Programming Mini-Project 1

Uninformed & Informed Search; Game Playing

Max possible score:

- 4308: 150 Points
- 5360: 150 Points

Part 1

Max: [4308: 75 Points, 5360: 75 Points]

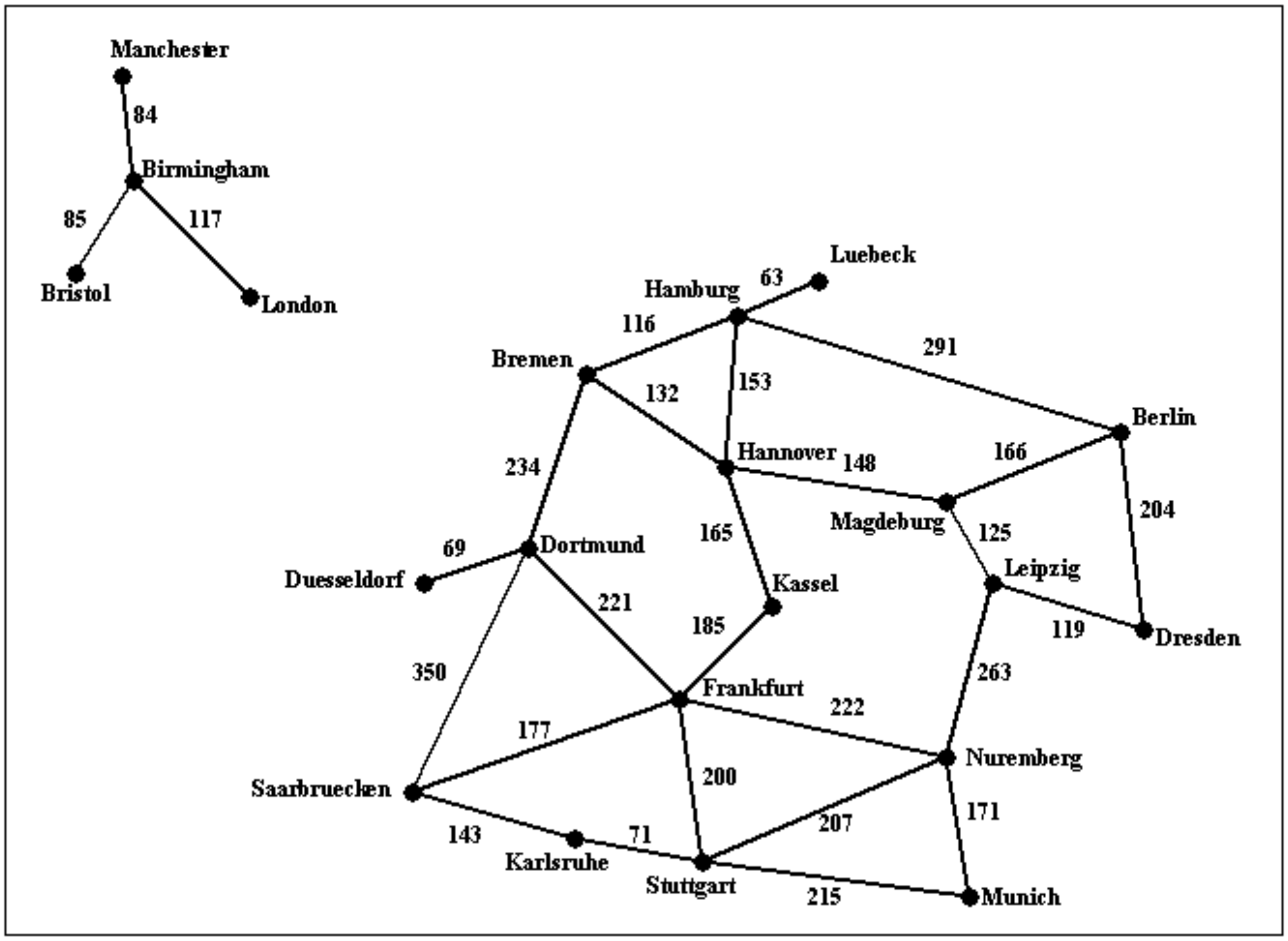


Figure 1: Graphical representation of information in input1.txt

Implement a search algorithm that can find a route between any two cities. Your program will be called find_route, and will take exactly commandline arguments as follows:

find_route input_filename origin_city destination_city heuristic_filename

An example command line is:

```
find_route input1.txt Bremen Kassel (For doing Uninformed search)
or
find_route input1.txt Bremen Kassel h_kassel.txt (For doing Informed search)
```

If heuristic is not provided then program must do uninformed search. Argument input_filename is the name of a text file such as, that describes road connections between cities in some part of the world. For example, the road system described by file input1.txt can be visualized in Figure 1 shown above. You can assume that the input file is formatted in the same way as input1.txt: each line contains three items. The last line contains the items "END OF INPUT", and that is how the program can detect that it has reached the end of the file. The other lines of the file contain, in this order, a source city, a destination city, and the length in kilometers of the road connecting directly those two cities. Each city name will be a single word (for example, we will use New_York instead of New York), consisting of upper and lowercase letters and possibly underscores.

IMPORTANT NOTE: MULTIPLE INPUT FILES WILL BE USED TO GRADE THE ASSIGNMENT, FILE IS JUST AN EXAMPLE. YOUR CODE SHOULD WORK WITH ANY INPUT FILE FORMATTED AS SPECIFIED ABOVE.

The program will compute a route between the origin city and the destination city, and will print out both the length of the route and the list of all cities that lie on that route. It should also display the number of nodes expanded and nodes generated. For example,

```
find_route input1.txt Bremen Kassel
```

should have the following output:

```
nodes expanded: 12
nodes generated: 20
distance: 297.0 km
route:
Bremen to Hannover, 132.0 km
Hannover to Kassel, 165.0 km
```

and

```
find_route input1.txt London Kassel
```

should have the following output:

```
nodes expanded: 7
nodes generated: 7
distance: infinity
route:
none
```

For full credit, you should produce outputs identical in format to the above two examples.

If a heuristic file is provided then program must perform Informed search. The heuristic file gives the estimate of what the cost could be to get to the given destination from any start state (note this is just an estimate). In this case the command line would look like

```
find_route input1.txt Bremen Kassel h_kassel.txt
```

Here the last argument contains a text file what has the heuristic values for every state wrt the given destination city (note different destinations will need different heuristic values). For example, you have been provided a sample file h_kassel.txt which gives the heuristic value for every state (assuming kassel is the goal). Your program should use this information to reduce the number of nodes it ends up expanding. Other than that, the solution returned by the program should be the same as the uninformed version. For example,

```
find_route input1.txt Bremen Kassel h_kassel.txt
```

should have the following output:

```
nodes expanded: 3
nodes generated: 8
distance: 297.0 km
route:
Bremen to Hannover, 132.0 km
Hannover to Kassel, 165.0 km
```

Suggestions

Grading

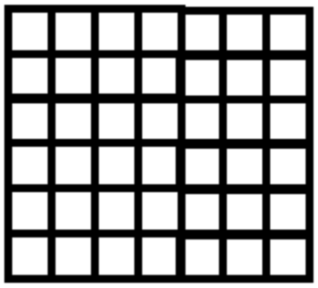
The assignments will be graded out of 75 points.

- 35 points: The program always finds a route between the origin and the destination, as long as such a route exists.
- 15 points: The program terminates and reports that no route can be found when indeed no route exists that connects source and destination (e.g., if source is London and destination is Berlin, in the above example).
- 15 points: In addition to the above requirements, the program always returns optimal routes. In other words, no shorter route exists than the one reported by the program.
- 10 points: Correct implementation of any informed search method. Please note that you only need to make one submission that meets this and all previous requirements to get credit for all the parts.
- Negative points: penalty points will be awarded by the instructor and TA generously and at will, for issues such as: submission not including precise and accurate instructions for how to run the code, wrong compression format for the submission, or other failures to comply with the instructions given for this assignment. Partial credit for incorrect solutions will be given ONLY for code that is well designed and well documented. Code that is badly designed and badly documented can still get credit only as long as it accomplishes the required tasks.

Part 2

Max: [4308: 75 Points, 5360: 75 Points]

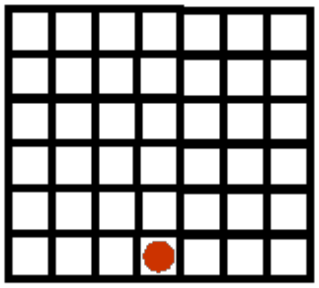
Last move: none



Score: 0-0
(red first)

Moves so far: 0

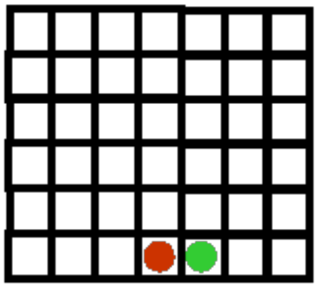
Last move: red



Score: 0-0
(red first)

Moves so far: 1

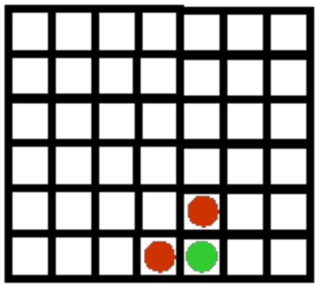
Last move: green



Score: 0-0
(red first)

Moves so far: 2

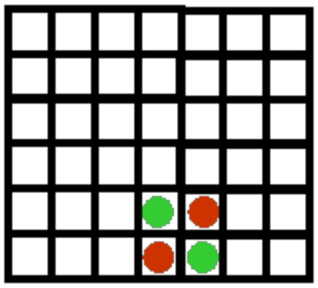
Last move: red



Score: 0-0
(red first)

Moves so far: 3

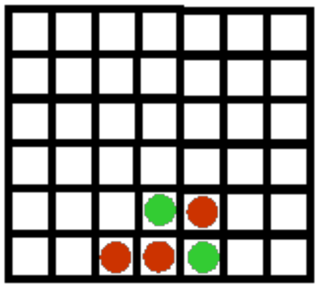
Last move: green



Score: 0-0
(red first)

Moves so far: 4

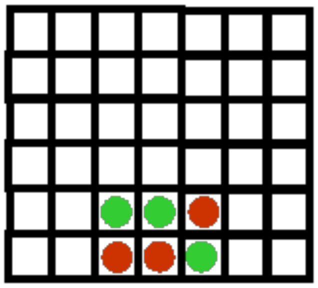
Last move: red



Score: 0-0
(red first)

Moves so far: 5

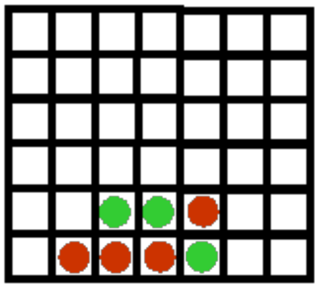
Last move: green



Score: 0-0
(red first)

Moves so far: 6

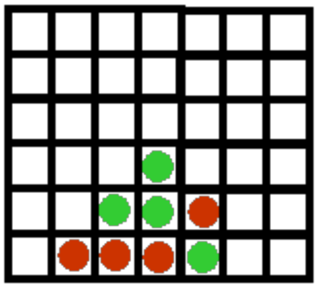
Last move: red



Score: 0-0
(red first)

Moves so far: 7

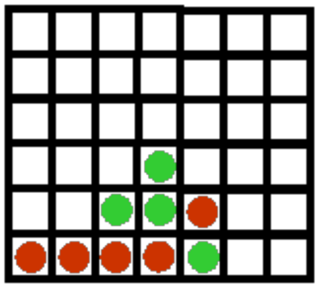
Last move: green



Score: 0-0
(red first)

Moves so far: 8

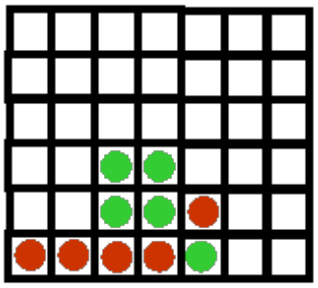
Last move: red



Score: 1-0
(red first)

Moves so far: 9

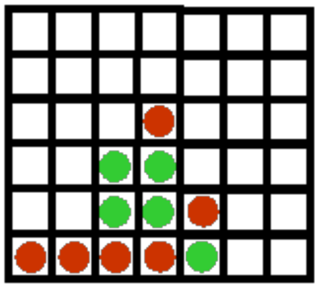
Last move: green



Score: 1-0
(red first)

Moves so far: 10

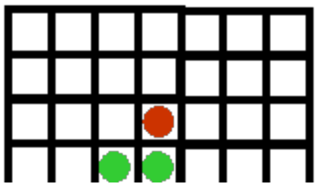
Last move: red



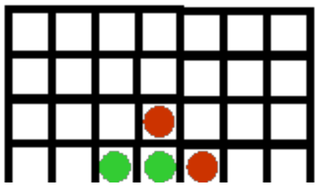
Score: 1-0
(red first)

Moves so far: 11

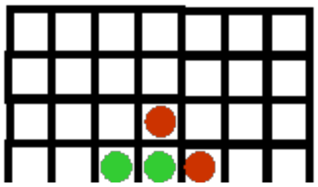
Last move: green



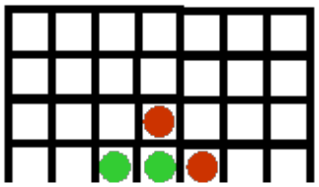
Last move: red



Last move: green



Last move: red



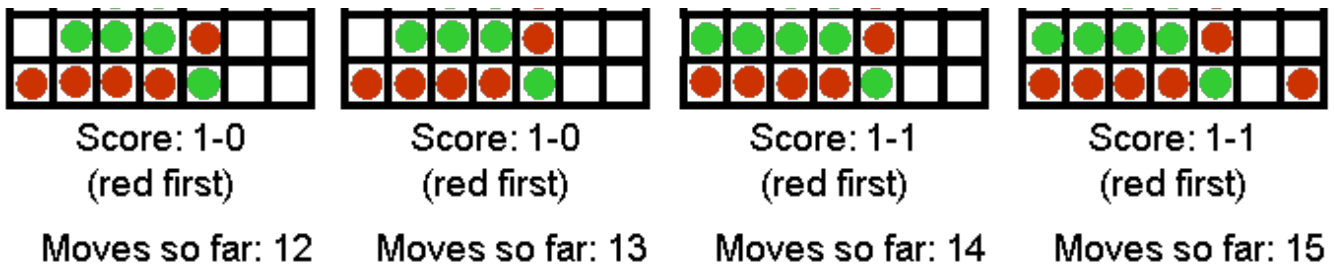


Figure 2: Examples of Moves made in a game of Max-Connect4

The task in this programming assignment is to implement an agent that plays the Max-Connect4 game using search. Figure 2 above shows the first few moves of a game. The game is played on a 6x7 grid, with six rows and seven columns. There are two players, player A (red) and player B (green). The two players take turns placing pieces on the board: the red player can only place red pieces, and the green player can only place green pieces.

It is best to think of the board as standing upright. We will assign a number to every row and column, as follows: columns are numbered from left to right, with numbers 1, 2, ..., 7. Rows are numbered from bottom to top, with numbers 1, 2, ..., 6. When a player makes a move, the move is completely determined by specifying the COLUMN where the piece will be placed. If all six positions in that column are occupied, then the move is invalid, and the program should reject it and force the player to make a valid move. In a valid move, once the column is specified, the piece is placed on that column and "falls down", until it reaches the lowest unoccupied row in that column.

The game is over when all positions are occupied. Obviously, every complete game consists of 42 moves, and each player makes 21 moves. The score, at the end of the game is determined as follows: consider each quadruple of four consecutive positions on board, either in the horizontal, vertical, or each of the two diagonal directions (from bottom left to top right and from bottom right to top left). The red player gets a point for each such quadruple where all four positions are occupied by red pieces. Similarly, the green player gets a point for each such quadruple where all four positions are occupied by green pieces. The player with the most points wins the game. Your program will run in two modes: an interactive mode, that is best suited for the program playing against a human player, and a one-move mode, where the program reads the current state of the game from an input file, makes a single move, and writes the resulting state to an output file. The one-move mode can be used to make programs play against each other. Note that THE PROGRAM MAY BE EITHER THE RED OR THE GREEN PLAYER, THAT WILL BE SPECIFIED BY THE STATE, AS SAVED IN THE INPUT FILE.

As part of this assignment, you will also need to measure and report the time that your program takes, as a function of the number of moves it explores.

Interactive Mode

In the interactive mode, the game should run from the command line with the following arguments (assuming a Java implementation, with obvious changes for C++ or other implementations):

java maxconnect4 interactive [input_file] [computer-next/human-next] [depth]

For example:

java maxconnect4 interactive input1.txt computer-next 7

- Argument interactive specifies that the program runs in interactive mode.
- Argument [input_file] specifies an input file that contains an initial board state. This way we can start the program from a non-empty board state. If the input file does not exist, the program should just create an empty board state and start again from there.
- Argument [computer-first/human-first] specifies whether the computer should make the next move or the human.
- Argument [depth] specifies the number of moves in advance that the computer should consider while searching for its next move. In other words, this argument specifies the depth of the search tree. Essentially, this argument will control the time takes for the computer to make a move.

After reading the input file, the program gets into the following loop:

1. If computer-next, goto 2, else goto 5.
2. Print the current board state and score. If the board is full, exit.
3. Choose and make the next move.
4. Save the current board state in a file called computer.txt (in same format as input file).
5. Print the current board state and score. If the board is full, exit.
6. Ask the human user to make a move (make sure that the move is valid, otherwise repeat request to the user).
7. Save the current board state in a file called human.txt (in same format as input file).
8. Goto 2.

One-Move Mode

The purpose of the one-move mode is to make it easy for programs to compete against each other, and communicate their moves to each other using text files. The one-move mode is invoked as follows:

java maxconnect4 one-move [input_file] [output_file] [depth]

For example:

java maxconnect4 one-move red_next.txt green_next.txt 5

In this case, the program simply makes a single move and terminates. In particular, the program should:

- Read the input file and initialize the board state and current score, as in interactive mode.
- Print the current board state and score. If the board is full, exit.
- Choose and make the next move.
- Print the current board state and score.
- Save the current board state to the output file **IN EXACTLY THE SAME FORMAT THAT IS USED FOR INPUT FILES**.
- Exit

Sample code

The sample code needs an input file to run. Sample input files that you can download are input1.txt and input2.txt. You are free to make other input files to experiment with, as long as they follow the same format. In the input files, a 0 stands for an empty spot, a 1 stands for a piece played by the red player, and a 2 stands for a piece played by the green player. The last number in the input file indicates which player plays NEXT (and NOT which player played last). Sample (omega-compatible) code is available in:

- Java: download files here. Compile using:

  ```
  javac maxconnect4.java GameBoard.java AiPlayer.java
  ```

  An example command line that runs the program (assuming that you have input1.txt saved in the same directory) is:

  ```
  java maxconnect4 one-move input1.txt output1.txt 10
  ```

- C++: download file here. Compile using:

  ```
  g++ -o maxconnect4 maxconnect.cpp
  ```

  An example command line that runs the program (assuming that you have input1.txt saved in the same directory) is:

  ```
  maxconnect4 one-move input1.txt output1.txt 10
  ```

- Python (Version 2.4): download file here.

  An example command line that runs the program (assuming that you have input1.txt saved in the same directory) is:

  ```
  ./maxconnect4.py one-move input1.txt output1.txt 10
  ```

The sample code implements a system playing max-connect4 (in one-move mode only) by making random moves. While the AI part of the sample code leaves much to be desired (your assignment is to fix that), the code can get you started by showing you how to represent and generate board states, how to save/load the game state to and from files in the desired format, and how to count the score (though faster score-counting methods are possible). You are welcome to use any part of this sample code in your implementation or ignore it completely and write your code from scratch.

Measuring Execution Time

You can measure the execution time for your program on linux/mac machines by inserting the word "time" in the beginning of your command line. For example, if you want to measure how much time it takes for your system to make one move with the depth parameter set to 10, try this:

```
time java maxconnect4 one-move red_next.txt green_next.txt 10
```

Your output will look something like:

```
real    0m0.003s
user    0m0.002s
sys     0m0.001s
```

Out of the above three lines, the **user** line is what you should consider.

Windows machines do not have a similar command. However, you can approximate it by setup a batch script like this:

```
@echo off
set startTime=%time%
java maxconnect4 one-move red_next.txt green_next.txt 10
echo Start Time: %startTime%
echo Finish Time: %time%
```

This will display text similar to the following after the output from your file

```
Start Time:  1:17:14.37
Finish Time:  1:17:14.44
```

The difference between the two gives you the time (0.07s).

Grading

The assignments will be graded out of 75 points.

- 30 points: Implementing plain minimax.
- 20 points: Implementing alpha-beta pruning (if correctly implemented, you also get the 30 points for plain minimax, you don't need to have separate implementations for it).
- 15 points: Implementing the depth-limited version of minimax (if correctly implemented, and includes alpha-beta pruning, you also get the 30 points for plain minimax and 20 points for alpha-beta search, you don't need to have separate implementations for those). For full credit, you obviously need to come up with a reasonable evaluation function to be used in the context of depth-limited search. A "reasonable" evaluation function is defined to be an evaluation function that allows your program to consistently beat a random player.
- 10 points: Include in your submission an table of CPU runtime (for making a single move) vs depth, when the board is empty (input1.txt). Your table should include every single depth, until (and including) the first depth for which the time exceeds one minute.

How to submit

For each part: Implementations in C, C++, Java, and Python will be accepted. Points will be taken off for failure to comply with this requirement.

Create a ZIPPED directory called <net-id>_proj1.zip (no other forms of compression accepted, contact the instructor or TA if you do not know how to produce .zip files). The directory should contain two folders (one for each part). Each folder should contain source code for each part. Each folder should also contain a file called readme.txt, which should specify precisely:

- Name and UTA ID of the student.
- What programming language is used for this task. (also mention if the code is omega compatible)
- How the code is structured.
- How to run the code, including very specific compilation instructions, if compilation is needed. Instructions such as "compile using g++" are NOT considered specific if the TA needs to do additional steps to get your code to run.

- Insufficient or unclear instructions will be penalized.
- **Code that the TA cannot run gets AT MOST 75% credit**.