Navigation in Adversarial Environments Guided by PRA* and a Local RL Planner

Debraj Ray[1], **Nathan R. Sturtevant**[1, 2]

[1] Department of Computing Science, University of Alberta, Canada
[2] Alberta Machine Intelligence Institute (Amii)
debraj1@ualberta.ca, nathanst@ualberta.ca

Abstract

Real-time strategy games require players to respond to short-term challenges (micromanagement) and long-term objectives (macromanagement) simultaneously to win. However, many players excel at one of these skills but not both. This research is motivated by the question of whether the burden of micromanagement can be reduced on human players through delegation of responsibility to autonomous agents. In particular, this research proposes an adversarial navigation architecture that enables units to autonomously navigate through places densely populated with enemies by learning to micromanage itself. Our approach models the adversarial pathfinding problem as a Markov Decision Process (MDP) and trains an agent with reinforcement learning on this MDP. We observed that our approach resulted in the agent taking less damage from adversaries while travelling shorter paths, compared to previous approaches for adversarial navigation. Our approach is also efficient in memory use and computation time. Interestingly, the agent using the proposed approach also outperformed baseline approaches while navigating through environments that are significantly different from the training environments. Furthermore, when the game design is modified, the agent discovers effective alternate strategies considering the updated design without any changes in the learning framework. This property is particularly useful because in game development the design of a game is often updated iteratively.

Introduction

In real-time strategy (RTS) games, one common metric predictive of player performance is actions per minute (APM) (Avontuur, Spronck, and Van Zaanen 2013). This often means the player is micromanaging the behaviour of all the units under their control to get the best possible behavior. While expert players do not necessarily want games to reduce their burden of micromanagement, since it gives them an advantage, one could imagine many scenarios where players could find increased enjoyment when their units have better default behaviors. In particular, game units may want to trade off path-length optimality for other objectives. For example, offensive RTS units might collaborate to encircle the enemy, while defensive units may prefer safer escape routes (Ontañón et al. 2013). Other strategies for safe navigation include keeping distance from enemies, avoiding enemy vantage points and areas commonly patrolled by enemy scouts to avoid detection (Critch and Churchill 2020).

In previous research, Levy et al. (2020) used deep reinforcement learning to generate safe routes in urban environments while optimizing path lengths. However, this work learns features from the entire map, which does not generalize well to unseen maps or scale to maps of arbitrary sizes. Other works on adversarial navigation uses influence maps (Critch and Churchill 2020) and potential fields (Hagelbäck 2016). PRA* (Sturtevant and Buro 2005; Sturtevant et al. 2019) is an incremental pathfinding and navigation technique well-suited for dynamic environments, including hostile ones. In dynamic environments, computed paths get outdated quickly. Since PRA* works by cheaply computing abstract paths with only partial refinement of the real path, it can afford to replan repeatedly to avoid local enemies. These approaches however require manual tuning of parameters that determine the magnitude of agent's attraction and repulsion from objects in the environment.

With the broader goal of better micromanagement in mind, this paper studies the specific problem of adversarial pathfinding, proposing a reinforcement-learning (RL) based approach combined with PRA* for hierarchical pathfinding. Reinforcement learning (RL) is used for learning a policy to counter adversaries during local movement while that also optimizes path suboptimality. The agent then uses PRA* for long-distance pathfinding, considering only static obstacles, using local strategies learned through RL for reaching nearby goals. While PRA* has been shown to be a better alternative to A* for pathfinding in dynamic environments, RL reduces the burden of refinement on PRA* at the lowest levels of abstraction (that considers dynamic obstacles and adversaries), thus resulting in a mutually beneficial relationship. Furthermore, our use of environmental abstractions enables the agent to execute adversarial navigation effectively on maps that are significantly different from the maps used to train the agent. It is also observed that the proposed approach is generalizable such that the agent can learn different strategies when the game design is updated without requiring any code changes. This is a desirable property because game designing is usually an iterative process (Macklin and Sharp 2016).

Experimental results show that agents using the proposed approach take about four times less damage than baseline approaches based on either potential fields or PRA* alone, while not requiring significantly more computation resources (memory and CPU time). Furthermore, the proposed approach is shown to generalize to different kinds of environments and to scale to those of varying sizes.

Related Work

Real-time strategy games usually take place in dynamic and hostile environments where players have to make high-level and low-level decisions under extreme time constraints (Lara-Cabrera, Cotta, and Fernández-Leiva 2013). Hagelbäck (2016), proposed a hybrid navigation system for the RTS game StarCraft combining potential fields with A*. Their paper states that A* alone is not well-suited to dynamic worlds because the environment can change as the unit navigates, and the path computed by A* could quickly become obsolete. An alternate approach called PRA* (Sturtevant and Buro 2005) involves constructing hierarchical abstractions on top of a given map and then partial pathfinding through refinement. Unlike A*, PRA* is suited for dynamic environments because it interleaves partial path planning and path execution, which makes it more robust to changes in the environment. This is because the last stage of partial pathfinding does not need to happen until an agent is nearby, meaning that the local environment is known with higher certainty. Desipite this flexibility, paths generated by PRA* can still be within 1% of the optimal path length, depending on the parameters used in practice.

In the field of adversarial pathfinding, stealth games like Tom Clancy's Splinter Cell (Ubisoft, 2002), Mark of the Ninja (Microsoft Studios, 2012), and Metal Gear Solid V (Konami, 2015) encourage players to follow covert paths and utilize light and sound to avoid enemy detection during navigation. Mendonça, Bernardino, and Neto (2015), proposed a method to construct such covert paths in real-time using navigation meshes. Other researchers have attempted to strike a balance between path suboptimality and path safety in adversarial navigation. Jong et al. (2015) used influence maps (IMs) to compute a safety score for each grid cell of the map and then used it in the heuristic function of A* to find safe paths. The paper observed that the search space of A* increases significantly due to this additional dimension. The paper then used PRA* to improve search performance. A similar idea based on influence maps and flocking was implemented by Danielsiek et al. (2008) for group navigation of units in the RTS game Glest. However, influence maps are computationally expensive without parallelization, since they must be updated frequently when the position of units in the world changes (Mark 2019). With parallelization, the implementation complexity of the algorithm increases considerably.

The game of capture-the-flag (CTF) is interesting in the context of adversarial pathfinding as it requires complex strategic navigation. Players need to defend their own flag while trying to steal their opponent's flag. Huang et al. (2011) calculated the winning regions of both the attacker and the defender using the Hamilton-Jacobi reachability. Like CTF, the game of cops and robbers requires adversarial navigation. Moldenhauer and Sturtevant (2009) explored different algorithms for the robber to evade capture by the cop for as long as possible. Besides video games, tactical adversarial navigation has applications in the real world too. For example, a wildlife security group (PAWS) optimized its use of human resources for patrolling large conservation areas while effectively combating poaching (Fang et al. 2016).

Wang et al. (2020) proposed a framework to execute navigation of mobile-robots in a dynamic environment. The method uses reinforcement learning to avoid collisions with static and dynamic obstacles while following a global guidance path computed using the A* algorithm. Our work extends these ideas to adversarial environments and explores PRA* for global guidance.

Defining Adversarial Navigation

The adversarial pathfinding problem is defined on a graph $G = (V, E)$ where V is the set of nodes and E is the set of valid edges between the nodes. Edges are represented as $E = \{(v, v') : (v \ \& \ v' \in V)\}$. Each edge has an associated positive cost represented by $c(v, v') \in \mathbb{R}^+$. A path ν is a sequence of nodes $\nu = v_0, v_1, v_2, v_3 \ldots v_k$ such that $(v_i, v_{i+1}) \in E$ for every $i \in 0, 1, 2, \ldots k-1$. The i-th node on the path is represented using ν^i. The cost of traversing a path is calculated by summing all the edges in ν:

$$cost(\nu) \ = \ \sum_{i=0}^{k-1} c(\nu^i, \nu^{i+1})$$

An enemy (ξ) is defined as a unit whose objective is to inflict damage on the agent. The adversarial path planning problem is modelled by considering enemies as part of the environment. Enemies have a defined set of actions, which in this research are: move to an adjacent vertex, stay at the same location, and attack the agent. An enemy ξ_i can cause damage of magnitude $\hat{A}_i$ to the agent. Graph G is an adversarial environment with N enemies located at vertices: $\hat{v_1}, \hat{v_2}, \hat{v_3}, \ldots \hat{v_N}$. The behaviour policy of ξ is not provided explicitly, but is assumed to be static.

The agent starts with an initial health H at location v_s. The agent's destination is v_d. The agent has a legal set of actions, which in this research are moves to adjacent vertices. If ξ_i is successful in its attack on the agent, then the health of the agent after the attack is: $H - \hat{A}_i$. The agent is killed if its health falls to 0 or below. The input to the adversarial navigation problem can be defined with the tuple $\langle G, c, \xi, \hat{A}, H, v_s, v_d \rangle$. The objective when training the agent is for it to reach the destination by simultaneously reducing the distance travelled and the damage it takes from the enemies. A trade-off occurs because reducing distance requires the agent to follow shortest distance paths which may cause frequent enemy encounters and higher damage. Reducing enemy encounters could require the agent to plan longer paths that avoid areas with many enemies.

In our work, G is extracted from 8-connected (octile) grid maps, but the approach generalizes to other pathfinding architectures.

Our MDP for Adversarial Navigation

We design a Markov decision process (MDP) that represents the general adversarial navigation problem. In our game design, the world is deterministic and consists of fixed obstacles and dynamic enemies. The agent can only observe a limited area of the map around it, called the agent's field of view (FOV) (Figure 1). The agent's FOV is much smaller than the area of the entire map. This means that changes in the map are not completely predictable when the agent moves, as new obstacles come into the FOV. Similarly, enemies have their own strategies for movement and may appear or disappear from the FOV. As a result, the agent's view of the dynamics of the game are modeled by an MDP.

States in the MDP are defined by the agent's field of view. The image of the FOV defines half of the MDP state. An additional set of features are computed using information from the agent's FOV as the other half of the MDP state. These derived features contain the locations of nearby enemies, and will be discussed in the Q-Network architecture section.

In this work our enemies are identical and have fixed strategies for chasing and attacking the agent. Similar to the agent, the enemies have visibility of the portion of the map that is within their FOV. An enemy will chase and attack the agent if the agent is in their FOV. To execute a chase, the enemies predict the agent's next location by tracking the agent's previous location, and planning a path towards the agent's predicted next location. Enemies attack the agent by collision. Further details of the game design are provided in the section on experimental setup.

Actions in the MDP are defined relative to the direction of the global PRA* path at its point closest to the agent. The legal actions include going straight and turning 45 or 90 degrees in either direction relative to the global path.

The agent is trained on the MDP using reinforcement learning, so the learned policy depends on the rewards in the MDP. We have two objectives for agents in the game: (1) they should take short paths from the source to the destination and (2) they should avoid taking damage from enemies on the way. The rewards in the MDP are used to balance these objectives during learning.

Encoding Objectives in the Reward Function

The states of the MDP are split into two sets with two different reward structures. The first set of rewards applies when the agent is safe with no enemies in the vicinity, and the second set when the agent is unsafe due to presence of enemies.

An agent's state at time t is represented using λ_t^U if the agent is in an unsafe state and λ_t^S if the agent's state is safe. In safe states the rewards are designed such that the agent learns to follow the global path. If the agent is at node ν^i on the global path then any action that moves the agent to node $\nu^j : j > i$ receives a positive reward *RewardAdvance* (all reward values are provided in Table 1). Safe states on the global path are terminal states of the MDP. That is, an episode ends when the agent reaches a location on the global path that is ahead of its previous location on the global path. Thus the agent's trajectory during gameplay from v_s to v_d consists of, from a learning perspective, many, potentially short, episodes. This simplifies the learning process and allows agents to start learning quickly.

Other safe states that are not on the global path, but have the global path in the agent's field of view, are non-terminal states of the MDP. If an agent's action transitions the agent to a non-terminal state at unit distance from the global path, the agent receives a reward *RewardNonTerminal_1_Safe*. Similarly, if the agent's action results in transition to a non-terminal state located 2 units from the global path, the reward is *RewardNonTerminal_2_Safe*. In the same manner, we have defined rewards (*RewardNonTerminal_3_Safe*) for actions that transition the agent to states that are at 3 or more units away from the global.

Safe states that do not have the global path in the agent's field of view are terminal states and represent the case that the agent is unrecoverable or lost. An action that transitions the agent to a lost state is rewarded negatively with *Reward-Lost*. This is also a terminal state. The agent starts a new episode by resuming at the same location with a new global path from PRA*.

In unsafe states the rewards are designed such that the agent learns to reduce damage and reach a safe state. Consider three possible sequences of states (ψ_1, ψ_2, and ψ_3) the agent may encounter while moving from v_s to v_d. The hyphens in the sequences are used indicate when separate episodes begin. A safe state on the global path is a terminal state and therefore the agent's trajectory can contain many episodes.

$$\psi_1 = \lambda_0^S - \lambda_1^S\lambda_2^S\lambda_3^U\lambda_4^U\lambda_5^U\lambda_6^U\lambda_7^S - \lambda_8^S$$

$$\psi_2 = \lambda_0^S - \lambda_1^S\lambda_2^S\lambda_3^U\lambda_4^U\lambda_5^S - \lambda_6^S - \lambda_7^S\lambda_8^S$$

$$\psi_3 = \lambda_0^S\lambda_1^S - \lambda_2^S\lambda_3^S\lambda_4^S - \lambda_5^S\lambda_6^S\lambda_7^S\lambda_8^S$$

The adversarial navigation trajectory represented by ψ_3 is simpler because it only requires making progress on the global path through safe states. Since ψ_1 has more unsafe states than ψ_2, we consider ψ_1 to be further from the solution to the problem of adversarial navigation than ψ_2. Therefore we designed our rewards such that the agent learns to avoid damage in the unsafe states while looking to move to a safe state on the global path. This reduces both path cost and damage from enemies. In practice, once the agent has learnt to navigate within safe states, it will learn to navigate unsafe states with strategies that neutralize enemies the fastest. For example, in a certain unsafe state, the agent can decide whether an escape strategy (defensive) is better than an attacking (offensive) strategy based on which strategy can defeat enemies more quickly.

We hypothesize ($\mathcal{H}$) that the design of our rewards will be able to solve the problem of adversarial navigation. The hypothesis $\mathcal{H}$ is validated if the proposed approach results in reduced damage and shorter paths than the baseline approaches. We will validate the hypothesis empirically through experiments.

The agent is provided a negative reward *RewardNonTerminalUnsafe* when its current state is an unsafe state, and its next action leads to another unsafe state. If an agent-action results in damage to the agent due to an attack from

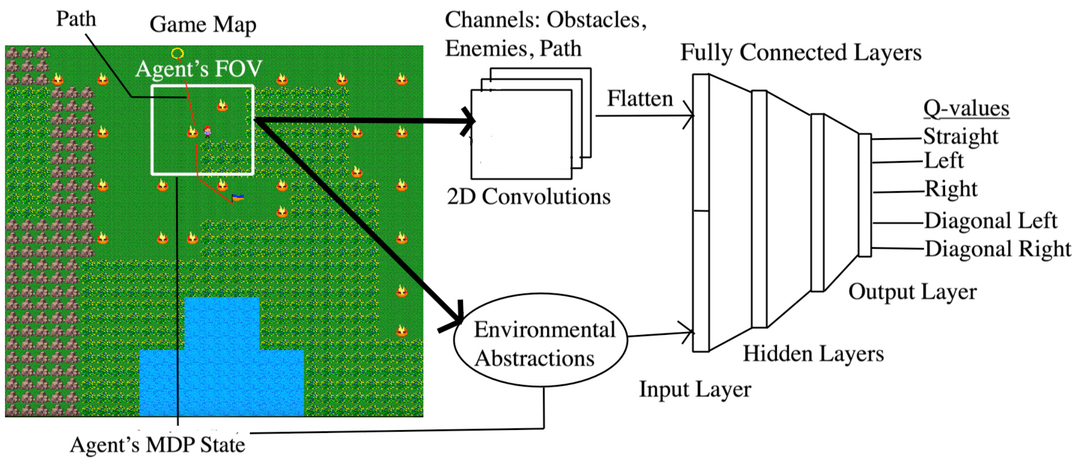

Figure 1: The neural network for learning navigation in adversarial environments

a nearby enemy, then it receives a large negative reward *RewardInjury*. Furthermore, an agent action that transitions the agent to a lost state also receives a negative reward *RewardLost*. Both agent states where the agent is either damaged or lost are terminal states. The agent starts a new episode by resuming at the same location where the previous episode terminated. The learning follows the reward structure of safe state to safe state transitions when the agent's action transitions it from an unsafe to a safe state.

Q-Network Architecture

This section describes the architecture of the learning framework, including the inputs and outputs of each module (Figure 1).

Features are extracted from the agent's field of view (FOV) through a combination of 2D convolution layers and environmental abstractions. Environmental abstractions are computed features that generalize well across different environments. For example, the agent scans outward to detect the position of obstacles around it, abstracting actual shapes and sizes of the obstacles. Furthermore, instead of tracking the entire section of the path inside the FOV, the agent abstracts the path by finding the nearest point on the path and the direction of the path at that point. The agent evaluates linear and angular distances to enemies in the FOV relative to its current direction of motion. These measures are invariant to FOV rotations which happen when the agent's direction changes as it navigates. Based on the linear and angular distances to an enemy, a risk score is assigned to it. Risk scores are higher for enemies located close to the agent or having a forward placement, while risk scores are lower for enemies further away or placed in the rear of the agent. The enemies are then sorted by their risk scores. This helps the agent prioritize threats and also reduces variance in the training data. Environmental abstractions are designed to reduce the size and variance of the feature space through extraction and processing of key information while abstracting unnecessary details. It also helps generalize the learning to unseen environments that are significantly different from the training environment.

Besides environmental abstractions, we observed that adding a single 2D convolution layer improves model accuracy without compromising on generalizability or speed of training. The dimensions of the agent's FOV is 9×9. The CNN has 3 input channels that capture the position of obstacles, enemies, and the global path in the FOV. The kernel size is 3 and there are 16 output channels of the CNN. The output of the CNN is then passed through a $ReLU$ activation function before being flattened into a one dimensional tensor.

The agent computes a number of abstract features through the environmental abstraction module and populates a single dimensional tensor with those features. The two tensors are concatenated into a tensor of size 834, which represents the agent's complete state tensor (Figure 1). The state tensor forms the input layer of a fully-connected neural network with 2 hidden layers and 1 output layer. The first hidden layer contains 520 neurons and the second hidden layer contains 400 neurons. The output of the hidden layers pass through a $ReLU$ activation before entering the next layer. The output layer contains the same number of neurons as the agent's action space. The agent's action space consists of 5 actions determined relative to the agent's current direction: move straight, move left, move right, move diagonal left, move diagonal right.

Given an agent state, the output layer produces Q-values associated with every action from that state. The neural network (also called the Q-network and represented with $Q(s,a;\theta)$) is trained by minimizing a sequence of loss functions $\delta_i(\theta_i)$ that are updated at every iteration i of the training (Mnih et al. 2013).

$$\delta_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(.)}[(y_i - Q(s,a;\theta_i))^2]$$

where,

$$y_i = \mathbb{E}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}|s,a)]$$

The target for the next iteration is represented as y_i. $\rho(s,a)$ is the behavior distribution that follows an ϵ-greedy strategy - choosing the action having highest Q-value with

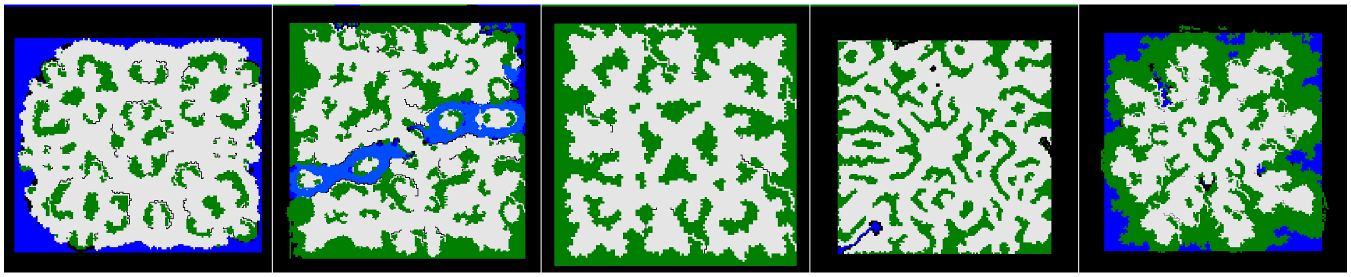

Figure 2: Warcraft III maps used to test the agent. From left to right: blastlands, divideandconquer, duskwood, gardenofwar, and thecrucible

a probability of $1 - \epsilon$ and a random action otherwise. The value of ϵ is initially set to 1 to encourage exploration over exploitation and then decayed gradually until the value of 0.25 is reached after which no more decay is allowed. The loss function is optimized with stochastic gradient descent using the Adam optimizer (Kingma and Ba 2017) and gradient clipping. A replay buffer of capacity 15000 is used to store past experiences, from which a batch of 4000 is randomly sampled in every epoch of training. Training is run for 300,000 epochs.

Experimental Setup

This section verifies the claim that adversarial navigation can be solved approximately with the proposed MDP design. Furthermore, we test whether the proposed approach can generalize to different types of game-environments and scale to large game-maps. A comparison with baseline approaches is provided in terms of the amount of damage the agent takes, path length suboptimality, the amount of memory used, and execution time across different path lengths.

The first experiment involves the design of an adversarial game. The objective of the agent, represented as the boy at the center of the map in Figure 3, is to navigate from the start position (marked with a circle) to the goal position (marked with a flag). The map is populated with enemies - the fire monsters. If the agent is in the field of view (of size 7×7) of the fire monsters, they will chase the agent. A collision with a monster causes 10 points of damage to the agent. Each fire monster is territorial and stays within a radius of 6 units from its starting location. All other objects, such as rocks, water, trees, etc., are considered obstacles in this experiment. The fire monsters have one weakness: If two or more monsters collide with each other, they are both destroyed and removed from the game. The agent is unaware of this weakness. However, we expect the agent to eventually discover this enemy weakness and learn to exploit it.

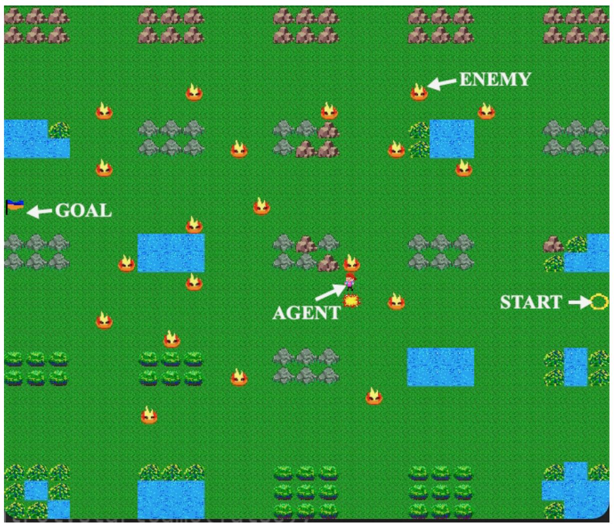

Figure 3: A generated map used to train the agent

The agent is trained on 10 maps of size 27×27 with obstacles placed uniformly as shown in Figure 3. These maps differ in the initial position of the fire monsters, which are calculated by random selection from a uniform distribution. At each iteration of training, the agent's start and goal location are decided randomly. The agent's trajectory from the start to the goal is split into many training episodes. An episode ends if the agent can reach a location on the global path that is safe and ahead of agent's last location on the global path. Additionally, an episode ends when the agent is killed by an enemy or has lost the global path from its FOV. When an episode ends, the agent starts a new episode by resuming at the location the previous episode ended. This process continues until the agent reaches the destination or a timeout occurs. A timeout will occur if the agent has travelled 5 times the optimal path length and yet not reached the destination. If the agent has reached the destination or a timeout occurred, the agent will start a new episode of adversarial navigation with new start and goal location on the same game map or a different one. The agent is trained with reinforcement learning using the reward structure shown in Table 1.

The agent is tested on five randomly selected maps of the RTS game Warcraft III, developed and published by Blizzard Entertainment. These maps are shown in Figure 2. The test maps are scaled versions of the original maps and have dimensions 512×512 (Sturtevant 2012). It is important to note that none of the test maps have been used for training. Moreover, the test maps have significantly different features compared to the training maps. The test data is created by generating 2000 random paths per test map, evenly binned by path lengths with a bin size of 50.

State	Name	Value
Safe	RewardAdvance	15.0
Safe	RewardNonTerminal_1_Safe	−1.0
Safe	RewardNonTerminal_2_Safe	−2.0
Safe	RewardNonTerminal_3_Safe	−3.5
Unsafe	RewardInjury	−45.0
Unsafe	RewardNonTerminal_Unsafe	−3.5
Both	RewardLost	−45.0

Table 1: Rewards used to train the agent

There are two baseline agents for comparison. The first one is based on PRA* - it computes an abstract path from the source to the destination and then refines up to 8 abstract nodes in the real world. When pathfinding in the real world the agent considers enemies as virtual obstacles. It is possible to prevent this baseline agent from going close to the enemies by using virtual obstacles larger than the enemies. The second baseline is based on static potential fields. The goal is given a low potential, and the enemies and obstacles are assigned high potential values. The agent moves in the direction of the steepest drop in potential.

This experiment compares the proposed approach against the baseline approaches on the following measures: 1) Amount of damage the agent takes, 2) Path length suboptimality, 3) Execution time of episodes, and 4) Maximum memory used for pathfinding. For the purpose of measuring the amount of damage and comparing the approaches, the agent is given infinite life. At the end of an episode of adversarial navigation, the total damage incurred by the agent is recorded. Path length sub-optimality is calculated as $\frac{ActualPathLength}{OptimalPathLength}$. Since PRA* is based on the A* (Hart, Nilsson, and Raphael 1968) algorithm, it requires an open and a closed list for path finding. The maximum size of the open list is a measure of the maximum memory used. The execution time represents the total time to complete an episode in milliseconds.

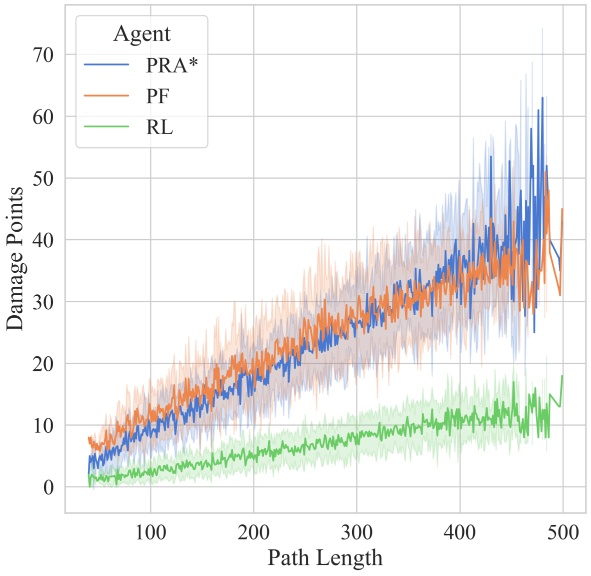

Figure 4: Amount of damage the agents take, plotted against the optimal path length

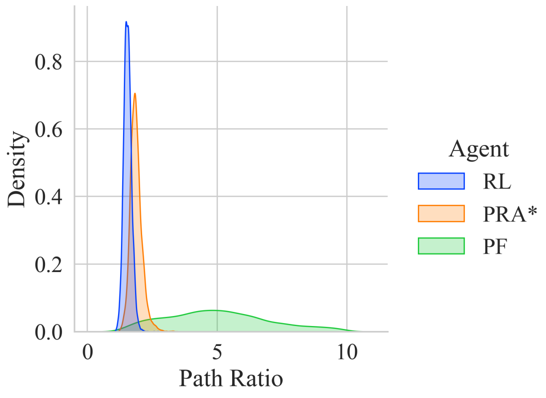

Figure 5: Path length suboptimality for different types of agents.

A second experiment updated the design of the game. The vulnerability of the fire monsters is eliminated. Now, the fire monsters fuse on collision to become a single stronger enemy with greater attack points and starting location at the fusion site. In this experiment, we observed that the agent based on the proposed approach was able to learn new strategies different from the previously learnt strategies, without any change to the agent's code or learning framework. The new strategies worked well on the updated game design. This means our proposed approach can generalize across game designs and types of adversaries. Unfortunately, due to limited space in this paper, we are excluding the results of the second experiment. However, the results are similar to the first experiment (in relative trends), with the key difference being in the magnitude of values (damage, path length suboptimality, memory use, and execution time). The values are larger in the second experiment than the first because of stronger enemies in the second. Full results are available elsewhere (Ray 2023).

Experimental Results

We describe our observations from the first experiment in this section and compare different approaches - the proposed approach and the two baseline approaches. For convenience, we call the agent using the proposed approach: RL agent; the baseline agent using PRA*: PRA* agent; and the baseline agent using potential field: PF agent.

The baseline approaches require hand-tuning of parameters for the different test maps used in the experiment. This process is often tedious and challenging. In the case of PRA*

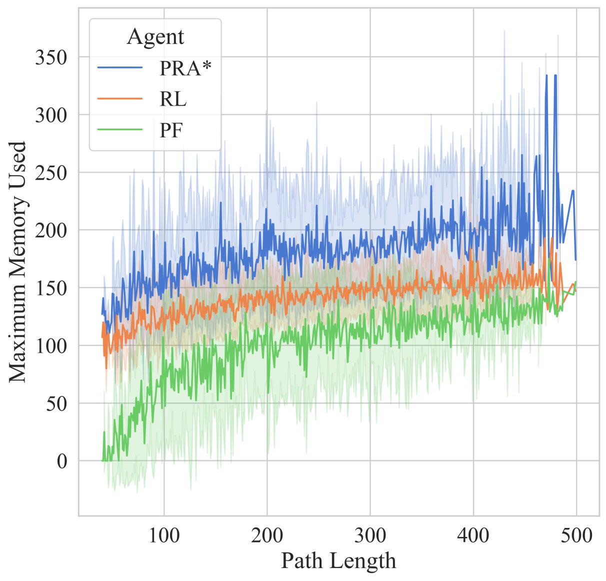

Figure 6: The maximum memory used for path finding (measured by the size of the open list), plotted against the optimal path length.

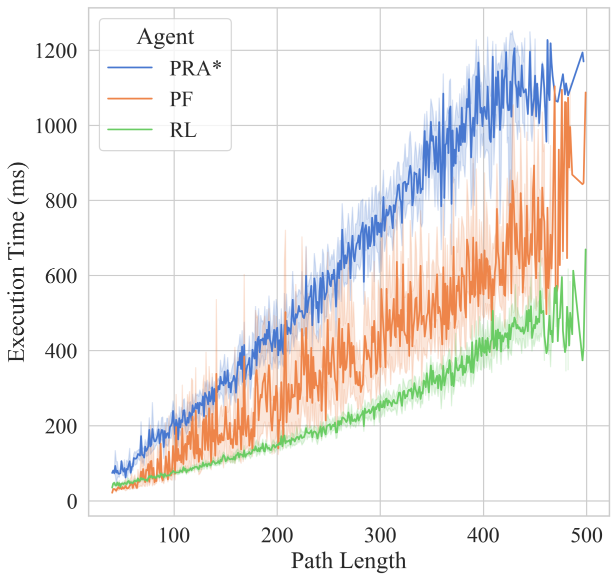

Figure 7: The amount of time in milliseconds that the agents take to reach the destination, plotted against the optimal path length.

baseline, there is a need to set the size of the virtual obstacles. For the potential field based baseline the attraction and repulsion values need to be tuned. We plotted all the figures in this section with our best possible configuration for PRA* and potential fields.

Since both the baseline agents have a similar evasion strategy, which is to stay away from the enemies, they accrue similar damage points for similar path lengths (Figure 4). However, the proposed approach (RL) takes significantly lesser damage, which implies that the learnt strategies are more effective than the simple enemy evasion strategy the baseline agents use. For example, the enemy evasion strategy is ineffective when the agent is encircled by adversaries. We observed that the RL agent is less vulnerable to encirclement as its strategies more aggressively exploit enemy weak points, allowing little time for enemy build-up around it.

The length of the actual path navigated by the RL agent and the PRA* agent are often closer to the optimal path than the PF agent (Figure 5). This is because both the approaches explicitly minimize path cost. However, the PF agent is not penalized in any way for taking longer paths. The RL-agent is less sub-optimal in path length than the PRA* agent because its strategies exploit enemy vulnerabilities while staying close to the optimal path, thereby minimizing path length suboptimality. This and the previous result affirms our hypothesis ($\mathcal{H}$).

Both the RL agent and the PRA* agent use additional memory for pathfinding. The PRA* paper defines a constrained A* search which significantly reduces the search space. This results in lower memory footprint and execution time compared to a regular A* search. However, we observed that in a dynamic environment refining narrow corridors as part of the constrained A* search can be unsuccessful when enemies accumulate in such corridors blocking the path completely. One solution to this problem is to expand the size of the corridors. However, since this amounts to relaxing the constraint of the constrained A* search it will increase memory use and execution time compared to the unrelaxed constrained A* search. In our experiment, we dropped constrained A* search from the PRA* baseline to resolve the problem of unsuccessful refinement due to enemy congestion. This resulted in higher memory usage of the PRA* agent (Figure 6). The RL agent does not suffer from the enemy-congestion problem despite using PRA* because it does not consider enemies (and dynamic obstacles) during the pathfinding step. It tackles enemies and dynamic obstacles locally with strategies during navigation. Therefore, RL and PRA* together in the proposed approach is a symbiosis that results in effective navigation in hostile environments.

The potential field approach does not use any additional memory except when it is stuck in local optima (Hagelbäck and Johansson 2021). We implemented repulsive trails to minimize the problem but still needed PRA* to recover the agent in the few instances the agent got stuck. Thus, we see the PF-agent using some additional memory in Figure 6.

The execution time of PRA* is the highest (Figure 7) because: 1) its heuristic function does not account for environment dynamics and is therefore less accurate, 2) it does not benefit from the constrained A* search due to the problem of enemy congestion. Note, PRA*-agent can benefit from constrained-A* search partially if the size of the corridors are expanded. The RL-agent fully benefits from the constrained A* search and is therefore significantly faster in computing paths than the PRA*-agent. Moreover, in the case of the RL agent, inference from neural networks is a constant time operation with respect to the path length. However, the PRA* agent repeatedly searches for new paths to avoid adversaries, which has an exponential worst case time complexity in path length (Korf and Reid 1998). Therefore we find execution time growing faster for the PRA* agent compared to RL agent as the path length increases.

The Time complexity of the potential field calculations is linear in the branching factor of the map. It involves checking the potential values of the grid cells adjacent to the agent's current cell. Since in our octile grid maps the branching factor is constant, we can treat potential field calculation as a constant time operation in path length. Therefore, the execution time growth rate of the PF agent is slower than the PRA* agent. However, the PF agent is significantly more suboptimal in path length compared to the RL agent and therefore requires a longer time to reach the destination.

Conclusion

The topic of adversarial navigation presents multiple objectives that must be simultaneously optimized to achieve the desired result. We showed that reinforcement learning combined with PRA* can reduce path length and damage from adversaries simultaneously, without compromising memory or CPU time. We trained the agent on custom maps of size 27×27 and showed that the agent can outperform baseline approaches on maps of size 512×512 of the RTS game Warcraft III. This demonstrates our approach's generalizability, and ability to scale to large maps. In a second experiment, we updated the game dynamics to answer the question whether our approach can generalize across game designs. This property is useful since game designing is usually an iterative process. We observed that the agent could learn new strategies which enabled it to outperform the baseline agents in the updated game.

Future research in this field can explore how the proposed approach can be extended to an environment containing enemies having dynamic strategies to attack the agent. Another interesting research direction is cooperative adversarial navigation where some units are allied while others are adversarial. There is also a need to study the impact of this research on actual game play experience of human players. Shortcomings observed in the study can then be addressed in future works.

Acknowledgments

This work was funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Avontuur, T.; Spronck, P.; and Van Zaanen, M. 2013. Player skill modeling in Starcraft II. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 9, 2–8.

Critch, L.; and Churchill, D. 2020. Combining Influence Maps with Heuristic Search for Executing Sneak-Attacks in RTS Games. In *2020 IEEE Conference on Games (CoG)*, 740–743.

Danielsiek, H.; Stuer, R.; Thom, A.; Beume, N.; Naujoks, B.; and Preuss, M. 2008. Intelligent moving of groups in real-time strategy games. In *2008 IEEE Symposium On Computational Intelligence and Games*, 71–78.

Fang, F.; Nguyen, T.; Pickles, R.; Lam, W.; Clements, G.; An, B.; Singh, A.; Tambe, M.; and Lemieux, A. 2016. Deploying PAWS: Field Optimization of the Protection Assistant for Wildlife Security. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(2): 3966–3973.

Hagelbäck, J. 2016. Hybrid Pathfinding in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(4): 319–324.

Hagelbäck, J.; and Johansson, S. 2021. The Rise of Potential Fields in Real Time Strategy Bots. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 4(1): 42–47.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Huang, H.; Ding, J.; Zhang, W.; and Tomlin, C. J. 2011. A differential game approach to planning in adversarial scenarios: A case study on capture-the-flag. In *2011 IEEE International Conference on Robotics and Automation*, 1451–1456.

Jong, D.; Kwon, I.; Goo, D.; and Lee, D. 2015. Safe Pathfinding Using Abstract Hierarchical Graph and Influence Map. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, 860–865.

Kingma, D. P.; and Ba, J. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980.

Korf, R. E.; and Reid, M. 1998. Complexity analysis of admissible heuristic search. In *AAAI/IAAI*, 305–310.

Lara-Cabrera, R.; Cotta, C.; and Fernández-Leiva, A. J. 2013. A review of computational intelligence in RTS games. In *2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI)*, 114–121.

Levy, S.; Xiong, W.; Belding, E.; and Wang, W. Y. 2020. SafeRoute: Learning to Navigate Streets Safely in an Urban Environment. *ACM Trans. Intell. Syst. Technol.*, 11(6).

Macklin, C.; and Sharp, J. 2016. *Games, Design and Play: A detailed approach to iterative game design*. Addison-Wesley Professional.

Mark, D. 2019. Modular tactical influence maps. In *Game AI Pro 360*, 103–124. CRC Press.

Mendonça, M. R.; Bernardino, H. S.; and Neto, R. F. 2015. Stealthy Path Planning Using Navigation Meshes. In *2015*

Brazilian Conference on Intelligent Systems (BRACIS), 31–36.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602.

Moldenhauer, C.; and Sturtevant, N. R. 2009. Evaluating strategies for running from the cops. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4): 293–311.

Ray, D. 2023. Navigation in Adversarial Environments Guided by PRA* and a Local RL Planner. Forthcoming.

Sturtevant, N.; and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, volume 5, 1392–1397.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2): 144–148.

Sturtevant, N. R.; Sigurdson, D.; Taylor, B.; and Gibson, T. 2019. Pathfinding and Abstraction with Dynamic Terrain Costs. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 15(1): 80–86.

Wang, B.; Liu, Z.; Li, Q.; and Prorok, A. 2020. Mobile Robot Path Planning in Dynamic Environments Through Globally Guided Reinforcement Learning. *IEEE Robotics and Automation Letters*, 5(4): 6932–6939.