Artificial Intelligence (521495A), Spring 2022
Exercise 1: Answers to Problems 1-5

Problem 1. State-space search problem definition

(a) State can be represented by a pair (x, y), where x and y are the amounts of water in pot A and pot B, respectively.

The initial state is (x, y) = (0, 0).

The goal test is: (x = 2) OR (y = 2). In this case, the other pot may contain some water.

If it is wanted, that it is empty, the goal test could be for example ((x = 2) OR (y = 2)) AND (x+y = 2).

Six actions can be recognized:

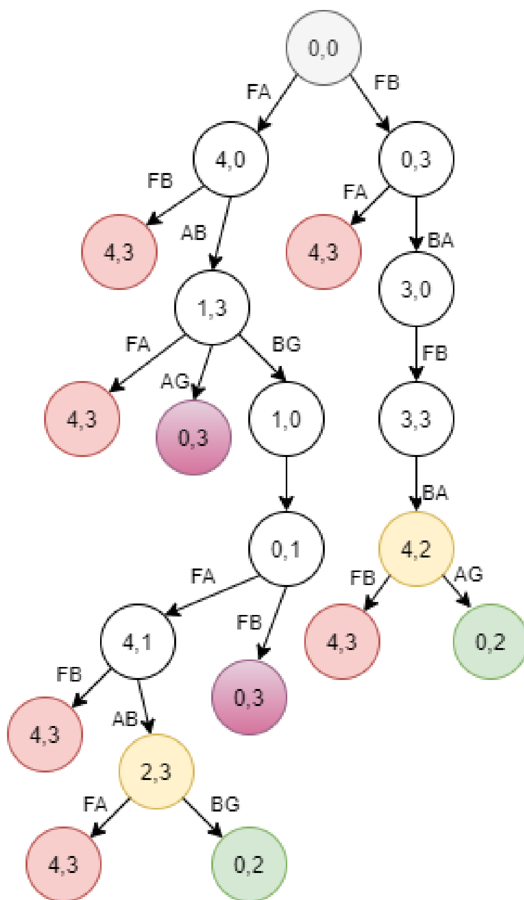
Id	Action	Transition	Applicable, if	Cost 1	Cost 2
FA	Fill A from tap	(X,Y) -> (4,Y)	X < 4	0	4-X
FB	Fill B from tap	(X,Y) -> (X,3)	Y < 3	0	3-Y
AG	Pour A to ground	(X,Y) -> (0,Y)	X > 0	X	0
BG	Pour B to ground	(X,Y) -> (X,0)	Y > 0	Y	0
AB	Pour from A to B as much as possible	(X,Y) -> (X-d,Y+d), where d = min(X,3-Y)	X > 0 AND Y < 3	0	0
BA	Pour from B to A as much as possible	(X,Y) -> (X+d,Y-d), where d = min(4-X,Y)	X < 4 AND Y > 0	0	0

Cost 1: Penalizes for pouring water to ground.
Cost 2: Penalizes for taking water from the tap.

The column "Transition" specifies the successor function and the column "Applicable, if" provides a condition for state, that must be fulfilled. Different kinds of costs can be defined for the actions can be given. The stated requirement was to minimize the waste of water, so the Cost 1 seems to be the right one. It could also be Cost 2.

Note how you can use variables to specify the actions. In the specification of transitions, the current state is referred to with pair (X,Y) and the next states are specified using those variables. State space conditions and associated costs refer to the variables of introduced in transition descriptions.

(b) Search tree



Actions that lead to the state of any ancestor not shown. In addition, expansion not shown from

4,3 — because the next state would be a state seen already in level 1.

0,3 — because the state has been seen in level 1.

Goal fulfilled:

Terminal for goal test 1: Other pot contains 2 liters.

Terminal node for goal test 2: In addition, other pot is empty.

(c) There are loops in the state space. Therefore, the depth-first strategy would not work with the tree-search algorithm. In general, the tree-search would do unnecessary node expansions (note what happens with the nodes labelled (4,3) and (0,3) in the search tree). Counterweight for the need for expanding is that the tree-search does not have to maintain the closed set.

Problem 2. Graph-search algorithm applied

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

a) Depth-first search.

States expanded: Start, A, C, D, B, Goal

Path returned: Start-A-C-D-Goal

b) Breadth-first search.

States expanded: Start, A, B, D, C, Goal

Path returned: Start-D-Goal

c)Uniform cost search.

States expanded: Start, A, B, D, C, Goal

Path returned: Start-A-C-Goal

d) Greedy search with the heuristics shown on the graph.

States expanded: Start, D, Goal

Path returned: Start-D-Goal

e) A* search with the same heuristic.

States expanded: Start, A, D, C, Goal

Path returned: Start-A-C-Goal

Problem 3. Tree-search algorithm applied

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Note that the goal test is applied to the node, when it is removed from there.

Detailed explanations of processing:

a) Depth-first: (LIFO queue)
Tree-search algorithm expands like this: Start-A-C-D-B-Start-A….
So, it runs into loop and will not manage to find the solution.

b) Breadth-first: FIFO queue used
1. Into the queue: Start
2. Remove "Start". Into the queue: Start-A, Start-B, Start-D
3. Remove "Start-A". Into the queue: Start-A-C
4. Remove "Start-B". Into the queue: Start-B-D
5. Remove "Start-D". Into the queue: Start-D-B, Start-D-C, Start-D-Goal
6. Remove "Start-A-C". Into the queue: Start-A-C-D, Start-A-C-Goal
7. Remove "Start-B-D". Into the queue: Start-B-D-Start, Start-B-D-C, Start-B-D-C-Goal
8. Remove "Start-D-B". Into the queue: Start-D-B-Start
9. Remove "Start-D-C". Into the queue: Start-D-C-A, Start-D-C-Goal
10. Remove "Start-D-Goal". Terminate, goal reached.
Found path: **Start-D-Goal**

c) Uniform-cost search (priority queue based on cost)
1. Into the queue: Start (cost = 0)
2. Remove "Start" (0). Into the queue: Start-A (cost = 2), Start-B (cost = 3), Start-D (cost = 4)
3. Remove "Start-A" (2). Into the queue: Start-A-C (cost = 6)
4. Remove "Start-B" (3). Into the queue: Start-B-D (cost = 7)
5. Remove "Start-D" (4). Into the queue: Start-D-C (cost = 6), Start-D-B (cost = 9), Start-D-Goal (cost = 10)
6. Remove "Start-A-C" (6). Into the queue: Start-A-C-D (cost = 7), Start-A-C-Goal (cost = 8)
7. Remove "Start-D-C" (6). Into the queue: Start-D-C-Goal (cost = 8), Start-D-C-A (cost = 10)
8. Remove "Start-B-D" (7). Into the queue: Start-B-D-C (cost = 8), Start-B-D-Goal (cost = 12)
9. Remove "Start-A-C-D" (7). Into the queue: Start-A-C-D-B (cost = 11), Start-A-C-D-Start (cost = 12),
Start-A-C-D-Goal (cost = 12)
10. Remove "Start-A-C-Goal" (8). Terminate, goal reached.
Found path: **Start-A-C-Goal** (optimal path)

d) Greedy search: (priority queue based on heuristic) Assuming heuristic h = 10 for Start.
1. Into the queue: Start (h = 10)
2. Remove "Start". Into the queue: Start-D (h = 1), Start-A (h = 2), Start-B (h = 5)
3. Remove "Start-D". Into the queue: Start-D-Goal (h = 0), Start-D-C (h = 2), Start-D-B (h = 5)
4. Remove "Start-D-Goal". Terminate, goal reached.
Found path: **Start-D-Goal** (note: not optimal)

e) A* search: (priority queue based on cost+heuristic)

1. Into the queue: Start (cost+h = 10)
2. Remove "Start". Into the queue: Start-A (cost+h = 4), Start-D (cost+h = 6), Start-B (cost+h = 8)
3. Remove "Start-A" (4). Into the queue: Start-A-C (cost+h = 8)
4. Remove "Start-D" (6). Into the queue: Start-D-C (cost+h = 8), Start-D-Goal (cost+h = 10),
   Start-D-B (cost+h = 14)
5. Remove "Start-B" (8). Into the queue: Start-B-D (cost+h = 8)
6. Remove "Start-A-C" (8). Start-A-C-D (cost+h = 8), Start-A-C-Goal (cost+h = 8)
7. Remove "Start-D-C" (8). Into the queue: Start-D-C-Goal (cost = 8), Start-D-C-A (cost+h = 12)
8. Remove "Start-B-D" (8). Into the queue: Start-B-D-C (cost+h = 10), Start-B-D-Goal (cost+h = 12)
9. Remove "Start-A-C-D" (7). Into the queue: Start-A-C-D-Goal (cost = 12), Start-A-C-D-B (cost = 16)
   Start-A-C-D-Start (cost = 22)
10. Remove "Start-A-C-Goal" (8). Terminate, goal reached.

Found path: **Start-A-C-Goal** (an optimal path)

Problem 4. "Breakout Method" paper by Morris

The paper considers a method for solving **constraint satisfaction problems** (CSPs)[1] using local search. Specifically, one starts from a configuration (state)[2] where some constraints are violated. Then, one looks at the nearby configurations obtained by local changes and selects a configuration, which reduces the number of constraint violations. So, the basic technique is doing **hill-climbing** search and it may get stuck to some local minimum (= situation, where some constraints are still violated, but there is no better nearby configuration).

Author's concern is how to solve the local minimum problem. He takes intuition from "a physical force metaphor". The idea is that local minimum can be thought as an equilibrium state between objects (variables) surrounded by a potential barrier (local changes lead to worse solutions). One must break through the barrier in order to reach an equilibrium state with a lower energy.

This is done as follows (algorithm is provided in Figure 2):

1. Cost function to be minimized (to zero) is defined as the sum C = $\sum_{i \in V} w_i N_i$, where ***V*** denotes the set of variables, w_i is a weight associated with the variable i, and N_i is the "nogood" value:

$$N_i = \begin{cases} 0 & \text{if all constraints are satisfied for the variable } i \\ 1 & \text{otherwise} \end{cases}$$

Note that at the global minimum C = 0.

2. Initially, the weight of each variable is equal to one.

3. When the search reaches a local minimum, the weights of nogoods are increased by unit increments until the state is not a local minimum anymore. So, the idea is to modify the cost function in this manner in order to make breakout.

Author provides experimental results for different kinds of CSPs. The algorithm has intuitive grounds and to provide more support to the proposed method, he compares it to a closely related "Fill" algorithm, which is shown to be complete (that is, quaranteed to find a solution).

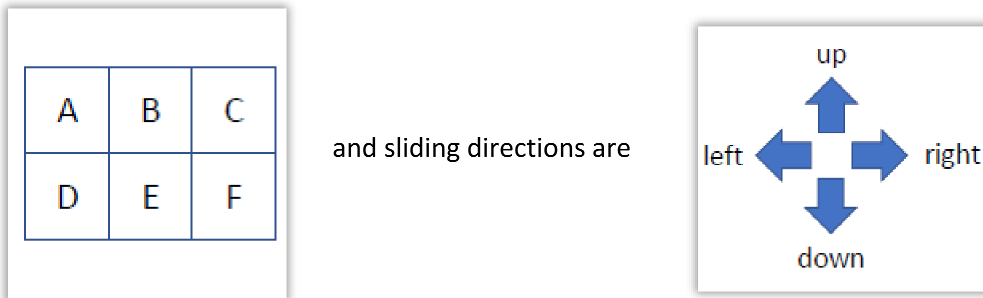
[1] CSPs are discussed in Lecture 6.

[2] A state is a complete set of assignments for the variables.

Problem 5. State-space search for 6 tiles

(a) 5 tiles and empty position can be placed in 6! times, which is equal to 720. However, only half of those states is reachable from the initial state, so the size of the state space is 360. For example,

(b) One possibility is to specify actions as pairs (L, D), where L specifies the location of the tile to be moved and D specifies the sliding direction. Tile locations can be labelled as



and sliding directions are

In the table below, all 14 actions are listed in the first column. Sliding direction is shown with its initial letter.

(c) Precondition for an action is that some tile location is empty. Conditions are shown in the second column of the table.

(d) Current state can be encoded as a vector (a, b, c, d, e, f), where components make a permutation of values (0, 1, 2, 3, 4, 5), 0 indicating an empty tile position. Vectors for the next state can then be expressed as shown in the third column of the table.

(e) Last column in the table.

Action	Applicable, if empty	Next state, when Current state (a,b,c,d,e,f)	Cost
(A,r)	B	(b,a,c,d,e,f)	3
(A,d)	D	(d,b,c,a,e,f)	1
(B,l)	A	(b,a,c,d,e,f)	3
(B,r)	C	(a,c,b,d,e,f)	3
(B,d)	E	(a,e,c,d,b,f)	3
(C,l)	B	(a,c,b,d,e,f)	3
(C,d)	F	(a,b,f,d,e,c)	1
(D,u)	A	(d,b,c,a,e,f)	1
(D,r)	E	(a,b,c,e,d,f)	1
(E,l)	D	(a,b,c,e,d,f)	1
(E,u)	B	(a,e,c,d,b,f)	3
(E,r)	F	(a,b,c,d,f,e)	1
(F,l)	E	(a,b,c,d,f,e)	1
(F,u)	C	(a,b,f,d,e,c)	1

(f) A good heuristic should estimate the cost of the action sequence as closely as possible. Also it should not overestimate the cost.

In the lectures, two heuristics for 8-puzzle were discussed. Russell & Norvig's experimental results showed that heuristic $\mathbf{h_2}$ based on Manhattan distance is better than $\mathbf{h_1}$ (number of misplaced tiles).

Similarly, one could use a distance-based heuristic, which also considers the action costs. The heuristic is

$$h(STATE) = \sum_{TILE \in \{1,2,3,4,5\}} D(TILE)$$

where *D(TILE)* is the minimum sliding cost for *TILE*. Specifically, if there are two minimum step routes between two positions, the value of *D(TILE)* is the cost of the route, which minimizes the cost. Then the cost is not overestimated.

For example, consider moving tile "2" from the position D to the target C. The cost to be used in the heuristic function is 3. Also, if the initial position of "2" is A, the heuristic is based on the cost of the longer route, whose cost is 4.