Pursuit Evasion Algorithms
Designing Cooperative Algorithms

Ahmed Luqman (24100041) - *CS junior*

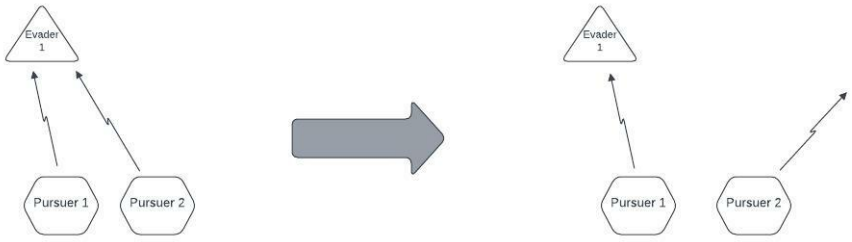
Problem Motivation and Description

Pursuit evasion and motion planning is a very important topic in the design of autonomous robots. It is also a rich area of research.

Our project centers around developing algorithms for robot pursuit and evasion. The problem involves a number of chasing robots (pursuers) and a number of fleeing robots (evaders). The goal of the pursuers is to capture the evaders in as less time as possible and the evaders need to avoid capture for as long as possible. Although this is a very well researched problem, we have decided to look at it from the aspect of Cooperative game theory and system design. We do this by introducing cooperative behaviour in both the pursuers and the evaders. The pursuers interact with one another and based on their common knowledge they can then make decisions on whether a particular pursuit is globally optimal or not. For example if two pursuers are chasing after one evader, and the evader is close to getting caught then it is in the pursuers interest that one pursuer abandon his chase as the other pursuer will suffice in catching that particular evader.

Similarly if two evaders are close to getting caught by one pursuer, than the evaders know that going as far from one another will be optimal and one will in essence "sacrifice" itself - keep the pursuer busy for as long as possible while the other evader gets away - for the survival of their team as a whole.

Figure 1: The evaders agree to separate and Pursuer 2 chooses a different target



Problem abstract

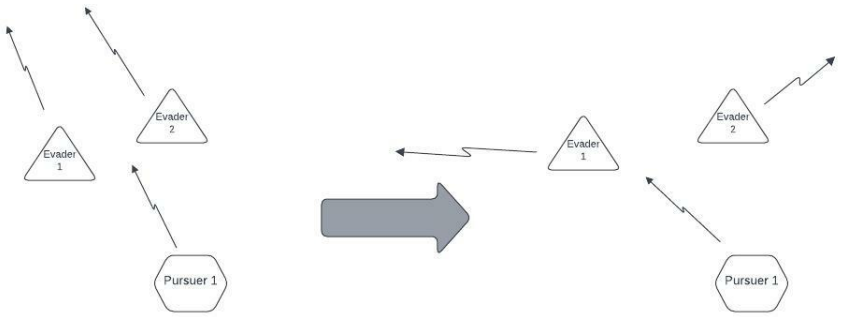
The simulations will be run in a 2-D Grid world, where the number of pursuers "m" and evaders "n" will vary. To thoroughly test cooperative behaviour in different setups we will develop a number of algorithms for both evaders and pursuers

Regarding the actual algorithms, some behavioural specification can be as follows:

(i) For one evader algorithm we can specify an optimal sacrifice scenario, where if certain conditions are met, the particular evader decides that he must now sacrifice himself for the good of the team.

(ii) Another evader algorithm can focus on pre-planned coordination, at the time the pursuit starts, based on the current positions of pursuers and evaders the evaders determine an optimal plan of action and then carry it out regardless of how pursuer behaviour changes with time.

Figure 2: The pursuers decide to go in opposite direction and evader 1 “sacrifices” itself



(iii) One other algorithm which is essentially a milder version of the first sacrificial one is where evaders are willing to help out other evaders but only to the extent of some threshold. For example an evader may be willing to intercept a pursuer but only when it is at a distance of >5 blocks of it.

(iv) A possible algorithm for the pursuers can focus on them breaking into groups or pairs temporarily and when they are sure that the evader can be captured by a single pursuer the rest of them change targets.

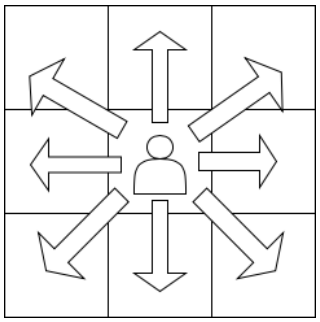
(v) We can test the other algorithms against a benchmark by creating a standard algorithm that uses vornoi partitioning to minimise the area of escape for the evaders, we can check whether introducing cooperative behaviour makes it easier for the pursuers to capture the evaders

Mathematical formulation of the problem

First we will describe the environment that these algorithms will be tested in. We assume a m by n grid with certain obstacles (We denote obstacles as inaccessible grid sections). An example of a viable “map” is given on the next page:

The distance metric used for path finding is the Chebyshev distance.This restricts our robots movements to 8 moves.

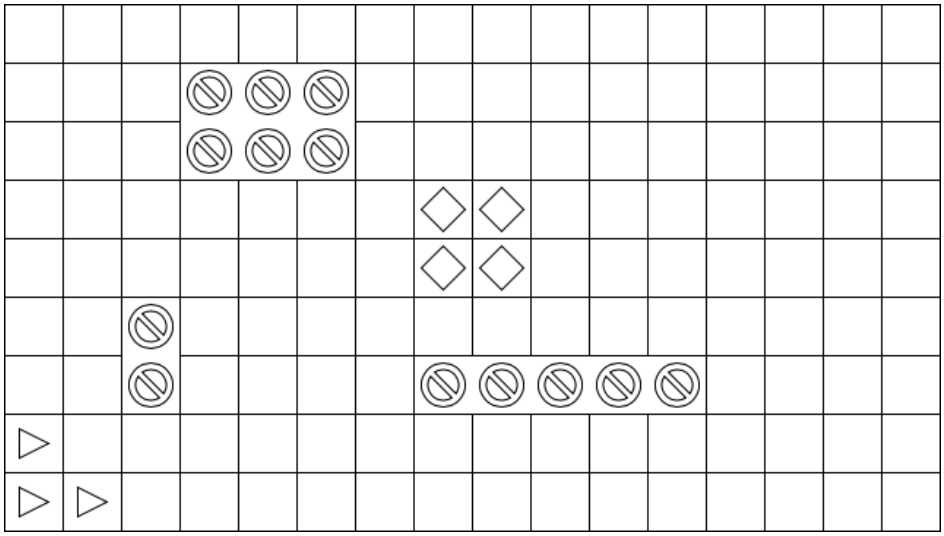
Figure 3: Namely moving to the upper right, upper, upper left, right, bottom right, bottom, bottom left, and left cells.



All entities on a given map move simultaneously i.e per movement round all entities can only move to one cell each or stay in their current cell. However, if an evader were to move to a cell that was occupied in the previous round by a pursuer and the pursuer moves to a cell occupied by an evader than this will be counted as an “encounter” and the evader will be considered as caught.

The chase begins after generating a grid map and placing the pursuers and evaders onto their starting cells. We will now go over the algorithms used to model the chase, both for the evaders and the pursuers.

Figure 4: A 9x16 grid map with 3 different obstacles denoted my the area consisting of stop symbols. The triangles depict the starting position of the pursuers and the diamonds are the evaders



Algorithm Specifications and Design

Pursuers

We begin by introducing a naive algorithm that employs no communication and cooperation among the pursuers. To think about such an algorithm we simply take a greedy approach and assume that at the start of the game every pursuer is "assigned" an evader and for the rest of the game his sole purpose is to catch this evader, once this evader is caught, he is then assigned a new evader. This naturally formulates as a greedy approach for which at every step the pursuer simply wishes to minimize his distance to his assigned target. The algorithm ends up taking this kind of a form:[1]

Algorithm 1 Func: Pursuer(Assigned Target - x, Targets occupied cell - s[i,j])

```
Ensure: next_cell is in the map and not coinciding with an obstacle (Dist() takes these limitations into
account)
curr_state ← s[i,j]
while All Evaders not captured do
  next_cell ← Dist(curr_state)
  curr_state ← next_cell
  if Evader captured then
    Rem_evaders[].pop(x)
    next_target ← Rem_evaders[0]
  end if
end while
```

Here "Dist()" is a helper function which simply tests all possible positions the robot can move to and chooses the position that minimises its distance from its target and returns that move to be played next

[1]The algorithms actually written in python involve a lot more array manipulation, those details are abstracted away here

round.[2]

Now we try to refine this algorithm to introduce some cooperative behaviour.We will try to bring improvements in our naive algorithm one by one.First of all we can notice that in the sample map given above, there are objects around which the evaders can simply circle around infinitely, thus evading capture forever, so assigning one pursuer per evader instantly seems like a bad idea. So we can see that it would fair the pursuers better if they "ganged up" on a particular evader. So our next improvement on the naive approach would be to introduce this very basic notion of teamwork. So we add an extra argument for the positions of the "teammates" of the pursuer. Our Distance formula now becomes more complicated since it will give the next optimal cell keeping in mind the position of the other teammates. For now we abstract away the details of how exactly this optimisation happens but we can prove that this indeed does solve the problem of pursuers circling around obstacles to avoid capture indefinitely.

Take any evader k, currently residing in cell (i, j), assume (without loss of generality), the obstacle the evader uses to pivot around to evade capture is a 2x2 square. For any possible direction of approach from a pursuer, the evader can pivot around this square by either moving to the right or the left.Thus in this simple case two evaders suffice to block both directions of escape (view fig 5). Now we move onto a more complicated obstacle or a set of obstacles, we can show that even given this obstacle, two pursuers are enough to remove the "infinte circling" strategy of an evader.

Consult Figure (5b). Take the starting cell for the evader as (i, j), possible moves:

$$(i, j) \rightarrow (i, j + 1)$$

$$(i, j) \rightarrow (i, j - 1)$$

$$(i, j) \rightarrow (i + 1, j)$$

$$(i, j) \rightarrow (i - 1, j)$$

Every such possibility signals the evaders intention to try to pivot around either of the four separate "blocks". For $(i, j + 1)$, he wants to use either of the top two blocks for pivoting. For $(i, j - 1)$ he intends to use either of the bottom blocks, and so on either the left two blocks or the right two blocks for a left movement and right movement respectively.

Taking symmetry into account we just need to prove that even one of these decisions will not help him carry out "infinte circling" and the rest of them also end up being equally useless.

Assume $(i, j) = (4,4)$ the following set of steps for this evader and corresponding response moves by each pursuer below it.

$$(4, 5) \rightarrow (4, 6) \rightarrow (4, 7) \rightarrow (4, 8)$$

$$(3, 2) \rightarrow (4, 3) \rightarrow (4, 4) \rightarrow (4, 5)$$

$$(1, 3) \rightarrow (2, 4) \rightarrow (3, 4) \rightarrow (4, 5)$$

Algorithm 2 Func: Pursuer(Assigned Target - x, Targets occupied cell - s[i,j], Pos_Teammates[])

```
Ensure: next_cell is in the map and not coinciding with an obstacle (Dist() takes these limitations into account)
curr_state ← s[i,j]
while All Evaders not captured do
  next_cell ← Dist(curr_state, Pos_Teammates[])
  curr_state ← next_cell
  if Evader captured then
    Rem_evaders[].pop(x)
    next_target ← Rem_evaders[0]
  end if
end while
```

[2]Dist uses breadth-first-search at its core with some minor modifcations

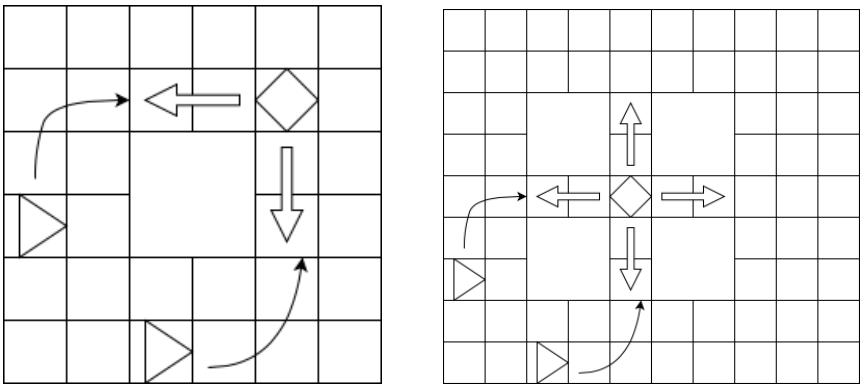


Figure 5: Figures 5(a) and 5(b) - obstacles denoted by white space

It seems wasteful to engage in pathing that puts two pursuers on the same grid cell but this is infact essential to "flush out" the evader from this potential pivoting cycle, once both pursuers reach this center cell $(4, 5)$, one pursuer begins to move towards the evader while the other waits for one round to see what the evader does, and then based on the evaders movement he then moves in the direction of escape and the chase resumes, and the evader is restricted from entering this object setup again.

So now we add this extra detail to our algorithm to avoid infinite chase scenarios, thus we have our third version of the algorithm, which introduces a function called Flush_out, which just coordinates movements in an attempt to get them to the center of a pivotable object setup:

Algorithm 3 Func: Pursuer(Assigned Target - x, Targets occupied cell - s[i,j], Pos_Teammates[])

```
Ensure: next_cell is in the map and not coinciding with an obstacle (Dist() takes these limitations into
account)
curr_state ← s[i,j]
while All Evaders not captured do
  if Evader in circuling attempt then
    next_cell ← Flush_out(s[i,j], curr_state, Pos_Teammates)
  end if
else
  next_cell ← Dist(curr_state, Pos_Teammates[])
  curr_state ← next_cell
  if Evader captured then
    Rem_evaders[].pop(x)
    next_target ← Rem_evaders[0]
  end if
end while
```

Now our algorithm guarantees eventual capture given at least two pursuers regardless of how the objects are positioned. We can now focus on efficiency and making our eventual capture scenario happen faster if we make the pursuers divide their efforts more effectively. For this we set up a type of cost sharing mechanism to encourage the pursuers to take up pursuits that give greater reward. That is, we need to find a way to get the pursuers to understand that chasing a target with 2-3 pursuers already on its trail, is not a desired decision and in a way "the more the difficulty of a pursuit the greater the benefit to the pursuer to engage in that chase". This is where we introduce our actual game theoretic concepts into the chase. We incentivize pursuers by offering a bounty on each capture, this bounty is periodically updated to reflect the difficulty of capture for each evader. The pursuers are now tasked with the goal of maximising their bounty by the end of the chase. To ensure that cooperation is still desirable we need to find an effective cost sharing mechanism for captures so that engaging in group chases does not end up only being beneficial for the pursuer who

ended up finally capturing the pursuer, but rather the helping agents also get compensation for their efforts. So a rough algorithm for to employ this cost-sharing mechanism would be as follows:

Algorithm 4 Func: Pursuer(Assigned Target - x, Targets occupied cell - s[i,j], Pos_Teammates[])

```
Ensure: next_cell is in the map and not coinciding with an obstacle (Dist() takes these limitations into
account)
(Target reassignment function gives new target and its curr_position)
x, curr_dist = Target_Reassign(s[i,j], curr_state, Pos_Teammates)
curr_state ← s[i,j]
while All Evaders not captured do
  if Evader in circling attempt then
    next_cell ← Flush_out(s[i,j], curr_state, Pos_Teammates)
  end if
else
  next_cell ← Dist(curr_state, Pos_Teammates[])
  curr_state ← next_cell
  if Evader captured then
    Rem_evaders[].pop(x)
    self.reward ← self.reward + bounty[x]
    x, curr_dist ← Target_Reassign(s[i,j], curr_state, Pos_Teammates)
  end if
end while
```

When thinking up a good mechanism for score allocation or cost per pursuing a particular evader we can think of possible factors that would wither make a specific evader more desirable to pursue or more undesirable.

For example let us take some metrics and see how they might affect these decisions.

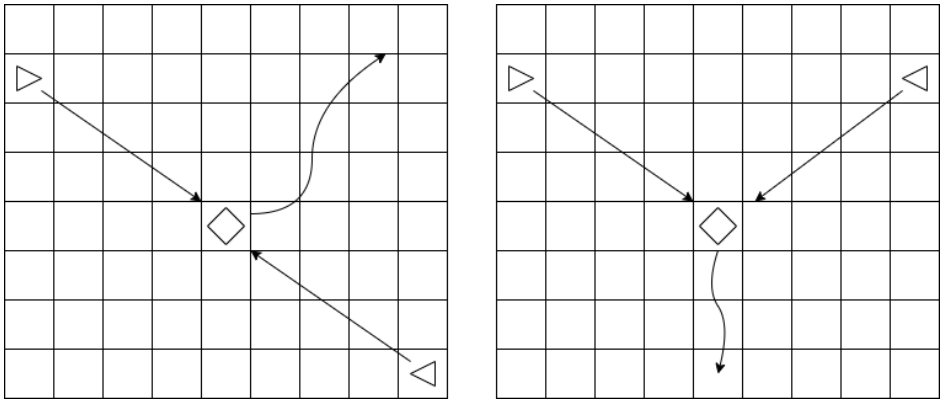
- First and foremost, the distance between the pursuer and the evader is the primary metric used to gauge desirability when choosing a target assignment, it seems reasonable enough that a target 20 blocks away seems less important to the pursuer than a target 5 blocks away from him, so naturally, the higher the distance the higher the cost assigned to that evader.
- This, however, does not seem to be much different from our initial naive algorithm, how do we incorporate an aspect of cooperation here, for example if the target 5 blocks away was already being chased by two other pursuers then in that case chasing it too seems like a waste of resources. In that case the distances between the other pursuers and that evader are also of interest to us so let's consider them when assigning a score too, we subtract the distances of each of the other pursuers from a particular evader from the cost of that evader. This is reasonable because lower distances from other pursuers makes this evader a more attractive target for our pursuer
- Another important metric is the distance of the other pursuers between each other, this ties in closely with the idea of converging to a specific point, the closer the other two evaders are to each other the better they will act as a sort of "net" to close off escape routes, this is illustrated in the image below. Two pursuers close to an evader in addition to them being close to each other makes that target undesirable for assignment
- There are also other factors we can consider like the distance of the evaders between each other but these three factors above will suffice to make our algorithm noticeably better than the initial algorithm we drafted at the start of the paper

Our final formula for assigning scores to evaders following a game theoretic framework is as follows:

$$cost = Dist\_from\_evader(p0) \; - \; Dist\_between \; * \; (Dist\_from\_evader(p1) \; + \; Dist\_from\_evader(p2))$$

cost is the assigned cost per evader for each pursuer, $Dist\_from\_evader(p0)$ is the Distance of a particular evader from the pursuer in consideration, *Dist_between* is the distance between the other two pursuers, $Dist\_from\_evader(x)$ is the distance between the pursuer "x" and the particular evader. *Dist_between* acts as a scaling factor, larger the distance between the two pursuers and the the distance between the pursuers and the evader, the greater the reduction in cost for us targeting that particular evader.

Figure 6: The distance between the attackers and the evader is the same in both cases, however, on the right they are better situated to capture the evader



Now we move onto some simulations to show that our new algorithm is indeed better at catching the evaders than simple A* search based non-cooperative algorithms.

The code is available at following repository:

`https://github.com/AhmedLuqman26/Pursuit_Algorithms.git`;[3]

Simulations and models

In our setup we use a 12x12 square grid by default, we've limited the evaders to 2 and the pursuers to 3, however, this is enough of scale to demonstrate noticeable ideas. The code can very simply be modified to cater to different scenarios and the algorithm we design quite naturally extends to more pursuers and evaders too as the fundamental factors affecting target assignment stay the same regardless of the scale of the problem.

In our first scenario the setup is as follows, the grid below represents the "map" for the game and labels "a", "b" and "c" correspond to the initial positions for each of the three pursuers, 1 and 2 are the labels for the evaders and the grid blocks marked with an "X" represent obstacles. To abstract out some complications and drawn out chases, Evaders are controlled by a human user and have four movements, up, down, left and right. Pursuers can also take two steps at any given moment within the game, which is termed a "jump", however, once they use this jump, they cant use it again for the remainder of the game. This also includes an extra element of strategy. We have spaced out all the entities to make for a more interesting setup.;[4]

In the initial setup we can see that a reasonable strategy seems to be to use evader 1 to distract c and b and let 2 go to the gap between the two obstacles to escape to the upper half of the map. This strategy works because b and c are not cooperating so both of them go after evader 1 instead of converging on both the evaders collectively. After a few more moves this strategy plays out as we wanted:

[3]The code requires some manual adjustment when shifting from one algorithm to the other, these sections to remove/add are contained within doc-strings i.e "'example code"'

[4]jumps may not be clearly visible in the simulation output, however for the actual code, when a pursuer makes a jump it is explicitly mentioned below the grid output for clarity

Figure 7: The initial configuration for the first setup

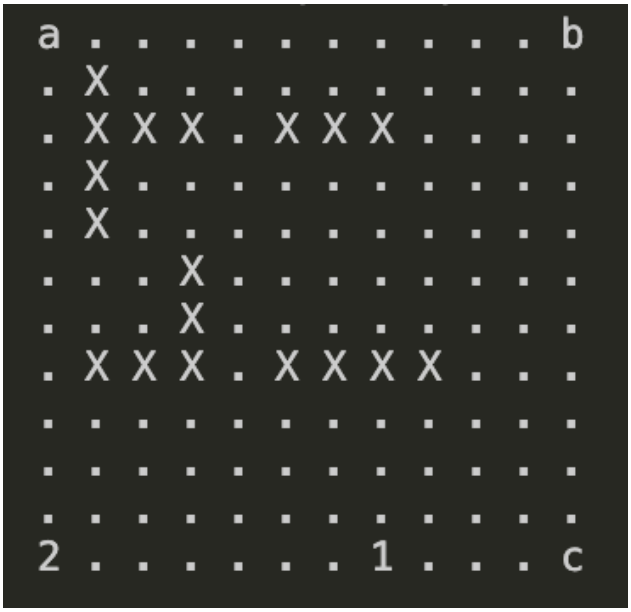
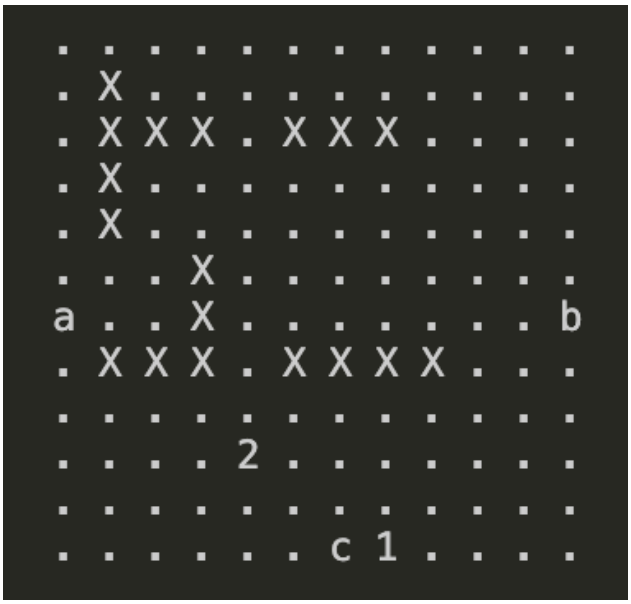


Figure 8: *b* starts to target evader 1 and essentially "takes the bait"



Evader 2 then successfully escapes and manages to prolong the chase to 25 moves

Figure 9: 2 has escaped with all evaders pursuing him (a and c happen to overlap since they are in the same cell)

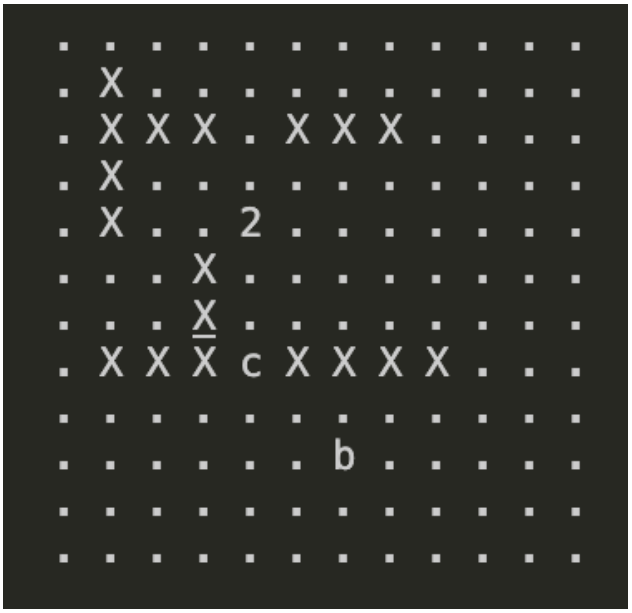
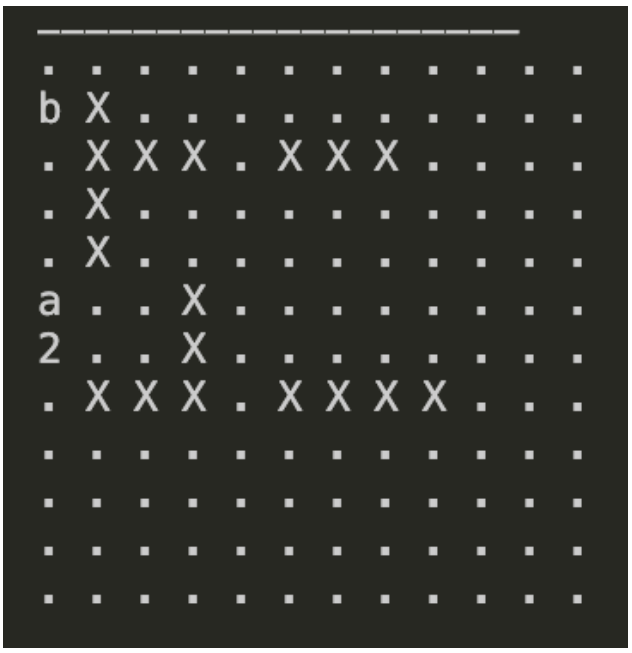


Figure 10: 2 gets captured finally after a uses his "jump" upon getting close



This chase was prolonged because b did not make the decision of closing off the exit for evader 2, we will now run the simulation using our cooperative algorithm and demonstrate how it is more optimal.

The initial setup is the same as the one used above;[5]

Figure 11: The initial configuration for the second setup

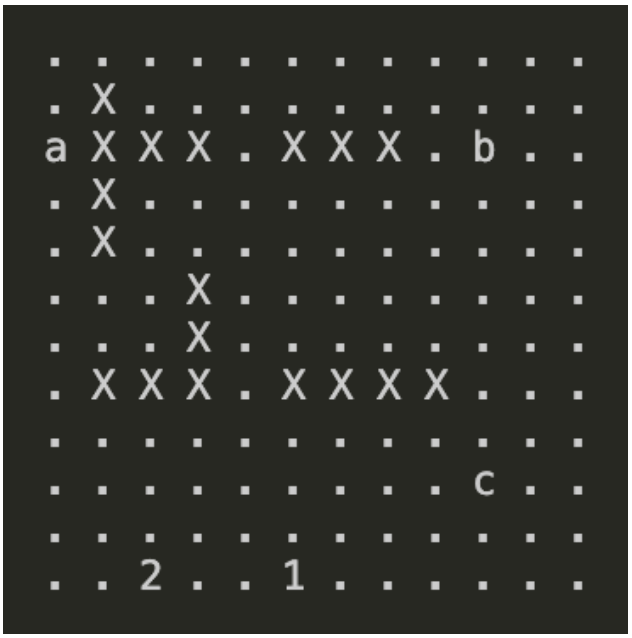
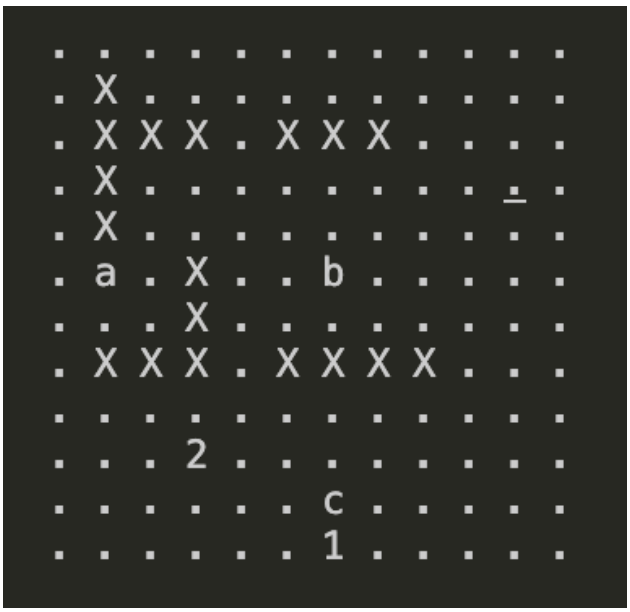
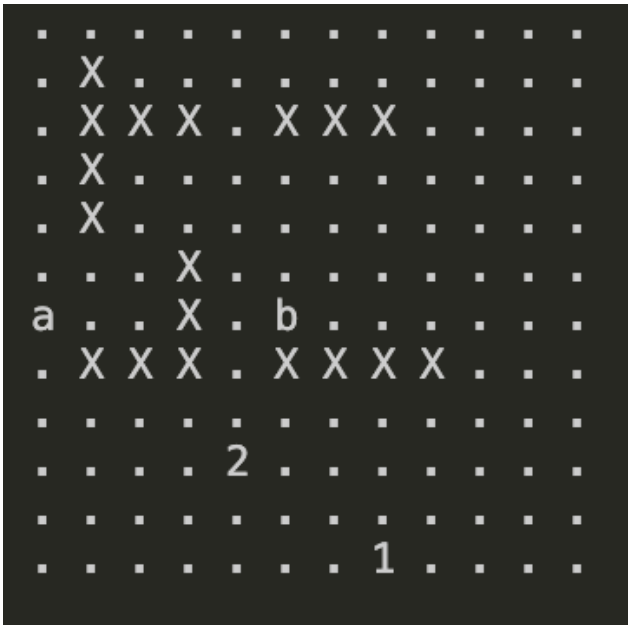


Figure 12: b paths differently due to our modified cost assignment



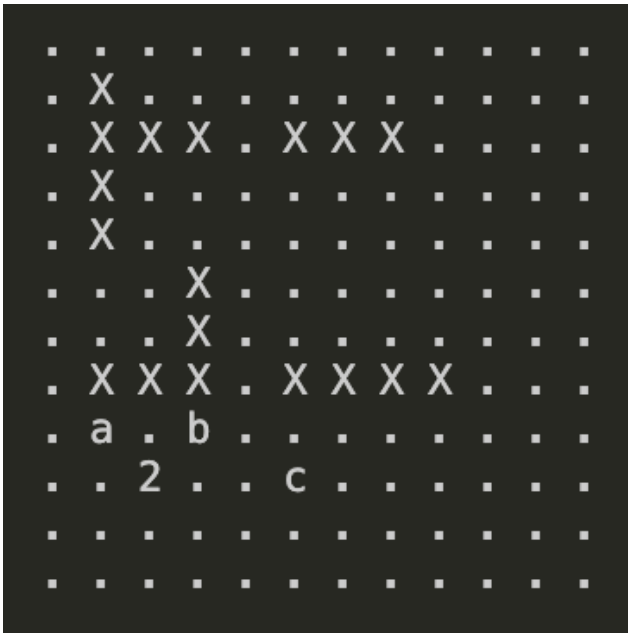
[5]Due to the way the grid is outputted, labels may overlap showing only one entity at a particular cell ,however, it actually may contain others as well

Figure 13: Evader 1 is already captured under 6 moves



[6]

Figure 14: Both evaders get captured within 11 moves, which is the optimal time for any capture on this map (since mathematically not moving would also result in capture within 11 moves)



Our target assignment mechanism gives the most significant improvements when the map includes several obstacles. Given any kind of map our new algorithm performs at least as good as the naive algorithm in the

[6]In figure 13, evader 1 is captured by c, however, due to the order of printing of labels, "1" overwrites "c", this is fixed on the next iteration

worst case and way better average performance practically. Since for this setup the moves taken for capture were brought down from 25 to 11.[7]

Evaders

In this paper we do not formally introduce an algorithm for the evaders, however some points that should be considered when designing an algorithm for the evaders are mentioned below:

- A straightforward approach would be (similar to the original pursuer algorithm) to always maximise the distance between the nearest pursuer and yourself. This is simply given by choosing the move that gives the highest path cost using a simple depth-first-search algorithm and execute that move - essentially the opposite of how we used depth-first-search for the pursuers.
- Cooperation can be introduced by using concepts such as misdirection, baiting and sacrificing etc.
- Unlike our algorithm for the pursuers evader algorithms would be very susceptible to the kind of obstacles the map has, which makes it difficult to make a general purpose efficient algorithm, That is why, for the purposes of the above simulations the evaders are user controlled.

For the sake of completeness, here is some basic psuedocode to build upon for a potential evader algorithm:

Algorithm 5 Func: Evader(curr_pos, pos_pursuers[n])

```
Ensure: next_cell is in the map and not coinciding with an obstacle (Dist() takes these limitations into account)
curr_state ← curr_pos
while All not captured do
  next_cell ← Modified_Dist(curr_state)
  curr_state ← next_cell
  if Alter_behaviour(curr_pos, pos_pursuers[n],pos_evaders[n])[0] is True then
    curr_state ← Alter_behaviour(curr_pos, pos_pursuers[n],pos_evaders[n])[1]
  end if
end while
```

For the sake of completeness, here is some basic psuedocode to build upon for a potential evader algorithm: Here *Alter_behaviour*() is our main function for deciding how the evaders cooperate with each other, we pass it our position along with the other evaders and the pursuers. It returns true/false depending on whether deviation from default path finding is beneficial followed by the next move that it suggests. It could decide this by assigning scores to all possible moves and weight the benefits. However, there could be various equally effective ways of calculating the best move, as such we do not delve further into these detail as they are out of scope of this paper

[7]Certain decisions made by the pursuers do incorporate some randomness, especially when multiple paths give the same distance, then one is randomly chosen, so exact results may vary by 2-3 rounds if the user chooses similar moves