

Lab 1: Search

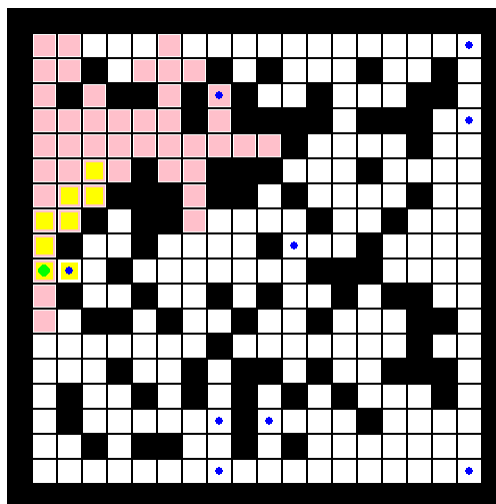
This lab's objective is to implement classic search algorithms to help a vacuum cleaner navigate a maze-like environment, and remove the dirt on various spots.

Deliverable Submit a zip file containing all of your code, along with a written report containing:

- For each file that you modified, a quick description of what you wrote
- Quick experimental results comparing the different search algorithms, as well as an analysis of the results
- Answers to the theoretical questions

1 Problem Domain: Vacuum Cleaner Agent

The vacuum cleaner agent operates in a maze-like environment where it must navigate to collect dirt particles. The agent has perfect knowledge of the environment layout and dirt locations, and aims at collecting the dirt particles one after the others, successively going for the next closest dirt particle by euclidean distance. Your goal is to make the agent autonomous, through an implementation of search algorithms that help it find paths to each dirt particle.



The problem is divided into several subproblems, since the agent collects dirt particles successively. With the current implementation, the agent automatically cleans the dirt on a cell where there is some, so that you only have to find a path there.

1.1 Setup

The provided codebase contains the environment described in the previous section, as well as an implementation of a very simple search algorithm that randomly picks an action until the problem is solved. You can test it with the following steps:

1. Extract the provided code archive
2. In case you don't have the PyGame package installed, install dependencies: `pip3 install -r requirements.txt`
3. Run the lab: `python3 run_lab.py --search random`

You can use the `--help` option when running file `main.py` to get an overview of the options you can pass. More details are provided in appendix.

1.2 Code Structure Overview

The codebase provides a few classes that you need to understand in order to implement the search algorithms for the grid navigation problem we are effectively tackling. As a start, we provide below a small description of the main classes that you will work with.

SearchProblem (`problem.py`): The class represents the grid navigation problems that you will have to successively solve, where the goal is to find a path from an initial state to a designated goal state. You will need the `get_initial_state()`, `is_goal_state()`, and `get_successors()` methods.

For performance analysis, the class also tracks the expanded nodes.

SearchNode (`search_node.py`): The class represents a node of the search tree. A path is represented by a list of such objects. If the parent nodes have been tracked properly, you can use the `get_path_from_root` method to return the path used to get there.

BaseSearch (`base_search.py`): An abstract base class that defines the common interface for all search algorithms that you should implement.

The main search algorithms should be implemented in method `search(problem) -> List[SearchNode]`. Note that your algorithms should return a list of `SearchNode`.

Be careful to also properly implement the getters provided in that class, since they are used by the visualizer to display information about the search process itself. In particular, **be sure to return `SearchNode` objects when needed** (even if only the `GridPos` state has to be correct for visualization purpose). Have a look at the signatures of the functions if need be.

An example of the usage of the classes above can be found in the implementation of **RandomSearch** (`random_search.py`).

2 Coding task

To complete this lab session, you must implement the following two to three search algorithms by completing their respective files. Each algorithm inherits from **BaseSearch** and must implement the required methods.

You can use the following Python classes to implement the data structures you will need:

- `list` - Can be used as a stack with `append()` and `pop()`
- `collections.deque` - Can be used as a queue with `append()` and `popleft()`
- `heapq` - A module that provides functions `heapify()`, `heappush()` and `heappop()`.
 - Note that objects of a given type A inserted into this structure must be comparable, i.e. operation $<$ must be supported. Support can be added by implementing the method `__lt__(other)` for class A .

2.1 Uninformed Search

In uninformed search (also called blind search), the algorithm only considers basic information about the states that it encounters, and is based on comparisons between states. During this lab session, you will implement two blind search algorithms that are fairly similar in their logic.

BreadthFirstSearch (`breadth_first_search.py`): This algorithm explores successive layers, by exploring all nodes at depth d before exploring nodes at depth $d + 1$.

Task: Implement Breadth First Search

DepthFirstSearch (`depth_first_search.py`): This algorithm keeps exploring a successor of the last expanded state until no more are available (or the goal is reached), and then backtracks if needs be.

Task: Implement Depth First Search

2.2 Heuristic Search

In informed search, the search is directed by a heuristic function, that gives an estimate of how close some node in the frontier is to the nearest goal state. The algorithm then favors the most promising nodes.

AStarSearch (a_star_search.py): A* search explores the top node provided by a priority queue ordered by $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost and $h(n)$ is the heuristic estimate.

Tasks:

1. Design a (or multiple) heuristic function(s) for the grid navigation problem
 - Recall that a good heuristic for A* is admissible: it never overestimates the cost of an optimal path to the closest goal.
2. (*Optional*) If you have the time, implement A*
 - Since A* requires more information about the nodes of the tree than blind search algorithms, you should implement class **AStarNode** that inherits from **SearchNode**

3 Theory

Answer the following questions in your report.

1. What is the difference between Breadth-First Search and Uniform Cost Search in a domain where the cost of each action is 1?
2. Suppose that h_1 and h_2 are admissible heuristics (used in A*). Which of the following are also admissible? Justify your answers.
 - (a) $(h_1 + h_2)/2$
 - (b) $2h_1$
 - (c) $\max(h_1, h_2)$
3. For each of the following search algorithms, determine whether they are complete and optimal. Give a quick explanation why.
 - Breadth-First Search
 - Depth-First Search
 - Uniform Cost Search
 - Iterative Deepening Search
 - Bidirectional Search
 - Greedy Best-First Search
 - A* Search

Appendix

Available options for the GUI:

- `--search`: Choose algorithm (bfs, dfs, astar, random)
- `--maze`: Choose maze type (default, simple, office, caves)
- `--cell-size`: Adjust cell size for visualization
- `--dirt`: Number of cells with dirt
- `--seed`: The seed of the random numbers generators. Set it to a fixed value for debugging purpose
- `--no-gui`: Run the program without the PyGame-based visualizer. You can use this to run benchmark tests.

Note that the parameters of the different generation functions for the mazes can be tweaked, to have denser or sparser grids.