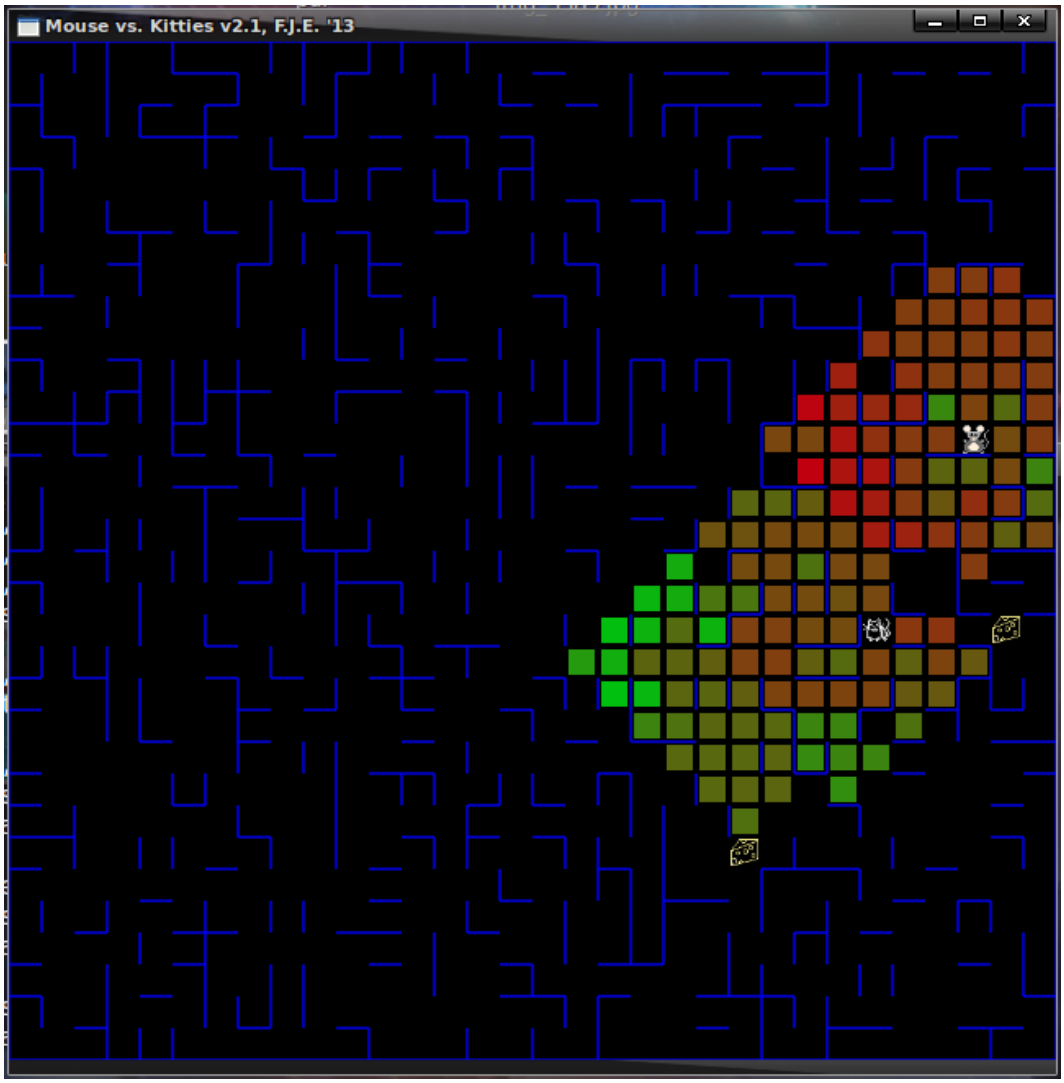
Assignment 2
Adversarial Games

(electronic submission on Quercus)

This assignment can be completed individually, or by a team of 2 students



CSC D84 – Assignment 2
Game Trees and Mini-Max

On assignment 1 you got to build clever mice that could run around a few not-too-bright kitties using ***search algorithms.***
In this assignment, you will learn to program agents that try to outsmart each other at every turn. Your code will determine the actions of both the mouse and the cats, and you will learn how to determine optimal sequences of actions for each agent that ***take into account what the other agents in the game will do.***
The result will be smart agents that do their best to win at every move!

Learning Objectives:

You will understand how the Mini-Max algorithm works, and how to use it to plan actions in adversarial games.

You will think about, and gain experience designing a good ***utility function***. You will see how this function affects the evolution of the game.

You will study empirically the complexity of exploring the game tree, and realize that even for simple games it is impractical to plan more than a handful of steps in advance.

You will understand how to put ***bounds*** on the expected utility value of a node, and how these bounds can be used to determine whether to continue searching from that node.

You will apply ***alpha-beta*** pruning to reduce the amount of search required to find a promising sequence of actions.

Skills Developed:

Thinking in terms of ***utility***. Deciding what factors to consider within the utility, and how to balance positive and negative factors to increase the chance of success for the mouse.

Implementing Mini-Max search, which is standard tool used in adversarial games.

Implementing alpha-beta pruning, a useful technique for early search termination.

Reference material:

Your in-class lecture notes on search.

Your course instructor and your TA!

Building Smarter Mice

In this assignment you will develop the framework for effectively playing an adversarial game. Specifically, you will explore the problem of determining the optimal sequence of moves for the mouse if the mouse's goal is to eat the cheese and not get eaten. What makes this interesting Is that this time around ***the cats know what the mouse's goal is*** and are working hard to prevent it from reaching the goal.

This is a typical setup for an adversarial game. Other common examples include ***Chess***, ***checkers***, and ***Go***.

There are roughly three components to think about in this assignment

- Designing and implementing a ***utility function*** that evaluates the 'quality' of each maze position in terms of the mouse's goals. This determines to a large extent the behaviour of the mouse and the cats.
- Writing a search algorithm that will determine the optimal sequence of moves. Unlike the search methods in A1; however, ***this time the search must account for the cat's potential actions***
- Optimizing your search algorithm via a ***branch and bound*** technique called ***alpha-beta pruning*** that significantly reduces the amount of search.

All the code to implement the initialization and updates is given to you. You only need to add code for two functions that do all the heavy lifting.

Step 1

Download and uncompress the starter code into an empty directory.

This code is designed for Linux.
Our, you can install Linux on your laptop/desktop machine. This code runs on Ubuntu based Linux distros from 12.04 onward.

The starter code contains the following files:

board_layout.h
MiniMax_search.c
MiniMax_search.h
MiniMax_search_core_GL.o
compile.sh
REPORT.TXT
**** a set of .ppm images used by the code***

Read all the comments in MiniMax_search.c. The comments describe what needs to be implemented, and the conditions and constraints placed on your solution.

Feel free to ask for clarification at any time. But I expect you to have ***read this handout and all the comments carefully***.

Building Smarter Mice

Step 2

Test-run the starter code:

1) Compile the code with the script provided

2) From the command shell, execute

```
>./MiniMax_search 2415 1 1 0 10 0
```

Parameters of the code are: random seed, number of cats, number of cheese, search mode, maximum depth, and cat smartness. The code will print information about these if called with no parameters.

You should see a maze, cats wandering randomly, and a static mouse. Press 'q' to quit.

Step 3

Read and understand the starter code in ***MiniMax_search.c***. This file specifies what each of the two functions you will work on is supposed to do. The behaviour of each function is carefully documented, and ***you have to ensure your code complies with the specs.***

Do not start coding until you have understood what you are being asked to implement.

Note: While you work on your solution, use a fixed random seed, one cat, and one cheese. Try changing things around once you have working mini-max code.

Building Smarter Mice

Step 4

Understand the GameState

It is important that you understand that the mini-max function is recursive, and each recursive call has its own game state corresponding to the positions of agents at some point in the game. You must ensure that each recursive call is passed the state it is supposed to use.

For example, assume you are working at level 0, for the mouse's move. You will need to call recursively the mini-max function to evaluate the score of all possible mouse moves, top, right, bottom, and left.

This will require calling the mini-max function recursively. You must ensure that the mouse position you pass to the recursive call corresponds to the mouse position you want to evaluate!

The same goes for cat moves – if your current level needs to evaluate the score for different cat moves, recursive calls to mini-max must have that cat's position updated accordingly. Pay close attention to the starter code. And ***print*** the positions of agents at different levels of the recursion to make sure things are being handled properly.

Step 5

Write the utility function – this is the part that requires a human designer!

You can not test the MiniMax component of the assignment without a utility function, so start by implementing this bit. See the starter to find out what information is available to you for this. Think hard about how to write a good utility function! The success of your mouse depends on it.

- Your utility function should return a positive value when the game configuration is favorable to the mouse.
- It should return a negative value when the game configuration is bad for the mouse (or, another way to think about it, when it favours the cats).

You are free to interpret the above instructions in any ***reasonable*** way. And I won't give you specific hints. I want you to think about this carefully and write ***a good utility function***. you will document and explain your utility function design in ***REPORT.TXT***.

Note that the amount of 'smartness' you will see in your mouse is directly affected by how good an utility function you can write.

Implementation suggestion: Write a ***simple*** utility function first, something easy to test that you can use while writing the ***MiniMax*** code. Once you are sure that mini-max is working properly go back and make your utility function really ***crunchy***.

Building Smarter Mice

Step 6

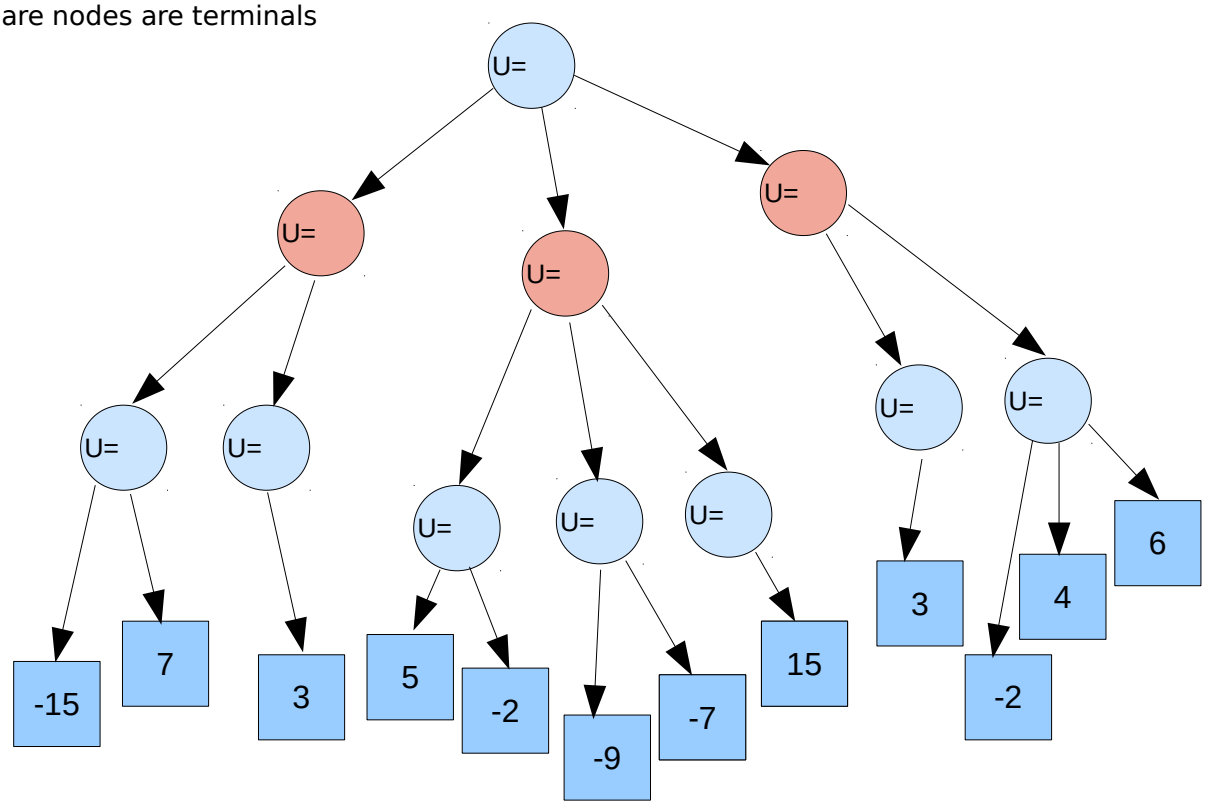
Write the MiniMax procedure – you can use A.I. tools to help you with this part

The core of this assignment consists of implementing the mini-max search algorithm to determine an optimal sequence of moves for each agent. ***Review your notes on mini-max*** and come to see me if you have questions. ***Do not start coding until you are sure you understand how mini-max works.***

As an exercise for you to test your understanding of the mini-max process, complete the diagram below by writing down the missing ***utilities*** at each node (note that for terminal nodes the utilities are already given!)

You do not have to hand this in, it will not be graded. It is just for you to realize either that you fully understand mini-max, or that you need help and should come and talk to us.

BLUE circle nodes are MAX nodes
RED circle nodes are MIN nodes
square nodes are terminals



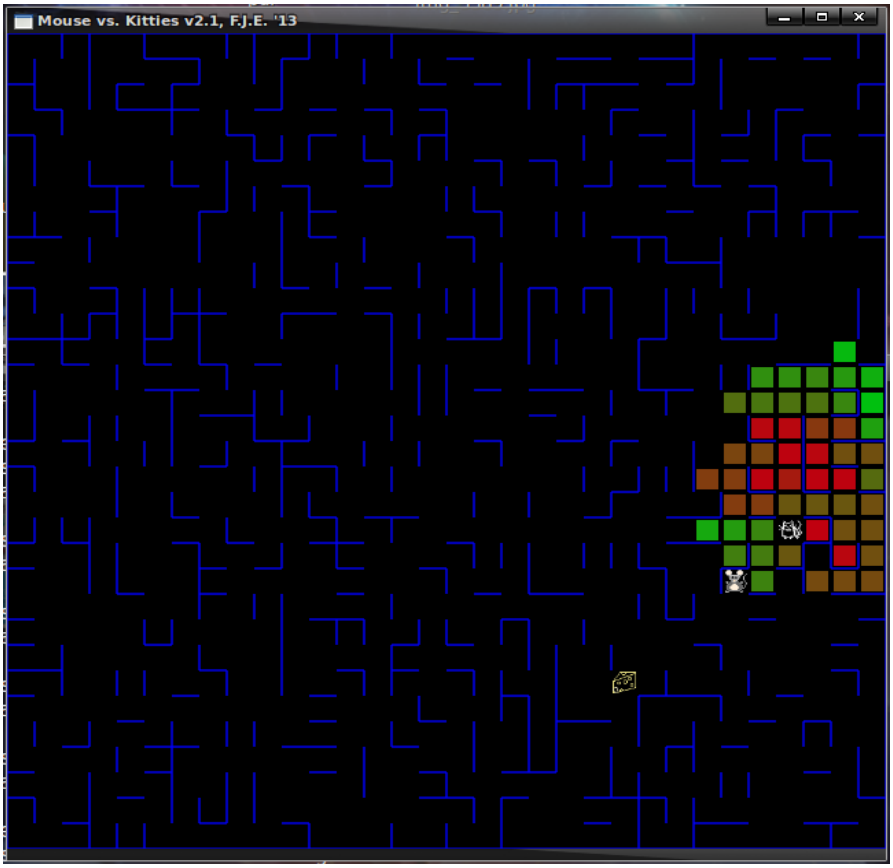
Once you have determined the utilities, indicate which path is chosen.

Building Smarter Mice

Step 6 (cont.)

Once you have understood the mini-max algorithm, implement it and test it. You will probably need to print debugging information along the way, however ***make sure to turn off any print statements in MiniMax_search.c before submitting.***

If all goes well, once you have implemented mini-max and a utility function, you will be able to run a full game and see the mouse and cat chase each other around the maze. You should see an image similar to the one below***.***



There will be coloured squares around the mouse and cats. Colours correspond to utilities, you should see shades of green and red corresponding to positive and negative utility values. The brighter the colour, the larger the magnitude of the utility for that square.

Your mouse's actions are determined by the result of the call to your mini-max code. The cats... Well, they pretty much do what they want, as everyone knows.

Building Smarter Mice

Step 6 (cont.)

If you are doing the right thing, the mouse should be pretty smart and should win reasonably often against one cat (even with cat smartness set to 1).

Note that the ***max search depth*** value has a strong influence in the result. You have to experiment a bit to understand exactly how it affects the evolution of the game.

At this time:

Test! Test! Test!

- Make sure your MiniMax function and your utility function are doing the right thing.

- If you started with a simple utility, now is the time to go back and make it very ***crunchy***.

- Try and see what happens with different seeds, varying amounts of cheese, changing the cats smartness.

Answer the relevant questions in REPORT.TXT

Step 7

Implement Alpha-Beta Pruning – once more, you can use A.I. tools to help you

Alpha-beta pruning is a technique for quickly eliminating from the search tree branches that ***can not possibly be chosen as part of the optimal path***. Please review your notes on alpha-beta pruning and make sure you understand how it works.

The actual amount of code required to implement alpha-beta pruning is very small, so it is short, but it is also easy to get it wrong if you don't know what you are doing. Once again, feel free to add debugging printouts and test with a single cat, a single mouse, and a single cheese.

Debugging suggestion: Choose a random seed. Using a single cat/mouse/cheese, and a low value for ***max search depth*** (e.g. 4) print out the utilities for each node visited for ***a single round*** of the game using ***standard minimax***. Draw this in the form of a search tree.

Then, turn on your alpha-beta pruning, print out the utilities for nodes being expanded (evidently you won't see all the nodes now that some are being pruned). Draw these nodes as well in tree form. ***Now compare the two trees and make sure alpha-beta pruning is cutting the correct branches off the search tree.***

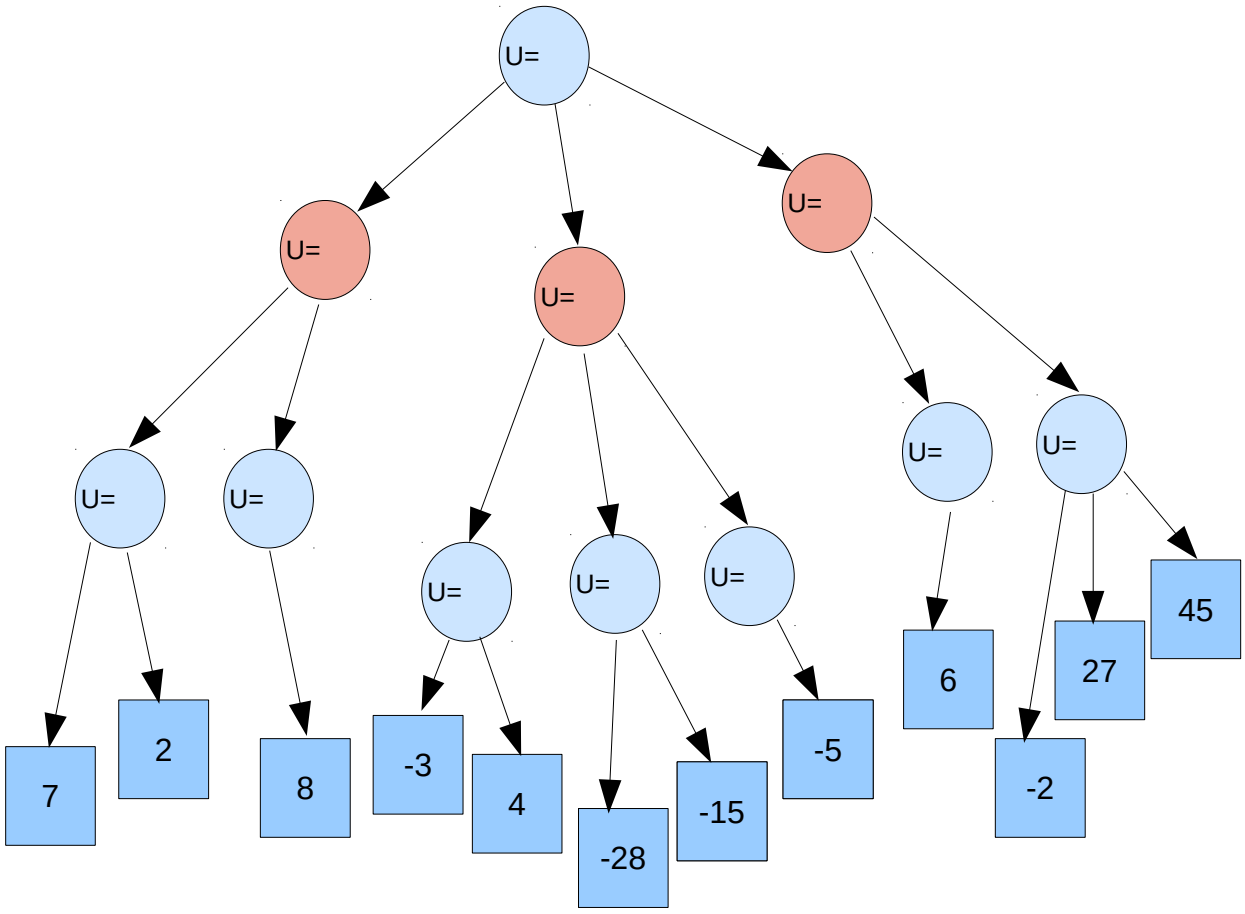
Another way to verify your alpha-beta pruning is working: The mouse makes exactly the same moves with and without alpha beta pruning, but the time take for each move is much smaller with alpha-beta pruning on.

Building Smarter Mice

Step 7 (cont.)

To make sure you fully understand alpha-beta pruning, complete the exercise below by writing The ***utilities*** at each node, and indicating ***which branch(es) are pruned*** by the algorithm (and why – you should note the alpha or beta value that causes a branch to be cut).

Once again, you do not have to hand this in, and it will not be graded. But if you find this difficult to do, you may want to study the notes and then come and ask about anything that seems confusing.



Once your alpha-beta pruning code is working, it's time to ***Test! Test! Test!***

- Choose a random seed, one cat, one cheese. Run the ***standard minimax and note the sequence of moves played by the cat and mouse***. Using the same random seed, run the game using ***alpha-beta pruning***. You should obtain the ***same sequence of moves***.

Building Smarter Mice

Step 7 (cont.)

- You should now be able to comfortably run games with a ***max search depth*** of 14 or 15. The mouse and cats should be pretty interesting to watch if you implemented everything correctly.

Answer the relevant questions in REPORT.TXT

Step 8

Submit your work

Create a single compressed ***.zip*** file – and that means it should actually be ***in .zip format***. If you submit a compressed tar file, a .ar, .arj, .arc, .7z, or anything else renamed to have the .zip extension you will incur the wrath of your TA and lose marks.

The .zip file should contain your entire code directory (that means, your code, the .ppm images, compile script, and object files):

Your .zip file should be named:

`MiniMaxSolution_studentNo1_studentNo2.zip`

(e.g. MiniMaxSolution_012345678_9876543210.zip)

Submit your file on Quercus as usual

Before you submit, make sure your compressed file in fact contains all your files and code, and that it decompresses properly on mathlab. Non-working .zip files will get zero marks.

If you have an emergency and can't submit on Quercus, email me your code, but note I will impose a 5 to 10 marks penalty for not leaving yourself enough time to account for possible issues while submitting.