COMP3411/9814 Artificial Intelligence
Term 1, 2025

Assignment 1 – Search, Pruning and Treasure Hunting

Due: Friday 21 March, 10pm
Marks: 25% of final assessment

In this assignment you will be examining search strategies for the 15-puzzle, and pruning in alpha-beta search trees. You will also implement an AI strategy for an agent to play a text-based adventure game. You should provide answers for Questions 1 to 3 (Part A) in a written report, and implement your agent to interact with the provided game engine (Part B).

Note: Parts A and B must be submitted separately ! Submission details are at the end of this specification.

Part A: Search Strategies and Alpha-Beta Pruning

Question 1: Search Strategies for the 15-Puzzle (2 marks)

For this question you will construct a table showing the number of states expanded when the 15-puzzle is solved, from various starting positions, using four different search strategies:

(i) Breadth First Search
(ii) Iterative Deepening Search
(iii) Greedy Search (using the Manhattan Distance heuristic)
(iv) A* Search (using the Manhattan Distance heuristic)

Download the file `path_search.zip` from this directory:
`https://www.cse.unsw.edu.au/~cs3411/25T1/code/`
(or download it from here).

Unzip the file and change directory to `path_search`:

```
unzip path_search.zip
cd path_search
```

Run the code by typing:

```
python3 search.py --start 2634-5178-AB0C-9DEF --s bfs
```

The `--start` argument specifies the starting position, which in this case is:

2	6	3	4
5	1	7	8
A	B		C
9	D	E	F

Start State

1	2	3	4
5	6	7	8
9	A	B	C
D	E	F	

Goal State

The Goal State is shown on the right. The `--s` argument specifies the search strategy (`bfs` for Breadth First Search).

The code should print out the number of expanded nodes (by thousands) as it searches. It should then print a path from the Start State to the Goal State, followed by the number of nodes Generated and Expanded, and the Length and Cost of the path (which are both equal to 12 in this case).

(a) Draw up a table in this format:

Start State	BFS	IDS	Greedy	A*
start1				
start2				
start3				

Run each of the four search strategies from three specified starting positions, using the following combinations of command-line arguments:

Starting Positions:

start1: `--start 1237-5A46-09B8-DEFC`
start2: `--start 134B-5287-960C-DEAF`
start3: `--start 7203-16B4-5AC8-9DEF`

Search Strategies:

BFS: `--s bfs`
IDS: `--s dfs --id`
Greedy: `--s greedy`
A*Search: `--s astar`

In each case, record in your table the number of nodes Expanded during the search.

(b) Briefly discuss the efficiency of these four search strategies, based on the results from part (a).

Question 2: Heuristic Path Search for 15-Puzzle (3 marks)

In this question you will be exploring a search strategy known as **Heuristic Path Search**, which is a best-first search using the objective function:

$$f_w(n) = (2 - w)g(n) + wh(n),$$

where $h()$ is an admissible heuristic and w is a number between 0 and 2. Heuristic Path Search is equivalent to Uniform Cost Search when $w = 0$, to A* Search when $w = 1$, and Greedy Search when $w = 2$. It is Complete for all w between 0 and 2.

(a) Prove that Heuristic Path Search is **optimal** when $0 \leq w \leq 1$.

Hint: show that minimizing $f(n) = (2 - w)g(n) + wh(n)$ is the same as minimizing $f'(n) = g(n) + h'(n)$ for some function $h'(n)$ with the property that $h'(n) \leq h(n)$ for all n.

(b) Draw up a table in this format (the top row has been filled in for you):

	start4		start5		start6	
IDA* Search	48	1606468	52	3534563	54	76653772
HPS, $w = 1.1$						
HPS, $w = 1.2$						
HPS, $w = 1.3$						
HPS, $w = 1.4$						

Run the code on each of the three start states shown below, using Heuristic Path Search with $w = 1.1, 1.2, 1.3$ and 1.4 .

Starting Positions:

start4: `--start 8192-6DA4-0C5E-B3F7`
start5: `--start 297F-DEB4-A601-C385`
start6: `--start F5B6-C170-E892-DA34`

Search Strategies:

HPS, $w = 1.1$: `--s heuristic --w 1.1`
HPS, $w = 1.2$: `--s heuristic --w 1.2`
HPS, $w = 1.3$: `--s heuristic --w 1.3`
HPS, $w = 1.4$: `--s heuristic --w 1.4`

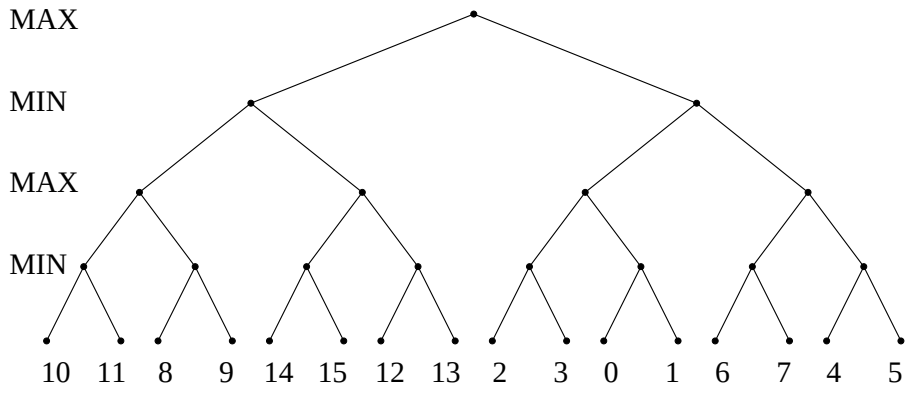
In each case, record in your table the length of the path that was found, and the number of nodes Expanded during the search. Include the completed table in your report.

If the process runs out of memory (particularly for `start6` with $w = 1.1$), try running it again but with "`--id`" added to the command-line arguments (this will make it switch to the Iterative Deepening version of Heuristic Path Search, which expands a similar number of nodes but uses far less memory).

(c) Briefly discuss the tradeoff between speed and quality of solution for Heuristic Path Search with different values of w, based on the results from part (b).

Question 3: Game Trees and Pruning (4 marks)

(a) The following game tree is designed so that alpha-beta search will prune as many nodes as possible. At each node of the tree, all the leaves in the left subtree are preferable to all the leaves in the right subtree (for the player whose turn it is to move).



Trace through the alpha-beta search algorithm on this tree, showing the values of alpha and beta at each node as the algorithm progresses, and clearly indicate which of the original 16 leaves are evaluated (i.e. not pruned).

(b) Now consider another game tree of depth 4, but where each internal node has exactly **three** children. Assume that the leaves have been assigned in such a way that alpha-beta search prunes as many nodes as possible. Draw the shape of the pruned tree. How many of the original 81 leaves will be evaluated?

Hint: If you look closely at the pruned tree from part (a) you will see a pattern. Some nodes explore all of their children; other nodes explore only their leftmost child and prune the other children. The path down the extreme left side of the tree is called the line of best play or Principal Variation (PV). Nodes along this path are called PV-nodes. PV-nodes explore all of their children. If we follow a path starting from a PV-node but proceeding through non-PV nodes, we see an alternation between nodes which explore all of their children, and those which explore only one child. By reproducing this pattern for the tree in part (b), you should be able to draw the shape of the pruned tree (without actually assigning values to the leaves or tracing through the alpha-beta search algorithm).

(c) What is the time complexity of alpha-beta search, if the best move is always examined first (at every branch of the tree)? Explain why.

Part B: Treasure Hunt (16 marks)

For this part you will be implementing an agent to play a simple text-based adventure game. The agent is considered to be stranded on a small group of islands, with a few trees and the ruins of some ancient buildings. The agent is required to move around a rectangular environment, collecting tools and avoiding (or removing) obstacles along the way.

The obstacles and tools within the environment are represented as follows:

Obstacles		**Tools**	
`T`	tree	`a`	axe
`-`	door	`k`	key
`*`	wall	`d`	dynamite
`~`	water	`$`	treasure

The agent will be represented by one of the characters `^`, `v`, $<$ or $>$, depending on which direction it is pointing. The agent is capable of the following instructions:

L turn left
R turn right
F (try to) move forward
U (try to) unlock a door, using an key
C (try to) chop down a tree, using an axe
B (try to) blast a wall, tree or door, using dynamite

When it executes an L or R instruction, the agent remains in the same location and only its direction changes. When it executes an F instruction, the agent attempts to move a single step in whichever direction it is pointing. The F instruction will fail (have no effect) if there is a wall or tree directly in front of the agent.

When the agent moves to a location occupied by a tool, it automatically picks up the tool. The agent may use a C, U or B instruction to remove an obstacle immediately in front of it, if it is carrying the appropriate tool. A tree may be removed with a C (chop) instruction, if an axe is held. A door may be removed with a U (unlock) instruction, if a key is held. A wall, tree or door may be removed with a B (blast) instruction, if dynamite is held.

Whenever a tree is chopped, the tree automatically becomes a raft which the agent can use as a tool to move across the water. If the agent is not holding a raft and moves forward into the water, it will drown. If the agent is holding a raft, it can safely move forward into the water, and continue to move around on the water, using the raft. When the agent steps back onto the land, the raft it was using will sink and cannot be used again. The agent will need to chop down another tree in order to get a new raft.

If the agent attempts to move off the edge of the environment, it dies.

To win the game, the agent must pick up the treasure and then return to its initial location.

Running as a Single Process

Download the file `src.zip` from this directory:
`https://www.cse.unsw.edu.au/~cs3411/25T1/hw1raft`
(or download it from here).
Copy the archive into your own filespace, unzip it, then type

```
cd src
javac *.java
java Raft -i s0.in
```

You should then see something like this:

```
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
~~  d   *     T   a  ~~
~~    *-*     ***    ~~
~~****     v     ****~~
~~TTT**         **TTT~~
~~  $ **   k   **    ~~
~~     **     **     ~~
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~

Enter Action(s):
```

This allows you to play the role of the agent by typing commands at the keyboard (followed by <Enter>). Note:

- a key can be used to open any door; once a door is opened, it has effectively been removed from the environment and can never be "closed" again.
- an axe or key can be used multiple times, but each dynamite can be used only once.
- C, U or B instructions will fail (have no effect) if the appropriate tool is not held, or if the location immediately in front of the agent does not contain an appropriate obstacle.

Running in Network Mode

Follow these instructions to see how the game runs in network mode:

1. open two windows, and cd to the src directory in both of them.
2. choose a port number between 1025 and 65535 – let's suppose you choose 31415.
3. type this in one window:

   ```
   java Raft -p 31415 -i s0.in
   ```

4. type this in the other window:

   ```
   java Agent -p 31415
   ```

In network mode, the agent runs as a separate process and communicates with the game engine through a TCPIP socket. Notice that the agent cannot see the whole environment, but only a 5-by-5 "window" around its current location, appropriately rotated. From the agent's point of view, locations off the edge of the environment appear as a dot.

We have also provided a C version of the agent, which you can run by typing

```
make
```

```
./agent -p 31415
```

Writing an Agent

At each time step, the environment will send a series of 24 characters to the agent, constituting a scan of the 5-by-5 window it is currently seeing; the agent must send back a single character to indicate the action it has chosen.

You are free to write the agent in any language of your choosing.

- If you are coding in Java, your main file should be called `Agent.java` (you are free to use the supplied file `Agent.java` as a starting point)
- If you are coding in Python, your main file should be called `agent.py` (you are free to use the supplied file `agent.py` as a starting point) and the first line should specify the version of Python you are using, e.g.

  ```
  #!/usr/bin/python3
  ```

- If you are coding in C, you are free to use the files `agent.c`, `pipe.c` and `pipe.h` as a starting point. You must include a `Makefile` with your submission which, when invoked with the command "make", will produce an executable called `agent`.
- In other languages, you will have to write the socket code for yourself.

You may assume that the specified environment is no larger than 80 by 80, but the agent can begin anywhere inside it.

Additional examples of input environments can be found in the directory
`https://www.cse.unsw.edu.au/~cs3411/25T1/hw1raft/sample`
(or download it from here).

Question

At the top of your code, in a block of comments, you must provide a brief answer (one or two paragraphs) to this Question:

> Briefly describe how your program works, including any algorithms and data structures employed, and explain any design decisions you made along the way.

Submission

Parts A and B should be submitted separately.

You should submit your report for Part A by typing

```
give cs3411 hw1a hw1a.pdf
```

You should submit your code for Part B by typing

```
give cs3411 hw1raft ...
```

(Replace `...` with the names of your submitted files)

You can submit as many times as you like – later submissions will overwrite earlier ones. You can check that your submission has been received by using one of this command:

```
3411 classrun -check
```

The submission deadline is **Friday 21 March, 10 pm**.

A penalty of 5% will be applied to the mark for every 24 hours late after the deadline, up to a maximum of 5 days (in accordance with UNSW policy).

Additional information may be found in the FAQ and will be considered as part of the specification for the project. Questions relating to the project can also be posted to the course Forums. If you have a question that has not already been answered on the FAQ or the Forums, you can email it to `cs3411@cse.unsw.edu.au`

Please ensure that you submit the source files and NOT any binary files. The `give` system will compile your program using your `Makefile` and check that it produces a binary file (or java class files) with the correct name.

Assessment

Your program will be tested on a series of sample inputs with successively more challenging environments. There will be:

- 10 marks for functionality (automarking)
- 6 marks for Algorithms, Style, Comments and answer to the Question

You should always adhere to good coding practices and style. In general, a program that attempts a substantial part of the job but does that part correctly will receive more marks than one attempting to do the entire job but with many errors.

Plagiarism Policy Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar assignments in previous years, if applicable) and serious penalties will be applied, including an entry on UNSW's plagiarism register.

You are also not allowed to submit code obtained with the help of ChatGPT, Claude, GitHub Copilot, Gemini or similar automatic tools.

- Do not copy code from others; do not allow anyone to see your code.
- Do not copy code from the Internet; do not develop or upload your own code on a publicly accessible repository.
- Code generated by ChatGPT, Claude, GitHub Copilot, Gemini and similar tools will be treated as plagiarism.

Please refer to the on-line resources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Academic Integrity and Plagiarism
- UNSW Plagiarism Policy

Good luck!