Introduction to Artificial Intelligence
198:440
Project 1

Larry Scanniello

June 2024

1 Preliminaries/Bots 1, 2, and 3

I modeled the ship using NumPy arrays. I set open squares as 0's, blocked squares as 1's, fire squares as 2's, the button as 5, and bots $1, 2, 3, 4$ as $10, 100, 1000, 10000$ respectively, so that operations on the ship could be performed with modular arithmetic. I will give a quick summary of how I implemented the first three bots.

Bot 1, the simplest bot of the four, runs a breadth-first search at $t = 0$, then commits to the path returned by this BFS regardless of how the fire behaves.

Bot 2, rather than just running a BFS at $t = 0$, runs a BFS at every t in the range of the length of the simulation, reevaluating the shortest path to avoid the fire at every step of the simulation.

Bot 3 at every step first creates an altered grid, where every open cell adjacent to a fire cell is transformed to a 3. Bot 3 then runs a BFS on this altered grid, to find a shortest path to the button that does not travel through a 3-cell adjacently to a fire cell. If no such path is found, bot 3 reverts to doing what bot 2 would do, and finds the shortest path to the fire via a normal BFS that gives a route that travels through fire-adjacent cells.

As the data will show, bot 3 usually outperforms bot 2, and bot 2 pretty much always outperforms bot 1.

```
t= 9
[[   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    1.    0.    1.    0.    0.    1.    0.    0.    0.    0.    0.]
 [   1.    0.    0.    0.    0.    1.    0.    0.    0.    0.    0.    0.    0.    1.    1.]
 [   0.    0.    0.    0.    0.    0.    1.    0.    0.    1.    0.    0.    0.    0.    1.]
 [   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    1.    0.    0.    0.    0.]
 [   0.    0.    0.    1.    0.    1.    1.    0.    0.    1.    0.    0.    1.    0.    0.]
 [   1.    0.    0.    0.    0.    1.    0.    1.    0.    0.    0.    1.    1.    0.    0.]
 [   0.    0.    0.    1. 11110.    0.    0.    0.    0.    1.    0.    1.    1.    0.    0.]
 [   0.    1.    0.    0.    5.    0.    0.    1.    0.    0.    0.    0.    0.    0.    0.]
 [   0.    1.    0.    0.    0.    1.    0.    0.    1.    1.    1.    0.    0.    0.    0.]
 [   1.    0.    0.    0.    0.    0.    0.    1.    0.    0.    1.    0.    0.    0.    0.]
 [   0.    0.    1.    0.    1.    0.    1.    2.    0.    1.    2.    2.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    2.    2.    1.    2.    2.    1.    0.    1.    0.]
 [   0.    0.    0.    1.    0.    0.    2.    2.    2.    2.    1.    2.    2.    0.    0.]
 [   0.    0.    0.    0.    0.    1.    2.    2.    1.    2.    2.    2.    1.    0.    1.]]
```

Figure 1: A frame of a random simulation, D=15. The bots are almost at the button. The fire is the cluster of 2's.

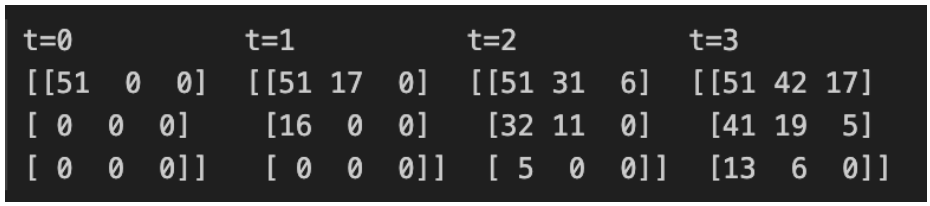

Figure 2: The first few frames, $t_k = 0, 1, 2, 3$ from a hypothetical *weightarrlist* made for a 3×3 grid instead of 25×25. The function **get_bot_4_weights** creates a list with 70 arrays like these total instead of four, made by adding one to a cell for every time that in one of the 50 hypothetical fire simulations there is a fire in that cell at that point in time.

2 Bot 4

The algorithm I created for bot 4 is much more involved than the algorithms for any of bots $1, 2$, or 3. From here, the "main simulation" refers to the actual simulation of the fire spreading and bots traversing the grid for real, as opposed to the "hypothetical simulations" which are the simulated fires by bot 4 that happen prior to the main simulation.

The first step of this algorithm is to, at $t = 0$ of the main simulation, pass the starting grid to a function called **get_bot_4_weights**. The main idea of the function **get_bot_4_weights** is to run 50 hypothetical simulations of how the fire might unfold, and then create a weighting system so that the path of bot 4 reflects the probability of there being fire at a particular space at a particular point in time. Since in all of my tests, I worked with $D = 25$, I chose to run each of these 50 simulations 70 time frames deep ($t_k = 0, 1, 2, ..., 69$, where t_k is the time at the kth hypothetical simulation, $k = 1, 2, ..., 50$) from the beginning grid. The assumption here is that the longest possible shortest path from one point to another on a 25×25 array has length 50, so we should never need more than 70 steps to run a full fire. The result of these simulations is a list of 70 NumPy arrays, each array of size $D \times D = 25 \times 25$. I called this list *weightarrlist*, which is essentially a $25 \times 25 \times 70$ array.

Here is how *weightarrlist* is created. Say the main grid has just been passed to **get_bot_4_weights**. The list *weightarrlist* is initialized as a list of seventy 25×25 arrays where every cell equals 1. We now must go through our first of fifty hypothetical fire simulations. For the first hypothetical simulation, at $t_1 = 0$, for any square there is a fire in the hypothetical grid, the array *weightarrlist*[0] gets 1 added to it in all corresponding squares. At $t_1 = 1$, again in this first hypothetical simulation, for any square in the main grid that there is a fire in, the array *weightarrlist*[1] gets one added to any corresponding square. We continue in this way until $t_1 = 69$, which marks the end of the first of 50 total hypothetical simulations. The result of this process after 50 iterations is the final version of *weightarrlist*, which is then returned. The list *weightarrlist* can be visualized in figure 2.

The next step is to pass *weightarrlist* to the function **bot_4_UFCS**. The idea of this function is to run a uniform cost search through *weightarrlist*. I wanted to make it so that the UFCS algorithm traversed *weightarrlist* in sequence. In other words, we are traversing *weightarrlist* by starting in the starting spot at $t = 0$ in *weightarrlist*[0], then moving to an adjacent tile at $t = 1$ but in *weightarrlist*[1], then again moving to an adjacent tile but in *weightarrlist*[2], and so on. In this way, bot 4 accounts for the likelihood of fire at the time bot 4 will be at the square.

The first thing **bot_4_UFCS** does is create an adjacency list to make the search feasible. I think of this as converting a 3D array, *weightarrlist*, into a 2D directed graph, where if a node on the graph corresponds to a point at $t = k$, then it points to an adjacent node at $t = k + 1$. For instance, given *weightarrlist* with dimensions $D \times D \times$ range t, the point $(0, 0, 4)$ can point to $(1, 0, 5)$ or $(0, 1, 5)$ on the adjacency list but not $(0, 1, 6)$ or $(0, 1, 4)$ or $(0, 1, 3)$. In my code, the nodes are stored as vectors of the form $(w, (a, b), t)$, where w is the weight given from *weightarrlist*, (a, b) denotes a position on a 25×25 array, and t denotes

time. The making of the adjacency list is essentially BFS. The algorithm starts at the starting point at $t = 0$, then adds adjacent nodes at $t = 1$, and for all of those nodes, adjacent nodes are added at $t = 2$, and so on. All the while, nodes that have been seen are kept track of, which uses the assumption that bot 4 backtracking would never be desirable. There may have been a way to skip the adjacency list step and have the UFCS run directly on *weightarrlist*, but this seemed like the easier option in the time that I had.

Once we have our adjacency list, we can run a straightforward uniform cost search. I used Python's heapq for the priority queue, which conveniently makes comparisons based on the first component such as $(1, (1, 0), 1) < (5, (0, 1), 1)$. Then the uniform cost search returns a list of tuples that represent the path that bot 4 will take. For each main simulation, this hypothetical simulation process only happens once. I am sure there is a way to improve the bot by making it do more simulations at more steps, but I found I needed a lot of tests to get statistically significant results, and simulating fires to plan for the main fire at multiple steps was very computationally expensive, very fickle, and difficult to get to work correctly.

The last, and an important feature of my bot 4 is a safeguard/fire detection capability. At every step of the main/actual simulation, the bot looks ahead to see if at any tile in its planned path, there is an adjacent fire cell. If this is the case, then the bot reverts to doing whatever bot 3 would do.

3 is_possible_sol

I implemented a feature in my code that estimated how often success was possible. For each main simulation, I stored the main simulation in a list. If every bot failed, I retroactively ran a search through the grids to see if in hindsight, a path was possible, and my code would print the path that could have been taken. The reason my numbers shown are estimates is because if there was a path possible, but the path required t greater than when the main/actual simulation ended (which is when all bots were either on fire on a button), it would not be counted. I figured that since this was probably rare, and I had limited time, this underestimate would suffice, and still be a good estimate to how much the bots could possibly improve. If my code is run and one of these hindsight paths are printed, 200 represents the path that could have been taken but never was. I will call this path the "hindsight bot".

4 Results and Analysis

For each $q \in \{0, 0.05, 0.1, 0.15, ..., 1\}$, I tested the bots 500 times. Each trial was a random placement of bots/button/fire (each bot on the same cell), each trial was on a randomly generated layout, and each trial continuing until each bot is on fire on the button. The results can be seen in figures 3 and 4. In total, the data suggests that bot 4 has a modest advantage over bot 3, which has a modest advantage over bot 2, which has a greater advantage over bot 1.

I was hoping to get a more resounding result, but I think that considering the estimated ceiling of success was an average of .767 across all values of q, and the success rate of bot 4 was an average of .754 across all values of q, it seems to me like that while there is some room for improvement, the ceiling for how much better a bot can be might not dramatically higher than what I achieved with bot 4. According to the graph, it seems like there might be most room for improvement around $q = .3$.

One immediate observation is that bot 4 tends to consistently outperform bot 3 for higher values of q, but not so much for lower values of q. Specifically, for $q \geq .35$, bot 4 outperformed bot 3 a decisive 13 out of 14 times, only drawing once when $q = .95$. However, for $q < .35$, bot 4 loses three times, draws twice, and only wins once. This suggests that bot 4's ability to think ahead about where the fire is is most useful when the flammability of the ship is higher and the fire spreads faster.

A feature of the data is that there is a lot of noise. With 500 trials at each q, the success rates for all of the bots are situated within a relatively small range. Between this, and the clear correlation we can see on the graph, it is safe to say that a bot's success depends mostly on q and where it spawned and the layout. The

q	Bot 1 Success Ratio	Bot 2 Success Ratio	Bot 3 Success Ratio	Bot 4 Success Ratio	Success Possible Ratio
0	1	1	1	1	1
0.05	0.972	0.998	0.998	0.998	0.998
0.1	0.912	0.976	0.988	0.986	0.996
0.15	0.892	0.948	0.96	0.958	0.97
0.2	0.854	0.908	0.93	0.928	0.94
0.25	0.822	0.874	0.908	0.91	0.934
0.3	0.792	0.836	0.852	0.852	0.89
0.35	0.788	0.814	0.836	0.84	0.866
0.4	0.794	0.82	0.828	0.846	0.882
0.45	0.716	0.732	0.742	0.748	0.776
0.5	0.666	0.692	0.708	0.73	0.75
0.55	0.654	0.666	0.684	0.69	0.704
0.6	0.642	0.652	0.662	0.674	0.694
0.65	0.616	0.62	0.618	0.636	0.648
0.7	0.576	0.582	0.602	0.612	0.614
0.75	0.582	0.592	0.59	0.608	0.61
0.8	0.586	0.598	0.61	0.616	0.622
0.85	0.546	0.556	0.562	0.568	0.574
0.9	0.528	0.53	0.536	0.546	0.55
0.95	0.532	0.54	0.542	0.542	0.544
1	0.522	0.524	0.534	0.54	0.548
Average	0.714	0.736	0.747	0.754	0.767

Figure 3: Test results, 500 main simulations per q

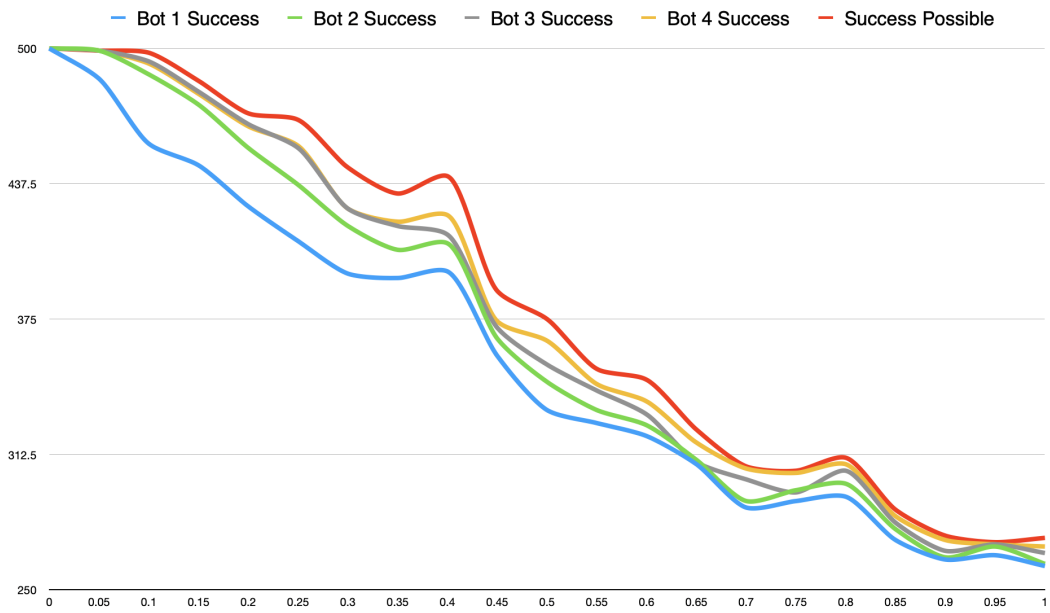

Figure 4: Success out of 500 as a function of q

best thing to do would have been to control for this experimentally, but time did not permit this. However, there is still a way to utilize controls in the experiment, namely, we can have the controls be the success rate of bot 1 and the success possible rates. The idea is that if bot 1 succeeded, then the success of bot 4 probably doesn't mean much, because the shortest, easiest path worked. On the other hand, if the bots all spawned in a way where the fire was impossible to traverse, and no success was possible, then the failure of bot 4 also doesn't say much about how good bot 4 is. It will be helpful to examine the success rate when success is not too easy, and also not impossible. This leads us to a useful statistic for bot $i = 2, 3, 4$:

$$(\#\text{ bot } i \text{ success} - \#\text{ bot 1 success})/(\#\text{ success possible} - \#\text{ bot 1 success})$$

which is approximately the ratio of success in the most meaningful trials, or the ratio of success when controlling for trials where bot 1 succeeds or trials when success is impossible. This statistic has been calculated and can be seen in figures 5 and 6, where n is the approximate number of trials where success is not too easy or too hard. We note that it is rare but possible for bot 1 to succeed when others fail, so this is not a perfect representation of the ratio of success in meaningful trials, but it is a reasonable estimate.

Where before, the data showed relatively small differences between the bots, with our data transformed into this arguably more meaningful context, we can see more drastic differences in the success rates of the bots. For example, for $q = 0.4$, the data indicates with $n = 44$ that the success rate of bot 4 is more than .2 higher. For other q values, the difference is even greater. This data shows pronounced success of bot 4 for $q > .3$, becoming less statistically significant but showing significant advantages to bot 4 as q goes higher, and showing that bot 4 did just a bit worse than bot 3 for low values of q.

It is fair to say from the data that most of the time, q, the layout, and the spawn points will determine whether a bot is successful. But in those critical times when the shortest path fails but success is possible,

q	Bot 2	Bot 3	Bot 4	n
0.05	1	1	1	13
0.1	0.762	0.905	0.881	42
0.15	0.718	0.872	0.846	39
0.2	0.628	0.884	0.86	43
0.25	0.464	0.768	0.786	56
0.3	0.449	0.612	0.612	49
0.35	0.333	0.615	0.667	39
0.4	0.295	0.386	0.591	44
0.45	0.267	0.433	0.533	30
0.5	0.31	0.5	0.762	42
0.55	0.24	0.6	0.72	25
0.6	0.192	0.385	0.615	26
0.65	0.125	0.063	0.625	16
0.7	0.158	0.684	0.947	19
0.75	0.357	0.286	0.929	14
0.8	0.333	0.667	0.833	18
0.85	0.357	0.571	0.786	14
0.9	0.091	0.364	0.818	11
0.95	0.667	0.833	0.833	6
1	0.077	0.462	0.692	13

Figure 5: (# bot i success − # bot 1 success)/(# success possible − # bot 1 success), and n = # success possible − # bot 1 success

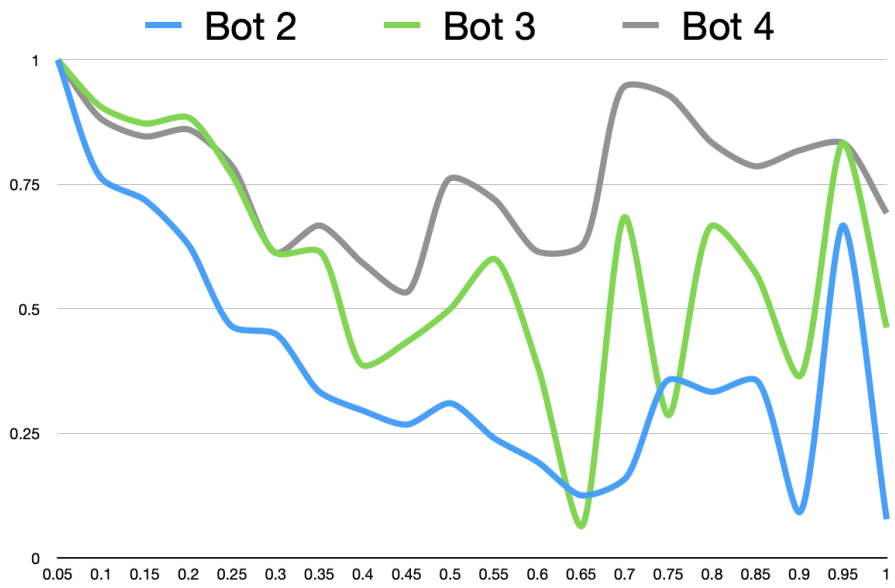

Figure 6: The data from figure 5 graphed

we see a much more pronounced difference in success rates. In the times when it counts, and when q is not too small, bot 4 performs quite well compared to the other bots.

A more thorough statistical analysis would have to be applied to see how likely it is that our results are the result of random chance, as I have observed a good amount of variance in success rates when running different trials with fixed q, but I think my data supports that my bot 4 has serious advantages when q is not too small.

5 Reflections

All in all, the improvements with bot 4 were hard-earned, and were the result of a lot of effort and dead ends (and a helpful suggestion by the professor in office hours).

Of the many dead ends I ran into, there were a few noteworthy ones. There was one bot that I was certain was going to work. It involved running 50 simulations of hypothetical simulations of fires, like the bot that I ended up on, except that it ran a BFS after each full fire to find the shortest path that would've worked, and then created weights so that the path the bot traveled on was the shortest one that worked most often. The reason this failed to compete with bot 3 was that it prioritized finding short paths over finding safe paths. The bot 4 that I am submitting does a much better job of accounting for the fire at particular tiles at the exact time the bot is at that tile.

I also had problems when trying to make the bot smarter. For example, I made a bot that at every step, would run simulations of bot 3 starting at every adjacent tile, and would pick the tile that had the highest success rate. This bot was both incredibly slow computationally and also performed badly in terms of success rates. While it seemed like this bot should have been smarter, it was not evaluating the fire and processing information optimally. Every bot that I made that involved processing simulations at every step suffered from indecisiveness and unnecessary backtracking as well as bad decisions. With some more effort, it's possible that I could get a bot like this to work, but I couldn't get one of these bots to work in time.

As far as bots 1, 2, and 3, it seems like the biggest reason they fail is because they are too short-sighted. This is most evident with bots 1 and 2. They set out on a path but they fail to account where the fire will

be in say, 10 moves. Using the hindsight bot, I was able to see many instances of when the bots fail but there was a path available. Often, this hindsight path suggested the correct move was to decisively set out on a specific path from the start, but bots 1 through 3 would usually miss this path. Bot 4, the only bot with the capability to look ahead many steps, would sometimes be the only one to set out on this correct hindsight path.

As far as failures of my current bot 4, I would say based on the trials I have run, there is room for improvement. There were sometimes were it just seemed to not to do what I intended it to do. It would sometimes run next to a fire when an equivalent but safer path was available, leading to its destruction. With some effort I think I could iron out some of these bugs and marginally improve it. However, there are sometimes when it simply does not make the correct the decision, when the hindsight bot shows a path and my bot 4 just doesn't pick up on it. I think this is harder to fix. I played around with the weights, for example doubling all numbers on each weight array greater than 1, but it was not clear what effect that had. The fact that each cell in each entry in *weightarrlist* is a number between 1 and 51 seems arbitrary to me. I wonder if there was something to do like squaring the weights, doubling the weights, taking the log of the weights that would yield a better program.

To make the ideal bot, one improvement that I thought of was to actually calculate the expected value of there being fire at each tile at each point in time, rather than run simulations. Since bot 4 knows q, it might not be that hard to actually calculate the specific probabilities of there being fire at each tile, and this might be a more accurate and computationally efficient way of doing things, and this computational efficiency may lead to the bot being able to process more information at each step.

I think the ideal bot would take to completion this very idea that I tried and failed at: I think the ideal bot should reevaluate its surroundings at every step. I think there is a way you can get this to work, although I do not know how feasible the computation time would be, and I think it might be tricky to get it to work just right. It might work if you tell the bot it isn't allowed to backtrack; this might fix the indecisiveness problem. But I think even the ideal bot will not be successful every time it's possible. This simulation involves chance, and a bot could make the correct decisions, but get unlucky.

I really did learn a lot from this project. Programming a smart bot was much more finicky than I expected. Getting modest improvements was pretty difficult. But all things considered, I think I was reasonably successful in designing a bot.

6 Bonus

With more time, I think the ideal way I'd try to make a safest layout is to try and implement the evolutionary algorithm we learned in class. I would parent test layouts for safety, create "offspring", and try to figure out a safest layout via natural selection.

One priority of a safe layout might be to minimize the number of times an open square has more than one open neighbor, since the fire is more likely to spread to an open square if it has more than one neighbor on fire. But this has to be balanced against maneuverability of the bots.

My first conjecture was that the safest layout is one where, given a 25×25 grid, all squares with index (i, j) satisfying both $i\%2 == 1$ and $j\%2 == 1$ are blocked off. While there are many squares with four neighbors, I thought this layout would stifle the fire and still have enough maneuverability for the bots to be successful. But upon testing, the success rates for the bots with this layout were very close to the success rates with random layouts, at least at $q = 0.5$.

I did try another layout with the goal of minimizing fire neighbors, and at first was pretty astounded by the results. The layout is 25×25 with all (i, j) open except those points satisfying ($i\%2 == 1$ and not $j == 0$ and not $j == 24$). That is, the layout consists of long hallways stretching over the length of the grid. This gave me the following successes out of 500 at $q = 0.5$, with random spawn points for bots/button/ on the grid:

Bot 1: 443 Bot 2: 476 Bot 3: 479 Bot 4: 479 Success possible: 492

Compare this with random layouts, which gave for $q = 0.5$:

Bot 1: 333 Bot 2: 346 Bot 3: 354 Bot 4: 365 Success possible: 375

Even at a higher $q = .75$, we get remarkable results, with success rates:

Bot 1: 404 Bot 2: 424 Bot 3: 427 Bot 4: 428 Success possible: 442

Compare this with random layouts:

Bot 1: 291 Bot 2: 296 Bot 3: 295 Bot 4: 304 Success possible: 305

These early tests at first glance indicate that this new layout is tremendously safer than the average layout. However, I must disclose an important feature of my project that is probably causing this: *the button cannot catch on fire*. (In all of the previous tests, I don't think this really impacted the tests of quality of the bots). In this particular layout, more testing is needed to determine if this has an inordinate effect on the success rates, and some early testing indicates that this is in fact the case.