Developing Intelligent Agents for Physics-Based Adversarial Action Games

Gabriel Montague, Brandon Price
Harvard University

1. INTRODUCTION

The genre of computer "action" games typically involve a player with control over a single game character. To succeed in such games, players typically require hand-eye coordination and the ability to react quickly to new information. Although real-time strategy (RTS) games also require quick reaction to information, RTS games generally do not emphasize hand-eye coordination and fine-tuned control over player movement. We use the term *physics-based* to describe action games that take place in worlds that are loosely governed by the laws of physics. In physics-based games, the positions of various game objects are continuously updated according to some velocity, acceleration, or collision force.

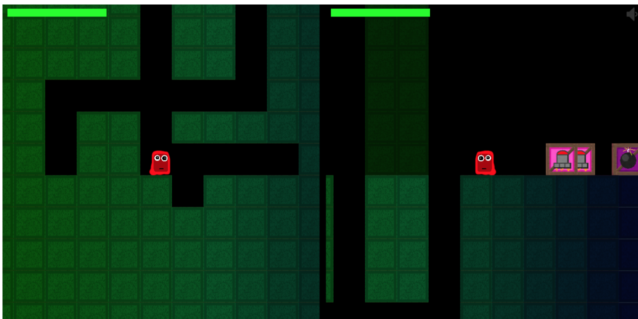

Figure 1: Split screen view of the *Tunnel Wars* game

Most published research in the development of computer-game-playing agents has been focused towards RTS games, with particular emphasis on the popular *StarCraft* series of RTS games (Blizzard Entertainment, 1998–present). As is exemplified by (Stanescu et. al., 2014), successful RTS agents often employ a hierarchy of solution space searches – one for each strategic aspect of gameplay.[1] Such systems are often divided into "layers": the topmost layer directs the behaviors of levels beneath through the selection of general strategy, while the bottommost layer solve specific aspects of gameplay and produce actions. In this paper we apply a similar layer-based divide-and-conquer approach to physics-based action games. Our method employs three layer: a strategy-selection layer, navigation layer, and action layer, each of which will be discussed in detail.

We test the methodology on the *Tunnel Wars* action game (www.tunnelwars.com), which was reworked to provide an abstraction of the state space. The *Tunnel Wars* game involves competition between two characters, originally both controlled by keyboard input. The characters can move continuously in the left and right directions and are pulled downwards by gravity, but can jump if they are touching the ground or walls. A character can also fire projectiles to reduce the health of the opponent, or to tunnel through the 30x46 destructible grid of ground blocks that the game is set upon. If a player's character runs out of health then then they lose the game, and their opponent wins. Despite the simple mechanics of the game, complex strategies are often needed to win due to the many configurations of tunnels that can arise. The *Tunnel Wars* website contains further details. It is worth noting that the gameplay is nondeterministic, with random outcomes influencing the motion of some projectiles and power-up spawning.

In applying and training our method with the *Tunnel Wars* game, our aimed was to achieve a solution the was both runnable in real time and effective in competing against human players.

2. METHOD SUMMARY

The proposed methodology operates using a hierarchical organization of decision-making processes, similar to the work of Stanescu et. al. Although our Tunnel Wars implementation uses perfect knowledge of the game state, as we will see this is not a hard requirement. However, given a model of opponent behavior, the agent should be able to approximatively simulate "steps" of the game, which occur at a rate of 60 per second. The agent also will rely on the implementation of various game-specific heuristics to estimate closeness to the game's end or to the end of a particular objective. Given these requirements, along with a current game state, the agent produces a calculated action. In the Tunnel Wars context, there are only four actions available to the agent: left, right, jump, and fire (projectile), and each state represents the game-state at a given step or frame.

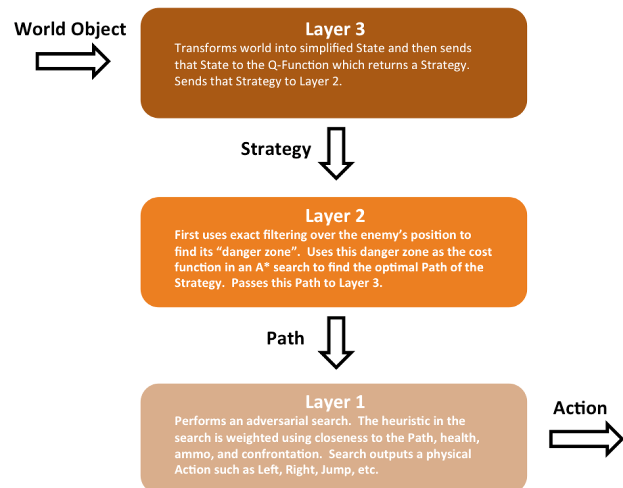

Figure 2: Interaction between layers of the Tunnel Wars agent

[1] Stanescu, Marius, Nicolas A. Barriga, and Michael Buro. *Hierarchical Adversarial Search Applied to Real-Time Strategy Games*. Edmonton, CA: University of Alberta

As mentioned, the agent is composed of three layers. The highest level layer (layer 3) employs Q-Learning and selects the optimal strategy based on a simplified state representation. The middle level layer (layer 2) uses classical search and exact filtering in order to find a motion path that conforms to the optimal strategy. The lowest level layer (layer 1) performs a variation of adversarial search to find the appropriate game action. The functionality and interaction of the layers is diagrammed in Figure 2.

2.1 ACTION SELECTION

Given that we are provided with a way to advance the state space, and that there are a discrete set of actions to choose from at each step, a natural method for probabilistic adversarial games is the expectiminimax adversarial search algorithm. However, there are a number of reasons why this classical approach is ill-fitted for physics-based action games:

(1) This model assumes that the other player performs perfectly rational actions at each step. This assumption has been shown to expose weaknesses, especially in games that involve "metareasoning", or strategy selection based on the opponent's strategy. For nondeterministic games (such as Tunnel Wars), it has been shown that the optimal behavior in certain states may be random behavior.[2]

(2) Due to a real-time computation cap of 1/60 or 1/30 seconds per frame, the expansion depth of performing an adversarial search of any kind is severely limited, even with deterministic alpha-beta pruning.[3] Once probabilistic outcomes are incorporated into the state advancement model, the expansion depth becomes even more limited.

(3) Given the small expansion depth, it becomes unclear, given the complexity of strategy-based gameplay, how to devise a general heuristic (oracle) that can evaluate the utility of a future game state. A heuristic function evaluated for each leaf node in the search that encapsulated considerations such as those of (1) of semi-stochastic complexity in a would make the game unplayably slow.

(4) At 30 to 60 frames per second, human players are not adequately mimicked in the adversarial simulation of the search, as visual signals take humans on average roughly 250 milliseconds (15 game state transitions at 60 frames per second) to react to.[4]

In Tunnel Wars, it was found that even with a simple alpha-beta pruned adversarial search, human testing revealed that the AI agent could not feasibly expand far enough into the future to dodge a single projectile. Caching of search expansions was also found to be of little help. Due to the strong deviation of human adversarial behavior from the expected minimax behavior (reaction time, stochastic strategies), new expansions needed to be calculated each frame despite cached results.

To combat each of these problems, we suggest the use of action-strings (predefined sequences of actions) in the place of single actions. Although this substitution sacrifices a degree of fine-tuned control over movement, the effect is no worse than that of slowing the reaction-time of the AI, which is already significantly faster than that of its human opponent. The exact actions to be placed together in an action string will vary depending on the game, but the action-strings should together form a basis by which the effect of taking a single action can be achieved or approximated given multiple frames. With action-strings replacing single actions, the standard expectiminimax search can be performed over the state space to achieve generally better results.

In the Tunnel Wars case, human testing revealed an action-string set of just four action-strings of length four was effective for reacting to risks and rewards close to present. Although adopting this methodology caused the agent to climb walls and fired weapons at a slower, less optimal, rate – drawbacks often observed in human players with less dexterity – the agent could now approximately expand game states four times farther into the future, which was enough to dodge projectiles. An additional benefit was gained because the adversarial search only needed to be conducted every four frames (the length of the action string).

This adapted version of the expectiminimax search forms the basis of the first layer. Although action-strings are insufficient to combat the problems of (1) and (3), we found it to be effective in combating the issues of (2) and (4). Action-strings more closely mimic human adversarial behavior in terms of reaction time, and are thus often more appropriate for state-expansions of minimax. Additionally, the expansion depth is enhanced at least by a factor of the expected length of an action-string. In a real-time setting, when an action-string is selected, computational power may be freed in subsequent frames by allowing the agent to perform each of the selected actions in the action string without re-computation of the optimal string. Although a fix for (1) is not offered here, (3) is solved by the upper layers 2 and 3.

2.2 STRATEGY SELECTION

We now describe the uppermost layer of long-term (lasting multiple seconds) strategy selection. The problem of metareasoning becomes more acute at this level, as long-term behavior is more random and frequently deviates from optimal. The problem of combating metareasoning and finding an accurate heuristic for the utility of a single state can be naturally linked together. A good utility-evaluating heuristic would learn to account for strategies and metareasoning and adjust the agent's actions accordingly. Our model uses Q-learning to address the issue. Q-learning requires reinforcement from human players, and develops a "Q-function" of state-action utilities that reflects the various strategies used upon it, as well as its own strategies discovered through experimentation. Using an approximative reward function or heuristic the utilities can be learned. For example, reward is given out for Tunnel Wars agents winning or losing health, or winning or losing the game.

Ideally, the Q-function would directly influence the heuristic function of layer 1, providing the learned values of each state by marginalizing over the actions available at that state. Also ideally,

---

[2] Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Second ed. Upper Saddle River, NJ: Pearson Education, 1995. Page 613.

[3] Our original intention with the Tunnel Wars agent was to embed it in a web browser, so we did not consider GPU parallelism as an option for optimization.

[4]"Reaction Time Statistics." *Human Benchmark*. Web. 9 Dec. 2015. <http://www.humanbenchmark.com/tests/reactiontime/statistics>.

each state-action pair would be represented in a dictionary of utility values holding mapping from every state in the game world. Although many action spaces (including that of Tunnel Wars) are sufficiently small for this, the state of any real action game involving continuous positions is too large to be stored on any system. To approximate similar state spaces, various grouping algorithms were considered. Not only must the continuous variables of various positions, velocities, and forces be considered, but also discrete state spaces such as the Tunnel Wars 30x46 array of ground blocks also carries important information for the evaluation of utility. Although continuous state spaces are addressed by a number of algorithms outlined by Gaskett, Wettergreen, and Zelinsky's review of the subject, none account for mixed continuous and discrete state spaces.[5] A method proposed by Gasket et. al, involves interpolation of states between "wires" adjusted by a neural network. However, this can overlook important pieces of the game state. In Tunnel Wars this overlooks the interconnectedness between the players through tunnels, information the would be lost in most of the Q-learning variants proposed.

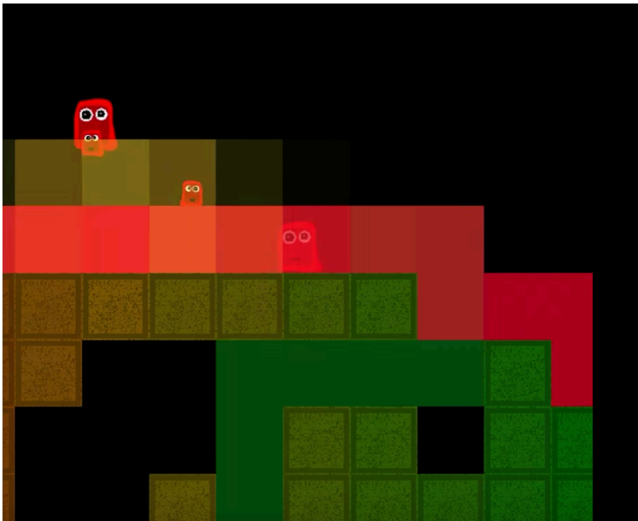

Figure 3: The filtered danger zone (red, yellow) and A* path (green), for an agent on the ground executing "Run Away"

Without an accurate, feasible state representation, the state space of the Q-learning layer must be generalized and discretized. The more generalized the state representation becomes, the less useful it is to learn the utility of something as simple as "moving left a step" given a generalized state. It is for this reason that Q-learning is confined to the topmost layer of strategy selection. Adopting strategies at the highest layer allows for hierarchical search chains as exemplified by Stanescu et. al's published solution to StarCraft, a similarly realtime and strategic game.[6] In our model, strategies are realized by specific targets-of-interest in space that the player must move to. In physics-based action games, there is often little difference between enacting a strategy and moving towards a target. Layer 3 therefore takes in simplified state representations and produces a target-point for the agent to move to. Layer 2 is then responsible for coordinating the objective produced by Q-learning with the heuristic or oracle function of the expectiminimax of layer 1. When layer 1 reaches the maximal expansion depth, layer 2 evaluates the conformance of the current state to the strategy of layer 3.

A generalized form of simplifying the state spaces is still needed to further generalize this methodology. However, in Tunnel Wars we achieved satisfactory state representations by categorizing states based on discrete settings of the variables "Projectile", "Ammo Amount", "Opponent Projectile", "Opponent Ammo Amount", "X Distance from Opponent", "Y Displacement from Opponent". The Tunnel Wars strategies, each corresponding to a target position, were "Attack", "Run Away", "Get Ammo", and "Dig Down".

The way the algorithm initialized a transition and thus got a new strategy had to be done in a very specific way because when a transition occurs between strategies is not obvious. The way in which we decided to go about it is if any of the following occurs: the current state changes, the player completes its current path described by layer 2, the player deviates from its current path by an fixed distance, the enemy's danger zone changes by an fixed amount, or if an fixed amount of time passes by. In any one of these cases a transition occurs: the player will update reevaluate the current state to determine a new strategy. In order to build a large enough dictionary of utility values, the Tunnel Wars system was trained for 6 hours with the help of willing human participants to obtain the 13,000 utility values.

2.3 STRATEGY EXECUTION

As mentioned, each strategy provides a different heuristic to the adversarial search, but all incorporate a metric of closeness to a position. As physics-based action games often provide complex worlds that must be navigated through, merely providing the strategy target position is insufficient to produce real navigation behavior. For this reason, layer 2 conducts pathfinding (in the case of Tunnel Wars an A* search) to guide the agent to the target position. Apart from navigating around physical obstacles, the path-finding must also avoid zones in which the agent could lose the game or lose Q-learning reward (health). In many action-based games such zones take the form of areas in which enemies, projectiles, or some other form of danger could exist. Such danger zones can often be computed using Bayesian methods such as exact or particle filtering. As the danger zone only will be encountered after many frames go by, it is appropriate to simulate exact filtering for a time-length proportional to the distance from the target destination. To combat the computational demands of per-frame exact filtering, action-chaining should be again employed, or else an alternative discretization of state space. In the Tunnel Wars case, this was achieved by discretizing at the ground block level as shown in Figure 3.

Layer 2 then operates as follows: Each time a strategy is selected, a danger zone is computed from Bayesian filtering of possible dangerous positions. In the case of Tunnel Wars this was made feasible by snapping positions to the nearest ground block. An A* path is then computed from the agent to the goal specified by the strategy, in which the cost function is a mixture of length of the path, and a measure of the amount of overlap with the enemy's danger zone. Figure 3 displays an example danger zone and computed path. Layer 1's adversarial search oracle then is

[5]Gaskett, Chris, David Wettergreen, and Alexander Belinsky. *Q-Learning in Continuous State and Action Spaces*. Canberra, Australia: The Australian National University

[6] Stanescu, Marius, Nicolas A. Barriga, and Michael Buro. *Hierarchical Adversarial Search Applied to Real-Time Strategy Games*. Edmonton, CA: University of Alberta

adjusted to weight conformance to the path in order to have the agent follow what it thinks is the best path towards achieving its strategy. A heuristic function of the strategy can also be incorporated at this level. In this way, a general selected strategy such as “Get ammo” can be translated into low-level actions such as moving left or right.

3. RESULTS

In our implementation of this Tunnel Wars AI algorithm, we first started with Layer 1. We created the adversarial search algorithm described above without weighing any closeness to a specific path in its heuristic. It turned out that this algorithm could stand alone beat any human in as long as there were no tunnels. The search algorithm could anticipate rockets, bombs, etc. much quicker than a human, but could not tunnel or execute or save itself from complex strategies involving tunnels. This motivated the incorporation of learning and the creation of layer 3.

With all three layers, the AI agent was able to use all four strategies well to become a formidable opponent against human players. It exhibit particular strengths in:

- Gathering power-ups. It is able to find the quickest path to get power-ups, and opt for this strategy whenever power-ups are needed to win.
- Attacking. When the agent is armed with projectiles, it tracks down the enemy quickly and then use fires the projectiles precisely in manners that are quite difficult to dodge.
- Dodging. The AI can dodge anything thrown at it like Neo in the Matrix.

However, there are few issues that still hold back the agent from competing perfectly. It struggles with the following:

- Staying out of dead ends. Despite the danger zone and A* heuristic, the Tunnel Wars agent still occasionally goes down tunnels it should avoid, which means that it can be defeated somewhat easily if this happens. This is due to power ups spawning randomly and the danger zone shifting in a way that it had not anticipated, as it has no ability to equate dead-ends with danger without more advanced state collapse methods. The use of a convolutional neural network on the blocks around the agent would be likely to solve this, and recognize dead-ends.
- Fighting itself. The agent plays the game conservatively by avoiding the other player unless it knows that it can win, so simulating games against itself can occasionally take a while and still induce deadlock.
- Freezing. Although calculation of utilities and paths could easily be spread among free frames that are not used by layer 1, this was not implemented and as such strategy changes were more abrupt but also induced lag. This could easily be corrected by distributing the calculation among free frames.

4. GENERALIZATION

Although Tunnel Wars is unique in its block-based gameplay, physics-based action games share many qualities with Tunnel Wars that allow the technique to be generalized to other games of the same genre. As discussed, layer 3 must simplify the state space in some way which must be specific to the individual game, and simplify optimal gameplay into discrete strategies. Additionally each strategy of the game must come with its own heuristic of effectiveness of execution. The danger zone concept of layer 2 is similarly applicable to almost all action games. However, the discretized pathfinding may need to be adapted to the world-geometry of non-block-based games by applying action-chaining similar to that of layer 1.

5. APPENDIX: HOW TO PLAY

1. For instructions on building the game using Unity, follow setup instructions in the project’s README.md inside the game directory.
2. Then double click PlayerVsAI.app to run the game.
3. The AI agent plays on the right side, and the player plays on the left side. Move around your player with the W, A, S, D keys and press F to fire projectiles. You can climb walls by moving against them.
4. Collect the randomly appearing projectile boxes and use them to dig under the wall to the other side.
5. Use the projectiles to defeat the agent