Spec: https://studres.cs.st-andrews.ac.uk/CS3105/Practicals/P1/CS3105-AI-P1-BlackHole.pdf

### Overview

This practical focused on implementing a search algorithm that could attempt to solve variants of the game Patience (Solitaire), known as Black Hole and Worm Hole. The specification came as three distinct parts, the first one being a checker function which checked if a given solution for a puzzle was valid, and then the search implementations for the two variants. On top of this, further testing and evaluation of the implementations was required, discussing the search algorithms in detail. The specification requirements were all fully realised, with part 1, 2 and 3 all implemented. See the check table at the bottom of this report for more detail into the project completion.

## Design & Implementation

**Part 1**

The first requirement was the implemenation of the checker function. Before writing any code, I studied the practical specification and made sure I completely understood the input format that the checker should be given. The input, which was a list of integers was to be processed as pairs (one for the pile, one for the card) so it naturally lead to a simple iterative solution which would loop up until we see no more pairs. The thought process was simply to follow the steps of the puzzle and check once we are out of pairs, if we have found a valid solution.

When initially implementing, a few helper functions were required to step through the game. The BHLayout class especially had to be extended to support the new Checker class. These were fairly straightforward however, the algorithm to calculate the card rank of a card proved a slight difficulty. As the cards are numbered 1 to the number of cards in the deck (52 for a normal deck), in order to retrieve the rank we have to slice the deck up into its suits and locate the rank from these ‘slices’. The Checker class was designed to be instantiated as an object which had a BHLayout member. This layout member would be altered throughout the execution to represent the game at different states. The instances provided in the Tests folder don’t care if we don’t have every card that is in the deck present in the piles so we can simply check if we have solved the game by making sure every pile is empty.

Testing worked concurrently throughout development with stacscheck but I also started implementing a Junit framework to test the Checker class’ individual functions to make sure everything worked appropriately.

**Part 2**

For part 2, a new Solver class was created. Different approaches could be taken with depth-first search. I started with a list-based implementation which was described in the week 2 lectures here: https://studres.cs.st-andrews.ac.uk/CS3105/Lectures/CS3105-L02-Search-

1.pdf. It's a very straightforward implementation which worked perfectly for this problem. It simply loops until we have no more game states, picking off the first element in the list and expanding it, adding any new nodes we find to the front of the list (i.e. merging) . Some things I had to think about included how I was going to store nodes in the abstract search tree. To avoid cluttering the BHLayout class I created a new custom type called Node which stored a layout instance and an array list of a new type, Tuples. This list was a 'log' of all the moves made so far to get to this node. This came about because of the output requirement of part 2 which required outputting the step-by-step solution to the game. This decomposed the problem significantly and made it easier to manage as a whole.

Actually implementing this algorithm lead to more tweaks and ehancements to the overall program. A lot of functions and variables are reused across the project such as the Checker functions which were made public and static, allowing the Solver to use them freely. This was important particularly when expanding nodes to new game states. New game states will come about by finding an element off the top of a pile that we can move to the hole card if they are adjacent in rank. The search algorithm will have exhaustively searched the tree if we find no solution and the list is empty.

After running stacscheck it appeared it timed out on a few of the tests. I inspected the Solver class and added Junit tests for the program, making sure each component worked correctly. Actually designing Junit tests forced me to decompose the Solver class into smaller functions to test, leading to cleaner written code. To ehhance the Solver I implemented it so that it would discard nodes identical to the ones we have seen. This actually brought about a lot more work than originally thought. To directly compare the nodes I had to override the equals and hashcode behaviour for Node, Tuple and BHLayout, which Junit tests were promply added for. Once this was working, to implement this feature, I used a HashSet type which could easily detect any duplicate nodes. Thoroughly testing and decomposing the algorithm lead to it passing the stacscheck tests with further enhancements.

**Part 3**

Immediately I could tell from reading the specification that the worm hole variant would lead to this algorithm discovering and expanding a lot more game states than the Black hole variant since we can freely move any card into the worm hole card if its empty. First, I started at the problem of storing this new game layout with the worm hole card. It seemed appropriate to save time on extra coding and testing to extend the BHLayout class with an added member for the worm hole. I also noticed that some behaviours of the functions will need to change such as the topCard() function which needs to detect if the pile value given is -1 which represents the worm card. In addition to this, the equals function would need to be overidden to accommodate the news worm hole card.

The designed algorithm is very similar to that of the black hole solver. I decided to include both solving algorithms in the same class. This could be very easily managed with a boolean member that described whether our puzzle was a worm hole variant or not. The algorithm now needs to expand a node in 3 different ways. Along with the way of placing a card on the

hole card, we can place the top card of any pile into the worm hole, or move the worm hole to the hole card if they are adjacent. If we accommodate for these methods of creating new nodes and use instances of WHLayout, the stacscheck tests passed for all worm hole tests. Thorough Junit testing was then implemented for the WHLayout class and worm hole puzzle solving.

**Further Implementation (Code)**

The list-based implementation worked really well and passed all the tests but I wanted to compare this with a recursive algorithm. The design was very similar to the list-based implemenation in terms of how it expanded nodes but logically worked a little different due to it's recursive nature. See the testing and evaluation section on how these algorithms compared.

Small enhancements were made to the searching algorithms including the option to completely search the tree even if we find one or more solutions, a custom time out was added to the search algorithms which was continuously altered throughout testing but is set at 60s which seemed appropriate for giving the algorithm plenty time but also not hanging up any tests that were carried out. A lot of these additions contributed to testing such as counting the number of states visited which was added along with calculating the elapsed time of a search.

The project structure was also organised into packages. This actually allowed me to group up classes which contained similar behaviour and/or had similar purposes in the project. E.g. Node and Type were both new types for the project so were packaged together, along with the game layout classes in a new layouts package.

The program doesn't check for any class casting exceptions which is one bug/letdown of the program. It requires users to comply with how the functions are used and what is passed to them to avoid this, and ideally any interaction with the program would be through BHMain.

A Makefile was also added which can be used to build and run tests for the program easily through the command line, along with the build.sh script which was changed to use the Makefile.

## Testing

Testing took multiple forms throughout development; stacscheck, command line execution using 'Java BHMain' and Junit tests in Intellij. Junit tests focused mainly on testing the individual components of the classes created to make sure they all worked as singular functions but also when combined as a whole, while stacscheck was used to test the correctness of the algorithms. If I encountered any errors or inconsistent results I used the command line to focus on a particular input and check the output of the program.

Stacscheck tests were quite comprehensive including input tests with empty piles, logically impossible puzzles and puzzles with solutions. I added a couple input tests which included

having a card in a pile larger than that of the deck, a negative card number and negative values for ranks, suits and pile input all which should produce no solution. More trivial cases with empty inputs and no piles were also tested which do produce solutions since it is already solved. One alteration that came about due to this testing was that when inputting a negative value for the number of piles, it attempted to initialise an ArrayList of size -1 which produces a exception.

To get around this, in BHLayout, I changed the function to default to pile size 17 in this case, which may produce a puzzle with no solution, based on the piles that are then input.

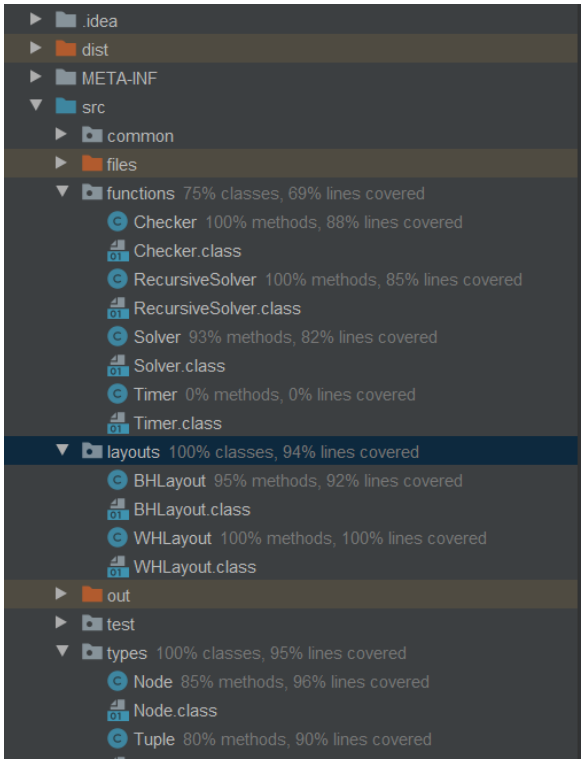
This combined with stacscheck shows that for the general input structure, the program can accept it, validate it and try and search for a solution correctly with the known inputs. However, more rigorous testing can always be applied.

One thing to note about the project structure is that the BHMain class sits in both the package ‘common’ and the src directory. This is to accommodate for both stacscheck and using the Junit framework. I struggled to find a way around this so just settled with two different copies. Javadoc was also used for any new classes or new functions added to existing classes.

Note: ‘make all’ will run stacscheck, Junit and the performance evaluation which may take a long time. Sometimes when running stacscheck the executable build.sh would refuse permission so this command was used: chmod u+x ./build.sh

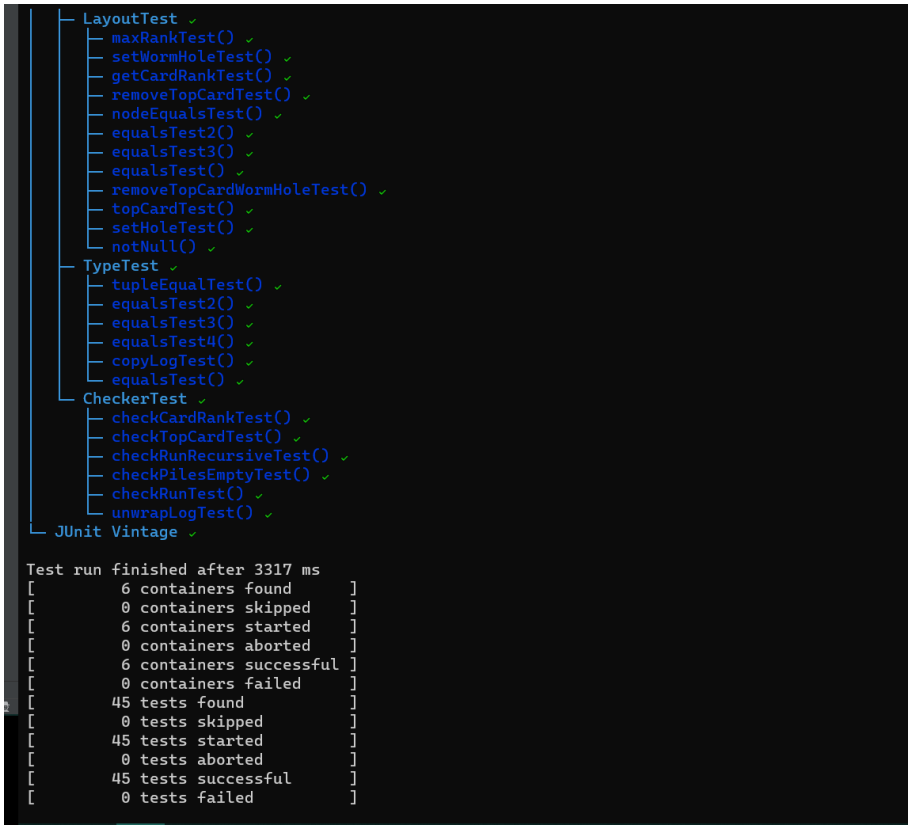
```
* COMPARISON TEST - Part3/WHSolve/prog-run-551-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolve/prog-run-651-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolve/prog-run-851-47-12-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolve/prog-run-951-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3/WHSolve/prog-run-951-51-13-4-2.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-01Solvable.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-1-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-1-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-1-5-3-2-1.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-1-5-3-2-2.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-101-5-3-2-1.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-151-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-251-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-301-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-351-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-401-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-51-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-551-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3/WHSolveAndCheck/prog-run-651-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-201-51-13-4-17.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-251-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-301-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-351-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-651-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-701-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-701-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-851-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-901-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveAndCheckHarder/prog-run-901-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-201-51-13-4-17.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-251-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-301-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-351-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-501-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-651-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-701-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-701-26-9-3-6.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-751-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-851-23-6-4-5.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-901-26-9-3-10.txt.out : pass
* COMPARISON TEST - Part3Hard/WHSolveHarder/prog-run-901-26-9-3-6.txt.out : pass
156 out of 156 tests passed
```

All stacscheck tests passed.




The Junit testing comprehensively tested function routines in the code. The coverage shown to the right is without the execution of the PerformanceTest class which generates random input for the puzzle solvers. This would likely lead to even more code coverage since a lot of lines not tested were exception handlers and states that could be reached with tailored input (e.g. causing a timeout).

Down below is a screen shot of the Junit tests passing using the Junit standalone command line artifact that is in the source directory.



# Evaluation

The evaluation of the algorithms was the main reason the elapsed time calculation and node count was added to the programs so we could compare time and space efficiency of both the different algorithms.

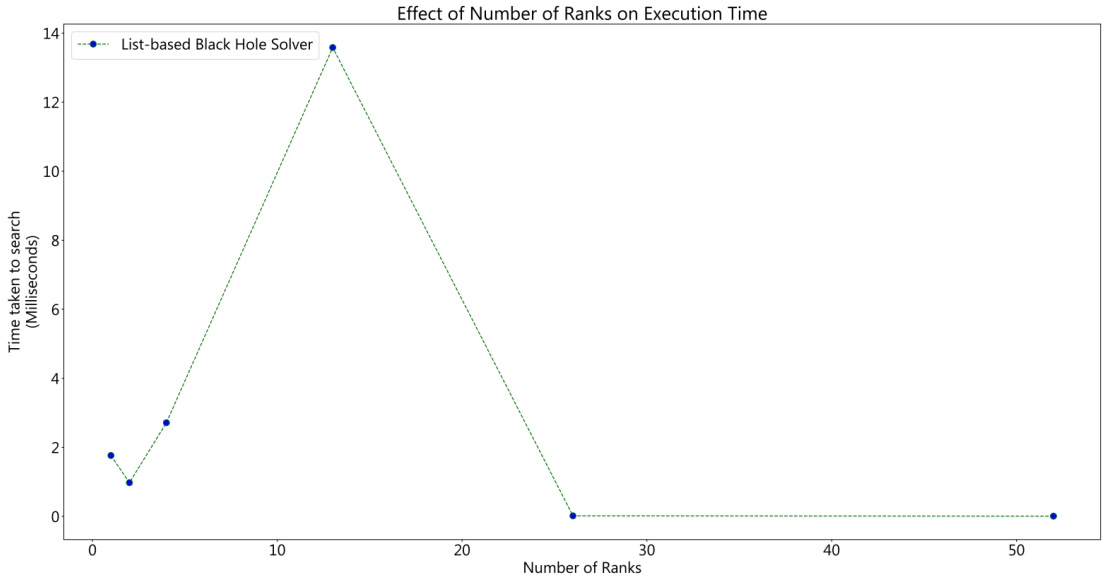
### List-based Black Hole Solver

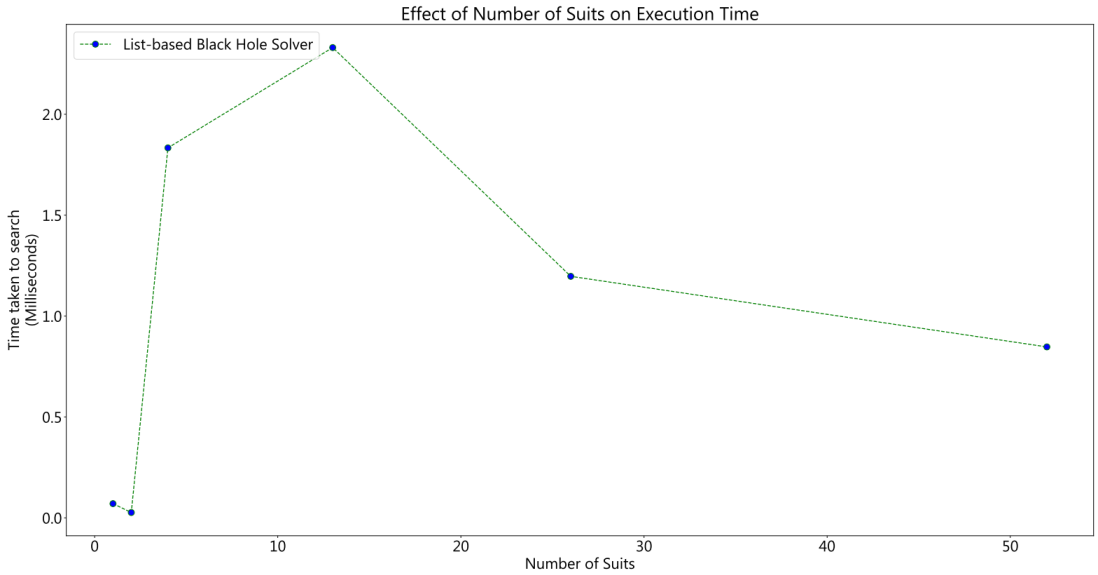
This algorithm works really well with the provided stacscheck tests as it doesn't take too long to find a solution for every test including the hard ones. I did further analyis with an additional testing class called Performance which runs using parameterised tests to test how the different permutations to the input (e.g. pile size, ranks etc.) affect the execution of the solvers with regard to time and space. Originally, this was represented through the command line as tables but I implemented a simple python program which could represent this data as graphs, making it a lot more readable. This also includes a new Writer class which simply writes the results of a solver to the appropriate file. This was actually based off of my CS2001 Week 10 practical which included graphing time complexities. For some tests a constant pile size of 10 was chosen in keeping with some of the stacscheck tests.

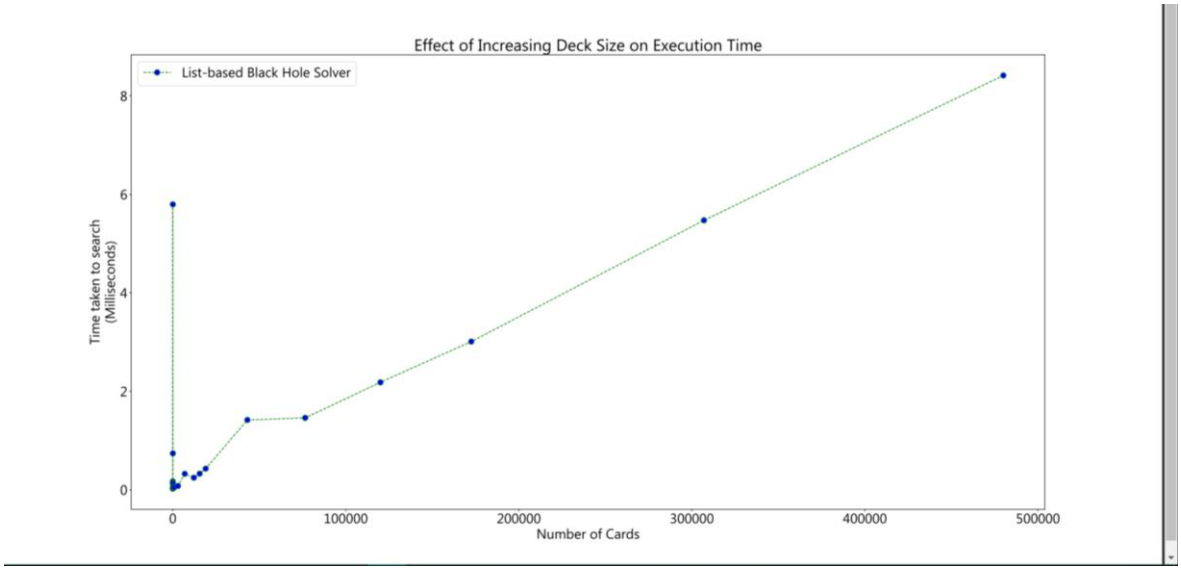
### Efficiency

Note: the graphs here may not be their final representation as the tests were subsequently carried out more times after reporting. They do however, share the same behaviour.

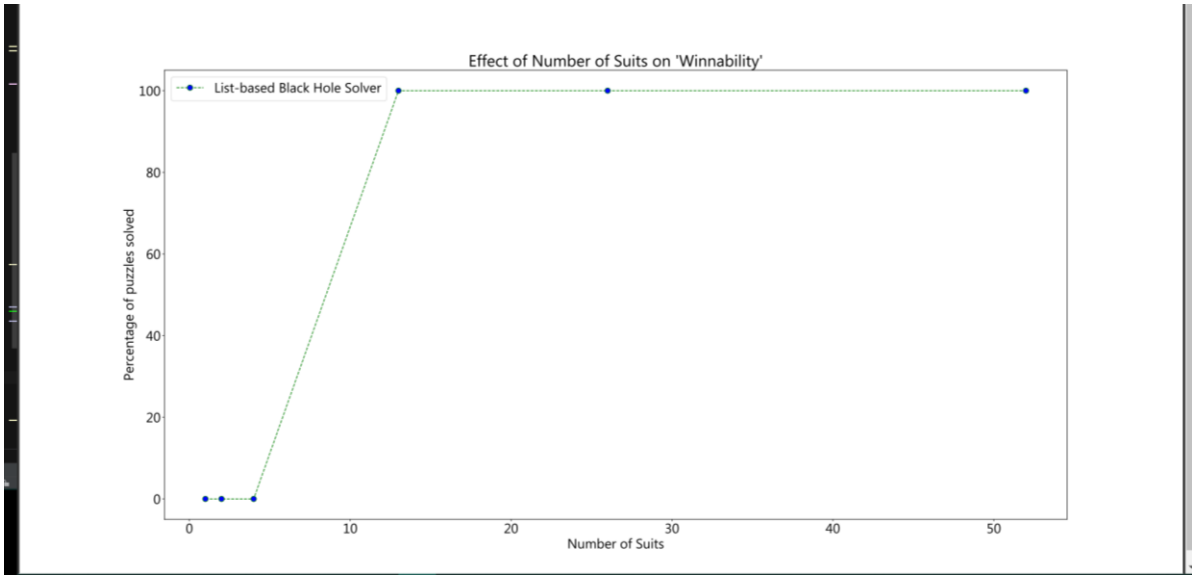
We have these three trivial cases of graphs:





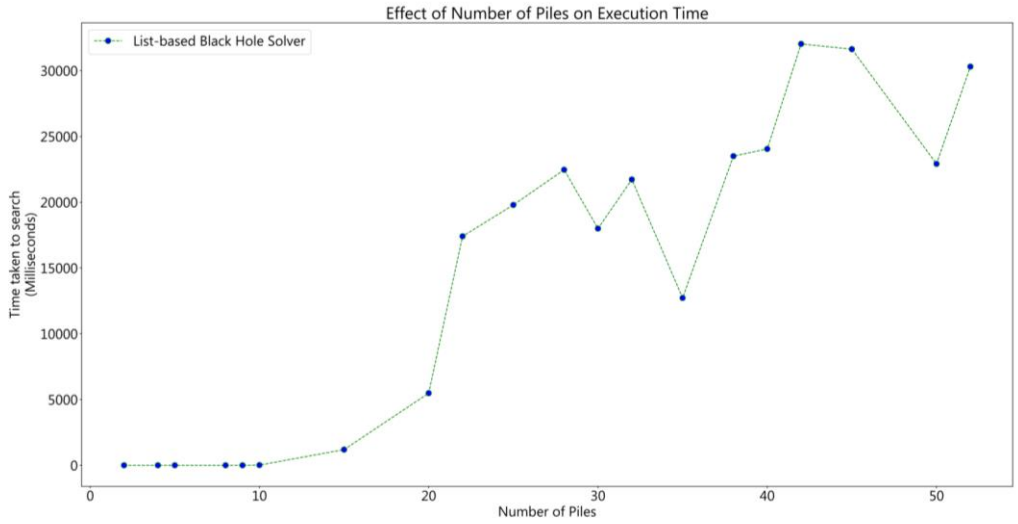

These three graphs show the time complexity when a) we increase the number of ranks in the deck, then the suits, then the deck size where a suit/deck ratio is kept in proportion. We can clearly see that when the ranks are 13, our solver takes longer time to search for a solution. This works in practise as if we increased the number of ranks across the deck then we decrease our suits, split across 10 piles it's increasingly likely we won't find the rank we are after on the top of a pile. If we increase the number of suits, we decrease the ranks. This is results in the converse of what happens with the ranks as we are more likely to find an adjacent rank and therefore can search the tree with more possible solutions. This is evident in the graph representing our chance of winnability with increasing suit:




We can see that our chance increases as we increase the suits.

If we take an increasing deck size with proportionate suits and ranks, hypothetically from our algorithm, it will now contain a lot more iterative cycles due to its programming (.e.g. finding a card's rank amongst thousands of cards) leading to a slower execution time. We can clearly see that from the graph which linearly increases in speed for an increase in deck size.

These are all trivial cases however, and if we take a normal deck of cards with the same proportion of decks and suits, what is more impactful on the winnability and execution time of the algorithm is the number of piles in the problem.



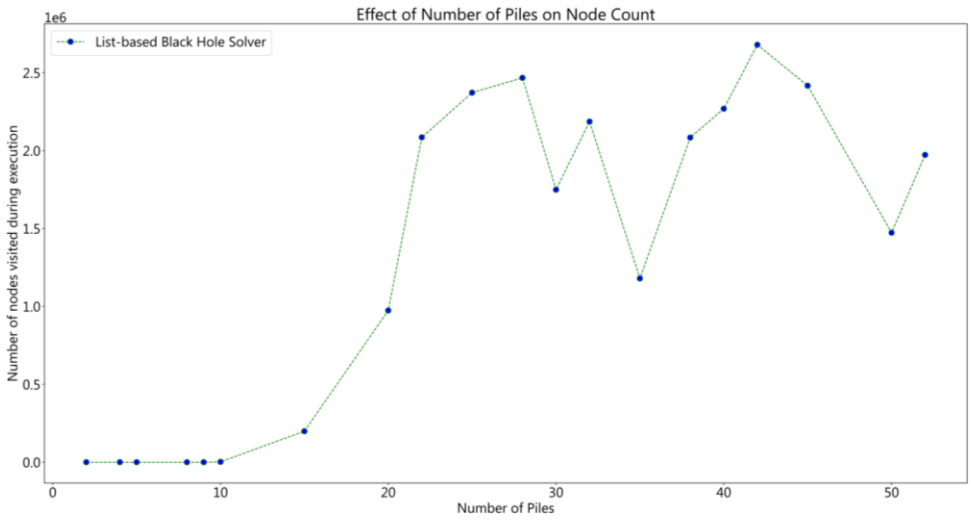

The pile size clearly has a large impact on the execution time of our algorithm. The jagged formation could be down to our use of random layout generation so in some instances it may be easier to solve than others. However we can see a clear rise in execution time as the

piles increase. If we look at the algorithm, the iterative cycles through the piles will contribute to this but also because the spread of cards is wider. For smaller piles, take 2 piles, we have 2 cards to look at. This decreases the probability of those 2 cards being adjacent to the hole so immediately the algorithm returns. If we then take 20, well we have more iterative cycles but also the probability of finding an adjacent card is higher so will lead to deeper and deeper search. We can assume then that the complexity of our algorithm is proportional to the pile size of a puzzle layout.

We can then conclude that since deck size is directly proportional to execution time along with the pile size, our algorithm will take an enormous amount of time (most likely time out with the 60s timeout mark) for a problem with a large deck size (into the thousands) and an increasing pile size (>= ½ of the number of cards in the deck).

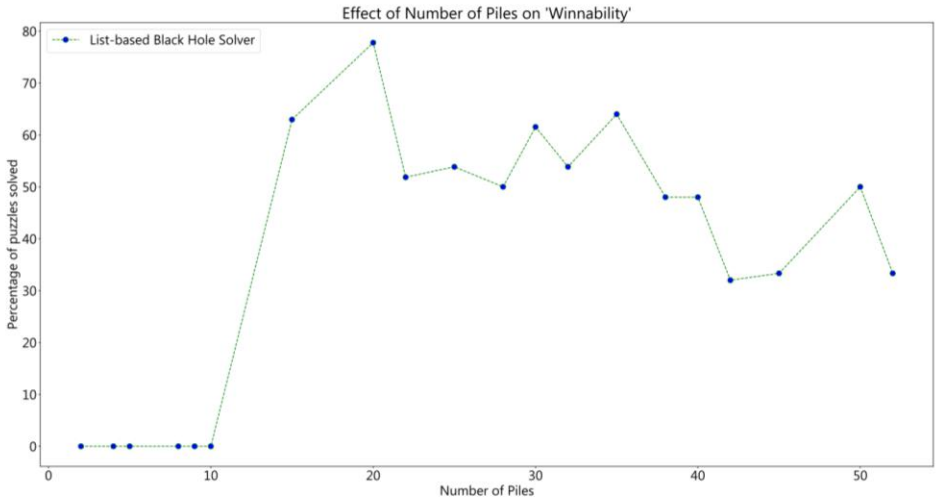
**Space**

It directly follows that if we spend more time looking at nodes, we are going to look at more nodes.



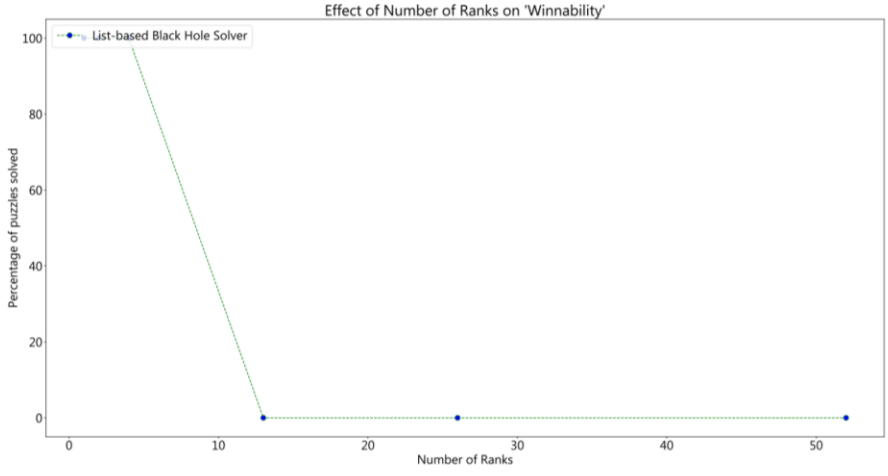

We can clearly see a rise in the node count for larger and larger amounts of piles in the puzzle.

**Winnability**




It must be noted that after debugging in Intellij, it was found that a lot of these tests timed out so the percentage of games that were solved may be even higher. We can clearly see an optimum number of piles for a deck of size 52. Around 20 piles gives us a great chance of finding a solution for any given puzzle. If we increase the size even further, we give ourselves a good chance but the execution time (see the graph of execution time) increases leading to the possibility of a time out. I propose that if we let the machine run for as long as required to run a complete search, it would discover there is a greater chance if we continue to increase the number of piles.

If we take 52 piles then surely we would have a solution every time since the cards are all on top. However, because we time out, we can't find out fully whether this is the case. This may be why the stacscheck tests opted for a pile size of 17 in some tests. From our graph, it gives approximately 80% chance of finding a solution, but also a smaller chance of a long execution time (or timing out).



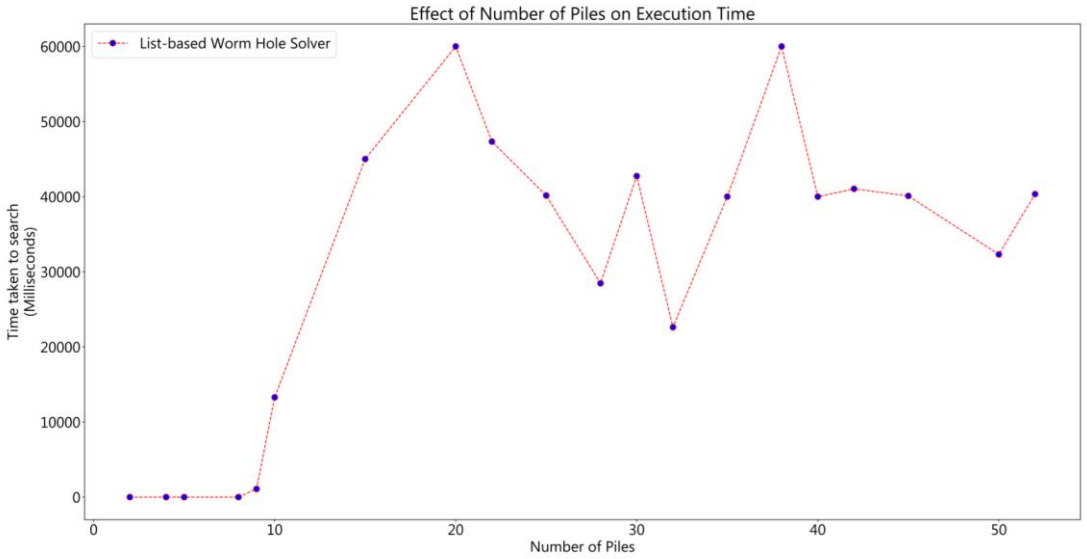

As discussed above, if we increase the number of ranks our chances of winning decreases due to the occurrences of any adjacent rank decreasing throughout the piles. We can see that clearly in the graph and that our best chance of winning is when the rank is 4. With our suits, since we increase the occurrences of adjacent ranks, the 'winnability' goes up.

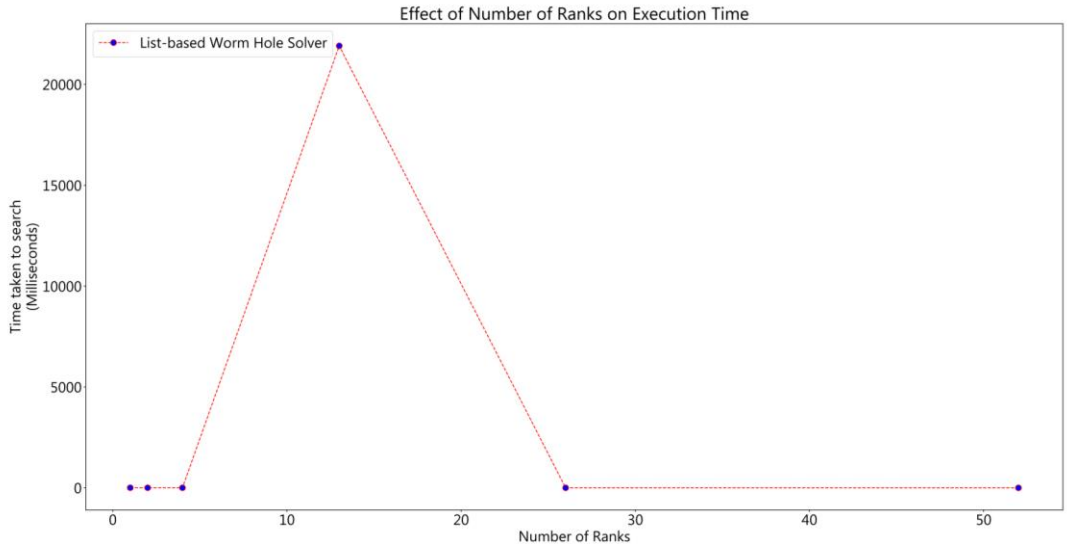
The algorithm I have implemented for solving Black Hole puzzles ahs a few drawbacks in terms of its iterative nature and how it is repeatedly searching and storing new nodes even if they don't get explored further. Out of the nature of this Patience variant, the algorithm works best and is suited for a deck size of 52 or at least having the correct deck proportion of ranks and suits. However, the main influence on the winnability, time and space of the algorithm is the pile sizes. A pile size of just less than half (17-22) for a deck of 52 cards seems to produce a higher chance of winning while also lending itself to not timing out as frequently as is the case with more piles.

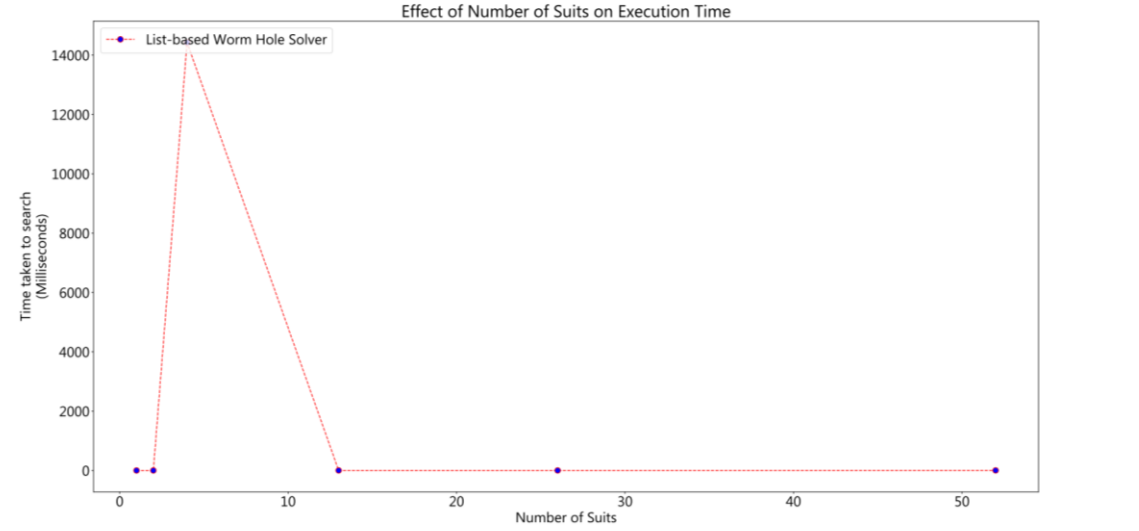
**List-based Worm Hole Solver**

Since the algorithm for solving Worm Hole variants discovers more unique states for every node, we can hypothesise that time and space complexity will increase dramatically in comparison to the black hole variant.

**Time**





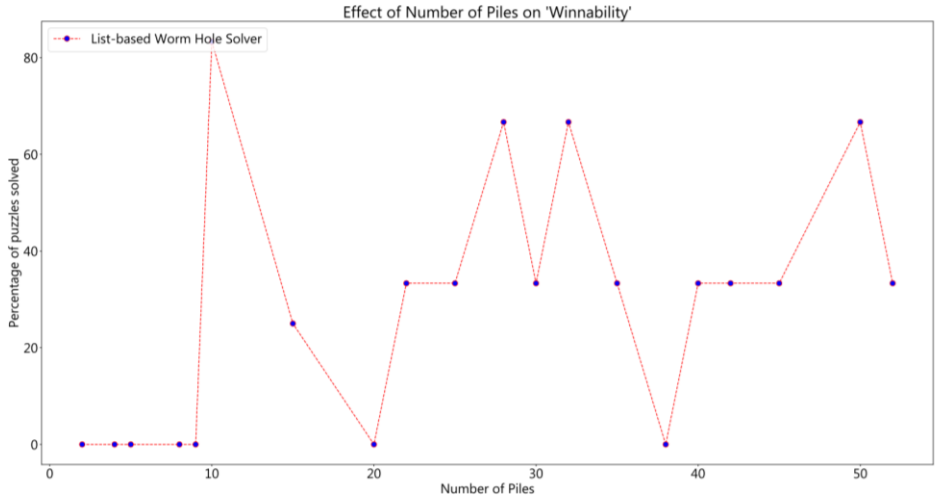




We are seeing the same behaviour as with Black hole searches whereby the time taken to solve dramatically rises for a deck of 52 cards when the suits are 4 and rank is 13. What's more important to us however, is the dramatic increase in time complexity when the number of piles increases. We are immediately timing out when we reach pile numbers of around 20 unlike in black hole searches. Because we have so much more to do for each state, if we increase the number of different permutations then we are of course going to increase execution time.

**Space**


Again as with the time complexity, we have a steep increase in node count around 20 different piles before the algorithm starts timing out.

**Winnability**




This is quite different from the 'winnability' of black hole. We now have a higher chance of finding a solution with 10 piles (which appears in stacscheck) than we do around 20. This, once again, is just before we time out. Because with worm hole we now have a lot more possibilities in terms of new states, we perhaps don't need as many piles to provide us with a higher chance of solving a given puzzle.

A direct comparison of the Worm Hole and Black Hole solvers maybe can't quite be fully made since one algorithm is clearly trying to do a lot more work than another. Hypothetically we can tell it will run a lot slower and take up more space and the graphs clearly show that. The key difference being we don't need as many piles to continually find solutions for the worm hole puzzle compared to black hole.

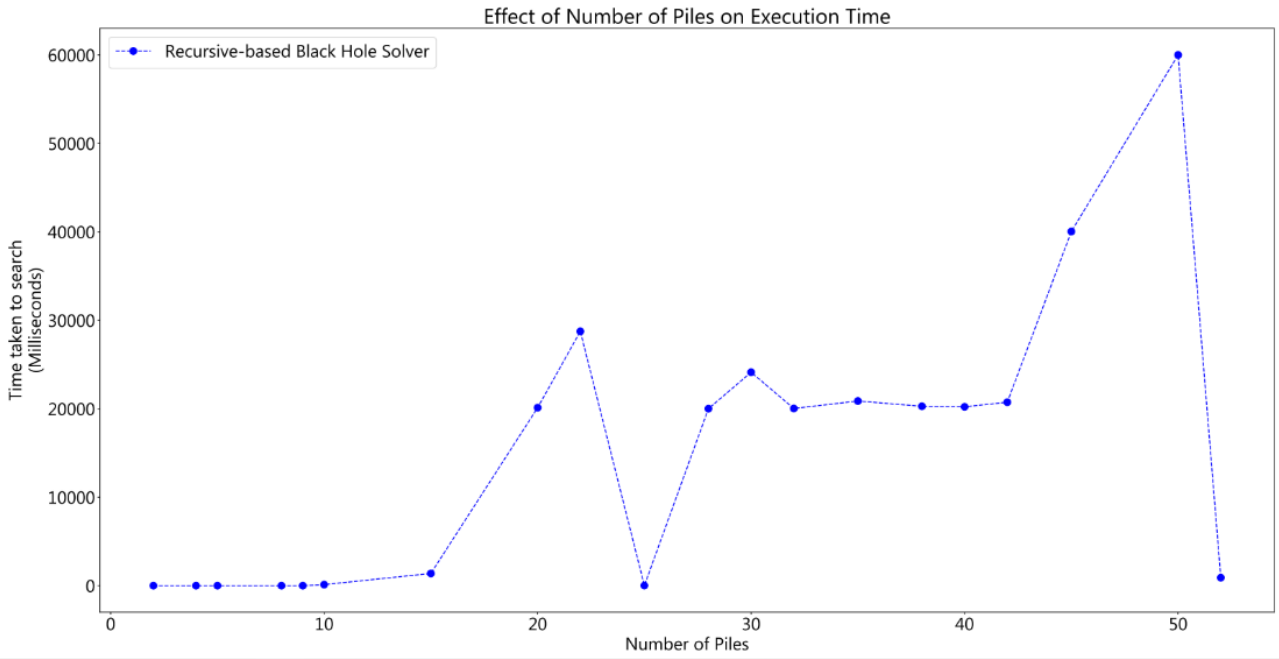
All in all, the evaluation here clearly shows that this algorithm works well up until a given limit. As soon as we introduce an excessive amount of piles, the solver takes a large amount

of time and space to search for a solution. We have to be realistic about our use of the algorithm and be very careful with how we use it. For small 52 deck puzzles with 10-20 piles, the algorithm will work quite well bar the few pathologies it may come across. However, as soon as we bombard the algorithm with a larger deck with more piles, our space and time increase dramatically even if more piles means a higher success rate.

**Recursive-based Black Hole Solver**

During implementation, I decided to design a recursive-based solver since the list-based solution was quite demanding in terms of it's iterative cycles. Before deepening on a new node it expands all possible nodes and pushes them to the list. The recursive algorithm however, only finds one node then recurses, if this deepening fails it then backtracks and expands. It avoid any unnecessary expansion that may occur in the list-based solution.

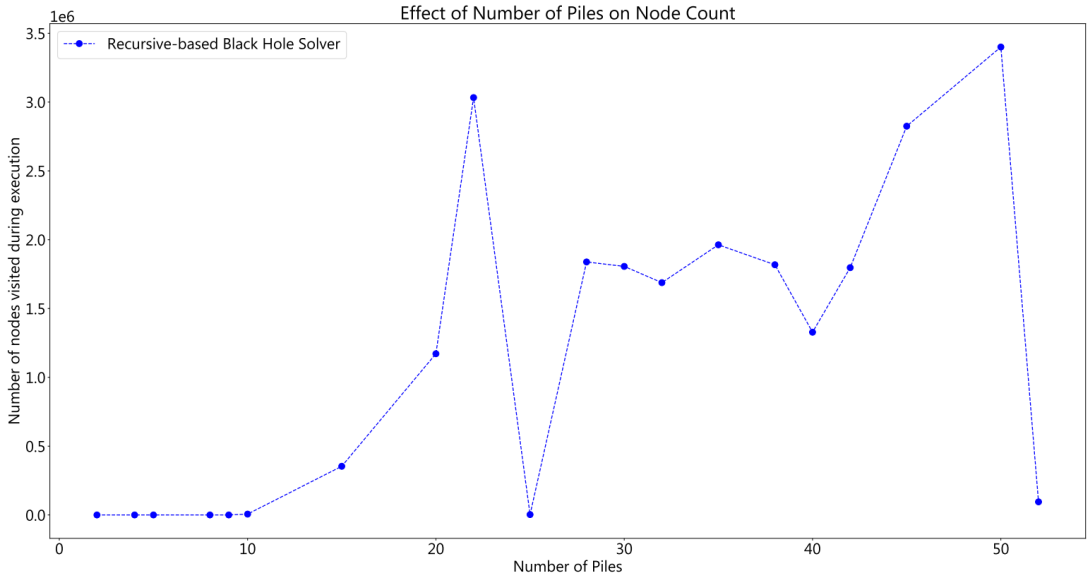
**Time**



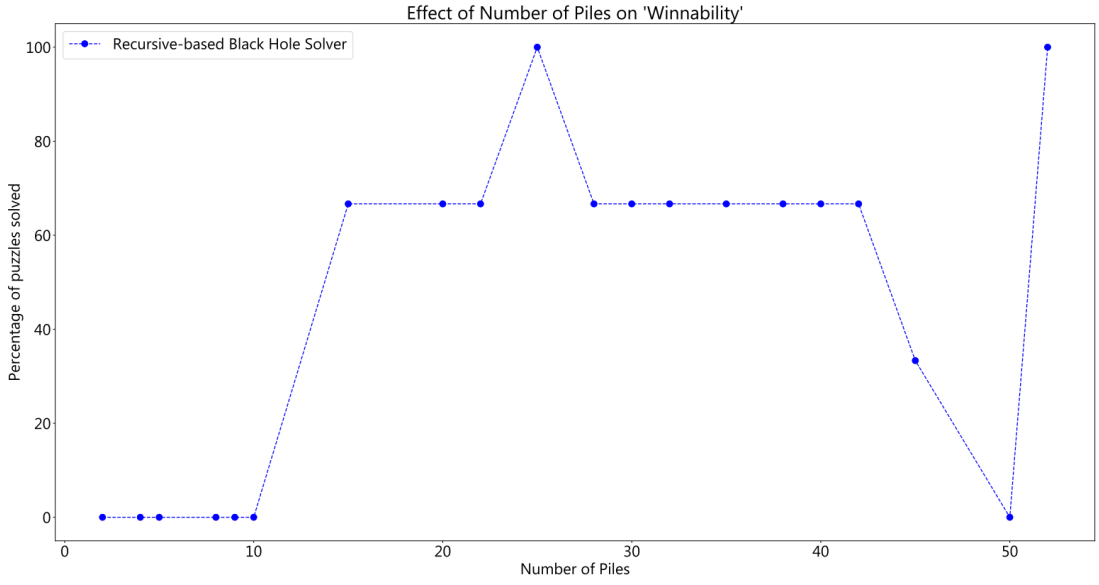

Immediately we can see that the recursive algorithm works a lot better when the number of piles increases. For starters, only until we have 50 piles does the algorithm time out, and when we have 52 piles, the expected quick response of solving occurs as it is almost at 0 ms. Our recursive approach is therefore a lot better at solving puzzles faster compared to our list-based implementation.

**Space**

Space here is a little bit more complex for the recursive approach as the stack space is something we must also take into account. Recursive approaches are destined to use up more stack space because of the repeated function calls but if we want to look at automatic memory allocation, it may once again be a better alternative as we are not keeping track of a list of nodes. However if we look at the node count:




It once again follows the pattern that if we spend more time looking at nodes, we look at more nodes. The recursive approach does look through a lot more nodes than the list-based implementation on average but perhaps comparing them isn't a true reflection of the algorithm since we are not keeping track of nodes pushed onto a list.




This looks a lot better than our iterative approach and largely down to the fact the algorithm does not time out as frequently. Because of this, our chances of finding a solution go up. If we let the iterative solution run for as long as need be perhaps it would look exactly the same as the graph above.

The behaviour seen with the suit and rank permutations with the list-based solutions is also seen with the recursive implementations.

Also see the graphs folder in the submission for graphs with different seeds. In some cases, randomly generated layouts have created pathological cases for the algorithms that lead to timeouts.

The specification prevents a very abstract idea of 'hardness' to do with instances of puzzles and how this correlates to the winnability. Apart from some pathological cases which are always going to arise, I would say hardness presents itself not because of some heuristic choices we have to make (because we make none) but because of the number of nodes we have to search which is directly tied to the deck size and number of piles. If we increase our piles, the 'hardness' goes up as we have to search many more states however, this directly leads to an increase in winnability since our deck is spread out across more piles we increase the probability of more adjacent cards being present.

All in all, I think the algorithms implemented are sound, well-formatted and work when taken with appropriate care (e.g. trying to solve a worm hole puzzle with a black hole layout). We have to be sensible about the puzzle we are trying to solve. Be aware that because of the length of time performing the tests above take, not many were taken. We might be able to see more refined patterns if we ran more tests.

Overall, I am very happy with this implementation and report. It is thoroughly tested however, the program requires sensible usage to avoid any runtime exceptions. The structure of the project is well done if a little large and running tests especially, takes up a lot of space. It could have been taken further perhaps by looking at complete search trees, adding new features to enhance the algorithm which I'm sure are available for a depth-first search, and possibly adding a heuristic. However, I didn't see the necessity for a heuristic as for a Black hole solution, all solutions must have the same number of steps so we could not quantify an 'optimal' one.

## Check Table

| Task | DONE | REPORTED |
|---|---|---|
| Part 1 – Checking | | |
| Design & Implementation | Yes | Yes |
| Provided Tests Passed | Yes | Yes |
| Additional Testing Undertaken | Yes | Yes |
| Part 2 – Black Hole Search | | |
| Design & Implementation | Yes | Yes |
| Provided Tests Passed | Yes | Yes |

| | | |
|---|---|---|
| Additional Testing Undertaken | Yes | Yes |
| Evaluation Performed & Reported | Yes | Yes |
| Other Evaluation Questions Answered | Yes | Yes |
| Part 3 – Worm Hole Search | | |
| Design & Implementation | Yes | Yes |
| Provided Tests Passed | Yes | Yes |
| Additional Testing Undertaken | Yes | Yes |
| Evaluation Perfomed & Reported | Yes | Yes |
| Other Evaluation Questions Answered | Yes | Yes |
| Code | | |
| Code well structured & written | Yes | Yes |
| Build.sh script included and working | Yes | Yes |
| Bugs/Inefficiences identified | Yes | Yes |
| Report | | |
| Report written covering required points | Yes | Yes |
| Any additional material/insight presented | No | No |
| Problems encountered/ Overcame in report | Yes | Yes |

Word count: 4287