# CS3105 Practical 1: Late Binding Solitaire Report

## **Overview**

For this practical I was asked to implement search algorithms for the game Late Binding Solitaire (accordion solitaire), then test and evaluate the code. I have managed to implement the code fairly well however I was unable to fully test and evaluate the code due to time constraints.

## ***Part 1 – Checking Solutions***

### Design

As with almost all the code in this practical, this section was just built on top of the starter code. First the input needs to be checked to make sure it is in the correct format, whilst the starter code does some of this, more filtering was needed. The program loops through every second item in the list instead of every item in the list of inputs as moves are given in a set of 2 integers: the card being moved and the card in the destination.

So, first in the loop simply for readability and efficiency's sake several variables are declared to represent the card the solution wants to move, what pile it's being moved to and the card at the target destination. The last one is declared after the first check because the first checks that the given card and card destination are within the bounds to be a valid card and a valid pile number given the current layout. If not, the program prints false and quits as the second a bad input is given, then the puzzle cannot be solved.

Then the program checks the card to be moved is actually in the current layout and the card destination is in the current number of piles and the card destination is either one space away or three to the spaces away. If so, then the program loops through to check if the card can be moved there with the rules of the game I.e., the card is either the same rank or suit. To accomplish this the program loops through the number of suits, using the index as a sort of benchmark to tell the upper and lower bounds of each suit since there can be a variable number of suits and ranks. It also checks if the cards are of the same suit by dividing them by the number of ranks and since they are integers then they should be the same whole number if they are within the same suit. If the card is within the same rank or suit then the loop quits, the move is legal and thus is made. Otherwise, if the program loops through all the suits without finding a suit that both the card and card at the destination are within then the program outputs false and returns.

Finally, once all moves have been carried out (assuming they are all legal), then the program checks how many piles are left in the current layout, if the puzzle was solved there should only be 1 left. If so, the program outputs true and quits, otherwise, it outputs false.

## Testing

Due to time constraints the program could not be thoroughly tested as I could not get stacscheck run in time.

# ***Part 2 – Depth First Search***

## Design

For the actual solve portions of this practical a new class was created. The LBS State class is a simple class that holds the layout of a game, the moves to get to that state from the initial state and for the saving grace versions a Boolean representing if the saving grace had been used. Then a few simple constructors and getter/setter methods were made. The only one of note is the method that sets the saving grace Boolean to true, since it should never be reset to false unless it's a different state object.

Now for the actual solving part of the program. There were two generic approaches to take when designing a way to solve the problem: iterative and recursive. I decided to go with iterative mainly due to the fact this was being programmed in java which while it technically supports recursion, it can often be incredibly intensive and the number of states and data that would be generated the program would likely have a stack overflow and not work due to the way java stores things when doing recursion.

The basic design of the algorithm is from the lecture slides (https://studres.cs.st-andrews.ac.uk/CS3105/Lectures/L03-Search-1.pdf Slides 31 & 32). Since this is depth first the list storing the states was actually a stack. A stack of state objects is created, creating a null state/initial state object that is pushed to the stack. Also, a Boolean variable is created to improve the efficiency of the algorithm as when a solution is found the loop will quit and the output will print.

Then the program loops until either the stack is empty, or a solution is found. The current state is then popped from the stack and the program loops through every card in that current state or until a solution is found. A temporary state is created so that the state being checked isn't changed as that would break the algorithm. The current card potentially being moved is then stored.

Then similarly to the solve method the program checks if the card can legally be moved one space or three spaces. Nested ifs are used to ensure that there is card within the one or three spaces respectively to ensure there are not out of bounds errors.

Then if the card can be moved to either one space or three spaces then the program makes that move and pushes the card being moved and the pile number its moved to, to the stack of moves in the state object. Then that state object is pushed to the search stack. Finally, once all the cards in a state have been checked the program checks that if that once the moves are made if the solution has been found I.e., there is only one pile left.

After that, the program repeats popping the next state off which should spawn more states in that branch so therefore following the depth first search algorithm.

Once the loop has finished the program checks if a solution were found, if not and the stack size is 0 then it is simply an impossible layout, but if the stack is not empty then an error has occurred and the program could not find a solution in time (I did not have time to program a way for the program to track when it has been too much time spent so this should not be able to happen). Otherwise, the program pops the winning state which should be at the top as it was just pushed and then prints out the size of the stack of moves and then prints them out one by one.

### Testing

Due to time constraints the program could not be tested as I could not get stacscheck run in time.

### Evaluation

Since testing could be performed the evaluation of this section is hard to do but I thought I would still discuss a few things. So, for this part of the program I believe for an iterative depth first search it is fairly efficient. While I believe that recursive would likely be faster, I was mainly worried about recursive restraints in java would lead to data issues. So, if this were done in other languages, I would have done a recursive model however since we had to stick to the starter code. Sadly, I believe there is a way to improve this by removing the nested loops used to check if the cards are within the same suit however, I do not believe I could figure it out in time. Another thing I could have done given more time would be a way to prevent repeat states as it would lead to shorter branches if they were unsuccessful.

## ***Part 3 – Depth First Search with Saving Grace***

### Design

The design for this is incredibly similar as it only really adds one element. Basically, the only difference from the non-saving grace solve is the program, when checking if a move is valid checks if the saving grace has been used and if not then creates a state where it is used. Then updates that in the state object to keep track of it.

### Testing

I sadly did not have time to test this part of the program due to similar reasons above.

### Evaluation

Now the evaluation is in a similar circumstances to part twos evaluation in terms of how I am conducting it and the points I made about the actual code itself there is one important thing to note. Mainly the saving grace mechanic of the game adds an insane number of states tanking the speed of the algorithm as it can be used at any point in the game spawning way more states which would not even be helped with a way to remember states as they would technically be different.

## Conclusion

To be honest I found this practical unexpectedly difficult. The starter code was incredibly difficult to wrap my head around, it would have been helpful to have more commentary on it. This caused part one to take way longer than expected. So, I had to crunch the last two parts. And due to the lab outage, I could not test part one, so I decided not to waste time and code parts two and three. Also, at the same time I had several personal issues that I will not discuss in the efforts to remain professional that made it increasingly difficult to work. I know this entire paragraph may come across as unprofessional regardless, however I feel it is important to explain why my practical is in this state and thus will hopefully allow a fairer grade.