A*pple Quest

Artificial Intelligence Course Project
Authors: Alessandro Querci, Andrea Lepori
Academic Year: 2024/2025

Source code and data available at: GitHub Repository

Introduction

This notebook constitutes the report for the final project of the AIF course, project's name is "A*pple Quest" and our group is composed by Alessandro Querci and Andrea Lepori. The goal of this project is to explore classical Artificial Intelligence **search and planning algorithms**, among the ones seen during the course, within the MiniHack environment, a reinforcement learning platform built on top of NetHack. Our work started by trying to identify a task suitable to apply the methodologies and algorithms seen in class (GOFAI, no learning).

In particular, we decided to focus on the use of state space search and planning algorithms. The high-level idea is to design some custom rooms/environments containing apples (with associated reward) and a custom reward manager. The reward manager assigns a positive reward for each collected apple and a (typically larger) reward when the agent reaches the downstairs by successfully completing the task. Additionally, to incentivize the agent to be as efficient as possible in terms of the number of steps, a (small) constant penalty is assigned for each step the agent moves through. Then, to evaluate the different search algorithms, they are used to generate a path to complete the task (reaching the downstairs) while maximizing the reward.

We decided to test the algorithms both in an *offline setting* with *full observability* and in an **online setting** with **partial observability**. The environment is deterministic, discrete, static and single agent. As mentioned above, if we are in the offline setting the observability is full as the agent knows the full map in advance, in the online setting the observability is partial as it's limited by darkness and walls.

To test our algorithms we defined different rooms/environments/mazes with different complexity. We started with a simple rectangular room, named "simple room", with some random apples and no obstacles, to start testing and debugging the algorithms, then we made a custom room with obstacles like lava in the so-called "lava room" that allow more difficult tasks while preserving the full observability.

Then we limit the vision of the agent by removing the premapped flag and using full obstacles like walls, by using the two maze maps provided: the "simple_maze.des" and in the more difficult "complex_maze.des".

To compare and evaluate each algorithm the more natural metric is the **total reward** collected by the agent accomplishing the task and computing automatically by the Reward Manager, however we decide to evaluate the algorithms across other metrics:

- **success rate** (task-completion/reaching the downstairs)
- **planning time** (time to elaborate a path/plan)
- **path length** (number of steps in the path)
- **apples collected** (numbers of collected apples during the path)

Once the algorithms were written and the environments defined, a model selection was carried out using grid search to optimize some of the hyperparameters. For all algorithms, given the same environment, several runs/simulations were executed with randomized apple positions, starting points, and downstairs, and the average of the metrics of interest was calculated in order to compare the algorithms across various dimensions and evaluate the trade-offs in using one over another.

Related works

To realize our project, we relied heavily on MiniHack.

> "MiniHack is a sandbox framework for easily designing rich and diverse environments for Reinforcement Learning (RL). Based on the game of NetHack, MiniHack uses the NetHack Learning Environment (NLE) to communicate with the game and to provide a convenient interface for customly created RL training and test environments of varying complexity."

> — https://github.com/samvelyan/minihack

In particular, we studied and heavily used the APIs provided by Minihack to create the map, customize our environment, customize the reward manager, manage the simulation, etc...

We took also inspiration, as a starting point, from the hands-on sessions of the course and from some projects of our colleagues that have applied search algorithms with different tasks and environments. In particular, we started with the concept of collecting most gold possible before reaching the stairs. Some choose to add opponents, like the leprechauns, studying scenarios with full observability and non-deterministic, but we decided to focus on an exploration task with restricted observability, where the agent has to explore the deterministic environment to find the apples and the stairs.

Methodologies

We used Python as programming language to work with Minihack's library, the README lists the project structure of the .py files, containing all the algorithms implementation and the code used to simulate and benchmark them.

To test our implementations, compare algorithms we use Jupyter notebooks. In the rest of this section, I will present briefly the two distinct approaches we follow, as well as the algorithms, in the **offline** setting and in the **online setting**.

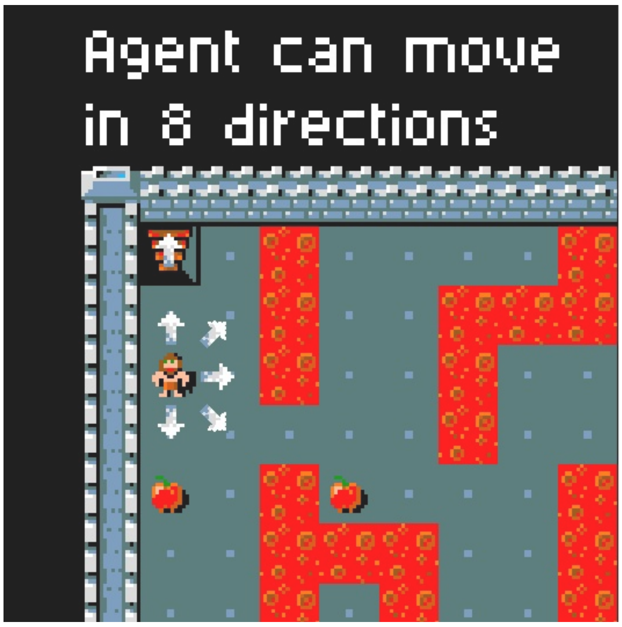
First, a shared operation for the two settings is the **env generation**. Precisely, we leverage level generator package in MiniHack to customize the environment, first by generating the .des file of a map with a function create_map, specifying with a specific string the map layout and possible presence of lava or walls.

As arguments of this method we can specify the starting position where the agent is spawned. While the number of apples is an argument, their position is decided either manually generating random positions with a seed or letting the built-in randomization place them. This distinction was made because the internal random will generate a new random sequence at each env.reset() even with a seed provided, and we needed to ensure repeatability during benchmarks.

As the .des is generated, the function create_env handles the creation of the gym env with the specified map and reward settings. The user can specify the reward to assign to apple collection as well as the penalty time for each step taken. In this function we define the set of actions that the agent is allowed to perform and the observation keys we want to track, we define the reward manager to register the specified rewards and penalty. Then using gym.make() we effectively create the env.

The rules given are:

- The agent can move in the four cardinal directions (up, down, left, right) and the four diagonal directions (up-left, up-right, down-left, down-right).
- The agent can't move through walls or lava, and it can't move diagonally if both adjacent cells are walls or lava.
- The agent can pick up an apple by moving onto it, and then eat it by performing the action "eat" and getting a reward.
- The agent need to move onto the stairs to end the task and receive a reward.
- The agent receives a penalty for each step it takes, to incentivize it to find the shortest path.



Pathfinding Algorithms

At the core of the offline setting task there are the pathfinding algorithms that are called to determine the path the agent has to execute. The implementations of all the used algorithms can be found in algorithms.py regarding search algorithms in state spaces, while the implementation of Monte Carlo Tree Search can be found in the MCTS.py file. Now a brief description of the implemented algorithms, for more details please refer to the code:

A* Star based Algorithm

As the title implies, we gave a bit more room to testing pathfinding algorithms based on the A* Star algorithm, which is a well-known and widely used pathfinding algorithm in AI. The A* algorithm is a best-first search algorithm that uses a heuristic to estimate the cost of reaching the goal from a given node, allowing it to efficiently find the shortest path in a weighted graph.

a_star_apple: This function implements a modified A* pathfind algorithm to find an optimal path from a start position to a target, in addition to the standard A* components (g-scores for actual cost, heuristic estimates, and a priority queue), the algorithm introduces a reward system for paths that pass near or through apples. Apples directly reduce the movement cost (g-score), with a stronger bonus for stepping onto one and a smaller bonus for being adjacent. This encourages the pathfinder to favor apple-rich routes without compromising its goal-directed behavior. A weight parameter allows tuning the influence of the heuristic, effectively enabling weighted A* behavior.

a_star_collect_apples: This function implements an A* pathfinding variant that aims to collect all apples on the grid before reaching the target. It uses a Minimum Spanning Tree (MST)-based heuristic to estimate the minimal cost required to visit all remaining apples and the target, combining this with the actual path cost (g) for A'*s total cost function (f = g + h). It prioritizes paths that visit uncollected apples efficiently, balancing exploration and optimization using a tunable weight parameter on the heuristic (weighted A*).

beam_search_apple: This algorithm applies Beam Search to find a path from a start point to a target while collecting apples for additional reward. At each step, it keeps only the top k candidate paths (defined by the beam width) that maximize a net score: total apple reward minus path cost. All pairwise paths between the start, apples, and target are precomputed using a classic A* for efficiency. The search explores new candidates by extending paths toward unvisited apples or the target, using the precomputed paths, avoiding cycles.

Potential Fields

potential_field_path: This simple algorithm, born in the field of robotics, uses a potential field approach to guide pathfinding toward a target while encouraging apple collection along the way. It assigns each grid position a scalar "potential" based on its attractive force toward the target and any remaining apples, using either the sum or maximum of these forces (modality_potential). Random noise is added to break ties and avoid local minima, and repeated visits to the same cell are penalized (repulsive attraction) to discourage loops. The agent greedily chooses the next move that maximizes total potential.

Monte Carlo Tree Search

mcts : This algorithm applies Monte Carlo Tree Search (MCTS) to navigate the state space. Each node in the search tree represents a game state defined by the agent’s position and the set of collected apples. At each iteration, the algorithm follows the standard four MCTS phases:

- **Selection**: Nodes are traversed using the UCB1 formula to balance exploration and exploitation.
- **Expansion**: New child nodes are added by simulating valid moves from the current state.
- **Simulation** (Rollout): A random policy simulates play from the new state to a terminal condition or step limit, assigning a reward based on apples collected and whether the target is reached.
- **Backpropagation**: The simulation result is propagated up the tree to update visit counts and cumulative rewards. The best path is then extracted by following the most visited child nodes from the root.

Greedy Best-First Search

greedy_best_first_search: This algorithm uses Greedy Best-First Search to find a path from a start point to a target while collecting all apples on a grid. It prioritizes nodes based solely on a heuristic estimate of the cost to reach the goal, ignoring the actual path cost. The heuristic evaluates the shortest possible path through all remaining apples and then to the target, using either Manhattan or cached BFS distances.

Simulator Offline

The simulator for the offline setting is implemented by the function **simulate_offline_planning**, that takes in input the environment to simulate and the pathfinding function to use for simulation (to compute the path). In practice, the simulator retrieves from the game map the position of the player, the target, and the apples in the environment, and makes a call to the pathfinding algorithm that returns the path found.

The path, expressed as a sequence of coordinates, is translated into a sequence of actions that the agent must perform. Subsequently, the path is executed with closed eyes in perfect offline mode. For each action in the list of actions, the function env.step is called. If you land on an apple, a series of actions is performed in order to eat it, and then you continue on your path. During the sequence of actions, the overall reward is accumulated by collecting the various instantaneous rewards.

The function finally returns the total reward, the path length, the time needed for planning, the apples eaten and whether the main task was successfully completed (the stairs down are reached).

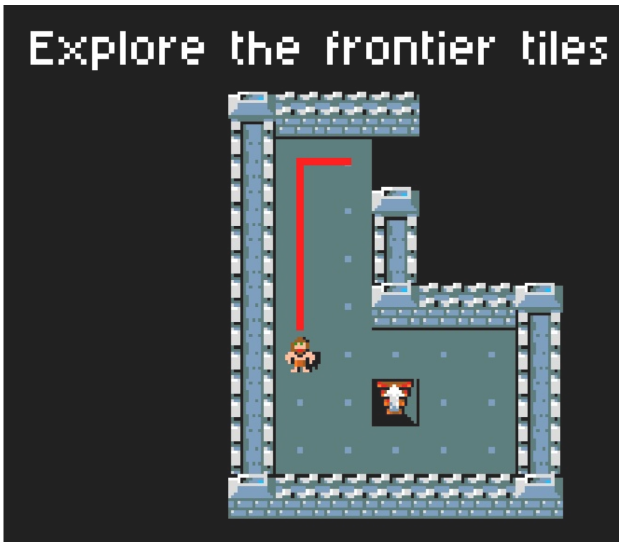
Simulator Online

The online version, for simulation in a partially observable environment, is implemented by the function simulate_online.

The main difference with the offline simulator is that the agent does not have access to the full map, but only to a limited radius around its position. The agent can only see the cells that are within this radius, and it has to explore the environment to discover new cells. For this reason, the agent has to continuously recalculate the path to the target (the nearest apple or the stairs) as it moves through the environment.

Once the target position has been decided the agent behaves similarly to the offline setting, using the same pathfinding algorithms. The path is then executed by the agent, which moves through the environment until it reaches the target or discovers a new apple. If the agent discovers a new apple during the execution of the current plan, it stops and recalculates a new plan, performing a new iteration of the loop. If the agent exhausts the actions of the current plan, a new iteration of the loop leads to the computation and identification of a new path.

The rest of the logic is similar to the offline simulator, with the agent collecting apples and accumulating rewards. The function returns the total reward, the path length, the planning time, the apples eaten and whether the main task was successfully completed (the stairs down are reached).



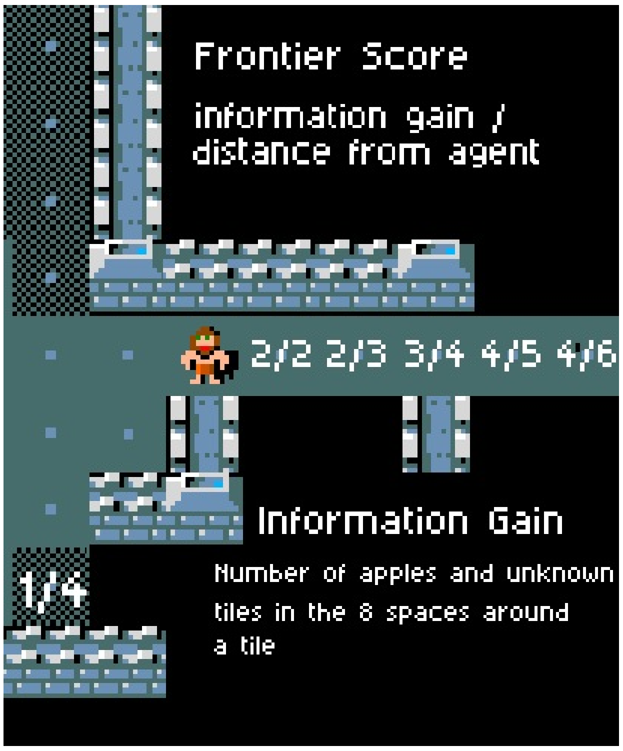
Frontier Search and Target Selection

To make the agent able to explore the environment, a Frontier Search algorithm is implemented, which looks for the frontier tiles, i.e. the tiles that are currently known and accessible but border unknown (unexplored) spaces.

To decide which tile to explore, the agent uses a scoring function that takes into account both the distance from the current position and the number of unknown tiles in its moore neighbour (information gain). This way, the agent is able to explore as much map as possible to discover new apples while trying to find the stairs. The scoring function is defined as follows:

```
frontier_score = information_gain / bfs_distance_to_agent
```

To not make it focus entirely on exploration of the frontier, the agent also compute the frontier score of the apple tiles with an information gain of 1 so the agent can decide whether to collect the apple if it's close enough before going to explore a far frontier tile. The agent will then choose the tile with the highest score as the target to explore next.


Frontier Score
information gain / distance from agent
2/2 2/3 3/4 4/5 4/6
Information Gain
Number of apples and unknown tiles in the 8 spaces around a tile
1/4

Assessment

Two distinct benchmarks were performed, one for the offline setting and another for the online setting, respectively in the notebooks Benchmark_Offline.ipynb and BenchMark_Online.ipynb. The metrics used for the benchmarking are:

- **total reward**: at the end the better agent is the one which maximizes the custom reward.
- **success rate**: the agent is able to accomplish the task (i.e. reaching downstairs)?
- **apples collected**: how many apples the agent eats? How biased is the algorithm toward collecting apples?
- **path length**: How long is the path in terms of number of steps.
- **planning time**: How long does the pathfinding algorithm take to calculate the path? The less, the better.

For the offline setting the algorithms performances were compared on the Lava Maze with penalty_time of -0.1, for the online setting we use the simple maze and the complex maze without the premapped flag enabled and with a lower penalty on steps needed (penalty_time=-0.01). The reward for collecting an apple is fixed to 0.75 and the reward for reaching the stairs is fixed to 1.0.

The algorithms are tested with different hyperparameters, in particular for the A* Star algorithm we vary the weight of the heuristic and the apple bonus, for MCTS we vary the exploration constant C, for Greedy BFS we vary the modality of the heuristic (Manhattan or cached BFS), for Potential Fields we vary the modality of the potential field (sum or max) and for Beam Search we vary the beam width and heuristic weight.

To run the tests the function benchmark_simulation was called for all the algorithms. This function benchmarks a pathfinding algorithm under different random seeds (for environment randomization ensuring repeatability across experiments) and parameter settings (to allow a minimal model selection and hyperparameter tuning). For all algorithms, the results across all the metrics, are averaged across all the different runs with the different seeds.

Results

In the following tables you can find the best averaged results in terms of reward for all the algorithms tested in the Offline setting on the lava maze, and in the online setting both in the simple and complex maze. The other metrics of interest are also shown.

To see also the effect of the different hyperparameters and the differences in performances for different hyperparams config. please refer to the related notebooks.

Offline benchmark on "lava env", best averaged result in terms of Reward for each algorithm

Algorithm	Avg Reward	Avg Path Length	Avg Apples	Avg Success Rate	Avg Planning Time
A* Star	1.165	34.7	4.9	1.0	0.004
MCTS	0.460	24.6	2.8	1.0	1.35
Greedy BFS	0.85	35.0	5.0	1.0	0.03
Pot. Fields	0.745	34.1	4.7	1.0	0.04
Beam Search	0.055	29.3	2.9	1.0	0.09

Online benchmark on "simple maze", best averaged result in terms of Reward for each algorithmn

Algorithm	Avg Reward	Avg Path Length	Avg Apples	Avg Success Rate	Avg Planning Time
A* Star	3.715	86.8	4.4	1.0	0.06
MCTS	3.687	75.6	4.3	1.0	0.77
Greedy BFS	4.059	81.4	4.8	1.0	0.06
Pot. Fields	3.686	120.4	4.8	1.0	0.06
Beam Search	3.557	74.3	4.1	1.0	0.14

Online benchmark on "complex maze", best averaged result in terms of Reward for each algorithmn

Algorithm	Avg Reward	Avg Path Length	Avg Apples	Avg Success Rate	Avg Planning Time
A* Star	3.005	128.9	4.0	1.0	0.18
MCTS	2.447	126.2	3.2	1.0	2.17
Greedy BFS	3.417	133.6	4.6	1.0	0.16
Pot. Fields	2.732	172.3	4.4	1.0	0.15
Beam Search	2.205	128.9	2.9	1.0	0.85

Conclusion

The experimental results highlight the trade-offs and performance characteristics of different path planning algorithms across both offline and online settings in maze environments of varying complexity.

In the offline lava maze environment, A* Star demonstrates the highest average reward and apple collection with negligible planning time, confirming its efficiency in static, fully observable environments. MCTS, while returning the shortest path, shows lower average rewards and fewer apples collected (it seems to favor direct paths towards the target), with significantly higher planning times, indicating that its stochastic search strategy trades off computational cost for exploration. Greedy BFS and Potential Fields, despite being very simple algorithms, offer competitive performance with moderate planning times, whereas Beam Search, falls short in reward and apple collection, while the length of the path is close to optimal.

Moving to the online setting in the simple maze, all algorithms maintain a perfect success rate, with Greedy BFS slightly outperforming others in average reward and apple collection. MCTS maintains competitive rewards but at a substantial increase in planning time compared to greedy methods, reflecting the cost of its sampling-based search. Notably, Beam Search lags behind in rewards and apple counts but offers the shortest path to the goal. Potential Fields exhibit longer paths, which may indicate less direct routes but sufficient exploration as it collects on average near all the apples.

In the more challenging complex maze, the distinctions become more pronounced. Greedy BFS leads in average reward and apple collection while balancing path length and planning time, suggesting its heuristic guidance scales well with complexity. A* Star remains efficient, preferring a shorter path instead of collecting one more apple. MCTS incurs the longest planning times, emphasizing its computational expense in complex environments, finding the shortest path but collecting less apples. Potential Fields incur in the worst path lengths but in terms of planning speed and apple collection it matches, but still not outperforms, greedy methods. Beam Search experiences further degradation in reward and apple collection, indicating difficulty navigating increased maze complexity under limited beam width constraints.

Overall, the results confirm that A Star and Greedy BFS offer strong, efficient performance across environments, with Greedy BFS showing greater adaptability in online and complex scenarios. MCTS comes with significant computational overhead that may limit real-time applicability. Beam Search and Potential Fields present interesting trade-offs, they are very simple algorithms, but generally underperform A* and Greedy BFS.

These findings suggest that combining heuristic guidance with efficient search strategies (as in Greedy BFS) provides a practical balance of reward maximization, computational efficiency, and adaptability, especially in online or dynamic environments.

Appendix

Cached BFS heuristic

The cached_bfs heuristic enhances pathfinding in grid environments with obstacles by computing the shortest path length between two points using a breadth-first search (BFS) that respects walls and barriers. To improve efficiency in some scenarios where the same pair of points might be evaluated multiple times, it caches previously computed distances to avoid redundant searches. This heuristic provides an accurate distance estimate that accounts for impassable terrain, making it well-suited for environments where direct straight-line metrics (like Manhattan distance) are insufficient due to obstacles.

Project Structure

- `Benchmark_Offline.ipynb` — Notebook for offline benchmark experiments
- `Benchmark_Online.ipynb` — Notebook for online benchmark experiments
- `MCTS.py` — Implementation of Monte Carlo Tree Search
- `algorithms.py` — Search & Planning algorithms
- `algorithms_online.py` — Online search algorithms
- `complex_maze.des` — Complex maze environment description file
- `simple_maze.des` — Simple maze environment
- `simulator.py` — Implementation of the various simulators
- `utils.py` — Utility functions
- `Report.ipynb` — Project Report (only Text/Markdown)

Team contributions

The project work was carried out by the two members of the group, in particular:

- **Alessandro Querci**: implemented the logic of the simulators, a_star_apple, beam search, online algorithms, frontier search and benchmarking
- **Andrea Lepori**: implemented a_star_collect_apples, MCTS, potential fields, Greedy Best First.

The work on the project involved some meetings and pair sessions for programming and writing the report,as well as individual work using GitHub as a collaboration platform.

Relationship with the course

The work of this project has strong relations with the topics of the AIF course, in particular with the section on state space search algorithms. Particular attention was paid to stick to the typology of classical AI algorithms studied in the course (no machine learning). However, we tried to experiment with algorithms not seen during the course such as Potential fields or lightly touched upon such as A* online or with heuristics specifically designed for our task such as bfs_cache. Furthermore, our work has a strong relationship with the practical lessons held during the hands-on sessions.