Combining Scripted Behavior with Game Tree Search for Stronger, More Robust Game AI

Nicolas A. Barriga, Marius Stanescu, and Michael Buro

[1 leave this spacer to make page count accurate]
[2 leave this spacer to make page count accurate]
[3 leave this spacer to make page count accurate]
[4 leave this spacer to make page count accurate]
[5 leave this spacer to make page count accurate]
[6 leave this spacer to make page count accurate]
[7 leave this spacer to make page count accurate]
[8 leave this spacer to make page count accurate]
[9 leave this spacer to make page count accurate]
[10 leave this spacer to make page count accurate]
[11 leave this spacer to make page count accurate]
[12 leave this spacer to make page count accurate]
[13 leave this spacer to make page count accurate]
[14 leave this spacer to make page count accurate]
[15 leave this spacer to make page count accurate]
[16 leave this spacer to make page count accurate]
[17 leave this spacer to make page count accurate]
[18 leave this spacer to make page count accurate]
[19 leave this spacer to make page count accurate]
[20 leave this spacer to make page count accurate]

1 Introduction

Fully scripted game AI systems are usually predictable and, due to statically defined behavior, susceptible to poor decision-making when facing unexpected opponent actions. In games with a small number of possible actions, like chess or checkers, a successful approach to overcome these issues is to use look-ahead search, i.e. simulating the effects of action sequences and choosing those that maximize the agent's utility. In this chapter we present an approach that adapts this process to complex video games, reducing action choices by means of scripts that expose choice points to look-ahead search. In this way, the game author maintains control over the range of possible AI behaviors and enables the system to better evaluate the consequences of its actions, resulting in smarter behavior.

The framework we introduce requires scripts that are able to play a full game. For example, a script for playing an RTS game will control workers to gather resources and construct buildings, train more workers and combat units, build base expansions and attack the enemy. Some details, such as which combat units to build and where or when to expand, might be very dependent on the situation and difficult to commit to in advance. These choices are better left open when defining the strategy, to be decided by a search algorithm which can dynamically pick the most favorable action.

In this chapter we chose to represent the scripts as decision trees because of the natural formulation of choice points as decision nodes. However, our approach is not limited to decision trees. Other types of scripted AI systems such as finite-state machines and behavior trees can be used instead, by exposing transitions and selector nodes respectively.

2 Scripts

For our purposes we define a script as a function that takes a game state and returns actions to perform now. The method used to generate actions is unimportant: it could be a rule based player hand coded with expert knowledge, or a machine learning or search based agent, etc. The only requisite is that it must be able to generate actions for any legal game state.

As an example consider a *rush*, a common type of strategy in RTS games that tries to build as many combat units as fast as possible in an effort to destroy the opponent's base before he has the time to build suitable defenses. A wide range of these aggressive attacks are possible. At one extreme, the fastest attack can be executed using workers, which usually deal very little damage and barely have any armor. Alternatively, the attack can be delayed until more powerful units are trained.

2.1 Adding Choices

Figure 1 shows a decision tree representing a script that first gathers resources, builds some defensive buildings, expands to a second base, trains an army and finally attacks the enemy. This decision tree is executed at every frame to decide what actions to issue. In a normal scripted strategy, there would be several hardcoded constants: the number of defensive buildings to build before expanding, the size of the army and when to attack. However, the script could expose these decisions as choice points, and let a search algorithm explore them to decide the best course of action.

When writing a script, we must make some potentially hard choices. Will the AI expand to a new base after training a certain number of workers or will it wait until the current bases' resources are depleted? Regardless of the decision, it will be hardcoded in the script, according to a set of static rules about the state of the game. Discovering predictable patterns in the way the AI acts might be frustrating for all but beginner players. Whether the behavior implemented is sensible or not in the given situation, they will quickly learn to exploit it and the game will likely lose some of its replay value in the process.

As script writers, we would like to be able to leave some choices open, such as which units to rush with. But the script also needs to deal with any and all possible events happening during the strategy execution. The base might be attacked before it is ready to launch its own attack, or maybe the base is undefended while our infantry units are out looking for the enemy. Should they continue in hope of destroying their base before they raze ours? Or should they come back to defend? What if when we arrive to the enemy's base, we realize we don't have the strength to defeat him? Should we push on nonetheless? Some, or all, of these decisions are best left open, so that they can be explored and the most appropriate choice taken during the game.

The number of choice points exposed can be a configurable parameter with an impact on the strength and speed of the system. Fewer options will produce a faster but more predictable AI, suitable for beginner players, while increasing their number will lead to a harder challenge, at the cost of increased computational work.

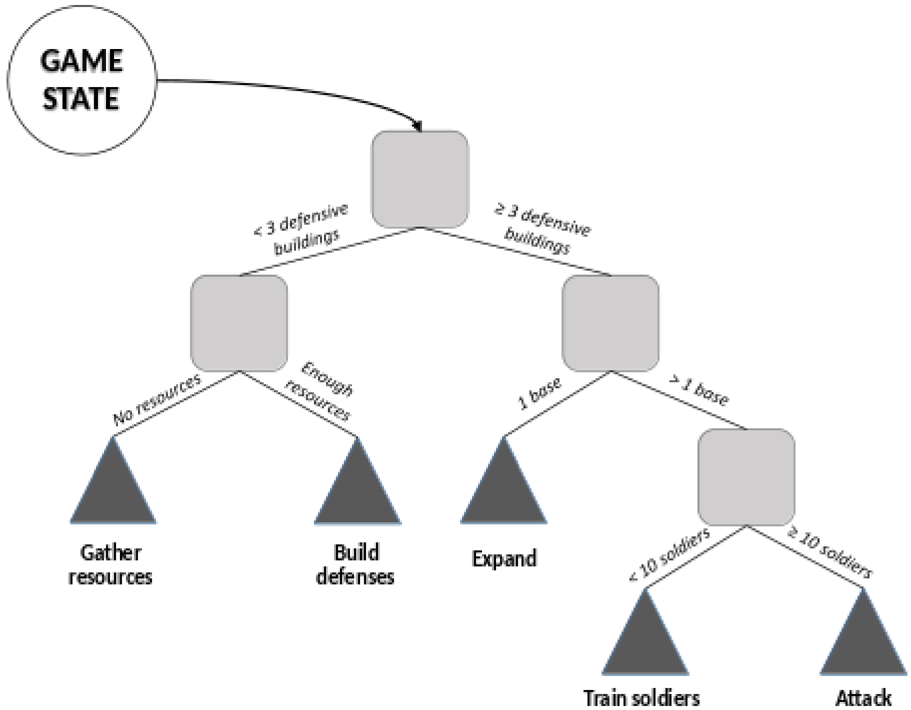

Figure 1 Decision tree representing script choices.

*3 A*dding Search

So far we have presented a flexible way to write AI scripts that include choice points in which multiple different actions can be taken. However, we have not mentioned how those decisions are made. Commonly, they would be hardcoded as a behavior or decision tree. But there are other techniques that can produce stronger AI systems without relying as heavily on expert knowledge: machine learning (ML) and look-ahead search.

An ML based agent relies on a function that takes the current game state as input, and produces a decision for each choice in the script. The parameters of that function would then be optimized either by supervised learning methods on a set of game traces, or by reinforcement learning, letting the agent play itself. However, once the parameters are learned, the model acts like a static rule based system and might become predictable. If the system is allowed to keep learning after the game has shipped, then there are no guarantees on how it will evolve, possibly leading to unwanted behavior.

The second approach, look-ahead search, involves executing action sequences and evaluating their outcomes. Both methods can work well. It is possible to have an unbeatable ML player if the features and training data are good enough and a perfect search based player if we explore the full search space. In practice, neither requirement is easy to meet: good representations are hard to design, and time constraints prevent covering the search space in most games. Good practical results are often achieved by combining both approaches [Silver 16].

3.1 Look-Ahead Search

To use look-ahead search, we need to be able to execute a script for a given timespan, look at the resulting state, and then go back to the original state to try other action choices. This has to happen without performing any actions in the actual game, and it has to be several orders of magnitude faster than the real game's speed because we want **a)** to look-ahead as far as possible into the future, to the end of the game if feasible, and **b)** to try as many choice combinations as possible before committing to one.

This means we need to be able to either save the current game state, copy it to a new state object, execute scripts on the copy, and then reload the original, or execute and undo the actions on the original game state. The latter approach is common in AI systems for board games, because it is usually faster to apply and undo a move than to copy a game state. In RTS games however, keeping track of several thousand complex actions and undoing them might prove difficult, so copying the state is preferable.

When performing look-ahead we need to issue actions for the opponent as well. Which scripts to use will depend on our knowledge about him. If we can reasonably predict the strategy he will use, we could simulate his behavior as accurately as possible and come up with a best response—a strategy that exploits our knowledge of the enemy. For example, if we know that a particular opponent always rushes on small maps, then we will only explore options in the choice points that apply to rushes to simulate his behavior, while fixing the other choices. If the script has a choice point with options **a)** rush, **b)** expand, and **c)** build defenses, and a second choice point with the type of combat units to build, we would fix option **a)** for the first choice point and let the search explore all options for the second choice point. At the same time, we will try all the possible choices for ourselves, to let the search algorithm decide the best counter strategy.

However, the more imprecise our opponent model is, the riskier it is to play a best response strategy. Likewise, if we play against an unknown player, the safest route is to try as many choices for the opponent as for ourselves. The aim is to find an equilibrium strategy that doesn't necessarily exploit the opponent's weaknesses, but can’t be easily exploited either.

3.2 State Evaluation

Forwarding the state using different choices is only useful if we can evaluate the merit of the resulting states. We need to decide which of those states is more desirable from the point of view of the player performing the search. In other words, we need to evaluate those states, assign each a numerical value and use it to compare them. In zero-sum games it is sufficient to consider symmetric evaluation functions `eval(state, player)` that return positive values for the winning player and negative values for the losing player with `eval(state, p1)=-eval(state, p2)`.

The most common approach to state evaluation in RTS games is to use a linear function that adds a set of values that are multiplied by a weight. The values usually represent simple features, such as the number of units of each type a player has, with different weights reflecting their estimated worth. Weights can be either hand-tuned or learned from records of past games using logistic regression or similar methods. An example of a popular metric in RTS games is Life-Time Damage, or LTD [Kovarsky 05], which tries to estimate the amount of damage a unit could deal to the enemy during its lifetime. Another feature could be the cost of building a unit, which takes advantage of the game balancing already performed by the game designers. Costlier units are highly likely to be more useful, thus the player that has a higher total unit cost has a better chance of winning. The chapter *Combat Outcome Prediction for RTS Games* [Stanescu 17] in this book describes a state-of-the-art evaluation method that takes into account combat unit types and their health.

A somewhat different state evaluation method involves Monte Carlo simulations. Instead of invoking a static function, one could have a pair of fast scripts, either deterministic or randomized, play out the remainder of the game, and assign a positive score to the winning player. The rationale behind this method is that, even if the scripts are not of high quality, as both players are using the same policy, it is likely that whoever wins more simulations is the one that was ahead in the first place.

If running a simulation until the end of the game is not feasible, a hybrid method can be used that performs a limited playout for a predetermined amount of frames, and then calls the evaluation function. Evaluation functions are usually more accurate closer to the end of a game, when the game outcome is easier to predict. Therefore, moving the application of the evaluation function to the end of the playout often results in a more accurate assessment of the value of the game state.

3.3 Minimax Search

So far we have considered the problem of looking ahead using different action choices in our scripts and evaluating the resulting states, but the fact that the opponent also has choices has to be taken into account. Lacking an accurate opponent model, we have to make some assumptions about his actions. For simplicity we'll assume he uses the same scripts and evaluates states the same way we do.

To select a move, we consider all possible script actions in the current state. For each we examine all possible opponent replies, and continue recursively until reaching a predefined depth or the end of the game. The evaluation function is then used to estimate the value of the resulting states, and the move which maximizes the player-to-move’s score is selected. This algorithm is called negamax [CPW 16-1]—a variant of the minimax algorithm—because in zero-sum games, the move that maximizes one player’s score is also

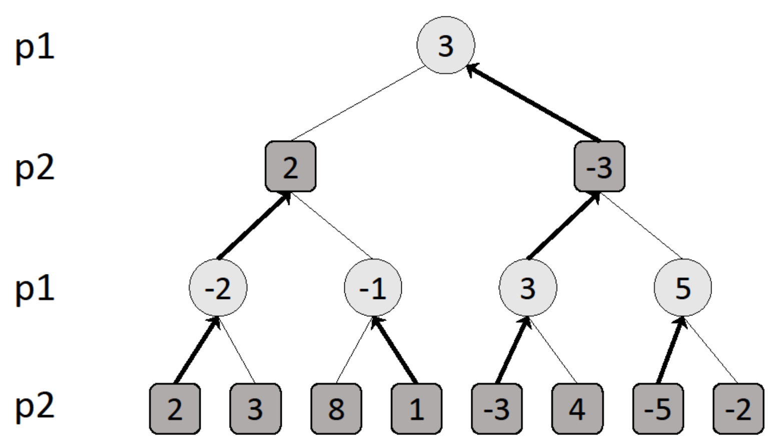

Figure 2 Negamax search example.

Listing 1 Negamax implementation in Python.

```
def negaMax(state, depth, player):
  if depth == 0 or terminal(state):
    return evaluate(state, player)
  max = -float('inf')
  for move in state.legal_moves(player):
    childState = state.apply(move)
    score = -negaMax(childState, depth-1, opponent(player))
    if score > max:
      max = score
  return max

#example call
#state: current game state
#depth: maximum search depth
#player: player to move
value = negaMax(state, depth, player)
```

the one that minimizes the other player's score. The move that maximizes the negated child score is selected and assigned to the parent state, and the recursion unrolls, as shown in Figure 2. Listing 1 shows a basic implementation returning the value of the current game state. Returning the best move as well is an easy addition.

A modification is needed for the minimax algorithm to work in our scripted AI. The moves in an RTS game are simultaneous, so they need to be serialized to fit the game tree search framework. Randomizing the player to move, or alternating in a p1-p2-p2-p1 fashion are common choices to mitigate a possible player bias [Churchill 12]. The resulting algorithm is shown in Listing 2.

In Listings 1 and 2, negamax takes as an input the height of the search tree to build, and being a depth-first algorithm, it only returns a solution when the tree has been fully searched. However, if the computation time is limited, we need an *anytime* algorithm that can be stopped at any point and returns a reasonable answer. The solution is to search to a shallow depth, 2 in our case, and then iteratively deepen the search by 2 levels until time runs out. At first it might look like a waste of resources, because the shallower levels of the tree are searched repeatedly, but if we add a *transposition table* information from previous iterations can be reused.

In this chapter we use the negamax version of the minimax algorithm for simplicity. In practice, we would use AlphaBeta search [CPW 16], an efficient version of minimax that prunes significant parts of the search tree, while still finding the optimal solution. AlphaBeta is more efficient when the best actions are examined first, and accordingly, there exist several move ordering techniques, such as using *hash moves* or *killer moves*, which make use of the information in the transposition table [CPW 16-2].

Listing 2 Simultaneous Moves Negamax.

```
def SMNegaMax(state, depth, previousMove=None):
  player = playerToMove(depth)
  if depth == 0 or terminal(state):
    return evaluate(state, player)
  max = -float('inf')
  for move in state.legal_moves(player):
    if previousMove == None:
      score = -SMNegaMax(state, depth-1, move)
    else
      childState = state.apply(previousMove, move)
      score = -SMNegaMax(childState, depth-1)
    if score > max:
      max = score
  return max

#Example call
#state: current game state
#depth: maximum search depth, has to be even
value = SMNegaMax(state, depth)
```

Another class of algorithms that can be used to explore the search tree is Monte Carlo Tree Search (MCTS) [Sturtevant 15]. Instead of sequentially analyzing sibling nodes, MCTS randomly samples them. A sampling policy like UCT [Kocsis 06] balances exploration and exploitation to grow the tree asymmetrically, concentrating on the more promising subtrees.

4 Final Considerations

So far, we have introduced scripts with choice points, a state evaluation function, and a search algorithm that uses look-ahead to decide which choices to take. Once the search produces an answer in the form of decisions at every choice point applicable in the current game state, it can be executed in the game. Given enough time, whenever the AI system needs to issue actions, it would start the search procedure, obtain an answer and execute it. However, in practice, actions have to be issued in almost every frame, with only a few milliseconds available per frame, so this can be impractical. Fortunately, as the scripts can play entire games, a previous answer can be used as a standing plan for multiple frames. The search can be restarted, and the process split across multiple frames until an answer is reached, while in the meantime the standing plan is being executed. At that point, the new solution becomes the standing plan. The search can be started again, either immediately, or once we find the opponent is acting inconsistently with the results of our search.

Experiments using *StarCraft: Brood War* have shown good results [Barriga 15]. A script with a single choice point that selects a particular type of rush was tested against state-of-the-art StarCraft bots. The resulting agent was more robust than any of the individual strategies on its own, and was able to defeat more opponents.

One topic we haven't touched on is fog-of-war. The described framework assumes it has access to the complete game state at the beginning of the search. If your particular game doesn't have perfect information, there are several choices. The easiest one is to let the AI cheat, by giving it full game state access. However, players might become suspicious of the unfair advantage if the AI system keeps correctly "guessing" and countering their surprise tactics. A better option is to implement an inference system. For instance, a particle filter can be used to estimate the positions of previously seen units [Weber 11], and Bayesian models have been use to recognize and predict opponent plans [Synnaeve 11].

5 Conclusion

In this chapter we have presented a search framework that combines scripted behavior and look-ahead search. By using scripts, it allows game designers to keep control over the range of behaviors the AI system can perform, while the adversarial look-ahead search enables it to better evaluate action outcomes, making it a stronger and more believable enemy.

The decision tree structure of the scripts ensures that only the choice combinations that make sense for a particular game state will be explored. This reduces the search effort considerably, and because scripts can play entire games, we can use the previous plan for as long as it takes to produce an updated one.

Finally, based on promising experimental results on RTS games, we expect this new search framework to perform well in any game for which scripted AI systems can be built.

6 References

[Barriga 15] Barriga, N.A., Stanescu, M. and Buro, M. 2015. Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games. *Proceedings of the Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.*

[CPW 16] Chess Programming Wiki. 2016. Alpha-Beta. http://chessprogramming.wikispaces.com/Alpha-Beta

[CPW 16-1] Chess Programming Wiki. 2016. Minimax. http://chessprogramming.wikispaces.com/Minimax

[CPW 16-2] Chess Programming Wiki. 2016. Transposition Table. http://chessprogramming.wikispaces.com/Transposition+Table

[Churchill 12] Churchill, D., Saffidine, A. and Buro, M. 2012. Fast Heuristic Search for RTS Game Combat Scenarios. *Proceedings of the Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Kocsis 06] Kocsis, L., Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. *17th European Conference on Machine Learning.*

[Kovarsky 05] Kovarsky, A. and Buro, M. 2005. Heuristic Search Applied to Abstract Combat Games. *Proceedings of the Eighteenth Canadian Conference on Artificial Intelligence*.

[Silver 16] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. and Dieleman, S., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*.

[Stanescu 17] Stanescu, M., Barriga, N.A. and Buro, M. 2017. Combat Outcome Prediction for RTS Games. In *Game AI Pro 3: Collected Wisdom of Game AI Professionals,* ed. Rabin, S., XXX-YYY. CRC Press.
[Sturtevant 15] Sturtevant, N.R. 2015. Monte Carlo Tree Search and Related Algorithms for Games. In *Game AI Pro 2: Collected Wisdom of Game AI Professionals*, ed. Rabin, S., 265-281. CRC Press.
[Synnaeve 11] Synnaeve, G. and Bessière, P. 2011. A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft. *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference.*
[Weber 11] Weber, B.G., Mateas, M. and Jhala, A. 2011. A Particle Model for State Estimation in Real-Time Strategy Games. *Proceedings of the Seventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.*

10 Biography

Nicolas A. Barriga is a Ph.D. candidate at the University of Alberta, Canada. He earned B.Sc., Engineer and M.Sc. degrees in Informatics Engineering at Universidad Técnica Federico Santa María, Chile. After a few years working as a software engineer for Gemini and ALMA astronomical observatories he came back to graduate school and he is currently working on state and action abstraction mechanisms for RTS games.

Marius Stanescu is a Ph.D. candidate at the University of Alberta, Canada. He completed his MSc in Artificial Intelligence at University of Edinburgh in 2011, and was a researcher at the Center of Nanosciences for Renewable & Alternative Energy Sources of University of Bucharest in 2012. Since 2013, he is helping organize the AIIDE StarCraft Competition. Marius' main areas of research interest are machine learning, AI and RTS games.

Michael Buro is a professor in the computing science department at the University of Alberta in Edmonton, Canada. He received his PhD in 1994 for his work on Logistello - an Othello program that defeated the reigning human World champion 6-0. His current research interests include heuristic search, pathfinding, abstraction, state inference, and opponent modeling applied to video games and card games. In these areas Michael and his students have made numerous contributions, culminating in developing fast geometric pathfinding algorithms and creating the World's best Skat playing program and one of the strongest *StarCraft: Brood War* bots.