AI 1: lab session 1

Agents and Search

Instructions

- Formulate your answers clearly and always provide a motivation for your answer.
- Read the book and use the information from the lectures. Read the questions carefully and feel free to ask for clarifications if something is not clear.
- You can use the helpdesk email `airug1718@gmail.com` to ask questions. Make sure you read the instructions to contact the helpdesk provided on Nestor. Only emails sent according to the instructions will be replied to.
- For the programming assignments, always supply all relevant (self-written) source codes such that the teaching assistants can test them. It is not allowed to use code supplied by others. If we suspect plagiarism, the exam committee will immediately be notified!

Rules for submission

- Submit a digital version of your team's report through Nestor (via the Submissions section). Reports need to be written in LaTeX. A template is available on Nestor.
- Submit your report by the deadline provided on Nestor. Deadlines are strict.
- Supply the names and student numbers of all members of your team. Also clearly write down your Learning Community.

Grading Every session is graded by the teaching assistants. The grade of this lab assignment will count for 15% of your final grade. Note that the deadlines are strict: we subtract 2^{n-1} grading points for a report that is n days late.

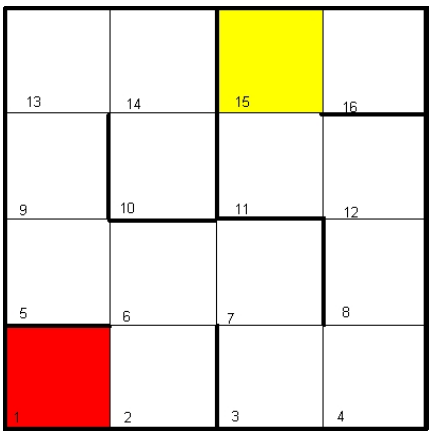

Figure 1: Maze

Theory assignment

1. **PEAS description (theory, do not make this exercise during the computer labs)**
   Study the material on PEAS descriptions from the book. Write a PEAS-description for the following agents:

   1. A reversi/othello computer (see `http://en.wikipedia.org/wiki/Reversi`)
   2. A robotic lawn mower (zie `http://en.wikipedia.org/wiki/Robotic_lawn_mower`).

   Characterize each domain as fully observable / partially observable, deterministic / stochastic, episodic / sequential, static / dynamic, discrete / continuous and single agent / multi-agent. Explain which agent architecture is best for each domain.

2. **Maze (theory, do not make this exercise during the computer labs)**

   Given is a simple maze environment (see Figure 1 above). An agent in this environment can choose from 4 actions: move one step north, east, south, or west.

   States : The squares of the maze.
   Actions: step in direction north, east, south, or west.
   Goal Test: Is the agent in the yellow colored square (square 15)?
   Path cost: Each step has cost 1.
   Initial state: The agent is in the red square (square 1).

   Furthermore, the agent is provided with a collision detector that indicates whether it has just collided against a wall. If the agent hits a wall, then it knows that the corresponding direction is blocked and it will try the next direction from the set of possible actions. The agent always tries actions in the order [N=north, E=east, S=south, W=west].

   First we try to solve the maze using the following *open list* version of depth first search (DFS).

```
1  procedure mazeDFS(maze, start, goal):
2     stack = []
3     stack.push(start)
4     while stack is not empty:
5        loc = stack.pop()
6        if loc == goal:
7          print "Goal found"
```

```
8          return
9        for move in [N,E,S,W]:   # in this order!
10         if allowedMove(loc, move):
11            stack.push(neighbour(maze, loc, move))
12    print "Goal not found"
```

For example, the sequence of nodes visited by the algorithm in the search for a DFS path from square 5 to square 10 (i.e. we call `MazeDFS(maze, 5, 10)`) is *N, N, E, S*.

1. Why is it not possible to use `mazeDFS()` to find a path from the red square to the yellow square? Where does it go wrong?
2. How should the (pseudo-)code be modified such that it is guaranteed to always find a solution (if one exists)?
3. In which order are states visited by the modified algorithm for the call `mazeDFS(maze, 1, 15)`?
4. In which order are states visited by the modified algorithm for the call `mazeDFS(maze, 1, 15)` if we change the order of visiting neighbours into `[N,S,W,E]`?
5. If we replace the stack in the original algorithm by a FIFO queue, then the algorithm is suddenly turned into (the open list version of) breadth first search (BFS). Will this algorithm always find a solution (if one exists)? Explain your answer.
6. In which order are states visited by the open list BFS algorithm for the call `mazeDFS(maze, 1, 15)`?
7. Can we reduce the number of visited states of the BFS algorithm? If yes, how should the program be modified?
8. Would you use (a variation of) DFS or (a variation of) BFS for path searching in large mazes? Explain your answer.

Programming assignment

3. **The answer to the Ultimate Question of Life, The Universe, and Everything is 42.** [*Note: If you do not understand the title of this exercise, just google for the number 42.*]

   On Nestor, you can find the file `f42.tar`. After extracting it, you will find a directory `find42` with some source files in it. Go into this directory, and compile the supplied code by simply typing `make`.

   In this exercise we consider the following simple mathematical puzzle. Given an integer n from the domain $D = [0..10^6)$, we can perform 6 actions on n:

   - $n \longrightarrow n + 1$: increment n (only allowed if $n + 1 \in D$). The cost of this operation is 1 unit.
   - $n \longrightarrow n - 1$: decrement n (only allowed if $n - 1 \in D$). The cost of this operation is 1 unit.
   - $n \longrightarrow 2 \cdot n$: double n (only allowed if $2 \cdot n \in D$). The cost of this operation is 2 units.
   - $n \longrightarrow 3 \cdot n$: triple n (only allowed if $3 \cdot n \in D$). The cost of this operation is 2 units.
   - $n \longrightarrow \lfloor n/2 \rfloor$: integer division of n by two. The cost of this operation is 3 units.
   - $n \longrightarrow \lfloor n/3 \rfloor$: integer division of n by three. The cost of this operation is 3 units.

   Using these actions, there exist many 'paths' from 0 to 42. For example

   $$0 \rightarrow_{+1} 1 \rightarrow_{.3} 3 \rightarrow_{.2} 6 \rightarrow_{.2} 12 \rightarrow_{.3} 36 \rightarrow_{+1} 37 \rightarrow_{+1} 38 \rightarrow_{+1} 39 \rightarrow_{+1} 40 \rightarrow_{+1} 41 \rightarrow_{+1} 42$$

The path has length 11 (it consist of 11 steps), but its cost is 1+2+2+2+2+1+1+1+1+1+1=15. There exist 'cheaper' paths from 0 to 42.

In this exercise we consider the problem of finding a path from some $a \in D$ to some $b \in D$. We will use DFS (Depth First Search), BFS (Breadth First Search), UCS (Uniform Cost Search), and IDS (Iterative Deepening Search).

The program accepts the command line: `search <STACK|FIFO|HEAP> [start] [goal]`
For example, if you want to find a path from 0 to 42 using BFS, type `./search FIFO 0 42`. The output will be:

```
Problem: route from 0 to 42
goal reached (5279 nodes visited)
#### fringe statistics:
 #size        :   23546
 #maximum size:   23547
 #insertions  :   28825
 #deletions   :    5279
####
```

The output reports that a path was found after visiting 5279 states (using a FIFO queue, so the algorithm is BFS). The search terminated after having found the goal (42). At that moment, there were still 23546 states in the FIFO queue left. The maximum size of the queue during the search process was 23547 states, and the total number of insertion in the queue was 28825. The number of deletions (which is of course the same as the number of visited states) was 5279.

Study the supplied source code. Make sure you understand the data structures used in the program. Especially the files `fringe.h` and `fringe.c` are important. In these files, a general (partly implemented) framework is given for building and manipulating fringes (which can be of type `LIFO(=STACK)`, `FIFO`, or `HEAP(=PRIO)`. The Abstract data type `Fringe` is completely implemented for the types `STACK/LIFO` and `FIFO`. The implementation of `HEAP/PRIO` is incomplete.

1. Try to use the program to find paths (using BFS) from 0 to 99, from 0 to 100, and from 0 to 102. Also try to find a path from 1 to 0, using DFS. Next, try to find a DFS path from 0 to 1. Explain the results.
2. Modify the program such that DFS and BFS can solve the above mentioned cases. Does this modification have effect on the performance of the algorithm?
3. Modify the program such that the algorithm prints a solution path and its length and cost. The above (example) path should be printed as:

   ```
   0 (+1)-> 1 (*3)-> 3 (*2)-> 6 (*2)-> 12 (*3)->36 (+1)-> 37 (+1)->38
   (+1)->39 (+1)->40 (+1)->41 (+1)-> 42
   length = 11, cost = 15
   ```

4. Complete the priority queue operations of the abstract data type `Fringe`.
5. Change the program such that the command line `./search PRIO 0 42` will search for the optimal cost path from 0 to 42 using the UCS algorithm. What is (are) the optimal paths from 0 to 42?
6. Make (from scratch) a program that solves the problem using iterative deepening. You are advised to use explicit recursion.
7. Compare the performance of the four search algorithms for several inputs. What are your conclusions?

4. **Knight distance using A***

In the game of chess, a knight can move two squares horizontally and one square vertically, or two squares vertically and one square horizontally (see left figure below).

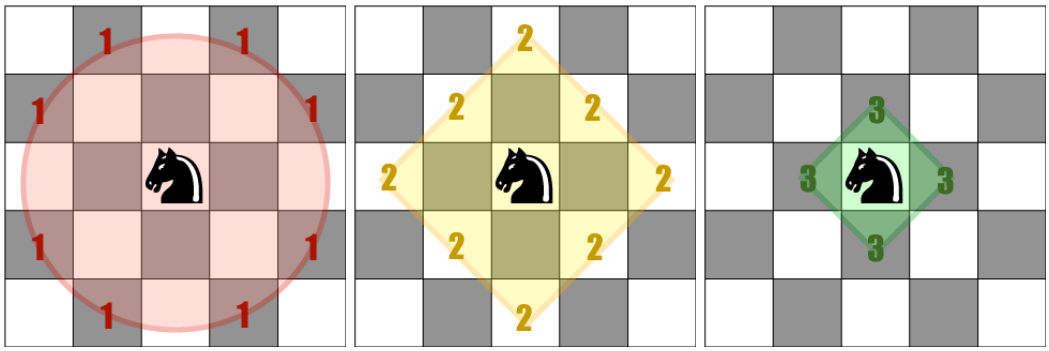

Figure 2: Knight distances on a 5×5 chessboard.

We consider a large 500×500 chessboard. A square is a pair (x, y) where $0 \leq x < 500$ and $0 \leq y < 500$. In this exercise we want to find the *length of the shortest path (number of knight moves)* from a starting location (x_0, y_0) to a goal location using the A^* algorithm.

On Nestor, you can find the file `idknight.c` which solves this problem. However it uses IDS (Iterative Deepening Search), and is therefore only supplied to verify your program. It is (really) slow for long paths. For example, the program visits 13980836253 states in the search for the optimal path from $(0, 0)$ to $(499, 499)$ (the length of the optimal path is 334 moves).

- Make a version of the program that uses the A^* algorithm.
- Invent at least two admissible heuristics for this problem, and compare their effectiveness by computing (approximately) their *effective branching factor*.