

# Lab 1

CS 5233 – Fall 2007  
Tom Bylander, Instructor

assigned August 22, 2007  
due midnight, Sept. 26, 2007

In Lab 1, you will program the functions needed by algorithms for iterative deepening (ID) and iterative deepening A\* (IDA\*) for playing Freecell. You will compare the performance of the search algorithms and the heuristic functions.

## Freecell

Freecell is a solitaire card game commonly found on machines with MS Windows operating systems. A typical card deck has 52 cards, distinguished by 4 suits (spades, hearts, clubs, and diamonds) and 13 ranks (ace, 2–10, jack, queen, and king). In our version of Freecell, the task can be simplified by specifying a smaller number of ranks.

The object of the game is to move all the cards to the four “home cells,” using the four “free cells” as place holders. To win, you make four stacks of cards on the home cells: one for each suit, stacked in ascending order of rank. Our version of freecell will stop the game sooner, when all columns are in descending order of rank.

The Freecell game area consists of the four home cells, four free cells, and the deck of cards, which is dealt face-up in eight “columns” at the beginning of the game. With 52 cards, four columns will have seven cards, and four columns will have six cards.

There are three ways you can move cards in Freecell:

1. Any card from the bottom of a column can be moved to an empty free cell.
2. A card in a free cell or the bottom of a column can be moved to a home cell if it is an ace or if the card with the same suit and next lowest rank has been moved to its home cell.
3. A card in a free cell or the bottom of a column can be moved to the bottom of another column if the column is empty or if the current bottom of the column has a different color and the next highest rank.

## Environment

The environment program for Freecell is in `/home/bylander/agents/sun/bin/freecell`.<sup>1</sup> The source for this program and other programs are in the `/home/bylander/agents/sun/src` directory and in `/home/bylander/agents/agents.tar`. This was developed in Linux. For the Sun version, I made some minor changes to the `Makefile` in `agents.tar` so it would properly compile and install on the Suns. Also, for the CS Sun network, I used `/usr/local/bin/make` (GNU make) rather than `/usr/ccs/bin/make` (SunOS make).

To run the environment with user input, do the following:

---

<sup>1</sup>This is the binary for our Sun Lab. Replace “sun” with “linux” for our Linux Lab. I hope to have a “windows” version soon.

```
setenv A /home/bylander/agents/sun/bin or A=/home/bylander/agents/sun/bin
$A/interact "$A/freecell -v" $A/freecell_user
```

You will see the 4 empty free cells and 4 empty home cells, and the 8 columns of cards. The freecell program accepts a line of two characters as input, the first character specifies the column or cell from which a card is taken. The second character specifies the column or cell to which the card is moved. 1-8 specifies one of the 8 columns. A-D (or a-d) specifies one of the four free cells. F (or f) and H (or h) can only be used as the second character. F can be used to move to the first empty free cell. H (or h) specifies a move to the home cell of the card's suit.

The `interact` program reads the stdout of each program and sends it to the stdin of the other program. Any output to stderr will be displayed to the user. There are two exceptions. If the user types in a line, this line will be sent to both programs with the prefix `"user "`. If the `interact` program reads a line from a program that starts with `"user "`, then that is displayed to the user and not sent to the other program. You do not need to understand the `interact` program, but it might be an instructive exercise in learning Perl. Do not change `interact`. It has been carefully written so that it won't be the cause of any deadlock.

For the Freecell game, `freecell_user` has been written to minimally interact with the freecell program. It is instructive to run `freecell` and `freecell -v` by themselves to see the output of this program. Running `freecell -h` will provide some help.

Simple programs that might be instructive for environment/agent interaction are in `/home/bylander/agents/src/picknum`. Note that `fflush` is used after every line printed. In this directory, `picknum` is the environment, and `guessnum` is the agent. Running `picknum -h` will give you some help. Run the following to see the interaction between the two programs.

```
$A/interact -v $A/picknum $A/guessnum
```

## Agent

Your task is to write an agent program to interact with the freecell environment program. Your program will read the initial state, find a sequence of moves/operators that goes from the initial state to a goal state, and print the sequence of operators so that the environment ends up in a goal state.

We will use the iterative deepening (ID) and iterative deepening A\* (IDA\*) algorithms to do the search for us. Talking about ID and IDA\* are lectures ahead of us on the syllabus, so what I have done is to implement ID and IDA\* for you, leaving you the job to implement the functions that it needs.

ID and IDA\* are implemented as the functions `idsearch` and `idastar`, respectively in `/home/bylander/agents/src/slide/idastar.c` and its companion header file `idastar.h`. You should use this code verbatim. That means don't make any changes to your copy of `idastar.c` and `idastar.h` unless you like looking for trouble. This code expects to find a definition of the `State` data type in `state.h`. One is provided for you in `/home/bylander/agents/src/freecell/state.h`. In this case, a `State` specifies the cards in the free cells, home cells, and columns, where each card is represented by a integer from 0 to 51. Allocating states will be done at runtime.

The file `/home/bylander/agents/src/freecell/state.c` contains the basic functions for allocating and freeing states. The initial state should be created by calling `freecell_alloc` and initializing the values of the game area to the initial state.

The `idsearch` and `idastar` functions are called with the initial state and the 4 functions below, one of which has already been implemented for you. See `/home/bylander/agents/src/slide/8-puzzle.c` for an example (the call to `idastar` can be replaced with `idsearch`). A little more documentation can be found in `idastar.h`.

1. `freecell_expand`. This function inputs a state and returns an array of states that can be reached in one move. I.e., each state in this array differs from the given state by one move. It is up to `freecell_expand` to allocate new space for the array. The last element of the array should be set to `NULL`.
2. `freecell_free`. This is implemented for you in `state.c`.
3. `freecell_goal`. This inputs a state and should return 1 if the state is a goal state, otherwise it should return 0. The ID algorithm will take care of finding the optimal path to a goal state.
4. `freecell_equal`. This inputs two states and should return 1 if they are equal, otherwise 0. Two states are equal if they have the same configuration of cards. The order of cards in the free cells do not matter.
5. `freecell_cost`. This function inputs two states and returns the cost of the edge from the first state to the second state. For Lab 1, this function should always return 1. This function is ignored by `idsearch`.
6. `freecell_heuristic`. This functions inputs a state and returns an estimate of how many moves are needed to reach a goal state. This function is ignored by `idsearch`.

Here are one idea to implement a heuristic function. The goal for our version of freecell is to put each column in descending (but not strictly descending) order. For each column, determine how many cards need to be removed before the remaining cards in the column are in descending order. At least this many moves must be made from this column. The heuristic is to add up these numbers for the columns.

It will be interesting to test of version of this heuristic which simplies multiplies the result by 2.

These functions will be passed as arguments to the `idsearch` and `idastar` functions. Again see `/home/bylander/agents/src/slide/8-puzzle.c` for an example.

You might find the function implemented in `/home/bylander/agents/src/split_stdin.c` to be useful. The `split_stdin` function parses a line of stdin into an array of strings. Again see the 8-puzzle example.

The `freecell` program has an `-r` option to control the difficulty of the problem. This options specifies the highest card rank to be used. Debug using `-r 3`, then try higher values until your program takes too long to run. Use the `-s` option to set the random number seed, so that you can debug and compare performance on the same instance(s). You should be able to run your programs with command lines like:

```
$A/interact "$A/freecell -r 3" ./lab1id
$A/interact "$A/freecell -r 3" ./lab1ida1
$A/interact "$A/freecell -r 3" ./lab1ida2
```

I know there are a number of things to digest before anything will run. I will be providing additional help based on the questions I get from the class.

## Performance Analysis

For performance analysis, I would like you to measure and prepare a report on the following:

1. For `idsearch`:
  - (a) Measure the number of calls to `idsearch`. This can be measured by including a counter in the `freecell_goal` function because each call to `idsearch` makes one call to `freecell_goal`.
  - (b) Measure the CPU time (preferably not clock time) that is used by `idsearch` to do the search. In C, use the `clock` function.
  - (c) Record the length of the optimal solution path.
2. For `idastar` and each heuristic you implement:
  - (a) Measure the number of calls to `dfs_contour`. This can be measured by including a counter in the `freecell_heuristic` function because each call to `dfs_contour` makes one call to `freecell_heuristic`.
  - (b) Measure the CPU time (preferably not clock time) that is used by your Lab 1 program to do the search. In C, use the `clock` function.
  - (c) Record the length of the optimal solution path.

Be sure to use instances that span a wide range of CPU time.

Your report should comment on the performance of the different search implementations.

Remember that the `-r` value is not the same as the length of the path from initial to final state. Also note that particular values for `-r` and `-s` on a particular machine will always result in the same instance.

## Turning in Your Lab

Somewhere in your directory, you should create a `lab1` subdirectory. This directory must have a `Makefile` (or *must* have an executable file `make_lab1`) that compiles and links your agent programs, which *must* be named `lab1id`, `lab1ida1`, `lab1ida2`, etc. Any source code that is linked to these programs *must* be in this directory. That is, you should be able to copy the files in your `lab1` directory to another directory and be able to run the following command sequence.

```
setenv A /home/bylander/agents/sun/bin or A=/home/bylander/agents/sun/bin
/bin/rm lab1id
source /etc/.login revert to default values for shell variables
make or ./make_lab1
$A/interact "$A/freecell -m 3" "./lab1id"
```

Your `lab1` directory should have a report named `lab1.html`, `lab1.pdf`, or `lab1.ps`, depending on whether it is in HTML, PDF, or Postscript format, respectively. I do not want to read other special formats such as Word files, Excel files, Latex files, etc. The HTML files can link to GIF or JPG pictures in your directory.

Zip or tar the entire directory and submit it using WebCT. If you submit multiple times, only the last one is kept. Be sure that WebCT has registered your submission.

## Grading

Task	Percentage of Grade
ID Implementation	30%
IDA* Implementation	
First Heuristic	10%
Second Heuristic	10%
Report	
Quality of Writing	20%
Quality of Measurements	20%
Quality of Conclusions	10%