Grid Runner:

Multi-Objective AI Pathfinding in 2D Environments

CS5100 - Final Project | Summer 2025 | Jack Einbinder

Introduction

Problem Statement

Pathfinding is a central challenge in Artificial Intelligence (AI) and robotics, where agents must determine an optimal route from a starting location to a goal in a known or unknown environment. In real-world applications such as autonomous vehicles, warehouse robots, and game engines, agents are often required to satisfy additional objectives, such as collecting items, maximizing score, or avoiding hazards under movement limitations, making the pathfinding task more complex and computationally intensive.

This project focuses on the problem of multi-objective pathfinding in a 2D grid environment. An agent must navigate toward a goal while simultaneously avoiding walls, collecting coins and picking up trash, all within a limited number of steps. To address this problem, the agent is equipped with informed search algorithms that use a reward-aware heuristic:

- Reward-aware informed algorithms
    - A* Search
    - Greedy Best-First Search

These are evaluated against local search algorithms that instead use a reward-aware evaluation function:

- Reward-aware local search algorithms
    - Randomized Hill Climbing
    - Simulated Annealing

Finally, both reward-aware approaches are compared with uninformed, reward-unaware complete search algorithms, which are generally more computationally expensive than local search and may have similar compute costs to the informed reward-aware algorithms:

- Reward-unaware uninformed algorithms
    - Breadth-First Search (BFS)
    - Uniform Cost Search (UCS)

Motivation and Goals

Planning and executing intelligent navigation strategies is imperative for agents operating in a traversable environment. The need for these strategies extends beyond agents moving through physical space, such as warehouse robots or autonomous vehicles, and includes agents traversing decision trees or game states where different states may carry different values. This underscores the importance of pathfinding approaches beyond simple spatial movement and highlights their relevance in broader planning tasks (Russell & Norvig, 2020).

Classical search algorithms provide structured methods for exploring and evaluating candidate states within a state space. Uninformed algorithms, like Breadth-First Search (BFS) and Uniform Cost Search (UCS), traverse the state space without using information about the goal's location, expanding nodes either uniformly or by lowest path cost (Russell & Norvig, 2020; Djikstra, 1959). Informed algorithms, such as A* Search and Greedy Best-First Search, use heuristics to guide exploration more efficiently, with A* Search balancing path cost and estimated distance while Greedy Best-First Search relies solely on its heuristic (Hart, Nilsson, & Raphael, 1968; Pearl, 1984). Reward-aware local search algorithms, including Randomized Hill Climbing and Simulated Annealing, evaluate states based on both estimated distance and potential rewards from collecting objects in the environment. These methods introduce probability and randomness to explore the state space more freely and attempt to escape local maxima, although they may still get stuck or fail to find an optimal solution (Russell & Norvig, 2020; Kirkpatrick, Gelatt & Vecchi, 1983).

The goal of this project is to evaluate these three algorithmic strategies (reward-unaware uninformed, reward-aware informed, and reward-aware stochastic) in the context of multi-objective pathfinding, where an agent must avoid obstacles, collect rewards, and reach a goal within a limited number of steps.

Background

Related Work

Classical search algorithms are foundational tools in Artificial Intelligence for solving planning and navigation problems. Early work in the field began in 1959 with Dijkstra's introduction of an optimal pathfinding algorithm for traversing weighted graphs with positive edge weights (Dijkstra 1959; Haepler et al., 2023). Dijkstra's algorithm laid the groundwork for Uniform Cost Search, which generalizes Breadth-First Search by expanding the node with the lowest cumulative path cost at each step. In 1968, Hart, Nilsson and Raphael proposed the A* algorithm, which combines path cost and a heuristic estimating the agent's distance to the goal (Hart, Nilsson, & Raphael, 1968). In 1984, Pearl introduced Greedy Best-First Search, a heuristic-based method that, while neither optimal nor complete, guides exploration using only the estimated distance to the goal (Pearl, 1984).

Inspired by annealing in metalworking, where heated metal is slowly cooled to remove internal stresses, Kirkpatrick, Gelatt, and Vecchi introduced Simulated Annealing in 1983 (Kirkpatrick, Gelatt & Vecchi, 1983). This approach incorporates probabilistic decision-making into pathfinding, allowing agents to escape local maxima by occasionally accepting worse solutions early in a search. The probability of selecting worse choices decreases over time according to a temperature-based cooling schedule. Simulated Annealing addresses a key limitation of more simple local search algorithms like Hill Climbing, which often become trapped in local maxima, plateaus, or ridges.

More recent work explores how classical search algorithms can be adapted for use in constrained and multi-objective domains, where agents must satisfy competing goals or operate under limited resources. In 2015, Aine et al. introduced Multi-Heuristic A* (MHA*), an extension of A* that uses multiple heuristics simultaneously to guide exploration. This allows the algorithm to balance different objectives more effectively, especially in cases where a single heuristic might not capture the full complexity of the environment (Aine et al., 2015).

Relevant AI Concepts

Pathfinding

Pathfinding is an algorithmic technique for finding the shortest route between a start and goal state. It is typically implemented using classical search algorithms like Uniform Cost Search or A*, with complex problems often requiring modified variants. Pathfinding algorithms are categorized as uninformed, which explore without heuristics, or informed, which use a heuristic to estimate the distance to the goal. Search methods are evaluated by completeness (whether a solution is found if one exists), optimality (whether the path is lowest-cost), and their space and time complexity.

Informed Pathfinding Algorithms and Heuristic Functions

Informed pathfinding algorithms use heuristic functions to estimate the distance to a goal state, guiding exploration through the state space. By prioritizing promising paths, they can bypass suboptimal states and traverse more efficiently than uninformed algorithms like Breadth-First Search and Uniform Cost Search. A* combines actual path cost and estimated distance, while Greedy Best-First Search uses only the heuristic, ignoring path cost.

Local Search Algorithms and Evaluation Functions

Local search algorithms such as Hill Climbing use a greedy strategy that makes the locally optimal choice at each step, often seeking to minimize or maximize a goal based on an evaluation function, a function that incorporates a heuristic to guide the search. These algorithms improve efficiency by avoiding full traversal of the state space, but they can fail to find a global optimum by getting stuck in local maxima, plateaus with no gradient, or ridges that require moving away from the gradient. Randomized Hill Climbing mitigates this by shuffling the order in which neighbors are considered, while Simulated Annealing goes further by occasionally accepting worse moves early in the search and reducing this behavior over time.

Multi-Objective Pathfinding

Classical pathfinding algorithms focus on finding the shortest distance, or path of least cost between a start state and a goal state. Multi-objective pathfinding algorithms, on the other hand, consider additional goals, such as reward collection, requiring either multiple heuristics, or custom heuristics that balance competing objectives.

Methodology

Project Overview

Grid Runner is an AI game where an agent navigates a 10x10 grid in search of a goal. The user sets a step limit, and the agent must plan the most efficient path to the goal within that constraint. Users can dynamically adjust the environment during execution by dragging the agent or goal to new positions, prompting the agent to replan its path in real time.

Users can also place the following objects on the grid:

- Walls
    - Impassable cells that restrict the agent's movement. These can be used to create obstacles and mazes.
- Coins
    - Collectible objects that provide a reward for the agent. Agents using informed or local search pathfinding algorithms, including A* Search, Greedy Best-First Search, Randomized Hill Climbing, and Simulated Annealing, will attempt to collect as many coins as they can while still reaching the goal within the step limit.
- Trash
    - A competing reward that the agent must balance with coins. The user can assign reward values for both trash and coins, and the agent will attempt to maximize its cumulative reward within the step limit.

Users can choose from a variety of pathfinding strategies. Grid Runner currently supports:

- Reward-unaware uninformed algorithms
    - Breadth-First Search (BFS)
    - Uniform Cost Search (UCS)

- Reward-aware informed algorithms
    - A* Search
    - Greedy Best-First Search

- Reward-aware local search algorithms
    - Randomized Hill Climbing
    - Simulated Annealing

Informed and reward-aware pathfinding algorithms use a heuristic function to estimate the agent's distance to the goal. By subtracting the reward value of a cell in the grid from the estimated (Manhattan) distance, more valuable cells effectively appear closer to the goal:

$$h_{reward}(n) = h(n) - reward\_value(row, col)$$

Reward-aware local search algorithms instead use an evaluation function that incorporates the heuristic, where the reward value of a cell is subtracted from its estimated (Manhattan) distance making closer rewards more favorable than distant ones:

$$e_{reward}(n) = reward\_value(row, col) - h(n)$$

Tools and Libraries

Grid Runner uses the following tools and frameworks:

Tool/Framework	Purpose
Python 3.13	Programming language used for this project
Pygame	Python game engine used to render the GUI and handle user input
Matplotlib	For visualization of performance data
NumPy	For efficient representation and manipulation of the game grid
random, os, json, time, itertools, heapq, collections, abc, enum	Standard Python libraries used for randomness, file I/O, time measurement, and internal pathfinder logic

Pathfinding Algorithms

All algorithm implementations are provided in the appendix.

Breadth-First Search (BFS)

BFS starts by adding the start state to a queue. While the queue is not empty, the algorithm pops the front node, checks if it is the goal, and adds all unvisited neighbors to the back of the queue. This process continues until the goal is found or the queue is exhausted, in which case the algorithm returns an empty list. BFS incurs a time complexity of $O(b^d)$ where b is the branching factor, and d is the depth of the goal node. Since the BFS implementation in Grid Runner stores visited nodes, the space complexity is also $O(b^d)$. BFS is both complete and optimal when all step costs are equal and the branching factor is finite.

Uniform Cost Search (UCS)

UCS functions similarly to BFS but uses a priority queue instead of a standard queue. This allows UCS to expand the node with the lowest total path cost at each step. When storing visited nodes, both the time and space complexity of UCS are $O(b^{\lfloor 1 + C^*/\varepsilon \rfloor})$ where b is the branching factor, C* is the cost of the optimal solution and ε is the minimum step cost. UCS is both complete and optimal when all step costs are positive and the branching factor is finite.

A* Search (A*)

A* functions similarly to UCS but incorporates a heuristic to guide its search. At each step, A* expands the node with the lowest estimated total cost *f(n)*, calculated as *f(n) = g(n) + h(n)*, where *g(n)* is the cost to reach the node and *h(n)* is a heuristic estimate of the cost to reach the goal. When *h(n)* is admissible, meaning it never overestimates the cost to reach the goal, A* is both complete and optimal. The time and space complexity of A* are $O(b^d)$ where *b* is the branching factor and *d* is the depth of the optimal solution. The A* implementation in Grid Runner uses the manhattan distance ($|x1 - x2| + |y1 - y2|$) as its heuristic and is modified to account for competing rewards by subtracting a weighted reward score from the total cost function. As a result, nodes with higher cumulative rewards are prioritized even if their path is longer, provided they do not exceed the step limit.

Greedy Best-First Search (GBFS)

GBFS functions similarly to A*, using both a priority queue and heuristic function to select the optimal node at each step. In contrast with A*, GBFS does not consider the path cost *g(n)*, instead focusing solely on the heuristic function *h(n)*. Because the path cost is ignored, GBFS is neither complete nor optimal. The time and space complexity of GBFS are $O(b^d)$ where *b* is the branching factor and *d* is the depth of the shallowest solution. The implementation of GBFS used in Grid Runner accounts for competing rewards by subtracting the cumulative reward from the Manhattan distance to the goal.

Randomized Hill Climbing (RHC)

RHC is a greedy local search algorithm that starts from the initial state and iteratively moves to the best neighbor based on a heuristic function. To mitigate the limitations of basic Hill Climbing, RHC randomizes the order in which neighbors are explored at each step. Despite this mitigation strategy, RHC still terminates early if none of the neighboring states offer an improvement over the current state, making it prone to getting stuck in local maxima and plateaus. Because RHC only follows one path and doesn't backtrack or explore alternatives, it is neither complete nor optimal. The time and space complexity of RHC are *O(d)* where *d* is the number of steps taken before terminating. The RHC implementation in Grid Runner accounts for competing objectives by subtracting the Manhattan distance to the goal from the reward value of a cell.

Simulated Annealing (SA)

SA is a variant of Hill Climbing that allows selecting worse states early in the search, based on a heuristic function, with this behavior gradually restricted as a temperature parameter decreases according to a cooling schedule. By occasionally abandoning a greedy strategy, SA is better at overcoming local maxima, plateaus, and ridges than basic Hill Climbing or RHC, which only select neighboring states that improve upon the current state. However, because SA's behavior depends on the cooling schedule, it can still terminate early if it fails to reach the goal before the temperature becomes too low. Like RHC, SA only follows one path and doesn't backtrack or explore alternatives. As a result, it is neither complete nor optimal. The time and space complexity of SA are *O(d)* where *d* is the number of steps taken before terminating. Similar to

RHC, the implementation of SA in Grid Runner accounts for competing objectives by subtracting the Manhattan distance to the goal from the reward value of a cell.

Results

Evaluation Strategy

To evaluate the different pathfinding strategies, each algorithm will guide the agent through 10 runs on a common evaluation map, shown below. The map includes both obstacles and rewards, allowing us to test the algorithms' handling of multiple objectives. A low step limit is imposed to force trade-offs, requiring the agent to prioritize between collecting rewards and reaching the goal. Since coins are worth twice as much as trash, a reward-aware pathfinder should prioritize coins when forced to choose. Algorithms will be compared based on their average performance across several key metrics.

Grid Runner Evaluation Map

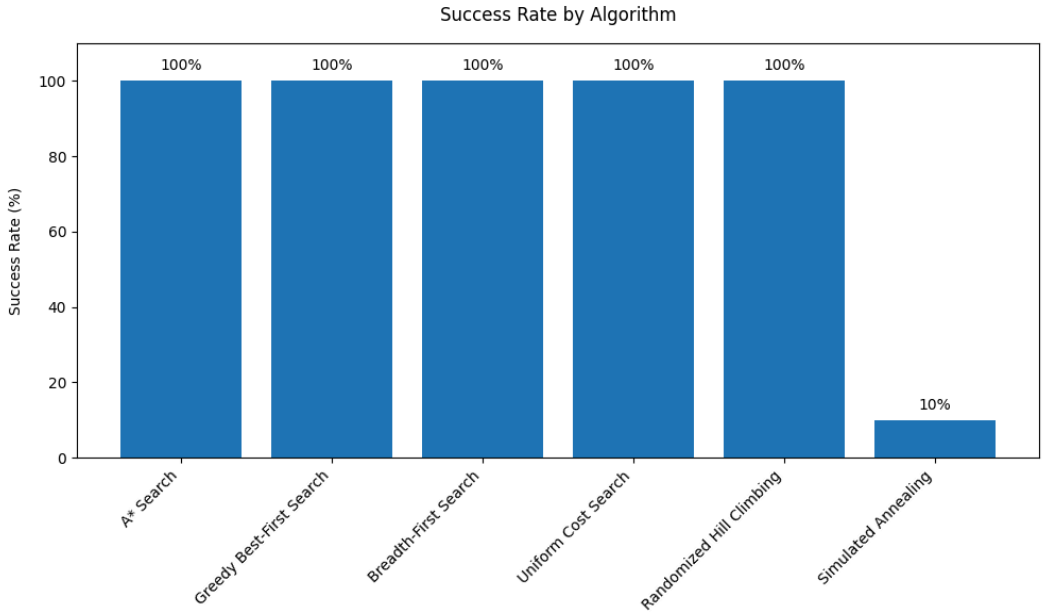
Evaluation Metrics

Metric	Description
Success Rate	Percentage of runs where the agent reaches the goal within the step limit
Average Score	Average cumulative reward (coins and trash) collected across successful runs
Average Steps Taken	Average number of steps taken across successful runs
Average States Explored	Average number of states expanded before reaching the goal
Average Compute Time	Average time in seconds to compute a path to the goal

These metrics allow us to evaluate the pathfinding algorithms based on their success in reaching the goal, the efficiency of the paths they generate, their ability to balance multiple objectives, and their overall computational performance.

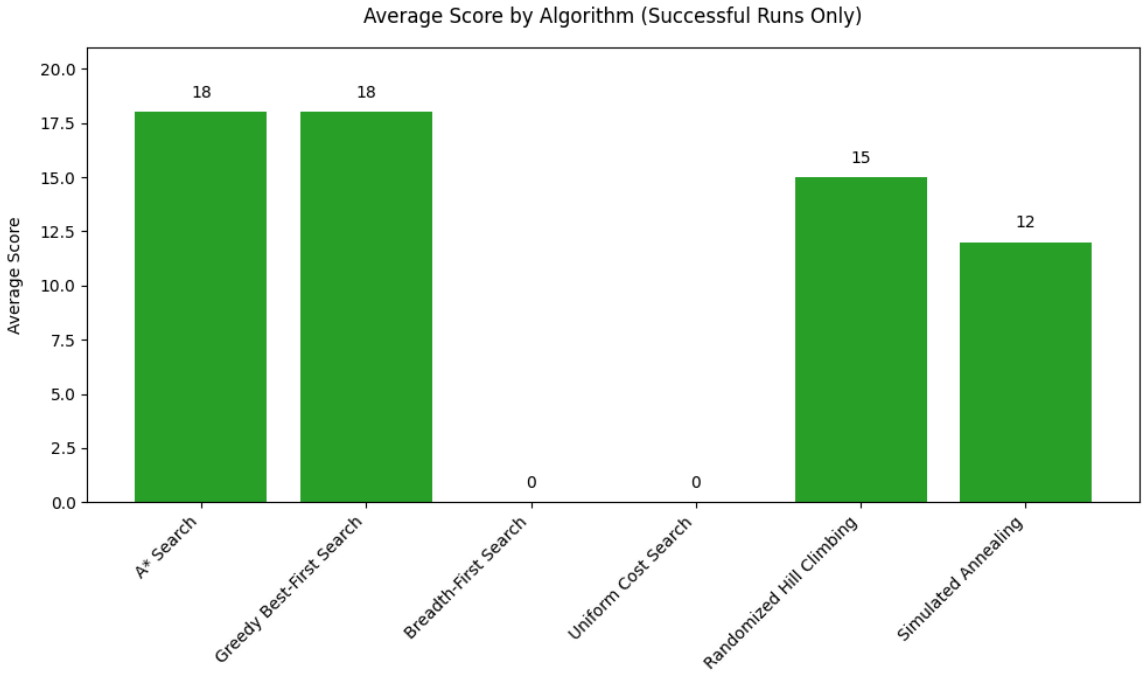
Visualizations

Success Rate



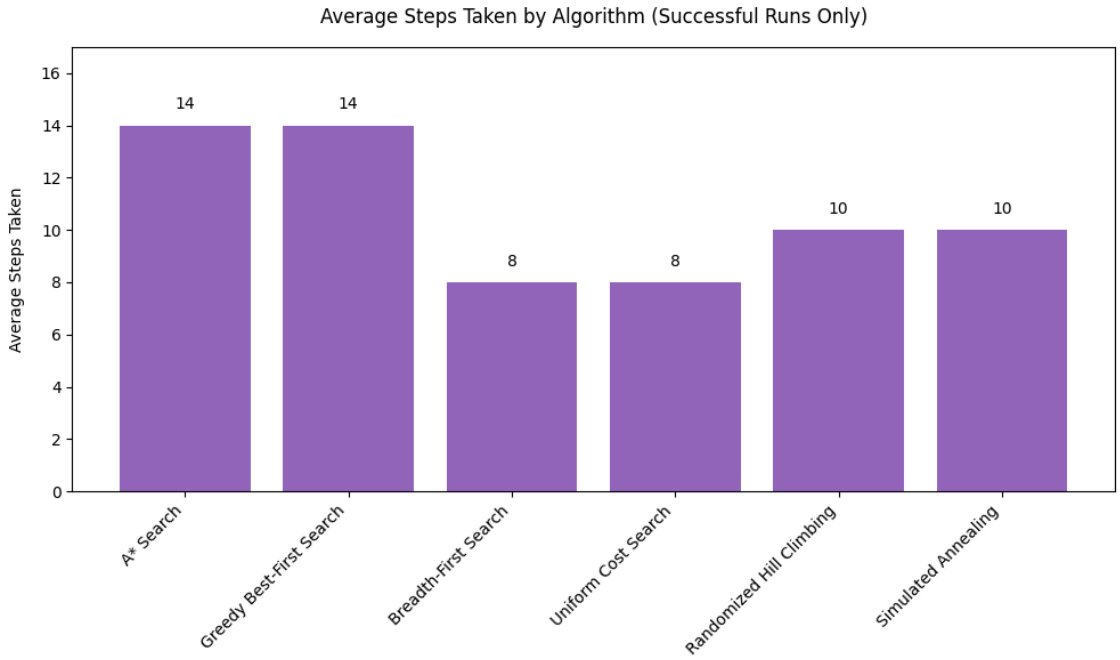
All algorithms consistently found the goal except Simulated Annealing, which often failed.

Average Score



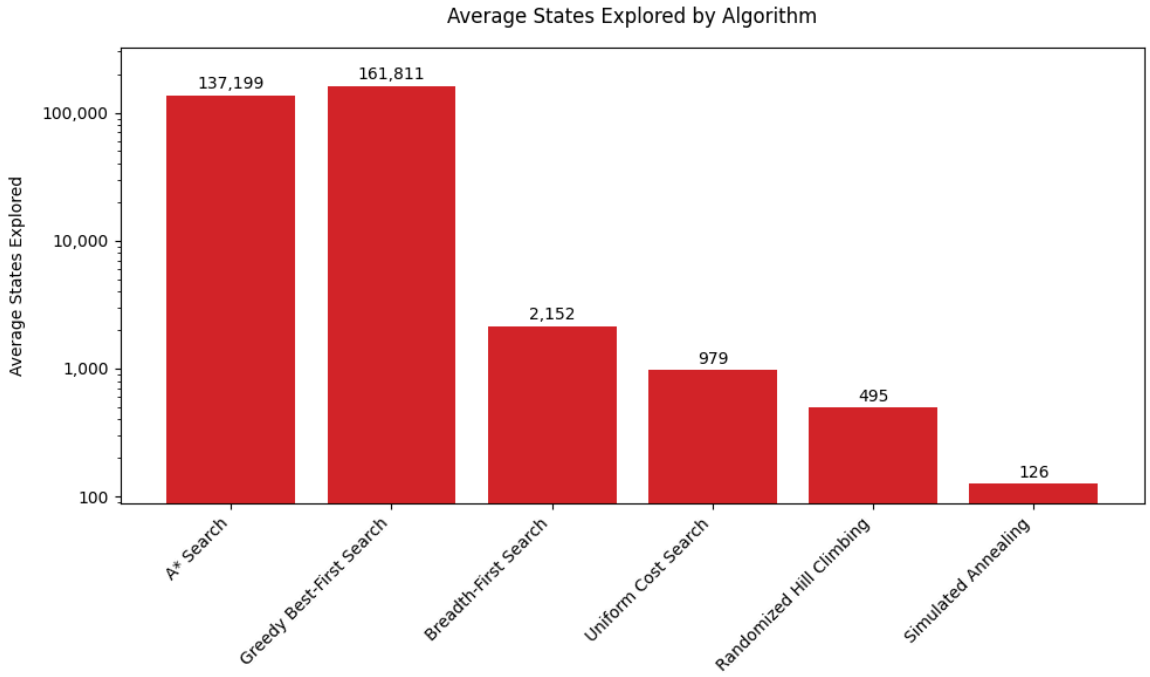
Reward-aware algorithms achieved the highest scores, while reward-unaware algorithms prioritized shorter paths over collecting rewards.

Average Steps Taken



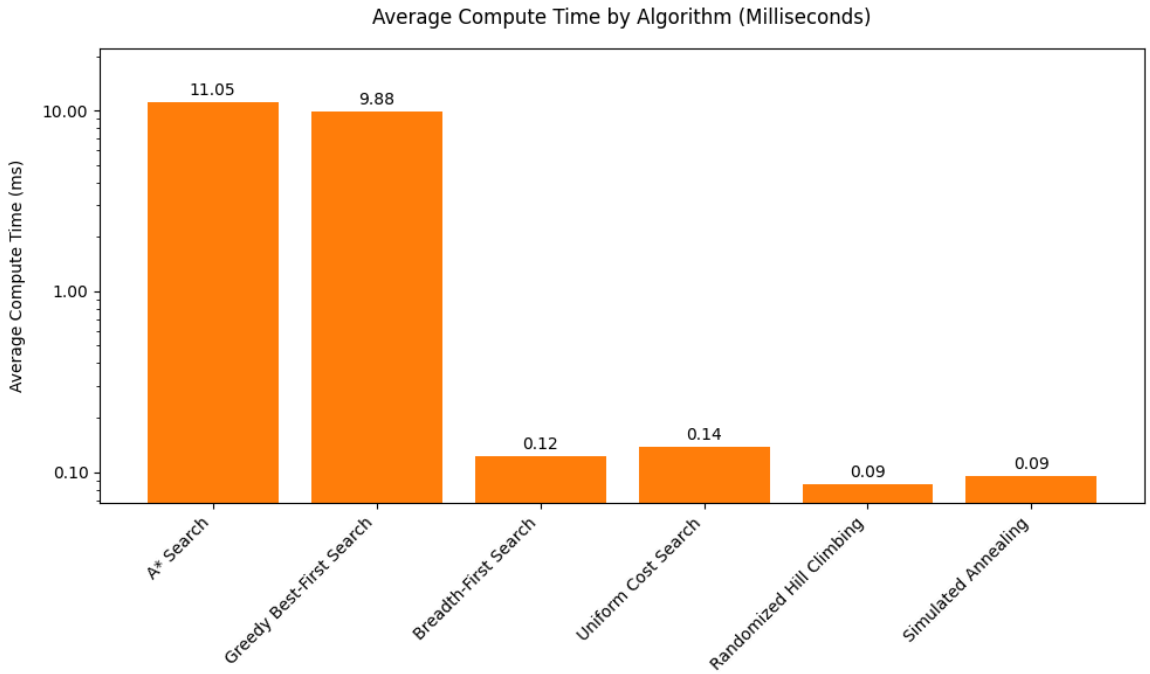
Reward-aware algorithms took longer routes in order to maximize rewards.

Average States Explored



Reward-aware informed algorithms expanded significantly more states than the rest, while local search algorithms expanded the fewest.

Average Compute Time



Reward-aware informed algorithms are significantly more computationally expensive than the others.

Discussion

Interpretation of Results

Reward-aware pathfinding algorithms (A*, GBFS, RHC, and SA) were able to prioritize reward collection while still reaching the goal within the step limit. This demonstrates that incorporating additional objectives into an informed algorithm's heuristic or evaluation function enables it to balance multiple goals effectively. A* and GBFS both achieved an average score of 18, suggesting that A*'s consideration of path cost was less important than reward collection, as long as the goal could still be reached in time.

Local search algorithms (RHC and SA) also collected rewards and reached the goal but faced greater difficulty finding solutions, as highlighted by SA's low success rate of 10%. RHC's shortcomings were less obvious here but became clear in earlier tests on more complex grids (excluded from final evaluation because SA could not complete them within 10 attempts), where RHC also struggled despite randomization. This suggests that the reward-aware evaluation function hindered these algorithms' ability to find the goal, likely because incorporating reward values obscured direct path finding.

As expected, the uninformed algorithms (BFS and UCS) did not consider rewards and instead prioritized reaching the goal in the fewest steps. Both achieved scores of 0 as well as the lowest average step count of 8, significantly lower than the rest.

Reward-aware informed algorithms were the most computationally expensive, expanding significantly more nodes than the others: 137,199 and 161,811 on average for A* and GBFS respectively. Their compute times were also the highest, at 11.05ms and 9.88ms respectively. This suggests that incorporating reward-awareness reduces their ability to efficiently prune the state space, which can likely be mitigated by using a more appropriate heuristic than estimated distance to the goal. Local search algorithms were the least expensive despite reward-awareness, expanding only 495 (RHC) and 126 (SA) nodes, with average compute times of 0.9ms. BFS and UCS were more expensive than local search but significantly cheaper than the reward-aware informed algorithms.

Overall, these results align with the well-documented time complexities of classical search algorithms. Incorporating additional objectives, such as reward collection, substantially increases computational cost, as demonstrated by the gap between reward-aware informed algorithms and their reward-unaware, uninformed, and complete counterparts. This underscores the need for improved heuristic functions that account for reward optimization more effectively.

Limitations

The evaluation of the pathfinding algorithms revealed limitations in both the implementations as well as the underlying game mechanics. As discussed previously, A* and GBFS were computationally expensive, occasionally causing the game to slow down or crash when too

many items were placed on the board. This was mitigated through compute-saving strategies such as bit masking to efficiently update the game board, pruning state re-expansion when paths offered no improvement, and by restricting pathfinding to the play state to avoid unnecessary replanning.

Local search algorithms struggled to find the goal on a 10×10 board, even with modest obstacles. To ensure all algorithms could complete the evaluation, a simplified map using only half the grid was created. Even then, SA barely completed the task successfully a single time.

Given the purpose of this project—to explore methods for enabling classical search algorithms to prioritize multiple objectives through reward-aware heuristic and evaluation functions—the immense computational cost introduced by suboptimal base heuristics, in this case Manhattan distance, was not fully recognized until the results became apparent through performance data. New insight into these costs suggests the need to explore more efficient heuristic functions, directly comparing them with their classical counterparts to better understand the computational trade-offs. Such comparisons could provide deeper insight into designing heuristics that efficiently support multi-objective pathfinding.

Potential Improvements

Grid Runner's limitations can be addressed through the following changes:

- Explore more efficient heuristic functions that reduce computational load.
- Expand the algorithm set for comparison. Including classical versions of A*, GBFS, and Hill Climbing would help clarify the impact of reward-awareness. Reward-aware variants of BFS and UCS could also be added to evaluate their performance against their classical counterparts.
- Reduce the grid size to 5×5. A smaller grid would allow local search algorithms to better compete with more complete pathfinding algorithms like A*, GBFS, BFS, and UCS. It would also reduce total node expansion across the board, making it possible to efficiently run more complex maps.
- Apply computation-saving techniques. Optimizations such as bit masking and pruning should be implemented across all algorithms to ensure no method is slower than necessary.

Conclusion

Summary of achievements

This project implemented and evaluated several pathfinding algorithms with the goal of addressing multiple objectives. Classical algorithms were adapted with reward-aware heuristics and evaluation functions to guide an agent through a 2D grid, collecting coins and trash while reaching a fixed goal within a limited number of steps. The evaluation highlighted the

computational cost introduced by reward-awareness and the difficulties local search algorithms face in reaching the goal due to their limited exploration of the state space.

Lessons Learned

This project demonstrates that multiple objectives can be effectively handled using a single reward-aware heuristic, suggesting that approaches like MHA*, which rely on multiple heuristics, may not always be necessary (Aine et al., 2015). The implementation and evaluation of reward-aware informed algorithms, such as A* and GBFS, also revealed their significant computational cost, underscoring the importance of efficient heuristics and optimization strategies in further applications. Future work might focus on finding the most appropriate heuristic functions for specific multi-objective problems.

References

Russell, S. J., & Norvig, P. (2020). Artificial intelligence: A modern approach (4th ed.). Pearson

Dijkstra, E. (1959). A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1, 269–271. https://ir.cwi.nl/pub/9256/9256D.pdf

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968, July). A Formal Basis for The Heuristic Determination of Minimum Cost Paths https://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf

Pearl, J. (1984). Heuristics: Intelligent search strategies for computer problem solving. Addison-Wesley.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. Science, 220(4598), 671–680. https://doi.org/10.1126/science.220.4598.671

Haeupler, B., Hladík, R., Rozhoň, V., Tarjan, R. E., & Tětek, J. (2023). Universal Optimality of Dijkstra via Beyond-Worst-Case Heaps. ArXiv.org. https://arxiv.org/abs/2311.11793

Aine, S., Swaminathan, S., Narayanan, V., Hwang, V., & Likhachev, M. (n.d.). Multi-Heuristic A* Journal name ©The Author(s) 2015. https://www.cs.cmu.edu/~maxim/files/mha_ijrr15.pdf

Appendix

Algorithm Implementations

Breadth-First Search (BFS)

```python
from collections import deque
from .base_pathfinder import BasePathfinder
from ..core.grid import Grid


class BFSPathfinder(BasePathfinder):
    def __init__(self, step_limit: int, coin_reward: int, trash_reward: int):
        super().__init__(step_limit, coin_reward, trash_reward)

    def search(self, state_space: Grid, start: tuple[int, int], goal: tuple[int, int]) -> list[tuple[int, int]]:
        frontier = deque()
        frontier.append((start, [start]))
        visited = set()
        while frontier:
            current, path = frontier.popleft()  # Pop node from frontier
            self.states_explored += 1
            if len(path) - 1 > self.step_limit or current in visited:  # Skip invalid or visited nodes
                continue
            visited.add(current)  # Add node to visited set
            if current == goal:  # Goal test
                self.final_path = path
                return path
            for neighbor in state_space.get_adjacent(*current):  # Enqueue neighbors
                if neighbor not in visited:
                    frontier.append((neighbor, path + [neighbor]))
        self.final_path = []
        return []
```

Uniform-Cost Search (UCS)

```python
import heapq
from .base_pathfinder import BasePathfinder
from ..core.grid import Grid


class UCSPathfinder(BasePathfinder):
    def __init__(self, step_limit: int, coin_reward: int, trash_reward: int):
        super().__init__(step_limit, coin_reward, trash_reward)

    def search(self, state_space: Grid, start: tuple[int, int], goal: tuple[int, int]) -> list[tuple[int, int]]:
        frontier = []
        heapq.heappush(frontier, __item: (0, start, [start]))  # Priority queue is frontier
        visited = {}
        while frontier:
            self.states_explored += 1
            cost_so_far, current, path = heapq.heappop(frontier)  # Pop node from frontier
            if len(path) - 1 > self.step_limit:  # Skip invalid nodes
                continue
            if current in visited and cost_so_far ≥ visited[current]:  # Skip if state is not improvement
                continue
            visited[current] = cost_so_far  # Add node to visited dict
            if current == goal:  # Goal test
                self.final_path = path
                return path
            for neighbor in state_space.get_adjacent(*current):  # Push neighbors to pqueue
                total_cost = cost_so_far + 1
                new_path = path + [neighbor]
                heapq.heappush(frontier, __item: (total_cost, neighbor, new_path))
        self.final_path = []
        return []
```

A* Search

```python
import heapq
from .reward_aware_pathfinder import RewardAwarePathfinder
from ..core.grid import Grid
from ..enums.game_object import GameObject
from itertools import count


class RewardAStarPathfinder(RewardAwarePathfinder):
    def __init__(self, step_limit, coin_reward, trash_reward, heuristic=None, reward_weight=1.0):
        super().__init__(step_limit, coin_reward, trash_reward)
        self.heuristic = heuristic or self.manhattan
        self.reward_weight = reward_weight

    @staticmethod
    def manhattan(a: tuple[int, int], b: tuple[int, int]) -> int:
        return abs(a[0] - b[0]) + abs(a[1] - b[1])  # Manhattan distance

    @staticmethod
    def pos_to_bit(row: int, col: int, cols: int) -> int:
        return 1 << (row * cols + col)

    def search(self, state_space: Grid, start: tuple[int, int], goal: tuple[int, int]) -> list[tuple[int, int]]:
        initial = {
            'pos': start,
            'score': 0,
            'steps': 0,
            'collected_mask': 0,
            'parent': None
        }
        frontier = []
        counter = count()
        heapq.heappush(frontier, __item: (0, next(counter), initial))  # Priority queue is frontier
        visited = {}
        best_goal_score = float('-inf')  # Track best score
        best_goal_node = None  # Track best node
        while frontier:
            self.states_explored += 1
            _, _, node = heapq.heappop(frontier)  # Pop node from frontier
            pos = node['pos']
            score = node['score']
            steps = node['steps']
            collected_mask = node['collected_mask']
            if pos == goal and steps <= self.step_limit:  # Goal test
                if score > best_goal_score:  # Skip state if no improvement
                    best_goal_score = score
                    best_goal_node = node
                continue
            if steps > self.step_limit:  # Skip invalid nodes
                continue
```

```
            state_key = (pos, collected_mask)
            if state_key in visited:  # Skip state if no improvement
                prev_score, prev_steps = visited[state_key]
                if score ⩽ prev_score and steps ⩾ prev_steps:
                    continue
            visited[state_key] = (score, steps)
            for neighbor in state_space.get_adjacent(*pos):
                r, c = neighbor
                cell_type = state_space.get_cell_type(r, c)
                bit = self.pos_to_bit(r, c, state_space.cols)  # Use bit masking to quickly check and modify grid
                reward = 0
                new_mask = collected_mask
                if cell_type in (GameObject.COIN, GameObject.TRASH) and not (collected_mask & bit):
                    reward = self.get_reward(cell_type)  # Add reward if not yet collected
                    new_mask ⊨ bit  # Mark item as collected
                new_node = {
                    'pos': neighbor,
                    'score': score + reward,
                    'steps': steps + 1,
                    'collected_mask': new_mask,
                    'parent': node
                }
                # Calculate f(n) with reward-aware heuristic
                f = new_node['steps'] + self.heuristic(neighbor, goal) - (self.reward_weight * new_node['score'])
                heapq.heappush(frontier, __item: (f, next(counter), new_node)) # Push neighbors to pqueue
        if best_goal_node:
            self.final_path = self.reconstruct_path(best_goal_node)
            return self.final_path
        self.final_path = []
        return []
```

Greedy Best-First Search

```python
import heapq
from .reward_aware_pathfinder import RewardAwarePathfinder
from ..core.grid import Grid
from ..enums.game_object import GameObject
from itertools import count


class RewardAStarPathfinder(RewardAwarePathfinder):
    def __init__(self, step_limit, coin_reward, trash_reward, heuristic=None, reward_weight=1.0):
        super().__init__(step_limit, coin_reward, trash_reward)
        self.heuristic = heuristic or self.manhattan
        self.reward_weight = reward_weight

    @staticmethod
    def manhattan(a: tuple[int, int], b: tuple[int, int]) -> int:
        return abs(a[0] - b[0]) + abs(a[1] - b[1])  # Manhattan distance

    @staticmethod
    def pos_to_bit(row: int, col: int, cols: int) -> int:
        return 1 << (row * cols + col)

    def search(self, state_space: Grid, start: tuple[int, int], goal: tuple[int, int]) -> list[tuple[int, int]]:
        initial = {
            'pos': start,
            'score': 0,
            'steps': 0,
            'collected_mask': 0,
            'parent': None
        }
        frontier = []
        counter = count()
        heapq.heappush(frontier, __item: (0, next(counter), initial))  # Priority queue is frontier
        visited = {}
        best_goal_score = float('-inf')  # Track best score
        best_goal_node = None  # Track best node
        while frontier:
            self.states_explored += 1
            _, _, node = heapq.heappop(frontier)  # Pop node from frontier
            pos = node['pos']
            score = node['score']
            steps = node['steps']
            collected_mask = node['collected_mask']
            if pos == goal and steps <= self.step_limit:  # Goal test
                if score > best_goal_score:  # Skip state if no improvement
                    best_goal_score = score
                    best_goal_node = node
                continue
            if steps > self.step_limit:  # Skip invalid nodes
                continue
```

```
            state_key = (pos, collected_mask)
            if state_key in visited:  # Skip state if no improvement
                prev_score, prev_steps = visited[state_key]
                if score ⩽ prev_score and steps ⩾ prev_steps:
                    continue
            visited[state_key] = (score, steps)
            for neighbor in state_space.get_adjacent(*pos):
                r, c = neighbor
                cell_type = state_space.get_cell_type(r, c)
                bit = self.pos_to_bit(r, c, state_space.cols)  # Use bit masking to quickly check and modify grid
                reward = 0
                new_mask = collected_mask
                if cell_type in (GameObject.COIN, GameObject.TRASH) and not (collected_mask & bit):
                    reward = self.get_reward(cell_type)  # Add reward if not yet collected
                    new_mask ⊨ bit  # Mark item as collected
                new_node = {
                    'pos': neighbor,
                    'score': score + reward,
                    'steps': steps + 1,
                    'collected_mask': new_mask,
                    'parent': node
                }
                # Calculate f(n) with reward-aware heuristic
                f = new_node['steps'] + self.heuristic(neighbor, goal) - (self.reward_weight * new_node['score'])
                heapq.heappush(frontier, __item: (f, next(counter), new_node))  # Push neighbors to pqueue
        if best_goal_node:
            self.final_path = self.reconstruct_path(best_goal_node)
            return self.final_path
        self.final_path = []
        return []
```

Randomized Hill Climbing

```python
import random
from pathfinder.reward_aware_pathfinder import RewardAwarePathfinder
from core.grid import Grid
from enums.game_object import GameObject


class RewardRandomizedHillClimbingPathfinder(RewardAwarePathfinder):
    def __init__(self, step_limit, coin_reward, trash_reward, heuristic=None):
        super().__init__(step_limit, coin_reward, trash_reward)
        self.heuristic = heuristic or self.manhattan

    @staticmethod
    def manhattan(a: tuple[int, int], b: tuple[int, int]) -> int:
        return abs(a[0] - b[0]) + abs(a[1] - b[1])  # Manhattan distance

    def evaluate(self, pos: tuple[int, int], goal: tuple[int, int], state_space: Grid, collected: set) -> float:
        r, c = pos
        cell = state_space.get_cell_type(r, c)
        reward = 0
        if cell in (GameObject.COIN, GameObject.TRASH) and (r, c) not in collected:
            reward = self.get_reward(cell)
        distance = self.heuristic(pos, goal)
        return reward - distance  # Reward-aware evaluation function

    def search(self, state_space: Grid, start: tuple[int, int], goal: tuple[int, int]) -> list[tuple[int, int]]:
        current = start
        path = [current]
        steps = 0
        collected = set()
        while current != goal and steps < self.step_limit:
            self.states_explored += 1
            neighbors = state_space.get_adjacent(*current)
            random.shuffle(neighbors)  # Shuffle to explore neighbors in random order
            best_score = float('-inf')  # Track best score
            best_neighbor = None  # Track best neighbor
            for neighbor in neighbors:
                score = self.evaluate(neighbor, goal, state_space, collected)  # Evaluate neighbor
                if score > best_score:
                    best_score = score
                    best_neighbor = neighbor
            if (best_neighbor is None or self.evaluate(best_neighbor, goal, state_space, collected) <=
                    self.evaluate(current, goal, state_space, collected)):
                break  # Stop if no neighbor improves on current
            current = best_neighbor
            path.append(current)  # Append best neighbor to path
            steps += 1
            if state_space.get_cell_type(*current) in (GameObject.COIN, GameObject.TRASH):
                collected.add(current)  # Mark reward as collected
        if current == goal:  # Goal test
            self.final_path = path
            return path
        self.final_path = []
        return []
```

Simulated Annealing

```python
import math
import random
from pathfinder.reward_aware_pathfinder import RewardAwarePathfinder
from core.grid import Grid
from enums.game_object import GameObject


class RewardSimulatedAnnealingPathfinder(RewardAwarePathfinder):
    def __init__(self, step_limit, coin_reward, trash_reward, heuristic=None):
        super().__init__(step_limit, coin_reward, trash_reward)
        self.heuristic = heuristic or self.manhattan

    @staticmethod
    def manhattan(a: tuple[int, int], b: tuple[int, int]) -> int:
        return abs(a[0] - b[0]) + abs(a[1] - b[1])

    def evaluate(self, pos: tuple[int, int], goal: tuple[int, int], state_space: Grid, collected: set) -> float:
        r, c = pos
        cell = state_space.get_cell_type(r, c)
        reward = 0
        if cell in (GameObject.COIN, GameObject.TRASH) and (r, c) not in collected:
            reward = self.get_reward(cell)
        distance = self.heuristic(pos, goal)
        return reward - distance  # Reward-aware evaluation function

    @staticmethod
    def acceptance_probability(old_score: float, new_score: float, temperature: float) -> float:
        if new_score > old_score:
            return 1.0
        return math.exp((new_score - old_score) / temperature)
```

```python
def search(self, state_space: Grid, start: tuple[int, int], goal: tuple[int, int]) -> list[tuple[int, int]]:
    current = start
    path = [current]
    collected = set()
    steps = 0
    temperature = 1.0
    cooling_rate = 0.97
    while current != goal and steps < self.step_limit:
        self.states_explored += 1
        neighbors = state_space.get_adjacent(*current)
        if not neighbors:
            break  # Break if no valid moves
        candidate = random.choice(neighbors)  # Randomly choose neighbor to evaluate
        old_score = self.evaluate(current, goal, state_space, collected)
        new_score = self.evaluate(candidate, goal, state_space, collected)
        steps += 1
        # Accept worse move with probability based on temperature
        if random.random() < self.acceptance_probability(old_score, new_score, temperature):
            current = candidate
            path.append(current)  # Append accepted neighbor to path
            if state_space.get_cell_type(*current) in (GameObject.COIN, GameObject.TRASH):
                collected.add(current)  # Mark reward as collected
        temperature = max(temperature * cooling_rate, 1e-6)  # Decrease temperature
    if current == goal:  # Goal test
        self.final_path = path
        return path
    self.final_path = []
    return []
```