ARTIFICIAL INTELLIGENCE FOR MONSTERS IN TOWER DEFENSE GAMES (AIMTD)

Federico Nusymowicz
fnusy@seas.upenn.edu

EAS 499 Senior Capstone Project
Advisor: Kostas Daniilidis
kostas@cis.upenn.edu

Coordinated by Max Mintz
University of Pennsylvania

ABSTRACT

Tower defense game developers typically use waypoints to predefine enemy attack routes. Unfortunately, these scripted attack patterns translate into predictable game experiences with low replay values. The TD genre would be far richer if enemies could avoid dangerous areas of the map, elect to sacrifice themselves for their teammates, or choose to work together in teams.

AIMTD is an open-source toolkit for TD game development. The AIMTD API allows developers to effortlessly evolve their TD enemies from mindless 'creeps' that follow scripted waypoints to intelligent monsters that react to player strategies in real time.

The project's ultimate goal is to help create more dynamic TD game experiences, and to lay the foundations that will aid developers create the next generation of TD games.

I. INTRODUCTION

Tower defense (TD) is a subtype of real-time strategy games. The TD player's goal is to defend a vulnerable objective from oncoming waves of enemies. To do so, the player must allocate resources for building defensive towers that shoot at enemies as they try to pass through the TD map. If enough creeps make their way through the map, the player loses. If the player survives each wave, on the other hand, the level is finished and the player wins.

TD games have been around for at least two decades [1]. The genre remained relatively obscure until 2003 however, when Blizzard Entertainment released a series of TD maps as part of their expansion to *Warcraft III: The Frozen Throne* [2]. As millions of *Warcraft III* players caught onto the genre, many started making their own levels using the game's built-in map editor. Players then began creating TD levels using map editors from other popular real-time strategy games like *Starcraft*. Hobbyist 'indie' game developers started releasing free Flash-based TDs shortly thereafter, and the genre kept growing steadily ever since.

One of the driving forces behind the genre's growth is its appeal to casual gamers. Since TD game interfaces are inherently simple, inexperienced gamers can understand game mechanics with relatively little effort. To ease players into their game even further, developers typically make the first few levels fairly easy to beat. Casual players can also complete a TD level within a short period (usually 2-10 minutes), so they need not invest large amounts of time to enjoy the game.

Despite the genre's appeal to casual gamers, 'hardcore' gamers also enjoy TDs. As levels progress, their difficulty also tends to increase; this in turn demands richer and more creative strategies on the player's behalf. Developers can introduce all kinds of creative features to make their games more challenging and interesting: they can make enemies become faster and stronger, make maps become more complicated, unlock different types of towers for the player to build, or introduce hundreds of other game variations.

Besides their appeal to a wide audience, TDs became popular because they're so easy to develop. A gamer with no programming knowledge could quickly put together a TD using a map editing GUI like the one bundled into

Starcraft II. A more enthusiastic hobbyist could also develop a full-featured TD through user-friendly technologies like Adobe Flash and ActionScript. Today, thousands of free-to-play TDs are available on the web, so TD gaming enthusiasts can always find more content.

A. Game Mechanics

In order to introduce the typical TD game's mechanics we will look at *Shock Defence* [3], one of the simplest TDs available on the Web. Figure 1 provides a screenshot of the game.

Figure 1: Shock Defence screenshot

Shock Defence begins with a 'setup' phase wherein the map is completely empty. The player starts off with $75 in resources, which can be spent on building three different types of Basic Towers. Each tower type has a different building cost, firing radius, firing speed, and firing damage. The player may build towers anywhere within the green areas of the map.

Once the player feels prepared to defend against the first wave of enemies, the "Next Level" button on the bottom right of the screen releases the first wave of creeps. Enemies then appear one by one from the point denoted "Spawn" and travel through the map's brown path in an attempt to reach the "Objective".

As enemies travel through the map's path, they come within range of the towers' firing radiuses. Each tower locks onto the first enemy it sees and shoots at it until the enemy dies or until the enemy runs past the tower's firing range. If one of the enemies reaches the objective, the player loses a life. The game ends if the player has no lives left.

Each time a tower kills one of the enemies, the player is awarded an amount of resources proportional to the enemy's difficulty (in the case of Figure 1, the level 1 sheep crossing the map award $1 each). The player can then allocate newly acquired resources to build new towers or upgrade existing ones.

Once all creeps are dead, the game goes back into its 'setup' phase. Here the player may once again calmly strategize resource allocation. When ready to face the next wave of creeps, the "Next Level" button sends out the second wave of enemies.

As the game progresses creeps become increasingly harder to kill. They also gain the ability to move faster or sustain more damage, or they may gain special attributes or other abilities. 'Flying' creeps, for example, can only sustain damage from certain tower types. This forces the player to strategize from early on: even though flying-damage towers may be useless investments in the early levels, being caught without them later on could translate into losing the game.

The player can also spend resources to unlock Advanced Towers, each with its own unique features. New tower combinations open up possibilities for different kinds of strategies that add depth and richness to the game.

The game ends once the player beats all 35 levels.

B. TD Variations

Most of today's popular TDs possess far more content than *Shock Defence*. In fact, the equivalent of the entire *Shock Defence* game represents only a single level in one of the larger games like *Kingdom Rush* [4]. Levels in *Kingdom Rush* consist of several waves of enemies; different levels utilize entirely different maps. Players can save their progress in between levels or upgrade the strength of specific tower types for all future games. They can also purchase special abilities, like the power to rain meteors down on creeps.

Other games like *Gemcraft Labyrinth* [5] use complicated maps with forking paths, multiple spawn points, and several different vulnerable objectives. These types of games are usually more challenging, since defending complicated terrains demands more strategic planning. Figure 2 provides a screenshot of the game.

Several "reverse TDs" like *Anomaly Warzone Earth* [6] take a different approach, flipping around player responsibilities entirely. In reverse TDs, players strategize creep deployment patterns rather than tower placement decisions. The list of TD variations goes on.

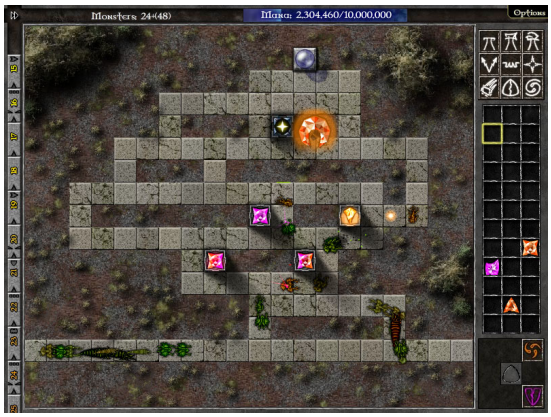

Figure 2: Gemcraft Labyrinth screenshot. Creeps spawn from the bottom left and right areas of the map, then follow scripted paths towards the vulnerable pale orb at the top-middle.

C. Game Elements

Rather than list out all the possible twists on the TD genre, we turn to examining and categorizing some of the most broadly shared features across TD games. The list of game elements below is by no means exhaustive, but is rather meant to help define the subset of TD games for which AIMTD will be relevant:

Creeps. Creeps typically possess predefined movement speeds and health points. In different types of TDs, the player may have varying amounts of control over creep deployment patterns and enemy movement behaviors. AIMTD focuses primarily on games wherein enemy movement patterns are controlled by the computer. Since the vast majority of TD games feature computer-controlled enemy paths, AIMTD should prove relevant to most TD games.

Towers. Each tower usually has a predetermined firing range, firing damage, and firing speed. Some TDs also allow the player to deploy 'blocking' towers that creeps must tear down before continuing on their paths. AIMTD will not be optimized to deal with blocking towers, since they slow down game flow and are thus relatively unpopular.

Terrain. Some TDs like *Plants vs. Zombies* [7] constrain creep movement to one dimension. Other TDs allow for 3-dimensional attack paths. AIMTD focuses on 2D terrain architectures like the one in *Shock Defence*, wherein enemy movement patterns can be modeled in the context of a Cartesian plane. This type of terrain design is by far the most popular across the TD genre.

D. Motivation

Once players figure out a winning tower build order, the level loses all replay value. Beating the level once again becomes a simple exercise of repeating an old winning strategy. To address this problem, developers began incorporating different maps, scripting new enemy attack routes, varying creeps' strength levels and deployment rates, and introducing more and more varied tower types.

Unfortunately, even with all the above content additions, TD game experiences still remain largely formulaic. Players face the exact same challenge every time they open up a map, which ensures they will eventually succeed by simple trial and error. Thus, TD games usually resemble glorified puzzles more so than true dynamic challenges.

One of the driving factors behind gamer satisfaction comes from the level of challenge a game presents [8]. Since TDs are not built to provide dynamic gaming experiences, 'hardcore' gamers commonly categorize TDs as a "simple" genre [2], meant only for casual entertainment.

AIMTD seeks to help create richer, more dynamic, and more rewarding TD games. The AIMTD API provides developers with the tools needed to implement enemy artificial intelligence. By allowing creeps to adapt to the player's strategy in real time, the formulaic nature of TD gaming will disappear and games will become far more challenging and engaging.

II. AIMTD Overview

AIMTD is an open-source toolkit for TD game development. The project's goal is to transform TD enemies from mindless 'creeps' that follow scripted waypoints to intelligent monsters who choose their paths in real time. All source code and documentation is available freely at http://www.fedenusy.com/aimtd.

With AIMTD, monster behavior patterns are put completely under the game developer's control. Monsters could follow the shortest path to the nearest objective, they could follow the path that's least likely to get them killed, or they could work together in teams in order to maximize their chances of reaching map objectives. The API handles all the details of monster movement, allowing developers to focus on other innovative game elements.

A. Features

AIMTD handles all aspects of monster movement. The API takes into account monster

health and speed, map layout, and as much of the player's strategy as the developer wishes to expose.

The monster AI engine currently supports scripted waypoints, forking paths in the map, multiple vulnerable objectives, and 2D monster movement in any direction. Additionally, the API allows developers to specify one of 3 distinct intelligence levels for each of their monsters. Different intelligence levels correspond to increasingly complex behavioral patterns.

B. Applications

AIMTD would integrate particularly well with TDs that use complicated maps like *Gemcraft Labyrinth* [5]. Monsters controlled by AIMTD would dynamically choose their paths based on the player's tower placement decisions, which would in turn lead to entirely customized game experiences. To make games even more dynamic, developers could begin using AIMTD to experiment with randomly generated maps. Game AI researchers have already encountered early successes with pseudorandom TD map generation techniques [9]; AIMTD would act as a perfect complement.

The API could also help developers balance TD level difficulties. AIMTD allows game designers to increase or decrease their monsters' intelligence levels effortlessly, which would in turn make their monsters harder or easier to kill. By extension, the API would also prove helpful for more sophisticated developers implementing dynamic difficulty adjustment techniques [10]. Since the API offers easy-to-change behavioral complexities, TDs could adjust enemy movement patterns to become more or less challenging based on the player's displayed skill level.

AIMTD also opens up interesting possibilities for player-versus-player TD variations. One of the players could take on the traditional role of 'tower builder' for example, while the other could play the reverse-TD role of 'creep deployer'. The 'creep deployer' could then unlock more and more advanced creep intelligence levels as the game progressed.

As a final example of a possible application, developers could combine AIMTD with several of the elements described above in order to create a massively multiplayer online (MMO) game. Randomly generated maps combined with player-versus-player interactions, for example, would give rise to the nearly endless gameplay possibilities that characterize most of today's commercially successful MMOs.

III. API

AIMTD's continued success will rely upon a growing user base. Since AIMTD users will consist exclusively of game developers, the API will only grow if it is easy to adopt and understand. To that end, the project's homepage hosts an API tutorial as well as frequently updated documentation.

Besides ease of adoption, API design was driven by two guiding principles. The first was to make AIMTD relevant to as many different TD variations as possible. The second was to provide a sound enough design with a high enough level of abstraction to make the interface easy to work with on an ongoing basis. When designing the AIMTD API, the two considerations often clashed with each other, but in the end we struck a sensible balance. We now describe the basic ways a game developer would interact with the AIMTD API when programming a TD game.

A. The Coordinator

This object acts as AIMTD's clock. The Coordinator's TICK() command signals each Monster to move according to the Map layout, the Monster's intelligence level and movement speed, and various other game factors. Most developers will probably call TICK() once before refreshing each frame.

B. The Map

The Map serves as an abstract representation of the TD level's terrain. For AIMTD's purposes, relevant Map coordinates include:

- Path coordinates that Monsters can travel over.
- Objective coordinates that Monsters are always trying to reach.
- Field coordinates over which the player could build towers.
- Rock coordinates that represent dead space.

Each of the above coordinate types corresponds to a distinct integer. Developers construct the Map object via an array of arrays of integers, where each integer's inner index corresponds to the coordinate's x-position and its outer index corresponds to its y-position. Figure 3 provides an example AIMTD Map layout.

By default, AIMTD allows Monsters to travel between diagonally adjacent Path coordinates.

Developers can disable diagonal path links at the time of Map construction.

C. Monsters

Creeps must inherit from the Monster class in order for AIMTD to control their movement. Developers need to supply the Monster class with the following creep-specific variables:

- Starting x- and y-coordinates.
- Health points: the amount of Tower damage this Monster can sustain before dying.
- Speed: measure of the Euclidean distance units this Monster can cover within a TICK(), defined as (speed/100) units per TICK(). A speed of 200 thus implies that if the Monster's initial position is (1,0), its speed is 200, and the Coordinator decides that the monster should move east, then after calling TICK() the Monster will be located at (3,0).
- Intelligence level: variable that determines this Monster's behavioral complexity. With an intelligence level of 1, the Monster will always follow the shortest path towards the nearest objective. With an intelligence level of 2, the Monster will follow the path that's least likely to get it killed. Intelligence level 3 is an improvement on level 2 that allows Monsters to coordinate group tactics in conjunction with other Monsters.

C. Towers

Similar to creeps, towers must inherit from the Tower class. AIMTD tracks Towers to factor player strategy into its calculations. Towers must define the following variables:

- X- and y-coordinates.
- Fire damage: amount of damage a single projectile will inflict on a Monster's hit points.
- Firing range: the Tower's firing radius in terms of distance units. A Tower located at (3,2) with firing range of 1, for example, will be able to fire at Monsters passing through (3,3), but not at Monsters passing through (4,3).
- Firing speed: frequency at which the tower fires, defined as (speed/100) shots per TICK(). A speed of 200 thus implies the tower can fire two projectiles within a single call of the Coordinator's TICK().

D. Putting It All Together

The developer begins by constructing a Map and passing it to the Coordinator. Every time the game adds a new Monster or Tower (or removes an old one), the game alerts the Coordinator.

```
{
{0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0},
{0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0},
{0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0},
{0,0,1,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,0,0},
{0,0,1,1,1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0},
{0,0,0,0,1,1,1,1,1,0,0,0,0,0,1,0,0,0,0,1,0,0},
{0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1,0},
{0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1,1,0,0},
{0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0},
{0,0,0,0,0,0,0,0,1,0,8,0,0,0,0,0,1,0,0,0,0},
{0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,0},
{0,0,0,0,1,0,8,1,1,1,1,0,0,0,0,0,0,0,0,1,0},
{0,0,0,1,1,0,0,1,0,0,0,1,1,0,0,0,0,0,1,0},
{0,0,1,0,0,0,8,1,0,0,1,0,0,1,0,0,0,0,0,1,0},
{0,0,1,0,0,0,0,1,0,0,1,0,0,1,0,0,0,0,0,1,0},
{0,0,1,0,8,0,1,0,0,0,1,0,0,0,1,1,1,1,1,1,0},
{0,0,1,0,0,1,0,0,0,0,1,0,0,0,1,0,0,0,0,0},
{0,1,0,0,1,0,0,0,0,0,2,0,0,0,0,1,1,0,0,0},
{0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0},
{0,2,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2}
}
```

Figure 3: AIMTD 20x20 terrain layout as a Java array of arrays. 1s represent Path points, 2s represent Objective coordinates, 0s represent Field points, and 8s represent dead space in the Map.

Before refreshing each frame, the developer calls TICK(). AIMTD then moves Monsters through the Map according to the terrain layout, the Monster's speed, and various other factors depending on the Monster's intelligence level. All these mechanics are hidden from the game developer.

After calling TICK(), developers update Monster positions on display by retrieving x- and y-coordinates from each Monster object.

IV. IMPLEMENTATION

At its core, AIMTD is a pathfinding toolkit adapted to the needs of TD game developers. The natural choices for data structures and algorithms thus revolved around graph theory.

Upon Map construction, AIMTD creates a node object representing each Path coordinate from the terrain layout. Bidirectional edges are then established between all adjacent Path nodes (we create bidirectional rather than undirected edges because future versions of AIMTD may need to support directed graph structures).

In the following subsections we describe AIMTD's internal mechanics.

A. Core Algorithms

From the Monster's perspective, we want to find the least costly path from the Monster's current position towards any one of the Map's objectives. This is an instance of the single-source shortest path problem [11]. While the problem could also be interpreted as a series of single-pair or single-destination shortest path problems [11], either of those approaches would have introduced

complications when implementing waypoints, which we will discuss later.

To solve the single-source shortest path problem, AIMTD relies upon Dijkstra's algorithm [11]. The algorithm fits AIMTD's needs perfectly because, given nonnegative edge costs, the procedure guarantees an optimal least-cost path towards each node (i.e. coordinate) in the graph. We can then check, for each of the Map's objectives, which objective the Monster can pursue by incurring the least cost.

An alternative implementation could have used the Bellman-Ford algorithm, which would have tolerated negative values from the edge cost function. The added flexibility, however, would have come at the cost of a slower runtime. Since none of our cost functions allowed for negative edge weights anyway, the Bellman-Ford algorithm proved unnecessary.

While we may develop a need for cost functions that go below zero in the future, for the moment this seems unlikely. To understand why, note that Monster movement almost always implies some positive cost. At the very least the Monster is spending time in the act of moving, which gives the player time to strategize. Additional time for the player to strategize in turn decreases the Monster's likelihood of survival. This implies a positive cost so long as the Monster stays on the Map, regardless of which action the Monster decides to take.

We now turn towards the Coordinator's function. First, note that the Coordinator needs to ensure that Monsters continuously re-evaluate their environment, since this is the only way Monsters can make adaptive real-time decisions. This requirement necessitates a high-level TICK() function similar to the following:

```
TICK()
    START-TICK()
    for each Monster m ∈ ActiveMonsters
        COMPUTE-DIJKSTRA(m, Map)
        Node obj = SELECT-OBJECTIVE(m, Map)
        MOVE-TOWARDS-NODE(m, obj)
```

Where we define ActiveMonsters to represent the set of living Monsters under the Coordinator's control that have not yet reached an objective.

START-TICK() begins by updating the set of ActiveMonsters, removing dead creeps and Monsters that have reached their objectives from the active set. The function also performs a series of other housekeeping tasks, like resetting node attributes to their default values. We will discuss these attributes further in later subsections.

As the name implies, COMPUTE-DIJKSTRA(*m*, Map) computes Dijkstra's algorithm. The function is modified to also take into account the Monster's intelligence level. Based on Monster's intelligence, COMPUTE-DIJKSTRA decides the appropriate cost function to use when computing edge costs.

SELECT-OBJECTIVE chooses the Monster's next objective node – that is, the endpoint of the Monster's chosen path. Then MOVE-TOWARDS-NODE pushes the Monster along the selected route. We now discuss these two functions together, along with waypoints.

B. Waypoints, Objective Selection, and Movement

Waypoints give developers an additional level of control over Monster paths. While the default AIMTD behaviors should usually suffice, certain levels may require specific Monsters to travel in preset patterns.

Monsters each hold a queue of waypoints. Developers can add to the queue in order to guide their Monster's trajectory however they like. The Monster then moves towards each waypoint successively before attempting to pursue one of the Map's objectives.

To allow for waypoints, the SELECT-OBJECTIVE function works as follows:

```
SELECT-OBJECTIVE(m, Map)
    if m.waypoints is not empty
        return m.waypoints.peek()
    leastCostlyObjective = ∞
    Node objective = null
    for each Node v ∈ Map.nodeSet
        if v is an objective node
            if v.cost < leastCostlyObjective
                objective = v
    return objective
```

Therefore waypoints are treated as intermediate objectives.

Finally, we define the movement function in more abstract terms. MOVE-TOWARDS-NODE(*m*, *obj*) incrementally moves Monster *m* towards the given objective *obj* (which could be a waypoint) through the following steps:

(1) Traverse backwards through the least-cost path ending at *obj*. COMPUTE-DIJKSTRA, which we calculated earlier, makes backward traversal possible. Follow the path until we find the next neighboring node *v* that *m* should move to, starting from its current position.

(2) So long as *m* can still move (as capped by its movement speed / movement capacity per turn), move *m* towards *v* in small increments. Subtract the small increment from *m*'s movement capacity for this TICK().
(3) After each small incremental move, check whether *m* can still move. If not, *m* is done moving for this TICK() and we return from the function. Otherwise *m* can still move, so we check whether *m* reached *v*. If it did not, we go to (2) and keep moving towards *v*. Otherwise *m* did reach *v*. If *v*≠*obj*, we jump back to (1) to figure out the next node in the path. Otherwise, *m* reached *v* and *v*==*obj*. If *v* is an objective node in the Map, *m* took one of the player's lives and is no longer active, so we return from the function. Otherwise *m* reached a waypoint objective, so we call *m.waypoints.poll()* to remove *v* from *m*'s waypoint queue, then go to (1) to continue moving *m* along the next chosen path.

We now discuss the different intelligence levels that separate AIMTD Monsters from common creeps.

C. Intelligence Level 1

Intelligence level 1 Monsters always follow the shortest path towards the nearest objective. Implementation is fairly straightforward.

As we progress through Dijkstra's algorithm [11], let v represent the last node whose fixed cost we determined with certainty. Let u be one of v's neighbors such that u's cost value is not yet fixed. Then we define u's new cost as:

$$cost_1(u) = \min[cost_1(u), cost_1(v) + cf_1(v, u)]$$

Where $cf_1(v, u)$ is the level 1 cost function from v to u. In this simple case, $cf_1(v, u)$ is simply calculated as the Euclidean distance between v and u's coordinates.

D. Intelligence Level 2

Intelligence level 2 Monsters seek to avoid as much damage as possible along their path. They dodge areas with heavy Tower influence and run through the most lightly guarded parts of the Map.

Let T represent the set of all Towers. Then, for every t∈T, define:

- t.*x* as t's x-coordinate
- t.*y* as t's y-coordinate
- t.*dmg* as t's firing damage
- t.*speed* as t's firing speed
- t.*range* as t's firing radius

Similarly, for every v∈Map.*nodeSet*, define:

- v.*x* as v's x-coordinate
- v.*y* as v's y-coordinate
- v.*dmg* as the amount of damage that a Monster standing still on v would receive within a TICK()

Our first goal is to calculate v.*dmg* for each node v. To do this, we begin by defining:

$$T_v = \{\forall t \in T \mid t.range \geq \sqrt{(t.x - v.x)^2 + (t.y - v.y)^2}\}$$

In other words, T_v represents the set of all Towers that can fire upon Monsters passing through v. Then we calculate v.*dmg* as

$$v.dmg = \sum_{t \in T_v} t.dmg * (t.speed / 100)$$

So that v.*dmg* equals the sum of the damages of all the Towers that can fire upon v within a TICK(), standardized for each Tower's firing speed (which is itself defined as 1 projectile per TICK() per 100 units of firing speed).

Now let M be the set of all Monsters. For every m∈M define:

- m.*x* as m's x-coordinate
- m.*y* as m's y-coordinate
- m.*hp* as m's health points
- m.*speed* as m's speed

We finally turn to defining the cost function $cf_2(v, u)$ for intelligence level 2 Monsters:

$$cf_2(v, u) = [cf_1(v, u) / (m.speed / 100)] * [0.5 * (v.dmg + u.dmg)]$$

In plain English, the cost function is equal the number of TICK()s it takes m to cross the edge between v and u, multiplied by the weighted average of the damage m would receive if it were standing on u or v for the length of a TICK().

In practice, one problem with the cost function above turned out to be that when no Towers were built along a path, the path always yielded a cost of 0 (even though the path may have been really long). Furthermore, when the cost function between two nodes was 0, Dijkstra's algorithm would sometimes choose winding paths that were clearly suboptimal.

The simple fix involved adding an additional small factor that would account for distance:

$$cf_2(v, u) = [cf_1(v, u) / (m.speed / 100)] * [0.5 * (v.dmg + u.dmg)] + cf_1(v, u) / 1000$$

With intelligence level 2 working properly, a number of further improvements became possible. For example, when m's cost for reaching an objective exceeded m.*hp*, m knew it had a chance of dying regardless of the path it chose. At that point, m could recalculate the Dijkstra function

under intelligence level 1 assumptions. By choosing the shortest path, m would sacrifice itself for its less intelligent teammates. We discuss simulation results in Section V.

E. Intelligence Level 3

Intelligence level 3 Monsters look to their teammates to employ group tactics. The intuition is that if several Monsters can pass through node v at the same time, then each $t \in T_v$ will need to split its attention amongst many Monsters, thus rendering each Tower t less effective. Our implementation draws some inspiration from [13], which uses influence maps to coordinate a group of boats in a naval capture-the-flag game.

We begin by defining, for every v∈Map.*nodeSet*:

- v.$\boldsymbol{t}$ as a row vector, where $\boldsymbol{t}_i$ represents the number of Monsters that could be traveling over v during the ith TICK().

In order to update all v.$\boldsymbol{t}$'s each TICK(), we first clear and then re-compute them as part of the START-TICK function. Computing each v.$\boldsymbol{t}$ is trivial, as it only involves running COMPUTE-DIJKSTRA under intelligence level 1 assumptions for each m∈M. The only modification to the function happens when a node v's final cost is decided; at that point, we:

- Choose $i = \text{round}(\text{cost}_1(v) / [m.speed / 100])$
- Set $v.\boldsymbol{t}_i = v.\boldsymbol{t}_i + 1$

We then define $cf_3(v, u)$ as a simple potential function [14]:

$$cf_3(v, u) = cf_2(v, u) / (1 + \alpha * [0.5*(v.\boldsymbol{t}_i + u.\boldsymbol{t}_i)-1])$$
$$\text{Where } i = 0.5 * [\ \text{round}(\text{cost}_1(v) / [m.speed / 100]) + \text{round}(\ \text{cost}_1(u) / [m.speed / 100]\ )\ ]$$
$$\text{And } 0 \le \alpha \le 1$$

The intuition behind the equation above is that potentially, there could be $0.5*(v.\boldsymbol{t}_i + u.\boldsymbol{t}_i) - 1$ other Monsters traveling along (u,v) at time i. Since those Monsters would be splitting all Tower damage with m, in order to calculate the adjusted damage cost function, we split $cf_2(v, u)$ by $(1 + [0.5*(v.\boldsymbol{t}_i + u.\boldsymbol{t}_i)-1])$ Monsters. However, not all those Monsters are guaranteed to be traveling through (u,v) at time i, since u.$\boldsymbol{t}_i$ and v.$\boldsymbol{t}_i$ represent a figure closer to the potential *maximum* number of Monsters traveling through u and v at time i. To account for this, we decay the adjustment factor by $0 \le \alpha \le 1$, thus leading to the $cf_3(v, u)$ formula (note: we haven't yet had a chance to run enough level 3 simulations to calibrate an α value, but values around 0.2 seemed to work quite well for the test conditions described in Section V).

V. PERFORMANCE

Defining the appropriate performance benchmarks for AIMTD poses more than a few challenges. The only real measures of the project's success will eventually come from how widely adopted the API becomes and whether AIMTD succeeds at creating a positive impact on the TD genre. Since the project is fairly new at the time of this writing, however, no such measures are yet available.

We turn to more quantifiable metrics instead. In order to test whether intelligence level 2 Monsters are in fact more intelligent than level 1 Monsters, we run a series of Monte Carlo simulations. We expect the smarter level 2 Monsters to leave the player with fewer lives on average.

A. Simulation Setup

The player starts out with 100 lives. Every time a Monster reaches one of the objectives, the player loses a life. Every time a Monster dies, the player earns $30.

For our simulation we used the Map layout depicted in Figure 3. Throughout the beginning of the simulation, two different types of Monsters were released each TICK(). This spawning phase lasted for 50 TICK()s, forming an enemy wave of 100 Monsters total.

The first Monster released during each of the spawning phase TICK()s had the following attributes:

- Starting coordinates (1,0)
- Started with 100 health points
- Had a movement speed of 20 (i.e. the Monster moved 0.2 units per TICK()).

The second Monster type released during each spawn TICK() had the following characteristics:

- Starting coordinates (17,0)
- 75 initial health points
- Movement speed of 32

Thus, during the spawning phase, two different types of Monsters appeared from the top left and top right corners of the Map. One of the Monsters was slightly weaker but faster than the other, which is a fairly common theme across TDs.

Each Monster then tried to make its way down the Map towards one of the three objectives located at (1,19), (10,17), or (19,19). At each TICK() Monsters would follow the paths calculated according to their intelligence levels.

Intelligence levels were uniform throughout a given simulation, meaning each scenario either had 100 intelligence level 1 or level 2 Monsters. The simulation ended when no Monsters were left, because they all either died or reached an objective.

B. Simulated Player

Whenever the player built a Tower, the Tower had the following attributes:

- 25 damage per projectile
- 100 firing speed (i.e. 1 projectile per TICK())
- Firing radius of 2 units
- $250 build cost

Before Monsters began spawning we implemented a setup phase. Here, the simulated player was given $1000 to set up initial defenses. All $1000 was then spent on 4 Towers, each one strategically placed at a high-impact location adjacent to multiple road tiles. The first 4 build decisions created Towers on coordinates (4,3), (17,1), (8,10), and (17,14).

After the setup phase, the simulated player would make a build decision as soon as enough resources became available. Build decisions were completely randomized. So immediately after the player acquired $250 or more, the player would build a Tower at an empty Field coordinate on the Map, chosen at random.

C. Results

We ran 10,000 simulations with level 1 Monsters. In the end, the player was left with an average of 71.1 lives. Maximum lives left were 100, minimum lives left were 44, and the sample standard deviation was 18.5 lives.

We also ran 10,000 simulations with level 2 Monsters. The player was left with an average of 48.2 lives. This was exactly as we expected; the more highly intelligent Monsters caused more damage overall. Maximum lives left were 100, minimum lives left were 31, and the sample standard deviation was 18.4.

Intelligence level 3 is still under development. A few early successful trials suggest average lives left were slightly below intelligence level 2, which is promising. We should note, however, that the marginal difference was by no means significant.

VI. OTHER APPLICATIONS

While AIMTD is meant for TD games, the API could also apply to just about any videogame needing 2D pathfinding intelligence. A developer could, for example, create a Pacman clone wherein enemy ghosts are guided by the AIMTD engine. The developer could place powerful (invisible) towers around areas where level 2 ghosts should not come near.

There could also be several real-world applications for AIMTD.

A. An Example

A taxi company dispatcher could use an AIMTD Map to model a city's coordinates. Taxis would be modeled as level 2 Monsters, customers as objectives, and areas of high traffic as Towers. Based on AIMTD's output, taxi drivers would be directed along the path that leads them to customers causing 'the least damage', i.e. the customers they can get to fastest.

If we switched from our Dijkstra implementation to the Bellman-Ford alternative, we could also deploy level 3 Monsters using a negative α parameter. Then, rather than crowd together around the same paths, taxis would instead move away from each other. From the dispatcher's perspective, this might be useful for idling taxis. The model would help spread cabs out throughout the city, thus increasing the company's coverage.

VII. FUTURE WORK

AIMTD hopes to eventually become a part of (or turn into) a full-on framework for TD game development. In order to do that, the project needs to grow and mature in several different ways:

A. Implement more advanced AI

AIMTD's immediate next step involves debugging, testing, and benchmarking intelligence level 3 Monsters. Refinements to the existing AI functionality will follow from there.

Other more advanced types of AI will eventually make their way into AIMTD as well. Developers could potentially be given the ability to customize their Monsters' fields of vision for example.

Another interesting possibility involves building a level 4 intelligence level wherein Monsters begin coordinating via agreement protocols rather than potential function heuristics.

B. ActionScript Integration

Most of today's freely available TDs are based on Flash and ActionScript. AIMTD will

eventually integrate with Adobe's technology to help spread adoption.

C. AIMTD-Based Game

One of the best ways to test AIMTD, identify needed features, and refine the API would involve building a TD game based on AIMTD.

The game would potentially be an MMO.

D. Tower Intelligence

Besides Monster intelligence, AIMTD could also begin coordinating Tower firing decisions with relative ease. Since the Coordinator is already aware of all Towers anyway, this functionality could be added without affecting the API in any way.

E. Support Additional Features

AIMTD does not currently support several classic TD features. TDs typically allow players to build 'splash damage' Towers for example, which damage all Monsters within a given area.

In order to support this specific game element, the AIMTD Tower class and the level 3 cost function would both need to change.

F. Dynamic Difficulty Adjustments

The AI engine could begin scaling Monster intelligence up or down based on the player's displayed skill level. Dynamic difficulty adjustments often improve the game experience [10].

Developing DDA would also likely provide further insights into player strategy, which would in turn help improve the AI engine.

G. Incremental Updating

Every time TICK() is called, the function re-computes every Monster's pathfinding algorithm. While we saw no real performance bottlenecks in our simulations, bigger TD games with larger Maps, more Monsters, and more Towers may experience runtime difficulties.

We may be able to incrementally update certain data structures in order to reduce complexity and increase performance.

H. Other

The project would be well suited to supply other types of game intelligence, such as pseudorandom map generators and Monster deployment intelligence.

VIII. CONCLUSION

Today's TD game architectures limit the genre's growth potential. Scripted attack patterns, predictable winning strategies, and lack of replay value each place ceilings on the TD game experience.

AIMTD seeks to remove these limitations by allowing developers to create intelligent Monsters. The project is an open-source initiative, driven by the idea that TD games can become far more than just a casual subgenre of real-time strategy.

The AIMTD API is designed for ease of adoption and implementation. The AI engine beneath the API uses graph structures in conjunction with various movement-cost heuristics in order to guide Monsters through a Map. AIMTD Monsters can display three distinct behavioral patterns, adding depth and richness to the game experience.

The project constantly evolves and finds new venues for growth. For the foreseeable future, focus will remain on growing the project's user base and gaining traction within the TD developer community. Eventually, AIMTD hopes to become a full-fledged framework for building a dynamic new breed of TD games.

REFERENCES

[1] "Rampart (video game)". http://en.wikipedia.org/wiki/Rampart_(arcade_game)/ (accessed April 21st, 2012). Wikipedia, April 2012.

[2] L. Mitchell. "Tower Defense: Bringing the genre back". http://palgn.com.au/11898/tower-defense-bringing-the-genre-back/ (accessed April 20th, 2012). Palgn, June 2008.

[3] Shock Arcade. *Shock Defence*. http://shockarcade.com/shock-defence-2/ (accessed April 20th, 2012). Shock Arcade, December 2011.

[4] Ironhide Game Studio. *Kingdom Rush*. http://kingdomrush.com/play.php/ (accessed April 20th, 2012). Ironhide Game Studio, 2011.

[5] GameInABottle.com. *Gemcraft Labyrinth*. http://www.gameinabottle.com/gcl.php/ (accessed April 20th, 2012). GameInABottle.com, 2011.

[6] Chillingo. *Anomaly Warzone Earth*. http://www.chillingo.com/games/anomaly-warzone-earth/ (accessed April 20th, 2012). Chillingo, August 2011.

[7] PopCap. *Plants Vs. Zombies*. http://www.popcap.com/games/plants-vs-zombies/ (accessed April 21st, 2012). Electronic Arts, 2012.

[8] G. N. Yannakakis. "How to Model and Augment Player Satisfaction: A Review." Proceedings of the 1st Workshop on Child, Computer and Interaction. Chania, Crete: ACM Press, October 2008.

[9] P. Avery, J. Togelius, E. Alistar, and R.P. van Leeuwen. "Computational Intelligence and Tower Defence Games." Proceedings of the 2011 IEEE Congress on Evolutionary Computation (CEC). IEEE Press, 2011.

[10] R. Hunicke and V. Chapman. "AI for dynamic difficulty adjustment in games." Proceedings of Artificial Intelligence and Interactive Digital Entertainment, 2004.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (Third Edition)*. Cambridge, MA: The MIT Press, 2009.

[12] P.H. Winston. *Artificial Intelligence (Third Edition).* Boston, MA: Addison Wesley, 1992.

[13] P. M. Avery, S. J. Louis, and B. Avery. "Evolving coordinated spatial tactics for autonomous agents using influence maps." Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Games. IEEE Press, 2009.

[14] D. Monderer and L.S. Shapley. "Potential Games". Journal of Economic Literature, January 1994. Academic Press, Inc. 1996.