StarCraft Unit Motion: Analysis and Search Enhancements

Douglas Schneider and Michael Buro
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada, T6G 2E8
{ds3|mburo}@ualberta.ca

Abstract

Real-time strategy (RTS) games pose challenges to AI research on many levels, ranging from selecting targets in unit combat situations, over efficient multi-unit pathfinding, to high-level economic decisions. Due to the complexity of RTS games, writing competitive AI systems for these games requires high speed adaptive algorithms and simplified models of the game world.

In this paper we focus on motion prediction and motion planning in StarCraft — a popular RTS game for which a C++ API exists that allows us to write AI systems to play the game. We explore our existing unit motion model of StarCraft and find and fix some inconsistencies to improve the model by accounting for systematic command execution delays and unit acceleration. We then investigate ways to improve existing combat motion planning systems that are based on discrete unit motion sets, and show that search-based algorithms and scripts can benefit from using a new direction set that considers moves towards the closest enemy unit, away from it, and perpendicular to both directions.

Introduction

In this paper we examine AI in the context of the game StarCraft. StarCraft is popular a real-time strategy (RTS) game which involves the simulation of a war between two or more players. There are many important components to this type of game such as resource management, unit production planning, and high level and low level combat strategies. When designing an AI system to play a complex game such as StarCraft, it makes sense to break the design of the bot into several smaller modules that each individually solve less complex AI problems (Ontañón et al. 2013). In this paper we focus on the UAlbertaBot StarCraft bot, the champion bot of the 2013 AIIDE StarCraft Competition (Churchill 2013) and its motion prediction and planning.

UAlbertaBot is broken down into modules designed to separately solve each of the problems mentioned above. Each of these modules has unique challenges. One of the weaker parts of UAlbertaBot is its low level combat, at the unit or squad level. This includes moving individual units around in combat, as well as choosing targets for each unit.

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this paper we analyze the model that is used by UAlbertaBot to simulate low level combat. We present measured error in the model, analyze its source, and present new error measurements after the model has been corrected. Improving the model of combat will allow UAlbertaBot to better model the results of it's actions and should lead to improved combat performance.

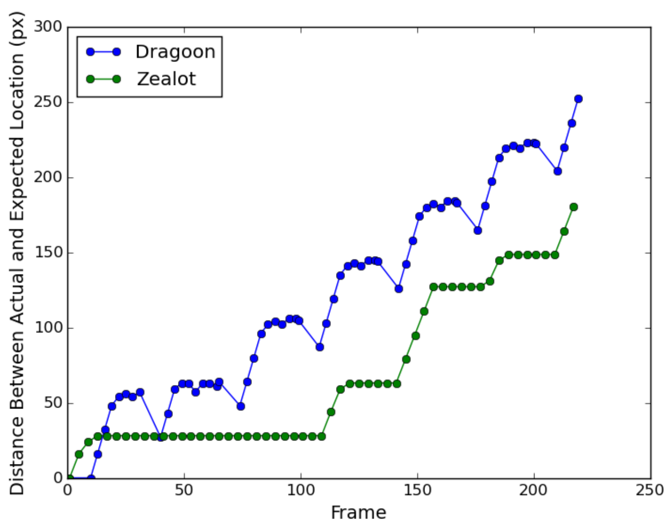
After improving UAlbertaBot's model of StarCraft combat, we analyze the actions that UAlbertaBot is able to take for each unit in combat. In specific we present a new motion planning strategy for units that yields an increase in the win rate in UAlbertaBot's combat simulation.

StarCraft Unit Motion Analysis

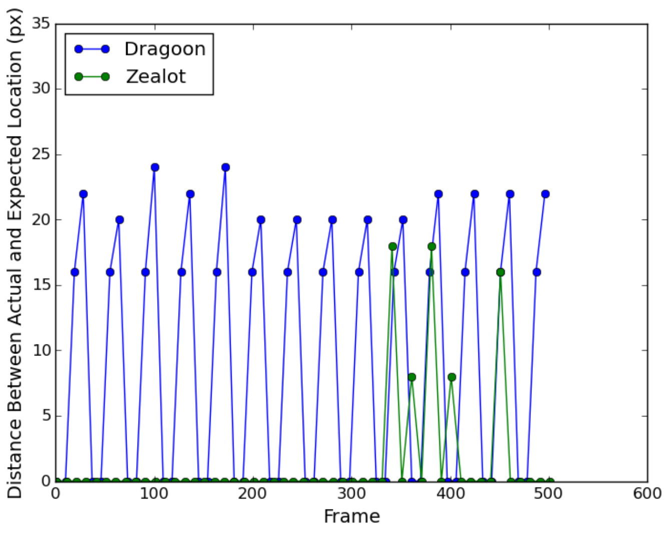
Combat in StarCraft is quite complex. Unit acceleration, deceleration, and delay caused by attack animation frames all add to the complexity of predicting combat outcomes. UAlbertaBot relies on the use of a simulation to perform combat. The simulation has two important roles, first it reduces the complexity of the game world, and second it provides a reliable and reproduceable way to predict the result of each action. UAlbertaBot relies on a simulation called SparCraft (Churchill 2015) to evaluate the current combat situation. SparCraft is a standalone module that given an input set of units and positions, is able to simulate combat, and predict the result. SparCraft is a model of StarCraft combat which is not entirely accurate. For example, SparCraft ignores unit collisions in order to speed up its simulation. This is required because AI systems for RTS game combat must act in real time. Other than this, SparCraft is a fairly accurate simulation, and most discrepancies between SparCraft and StarCraft are due to lack of access of the exact model the StarCraft game engine uses for combat.

Initial Experiment

As part of our work we wanted to measure any remaining discrepancies between SparCraft's model and StarCraft, and try to fix them. Of interest to us was the movement of units. In order to measure discrepancy, a combat simulation between a Dragoon (a ranged Protoss ground unit) and a Zealot (a Protoss melee unit) was run. Using the positions from a combat simulation allowed us to measure the error in a realistic scenario by gathering error from many sources such as attacking, stopping, and moving. The Dragoon and Zealot



A) not accounting for initial move delays



B) accounting for initial move delays

Figure 1: These graphs show the distance in pixels between the actual and expected positions of a Dragoon and Zealot in a combat simulated by SparCraft when played out in StarCraft before and after accounting for initial move delays.

were positioned on a horizontal line 143 pixels apart. This kept the Dragoon starting out of range of the Zealot forcing it to start the combat by moving. The actions generated by SparCraft at each frame were run inside of StarCraft on the corresponding frame. The combat action sequence produced by SparCraft consists of the Dragoon attacking the Zealot: retreating while its attack recharges, then moving into attack range, attacking, and repeating this pattern. This behaviour is called "kiting". The Zealot, on the other hand, charges steadily towards the Dragoon until it is in range. It then attacks and continues charging. This results in the Zealot taking a while to get its first attack. Then once the Zealot is closer, it gets an attack against the Dragoon each time it stops to attack. During combat the actual positions of each of the two units was gathered at each frame and compared to SparCraft's expected positions. The resulting graph shown in Fig. 1 A) demonstrates periodic spikes in the error of the units' positions, this is particularly obvious for the Zealot. When examining the combat and the frames that the spikes occur at, it was found that error is introduced each time a unit is ordered to move. This is easily identifiable when the Zealot is first ordered to move at frame zero. Each time one of the units stops to attack and starts moving again there is an increase in the error. Because the positions of the units diverge rapidly from the expected positions as time goes on, SparCraft may find what it thinks is a winning strategy is not a winning strategy in the actual StarCraft game. This inconsistency results in units getting attacked when SparCraft didn't think a unit could get attacked, or units not making it into range to attack when SparCraft thought they would.

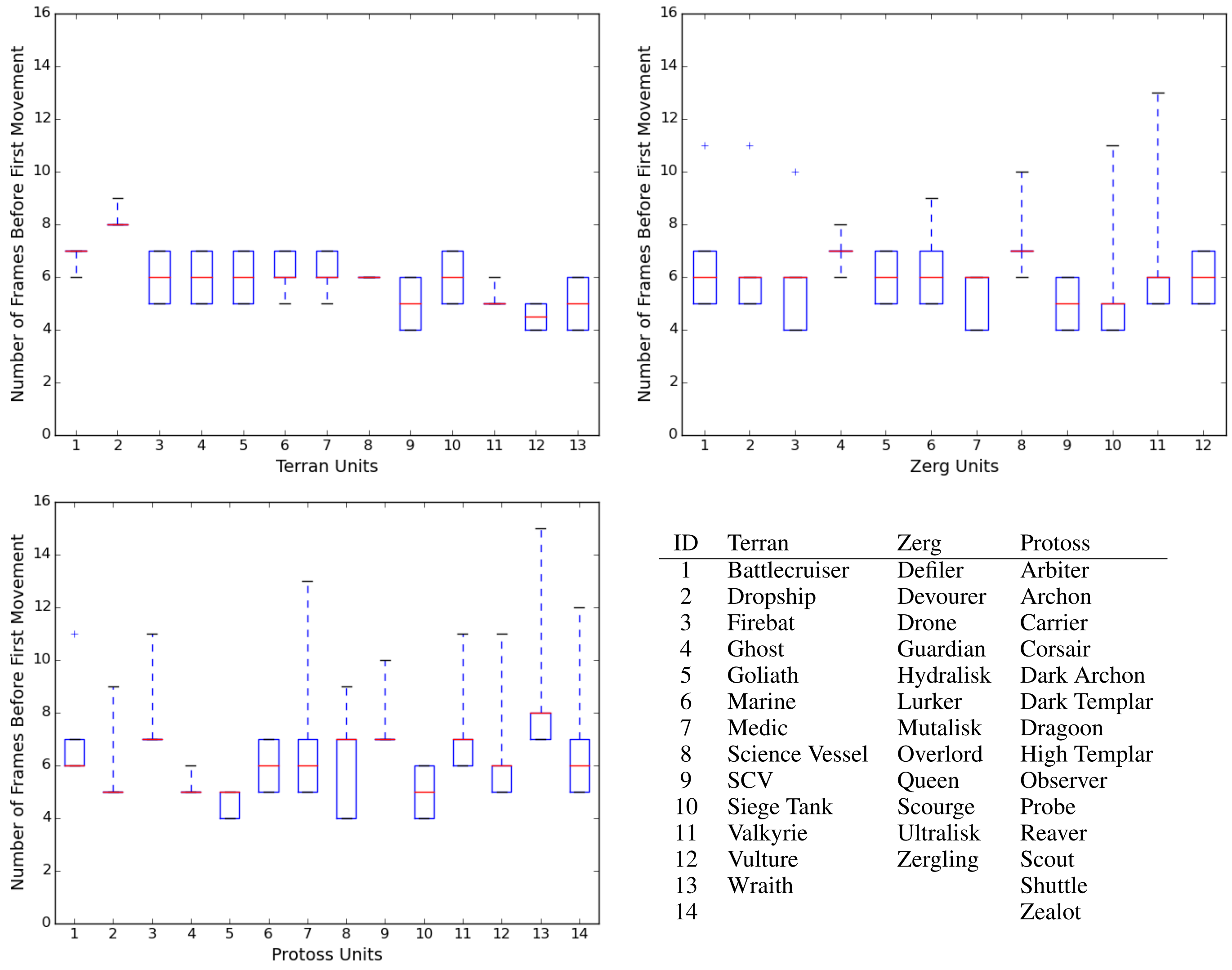
Measuring Initial Motion Delays

To fix this discrepancy we first needed to measure the initial command delay. For this, different units were marched back and forth in StarCraft. The number of frames between ordering a unit to move and the unit's position changing were measured. To prevent unit rotation from affecting the delay measurement, units were moved a small distance and then held still for a small amount of time. This was done in order to get the units facing the correct direction and to make sure they'd come to a complete stop. This experiment was run for each unit type, and 50 samples were gathered for each. The measurement statistics are shown in Fig. 2.

These graphs depict box plots of the delay between a unit being ordered to move and the position of the unit first changing. Based on the three graphs the average delay across all unit types appears to be $\approx$ 6 frames. We didn't have enough time to systematically investigate why there was variation in the delay for individual unit types. One explanation may be that the experiments were run in bulk, and this may have resulted in some units having a chance to rotate after stopping before moving again. Coming up with more accurate models of the unit delay is future work.

Without StarCraft's source code, we can't be certain of the cause of the systematic 6 frame delay, but one hypothesis is that this delay is used to prevent units from moving until the cursor click animation has completed. Watching the cursor animation in slow motion reveals it to last about the correct number of frames. However, we were not able to confirmed that this is the actual cause of the delay. It is also important to note, that the library UAlbertaBot uses adds some delay frames to account for latency, but because the experiments were gathered on a single player map, the latency added does not account for the entire delay witnessed.

After adding this fixed 6 frame delay to the SparCraft model a new combat simulation was run inside of StarCraft. The obtained results are shown in Fig 1 B). The error between the actual and expected positions of the Dragoon and Zealot have been drastically reduced. There are still error spikes that occur whenever one of the units goes to attack.



ID	Terran	Zerg	Protoss
1	Battlecruiser	Defiler	Arbiter
2	Dropship	Devourer	Archon
3	Firebat	Drone	Carrier
4	Ghost	Guardian	Corsair
5	Goliath	Hydralisk	Dark Archon
6	Marine	Lurker	Dark Templar
7	Medic	Mutalisk	Dragoon
8	Science Vessel	Overlord	High Templar
9	SCV	Queen	Observer
10	Siege Tank	Scourge	Probe
11	Valkyrie	Ultralisk	Reaver
12	Vulture	Zergling	Scout
13	Wraith		Shuttle
14			Zealot

Figure 2: Box plots for the delay between ordering a unit to move and the first change in its position measured in number of game frames. These graphs contain data for all Terran, Zerg, and Protoss units.

This demonstrates that more tuning needs to be done to properly account for attack delay. We hypothesize that the error for each attack doesn't accumulate because all or most of the delay from attacking is accounted for, it just isn't accounted for on the right frames. The error in position rises when the Dragoon goes to attack, because SparCraft thinks that the Dragoon should still be moving. Then the error begins to drop, because the Dragoon is moving and SparCraft thinks the Dragoon should be standing still. Thus at the end of each attack the error is reduced to, or near zero. We will leave investigations into this matter to future work. Overall, adding the movement delay to the model drastically increased its accuracy, and thus allows SparCraft to provide more accurate long term battle plans to UAlbertaBot.

Modelling Unit Acceleration

There's one more significant motion detail that isn't modelled by SparCraft yet: unit acceleration. This isn't apparent for the Dragoon's or Zealot's position error because neither of those two units accelerate. When units move in SparCraft they move at a constant speed and no acceleration time is taken into account. Because in combat units often start and stop moving, the prediction error can become large very quickly. To account for acceleration, rather than simulate it, we simplified the problem by finding the best frame to use as the frame where a unit's top speed is achieved. The best frame is the frame that will result in the lowest overall Mean Squared Error(MSE) between the actual and predicted position of the unit. This allows us to move the unit forward a distance by that frame, and then move the unit forward at a constant speed after that point. By removing the assumption of zero acceleration, units with acceleration will have more accurate positions in the model.

To find the best frame to use for hitting top speed, a linear regression was done using each possible frame between when the unit starts moving and frame 150, at which all units were at their top speed. After that point, the predicted values fall on the line fixed at the average distance travelled at the top speed, and the average distance travelled at frame 150. The MSE is then calculated based on the predicted and actual values. The top speed frame with the lowest overall MSE then is the frame that we use for when the unit reaches top speed. The samples for this data were gathered by moving units in a straight horizontal line for 1000 pixels. All units were moved from a stand still while facing the correct direction of movement. Fifty samples were gathered for each unit.

Assuming the StarCraft game engine implements simple kinematics laws using constant acceleration, we have $d = d_0 + v_0 t + \frac{1}{2}at^2$. Because we are moving the units from a stop, $d_0 = 0$ and $v_0 = 0$. So, $d = \frac{1}{2}at^2$. Now we can replace $\frac{1}{2}a$ with c to get $d = ct^2$. This is what we'll perform the linear regression on. By using the sample data we can find the best value for c and thus a. In order to find the best frame to use as the top speed frame, we try every possible frame, and perform the linear regression using data up to that frame. The frame that results in the lowest total MSE we use as the frame where top speed is achieved. Minimizing the MSE w.r.t. one variable is a straight-forward calculation: If we take the derivative of the error $\epsilon = \sum_{i=1}^{n}(ct_i^2 - d_i)^2$, then we get $\frac{d\epsilon}{dc} = c\sum_{i=1}^{n} t_i^4 - \sum_{i=1}^{n} d_i t_i^2 = 0$ since we want the c which results in the minimum error. Finally, $c\sum_{i=1}^{n} t_i^4 = \sum_{i=1}^{n} d_i t_i^2$ and $c = \sum_{i=1}^{n} d_i t_i^2 / \sum_{i=1}^{n} t_i^4$.

Having computed the top-speed frame that minimizes the overall MSE, we can then break the distance travelled by a unit into two stages. The stage involving acceleration and the stage involving constant velocity. By factoring this into SparCraft's model, we can obtain an even more accurate model of unit movement.

Fig. 3 depicts the result of running this experiment for a single unit, the Protoss arbiter. The blue line represents the average position of the unit over 50 samples. The green line represents the predicted position when using the indicated frame as the top speed. The top speed frame is indicated by a light blue vertical line. This line has two numbers beside it. The first is the frame in which top speed is attained. And the second is the number of frames before any movement was achieved. This is the delay from above. Thus subtracting the second number from the first yields the actual number of frames that acceleration occurred over. Note that the MSE measured between the predicted and actual distance travelled is multiplied by 10 to make the MSE more visible. The MSE has a max value of around seven. This means that during the whole movement our predicted distance travelled is off by just a few pixels.

In future work, this acceleration model can easily be added to SparCraft's unit movement prediction after the top-speed frame and the according c value have been estimated for each unit type.

Improving Motion Search

After improving SparCraft's motion model of StarCraft we were interested in improving unit motion planning, i.e., the process of coming up with each unit's physical movements during a battle.

SparCraft simulates combat by assuming that each player takes turns making moves. In each turn a player is able to move each unit they control and attack any unit in range of each of their units that are able to attack. One common restriction is that a unit that is currently attacking may neither move nor attack until it is done attacking (with the exception of Tanks in StarCraft which can shoot while moving). Representing the game in this way simplifies the problem and makes it easy to apply existing search algorithms, such as Alpha-Beta search or Monte-Carlo Tree search, because it is turn based.

Currently, the simulation relies on a few different algorithms for approximating the best actions to take. The first algorithm, Alpha-Beta search, assumes that each turn we take the best move for us, and our opponent takes the worst move for us. If the model of the world is good and the domain isn't that complex, Alpha-Beta search is often able to come up with a good move for us to take. Each node in the Alpha-Beta search is given a score based on a playout based evaluation. The playout runs the quick NOKDPS combat script (see below) for both sides on that node and uses a score based off of the result.

The second algorithm is Portfolio Greedy search (PGS, (Churchill and Buro 2013)). PGS uses a portfolio of scripts (or game play types), and chooses a script for each unit to use. The selection is then scored using a playout against an enemy whose units have been assigned scripts in the same manner. The players scripts are then improved by searching over each script in the portfolio exactly once for each unit. The same is done for the enemy, and the assignment of scripts is scored again using playout. This is done for

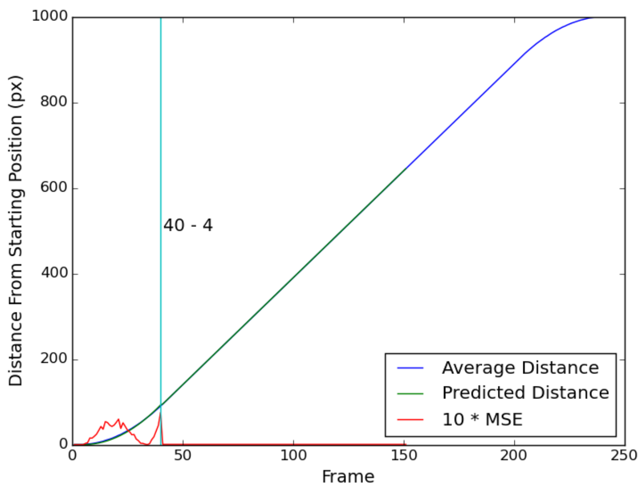


Figure 3: A graph depicting actual and predicted positions as well as prediction errors and the best frame to use for the top speed of the Protoss arbiter.

a number of iterations. In our experiments, the player uses the NOKDPS and KiterDPS scripts, and the enemy uses the NOKDPS script.

The final strategy we focus on in this paper is called "No-Overkill Damage Per Second" (NOKDPS, (Churchill and Buro 2013)). NOKDPS is a scripted strategy that doesn't involve search. It attempts to deal the most damage on each turn without causing any more damage to each enemy unit than is required to take its health down to zero — or as little overkill as possible because getting to exactly 0 isn't always possible. NOKDPS moves units to be the closest to an enemy. Thus, given a set of any number of movements it will choose the movement that takes the unit closest to an enemy unit. The KiterDPS script we mentioned above behaves by preferring moves that either move the unit closer to the closest enemy unit (in the case of attacking), or maximizes the distance to the nearest enemy unit (in the case of retreating) (Churchill and Buro 2013). When attacking, the KiterDPS script attacks the enemy with the highest ratio of damage per frame, to remaining health. Damage per frame is calculated by taking the total damage per attack and dividing it by the number of cooldown frames required after performing an attack.

When SparCraft uses above algorithms to compute a combat strategy, it moves units in the limited set of cardinal directions (North, East, South, and West). This provides each unit with a versatile set of movements, while keeping the number of possible movements low and search trees small. As a result, in the limited amount of time SparCraft has to compute a strategy, more of the search space can be explored which often yields better search results.

When looking to improve the small-scale unit combat of UAlbertaBot we wondered what limited set of movements can be used to achieve better performance. Here, performance is the measure of the percentage of wins a player using a new search space has over a player using the old search space inside of the SparCraft simulation. The current set of movements chosen are the four cardinal directions. Is there a set of four movements that performs better?

A couple of different sets of movements were considered. The first was to choose the two closest enemies and move in each of their directions, and choose the two closest allies and move towards each of them (Fig. 4). In the case that there are fewer than two enemies or fewer than two allies, the cardinal directions are added to supplement the current movements, and keep the branching factor at four. The idea behind these movement choices was that our units will attack, and when it's better to retreat, will regroup towards each other.

The second option was to plan movements based on the closest enemy (Fig. 5). Here, we consider moving towards the closest enemy and away from it, and the two movements orthogonal to the previous directions. The idea behind this selection of moves was to provide movement to the closest target, as well as three potential retreat plans. In addition, the movements are equally spaced which allows for a good motion spread.

To compare the two motion sets, experiments were run comparing each set to the original cardinal direction set. The experiments consisted of running a simulation of two

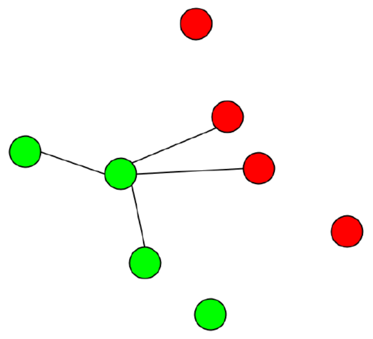


Figure 4: Closest enemy/ally movement directions.

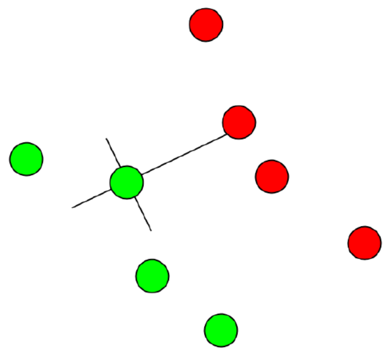


Figure 5: Closest enemy based perpendicular movement directions ("Closest-$\times$" method).

teams. One team using one of the proposed motion sets, and one team using the cardinal direction motion set. Each team consisted of four Dragoons. The units were laid out using SparCraft's SeparatedState placement. This places the two teams symmetrically across the diagonal, and separated so that they are not overlapping. This was done to ensure that the full benefit of the new motion sets were measured. Otherwise, units might be placed so that no motion was required. Each experiment consisted of nine configurations with 20 executions each. The configurations were the pairwise matching of the Alpha-Beta, PGS, and NOKDPS algorithms for both the proposed motion sets and the cardinal direction motion set. The PGS algorithm used NOKDPS and a melee script as portfolios for the Zealots, and NOKDPS and a kiter script as portfolios for the Dragoons.

Both motion sets were tried to see which had a higher win ratio. The first set performed worse than the simple cardinal movements. The movement of units seemed to be very limited resulting in a poor search space coverage. After some trials it occurred to us that even though four movement choices were available, as the battle progressed essentially only two choices would exist. The problem is that our units gather together in clumps. The result is that the two movements to our closest allies result in movements that lead to similar location. In addition, the two closest enemies are often very close to each other. The result is that we often only consider two movements: the first is towards our enemies and the second is to stay still. Because of the clumping of units this movement set is very limited.

The second movement set, however, showed an improvement over the simple cardinal direction scheme and performed quite well. This set seems to perform better because the motion directions are evenly spaced. In addition, NOKDPS works best when it is moving directly towards the closest target. So, giving it the option to move towards the closest target gives it the best option for charging, and leaves three potential retreat directions.

Now that the second motion set has been shown to be more effective than the first, we ran a larger set of experiments to measure its performance across a larger range of tests. For easy reference, the second motion set has been named the "Closest-×" motion set. The larger set of experiments consists of the same setup as the previous experiments, but variations of Dragoons and Zealots are used. The first experiment runs with each team containing four Dragoons as before. The second experiment consists of each team containing two Dragoons, and two Zealots. The final experiment consists of each team containing four Zealots. In each experiment, the total number of units was kept to eight to prevent the branching factor from reducing the Alpha-Beta search depth and thus decreasing performance. During combat, Alpha-Beta was usually able to make it to a search depth of four.

Table 1 contains the results of experiments comparing Closest-× motion planning to Cardinal direction motion planning for three team settings. As can be seen, Closest-× motion planning increases the win ratio of both Alpha-Beta search and the NOKDPS script when compared to the simple cardinal direction motion planning in all three experiments.

Table 1: The win ratio of Closest-× motion planning against cardinal direction motion planning for Alpha-Beta, PGS, and NOKDPS and three different unit configurations.

4 Dragoons each

	Cardinal Direction		
Closest-×	Alpha-Beta	PGS	NOKDPS
Alpha-Beta	0.65	0.35	1.00
PGS	0.15	0.50	0.95
NOKDPS	0.10	0.30	0.90

2 Dragoons and 2 Zealots each

	Cardinal Direction		
Closest-×	Alpha-Beta	PGS	NOKDPS
Alpha-Beta	0.65	0.05	1.00
PGS	0.25	0.48	1.00
NOKDPS	0.15	0.25	0.70

4 Zealots each

	Cardinal Direction		
Closest-×	Alpha-Beta	PGS	NOKDPS
Alpha-Beta	0.63	0.00	1.00
PGS	0.40	0.45	0.95
NOKDPS	0.45	0.45	0.90

Table 2: The win ratio of Closest-× motion planning against cardinal direction motion planning for PGS and NOKDPS using 6 bigger unit configurations.

8 Dragoons each

	Cardinal Direction	
Closest-×	PGS	NOKDPS
PGS	0.30	1.00
NOKDPS	0.30	1.00

16 Dragoons each

	Cardinal Direction	
Closest-×	PGS	NOKDPS
PGS	0.05	1.00
NOKDPS	0.25	0.90

4 Dragoons and 4 Zealots each

	Cardinal Direction	
Closest-×	PGS	NOKDPS
PGS	0.15	1.00
NOKDPS	0.10	0.90

8 Dragoons and 8 Zealots each

	Cardinal Direction	
Closest-×	PGS	NOKDPS
PGS	0.05	1.00
NOKDPS	0.05	0.90

8 Zealots each

	Cardinal Direction	
Closest-×	PGS	NOKDPS
PGS	0.30	1.00
NOKDPS	0.35	1.00

16 Zealots each

	Cardinal Direction	
Closest-×	PGS	NOKDPS
PGS	0.30	1.00
NOKDPS	0.30	0.90

The win ratio for PGS tells a different story. The win rate either stays the same or decreases. The likely reason for this is that PGS uses the NOKDPS script and the KiterDPS script to generate its actions. Thus it's possible that the KiterDPS script does not perform as well with the new motion plan as the NOKDPS script does and the performance decreases overall. But when using the NOKDPS script or Alpha-Beta search, the new motion plan provides a set of movements that increases the win ratio in all three experiments.

In order to test the performance of Closest-× motion planning when more units are involved, tests with similar unit breakdowns as the previous tests were run with eight and sixteen units per team. Table 2 contains results from larger tests, Alpha-Beta is not included in the table for the reason mentioned above.

In these cases the Closest-× motion planning continues to beat cardinal direction motion planning when using the NOKDPS script. The win rate is between 90% and 100%

in all of the tests. This makes sense because as mentioned above NOKPDS prefers to move toward the closest target, thus the optimal move is provided.

In the scaled up experiments Closest-$\times$ PGS continues to perform poorly against the cardinal direction PGS with a win rate ranging between 5% and 30%. This implies that scaling up the number of units does not improve the performance of the KiterDPS script when using Closest-$\times$ motion planning.

Conclusions and Future Work

In this paper we addressed two motion related problems in RTS games: accurate unit position prediction and motion planning based on small sets of discrete motion options.

Our work has demonstrated a previously unaccounted delay in movement when ordering a unit to move in StarCraft, as shown with before and after graphs of the unit positions error. Accounting for the delay and also accounting for unit acceleration we were able to remove most of the unit position error that remains in our motion prediction model.

Our experiments with unit motion planning showed that without increasing the latency of decisions we are able to come up with a new motion direction set for units considered during look-ahead search that increases low level combat performance. This result demonstrates the potential that changing motion planning for units can have on combat. It raises the question of what effects other new direction sets may have on the performance of low level combat of StarCraft bots.

We do have some thoughts on alternative direction sets that may increase performance further. One plan is to identify move directions based on influence maps which aggregate unit positions and firepower to identify high-valued targets or threats nearby. In addition to our work on motion planning, more future work is to come up with more precise measurements of the delay when ordering a unit to move and it beginning to move. For this it would be helpful to identify whether unit heading or other attributes influence the delay. Lastly, to make our model even more accurate, we'd also like to explore the spikes in the error of the unit position that remained in the model even after the movement delay was accounted for. This would involve examining what possible delay may not be accounted for when a unit goes to attack.

References

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8. IEEE.

Churchill, D. 2013. AIIDE StarCraft AI competition. http://www.starcraftaicompetition.com/.

Churchill, D. 2015. SparCraft: open source StarCraft combat simulation. github.com/davechurchill/ualbertabot/tree/master/SparCraft.

Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *TCIAIG* 5(4):293–311.