Reinforcement Learning Approaches for Locomotion Planning in Interactive Applications

Tomislav Pejsa and Sean Andrist

Abstract
Locomotion is one of the most important capabilities for virtual human agents in interactive applications, because it allows them to navigate their environment. Locomotion controllers in interactive applications typically work by blending and concatenating clips of keyframe or motion capture motion that represent individual locomotion actions (e.g. walk cycles), to generate sequences of natural-looking, task-appropriate character motion. The key challenge of locomotion implementation is planning - i.e. choosing an optimal sequence of locomotion actions that achieves a high-level navigation goal. In recent years researchers have successfully applied reinforcement learning to this problem. In this paper we give an overview of these efforts, and demonstrate our own application of reinforcement learning to a simple navigation task.

1. Introduction
The term locomotion refers to AI- or user-directed movement of a virtual character to a target location while avoiding both static and dynamic obstacles along the way. Locomotion in composite, dynamically changing environments is a fundamental challenge for virtual human characters in interactive applications. A potential solution is to employ motion planning, a set of techniques originating from robotics that use smart control policies for synthesis of long motion sequences which accurately and efficiently accomplish complex high-level tasks, e.g., reaching the target destination while avoiding all obstacles. The main difficulty arises from the fact that the space of states and actions to search is huge; the character can move in a potentially infinite number of directions in potentially infinite number of ways.

Characters are assumed to have a motion corpus of available motions they can use to move about their environment. This motion corpus can consist of either example-based motions from a motion capture system, or clips of keyframe animation. In the work we present in this paper, our character’s motion corpus consists of the latter. Unlike techniques that search for appropriate motion clips while using only local information, motion planning methods consider the entire relevant state space and generate motion sequences that are close to being globally optimal, that is they are near-guaranteed to achieve objectives in the best, most expedient manner possible. As performing global motion search at run-time is infeasible, it is necessary to perform as much of the planning computations as possible in a preprocessing step and then use the precomputed data at run-time to make near optimal local decisions. Efficient and near-optimal planners have been previously developed which employ control policies estimated through the iterative process of reinforcement learning.

In the next section we present background on the general area of motion synthesis for computer animation, especially motion planning in a data-driven framework. In Section 3 we detail how reinforcement learning is an effective approach for motion planning for locomotion. We also discuss our own implementation of locomotion planning using reinforcement learning for a character navigating to a target in an environment with numerous obstacles, which can be seen in Figure 1.
2. Motion Synthesis for Computer Animation

There are a number of possible approaches to motion synthesis for character animation. Procedural and physics-based animation methods employ a mathematical model to generate desired character motion according to high-level parameters. Because of its parametric nature, this kind of motion is intuitively controllable and suitable for interactive applications. However, it suffers greatly from a lack of naturalness. The underlying mathematical model is typically only a rough approximation of actual physics that govern the movement of living creatures, while more complex models are too computationally expensive to be of any use in interactive applications. For this reason procedural and physics-based methods are still typically used only in a few specific situations. Instead, it is quite common to use data-driven methods which rely on having a database of recorded or hand-crafted motion clips with which to synthesize new natural-looking motions.

2.1. Data-driven motion synthesis

The fundamental idea behind example-based or data-driven motion synthesis is to combine the controllability of procedural animation with realistic appearance of recorded clips of motion. This is accomplished with two techniques: motion concatenation and parametric motion synthesis. The former refers to the concatenation of short motion segments into sequences of actions. This is commonly done with motion graphs [KGP02], which are graph structures with motion clips at the nodes and transitions between clips as edges. These edges must be precomputed between points where the character poses are similar enough to make the transition between clips smooth and natural. The second technique, parametric motion synthesis, enables parametrically controlled interpolation of similar motion clips which correspond to the same logical action. Inverse kinematics (IK) techniques are used to enforce constraints, such as footplants to reduce foot sliding.

2.2. Motion planning

Motion graphs and parametric motion are only useful if they can be used by high-level application modules for synthesis of motion sequences that accomplish various goals at run-time. Translating these high-level goals into synthesis of low-level motion sequences is the principal function of the character controller. The character controller is mainly responsible for locomotion, as well as both environment and character interaction. The set of techniques used for solving the locomotion problem in a globally optimal way are collectively referred to as motion planning.

Motion planning for locomotion is a complex problem for online motion synthesis. In most practical implementations, example-based motion synthesis is still done by performing search of the character’s motion corpus with local information. Such methods do not at all guarantee achievement of objectives in an optimal manner and may fail altogether in more complex scenarios. These problems are illustrated in Figure 2, which depicts several characters failing to walk through an open gate. Similarly in our implementation, presented later, we show how a greedy search strategy almost always fails.

A number of more sophisticated approaches have been proposed. One of these is probabilistic roadmaps.
Researchers in recent years have embraced reinforcement learning as an effective technique for locomotion planning. Rather than precompute search results, reinforcement learning precomputes a control policy that makes near-optimal decisions.

3. Reinforcement Learning for Locomotion

In reinforcement learning, objectives to accomplish are formulated as reward functions, which compute rewards for being in a particular state and performing specific actions (motion clips). Agents learn from an indirect, delayed reward, and choose sequences of actions that produce the greatest cumulative reward. A control policy is precomputed and used at run-time to pick actions which maximize the long-term reward, thus ensuring synthesis of an optimal motion sequence.

For our implementation, we use a common algorithm that we use, a use a common algorithm for reinforcement learning called Q learning that can result in optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment.

The simplest approach to Q learning, which we use, involves the construction of a Q table which lists all possible state-action pairs. A valid concern is that the Q table becomes much too large, possibly even infinite. This can be alleviated by using different representations, such as scattered data interpolation [IAF05], linear combination of basis functions [TLP07], and regression trees [LZ08]. In the end, at any state s the optimal policy, π∗ will compute the best action to perform. This is done with the following equation:

\[ \pi^*(s) = \arg \max_a Q(s, a) \]  

We build up our Q table by iteratively refining its values until we have converged to the final values. We do this by exploring the state-action space and updating Q values based on the rewards, r, received.

\[ \hat{Q}(s, a) = r(s, a) + \gamma \max_{a'} \hat{Q}(s', a') \]  

3.1. Motion Synthesis

Once an optimal policy has been estimated, motion synthesis is straightforward. As the current action finishes, the control policy is used to pick the next action A′ based on the current state s. The next state s′ is computed using the state transition function f(s, A′). Motion blending and transitions can be implemented in different ways. Discrete motion graphs are a simple approach [LL04, IAF05], but they are not suitable when responsiveness is critical, as the character’s movement is constrained to the graph. Motion fragments [TLP07] are another approach, where example motions are divided into fragments, each holding a sequence of frames that represents a single walk cycle. Transitioning between motion fragments is done without a motion graph or constraint enforcement. This can also be extended to parameterized motion fragments [LZ08] to support parametric motions.

3.2. Our implementation

To demonstrate the applicability of reinforcement learning to the problem of interactive locomotion, we implemented a simple reinforcement learner capable of learning a near-optimal locomotion control policy for a virtual agent. Our planner can be embedded in a moderate-size environment containing a set of static obstacles and a navigation target.

We built our system using the Unity game engine. Unity provides out-of-the-box support for rendering, character animation with transitions and priority blending, scripting, and a number of other features needed for development of rich interactive applications. We wrote our motion planner and learner in Unity’s scripting language, on top of an example character controller provided with the engine.

3.2.1. State and reward formulation

We represent the character’s state as the following tuple:

\[ s = (\theta, d, \varnothing, o) \]  

where \( \theta, d \) is the position of the target relative to the agent, expressed in spherical coordinates (relative to the agent’s movement direction). Similarly, \( \varnothing, o \) is the relative position of the obstacle nearest to the agent. Similar state representations are used in other reinforcement learning papers, such as [TLP07].
The state reward is expressed as the following function:

\[ R(s) = w_a(1 - \frac{|\theta|}{180}) - w_d - w_{\text{OBS}}(1 - \frac{d}{180})e^{-0.75d} \]  

where \( w_a \) are weights assigned to different components of the reward. The reward function is designed to reward the agent for facing the target, and punish it for being distant from the target, for being in proximity of an obstacle, and for heading in the direction of the obstacle.

\subsection*{3.2.2. Learning algorithm}

For the sake of simplicity we implemented 1-step Q-learning for training our controller. The action-value function is expressed as a table of Q-values with an entry for every state-action pair. In every step of the learning algorithm chooses an action to take, computes new state \( s' \), and then updates the corresponding Q-value based on the following formula:

\[ Q(s, a) = R(s') + \gamma \max_{a'} Q(s', a') \]  

where \( a' \) is the optimal action to take in new state \( s' \), given the current Q-table. The discount factor \( \gamma \) is by default set to 0.1 in our implementation.

One challenge we faced was the fact that our state space and action space are both continuous, which makes them unsuitable for the Q-table representation. We solve this problem in the simplest possible manner - by uniformly sampling our state and action spaces, and only keeping Q-values for our sample states and actions in the Q-table. Then we query the table using a simple 1-nearest-neighbor method to determine the state nearest to our query state. While this approach works reasonably well for our small state space, a more scalable formulation of the Q function is an absolute necessity for more complex scenarios.

Our exploration strategy is based on probabilistically choosing which action to take. The probability of choosing a particular action \( a \) in the current state \( s \) is expressed as:

\[ P(s, a) = \frac{k^{Q(s, a)}}{\sum_{i \in A} k^{Q(s, i)}} \]

\( k \) is a constant that is initially set to 1, and we increment it in every learning episode. That way, all actions are initially equally likely, and the algorithm does a lot of exploration. But as learning progresses, higher probabilities are assigned to actions that have higher Q values, and the algorithm becomes increasingly exploitative.

\subsection*{3.2.3. Results}

We trained our locomotion planner on a moderate-size environment with a fixed layout of static obstacles. Locations of the agent and target were randomized at the start of every learning episode.

Once training was done, we tested the performance of the planner on a series of 12 environments of same size, but randomized obstacle layout, target and agent locations. We measured performance as the planner’s success rate in reaching the target, and the time taken to reach it. We compared its performance against a simple greedy search policy, designed to always choose the action with the highest immediate reward.

Our results show that the reinforcement learning-based planner was able to find the target in 91.6% of cases, on average taking 28.40s to do so \((M = 28.40, SD = 16.32)\). On the other hand, the greedy policy was successful in only 41.7% cases, though in the cases when it managed to reach the target, it took on average only 20.35s \((M = 20.35, SD = 13.95)\). Figure 3 shows the character using a greedy policy. It is never able to reach its target. Contrast this with Figure 4, in which the character, using a control policy computed with reinforcement learning, quickly navigates the obstacle-filled environment to reach its target.

Due to limitations of our testing framework and lack of time, we were unable to do more comprehensive testing with multiple iterations and formal hypothesis testing. Even so, these early results indicate clear superiority of the reinforcement learning-based control policy in environment navigation tasks. We observed that the greedy policy was only able to locate the target in simpler scenes, with few obstacles, and the target located close to the initial location of the agent.

Though our reinforcement learning-based planner performs better than greedy search, we did observe a number of issues with it. It often had trouble navigating past a large number of obstacles, and occasionally made decisions that were clearly suboptimal. We believe these issues are caused by a number of factors. One is overly coarse discretization of the state and action space, which was necessary to make our problem scalable, but has been shown to dramatically impair the navigational capabilities of locomotion controllers [LZ08]. Secondly, our state representation may be overly simple. For example, we only encode the relative location of the nearest obstacle in our state representation, even though our agent often finds itself in the proximity of several obstacles at once. Moreover, our reward function is relatively untested and...
its parameters may need to be adjusted further before it serves as a good measure of reward for our navigation problem. Finally, upon inspection we found that many states in our Q-table were never visited, indicating that the learner may be switching from the exploration to the exploitation strategy too quickly. Again, it may be necessary to experiment with different exploration parameter settings to make our learner sufficiently robust.

4. Conclusions

In this paper we discussed the use of reinforcement learning approaches for motion planning, specifically character locomotion. We show, both with a review of prior work and our own implementation, that planning character controllers based on reinforcement learning can synthesize higher-quality motion sequences than controllers using greedy search. Moreover, they are usually able to generate a good motion sequence even in scenarios where greedy controllers fail altogether.

References


