*Concepts in Multicore Programming* March 30, 2010
Massachusetts Institute of Technology 6.884
Charles E. Leiserson Handout 11

# Lab 5: Backtracking Search

In this lab, you will implement a parallel backtracking search to solve games of peg solitaire. The write-up for this lab is due on Wednesday, April 7 at 5:00 P.M.

## 1 Peg Solitaire

Peg solitaire is a classic one-player game which involves removing pegs from a board with holes. Peg solitaire is played on a ***board*** which is an $n$-by-$m$ grid of squares, with each square being either valid or invalid. A valid square, or ***hole***, may contain a peg. A hole is ***filled*** if it contains a peg, and it is ***empty*** otherwise. A ***configuration*** is a board with some subset of its holes filled. Thus, a board with $H$ holes has $2^H$ possible configurations.

In peg solitaire, a player transforms one board configuration into another via ***jumps***. A jump involves 3 (vertically or horizontally) consecutive holes, with the first square identified as the starting square for the jump. More precisely, for a given board, one can represent a ***jump*** as a triple $(i, j, dir)$, where $i$ and $j$ are the row and column on the board of the jump's starting hole (indexed from 0), and $dir$ is the direction of the jump (either `NORTH`, `EAST`, `SOUTH`, or `WEST`). A jump on a board is ***legal*** for a particular configuration if the jump involves 3 consecutive holes with the first and second holes filled and the third hole empty. A legal jump transforms the configuration by removing the two pegs from the first and second holes, and filling the third hole with a peg. For example, Figure 1(b) admits 3 legal jumps starting from square $(2,2)$: $(2,2,$`NORTH`$)$, $(2,2,$`EAST`$)$, and $(2,2,$`WEST`$)$. The jump $(2,2,$`SOUTH`$)$ is never legal for any configuration of this board, however, because $(4,3)$ is out of range.

The goal of peg solitaire is to find a sequence of legal jumps which transforms a starting configuration into a final configuration. In many games, the desired final configuration contains only a single peg. For example, in Figure 1, there exists a sequence of legal jumps which takes configuration (b) to configuration (f); in this case, (c), (d), and (e) represent the intermediate configurations for this sequence.

```
 --o-o      --o-o      --o-o      --o-o      --o-o      --o-o
 ooooo      ooXXo      ooXoo      ooooo      ooooo      oXooo
 ooooo      oXXXo      oXXoo      oXooo      oXooo      ooooo
 -oooo      -oooo      -ooXo      -oXXo      -Xooo      -oooo
  (a)        (b)        (c)        (d)        (e)        (f)
```

**Figure 1:** (a) A 4-by-5 peg solitaire board, with 16 holes, and (b) one possible configuration for this board. The character `-` represents an invalid square, `o` represents an empty hole, and `X` represents a filled hole. The following sequence of legal jumps transforms the starting board configuration (b) to the final configuration (f): $(1,3,$`SOUTH`$)$, $(1,2,$`SOUTH`$)$, $(3,3,$`WEST`$)$, and $(3,1,$`NORTH`$)$. The boards (c) through (e) represent the intermediate configurations for this sequence.

In this lab, you will write a program to solve peg solitaire: given a start and a final configuration, find a sequence of legal jumps connecting these configurations, or report if no such sequence exists. Since the problem of solving an arbitrary peg solitaire board is known to be NP-complete [3], a provably efficient polynomial-time algorithm for solving peg solitaire is unlikely to exist. Instead, in this lab you will implement a backtracking search algorithm, combined with some heuristics for pruning the search space.

```
FINDSOLUTION(start, final, path)
 1  if $start.numPegs \leq final.numPegs$
 2      return ($start = final$)
 3  else
 4      for each jump $J \in [0,n) \times [0,m) \times \{\texttt{NORTH}, \texttt{EAST}, \texttt{SOUTH}, \texttt{WEST}\}$
 5          if $J$ is a legal jump for $start$
 6              $start.makeMove(J)$
 7              $path.push(J)$
 8              $found$ = FINDSOLUTION($start, final, path$)
 9              if $found$
10                  return TRUE
11              else
12                  $start.makeReverseMove(J)$
13                  $path.pop()$
14      return FALSE
```

**Figure 2:** Pseudocode for a serial backtracking search to solve peg solitaire. The FINDSOLUTION method looks for a sequence of legal jumps connecting the *start* and *final* configurations, returning TRUE and the path if one exists, and FALSE otherwise.

## 2 Backtracking Search

One way to solve peg solitaire is to use a backtracking search algorithm, as shown in Figure 2. This algorithm searches for a solution from a given *start* configuration by making a legal jump *J* from *start*, after which it recursively searches for a solution from the new configuration. Since every jump reduces the number of pegs (filled holes) on the board by one, in the base case, the algorithm stops recursing when both *start* and *final* have the same number of pegs.

For this lab, we have provided code for a simple serial solver for peg solitaire. You can check out code using the following command:

```
git clone /afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab5/ lab5
```

This program takes in as input the starting and final board configurations from a text file, and searches for a solution using an algorithm similar to the one in Figure 2.

```
cagnode1:~$:  make peg_simple
cagnode1:~$:  ./peg_simple tests/test2.in
```

The `peg_test` application runs regression tests.

```
cagnode1:~$:  ./peg_simple --run_tests
```

For more details on the input format and on the current implementation, see the `README` file in the code directory.

**(a)** Construct some test inputs and benchmark the existing solver. For a fixed-size board, experiment and see if you can find the configurations which take the longest time to solve. For a worst-case configuration, how large of an input board can the solver finish searching within 2 minutes?

The backtracking algorithm in Figure 2 is correct, but can be inefficient because it does not make any attempt to prune the search space while looking for a solution. Pruning the search space is particularly useful for peg solitaire problems which have no solution; a simple solver may need to check many configurations to verify that no solution exists. For example, the code we provide does not solve the peg solitaire board shown in Figure 3 in a reasonable amount of time.

```
--XX--        --oo--
--XX--        --oo--
XXXXXX        oooooo
XXoXXX        oooXoo
XXXXXX        oooooo
--XX--        --oo--
--XX--        --oo--
 (a)           (b)
```

**Figure 3:** A peg solitaire problem on a 7-by-6 grid. This problem is solvable, but a simple solver without any heuristics takes a long time to find the solution.

Kiyomi and Matsui in [1] describe a heuristic for pruning the search space based on the number of times a particular jump can occur the solution sequence. Given particular start and final configurations, their heuristic is to compute an upper bound $R(J)$ on the number of times the jump $J = (i, j, dir)$ can appear in the solution. It turns out that this upper bound provides a useful way to prune the search space, since one need not search for solutions after a jump $J$ occurs more than $R(J)$ times. To compute this upper bound $R(J)$, one can use a (relatively small) integer linear program. (See [1] for more details.)

**(b)** We have extended the simple peg solver to compute the upper bounds $R(J)$ and use this quantity to prune the search space. Verify that this solver compiles and reports that the board shown in Figure 3 has a solution.

```
cagnode1:~$:  make peg
cagnode1:~$:  ./peg tests/long_test1.in
```

To compute $R(J)$, this solver uses the SYMPHONY library for mixed integer linear programming [2].

## 3 Optimizing the Solver

Your goal in this lab is to implement a peg solitaire solver which finds solutions as quickly as possible. Since different portions of the search space can be searched independently, conceptually, there is ample parallelism in a backtracking search that you can potentially exploit.

As written, however, the backtracking search in Figure 2 is serial, since it updates the *start* configuration and the current solution *path* in place when testing possible jumps $J$. Thus, in a parallel implementation, one may need to make copies of the board state to avoid race conditions and expose the potential parallelism. If one makes only a few copies, the parallelism of the algorithm may be limited and it may be difficult for the scheduler to efficiently load-balance the computation. On the other hand, repeatedly copying the board state introduces additional overhead as compared to the serial algorithm. One challenge for your implementation will be to find an appropriate trade-off between these options.

**(c)** Implement your own parallel peg solitaire solver which does not use Kiyomi and Matsui's heuristic. You may wish to optimize the provided implementation to use a more space-efficient representation of board configurations.

How large of an input board can your parallel solver finish searching within 2 minutes? What is the parallel speedup of your implementation?

**(d)** Modify your parallel peg solitaire solver to use Kiyomi and Matsui's heuristic to prune the search space. How large of an input board can the solver search within 2 minutes, assuming that the values of $R(J)$ for the problem have already been precomputed? What if one includes the time spent computing $R(J)$? What is the parallel speedup of your algorithm over the fastest serial solver you can implement?

Extend your solver by optimizing your implementation and finding solutions as quickly as possible. Some potential optimizations include:

- Accelerating the search by storing intermediate configurations in a hash table, and searching both forward from the start configuration and backwards from the final configuration.
- Speeding up the search by implementing other heuristics for pruning the search space. What is the impact of your heuristics on how easy it is to parallelize the search algorithm?
- Lazily computing the upper bounds $R(J)$ as needed, instead of relying on a precomputation step.

Be wary of unconsciously turning this assignment into your term project, however. You should spend only about 12 hours on this lab.

If you are interested in extending this lab into a term project, some ideas include:

- Parallelizing an integer linear solver such as SYMPHONY. In general, solvers rely on underlying branch-and-bound search algorithms which can themselves be parallelized.
- Parallelize another application which relies on a backtracking search (e.g., a SAT solver).
- Parallelizing a minmax searching algorithm for a game such as chess.

## References

[1] Masashi Kiyomi and Tomomi Matsui. Integer programming based algorithms for peg solitaire problems. In *Computers and Games*, page 229240. Springer-Verlag, 2001.

[2] Ted Ralphs and Menal Guzelsoy. The SYMPHONY callable library for mixed integer programming. In *Proceedings of the Ninth Conference of the INFORMS Computing Society*, 2005.

[3] Ryuhei Uehara and Shigeki Iwata. Generalized Hi-Q is NP-complete. *Transactions of the IEICE*, E73-E(2):270–273, February 1990.