Homework 2 – Artificial Intelligence

1. Generic missionaries problem

1.1 Formulation

1.1.1 State representation

A state can be encoded by the following means:

- **side1**: a tuple indicating how many cannibals and missionaries are located at side 1. E.g. (2,1) means that there are 2 cannibals and 1 missionary.
- **side2**: a tuple indicating how many cannibals and missionaries are located at side 2
- **boat_position**: indicates the current location fot he boat (0=>side1 and 1=>side2)

A state is encoded as: (side1, side2, boat_position). Please note that we do not need to explicitly write down **side2**, because it is always fixed based on the entities placed on **side1**. For example, ff there all M missionaries and C cannibals are on **side1**, no of them can be on **side2**.

A state is therefore even shorter encoded by the following tuples:

```
state = ((c,m), boat_position)
initial_state = ((C,M), 0)
goal_state = ((0,0), 1)

def is_goal_state(state=((c,m), boat_position)):
    return c==0 AND m==0 and boat_position==1
```

The initial state is $((C, M), 0)$ (all entities on **side1**, boat on **side1**). The final state is $((0, 0), 1)$ (all entities on **side2**, boat on **side2**).

1.1.2 Action representation

There are the following actions defined. Each action is a tuple which indicates how many cannibals and missionaries are transported to the other side. Thus, each action changes the current boat_position and the entities placed on both sides of the river. An action is therefore a tuple containing two integers:

```
action = (c,m)
```

In the general settings, all actions matching the following equation are valid: $(c, m); c <= m; c <= B <= C; m <= B <= M$

This formulation ensures, that during transport the number of cannibals will never exceed the number of missionaries. Moreover, the number of transported cannibals c and missionaries m can never exceed the total amount of cannibals C and missionaries M. Depending on the current state of the agent, however,

only a limited set of actions is valid. The following pseudo-code verifies if a given action (c_a, m_a) is valid based on the current state $((c, m), boat\_position)$:

```
global M, C, B

def is_valid_action(action=(c_a, m_a)):
    """
    Verifies if this action is valid (not considering the current state).
    """
    if c_a < 0 OR m_a < 0:
        # Invalid action, because no negative entities can be transported
        return False

    if c_a == 0 AND m_a == 0:
        # Invalid action, because zero entities
        return False

    if c_a > C OR m_a > M:
        # Invalid action, because exceeding the given entity limits
        return False

    if c_a > m_a:
        # Invalid action, because cannibals exceed missionaries
        return False

    return True


def can_perform_action(state=((c,m), boat_position), action=(c_a, m_a)):
    """
    Verifies if action is valid in given state.
    """
    if not is_valid_action(action):
        return False

    available_c = 0
    available_m = 0
    if boat_position == 0:
        # Boat is placed on side1 -> entities on side1 are given in c and m
        available_c = c
        available_m = m
    else:
        # boat is placed on side2 -> entities on side2 are given in (C-c) and (M-m)
        available_c = C-c
        available_m = M-m
    return  c_a <= available_c AND \
```

```
            m_a <= available_m AND
```

For a given state and a valid action, the following function computes the next state:

```
global C, M, B

def get_next_state(state=((c,m), boat_position), action=(c_a, m_a)):
    """
    Computes the next state based on the current state and the given action.
    Assumption: The given action is valid in the current state.
    """
    if boat_position == 0:
        # Subtract the transported entities from site1
        return (c - c_a, m - m_a), 1
    # Add the transported entities to site1
    return (c + c_a, m + m_a), 0
```

For a given state, the following function computes the set legal actions:

```
global C, M, B

def get_legal_actions(state=((c,m), boat_position)):
    """
    Return all legal actions in current state.
    """
    legal_actions = []
    # Iterate over all possible actions and find legal ones
    for c_a in range(C):
        for m_a in range(M):
            if can_perform_action((state, (c_a, m_a)):
                legal_actions.append((c_a, m_a))
    return legal_actions
```

1.1.3 Invalid states

A given state $((c, m), boat\_position)$ is assumed to be invalid, if the following equation evaluates to true: `(c > m AND m != 0) OR (c < m AND m != M)`

The left side of the equation evaluates to `true`, if cannibals can eat missionaries on **side1**. The right side of the equation is `true`, if cannibals can eat missionaries on **side2**.

This could be implemented with the following code:

```
global B, C, M
def is_state_invalid(state=((c,m), boat_position))
    return (c<m AND m != 0) OR (c < m AND m != M)
```

1.1.4 Path costs

We are interested in the solution, which requires the least number of river crossings. Thus, each action is assumed to have cost 1. The cost of a path to a certain state is the sum of all performed actions on this path.

1.2. Estimates

1.2.1 Average branching factor

The following is a estimate for the possible amount of actions: In each action, `m<=B` missionaries can pass the river. - if $m = 0$: Up to B cannibals can pass the river. This corresponds to B possible actions. - if $m = B$: There is only one action possible, since all seats are filled up with missionaries. - if $0 < m < B$: If all missionaries are seated, there are $B - m$ free seats on the boat. Since the number of cannibals can not exceed the number of missionaries, up to $c = min\{B - m; m\}$ cannibals can join the boat. There is also the possibility, that m missionaries head over with zero cannibals, thus we need to add one action: $min\{B - m; m\} + 1$

If we sum this up for all possible values of $m <= B$, we have the total amount of valid actions: $B + 1 + (\sum_{m=1}^{B-1} min\{B - m; m\} + 1)$

This computation delivers the maximum branching factor. With the value $B = 2$ this yields the result 5 (correct since we had 5 possible actions in homework 1). The minimum branching factor is 0, because invalid states do not have outgoing actions. I assume the average branching factor to be the median of the maximum and minimum factor. In the previous homework this was $(0 + 5)/2 = 2.5$. In this homework, this is the maximum branching factor divided by two: $\frac{B+1+(\sum_{m=1}^{B-1} min\{B-m;m\}+1)}{2}$

1.2.2 Depth of shortest solution

In order to estimate the depth of the shortest solution, I try to find a lower bound of actions to solve the problem. This corresponds to a underestimate of the depth of the shortest solution.

Assume, that we have to transport in total $T = M + C$ entities over the river. The best solution should carry a full boat of B entities in their final ride over the river. Thus, we need to carry $T - B$ entities over the river, while we need to make sure that the boat always is able to return to the other side. In a single round trip, we can carry at most $B - 1$ entities, since at least one entity needs to drive the boat back. The question is now, how many round trips are required to transport $T - B$ entities if one round trip can carry at most $B - 1$ entities? This can be computed by $math.ceil(\ \frac{T-B}{B-1}\ )$. A single round trip requires two actions. Additionally we need to add the final step. This yields to following formula for a lower bound of the required steps: $(2 * math.ceil(\ \frac{T-B}{B-1}\ )) + 1$

With the values $C = M = 3$ and $B = 3$ this yields to following lower bound of actions required to solve the problem: $2 * (math.ceil(\ \frac{4}{1}\ )) + 1 = 2 * 4 + 1 = 9$.

1.2.3 State space

To explore all possible states, we must only consider the placement of missionaries and cannibals on **side1** (since **side2** directly follows from this choice). Additionally, we must consider, that for each possible distribution of missionaries and cannibals the boat can in theory be placed on both sides of the river. The total state space can also contain invalid states. We can place up to C cannibals and up to M missionaries on **side1**. Since we can also place 0 cannibals or missionaries, we have the following amount of states: $2 * (C + 1) * (M + 1)$.

This matches the estimate made in homework1. For $M = C = 3$ this equation yields the result 32.

1.2.4 Size of search tree

To estimate the size of the search tree, I try to estimate the size of invalid states. I will then subtract the number of invalid states from the state space to estimate the size of the search tree. This is an underestimate, since the search algorithm will reach invalid states. Thus, invalid states will be part of the actual search tree. Moreover, the search tree might encounter loops, which are not reflected by this estimate.

Remember the state representation: $((c, m), boat\_position)$. In general, there are two options for invalid states: `- c > m AND m != 0 - c < m AND m != M`

Due two the symmetrical setup, both options have the same number of states. For the first option, we can observe the equation $\sum_{c=1}^{C}(c-1)$, because for a fixed number of cannibals c, there can be up to $c-1$ missionaries to create an invalid state. Since for each state, the boat can be on both sides of the river, the following is an estimation for all invalid states: $2 * 2 * \sum_{c=1}^{C}(c-1) = 4 * \sum_{c=1}^{C}(c-1)$

Compared to the last homework, this yields: $4 * \sum_{c=1}^{3}(c-1) = 4 * 3 = 12$ This number must be subtracted from the size of the state space (which was 32). In total, this estimates that the search tree contains 20 nodes. This is considered to be an under estimation.

1.3 Algorithms

We are facing a finite set of states. We might encounter loops. Thus, the search tree might be infinite if we do net discard re-visited states. We assume to have a reachable goal state and ech step cost is 1.

- DFS:
  - complete: complete, if we discard re-visited states

  - optimal: not optimal, because it stops immediately after a result was found (there might be a cheaper path though)
- BFS:
  - complete: is always complete
  - optimal: is optimal since we assume step cost 1
- Iterative-Deepening:
  - complete: is always complete
  - optimal: is optimal since we assume step cost 1
- Uniform-Search: Equal to BFS since we assume step cost 1
  - complete: yes
  - optimal: yes
- Best-First:
  - complete: complete, if we discard re-visited states, is incomplete if we do not discard re-visited states
  - optimal: not optimal (if we discard re-visited states, we might miss a better solution)
- A*:
  - complete: complete, if we do not discard re-visited states (however, this could lead to search trees exponential in size)
  - optimal: optimal, if we do not discard re-visited states (however, this could lead to search trees exponential in size)

We can discard DFS and Best-First, since they do not deliver an optimal result. Iterative-Deepening is preferred over BFS and Uniform-Search since it has better complexity estimates. So we have to choose between A* and Iterative-Deepening. The evaluation of A* highly depends on the used heuristics. - If the used heuristics is admissible (but not consistent), we must not discard revisited states. Then, A* would be complete and optimal. The search tree might be exponential in size, though. In the scenario of a admissible heuristics, I probably would prefer Iterative-Deepening to avoid exponential search trees. - If the used heuristics is consistent, we can handle revisited states much more efficient. Specifically, we do not suffer from the risk of exponential search trees. In this scenario, I would prefer A* because it uses a heuristic to estimate which node to expand next, while Iterative-Deepening simply expands the first node in the frontier.

2. Heuristics

2.1. Definitions

- *Admissible heuristics* are estimates of the cost of reaching a goal state from a given state that are always lower or equal to the actual cost. In other words, they never overestimate the cost of reaching a goal.
- *Consistent heuristics* are a type of admissible heuristics where the estimated cost of getting from one state to another plus the estimated cost of reaching the goal state from the next state is always less than or equal to the actual

cost of getting from one state to the goal state. It ensures that the estimated cost of reaching the goal state is always in line with the actual costs along the path.

2.2 When to use?

- If a heuristic is admissible, it means that it is "optimistic" in nature, which means that the algorithm is able to find the optimal solution of the problem. If a heuristic is not admissible, it overestimates the costs and might not detect the best path to the goal state.
- If a heuristic is consistent, then whenever A* expands a node, it has already found an optimal path to this node's state. Thus we do not need to track re-visited states because upon expansion of a node we've already found the optimal path. This eliminates the risk of search trees exponential in size, if A* is forced to not discard revisited states.

2.3 How to construct?

To construct admissable heuristics, we should consider a relaxed problem at each node. Hence, to solve the problem in a given state, we should simplify it and estimate, how many steps we need to solve the simplified problem. This is generally assumed to be an underestimate (and therefore an admissable heuristics), since the reduced complexity usually also reduces the costs to reach a goal node.