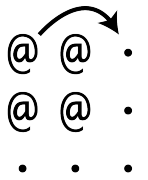
15-122: Principles of Imperative Computation, Spring 2014

Homework 6 Programming: Peg Solitaire



Due: Monday, March 24, 2014 by 22:00

For the programming portion of this week's homework, you will implement a program to solve peg solitaire puzzles.

The code handout for this assignment is at

`http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/prog6.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a 25 handin limit for this assignment. Additional handins will incur a 1-point penalty per handin.

Frequently and incrementally test your code. You can solve small boards without backtracking, and it is not necessary to use priority queues or hash tables to get full credit on this assignment. Make an effort to to catch conceptual errors early.

Modification to academic integrity and Piazza policy: The academic integrity policy for this course does not allow you to view other people's C0 code or share your C0 code with others. However, you may share peg solitaire boards that you find useful for testing in public Piazza posts.

Questions on Piazza that include Autolab error messages and ask "is there any test board that might help me figure out the problem" are fine, but will *not* be addressed by course staff. Tag these posts on Piazza by writing `#needaboard` in the post. Questions on Piazza about specific code problems will only be addressed by course staff if they reference a test board that demonstrates the problem. (Conceptual questions and clarifications are still fine on Piazza.)

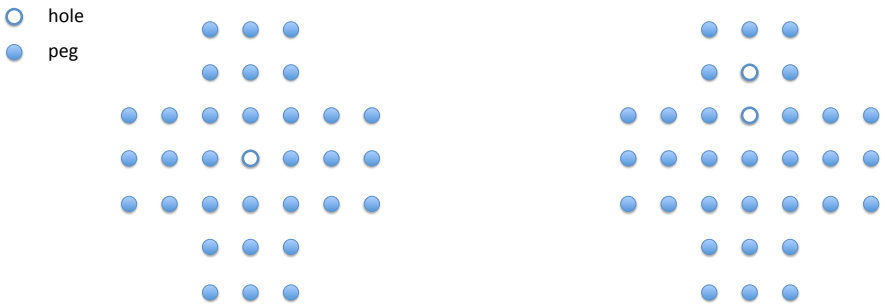
Style and contract testing We do not plan to do systematic style and contract checking for this assignment. However, we reserve the right to deduct points (up to 5) for significant issues not caught by the autograder (including contracts), and we may do some style grading, especially for code that does not pass the `-w` style checks that `cc0` performs.

One reason for this is that your code is liable to get pretty messy as you transform `peg1.c0` into `peg2.c0` and `peg2.c0` into `peg3.c0`. In order to succeed at this assignment, you will want to *refactor* your code – notice that it has gotten complicated and restructure it in a way that suits the task better. Refactoring a bit will probably help you write and debug the assignment, and one purpose of style grading this assignment is to help persuade you to do this.

Peg Solitaire

Peg solitaire is a one-player board game with the goal of removing all pegs except one from a board, starting with some initial board configuration consisting of holes, some of which are filled with pegs.

A move is always a vertical or horizontal jump of one peg over another, removing the peg that was jumped over from the board. For example, in the initial configuration of the standard English board on the left, there are 4 possible moves, all ending in the center. The peg arriving from the top leads to the configuration on the right.



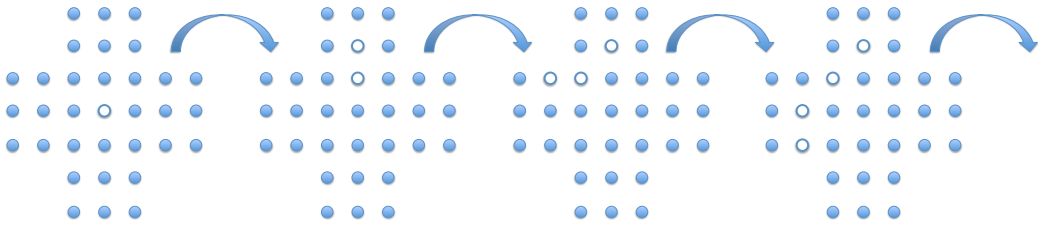
In the configuration on the right we have now have just 3 possible moves.

The goal of the game is to be left with just one peg. On the standard English board we start with 32 pegs, so any solution will require exactly 31 moves (each jump removes exactly one peg from the board). In some variations of the game we also stipulate where the final peg should come to rest, but in this assignment just reaching a board with a single peg is the only goal.

See `http://en.wikipedia.org/wiki/Peg_solitaire` for more on peg solitaire.

Games computers play

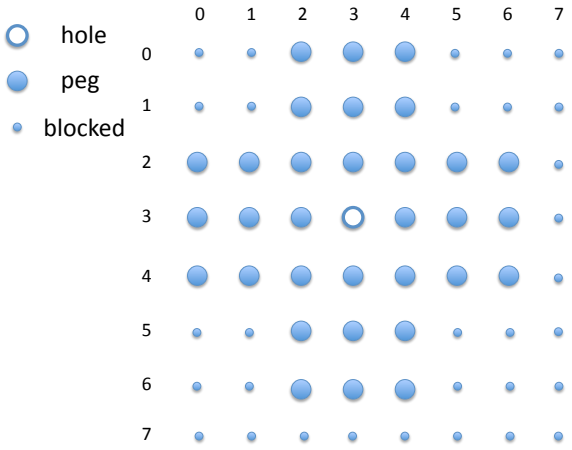
The structure of a peg solitaire is well suited to a solution involving recursion. If we want to play peg solitaire on a board with 32 pegs, then we can enumerate all the valid moves that can be made on that board. Then, after we make one of those moves, we're playing peg solitaire again – only this time on a board with 31 pegs.



If we ever reach a point where we're playing peg solitaire on a board with one peg, it is a very easy game – we've won! A *losing* game of peg solitaire is one where there are no valid moves and more than one peg. An *unsolvable* game of peg solitaire is one where every series of valid moves leads to a losing game.

Representation of the Board

In this programming assignment you will have to make some choices regarding the representation of moves, hash table keys, etc., but we specify the representation of boards. A board is always an array of size $8 \times 8 = 64$ integers, where -1 means the location is blocked (i.e. has no hole and cannot accept a peg), 0 means the location is a hole without a peg, and 1 means the location is a hole occupied by a peg. The board is laid out starting at the top left.



The notation we use for a location on the board is *row* : *col*, starting both *row* and *col* at 0 in the upper left-hand corner. For example, the first move illustrated in the example on the previous page would be from `1:3` to `3:3`. Remember that we represent the board as a one-dimensional array, so `1:3` corresponds to array index $1 * 8 + 3 = 11$ in the board array and `3:3` corresponds to array index $3 * 8 + 3 = 27$ in the board array. Please refer to the definitions in the file `lib/peg-util.c0`, including

```
typedef int[] board;
```

This file also contains functions to read board configurations from a file and print board configurations.

Representation of a Move

It is up to you how to represent moves.

```
typedef ________ move;
```

Some suggestions for representations: triples of integers (representing the three board indices involved), pairs of integers (the first and last index of the move), or four integers representing the row and column values of the peg before and after the move. These could be represented by structs, or could be compressed into something like a single `int`. This involves a tradeoff between compactness and ease or speed of determining the possible moves on a given board. We do **not** combine multiple jumps into a single move.

The testing harness will treat the type `move` as abstract. This means it only uses the following four functions to extract information from a given move m to check whether the move is valid:

```
int row_start(move m);
int col_start(move m);
int row_end(move m);
int col_end(move m);
```

You can find the testing harness in `peg-main.c0`. You cannot change this file, but you can inspect this code (and the code in `lib/peg-util.c0`) and reuse anything you find useful.

Representation of a Solution

A *solution* to peg solitaire from a given initial configuration consists of a *stack of moves*. The top of the stack should contain the first move, the next element the second move, etc. The function

```
bool verify_solution(board B, stack S);
```

(provided in the file `peg-main.c0`) verifies that the stack S is a valid solution for the initial configuration given by board B.

Because the solution is a stack of moves, your code needs to define the type of stack element, called `stackelem` in the file `peg-client.c0`, before the stack library is used. This file also contains a client implementation for hashtables, which you may use in `peg3.c0`.

1 Solving Deterministic Peg Solitaire

Task 1 is preliminary work for the subsequent tasks. For this task your code should try only one move (if any are possible) for each board configuration, reaching either a winning configuration (i.e. one peg left) or a losing configuration (more than one peg left, but no moves possible). This means that if it is given a *deterministic* board (one where there is always at most one move possible), it should find the solution if a solution exists. If given a nondeterministically solvable board (winnable, but with more than one possible move at some point), it might or might not find the solution. This won't matter - we will only test your `peg1.c0` implementation on both solvable and unsolvable determinstic boards.

The point of this task is to make sure that you can correctly determine possible moves and correctly generate solutions, and that you are providing appropriate implementations of the specified interfaces. You may find it very useful to adapt some of the verification code from `peg-main.c0` in the starter code, either directly or as a model. Just copy any useful code into `peg1.c0` and document where it came from in the comments. Also remember to look at `lib/peg-util.c0` (which is included when you compile) and call any useful functions you find there. (Don't copy them, just call them!)

Task 1 (7 pts) *Based on the problem description given, determine how you want to represent a move. In the file* `peg1.c0`*, define the type* `move`

```
typedef ________ move;
```

Also, in the file `peg1.c0`*, define the functions*

```
int row_start(move m);
int col_start(move m);
int row_end(move m);
int col_end(move m);
```

to return row and column information from a move where start refers to the starting position of the jumping peg, and end to its ending position.

Recall that a solution is represented as a stack of moves. In this assignment, our `stack` *is a stack of* `stackelem`*s. In the file* `peg-client.c0`*, define the type* `stackelem`

```
typedef _______ stackelem;
```

so `stackelem` *is consistent with the type* `move`*. This will allow the representation of a solution as a stack of moves. (We couldn't just write* `typedef move stackelem` *because the type* `move` *will not be defined until later in the compilation sequence.)*

In the file `peg1.c0`*, define the function*

```
int peg_solve(board B, stack S);
```

to attempt a deterministic solution to peg solitaire for the given board. The integer that `peg_solve` *returns is the number of pegs left at the end of the game.*

- *If* `peg_solve` *returns* `1`*, then our algorithm played peg solitaire and won. In this case,* `S` *should contain the solution in the form of a stack of moves.*
- *If* `peg_solve` *returns a number greater than* `1`*, then our algorithm played peg solitaire and lost when the board still had* `\result` *pegs left on the board. In this case,* `S` *can contain anything.*

Notice how `peg_solve` communicates information to its caller both by the `int` it returns and by the moves it pushes onto the stack. Since the stack is really a pointer, `peg_solve` and the function that calls it can both access the same area of memory through that pointer.

In order to write the `peg_solve` function, you should write a **recursive** helper function `solve` that is called from `peg_solve`. This helper function should take the board, the current stack of moves, and the number of pegs remaining on the board as parameters. It should return `1` if there is a solution for the given parameters. It should also alter the given solution stack by adding the moves for the solution. THINK: What is the base case here for this recursive function? When do you know you have a solution for the current board?

One strategy for this helper function is to consider all the possible moves that could be made for the current board. This could be written as an auxiliary function that generates all the possible valid "next moves," perhaps returning a new stack containing these valid moves. (It's a good idea to write the code to find all possible moves now, even though for this task you only need one move.)

Then, process the top move of the additional stack of first moves (ignore all of the others) and recursively try to see if there is a solution to the resulting board. If there is a solution to the resulting board, there is a solution to the current board. In case there is a solution, make sure you push the moves of the solution onto the solution stack in the correct order. It's easy for the final set of moves to end up in the wrong order on the solution stack.

Testing

You can test your `peg1.c0` solution by compiling it as described in `README.txt` and then running

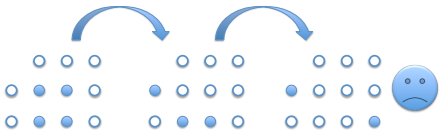
```
% ./peg1 german.txt
```

This one trivial test won't tell you very much, however, and you will need to test your code against other solutions as well. (You are encouraged to share such test cases on Piazza.)

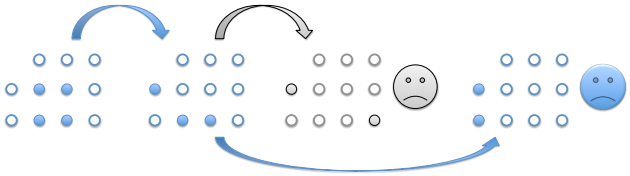
Our testing harness will only test your Task 1 code on peg solitaire boards that are deterministic (boards always have either 1 or 0 moves), but you may want to run your solution on more complicated boards. If you do, you should expect your code to sometime give up too early, failing to find a solution even when one exists.

2 Solving Nondeterministic Peg Solitaire

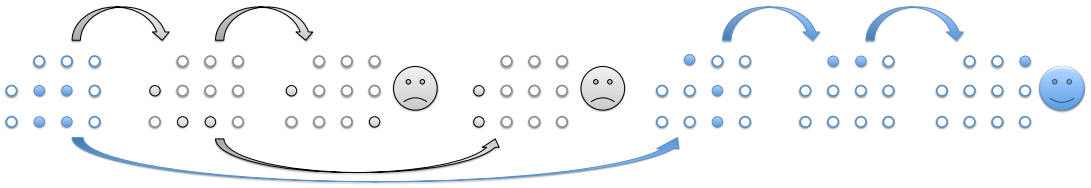
In the previous task, we only dealt with deterministic peg solitaire boars: with the board in its current state, we generated a stack of all possible immediate moves we could make from the current board, chose one, and played peg solitaire on the resulting board (recursively). Using this approach, if we hit a dead-end in our search for the solution, we're out of luck – we would return the number of pegs left on the board to indicate that we did not win the game. (The first move below is `1:2 to 1:0`, and the second move is `2:1 to 2:3`.)



We can do better with a strategy called *backtracking*. Right now we know that the third peg solitaire board above, with two pegs, is a losing game, which means it is also unsolvable. But we don't know that the *second* peg solitaire board, the one with three pegs, is unsolvable, because there's another valid move we could make – `2:2 to 2:0`. So we backtrack to this old board and try working forward from there.



Unfortunately, this second move also leads to a losing board. This means that the board with three pegs above *is* unsolvable – all the moves starting from that board lead to losing boards. We have to backtrack further, to the board with four pegs, and pick a different way forward. If we pick the valid move `2:1 to 0:1`, we will succeed.



In this task you will generalize the implementation from Task 1 so that it can solve boards that require backtracking for their solution.

Using the process of backtracking, we start with a board in some current state. We try a potential first move on a board, changing the board to a new state. If that move does not work (if the resulting board is unsolvable), we return back to the current state and try

another potential first move. We repeat this process as long as a first move does not lead to a solution and there are still potential first moves left to try. If none of the potential first moves lead to a solution, then we know the current board is unsolvable. We return back to the previous state of the board and try any remaining first moves still available, and so on. This backtracking process is also recursive. (Do you see why?)

To summarize, this task differs from the previous task in that your new code will try to definitively answer the question of whether a particular peg solitaire game can be won. If it hits a dead-end with no subsequent moves, it undoes the previous move and tries another move. It that fails, it undoes that move and continues this process.

Task 2 (8 pts) *Copy your working code from* `peg1.c0` *to* `peg2.c0`.

Extend the function `peg_solve` *so that it can handle boards that potentially require backtracking. We will test your code with simple problems requiring backtracking. The most complex one will be* `english.txt` *(the standard English peg solitaire board and initial configuration).*

However, if your `peg_solve` *function returns a number greater than* `1`*, indicating that the peg solitaire board was entirely unsolvable, then the returned integer should be the* smallest *number of pegs on any board you encountered. The stack* `S` *still only needs to contain a valid set of moves when the result is* `1`.

NOTE: For this task *do not use hash tables.* This will allow you to explore how complex the problems can be using this technique before you need a data structure like a hash table to reduce the expanding search space.

Testing

You should test your `peg2.c0` solution first with small unit tests and easy-to-solve boards with the `-d` option on as described in `README.txt`.

```
% cc0 -d -o peg2 peg-client.c0 lib/*.c0 peg2.c0 peg-main.c0
% ./peg2 mediumboard.txt
```

Then, by compiling with the "fast but unsafe" options described in the `README.txt`, you can try solving some trickier boards.

```
% cc0 -o peg2 -r unsafe -c-O2 peg-client.c0 lib/*.c0 peg2.c0 peg-main.c0
% ./peg2 english.txt
```

We will test your code with using several boards of increasing difficulty, culminating in `english.txt`, the standard English board and starting configuration. Depending on the order in which you pick moves, your code may be able to handle `english.txt`, even without the use of a hash table.

We will set timeouts on the Autolab server so that if your code it too slow it may fail some of the more difficult tests. So you should pay some attention to efficiency of your code. You should be aware that the Autolab server may run slower than your laptop (for instance).

Some Advice

Your solution should probably reuse most of the code from Task 1. You will mainly have to deal with the problem that applying a move *modifies* the board. When you backtrack by returning a number greater than `1` you must make sure to *undo* your change to the board. This can be stated as an invariant of the `solve` function: before it returns the board must be restored to the configuration it was in when it was called.

If you try to copy the board instead, your code will probably be too slow.

Efficiency is definitely a factor factor at this point. For example, it is probably worth considering how to *efficiently* check that you have reached a winning configuration since you may have to do this many times. We recommend sticking with the so-called *brute-force* search rather than trying to rank the possible moves based on their promise.

Most any definition for the type `move` in `peg1.c0` will good enough to get full credit. However, depending on how the rest of your code is designed, you may see a useful efficiency improvement if you represent moves as single integer (think about how ARGB pixels pack four values into a single int). Whether to do this is entirely up to you. If you do, don't forget to edit both `peg2.c0` and `peg-client.c0`!

3 Bonus: Memoization

The basic problem with the backtracking search from Task 2 is that it may visit the same unsolvable peg solitaire boards many, many times. In this (bonus) task, you will memoize your solve function using a hash table.

If you think of `peg_solve` as a function that maps a board to the smallest number of pegs reachable from that board, then the hash table will be the same type of mapping. Once a board B has been found to be unsolvable, it will be stored in the table with the number of pegs left in the final configuration.

When trying to solve a board we first check if the board has already been recorded as having no solution and, if so, we immediately return the answer stored with it in the hash table. Notice that everything in the hash table represents failure, so there is no need to store the move stack as part of the hashed element.

If we find a winning configuration, there is no need to store it since no further searching should occur.

Three crucial factors will determine the efficiency of your implementation: your choice of keys, your choice of hash function, and the order in which you try moves.

With respect to keys, we recommend compressing your board into a compact representation for the hash table containing enough information so that two boards in a given problem have the same key if and only if they represent the same configuration. The `key_equal` function

on this representation should be fast. Note that you could not use the board array itself anyway without copying, because then your `solve` function would change the contents of the hash table. If your key is a copy of the whole length 64 array of integers, then your implementation will almost certainly still be too slow. *It is not a good idea to use strings for keys, although you may be tempted since you saw a hash function for strings in lecture.*

When thinking about efficiency, remember that you're free to redefine your helper function `solve` to take as many parameters as you see fit. For instance, you might find it advantageous to update keys as you update boards, and pass them around together.

Task 3 (Bonus and scoreboard glory only) *Copy your code from* `peg2.c0` *to* `peg3.c0`.

In the file `peg-client.c0`*, provide the client-side implementations of the following types and functions that are necessary for the hash table to work correctly.*

```
typedef _____* htelem;
typedef _____ key;

int hash(key k, int m);
bool key_equal(key k1, key k2);
key htelem_key(htelem e);
```

A usable example has been provided for you. In the file `peg3.c0`*, extend your code to take advantage of the hash table to reduce the search space for a solution.*

Testing

We will test your code with

```
% cc0 -o peg3 -r unsafe -c-O2 peg-client.c0 lib/*.c0 peg3.c0 peg-main.c0
% ./peg3 english.txt
```

using several boards of increasing difficulties, culminating in the files `english.txt` and `french*.txt` (a French version of peg solitaire).

As in Task 2, we will set some timeouts to prevent your code from consuming unbounded system resources and verifying that your code is reasonably efficient.

Some Advice

Efficiency is becoming a major factor at this point. Move selection remains an important issue. The efficiency of your hash function, the compactness of your hash key representation, the size of your hash table, and the ability of your hash function to avoid collisions are also important factors. Some of these issues are discussed in `performance-debugging.txt`.

Be aware that, for hashing to work properly, the keys and elements must not be updated after they are put in the hash table. If you incrementally update any data that ends up in the hash table, be sure to pass a copy of it to `ht_insert`, not the updatable data itself.

Even with memoization and well-written code, some peg solitaire boards, including the French boards, can cause your program to run for a very, very long time. Don't assume that you have a bug if this happens. Remember that the French boards don't affect your score on this lab. Solving them is a bonus.

Going Further

Our solution is entirely *brute force*, that is, it does not employ any heuristic ordering among the possible moves. You might consider adding such heuristics to select the most promising moves first once you get this assignment done. All other aspects of the solution should be the same as in the required assignment. If you go beyond the required assignment, please include a `HEURISTIC.txt` file that explains your strategy.

A Move selection

The order in which you consider moves will not make a big difference to how fast you search unsolvable boards, but it can make a big difference to how fast you find the solution to a solvable board. If you pick moves by iterating over the board spaces, there are three obvious options:

- Every time you find a peg, see if it can jump in every direction (up, down, left, right).
- Every time you find a peg, see if it can be *jumped over* in every direction.
- Every time you find a hole, see if it can be jumped *into* from every direction.

We recommend you try the first of these three strategies and then experiment with different orderings of directions (up-down-left-right versus up-right-down-left and so on) to find one that works well on the English board. With the right move selection strategy, you can find a solution to `english.txt` in Task 3 very quickly (there can be less than 1100 boards in your hash table after solving the English board).

There is another choice you could experiment with: will you try all possible directions at each peg/hole before moving on to the next, or one direction at every peg/hole on the board before trying another direction? This can also make an enormous difference, depending on many things.