# 24

# Being Where It Counts

## *Telling* Paragon *Bots Where to Go*

*Mieszko Zieliński*



### 24.1 Introduction

From an AI point of view, multiplayer online battle arena (MOBA) games have two distinct layers: the tactical layer, which is responsible for the individual behavior of the AI-controlled players (aka "bots"), and the strategic layer, which is responsible for the broader, team-level, strategic decision-making. The goal of the tactical layer is to handle moment-to-moment combat in a fashion that will not break the players' suspension of disbelief. Although this is visually appealing, it has a relatively small impact on the game's outcome (assuming a certain level of skill among the players). The strategic layer, on the other hand, has a much more important goal of providing the bots with their strategic, long-term goals. It is responsible for knowing where the enemies are, deducing what the opponent's next strategic step might be, and determining which defensive structures are in danger, which enemy structures to attack, who should attack them, and so on. This is the part of the AI that can win or lose the game and that can keep players engaged over the course of the entire match.

*Paragon* is the latest MOBA from Epic Games. This chapter will describe the approach taken while developing the first version of the strategic layer for *Paragon's* bots. It is a simple solution that can be applied to a wide range of games, but despite its simplicity, it has achieved powerful results.

## 24.2 Problem Description

From a strategic point of view, being in the right place at the right time is the key to success in a MOBA. Assuming that all players are more or less on the same skill level, the only way to win an encounter is to have an advantage in numbers or the element of surprise. You need to be able to concentrate your forces to attack or defend in the right places, at the right times, without leaving an opening that the enemy can exploit. Both are extremely difficult to achieve if every bot thinks for itself. This is not an AI-specific problem—in Player versus Player (PvP) games, the teams using voice communication have an enormous advantage over the teams that do not.

Any kind of cooperation is better than no cooperation. The simplest form of cooperation is "going together." If two teammates head down the same map route, they will help each other even if only by accident, splitting the danger in two, doubling the firepower. From this point of view, it should be enough, at least initially, to just make sure bots are where they are needed. If there are three enemy heroes attacking a bot team's tower and only one bot is defending it, for example, then the obvious course of action is to get some teammates over to boost the defense. Merely having bots show up in such a scenario is enough to create an illusion of cooperation—anything achieved on top of that is a bonus.

Before the systems described in this chapter were implemented, the *Paragon* bots already had their individual behaviors in place. They had a basic understanding of how their abilities should be used. They could use healing potions when low on health points. They had reasonable target selection policies. They also knew to run for their lives when in mortal danger. Although the bots' behavior was perceived as correct, the absence of cooperation between the bots lacked the strategy necessary to compete against humans.

In summary, the problem can be formulated in simple terms as *making sure bots go where they are needed*. Once they arrive, the tactical layer can take over and do the rest.

## 24.3 The Graph

The very first thing needed when deciding where to go is a knowledge of the places that one can go. Experienced human players have it easy: they look at the mini-map and get information about all important places at a glance. They see where their teammates are, where the minions are going, which towers are still alive, and so on. Tracking minions, the lesser AI-controller soldiers, is especially important since players need minions to be able to attack enemy towers. The AI needs the same information, but those data need to be somehow gathered or generated, which is done using specialized AI subsystems.

Every MOBA game map has a similar structure: There are two team bases at far ends of the map, and there are defensive structures (called *towers* in *Paragon*), which are arranged into chains, forming *lanes*. There is usually more than one lane, and in between the lanes, there is a *jungle* that hides additional interesting places. In *Paragon*, those include *experience (XP) wells* and *jungle creeps' camps*. XP wells are places where teams can place *harvesters* to extract and store XP. The stored XP needs to be retrieved on a regular basis by a player hero because harvesters have limited capacity. Jungle creeps' camps are places where heroes can kill neutral inhabitants of the jungle for experience and buffs (temporary stat improvements).

The strategic AI needs to know about all such places—both the towers and the contents of the jungle—because every one of them has some strategic significance and can become an objective for the AI. Using information regarding the important places on the map, we can build a graph for the AI to use as the game-world's abstraction. Since nodes represent potential objectives, we call it the *Objective Graph*.

Let us take a step back from *Paragon* for a second. Our approach for generating a graph from relevant locations on the map does not depend on *Paragon's* specifics and can be easily applied to nearly any game that deals with spatial reasoning. The algorithm will be described in the following subsections.

### 24.3.1 Nodes

Every important location (such as a tower or XP well in *Paragon*) should have a corresponding graph node. Nodes are packed nicely in an array. Every node has an index, a pointer to the related structure on the map, a location (which is polled from said structure), and some additional cached properties. Examples of these properties include the structure type, which team currently owns it, and in which lane (if any) the node lies.

In addition to structures, places of strategic importance like vantage points, spawn locations, good places for an ambush, and so on, are all good candidates for graph nodes. In short, any location that bots might be interested in for strategic reasons should be represented as a node in the graph.

### 24.3.2 Edges

The edges in the graph represent the game-world connections between the locations symbolized by nodes. A connection is created between two game-world locations if there is an AI navigable path between them. Building edge information can be done in two steps. The first step is to generate all possible connections so that every node has information about every other reachable node. A game-world pathfinding query from every node to every other node is performed; if the path is found, the path's length is stored on the edge. This information is referred to as the edge's *cost*. Connections need to be tested both ways because path-length information is going to be used in a meaningful way later, and for nontrivial maps, A-to-B and B-to-A paths will sometimes have different lengths.

In most games, there are no isolated locations. Given enough time, a player can reach every location on a map from any other location—otherwise, why include that location in the game at all? With that in mind, the resulting graph is a *complete digraph*, which means it has an edge between every pair of nodes in the graph. This graph may sound expensive to create, but for our maps, it took less than three seconds. In any case, the computation is done offline when the game is created, so the cost does not impact the player's experience.

The second step of the edge-building process is to prune unnecessary connections. By unnecessary, we mean edges that, when replaced with a combination of other edges, still sum up to a similar cost (within some configurable tolerance). Listing 24.1 describes the algorithm with pseudocode.

The `EdgeCostOffset` variable defines the tolerance that we allow in the combined cost of the edges that are replacing a single edge and is designer-configurable, which gives the map maker a degree of control over the edge generation process. The offset value should be ideally configurable for each map, as the values that work well on one map may

Listing 24.1. Pruning of graph edges.

```
Graph::PruneEdges(InNodes, InOutEdges)
{
  SortEdgesByCostDescending(InOutEdges)
  for Edge in InOutEdges:
    for Node in (InNodes - {Edge.Start, Edge.End}):
      if Edge.Cost >= InOutEdges[Edge.StartNode][Node].Cost
        + InOutEdges[Node][Edge.EndNode].Cost
        + EdgeCostOffset:
            Edge.IsPruned = true
            break
}
```

not be as good on another. Having a way to force connections (i.e., shield them from pruning), as well as a way to manually prune edges, can come in handy as well.

Note that in the algorithm presented, pruned edges are not being removed, just marked as pruned. This is an important point. As described in Section 24.4, in some cases, it is advantageous to use the pruned edges, whereas in others, it is not.

### 24.3.3 Closest Graph Node Lookup Grid

There is one more piece of information that we incorporate as part of the graph building process—a *closest graph node lookup grid*. This is a coarse grid covering the whole map that stores calculated information regarding the closest nodes. Since this information is calculated offline, in the editor or as part of some build process, it is not limited to simple-and-fast distance checks. Instead, full-featured path-length testing can be used to maximize the quality of the data stored. For every grid cell, an arbitrary location is picked (center is arbitrary enough), and paths are found to every node in the graph. Then a note is taken of the node closest in terms of path length. Optionally, the specific path-length value can be stored as well; there are a plenty of ways to prove that this kind of information is useful.

Note that the grid's resolution is arbitrary, but it has consequences—the smaller the cells, the higher the resolution of data, but it will take more time to build that information (a lesser problem), and it will take up more memory (a potential deal breaker).

## 24.4 Enemy Presence

One type of data that would be beneficial to associate with graph nodes, which is available only at runtime, is *enemy presence* or *influence* (Dill 2015). On the strategic level, where the graph lives, it does not really matter which hero is where exactly, or how many minions are left alive from which wave (minions move in waves). The relevant information is what the "combat potential" is or how "in danger" a given area is.

A simple runtime update step can be performed on the graph periodically. There should already be a way to query other game systems for information regarding the location and state of all heroes and all minion waves. This information can be used to build influence information with the algorithm shown in Listing 24.2.

Listing 24.2. Influence calculations.

```
Graph::UpdateInfluence(InAllHeroes, InAllMinionWaves)
{
  ResetInfluenceInformation()

  for Hero in InAllHeroes:
    Node = LookupClosestGraphNode(Hero.Location)
    Influence = CalculateHeroInfluence(Hero, 0)
    Node.ApplyInfluence(Influence)
    for Edge in Node.Edges:
      Influence = CalculateHeroInfluence(Hero, Edge.Cost)
      Edge.End.ApplyInfluence(Hero.Team, Influence)

  for Wave in InMinionWaves:
    Node = LookupClosestGraphNode(Wave.CenterLocation)
    Influence = CalculateWaveInfluence(Wave, 0)
    Node.ApplyInfluence(Influence)
    for Edge in Node.Edges:
      if Edge.EndNode.LaneID != Wave.LaneID:
        continue
      Influence = CalculateWaveInfluence(Wave, Edge.Cost)
      Edge.End.ApplyInfluence(Wave.Team, Influence)
}
```

Note that `LookupClosestGraphNode` takes advantage of the closest graph node lookup grid described in the previous section.

In essence, what the code in Listing 24.2 is doing is calculating the influence score for every relevant source and distributing it across the graph. This is one of the places where information from pruned graph edges is being used. `Cost` information is relevant regardless of whether an edge is pruned or not, and here it is being used as a reliable indication of game map distance.

Minions' influence is limited to lane nodes because minions are restricted to a specific lane. Minion influence should not seep through the jungle to another lane, since that would result in misinformation—minions are not allowed to change lanes.

## 24.5 Putting the Graph to Use

By now, it should be clear that the graph described in previous sections is the top level of a *hierarchical navigation graph*. It has nodes corresponding to actual locations in the game world. It has connections corresponding to paths in the game world. This means any path calculated in the graph can be translated into a regular navigation path in the game world.

Using the information regarding dangers on the map, strategically smart paths can be found. Pathfinding can be configured to find paths serving multiple purposes, like paths that avoid an enemy, paths that go through enemy-dense areas, paths that stick to lanes as much as possible, and so on. With this ability, there are a large number of possibilities for an AI programmer to explore.

Regular A* search (Buckland 2004) can be used to find paths along unpruned edges of the graph. Using unpruned edges promotes bots moving between interesting locations. The resulting paths are sequences of nodes that the AI should visit to reach its goals. If we do not want the AI to visit "interesting locations," then we can skip this or better yet apply a heuristic to the search which forces it to avoid "bad" nodes (such as those with high enemy influence). This ensures that we pick a smart path, and the graph node is small enough that the use of a heuristic will not make our searches overly expensive.

The bot's path-following code uses a graph path to generate consecutive navigation paths. When a bot following a graph path reaches a graph node on the path, it then searches a regular path to the next node on the path, repeating the process until the last node is reached. This saves us some path-finding performance cost, since finding a series of short paths is cheaper than finding one long path. Building the full path ahead of time would be a waste: the bot usually will not even utilize the whole hierarchical path. Often, it will get distracted by petty things like enemies or death.

## 24.6 The Objectives

Every node in the graph has some strategic importance. They might be places to attack, or to defend, or places where a bot can go to gather XP or wait in an ambush, for example. Thus, every node of the graph can have some kind of objective associated with it.

A *Bot Objective* in *Paragon* is a type of behavior modifier. When a bot is assigned to an objective, the objective tells it where to go and what to do when it gets there. An objective is usually associated with a structure on the game map and, by extension, with a node in the objective graph. An objective knows how to deal with the structure: for example, defend it, attack it, or stand on it to get XP. It also knows which agents would be best suited to deal with the structure and how many agents are required. An objective can also influence how agents are scored.

Not all objectives are equally important. An objective's priority is based on a number of factors. These include the objective's type (for example, defending is more important than attacking) and its location (e.g., defending the base is more important than defending a tower somewhere down the lane). It is also related to the structure's health and enemy presence in the area. The enemy presence is read directly from the Objective Graph, using the influence information described in Section 24.4.

There are also objective-specific factors influencing the objective's priority. The "experience well" objective is an interesting example in this regard. The longer an XP harvester is waiting at full capacity, the more XP potential gets wasted and the more important it is to retrieve XP from it. On the other hand, the higher the level of heroes on the team, the less important it is to acquire that XP.

### 24.6.1 Where do Objectives Come From

Every type of objective has its own generator, and all generators create instances of objectives before the match starts. These objectives then register themselves with their respective graph nodes, as well as with any game system they need notifications from. For example, a defend objective would register to be notified when its tower gets destroyed, and an XP harvesting objective would register to be notified when the XP harvester on a particular well is full. In all cases, these objectives register with the *AI Commander*.

## 24.7 The AI Commander

The AI Commander is responsible for assigning objectives to bots based on all the information described above. Objective assignment is done as a response to events in the game world. Each team is processed separately, and human-only teams get skipped altogether. The pool of available objectives is also separate for each team, which is pretty intuitive—bots should not know whether the other team is planning to attack a given tower that would be cheating! However, some of the objectives do care about aspects of the other team's actions or state. For example, the "experience well" objective has two modes—it is "regular" when the well is unoccupied, or owned by the objective's team, and it switches to "offensive" when the enemy team takes possession of the well by placing its own harvester on it. This kind of knowledge gathering is encapsulated inside each specific objective's logic.

The objective assignment process can be split into multiple steps. As a first step, each team's objectives have a chance to update their own state. Every objective can be in either a "Dormant" or "Available" state. It is up to each objective's internal logic to determine which state it is in. If an objective is dormant, then it is skipped by the AI Commander during the objective assignment process. As part of the update, objective priorities are calculated as well. As previously mentioned, the objective's priority is based on multiple factors, and every objective can also specify how much a given factor influences its score. For example, "experience well" objectives are not concerned with enemy influence, while tower defense objectives treat it very seriously. Once every objective is updated and scored, all the available objectives are sorted by priority.

In step two, each available objective filters and scores all of the teammates, and stores the results for use in step three. Filtering gives objectives a chance to exclude unfit agents on a case-by-case basis. For example, a special ability is required to place a harvester on an experience well, so that objective excludes heroes who do not have that ability. Agent scoring is described in greater detail in Section 24.7.1. It would actually be more efficient to do the scoring and filtering as part of step three, but splitting the process this way makes it easier to explain.

The third step of the objectives assignment process is responsible for assigning agents to objectives. We do this by iterating over the following loop until all agents have been assigned:

1. Find the highest priority objective.
2. Allow that objective to pick a minimum set of resources (e.g., a single Brawler, or a Support and Caster pair).
3. Reduce the priority of that objective according to the amount of resources it took. The amount of priority reduction per agent assigned is in relation to all active objectives' max priority. It means that with every agent assigned, an objective will lose MaxPriority/TeamSize priority.

One potential tweak some adopters might want to consider is to use different limits to how many resources an objective is allowed to take. Allowing only the minimum amount will result in carrying out as many objectives as possible, but for some games, it would make more sense to focus all the power on just one or two objectives.

It is important to note that the result of this process depends heavily on how the objectives' priority is calculated and on how the individual objectives calculate their agent scores. Tweaking those two elements is necessary for good results.

### 24.7.1 Agent Scoring

There are two ways the agents' scoring can be performed by an objective. One is fully custom scoring, where the objective is the only scoring authority and calculates the scores itself. The other way, the default method, allows an objective to specify a hero role preference (Support, Tank, Caster, etc., as a single hero can have multiple roles). The preference is expressed as a set of multipliers for every role, and the score is based on the best role that the agent has. For example, if a hero is a Tank and a Brawler, and the objective has 0.1 preference for Tanks but 0.3 for Brawlers, then that given hero's score will be 0.3.

Regardless of the method, the agents' scores are calculated per objective. This makes it easier to tweak the results of the whole objective assignment process since potential mistakes in agent scoring will be localized within individual objectives and will have limited effect on which objectives are being carried out.

Distance (or travel time, if you have agents that move at different speeds) is another component of agent score. It usually makes most sense to assign a medium-scoring agent that is close to an objective rather than picking the highest scoring one that is on the other side of the map. Calculating a reliable distance-based portion of the score is really fast, since that information is already available within the Objective Graph; it already knows the path cost for every node pair in the graph! The path distance is retrieved from the graph and multiplied by a factor supplied by the objective.

Using one fun trick, a distance to the objective can be calculated even for dead heroes. To do this, we have to use travel time (that is, how long will it take the unit to travel to the objective) rather than distance. We then treat dead heroes as standing in their team's base but needing an additional X seconds (X being the time left to respawn) to arrive at the destination. This way, if one additional agent is needed to defend the base, then from two similarly adequate bots, the one that respawns in 10 seconds will be picked over the one that would need 20 seconds to get there.

### 24.7.2 Humans

The AI Commander treats human players in mixed human-bot teams just like other agents. The objective assignment code makes no exception for human agents. An assumption is made that the AI Commander will pick a reasonable objective for every agent, including players. If so, a player can be expected to carry out the assigned objective without being directly controlled by the game AI. Moreover, the on-screen team communication system can be used to request players to do something specific, like "defend left" or "group up!" (just like human players can do in regular PvP games). In any case, both the game and AI Commander are flexible in this regard, so even if an agent does not fulfill its role, the game will not break, and adjustments will be made during the next objective assigning iteration.

An interesting possible extension should be pointed out here. Since human players can use team communication, we could include messages from them as hints and temporarily increase the priority of the objectives associated with received messages. We cannot have human players control which objectives the AI Commander picks directly, since that

would have a potential of ruining the experience by sending the AI to all the wrong places, but we can use it to influence the decisions in a weaker way.

### 24.7.3 Opportunistic Objectives

When a bot carries out an objective, it usually involves following an objective graph path. The path-finding algorithm can be configured to prefer graph nodes containing unassigned objectives. Then as a bot progresses through the graph path on every node, a check can be done to see if there is an unassigned objective that said bot can carry out. The objective gets a chance to specify if it is interested in being assigned this way. For some objective types, it simply does not make sense to be picked up "on the way." Good opportunistic objectives should be easy and quick to carry out, for example, gathering XP from wells or destroying a jungle creeps' camp.

## 24.8 Future Work

The ideas described in this chapter have a lot more potential uses. The Objective Graph is a convenient abstraction of the map, and more data can be associated with every node. Below are some examples.

### 24.8.1 Probabilistic "Presence" Propagation

As of this writing, the *Paragon* AI Commander has perfect knowledge of all heroes' locations. Human players only know about enemy heroes that they have seen themselves or that have been sighted by the team. Although this is not completely fair, it helps compensate for other shortcomings (such as not being able to synchronize ability usage with their teammates).

It would be possible to build a probabilistic net of locations of all heroes based on the objective graph. If an enemy hero is visible to any of the team's heroes or minions, the graph node closest to the enemy's location gets annotated with information that the given hero is at that node with probability of 1. If a hero is not visible, then that hero's last known location is used; based on the time of last sighting, propagate information to all neighbors of the last known location node. The probability of a hero being at any other graph node is proportional to the distance (read directly from the graph nodes) and the time passed; it may also be influenced by the knowledge of currently hot locations on the map and where a hero could be interested in being. An information diffusion step could also be added to the process. This whole idea is a variation of Occupancy Maps (Isla 2006).

### 24.8.2 Map Evaluation

One of the few reliable use cases of neural networks (NNs) in game AI is data classification. It is possible to imagine a NN that would take the "world state" as input and generate a sort of "quality" or "desirability" value as output. Having a map abstraction such as a graph already at hand makes converting the world state into a vector of values into a relatively straightforward process. Data from games played online by human players can be used to construct a training set. Grabbing graph snapshots at regular intervals and associating them with the final result of the match would be a good start. Once trained, the net could be used to help the AI Commander to evaluate current game's state and guide high-level strategy, such as by hinting whether the current world state requires a more defensive or a more offensive stance.

## 24.9 Conclusion

As proven by our internal user experience tests, the introduction of a strategy layer to the *Paragon* bot AI greatly improved players' experiences. The game did not instantly become harder, because no behavioral changes have been made to the bots' individual behaviors, but users did notice the game being more interesting. Bots started showing signs of a deeper strategic understanding of the game: filling in for fallen comrades, switching lanes, attacking enemy harvesters, and even *ganking* (which is MOBA-speak for *ambushing*), although the latter behavior was entirely emergent. The system telling the bots where to go was simple but competent. Players will generate their own explanations for what is going on in the game as long as the AI is doing its job well enough!

A graph representation, due to its discrete nature, is an easy way to represent complex data like a level filled with gameplay. Pathfinding over long distances is a breeze. Estimating enemy danger at a node location is a simple lookup operation (provided regular influence updates are performed). Last but not least, the Objective Graph gives spatial context to the otherwise abstract concept of objectives. This is just a start; there is so much more that could be done with a graph abstraction of the game map. There is no (good) excuse not to give it a try and build one of your own!

## References

Buckland, M. 2004. *Programming Game AI by Example*. Jones & Bartlett Learning.

Dill, K. 2015. Spatial reasoning for strategic decision making. In *Game AI Pro 2: Collected Wisdom of AI Professionals*, ed. S. Rabin. Boca Raton, FL: A. K. Peters/CRC Press.

Isla, D. 2006. Probabilistic target tracking and search using occupancy maps. In *AI Game Programming Wisdom 3*, ed. S. Rabin. Hingham, MA: Charles River Media, pp. 379–388.