Pacman – Capture the Flag

Abstract

Inspired by the original classic game, the multiplayer variant of Pacman called Capture the Flag[1] aims to provide a challenging planning domain in which teams of AI controlled agents go head-to-head in order to eat the most food and survive to tell the tale. This report explores various strategies and techniques in cooperative AI planning in order to create a team of agents that can effectively operate in Pacman CTF domain, utilising PDDL, heuristic search and approximate Q-learning in achieve these goals.

Workflow Overview

Agents operate according to a specific workflow, which aims to split computation of high- and low-level goals and plans into distinct but related steps. First, the game state is converted into a PDDL representation to be reasoned over. From this, an overall strategy is chosen by solving a high-level PDDL problem, where the goal is definitive but not descriptive. From this high-level goal, a high- and a low-level plan is constructed. The high-level plan outlines the broad strategy to be taken, such as *attack*, *defend*, *go home* and *patrol* being the default PDDL actions provided in *myTeam.pddl*.

The low-level plan is generated by taken the high-level action generated from the PDDL solver and converts it into concrete actions to be taken. For example, if the high-level goal is to *attack*, the low-level plan would be taking one of North, South, East or West cardinal direction steps to reach the enemy's territory. If this plan cannot be satisfied mid execution, then both the high- and low-level plans should be revaluated and replanned.

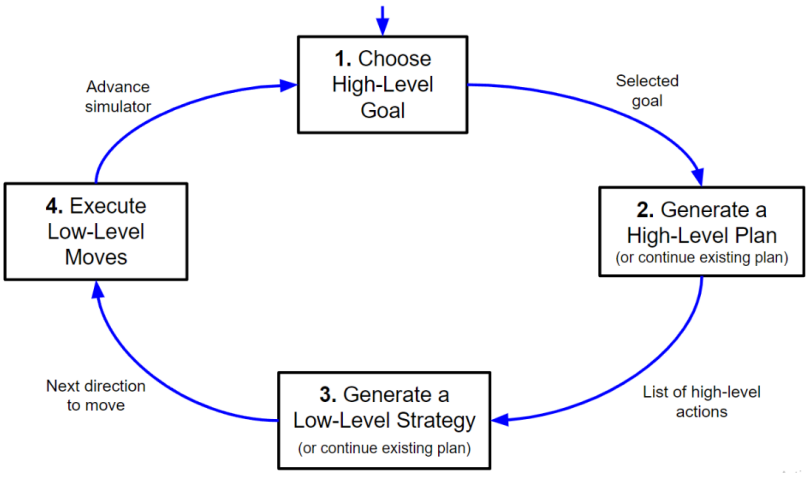

Figure 1 - Agent workflow transition model

As part of this workflow, different strategies for generating these plans can be mixed and matched in order to get more complex behaviour. A high-level goal can have multiple low-level plans which are all feasible and satisfy the goal. Depending on the scenario, an approximate Q-learning approach may be more desirable for one situation, while a simple A* heuristic search may suffice for another. This allows many avenues for expression and freedom in designing strategies for the agents to use.

[1] UC Berkeley CS188 Intro to AI – Course Materials: http://ai.berkeley.edu/contest.html

Baseline approach analysis

It is important to understand the abilities and limitations of the baseline implementation before improvements can start being made.

High-level actions

Firstly, the high-level actions are described as a decision chart below. The PDDL actions and relevant pre-conditions and effects are described in text to illustrate the general flow of the baseline agent.

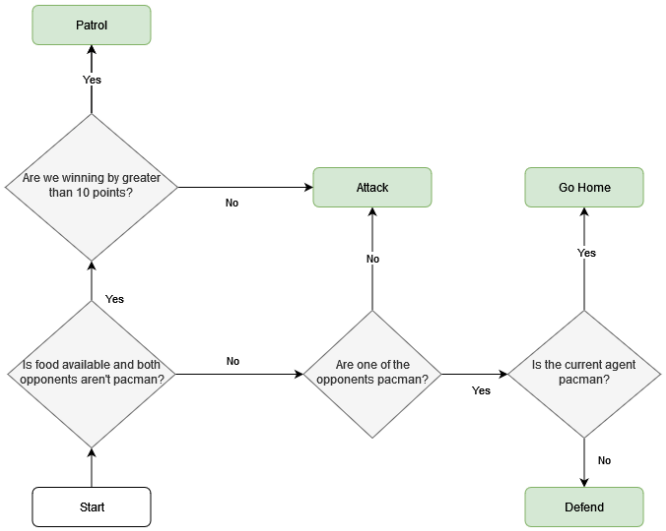

Figure 2 - Baseline agent high-level action flow

There are only four possible actions to be chosen from: attack, defend, go home and patrol. The overall decisions that prompt these actions are described in the triangle boxes. The first thing to note is that this approach is complete albeit rudimentary: all basic game mechanics such as movement, scoring and defensive behaviour are covered.

Beginning in the bottom left, we first want to know if there is any food still available, and if both opponents are not Pacman (i.e., they're not in our territory). If yes, we check whether we are already leading by 10 points, in which case we assume guaranteed victory and just **patrol** our agent's territory using the defence approximate Q-learning strategy. If no, we **attack** which involves invading enemy territory. If the high-level plan evaluated to no at the initial step, then we check if one of the opponents are Pacman. If not, again we **attack**. If yes, then we need to check whether we are in our own territory or in the opponents. If we are Pacman (i.e., we're in the opponent's territory), we first need to **go home**. If we are a ghost (in our territory), we commence the **defence** action.

Baseline Limitations

There are some limitations with this baseline implementation. When the current agent Pacman is powered up by a capsule, it does not know that it can eat the enemy, as the offensive feature *#-of-ghosts-1-step-away* causes the attacking agent to recoil and move away from the enemy ghost.

The defence action often results in the agents just hanging around some food or in their spawn point, allowing the invaders to gobble up food while staying outside the visible range. The baseline is limited in its predictive power to potentially move towards areas where invading opponents can be caught out.

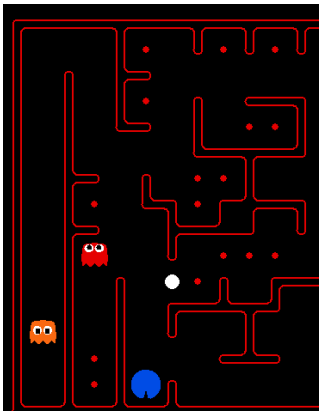

Figure 3 – In the baseline defence, the left most ghost continues to patrol the spawn though there is no food there.

Overall, the baseline agent is underpowered when it comes to defensive abilities and is very simplistic in its overall offensive goals. This is highlighted in the experiments section, where the baseline defence can be very easily exploited.

High-level goals

There are only two PDDL goals in the baseline implementation; *goalScoring* and *goalDefWinning*. These are described below, split into positive and negative goals, only with any relevant pre-conditions.

Goal	Description	Pre-condition(s)	Positive goal(s)	Negative goal(s)
goalDefWinning	Want to defend all foods after the score is greater than 10.	winning_gt10	defend_foods	-
goalScoring	Goal is to eat all foods on the opponent's side.	-	-	food_available is_pacman enemy1 is_pacman enemy2

These goals are very basic and essentially represent a very fixed gameplay loop: unless you're winning by more than 10 points, the goal is to eat all of the opponent's food and keep the enemy outside of your territory. Once you have a large lead, the goal is to defend your territory indefinitely until the score changes and the 10-point lead is lost.

Baseline features

The baseline implementation is a purely approximate Q-learning agent, with three feature sets for offensive, defensive and escape actions. When *maze_distance* is referred below, this refers simply to manhattan distance as that is the default behaviour when a distancer has not been instantiated by the CaptureAgent object.

Offensive features

The following features are for offensive behaviour, mapping to the *attack* high-level action. These features promote agent scoring and aggressive evasion behaviour.

Feature	Description	Value
successorScore	Describes the score function used to evaluate the reward from one state to the next.	$\frac{score(succ)}{width + height} * 10$
bias	Captures bias such that maximisation of Q-function	1

	does not lead to overestimation of Q-value.	
#-of-ghosts-1-step-away	Sum of Boolean evaluations where the current agent's next position intersects with a possible successor position of a ghost agent.	$\sum_a succ\_pos_a \in ghost_{pos_e} \forall enemies$
chance-return-food	Weights the number of food carried by the normalised maze distance from home	$nCarry * (1 - \frac{dist\_home}{width + height})$
closest-food	Normalised maze distance to the closest food if it exists	$\frac{dist\_food}{width + height}$

Defensive features

The following features are for defensive behaviour, mapping to the *defence* and *patrol* high-level actions. These features promote defensive capabilities, though are a little bit too passive. One thing to note is that features are not normalised, therefore when trained these weights increase drastically.

Feature	**Description**	**Value**
onDefence	Describes if current agent is a ghost. Acts as the bias for defensive moves.	$1\ if\ ghost$
teamDistance	The maze distance between agents on the same team.	$maze\_dist(pos_{a_0}, pos_{a_1})$
numInvaders	The number of invaders the agent can see.	$\lvert invaders \rvert$
invaderDistance	Maze distance from closest invader the agent can see.	$\min_{i \in invaders} maze\_dist(pos_a, pos_i)$
stop	If action is STOP set to 1	$1\ if\ action = STOP$
reverse	If next action is reverse of current set to 1.	$1\ if\ action = REVERSE$

Escape features

The following features are for escape behaviour, mapping to the *go_home* action. These actions promote returning to home and avoiding enemies while doing so. One thing to note is that features are not normalised, therefore when trained these weights increase drastically.

Feature	**Description**	**Value**
onDefence	Describes if current agent is a ghost.	$1\ if\ ghost$
invaderDistance	Maze distance from closest invader the agent can see.	$\min_{i \in enemies} maze\_dist(pos_a, pos_i)$
stop	If action is STOP set to 1	$1\ if\ action = STOP$
distanceToHome	Maze distance to spawn point of current agent.	$maze\_distance(pos_a, spawn\_pos_a)$

Rewards

The baseline implementation of Q-learning only contains rewards for offensive actions. The reward for offensive actions is given as below, where final reward is a sum over all the rewards.

Description	**Value**

Base reward for number of returned food and number of carried food for the next state.	$-50 + nReturned + nCarry$
If there's a ghost one step away	-5
New food has been returned	$10 * nReturned$
If score < 0	$score$

The base reward for any given state is therefore given by the linear sum below:

$$reward = -50 + nReturned + nCarry - 5 + (10 * nReturned) + 1_{score}$$

$$where\ 1_{score} = \begin{cases} score\ if\ score < 0 \\ 0 \qquad else \end{cases}$$

There are many examples online of Pacman Capture the Flag projects being completed by students. Many design decisions are inspired and tested from a variety of these projects. The unique aspect of this report is that the workflow has been explicitly set as described in the previous section, thus the strategies in projects found online by nature must be translated to the high-level plan, low-level plan workflow structure. Nonetheless, the ideas are very much applicable.

To start understanding the Pacman environment and get immersed in the sorts of strategies that may be feasible, a good place to begin developing the agents are to simply try and recreate some of the strategies seen online and evaluate what may make them work and some improvements that should be made.

Implementing heuristic search

Sometimes it is simpler and more appropriate to specify a goal location for an agent to reach. Heuristic search planners enable us to provide concrete direction for our agents. Agents utilise the A* search algorithm it is guaranteed to find optimal paths given an admissible and consistent heuristic.

A* finds the lowest cost solution by expanding nodes in the open list ordered by the function $\mathrm{f(n)} = \mathrm{g(n)} + \mathrm{h(n)}$, where $\mathrm{g(n)}$ is the current cost from the start node to node *n* and $\mathrm{h(n)}$ is the heuristic function calculated between node *n* and the goal node. To facilitate this, the implementation in *piglet* allows for a seamless integration with the Pacman domain, with some small added checks.

Heuristic search attacking

The first test was to implement heuristic search for attacking. The search goal node would be the position of the closest food. This ended up creating a greedy agent, which would simply path find to the next food. It was also a very basic and foolish agent that would never deviate from the attack path and run into ghosts. In order to handle enemy ghosts, one could implement the ghost locations as obstacles, and in every iteration of the game check if the path collides with the enemy location. The agent would be replanned if the ghost ends up on the path of Pacman, allowing the attacking agent to plan around the ghosts.

However, this came with some issues, where a plan could not be found for Pacman as all available actions would result in either colliding with a wall or a ghost, returning an empty low-level plan. This would also mean that Pacman's behaviour would need to be handled almost like a decision tree in attacking, and this could leave holes in the attacking strategy. Thus, the decision to utilise heuristic search for offensive actions was abandoned, however it was found to be very useful for defensive actions as discussed later.

Rushing into enemy territory using heuristic search

Heuristic search does have its uses for simpler behaviours. A separate move for getting into enemy territory as quickly as possible is planned using heuristic search, which finds the shortest path to a given goal location in the enemy territory. In this case, either the closest border point or a random border point is chosen, depending on proximity to the border. This is discussed more in the high-level actions section under the *Rush* action.

Developing smarter agents

The game state offers a large amount of information that can be directly and indirectly leveraged to build smarter agents. Some of the strategies rely on inference made from the game state information, which is not provided in the baseline implementation. Below are some of the information and strategies used to attempt towards building more competent Pacman agents.

Tracking disappearing food

On the defending side, we have full information of the food we are defending. Given we have noisy readings to enemy agents but perfect knowledge of food, we can leverage this information to locate enemies more accurately. When defending, we can use our own food as an impromptu "radar", where any food that disappears "pings" our agent where they can utilise that signal and respond to it.

The *CaptureAgent* class contains the history of game state observations. As part of this, we can directly compare the current and previous game states, get the food as a list, and perform a set difference to discover which locations have had missing foods.

Food density

Many areas contain multiple foods in a short space. Depending on the current state, it may be beneficial to target areas where there is a higher food density for either offence or defence over alternative locations that are less dense. Inspired by Team Alpha [5], the concept of a density dictionary[2] is created, where for each food dot location, the density is calculated as the number of food dots within a certain radius.

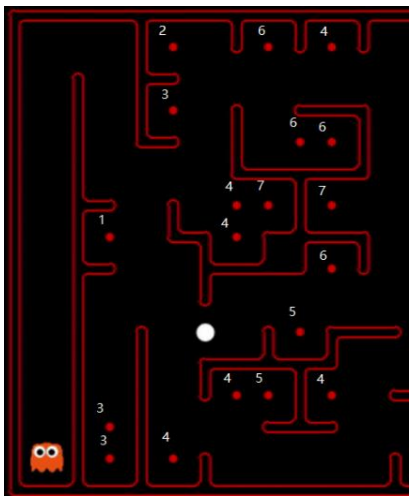

Figure 4 - Food densities for red team with radius=3

[2] Adapted from Wang. Y, Fu. H, Liang.Y. (Team Alpha). Project: https://github.com/infinityglow/COMP90054-Pacman-Contest-Project.

This is a simple calculation that allows the defending agent to prioritise certain areas over others. For example, the area where the food densities are six and seven would be better to defend than the bottom left where realistically you'd only get two food dots in the long dead end and would be difficult to spot and respond to invaders.

	Algorithm 1: FoodDensity
1	**Input**: map width w and height h, list of foods F, radius r
2	density = {}
3	**foreach** $(x, y) \in F$
4	$x_{range} = max(1, x - r) \ldots min(x + r, w - 1)$
5	$y_{range} = max(1, y - r) \ldots min(y + r, h - 1)$
6	$density_{x,y} = \sum_{xr \in x_{range}, yr \in y_{range}} food(xr, yr)$
7	**return** density

Algorithm 1 - Food density calculation algorithm

Algorithm 1 above describes the calculation of food densities. It takes a list of foods, along with a radius for which densities will be checked. Then for each food location, we count the number of foods within that radius, and store it in a dictionary density. Boundary checks for map height and width are accounted for.

Border as a landmark

Some actions and strategies the agent would like to take would require knowledge of where the border between territories is. For example, how should the agent know where is the closest food to the enemy side is? The border positions are simply calculated as all traversable y co-ordinates where the x co-ordinate is either half the grid width if the agent is on red team, or half + 1 if on the blue team. This gives another landmark to inform potential actions, such as the border patrol action for defensive moves.

Improving approximate Q-learning offensive behaviour

The baseline offensive agent is complete with regards to offense, but it is not as effective as it could be. Some simple improvements are made to the baseline that increases the effectiveness of the agent.

Eating capsules

The offensive baseline agent did not have any knowledge of what a capsule can do. The Q-learning features only rewarded interactions with food. The first trial was to add knowledge of capsules and reward the agent for being near the capsule. A large reward of 25 was given for being within a step of the capsule. What this resulted in was the agents diving straight into enemy territory for the capsule, ignoring food along the way.

The new improvement was to positively reward agents eating the capsule and being near enemies, while negatively reward agents if they were near enemies but not near capsules. The reward for being within a step of a capsule in general was reduced to just 5, the inverse of the -5 for being within a step of an enemy. If an agent was within one step of an enemy and one step of a capsule, they'd receive a reward of 10. However, this resulted in strongly biased capsule eating, the solution instead was to only reward the agent for being new a capsule when there was also an enemy ghost within one step of them.

The final capsule feature was implemented to improve the reward shaping. If there was a ghost one step away and a capsule one step away, this gave a +2 reward for that state. This enabled the

offensive agent to be motivated towards the capsule, but not overly skew the agent to always go for it at the start of the attack.

Keeping distance on attack

Agents on attack are promoted to keep distance from each other by at least two tiles. This is handled during training, where a reward of +1 is provided every time there are two attackers and they are distant from each other.

Feature	Description	Value
teamDistance	Describes the distance held between two attackers.	$\begin{cases} \frac{maze\_distance(a1, a2)}{width + height} & if\ nAttackers = 2 \\ 0 & else \end{cases}$
closest-capsule	Distance to the closest capsule if one exists.	$\min_{c \in capsules} maze\_distance(a, c)$

Table 1 - Additional offensive features

Description	Value
If there are two attackers and their distance is at least two apart	$+1$
If there's a ghost one step away and a capsule one step away	$+2$
If there's a ghost one step away and no capsule one step away	-5

Table 2 - Additional and altered rewards for offensive behaviour

Heuristic based defence strategy

Initially rewards were created for the defence approximate Q-learning agent. However, it is somewhat difficult to define what a good reward should be for a defending agent in the scope of the game state, especially when we wish to isolate the training to just the defensive agent. How do we reward being near enemy agents if we can't see them? One tactic is to have agents be rewarded for being near the map and near dense food clusters. However, the rewards are sparse, and in the end the agents would just cower in base, waiting for the opponents to just eat everything up and end their suffering. This can be overcome by only training weights once the agent is in the centre of the map, but it is still a sparse reward problem.

The solution instead was to build a hierarchical heuristic searcher, that would change goals based on new information in the game state. The below flow describes the defensive actions that can be taken by our low-level heuristic agent. This logic is inspired by Team Alpha's [3] BFS defence agent[3], which follows a similar workflow for their defensive agent.

[3] Adapted from Wang. Y, Fu. H, Liang. Y. (Team Alpha). Defensive agent governing strategy: https://github.com/infinityglow/COMP90054-Pacman-Contest-Project/wiki/AI-Method-3.md

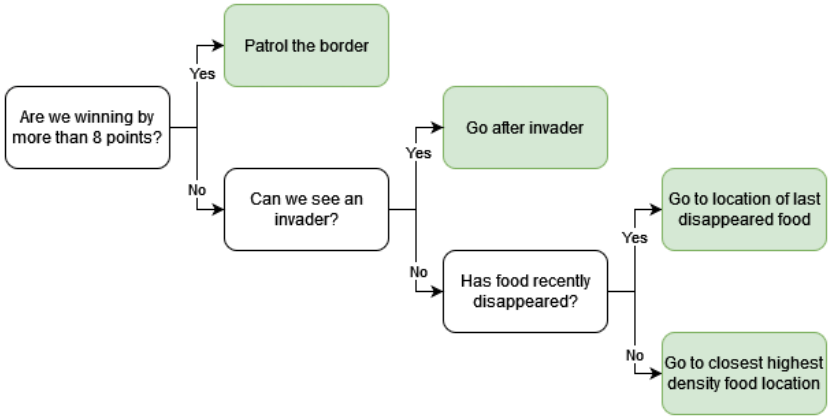

Figure 5 - Decision logic for heuristic defensive behaviour

This simple logic inspired by Team Alpha's [5] agent allows the defensive agent to behave exactly as expected at any point in time, though there are limitations which are discussed next, alongside improvements made.

Limitation: Fear is the mind killer

This workflow does not consider whether defensive agents are scared or not. As such, any Pacman that is powered by a capsule is considered the same as a regular invader, and as such the defensive agents will still target them, not knowing that they are now vulnerable.

This behaviour may be difficult to program using heuristic search, as the goal position changes every iteration with the invader likely chasing the defending agent. Therefore, scared defensive agents are handled using approximate Q-learning, with features specifically for staying close but avoiding the invader when scared. This is discussed specifically in the high-level actions section under the *Defence (Avoid)* action.

Limitation: Patrol pre-condition is strict

The patrol action is invoked whenever the team is winning by more than 8 or 10 points. However, in closer games, this can result in the patrol action never being taken where it otherwise would be a good idea. To overcome this, the pre-condition for patrolling is changed to account for the relative difference in foods remaining between the teams. In particular, the percentage difference relative to the friendly team is taken, where if the difference is greater than 65% and the agent's team has a positive score, patrolling is triggered in order to protect what's left. This 65% threshold was found simply through empirical evaluation as there is no real general way to pre-determine this without simulating large numbers of games. Additionally, if there is no food left on the opponent's side, there is no reason to continue fighting.

$$\frac{food_a - food_{opp}}{food_a} * 100 > 65 \ \cap score > 0 \cup food_{opp} = 0$$

Limitation: Food density can be misleading

This defensive strategy inspired by Team Alpha [4] has deliberately been implemented first to demonstrate a significant flaw: the area with the densest food is not necessarily the best place to defend depending on the position of the agent. Figure 6 demonstrates a situation where path finding the highest density food may not be adequate for maximal defensive coverage. Assuming the invader does not eat any dots along the way, the defensive agent has placed itself in an inappropriate location for defence, despite the food density saying it is a good location.

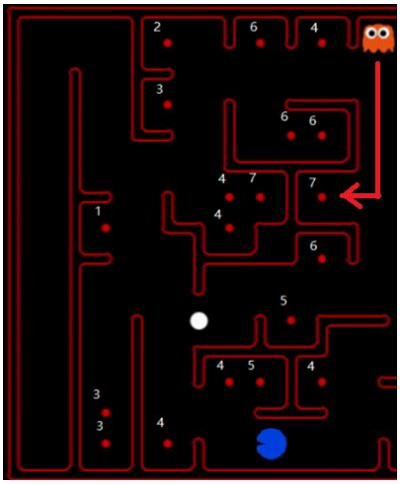

Figure 6 - Ghost agent goes to closest high density food dot, while invader can sneak around.

In more standard games, the baseline agent would eat that first dot in figure 6. This would result in the next closest dense dots being the two dots with a density of 6, which is in a dead end. These dots provided less strategic value again, and the enemy invader would be free to eat everything else up.

To further improve on this strategy, we would ideally like to be more informed in our positioning of the defensive agent. This brings us to the concept of maximal flow and the Ford-Fulkerson algorithm, which was experimented with but ultimately not used in the final submission as explained.

Determining bottleneck positions using the Ford-Fulkerson algorithm

The issue with the situation in figure 6 is that there is virtually no strategic value in being there. Ideally, the ghost would be better off blocking areas from the invader such that escape options are limited, and that cover the most amount of food.

Team pacamon [4] showed that utilising the Ford-Fulkerson algorithm [3], a greedy algorithm for computing the maximum flow in a flow network, the bottleneck positions for a given grid map can be found. They were able to achieve great performance with their agent by including this strategy.

	Algorithm 2: Ford-Fulkerson algorithm
1	**Input**: Network G = (V,E), flow capacity c, source node s, sink node t
2	**foreach** edge (u,v) in G
3	flow(u,v) = 0
4	**while** there exists path p from s to t in G such that capacity of edge $(u, v) > 0 \; \forall (u, v) \in p$
5	$c_f(p) = \min\{c_f(u, v) \mid \forall (u, v) \in p\}$
6	**foreach** $(u, v) \in p$
7	$flow(u, v) = flow(u, v) + c_f(p)$
8	$flow(v, u) = flow(v, u) - c_f(p)$

Algorithm 2 - Ford-Fulkerson algorithm for finding the maximum flow.

The above algorithm describes the pseudocode for finding the maximum flow in a generic flow graph. We initialise the algorithm with a network G, which has an equivalent residual network G_f with edges E_f and capacities c_f. These networks have some source node *s* and sink node t. The sink node describes the end point at which we want to measure the flow from. Line 4 states that while there exists a path (commonly found with BFS) from the source to the sink node such that the capacity of all edges is greater than 0, we find the minimum residual flow (line 5). Then for each edge in the path, we assign the flow of edge u to v as that flow plus the residual flow, and then assign the flow of the reverse edge v to u by subtracting the residual flow.

How this is utilised by the Pacman domain is that a flow graph is generated by artificially creating source and sink nodes at (-1,-1) and (-2,-2). Then all the possible start positions (usually the border positions), and the end positions are the locations of all the foods and capsules are attached to the source and the sink nodes, respectively. Then, all possible positions in the red or blue team's half of the map are added as vertices to the flow graph, with edges added between all the vertices where a legal action is possible, and flows being initialised to 1. The flow from the start positions to the source node, and the end positions to the sink node are set as infinite. The implementation of Ford-Fulkerson is adapted from Team pacaman [4], however it is not utilised in the final submission, though the adapted code is provided for reference in *maxflow.py*.

Map	Time to run (s)	Number of edges
jumboCapture.lay	0.663	329
defaultCapture.lay	0.559	133
officeCapture.lay	3.453	294
strategicCapture.lay	0.070	147
mediumCapture.lay	0.086	77
distantCapture.lay	0.522	165

Table 3 - Statistics for Ford-Fulkerson on different maps.

Table 1 demonstrates the algorithmic performance of the Ford-Fulkerson algorithm on select maps. In general, there is a range of runtimes that is governed by not just the number of edges but the maxflow as well. FFA is of the complexity of $O(maxflow * E)$. Using the default map and a starting position of (15, 8), there are 133 edges that are added to the flow network. This meant that it would not be feasible (or necessary) to recalculate every game iteration. These bottlenecks are therefore recalculated only when a) at the beginning of the match and b) when food has disappeared.

Figure 7 shows that when the Ford-Fulkerson algorithm is run, it finds the positions (6,10) and (8,13) as the positions that directly cover off the most amount of food when utilising the source node as the middle of the border (15,8). These cover off three food directly, and indirectly a large amount of the map.

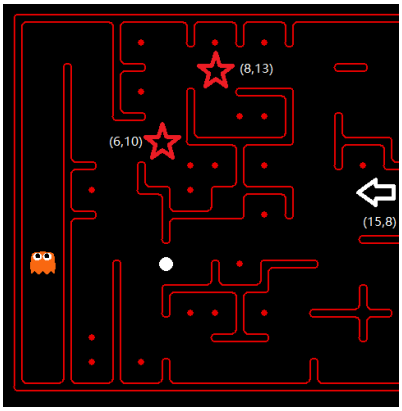

Figure 7 - Top 2 bottleneck positions identified that cover off the most food, using (15,8) as the source node.

It should be noted that there are other algorithms for finding the maximal flow through a flow network such as Edmonds-Karp [2] and Dinic's [1] algorithms which aim to improve the time complexity of the original Ford-Fulkerson algorithm. These alternatives were not implemented as there was not enough time, and this may be an area for future exploration.

Bottleneck positions may be implemented as part of the defensive agent's strategy, replacing maximal food density as the target for path finding. Below is the revised logic for the defensive behaviour used in some agents.

However, there were issues when it came to actually executing this strategy on contest server submissions due to crashing agents, further explained in the *Challenges* section. Thus unfortunately, reasoning over bottlenecks was not a feature in the final submission.

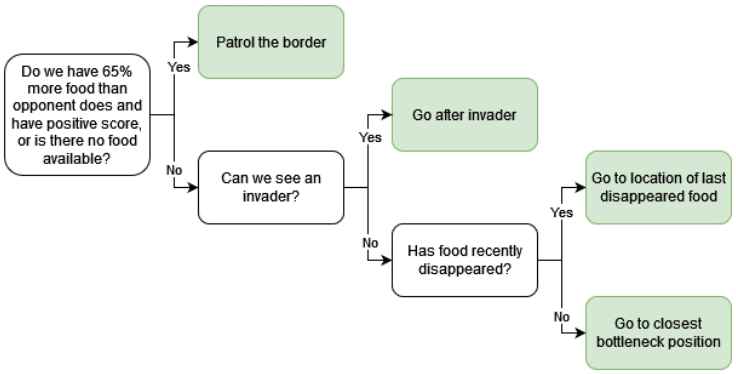

Figure 8 - Revised logic defensive behaviour using bottlenecks and patrol condition. Final submission used food densities instead of bottlenecks due to incompatibility with server execution.

High-level actions

In order to perform more complex actions, more high-level actions need to be included. The difficulty comes in striking a balance between having exact 1:1 mappings from high-low actions for every behaviour and having a generally solvable PDDL problem. The result was to define a range of high-level actions that describe some granularity of overall offensive and defensive behaviour, and then fully describe them in the low-level planning.

Attack

The same effect as the baseline is utilised, however the pre-conditions change such that attack is only in effect whenever the agent is in the enemy's territory. The attack action is overall improved as discussed in the previous section on improving the offensive Q-learning agent.

Rush

The rush action is taken whenever there are no invaders, the agent is not in opponent territory, or they are not in an end game state. This action describes the agent's team taking different attack routes through to the opponent's territory. This utilises heuristic search, with the goal location being split up into two cases:

1) If the distance to the closest border position is within three tiles, set goal as there.
2) Otherwise choose a random target position and path find there.

This covers a range of behaviours in our agent for the rush action. When spawning in, the agent is almost always far enough from the border to pick a random spot, which enables the agents to split up. If the agent has just returned from enemy territory (to eat food), the rush action means it will simply re-enter the same position.

Consume

A separate action to consume the backpack has been created as the pre-conditions for the attack action has changed. This action describes the agent returning to the closest border position using the baseline approximate Q-learning features for escape, as that seems to be sufficient.

Return

Similar to the consume action, however the pre-condition for this move is that there are no foods left available. This move was necessitated as there were instances when the game would have no food remaining, and attempt to find a high-level plan but the solver could not find one.

Defence (Passive)

The passive defence action is taken whenever there are invaders, but they are out of sight. This corresponds to the agent moving towards the bottleneck position using the Ford-Fulkerson algorithm that determines maximum flow.

Defence (Ping)

When food disappears from the defending side, this triggers a "ping" that lets the agent know that they can path find to the most recent disappeared food in hopes of catching an invader either along the way or nearby. If more food is eaten, then this is handled as a replan check.

Defence (Active)

The active defence action is taken when an invader is within sight. As part of this action, the agent replans every time step in order to continue moving towards the invader as they would also be moving.

Defence (Avoid)

The defence avoid action triggers when an agent is scared. The objective is to stay within sight of the hunting Pacman agent, but always try to be a few steps away. The corresponding low-level behaviour would trigger an approximate Q-learning action that is similar to the baseline defensive action, except a feature to capture if a scared ghost is close to a powered-up Pacman is used, with a huge -5000 manual weighting used to force the ghost to always be at least 3 distance from the Pacman but still in sight. This is simply an extension on the baseline defensive Q-learning behaviour, and only use the Q-learning action for this scared avoidance behaviour.

Patrol

The patrol action is extended to be a context dependent action, which can serve as the initially intentioned endgame action, or can allow an offensive-defensive agent to split. While initially included as part of the baseline implementation, it is expanded by having the agent patrol the border of their territory while waiting for invaders to approach.

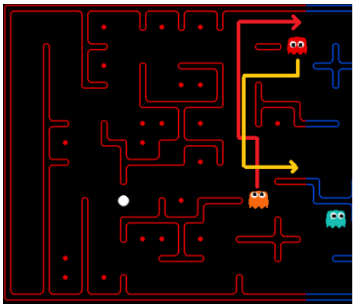

Figure 9 - Patrol action in effect having both agents patrolling the border to either a random position or the furthest border position.

Dive bomb

This is an action triggered specifically under specific end game circumstances. If the timer is less than 80 and the enemy has 50% more food than we do and we have a negative score, this triggers greedy offensive behaviour in the form of greedy heuristic search. Here we simply aim to make one last ditch effort to eat everything, even though it is very unlikely the dive-bombing agent will survive as enemy positions are no longer accounted for. This is represented as simply finding the shortest heuristic search path to the closest enemy food.

Limitations: Representing certain behaviours in PDDL

Some desirable behaviours are difficult to represent at a high level in PDDL for goals. For example, we have two defence actions: defence and defence_ping. The first action is triggered when there are invaders in our territory, the second action is triggered when food has gone missing. These high-level actions would map to different low-level behaviours such as regular defensive patrolling and specifically targeting the position food has disappeared from.

However, by nature of the high-low-level workflow, representing food disappearing as a predicate would only last for a single game state. If we pre-condition defence_ping to activate when food has disappeared, in the next iteration of the game the logic to check disappeared food would see there is no longer any food that has disappeared, as only the most recent two historical observations are used to check that logic, and that is recomputed at the beginning of goal selection.

High-level goals

In contrast to numerous additions of high-level actions, only a handful of goals were added as the fundamentals of the baseline goals were already deemed sufficient to handle the majority of the scenarios that might arise in game. The additions were made to complement certain desirable actions and behaviours as follows.

Goal	**Description**	**Pre-condition(s)**	**Positive goal(s)**	**Negative goal(s)**
goalDefWinning	Defend foods and patrol friendly area.	patrol_threshold (as described in the *Patrol* section)	defend_foods	-
goalScoring	Eat food on the opponent's side.		-	food_available is_pacman enemy1 is_pacman enemy2
goalRush	Get into enemy territory.	For any agent, ~in_enemy_territory	is_pacman	
goalTargetEater	Go towards disappearing food.	food_disappeared ~defence_active	-	food_disappeared is_pacman enemy1 is_pacman enemy2
goalConsumeBackpack	Eat backpack when you've got a substantial amount.	15_food_in_backpack		15_food_in_backpack

In particular, the target eater goal is included to specifically switch defensive behaviour whenever food disappears. Likewise, the goal rush is to differentiate between the *attack* and *rush* actions as they use Q-learning and heuristic search respectively. Consume backpack is included so that the

agent does not get too greedy and can return home when it has a substantial amount. Generally, 15 is a good amount.

Co-operative behaviour

Agents in the Pacman domain are initially independent. This means that when agents are instantiated by the game controller, they may follow the exact same policy if no knowledge of their teammates is given. Co-operative behaviour between agents thus needs to be explicitly handled through a few mechanisms:

1) Separate high-level PDDL actions for each member of the team. This maintains a 1-1 mapping between high- and low- level actions but can become too verbose and numerous.
2) High-level PDDL actions which utilise co-operative predicates. This reduces the total number of actions required but requires a more sophisticated high-level representation of the game.
3) Handling co-operative behaviour at the low-level. This reduces the complexity of the high-level plan but can result in less explainable AI agents.

The second option is likely the preferred way to handle co-operative behaviour through the lens of classical planning. However as explained in the challenges section, this was unable to be successfully implemented though not without trial and error. As such, options one and three were used for co-operative behaviour.

In order to handle co-operative behaviour in the agents, the choice to maintain a class variable for low-level goal nodes and to implicitly capture co-operative behaviour in the approximate Q-learning features was chosen for both offensive and defensive behaviour. Similar to the current agent class variable, another class variable dictionary contains the target location of each member of the team if they are being planned via heuristic search. This information can be used when planning between agents, such that if one agent moves to a given location, the next agent should probably go somewhere else depending on the context of the game state.

Co-operative defence

The defensive actions in this implementation rely on visibility of the enemy to a particular agent. As such, different agents can handle different invaders and scenarios depending on whether they are in view or not. For example, if one agent sees an invader, they can path find and go after them, while the other agent can continue to patrol or passively guard areas of interest. This enables the agents to perform general actions under the *defence* umbrella, but still co-operate differently depending on whether an agent sees another or not, rather than both agents reacting to the same enemy.

Utilising the global information of the current goal location for an agent, as defence is almost entirely a heuristic search problem, agents are able to be informed of where their teammates are heading to. This information is used to separately plan defenders where they may be in the same general location and thus the closest bottleneck or food density is the same for both.

Co-operative offense

When both agents are on attack, a feature capturing the distance between them is utilised to keep their distance at least two away from each other. This was such that both agents can be spaced away from each other.

The rush action maps to the low-level heuristic search planner that finds a random entry point for each agent when they are at least three tiles away from the closest border position. This implicitly

forces co-operation between the agents when rushing to the enemy side, as they do not follow each other but choose different entry points.

Experimental results

The final submitted agent was tested over a range of maps, with 50 games played each. The win rate, average score, and distribution of scores for each map is documented to demonstrate results of the agent. The agent was always the red agent. The *tinyCapture.lay* map was unable to be evaluated as the staff team baseline agent encountered errors on that particular map over multiple runs where no high-level plan could be found.

Map	Red win rate (%)	Average score
defaultCapture.lay	48%	1.74
alleyCapture.lay	52%	0.46
fastCapture.lay	66%	3.08
mediumCapture.lay	56%	4.50
jumboCapture.lay	78%	19.20
strategicCapture.lay	78%	11.50
crowdedCapture.lay	100%	35.24
bloxCapture.lay	58%	8.62
distantCapture.lay	58%	2.50
officeCapture.lay	94%	36.50

Table 4 - Summary results of agent against baseline.

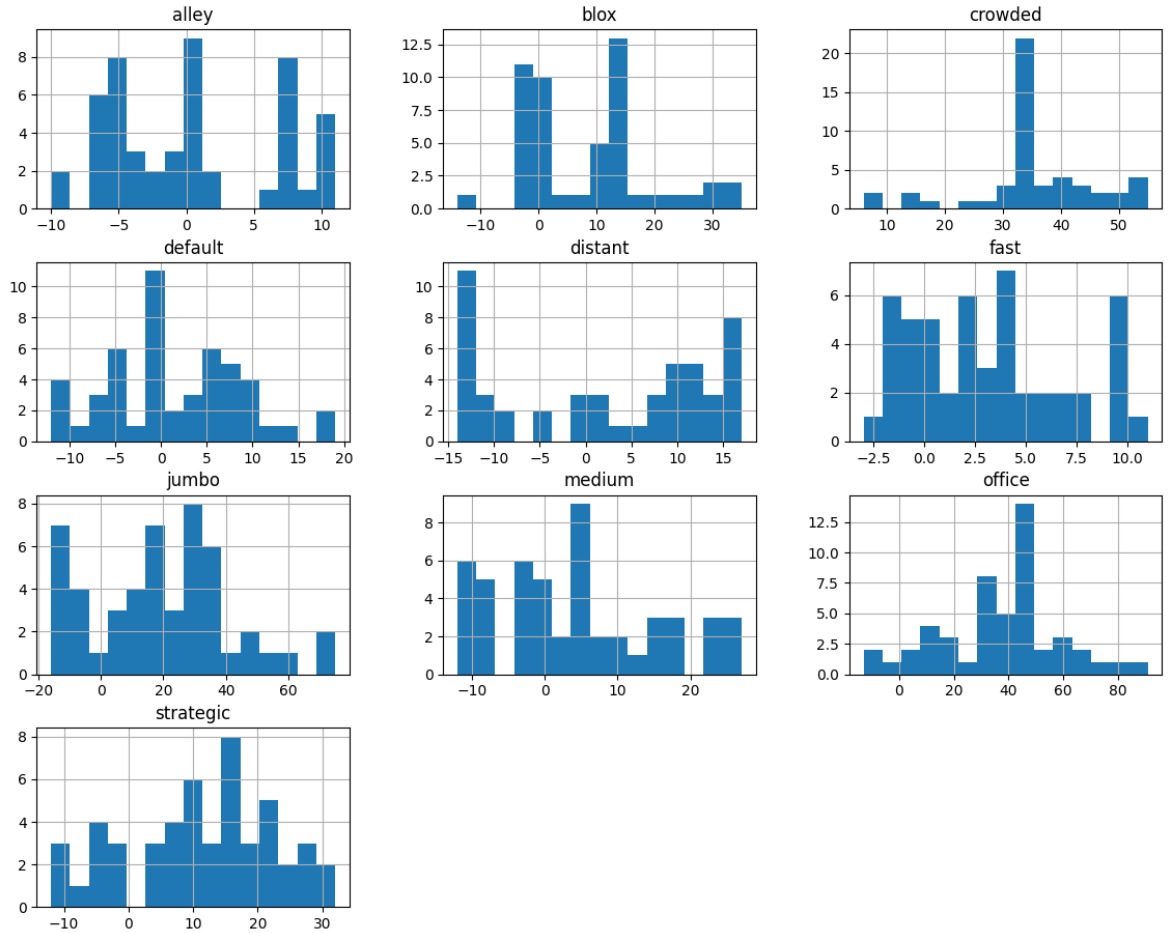

Figure 10 - Histogram of scores over 50 games on each map.

What is interesting to note is that the default capture map was the only map where the agent failed to win more than 50% of the time. By the standards of a convincing win (28 out of 49 games won i.e., 57%), the agent convincingly won over the baseline on 7 out of the 10 maps. The average score was generally quite low compared to the large amount of food available, and this could be explained by the fact that the patrol action meant that after a certain amount of game time the agents would simply guard their side instead of continue to be aggressive, artificially limiting the total score.

Exploiting the baseline defence strategy

In general, it seemed that on larger maps (such as jumbo, office, strategic), our agent was able to perform much better than the baseline. This would likely be due to how far away spawns are, exploiting the noisy distance of the defensive baseline agents to eat food while the defensive agents were stuck in loops in spawn. A good example is *officeCapture.lay* in the figure below, which features this sort of exploitative behaviour. This is a similar case with *jumboCapture.lay* though to a less extreme extent.

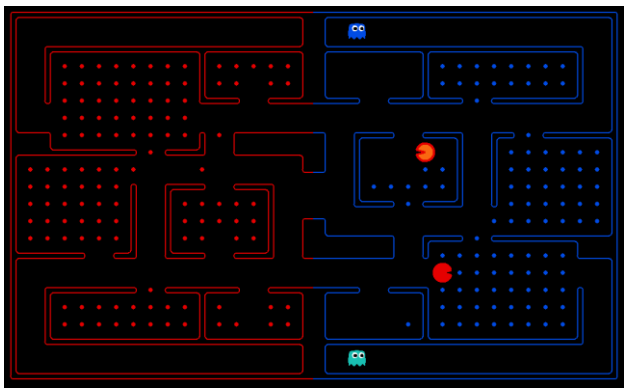

Figure 11 - Blue agents stuck in spawn in defensive loops while red Pacmen go to town.

Most interestingly of the results was the perfect score achieved on the *crowdedCapture.lay* map. This map begins with one agent on each side at the border. However, the baseline agent does not cross the border, instead the agent starting all the way in spawn is the one moving forward, while the baseline agent at the border is stuck in an infinite loop of moving left and right. This allows our agent to freely eat all the food at the top of the map and get back home before getting home safely and scoring easy points.

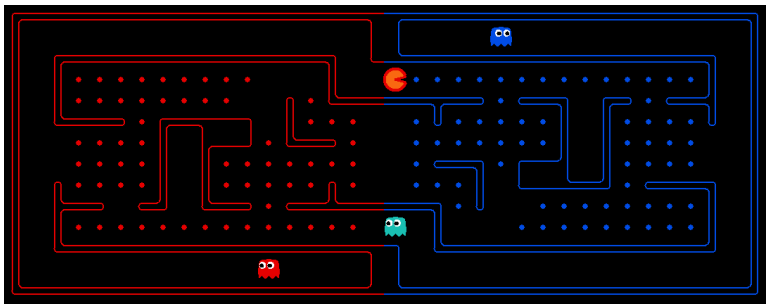

Figure 12 - Bottom blue ghost stuck in infinite loop, allowing red Pacman to move freely.

While this was able to dominate the baseline agent, this behaviour certainly cannot be replicated against the more sophisticated strategies of the other agents on the submission board, which is likely why this agent's strength against the baseline was also its weakness against the other teams.

Submissions

Several agents were submitted and tested, each with different combinations of techniques and strategies. This section lists out the notable submissions, which either feature convincing wins against the baseline or notable introductions of varying strategies.

Notable Agent 1: First convincing win

The first agent team to get a convincing win against the baseline and record victory points utilised a retrained baseline attack agent which considers enemy capsules, has a separate rush action, and utilises the heuristic search defence strategy, with the passive defence action being to path find to the highest food density area. The key attack strategy was that the agent would eat food until there were three in the backpack, and then return home. Interestingly, utilising a threshold of five foods in the backpack instead of three resulted in the baseline failing. The patrol threshold was invoked as simply having eaten more than 8 foods.

At the time of submission, this agent was able to secure 4.5 victory points, winning one match convincingly with three other matches resulting in ties.

Notable Agent 2: Bottlenecks and keeping distances

The next submitted agent replaced food densities with bottlenecks as the passive defensive action. The attack action had another feature added to measure the distance between the teammates if they were both on attack, rewarding agents during training if they kept their distance by at least two tiles. The patrol action was also modified so that only paths that stay within friendly territory could be taken. This agent somehow failed to find convincingly against the baseline. After some experimentation with different permutations, it seemed that in this particular case the patrol action that crossed into enemy territory actually seemed to perform better and win against the baseline, possibly because it would promote the agent to remain aggressive.

Unfortunately, this agent was unable to beat the baseline due to it crashing in the submission server repeatedly. The usage of bottlenecks was therefore abandoned.

Notable Agent 3: Improved pinging and modified patrol threshold

To get around disappearing food pings not persisting between game states, an instance variable is maintained called *disappearing_food_lock* which sets whenever food disappears via setter methods. This food lock is the position of the disappeared food and is used to maintain the predicate *food_disappeared* until an enemy is found or the agent reaches the ping, at which point it will switch back to passive defence.

The agents patrol threshold was modified so that it would patrol only when there were 65% more foods on its side than the opponent's side, or there is no food left on the opponent's side. This resulted in more selective patrolling behaviour, which may have been better for larger maps, where winning by more than 8 points is not necessarily a game winner or an end game state, and it would be better to still remain aggressive.

At the time of submission, the agent was able to secure 8.5 victory points, winning two matches and another five resulting in ties. This was the best performing agent that could be tested on the server before submissions time ended.

Challenges

There were some difficulties encountered which affected the breadth and scope of abilities of the final agent. They are listed out in this section.

High level co-operative actions

The workflow of high- and low-level actions facilitates a specific type of governing strategy for the agents, where it was relatively simple to describe which actions an agent should be able to take and given concrete goals, have the PDDL solver find the best set of high-level actions to take.

Implementing this behaviour in PDDL proved somewhat challenging, as when certain actions were given pre-conditions to account for the ally's action, a feasible solution could not be found. This in turn produced an unfortunate result where the agents at a high level could not interchangeably mix their behaviour as intended. The workaround solutions described above produce a form of co-operative behaviour either using separate actions at the high-level or in the low-level solver. However, this can produce inconsistent behaviour, and through the lens of explainable AI results in an agent whose behaviour can deviate from the mental model that the human may have of the AI agent.

Heuristic search defence strategy and food pings

The defensive strategy provided a concrete way to cover a number of defensive behaviours with relative control. However, the largest difficulty was handling food disappearing within the workflow of the high- and low-level plans. When a food disappeared, we would ideally like to lock our agent into the path towards the disappeared food. However, the next game instance would determine that this plan is not satisfiable, and revert the behaviour to defence passive, which resulted in agents not reacting to food disappearances. This was eventually resolved in agent 3 but was difficult to reconcile within the workflow.

Agent crashing on server using bottlenecks strategy

Unfortunately, it was found that the agent would crash on the server and fail the baseline when utilising strategies that used bottleneck calculations. This was an unfortunate circumstance as locally the agents were able to run successfully, while server runs would immediately fail the baseline within a second, and there was no clear way to understand how the agents were crashing. This meant that the bottleneck strategy had to be discarded as it could not produce the desired behaviour, leaving a more flawed food density passive defence strategy to replace it.

Conclusions

The submitted agents showed that they were able to convincingly win against the baseline agent across a large range of maps in local testing and on the submission server, utilising a mixture of heuristic search defensive capabilities and various auxiliary actions such as rushing and patrolling, and an enhanced approximate Q-learning strategy for offensive scoring behaviour.

Taking inspiration from various agents found online, it was found that more complex and interesting information can be derived from the game state and the domain such as discovering bottlenecks using the Ford-Fulkerson algorithm, utilising disappearing food as locational information and food densities for example. These pieces of information were utilised to further enhance the capabilities of the agent, though not all were able to translate well to the submission server, and as such strategies that involved reasoning over bottlenecks were abandoned.

The challenging aspects of the experimentation included wrangling with the high-level plan and attempting to utilise co-operative actions and predicates that would find solutions with given PDDL goals. This resulted in a large amount of time being spent attempting to integrate high-level co-operative planning within action pre-conditions and effects, which detracted from being able to experiment with other low-level strategies.

There are a large number of ways the agents can be improved. These range from improvements to calculating bottlenecks through implementing more efficient maximum flow algorithms, to an overall strategic overhaul of the agents' actions and behaviours. Being able to co-operatively plan agents using high-level co-operative predicates would have allowed for much deeper exploration of agent strategies that would have also allowed for the agents to be much more explainable. Adding predictive power through Monte-Carlo Tree Search, reasoning over dead-ends and corridors, and handling blind spots are some instances where the behaviours of the agents can become more sophisticated.

References

[1] Dinic, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". Soviet Mathematics - Doklady. Doklady. 11: 1277–1280

[2] Edmonds, Jack; Karp, Richard M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". Journal of the ACM. 19 (2): 248–264. doi:10.1145/321694.321699

[3] Ford, L. R.; Fulkerson, D. R. (1956). "Maximal flow through a network". Canadian Journal of Mathematics. 8: 399–404.

[4] Sharma. A, Winter. S, Khandelwal. S and Gupta. R (Team pacamon). MaxFlow Algorithm: Identifying bottlenecks in the map: https://abhinavcreed13.github.io/projects/ai-team-pacamon/#maxflow-algorithm-identifying-bottlenecks-in-the-map

[5] Wang. Y, Fu. H, Liang.Y. (Team Alpha). Project: https://github.com/infinityglow/COMP90054-Pacman-Contest-Project.