22

Introduction to Search for Games

Nathan R. Sturtevant

22.1 Introduction

Search techniques are broadly used outside of video games, but search is not widely used in current video games, outside of its ubiquitous use in path planning. As a result, many books about artificial intelligence (AI) in games do not usually dedicate significant resources to describing general search techniques and their application. However, the use of search appears to be growing, and thus a section of this book is dedicated to search-specific techniques. The purpose of this chapter is to give some introduction and justification of why and when search will be a useful technique. As the importance and use of AI in games grows, particularly with the availability of parallel cores for computation, we expect that the role of search in games is also going to increase.

22.2 Illustrative Example

To illustrate the role and potential power of search, consider a battle in a role-playing game (RPG) where a nonplayer character (NPC) in the player's party has the choice of using several weapons to attack an enemy. There are a range of approaches that can be used when this NPC decides which weapon to use.

The simplest approach offloads the choice onto the human player, by asking them to equip a default weapon or specifying simple decision rules that help the NPC make the decision. While this might be strategically interesting for the human, it isn't interesting from an AI perspective.

The next approach is based on static analysis of the situation. The designer in charge of the AI might craft a number of rules or even build a more complicated decision tree or behavior tree to author the behavior of the NPC in battles. If the battles are relatively simple, or the NPC faces relatively uniform challenges, this approach can be sufficient. But, because it relies on static analysis, the resulting behavior can be brittle when the NPC is faced with new or unforeseen situations. Note that the input to this static analysis is a state of the game, and the output is one or more actions to be applied.

We can now introduce a basic search approach. Instead of using static analysis, we could build into our game engine the ability to query the expected damage by any weapon against any foe. Since the engine already has the ability to simulate the attack, it just needs a flag to determine whether to apply the damage from the attack or to simulate the damage and return the expected value. After doing this for all possible attacks, the NPC can, for instance, choose the attack that has the highest expected damage. In many ways, this is far simpler than the previous static analysis, because nothing needs to be hand authored for this to work. If a particular weapon is ineffective against a particular enemy, the NPC will know not to use it, and if an enemy is vulnerable to a particular weapon, the NPC will surely use that weapon. This is a 1-ply analysis, because it searches one step into the future.

The drawback of this approach is that by taking the action that maximizes damage, it ignores other resource considerations. Consider, for instance, a weapon that has a limited number of death strikes that will immediately kill an opponent. This power should obviously be saved for use against strong opponents, but the search would not discriminate between easy and hard foes. The approach can be generalized by mixing static analysis with search. Instead of choosing the attack that maximizes damage, the best action will balance the short-term damage to an opponent against the long-term use of resources. So, a static evaluation function must be designed to evaluate the result of a 1-ply search. The static evaluation function will need less sophistication than without the search, as some of the information normally captured in the evaluation function will now be captured in the search. In this example, the role of the static analysis shifts. Instead of suggesting actions, the static evaluation must evaluate states after an action has been taken.

Now, suppose that the designers change the parameters of the game, shifting the damage done by all types of weapons and introducing new weapon buffs. Any AI designed around a static evaluation would have to be completely retuned, and information about the new weapon damages and buffs would have to be added to the AI logic. But, with the 1-ply search, significantly fewer changes are needed. Changing the game engine immediately enables the AI to reason with the new weapon models. The static evaluation may have to be retuned to balance the new buffs, but this is relatively small in comparison.

Unless the search considers all possibilities, there is always the chance that it will miss something important in the future, whether it is the next battle immediately after this one or the boss at the end of the level. But, practically speaking, even a small amount of search will capture dynamics that are difficult to capture in a static evaluation function.

This can easily be generalized to deeper search. In general, the stronger and more exhaustive a search becomes, the weaker the static evaluation function can be. But, there are diminishing gains from search relative to its cost. Each game will have its own sweet spot where the intelligence of the AI is balanced against the resources required to achieve that performance.

22.3 Basic Search Requirements

In order to apply search to a game, there are a few requirements that must usually be met. In general, search begins with the *state* of the world. This doesn't have to be the full state of everything going on everywhere, but the state must capture everything that is of interest to the AI. The state is usually composed of a set of variables and their values. This might include all characters in a battle and their stats, such as health and inventory, but could also include an arrangement of puzzle pieces on a board.

Given the state of the world, you need some way to search over different states of the world through the application of valid *operators*. Operators can vary wildly between different types of games or different parts of the same game. In our battle example, different operators might attack with different weapons. But, in a puzzle game, operators might correspond to moving different pieces of a puzzle around. Because the available operators might change based on the state of the world, it is usually useful to have a function that can return the valid operators in a given state.

The final piece needed for search is the ability to *apply* a valid operator to a given state. This is the only step that actually changes the state of the world. Sometimes, it is useful to be able to *undo* operators to extract the parent of a particular state, but this isn't possible in all state spaces.

These three pieces are the basic foundation of many types of search. There are other distinctions, such as whether the search is adversarial, whether the world is deterministic or stochastic, whether actions are taken simultaneously or sequentially, and whether characters have full knowledge of the world. Each of these distinctions require different types of search, something we will not cover here, but it is worth mentioning that search can also be exhaustive in nature (such as A* or a breadth-first search) or local (such as a genetic algorithm or hill-climbing approach). Each type of algorithm has its own strengths and weaknesses.

Because search techniques tend to look at large numbers of states, it is important that these operations are efficient. If the state is too large, the number of operators is too large, or the cost of applying operators is too expensive, search may not be the best approach, or a more efficient representation might be necessary.

22.3.1 Efficient State Representation

There are significant advantages when the search representation corresponds exactly to the representation used by the game engine. This ensures that any reasoning done by the AI matches exactly with the world. But, sometimes, the world is too complicated to simulate exactly for search. The classical example for this is in path planning. It is often the case that movement restrictions for game characters are defined by the physics engine, which performs collision checking and other tasks that influence path following and locomotion. But, planning directly in the physics engine is usually too expensive. Thus, many path planning representations are an abstraction of the physical world that is more suitable for search than the physics engine. The drawback of this abstraction is that when the abstraction does not correspond with the real world, strange behavior results, such as characters that get caught on world geometry when trying to move around the world.

Despite this, there are many ways in which abstraction can simplify and decompose the world, making it feasible for planning. In our RPG example, for instance, the physical

movement of characters might be abstracted in order to eliminate the need for path planning during search. In a real-time strategy game (RTS), battles could be abstracted instead of simulated to allow for higher-level planning. The selection of operators can also abstract the choices available. Instead of allowing any character to attack any opponent with any weapon, the operators might only allow a choice between melee and ranged attacks. Choosing an appropriate abstraction of the world is important for controlling the cost of the search and the quality of the results.

Once the world is defined appropriately for search, there are additional techniques for reducing the cost of search. One important technique is to avoid storing multiple copies of the state of the world. Operators are usually much smaller than the world state, and so it is easier to store a sequence of operators than a sequence of states. This is particularly efficient when it is possible to *undo* operators, as a depth-first search can then just keep a single copy of the world state along with a stack of operators in the search. It is also important to avoid memory allocations during search. As much as is possible, data structures for providing legal operators should be preallocated, and runtime memory allocation should be avoided.

22.4 Alternate Uses of Search

When search capabilities are well integrated into a game engine, we have used them to support many other necessary functions of a game. For instance, the search engine can be used to determine what parts of the game graphical user interface (GUI) should be activated. This works when the search engine is able to produce a set of valid operators that correspond to GUI elements that a human player can interact with. Such elements are often enabled or disabled depending on the state of the game, and custom logic is often written to do this. But, the search code already encodes this knowledge with the available operators. Thus, instead of using custom logic for the GUI, the GUI can just query the search engine to determine what GUI elements should be activated. Related to this, the search engine can provide the ability for users to undo their actions or replay games by storing the sequences of operators that were used during the game.

In a similar manner, a search engine can be used to build a dynamic tutorial for a game. Tutorials, particularly in puzzle games, often have simple puzzles for players to solve. If the search engine is able to solve these puzzles automatically, then the tutorial does not have to use a fixed set of puzzles for training. Instead, the search engine can be used to find solutions and train users. Similarly, if players appear to be stuck, a search engine can be used to find solutions and suggest moves for players.

Finally, search engines are useful for debugging purposes, as even a simple depth-first search can quickly reach a large range of states, potentially exposing unintended consequences of the game design or other bugs.

22.5 Bottlenecks for Search

Search is not a universal antidote to underperforming AI; many compelling games have been created without using search. Thus, it is worth noting the situations when search might not be appropriate.

Many optimization problems are more suited to be solved by linear programming (LP). In these types of problems, the question often isn't what particular action to take, but, for instance, how to allocate resources along a continuous spectrum. LPs can also be used to compute action probabilities for games of imperfect information, although non-LP methods can also be used for this problem.

The value of search is also limited when it is very costly to apply operators, when the number of legal operators is very large, or when the static evaluation of states is difficult. In many cases, these difficulties can be overcome through the use of abstraction, although it isn't always apparent what form of abstraction will result in the best behavior. But, broadening use of search in games will help reveal the best abstractions for achieving high performance. The following chapters in this book will illustrate new and interesting ways to apply search for a variety of games.

22.6 Conclusion

The goal of this chapter was to provide a basic background on search techniques to introduce the chapters in the general search section of this book. The astute reader will find many of the ideas in this article reflected in the following chapters in this section. We hope that it will provide a valuable framework for thinking about search and its applications.