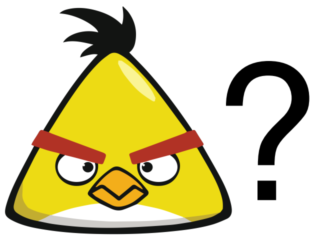
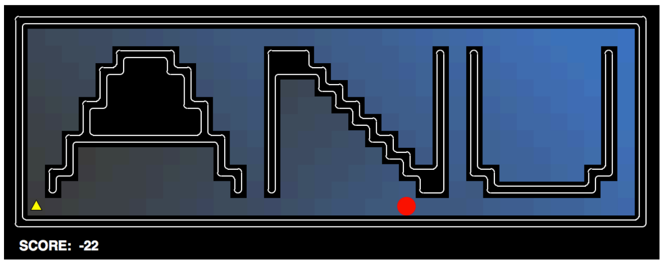
COMP3620/6320 Artificial Intelligence

Assignment 1: Search - Where is the Yellow Bird

The Australian National University
Semster 1, 2014





1 Background

Chuck, the Yellow Bird, is lost in a maze. Time for you to guide Red, the Red Bird, to look for him!

Throughout this assignment you will implement some search strategies to help Red find a path to Chuck. You will write an admissible heuristic to help Red find multiple clones of Chuck in a maze. You will also write parts of a minimax algorithm to help Red find more Chuck clones than Black, the black bird.

To do this assignment, you mainly need to look at the files `search_strategies.py`, `heuristics.py`, and `minimax_agent.py`. Along with this handout, the comments in these files should tell you everything that you need to know to get started.

2 Preliminary: Human Intelligence

Before you create some AI algorithms to solve the search problems in this assignment, you can try solving them yourself by using either the arrow keys or `w`, `a`, `s`, and `d` to move Red around the maze With the following command:

```
./testKeyboard.sh layout
```

where `layout` can be any one of:

- aiSearch, anuSearch, or mazeSearch to help Red find Chuck;
- aiMultiSearch, anuMultiSearch, or mazeMultiSearch to help Red find all of the Chuck clones; or
- smallAdversarial, testAdversarial, aiAdversarial, or anuAdversarial to help Red find more Chuck clones than Black.

3 Q1: Depth-First Search (2 marks)

In the depth_first_search function in search_strategies.py, implement the depth-first search algorithm to find Chuck. You can test your solution with the following commands:

```
./testDFS.sh aiSearch
./testDFS.sh anuSearch
./testDFS.sh mazeSearch
```

You should see the states expanded in your search in a blue hue, where the brighter the hue indicates the earlier the expansion.

At the bottom of the UI you will see a score. For Q1-Q4, the Red loses one point every step and then gets 100 points for each yellow bird he saves. Q5 has a different scoring function, which is explained there.

You can implement a generic search function for handling Q1-Q3. However, this is not necessary for you to receive full marks for the questions.

Look in frontiers.py for a Stack (LIFO) data-type that you may find useful for this exercise.

4 Q2: Breadth-First Search (2 marks)

The depth-first search algorithm may not find the shortest path to Chuck, but as you should have seen from the expanded nodes, it may not have to explore much of the search space to find a solution.

To get the shortest solution, implement the breadth-first search algorithm in the breadth_first_search function in search_strategies.py. To test your implementation, use the following commands:

```
./testBFS.sh aiSearch
./testBFS.sh anuSearch
```

```
./testBFS.sh mazeSearch
```

If you have implemented BFS correctly, Red should find Chuck in 26 steps (with a score of 74) in aiSearch, 45 steps (with a score of 55) in anuSearch, and 68 steps (with a score of 32) in mazeSearch.

Look in `frontiers.py` for a `Queue` (FIFO) data-type that you may find useful for this exercise.

5 Q3: A-Star Search (3 marks)

Breadth-first search always find the shortest solution, but it can be very costly in expanding a lot of search nodes. To minimise search cost, a smarter search is needed.

A-star search is a smart search if you use the right heuristic. Implement A-star search in the function `a_star_search` in `search_strategies.py`. You can then test your solution with the Manhattan Distance heuristic (already implemented) with the following commands:

```
./testAStar.sh aiSearch manhattan
./testAStar.sh anuSearch manhattan
./testAStar.sh mazeSearch manhattan
```

If you have implemented A-star search correctly, you'll find that Red finds Chuck with the same path length, but with fewer search nodes expanded.

You can also try this using Euclidean distance as a heuristic by substituting `manhattan` for `euclidean` in the above examples.

Look in `frontiers.py` for a `PriorityQueue` data-type that you may find useful for this exercise. Additionally, there is an optional `PriorityQueueWithFunction` class which allows you to pass an evaluation function to make it easier to write a generic search function for Q1-Q3 if you want to.

Once you are finished, you may wish to think of other, stronger heuristics here. You may notice that it can be quite difficult to think of admissible heuristics which are not just as expensive to compute as the original problem is to solve!

6 Q4: Find Every Yellow Bird (4 marks)

Now there are many Yellow Birds lost in the maze. You have to guide Red to find them all! This is a much more difficult problem than finding a single bird. A solution consists of a path that find every yellow bird.

If you have implemented A-star search in the previous question correctly, you can already solve small problems such as `testSearch` by typing:

```
./testEveryBirdNull.sh aiMultiSearch
./testEveryBirdNull.sh anuMultiSearch
./testEveryBirdNull.sh smallMultiSearch
./testEveryBirdNull.sh mazeMultiSearch
```

However, you'll find that it can be very slow for some problems, such as `mazeMultiSearch`. To solve this last problem with the null heuristic, A-star has to expand over 500000 nodes! To optimise your search, you need to build your own heuristic to find every yellow bird in the maze. Implement `every_bird_heuristic` with an **admissible heuristic** in `heuristics.py`. You can test your heuristic with the following commands:

```
./testEveryBird.sh aiMultiSearch
./testEveryBird.sh anuMultiSearch
./testEveryBird.sh smallMultiSearch
./testEveryBird.sh mazeMultiSearch
```

You will be graded on how many nodes your search expanded on the `smallMultiSearch` map. The fewer search nodes explored, the more points you'll get:

- $>$ 15000 nodes: 1/4
- $\leq$ 15000 nodes: 2/4
- $\leq$ 8000 nodes: 3/4
- $\leq$ 7000 nodes: 4/4

If you can solve `smallMultiSearch` and `mazeMultiSearch`, expanding less than 2000 nodes in less than 30 seconds (on our test machine) with an admissible heuristic then you will be awarded one bonus point to make up for marks if you lose them elsewhere in the assignment. We have a solution to this problem which expands 698 nodes and takes 0.8 seconds.

You need to explain in the comments of your heuristic why it is admissible. You will suffer a two point penalty to the mark awarded to your heuristic if it is not admissible, so be careful!

The comments in `every_bird_heuristic` detail the data structures you will need to use. In particular, we have gone to the trouble of pre-computing the shortest distance between every pair of positions on every map. As described in the comments this information is available to you using the `maze_distance` function.

7 Q5: Minimax (4 marks)

Now Bomb, the black bird and Red are having a race to see who can find the most lost yellow birds.

Bomb really wants to help as many yellow birds as possible and greedily runs towards the closest yellow bird to him at any time. He will only stand still if there is nowhere he can move. To see how he behaves, look at the class `GreedyBlackBirdAgent` at the bottom of `agents.py`. To help Red win the race we are going to use the *Minimax algorithm.*

In the score displayed in the UI for Q1-Q4 Red loses 1 point every step and scores 100 points for every yellow bird he saves. In this question, the scoring function, which is used for the utility of terminal states, is different. The score does not decrease every step. Instead, Red scores a number of points for every yellow bird he saves and loses points for every yellow bird Bomb saves. The yellow birds are initially worth 100 points, but the value of each unsaved yellow bird decreases by 1 at every move (this helps giving Red and Bomb a sense of urgency!). The yellow birds will never be worth less than 1 point. Red has to get a positive score to win. Red wants to have the highest score overall, even if he ends up losing.

Implement the required methods in `MinimaxAgent` in `minimax_agent.py`. First, you will need to implement the methods `maximize` and `minimize`.

You can then test your implementation with:

```
./testMinimax.sh testAdversarial depth
./testMinimax.sh smallAdversarial depth
./testMinimax.sh aiAdversarial depth
./testMinimax.sh anuAdversarial depth
```

Here, `depth` is the maximal depth of the game tree explored. In your implementation each maximization or minimization should count as one depth step.

You should notice that it is too computationally expensive to set the depth high enough to explore the whole search space. When the depth limit is reached the algorithm calls `evaluation` instead of `utility`. This evaluation represents an estimate of the utility.

The implementation of `evaluation` we have provided the search just returns the utility of the reached state, rather than the expected utility at the end of the game. You need to implement your own, more powerful evaluation in this method.

You should hopefully be able to perform competitively with the Black Bird on these maps with a depth of $10 - 12$.

You will be awarded 2 of the 4 points for this exercise for the correctness of your `minimize` and `maximize` methods and a further 2 points for the quality of your `evaluation` method.

8 Submission

When you have finished the assignment, make and submit (on Wattle) a `uxxxxxx.tgz` file containing the following files:

```
search_strategies.py
heuristics.py
minimax_agent.py
```

Auto-grading will be used to assess the correctness of your code, but all submissions will be looked at manually as well. Please comment your code as it helps us to determine your intent and give part marks when things don't work as intended.