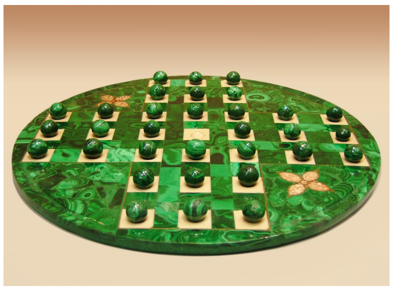
COMP20003 Algorithms and Data Structures
Second (Spring) Semester 2020

[Assignment 3]
AI Solver for Peg Solitaire: Graph Search

Handed out: Monday, 19 of October
Due: 23:59, Sunday, 1 of November



Purpose

The purpose of this assignment is for you to:

- Increase your proficiency in C programming, your dexterity with dynamic memory allocation and your understanding of data structures, through programming a traversal algorithm over Graphs.
- Gain experience with applications of graphs and graph algorithms to solving games, one form of artificial intelligence.

Assignment description

In this programming assignment you'll be expected to build an *AI algorithm* to solve Peg Solitaire (known as Brainvita in India). The game was invented in the XVII century in Madagascar. The first reference to the rules appeared in 1687 in a french cultural magazine. It is one of the classic board game puzzles, and several boards that differ in shape and size appeared through time. You can play the game using the keyboard and compiling the code given to you, or using this web implementation.

The Peg Solitaire game enginge code in this assignment was adapted from the open-source terminal version made available by Maurits van der Schee[1].

The Peg Solitaire game

As explained in the wikipedia entry, The player can move a peg *jumping* on top of another adjacent peg, if there is a free adjacent cell to land. There are 4 valid jumps: Left, Right, Up and Down.

The objective is to clean the board until there is only 1 peg left. Some variants require that the last remaining peg sits on the middle of the board. In this game we ignore that variant and win the game if only 1 peg is left, no matter its final position. An AI agent or human player can chose the sequence of jumps to win the game.

Solving Pegsol belongs to the class of problemns known as *NP-Complete* problems (paper). NP-complete problems are hard to solve. The best algorithms to solve NP-Complete problems run in

[1]https://github.com/mevdschee/peg-solitaire.c

exponential time as a function of the size of the problem. Hence, be aware that your AI solver will struggle more as the number of pegs increases. We talked in the lectures about the Travelling Salesman Problem as one example of an NP-Complete problem. In fact, many real-world problems fall under this category. We are going to learn more about NP-Complete problems in the last lecture of the course.

As a curiosity, you can see the number of distinct board positions on the standard 33-hole cross-shaped layout as an entry on the encyclopedia of integer sequences.

Human Player Mode

In the code provided, you can move the cursor in four directions using the arrow keys: up, down, left, and right. Use the enter/return key to select (or deselect) a peg. Pegs can then be moved using the arrow keys. Quit, restart and undo keys are 'q', 'r' and 'u'. A peg can only move if it can jump over a adjacent peg and land on an empty space. A peg is represented by a circle and a empty space by a dot. You win the game when you end with one peg (anywhere in the board). When there are no more moves possible the game is over.

The Algorithm

Each possible configuration of the Peg Solitaire (Pegsol) is a tuple made of: $m \times m$ grid board, the position of the cursor and whether the peg under the cursor has been selected. This tuple is called a *state*. The Pegsol Graph $G = \langle V, E \rangle$ is implicitly defined. The vertex set V is defined as all the possible configurations (states), and the edges E connecting two vertexes are defined by the legal jump actions (right, left, up, down).

Your task is to find the path leading to the best solution, i.e. leading to the vertex (state) with the least number of remaining pegs. A path is a sequence of actions. You are going to use Depth First Search to find the best solution, up to a **maximum budget** of expanded/explored nodes (nodes for which you've generated its children).

When the AI solver is called (Algorithm 1), it should explore all possible paths (sequence of jump actions) following a Depth First Search (DFS) strategy, until consuming the budget or until a path solving the game is found. Note that we do not include duplicate states in the search. If a state was already generated, we will not include it again in the stack (line 21). The algorithm should return the **best solution found**, the path leading to the least number of remaining pegs. This path will then be executed by the game engine.

You might have multiple paths leading to a solution. **Your algorithm should consider the possible action by scanning the board in this order**: traverse coordinate $x = 0, \ldots, m$ first, and then $y = 0, \ldots, m$ looking for a peg that can jump, and then selecting jumping actions left, right, up or down.

Make sure you manage the memory well. When you finish running the algorithm, you have to free all the nodes from the memory, otherwise you will have memory leaks. You will notice that the algorithm can run out of memory fairly fast after expanding a few milion nodes.

When you *applyAction* you have to create a new node, that

1. points to the *parent*,
2. updates the *state* with the action chosen,
3. updates the *depth* of the node.
4. updates the *action* used to create the node

Algorithm 1 AI Peg Solitaire Algorithm

```
procedure PEGSOLVER(start, budget)
    $n \leftarrow$ INITNODE($start$)
    STACKPUSH(n)
    $remainingPegs \leftarrow$ NUMPEGS($n$)
    while $stack \neq empty$ do
        $n \leftarrow stack.pop()$
        $exploredNodes \leftarrow exploredNodes + 1$
        if NUMPEGS($n$) $< remainingPegs$ then                              ▷ Found a better solution
            SAVESOLUTION(n)
            $remainingPegs \leftarrow$ NUMPEGS($n$)
        end if
        for each jump action $a \in [0, m) \times [0, m) \times \{Left, Right, Up, Down\}$ do
            if $a$ is a legal action for node $n$ then
                $newNode \leftarrow$ APPLYACTION($n, a$)                       ▷ Create Child node
                $generatedNodes \leftarrow generatedNodes + 1$
                if WON($newNode$) then                                       ▷ Peg Solitaire Solved
                    SAVESOLUTION(newNode)
                    $remainingPegs \leftarrow$ NUMPEGS($newNode$)
                    return
                end if
                if newNode.state.board seen for the first time then          ▷ Avoid duplicates
                    STACKPUSH(newNode)                                        ▷ DFS strategy
                end if
            end if
        end for
        if $explored\_nodes \geq budget$ then
            return                                                            ▷ Budget exhausted
        end if
    end while
end procedure
```

Deliverables, evaluation and delivery rules

Deliverable 1 – *Solver* source code

You are expected to hand in the source code for your solver, written in C. Obviously, your source code is expected to compile and execute flawlessly using the following makefile command: `make` generating an executable called `pegsol`. Remember to compile using the optimization flag `gcc -O3` for doing your experiments, it will run twice as quickly than compiling with the debugging flag `gcc -g` (see `Makefile`, and change the `CFLAGS` accordingly). For the submission, please **submit your makefile with `gcc -g` option**, as our scripts need this flag for testing. Your program must not be compiled under any flags that prevents it from working under gdb or valgrind

Your implementation should work well over the first 6 layouts, but it will not be expected to find a solution to win the last three layouts with budget limits up to 3M expansions requiring around 2GB of RAM.

Try different budgets and explain why your agent works well (or doesn't) in each of the test cases.

Base Code

You are given a base code. You can compile the code and play with the keyboard. You are going to have to program your solver in the file ai.c. Look at the file peg_solitaire.c (main function) to know which function is called to call the algorithm.

You are given the structure of a node, the state, a stack and a hashtable implementation to check for duplicate states efficiently (line 21 in the algorithm). Look into the utils.* files to know about the functions you can call to apply an action to update a game state.

In your final submission, you are free to change any file, but make sure not to change the name of the variable in layouts.h file, as we will change that file to test your algorithm with new layouts.

Input

You can play the game with the keyboard by executing

```
./pegsol <level>
```

where level $\in \{0, \ldots, 8\}$ for standard levels.

In order to execute your solver use the following command:

```
./pegsol <level> AI <budget> play_solution
```

Where `AI` calls your algorithm. `play_solution` is optional, if typed in as an argument, the program will play the solution found by your algorithm once it finishes. <budget> is an integer number indicating the budget of your search.

for example:

```
./pegsol 5 AI 1200000 play_solution
```

Will run the 6th layout expanding maximum 1.2M nodes and will play the soltution found.

Output

Your solver will print into an `output.txt file` the following information:

1. Number of expanded nodes.
2. Number of generated nodes.
3. Number of pegs left in the solution.
4. Number of nodes expanded per second.
5. Total search time, in seconds.

For example, the output of your solver `./pegsol 5 AI 1200000` could be:

```
STATS:
Expanded nodes:  1090275
Generated nodes:  4898609
Solution Length:  35
Number of Pegs Left:  1
Expanded/seconds:  329924
Time (seconds):  3.304622
```

Expanded/Second can be computed by dividing the total number of expanded nodes by the time it took to solve the game. A node is expanded if it was popped out from the stack, and a node is generated if it was created using the applyAction function. You can print all this information by adapting the current code in the `main()` function in `peg_solitaire.c`

Deliverable 2 – Experimentation

Besides handing in the solver source code, you're required to provide a table with the number of pegs left, generated nodes, expanded nodes, expanded/second, and total execution time for each layout and each max budget of 10k,100k,1M,1.5M.

Explain your results using your figures and tables. Plot the number of pegs left as a function of the number of pegs initially in the board. Also plot how the budgets affect solution quality in the last 4 layouts. You can include any other plot that may give you insights into how your algorithm works.

If you do any of the optimizations over the algorithm, please report and discuss it in your experimentation.

Answer concisely. **Please include your Username, Student ID and Full Name** in your Document.

Evaluation

Assignment marks will be divided into three different components.

1. Solver (11)
2. Code Style (1)
3. Experimentation (3)

Please note that you should be ready to answer any question we might have on the details of your assignment solution by e–mail, or even attending a brief interview with me, in order to clarify any doubts we might have.

Code Style

You can improve the base code according to the guidelines given in the first assignments. Feel free to add comments wherever you find convenient. From your comments it should be very clear exactly which lines implement each line of the pseudocode. The base code was minimally modified to allow easy deployment of your AI algorithm and the coding style belongs to the original author.

Delivery rules

You will need to make *two* submissions for this assignment:

- Your C code files (including your `Makefile`) will be submitted through the LMS page for this subject: *Assignments* → *Assignment 3* → *Assignment 3: Code.*
- Your experiments report file will be submitted through the LMS page for this subject: *Assignments* → *Assignment 3* → *Assignment 3: Experimentation.* This file can be of any format, e.g. .pdf, text or other.

Program files submitted (Code)

Submit the program files for your assignment and your `Makefile`.

Your programs *must* compile and run correctly on the JupyterHub server. You may have developed your program in another environment, but it still *must* run on the JupyterHub server at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on the JupyterHub server at reasonably frequent intervals.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary.

Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

"Borrowing" of someone else's code without acknowledgement is plagiarism, e.x. taking code from a book without acknowledgement. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic honesty and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) on the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Administrative issues

When is late? What do I do if I am late? The due date and time are printed on the front of this document. The lateness policy is on the handout provided at the first lecture and also available on the subject LMS page. If you decide to make a late submission, you should send an email directly to the lecturer as soon as possible and he will provide instructions for making a late submission.

What are the marks and the marking criteria Recall that this project is worth 15% of your final score. There is also a hurdle requirement: you must earn at least 20 marks out of a subtotal of 40 for the projects to pass this subject.

Finally Despite all these stern words, **we are here to help!** There is information about getting help in this subject on the LMS pages. Frequently asked questions about the project will be answered in the LMS discussion group.

Have Fun!

COMP20003 team.