ComS 327 Project Part 3

Solitaire: determine process moves

Spring 2020

1 Summary

For the third part of the project, you must read an input file that describes a game configuration (possibly in the middle of a game). If the input file is valid, then you must find a sequence of moves that will advance the game configuration to a winning state, or indicate that the game is not winnable. For this part of the project, all source code must be C or C++.

Specifically, your `Makefile` must build your `check` executable from part 1, your `advance` executable from part 2, and a new executable named `winnable`. Program `winnable` should accept the following optional command-line switches and arguments, which may appear any number of times and in any order. If two or more switches conflict with each other, the last switch takes precedence. You may implement additional switches, for example to display debugging information, if you wish.

- Switch `-m N`, indicating that the search should be limited to sequences of at most N moves. If omitted, the default value of N is 1000 (this value is *extremely* large).
- Switch `-c`, indicating that a hash table *cache* should be used during the search (discussed in detail below). This switch is extra credit.
- Switch `-f`, indicating that safe moves to the foundations should be *forced* if they are available (discussed in detail below). This switch is extra credit.
- Switch `-v`, indicating *verbose* mode. If this switch is given, you are *strongly* encouraged to display diagnostic information to `stderr` periodically, so the TAs know that your code is not stuck in an infinite loop.

At most one filename argument may be passed, in any position. When testing your executable, we will not use filenames that begin with a `'-'` character. If no filename is given, the program should read from standard input (just like program `check` in part 1).

2 Determining if a game is winnable

For any game configuration, we say that a sequence of moves is a *winning sequence of length* `N` if, the sequence contains `N` moves, and starting from the game configuration, we can play each move in the sequence in order (it is a valid move), and at the end of the sequence, the game state satisfies the winning condition (see below). The goal of the program is to find a winning sequence of length `N` where `N` is not greater than the the maximum number of moves specified. If no such sequence exists, then your program should indicate this. Otherwise, your program should display one of the winning sequences. You do not need to find a *shortest* sequence.

2.1 Winning condition

Technically, a solitaire game is won when all cards are in the foundations. However, a game is *guaranteed* to be winnable if there are no covered cards, no stock cards, and at most one waste card. This should be the condition that you search for.

2.2 Depth first search

One way to examine all sequences of moves is to use *depth first search*. This can be done using recursion: given a game configuration, check all the single moves from that game configuration. If a move is valid, then this leads to a new game configuration; check that one recursively. The maximum number of moves is used to limit the depth of the recursion. If a winning game configuration is found, the recursion can be terminated, but this must be done carefully so that the winning sequence of moves just discovered can be written to standard output.

This is a brute-force approach that will work, but is not very computationally efficient, because the number of sequences to check can be huge.

2.3 Using a cache (`-c` switch: optional)

One of the problems of using the straightforward depth-first search is that, many different sequences may lead to the same game configuration. For example, if moves `2->f`, `4->f`, and `7->f` are all valid moves, then doing these in any order will lead to the same game configuration. Determining if the same game configuration is winnable within m moves, several times, leads to unnecessarily long computation times.

To fix this, one could use a hash table of game configurations (along with number of moves checked) to keep track of game configurations that were already checked: if we already know that game configuration G is not winnable within m moves, then we do not need to check again that G is winnable within n moves if $n \leq m$.

Doing this well means using a large (say, 10 million entries) hash table and finding a good hash function for game configurations. If two game configurations hash to the same location, it is reasonable to discard the older one. In this sense the hash table acts as a cache.

2.4 Forcing moves to the foundations (`-f` switch: optional)

Another way to reduce computation time is to reduce the number of choices of moves. But this must be done in such a way that we can still check if a game is winnable. Forcing moves to the foundations, if done correctly, can reduce the number of move choices and not affect if a winnable sequence can be found. It may, however, affect the *length* of the winnable sequence.

We say it is *safe* to move a card to the foundations if (1) the card is not being moved from the waste pile, unless we are turning 1 card over at a time; and (2) the current rank of the foundation suit we are adding to is at least as large as the ranks of the foundation cards on the opposite color. For example, it is safe to move `7s` to the foundations from the tableau, if the hearts and diamonds foundations have rank 6 or higher (because the current top of spades must be `6s`). If (say) the hearts foundation has top card `5h`, then it is not safe to move `7s` because we might need the `7s` in the tableau so we could play the `6h` on it. Some versions of computer solitaire will automatically move cards to the foundations using this kind of rule.

The `-f` switch should force safe moves to the foundations. Note that these moves must still be included in the move sequence. While this reduces the search space for a particular move depth, it could increase the number of moves required to win, because building up the foundations might not be needed to win the game.

3 Input

The input file format is the same as for parts 1 and 2. You may either ignore the `MOVES:` section of the input file, or consider the starting game state to be the state reached after processing all `MOVES:` in the input file. For this part of the project, your executable will be tested on input files with no moves. Example input files are shown in Section 6.

4 Output

The idea is that your program will write, to standard output, a portion of an exchange file that can be appended to the input file.

4.1 Unwinnable games

For games that are not winnable, or are not winnable in the specified number of moves, your program should write

```
# Game is not winnable within M moves
```

to standard output, where `M` is replaced by the number of moves checked. Nothing else should be written to standard output.

4.2 Winnable games

For games that are winnable, your program should write to standard ouput

```
# Game is winnable in M moves
```

followed by a sequence of moves (in exchange format) of length `M`. If the input file has no moves specified, then appending this output to the input file should produce a file that, when run through your `advance` executable, has all valid moves and leads to a winning game configuration. Example outputs are shown in Section 6.

5 Grading

The project is worth 100 points for individuals, and 110 points for groups of 2. You must handle the following `RULES`:

- `turn 1` (turn over one card at a time)
- `unlimited` (no limits to resetting the stock pile)

Additional rules are worth extra credit. The points breakdown for various features of the project are listed below. You may implement more features than required, for extra credit.

10 pts	`README` file, that describes implemented features
10 pts	`DEVELOPERS` file, that gives an overview of your implementation, including a breakdown of source files and functions, and who authored each function.
10 pts	Working `Makefile`. Typing "`make`" should build all executables.
6 pts	Reads from `stdin` if no filename argument
6 pts	Reads from filename passed as argument
3 pts	Output written to `stdout`
5 pts	Output "`# Game is winnable/not winnable`"
5 pts	Move sequences formatted correctly
10 pts	`-m` switch works
35 pts	Finds a winning sequence if one exists
5 pts	Works for `turn 3`
5 pts	Works for `limit R`
10 pts	`-f` switch works
20 pts	`-c` switch works

6 Examples

6.1 File testwin1.txt

```
#
# It is possible to reach the winning condition
#   (no covered cards, at most 1 waste card, no stock cards)
# for this game in 5 moves: . w->3 . . w->3
#
RULES:
  turn 1
  unlimited
FOUNDATIONS:
  2c
  4d
  3h
  3s
TABLEAU:
  | 4c
  |
  | Kc Qd Js Td 9s 8d 7c 6h 5s 4h 3c
  | 6s 5h
  | Kd Qc Jh Ts
  | Kh Qs Jd Tc 9d 8c 7d 6c 5d 4s
  | Ks Qh Jc Th 9c 8h 7s 6d 5c
STOCK:
| 9h 7h 8s
MOVES:
```

6.1.1 Output example 1

If we run

```
./winnable -m 4 testwin1.txt
```

then the output should be

```
# Game is not winnable within 4 moves
```

because there is no winning sequence of moves with length 4 or less.

6.1.2 Output example 2

If we run

```
./winnable -m 6 testwin1.txt
```

then the output could be

```
# Winnable in 6 moves:
  5->f
  .
  w->3
  .
  .
  w->3
```

where your output may be different depending on the order in which you explore the sequences of moves. Note that the output sequence shown here includes a useless move (the first move, “5->f”, can be omitted).

Also note that if this output is appended to the input file, we obtain a valid exchange-format file that could be fed into our `advance` executable for checking.

6.2 File `testwin2.txt`

```
RULES:
  turn 1
  unlimited
FOUNDATIONS:
  2c
  _d
  2h
  3s
TABLEAU:
  Ad 7d | 5d 4c 3h
  | Ks Qh Jc Th 9c 8d 7c 6d 5s 4h 3c 2d
  |
  | Js Td 9s 8h 7s 6h 5c 4d
  | Kh Qs Jd Tc 9d 8c
  | Kc Qd
  | Kd Qc Jh Ts 9h 8s 7h 6s 5h 4s 3d
STOCK:
6c |
MOVES:
```

6.2.1 Output example 3

If we run

```
./winnable -m 8 testwin2.txt
```

then the output could be

```
# Winnable in 8 moves:
  7->f
  6->4
  6->f
  1->7
  1->f
  1->f
  7->1
  7->3
```

6.2.2 Output example 4

If we run

```
./winnable -v -m 7 testwin2.txt
```

then the output could be

```
Using DFS search
1,000,000 configurations checked so far
1,535,072 configurations checked.
# Game is not winnable within 7 moves
```

where all lines except "`# Game is not winnable...`" are due to the `-v` switch and are sent to stderr. This test takes about 10 seconds for my executable.

6.3 File testwin3.txt

```
#
# Not winnable in any number of moves
#
RULES: turn 1 unlimited
FOUNDATIONS: 2c Ad 2h 4s
TABLEAU:
  6h 2d | Ts
  Qd 8s 3h | Kc Qh Js Td 9c 8h 7c 6d 5c 4h
  Jd Jc 9s | 8d
  8c 3c 9d | Kh Qc
  7s | Th
  | Kd Qs Jh Tc 9h
  | 6s 5h 4c 3d
STOCK:
  | 5d 7h Ks 4d 7d 6c 5s
MOVES:
```

6.3.1 Output example 5

If we run

```
./winnable -m 11 -v testwin3.txt
```

then the output could be:

```
Using DFS search
1,000,000 configurations checked so far
2,000,000 configurations checked so far
3,000,000 configurations checked so far
4,000,000 configurations checked so far
5,000,000 configurations checked so far
6,000,000 configurations checked so far
7,000,000 configurations checked so far
8,000,000 configurations checked so far
9,000,000 configurations checked so far
10,000,000 configurations checked so far
11,000,000 configurations checked so far
12,000,000 configurations checked so far
12,207,830 configurations checked.
# Game is not winnable within 11 moves
```

This test takes about a minute for my executable.

6.3.2 Output example 6

If the hash table cache is implemented (for extra credit), then running

```
./winnable -c -m 11 -v testwin3.txt
```

could instead produce the output:

```
Using DFS search
377 configurations checked.
Cache had 144 entries.
# Game is not winnable within 11 moves
```

This test takes about a second for my executable.

6.4 File testwin4.txt

```
RULES:
  turn 1
  unlimited
FOUNDATIONS:
  2c 3d 3h 3s
TABLEAU:
  4c 5h | Ts
  |
  | Kc Qd Js Td 9s 8d 7c 6h 5s 4h 3c
  | 6s
  | Kd Qc
  | Kh Qs Jd Tc 9d 8c 7d 6c 5d 4s
  | Ks Qh Jc Th 9c 8h 7s 6d
STOCK:
  5c 4d Jh 9h 7h | 8s
MOVES:
```

6.4.1 Output example 7

If the hash table cache is implemented (for extra credit), then running

```
./winnable -c -v -m 15 testwin4.txt
```

could produce output:

```
Using DFS search
1,000,000 configurations checked so far
1,165,094 configurations checked.
Cache had 717984 entries.
# Winnable in 15 moves:
  .
  r
  .
  w->1
  .
  w->f
  .
  w->3
  7->3
  7->4
  .
  w->3
  .
  .
  w->3
```

This test takes about ten seconds for my executable.

6.4.2 Output example 8

If the hash table cache *and* forced moves to foundations are implemented (for extra credit), then running

```
./winnable -f -c -v -m 20 testwin4.txt
```

could instead produce the output:

```
Using DFS search
392,569 configurations checked.
Cache had 177676 entries.
# Winnable in 20 moves:
  5->f
  5->f
  2->f
  5->f
  4->f
  .
  r
  .
  w->1
  .
  w->f
  .
  w->3
  7->3
  .
  w->3
  .
  .
  w->3
  7->f
```

This test takes about three seconds for my executable.

7 What to submit

7.1 In your git repository

Remember to include the following in your git repository.

- All TAs and the instructor as "reporters" (with read access).
- C and/or C++ Source code.
- A `Makefile`, so your executables can be built by simply typing "`make`". The TAs will **build and test your code on** `pyrite`.
- A `README` file, to indicate which features are implemented.
- A `DEVELOPERS` file, with necessary information about the source code for other developers. For group submissions, this file also should indicate which student(s) implemented which function(s).
- A tag with an appropriate name (e.g., "Part3", or "Part3a", "Part3b" in case you need to create more than one tag) as a frozen snapshot for the TAs to grade.

7.2 In Canvas

Please submit under the appropriate assignment in Canvas: either as an individual submission (you worked by yourself) or as a group submission (you worked in a group of 2 students). In the text submission box, indicate:

- The webpage URL for your (group's) git repository. Please indicate if this is a different repository than the one(s) you used previously.

- The name of the git tag for the TAs to grade.
- For group submissions, indicate which students (names and ISU netIDs) are part of the group.
- For group submissions, *all* students in the group should submit this information in Canvas.