Using Search to Make a Chase Game AI.

CSC384 – Fall 2015
Due: Electronic Submission Tuesday Dec 8th, 7:00pm
Project Type: Search

Robert Staskiewicz (g4staski)
Jaisie Sin (g2crypt)

I. EFFICIENCY OF A* SEARCH FOR GAME AI

The thrill of a chase is fun and exciting. For our project, we created a classic chase game where the goal is for the player to avoid being caught by a computer-controlled enemy. The game begins with both the player and the computer being placed in a space in an n-by-n grid at random, non-identical starting locations, with each at least the floor of n/4 squares vertically and horizontally apart from each other.

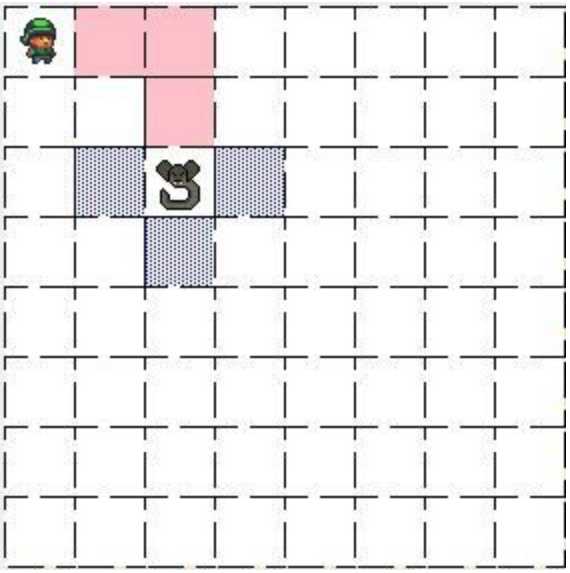

Figure 1. The initial state of our game. Blue tiles are on open list, while red tiles is the solution path as determined by A* search.

From here, the player's goal is to avoid being caught by the computer. "Being caught" is defined by the computer's position being on the same square as the player. The user can use the up, down, left, and right arrow keys on the keyboard to change their position in the grid at a rate of one grid space per press. Based on the player's position at any given time, the computer makes a decision about where it should move in order to catch the player. Like the player, the computer can only move up, down, left or right. The player and computer may move asynchronously of each other, with the player being moved at the user's will, and the computer moving at a particular rate per second.

As well, at the start of the game, the player chooses the difficulty level they would like to play at:

- *Easy has the opponent moving at a speed of 1 block every 1 seconds.*
- *Medium has the opponent moving at a speed 1 block every 0.5 seconds*
- *Hard has the opponent moving at a speed 1 block every 0.25 seconds*

II. APPROACH DESCRIPTION

A. OVERVIEW

Search is a basic technique in the field of artificial intelligence. Basic and practical, it serves as a successful model for several aspects of intelligent behaviour -- and for games such as ours. Search has proved to be successful in solving for the optimal path the computer has to traverse in order to catch the player. Search enables a system to determine a particular sequence of actions to take to reach a goal state, which perfectly models our game's requirement to find a sequence of steps to take to form a path to the player's location.

Our approach was implemented with Python 3 and relies on three interfaces: `grid.py`, `Search.py`, and `game.py`.

`grid.py` uses the Python package `tkinter` to provide a graphical user interface for a user to play the game. This interface allows for the user to move their player icon around the grid in order to keep away from the computer enemy that is chasing them. The GUI also visualizes the path the computer has determined to follow in order to catch the player and its unexpanded but open nodes on the successor queue. This interface also handles the event loop that moves the enemy along a path, as decided by search procedures, to catch the player. Additionally, every time that the player moves, this interface calls search procedures with the updated player coordinates to determine a new path for the enemy to take to try to catch the player, the implication being a computer opponent computes an optimal path via search every time a human player moves – quite reasonable with modern a CPU.

`Search.py` was supplied to us in Assignment 1 of the course, and it is this file that forms the basis of our search routines. It contains the `StateSpace` and `SearchEngine` classes for performing search, and their details can be found in the assignment handout.

`game.py` inherits from `StateSpace` and contains our implementation of the game's search problem. The game has a player and an enemy. The player and enemy are initialized in separate locations that are at least the floor of n/4 squares vertically and horizontally apart from each other. The player and enemy can move around the game board.

In this domain, the enemy face the problem of finding a path to traverse in order to catch the player. The problem is solved when a path to the player is found.

B. SEARCH FORMALISM

Our path finding problem can be easily formulated as a search problem, as it has the four required components of a search problem:

1) STATE SPACE

As explained above, `game.py` inherits from the abstract class `StateSpace` and contains

a) The class initializer method "`__init__`". This method initializes the specific state data structures for a `game` object, and calls the base class init method to initialize the general data shared by all state space objects.

b) The `successors` method, where each game object represents a state in the game state space. `successors` is implemented as a "self" function and is used to generate the state's successors. It returns a list of `game` objects, each reachable from "self" by a single action. Each `game` object contains a reference to its parent and the action used to obtain it (represented as a string), the cost of getting to this state (the g-value), and the game specific data structures.

c) The `hashable_state` method, which overrides the function by an identical name within `StateSpace` in order to employ its use of a python dictionary to implement cycle checking.This function returns a data item that can be indexed into the dictionary. Two states generate the same state if they have the same x and y coordinates as their respective `enemy` coordinates.

2) ACTIONS or STATE SPACE Transitions

Our state space `game` employs at most four actions at a time, one for each possible direction the enemy player can move: left, right, up, and down. These actions cost an amount equal to the move time forward (i.e. one time step each).

At times, a state will have fewer than four actions. For example, when the enemy has arrived at the top-left corner of the map, with its previous action having been to move up, its only possible action to take is to move right, since there is no where on the board to move if the enemy were to move up or left, and moving down would return it to its previous state.

3) INITIAL or START STATE and GOAL

In our start state, time is 0 to represent that 0 steps are taken by the enemy towards the player. Additionally, the start state has a particular (x, y) coordinate for the enemy and a non-identical (x, y) coordinate for the player.

In our goal state, the (x, y) coordinate of the enemy is the same as the (x, y) coordinate of the player. This represents that the enemy has found a path to the player.

4) Heuristics

Our heuristic, that is, the estimate of the cost of achieving the goal from any particular set of (x,y) coordinates, is $|x_1 - x_2| + |y_1 - y_2|$, where (x_1, y_1) are the coordinates of the player and (x_2, y_2) are the coordinates of the computer enemy.

With these components in mind, we were able to formulate our chase game path finding problem as a search problem, where the actions defined in II.(2) of the solution path represent the actual path that the the enemy could take to find the player.

III. APPROACH EVALUATION

In this section, we explore the various methods to order the paths on our Frontier or OPEN set, depth-first, breadth-first, and A*, in order to evaluate the overall value of search algorithms in order to solve our chase path-finding problem. We consider both (a) space complexity and (b) time complexity.

Note that we had considered best-first search, but decided against using it since our problem model does not involve the use of ranked nodes.

A. TIME COMPLEXITY

Time complexity was measured by running the search algorithm given varying starting coordinates for the player and the enemy. The coordinates categorized by the resulting distance between the player and the enemy, as given by $|x_1 - x_2| + |y_1 - y_2|$, where (x_1, y_1) are the coordinates of the player and (x_2, y_2) are the coordinates of the computer enemy.

i. Depth-First Search (DFS)

Case	n	Player (x,y)	Enemy (x,y)	Minimum Steps to Completion	Time to find solution (s)		
					Cycle Checking		
					None	Path	Full
a)	10	(0,0)	(1,0)	1	> 15 mins	> 15 mins	> 15 mins
b)		(0,1)	(0,0)	1	0	0	0
c)		(0,0)	(0,0)	2	> 15 mins	> 15 mins	> 15 mins
d)		(0,2)	(0,0)	2	0	0	0
e)		(1,1)	(0,0)	2	> 15 mins	> 15 mins	> 15 mins

Figure 2. DFS Run Time Evaluation. Sampling of time required for DFS to run given various starting values for board size n, player (x,y), and enemy (x,y). Minimum Steps to Completion is the heuristic measure of the number of steps needed to reach the player from the enemy's initial position.

Depth-first search is fast – but only when the player is not in the last successor to be opened. In our algorithm, our successors are added to OPEN in the order of left, right, up, and then down. As a result, the 'down' successor is the last to be added to OPEN and so this successor becomes the first to be explored next. This means that if our player is in that direction, then it can be explored very soon, as in the case b) and d) in Figure 2, which takes nearly no time at all to find a solution, as opposed to the other cases that take over 15 minutes and still do not find a solution.

We further explored the implications of the degree of depth by testing the case where the player starts at (1,1) and the enemy starts at (0,0), that is, the enemy needs to move one step down and one step right in order to find the player. The only parameter to vary is n, thus leading to a larger depth of exploration. The two directions down and right correspond to the latest vertical and horizontally moving successor types to be added to OPEN, and by that virtue, they will be explored the soonest. This allowed us to test the consequences of the size of the board, which in turn impacts the depth of the search.

Figure 3 (below) demonstrates the consequence of an increasing longest path in the state space. The longer the path, the more states must be checked for cycles, and in the case of full cycle checking, the time required increases exponentially with increasing board size (path checking requires less time since fewer states need to be checked). The time complexity of DFS is $O(b^m)$, where m is the length of the longest path in the state space, and in cases like those in Figure 3, much time is wasted by searching down one path and not another, shorter route.

Case	n	Player (x,y)	Enemy (x,y)	Minimum Steps to Completion	Time to find solution (s)		
					Cycle Checking		
					None	Path	Full
a)	2	(1,1)	(0,0)	2	> 5 mins	0.01	0
b)	3	(1,1)	(0,0)	2	> 5 mins	0	0.02
c)	4	(1,1)	(0,0)	2	> 5 mins	0	0.13
d)	5	(1,1)	(0,0)	2	> 5 mins	0.01	0.57
e)	6	(1,1)	(0,0)	2	> 5 mins	0	1.96
f)	7	(1,1)	(0,0)	2	> 5 mins	0.01	8.47
g)	8	(1,1)	(0,0)	2	> 5 mins	0.01	35.83
h)	9	(1,1)	(0,0)	2	> 5 mins	0.01	147.67
i)	10	(1,1)	(0,0)	2	> 5 mins	0	612.63

Figure 3. Testing implications of DFS depth. This chart visualizes the time required to perform DFS given a varying size of the game board and the state of cycle checking.

The results of Figure 3 also demonstrates the futility of DFS without cycle checking. Even in a board of size n=2, DFS can continue to go indefinitely with the states perpetually bouncing back between the parent state and a child state.

ii. Breadth-First Search (BFS)

Breadth-first search offered overall the second best results by time. We tested the effect of distance on the runtime of BFS. Figure 4 (next page) illustrates our results.

Case	n	Player (x,y)	Enemy (x,y)	Minimum Steps to Completion	Time to find solution (s)		
					Cycle Checking		
					None	Path	Full
a)		(0,0)	(1,0)	1	0	0	0
b)		(0,1)	(0,0)	1	0	0	0
c)		(0,0)	(2,0)	2	0	0	0.01
d)		(0,0)	(1,1)	2	0.01	0	0
e)		(0,0)	(3,0)	3	0	0	0.01
f)		(0,0)	(1,2)	3	0	0.01	0
g)		(0,0)	(4,0)	4	0.01	0.01	0.01
h)	10	(0,0)	(2,2)	4	0.01	0	0.01
i)		(0,0)	(3,3)	6	0.18	0.08	0.03
j)		(0,0)	(4,4)	8	3.33	0.94	0.16
k)		(0,0)	(5,5)	10	58.09	4.93	0.29
l)		(0,0)	(6,6)	12	> 10 mins	30.99	0.55
m)		(0,0)	(7,7)	14	> 10 mins	169.48	1.51
n)		(0,0)	(8,8)	16	> 10 mins	> 10 mins	4.98
o)		(0,0)	(9,9)	28	> 10 mins	> 10 mins	25.55

Figure 4. Testing implications of BFS depth. This chart visualizes the time required to perform BFS given a varying size of the game board and the state of cycle checking.

BFS produced mediocre results. Its efficiency began deteriorating (go over 1 s) when the path required a distance over 8 when no cycle checking was involved, 10 with path checking, and at 12 with cycle checking. It would seem that BFS is effective only with small sized boards that did not require a long path, and thus many states to be verified and successors expanded.

This makes sense, since in BFS, new paths are added to the end of OPEN. In the cases of Figure 4, the distance continually increases, and so the actual solution to the path will not be reached until all of the other states has been expanded. Expanding the entirety of OPEN can take a lot of time, especially the larger the distance between the player and the enemy becomes. With BFS, there is no doubt that a solution to the search will be reached eventually, however it could take a very long time to do so.

iii. A* Search

Out of the three types of search tested, A* Search generated the best results for our purposes by far. Figure 5 illustrates the runtime using A* Search given various distances between the player and the enemy for the three types of path cycle checking. From the visualization, it is easy to see that A* can quickly generate a solution path even when the player-enemy distance is vast. In particular, A* with full cycle checking can generate solutions in under a second when working under a distance of about 800 steps between the player and the enemy.

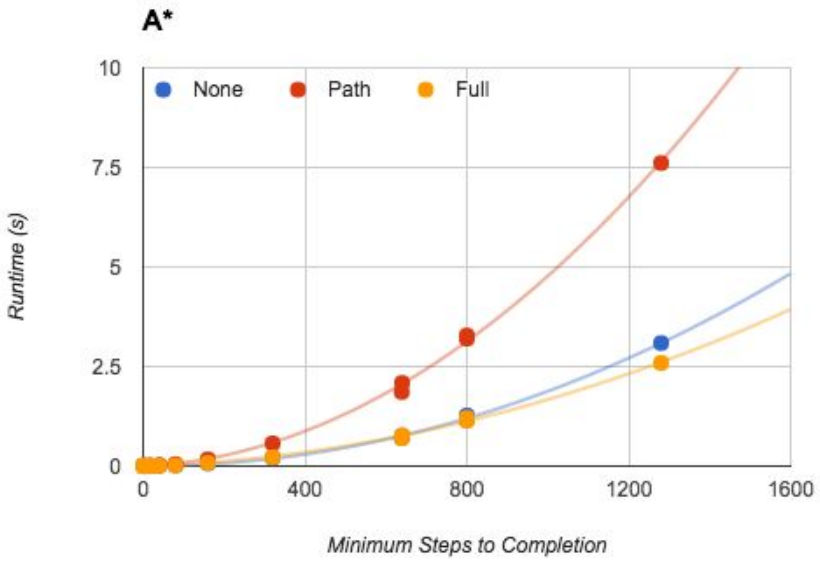

Figure 5. A* Search used to find solutions of various distances. This chart visualizes the time required to perform A* given a varying distance between the player and the enemy and the state of cycle checking.

Our A* search looked ahead to evaluate a guess on the number of steps it would need to take to reach the player. That is, it evaluated $|x_1 - x_2| + |y_1 - y_2|$, where (x_1, y_1) are the coordinates of the player and (x_2, y_2) are the coordinates of the computer enemy. Based on this, the node that is expanded will be the one that has taken the least number of steps already towards the player (i.e. the g(n) in the function for f-value) combined with the heuristic estimate of reaching the player (i.e. the h(n)).

Through the use of a heuristic, A* takes a more targeted approach when expanding nodes, and thus expectedly takes less time to find an answer than DFS or BFS.

B. SPACE COMPLEXITY

To evaluate space complexity, we modified `search.py` to keep track of the maximum number of nodes on open at a time in each of the three search algorithms. For each algorithm, we evaluate the effect that player-enemy distance and game board size, the two variables defining various versions of our path search problem, have on space complexity.

Player (x,y)	Enemy (x,y)	Min Steps to Completion	Maximum Number of Nodes on OPEN at a time								
			DFS			BFS			A*		
			N	P	F	N	P	F	N	P	F
(0,1)	(0,0)	1	2	2	2	4	3	3	2	2	2
(0,2)	(0,0)	2	4	3	3	16	9	7	4	3	3
(0,3)	(0,0)	3	6	4	4	58	23	15	6	4	4

Figure 6. Testing Space Complexity of Search. The maximum number of nodes stored in memory (i.e., the length of OPEN) is visualized in this chart to compare the space complexity of DFS, BFS, and A*. This figure looks at a set of cases that can be solved by even DFS with no cycle checking, where the board size was fixed at 10. Overall, BFS uses the most space, and DFS and A* were comparable. Note: N - No path cycle checking, P - path checking, C - cycle checking

n	Maximum Number of Nodes on OPEN at a time							
	DFS		BFS			A*		
	P	F	N	P	F	N	P	F
2	2	2	8	4	4	4	3	3
3	4	5	9	5	5	4	3	3
4	6	10	9	5	5	4	3	3
5	8	13	9	5	5	4	3	3
6	10	21	9	5	5	4	3	3
7	12	25	9	5	5	4	3	3
8	14	36	9	5	5	4	3	3
9	16	41	9	5	5	4	3	3
10	18	55	9	5	5	4	3	3

Figure 7. Testing Space Complexity of Search with varying board size. In each of these cases, player has coordinates (1,1) and the enemy has coordinates (0,0). DFS with no cycle checking as the solution could not be solved by this approach; the algorithm remained in an infinite loop.

i. Depth-First Search (DFS)

DFS is known to have a linear space complexity, O(bm), as only one path is searched at a time: OPEN only keeps track of the deepest node on the current path in addition to the backtrack points. Our results reflect this, as going down any of N, P or C under DFS in Figure 6 and Figure 7, the values are increasing linearly. These results emphasize the advantage DFS has in space.

That said, when DFS has a more difficult time searching for an ideal path, its space complexity could be worse than the other two search algorithms. In particular, in the case in Figure 7 where its first searched path to the deepest node, the number of nodes stored becomes quite large in comparison to those used by BFS or A*. DFS efficiency is dependent on the board size.

ii. Breadth-First Search (BFS)

BFS has a space complexity of $O(b^{d+1})$, where d is the depth of the goal node. As expected, BFS has the worst space complexity of the three algorithms with increasing distance between the player and the enemy, which is reflected in the larger values in the BFS section of Figure 6. In the worst case in our game, we have n^n nodes stored, where n is the size of the length of the board. This is far from ideal.

BFS is not dependent on the board size, as demonstrated by Figure 7. Going down any column for BFS, the maximum number of nodes stored does not change. This makes sense, as BFS searches nearer spaces first before searching further spaces on the game grid. The size of the game board has no effect.

iii. A* Search

Generally, the complexity of A* search varies with the heuristic used for any given problem, and in our case, A* search has a space complexity of twice the Euclidean distance between the player and the enemy, as the enemy is not allowed to traverse the grid diagonally and must advance in an L-SHAPE. For example, to go from (0,0) to (1,1) is 1 step diagonally, but is 2 steps with our given rules via a path right and then down, or down and then right. The OPEN would store the nodes consisting of the path [<0,1>,<1,0>].

Empirically, from Figure 6 we see it is the case that the maximum number of nodes in OPEN is directly related to the distance between the enemy and the player. As a result, A* has a space complexity that is comparable to DFS.

A* search is also not dependent on the board size, as can be seen by Figure 7. The maximum number of nodes stored does not change when going down any column for A*. This makes sense as, out of all of the options of a path that can be expanded, A* picks the one that is the best based on its estimated number of steps needed for the enemy to reach the player. Thus, like in the case of BFS, the size of the game board has no effect on the space complexity of A* Search.

IV. FINAL CONCLUSIONS

As in many search problems, efficiency is an important element. For us, efficiency and the minimizing of runtime was highly vital as the game needs to be playable in real time. The speed of search would determine the limits of our game: the slower the search, the smaller is too the maximum speed of the enemy, as time needs to be allotted for the enemy to evaluate an appropriate solution path to follow to reach the player. Unfortunately, a slow chase game would not make a very fun game for a human player!

In our case, the search needs to be completed within 0.25 seconds, which corresponds to the ‘hard’ difficulty of our game. Based on our results from this project, A* search with cycle checking does exactly this as long as the distance between the player and the enemy does not go much past 320 steps(Fig 5), which empirically has a runtime of 0.22 seconds. In fact, A* search’s runtime curve beats that of DFS and BFS, thus making it the most realistic search mechanism to use for our game of chase in terms of time. As it turns out, A* search also has the best space complexity regardless of the number of steps for the enemy to reach the player and the size of the board.

In all, our experiments have shown that our path finding problem in our chase game can be modelled as a search problem effectively and efficiently, in particular, with A* search with cycle checking.

V. REFERENCES

1. Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig. 3rd Edition.

2. AI demo by Alexander Biggs, located at: http://akbiggs.github.io/utgddc-ai/