Assignment 1 - Search 1 and 2

Artificial Intelligence

WS 2018

Due: 2018-10-28, 23:59

Andreas Arzt, Verena Haunschmid, Rainer Kelz

General Information

Before you start, make sure you use the most current version of the framework (section "ASSIGNMENT - 1" in Moodle).

Empty classes and methods of the algorithms to be implemented are available in the package `at.jku.cp.ai.search.algorithms`. This should be familiar to you, due to how things worked in `assignment0`.

You can test your implementations on a set of unit tests available in the package `at.jku.cp.ai.tests.assignment1`. The unit tests compare the results of your implementations to those of our reference implementations on the problem instances in `assets/assignment1`. Note that passing all these tests does not necessarily mean that your implementations have no flaws, since the tests do not (can not) cover all possible invariants!

For implementation details of the framework, please refer to the handbook and/or source code comments where available. Remember your solution of `assignment0` when deciding which data structures to use.

1 Report for the theoretical part

For handing in your solutions to the theoretical part, we would like you to create a PDF file in the directory named 'reports'. Ideally you would use Latex to produce your reports, but exporting a PDF from any other program is fine too.

Where to put your reports:

```
<your-exercise-directory>
├── reports
│   └── report.pdf (you need to put your theory answers into this file)
├── assets
├── libs
└── src
```

IMPORTANT:

- Your report needs to **contain** your **names** and **immatriculation numbers!**
- **No photos** of handwritten reports!

2 Coding Guidelines

- You write code mainly for other **humans** to read.
- Make sure your code is **readable** and **understandable**.
- Add **comments** to explain what you are doing and why.
- When using non-primitive data types (List, Set, ...), **explain why you chose exactly this type over others!**

  (e.g. "We use `<DataStructure>` because we need a fast `<operation>` with runtime complexity `O(<something>)`, so that we don't increase the runtime complexity of `<other-operation>` that uses this datastructure.")

3 (Theory) - Purpose of Datastructures

Answer these questions about the purpose of certain datastructures:

- What is the main purpose of a priority queue?
- What is special about the priority queue provided with the framework?
  `at.jku.cp.ai.search.datastructures.StablePriorityQueue`
- What is special about the stack provided with the framework?
  `at.jku.cp.ai.search.datastructures.StackWithFastContains`
- Which data structure should we choose to implement a “closed list” as it is often called in the literature?

(2 points)

4 (Programming) Uninformed Search

Implement the following uninformed search algorithms within the provided framework. Take care to avoid repeatedly expanding visited nodes as far as it is possible without increasing the big-O space complexity of the algorithms.

Look at the sub-package `at.jku.cp.ai.search.datastructures`, and view the data-structures contained within. If you need a priority queue, please use `StablePriorityQueue`, in order to make sure your node expansion order is the same as ours. If you need a stack, and you need to check if something is already on the stack, please use `StackWithFastContains`.

Process successor states returned by the `adjacent()` method in the order they are returned (so, if you get `[x, y]` back from `adjacent()`, the algorithm should explore `x` first). Lastly, to keep track of costs for nodes, you will need additional data structures.

(A) Breadth-First Search
`src/at/jku/cp/ai/search/algorithms/BFS.java` (3 points)

(B) Uniform Cost Search
`src/at/jku/cp/ai/search/algorithms/UCS.java` (2 points)

(C) (Self-Avoiding) Depth-limited Depth-First Search
`src/at/jku/cp/ai/search/algorithms/DLDFS.java` (3 points)

(D) Iterative Deepening Search
`src/at/jku/cp/ai/search/algorithms/IDS.java` (1 points)

HINTS:

- Read the handbook, section 6, “Search in graphs”.
- Look at the extended DLDFS examples (slides in Moodle) to understand why self-avoiding DLDFS is a good idea.
- For implementing the `cost` and `heuristic` functions, the `java.util.Function<T, R>` interface is used. To execute the function, call `cost.apply(t)` (or `heuristic.apply(t)`) by passing a parameter of type `T`, an object of type `R` will be returned.

5 (Programming & Theory) Heuristic Search

Implement the following heuristic search algorithms. If you need a priority queue, please use `at.jku.cp.ai.search.datastructures.StablePriorityQueue`, in order to make sure your expansion order is the same as ours.

(A) Greedy Best-First Search
`src/at/jku/cp/ai/search/algorithms/GBFS.java` (4 points)

(B) A* Search
`src/at/jku/cp/ai/search/algorithms/ASTAR.java` (4 points)

Consider the two heuristics implemented in the framework and used to test the algorithms. Answer the following questions, and **explain your answers** in detail:

(C) Which of the heuristics guarantees that Greedy Best-First Search will lead to an optimal solution? Which of them guarantees obtaining an optimal solution using A* Search? (1 points)

(D) Considering only the game world, which of the heuristics is better? (1 points)

6 (Dry running) Create a Level

Look into the directory `assets/assignment1`. Each of its subdirectories contains a level that your implementations are tested against.

The `level` file contains a definition of the board, where '`#`' is a wall, '`.`' is a tile the player can move on, '`p`' is the player, and '`f`' is a fountain (the goal).

The `costs` file is of the same structure as the `level` file, but contains numbers for each reachable tile (everything except walls). These numbers correspond to the `cost` incurred by stepping on this tile.

The `*.path` files contain the paths returned by the reference implementations of the search algorithms. For A* and GBFS, there are two files corresponding to the two different heuristics used. E.g. `astar_mh.path` contains the path that the A* implementation using the Manhattan Distance heuristic should return. Note that both the starting point and the goal position are included!

(A) Create a new level for which each of the search algorithms {BFS, IDS, DLDFS, UCS} returns a **different path**. If this is fundamentally not possible for certain pairings of algorithms, explain **why**.
Create a L101 `level` file and corresponding `costs` file.
Create `*.path` files for every algorithm listed here. (4 points)

(B) Describe **how** all of the search algorithms in this list {BFS, IDS, DLDFS, UCS, GBFS, ASTAR} compute their solution **for the level you made in (A)**. You can do this e.g. by drawing the **search space** as a tree, clearly **indicating the expansion order** of the nodes and including all necessary information like accumulated costs, etc... (6 points)

HINT: try and create the smallest board possible! both (A) and (B) are possible with very small boards ...

Include the `level` and the `costs` file in your submission, in a directory called "`assignment1/L101`". Any drawings and textual description should go into your PDF report.