EECS 592: Introduction to Artificial Intelligence, Fall 2017

Homework 1 – Knight's Tour

Assigned: Sept 11, 2017

Due: Sept 25, 2017 at 11:59 PM

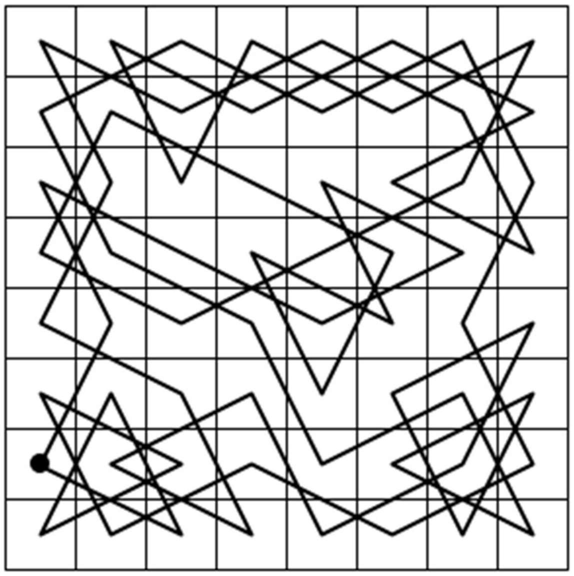
Submit your work to Canvas (https://umich.instructure.com). In the page for this course there is a tab for Assignments, under Homework 1 there will be a place to upload your files and answers. Please upload a single zip file containing all your files and answers. Below we give the details of what should be included.

Knight's Tour

"A **knight's tour** is a sequence of moves of a knight on a chessboard such that the knight visits every square only once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is *closed*, and otherwise it is *open*." – Wikipedia.

As a reminder, a chessboard consists of 8 rows and 8 columns (64 squares). A knight moves in an L, two squares in one direction and one in an orthogonal direction. For the purposes of this assignment, we define the squares that are a knight's move away from a square to be their *neighbors*. We define an *available* move from a square to be a move to a *neighbor* square that is not already a part of the current tour.

An example closed Knight's Tour, including a move back to the starting point:



In this problem, you will write a Java program to generate closed knight's tours as fast as possible. The program will use brute-force depth-first search, and then you will make modifications to it to incorporate constraints implicit in the problem so that it generates solutions faster. You will also experiment with a simple heuristic.

Here are some of the parameters of our version of the problem:

- The knight starts in position 2, 3 of an 8x8 board.
- A solution consists of a sequences of 64 legal moves for a knight that bring it to every square on the board and back to its origin on the last step.
- The approaches you will use will all be variants of depth-first search, where each extension of a sequence by a movement of the knight is a new state.
- Your program will run each of the variants, until each has visited 1,000,000 states (does not count backtracking).
- Your program will record the following:
  - Number of states visited (1,000,000)
  - Number of solutions found
  - The time (in microseconds) required.
- We will give a precise definition of the format of the output below.
- You will submit your program as a single Java source file, using no other libraries except java.util.
- We will run a program on the output of your program (format defined below) to confirm your solutions are all correct.

Strategies

You will be implemented a basic depth-first search, but with several possible strategies for how to choose the next move in a given state. All searches should start in location 2, 3. Here are the strategies we want you to implement:

0. A simple depth-first search where the program chooses randomly from the available moves at each step. This is useful for basic experimenting on small boards, but it is unlikely you will ever find an 8x8 solution this way. A key aspect of this is backing up when it hits failure (there are no available moves), and keeping track of which moves have been tried from a given square for this point in the search (so your system tries other moves after backup).
1. The degree of a square is the number of neighbors, for this square. For example, square 1, 1 has a degree of 2 as there are moves to squares 2, 3 and 3, 2. The initial degree varies from 2 (for 1, 1; 1, 8; 8, 1; 8, 8) to 8 (for squares at least two from a border).We define the *fixed degree* to be the number of available moves at the start of the search. This does not change during the search. For this strategy, your program must pick randomly from the available moves to unvisited squares and squares it has not yet searched as an extension to its current path that have the lowest *fixed* degree. For example, if a square has six moves with fixed degree of 3, 4, 6, 6, 4, 3, it will pick between the two with degree 3.
2. We define *dynamic degree* as the number of *available* moves from a square given the search so far. The dynamic degree is initialized to the fixed degree,

but every time the program moves to a square, it decrements the dynamic degree of all its neighbors that have not been visited yet. When your program reaches a dead end (or a solution!), and backs up out of a square, it must increment the dynamic degree of all its unvisited neighbors to reset them. Thus at any given moment the dynamic degree of a square is the number of its neighbors which are legal moves and are not yet part of the current path. With Strategy 2, your program will choose randomly from the *available* moves with the *lowest dynamic* degree (Warnsdorf's rule) (and hasn't backed up from along this path).

3. When selecting a move, if more than one of the *available* neighbors has a *dynamic degree of 1*, this is a failure (we leave it as an exercise for you to figure out why!), and your program should back up out of this square (updating the dynamic degree). Do not include the heuristic from #2, but you can include strategy #1 for cases where this one does not apply.
4. When selecting a move, if there is only one available neighbor with a dynamic degree of 1, move there. In addition, in backing up from such a move (after success or failure), the other available moves should not be tried, but the system should continue to back up to the previous move. That is because no other move besides to the move with the dynamic degree of 1 can ever lead to a successful path. Do not include #2 or #3, but you can include strategy #1 for cases where this one does not apply.
5. Combine 2, 3, and 4 into a single program so all of them are used at the same time.

Suggested approaches

Programming the Knight's Tour can be a rather complex problem. On an 8x8 board it may take 100's of millions of moves to find many solutions. Here we give some suggestions on how to attack the problem.

1. Read about Knight's Tour on the web.
2. Develop you depth-first search algorithm. We recommend an iterative approach, where there are moves and backups, as opposed to a recursive version. The recursive version can be conceptually simpler, but because of the stack management in Java, it could be much slower. We require that your searches always start at square 2, 3. This will make everyone's search more uniform and it speeds up some of the strategies.
3. Start small – Working with a 5x5 board is much more tractable where it is easy for you to track your program's moves. However, on a 5x5 board there are no closed solutions, only open ones. Wikipedia tells you how many open solutions there are, so see if you can match that number.
4. Move up to 6x6 – A 6x6 board has almost 10,000 closed ones. It's not necessary to find them all, but make sure you can easily get at least 100. Detecting that you have a closed solution can be tricky.
5. Move up to 8x8 – An 8x8 board has 1.9x10^16 open tours and 2.6x10^13 closed ones. Obviously, you are not going to find them all.

6. Test your results – We will be testing the output of your program to see that the solutions it produces are correct, so be sure you've confirmed that for yourself.
7. Timing measurements – Once you have your program finding correct tours reliably, you will need to instrument it to measure the time it is taking. For each run you should be able to measure: the number of moves made (states tried – do not include backups), the number of solutions found, and the number of seconds it took. Then you will calculate for each run the number of moves/second, the number of solutions/ second, and the number of moves/solution.

Analysis document

The main point of this homework is to study how the different strategies for choosing moves affect the performance of the search algorithm. Your submission will include a PDF document with some analyses of your results. For each of the strategies you implement, including numbers 0-5 above (we expect that for strategy 0 above, your program will generate 0 solutions) and any others you choose to try, we expect you to report the following information in your analysis document:

1. A description of the strategy. For 0-5 this can be based on our descriptions above)
2. Summary data for 30 runs on an 8x8 board, including the mean, median, and standard deviation of each of the following measures:
    a. The time it took in seconds
    b. The number of moves made [1,000,000]
    c. The number of moves/second
    d. The number of solutions/second
    e. The number of solutions found for each run

Once you have all these statistics gathered, make a plot, a bar chart is OK, that includes solutions/second and moves/second over the set of strategies you used. Write a paragraph of analysis on your takeaways from this graph: why does the data do what they do?

Output text file

In addition to the analysis document, you must provide us with a text file that reports 100 solutions on an 8x8 board for each strategy you used (assuming they found 100 solutions). The report of each strategy will include a single header line and 100 solution lines (or less if 100 were not found). At least one blank line should separate the reports for different strategies, and other than that, there should be no blank lines.

A typical report will conform to the following format:

```
KT: 8x8, strategy = 1, start = 2,3
001: 2,3 3,1 4,3 ... 2,3
001: 2,3 1,5 4,3 ... 2,3
...
100: 2,3 1,1 3,2 ... 2,3

KT: 8x8, strategy = 2, start = 2,3
...
```

Each line in the report should conform to the format above. Notice that a state is represented by the row number followed by a comma followed by the column number, with a space separating the states. These numbers are 1-based, so the upper left corner should be 1,1 and the lower right corner 8,8. Notice that each solution line will be 264 characters long, and that the starting square coordinates appear twice, at the beginning and the end.

Deliverables

On Canvas you will submit a single compressed file called <your-uom-unique-name>.zip. In that file we expect to find the following files:

1. `analysis.pdf` – This is your analysis document described above, as a PDF document.
2. `solutions.txt` – A text file reporting a single run for each strategy you used with 100 (if possible) solutions for that one run in the format given above.
3. `KnightsTour.java` – A single Java source file containing all your program code, and which does not refer to any external libraries other than java.util. We will probably not run this file, but we might. We will use a program to analyze these source files to detect plagiarism. We expect your code to be your own original work and not copied either from the web or from a friend.

Grading

If you do all the things described above, including getting reasonable data for strategies 1 through 5, you will get an A-. To get an A you must also do at least one of the following:

1. Implement an additional strategy that improves (decreases) the number of moves/solutions and document it along with the others.
2. Do both an iterative and a recursive implementation and compare their respective real-time performance on strategy 3. Why do you think one is faster than the other?