Humanlike Combat Behavior via Multiobjective Neuroevolution

Jacob Schrum, Igor V. Karpov and Risto Miikkulainen

Abstract Although evolution has proven to be a powerful search method for discovering effective behavior for sequential decision-making problems, it seems unlikely that evolving for raw performance could result in behavior that is distinctly humanlike. This chapter demonstrates how humanlike behavior can be evolved by restricting a bot's actions in a way consistent with human limitations and predilections. This approach evolves good behavior, but assures that it is consistent with how humans behave. The approach is demonstrated in the `UT^2` bot for the commercial first-person shooter videogame Unreal Tournament 2004. `UT^2`'s humanlike qualities allowed it to take 2nd place in BotPrize 2010, a competition to develop humanlike bots for Unreal Tournament 2004. This chapter analyzes `UT^2`, explains how it achieved its current level of humanness, and discusses insights gained from the competition results that should lead to improved humanlike bot performance in future competitions and in videogames in general.

1 Introduction

Simulated evolution has proven to be a powerful policy-search method for solving challenging reinforcement learning problems [6, 11, 19, 22, 24, 28]. However, evolutionary methods are also notorious for taking advantage of any trick available to achieve high fitness: any loopholes present in the domain simulation software are sure to be exploited. A similar problem arises in the context of evolving humanlike behavior for videogames. Because humans are skilled at videogames, it is reasonable to evolve bots for performance in order to get humanlike behavior. However, evolution may exploit domain tricks for the sake of performance, which results in bots behaving in a non-humanlike-manner.

However, if the senses and actions available to the bot are constrained such that they both simulate the restrictions humans deal with, and make common human actions easy to carry out, then it is possible to achieve humanlike behavior by evolving for good performance, even when good performance is defined in terms of multi-

Jacob Schrum, Igor V. Karpov and Risto Miikkulainen
University of Texas at Austin, Austin, TX 78712 USA,
e-mail: {schrum2,ikarpov,risto}@cs.utexas.edu

ple conflicting objectives. This maxim is demonstrated by the UT^2 bot, which placed 2nd in BotPrize 2010, a competition to develop humanlike bots for the commercial First-Person Shooter (FPS) videogame Unreal Tournament 2004 (UT2004).

This chapter describes the UT^2 bot, with emphasis on its combat behavior, the policy for which was determined by a neural network whose weights and topology were evolved using Evolutionary Multiobjective Optimization (EMO). The UT^2 bot is further discussed in [17], also in this book, which describes how UT^2 makes use of human trace data to navigate when pathfinding fails. The two techniques are complimentary, and can be used separately or together, as was done in UT^2.

Understanding how UT^2 exhibits humanlike behavior requires an understanding of the role of bots in the FPS genre (section 2). The particulars of UT2004 and BotPrize are discussed in sections 2.1 and 2.2 respectively. Given this context, UT^2 can be discussed in detail (section 3) with emphasis on its combat behavior (section 3.2). The combat behavior was learned using neuroevolution and evolutionary multiobjective optimization, which are discussed in sections 4.1 and 4.2 respectively. How these methods were used to produce the final combat behavior for UT^2 is discussed in section 4.3. After fully describing the bot, it is evaluated in section 5. This evaluation leads to discussion and ideas for future work in section 6. Then section 7 concludes the chapter.

2 Bots in First-Person Shooters

FPS games display the game world to the player through the first-person perspective of the agent controlled in the game. Early games pitted players against simplistic computer controlled opponents. Since the available weapons, ammo, health, and general capabilities of players differed so much from that of the computer opponents, it mattered little if the enemies behaved in a humanlike manner.

However, FPS games eventually began incorporating multiplayer modes that allowed players to compete against other humans over a network connection. A free-for-all competition between several human competitors is called a Deathmatch. In this style of play, all players are on equal footing with regards to weapons, health and abilities. From the advent of human multiplayer combat, it was only a small step to FPS games entirely based around the concept of multiplayer-style play.

2.1 Unreal Tournament 2004

The original Unreal Tournament (1999) was the first FPS to fully embrace the multiplayer style of gameplay. Although the game had a single-player mode, this mode consisted exclusively of a series of matches against bots played with the exact same rules used in multiplayer mode against humans. Thus arose the need for convincingly human bots in FPS games.

UT2004 is the second sequel to the original Unreal Tournament, and continues the trend of focusing on multiplayer-style play against humans. In addition to Deathmatch mode, all Unreal Tournament games feature several additional types of team play, such as team Deathmatch and capture the flag, but since these modes of play are not yet part of BotPrize, they will not be discussed further in this chapter.

In a Deathmatch, players spawn at random spawn points with only the most basic weapons. They then run around the level, accruing more powerful weapons and other useful items in order to help them kill each other in combat. An event where

one player kills another is called a frag, and is worth one point. After dying, players immediately respawn at a new, randomly chosen spawn point with full health, but only rudimentary weapons, as at the start of the match. If a player kills himself or herself, for example by jumping in a pit or by firing a rocket at a nearby wall, the penalty is the loss of one point, which can result in a negative score. The goal of a Deathmatch is to either get the most points within a preset time limit, or be the first to attain a preset number of points.

Because this chapter deals primarily with bot combat behavior, the specific weapons available in UT2004 will be reviewed in detail. Each weapon has both primary and alternate firing modes which are often very different from each other. Sometimes the alternate firing mode does not fire at all, but instead activates some special ability of the weapon. Several weapons also have a charging attack, which requires holding down the fire button to charge up a projectile whose properties depend on how long the weapon is charged before being released. Each weapon is explained in detail so that later descriptions (section 3.2.3) of how the bot handles each weapon will be understood:

- Shield Gun: A last resort weapon whose ammo recharges automatically. Players spawn with this weapon.
  - Primary: Charges weapon until the player is close enough to touch an opponent, at which point the weapon automatically discharges to deal an amount of damage proportional (within bounds) to how long the weapon was charged.
  - Alternate: Creates a defensive shield in front of the player that deflects projectiles while the fire button is held.
- Assault Rifle: A weak but rapid firing gun that all players spawn with.
  - Primary: Automatic fire that is rapid but weak.
  - Alternate: Charges a grenade that is launched in an arc on release. The grenade bounces off of level geometry but explodes on impact with players. Powerful, but difficult to aim.
- Shock Rifle: Weapon with both a fast, focused attack and a slower attack that explodes to affect a large area on impact.
  - Primary: Immediately hits target in the crosshairs and knocks players back on impact, which can disorient them. However, the delay between subsequent shots is significant.
  - Alternate: Fires a large, slow moving orb that explodes on impact. There is also a special combo attack that creates a larger, more powerful explosion if the primary fire mode is used to shoot the orb out of the air. Bots can only perform this "shock combo" by chance because they cannot determine the locations of their own projectiles.
- Bio-Rifle: Weapon whose projectiles fire in an arc and linger on the ground, where they explode on impact with any player that comes into contact with them. Note that the bots in BotPrize have no way of seeing these potential traps.

  - Primary: Rapidly fires small explosive green blobs.
  - Alternate: Charges the weapon in preparation for firing a large blob that deals an amount of damage proportional (within bounds) to the duration of the charge. If the shot misses, then the large blob explodes into a batch of small blobs upon hitting the ground.

- Minigun: A rapid fire machine gun.
  - Primary: High rate of fire, but slightly inaccurate, and therefore best suited to close quarters combat.
  - Alternate: Slower rate of fire, but is more accurate and fires shots that deal more damage.

- Flak Cannon: Versatile weapon whose primary firing mode is effective at close range and whose alternate firing mode works well at medium range.
  - Primary: Several small shards of flak are launched in a wide spread, each doing little damage, but dealing a great deal of damage together.
  - Alternate: Launches a flak grenade in an arc. Damaging flak is spread in all directions on impact.

- Rocket Launcher: Fires slow but powerful explosive projectiles.
  - Primary: Immediately fires a single rocket.
  - Alternate: Charges up to three rockets to be fired simultaneously. When released, however many rockets are currently loaded will be fired. The default firing pattern is a wide spread that becomes wider as the rockets get farther away. However, pressing the primary fire button while still charging causes the rockets to shoot in a tighter, forward moving spiral.

- Sniper Rifle: Very accurate and powerful, but slow firing weapon.
  - Primary: Fires a single shot that instantly hits whatever is in the crosshairs.
  - Alternate: For humans, alternate fire activates the sniper scope. Holding down the alternate fire button zooms in to allow the player a better view of what is in the distance at the cost of not being able to see nearby surroundings. While zoomed in, the player can use primary fire to shoot. However, bots are unable to use this feature because they do not see the world the way humans do.

- Lightning Gun: Functionally the same as the Sniper Rifle, except that the bolt of lightning fired by this gun can be seen by humans, making it easier to trace an attack back to its source. Bots cannot see these lightning bolts.
  - Primary: Fires a single bolt of lightning that instantly hits its target.
  - Alternate: Switches to a sniper scope, as with the Sniper Rifle.

This list shows that UT2004 provides viable weapons for any combat situation. Certain weapons are only useful within certain ranges, though when under attack players may be forced to improvise with the weapons and ammo currently available to them. Given a choice of what weapon to use in combat, there are several

salient features that can be used to choose an appropriate weapon. Perhaps more importantly, these features dictate how the weapon is used once it has been chosen.

- Rate of Fire: Rapid firing weapons work best in hectic, mid-range combat scenarios when players are actively dodging, whereas slow-firing weapons tend to be better at longer range, in which case the shooter can take time to make each shot. The latter statement is especially true of the sniping weapons.
- Projectile Speed: Some weapon shots instantly hit any target in the crosshairs, while others take time to reach their destinations. Humans using weapons with slower projectiles tend to compensate for the slowness by anticipating where their opponents will be in the next few seconds.
- Firing Trajectory: The alternate firing modes of both the Flak Cannon and the Assault Rifle launch projectiles in curved arcs that tend towards the ground. Both firing modes of the Bio-Rifle also fire in an arc. When using these weapons, players must account for gravity, which usually means aiming higher than one would aim with a straight firing weapon.
- Splash Damage: Weapons with an explosive component deal "splash" damage. Splash damage is particularly useful against players that dodge well, and are therefore hard to hit, since near misses will also damage them. However, splash damage weapons are also dangerous since they can damage the shooter as well. For this reason, splash damage weapons are not preferred in close quarters combat. When fired, it makes sense to aim at an opponent's feet, since the explosion from hitting the ground may damage the opponent even when the shot misses.

These weapon features are all relevant in defining the combat behavior of `UTˆ2`. However, `UTˆ2` was designed not only to perform well in UT2004, but in the modified version used in the 2010 BotPrize competition, which is described next.

2.2 BotPrize 2010

The original 2008 BotPrize competition [13] was billed as a "Turing Test for Bots" in which, as in a traditional Turing Test [27], each judge attempted to distinguish between a computer controlled bot and a human confederate in a three-player match. Many changes to this scheme were introduced in the 2010 competition [14]. The most important is the inclusion of a judging gun, which replaces a weapon not mentioned in section 2.1: the Link Gun. All human players and bots spawn with the judging gun, which has infinite ammo. Both the primary and alternate fire modes of the gun look and sound the same to all observers, but these two modes are different in that one is meant to be fired at bots and the other is meant to be fired at humans. If a bot is shot using the primary firing mode, then the bot instantly dies and the shooter gains 10 points. Similarly, if a human-controlled agent is shot using the alternate firing mode, then the human-controlled agent instantly dies and the shooter gains 10 points. In contrast, if either firing mode is used against an agent that is the opposite of the intended type, then the shooter instantly dies, and loses 10 points. In any case, a player is allowed to judge any other player only once; subsequent attempts to judge the same player will have no effect.

The judging gun not only changes how judging is done, but completely changes the game from a pure Deathmatch to a judging game. Since the bots are being tested

in this new judging game, they also have access to the judging gun, which adds the challenge of deciding if and when a bot should use the judging gun. Unfortunately, humans can now benefit from pretending to be bots. Such "distortion effects" are discussed in [30].

Because all players have the judging gun, there is no longer a division between human judges and human confederates. Furthermore, matches are no longer limited to three players. Several bots and a roughly equal number of humans play simultaneously. All human players are judges, but they are ultimately competing for the highest score. Of course, judging correctly is a good way to get a high score, since correct judgments are worth 10 points each.

Other than the judging gun, all weapons function as usual, except that all damage dealt is only 40% of normal, in order to give humans ample chance to observe opponents before one of them dies. The levels used were three publicly available maps designed by members of the UT2004 community: `DM-DG-Colosseum` (Colosseum), `DM-IceHenge` (IceHenge), and `DM-GoatswoodPlay` (Goatswood). Each match lasted 15 minutes between the five competing bots, one to two native UT2004 bots, and six to seven humans. There were a total of 12 matches conducted during three separate one-hour sessions.

All of this information, along with the maps and the game modification which implemented the competition rules, were available to the entrants before the competition. `UT^2` was designed to compete within the parameters of this competition.

3 The `UT^2` Bot

The `UT^2` bot was developed at the **U**niversity of **T**exas at Austin for use in the game **U**nreal **T**ournament 2004, hence the exponent of 2 after `UT` in the name. Specifically, the bot was designed for BotPrize 2010 using Pogamut 3 [10], a platform for writing Java code to control UT2004 bots via a customized version of the Gamebots message protocol [1]. This section outlines the overall architecture of the `UT^2` bot, and then focuses on the bot's battle controller.

3.1 Architecture

The architecture controlling `UT^2` is a behavior-based approach similar to both the POSH interface [5], which is integrated into Pogamut 3, and behavior trees [16], which were introduced in the commercial videogame Halo 2. The bot has a list of behavior modules, each with its own triggers. On every time step the bot iterates through the list, checking triggers for each module until one of them evaluates to `true`. The module associated with the chosen trigger takes control of the bot for the current time step. Each module can potentially have its own set of internal triggers that further subdivide the range of available behavioral modes.

The specific bot architecture is shown in Fig. 1. The highest priority action is getting *UNSTUCK*. Several triggers detect if the bot is stuck, but if any of them fire, it means the bot's ability to navigate has failed, and emergency action is needed to return to a state where the bot can function as normal. `UT^2`'s method for getting unstuck is based on human trace data, and is explained in full detail in another chapter in this book [17].

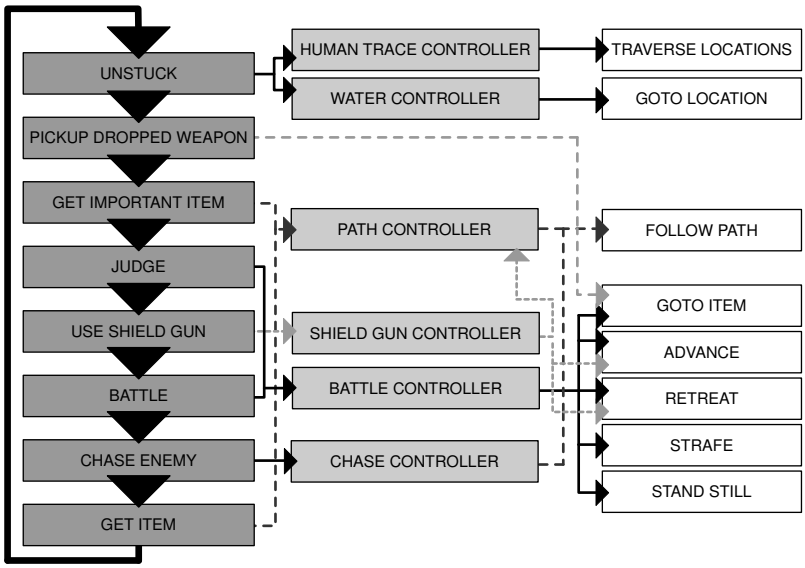


Fig. 1 Agent architecture for the `UT^2` bot. The left column shows the behavior modules in priority order from top to bottom. The middle column shows the individual controllers used by each module. Notice that the battle controller is used by both the *JUDGE* and *BATTLE* modules. The right column shows the individual actions available to each controller. This architecture can also be thought of as a POSH plan or a two-level behavior tree. This behavior-based architecture modularized bot behaviors, making the overall behavior easier to understand, and making programming and troubleshooting easier.

The next highest priority action is picking up weapons that have been dropped by killed opponents (*PICKUP DROPPED WEAPON*). Whenever an opponent dies, the weapon the opponent was using, along with whatever ammo it had, becomes available for pickup for a short time before disappearing. Humans tend to pick up these weapons immediately when they are dropped provided they are close enough, so it was decided that a humanlike bot should do the same.

The next highest priority module is *GET IMPORTANT ITEM*. Some items are highly desirable, either in absolute terms or in certain contexts, and should be pursued even if it means running away from combat. One such item is the Keg o' Health, which gives a player 100 health points, exceeding the normal limit of 100. The Double Damage powerup is always desirable as well. It makes a player's weapons deal twice the normal amount of damage for a period of 30 seconds. Items that are circumstantially important are health items when the bot is low on health, and weapons/ammo when the bot can only use the basic starting weapons. The *GET IMPORTANT ITEM* module makes the bot focus on and pursue any important item that is visible and close enough to obtain in a relatively short amount of time.

The next module is the *JUDGE* module, which uses the battle controller, the primary focus of this chapter. The bot remembers all opponents that it has judged so that it will not attempt to judge anyone twice. The bot's decision to judge is based

on how much interaction it has had with a given opponent, and how much time is remaining in the match. Judging is more likely if the bot has interacted a lot with an opponent, and if there is little time remaining in the match. Once the decision to judge has been made, the actual decision is based on knowledge of previous judgments and an assumption (only approximately true) that the number of bots in a match equals the number of humans. Whenever the bot judges an opponent, it knows the identity of the opponent after the judgment, regardless of the outcome. This knowledge is used to determine the probability that any remaining player is a human or a bot, which is in turn used to make a random but informed decision about how to judge an opponent. As for how the bot behaves while making its judgment, this is determined by the battle controller, which is described below in section 3.2.

The *JUDGE* module is high in the priority list because every player has infinite ammo for the judging gun, which makes its use a viable option at all times. However, if the bot chooses not to judge, but has no other ranged weapons available, it will resort to the *USE SHIELD GUN* module. Proper Shield Gun usage separates the good players from the experts, but typical players avoid using it because the ranged weapons are so much easier to use in comparison. Most players only resort to the Shield Gun when there is no other option. Because the Shield Gun is so different from the other weapons in the game, it has its own scripted controller.

Without ranged weapons, a player is more vulnerable. Human players will typically seek out better weapons rather than risk fighting with just the Shield Gun. Therefore, the controller's design is based on the idea that a human's primary concern when using the Shield Gun is getting a better weapon. The bot is programmed to approach the nearest relevant item while facing the nearest opponent and using the shield mode of the gun to defend itself. Relevant items consist of ammo for weapons that the bot has, and weapons that the bot does not have (picking up a previously possessed weapon provides no extra ammo for that weapon). However, the bot only pursues such a relevant item if it is closer than the nearest enemy. If the enemy is very close, then the bot will rush in with the attack mode of the Shield Gun. If the enemy is closer than a relevant item, but not close enough to do a Shield Gun rush, then the bot will simply try to put as much distance between itself and the opponent as possible while using the shield mode to defend itself.

The Shield Gun is only used if ranged weapons are unavailable. For ranged weapons, the *BATTLE* module takes over. The bot avoids combat if its health is very low, or if it is very far away from visible opponents, but otherwise it equips whatever available weapon is best for the given circumstances and uses the battle controller to drive its behavior, as described in the next section (section 3.2).

A static lookup table indexed by distance from the opponent determines the best weapon in each situation. Distance from the opponent is partitioned into close (less than 100 UT units), medium (100 to 2000 UT units) and long range (greater than 2000 UT units). In general, sniping weapons and splash damage weapons are favored at long range. At medium range, the Flak Cannon is favored, followed by rapid fire weapons and splash damage weapons. At close range, splash damage weapons and sniping weapons have lowest priority, and rapid fire weapons are favored. This table was tuned based on experience in UT2004, as well as trial and error.

If the bot is otherwise ready for battle, but sees no enemy, it checks its memory of where it last saw an opponent, and uses the *CHASE ENEMY* module. The bot runs to the last location that it remembers seeing an enemy in hopes of reacquiring its target and reengaging in combat, though it will break off the chase for any opponent that it sees. If the bot reaches the last known location of an enemy and still sees no opponents, it gives up the search. The bot also gives up the search after too long a period passes without encountering an enemy.

Given nothing better to do, the bot will simply head towards the nearest desirable item (*GET ITEM*), where desirability is based on current equipment and vital statistics. Basically, the bot will pursue weapons it does not have, ammo for weapons it does have, and health and armor if it has less than the full allowance.

Note that although the battle controller is primarily responsible for all combat actions, the bot is still capable of firing at opponents while using any of the non-combat-oriented modules above. In particular, the bot does not stop shooting if it needs to get unstuck, and it will fire on enemies it sees while attempting to pick up an important item. This sort of behavior is common among human players, and was deemed an essential component of a humanlike bot.

Still, the majority of `UT^2`'s combat behavior can be attributed to the battle controller. Furthermore, most interactions between the bot and other players occurs during combat, so the bot's capacity to appear human depends very much on the battle controller, which is described next.

3.2 Battle Controller

The battle controller is used by the combat and judging modules. It controls the bot using an artificial neural network. Artificial neural networks mimic some of the information processing capabilities of organic brains, but at their most basic level they can be thought of as universal function approximators between $\mathbb{R}^N$ and $\mathbb{R}^M$ for arbitrary integers N and M [12]. Some network architectures also have an internal recurrent state which influences network output [12], thus making the network behavior a function of all previous inputs rather than just the current input.

This section describes the input sensors of the battle controller's neural network, which is evolved (section 4.1), followed by a discussion of the network's outputs, including how these outputs are interpreted and filtered to produce behavior that is both effective and humanlike.

3.2.1 Network Inputs

The network used by `UT^2` processes inputs based on the bot's sensors every time step the battle controller is in use. The network produces several outputs for each set of inputs, and the outputs are used to produce an action for `UT^2`. The numerical inputs to `UT^2`'s neural network are:

- Ten Pie Slice Enemy Sensors: These sensors are identical to those used by van Hoorn et al. [15] to evolve combat behavior for a UT2004 bot. From an overhead perspective, the space around the bot is divided into slices, with the slices near the front of the bot narrower (and therefore more precise) than the slices near the rear of the bot (Fig. 2). For each sensor, the value of the input is higher if an

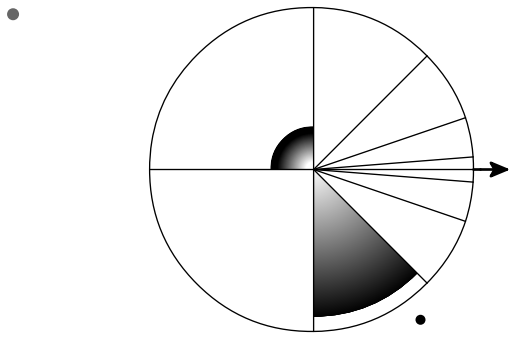


Fig. 2 Pie Slice Enemy Sensors. There are more slices of smaller size near the front (right) so the bot can better distinguish locations in front of it. The dots represent enemies, and filled portions of the pie slices show the relative activations for opponents at different distances. Activation increases as opponents get nearer, which is why the opponent in the upper left causes the corresponding pie slice to be filled less than the slice for the nearer opponent on the lower right side of the figure. Adapted from [15].

enemy sensed within that slice is closer. Given multiple enemies in one slice, the distance of the closest enemy defines the sensor value.

- Twenty-two Ray-Tracing Level Geometry Sensors: Gamebots provides a way to define periodically updated ray traces, each of which senses the distance to the first piece of level geometry that the ray trace intersects. The bot is surrounded by twelve such ray traces which are parallel to level ground. These twelve sensors are identical to the wall sensors used in [15]. However, because BotPrize levels have complicated 3D geometry, additional ray traces were added which radiated out at 45° angles both above and below the bot to sense unusual ground and ceiling geometry. There were six ceiling traces and four ground traces. Traces at each level were spread evenly around the bot (Fig. 3). However, it was discovered after the competition that the BotPrize version of Gamebots actually disabled all ray traces, meaning all these network sensors returned a value of 0. This problem is being fixed for future competitions, which will give future versions of the bot better awareness of their surroundings.
- One Crosshair Sensor: There is an additional ray trace projecting straight in front of the bot which can sense agents. If this ray trace hits an agent, then this sensor is 1.0; it is 0.0 otherwise. As with the ray traces for level geometry, this sensor only returned a value of 0 during the competition because the BotPrize version of Gamebots did not support ray traces.
- One Damage Sensor: 1.0 if the bot is currently being damaged, 0.0 otherwise.
- One Movement Sensor: 1.0 if the bot is currently moving, 0.0 otherwise.
- One Shooting Sensor: 1.0 if the bot is currently shooting, 0.0 otherwise.
- One Damage Inflicting Sensor: 1.0 if the bot is currently inflicting damage, 0.0 otherwise.
- One Ledge Sensor: 1.0 if the bot is on a ledge, 0.0 otherwise (potentially helps the bot avoid falling off of cliffs).
- One Enemy Shooting Sensor: 1.0 if the currently targeted enemy is shooting, 0.0 otherwise. `UT^2` usually targets whichever enemy is closest, but if another enemy is damaging the bot, then the threatening enemy will be targeted. Also, if the

Fig. 3 Ray-Tracing Level Geometry Sensors. Gamebots has a debugging option for viewing ray traces on a bot. The figure shows all 22 ray trace sensors around the bot, with the contrast heightened to improve visibility. Some of the rays aiming upward are brighter because they are not colliding with any level geometry. These sensors provide the bot with information about the structure of its immediate environment, which helps it reason about how best to dodge enemy attacks. Though disabled in BotPrize 2010, these sensors should help the bot be more aware of its surroundings in future competitions.

bot has already invested time damaging a particular enemy, it continues targeting that enemy unless it gets very far away, while another enemy gets much closer.

- Eight Current Weapon Sensors: For two of these sensors, 1.0/0.0 values represent yes/no answers to the following questions: Is it a sniping weapon? Does either fire mode deal splash damage? The remaining sensors report the rates of fire of both firing modes, the start-up times for firing with both modes, and the damage dealt by both modes. However, it was discovered after evolving the bot that the values Pogamut 3 returns for the damage of some weapons is incorrectly set to zero. Furthermore, alternate fire damage values for the Rocket Launcher and the Bio-Rifle are equal to the primary damage values, which does not indicate the high damage potential that these modes actually have. However, evolution seems to have been robust enough to account for these deficiencies.
- Six Nearest Item Sensors: 1.0/0.0 values represent yes/no answers regarding properties of the closest item to the bot: Is it visible? Is it health? Is it armor? Is it a shield? Is it a weapon? Is it a Double Damage powerup?
- Four Nearest Health Item Sensors: Scaled relative distances to the nearest health giving item along the x, y and z axes, as well as the scaled direct distance.

Though some of the inputs used by `UT^2` were based on sensors used in other work, some sensors were provided simply because there was a chance they would be useful. Though this particular set of inputs proved sufficient to generate good combat behavior for the 2010 competition, the task of trying to find an ideal set of inputs with which to evolve is future work.

3.2.2 Network Outputs

The outputs of the network were chosen to assure that in battle the bot would choose among actions similar to those commonly used by humans. When evolving neural networks (as described below in section 4.1) to control agents, it is common for both the inputs and the outputs to be ego-centric (cf. [15, 23]). The inputs listed above are ego-centric, but the outputs are defined both in terms of the `UT^2` bot and the opponent that it is currently targeting. This approach works because the

battle controller is only used when there is an opponent to fight, and it makes sense because human opponents pay attention to the opponents they face. Focusing on opponents is both good strategy and typical human behavior.

Specifically, the network has eight outputs: five compete to define the type of opponent-relative movement action taken by the bot, and three determine whether the bot shoots, which firing mode to use, and whether or not to jump. The five available movement actions are *ADVANCE* towards opponent, *RETREAT* from opponent, *STRAFE* left around opponent, *STRAFE* right around opponent, *GOTO ITEM* which is nearest, and *STAND STILL* (Fig. 4). The *GOTO ITEM* action is the only non-opponent-relative movement action. The movement action performed by the bot is the action whose network output has the highest activation.

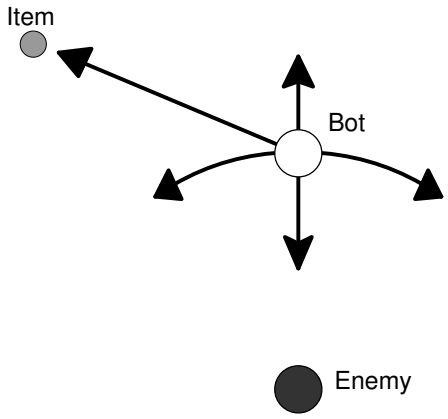


Fig. 4 Opponent-Relative Movement Actions. During combat the bot has six available movement actions depicted by the arrows in the figure (*STAND STILL* is not shown). These actions are defined with respect to the opponent the bot is currently targeting. Forcing the bot to always focus on an opponent makes it seem interested in the opponent, and therefore more humanlike.

While executing all actions, the bot looks at the targeted opponent. It is important that the bot seems interested in the human opponents it fights. The bot can also fire its weapon at the targeted opponent during any movement action. If the shooting output of the network is in the upper half of the output range, the bot shoots. The mode of fire depends on whether the fire mode output is in the lower, for primary fire, or upper, for alternate fire, half of the range. If the jumping output is in the upper half of the range, then the bot jumps while performing its movement action.

This scheme is enough to evolve effective combat behavior in UT2004, but because the objective is to evolve humanlike behavior, some additional restrictions are required to filter and adjust certain actions.

3.2.3 Action Filtering

In terms of movement, the bot will not move towards items that are not desirable (as defined with respect to the *GET ITEM* module from section 3.1). Also, when using a sniping weapon or the dangerously explosive Rocket Launcher, the bot will not *ADVANCE* towards enemies to which it is already close enough. In these cases, the action with the next highest activation is considered until a suitable action is found. The bot is also not allowed to jump if it is performing the *STAND STILL* action, since jumping in place generally looks very bot-like.

In terms of weapon usage, one of the clearest signs that an opponent is a bot is superhuman accuracy, particularly with single-shot, instant-hit weapons. Therefore, to make the accuracy of the bot more humanlike when using such weapons, the bot is actually commanded to fire at a point equal to the location of the target

plus some random noise. The maximum potential magnitude of the noise depends on both the distance between the bot and the opponent, and the relative velocities of the two agents. To account for a human's difficulty in aiming at targets that are far away, greater distances between the bot and the opponent result in greater random noise potential. To account for human difficulty in hitting moving targets, the magnitude of the noise added along the x, y and z directions is also proportional to the differences in velocity between the bot and the opponent along each of these axes. Therefore, if both the bot and the opponent are moving in the same direction at the same speed, then they are both standing still relative to each other, and no noise is added. However, such perfect synchronicity is unlikely, and in most cases the faster either agent moves, particularly when moving in different directions, the greater the noise will be and the harder it will be to aim with an instant-hit weapon.

The standard setup also needs to be more humanlike regarding how automatic weapons are used. Humans generally fire these weapons in continuous bursts as long as they can keep roughly on target. Because one network needs to handle proper control of all weapon types, automatic weapon use can become choppy and intermittent, which only makes sense with single-shot weapons. Therefore, whenever the bot initiates fire with an automatic weapon, it will remain firing as long as its target is available, regardless of whether the network commands it to shoot or not.

Weapons that need to be charged are similar to automatic weapons in that the network controller is likely to release the fire button while charging. The fix for the problem is similar, except that releasing the charge needs to happen while still facing the opponent. In order to make the bot effectively use charged weapons, a random check is used for as long as the bot is charging the weapon: for every time step after starting to charge a weapon, the chance of releasing the charge and firing is 25%. Because the triple rocket attack of the Rocket Launcher takes longer to charge, and is a very useful attack, this percentage is reduced to 15% for this weapon.

Other important features of the Rocket Launcher are that its projectiles are explosive, and take extra time to reach their target. The secondary fire of the Shock Rifle shares these features. Humans adjust to the slowness by firing at locations where they believe their target will be by the time the projectile hits. Therefore, when using the Rocket Launcher or the alternate fire of the Shock Rifle, `UT^2` adjusts its target along the direction of enemy movement with a small amount of random noise whose maximum magnitude along each axis is proportional to the corresponding components of the target's velocity along each axis. In other words, the bot will always aim slightly ahead of its target along the target's direction of movement. Additionally, in order to take advantage of splash damage from explosions that hit the ground near opponents, the bot will further adjust its target down by a small amount whenever its current position is higher than that of the opponent.

Weapons that lob projectiles in an arc are also problematic. Gamebots uses a one-size-fits-all firing command that does not work well for lobbing projectiles. The behavior of the default fire command is neither humanlike nor particularly accurate. The default behavior often results in projectiles lobbed over the heads of opponents. To compensate for this problem, the target for all lobbing projectiles is adjusted to

be a point slightly in front of the opponent along the line between the bot and the opponent. Random noise is used to determine exactly how much to adjust the aim.

Finally, weapons primarily intended for close to middle range are prevented from firing when the bot is too far away from its target. The decisions over which weapons to restrict and to what ranges were made with the help of volunteer human players.

Some of these modifications could effectively be added to the bot after the controlling network is evolved, but one of the main ideas of this chapter is that having these constraints and filters in place before evolution takes place requires evolution to find policies that perform well within the context of these constraints. For example, reducing the accuracy of the Sniper Rifle when moving at high speeds makes the bot more likely to evolve to stand still when using it, which is what humans do.

However, creating a network for the battle controller requires a method for evolving neural networks, which is the topic of the next section.

4 Evolution

Evolutionary Algorithms (EAs) are inspired by Darwin's Theory of Evolution by Natural Selection [7]. Though there are many different types of EAs, they are all population-based search methods. They depend on mutation operators to modify existing solution representations in order to search the space of available solutions. Selection is applied to favor the better solutions for inclusion in the next generation.

Many EAs also involve some form of crossover, which takes two existing solutions and recombines them to form a new solution, sharing traits of each parent. Though crossover is generally considered to be advantageous, there is some evidence [9] that crossover is unnecessary in evolutionary search, and in some circumstances detrimental, since the crossover operation often creates individuals that are highly dissimilar from either parent despite being derived from both of them. Simple mutation, on the other hand, always results in an individual that is a slight variation from its "parent" genotype. In fact, the Evolution Strategy (ES) paradigm relies exclusively on mutation [3]. Based on these arguments and preliminary work evolving with and without crossover, the decision was made to not use crossover in the evolution of `UT^2`. Further details about what methods were used to evolve `UT^2`'s combat behavior are given next.

4.1 Neuroevolution

Neuroevolution is the application of an EA to artificial neural networks. `UT^2`'s combat behavior was learned via constructive neuroevolution, meaning that the networks start with minimal structure and only become more complex as a result of mutations across several generations. The initial population of networks consists of individuals with no hidden layers, i.e. only input and output nodes. Furthermore, these networks are sparsely connected in a style similar to Feature Selective NeuroEvolution of Augmenting Topologies (FS-NEAT [29]). Initializing the networks in this way allows them to easily ignore any inputs that are not, or at least not *yet*, useful. Given the large number of inputs available to `UT^2`, it is important to be able to ignore certain inputs early in evolution, when establishing a baseline policy is more important than refining the policy.

Three mutation operators were used to change network behavior. The weight mutation perturbs the weights of existing network connections, the link mutation adds

new (potentially recurrent) connections between existing nodes, and the node mutation splices new nodes along existing connections. Recurrent connections transmit signals that are not processed by the network until the following time step, which makes them particularly useful in partially observable domains. In the context of reinforcement learning problems [25], such as UT2004, an environment is partially observable if the current observed state cannot be distinguished from other observed states without memory of past states. Recurrent connections help in these situations because they encode and transmit memory of past states. These mutation operators are similar to those used in NEAT [24].

This section explained the representation that was used to evolve policies for `UT^2`. The next section explains the algorithm controlling how the space of policies was searched.

4.2 Evolutionary Multiobjective Optimization

In multiobjective optimization, two or more conflicting objectives are optimized simultaneously. A multiobjective approach is important for domains like UT2004, which involve many conflicting objectives: kill opponents, conserve ammo, avoid damage, etc. Important concepts in dealing with multiple objectives are Pareto dominance and optimality. The following definitions assume a maximization problem. Objectives that are to be minimized can simply have their values multiplied by -1.

Definition 1 (Pareto Dominance). Vector $\mathbf{v} = (v_1, \ldots, v_n)$ dominates $\mathbf{u} = (u_1, \ldots, u_n)$ if and only if the following conditions hold:

1. $\forall i \in \{1, \ldots, n\} : v_i \geq u_i$, and
2. $\exists i \in \{1, \ldots, n\} : v_i > u_i$.

The expression $\mathbf{v} \succ \mathbf{u}$ denotes that $\mathbf{v}$ dominates $\mathbf{u}$.

Definition 2 (Pareto Optimality). A set of points $\mathscr{A} \subseteq \mathscr{F}$ is Pareto optimal if and only if it contains all points such that $\forall \mathbf{x} \in \mathscr{A}$: $\neg \exists \mathbf{y} \in \mathscr{F}$ such that $\mathbf{y} \succ \mathbf{x}$. The points in $\mathscr{A}$ are non-dominated, and make up the non-dominated Pareto front of $\mathscr{F}$.

The above definitions indicate that one solution is better than (i.e. dominates) another solution if it is strictly better in at least one objective and no worse in the others. The best solutions are not dominated by any other solutions, and make up the Pareto front of the search space. Therefore, solving a multiobjective optimization problem involves approximating the Pareto front as best as possible, which is exactly what EMO methods do. In particular, the EMO method used in this work is the Non-Dominated Sorting Genetic Algorithm II (NSGA-II [8]).

NSGA-II uses a $(\mu + \lambda)$ selection strategy. In this paradigm, a parent population of size μ is evaluated, and then used to produce a child population of size λ. Selection is performed on the combined parent and child population to give rise to a new parent population of size μ. NSGA-II uses $\mu = \lambda$.

NSGA-II sorts the population into non-dominated layers in terms of each individual's fitness scores. For a given population, the first non-dominated layer is simply the Pareto front of that population (usually not the same as the true Pareto front of

the search space). If this first layer is removed, then the second layer is the Pareto front of the remaining population. By removing layers and recalculating the Pareto front, the whole population can be sorted. Individuals in layers dominated by fewer other layers are considered more desirable by evolution.

Elitist selection favors these individuals for inclusion in the next parent generation. However, a cutoff is often reached such that the non-dominated layer under consideration holds more individuals than there are remaining slots in the next parent population. These slots are filled by selecting individuals from the current layer based on a metric called *crowding distance*.

The crowding distance for a point p in objective space is the average distance between all pairs of points on either side of p along each objective. Points having an objective score that is the maximum or minimum for the particular objective are considered to have a crowding distance of infinity. For other points, the crowding distance tends to be bigger the more isolated the point is. NSGA-II favors solutions with high crowding distance during selection, because the more isolated points in objective space are filling a niche in the trade-off surface with less competition.

By combining the notions of non-dominance and crowding distance, a total ordering of the population arises by which individuals in different layers are sorted based on the dominance criteria, and individuals in the same layer are sorted based on crowding distance. The resulting comparison operator for this total ordering is also used by NSGA-II: The way that a new child population is derived from a parent population is via binary tournament selection based on this comparison operator.

Applying NSGA-II to a problem results in a population containing a close approximation to the true Pareto front (an approximation set) with individuals spread out evenly across the trade-off surface between objectives. The details of how this process was carried out in UT2004, as well as an explanation of how one network was selected from the resulting Pareto front, are covered in the next section.

4.3 Evolution of `UT^2`

How can the above techniques be used to generate a network for `UT^2`'s battle controller? In order to evolve bots for UT2004, fitness objectives need to be designed to favor good behavior, opponents against which the bots can evolve need to be chosen, and maps within which the Deathmatches will occur are needed. After evolving a population in this manner, the results of evolution need to be examined in order to pick an appropriate network to serve as the brain for `UT^2`'s battle controller.

4.3.1 Fitness Objectives

Some of the objectives used to evolve `UT^2` were the same as those used in [15] (Damage Dealt, Accuracy, Damage Received), though additional objectives were added to discourage collisions with level geometry and other agents, since such collisions are characteristic of bot-like behavior.

- Damage Dealt: This objective measures both kills and damage dealt by the bot. It is possible to kill an opponent without being responsible for depleting all of its hit points, but the kill is still attributed to whoever delivered the final hit. Therefore, for each of the bot's kills, this fitness measure rewards it with an extra 100 fitness,

since 100 is the starting health of all agents. Additionally, the bot keeps track of how much damage it has dealt so far to each opponent. These amounts are reset to zero when the corresponding opponent dies. At the end of the match, whichever value is highest is added to the fitness score. Thus this fitness measure rewards kills, as well as additional damage that comes short of a successful kill.

- Accuracy: This objective measures the accuracy of the bot in hitting opponents. In section 3.2.3 some restrictions on `UT^2`'s accuracy were described. These restrictions can be overcome if the bot chooses to stand still or otherwise move such that it can aim better. The exact measure used is the number of hits divided by the amount of ammo used. This measure works well for most weapons, but it has become clear since the competition that it does not make sense for some weapons. For example, each shard fired by the Flak Cannon registers as a separate hit, and the secondary fire of some weapons consumes more than one unit of ammunition even though they may only register a single hit when successful.
- Damage Received: This objective needs to be minimized. Each time the bot dies, it counts as 100 damage received. However many hit points fewer than 100 the bot has at the end of a match are added to this amount.
- Level Collisions: A level collision registers whenever the bot bumps into some aspect of level geometry, usually a wall. Because these collisions look awkward, and can lead to the bot getting stuck, the goal of looking human requires the bot to minimize collisions of this type.
- Agent Collisions: Bumping into other agents in the world can also look awkward and should be avoided, so this is another objective to be minimized.

Though each of the above objectives measures an aspect of performance that is important in a skilled bot, it is not necessarily the case that this is the best set to evolve with in order to discover quality Deathmatch behavior. In particular, it would probably have been better to evolve with fewer objectives, since NSGA-II's performance is known to degrade with increased numbers of objectives. In future work, it would make sense to combine the collision objectives into a single objective, or perhaps simply drop them both. The accuracy objective is also problematic, as described above, and will need to be fixed before use in future competitions.

One meta-objective was also used in order to help evolution effectively explore the range of possible behaviors. Behavioral diversity [20, 22] was used to encourage different types of behaviors to assure that evolution did not get stuck in local optima. The objective is a generalized form of behavioral diversity [22] that uses a different set of randomized input vectors per generation to generate a behavior vector for each individual in the population. A behavior vector is the concatenation of all output vectors derived from processing each of the randomized input vectors through an individual's neural network. The behavioral diversity objective is to maximize the average distance of an individual's behavior vector in Euclidean space from all other behavior vectors in the population, thus favoring diverse network/agent behavior.

4.3.2 Agents

Given these objectives, decisions still need to be made regarding the scenario in which the bot will evolve. Because BotPrize involves competing simultaneously against multiple opponents, it was decided that the bot should evolve in a similar

scenario. Evolving against native UT2004 bots would have been ideal, but because Pogamut 3 was fairly new at the time UT^2 was being developed, an easy way to do this was not yet available (support has since been added). Therefore, the opponents for the evolving bots were instances of the Hunter bot, a simple but effective scripted bot that is provided with the Pogamut 3 platform. Specifically, during evolution one bot participated in a Deathmatch against five Hunter bots per evaluation.

In order to evolve a battle controller that would eventually be used by UT^2 in the competition, a slightly modified version of UT^2's architecture was used. The architecture presented earlier (section 3.1) was changed in two ways. First, the *JUDGE* module was disabled (the Hunters could not judge either), since in a scenario consisting entirely of bots there is no sense in judging. Secondly, the *HUMAN TRACE CONTROLLER* of the *UNSTUCK* module was replaced with a simple hand-coded controller for getting unstuck. This SimpleUnstuckController tries to move away from any obstacle that it collides with, and resorts to one of several random movement actions if it is stuck for some other reason. The SimpleUnstuckController was used because during evolution the version based on human traces was not yet fully developed. Both the SimpleUnstuckController and the *HUMAN TRACE CONTROLLER*, which was actually used by UT^2 in the competition, are described in full detail elsewhere in this book [17].

4.3.3 Maps/Levels

When evolving UT^2's battle controller, the exact level used and the length of evaluation depended on the current generation. The sequence of levels used was meant to increase in size and challenge, so that early generations would have a chance to learn basic behaviors before having to deal with more complicated situations. Evaluations in earlier levels were shorter than in later levels, both because more time is required to find enemies in larger levels, and because it is not worthwhile to evaluate networks for a long time in early generations, since much time would likely be wasted on bad solutions. The exact level/duration sequence was DM-TrainingDay/100 seconds, DM-Corrugation/200 seconds, DM-DG-Colosseum/300 seconds, DM-GoatswoodPlay/400 seconds, and then DM-IceHenge/500 seconds.

Though this sequence served well for the purpose of evolving combat behavior, competition experience has indicated that a better sequence is likely possible. For example, although Goatswood and IceHenge have challenging water hazards, Colosseum is difficult for the bot to deal with because it is easy to get lost and stuck in the columns. Furthermore, lessons learned by bots in earlier levels may have been forgotten in order to better specialize in later levels. Though the cost in evaluation time would be high, in future work it might be better to have bots face Deathmatches in multiple levels per evaluation.

Twenty generations were spent on DM-TrainingDay, and ten generations on each subsequent level, for a total of 60 generations. This is a very small number of generations, and better performance could likely have been achieved given more time. However, evaluation time is a major bottleneck in UT2004, so a lesser number of generations was used. For the same reason, the population size was only 20, which due to NSGA-II's $(\mu + \lambda)$ selection strategy meant that each generation in-

volved selection upon a population of size 40. This number is fairly small for evolutionary computation, but good results were obtained despite this practical restriction.

4.3.4 Results of Evolution

Fig. 5 compares values of the hypervolume indicator [32] for both the starting and final parent populations in each of the levels of BotPrize. The hypervolume indicator measures the hypervolume of the region dominated by all points in a given approximation to a Pareto front. The hypervolume indicator is special in that it is a Pareto-compliant metric [31], meaning that an approximation set that completely dominates another approximation set is guaranteed to have a higher hypervolume.

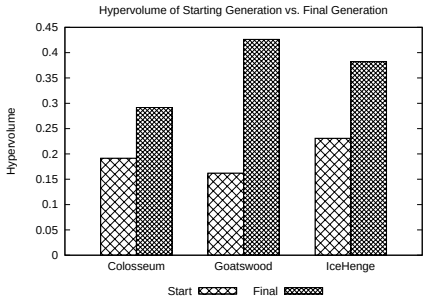


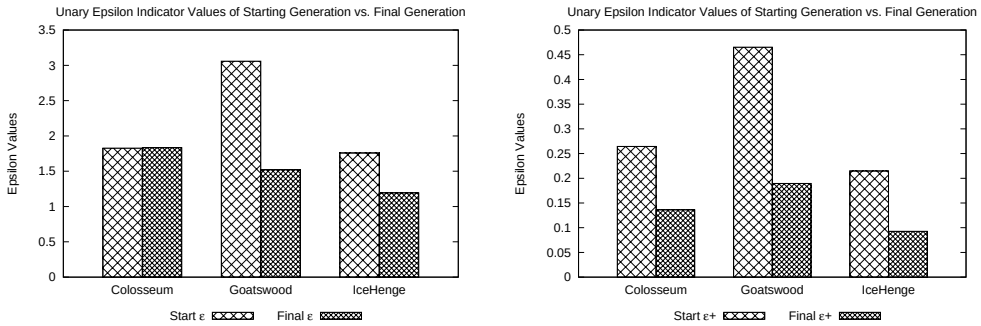
Fig. 5 Hypervolume of First Generation vs. Final Generation. This figure shows the gains made by evolution in each of the three levels used in BotPrize. In order to get accurate scores, each of the 20 members of both the start and final parent generations was evaluated in each level for 500 seconds against five Hunter bots ten times each. The objective scores for each individual were the averages of scores attained in each objective across the ten trials. Pareto fronts of the resulting scores were calculated, and the scores for each objective were normalized according to the maximum magnitude scores for each objective in the given map (hypervolumes between maps are not comparable). These normalized fronts were used to calculate hypervolume. In each level, the hypervolume in the final generation is greater than in the start generation, showing that the population evolved to dominate a larger region of objective space across generations.

Other Pareto-compliant metrics are the multiplicative (I^1_ε) and additive ($I^1_{\varepsilon+}$) unary epsilon indicators [18]. Both indicators are defined with respect to a reference set R. The multiplicative indicator I^1_ε measures how much each objective for each solution in a set would have to be multiplied (divided for minimization) by such that each solution in R would be dominated by or equal to a point in the resulting set. The additive indicator $I^1_{\varepsilon+}$ measures how much would have to be added (subtracted for minimization) to each objective in each solution such that each point in R would be dominated by or equal to a point in the modified set. For both indicators, smaller values are better because they indicate that a smaller adjustment is needed to dominate the reference set.

The scores from the start and end generations were compared using these unary epsilon indicators with a separate reference set for each level defined as the super Pareto front (Pareto front of several Pareto fronts) of the fronts from the start and end generations, as suggested in [18]. The results are shown in Fig. 6.

The evolved population has better hypervolume and epsilon values, but it is actually not the case that the approximation sets from the final generation are strictly better than those from the starting generation, although the sets from the final generation do tend to contain points that completely dominate points in the first generation sets. The lack of complete domination is likely caused by use of such small popula-

tions and so many objectives, which in combination make it hard for the population to cover all trade-offs.



(a) Multiplicative Indicator I_{ε}^{1} (b) Additive Indicator $I_{\varepsilon+}^{1}$

Fig. 6 Epsilon Indicator Values of Starting Generation vs. Final Generation. The normalized Pareto fronts used to compute the hypervolumes for the first and final generations in each level were used to compute unary epsilon indicator values with respect to reference sets, which were the super Pareto fronts of the two approximation sets under consideration in each level. With the exception of the I_{ε}^{1} values for Colosseum, all epsilon values indicate that the solutions in the final generation are better than those in the first generation.

4.3.5 Network Selection

There are still many trade-offs to consider, however, and one network had to be selected from all those available to compete in BotPrize 2010. In order to get a bot that performed well, the population was first filtered based on the highest Deathmatch scores across all levels, in a manner similar to [15]. This process resulted in a set of three high-scoring networks. Each of these three networks attained a high score by being aggressive, which was considered a human trait. These bots also tended to die more as a result of their aggressiveness (more on this in section 5.1).

The final decision of which network to use in BotPrize was made by the authors along with the help of two human volunteers. In Deathmatches between four humans (two of which were involved in programming the bot), the three candidate bots, and one native UT2004 bot, the humanness ratings (number of human judgements divided by total judgements) across multiple matches were used to single out the most humanlike bot of the available candidates. This bot became `UT^2` in the 2010 BotPrize competition.

5 Evaluation

Having fully described how `UT^2` was developed, it is now time to evaluate `UT^2` to see how well it performs in UT2004. `UT^2`'s performance is analyzed both in terms of its ability to achieve high fitness scores, and in terms of how humanlike the judges in BotPrize 2010 considered it to be.

5.1 Evaluation of Objective Scores

The ultimate goal of `UT^2` is to look as humanlike as possible, but the route to accomplishing this goal was evolving for good objective performance. This section deals with the quality of `UT^2`'s performance with respect to the objectives used in evolution as well as Deathmatch score, which played an important role in deciding which network from the Pareto front to use in the competition.

Fig. 7 compares the performance of `UT^2` using the evolved network chosen for BotPrize 2010 with the same bot using a randomized action selector for the battle controller. Specifically, randomized vectors are treated like output vectors from a network in order to determine combat behavior. In these evaluations, both versions of the bot had access to the complete human-trace-based *UNSTUCK* module [17] used in the final competition. Evaluations were performed for 500 seconds in each competition level against five Hunter bots.

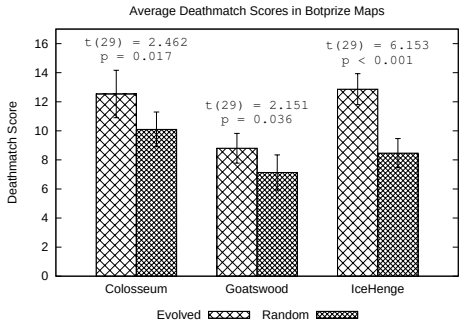
The results show that purposefully picking a network based on Deathmatch score and aggressiveness has resulted in an ability to deal significantly more damage, and therefore get significantly better scores, than a random battle controller. Accuracy was generally better too, though only significantly so in Colosseum.

However, favoring aggressive, score-increasing behavior has resulted in significantly more damage received in all levels. This result highlights the importance of a multiobjective approach in helping to find the best trade-off between objectives. It makes sense that aggressively pursuing enemies and actively engaging in combat will result in both more frags earned, as well as more deaths experienced.

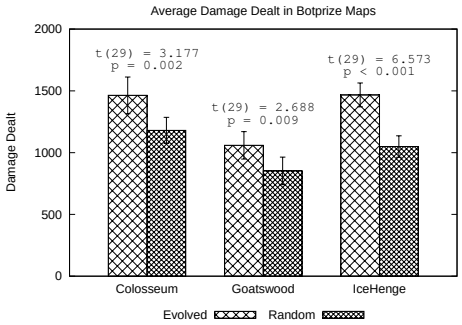
In terms of collisions with level geometry, differences between the evolved network and the random bot were inconsistent across levels. In general, this behavior seems more level-dependent than bot-dependent: level collisions are more common in Goatswood than in IceHenge and more common in Colosseum than in either of the other levels. Since the design of the bot depended primarily on ray traces to detect surrounding obstacles, and these ray traces were unavailable in the competition (section 3.2.1), it is not surprising that collision behavior does not seem strongly affected by evolution. Though battle style clearly affects collision frequency, this objective did not play as important a role in final network selection as the others did; this decision may have been a mistake (section 5.2).

In terms of collisions with enemy agents, the evolved network is worse than the random controller. Once again, this behavior is a result of favoring aggressive combat behavior. Many collisions occur because the bot is chasing the opponent. An aggressive player is more likely to be near its opponents, and therefore also likely to bump into them more often. Furthermore, these results are based on battle against Hunter bots, which mindlessly rush at their opponents in a way unlike humans. The behavior of the Hunter bots made agent collisions even more likely. Since in all cases the actual number of collisions is fairly small (averages below 13), it was assumed that the number of collisions would drop to an insignificant level when fighting human opponents.

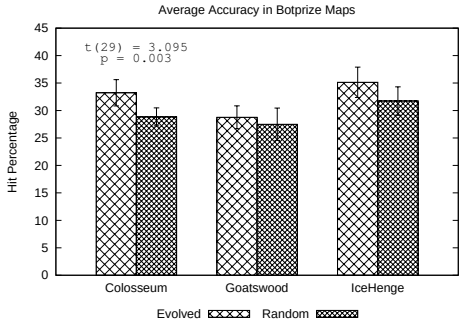
Given the priorities across objectives, the evolved network has succeeded in performing well in the Deathmatch domain. To what degree this good empirical performance translated into humanlike performance in the competition is discussed next.



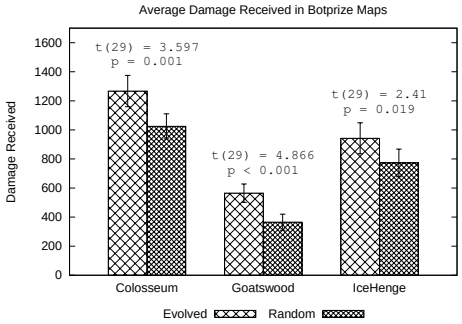
(a) Average Deathmatch Score



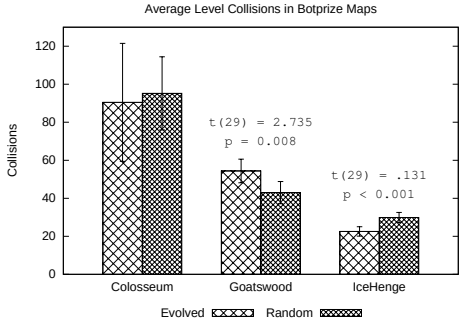
(b) Average Damage Dealt



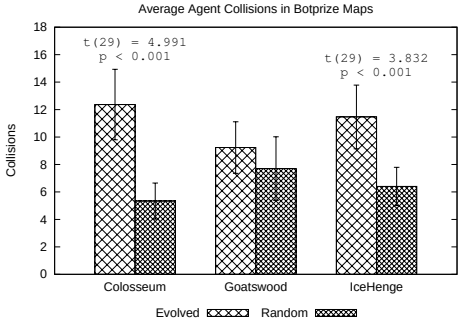
(c) Average Accuracy



(d) Average Damage Received



(e) Average Level Collisions



(f) Average Agent Collisions

Fig. 7 Each figure compares the average performance of the evolved network from the competition bot to that of the bot using random action selection in the battle controller. Performance in each of the three levels used in BotPrize 2010 is shown. Bot performance is measured in competition with five Hunter bots in 500 second matches. Averages are across 30 trials, and 95% confidence intervals are shown. For each score and level, t-tests were done to compare the evolved network to random action selection. When the difference is significant, the resulting t and p values for the test are shown above the bars. Though the evolved network is not significantly better than a random controller in some objectives, its performance in these objectives can be attributed to focusing on high-scoring networks when the evolved network was chosen from those available on the trade-off surface.

5.2 Evaluation of Humanlike Performance

Bot	Humanness
Native UT2004 Bot	35.3982%
Conscious-Robots	31.8182%
UT^2	27.2727%
ICE-2010	23.3333%
Discordia	17.7778%
w00t	9.3023%

Table 1 BotPrize 2010 Results (UT^2 highlighted). Humanness equals the number of human judgments divided by the total judgments, all multiplied by 100. UT^2 beat three entries to get 2nd place.

The results from BotPrize 2010 are in Table 1. UT^2 placed 2nd among entrants, though the humanness rating of the native UT2004 bots is also shown (native bots have the advantage of being written in UT2004's UnrealScript, which has some advantages over Gamebots in terms of sensing, latency and action execution). The lowest humanness rating for a human was 35.4839%.

UT^2 did not win, but it did beat three other entrees, losing only to Conscious-Robots. No competitor has yet won the grand prize, requiring a humanness rating of at least 50%. In fact, of the seven human judges, only two had humanness ratings over 50%. However, this result could be in part due to the fact that the new judging format actually encourages humans to act like bots in order to trick opponents into losing points for bad judgments. This strategy is one of the distortion effects mentioned in [30].

In any case, the humans were still clearly more human than the bots, although compared to previous competitions the gap is narrowing. Based on the many demo files made during the competition, some analysis of bot behaviors related to specific judgments is possible. Demos and logs of the competition are available online[1]. Each one allows a viewer to see exactly what any given judge saw during specific matches of the competition.

Most actions taken by UT^2 seem fairly human, or are at least difficult to distinguish from human actions. This statement is based on the fact that most humans interacted with UT^2 on several distinct occasions before making any sort of judgment. However, such extensive interactions make it hard to discern what aspect of the bot's behavior influenced the judgment. Despite this difficulty, certain behaviors were noticed that judges tended to associate, though not always correctly, with either bots or humans.

Most judges assumed it was human to stand still for long periods with sniping weapons while being oblivious to nearby surroundings. Although UT^2 would sometimes stand still to get a better shot when firing, these pauses were usually brief moments between dodging actions. This behavior is one case where UT^2's behavior was more effective in combat, but less humanlike. However, lack of this behavior did not seem to cause UT^2 to be judged as a bot; it simply meant that UT^2 missed some chances to be judged as human.

One problematic behavior exhibited by UT^2 is actually a bug that is not consistent with the description of the battle controller in section 3.2. In order to appear attentive, the bot is supposed to look at the targeted opponent during all combat actions. However, this was actually not the case for the *GOTO ITEM* action: the bot

[1] http://botprize.org/result.html

would look at, and sometimes even shoot in the direction of, the item towards which it was moving instead of the opponent it was fighting. This bug caused UT^2 to be judged as a bot on several occasions.

Other issues seem to be more level specific. Table 2 breaks down judgments against UT^2 by level, and shows that the bot faired best in IceHenge and worst in Goatswood. The effect of the level on the humanness rating of the bot is closely tied to its ability to navigate within that level. Judgements of UT^2 that seem to have been based on its navigational abilities are discussed in [17].

Map	Human	Bot	Humanness
Colosseum	2	6	25.00%
Goatswood	2	9	18.18%
IceHenge	5	9	35.71%

Table 2 Number of judgments of each type against UT^2 and the resulting humanness, divided by map. The bot was least human in Goatswood, which is unfortunate because five sessions were played in Goatswood, whereas four were played in IceHenge and three in Colosseum.

The current level also affected the bot's combat behavior. UT^2 likely faired well in IceHenge because most areas are wide open with few obstacles. Also, the fact that the last ten generations of evolution were spent in IceHenge probably made the bot's behavior better tailored to this level than others. In contrast to IceHenge, Goatswood is mostly comprised of narrow corridors and has several waist-high obstacles over which the players must jump. A few of the bot judgments that UT^2 received in Goatswood seem to be the result of the bot unnecessarily colliding with walls, however briefly, in the midst of dodging during combat. The judges presumably expected humans to be more aware of their surroundings so as to avoid such contact. These judgments indicate that the Level Collisions objective should have been considered more important when deciding which network to use in BotPrize.

Humans also expected other humans to be aware of the judging aspect of the competition. Some judges would purposefully miss with the judging gun in combat to see if they could elicit human reactions from their opponents. It is impossible to know what individual judges expected in these situations, but completely ignoring the judging gun and attacking as normal seems to have been considered bot-like. UT^2 was labelled a bot at least once for such behavior.

Humans also expected humans to use the judging gun. There are some occasions where UT^2 killed a human with a correct judgment, and was in turn immediately judged as a human by the judge the next time the two met. There are other occasions where the exact reason UT^2 was judged as a human was unclear due to the large number of interactions preceding the judgment, but in most cases where a judge saw UT^2 several times before judging it as human, at least one of the things the judge witnessed was UT^2 using the judging gun.

However, despite the role that judging behavior may have played in earning human judgments for UT^2, it is not necessarily true that judging behavior is vital to the competition. Neither the winner, Conscious-Robots [2], nor the more human native UT2004 bot did any judging at all. It seems that bots can get away with not judging and still look human due to the fact that most interactions are brief and spaced out across the match. In other words, there are many chances to use the judg-

ing gun out of sight of any given opponent, so no human would necessarily expect to see every opponent use the judging gun.

Other judgments against UT^2 are harder to interpret. Sometimes a judge saw UT^2 many times in a match, and eventually judged the bot as a human near the very end. Such judgments likely indicate that over the course of several interactions the bot did nothing overtly bot-like, and the most sensible course of action given little remaining time was to judge the bot as human.

In a few cases, UT^2 was quickly judged based on very little interaction. It is not clear from the demo replays what criteria the judges were using in these cases. It is possible in some cases that judges are able to discern the identity of an opponent simply by subtle movement patterns within mere seconds, but it is also possible that some judgments are the result of errors, such as mistaken identity or weapon misfire. Throughout all of the demo files there are many instances of snap judgments, both correct and incorrect.

Still, regardless of the reason behind such judgments, they must be accounted for in order to succeed at BotPrize. Ideas on how to do this, as well as some general ideas about how a bot can appear more human, are the topic of the next section.

6 Discussion and Future Work

Most of UT^2's behavior seems to be passably human. Many judges were unable to come to a conclusion about the bot's humanness, even after three or more interactions. However, UT^2 would look more human if it both performed certain actions that most humans are certain a human would do, and if it avoided the few very bot-like actions that crept into its behavior.

Fixing the bug caused by the *GOTO ITEM* action is simple. The availability of working ray-traces in future competitions should also help the bot avoid bumping into obstacles as often. There are also ideas for improving navigation with the use of human traces, discussed in [17]. With regards to how the combat behavior was evolved, there is room for improvement.

Obvious steps to improve the performance of the bot would be to evolve with a larger population for more generations, but there are also ways in which the basic evolutionary setup can be improved. These improvements are discussed below.

6.1 Opponent Interactions

One issue regards the opponents against which bots evolve. For practical reasons, these opponents are themselves bots. The Hunter bot was used to evolve UT^2, though the native bots would probably make better opponents. However, it would likely be even better to evolve against many different types of bots. The justification for this approach is that each of the BotPrize participants had a different play style and skill level. An evolved bot should be accustomed to the possibility that different opponents behave differently, and more importantly, a bot evolved against varied opponents is more likely to learn behaviors to deal with different types of players. In retrospect, evolving against the Hunter only may have resulted in the evolution of a one-size-fits-all behavior that is mostly effective, but perhaps too predictable and/or bot-like. Humans are very good at adapting and improvising. Having learned how to respond to a wide array of opponent strategies should at least give the impression that the bot is improvising.

An important concept when considering how agents interact is "attention": humans pay attention to the agents they interact with, and generally continue to do so until some note-worthy event shifts that attention. The opponent-relative movement commands of UT^2 assure that it pays attention to whichever opponent it is fighting, but when multiple opponents are present, UT^2 picks one of them to pay attention to according to a scripted routine (see under "Enemy Shooting Sensor" in section 3.2.1). It might be more humanlike to use a cognitive approach to this attention problem as done by the winning bot Conscious-Robots [2].

Linked to the issue of interaction is the idea of mimicry. Mimicry is important because it establishes an agent's ability to comprehend what another agent is doing, and utilize that knowledge for its own gain. Mimicry can involve copying what an opponent is doing at the moment, or it can mean that agents mirror each other's behavior, such that one is always countering the other to maintain equilibrium. An example of the first type would be jumping or dodging in a similar manner to an opponent. An example of the second type would be maintaining distance during combat, such that the bot moves forward when its opponent moves backward and vice versa. In either case, such behavior would make a bot look more human when fighting a human judge, since the bot would be acting the way the human judge acts.

One potential way to make such mimicry evolvable is to have opponent-relative input sensors in addition to opponent-relative actions. Rather than simple awareness of where an opponent is, the bot could sense whether the opponent was advancing, retreating, strafing, jumping, etc. If the bot can sense when an opponent is performing an action that it can also perform, learning to act the same way via a neural network would be quite easy. Of course, such behavior would only be favored by evolution if it also improved fitness, but given that humans favor such strategies it is believed that mimicking behavior will indeed lead to increased fitness.

Mimicry could be more directly encouraged by evolution if some measure of mimicry were used as a fitness function. However, rewarding mimicry directly could result in evolved bots that behave in non-human ways when interacting with human judges that behave stupidly as a ruse to gauge humanness, or with other bots, which are of course bot-like.

6.2 Scope of Evolved Policy

A limitation on how UT^2's behavior was evolved is that only the battle controller was evolving within a bot that had many other components. While this approach assured that its combat behavior would make sense in the context of its other behaviors, the approach is perhaps too inefficient. It may be better to evolve the combat behavior separately, at least initially, within a specialized combat scenario where the majority of the bot's other modules are disabled. For example, the bot could be given infinite ammo in a small level, thus making navigation and weapon collection irrelevant, and freeing evolution to focus on how the bot behaves in combat.

Such an evolved battle controller could be integrated with other evolved subcontrollers to build up a hierarchical controller as was done by van Hoorn et al. [15], whose approach was directly based on that of Togelius [26]. The method used in these works involves evolving the components of a subsumption architecture [4] within several separate subtasks leading up to the full task. Learning good behav-

ior for many small tasks in an incremental way is easier for most machine learning methods, but requires a knowledgeable human to construct the training and control hierarchies that define the agent's final behavior.

Though such approaches may make learning easier, for any evolved subcomponent, one must keep in mind that evolution will favor increased fitness, potentially at the cost of human behavior. In order for an evolved controller to act like a human, it must be both constrained as humans are and allowed to easily carry out actions that are common and easy for humans to carry out. However, the work of determining the proper constraints is task specific, and requires some knowledge both of how humans perform in the task, and of how a bot is likely to cheat at the task.

6.3 Weapon Usage

Evolving the battle controller in isolation would make it easier to control what weapons the bot has. It would then be possible to evolve a specialized controller for each weapon, or at least each class of weapons. Humans expect other humans to have varied behavior across weapons. Sniping weapons are an obvious example. Although `UT^2` knew some information about its current weapon, this information may not have been enough to serve as a basis for different combat styles, especially since some of the information was faulty (section 3.2.1). Simply having separate controllers for each weapon would assure the bot's behavior matched the weapon. However, such an approach would take even more time to evolve, and would be brittle with respect to new weapons, even if they were similar to existing ones.

An alternative option is to evolve multi-modal networks [21]. The networks have distinct output modes for different situations, which seems well-suited to having different behaviors for different types of weapons.

Knowing which weapon to use in a given situation is also an important aspect of gameplay in UT2004. In past BotPrize competitions, the University of Texas at Austin's entry (named `U Texas`) learned weapon preferences automatically [13]. The bot would learn estimates of the expected damage and accuracy of each weapon in each of the three ranges used by `UT^2`'s static weapon lookup rules.

The original intent was to integrate weapon preference learning into `UT^2`, but this learning method was afflicted with the same problems that make the Accuracy objective (section 4.3) problematic. Basically, the Gamebots protocol registers each source of damage separately, which makes gauging the accuracy of weapons that fire multiple projectiles at once difficult. Furthermore, accuracy is not as important for weapons that have higher rates of fire. Because of these difficulties, it was decided that `UT^2` would use static weapon preferences instead. However, including weapon preference learning is still a good idea, which future versions of `UT^2` will likely include, once a way to work around the limitations of Gamebots is found.

7 Conclusion

Evolving neural networks to provide the combat behavior for the `UT^2` bot in UT2004 helped it earn 2nd place in the 2010 BotPrize competition. The key to evolving humanlike behavior, despite evolving for raw performance, is to restrict the actions available to the bot to common human actions, and to filter the overall bot behavior such that the bot is restricted in ways that humans are. Evolving the bot to perform well in the context of human limitations naturally results in humanlike

performance. The UT^2 bot focused on the most obvious of such limitations, but much more is possible. By further tailoring bot actuators and sensors in this manner it should be possible to evolve more humanlike bots for UT2004 and other domains in the future.

Acknowledgements The authors would like to thank Niels van Hoorn for the use of his source code in getting started evolving bots in UT2004. They would also like to thank Christopher Tanguay and Peter Djeu for volunteering to critique and evaluate versions of UT^2. This research was supported in part by the NSF under grants DBI-0939454 and IIS-0915038 and Texas Higher Education Coordinating Board grant 003658-0036-2007.

References

1. Adobbati, R., Marshall, A.N., Scholer, A., Tejada, S.: Gamebots: A 3D virtual world testbed for multi-agent research. In: In Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS (2001)
2. Arrabales, R., Munoz, J., Ledezma, A., Gutierrez, G., Sanchis, A.: A Machine Consciousness Approach to the Design of Human-like Bots. In: P.F. Hingston (ed.) Believable Bots. Springer (2011). (To Appear)
3. Bäck, T., Hoffmeister, F., Schwefel, H.P.: A survey of evolution strategies. In: Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 2–9 (1991)
4. Brooks, R.A.: A robust layered control system for a mobile robot. IEEE Journal of Robotics and Automation **2**(10) (1986)
5. Bryson, J.J.: Intelligence by design: principles of modularity and coordination for engineering complex adaptive agents. Ph.D. thesis, Massachusetts Institute of Technology (2001)
6. Butz, M., Lonneker, T.: Optimized sensory-motor couplings plus strategy extensions for the TORCS car racing challenge. In: Computational Intelligence and Games, pp. 317–324 (2009)
7. Darwin, C.: On the Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life. Murray, London (1859)
8. Deb, K., Agrawal, S., Pratab, A., Meyarivan, T.: A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. PPSN VI pp. 849–858 (2000)
9. Fogel, D.B., Atmar, J.W.: Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems. Biological Cybernetics **63**(2), 111–114 (1990)
10. Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., C., B.: Pogamut 3 can assist developers in building AI (not only) for their videogame agents. Agents for Games and Simulations, LNCS 5920 (2009)
11. Gomez, F., Miikkulainen, R.: Active guidance for a finless rocket using neuroevolution. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 2084–2095. Morgan Kaufmann, San Francisco (2003). URL http://nn.cs.utexas.edu/keyword?gomez:gecco03
12. Haykin, S.: Neural Networks, A Comprehensive Foundation. Prentice Hall, Upper Saddle River, New Jersey (1999)
13. Hingston, P.: A Turing Test for Computer Game Bots. IEEE Transactions on Computational Intelligence and AI in Games **1**(3), 169–186 (2009)
14. Hingston, P.: A New Design for a Turing Test for Bots. In: Computational Intelligence and Games (2010)
15. van Hoorn, N., Togelius, J., Schmidhuber, J.: Hierarchical controller learning in a first-person shooter. In: Computational Intelligence and Games, pp. 294–301 (2009)
16. Isla, D.: Managing Complexity in the Halo 2 AI System. In: Proceedings of the Game Developers Conference. San Francisco, CA (2005). URL http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml
17. Karpov, I.V., Schrum, J., Miikkulainen, R.: Believable Bot Navigation via Playback of Human Traces. In: P.F. Hingston (ed.) Believable Bots. Springer (2011). (To Appear)
18. Knowles, J., Thiele, L., Zitzler, E.: A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. TIK Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich (2006)

19. Kohl, N., Miikkulainen, R.: Evolving neural networks for strategic decision-making problems. Neural Networks, Special issue on Goal-Directed Neural Systems (2009)
20. Mouret, J.B., Doncieux, S.: Using behavioral exploration objectives to solve deceptive problems in neuro-evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 627–634. ACM (2009). DOI http://doi.acm.org/10.1145/1569901.1569988
21. Schrum, J., Miikkulainen, R.: Evolving Multi-modal Behavior in NPCs. In: Computational Intelligence and Games, pp. 325–332 (2009). URL http://nn.cs.utexas.edu/?schrum:cig09
22. Schrum, J., Miikkulainen, R.: Evolving agent behavior in multiobjective domains using fitness-based shaping. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 439–446. Portland, Oregon (2010). URL http://nn.cs.utexas.edu/?schrum:gecco10
23. Stanley, K.O., Bryant, B.D., Karpov, I., Miikkulainen, R.: Real-time evolution of neural networks in the NERO video game. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence (2006). URL http://nn.cs.utexas.edu/keyword?stanley:aaai06
24. Stanley, K.O., Miikkulainen, R.: Evolving Neural Networks Through Augmenting Topologies. Evolutionary Computation **10**, 99–127 (2002). URL http://nn.cs.utexas.edu/keyword?stanley:ec02
25. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998). URL citeseer.ist.psu.edu/sutton98reinforcement.html
26. Togelius, J.: Evolution of a subsumption architecture neurocontroller. Journal of Intelligent and Fuzzy Systems pp. 15–20 (2004)
27. Turing, A.M.: Computing machinery and intelligence. Mind **59**(236), 433–460 (1950)
28. Waibel, M., Keller, L., Floreano, D.: Genetic Team Composition and Level of Selection in the Evolution of Multi-Agent Systems. Evolutionary Computation **13**(3), 648–660 (2009)
29. Whiteson, S., Stone, P., Stanley, K.O., Miikkulainen, R., Kohl, N.: Automatic Feature Selection in Neuroevolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (2005). URL http://nn.cs.utexas.edu/keyword?whiteson:gecco05
30. Yannakakis, G.N., Togelius, J., Shaker, N.: Assessing Believability as a Spectator. In: P.F. Hingston (ed.) Believable Bots. Springer (2011). (To Appear)
31. Zitzler, E., Brockhoff, D., Thiele, L.: The Hypervolume Indicator Revisited: On the Design of Pareto-compliant Indicators Via Weighted Integration. In: Conference on Evolutionary Multi-Criterion Optimization (EMO 2007), vol. 4403, pp. 862–876 (2007)
32. Zitzler, E., Thiele, L.: Multiobjective optimization using evolutionary algorithms - a comparative case study. In: Parallel Problem Solving from Nature, pp. 292–304 (1998)

Index