

CMPE540 - Project 1

Announced: 30th of September 2016

Due: 16th of October, 23:59

Submission Instructions:

- Please follow these instructions, otherwise you will lose half of the points.
- The project will be automatically evaluated. Still you need to prepare a very short report. See the end of this document for more information about what I expect in the report.
- Name your report as student-id.pdf. e.g.:
 - 2014301162.pdf
- Copy this report into p1/ directory.
- Create an archive. Name it student-id.tgz with the following command.

```
$ tar -czvf 2014301162.tgz p1/
```

- Attach the archive file to an email and send it to emre.ugur@boun.edu.tr with the following subject line:
 - [CMPE540 Project 1]

Late submissions:

- The policy is as it was defined before: – (10 x days) up-to 3 days
- In case there is a serious mistake in the project description or significant change, this date will be postponed.

Discussion and cheating:

- I don't like this section, but still should write about this..
- You are welcome to discuss the topics about the project. But please do not cheat. There are programs that can automatically check the similarity of the source code and/or execution. Read the department policy about what is considered as cheating. If I am convinced that you cheated, then you will get F.

Questions about the project:

- The details will be explained in the lecture on 1st of October.
- Please warn me as soon as you see a mistake through email or mailing-list.
- In case you have questions, please send them to the mailing-list so that
 - Everybody can see the answer to that question
 - People other than me can reply the questions as well

Acknowledgement:

- This project is adapted from the following webpage:
<https://inst.eecs.berkeley.edu/~cs188/fa11/projects/search/search.html>
- The same layouts are used for test
- The same questions are used except:
 - Cost is defined in a different way in our project.
 - Sub-optimal search is omitted in our project.

In this project, you will

- ## Introduction to the problem:

- [illegible]

- ```
Agent agent (searchType);
agent.setProblemType (problemType);
agent.perceive (layoutFileName, weatherType);
agent.search ();
agent.printSolution (msec);
```

- ## Compilation:

- ```

@ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
@ F                               F F @      @
@ F @ @ F @ @ F @ @ F @ @ F @ @ @ @ @ @
@                               A @ @ @
@ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
@ F F F F F @
@ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @

```

- If the terminal does not support coloring, compile with
 - `make NO_COLOR=1`

Execution:

- Execution command line arguments are provided if you make a mistake in use
 - Simply type “./p1 -help” which is a mistake :). You will get the following

```
Usage:
./p1 -p eat-food/visit-corners/eat-all-foods -l <layout-file> -w <weather-condition> -s
RAND/BFS/DFS/UCS/A_STAR -u <msec>

where <weather-condition> might be:
    calm/wind-from-west/wind-from-east/wind-from-north/wind-from-south/
    /lightning-at-west/lightning-at-east/lightning-at-north/lightning-at-south/
-p for problem
-l for layout file
-w for weather conditions
-s for search algorithm
-u for update speed in printing the solution path (no printing if -1)
[Error] in Wrong command line arguments
```

- You will have 3 different **problem** types:
 - eat-food (Q1-Q3): eat any one food in the environment.
 - visit-corners (Q4-Q5): visit all four different corners (do not eat the foods on the way).
 - eat-all-foods (Q6): eat all the foods in the environment.
- We can set any of the **layout** files for any problem. I will use different layout files for evaluation.
- **Weather** conditions define the costs of states and actions
 - **calm** weather: all actions cost 1.
 - **wind**: the robot action against the wind has high cost, the action along with the wind has low cost. Otherwise, medium cost.
 - **lightning**: as agent approaches to the lightning areas, the cost increases.
- **Search** algorithms:
 - **RAND** random search is implemented for you (Agent::randomSearch()). You can run it from the beginning for any layout, weather condition and problem type. If you are lucky, this will find a path:
 - ./p1 -p eat-food -l layouts/tinyMaze.lay -w calm -s RAND -u 100
 - Other algorithms should be implemented by you.
- You (or I) might want to see the path found by the search algorithm. Then set **screen update speed** (-u) to the desired milliseconds. If “-u -1”, then no solution output.
- This program, after execution (if your search completes!), prints out:
 1. the number of nodes expanded during searches
 2. the path cost
 - I will generally evaluate the projects based on these numbers (# of expanded nodes & path cost)
- Do not modify p1.cpp

Some data structures:

Agent:

- As you remember, agent perceives, searches for solution and acts. In our case, it will perceive, search for solution and print-out that solution. See main() in p1.cpp
- Agent.h includes the basic structures (e.g. Node) and functions that an agent uses in our project.
- Do not modify Agent.h.
- You will need to modify some parts in Agent.cpp in implementing search algorithms (see the details later). The functions that require no modification:
 - Agent::Agent()
 - Agent::setProblemType()
 - Agent::percept()
 - Agent::search()
 - Agent::randomSearch()
 - Agent::printSolution()
 - Agent::printPath()
- The functions that should be implemented:
 - Agent::add2Fringe () → INSERT function in the pseudo-code
 - Agent::removeFromFringe () → REMOVE-FRONT function in pseudo-code
 - Agent::graphSearch()
 - Agent::isInClosedList() // check if the state was visited before.
 - Agent::add2ClosedList() // adds nodes to the visited lists
 - Agent::initClosedList() // initializes the list of visited nodes

Node:

- Search tree node is implemented with Node class in Agent.cpp & Agent.h
 - Node::Node() should be kept intact
 - Node::expand() should be extended. Do not remove the lines that are commented. Those lines control if the heuristic is consistent.

Problem:

- Problem description is included in the classes inherited from the Problem class. See Problem.h for details. This is an abstract class that only defines the functions that should be implemented by the extended problems such as MazeProblem, CornersProblem, AllFoodsProblem.

Implementation:

- You are generally expected to implement the code within the comments in the source or header files, e.g. from Agent.cpp:
- These spaces are designed to make your job easier.
- The pointers for each space are detailed below.
- Feel free to use your own structures/functions unless explicitly said no.

```
// Implement for Q1-Q3 and use in the rest
Node *Agent::removeFromFringe (){
    Node *node = 0;
    if (searchType == BFS){
        /***** FILL-IN FROM HERE *****/
        /***** FILL-IN UNTIL HERE *****/
    }else if (searchType == DFS){
        /***** FILL-IN FROM HERE *****/
        /***** FILL-IN UNTIL HERE *****/
    }else if (searchType == UCS){
        /***** FILL-IN FROM HERE *****/
        /***** FILL-IN UNTIL HERE *****/
    }else if (searchType == A_STAR){
        /***** FILL-IN FROM HERE *****/
        /***** FILL-IN UNTIL HERE *****/
    }
    return node;
}
```

Questions 1-3: Eat any food in the environment (-p eat-food)

With the search functions written in this question, our agent should be able to eat one (any) of the foods available in the environment. If there is one food, it finds the path to that food. If there are more than one food, agent is free to find a path to any of those foods. This requires defining the problem and state representation:

- **State**
 - There is one food and one agent. Therefore the state includes only the position of the agent. The state is represented in **MazeState** class which is implemented for you. See MazeState.h and MazeState.cpp.
 - MazeState::isSameState() is also implemented.
- All relevant information about the problem is included in **MazeProblem** class:
 - The location of foods, location of walls, initial location of the agent, a 2d map with items, number of foods, number of wall, weather condition.
- A lot of useful functions for the problem are also included in the **MazeProblem** class. Some utility functions are:
 - **void int getPosFromRowCol (int r, int c):** function that finds the index of 2d (row,col) position:
 - **getRowColFromPos (int &r, int &c, int index):** function that finds 2d (row,col) position from index:
 - **void getActionEffectRC (int &dr, int &dc, int action):** function that returns the change in 2d position with an action
- Some implemented search related functions are:
 - void MazeProblem::getSuccessorStateActionPairs (vector<State *> &states, vector<int> &actions, State *curState)
 - See how this function is used on Agent::randomSearch()
 - Use this function while implementing your expand() function (see week2-slides, pg.15)
 - double MazeProblem::getStateActionCost (State *curState, int action)
 - Depending on the weather, current state and action, this function returns the cost. Do not change.
 - bool MazeProblem::isGoalState (State *curState)

Q1. BFS and DFS– 2 points.

In this question, you will implement BFS and DFS search functions so that the agent can reach to the goal. You might want to implement/extend:

- Agent::graphSearch()
- Agent::add2Fringe()
- Agent::removeFromFringe() “REMOVE-FRONT (fringe)” (pg. 15, week2).
- Agent::isInClosedList ()
- Node::expand ()
 - While expanding, do not change the order of state-action pairs that come from getSuccessorStateActionPairs() function.
 - If you change the order (i.e. order of actions), then DFS will explore different ways. To be consistent (and to get the grade depending on the number of expanded node), use that order. Action order (unless you change) would automatically be: MOVE_UP, MOVE_DOWN,

MOVE_LEFT, MOVE_RIGHT

DFS algorithm should find solutions quickly for even the following big maze example:

```
$ ./p1 -p eat-food -l layouts/bigMaze.lay -w calm -s BFS -u 200
$ ./p1 -p eat-food -l layouts/bigMaze.lay -w calm -s BFS -u 200
```

BFS algorithm should find optimal solutions for the above problems (and probably expands more nodes):

```
$ ./p1 -p eat-food -l layouts/bigMaze.lay -w calm -s BFS -u 200
$ ./p1 -p eat-food -l layouts/bigMaze.lay -w calm -s BFS -u 200
```

Q2. Uniform Cost Search – 2 points

In this question, you will implement uniform cost search. Depending on the weather conditions, the `getStateActionCost()` function will give different costs for different state-action pairs. May need to implement/extend:

- `Agent::add2Fringe()`
- `Agent::removeFromFringe()`
- `Agent::isInClosedList ()`

The agent should find optimal path for the corresponding costs. Please check your solutions and see if agent behaves properly in the corresponding weather conditions:

```
$ ./p1 -p eat-food -l layouts/mediumMaze.lay -w calm -s UCS -u 500
$ ./p1 -p eat-food -l layouts/mediumMaze.lay -w lightning-at-west -s UCS -u 500
$ ./p1 -p eat-food -l layouts/mediumMaze.lay -w lightning-at-east -s UCS -u 500
$ ./p1 -p eat-food -l layouts/mediumMaze.lay -w lightning-at-north -s UCS -u 500
$ ./p1 -p eat-food -l layouts/mediumMaze.lay -w lightning-at-south -s UCS -u 500
```

Q3. (A* search) – 2 points

Implement A* search using MANHATTAN distance heuristics (Problems.h). You might need to implement/extend the following functions:

- `Agent::add2Fringe()`
- `Agent::removeFromFringe()`
- `Agent::isInClosedList ()`

You will need to implement an admissible and consistent heuristic (Manhattan Distance) here:

- `MazeProblem::heuristicFunc ()`

Now our agent can use problem-specific information for informed search! A* search should find optimal solution for bigMaze slightly faster (less number of expanded nodes) than UCS (around 550-620 nodes)

```
$ ./p1 -p eat-food -l layouts/bigMaze.lay -w calm -s UCS -u 100
$ ./p1 -p eat-food -l layouts/bigMaze.lay -w calm -s A_STAR -u 100
```

Questions 4-5 Finding all the corners (-p visit-corners)

Now we have a new task/problem: The agent will visit all the corners without caring about the food. The CornersState represents the state and CornersProblem represents the problem. CornersState is inherited from MazeState and CornersProblem is inherited from MazeProblem.

Q4. State representation – 3 points

Choose a state representation for this problem. The state structure should be compact and not contain any unnecessary information. Otherwise, you will lose significant point from this question. In other words, no irrelevant information in state description. Extend the following:

- class CornersState in CornersProblem.h
- CornersState::CornersState() in .cpp
- CornersState::isSameState () in .cpp
- CornersProblem::getNextState() in .cpp
- CornersProblem::setInitState () in .cpp
- CornersProblem::isGoalState() in .cpp

Feel free to add any new methods or data structures. I do not think there is a need to add new methods, though.

Any uninformed search should solve this task now for easy layouts(note the -p argument):

```
$ ./p1 -p visit-corners -l layouts/tinyCorners.lay -w calm -s BFS -u 100
$ ./p1 -p visit-corners -l layouts/mediumCorners.lay -w calm -s UCS -u 100
```

Q5. Heuristics for visit-corners – 3 points

Design and implement a non-trivial heuristics (which should be consistent). Not-admissible or inconsistent heuristics will get no credit.

Implement/extend the following:

- CornersProblem::heuristicFunc()

```
$ ./p1 -p visit-corners -l layouts/tinyCorners.lay -w calm -s A_STAR -u 100
```

If your heuristic is consistent, the grading will be as follows:

- 1 point for any non-trivial consistent heuristics
- 3 points for expanding fewer than 1600 nodes.
- 4 point for expanding fewer fewer than 1200 nodes.

My heuristic expands 1133 nodes :) But it is possible to do even better!

Please read the definitions of **admissible heuristics** and **consistent heuristics**. If you use inconsistent heuristics, it will be (probably) detected by Node::expand().

Q6. Eating all the food (-p eat-all-foods) – 6 points

Now, we have a new task/problem: the agent needs to eat all the food around with smallest cost. For example shortest path in calm weathers. For this, you need to implement a new state description and problem description. Then, you need to write an admissible and consistent heuristics so that A* search finds an optimal path quickly. The files you need to extend are:

- AllFoodsMazeProblem.h
- AllFoodsMazeProblem.cpp

Feel free to add any new methods or data structures: you will probably need them.

Try your agent with trickySearch.lay:

```
$ ./p1 -p eat-all-foods -l layouts/trickySearch.lay -w calm -s BFS -u 100
$ ./p1 -p eat-all-foods -l layouts/trickySearch.lay -w calm -s A_STAR -u 100
```

My BFS implementation expands ~16000 nodes. A* start search should perform much better. In this question, you will be graded based on performance of your A* search.

Not-admissible or inconsistent heuristics will get no credit.

If your heuristic is consistent, the grading will be as follows:

- Fewer than 7000: 6 points
- Fewer than 9000: 5 points
- Fewer than 12000: 4 points
- Fewer than 15000: 3 points

Report:

- Q4: Describe how you defined state with 2 sentences at most. Copy class definition and same state check:
 - `class CornersState{}`;
 - `bool CornersState::isSameState (State *state) {`
- Q5: Describe the heuristic used for visit-corners problem with no more than 2 sentences. Please be clear.
- Q6: Describe the heuristic used for eat-all-foods problem with no more than 2 sentences. Please be clear.