159.372
Intelligent Machines
Assignment 4 Report

Chris McDonald
20009360

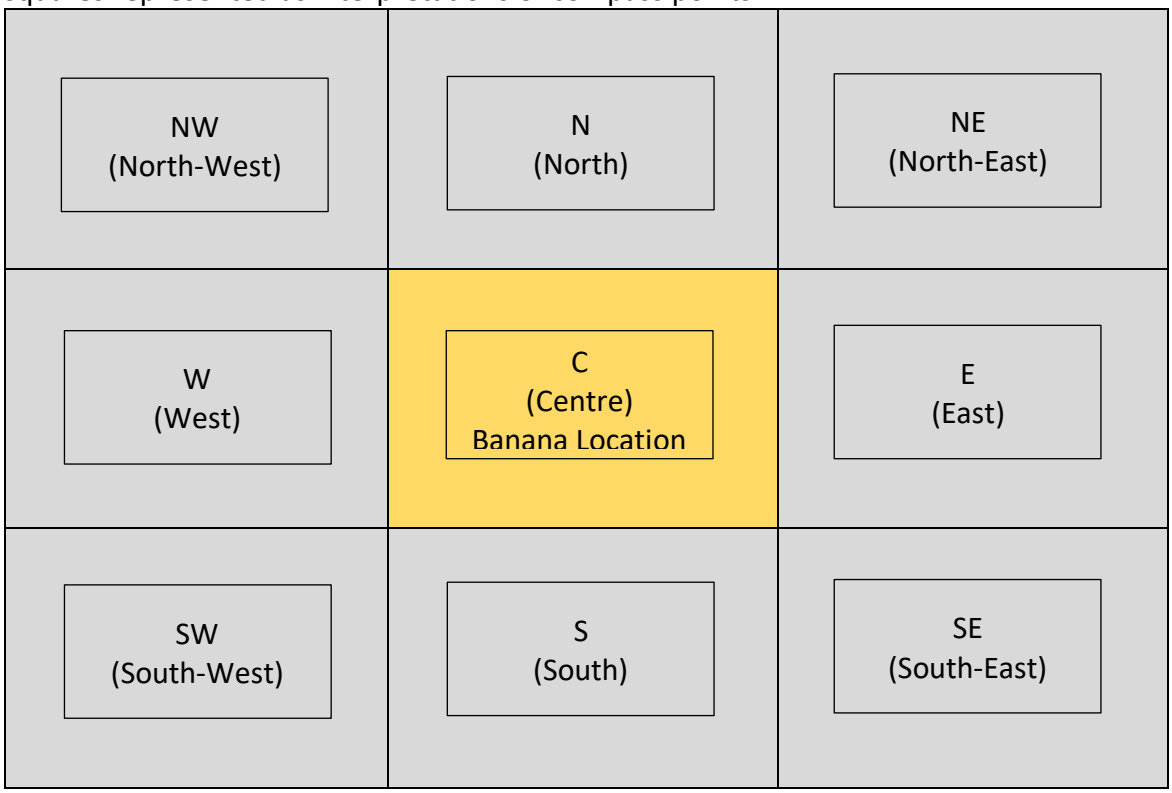
Chris McDonald - 20009360

Part 1:

The STRIPS planner I used for this assignment was the *stripsForwardPlanner* from AIPython, utilising an A* search with multi-path pruning.
I used the forward planner rather than the regression planner primarily due to computational effort related to my second problem. The first problem saw no significant difference in the runtime or number of paths explored in either the forward or regression planners. But the second problem (of which there is only one valid solution), saw a significant difference in the number of paths searched to find the solution. For the forward planner, there was typically only around 55-60 paths that needed to be explored before the planner found the solution. But for the regression planner, it took on average around 650-655 paths that needed to be searched in order to find the solution. So, for this reason, I elected to go with the forward planner for this assignment.
Due to the nature of the problems I chose, I didn't have to add any additional functionality to the forward planner to tailor it to my problems. I simply formatted both problems to fit with the predefined parameters for the forward planner.

Part 2:

The first problem I chose for this assignment was the **Monkey-Banana** problem. The scenario is as follows: a monkey enters a room that has a banana hanging from the ceiling in the centre of the room, and a stool placed randomly somewhere in the room. He cannot reach the banana without the stool, so must find the stool, drag it to the centre of the room and stand atop it in order to reach the banana.
In my interpretation of this problem, I envisioned the room as a 3x3 grid, with each of the 9 squares represented as interpretations of compass points.



The monkey and the stool would randomly spawn in any of these squares (possibly even in the same one) but regardless, the monkey would always have to take the stool to the centre square to get the banana. The monkey can only move one square at a time and cannot travel diagonally. Also, all movements have an equal cost.
The STRIPS features I used for defining the domain of the problem were the following:

- Monkey Location (*MLoc*): To determine which area of the room the monkey is in. Is represented by the symbols for each room (NW, C, SE etc.)
- Monkey Has Stool (*MHasStool*): A Boolean to signify if the monkey is in possession of the stool or not.
- Monkey On Stool (*MOnStool*): A Boolean to signify if the monkey is standing on the stool or not.
- Monkey Has Banana (*MHasBanana*): A Boolean to signify if the monkey has grabbed the banana. This Boolean is also the goal state for the problem when it becomes True.

The initial state of the problem had all the Boolean features set to false and the *MLoc* set to one of the squares, chosen randomly.
There were 5 main classes of actions in this problem:

- Monkey movement actions (*without stool*) : These actions were used when the monkey would move from one square to the next. Only the *MLoc* feature would be changed during these actions to signify the new square it was occupying. Each square would have an movement action for every possible path from that particular square and these would be shown in the action's name.
- Grabbing the stool: Once the monkey was occupying the same square as the stool, it could make an action to grab the stool, which would set the *MHasStool* Boolean from False to True.
- Monkey movement actions (*with stool*): These actions were used when the monkey would move from one square to the next while in possession of the stool. It works the same as the previous movement actions, but these have the added precondition that the *MHasStool* Boolean must be set to True. The action name would also specify that the stool is being moved, so it was necessary to distinguish these 2 movement action sets.
- Climbing the stool: Once the monkey had dragged the stool to the centre square, it now satisfied the precondition for being able to stand on the stool. The precondition was set so that climbing the stool was only possible from the centre square, where the banana is. This means the *MOnStool* Boolean would now be set to True.
- Grabbing the banana: Once all the preconditions were met (*MLoc*=C, *MHasStool*=True, *MOnStool*=True), the monkey is now able to take the action of grabbing the banana and thus achieve the goal state of *MHasBanana*=True.

These actions were used as they covered all aspects of the problem. All possible movement actions were covered, making sure to distinguish between movement with and without the stool. Actions to change Boolean values were also necessary to ensure the parameters of the problem were not broken and the rules were adhered to.

The second problem is the **Wolf, Goat & Cabbage** problem (this problem has many different names and variations), which is an old river crossing problem. The scenario is as follows:
A farmer is trying to transport a wolf, a goat and a cabbage across a river on his boat. Due to the small size of his boat, he can only take one of his items over the river at a time. But he cannot leave the wolf and the goat alone together, otherwise the wolf will eat the goat. He also cannot leave the goat and the cabbage alone together, as the goat will eat the cabbage. So he needs to figure out a way to transport all 3 across the river without any casualties. It should also be noted that the farmer can transport the items in both directions across the river.
This problem is interesting in that there is only one valid solution that the planner can find. Due to the higher complexity of this problem, it required many more features for setting up the domain than the previous problem. Here are the ones I selected:

- Boat/Wolf/Goat/Cabbage Location (*BoatLoc,WolfLoc,GoatLoc,CabLoc*): It was necessary to know the location of all components of this problem at all times, so the above 4 features could either have the value of *LeftBank* (the starting/initial bank) or *RightBank* (the goal bank).
- Goat & Cabbage together (*Goat&CabbageTogether*): This Boolean was used to check that the goat and the cabbage weren't left alone together on the same riverbank. This was a commonly used precondition to ensure that the rules of the problem were not broken.
- Wolf & Goat together (*Wolf&GoatTogether*): This Boolean, like the one above, was a widely used and necessary precondition to ensure that the wolf and goat weren't left alone on the same riverbank, thus voiding the rules of the problem.
- Boat is Empty (*BoatEmpty*): A Boolean to check if the boat is empty (apart from the farmer).
- Wolf/Goat/Cabbage on the boat (*WolfOnBoat,GoatOnBoat,CabbageOnBoat*): Booleans that signify if the items are on the boat or not.

The initial state of the problem had the location of all 3 items and the boat on the LeftBank, the *Goat&CabbageTogether*, *Wolf&GoatTogether* and *BoatEmpty* Booleans set to True and the remaining Booleans set to False.
The goal state was simply to have the *WolfLoc, GoatLoc* and *CabLoc* features all set to 'RightBank', signifying that all 3 items had crossed the river safely.
The actions for this problem are divided into 3 main categories:

- Boat Crossing to LeftBank/RightBank: A action to signify the boat moving from one side of the river to the other. It would simply change the *BoatLoc* feature to the opposite riverbank.
- Getting on the boat: For each of the riverbanks, the wolf, goat and cabbage all had their own actions for getting on the boat. The preconditions would ensure that both the boat and the particular item were on the same riverbank and that the boat was empty (*BoatEmpty*=True). Additionally, for the wolf, it was required that the *Goat&CabbageTogether* Boolean was set to False, since having the wolf board the boat and leave the other 2 on the same riverbank would break the rules of the

problem. The same thing was done for the cabbage, where the *Wolf&GoatTogether* Boolean had to be False to satisfy the precondition for boarding.

- Getting off the boat: This set of actions are far more complex than the previous two. It was necessary to have multiple actions for the same movement, depending on the locations of the other items.
  For all of these actions, the prerequisites required that the *BoatEmpty* Boolean was False (indicating that there was something on the boat the needed to disembark), the specified item was on the boat (*Wolf/Goat/CabbageOnBoat*=True) and that the location of the boat was correct.
  From here, there would be actions with the same name that would have different prerequisites based on the locations of the other items. For example, the wolf getting off the boat on the RightBank has one action for if the Goat is on the LeftBank and another for if the goat is on the RightBank. For both, the *WolfLoc* feature is changed to 'RightBank', *BoatEmpty* is set to True and *WolfOnBoat* is set to False. But if the goat is on the RightBank, the *Wolf&GoatTogether* feature will be set to True, which doesn't occur if the goat is on the other bank.
  So for each riverbank, the wolf and the cabbage both have two 'getting of the boat' actions with the same name, and the location of the goat determines which of the two actions is chosen. But because the goat is involved in both the preconditions for the other 2 items, the action of the goat getting off the boat is split into 4 different ones based on the possible location combinations of the other two items.
  To best illustrate this, I've included two screen shots below. The first is an example of the wolf's disembarking actions and the second is the goat's disembarking actions, both for a single riverbank.

```
Strips('Wolf off boat at RightBank',{'BoatEmpty':False,'WolfOnBoat':True,
                                     'BoatLoc':'RightBank','GoatLoc':'LeftBank'},
       {'WolfLoc':'RightBank','BoatEmpty':True,'WolfOnBoat':False}),
Strips('Wolf off boat at RightBank',{'BoatEmpty':False,'WolfOnBoat':True,
                                     'BoatLoc':'RightBank','GoatLoc':'RightBank'},
       {'WolfLoc':'RightBank','BoatEmpty':True,'WolfOnBoat':False,'Wolf&GoatTogether':True}),
```

```
Strips('Goat off boat at RightBank', {'BoatEmpty': False,'GoatOnBoat': True,
                                      'BoatLoc': 'RightBank','WolfLoc':'LeftBank','CabLoc':'LeftBank'},
       {'GoatLoc': 'RightBank', 'BoatEmpty': True, 'GoatOnBoat': False}),
Strips('Goat off boat at RightBank', {'BoatEmpty': False,'GoatOnBoat': True,
                                      'BoatLoc': 'RightBank','WolfLoc':'RightBank','CabLoc':'LeftBank'},
       {'GoatLoc': 'RightBank', 'BoatEmpty': True, 'GoatOnBoat': False,'Wolf&GoatTogether':True}),
Strips('Goat off boat at RightBank', {'BoatEmpty': False,'GoatOnBoat': True,
                                      'BoatLoc': 'RightBank','CabLoc':'RightBank','WolfLoc':'LeftBank'},
       {'GoatLoc': 'RightBank', 'BoatEmpty': True, 'GoatOnBoat': False,'Cabbage&GoatTogether':True}),
Strips('Goat off boat at RightBank', {'BoatEmpty': False,'GoatOnBoat': True,
                                      'BoatLoc': 'RightBank','CabLoc':'RightBank','WolfLoc':'RightBank'},
       {'GoatLoc': 'RightBank', 'BoatEmpty': True, 'GoatOnBoat': False,
        'Cabbage&GoatTogether':True,'Wolf&GoatTogether':True}),
```

This problem required a lot of actions to ensure that the rules of the problem were adhered to. The locations of the items in relation to each other effects how the problem is solved, so it was necessary to include actions for all possible outcomes.

Part 3:

*Note: A full file of console output can be found in the **Output.txt** file submitted with this report.*

To get clear console output when running the planner, it required reworking some of the print statements in the *searchMPP* file.

```
while not self.empty_frontier():
    path = self.frontier.pop()
    if path.end() not in self.explored:
        self.display(2, "Expanding:",path,"(cost:",path.cost,")")
        self.explored.add(path.end())
        self.num_expanded += 1
        if self.problem.is_goal(path.end()):
            self.display(1, self.num_expanded, "paths have been expanded and",
                    len(self.frontier), "paths remain in the frontier")
            self.solution = path   # store the solution found
            print('*SOLUTION FOUND*')
            p = str(path).split('}')
            print('*INITIAL STATE*: '+p[0]+'}')
            print('*BEST PATH*')
            count = 1
            for i in p[1:len(p)-1]:
                print(str(count)+') -> '+i+'}')
                count += 1
            print('-----------------------------------------------------------------------')
            return path
        else:
            l = ''
            pa = str(path).split('}')
            for x in pa[1:len(pa)-1]:
                xx = x.split('{')
                xxx = xx[0]
                xxxx = xxx+'\n'
                l = l+xxxx
            if l:
                print('*PATH EXPLORED*')
                print(str(l))
            neighs = self.problem.neighbors(path.end())
            self.display(3,"Neighbors are", neighs)
            for arc in neighs:
                self.add_to_frontier(Path(path,arc))
```

This would provide clarity to the statements outputted by the console and also show all paths explored by the planner while it searches to find the solution to the problems.
Here are some excerpts from a runtime trace of the **Monkey-Banana** problem:

```
*MONKEY-BANANA PROBLEM*
MONKEY LOCATION: NE
STOOL LOCATION: SE

*PATH EXPLORED*
Move NE->N

*PATH EXPLORED*
Move NE->E

*PATH EXPLORED*
Move NE->E
Move E->C
```

…….

```
*PATH EXPLORED*
Move NE->E
Move E->SE
Grab-Stool
Move-Stool SE->E
Move-Stool E->C
Climb-Stool

15 paths have been expanded and 0 paths remain in the frontier
*SOLUTION FOUND*
*INITIAL STATE*: {'MLoc': 'NE', 'MHasStool': False, 'MOnStool': False, 'MHasBanana': False}
*BEST PATH*
1) -> Move NE->E{'MLoc': 'E', 'MHasStool': False, 'MOnStool': False, 'MHasBanana': False}
2) -> Move E->SE{'MLoc': 'SE', 'MHasStool': False, 'MOnStool': False, 'MHasBanana': False}
3) -> Grab-Stool{'MLoc': 'SE', 'MHasStool': True, 'MOnStool': False, 'MHasBanana': False}
4) -> Move-Stool SE->E{'MLoc': 'E', 'MHasStool': True, 'MOnStool': False, 'MHasBanana': False}
5) -> Move-Stool E->C{'MLoc': 'C', 'MHasStool': True, 'MOnStool': False, 'MHasBanana': False}
6) -> Climb-Stool{'MLoc': 'C', 'MHasStool': True, 'MOnStool': True, 'MHasBanana': False}
7) -> Grab-Banana{'MLoc': 'C', 'MHasStool': True, 'MOnStool': True, 'MHasBanana': True}
```

So after displaying both the Monkey's start location and the location of the stool, the planner runs and shows every path explored by displaying the name of the action undertaken. This continues until the solution is found, then both the problem's initial state and the solution/best path are printed to the console (showing the order of actions taken then the name of the action, followed by the state once that action has occurred).

Now I'll show some excerpts from a runtime trace of the **River-Crossing** problem:

```
*RIVER CROSSING PROBLEM*
*PATH EXPLORED*
Goat on boat at LeftBank

*PATH EXPLORED*
Boat travelling to RightBank

*PATH EXPLORED*
Goat on boat at LeftBank
Boat travelling to RightBank
```

……

```
57 paths have been expanded and 4 paths remain in the frontier
*SOLUTION FOUND*
*INITIAL STATE*: {'BoatLoc': 'LeftBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'LeftBank', 'CabLoc': 'LeftBank'
*BEST PATH*
1) -> Goat on boat at LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'LeftBank', 'CabLoc
2) -> Boat travelling to RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'LeftBank', '
3) -> Goat off boat at RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'RightBank', 'Ca
4) -> Boat travelling to LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'RightBank', 'Ca
5) -> Cabbage on boat at LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'RightBank', 'Ca
6) -> Boat travelling to RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'RightBank',
7) -> Cabbage off boat at RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'RightBank',
8) -> Goat on boat at RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'RightBank', 'Cab
9) -> Boat travelling to LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'RightBank', 'Ca
10) -> Goat off boat at LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'LeftBank', 'CabL
11) -> Wolf on boat at LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'LeftBank', 'CabLo
12) -> Boat travelling to RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'LeftBank', 'GoatLoc': 'LeftBank',
13) -> Wolf off boat at RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'RightBank', 'GoatLoc': 'LeftBank', '
14) -> Boat travelling to LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'RightBank', 'GoatLoc': 'LeftBank', '
15) -> Goat on boat at LeftBank{'BoatLoc': 'LeftBank', 'WolfLoc': 'RightBank', 'GoatLoc': 'LeftBank', 'CabL
16) -> Boat travelling to RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'RightBank', 'GoatLoc': 'LeftBank',
17) -> Goat off boat at RightBank{'BoatLoc': 'RightBank', 'WolfLoc': 'RightBank', 'GoatLoc': 'RightBank',
----------------------------------------------------------------------
```

Just like the Monkey-Banana problem, the planner prints out the names of all actions taken in every path searched before finally displaying the final solution actions once found.

In terms of the comparative complexity of the problems, the Monkey Banana problem is the less complex of the two. No matter where the monkey and stool are placed, the final solution will always have less actions taken than the 17 required to solve the River-Crossing problem. It also requires less paths to be explored and has less features, actions and action prerequisites than the River-Crossing problem.