

# Lab 1

CS 3793 – Fall 2008  
Tom Bylander, Instructor

assigned August 27, 2008  
due midnight, October 3, 2008

In Lab 1, you will program the routines for performing heuristic search on the “solitaire Mancala” problem. The general search procedure is provided for you. What you have to do is to define the functions that it needs. Your grade on the lab will depend on the performance of your lab when I run it on test problems.

## Solitaire Mancala

Solitaire Mancala is a game that I made up based on the two-person game of Mancala (which we will play for Lab 3). The object of the game is to move the “stones” from your pockets to your Mancala using as few moves as possible. The rules are described in more detail below.

## Environment

The environment program for solitaire Mancala is `/home/bylander/agents/bin/smancala`. The source for this program and other programs are in the `/home/bylander/agents/src` directory and in `/home/bylander/agents/agents.tar`. This was developed in Linux. I made some minor changes to `Makefile` so it would properly compile and install on the Suns.

To run the environment, do the following:

```
setenv A /home/bylander/agents/linux/bin or A=/home/bylander/agents/linux/bin
$A/smancala -n 5
```

The `smancala` program will first print out the instance of the problem, which will look like:

```
0 0 0 2 3 0
```

The 6 numbers show the number of stones in the 6 pockets, from pocket 1 to pocket 6. Now the `smancala` program expects to read in a pocket number, i.e., the pocket whose stones are going to be moved.

```
0 0 0 2 3 0
5
1 0 0 2 0 1
6
1 0 0 2 0 0
1
0 1 0 2 0 0
2
0 0 1 2 0 0
3
0 0 0 3 0 0
```

```

4
0 0 0 0 1 1
6
0 0 0 0 1 0
5
0 0 0 0 0 1
6
0 0 0 0 0 0

```

In my first move, I chose pocket 5. In this move, all the stones in that pocket are “picked up” and then, one stone is placed in each pocket to the right. If there still is a stone at the end of the pockets, then one stone is placed in the Mancala. If there are any stones left, then one stone at a time is placed in the pockets starting at the left and ending with the Mancala until there are no more stones. In this move, the 3 stones in pocket 5 are put into pocket 6, the Mancala, and pocket 1. Note that 9 moves were used to move all 5 stones to the Mancala.

In this lab, we want a program to read in the initial information, search for and find an optimal sequence of moves, and print out the sequence one line at a time. We can set this up using the `interact` program in `/home/bylander/agents/linux/bin`. The `interact` program inputs two programs on the command line, and then it reads stdout of each program and prints it to stdin of the other program. Any output to stderr will be printed to the user. There are two exceptions. If the user types in a line, this line will be printed to both programs with the prefix “user”. If the `interact` program reads a line that starts with “user”, then that is printed to the user and not to the programs. 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.

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 (otherwise, the programs would deadlock with unflushed buffers). 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 `smancala` environment program. Your program will read the initial state (the number of stones in each pocket), find the optimal sequence of moves, and print the moves to the environment.

We will use the IDA\* algorithm to do the search for us. Talking about IDA\* is a couple of weeks ahead of us on the syllabus, so what I have done is to implement IDA\* for you, leaving you the job to implement the functions that it needs.

IDA\* is implemented 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 really like debugging. 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/mancala/state.h`. In this case, a `State` is a pointer corresponding to an array of 7 integers. Allocating states will be done at runtime. The integers store the number of stones in each pocket, and, if you wish, the number of stones in the Mancala. The search will be performed by trying one move at a time.

The file `/home/bylander/agents/src/mancala/state.c` contains the basic functions for allocating and freeing states. The initial state should be created by calling `state_alloc` and initializing the values of the array to the number of stones in each pocket.

The `idastar` function is called with the initial state and a data structure containing the 6 functions described below. 5 of the functions are left to you to define. See `/home/bylander/agents/src/slide/8-puzzle.c` for an example. A little more documentation can be found in `idastar.h`.

1. `state_expand`. This function inputs the “current” state and returns an array of states, which are called “successors”. Each is a copy of the current state (see `state_copy` in `state.c`) with one additional move performed. The last element of the array should be a NULL pointer.
2. `state_free`. This is implemented for you in `state.c`.
3. `state_goal`. This inputs a state and should return 1 if the state is a “goal” state, otherwise it should return 0. A state is goal state if there are no stones in any of the pockets. The IDA\* algorithm will take care of finding the optimal goal state.
4. `state_equal`. This inputs two states and should return 1 if they are equal, otherwise 0. Two states are equal if they have the same number of stones in each pocket.
5. `state_heuristic`. This inputs a state and should return an estimate of the number of moves that need to be performed to finish the game. A function that always returns 0 will work for a small number of stones (less than 10 or so), but will make the search very inefficient for a larger number of stones.
6. `state_cost`. This inputs a parent state and a child state and returns the “cost” of going from the parent state to a child state. For this lab, this should always return 1. In other search problems, there might be different costs for different moves.

These functions will be put into a data structure, which is then passed as an argument to the `idastar` function. 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.

## Heuristic Function

Your heuristic function should return the total number of stones in the pockets, except that if there are more than 18 stones, then it returns the total stones minus one. The simpler heuristic of returning the total stones in all cases can overestimate the number of moves. For

example, one or more pockets might have so many stones that a single move can put more than one stone in the Mancala, and all other moves put one stone apiece in the Mancala. This is possible with just 19 stones, so the total number minus one is safer when there is more than 18 stones.

Your program should print out the number of states that were visited in the form:

```
printf("user %d states were visited.\n", states_visited);
```

The variable `states_visited` should be incremented for every call to `state_goal`.

## Turning in Your Lab

Somewhere in your directory, you should create a `lab3` 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 `lab1`. 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/linux/bin or A=/home/bylander/agents/linux/bin
/bin/rm lab1
source /etc/.login revert to default values for shell variables
make or ./make_lab1
$A/interact $A/smancala ./lab1
```

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.

In addition, provide a brief report describing the performance of your program.

I will be running your lab on successively larger instances and will be looking at the quality of your answers and the amount of time it takes. You will get a grade of 0 until I can run your lab.