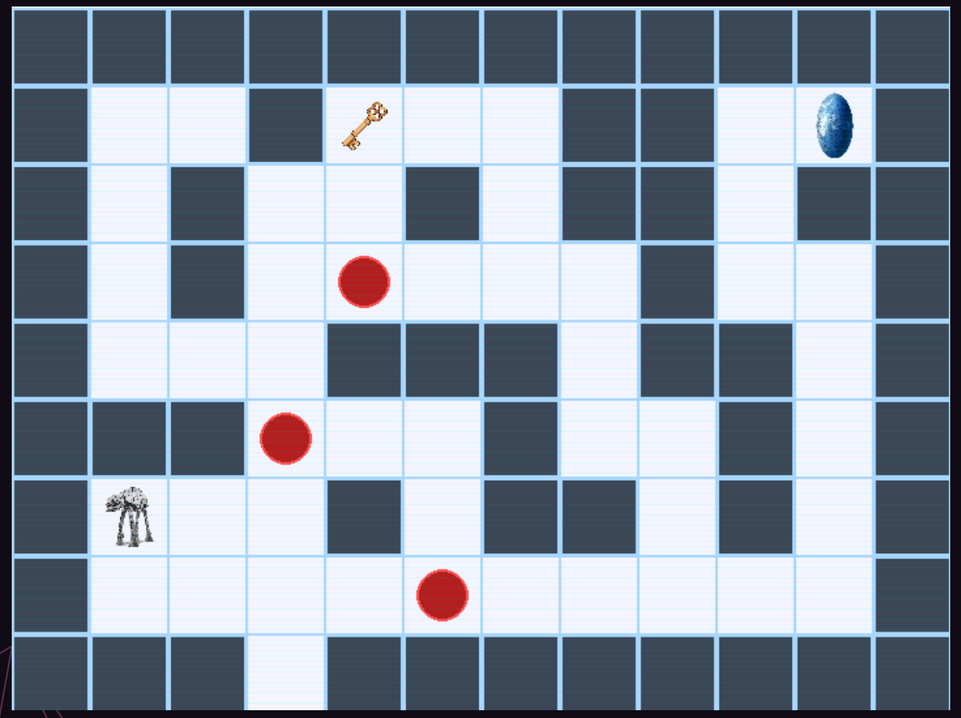


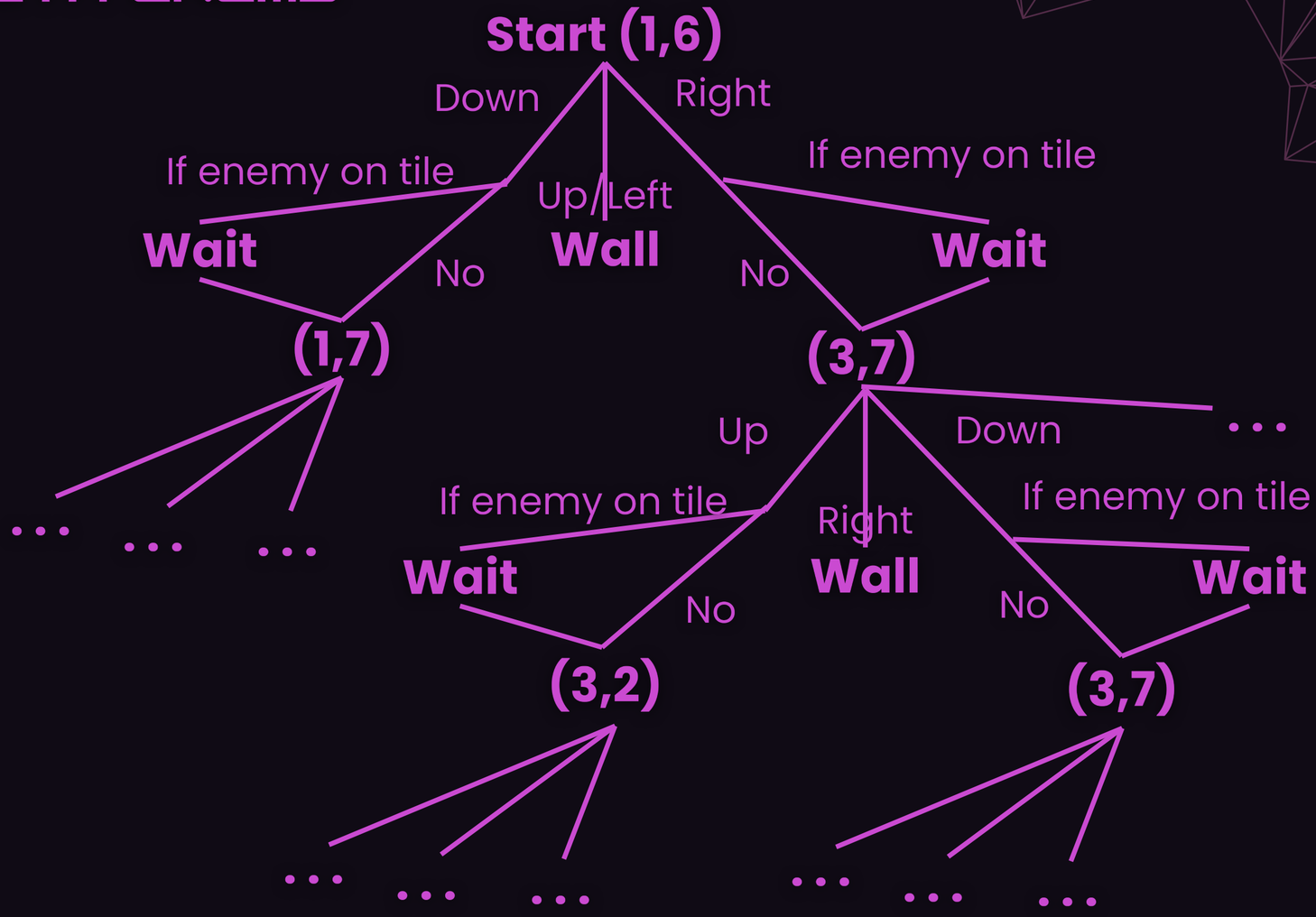


PARTIAL SEARCH TREE

WITH ENEMY

Start State





CODE IMPLEMENTATION

BFS Implementation

```
def bfs_with_key(self, start, goal, key_pos,
has_key_start=False):
        self.reset()
        self.algorithm_used = "BFS with Key Collection"

        q = deque([(start, has_key_start, [start])])
        seen = {(start, has_key_start)}

        while q:
            cur, has_key, path = q.popleft()
            self.search_order.append(cur)
            self.nodes_expanded += 1
            self.explored.add(cur)

            if cur == key_pos:
                has_key = True
            if cur == goal and has_key:
                self.path = path
                return True

            for nb in self.get_neighbors(cur):
                if nb == goal and not has_key:
                    continue
                st = (nb, has_key)
                if st not in seen:
                    seen.add(st)
                    q.append((nb, has_key, path + [nb]))
        return False
```

1. Queue and State Initialization

q = deque([(start, has_key_start, [start])])seen = {(start, has_key_start)}

- Queue Element Structure: (position, key_status, path_history)
- Seen Set: Tracks compound states (x, y, has_key) to prevent cycles
- Path Tracking: Each state maintains its complete path for solution reconstruction

2. BFS Main Loop Structure

while q: cur, has_key, path = q.popleft() # FIFO: First In, First Out

- FIFO Processing: Ensures level-by-level exploration
- State Unpacking: Extracts position, key status, and path
- Guarantees Optimality: Shortest path found first due to breadth-first nature

3. State Transition Logic

if cur == key_pos: has_key = True

- Automatic Key Collection: State change occurs when reaching key position
- Critical for Correctness: Must happen before goal testing
- State Persistence: Key status carries forward to all subsequent states

4. Goal Testing with Dependencies

if cur == goal and has_key: self.path = path
return True

- Two-Condition Success: Must have both correct position AND key
- Solution Storage: Saves complete path for execution/visualization
- Early Termination: Returns immediately upon finding solution

CODE IMPLEMENTATION

DFS Implementation

```python
def dfs_with_key(self, start, goal, key_pos, has_key_start=False):
        self.reset()
        self.algorithm_used = "DFS with Key Collection"

        stack = [(start, has_key_start, [start])]
        seen = set()

        while stack:
            cur, has_key, path = stack.pop()
            st = (cur, has_key)
            if st in seen:
                continue
            seen.add(st)

            self.search_order.append(cur)
            self.nodes_expanded += 1
            self.explored.add(cur)

            if cur == key_pos:
                has_key = True
            if cur == goal and has_key:
                self.path = path
                return True

            for nb in reversed(self.get_neighbors(cur)):
                if nb == goal and not has_key:
                    continue
                nst = (nb, has_key)
                if nst not in seen:
                    stack.append((nb, has_key, path + [nb]))
        return False
```

1. Stack-Based Exploration (LIFO)

stack = [(start, has_key_start, [start])]
cur, has_key, path = stack.pop()

- LIFO Behavior: Explores deepest path first
- Memory Efficient: Only stores current path, not all frontier nodes
- Different from BFS: May find suboptimal (longer) solutions first

2. Duplicate State Detection

st = (cur, has_key)
if st in seen:
continue
seen.add(st)

- Post-pop checking: Tests for duplicates after removing from stack
- Compound state tracking: Same position with different key status = different states
- Cycle prevention: Avoids infinite loops in graph traversal

3. Reversed Neighbor Addition

for nb in reversed(self.get_neighbors(cur)):
stack.append((nb, has_key, path + [nb]))

- Why reversed?: Maintains consistent exploration order with standard DFS
- Stack effect: First neighbor becomes deepest in stack, explored last
- Path reconstruction: Each state carries its complete path history

CODE IMPLEMENTATION

Visualization Features Real-time Search Visualization

```
def draw_grid(self):

      if not self.manual_mode and self.search.search_order:
          for i in range(min(self.visualization_step, len(self.search.search_order))):
              pos = self.search.search_order[i]
              color = SABER_BLUE if "BFS" in str(self.current_algorithm) else EMPIRE_RED
              s = pygame.Surface((CELL_SIZE - 10, CELL_SIZE - 10))
              s.set_alpha(80); s.fill(color)
              self.screen.blit(s, (pos[0]*CELL_SIZE + 5, pos[1]*CELL_SIZE + 5))

      if not self.manual_mode and not self.animating and self.solution_found and self.search.path:
          for pos in self.search.path:
              if pos in (self.maze.start_pos, self.maze.goal_pos): continue
              s = pygame.Surface((CELL_SIZE - 30, CELL_SIZE - 30))
              s.set_alpha(128); s.fill(JEDI_GREEN)
              self.screen.blit(s, (pos[0]*CELL_SIZE + 15, pos[1]*CELL_SIZE + 15))
```

Animation System

- **Step-by-step expansion:** Shows algorithm thinking process
- **Color coding:** Different colors for BFS vs DFS exploration
- **Solution highlighting:** Final path shown in green
- **Real-time metrics:** Node count, algorithm type display

CODE IMPLEMENTATION

Advanced Features

Enemy System Integration

```
def _step_enemies(self):
        if not self.enemies or self.game_completed or self.game_over:
            return
        now = time.time()
        if now - self._enemy_last_step < self.enemy_step_interval:
            return
        self._enemy_last_step = now

        for e in self.enemies:
            nx, ny = e["x"] + e["dx"], e["y"] + e["dy"]

            if self.maze.is_wall(nx, ny):
                e["dx"] *= -1
                e["dy"] *= -1
                nx, ny = e["x"] + e["dx"], e["y"] + e["dy"]
                if self.maze.is_wall(nx, ny):
                    for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
                        tx, ty = e["x"] + dx, e["y"] + dy
                        if not self.maze.is_wall(tx, ty):
                            e["dx"], e["dy"] = dx, dy
                            nx, ny = tx, ty
                            break
                    else:
                        continue

            e["x"], e["y"] = nx, ny

            if self.manual_mode and (e["x"], e["y"]) == self.player_pos:
                self._trigger_game_over()
```

1. Timing System

now = time.time()
if now - self._enemy_last_step < self.enemy_step_interval:
return

- This creates frame-rate independent movement. Enemies move every 0.35 seconds regardless of how fast the game runs.

2. Simple AI Behavior

nx, ny = e["x"] + e["dx"], e["y"] + e["dy"]

- Each enemy has a direction vector (dx, dy) that they follow persistently. This creates predictable patrol patterns.

3. Collision Recovery Algorithm

*e["dx"] *= -1*
*e["dy"] *= -1*

- The bounce behavior when hitting walls. The fallback system that searches all four directions shows robust error handling.

4. Mode-Specific Interaction

if self.manual_mode and (e["x"], e["y"]) == self.player_pos:
self._trigger_game_over()

- This is clever design - enemies pose a threat to human players but don't interfere with AI algorithm demonstrations.

CODE IMPLEMENTATION

Technical Architecture

Class Structure Overview

```
class Maze:
    # Level management and physics
    def load_level(self, level)          # 5 different maze layouts
    def slide_move(self, x, y, direction)  # Ice physics simulation
    def collect_key(self, player_pos)    # State transition trigger
    def can_exit(self, player_pos)       # Goal validation

class SearchAlgorithm:
    # Blind search implementations
    def bfs_with_key(...)                # Breadth-first search
    def dfs_with_key(...)                # Depth-first search
    def get_neighbors(self, pos)         # Action space generation

class StarWarsIceMazeGame:
    # Game engine and visualization
    def handle_events(self)              # Input processing
    def update(self)                     # Game state updates
    def draw(self)                       # Rendering pipeline
```

System Integration and Data Flow

1. Maze Class: Provides the problem domain and physics constraints
2. SearchAlgorithm Class: Implements the AI problem-solving logic
3. Game Class: Orchestrates interaction between human player and AI systems
4. Bidirectional Communication: Game calls search algorithms; algorithms query maze physics

Design Patterns Used

- State Pattern: Different behaviors for human vs AI modes
- Strategy Pattern: Interchangeable search algorithms (BFS/DFS)
- Observer Pattern: Search progress updates visualization system
- Command Pattern: Action queue for autopilot execution

CODE IMPLEMENTATION

Multi-Level Design

Progressive Difficulty Scaling

```
def load_level(self, level):
    if level == 1:
        self.grid = [
            [1,1,1,1,1,1,1,1,1,1,1,1],
            [1,0,0,0,1,0,0,0,0,0,0,1],
            [1,0,1,0,1,0,1,1,1,1,0,1],
            [1,0,1,0,0,0,0,0,0,1,0,1],
            [1,0,1,1,1,1,1,1,1,0,1,0,1],
            [1,0,0,0,0,0,0,1,0,0,0,1],
            [1,1,1,0,1,1,0,1,1,1,0,1],
            [1,0,0,0,0,0,0,0,0,1,0,1],
            [1,1,1,1,1,1,1,1,1,1,1,1],
        ]
        self.start_pos = (1, 1)
        self.goal_pos = (10, 7)
        self.key_pos = (5, 3)
        self.enemy_spawns = []
    elif level == 2:
        # Same size, add patrol enemies
        self.enemy_spawns = [
            (2, 7, 1, 0),   # Horizontal patrol
            (5, 1, 0, 1),   # Vertical patrol
        ]
```

Level Progression Features

- Grid Size: 12×9 → 16×10 (levels 1-3 vs 4-5)
- Enemy Count: 0 → 4 patrols
- Layout Complexity: Simple paths → Intricate maze designs
- Search Challenge: 15-25 nodes → 45-120 nodes expanded
- Start position: (1, 1) → top left corner inside the maze.
- Goal position: (10, 7) → exit point.
- Key position: (5, 3) → must be collected before finishing.
- Enemies: none on level 1 (self.enemy_spawns = []).

THANK YOU