SEPIA: A Scalable Game Environment for Artificial Intelligence Teaching and Research

Scott Sosnowski and **Tim Ernsberger** and **Feng Cao** and **Soumya Ray**
Department of Electrical Engineering and Computer Science
Case Western Reserve University, USA
{sts12, tse6, fxc100, sray}@case.edu

Abstract

We describe a game environment we have developed that we call the Strategy Engine for Programming Intelligent Agents (SEPIA). SEPIA is based on real-time strategy games, but modified extensively to preferentially support the development of artificial agents rather than human play. Through flexible configuration options, SEPIA is designed to be *pedagogically* scalable: suitable for use at the undergraduate and graduate levels, and also as a research testbed. We also describe assignments and our experiences with this environment in undergraduate and graduate classes.

Introduction

It is now well known that games can be used for purposes other than just entertainment. A genre of "serious games" (Eiben et al. 2012) have emerged that use games for problem solving, often through detailed simulations. "Gamification" techniques (Huotari and Hamari 2012) insert game mechanics into everyday tasks, drawing upon human identification of these mechanics as sources of entertainment to motivate participation in or completion of otherwise mundane activities. Likewise in pedagogy, learning through play (Kahn and Wright 1980) is recognized to be an effective strategy in developing skills in children. Computer science is fortunate in that the field naturally gives us the opportunity to use games, in particular video games, as teaching tools. Many students enjoy these games coming into our classes; this foundation can be built on for example by using games to illustrate key ideas, and then, enabling students to learn by programming those concepts in a game environment.

Artificial Intelligence (AI), of course, has long had a close relationship to games even within computer science. Many structurally simple board games such as chess have surprising depth, and this has sometimes led to the ability to play these games as being characterized as "proof-of-intelligence" of a sort. However, such board games are somewhat limited for pedagogical purposes in two ways: first, they may not provide the capability to demonstrate newer and more complex AI techniques, and second, perhaps more crucially, they generally do not pass the "coolness" test for a modern student body. Recent work has therefore looked at the possibility of using more complex games, such as Pac-Man (DeNero and Klein 2010) and Mario (Taylor 2011), to teach AI fundamentals.

In our work, we focus on the genre of *real-time strategy* (RTS) games. We use this genre as a framework to develop a game environment we call the **S**trategy **E**ngine for **P**rogramming **I**ntelligent **A**gents (SEPIA). SEPIA is written in Java and has been coded entirely by students at CWRU. In the following, we motivate the choice of genre, why we decided to create our own environment and how SEPIA fits into the curriculum at the undergraduate and graduate levels. Next, we describe the environment and its capabilities. Finally, we describe possible usage scenarios along with assignments, and document some experiences we have had using SEPIA in our classes over the past year and a half.

Background and Motivation

We first briefly introduce real-time strategy game mechanics. Some examples of these games are Warcraft and Starcraft by Blizzard Entertainment and Age of Empires by Microsoft. Gameplay takes place on a world map. The map contains multiple types of *resources* that can be harvested. Each player possesses units that can harvest resources and build buildings. Buildings support multiple functions, such as serving as resource deposit points, providing upgrades to units or enabling the production of other types of units or buildings. Some units are combat units: they can attack units and buildings belonging to other players, and defend their own units and buildings. Each unit and building possesses *health*, *armor* and a *damage rating*. These factors determine how combat is resolved between units; in particular, once a unit/building's health reaches zero it is destroyed and removed from the world. In addition to these mechanics, most such games include a "technology tree." Some buildings allow specific technologies to be researched that improve the characteristics (e.g. health, armor etc) of units or buildings. Actions such as researching technologies or building units cost resources. Thus successfully playing an RTS requires a delicate balance of three interacting sub-games: the *economic* game where resources are collected and buildings and units built, the *combat* game where own units are defended and enemy units attacked, and the *research* game where new technologies are researched, which potentially provide large benefits but also incur large resource costs. Generally, there is also a "scouting" aspect in these games in that the world map is partially observable: anything outside the sight radius of a player's units is unknown. The general mode-of-play in these games is "last player standing," i.e. the player (or team) that can destroy all opponents wins.

Even from the brief description above, it should be clear that typical RTS games allow a staggering degree of freedom to players. Large numbers of units, buildings and technologies, each with their own strengths and weaknesses, and changing maps with different spatial and resource constraints allows for very complex strategy and tactics to be developed. This makes mastering such games challenging even for human players. At the same time, these games are easy to get into because the mechanics often employ familiar motifs: a sword wielding unit is a melee combat unit, a unit with a bow is a ranged unit, a "gold" resource can be harvested from a mine, "wood" can be cut from trees, etc. Thus these games are "easy to learn but difficult to master," [1] they possess significant replay value and generally do not have pathological characteristics that result in losing because of a minor error at some early stage. For all these reasons, RTS games are very popular; as anecdotal evidence, every student in our current undergraduate Introduction to AI class was already familiar with them.

The same properties that make RTS games challenging to humans also make them a good testbed for AI research. They possess dynamic environments where objects are created and destroyed, large numbers of coordinating objects, complex spatial relationships in partially observable maps, multiple competing agents, the need to do long term decision making with a large number of possible actions at each step, though few are good in the long term, and time/resource constrained reasoning. These are some of the hardest problems in AI. Much recent research has therefore focused on solving fragments of RTS games, e.g. (Buro and Churchill 2012; Weber, Mateas, and Jhala 2011), and there are also AI competitions, e.g. the Starcraft AI competition, that use RTS games as benchmarks to evaluate AI techniques.

While a good solution to full RTS games remains beyond the reach of current AI techniques, it is important to note that, for pedagogical purposes, it is easy to simplify the setting. For example, we might disable partial observability or the technology tree, or disable specific units or buildings, or alter the victory conditions. In this way, we can construct a variety of scenarios which can be tackled by simple agents. Thus, we believe this domain has the property of being "pedagogically scalable:" small changes to the domain mechanics can result in a problem sequence that varies in difficulty. This allows us to use this domain through our undergraduate and graduate classes, with increasingly difficult scenarios and increasingly complex algorithms being implemented. For students going on to take graduate AI classes, a key benefit is the amortization of the initial cost of having to become familiar with a complex environment. This is even more the case for students who go on to do AI research in reinforcement learning and planning. The ability to create a rich sequence of problems of varying difficulty, scaling from basic AI concepts to research level development, was a key driver behind our decision to focus on the RTS domain.

[1] A quote due to Nolan Bushnell, the creator of *Pong*.

While part of the pedagogical motivation for using SEPIA came from the undergraduate and graduate AI classes we teach, CWRU also offers a minor in Computer Gaming. In this minor, students from our department along with students from the Cleveland Institute of Art team up to learn how to design interactive games, with a final project that involves creating a full game, which is judged by representatives from a large game studio. These students are required to take the Introduction to AI class. Thus teaching AI techniques in the context of a game is directly relevant to these students.

A question one might ask is why we chose to make our own environment rather than use one that was already available. Indeed, the Stratagus open source RTS engine is available for use and we had some prior experience using it. Likewise, the Starcraft AI competition makes available an API to interact with Starcraft. There are several issues with these systems, however. A major issue is that these systems are primarily geared towards human play, rather than AI development. This results in a different set of priorities. For example, detailed attention might be given to the rendering pipeline, which is useless to an AI agent, but insufficient support might be available for simulation, which is essential to most modern AI techniques (e.g. Monte Carlo planning (Kocsis and Szepesvari 2006)) applied to such settings. While it is possible to at least superficially disable the rendering pipeline in some cases, the complexity of the code means it is unclear if internal rendering operations are still being carried out (and finally discarded). Secondly, the environments do not come with experimental infrastructure that one might want to use to evaluate different AI techniques. Third, a variety of practical issues tend to arise when interacting with the existing code: for example, while an API might exist, the underlying engine might be proprietary, leading to "inherent" partial observability and lack of understanding of the precise mechanics. Similarly, in some cases communication with the underlying engine has to be done through TCP, which causes bottlenecks due to large amounts of state information needing to be passed. Finally, these environments may not be very flexible: altering the game mechanics or configuration might require modifying the engine, which is problematic .

We also briefly considered if the RL-Glue (Tanner and White 2009) interface could be useful in this work. However, the goal of that project is to provide a uniform interface to a variety of different domains. Here, we are concerned only with a single domain, so we forgo using RL-Glue in favor of tighter integration with the underlying engine.

For the reasons above, we decided to design an RTS environment that would be highly configurable and would provide detailed support for AI development primarily focused on teaching and research. In the following, we describe the capabilities of SEPIA in detail.

The SEPIA Environment

In order to run, SEPIA takes one input, a configuration file in XML (Figure 1 left). This file specifies the world map to load and for each player on this map, the file specifies the agent class controlling that player and any parameters

```
<Configuration>
  <Map>data/map.xml</Map>
  <Player Id="0">
    <AgentClass>
     <ClassName>edu.[xyz].sepia.agent.visual.VisualAg
     ent</ClassName>
       <Argument>true</Argument>
       <Argument>false</Argument>
    </AgentClass>
  </Player>
  <ModelParameters>
    <Conquest>true</Conquest>
    <Midas>false</Midas>
    <ManifestDestiny>false</ManifestDestiny>
    <TimeLimit>1000</TimeLimit>
  </ModelParameters>
  <Runner>
    <RunnerClass>edu.[xyz].sepia.experiment.Exampl
     eRunner</RunnerClass>
    <Parameter Name= "experiment.NumEpisodes"
     Value="5"/>
  </Runner>
</Configuration>
```

```
<state xExtent="15" yExtent="15" fogOfWar="false">
  <player>
    <ID>0</ID>
    <unit>
      <ID>0</ID>
      <currentHealth>60</currentHealth>
      <xPosition>2</xPosition>
      <yPosition>3</yPosition>
      <templateID>4</templateID>…
    </unit>
    <template xsi:type="UnitTemplate">
      <ID>4</ID>
      <goldCost>600</goldCost>
      <name>Footman</name>
      <baseHealth>60</baseHealth>
      <baseAttack>6</baseAttack>
      <armor>2</armor>…
    </template>
    <resourceAmount>
      <quantity>100</quantity>
      <type>GOLD</type>
    </resourceAmount>…
</player>
</state>
```

Figure 1: **Left:** Fragment of an XML configuration file input to SEPIA. It specifies the map, the agents that will run each player, game parameters including victory conditions and time limits, and the experiment runner. **Right:** Fragment of an XML map file. This map is 15-by-15 and fully observable. In position (2,3) there is a unit belonging to Player 0; this unit is a "footman" (a melee combat unit). Maps can be created as text or using a graphical map editor.

needed by that class. Multiple agent classes sharing the same player ID play on the same "team" and have access to all units in that team. The configuration file also specifies victory conditions; in addition to the standard victory conditions, SEPIA also supports a variety of other conditions such as resource and unit totals. Finally, the configuration file specifies the experiment being run and the specific environment model to be used. Typically, this will specify episodic gameplay for a certain number of episodes. The environment model affects the underlying mechanics. Using a suitable environment model, SEPIA can (i) switch durative actions on and off, (ii) modify sequencing between agents every epoch to be turn-taking (agents are processed in round-robin order) or simultaneous (all agent actions are collected and simultaneously applied to the state) and (iii) if sequencing is simultaneous, determine how consistent next states result when agent actions interfere with each other. Here "consistent" means an agent with full knowledge of the current state, action models and its own action would be able to predict the resulting state with nonzero probability (intuitively, this means actions do not have "unexpected" outcomes from the agent's perspective. In particular this means interfering actions have to be handled systematically.) Similar consistency criteria are applied to units available to a single agent. The configuration file also specifies whether decision making is real-time. If so, each agent is given a specified amount of time each epoch to propose a set of actions. Otherwise, an agent may take as much time as it needs.

The map specified in the configuration file is also stored in XML format (Figure 1 right). The map structure specifies whether it is fully or partially observable, and a list of units belonging to each player and their initial locations and conditions. Static unit characteristics (that do not change between maps, like armor) are filled in using a unit template (also a text file). Modifying the unit template allows the creation of new types of units. Maps can be created in a text editor or in a graphical map editor we provide with SEPIA.

Once the configuration and map are loaded, SEPIA will load the agents that will play the scenario. All agents in SEPIA implement a standard Agent interface. Using this interface, students can write their own agents and compile them without having to recompile the entire environment, as long as the jar file containing the SEPIA engine is in the classpath. Running a user-written agent is possible simply by specifying it as the agent class in the configuration file. We provide several simple agents as examples, including a simple agent that collects resources, a simple agent that combats enemies and a scripted agent that runs a sequence of commands written in a simple scripted language.

Each agent in SEPIA runs in its own thread. In each epoch, the engine provides all agents the current state (modified for partial observability if needed). After processing, each agent supplies the engine with actions for their units. SEPIA supports compound actions (e.g. move to a unit, then attack) that are resolved to primitive actions with an internal planner. The list of primitive actions for the current time

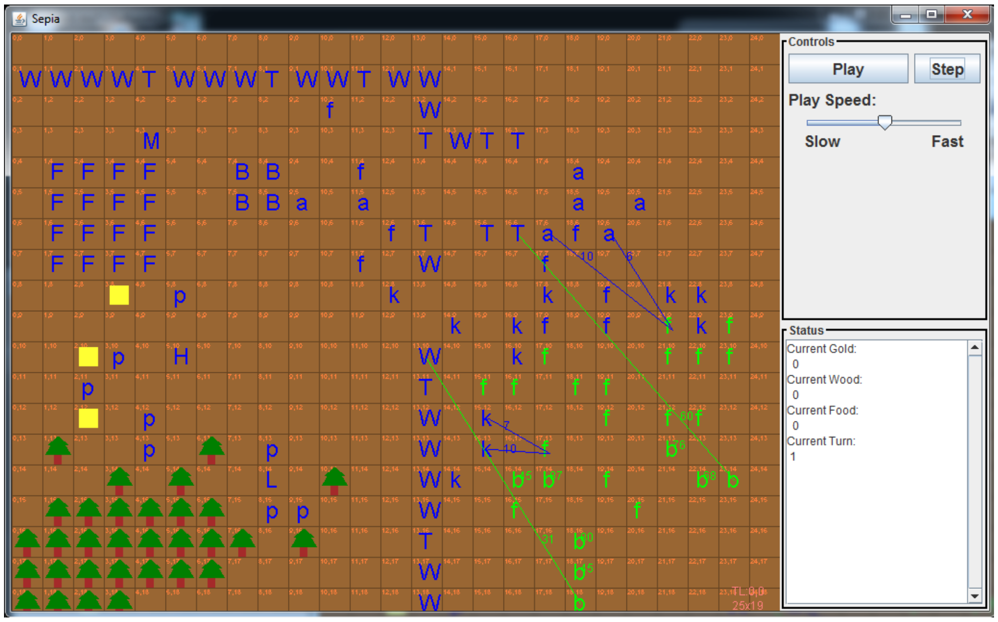

Figure 2: Screenshot of SEPIA's graphical interface, VisualAgent. Each letter or symbol denotes a unit or building, for example "W" denotes a "wall" and "k" denotes a "knight." Lines denote targets of targeted actions. There are two players (colored blue and green). This map is fully observable, so the entire map is drawn.

step are then executed using an execution order determined by the choice of model.

SEPIA comes with a graphical interface (Figure 2). This interface is also an agent, VisualAgent, that can be attached to any other agent playing a scenario in the configuration file. VisualAgent then shows the scenario as observed by that agent. Multiple VisualAgents, each showing the map from a different agent's perspective, can be run; this can be useful for debugging. The VisualAgent also allows human control of units with the keyboard and mouse, with appropriate arguments in the configuration file. If a VisualAgent is not specified in the configuration, nothing is displayed graphically by SEPIA. However it is still possible to track what agents are doing through log messages, which open in their own window when an agent writes to their log.

Many modern AI techniques for decision making, including Monte Carlo planning techniques, rely on lookahead-based simulation to work. SEPIA provides support for such algorithms as follows. An agent can internally *create* an environment model using the current (possibly best-belief) state. In this simulation model the agent can act, dictate other agents' actions and observe the consequences of its choices. These actions do not affect the "real" model that is running the game. Observations in these simulated models can then be used to update policies. It is also possible to attach VisualAgents to these simulated models, showing the "thought process" of an agent.

As a final aid to agent development, SEPIA supports detailed history and save functionality. Saves can be of three types: (i) the game (map) state, (ii) the agent state, which can include learned information, such as Q-tables or policies and (iii) history information, which stores the sequence of actions executed in the scenario. Storing each type of information separately has many benefits. Storing the map state separately allows different agents to play the same scenario. Storing the agent state allows knowledge transfer as the same agent plays different scenarios. Finally, history information enables action replay, which is a known technique to speed up convergence in temporal difference methods like Q-learning (Lin 1993). History and agent state are saved as Java serializable objects.

Along with history, SEPIA also supports detailed logging functionality within the game to support debugging agents, and provides descriptive feedback to agents about the status of actions being carried out. For example, if an action failed, the return code describes why: maybe the unit's path was blocked, or the unit or action's target was destroyed, etc.

We note a limitation of the current environment: it does not provide explicit support for communication between competing agents (collaborative agents share the state and action information). This means in particular that SEPIA does not explicitly support features like "diplomacy" and negotiation that are features of some RTS settings.

SEPIA is distributed as a single jar file. As we mention above, agents can be written, compiled and run so long as the jar file is in the classpath; the engine itself needs no compilation at those points. This is useful for classroom usage.

Using SEPIA: Assignments and Experiences

We have used SEPIA in two offerings of graduate AI and one offering of undergraduate AI so far (we are using it again in undergraduate AI this semester). In this section we outline the types of assignments we use in these classes and discuss some of the feedback we have received from students.

The undergraduate course is a standard introduction to AI using the Russell and Norvig textbook, "Artificial Intelligence: A Modern Approach", 3ed. The broad areas we cover are (i) problem solving with search, (ii) propositional and first order logic and automated planning, (iii) probabilistic reasoning, Bayesian networks and machine learning and (iv) Markov decision processes and reinforcement learning. There are five programming assignments associated with these topics: two on search, and one for each of the three others. Each assignment is focused on developing an agent for SEPIA that plays a scenario using concepts explained in class. At the start of the semester, we give a tutorial on writing agents for SEPIA. Students are allowed to do assignments in pairs if they wish.

The first assignment consists of two parts. The first part is meant to familiarize students with the SEPIA environment. Here, students write an agent that is able to take a variety of simple actions in the environment including collecting some resources and defeating an enemy unit. To facilitate the familiarization process, demonstration agents are provided from which the basic agent skeleton as well as API calls to get state and status information and execute actions are easy to figure out. In the second part of this assignment, students write an agent that can plan paths using A* search on fixed maps, with a given heuristic function. SEPIA internally uses A* for path planning (with the same heuristic). Students cannot use calls to that SEPIA code to do their assignment (obviously), but they can check their solution against the solution returned by SEPIA using VisualAgent. In this assignment, the average grade of the students was 94.5 with a deviation of 10.9.

In the second assignment, students implement alpha-beta pruning on game trees to solve a "chase-and-capture" scenario on a fully observable map. Here, some ranged enemy units are fleeing from some melee friendly units. The friendly units need to coordinate and plan the best sequence of moves to capture the enemy units. The ranged units are controlled by a provided fixed agent that balances running away, shooting their enemies if they are safe and staying away from walls. Since the game tree is too big to explore exhaustively, a state evaluation function is needed. In this assignment, the students are allowed to write their own evaluation functions. In this assignment, the average grade was 97.5 with a deviation of 8.0 (36 students total).

In the third assignment, students write an automated planning system to collect resources to meet a target resource goal using a single resource collection unit (called a Peasant). The planner is a simple state-space planner that uses forward search to find a plan. The A* code from the first assignment can be reused to solve this assignment. A second part of this assignment involves the more challenging case where additional Peasants (up to three) can be built. Building a Peasant moves the planner farther from the resource goal, but offers opportunities for parallelizing the remaining plan and thereby shortening the makespan. As in the second assignment, students are allowed to write heuristics for the planner. In this assignment, the average grade of the students was 83.3 with a deviation of 18.2.

In the fourth assignment, the students implement probabilistic inference to guide Peasants around on a map to collect resources. This map is partially observable; several locations contain enemy Towers that are static ranged units that shoot arrows. Towers see Peasants before Peasants can spot Towers. Solving the full POMDP is impractical at this stage; however a simple myopic strategy suffices in which the students maintain the probabilities that each location has a Tower, update the probabilities based on observations, and perform inference to make myopic decisions on the best move by a Peasant. The students start with multiple Peasants, and win by collecting a target amount of resources. Each map provided in this assignment has at least two completely safe paths, and we also provide fully observable versions of these maps so students can check their solutions. In this assignment, the average grade of the students was 90.8 with a deviation of 12.6.

In the final assignment, students implement Q-learning with linear function approximation to solve a combat scenario where a number of melee units fight each other. A fixed enemy agent is provided that occasionally does silly things, tuned so that human players (grad students) can beat it about 75% of the time. Several useful features are provided to the students to help construct the representation of the Q-function, and students are encouraged to design other useful features on their own. To keep things tractable, the set of actions is limited simply to Attack actions (there is no option to move to arbitrary locations, run away in fear, etc). In this assignment, the average grade of the students was 86.1 with a deviation of 8.3.

SEPIA was very favorably received in our undergraduate class last year (the first time we used it at this level). Prior to Spring 2012, we used "standard" programming assignments; for example, the programming assignment that implemented search used the eight-puzzle. In the student evaluations for these classes, there was no special mention of these assignments. Last year, however, most students who responded in the course evaluations praised SEPIA. Some comments we received:

- "The programming assignments, while difficult, were by far the most enjoyable part of the course. It's cool to see what you are taught in action."
- "[SEPIA] was a great learning tool - not only is it a fun platform for testing algorithms, it also has really interesting coding."
- "The programming assignments, for the most part, are enjoyable. [SEPIA] is a good way to see the concepts we learn in class play out."

Five students in the class (14%) then went on to take the graduate level AI and machine learning classes as electives in the Fall semester.

The course evaluations (conducted at the university level) do not have any separate questions on programming assignments, and this course does not formally have a "lab," so most students responded N/A to a question about the lab. The overall course rating was 3.22 ($n = 18$), as opposed to a rating of 3.0 ($n = 15$) for Spring 2010. [2] Further, while

[2]For Spring 2011, we experimented with two faculty members

for Spring 2010, no respondents rated the course as "excellent," in 2012 28% rated it as such. However, the fraction of "poor" ratings also increased from 13% to 22%, possibly indicating the increased difficulty of the assignments and teething troubles with SEPIA that caused some frustration.

Among the issues faced by students was the need for better documentation. Some students struggled with the API because they were not able to find the functions they needed to work with the state or did not fully understand what the functions did. However, based on student feedback, we also realized that some functionality involving the ability to copy objects storing the state was missing in SEPIA's API, so students had to implement this, which led to some frustration. We have since implemented this functionality in SEPIA. We also found some bugs in the implementation that we subsequently fixed. Fortunately, these were all minor bugs, easy to work around, so we did not have to patch SEPIA within the course timeline. Everything considered, we feel that SEPIA served well as an interesting and fun environment to demonstrate AI algorithms at this introductory level, and hope to continue the process this year.

We now discuss SEPIA in our graduate AI course. This course is divided into two parts; the first covers graphical models while the second focuses on reinforcement learning and planning. SEPIA is used for two programming assignments in the second part. One assignment deals with advanced planning. The scenario is again a resource collection scenario, but in this case the actions are durative and an arbitrary number of new Peasants can be constructed along with Farms, which provide food to support additional units. Further, the resource targets are very large and require long plans to fulfill. The students implement an online planner that also searches for renewable resources while planning (Chan et al. 2007). The second assignment is on reinforcement learning and again focuses on a combat scenario. In this case, the students implement a policy gradient approach based on GPOMDP (Baxter and Bartlett 2001) to solve the scenario.

We have had a somewhat more mixed response to SEPIA in the graduate class. A key issue seems to be that although our undergraduate class is a pre-requisite for the graduate class, a number of students in the class do not possess it (for example if they were international students who took a similar class elsewhere). This means they are unfamiliar with the SEPIA environment in this class. The pace of this class is more rapid than the undergraduate class, and we have not offered tutorials and introductory assignments so far. We plan on changing this, so that it is easier for incoming graduate students to get accustomed to SEPIA.

On the whole, we feel that SEPIA has great potential to be an interesting environment where students can learn AI concepts by programming agents that can play a contemporary video game. We plan to continue to develop it further, and make it available to other instructors for their use.

co-teaching the course. For this year, the overall course rating was 2.88. However, this may have been an artifact of the co-teaching, so we do not compare to this year.

Conclusion

We have introduced SEPIA, a game environment for AI teaching and research. SEPIA is based on real time strategy games, but modified for artificial agent development with support for simulation, different execution strategies, detailed logging and action replay among other features. We have used SEPIA in both undergraduate and graduate classes and received encouraging feedback from students, particularly in the undergraduate class. We feel that SEPIA provides a rich and fun environment to learn AI concepts, and plan to continue adding features to it and developing interesting scenarios that can be used to demonstrate how AI algorithms work in practice.

References

Baxter, J., and Bartlett, P. L. 2001. Infinite-horizon policy gradient estimation. *Journal of Artificial Research* 15:319–350.

Buro, M., and Churchill, D. 2012. Real-time strategy game competitions. *AI Magazine* 33(3):106–108.

Chan, H.; Fern, A.; Ray, S.; Wilson, N.; and Ventura, C. 2007. Online planning for resource production in real-time strategy games. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 65–72.

DeNero, J., and Klein, D. 2010. Teaching introductory artificial intelligence with pacman. In *Proceedings of the First Educational Advances in Artificial Intelligence Symposium*.

Eiben, C. B.; Siegel, J. B.; Bale, J. B.; Cooper, S.; Khatib, F.; Shen, B. W.; Players, F.; Stoddard, B. L.; Popovic, Z.; and Baker, D. 2012. Increased diels-alderase activity through backbone remodeling guided by foldit players. *Nature Biotechnology* 30:190–192.

Huotari, K., and Hamari, J. 2012. Defining gamification - a service marketing perspective. In *Proceedings of the 16th International Academic MindTrek Conference*.

Kahn, J., and Wright, S. E. 1980. *Human growth and the development of personality*. Pergamon Press.

Kocsis, L., and Szepesvari, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the 2006 European Conference on Machine Learning*. Springer.

Lin, L. 1993. *Reinforcement Learning for Robots using Neural Networks*. Ph.D. Dissertation, Carnegie Mellion University.

Tanner, B., and White, A. 2009. Rl-glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*.

Taylor, M. 2011. Teaching reinforcement learning with mario: An argument and case study. In *Proceedings of the Second Educational Advances in Artificial Intelligence Symposium*.

Weber, B.; Mateas, M.; and Jhala, A. 2011. Building human-level ai for real-time strategy games. In *AAAI Fall Symposium on Advances in Cognitive Systems (ACS)*.