CS3105 Practical 1: Search in Black Hole Patience

Ian Gent
Mun See Chang, Nguyen Dang

September 20, 2021

This practical carries 50% of the coursework assessment for the module and is due at 21:00 on 8th October, 2021. (Please note that MMS deadlines are definitive and also subject to change so should be checked.)

This practical involves implementing search algorithms for the patience game Black Hole.

The practical comes in three parts. In Part 1 you are to implement code to check potential solutions are correct. In Part 2, you will implement and evaluate Depth-First Search for Black Hole. In Part 3 you are to add a solver for a variant of Black Hole called 'Worm Hole' which adds one different rule, and test and evaluate it.

Marking: As always, each submission will be assessed by the School's standard mark descriptors, but we are providing the following guidance:

- An excellent submission which completes Part 1 excellently but does not attempt Part 2 is likely to get a mark up to 11.
- A submission which completes both Parts 1 and 2 excellently is eligible for marks up to 17.
- A submission which completes all parts excellently is eligible for marks up to 20, without necessarily having to include any extensions beyond the work suggested here.

REMEMBER:

- INCLUDE your source code! Work without source code, however good, is likely to attract a very much lower mark than it would otherwise get.
- INCLUDE your report! Work without a report, however good, is likely to attract a very much lower mark than it would otherwise get!

Black Hole Patience

There are many single player card games, called in general 'Patience' in British English and 'Solitaire' in American English. Each game has a particular set of rules, and we are going to look at a game called 'Black Hole'. Black Hole was invented by David Parlett[1] with these rules:

[1]The Penguin Book of Patience, David Parlett, 1979

“*Layout:* Put the Ace of Spades in the middle of the board as the base or ‘black hole’. Deal all the other cards face up in seventeen fans [i.e. piles] of three, orbiting the black hole.

Object: To build the whole pack into a single suite [i.e. sequence] based on the black hole.

Play: The exposed card of each fan is available for building. Build in ascending or descending sequence regardless of suit, going up or down ad lib and changing direction as often as necessary. Ranking is continuous between Ace and King.”

The table below shows an instance of the game: the 18 columns represent the A♠ in the black hole and the 17 piles of 3 cards each. We write T for 10, A for Ace (ranked 1), J for Jack (ranked 11), Q for Queen (ranked 12) and K for King (ranked 13).

	4♢	7♡	7♠	3♢	5♠	T♣	6♠	J♣	J♠	9♢	7♢	2♣	3♡	7♣	3♠	6♢	9♣
	9♠	9♡	J♡	4♠	K♢	Q♢	T♠	T♢	A♣	Q♠	K♠	Q♡	5♡	K♣	8♡	J♢	2♢
A♠	8♠	5♢	2♡	5♣	T♡	3♣	8♣	A♡	2♠	K♡	Q♣	4♡	6♣	6♡	A♢	4♣	8♢

A solution to this game is:

A♠, 2♣, 3♠, 4♢, 5♠, 6♠, 7♠, 8♡, 9♠, 8♠, 9♣, T♠, J♠, Q♡, J♡, T♣, J♣, Q♢,
K♢, A♣, 2♠, 3♡, 2♢, 3♣, 4♡, 5♡, 6♣, 7♡, 8♣, 7♣, 6♢, 7♢, 8♢, 9♡, T♡,
9♢, T♢, J♢, Q♠, K♠, A♡, K♡, Q♣, K♣, A♢, 2♡, 3♢, 4♠, 5♣, 6♡, 5♢, 4♣.

If you want to play the game yourself, you can of course use a pack of cards, but also there are free websites which let you play the game online (but check they have the rules correctly as at least one online version actually implements ‘Worm Hole’, see below).[2]

Input Format: To allow for comparison and testing, we will use an incredibly simple input format. Its simplicity is from the point of view of parsing into your program, it may not be the clearest for reading as a human. For example, a valid but unsolvable input is as follows, with the explanation below:

```
13 4 5
1
-1
-1
-1
15 42 -1
-1
```

- We will represent cards as integers from 1 to 52 (for a standard deck) or 1 to number of suits times number of ranks in each suit for a variant deck.
- Each suit is a consecutive sequence of cards. E.g. in a standard deck we will represent the Spades by 1 to 13, the Hearts by 14 to 26, Diamonds by 27 to 39, and Clubs by 40 to 52. This may be a bit confusing but fortunately it is not important. For this practical you are not required to convert between regular cards and integers. Instances are supplied as sets of integers.
- The first three numbers (on the top row) represent, in order: the number of ranks in each suit; the number of suits in the deck of cards, and the number of piles in the layout.
- The fourth number (on the second row) represents the hole card, in this case the Ace of Spades.

[2]E.g. `https://www.solitaireparadise.com/games_list/black-hole-solitaire.html`

- The remaining lines give the piles (which should match the number in the first line). Each pile must be terminated by a -1.
- The first card represents the bottom card in the pile, the second the one on top of that, and so on.
- A pile is allowed to be empty. There is no upper limit to how large a pile is.
- The piles are indexed from zero.
- A simple reader is provided in Java for reading this input format. Note that it ignores line breaks so the input would be valid all one line or with every number on a separate line. If you are creating your own instances you can format them how you wish, but there is no scope for comments except after the last pile is completely specified, when all text is ignored.

As another example, a very simple solvable problem would be:

```
5 1 4
1
4 -1
2 -1
3 -1
5 -1
```

This represents a trivial problem with a single suit of 5 cards, the ace in the hole, and one card in each pile. It is obviously easily solvable.

Output Format: To simplify checking we are requiring the following output format, but it only applies to the FIRST line of your output. Following that, you may choose to add any additional output that you wish.

The FIRST line of your output should be a sequence of integers as follows:

- It should start with either -1, 0 or 1, with -1 meaning your program could not determine the answer, 0 meaning the program proved the instance had no solution, and 1 meaning your program found a solution.
- If the first number is -1 or 0 that should be the end of the first line.
- If the first number was 1 (so there is a solution) the solution should be given in the rest of the first line.
  - The solution is given as two numbers for each move.
  - The first number in each pair is the pile that the card was moved from into the hole.
  - The second number represents the card that was moved into the hole, i.e. the card that was on top of the pile given by the first number.

As an example, one solution for the simple example puzzle above would be:

```
1 1 2 2 3 0 4 3 5
```

The initial 1 indicates success, then that we moved card 2 from pile 1, followed by card 3 from pile 2, card 4 from pile 0, and card 5 from pile 3.

Starter Code

You have been given some code to start you off. In particular BHMain.java and BHLayout.java. BHMain deals with command line arguments as required in the practical. BHLayout gives a data structure for storing the initial layout. This also contains a simple instance reader and a random generator of instances.

Part 1: Checking Solutions for Black Hole

The first part of this practical lays groundwork for the later parts, helping you to understand and design the necessary structures. It will also provide useful functionality for testing. You are to implement the following checking function:

`java BHMain CHECK PUZZLE` checks that the puzzle is correctly solved by the solution given on the standard input stream. It should output `true` if this is correct and `false` if not. Specifically the requirements are:

- Each pair of numbers in the solution sequence correctly identifies a card. That is, the given pile must be non-empty at the time the card is to be played, and its top card at that time must be the one listed in the solution.
- The first card is adjacent in rank to the initial hole card, i.e. is either one more or one less than the hole card.

  Throughout this document, 'adjacent in rank' includes the circular case where one card is the lowest rank and the other is the highest. For example, if there are 13 ranks then ranks 1 and 13 are adjacent. This captures the idea in the game that Ace and King are considered adjacent.
- Each card after the first is adjacent in rank to its predecessor.
- When all cards in the sequence have been played, all piles are empty.
- Additional input after a correct sequence can be ignored.

stacscheck: To test your program using `stacscheck`, run the following command from *the home folder of your submission* on a lab machine:

```
stacscheck /cs/studres/CS3105/Practicals/P1/Tests/Part1/
```

REMEMBER: `stacscheck` is only part of the process of making sure your code is working correctly! If you don't do more testing, then you risk your code not working.

Part 2: Depth-First Search for Black Hole

Implement, test, and evaluate depth-first search for Black Hole. We are not requiring any particular implementation technique. You can use recursive or iterative methods. You may certainly use the list based method for implementing search provided in lectures. But also you might want to think about ways to make search as fast as possible and optimise the implementation. The key point is to implement the search process correctly, and evaluate it well.

Your code should run with the following command and print output in the format specified above.

```
java BHMain SOLVE PUZZLE
```

Testing: Knowing that your code is giving correct results is extremely important. As well as giving results on stacscheck cases that we give you, please undertake additional testing and report this. Show that your depth-first implementation works correctly. Please be sure to look out for issues such as reporting incorrect solutions and incorrectly stating there is no solution.

Evaluation: As well as description of your program plus report on testing etc, your report should contain an evaluation of its performance. Evaluate your algorithms on a range of problems - both test data and additional instances that you might want to generate by hand or randomly. To help evaluate you might wish to add reporting features to your program such as time taken and number of search states visited.

Answer the following questions in your report:

- To what size of problem can your program reasonably reliably find solutions? [3]
- How efficient is your program? Can you identify particular aspects of it that could be optimised?
- How do the numbers of piles and cards in the deck affect the winnability of Black Hole, and how do they affect how difficult the instances are to solve? Is there any correlation between winnability and hardness?

In reporting how you answered questions like these, feel free to include graphs or other means of presentation that make your points most effectively. Extremely good evaluations might extend the evaluation significantly beyond the questions asked above.

stacscheck: To test your program using `stacscheck`, run the following command from *the home folder of your submission* on a lab machine:

```
stacscheck /cs/studres/CS3105/Practicals/P1/Tests/Part2/
```

Part 3: Worm Hole - a variant of Black Hole

In this part you will adapt your program to solve a variant game called 'Worm Hole'. This game is mainly the same as Black Hole but with one important change. As well as the black hole where every card must end up, there is also a 'worm hole' which a card can temporarily move through. At any time a card at the top of the pile can be put into the worm hole but there can only be one card at a time in the worm hole. When it is convenient the card can be moved from the worm hole to the black hole, following the normal rule about the card being adjacent in rank. Moving the card from the worm hole to the black hole frees the worm hole to take up another card.

Input Format: An input instance in the Black Hole described above is a valid instance for Worm Hole, and we consider the worm hole to be initially empty.

Output Format: The format is similar to that for Black Hole with some changes to allow moves from/to the worm hole.

- As before the first number should be 1/0/-1.
- In the case of the first number being 1, the remainder of the first line should again be pairs of numbers.
- Moves from a regular pile to the black hole are exactly as before.

[3]We have set stacscheck to time out after 1 minute but you are welcome to do experiments with either shorter or longer time limit.

- For a move FROM the worm hole to the black hole, the pile number is given as -1. The worm hole must not be empty and the card number must be the card in the worm hole. The card moved from the worm hole must be adjacent in rank to the card in the black hole.
- For a move TO the worm hole from a pile, the pile number is given as normal. However, the card number is negated, so e.g. if the card being moved is 17 from the top of pile 4, we would represent this as 4 -17. The worm hole must be empty but the card being moved does not need to be adjacent in rank to the hole card.

For example, a solution for the first example puzzle in page 2 would be:

```
1 3 -42 3 15 -1 42
```

Implement the code necessary to allow correct running of the two commands

`java BHMain CHECKWORM PUZZLE SOLUTION` checks that a claimed solution actually does solve the intended puzzle as a game of Worm Hole.

`java BHMain SOLVEWORM PUZZLE` which attempts to solve a problem as an instance of Worm Hole.

Test, evaluate and report on your implementation and how its performance compares with that of Black Hole.

stacscheck: To test your program using `stacscheck`, run the following command from *the home folder of your submission* on a lab machine:

```
stacscheck /cs/studres/CS3105/Practicals/P1/Tests/Part3/
```

Hand-in and Report requirements

Code Requirements

MAKE SURE TO INCLUDE SOURCE CODE IN YOUR SUBMISSION!!

Your program must be written in Java. It must be buildable on lab machines without further software installation. Source code should be in a directory called `src` within your submission. It must be buildable from this script - a simple version is supplied but you can edit it freely.

```
./build.sh
```

Your source code should be well structured and well commented. Where any code from other sources is used (except for the provided starter code), you must provide clear description of which code is yours.

Your program should run from the command-line, and send its output to standard output. Examples of what is expected can also be seen from the provided stacscheck tests.

Report Requirements

MAKE SURE TO INCLUDE YOUR REPORT IN YOUR SUBMISSION!

There is an ADVISORY word limit of 5400 words for this practical. A much shorter submission is likely to be able to get high marks. A submission which is significantly above the suggested count may be penalised if the length is disproportionate to the content. The word count does not include the table page, bibliography, or additional pages on additional details like results of tests.

You need to state if your program works fully or if there are non-working aspects. If the latter is the case, you should provide clear details. Instructions for compiling and running must be confirmed through clear screen snapshots of them working successfully on a specified lab machine, not just your own laptop.

Check Table: Please include a copy of the check table (see below) filled in accurately at the end of your report. Write your report so the attached check table elements are clearly and directly addressed by using the left hand column of the rows as section headings. Incorporate the table into the report on its own page to show which aspects have been done and provide a cross-reference into the report describing the relevant aspect where feasible.

Use of Lab Machines

You are STRONGLY ENCOURAGED to use School machines for building and testing, either remotely or in person (obeying the health and safety guidelines for lab use). Use of lab machines helps in two major ways: first it ensures that your programs are working in the environment in which they will be tested when marking; and second it gives a measure of protection in case of your own machine breaking down.

If there is a technical reason why you cannot use lab machines, please contact us in advance of submission.

If at all possible, please ensure that anything you submit can be viewed / executed / processed on the Linux lab machines (not the host servers), without any additional third party tools or libraries. If your submission cannot be checked on the CS lab machines during marking, then your final mark for this practical may be impacted.

If using School machines, please use Linux lab machines via remote access instead of host servers. See `https://systems.wiki.cs.st-andrews.ac.uk/index.php/Lab_PCs`

What to hand-in: Submit the report, source code and (if possible) jar file, via MMS. The format of the report must be that of PDF. The submission must be zipped into a single archive for MMS. Before submitting, remember that the University policy on Good Academic Practice applies, see: `https://www.st-andrews.ac.uk/students/rules/academicpractice/`

Marking: See the standard mark descriptors in the School Student Handbook, `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`

Lateness: Late work will be penalised by MMS, see: `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Check Table

Task	DONE?	REPORTED?
Part 1 - Checking Solutions		
Design + Implementation		
Provided Tests passed		
Additional Testing Undertaken		
Part 2 - Black Hole Search		
Design + Implementation		
Provided Tests passed		
Additional Testing Undertaken		
Evaluation Performed + Reported		
Other Evaluation questions answered		
Part 3 - Worm Hole Search		
Design + Implementation		
Provided Tests passed		
Additional Testing Undertaken		
Evaluation Performed + Reported		
Other Evaluation questions answered		
Code		
Code well structured + written		
Makefile + Build Instructions		
Bugs / Inefficiencies Identified		
Report		
Report written covering required points		
Any additional material/insights presented		
Problems Encountered / Overcame in report		