1. Introduction

Purpose
The purpose of the project is to apply relevant AI techniques learned during the course with a view to develop an intelligent game playing bot for the game of ‘Planet Wars’. Planet Wars is part of the Google AI Challenge 2010 and is based on the original game ‘Galcon’. The main objective of the project is to create a game playing bot that competes with the other computer bots. There are no human players involved.

Problem Scope / Description
A game of Planet Wars takes place on a map which contains several planets, each of which has some number of ships on it. Each planet contains a different number of ships during the start of the game. The owner of the planet can be either one of the players or it can be neutral. A neutral owner means the planet has not yet been captured. The game has a certain maximum number of turns, so that the game does not continue indefinitely. Provided that neither player performs an invalid action, the player with the most ships at the end of the game wins. The other way to win the game is to take control of all the enemy planets i.e. by defeating /destroying all the ships of the enemy planet. In this case a win is immediately declared to the player that has ships left in the game. If both players have the same number of ships when the game ends, it is declared as a draw match.

In every turn, a player can decide to send ships to any other planet in the map. The planet could be a neutral or enemy planet or could be the player’s planet itself. The player can send any number of ships to the destination planet, as long as the number of ships sent is less than or equal to the number of ships available in the planet at that time. In every turn, the number of ships in the planet will increase. The increase in the number of planets is defined by the growth rate. Different planets have varying growth rates.

The map is designed in the form of a Cartesian co-ordinate system. Each planet has an X and Y co-ordinate. The distance between any two planets can be calculated using the distance formula- \( \sqrt{(dx - sx)^2 + (dy - sy)^2} \). The distance decides the number of turns that the fleet of ships sent will take to reach the destination planet. The order for the ships once issued cannot be reversed and the destination cannot be changed.

Planet Wars Specification

Planets
The planets are described in the map with 5 attributes:

a. The X position of the planet.
b. The Y position of the planet.c. The number of ships at the beginning of the game.
d. The growth rate in the number of ship in the planet.e. The owner of the planet.

The planet is a stationary object and its position does not change during the game. The owner can be neutral, player 1 or player 2. The ID’s are 0, 1 or 2. The planet ID is also given to identify a specific planet.

Fleets
Each fleet is described by the following details:

a. The owner of the fleet.
b. The number of ships in the fleet.c. The source planet from where the fleet has been sent.d. The destination planet of the fleet.e. The distance between the source and destination planet.f. The number of turns remaining, i.e., the number of turns in which the fleet will reach its destination.

The game engine, during each turn, sends the game state to each player. This state is available in the form of a standard input statement (stdin). Once the players compute their next moves, the game engine receives these orders and updates the game state. During updating of the game state, it checks for the end game conditions.
**Game Updates**

The game state is updated in the following conditions:

a. When a fleet order is issued:
   1. The number of ships sent is subtracted from the source planet.
   2. The number of turns remaining is decremented by 1.
   3. Ships are added to a planet, according to the growth rate.

b. When the fleet arrives:
   1. The number of ships in the destination planet is subtracted.
   2. If the number is less than 0, it means the attack was successful and the subtracted value is set as the new number of ships in the planet. The ownership of the planet is also changed.

**End Game Conditions**

Currently the game is allowed to be played till one of the player wins.

**Software & Hardware Requirements**

a. Platform: Ubuntu Linux 10.04.1
b. Language: CPP
c. Compiler: g++

**2. Approach**

We have applied two different AI techniques and have presented the performance of each. We have implemented the first bot using a game tree. The bot can generate an “n-ply” tree. Game tree uses the **Minimax Search** algorithm. To improve the performance of the bot (since it has to search a very large space), we have also implemented **Alpha-Beta Pruning**. The improvement in performance due to ‘Alpha-Beta Pruning’ is significant as can be seen in the analysis section. Both bots have been implemented in C++. This is because the game engine has a timeout period for each turn. The amount of time in which a bot should issue an order is 1 second. With a strict time constraint, we were unable to use interpreted languages such as python. The execution time of the recursive code in python was very large; the bot was able to generate a single ply of the game tree. Apart from generating the game tree, the bot had to update all the states and also calculate the cost of each game state (i.e. in the last ply) in the game tree. Given a large amount of computation to be done in a limited time frame, C++ was used for the bot implementation.

The second bot has been implemented using a learning algorithm called **Temporal Difference Learning**. Temporal Difference Learning has been used in games such as Backgammon [1]. The bot has been tested for different learning rates (different values of $\alpha$). The rewards are based on the cost function described below.

The cost of game state is considered, because the cost is a cumulative value of the entire game state, which has a value attached for the fleets generated in various turns while playing the game and it also evaluates the opponent’s fleets. The planets owned by the bot and those that are owned by the opponent are also evaluated.

The function to evaluate the cost of the game state has been used in both Minimax algorithm and with Temporal Difference Learning.

**Game State**

Each game state consists of the following attributes:

a. The number of planets in the game.
b. The planets owned by the player and the adversary.
c. The growth rate for each planet.
d. The positions of each planet on the map.
e. The number of ships sent to each planet, i.e., the fleets that are in transit from one planet to another.
f. The number of ships under the player’s control.
g. The number of turns remaining. This is required to keep a check on the number turns used and number of turns left as the there is a limit on the maximum number of turns by each player.

The positions of the planets can be used to compute the distances between the planets to decide the number of turns required for the ships to reach the destination planet from the source planet.
**Minimax Game Tree Implementation**

Steps: (The steps described below are taken for each turn).

a. Expand the current game state and construct an n-ply game tree.

b. Calculate the cost of each game state at which the leaf node and cost the operations till the leaf node.

c. Back up the value and choose the operation to be conducted according to the Minimax Algorithm.

**Temporal Difference Learning Implementation**

a. Expand the current game state and list all the possible operations.

b. Calculate the Value of state and action:

\[ Q(a,s) = Q(a,s) + \alpha(r + \gamma maxa'(Q(a',s')) - Q(a,s)) \]

c. Choose the action with the highest value.

In case of Planet Wars, the game state contains information on all the planets and the fleets which are travelling in the game during the current turn. Since the amount information in the state is very large, the number of unique states is very high. The bot creates a lot of new states and is unable to update the value of a given state during the next round. The size of the knowledge base becomes very large. Hence searching the entire knowledge base for a specific state – action pair is very time consuming. The bot times out at the start of the game, because loading the knowledge base is very time consuming. Hence the state-action pair in the knowledge base contains the following information:

a. Source Planet.

b. Destination Planet of the fleet.

c. The turn when the fleet was sent.

d. The value of the action i.e. \( Q(a,s) \).

A single map in the game contains about 20 planets. If we create the state with only the source and the destination planet, the total number of states is \( P_2 \). The total number of states maintained would be 380. Also, attacking a destination planet is not only dependent on the source planet, but also on the turn when it can be done (considering where the opponent fleets and the fleets owned by the bot are). Hence the turn has also been added to the state, to make the knowledge base of states more detailed as well as to keep it small enough so that the bot can process it. The values for the all the states-action pairs are maintained in a file "mylearning<turn number>.txt". All the values in the file are sorted and maintained in the ascending order of the value of their turns.

The bot loads the entire file at the start of each turn. The file loading activity could have been reduced by reading and writing to the file only at the start and the end of the game (rather than at each turn), but the end of the game is unknown. This causes the game to stop abruptly and contents of the states generated and updated during the game may not be written at the end of the game.

Since the file has the potential to become big, the bot may timeout while reading the file during a turn. Hence to enhance the performance, there is separate file created for each turn, which contains the starting location of all the states-action pairs having the same turn. This creates a large number of database files.

Example value (in the knowledge base): 1, 0, 1, 159. The first “1” denotes the source planet, the “0” denotes the destination planet, the second “1” denotes the turn when the action was played. 159 is the value of the state.

**Cost Function**

The following attributes have been considered to calculate the cost of the game state and cost of sending the fleet:

a. Distance between the source and the destination planet.

b. The number of ships in the destination planet.

c. The number of ships in the source planet.

d. Ownership of the destination planet.

e. Growth rate of ships in the destination planet.

f. Growth rate of ships in the source planet.

g. A set of planets could send ships to a single neutral/ enemy planet. This is called “Gang Up”. There is a weight attached to this attribute.

h. The number of turns remaining before the fleet reaches the destination planet is also considered.
The planets have a value (owner field) as 0, 1 or 2. 0 is for a neutral planet, 1 means the planet is owned by the player (in this case our bot) and 2 means the planet is owned by the opponent. By attaching a weight to this attribute, the value of the operation which attacks an enemy planet increases. Thus by making the weight a large value, we can make the bot more aggressive towards attacking an enemy planet.

The game state contains the fleets owned by the bot and the opponent’s fleets. Hence the cost of both players fleet can be calculated. Thus the same attributes are maintained for the opponent also. They have a different set of weights. The total number of attributes is 16. A total of 16 weights are defined.

The values of the weights for our bot are:
1. -1 for the distance between the source and the destination planet.
2. -1 for the number of ships in the destination planet.
3. 5 for the number of ships in the source planet.
4. 1 for the ownership of the destination planet.
5. 10 for the growth rate of ships in the destination planet.
6. 10 for the growth rate of ships in the source planet.
7. -1 for total number of turns remaining.
8. 1 for the cost of “Ganging up”.

The values of the weights for opponent are currently maintained the same. The reason to maintain them as different set of attributes is because in case a different strategy is applied by the opponent, then that strategy can be mimicked by the changing values of the attribute weights for the opponent.

**Heuristic Calculation**

Heuristic Function value is calculated as follows:

\[ H(x) = A_1 \times W_1 + A_2 \times W_2 + A_3 \times W_3 + A_4 \times W_4 + A_5 \times W_5 + A_6 \times W_6 + A_7 \times W_7 + A_8 \times W_8 - A_9 \times W_9 - A_{10} \times W_{10} - A_{11} \times W_{11} - A_{12} \times W_{12} - A_{13} \times W_{13} - A_{14} \times W_{14} - A_{15} \times W_{15} - A_{16} \times W_{16} \]

Where,
A1 – A8 are the attributes of our bot.
W1 – W8 are the weights attached to each attribute of our bot.
A9 – A16 are the attributes of the opponent.
W9 – W16 are the weights attached to each attribute of the opponent.

The values (mentioned above) for the weights have been calculated using trial and error method.

The Game Tree and Temporal Difference Learning method could not be merged together into a single bot as the processing time is high.

3. **Testing and Performance Analysis**

The bot has been pitted against a different set of bots available with the Google AI package. The different bots are: the DualBot, RageBot, ProspectorBot and the BullyBot.

**Bot Description**

**DualBot**

The DualBot is a bot which maintains a static number of fleets in transit during the course of the game. When the total number of fleets goes below a certain threshold, it issues an order for a new fleet to be created during the next turn. The source planet (from where the fleet starts) is always the strongest planet it owns and it attacks the weakest enemy or neutral planet. The strongest planet is calculated by a score function.

\[ score = \frac{Number \ of \ ships \ in \ planet}{1 + planet's \ growth \ rate} \]
**RageBot**
The RageBot attacks only the opponent’s planets and not the neutral planets. It first makes a list of all the planets it owns and selects only those planets which match the criteria:

\[
\text{Number of Ships in Planet} \geq 10 \times \text{Growth rate of the planet}
\]

Thereafter, for each of the planets matching the above description, it finds the opponent’s planet which is at the shortest distance from it. It issues an order for the planet to attack the opponent’s planet with a fleet containing all the ships in the source planet. It issues orders for each such combination it can find. Thus, multiple orders are issued in a single turn.

**ProspectorBot**
The ProspectorBot has the same function as the DualBot. The only difference is that the ProspectorBot is designed to have a single fleet in transit at any given time in the game.

**BullyBot**
The BullyBot is designed to attack the opponent’s strongest planet. It first finds its strongest planet based solely on the number of ships the planet has. It then attacks the opponent’s strongest planet with fleet containing half the ships in the planet it chose.

**RandomBot**
The RandomBot is designed to pick an opponent’s planet at random and play the game.

**Maps Description**
The maps defined in the results, are the files containing the information required by the game to create the initial system (essentially the map of the entire game world). It contains the list of all the planets in that game, their X & Y co-ordinates, the number of ships they contain initially, the owner of the planet and the growth rate of each planet. It also has information on which player owns which planet at the start of the game. Different maps have different number of planets and these planets are position different locations on the co-ordinate space.

**Game Tree & Minimax Algorithm Tests Results**

1. **Dual Bot Results**

![Number of Turns in The Battle (Dual Bot)](image-url)
2. **Random Bot Results**

![Node Generation for DualBot (2-Ply Game)](image1)

![Node Generation for DualBot (3-Ply Game)](image2)

![Number of Turns in The Battle (RandomBot)](image3)
3. **Bully Bot Results**

![Graph of Node Generation for RandomBot (2-Ply Game)](image1)

![Graph of Node Generation for RandomBot (3-Ply Game)](image2)

![Graph of Number of Turns in The Battle (Bully Bot)](image3)
4. **Prospector Bot Results**
Analysis

Our bot was able to win all the games, except 3 games from the list. All the nodes were generated under a strict timing constraint of 1 second. The number of nodes generated for each ply increases exponentially. The number of nodes generated in the first ply is around 25 – 100. The second ply increases exponentially to around 1000 or more. One observation seen is that, for all the games where a 3 PLY game tree was generated and the bot did not timeout, our bot won the game with lesser or equal number of moves, compared to number of turns taken when a 2 PLY game tree was generated. This is because, with a 3 PLY game tree, the bot has a better look ahead into the moves the opponent bot can do and it also calculates the operations it can do when the opponent has finished the move. Thus the heuristic generated is much more accurate and chosen operation is much better.

Although, the decrease in the number of turns is seen between a game using a 2 PLY and 3 PLY game trees, the difference is not very high. There are a number of factors:

a. The weights for the attributes are not be configured for a 3 PLY Game tree.
b. There is a distance between the source and the destination planet, which has to be covered by the fleet. This distance is shown in the form of the number of turns remaining. The minimum distance between the planets in most cases is more than 4-5 turns. This means that the fleet generated in the first ply, never reaches its destination planet, when the game tree is expanded at every Ply, as the minimum number of Ply’s that need to be generated at 4. Thus, the effect of the attack by a fleet cannot be measured in the game tree, limiting the effectiveness of the heuristic calculation.
The number of nodes generated has been reduced by applying some simple rules:

a. A planet which does not have a minimum number of ships is not allowed to be of the game tree because by using the planet which does not have sufficient ships to build a fleet, the size of the fleet becomes small. This also makes the planet vulnerable to an attack and can be easily taken over by the opponent. The cost function in any case, tries to maximize the final value and the attribute weight for the number of ships in the source planet is high. This makes a planet with a low number of ships, an unlikely option to be accepted at the Final Max level.

b. The number of nodes has also been reduced by adding an additional condition which checks if the growth rate of a planet is 0. A planet with a growth rate of zero does not produce any ships, when taken over. Hence owning the planet is useless. Such a planet cannot be a destination planet to send the fleet and hence can be eliminated during the game tree construction.

Another improvement implemented which increased the performance, was to reduce the number of times the game state is updated. Whenever a node is generated in a ply (for some fleet which is created), the child nodes are recursively created by calling the game-tree/Minimax function again. When the recursive call is done, the game state needs to be updated as planets would have generated new ships and all fleets would have moved by one position for the next turn. If we carefully check the update function, the update is same for the next turn, irrespective of which fleet (operation) is created. Hence we update the state, in the previous level of the game tree and add only the fleet information when the child nodes are produced.

**Temporal Difference Learning Tests**

*Tests with the Rage Bot*
Analysis
The Temporal Difference Learning Bot has been tested with the RageBot. As the graph shows, initially the number of turns taken by the bot is 140. As the bot is able to learn the moves of the opponent, the number of turns reduces during every attempt or in a new game. This is because the bot refines its steps for every new game. The spikes in the graph are because when the bot travels a new path, there is a possibility that in the end, the new path might take more turns that the previous game.

Another reason for the spike is because using the Cost Function as a reward scheme may not give a proper reward value at times, even though the move may be correct. This is because the cost of the game state is a combination of all the fleets in the system and hence the reward value also has not only the effect of the action taken by the bot, but also the effect of the opponent’s moves.

We tried, executing the bot on a different map, and were able to see a set of losses and then the bot starts winning again. This is because the initial values of the state may be proper for the bot, but as the turns increase, the values of the states are not correct. An improvement here is to program the bot to see the changes between maps. Currently it recognizes the planet only on the basis of the planet ID. This means that, it rates the planet 1 in map 1 and planet 1 in map 2 as the same. But since their positions and fleet sizes are different, the bot should be able to change the value of the states to compensate for the change in the planet.

Once the bot won the games with the RageBot, we tested the bot against the DualBot. It lost most of the matches. This shows that the learning algorithm implemented now, is not able to adapt to a new strategy immediately. It takes time and more games to change its strategy for the new player.

The interesting observation here was that the bot able to learn an important action i.e. “Do Nothing”. In certain turns it did not send a fleet. This is because the bot generates combinations not only between its planets and the opponents/neural planets, but also combines combinations with its own planets. This helps the bot to evaluate if sending a fleet to one of its own planets is a good move right now to reinforce it against a possible attack. It also generates the combination of sending a fleet to its own planet i.e. it sends a fleet to a planet where the source and destination planet are the same. If the value of doing this is better than other actions, it submits a “Do Nothing” operation.

The tests were conducted with different values of $\alpha$. When $\alpha$ is 0.9, the bot is able to adapt quickly to the new strategy and reach a stable or optimal state. In case of the first graph, the stable numbers of turns at the end of the tests are 51. In case of $\alpha$ with a 0.1 value, the immediate reward tends to increase the number of turns in the intermediate tests, before it can reduce any further.
4. Conclusion
Although the learning bot was able to win, the game tree performs much better, if the attribute list is tweaked better. Because of the use of the cost function as a reward, the learning bot tends to be unpredictable in certain matches. The number of turns taken to win the game dramatically increases when the map or opponent is changed.

A better application of the two approaches is to use them together. The game tree can generate a set of states and the Temporal Difference learning can be used to provide a look-ahead on the value of the possible actions. This could be done if the timing constraint does not exist.

5. Future Work
There are a number of improvements that can be made:

a. The game bot can be made to play multiple moves at the same time.
b. The cost of having multiple planets attack simultaneously is known, but only a single move can be applied. This can be changed with a better look-ahead function.
c. The game bot does not maintain a history of all the moves done during the game.
d. The calculation of a “Do Nothing” operation can be improved can for the game tree.
e. The rewards used in the learning bot can be improvised. One of the problems faced is that, there is no way to update the bot about who won or lost the game. This is because even if the bot wins or loses, the game engine ends the game abruptly, giving no chance for the bot to be updated with this information.

6. Sources and References


