5-2019

Video Game AI Algorithms

Bowen Li
Trinity University, bli4personal@gmail.com

Video Game AI Algorithms

Bowen Li

A departmental senior thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for graduation with departmental honors.

4/22/2019

_____	_____
Thesis Advisor	Department Chair

_____

Michael Soto, AVPAA

Student Agreement

I grant Trinity University (“Institution”), my academic department (“Department”), and the Texas Digital Library ("TDL") the non-exclusive rights to copy, display, perform, distribute and publish the content I submit to this repository (hereafter called "Work") and to make the Work available in any format in perpetuity as part of a TDL, Digital Preservation Network (“DPN”), Institution or Department repository communication or distribution effort.

I understand that once the Work is submitted, a bibliographic citation to the Work can remain visible in perpetuity, even if the Work is updated or removed.

I understand that the Work's copyright owner(s) will continue to own copyright outside these non-exclusive granted rights.

I warrant that:

1) I am the copyright owner of the Work, or
2) I am one of the copyright owners and have permission from the other owners to submit the Work, or
3) My Institution or Department is the copyright owner and I have permission to submit the Work, or
4) Another party is the copyright owner and I have permission to submit the Work.

Based on this, I further warrant to my knowledge:

1) The Work does not infringe any copyright, patent, or trade secrets of any third party,
2) The Work does not contain any libelous matter, nor invade the privacy of any person or third party, and
3) That no right in the Work has been sold, mortgaged, or otherwise disposed of, and is free from all claims.

I agree to hold TDL, DPN, Institution, Department, and their agents harmless for any liability arising from any breach of the above warranties or any claim of intellectual property infringement arising from the exercise of these non-exclusive granted rights.”

I choose the following option for sharing my thesis (required):

[X] Open Access (full-text discoverable via search engines)
[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

I choose to append the following Creative Commons license (optional):

Video Game AI Algorithms

Bowen Li

Abstract

The ubiquity of human-like characters in video games presents the challenge of implementing human-like behaviors. To address the pathfinding and behavior selection problems faced in a real project, we came up with two improved methods based upon mainstream solutions. To make pathfinding agent take into account more incentives than only a destination, We designed a new pathfinding algorithm named Cost Radiation A* (CRA*), based on the A* heuristic search algorithm. CRA* incorporates the agent's preference for other objects, represented as cost radiators in our scheme. We also want to enable non-player characters (NPCs) to learn in real-time in response to a player's actions. We adopt the behavior tree framework, and design a new composite node for it, named learner node, which enables developers to design learning behaviors. The learner node achieves basic reinforcement learning but is also open to more sophisticated use.

Acknowledgments

I would like to thank my parents first, not only for supporting me financially but also supporting all the decisions I made with my ever-changing mind, despite the disagreement we had. I would like to thank my friends and family members for their generous help in my life.

I would like to thank Dr. Matthew A. Hibbs, Dr. Yu Zhang, and Dr. Albert X. Jiang from the Computer Science Department. I would like to thank Dr. Hibbs for all the help I received from him in all aspects throughout my college years. Besides being my academic advisor and thesis instructor, Dr. Hibbs has provided insightful advice with respect to my research and career choice. I also developed my interest in video gaming in his upper-division classes, the experience of which contributes to my thesis. I would like to thank Dr. Yu Zhang and Dr. Albert X. Jiang for serving on my thesis committee. I would also like to thank Dr. Zhang for being my summer research instructor in my early research in deep learning.

I would like to thank Dr. John H. Huston and Dr. Nels P. Christiansen from the Economics Department. In their classes, I developed life-long curiosity and passion for human intelligence and behaviors, which have also largely affected my research interest.

With respect to my summer research, I would like to thank Murchison Research Fellowships for providing the opportunity and funding my research. Finally, I want to thank Trinity University for financially supporting my college education.

Video Game AI Algorithms

Bowen Li

Contents

List of Figures

Chapter 1

Introduction

Artificial Intelligence in video games is a very special branch of AI, to the extent that someone would argue that it does not counts as one [9]. Video game developers very often use models and techniques that are or used to be popular among AI scholars, but often with radically different goals and methodologies. Sections of this chapter are dedicated to providing background knowledge for video game AI.

In Section 1.1 and Section 1.2 we talk about what is video game AI, and why and how video game AI is different from traditional AI. In Section 1.3 and Section 1.4 we introduce behavior selection and pathfinding in video games. In Section 1.5 we introduce our research goals and contributions in the context of video game AI.

1.1 What is Video Game AI

Art imitates life, and so do video games. Over the short history of video games, the passion for faithful imitation of life has never ceased or declined. The ever-growing graphic fidelity and demand for it is one aspect, but it doesn't stand up alone to convince players.

Developers also need to make the logic that runs video game worlds also look real. This is where Artificial Intelligence is introduced. The use of the term "Artificial Intelligence" is controversial, for reasons that we will talk about in Section 1.2. Here we explicitly limit the use to within the context of video games.

Video games are populated with human and human-like NPCs (non-player characters). The presence of them brings up a challenge that, for the video game world to hold together, the behaviors of those NPCs need to appear intelligent enough to resemble human behaviors.

Another advantage of AI is that, a general-purpose decision-making framework opens up the possibility for sophisticated and interesting game design. With respect to the ubiquity of both demands, video game developers always want better ways to model an intelligent decision-making process. Video game developers call it an AI problem.

In the field of video game, sometimes other systems or mechanisms are also referred to as AI, such as a dynamic difficulty adjustment system [21]. But in fact, the system-level AI can be far less generalizable given that different games have different core mechanics. Here we narrow the discussion to NPC AI, the main functionality of which is to direct the behaviors of an individual agent. It is often referred to as a behavior selection problem.

1.2 Goals and Requirements of Video Game AI

It's important to note that, the use of the word "AI" here differs significantly from what is studied in academia, often related to machine learning. Academic AI study aims to achieve one single goal of creating intelligent agents: any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals [20]. The goal is pure and straight-forward, whereas video game developers face more engineering tradeoffs and are more flexible with "intelligence."

Here we talk about three important aspects that makes video game AI different.

1.2.1 Not Necessarily Intelligence

The distinct goal is due to the fact that video games are sold as entertainment products with carefully designed experiences. NPCs in games are either part of the story delivered to players, or challenges disguised in a form of character interaction. They need to be intelligent only to a certain extent and in a certain way, so that the storytelling feel reals and the challenges are just challenging enough.

In short, there is no need for real or advanced intelligence if the naive form serves the design goal. This partly explains why the video game industry is less excited with the rise of deep learning.

In some cases, the low standard can ironically become a requirement that stupidity in decision-making is needed. For sophisticated game design, AI agents might need to be capable of making human-like mistakes to either imitate the imperfection in our rationality or be vulnerable enough to players to make a reasonable difficulty level.

1.2.2 Controllability and Usability

As said, video games are sold as entertainment products, and therefore any undesired behaviors unthought of might cost that experience by causing disillusion or frustration. Game developers are very demanding with respect to the controllability of AI. Even when stochastic behaviors are expected, all possible permutations of actions must be consistent with the design goal. The more well-defined the behavior selection model is, the more likely it would be stable at runtime, and the better that model is considered.

However, players' preference for interesting stories and challenges motivates game designers to come up with sophisticated and complex AI behaviors, including the desired

stupidity mentioned earlier. Another important dimension for an AI model is then usability, as how easily developers can design complex behaviors with it.

1.2.3 Efficiency

Video games often aim to be sold to a wide range of players, and therefore have to be friendly for relatively low-end hardware. If by the design of a game, its AI agents are supposed to make decisions in real time, which is often desired in order for the game to be fun, the decision-making algorithm needs to be computationally friendly for the hardware used by most target players. As long as it runs frequently in real time, it faces the baseline requirement of not lowering the target frame rate.

To be more explicit, if the decision-making process is to be run at every moment when there is a change in the game world, it should finish within at most 1/30 second (30 FPS is often the bottom line for video games to feel smooth enough), and ideally within 1/100 second at a high standard of 60 FPS to leave reasonable time for other routines that runs every frame. Considering the possibility of multiple intelligent agents, we can never be too satisfied with efficiency performance.

1.3 Behavior Selection Algorithm

A behavior selection algorithm is the high-level framework that directs an AI agent to make choices between low-level actions. They are referred to as algorithms, but are closer to general-purpose frameworks with which a designer can describe any decision-making process.

There are two major types: state-based frameworks and utility-based frameworks [7].

The state-based frameworks (including finite-state machines and behavior trees) are

highly descriptive and logical in that they explicitly define the condition where an action happens. Designers have complete control over what an agent does. On the other hand, they inevitably involve a lot of human labor to figure out the logic chain that makes the most sense.

The utility-based frameworks direct behaviors at a lower level. They often define only the most basic actions and rewards, and let agents figure out what is best for them. The advantage is that its empirical approach often generates pretty rational behaviors without a lot of design wisdom. It closely resembles reinforcement learning [25], which is a popular solution to behavioral problems in mainstream AI academia.

It is hard to win the advantages of both approaches. For most video games, where experience is pre-designed (good counter examples include the Sims series), we believe that controllability is more important. In this project, when we desire some learning ability, we choose to borrow ideas from utility-based approach under the state-based framework.

1.4 Pathfinding

Pathfinding is an AI problem where an agent tries to find the optimal path from one location to another. It deserves special attention within a behavior selection framework because it is highly mathematical, generalizable, and often has a lot of overhead to be optimized.

Unlike the highly continuous real world with infinitely many locations, the representation of a space in video games faces the limitations such as memory use. A common approach is to summarize the space as a graph which consists of a finite collection of traversable nodes. The pathfinding problem is then conveniently transformed into a graph search problem.

The most fundamental algorithm in this field is A* [10], a heuristic search algorithm that guarantees to find the shortest path between two locations in considerably short runtime

with good heuristic. Most popular shortest-path algorithms are variants of A*, because the heuristic approach is unbeatable most of the time in terms of efficiency.

However, the shortest-path approach is actually not how human-beings do pathfinding in real life. Most importantly, there are more factors to take into account than the distance between two locations, such as safety and easiness. In this paper, we call them secondary incentives, with closing up the distance to a destination being the primary incentive.

1.5 Research Goals and Contributions

The goals of this research emerge from the development of a real video game project. We encountered two AI problems to which the existing solutions are either imperfect or irrelevant in one way or the other:

1) We want an AI navigation scheme that incorporates secondary spatial information of reward and cost.

2) We want an AI behavior selection framework with real-time learning ability.

For the first problem, we design an auxiliary feature for waypoint graph, called cost radiation, with which waypoint-based navigation can utilize secondary information.

For the second problem, we design a learner node for behavior trees that enables agents to learn from past experience.

They will be discussed later in sections of methods.

1.5.1 Optimal Path Planning with Secondary Incentives

We at first envisioned an AI agent that is capable of keeping a shortest path to a moving target in a 2D environment. There exists quite a pathfinding few algorithms, mainly A* variants that address different aspects of the shortest-path problem. But in a video game

scenario, there might also be secondary incentives, such as rewards or risks, present in the environment in addition to the distance cost of traversing. If the agent is really to be intelligent (or to look intelligent), it should maximize utility by considering all kinds of rewards and risks. Then the shortest path might not be the optimal path.

For example, when an agent attempts to navigate to a location, there might be threatening enemies that the agent may like to stay away from. To a human player, there are now some secondary incentives affecting how the player plans, and would go for a longer but safer path. Most pathfinding algorithms only take into account the position of the target and the traversable space. If they ever attempt to incorporate secondary incentives, the common practice is to treat the presence of enemies as a boolean property that triggers different behaviors, instead of one of many variables that the path is calculated with.

We try to find a better way to incorporate secondary spatial incentives so that the agent acts more like a human and intelligently. Our contribution is a variant of the A* algorithm that takes into account the distance to secondary incentives, based on a special waypoint graph implementation.

As the scenario is a common one is modern video games, we believe that a solution to the problem described would be highly generalizable to most current video games with a sophisticated navigation demand.

1.5.2 Behavior Selection with Real-Time Learning

We envisioned a general behavior selection framework that supports in-game real-time learning. A usage scenario might be where an NPC is supposed to be capable of reacting to player's strategy, or where an NPC reacts to other NPCs' behaviors in a way not previously designed. The goal is always to make games more interesting.

It might seem that deep reinforcement learning will be the new approach to tackle the

problem due to their power in learning complex behaviors. However, it turns out not to be an ideal solution with respect to the requirements talked about in 1.3, which we will discuss it in more detail in the related works sections below.

The requirements of controllability and stability creates a dilemma in contrast with the unpredictability of learning processes, which partly accounts for why traditional reinforcement learning approaches generally aren't the best fit. Essentially, we want to find a highly predictable learning approach.

Therefore instead we aim to adopt a rather flexible framework that enables game designers to design learning behaviors on a fairly high level so that the behaviors that can be chosen from are already well-defined and sensible.

We chose behavior trees [5] as our framework, and designed a new node with learning features for behavior trees that makes very basic reinforcement learning possible.

Besides the fact that behavior trees are one of the most widely used frameworks in industry [13], the framework works with arbitrary granularity of actions to be learned. Therefore, one can easily control the learning between high-level actions, such as "approach the target in an optimal path" or "perform an attack combo," and avoid the common problem of arbitrary combinations of low-level actions arising from reinforcement learning, such as "bring up the sword and put it back."

Chapter 2

Optimal Path Planning with Secondary Incentives

2.1 Related Works

2.1.1 Space Representation

As in many video games, our search space is a continuous 2D map of arbitrary shape. The most common representations include waypoint graphs, grids, and navigation meshes. We will use the hypothetical map shown in Figure 2.1 as an example to explain these ideas. The figures used in this chapter are credited to [4].

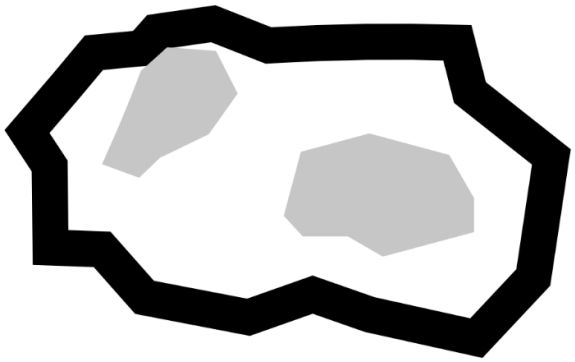

Figure 2.1: A map to be represent.

Waypoint Graph

Waypoint graphs, probably the most traditional representation of 2D space, represents the space as a finite collection of traversable locations. An agent can traverse from a waypoint (or node) to neighbors of the waypoint. The cost to traverse from one node to another can be calculated as the distance between them, or assigned with arbitrary rules. Graph-based traversal is one of the most common search problems in AI, and therefore leaves us with a lot of algorithms to choose from.

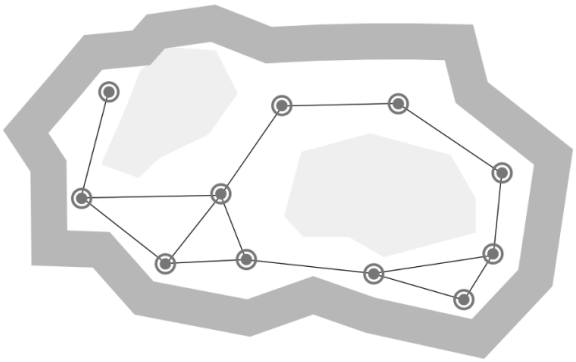

Figure 2.2: A waypoint graph representation.

Grid

The spatial distribution of waypoints can be quite arbitrary. In practice, it is often implemented as a grid. A grid assumes that a node has 4 neighbors (vertically and horizontally adjacent nodes) or 8 neighbors (including diagonally adjacent nodes), and it keeps a separate cost matrix where cost is assigned to each cell, instead of each traversal from one cell to another. With a grid, the cost from node A to node B cannot be assigned arbitrarily. It is either the cost of node B when they are adjacent, or the cost of the lowest-cost path from A to B when they are not adjacent.

Grids have some nice properties. 1) Nodes line up vertically and horizontally which is friendly for algorithms and data structures. 2) Assuming regular spatial distribution makes distance calculations easy. 3) Thanks to property 2, it is easier to work with a secondary cost system. In Figure 2.3, each traversable cell has a cost of 1.

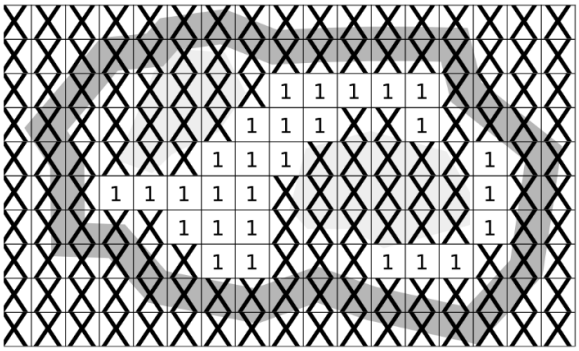

Figure 2.3: A grid representation.

Navigation Mesh

A navigation mesh [23] represents the space as a collection of triangles (or convex polygons with a different implementation). Each triangle is represented with three location vectors on

vertices. Its simplicity and efficiency are apparent when compared to waypoint graphs, and it is a great representation in many aspects. However, it is inappropriate for our purpose.

In theory, with the same generative precision, a navigation mesh is simply a subset of a waypoint graph, keeping only the waypoints on vertices of triangles. It includes the minimum information a shortest-path algorithm needs to know, because of the property that the shortest path (a sequence of nodes) can only consist of triangle vertices other than the given start and target locations [23]. Its simplicity is due to correctly ignoring all the information unnecessary to finding the shortest path.

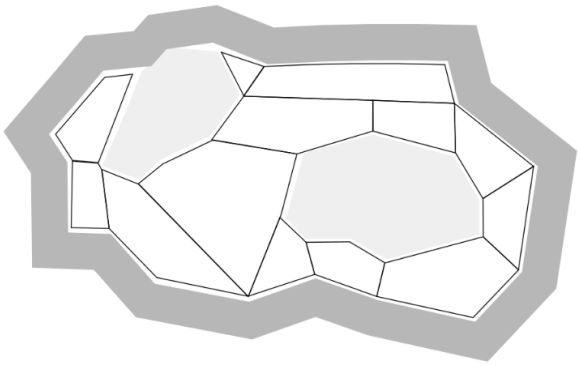

Figure 2.4: A navigation mesh representation.

Its efficiency in terms of memory use and runtime allows navigation meshes to be generated at a higher precision. Or from another perspective, significantly more nodes are required for higher precision for waypoint graph, but less so for navigation meshes.

2.1.2 Attepts to Include Secondary Incentives

So far, navigation mesh seems without doubt the representation to use for video games. However, when more incentives than the cost of distance are present, the optimal path might deviate from the shortest path, and breaks the nice property that ensures that navigation

meshes include the desired path.

The Unity game development engine [26] provides a feature to include some secondary incentives in pathfinding. Their navigation mesh system allows users to tag polygons with different cost. A navigation agent therefore would prefer paths through lowest cost polygons over others. The cost tagging practice provides some intelligence to the extent of choosing a better polygon to traverse through. But as long as we are to keep the navigation mesh simple with large polygons as it usually is, we cannot make distinctions between positions within the polygon that are too large to be treated uniformly.

The problem can be relieved if the navigation mesh is generated at a finer granularity, so that the agent can tell a difference between sub-polygons and make its path through the preferred one. But as the polygon size becomes small enough to make a local difference, this becomes more inefficient that it only resembles an awkwardly generated waypoint graph.

Similarly, in robotics research, scholars refer to this issue as a "weighted region problem" [24]. They attempt to divide the traversable area into regions with different manually assigned costs to find a path that makes more sense than the shortest path. The approach is similar to Unity's cost-tagging and therefore shares the same problem: when the discrete division preserves more information, it leaves daunting work for users to do.

A closer look at the problem reveals that, within the navigation mesh, people can only treat the secondary incentives as some discrete properties of regions, due to the fact that every point within a polygon is considered equivalently preferred. But in a real-life sense, the distance to an incentive does matter as a continuous variable.

2.1.3 Pathfinding Algorithms

Since grids and navigation meshes are only variants of waypoint graphs, most graph search algorithms work for all three representations. That is to say, under any of the three, most

A* variants will be available to us.

Many A* variants are created to tackle different aspects of the pathfinding problem, and their advantages are rarely incompatible with each other. With proper implementation, one can come up with a compound algorithm that is capable of performing a compound task. Most of them are optimizations targeted at specific limitations of A* or what has been ignored by A*.

We will introduce A* in details in the methods section as it is the basis of our variant algorithm. We do not dive into the discussion of those A variants, because our target solution should be compatible with most of the common variants. Useful algorithms that are worth attention with respect to our problem include LPA.[15]*, D* Lite[14], Theta*[6], HPA*[2], etc .

In our project, we experient with A* due to the small number of AI agents and small graph that we have in our testing examples. But it appears that we do face some efficiency bottleneck that can be relieved by incorporating some A* solutions. Developers are encouraged to make more efficient adaptions to tackle their unique problems.

2.2 Methods

In this section we introduce cost radiation as a feature of a waypoint graph. We then introduce the corresponding pathfinding algorithm that finds an optimal path with secondary incentives present, and show how and why it works.

2.2.1 Search Space

Based on the reasons talked about in Section 2.1, we consider grid a better representation with respect to our demands. The grid to be searched is programmatically generated with

a loop over an N by M area with step size i. At each step, the program checks colliders at that location. If no collision is detected, a node is created for that very location. The agent is to find an optimal path given start and end nodes within the grid.

2.2.2 A* Algorithm

Since our algorithm is modified from A*[10], here we introduce the algorithm and briefly explain how it works.

A* is based on a graph of traversable nodes. The cost to traverse from one node to another adjacent node is known, but the cost to travel to a non-adjacent node can have many possibilities, as there can be more than one paths. The job of the algorithm is to figure out, given a start node and a goal node, which path has the lowest cost. The algorithm would return the shortest path as an array of nodes, or null if no path can be found between the two nodes.

The difficulty of finding a shortest path comes from the huge number of permutations of nodes as candidates for shortest path. A* is a heuristic search algorithm, which makes it more efficient than a brute-force search. The idea is that, with a good heuristic, A* is able to make good guesses on which node is likely to be a node on a shortest path. Therefore it doesn't waste time on checking nodes that are less hopeful.

In the A* algorithm, each node has a g cost and a h cost. They together represent how likely a node is to be on the shortest path. The sum of g and h is denoted as f.

g is the cost of traveling from *start* to that node along the shortest path found so far. g is updated as the agent finds a better path to reach the node. When the graph is fully explored, g is the lowest possible cost to travel from the start to a given node. g can be overestimated prior to completion of enough exploration.

h is the heuristic cost, evaluated by the heuristic function provided to the algorithm.

It represents a guess of the cost of traveling from a node to the goal. The actual cost is unknown until the goal is reached, but a close enough heuristic guess guides the algorithm to explore more hopeful nodes first.

A* is only guaranteed to find the shortest path if the heuristic is admissible [11], requiring that it never overestimates the cost of traveling to the goal. A common heuristic is the straight-line distance between nodes when costs only depend on distance.

The algorithm keeps a closed set and an open set, where it puts nodes already evaluated, and nodes newly reached and not evaluated yet. In implementation, it is efficient to use priority queue or ordered set for the open set, to save the runtime of finding the node with lowest f in it.

The key step of the A* algorithm is where a new g cost is assigned for a node. At this step, a neighbor's g cost is replaced by the current node's g cost plus the cost to travel from *current* to *neighbor*. It means either that the node is visited for the first time since its initial g is infinity, or that the shortest path which its original g represents is found to be worse than the new path with the current node as its parent. And the shortest path leading to it is updated. This operation guarantees that when a node is in the open set, its g cost always represents the shortest path A* has found yet. And if it is the first element in an ordered open set, it is the shortest possible path on the leaf. When it is found to be the goal node, a shortest path to the goal is found.

2.2.3 Cost Radiation

We designed a new mechanism based on the basic grid implementation and A* pathfinding, which we call cost radiation.

A cost radiator represents a secondary incentive, either a threat with positive cost value, or a reward with negative cost value, and it and can be static or dynamic.

Algorithm 1 A* Pathfinding

```
function A_Star(start, goal, heuristic())
    closed = {};
    open = {};
    for each node n do
        node.g = ∞;
        node.h = heuristic(node)
        node.f = ∞
    end for
    start.g = 0
    open.add(start)
    while open is not empty do
        current = node in open with lowest f
        open.remove(current)
        closed.add(current)
        if current == goal then
            RecoverPath()
        else
            for neighbor ∈ current.neighbors do
                if n ∈ closed then
                    continue
                new_g = current.g + cost(current, neighbor)
                if new_g < neighbor.g then
                    neighbor.g = new_g
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current
                if n ∉ open then
                    open.add(n)
            end for
    end while
end function

function RecoverPath(start, goal)
    Initialize path = {}
    path.add(goal)
    current = goal
    while current ≠ start do
        current = current.parent
        path.add(current)
    end while
    path = reverse(path)
    Return path
end function
```

The effect of a cost radiator is imposing additional cost to its surrounding nodes. The cost is to be added to a node's g cost. Whether it should be added to h is discussed in the next section with respect to repellers (radiator with positive cost) and attractors (radiator with negative cost). As its name suggests, we designed the effect to be handled in a radiating manner, so that the farther away a node is from the radiator, the cost it gains from it would be discounted more heavily. We call the cost assigned base cost and the cost received effective cost.

The cost distribution incentivizes an pathfinding agent to stay away from a repeller and to stay close to an attractor. It's very important to notice that, **repellers and attractors are not theoretically equivalent**, although they seem to be. Some implications with respect to the heuristic function will be discussed shortly in next section.

A node can accumulate effective costs from multiple radiators, so that a node within range of multiple repellers (radiators with positive cost) is even less favored. We denote the total effective cost as r for radiation.

The formula for effective cost can be arbitrary. With some experiments, we choose the discounting coefficient to be the inverse of the distance between the node and a radiator plus 1. The +1 term is to make sure that when a radiator locates exactly at a node, the node gets exactly full effect of the radiator's base cost.

Algorithm 2 Total Effective Radiation Cost

Require: $r(n)$: total effective radiation cost for node n
  Initilize $r(n) = 0$
for each: radiator i in range
  $dist = distance(i, n)$
  $r_i(n) = base_i/(1 + dist)$
  $r(n)$ += $r_i(n)$

2.2.4 Repeller vs. Attractor

A* is guaranteed to find the shortest path if the heuristic cost of a node is always equal to or smaller than the actual cost between it and the goal. For this specific reason, a repeller is not theoretically equivalent to an attractor. A repeller is often safer to use than repeller.

A repeller makes the g costs of its surrounding nodes larger, and an attractor makes them smaller. When using common admissible heuristics such as straight-line distance, a repeller does not break the admissibility because the all actual costs are simply even larger than the heuristic cost. However, an attractor might bring down the actual costs by imposing negative r cost so that the actual cost can be smaller than the heuristic cost. In our project, the straight-line heuristic is one of those affected by attractor.

A solution is to also add r to h as well so that g and h increase by the same amount. Notice, when calculating $f = g+h$, the radiation cost is added twice. Further, it is also good practice for attractor because the additional r cost in g makes h less close to g and results in bad predictions. The performance results and some other interesting consequences will be discussed in detail in the results sections below.

2.2.5 Cost Radiation A*

We call our algorithm Cost Radiation A*. Based on the discussion we make above, we choose to also incorporate the radiation cost r into heuristic cost h. The resulting algorithm is Algorithm 3.

Algorithm 3 Cost Radiation A*

```
function A_Star(start, goal, heuristic())
    closed = {};
    open = {};
    for each node n do
        node.g = ∞;
        node.h = heuristic(node)
        node.f = ∞
    end for
    start.g = start.r
    open.add(start)
    while open is not empty do
        current = node in open with lowest f
        open.remove(current)
        closed.add(current)
        if current == goal then
            RecoverPath()
        else
            for neighbor ∈ current.neighbors do
                if n ∈ closed then
                    continue
                new_g = current.g + cost(current, neighbor) + neighbor.r
                if new_g < neighbor.g then
                    neighbor.g = new_g
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current
                if n ∉ open then
                    open.add(n)
            end for
    end while
end function
```

2.3 Results

2.3.1 Behaviors

The behavior pattern for pathfinding with repellers matches what we expected. The agent would find a longer path curved away from repellers. The shape of the curve depends on how strong the repellers are (magnitude of base cost). Figure 2.5, Figure 2.6, and Figure 2.7 show the optimal paths found with only different base cost and everything else equal. Figure 2.8, Figure 2.9, and Figure 2.17 show the optimal paths found with multiple repellers all with base cost of 100. We indicate the order or nodes in a path with little black arrows as shown in the figures. In all of the following figures, the start node is indicated by a red cube, the destination is shown as a smaller blue cube, and the dark blue cubes are radiators.

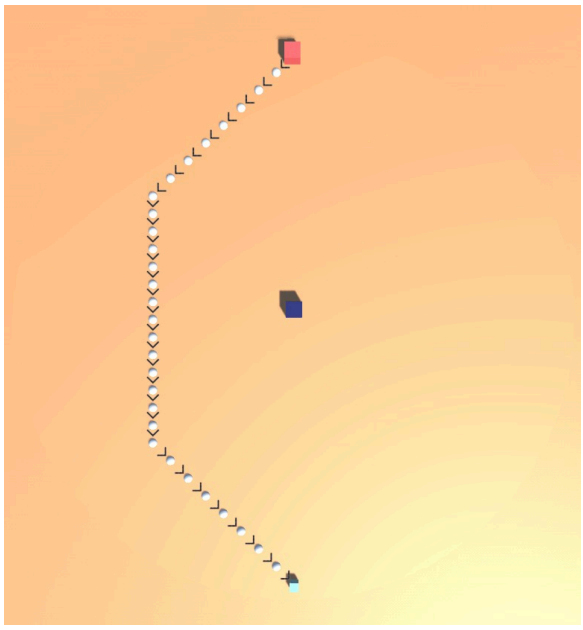

Figure 2.5: Repeller with base cost of 100.

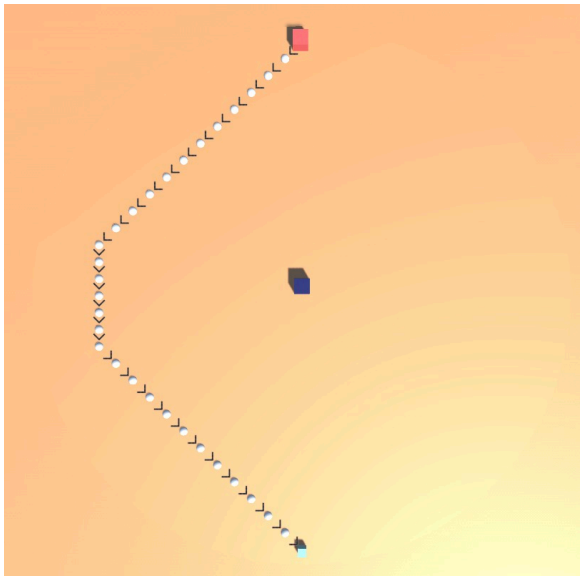

Figure 2.6: Repeller with base cost of 1000.

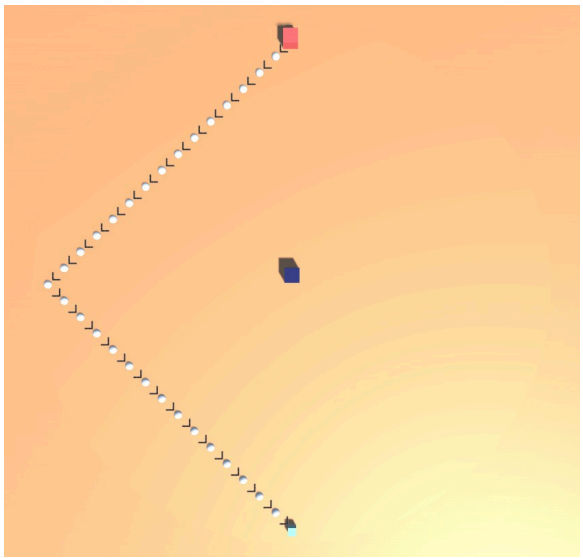

Figure 2.7: Repeller with base cost of 5000.

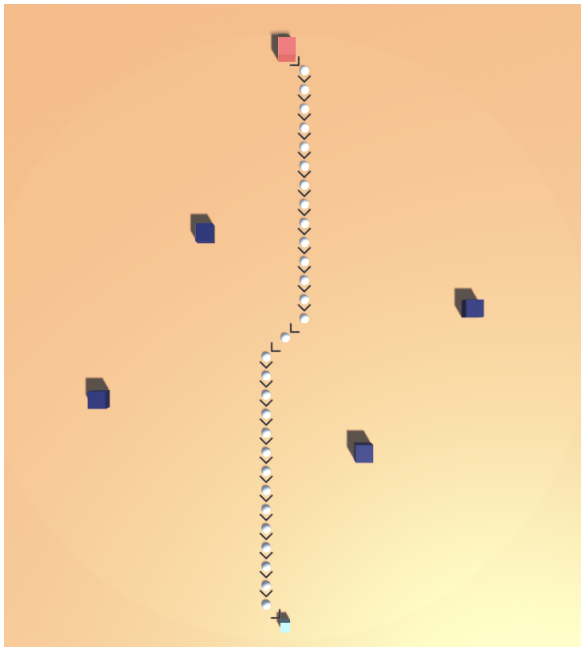

Figure 2.8: Repellers with base cost of 100.

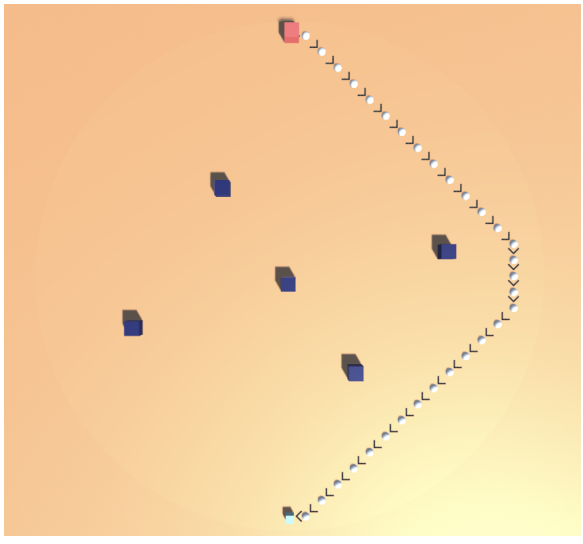

Figure 2.9: Repellers with base cost of 100.

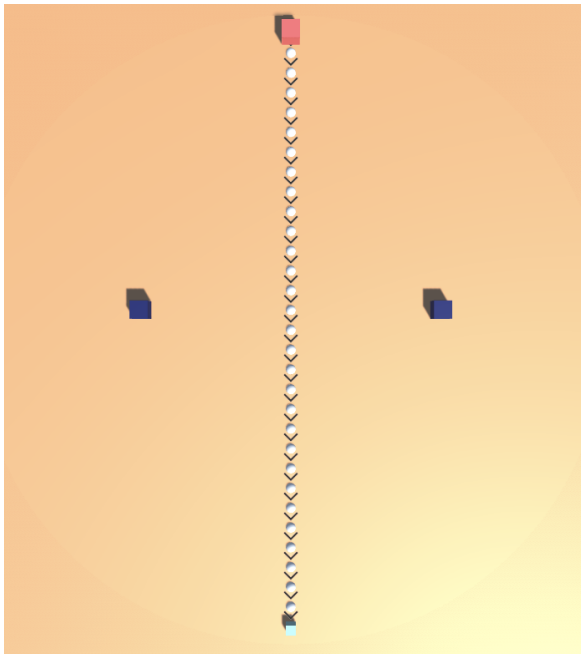

Figure 2.10: Repellers with base cost of 100.

The behavior pattern with attractors is quite surprising but also makes a lot of sense. In the case of a single attractor or closely located attractors, if the attractor is strong enough but not too strong, the path found by the agent would be a nice curve towards the attractors. However, if the attractor is so strong that the agent eventually reaches the attractor's position, the agent then would meander around the attractors for a while. Depending on magnitude of the base cost and spatial distribution of all objects, the path from attractor to goal can be of an S-shape towards the goal or a spiral shape around the attractor until the agent is far away enough, as if the agent is collecting rewards in that area.

Figure 2.13, Figure 2.11, and Figure 2.12 show the paths found with different initial locations but same base cost. Figure 2.14 shows the path found with the same location

setup as Figure 2.13 but 10 times of base cost.

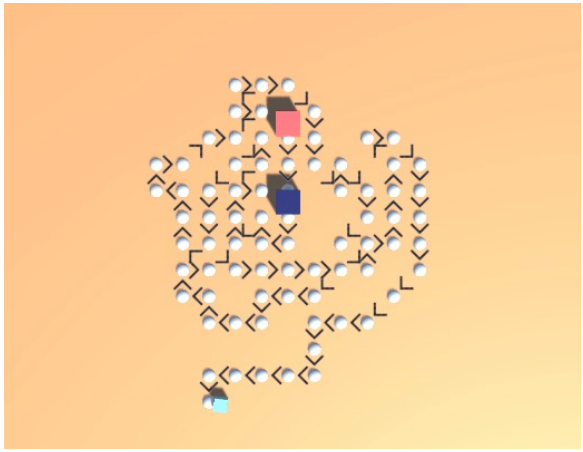

Figure 2.11: Attractor with base cost of -100.

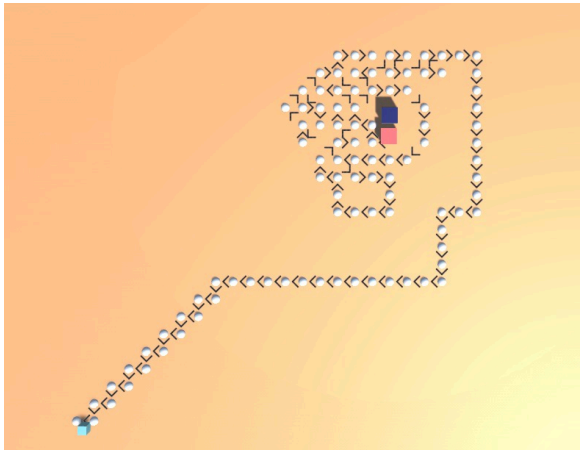

Figure 2.12: Attractor with base cost of -100.

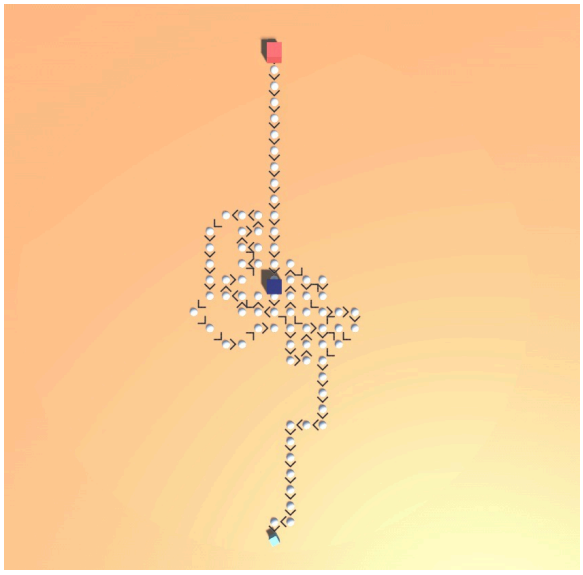

Figure 2.13: Attractor with base cost of -100.

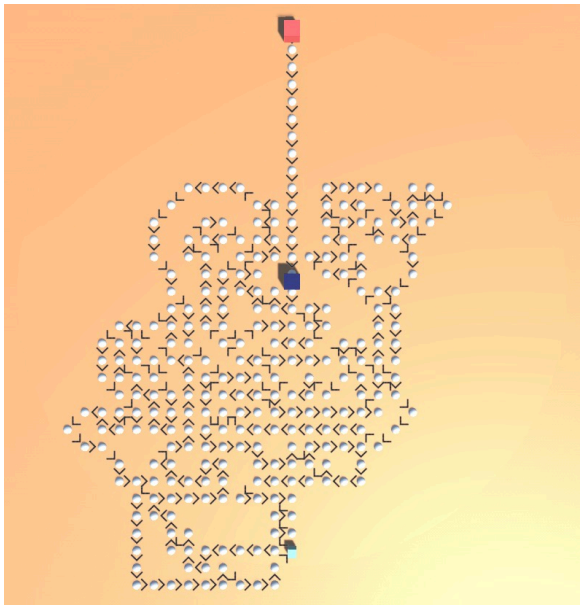

Figure 2.14: Attractor with base cost of -1000.

In case of multiple attractors located far from each other, if they are all reasonably close to a shortest path from the agent to the goal, the agent would try to reach as many attractors as it can in a reward-collecting manner, sometimes temporarily moving away from its goal, as shown in Figure 2.15, Figure 2.16, and Figure 2.17

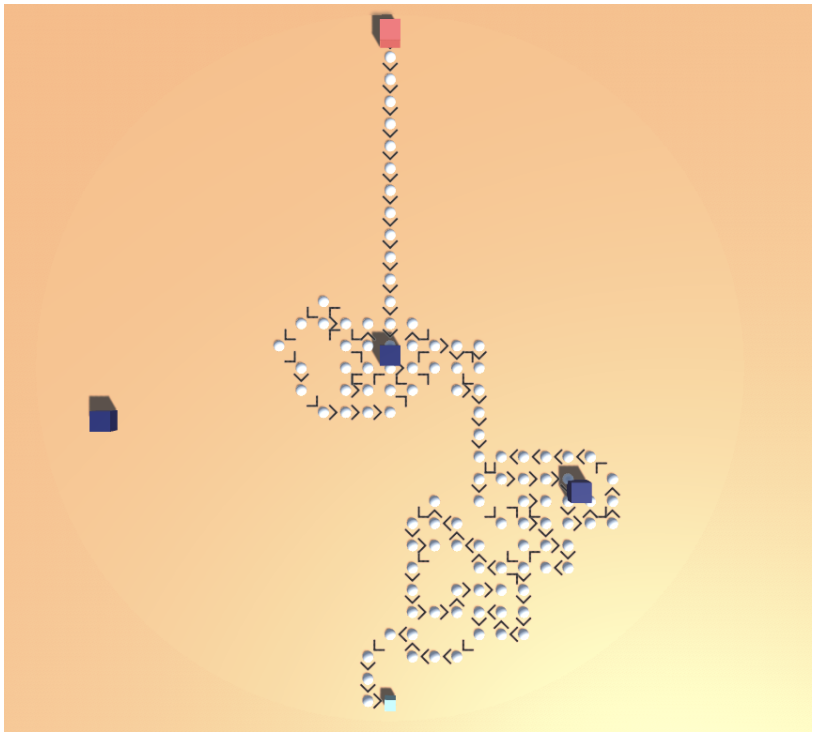

Figure 2.15: Attractors with base cost of -100.

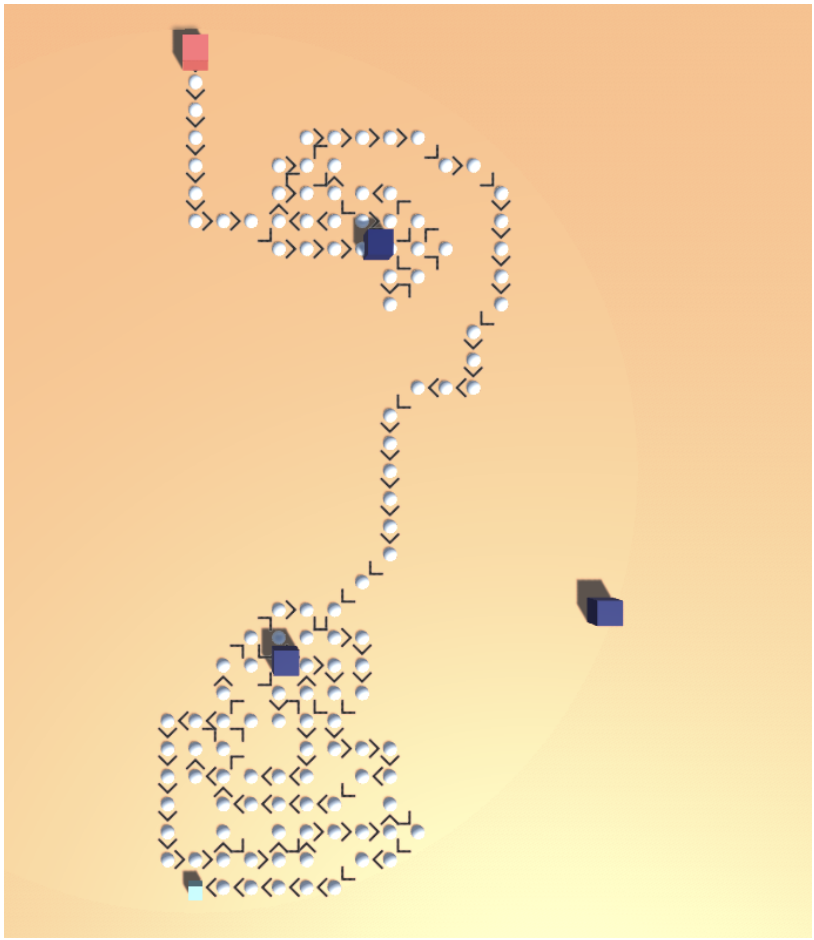

Figure 2.16: Attractors with base cost of -100.

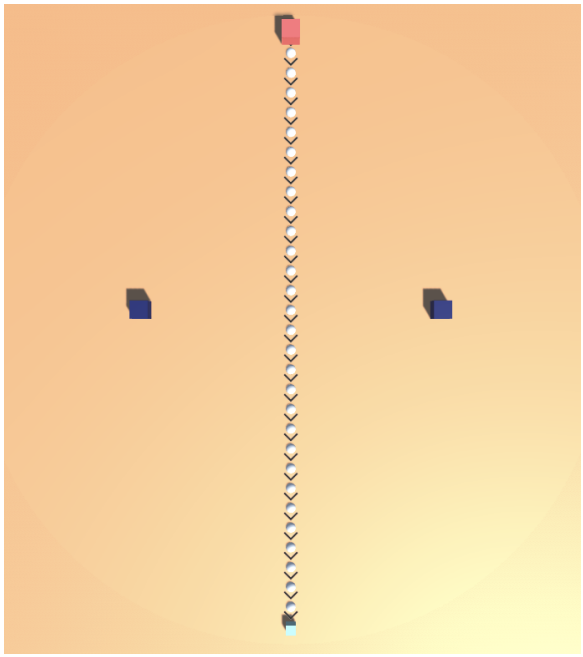

Figure 2.17: Attractors with base cost of -100.

For both kinds of radiators, it is possible that evenly distributed radiators cancel off each other, so the agent moves like on a shortest path.

However, the meandering behavior happens only if the agent follows one single path from start to finish. Otherwise, once the agent reaches an attractor, it is trapped at the attractor due to path inconsistency.

2.3.2 Path Inconsistency

For pathfinding with attractors, the resulting optimal path is no longer self-consistent. More specifically, if an agent follows the optimal path it finds, and if it re-runs the pathfinding algorithm after it has moved away from the start node, it might find the rest of the current path sub-optimal. If it constantly re-runs the pathfinding algorithm, it would change its

plan as it moves, and results in inconsistent behaviors.

Sometimes it is not obvious unless the optimal path is visualized. But a serious scenario of it would be an agent constantly changing its planned path but never getting closer to the goal, because it is constantly meandering around. In extreme cases, especially common when the agent gets close to an attractors, it moves back and forward between the node the attractor is located at and an adjacent node, because each time it moves away from the attractor, the next plannned path would include the attractor position again, and is then stuck forever. We make it clear that this is not caused by an infinitely long path which is created to "collect" rewards.

This situation can be avoided through some engineering workarounds. We can make the agent run the algorithm only for one time if the target does not move during a navigation. If the target does move frequently, one way is to make an agent blind to nodes that it has covered. Another way, depending on the scenario, is to reduce the base cost of attractors that the agent has reached. Alternatively but less desirably, we can also limit the magnitude of base cost for attractors, so the attraction is never so strong that an agent would get too close.

The inconsistency deserves more attention if the attractor is to be used frequently. It might also reveal some more important issue with the algorithm that we have not spotted.

2.3.3 Performance

The performance results are quite interesting.

Depending on its location, attractor affects the runtime differently. When it is located between the start and the goal, or reasonably closely along the same direction, it significantly shortens the runtime compared to a normal A* pathfinding. It is possible that the attractor, since its effect is included into h cost, serves as a guide to optimal path, and "pulls" the

search towards it.

On the other hand, when it is located away from the shortest path, the algorithm takes longer time because the path is curly and longer and/or the location the of attractor makes the straight-line heuristic less effective.

The magnitude of base cost makes the effects mentioned stronger in both cases.

But this is not the case for repellers. Depending on location of repeller, the runtime only becomes longer compared to a normal A* search. The performance is worst when the repeller is located between the agent and its target, and best (still worse than A*) when it is as far as it can be from the shortest path. Even when it sits behind the agent, when one would guess that it could "push" the search towards shortest path just as attractor "pulls" the search, it also makes the search significantly slower.

Future investigations and tests are required to determine the cause of this slowdown.

In our experiments, we run the algorithm at a 50x50 empty graph, where the agent and its target are located 45 nodes away from each other (shortest path has 45 nodes). A repeller of 100 base cost sitting in between causes the program to run about 6 times longer than a normal A*, whereas an attractor of -100 base cost at the same location causes the program to run only in about 1/3 of the time.

Chapter 3

Adding Learning Ability to Video Game AI

3.1 Related Works

In the context of video games, an AI agent with general learning ability implies infinitely many possibilities for good game design. From a storytelling perspective, an NPC with learning ability appears to be much more human-like. From a gameplay perspective, an NPC that adapts to players' strategy is simply more interesting.

Traditional video game AI rarely learns. Most of the time, NPCs only behave as the pre-scripted input-reaction function guides. The major difficulty of having a learning ability is that we don't have a robust enough learning framework. In recent years, deep learning sheds light on how behavioral tasks can be solved.

However, video game industries seem quite unmoved compared to the rest of the world. Of course, there are video game developers exploring the possibility that deep learning might provide, but so far we have observed no significant case where machine learning

creates significantly different game experiences. The most relevant noteworthy achievement belongs to machine learning but not video games–AI that plays video games at an expert level [17] [19].

After our experiments, we believe the current deep learning approach is still not mature enough to be used on video game AI. We, therefore, consider the more traditional approaches. Later in this chapter, we will discuss our attempts to incorporate traditional reinforcement learning into behavior selection frameworks.

3.1.1 Why Not Deep Learning

In recent years, deep learning has proven to be a promising solution to many difficult problems. At its core is the neural network, a model that theoretically can approximate any continuous convex functions [12]. The idea of using neural networks is not brand new, but only recently, better understanding and more powerful hardware start to make the model more accessible and useful.

Deep reinforcement learning is proposed as a result of a better use of neural networks, as the general-purpose model largely transforms the question of how to figure out policy function to be how to figure it out more efficiently. After all, a well-trained neural network should be able to describe the most meaningful behaviors.

After our experiments with Unity ML-Agents, we also conclude that deep learning is still not mature enough to model NPC behaviors in video games with respect to the criteria we have covered in Chapter 1. Those criteria closely match the current problems we observed in our deep learning experiments, and we will talk about them in details shortly.

What is Deep Reinforcement Learning

Deep reinforcement learning [16] is a modern approach to reinforcement learning problem with the help of neural networks. Besides techniques required to train a neural network, the fundamental idea is still reinforcement learning. A reinforcement learning agent learns to perform a task on its own (unsupervised) by observing the correlation between pre-defined rewards and its actions and improving its behavior model in order to maximize rewards.

The behavior model is called a policy function, taking inputs from the environment, and returning actions as outputs. Since it is likely to be a continuous convex function, we can model it with neural networks. The advantage is that it goes around the hard work of figuring out state space, provided that neural networks can approximate virtually any arbitrary function. It is most handy when a policy function is complex and difficult to manually design.

Reinforcement learning is only successful on behavioral problems, where rewards are easier to define in code, such as playing video games, or perform walking like a human being. The reward can simply be defined respectively as "getting a higher score" and "moving forward and not touching the ground with the upper body." Whereas for problems such as image recognition, supervision (providing the correct answer) is required, since telling whether an image contains a dog cannot be done programmatically.

Training Samples and Convergence

Compared to a design approach, using reinforcement learning to train an AI has the advantage that once the environment is set up correctly, the learning can go on its own. This is true for most agents, but in video games, the environment necessarily involves players. Those NPCs exist for the single reason for interacting with players.

Deep learning relies on a large number of samples to converge. That means for an agent to learning anything meaningful, it needs to repeat the game for a lot of times. That is not realistic given the limited time and patience a player would have for a video game.

Even the so-called "one-shot" learning [8] approach is not truly "one-shot," but relies on a pre-training that also demands a significant amount of samples. For one-shot learning agents to be robust in real time, they need to be trained with a significant number of different interactions with human players during development. It not only misses the point of deep learning, but is also unlikely given the complexity of neural networks.

Even with a lot of samples, convergence to an ideal or working level is never guaranteed for deep learning, despite the theoretical proof of convergence. Deep reinforcement learning is infamous for only more difficult to make it converge.

But here we mention another direction that might be hopeful. Perhaps a well-trained player AI can serve the role for training NPC AI. The idea of adversarial networks might be helpful.

Controllability

Neural networks also are known for its similarity to a black box [1]. It approximates a function in layers of nodes. The roles of nodes and the relationship between them are often unknown, especially for complex target functions. If we don't know what a neural network does exactly, we can't predict what it would do until we see it. But since the function is continuous, we can never exhaust all the possible situations. Even if we have a seemingly robust model, there is no way to guarantee that it would behave correctly in response to actions by players unthought of during development.

In terms of behavior design, deep reinforcement learning is similar to a utility-based framework. These two approaches only allow the design of preference and unit actions,

but not the resulting high-level behaviors, which are supposed to emerge as a result of interaction with the environment. This is good when more freedom is preferred, but bad if NPCs are to be carefully designed.

Undesired behaviors include "jittering," meaning that an agent switches back and forth between unit actions because the actions are closely preferred. A lot of cases that are considered successful in terms of performance also cannot avoid intuitively disturbing behaviors. In general, utility-based approaches lack a way to guarantee smooth, meaningful sequences of unit actions on a high level.

Efficiency

The training time for deep learning is never insignificant. Some models take hours to converge, and others take days. The long training time is due to the way neural networks are updated at each iteration. Most, if not all, deep learning approaches use stochastic gradient descent. With policy function (a neural network) and feedback reward, it is possible to formulate a reward function in terms of the parameters of the policy function. During each iteration, the program calculates an approximate partial derivative of the reward function with respect to parameters of the neural network. It updates the parameters with a tiny step towards the direction of a higher reward. The step has to be tiny because computers can only approximate the partial derivative since the derivative cannot be calculated, and a large step would not necessarily be in the correct direction.

For training to look noticeable in real-time, policy update has to be radical. Unless one can achieve true one-shot learning, the deep learning approach is not feasible.

3.1.2 Behavior Tree

Behavior trees are one of the most popular behavior selection frameworks in video game industries. It closely resembles an HFSM (hierarchical finite state machine) [22] as a highly modularized state-based framework [18]. Figure 3.1 shows an example of a behavior tree, performing the task of going through a door.

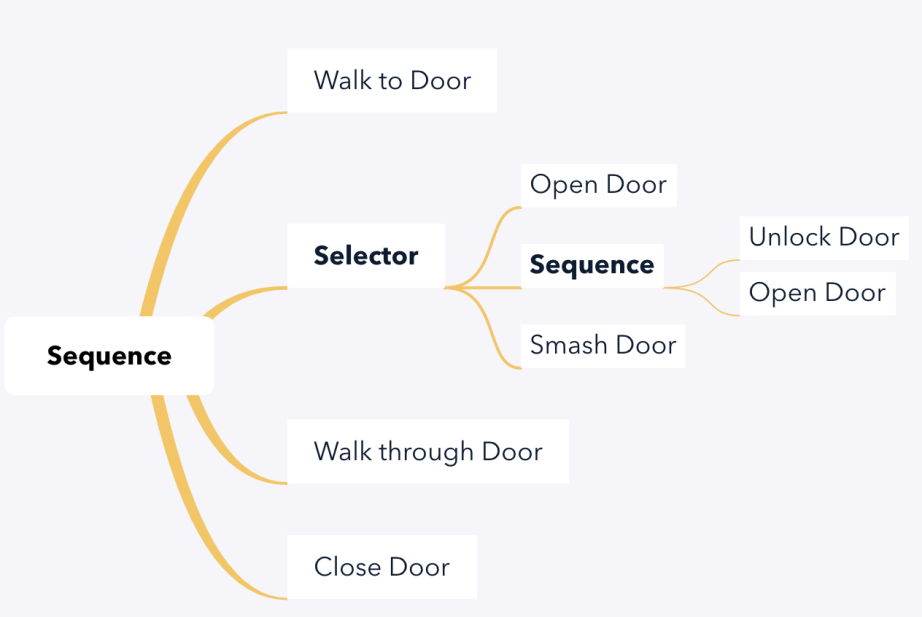

Figure 3.1: An example of behavior tree.

Whenever an agent does not have an ongoing action, it ticks the root node of the behavior tree. If the node is an action node, it performs a certain task. If the node is a control flow node, it ticks one or more of its children. The nodes on the leave are all action nodes, and nodes within the tree are all control flow nodes.

The control flow nodes generally don't perform any actual task but determine which actions are to be executed and in what order. Each action node has explicitly defined

failure and success status. When the node finishes its own action, it returns one of the statuses back to its parent. And its parent, a control flow node, determines either to execute another child or return to a higher parent depending on the status returned and its own type. When it returns, it ignores its unexecuted children. Multiple actions can be performed before the root node returns. After that, the behavior tree is to be ticked again.

There are some common default control flow nodes by convention. Nodes with multiple children are called composite nodes, including sequence node and selector node. A sequence node is supposed to successfully execute all of its children. If any of them returns a failure, the node immediately returns failure. Otherwise, it returns a success. A selector node is supposed successfully execute only one child. If any of them returns a success, the node immediately returns a success. Otherwise, it returns a failure.

Control flow nodes with only and only one child are called decorator nodes. They typically create a special condition for its child. Implementation of behavior tree by Unreal Engine 4 includes "conditional loop," which repeats its child until a certain condition is met or unmet, "force success," which returns success regardless of its child's status, etc.

Behavior trees are popular among game development community because of its highly logical form, and thus ease of understanding and using. It is also not difficult to implement since the idea is as simple as a finite state machine.

Another major advantage of behavior trees, which is why we consider it a good framework, is its arbitrary level of modularity. Based on developers' demand, behavior tree can define either a large tree of carefully designed complex behaviors, or only a few unit actions not far away from the root, or both situations branched separately. The flexibility leads us to consider behavior tree a good framework for our purpose since we want the "unit action" chosen by a learning entity to be already a meaningful high-level action by itself. There are some other studies focusing on the modularity characteristic of behavior trees.

3.2 Methods

We propose a new type of composite node, named learner node. The learner node executes one and only one child, and returns the same status returned from that child. The learner node keeps a list of weights for its children, each representing the node's preference for a child. When the learner node is ticked, it chooses the child to be executed based on the weights. If the child fails, learner node punishes it by reducing its weight, and otherwise rewards it by raising its weight.

The rules for choosing a child and modifying weights can be arbitrary. In our project, we implement the rule as shown in Algorithm 4.

Algorithm 4 Learner Node

```
Initialize parent, children, weights, lastTicked
function TICK
    softmaxWeights = Softmax(weights)
    childIndex =WeightedRandom(softmaxWeights)
    lastTicked = childIndex
    children[childIndex].Tick()
end function

function FINISH(childStatus)
    if childStatus == Success then
        weights[lastTicked] += 1
    else
        weights[lastTicked] −= 1
    parent.Finish(childStatus)
end function
```

In addition to learning, we choose to include stochastic behaviors. The softmax function is a function that normalizes a vector of real numbers into a vector of a real number between 0 and 1. The output vector can be interpreted as probabilities because the numbers always sum to 1. Therefore, regardless of the sign and absolute value of the weight, the correspond-

ing action always has a chance to be executed. Another parameter of softmax determines how concentrated the resulting distribution of probabilities would be, corresponding to how likely the agent is willing to try actions with low weights.

Softmax [3] is a typical function for action-choosing in reinforcement learning to encourage exploration by chance while focusing on exploitation most of the time, known as explore-exploit dilemma [27]. Our implementation actually resembles a naive version of reinforcement learning. But it is only a small step from a state-based approach towards a utility-based approach. The node neither observes environmental variables nor base its decisions on them.

The complexity of the learning to be implemented with learner node is only a design choice and a matter of degree. If needed, one should feel free to implement it with more sophisticated reinforcement learning features. But the downside is not only complexity and efficiency issues. The more it resembles a reinforcement learning agent, the more heavily it would be affected by the issues discussed in related works. As a step of testing of concept, we choose to withhold from more complexity for now.

3.3 Usage Scenario

To test the concept, we implement our own behavior tree in a Unity environment and make a game demo for it. Here we describe how we design the AI and how it fits the design goal. The behavior tree is shown in Figure 3.2.

We design an AI agent with a single goal of catching an escaping player, and design a few pre-defined chasing strategies for it. For example, a pair of behaviors can be a slower, tracking follow and a faster, fixed-direction dash. The strategies are to be put under a learner node. In our example, we make sure the agent approach the player first, and when

the player is within a certain distance, it decides whether to do a dash or a follow.

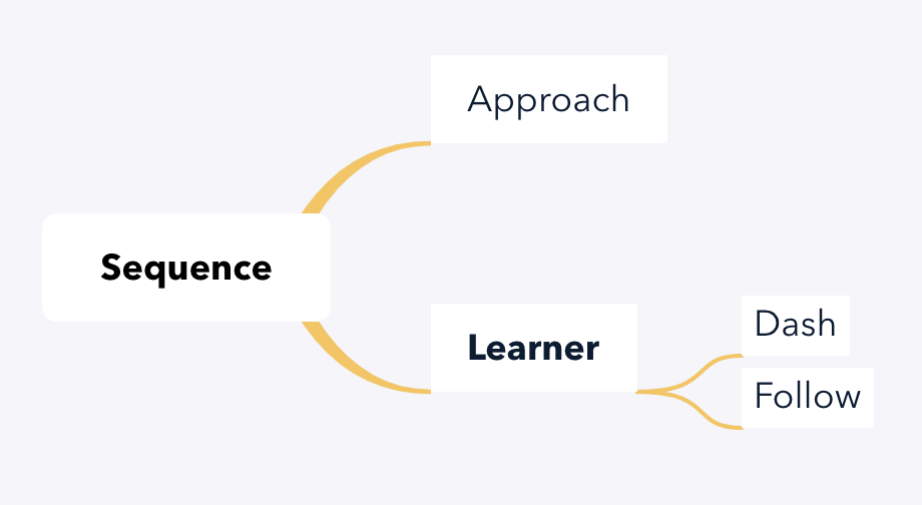

Figure 3.2: An agent that chases the player.

In general, the competitive behaviors should be no significantly inferior or superior compared to each other. Otherwise, unless the agent is intended to appear clumsy or stupid, there is no point to include obviously bad behaviors.

With a set of comparable strategies, the scenario becomes a dynamic one in that, as the agent learns which strategy appears to be working better, the player is also forced to learn and react differently since all strategies have comparable difficulty to tackle. An analogy we consistently visit is a rock-paper-scissors game, which demonstrates the idea that no strategy is better in its purest form. Realistic design in modern video games can be, for example, aggressive behaviors versus defensive behaviors in combat.

3.4 Results

The results should be expected by those familiar with reinforcement learning. With a shallow behavior tree, the agent does not act significantly different from an agent performing naive reinforcement learning.

In our experiments, the agent learns to adapt to player's behaviors. If the player always escapes in a straight line, the agent would learn to catch the player with a faster dash. If the player does not move in a straight line, which means the player moves away from the agent at a slower speed, the agent would learn to favor the tracking follow, with more accurate direction. As players become skilled at handling both strategies, the probabilities for both behaviors become close to each other.

But if a more complex hierarchy of behavior tree, where the learner node locates relatively deeper, is to be implemented, the performance or learning process can be a little different. The state space to be learned in is not the entire space anymore. As other parts of the behavior tree can constantly change the state the agent is in, the state space that can be visited by a learner node might have a radically different distribution, if not a subset of the original state space. The implication is that, the design of a learning agent is no longer with respect to the general environment anymore, but requires more consideration with respect to the specific situation the branch is likely to put the learner node in.

Chapter 4

Discussion

4.1 Contributions

In this thesis, we present two contributions. The first contribution is a pathfinding algorithm, named Cost Radiation A*, that finds an optimal path in an environment with secondary incentives. It achieves a human-like pathfinding behavior by consistently finding a path bent towards an attractor or away from a repeller.

The second contribution is a new component, named learner node, for behavior tree that enables an agent using the framework to learn and adapt in real time. In our demo environment, the node achieves naive reinforcement learning and is open to more sophisticated use based on demands.

In the following sections, we talk about the shortcomings of our methods and directions in which future studies can be done.

4.2 Cost Radiation A*

In our experiments, a bad location of radiator could result in a runtime of about 0.06 seconds for CRA* in worst cases. As mentioned in the section of the introduction, smooth video game experience requires at least 30 FPS, and 60 FPS ideally. Generally speaking, the total runtime of all algorithms needs to be shorter than 0.03 seconds. Optimization is a big topic if CRA* is to be used in a real project.

A* Complements and Algorithm Optimization

What we did not manage to experiment with are other A* variants. There are quite a few of them dealing with a dynamic environment and an ongoing search. Most of them are created to work around the need to re-run A* all the time. If useful features can be incorporated from those algorithms, we believe that the performance issue would be relieved to some extent.

The algorithm and grid implementation might also have non-trivial optimizations that can be made. We are currently unaware of them.

Heuristics and Alternative Implementation

We believe that CRA* runs slower than A* in our experiments mainly because the straight-line heuristic we used becomes worse at predicting better nodes when other factors are introduced. Therefore the algorithm makes more unnecessary exploration than A* does. Making h absorb r is only a rough way of maintaining the admissibility rule, and it is not necessarily an efficient way of utilizing information.

For future studies, an important direction is to explore proper heuristics for search with secondary incentives and what additional properties would the heuristic have. It will be

useful to figure out what causes the performance difference between repellers and attractors. An alternative implementation that incorporates r differently is also encouraged.

4.3 Behavior Tree

As we have shown, one single node for behavior tree can radically change what the framework can do. The learner node we present here is simple and naive in many aspects. Is for the only purpose of testing the concept, and might not be mature for a real project. We hope our attempt can be inspiring to others. In a real project, based on real demands, developers might come up with their own nodes that fit their design goals.

Bibliography

[1] José Manuel Benítez, Juan Luis Castro, and Ignacio Requena. Are artificial neural networks black boxes? *IEEE Transactions on neural networks*, 8(5):1156–1164, 1997.

[2] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 1(1):7–28, 2004.

[3] John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer, 1990.

[4] Julian Ceipek. Game path planning, 2019.

[5] Michele Colledanchise and Petter Ögren. Behavior trees in robotics and ai: an introduction. *arXiv preprint arXiv:1709.00084*, 2017.

[6] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[7] Michael Dawe, Steve Gargolinski, Luke Dicken, Troy Humphreys, and Dave Mark. Behavior selection algorithms. *Game AI Pro: Collected Wisdom of Game AI Professionals*, 47, 2013.

[8] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. In *Advances in neural information processing systems*, pages 1087–1098, 2017.

[9] Kris Graft. When artificial intelligence in video games becomes artificially intelligent. *Gamasutra: The Art & Business of Making Games*, 2015.

[10] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[11] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[12] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[13] Damian Isla. Gdc 2005 proceeding: Handling complexity in the halo 2 ai. *Retrieved October*, 21:2009, 2005.

[14] Sven Koenig and Maxim Likhachev. Dˆ* lite. *Aaai/iaai*, 15, 2002.

[15] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[18] Petter Ogren. Increasing modularity of uav control systems using computer game behavior trees. In *Aiaa guidance, navigation, and control conference*, page 4458, 2012.

[19] OpenAI. Openai five, 2019.

[20] David Lynton Poole, Alan K Mackworth, and Randy Goebel. *Computational intelligence: a logical approach*, volume 1. Oxford University Press New York, 1998.

[21] B Serviss. The discomfort zone: The hidden potential of valves ai director. *Gamasutra, February. http://www. gamasutra. com/blogs/BenServiss/20130207/186193/The_Discomfort_Zone_The_Hidden_Potential_of_Valves_AI_ Director. php*, 2013.

[22] Valery Sklyarov. Hierarchical finite-state machines and their use for digital control. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):222–228, 1999.

[23] Greg Snook. Simplified 3d movement and pathfinding using navigation meshes. *Game programming gems*, 1(1):288–304, 2000.

[24] Zheng Sun and John H Reif. On finding approximate optimal paths in weighted regions. *Journal of Algorithms*, 58(1):1–32, 2006.

[25] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[26] Unity Technologies. Unity - manual: Navigation and pathfinding, 2019.

[27] Stewart W Wilson et al. Explore/exploit strategies in autonomy. In *Proc. of the Fourth International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, volume 4, pages 325–332, 1996.