# Evolving Dota 2 Shadow Fiend Bots using Genetic Programming with External Memory

Robert J. Smith
Dalhousie University, Halifax, Nova Scotia, Canada

Malcolm I. Heywood
Dalhousie University, Halifax, Nova Scotia, Canada

## ABSTRACT

The capacity of genetic programming (GP) to evolve a 'hero' character in the Dota 2 video game is investigated. A reinforcement learning context is assumed in which the only input is a 320-dimensional state vector and performance is expressed in terms of kills and net worth. Minimal assumptions are made to initialize the GP game playing agents – evolution from a tabula rasa starting point – implying that: 1) the instruction set is not task specific; 2) end of game performance feedback reflects quantitive properties a player experiences; 3) no attempt is made to impart game specific knowledge into GP, such as heuristics for improving navigation, minimizing partial observability, improving team work or prioritizing the protection of specific strategically important structures. In short, GP has to actively develop its own strategies for all aspects of the game. We are able to demonstrate competitive play with the built in game opponents assuming 1-on-1 competitions using the 'Shadow Fiend' hero. The single most important contributing factor to this result is the provision of external memory to GP. Without this, the resulting Dota 2 bots are not able to identify strategies that match those of the built-in game bot.

## CCS CONCEPTS

• **Computing methodologies → Reinforcement learning**; **Genetic programming**.

## KEYWORDS

Dota 2, Genetic programming, Reinforcement learning, External Memory, Partial Observability, Coevolution

**ACM Reference Format:**
Robert J. Smith and Malcolm I. Heywood. 2019. Evolving Dota 2 Shadow Fiend Bots using Genetic Programming with External Memory. In *Genetic and Evolutionary Computation Conference (GECCO'19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3321707.3321866

## 1 INTRODUCTION

Dota 2 is a multiplayer online battle arena video game in which two to five players challenge each other through simultaneously defending their base / attacking the opponent's base. To do so, the player controls a 'hero' character, where heroes posses specific abilities, resulting in different performance tradeoffs. Moreover, a hero does not act alone, but operates with a team of 'creeps' that engage the opponent's creeps (and hero) through pre-defined behaviours. This means that a hero has to operate collaboratively with its own creeps and defensive structures called 'towers' to win a round of the game (destroy an opponent's tower). The game also has an economy in which wealth is collected and traded, so improving the capabilities of their hero, thus decisions also need to be made regarding when to trade and what to trade for.

The game has multiple properties that make it a particularly interesting testbed for reinforcement learning, even in 1-on-1 play against a single opponent (the case considered here). Game state is expressed as a map in 3-D isometric perspective. Moreover, each team is subject to partial observability, implying that they only see what lies in their respective lines-of-sight. We demonstrate that genetic programming (GP) is able to develop policies for a 'Shadow Fiend' hero character sufficient to reliably defeat the predefined opponent Shadow Fiend teams from a 'tabula rasa' starting point. By tabula rasa we imply that: 1) a generic instruction set is assumed, thus no attempt is made to tailor the instruction set to the task; 2) performance feedback is only provided at end of game and reflects the 2 properties a player experiences (kills, net worth) and basic skills of 'last hits' and 'denying', 3) state is defined by a 320-dimensional attribute vector subject to partial observability, 4) 30 different actions are possible (where this represents a minimalist set of actions available to the Shadow Fiend character at initialization).

We demonstrate that support for external memory is particularly important, without which agents are unable to evolve any useful behaviours at all. Conversely, when external memory is provided, GP spontaneously identifies appropriate strategies for navigation and engagement with the competing hero's team. In short, evolving bots for Dota 2 is a challenging problem in which the environment is characterized by high-dimensional spatial-temporal information of a non-stationary, partially observable nature. This research represents an investigation of some of these complexities, from the general perspective of demonstrating that *plausible agent behaviours* can be evolved without recourse to forward models (MCTS), game specific 'cheats' or assuming some form of neural representation.

## 2 RELATED RESEARCH

Real-time strategy games such as StarCraft [14] and Dota 2 [5] represent one of the most recent challenges posed for learning agents. As the number of potential decisions increases, learning agents now have to operate cooperatively with other bots on their team in order to defeat the opposing team. Game state is also incomplete and expressed using a vector of egocentric parameters (up to 20,000 in the case of Dota 2). In addition, there are multiple means by which you can win/lose a game, thus making it more difficult to successfully play the game using reactive behaviours alone, i.e.

strategy is important. StarCraft has seen a significant number of bots developed and competitions taken place on a regular basis (e.g. [14, 21]). Specific examples include bots using hybrid combinations of AI techniques (e.g. [14, 21]), Bayesian models [19] and GP alone [6], with the latter making extensive use of instructions specific to the task and a prior game specific decomposition.

In this work, we concentrate on the case of Dota 2, in part because: 1) a bot interface has recently became available [5], 2) the 1-on-1 mid game competition represents a well parameterized starting point for an attempt at tabula rasa agent learning (See § 6), 3) Dota 2 bots have been demonstrated using a form of neuroevolution[1], but only with extensive computational support – 180 years of gameplay per day are generated through self-play, requiring 128,000 CPU cores and 256 GPUs. Conversely, we are interested in evolving agents capable of playing the bots from the game engine with limited computing infrastructure (i.e. a single desktop).

## 3 THE DOTA 2 1-ON-1 MID-LANE TASK

The Dota 2 environment is defined in terms of a map in 3-D isometric perspective, however, all the agents playing the game are subject to sensor information constrained by partial observability (line-of-sight/fog-of-war). Thus, from an agent's perspective it is not possible to see 'through' a forest, or above a flight of stairs. Game play in Dota 2 revolves around developing strategies for defining the behaviour of a 'hero' character. There are three basic categories of hero (strength, agility, intelligence) resulting in a total of 117 specific heroes.[2] Indeed, when playing Dota 2 in teams, one of the most important decisions is what combination of heroes to assume. We will not be addressing the hero selection issue in our work. Instead, we will assume a specific hero from the agile category – a Shadow Fiend – where the agile category provides heroes with the potential to perform well in a broad range of gaming scenarios [15]. Future research will consider whether we can evolve heroes from the other categories.

There are two teams, 'The Radiant' represents our team (base at the bottom LHS, Figure 3) and 'The Dire' represent the opponent team (base at the top RHS, Figure 3). The base of each team periodically issue waves of 'creeps' which are simple bots with default behaviours. The creeps advance from each base down each 'lane', where the number of active lanes is a game parameter (the more active lanes, the more difficult the game). In this work, we are interested in the single mid-lane case, thus creeps will advance from each tower down the leading diagonal (Figure 3). When the creeps meet at the river feature in the centre of the map, they attack each other. Note that the river is lower than other features, hence once in the river, bots cannot see beyond the river. The hero character for each team attempts to use their powers to develop a strategy to enable them to advance specific aspects of the game to their advantage. Examples might be to support the creep attack or collect powerup and bounty runes from the environment (increasing team wealth and therefore supports the collection of items which provide additional powers). In addition, creeps also wander through the forest areas, engaging in heroes attempting to bypass the mid-lane. A game ends when one side either kills the opponent hero twice, or one tower is taken over.

The goal we set in this work is to determine whether we can evolve behaviour for the Shadow Fiend hero against the built in Shadow Fiend hero bot under the 'mid-lane' parameterization. We will assume that the opponent hero bot operates under the hard as opposed to easy or medium level of difficulty. We will then test the resulting evolved Shadow Fiend hero against all three settings for the opponent hero bot. This is not a straightforward goal, as we are potentially subject to several pathologies: 1) fitness disengagement – the opponent already has a challenging strategy, implying that we are not able to establish a useful gradient for directing evolution [4]. 2) sparse fitness function – many skills need to be attained in order to develop effective hero strategies, including navigation, defending, attacking and collecting bounty. Some of these properties have no direct reward (i.e. navigation) and others are only indirectly quantified in the fitness function (i.e. defending and attacking). Moreover, establishing an effective strategy for navigation facilitates the development of many other abilities, but it is not characterized directly in the fitness function.

## 4 TANGLED PROGRAM GRAPHS

In this work, we will assume the Tangled Program Graphs (TPG) framework [9], on account of its success at discovering light weight emergent solutions to ALE [10] and ViZDoom [16] tasks directly from the frame buffer. TPG achieves this by supporting multiple forms of modularity, beginning evolution with teams of programs (i.e. single node graphs), and incrementally discovering connections between teams to compose solutions as graphs of teams (of programs). In the following we summarize the TPG approach, and refer readers to [10, 11] for a more detailed characterization.

### 4.1 Programs

Programs express two properties, an *action* and the *context* for its action. To do so, TPG assumes an earlier formulation – Bid based GP [13] – in which context is the single output from the program after execution (e.g. the value at the root node of Tree structured GP). The program's action, $a$, is a scalar assigned at program initialization from the set of application specific atomic actions, $a \in \mathcal{A}$. Thus, a program always has the same action, but its context (i.e. program output) will vary as a function of the input. This also means that a program has to operate with other programs in order to specify a non-trivial behaviour (§ 4.2).

Without loss of generality, we assume a linear GP representation [2], with instructions defining operands (target, source1 (= target), source2) and an opcode $\langle op_i \rangle$. Table 1 summarizes the instruction set, which is unchanged since its original use for classification tasks [13]. A mode bit is also employed to enable the source2 field to support 3 addressing modes: 1) Register-Register – a set of registers specific to the program, 2) Register-Input – inputs characterizing state of the task $\vec{s}(t)$, or 3) Register-Memory – a reference to an external memory location (§ 5).

### 4.2 Nodes

Programs on their own are not capable of expressing a useful behaviour because their action is a single unchanging atomic action,

[1]https://openai.com/five/
[2]https://dota2.gamepedia.com/Heroes

**Table 1: Relation between opcodes and operands. Register-Register instructions define operands in terms of register references alone, or $x, y \in \{0, ..., R_{max} - 1\}$. Register-Input and Register-Memory references introduce different ranges for the *y* operand: Register-Input $y \in \{0, ..., N - 1\}$, Register-Memory $y \in \{0, ..., M - 1\}$. *N* is the number of pixels in the input, and *M* is the number of indexible locations in external memory (§ 5). *ln* and *exp* are (protected) natural logarithm and exponential operators (e.g. absolute value of the operand is assumed) and NaN is trapped in the case of division.**

| Opcode | Instruction |
|---|---|
| $\langle op_0 \rangle \in \{<\}$ | IF $R[x]\langle op_0 \rangle R[y]$ THEN $R[x] = -R[x]$ |
| $\langle op_1 \rangle \in \{cosine, ln, exp\}$ | $R[x] = \langle op_1 \rangle (R[y])$ |
| $\langle op_2 \rangle \in \{+, -, \div, \times\}$ | $R[x] = R[x]\langle op_2 \rangle R[y]$ |

*a*. Thus, a valid solution has to consist of a team of at least two programs with different actions. Moreover, we do not know what constitutes useful team complement, so this should be an evolved property. To do so, TPG makes use of a symbiotic model of evolution in which two populations are coevolved, the team (or node) population ($N$) and the program population [13]. The node population defines each individual in terms of a unique set of pointers to programs from the program population, so multiple node individuals can index the same program more than once. The number of pointers per individual is initialized to the value $\omega$ and is free to evolve under the action of the variation operators: $P_d, P_a, P_m$ for deleting, adding or (cloning and then) modifying a program.

Fitness is only defined for individuals in the Node population. Evaluation of node $i$ executes all programs indexed by this node ($p_j \in$ node$(i)$) under the current state from the environment, $\vec{s}(t)$. If there are 5 programs there will be a corresponding vector of 5 outputs, one from each program. However, only the program with the largest single output 'wins' the right to suggest its action, which is defined in terms of one of the task specific atomic actions, $a \in \mathcal{A}$ (atomic actions are defined in § 6).

Under reinforcement learning tasks each node denotes the policy for a decision making agent operating under delayed rewards. At each time step, $t$, the agent is presented with a corresponding state, $\vec{s}(t)$, and the agent's policy identifies an action. As the agent is embedded in a gaming environment, the agent's action will result in a change as recorded by the next game state, $\vec{s}(t+1)$. At this point one of two basic outcomes can happen, either an end-of-game state is encountered such as winning the game or a limit to the number of game interactions has been reached ($t = \tau$), or evaluation of the policy represented by node$(i)$ can continue.

On encountering an end-of-game state, the game score is used to define fitness (see § 6) for the agent (i.e. fitness of node$(i)$). Fitness evaluation is then repeated over all nodes, with the worst *Gap* nodes deleted at each generation. Variation operators stochastically clone *Gap* surviving nodes and the genetic content of the cloned nodes varied. However, before the cloning step (but after the worst *Gap* nodes deleted) any programs that are not associated with a Node are deleted, thus the size of the program population is free to float.

## 4.3 Graphs

At this point we have a population of nodes where each node is a unique combination of pointers to individuals in the program population, and we have a way of evaluating each node under a reinforcement learning task (§ 4.2). Over each generation, nodes will be rewarded for finding useful complements of programs for either: solving the entire task out-right (i.e. evolutionary cycle finishes with a champion node), or a subset of nodes that address different aspects of the overall task are identified. The TPG algorithm provides a mechanism for stitching together the different node behaviours in a context specific way [10, 11]. Thus, the end result is a graph of nodes (of programs) in which an overall agent policy is expressed.

The key mechanism by which this is achieved in an emergent way is through the operation of the variation operators. After cloning *Gap* individuals from the node population the probabilistic application of deletion/addition operators mix the node's program complement, while the $P_m$ operator identifies a program for cloning and modification (i.e. introduces a new program into the program population). This latter step results in the application of a set of variation operators to the cloned program, one of which tests for changes to the program's action, $p_{nm}$. When $p_{nm}$ tests true, then TPG tests for what *type of change* to introduce. Thus, if $rnd(0, 1] < p_{atomic}$ tests true, then a different atomic action is selected, otherwise an action is selected from the set of nodes currently in the node population. In the latter case, the action has became is a pointer to another node, hence the node in question is evaluated under the same environmental state. The process repeating recursively until an atomic action is returned [10, 11].

TPG also introduces a scheme for marking nodes visited during the evaluation of the same agent. This is necessary in order to ensure that infinite loops around a graph are not encountered. Thus, if a node in the same graph represents the winning action, but was previously evaluated, then the action of the runner up program would get to suggest the action instead. As long as each node has one program with an atomic action, TPG will guarantee that an atomic action can be identified no matter what state is encountered.

## 5 MEMORY MODEL

TPG as deployed by earlier work has assumed a purely reactive formulation (e.g. [9–11, 16]). That is to say, the ALE suite of game titles define state in terms of complete information. Thus, strong performance is possible without having to formulate approaches to reinforcement learning that are non-Markovian. However, once the task displays significant amounts of partial observability, it is necessary to build policies that are capable of recording previous state. Deep learning approaches have been pursuing a combined approach, using both recurrent connectivity about specific subsets of neurones (e.g. long-short term memory) and external memory (e.g. some form of indexed memory), see for example [8]. We aim to provide a parallel set of capabilities through support for scalar stateful memory and external indexed memory using a recently proposed probabilistic model of memory [17].

### 5.1 Scalar stateful memory

Scalar stateful memory characterizes the state of registers associated with each program [7]. Let $\mathcal{R}_i(t)$ denote the value of each of the $R_{max}$ registers associated with program $i$ at interaction $t$ with the environment. Execution of program $i$ will modify state as expressed by $\mathcal{R}_i(t)$. Thus, in a non-Markovian task, $\mathcal{R}_i(t)$ potentially carries useful information between states. That said, Scalar stateful memory has two basic limitations: 1) any instruction can modify some aspect of $\mathcal{R}_i(t)$ state, making it more difficult to retain long term properties (high likelihood of disruption), and 2) $\mathcal{R}_i(t)$ state is specific to each program, thus it is not possible to pass 'memory' between the multiple programs that make up TPG individuals.

## 5.2 External indexed memory

External indexed memory is synonymous with the manner in which a CPU perceives 'global' memory, i.e. with specific read/write instructions. Previous research has considered the role of indexed memory in the evolution of basic data structures [12, 20]. Others considered the evolution of 'mental models' in which a one-to-one relation between memory and $4 \times 4$ grid world was assumed [3]. A two phase evolutionary cycle was also required, with cycle 1 only allowed to write content and cycle 2 rewarded for reading back the relevant content. Such a two cycle process assumes that the agent is able to navigate the environment in the first cycle, whereas in the tasks posed in this work, memory is a pre-requisite for navigation. External memory models have also been proposed for use with simple robot controllers e.g. [1]. In this case, a performance measure was needed in order to define what memory content was actually saved and the criteria for replacement. Moreover, each write operation wrote the entire state space to memory. Neither would be feasible in this work.

One theme we do explicitly develop is the concept of retaining a single 'instance' of external memory [18]. From our perspective this implies that the state of external memory, $\mathcal{M}$, is *never* reset. Thus, each TPG agent inherits the external memory state as left from the previous agent interaction. Our motivation being to ensure that all agents evolve to a common/shared concept for what constitutes useful memory content, and where to find it. Thus, only at the evaluation of the first TPG on the first generation does $\mathcal{M}$ = NULL.

External memory, $\mathcal{M}$, adopted in our approach is formulated to provide a probabilistic write operation and indexed read [17]. We assume that the data written to memory takes the form of the vector of register state, $\mathcal{R}_i(t)$, from program $i$ at the point where the $Write(\mathcal{R}_i(t))$ instruction is executed.[3] Thus, we assume that register values are a suitable low dimensional encoding for articulating what the state of the agent is, as opposed to $\vec{s}(t)$.

A write operation also has to define were in memory to write data. In our case we assume a probabilistic model in which the vector $\mathcal{R}_i(t)$ is distributed across $L$ columns of external memory [17]. Thus, from the perspective of a write operation, $\mathcal{M}$ is organized as a $R_{max} \times L$ matrix. The probability of the write operation itself is defined to write columns about $\frac{L}{2}$ more frequently than columns about 0 or $L-1$ (Algorithm 1), hence defining short- and-long term memory regions within $\mathcal{M}$ respectively.

Read operations assume a memory model in which $\mathcal{M}$ is perceived as a single array of consecutively indexed memory locations [17]. In effect, we anticipate that over time read references to external memory will learn to distinguish between short- and long-term memory locations, whereas write references will learn what (register) states are actually useful to record. Figure 1 summarizes the difference in the organization of memory from the perspective of read and write operations.

[3]We do not preclude the same program performing multiple write (or read) operations.

**Algorithm 1** Write function for External memory $\mathcal{M}$. Function called by a write instruction of the form: Write($\mathcal{R}$) where $\mathcal{R}$ is the vector of register content of the program when the write instruction is called. Step 1 identifies the mid point of memory $\mathcal{M}$, effectively dividing memory into upper and lower memory banks (Figure 1). Step 2 sets up the indexing for each bank such that the likelihood of performing a write decreases as a function of the distribution defined in Step 2a. Step 2(a)i defines the inner loop in terms of the number of registers that can source data for a write. Step 2(a)iA tests for a write to the upper memory bank and Step 2(a)iB repeats the process for the lower bank.

```
Call: Write($\mathcal{R}$)
(1) mid = $\frac{L}{2}$
(2) for (offset := 0 < mid)
    (a) $p_{write} = 0.25 - (\beta \times \text{offset})^2$
        (i) for ($j := 0 < R_{max}$)
            (A) IF (rnd[0, 1) $\leq p_{write}$)
                THEN ($\mathcal{M}$[mid + offset][$j$] = $\mathcal{R}[j]$)
            (B) IF (rnd[0, 1) $\leq p_{write}$)
                THEN ($\mathcal{M}$[mid − offset][$j$] = $\mathcal{R}[j]$)
```

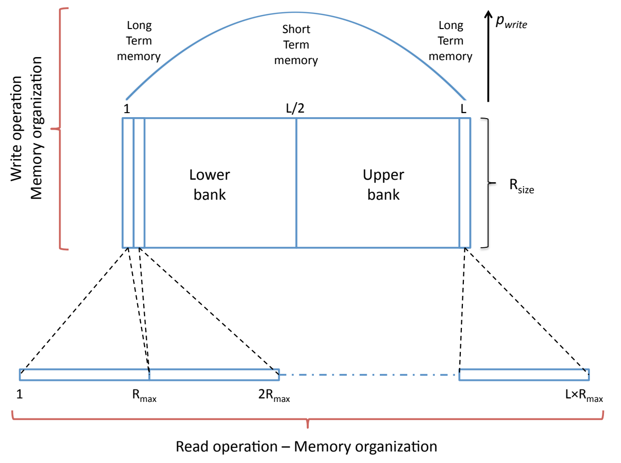

**Figure 1: View of external memory, $\mathcal{M}$, as perceived from the perspective of read and write operations. A write operation probabilistically ($p_{write}$) distributes the contents of its register set $\mathcal{R}$ over the upper and lower 'banks' of the $R_{max} \times L$ locations. Locations in the region of $L/2$ are more likely to be written to, whereas locations towards 1 and $L$ are less likely to be written to (Algorithm 1). A read operation is a reference from 'source2' with the mode bit set to 'Register-Memory'. The range of register references is cast using the mod $(\cdot)$ operator to address $L \times R_{max}$ locations in this case.**

## 6 EXPERIMENTAL DESIGN

Section 3 summarized the Dota 2 task as assumed for our goal of evolving a Shadow Fiend hero tabula rasa, which is to say, without first designing specific task decompositions to learn specific skills [22]. Naturally, more difficult formulations are possible, but we are looking at this as how a novice player might go about learning how to play the game. Hence, we adopt the 1-on-1 mid-lane set up with a single hero opponent (§ 3), with the opponent hero (bot) taking the same hero character as we evolve. In the following, we establish the action and state spaces as well as the fitness function and finish by stating the TPG parameterization adopted. Appendix A provides additional information regarding interfacing to the Dota 2 game engine.

### 6.1 Defining State space

Valve provides state functions, which can provide up to a 20,000 dimensional vector of game state at each frame, of which we only employ 320, Table 2. A total of eight different categories of feature were assumed. The *Self* and *Opponent* categories summarize the basic statistics for each team. Four of these properties represent internal bot state (Current Gold/Net worth, Last hits, Denies) and are therefore unknown in the case of the opponent. *Match time* is merely the duration of the current game. *Abilities* reflect further state information for each hero, and are repeated for all 6 of the Shadow Fiend hero abilities. As such abilities represent skills available to the hero character and change as the character 'level up' during each game (games start with heroes at the lowest level of ability). Thus, one of a hero's decisions during a game is what ability to upgrade. *Creep* state is characterized in terms of a vector of 14 attributes per creep, for the nearest 10 creeps. One way in which the game can be won/lost is when a tower is taken. Thus, there are 6 attributes defining state of Radiant and Dire's *mid-lane tower*. *Runes* represent gems that can be collected in the environment, 8 attributes summarize type, location and state of such items, and another 7 attributes summarize *Items* bought during the course of a game by each hero. An attribute value of '-1' is assumed for any attributes that are not measurable due to partial observability or if there is not actually enough of the property to measure (e.g. there are less than 10 creeps observable/alive).

### 6.2 Defining the Shadow Fiend action space

We will assume a minimalist scenario in which we want to provide enough functionality for our Shadow Fiend hero to support the following five properties, resulting in a **total of 30 actions**.

**Movement:** at any point in time heroes have the ability to walk in one of eight cardinal directions (north, north-east, east, south-east, south, south-west, west, north-west) or not move, resulting in *9 actions*. When issued, the move command is applied to a ground location 100 units from the heroes current location. **Attacking opponents:** an 'attack action' declares which opponent to attack, and assumes a default integer value (future work could potentially tune these defaults). In total TPG will distinguish between 12 possible targets: the opponent tower, the opponent hero, or the ten nearest opponent creeps, so a total of *12 actions*. **Casting spells:** a Shadow Fiend character has a unique set of spell casting abilities.[4]

[4]https://dota2.gamepedia.com/Shadow_Fiend

**Table 2: Attributes used to characterize game state. In the case of attributes estimated for both Self and Opponent, the † symbol indicates a Self specific attribute.**

| Attribute type | Attribute count |
|---|---|
| Self (Opponent) | 30 (26) |
| Team, Level, Health (avg, max, regen), Mana (avg, max, regen), Movement Speed (Base, Current), Base Damage (avg., variance), Attack (Damage, Range, Speed), Sec. per attack, Attack (Hit location, Time since last, Target of current), Vision range, Strength, Agility, Intelligence, Gold†, Net Worth†, Last hits†, Denies†, Location (x-, y-coordinate), Orientation | |
| Match duration | 1 |
| Abilities – 6 per Self (Opponent) | 48 (42) |
| Level, Mana cost, Damage, Range, Num. of Souls (Necromaster), Cooldown†, Target type, Effect flag area | |
| Creeps (up to 10 nearest) | 140 |
| Team, Health, Max Health, Movement speed (Base, Current), Base damage (avg., variance), Attack (Damage, Range, Speed), Sec. per attack, Location (x-, y-coordinate) | |
| Towers – Self (Opponent) | 6 (6) |
| Team, Health (avg., Max), Attack (Damage, Range, Speed) | |
| Runes | 8 |
| Bounty rune location (top, bottom), Top Regular rune location, Top Powerup rune (Type, Status), Bottom Powerup rune (Location, Type, Status) | |
| Items – Self (Opponent) | 7 (6) |
| Inventory (Slot 1, ..., Slot 6), Healing Salve flag† | |

The 'Shadowraze' is a ranged spell cast with three distances (near, medium, far), whereas a 'Requiem of Souls' is either cast or not. This results in another *4 actions*. **Collecting items:** a hero can collect 'powerup' and 'bounty' runes while navigating the environment, but to do so it has to explicitly deploy the relevant action. This results in another *4 actions*. **Hero's bot:** the Shadow Fiend hero can control a 'Healing Salve' for which there is a *single action*, deploy or not.

### 6.3 Fitness function

There are many tactics in a good hero strategy which relate to the relationship between the hero, friendly creeps, and defensive towers. In the following we reward our hero explicitly for successful 'Last Hitting' and 'Denies', but ignore other tactics such as 'Creep Blocking' and 'Pulling'; both of which redirect friendly creeps in an attempt to move the opponent creeps into positions that are easier for you to attack.[5] In addition, we include three factors explicitly scored by the game engine: Net worth, Kills, and Match points.

**Last hits (LH):** A 'last hit' refers to the killing blow applied to an opponent or neutral creep. If your hero is the last hitter, then a gold bonus is payed to your hero, increasing your ability to purchase items for improving the capabilities of your hero. Each last hit is awarded ***10 points*** in the fitness function. **Denies (D):** When creep health decreases below 50% you can get a last hit on your own creeps. This is useful because it 'denies' a last hit to the opponent hero, i.e. you prevent the opponent hero from gaining

[5]https://dota2.gamepedia.com/Creep_control_techniques

**Table 3: TPG Parameterization. For the most part this follows an earlier work deploying TPG in ViZDoom without External Memory and restricted forms of environment that did not require significant amounts of navigation [16].**

| Team Population | Program Population |
|---|---|
| Team Population Size ($P$): 360 | Max. Instructions: 64 |
| *Gap*: 50% of Root Teams | Prob. Delete Instr. ($P_{del}$): 0.5 |
| Initial Prog. per Team ($\omega$): 40 | Prob. Add Instr. ($P_{add}$): 0.5 |
| $P_d$: 0.7 | Prob. Mutate Instr. ($P_{mut}$): 1.0 |
| $P_a$: 0.7 | Prob. Swap Instr. ($P_{swp}$): 1.0 |
| $P_m$: 0.2 | $P_{nm}, P_{atomic}$: 0.2, 0.5 |

gold or experience points. Each 'deny' is awarded ***15 points*** in the fitness function, i.e. significantly less common than a last hit. **Net worth (NW):** is the overall wealth of your team, so includes all bought items, gold, and any inventory accumulated. We use the net worth as calculated by the game engine. **Kills (K):** Each time an opponent hero is killed a reward of ***150 points*** is given. **Match (M):** A match ends with either a hero dying twice, or the successful destruction of the team's tower (Figure 3). If the agent is the winner, the game engine awards ***2000 points***. **Not dead (ND):** A bonus of ***150 points*** is awarded if the hero did not die during the match. TPG Shadow Fiend hero fitness, $f_i$ now has the form:

$$f_i = 10 \times LH + 15 \times D + NW + 150 \times K + 150 \times ND + 2000 \times M \quad (1)$$

where *LH, D, K* reflect counts for the number of times each property is achieved per match, $ND \in [0, 1, 2]$ reflecting the number of times the agent is 'not dead' per match, $NW$ assumes the value estimated by the game engine at the end of the match, and $M \in [0, 1]$.

## 6.4 TPG parameterization

External memory is parameterized in terms of the number of registers a program indexes ($R_{max}$, § 5) and the number of columns, $L$ (Figure 1). Thus, from the perspective of a read operation, Figure 1, there are $M = R_{max} \times L$ indexible locations. In this work all our experiments assume $L = 100$ therefore the range of indexes supported during a read operation is $M = 800$. The probability distribution defined in Step 2a of Algorithm 1 is parameterized with $\beta = 0.01$. This means that for 'offset' values $\approx 0$ the likelihood of a write operation is 25% (or 4 out of $R_{max} = 8$ registers will be written to the shortest term memory location as indexed about $\approx \frac{L}{2}$). However, as 'offset' $\longrightarrow L$ (and respectively 1), then the likelihood of performing a write tends to 1%. Overall a write operation will result in 17% of $\mathcal{M}$ changing value under this parameterization. No claims are made regarding the relative optimality of such a parameter choice. Table 3 summarizes the parameters assumed for TPG (§ 4) and Table 1 details the instruction set.

## 7 RESULTS

Two experiments are performed: **reactive TPG** has scalar stateful memory, but no external indexed memory, **memory TPG** supports both scalar stateful and external indexed memory. In both cases TPG Shadow Fiend agents are trained against the Shadow Fiend bot at 'Hard' difficulty for 250 generations (about 2 weeks, single threaded, Intel CPU clocked at 4GHz). Parameters specific to the external indexed memory configuration are summarized in Section 6.4. All other TPG parameters are common to both experiments and detailed in Table 3. Post training a single champion Shadow Fiend agent is identified as the TPG agent with highest fitness.

**Table 4: Evaluation of champion TPG agent against opponent Shadow Fiend Bot. A total of 50 games are played at each difficulty level. XP is the median 'experience points' as awarded to memory TPG play over the 50 games. Median 'gold' reflects the ability of memory TPG to monetize during game play.**

| Opponent Bot | reactive TPG | memory TPG | | |
|---|---|---|---|---|
| Difficulty | # wins | # wins | XP | Gold |
| Easy | 3 (6%) | 31 (62%) | 8,760 | 3,633 |
| Medium | 0 (0%) | 22 (44%) | 10,111 | 3,860 |
| Hard | 0 (0%) | 12 (24%) | 6,166 | 3,170 |

### 7.1 Overall performance assessment

Champion TPG agent performance is assessed post training using a tournament between TPG Shadow Fiend and the Shadow Fiend bot from the Dota 2 game engine deployed under each of the three difficulty levels: easy, medium, hard. A total of 50 games are played for each difficulty level. Each game is played until there is a definitive winner. Table 4 summarizes the outcome of this tournament. Clearly the reactive TPG agent was unable to learn any useful strategies for playing against the Opponent Bot. The memory TPG agent, however, was able to develop effective strategies for competing with the Opponent Bot when set to 'easy' and 'medium' difficulty settings. The Opponent Bot was still clearly much better when set to the 'hard' setting. Also quoted are the median experience points (XP)[6] and Gold monetized by memory TPG during games and larger values are better. When playing the Opponent Bot with an 'easy' or 'hard' difficulty, it is apparent that the games are won/lost relatively quickly, not giving as much opportunity to increase XP and Gold. Conversely, under the 'medium' setting, games are much more evenly matched, with a hero character able to demonstrate a wider scope of behaviours.

A tournament using only the Opponent Shadow Fiend Bot was also attempted, i.e. 'Hard' against 'Medium' and 'Hard' against 'Easy'. The 'Hard' Shadow Fiend Bot *always won*. There are several factors that result in this. Easy, Medium and Hard difficulty levels are differentiated on the reaction times for ability and item usage (0.3 sec for Easy, 0.15 sec for Medium and 0.075 sec for Hard). Easy and Medium also have a delay for last-hitting whereas Hard has none.[7] Easy and Medium also have abilities subject to a 'recovery' delay once used (3 and 6 seconds respectively), whereas Hard has none. This impacts on the Shadow Fiend character's ability to cast its signature ranged attack. Thus, it is noteworthy that the evolved memory TPG agent is able to win some matches with the Opponent Bot set to a hard difficulty.

[6]XP reflect up to 11 different hero performance factors. https://dota2.gamepedia.com/Experience
[7]https://dev.dota2.com/showthread.php?t=19027

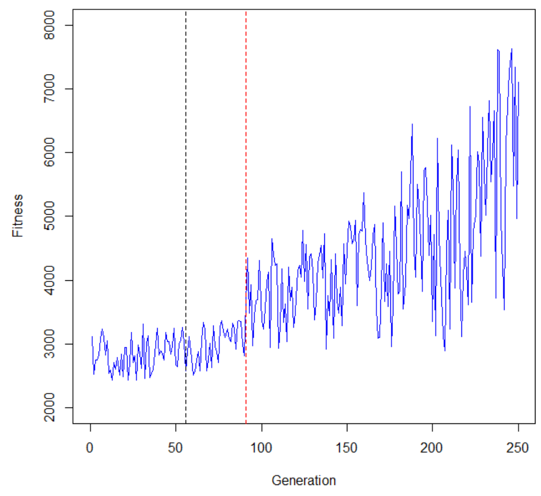

**Figure 2: Example fitness curve with the point at which memory TPG identified mid-lane (generation 56) and the point at which memory TPG discovered how to last hit creeps (generation 91).**

A typical fitness curve for TPG with memory is summarized in Figure 2. The curve is certainly noisy, due to the stochastic properties of the game, non-Markovian state space, and the multi-modal nature of the performance function. However, we can identify three specific regions of agent development. The region prior to the first horizontal dashed line represents performance before the TPG agent reliably navigated the Shadow Fiend hero to the mid-lane region. That is to say, once in the mid-lane region, the TPG Shadow Fiend would set up a patrolling like behaviour just prior to the river feature. This is important, as only at this point can the hero develop strategies for defending/attacking using its creeps.

The region following the second horizontal dashed line represents the point where TPG with memory manages to reliably demonstrate 'last hitting' on creeps (§ 6.3). This is particularly important as it accelerates the ability to generate 'gold', hence the hero can trade for items and improve its ability to defend/attack the Opponent team. This is immediately reflected in the fitness curve which is on an upward trend thereafter.

## 7.2 Navigation behaviours

We summarize the navigation behaviour observed in our champion TPG Shadow Fiend agents by plotting the movement of the agents over the course of a game and plotting them on a map of the environment.[8] Figure 3 illustrates the paths typically assumed by the reactive TPG agents. In effect, the agent would fail to note the significance of the mid-lane to the game (creeps will advance along the leading diagonal between the two team's bases). This left the creeps for the reactive TPG to fend for themselves while the hero wandered about the forest areas above or below the mid-lane. Ultimately, the reactive TPG hero would die once it encountered

[8]This was more effective than attempting to plot heat maps over all tournaments as the heat maps would tend to cover too much due to the aimless nature of the reactive TPG agents.

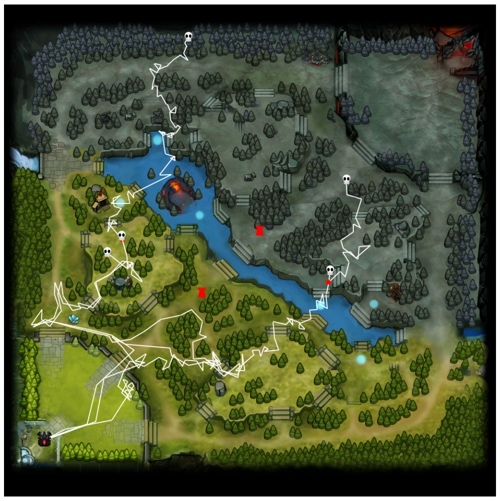

**Figure 3: Typical paths navigated (in white) by *reactive* TPG Shadow Fiend agent. 'Skull' icon mark where hero lost a life.**

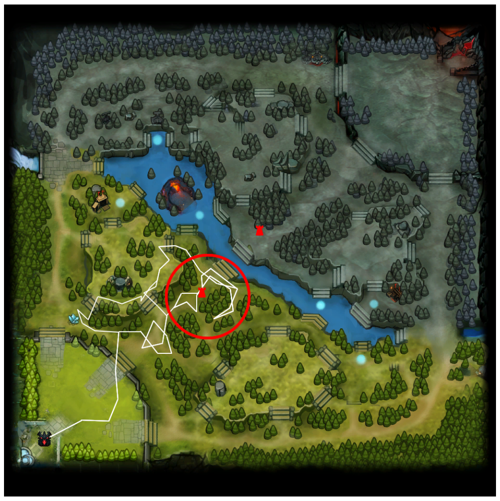

**Figure 4: Typical paths navigated by *memory* TPG Shadow Fiend agent. Red circle represents region patrolled by hero.**

neutral or opponent creeps or the opponent hero. In one case, it entered the forest and could not leave.

Conversely, the TPG with memory is extremely consistent with the navigation profile adopted (Figure 4). The red circle emphasizes the region in which the TPG with memory agent ultimately patrols. This corresponds to where the agent's tower feature is. This is particularly important because the tower and creeps need to be deployed in combination in order to set up a strong defensive position from which attacks can be launched across the river. Note also, that the hero does not descend into the river, as this would immediately result in an inability to see anything on land.

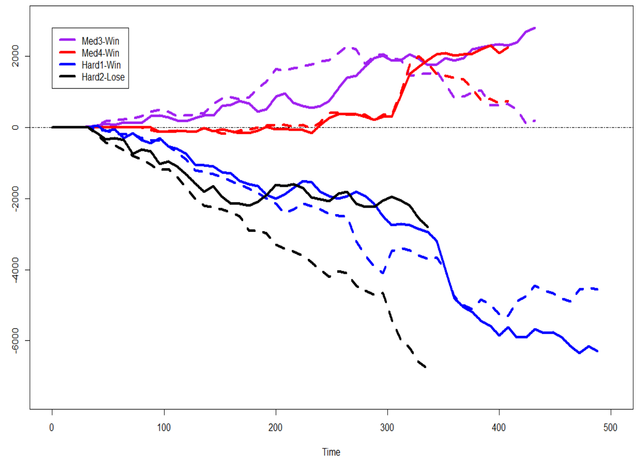

**Figure 5: Interaction of 'Experience' (solid line) and 'Net worth' (dashed line) under test conditions for memory TPG champion during game play. The *x*-axis implies that Self = Opponent for the metric, Self > (<) Opponent are positive (negative) *y*-values. 'Med' denote games against a medium difficulty opponent, 'Hard' a game against the hardest opponent. (Figure best viewed in colour.)**

## 7.3 Game dynamics during a tournament

There are many aspects of the Dota 2 game that make it a challenging. We can illustrate some of this by plotting the interaction between the two principle performance metrics reported during the course of a game: net worth and experience.

Figure 5 summarizes the development of Experience and Net worth over the course of four tournaments. Note that these are all strategies demonstrated by the *same* TPG Shadow Fiend agent. In the *purple* curves the agent emphasizes collecting last hits, building up the Net worth property. At around a time step of 300 the TPG hero switches strategy to one of killing as many opponent creeps as possible. This resulted in the destruction of the opponents tower, hence winning the game. The *red* curves illustrate a close game until the TPG hero managed to get a small Experience/Net worth advantage around a time of 250, ultimately resulting in the opponent hero being killed.[9] This produced a step increase in Experience/Net worth for the TPG hero, that was capitalized on, resulting in the opponent tower then being taken, so winning the match.

The *blue* curves represent a game against a hard Opponent Bot in which the TPG agent ultimately won, despite both Experience/Net worth curves being strongly against the TPG agent. Indeed, the sudden drop around a time of 350 reflects the TPG hero being killed. However, after re-spawning the TPG agent continued to put pressure on the opponent tower, ultimately destroying the tower, and winning the game. The *black* curves demonstrate a loss against the hard Opponent. The Opponent gained much more Net worth and destroyed the TPG agent's tower.

[9]Heroes can re-spawn once during the same match.

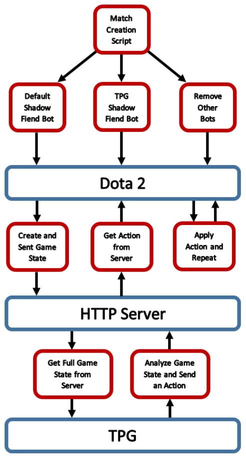

**Figure 6: Dota 2 interface assumed for TPG interface. Source code available at https://web.cs.dal.ca/~dota2/**

## 8 CONCLUSION

The recent availability of a bot interface for Dota 2 [5] presents the opportunity to experiment with reinforcement learning in a task environment presenting a unique set of challenges. Even the 1-on-1 mid-lane presents a task in which multiple sources of uncertainty exist: partial observability, stochastic, multimodal objectives. We demonstrate that agents can be evolved directly from the 'hard' level of built-in Shadow Fiend Bot when the TPG is augmented with external indexed memory. This implies that: 1) register state is capable of finding encodings that summarize high dimensional state, and 2) programs composing TPG solutions emerged that successfully make use of memory content.

The resulting TPG Shadow Fiend agent is able to focus on destroying the Opponent towers as a long term strategy, but only because it is capable of first navigating to the relevant region of the map, and can switch between emphasizing last hits versus a focused attack on towers. No attempt was made to sequentially reward success in each objective, and the instruction set does not include any task specific operations.

There are many potential avenues for future work, from experimenting with transfer of agent policies between different hero characters, to multi-lane battles, increased enemy bot difficulty, or matches against multiple opponents.

## A DOTA 2 BOT INTERFACE

Developing the Dota 2 agent interface was a significant undertaking in itself. Figure 6 provides a summary of the interfacing assumed to facilitate the evolution of agents in Dota 2. Scripts are used to configure the Dota 2 game engine (single lane, opponent hero type, etc.) and collect game statistics, whereas TPG interfaces to the game engine during play through a HTTP interface. For further details and source code, see https://web.cs.dal.ca/~dota2/

## ACKNOWLEDGMENTS

This research was supported by NSERC grant CRDJ 499792.

## REFERENCES

[1] B. Andersson, P. Nordin, and M. Nordahl. 1999. Reactive and memory-based genetic programming for robot control. In *European Conference on Genetic Programming (LNCS)*, Vol. 1598. 161–172.

[2] M. Brameier and W. Banzhaf. 2007. *Linear Genetic Programming.* Springer.

[3] S. Brave. 1996. The evolution of memory and mental models using genetic programming. In *Proceedings of the Annual Conference on Genetic Programming.*

[4] J. Cartlidge and S. Bullock. 2004. Combating coevolutionary disengagement by reducing parasite virulence. *Evolutionary Computation* 12, 2 (2004), 193–222.

[5] J. M. Font and T. Mahlmann. 2019. The Dota 2 Bot Competiton. *IEEE Transactions on Games* (2019). to appear.

[6] P. García-Sánchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo. 2015. Towards automatic StarCraft strategy generation using genetic programming. In *IEEE Conference on Computational Intelligence and Games*. 284–231.

[7] F. Haddadi, H. Günes Kayacik, A. N. Zincir-Heywood, and M. I. Heywood. 2013. Malicious Automatically Generated Domain Name Detection Using Stateful-SBB. In *EvoApplications (LNCS)*, Vol. 7835. 529–539.

[8] Max Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. García Castañeda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, N. Sonnerat, T. Green, L. Deason, J. Z. Leibo, D. Silver, D. Hassabis, K. Kavukcuoglu, and T. Graepel. 2018. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *CoRR* abs/1807.01281 (2018).

[9] S. Kelly and M. I. Heywood. 2017. Emergent Tangled Graph Representations for Atari Game Playing Agents. In *European Conference on Genetic Programming (LNCS)*, Vol. 10196. 64–79.

[10] S. Kelly and M. I. Heywood. 2018. Emergent Solutions to High-Dimensional Multitask Reinforcement Learning. *Evolutionary Computation* 26, 3 (2018), 347–380.

[11] S. Kelly, Robert J. Smith, and M. I. Heywood. 2019. Emergent policy discovery for visual reinforcement learning through tangled program graphs: A tutorial. In *Genetic Programming Theory and Practice*, Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman (Eds.). Vol. XVI. Chapter 3, 37–57.

[12] W. B. Langdon. 1998. *Genetic Programming and Data Structures.* Kluwer Academic.

[13] P. Lichodzijewski and M. I. Heywood. 2010. Symbiosis, complexification and simplicity under GP. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference.* 853–860.

[14] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games* 5, 4 (2013), 293–311.

[15] A. Sapienza, H. Peng, and E. Ferrara. 2017. Performance dynamics and success in online games. In *IEEE International Conference on Data Mining Workshops*. 902–909.

[16] R. J. Smith and M. I. Heywood. 2018. Scaling Tangled Program Graphs to Visual Reinforcement Learning in ViZDoom. In *European Conference on Genetic Programming (LNCS)*, Vol. 10781. 135–150.

[17] R. J. Smith and M. I. Heywood. 2019. A Model of External Memory for Navigation in Partially Observable Visual Reinforcement Learning Tasks. In *European Conference on Genetic Programming (LNCS)*, Vol. 11451.

[18] L. Spector and S. Luke. 1996. Cultural Transmission of Information in Genetic Programming. In *Annual Conference on Genetic Programming.* 209–214.

[19] G. Synnaeve and P. Bessière. 2016. Multiscale Bayesian modeling for RTS games: An application to StarCraft AI. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 4 (2016), 338–350.

[20] A. Teller. 1994. The evolution of mental models. In *Advances in Genetic Programming*, K. E. Kinnear (Ed.). MIT Press, 199–220.

[21] M. Čertický and D. Churchill. 2017. The current state of StarCraft AI competitions and bots. In *AIIDE Workshop on Artificial Intelligence for Strategy Games.* 1–7.

[22] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. 2005. Evolving soccer keepaway players through task decomposition. *Machine Learning* 59, 1 (2005), 5–30.