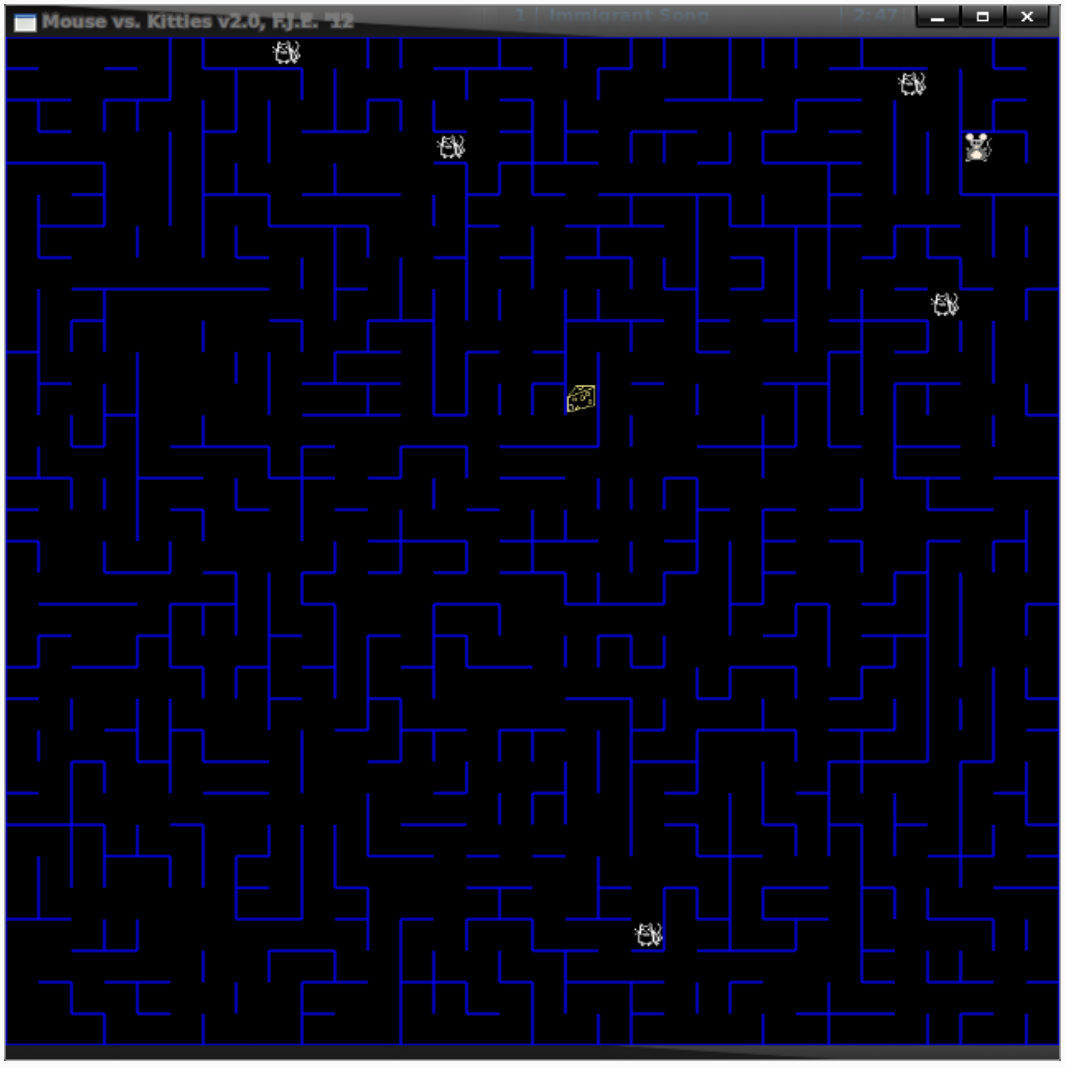
Assignment 1
A Mouse's New Hope

Due date: Monday, Jan 21st at 9am
(electronic submission on Mathlab)

This assignment is worth 10 units toward the total assigned work you will do during the term



CSC D84 – Winter 2013
Search problems and search techniques

The goal of this assignment is to give you practical experience in implementing ***search algorithms.*** As discussed in class, ***search problems*** are very common in A.I., and have direct applications in robotics, game playing A.I., and pattern recognition. For this assignment you will be working with a single agent that must use search to find its path to a goal. You will try different search techniques, and observe the search patterns that arise and their effect on the agents eventual success in finding the goal efficiently.

Learning Objectives:

You will understand the problem of path planning defined over a graph of locations.

You will learn the differences between common search methods: ***DFS, BFS, and A**** search.

You will experiment with different search algorithm and observe the search patterns generated by each. This will provide you with experience regarding what search methods may be more appropriate for path planning applications.

You will explore the concept of ***admissible heuristic*** and develop a valid ***A**** search Procedure based on proximity from the goal.

You will experiment with different heuristics and attempt to develop a cost function that prevents the agent from being caught before it achieves its goal.

You will gain practical experience in implementing the different search methods discussed In lecture.

Skills Developed:

Working with graph-based map descriptions. Performing graph traversals and checking for obstacles.

Implementing search algorithms. In particular, you will develop the ability to code both recursive and iterative search methods, and understand which implementation is better suited to each search method.

Developing good heuristic search functions for a given problem.

Reference material:

Your in-class lecture notes on search.

Sebastian Thrun's on-line short lectures on search (thanks Sebastian!). Start here:

http://www.youtube.com/watch?v=TPIFP4E7DVo&list=PLAADAB4F235FE8D65&index=14&feature=plpp_video

Then follow the trail to where it leads. Mind that Sebastian's setup and ours are quite different, so ***ignore the Python and programming details!*** Your solution ***will*** look different! However, the algorithm descriptions may be useful for you.

Inspired by UC Berkeley's PacMan Project, with thanks to John DeNero.

The Cat & Mouse Game v2.0

The assignment consists of implementing a set of search methods that will enable a poor hungry mouse to eat all the cheese hidden somewhere inside a cat-populated maze. For starters, the cats will be ***not too bright*** and they move randomly, with a bias toward moving in the general direction of the mouse.

The cats won't even react to 'seeing' the mouse. However, there is many of them and only one mouse so you don't want to make a mistake.

The general flow of the game is as follows:

Game is initialized – random maze, random locations for cats, mouse, cheese
Loop until cheese is all gone or mouse has been eaten:
    Update cat positions
    Update mouse position (this will call ***your*** search code)
    Check for mouse/cat collision, check for mouse/cheese collision

All the code to implement the initialization and updates is given to you, all you need to do is code the actual search methods.

Step 1

Download and uncompress the starter code into an empty directory.

This code is designed for Linux. I recommend you do all your work on the CS lab at IC 406. You can work remotely by logging into ***mathlab.utsc.utoronto.ca*** but note that the program uses OpenGL graphical display and will run very slowly over ssh.

We will not be able to support Windows, if you choose to develop on Windows you will have to set up your computer properly, and you ***must*** ensure your final submission ***works on mathlab***.

The starter code contains the following files:

AI_search_core_GL.pyc
AI_global_data.pyc
AI_search.py
REPORT.TXT

The only program file you need to look at is ***AI_search.py***. All your code will be written here. ***Do not add any extra Python program files.*** Also, you ***can not add any global data***.

Read all the comments in A1_search.py. The comments describe the global data that is Available to you (and which specifies the maze structure and locations of all the agents). 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***.

The Cat & Mouse Game v2.0

Step 2

Test-run the starter code:

1) Start an interactive Python shell ***(please make sure you're using Python 2.6 or 2.7!)***

2) On the shell prompt execute:

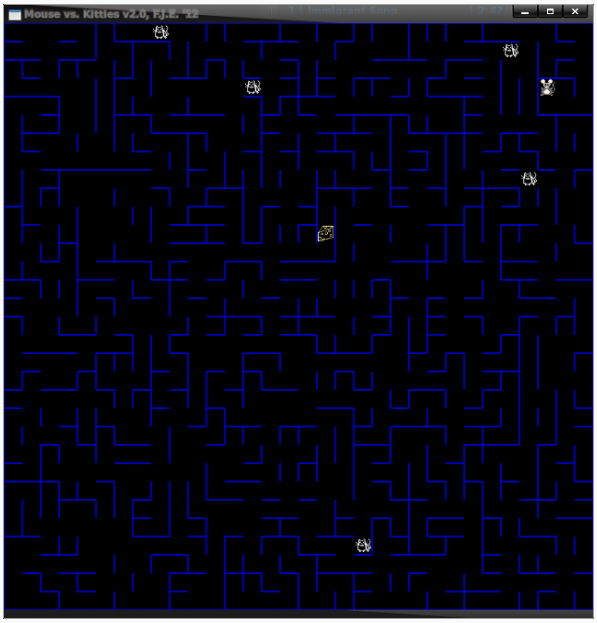
```
>>>from AI_search_core_GL import *
>>>initGame(1522,5,1)
>>>doSearch(0)
```

The first line imports all the code needed to run the Cat and Mouse game. Note that ***AI_search_core_GL*** will import your search code, so you don't need to import anything else to run the game.

The second line initializes the maze and agents. The first parameter is the ***random seed*** that will be used to generate the game. Choose any integer number you like, but use always the same number for testing and debugging. The second parameter is the number of cats, and the last parameter is the number of cheese pieces. In this case, 5 cats and 1 cheese.

The last line launches the search loop, the number 0 specifies that search will use the (included) dummy search algorithm that just moves the mouse randomly.

If all works well, you will see the following screen display:



Both the cats and mouse should move around. Press 'Q' to quit while on the graphical window, or wait for the cheese or mouse to get eaten.

The Cat & Mouse Game v2.0

Step 3

Understand the global data – understand how the maze is stored and where things are.

- The maze has a dimension of (32 x 32) locations. These are numbered 0 to 31 along each direction, with (0,0) at the top-left, and (31,31) at the bottom right. Ordering is ***row-major***, so (1,0) is row 0, column 1. An easier way to think about this is that coordinates are given as (x,y).

- The Maze is represented by a ***graph with one node per maze location***. Therefore, there are 32X32=1024 nodes in the graph.

- The graph is stored in an ***adjacency list*** called ***A[][]***. This list contains ***one row per maze location*** and ***4 columns***. The columns store the connectivity of each maze location with the ***top, right, bottom, and left*** neighbours respectively.

  Example: Suppose you want to find out to which neighbours maze location (5,3) is connected.

  1) Determine the ***index*** of data for maze location (5,3) in the A[][].

  *Index = (x + (y * 32))*
  *= (5 + (3*32))*
  = 101

  2) Check which neighbours are connected to (5,3). This is given by A[101][:] so:

  A[101][0] → if this is '1', (5,3) is connected to top neighbour (5,2)
  A[101][1] → if this is '1', (5,3) is connected to right neighbour (6,3)
  A[101][2] → if this is '1', (5,3) is connected to bottom neighbour (5,4)
  A[101][3] → if this is '1', (5,3) is connected to left neighbour (4,3)

  If any of the entries in A[101][:] is '0', that means there is a wall between (5,3) and the corresponding neighbour.

- The agent locations are stored in

  Mouse[][] → a single entry list, the mouse location is (Mouse[0][0], Mouse[0][1])
  Cats[][] → a list of pairs of coordinates, one per cat. The location of the *ith* cat (Cats[i][0], Cats[i][1])
  Cheese[][] → A list of pairs of coordinates, one per cheese. Analogous to Cats[][]
  Ncats → Number of cats in the game
  Ncheese → Number of pieces of cheese left at a given time

- All your search functions will update MousePath[][], this is a list of coordinate pairs describing a path from the mouse to a/the cheese chunk. Please make sure to read the comments in the code that explain how to store the path.

- Maze[][] array, of size (32 x 32), used to keep track of the order in which maze locations were explored by the different search algorithms.

The Cat & Mouse Game v2.0

Step 4

Implement the search algorithms:

1 – Breadth-First Search (BFS)

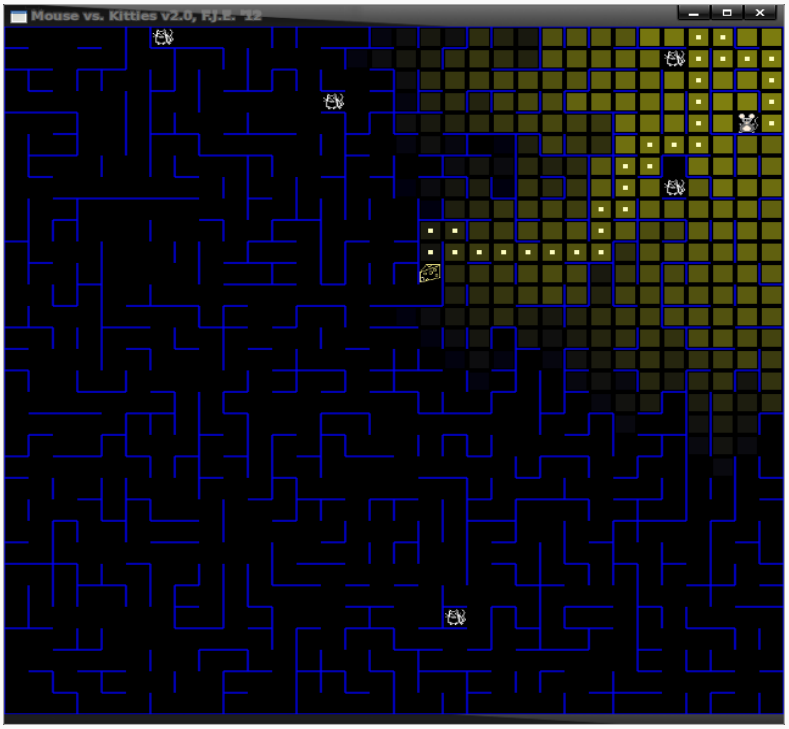
Please read the comments in the relevant section of ***AI_searc.py***. Your code will be called on each turn to compute a path to the cheese. The path may change due to the cats blocking parts of the maze.

As a reminder, BFS expands nodes in order of distance from the starting point. Distance here is measured as ***Manhattan*** distance. The path returned will be the shortest path that leads from the mouse to the cheese and does not attempt to go over any cats.

Test your implementation by calling the search function with argument 1:

```
>>>doSearch(1)
```

Which will display an image similar to the one below



Note:

- The path found by BFS is marked by yellow dots.

- Maze locations are coloured, brightness indicates the order in which different locations Where expanded by BFS. Brighter squares were expanded earlier.

- Notice the typical pattern of expansion with locations closer to the mouse being explored before those farther away.

- For this to work, you must ensure your search function updates the global Maze[][] array.

The Cat & Mouse Game v2.0

Step 4

Implement the search algorithms:

2 – Depth-First Search (DFS)

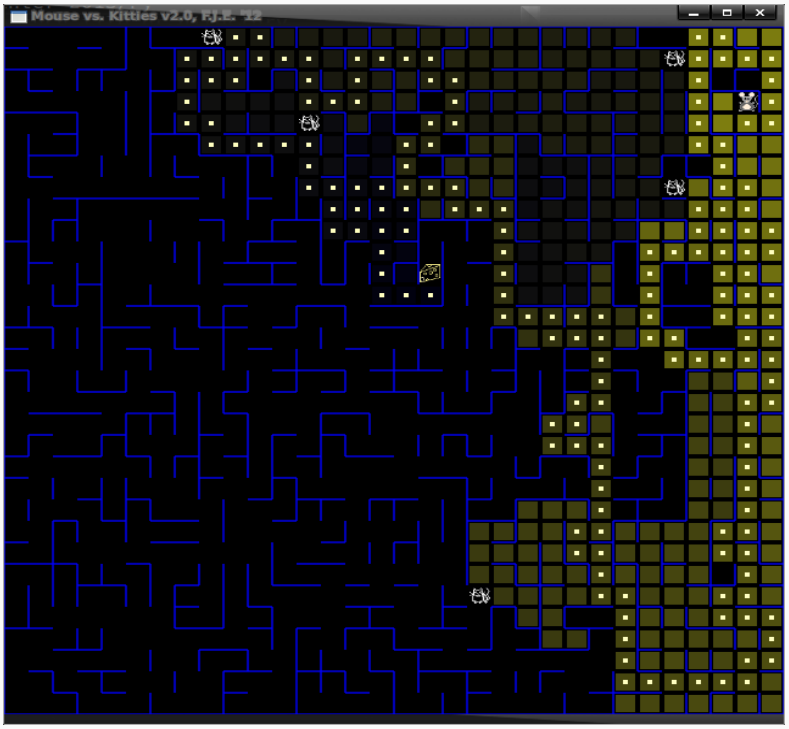
Please read the comments in the relevant section of ***AI_searc.py***. Your code will be called only when the current path is blocked by a cat at less than 10 moves away from the mouse (unlike BFS, which is called for each move).

DFS expands nodes moving away from the initial location and traveling as far as possible until no further expansion is possible or the goal has been found.

Test your implementation by calling the search function with argument 2:

```
>>>doSearch(2)
```

Which will display an image similar to the one below



Note:

- The typical pattern of exploration now moves far away from the mouse.

- The shape and length of the Path created by DFS. Compare with BFS.

- Consider the behaviour and stability of the path as the game progresses, compare with BFS.

- Compare the number of nodes expanded (search effort) against that of BFS.

- Which of BFS/DFS works best for this particular game?

The Cat & Mouse Game v2.0

Step 4

Implement the search algorithms:

3 – A* search

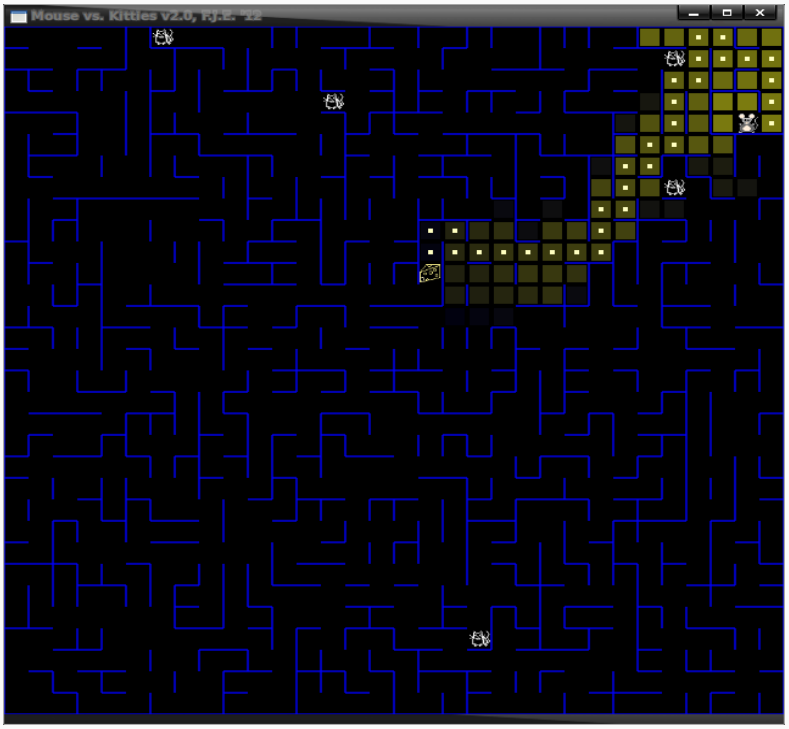
Please read the comments in the relevant section of ***AI_searc.py***. Your code will be called for every turn to re-compute the path to the cheese so as to account for motions by the cats.

The order of expansion depends on distance from the initial location, and on the heuristic cost function. You must define and implement your heuristic cost function, and ensure it is ***admissible***.

Test your implementation by calling the search function with argument 3:

```
>>>doSearch(3)
```

Which will display an image similar to the one below



Note:

- The typical pattern of exploration now moves toward the cheese.
- The length of the path created. Compare with BFS.
- Consider the behaviour and stability of the path as the game progresses, compare with BFS and DFS.
- Compare the number of nodes Expanded (search effort) against that of BFS/DFS.
- Which of BFS/DFS/A* works best for this particular game?

The Cat & Mouse Game v2.0

Step 4

Implement the search algorithms:

4 – A* search – no kitties!

Please read the comments in the relevant section of ***AI_searc.py***. Your code will be called for every turn to re-compute the path to the cheese so as to account for motions by the cats.

Your task here is to make the mouse as hard to beat as possible! You must design a heuristic function that helps the mouse escape being eaten, and consume all the cheese in the maze. Think carefully about your heuristic function and what it should include, and also reflect on whether it is admissible or not (and what that tells you about heuristic functions).

Test your implementation by calling the search function with argument 4:

```
>>>doSearch(4)
```

I won't show you my solution here. I want you to think about the problem and come up with an unbeatable mouse!

Feeling good about your heuristic? Try it against:

Smart Kitties → doSearch(5)
Evil Kitties of Doom → doSearch(6)

Step 5

Test! Test! Test!

Be sure to thoroughly test your implementation of each search method and make sure it does what is expected.

We will test your code with:

- Different random seeds
- Different numbers of cats (5 to 10)
- Different numbers of cheese (1 to 10)
- All search algorithms

Observe the behaviour of your algorithm and think about the following questions:

- Which search algorithm is best/worst for this game in terms of:

* Overall success: Helping the mouse 'win' the game by eating all the cheese regardless of the number of cheese/cats in the game.
* Search effort: Finding paths to cheese with the least amount of nodes expanded

The Cat & Mouse Game v2.0

Step 9

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:

```
AI_search.py
REPORT.TXT
AI_search_core_GL.pyc
AI_global_data.pyc
```

Your .zip file should be named:

```
SearchSolutions_studentNo.zip
```

(e.g. SearchSolutions_012345678.zip)

Submit your file on mathlab with the command:

```
submit -c cscd84w13 -a A1 -f SearchSolutions_studentNo.zip
```

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 mathlab, email me your code, but note I will impose a 5 to 10 marks penalty for not testing well in advance that you have access to mathlab and can submit electronically as required.