UNIVERSIDADE FEDERAL DO ABC

Jonathan Ohara de Araujo

Agente inteligente para Angry Birds

Santo André

Maio de 2015

Sumário

1 Resumo

O objetivo desse trabalho é criar um algoritmo usando alguma técnica de Inteligência Artificial para jogar o jogo Angry Birds.

Não existe uma estratégia universal para achar a melhor jogada cada fase (cenário).

O agente desenvolvido utiliza de técnicas de grafos para escolha da melhor jogada, baseado em um treinamento executado anteriormente.

2 Introdução

2.1 Contextualização

O Angry Birds é um jogo do estilo quebra-cabeça desenvolvido pela Rovio Entertainment, sua primeira versão foi desenvolvida para IOS em dezembro de 2009. O algoritmo é feito para a versão para Google Chrome que foi lançada em maio de 2011. (Rovio, 2015).

Angry Birds AI Competition é uma competição mantida pela IJCAI (International Joint Conference on Artificial Intelligence) o primeiro campeonato aconteceu em 2012 em Sidney – Austrália. (Aibirds.org, 2014)

O programa para interagir com o jogo Angry Birds é chamado de Angry Birds Game Playing Software está atualmente na versão 1.32 e foi desenvolvido por XiaoYu Ge, Stephen Gould, Jochen Renz, Sahan Abeyasinghe, Jim Keys, Andrew Wang, Peng Zhang, (Aibirds.org, 2014).

2.2 Problema

Existe um número muito grande de componentes que influem no resultado do jogo, fazendo com que cada fase tenha sua peculiaridade. Muitas vezes uma estratégia de jogadas funciona muito bem em uma determinada fase, enquanto em outro cenário não é possível nem atingir o objetivo básico.

2.3 Objetivos

Criar um algoritmo utilizando alguma técnica de Inteligência Artificial para jogar o Angry Birds que consiga fazer mais pontos que o algoritmo de jogo padrão (Angry Birds Game Playing Software).

2.4 O jogo

O Angry Birds é um jogo com mecânica bem simples. O objetivo é acabar com os porquinhos do cenário. Para destruir esses porquinhos o jogador conta com um estilingue que atira pássaros.

Existem diversos tipos de obstáculos no mapa como: pedaços de madeira, gelo, paredes, colinas e etc. Esses objetos podem atrapalhar ou ajudar o jogador a atingir seu objetivo.

Além dos diferentes tipos de obstáculos existem diferentes tipos de pássaros onde cada um tem um poder especial. O pássaro vermelho é o comum ele não detém nenhum poder extra. Também existem alguns tipos de porcos, onde cada um tem resistência maior que outro.

Cada objeto destruído ou danificado no cenário dá mais pontos ao jogador. Ao destruir todos os porcos do cenário a fase é finalizada e, são contabilizados quantos passarinhos não foram utilizados, cada pássaro não utilizado rende pontos extras ao jogador.

3. Agente

3.1 Histórico

Antes de a ideia final ser desenvolvida alguns outros algoritmos foram desenvolvidos e falharam ou não apresentaram um desempenho satisfatório por alguma circunstância.

Esses algoritmos feitos anteriormente deram ideias para construção do algoritmo atual e para futuras melhorias e, além disso, desenvolvendo esse algoritmo algumas premissas foram descobertas.

3.1.1 Perceptron

A primeira implementação do algoritmo foi criar um Percetron, que leria pequenos padrões em cada fase. Ao achar um a padrão esse algoritmo tentava todas possíveis jogadas e guardava a melhor para posterior uso.

Angry Birds Game Playing Software além de disponibilizar um agente jogador básico, também disponibiliza uma biblioteca com diversos recursos que torna o jogo completamente observável, ou seja, através de algumas funções é possível ter uma visão completa do cenário do jogo.

O algoritmo buscava por porcos no cenário e a partir disso buscava objetos mais próximos horizontalmente e verticalmente, ou seja, eram localizados as coordenadas x e y do porquinho e era procurado o primeiro objeto acima, abaixo, do lado direito e esquerdo.

Com esses objetos em mãos eu criava um padrão para esses objetos a partir da posição x, y e com a altura e largura de cada objeto que era chamado de *cage* (gaiola).

Porém, logo nos primeiros testes, foi identificado que as pequenas partes do cenário não representavam todas as possibilidades de jogadas comparadas ao cenário todo.

Na imagem a seguir isso fica bem claro:

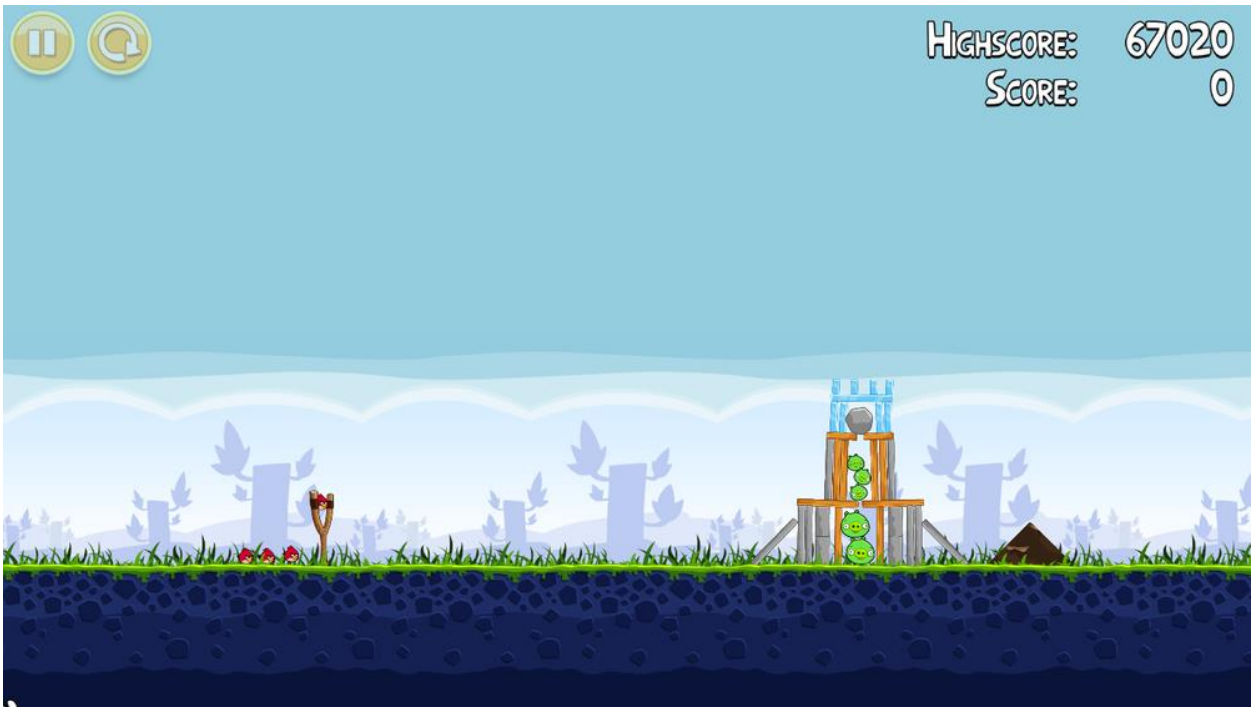

Figura 1: Fase cinco

Nesse cenário a heurística mais importante é de fazer a bola de pedra cair e destruir os porquinhos abaixo dela. Porém com o uso do perceptron ele dava mais valor aos objetos em volta do porco.

3.1.2 Aprendizado por reforço

Como cada configuração de cenário pode mudar totalmente a heurística de resolução, o algoritmo de aprendizado por reforço pareceu atender bem o problema.

O algoritmo funcionava da seguinte forma:

```
While state <> WON {
       state <- readState()
       state.possibleShots <- findPossibleShots( state )
       shot <- chooseOneShot( state.possibleShots )
       newstate <-doShot( shot )
       reinforcePastStates( state, newstate.points )
       state = newstate
}
```

Ele apresentava bons resultados em maioria das execuções, porém alguns cenários e situações ele falhava. Isso acontece porque, apesar de completamente observável o ambiente não é determinístico, ou seja, por motivos do jogo ou do algoritmo que realiza o lançamento dos pássaros, existem situações que dado um mesmo cenário e realizando a mesma ação o resultado obtido é diferente, em algumas fases essa diferença é tão drástica que a possibilidade não pode ser descartada.

Resumindo se dado um estado(estado inicial da fase por exemplo) se realizarmos um mesmo tiro N vezes podem podemos ter M resultados diferentes.

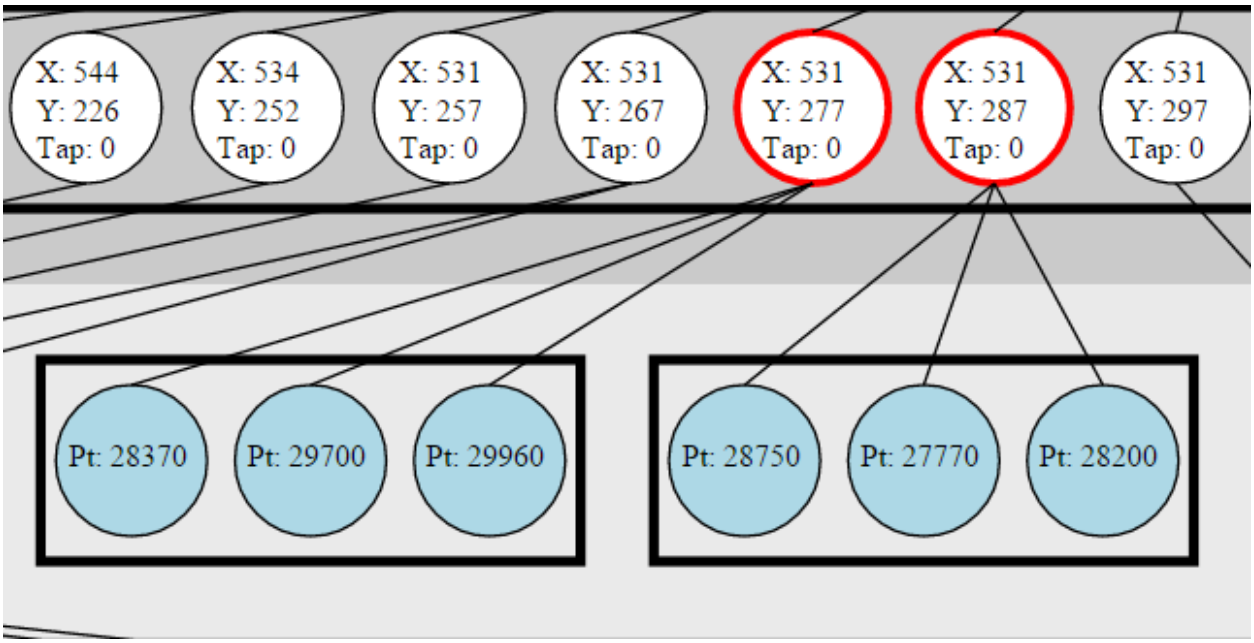

Figura 2: Trecho do grafo da Fase um

O pedaço do grafo da imagem acima foi criado baseado na primeira fase. Nesse trecho os nós destacados mostram que o mesmo tiro pode ter três possíveis resultados.

Essa característica estocástica acabou descartando a ideia de utilizar o algoritmo de aprendizado por reforço.

3.2 Algoritmo baseado em grafo

Apesar das ideias citadas acima serem descartadas, algumas ideias e alguns métodos dos algoritmos anteriores foram amplamente utilizadas, uma delas foi à utilização de grafos para descrever possíveis jogadas, estados e suas conexões.

O tipo de grafo usado foi o de árvore, ou seja, um grafo com nó raiz, acíclico e conexo.

Algumas características desse grafo precisam ser destacadas para melhor compreensão. São elas:

- Existem dois tipos de nós: tiros(*shot*) e estados(*state*);
- Todo nó raiz é um estado;
- Se o grafo estiver completo todo nó folha também é um estado;

- Todos nós filhos de um estado são nós de tiros;
- Todos nós filhos de um tiro são nós de estados;
- Todos nós em altura impar são estados;
- Todos nós em altura par são tiros;
- Cada nó pode ter N filhos.

A imagem a seguir mostra o grafo incompleto (o algoritmo de aprendizado ainda não percorreu todas possíveis jogadas, por isso, alguns nós folhas são tiros) da primeira fase.

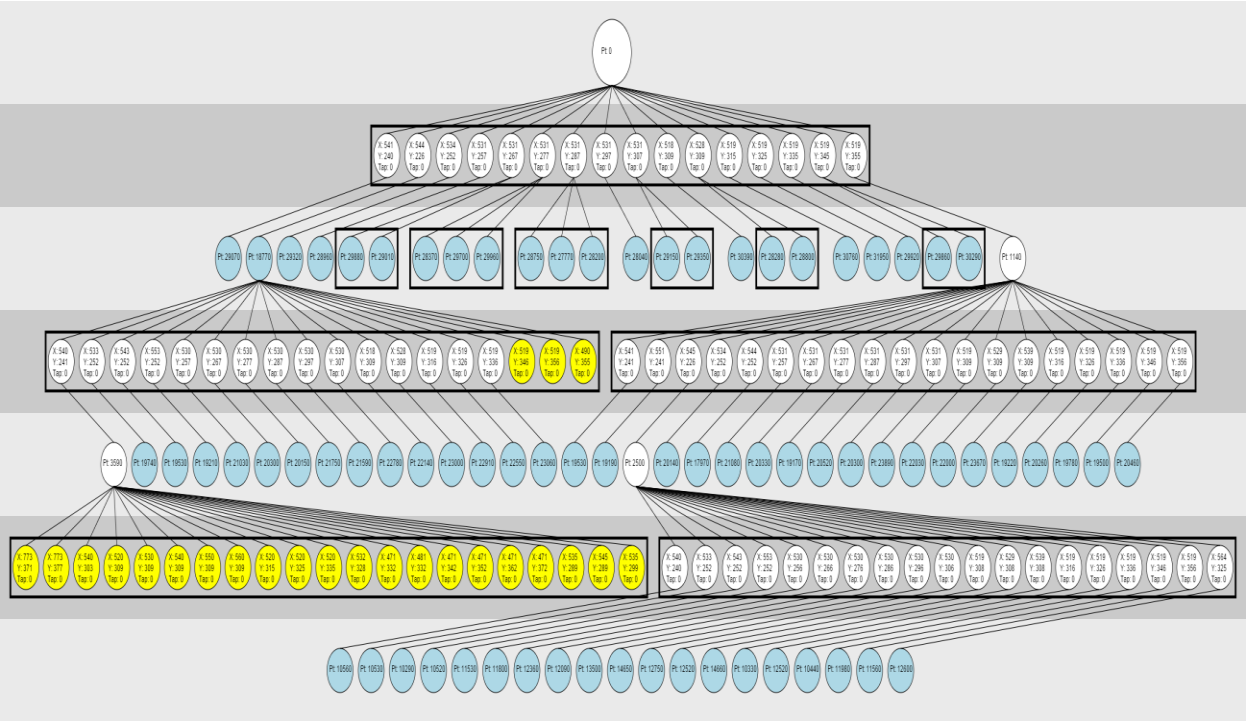

Figura 3:Grafo da Fase um

Existem algumas características visuais que são importantes nesse grafo:

- Nós brancos: nós não terminais, ou seja, não folhas;
- Nós amarelos: nós de possíveis tiros que o algoritmo nunca percorreu no modo de aprendizado.
- Nós azuis: nós de estado que terminam a fase com vitória, ou seja, nós folhas vencedores;

- Nós vermelhos: nós de estado que terminam a fase com derrota, ou seja, nós folhas perdedores;
- Nós agrupados por retângulos pretos: nós irmãos, filhos do mesmo pai.
- Nós de estados tem preenchido neles a quantidade de pontos;
- Nós de tiros tem preenchido a coordenada x, y e o *tapInterval*(será futuramente explicado nesse trabalho) da jogada.

3.2.1 Construção do grafo – Aprendizagem

O agente inteligente executa em dois modos: aprendizado e execução. Por padrão o modo utilizado é o de execução. Para ativar o modo aprendizado o programa precisa rodar com o seguinte argumento:

```
-learn=TYPE_OF_LEARNING
```

Onde em TYPE_OF_LEARNING os parâmetros aceitos são:

- confirmBestResults: Modo muito parecido com o modo RUN, porém grava os resultados na base de dados;
- roundRobin: Escolhe os tiros menos utilizados até que todos estejam com o mesmo número de tentativas;
- allShots: Escolhe o tiro com mais tiros filhos não testados;
- random: Modo randômico porém com um pouco mais de prioridade para tiros ainda não testados;

A seguir temos o pseudocódigo do modo aprendizagem:

```
While programIsRunning {
	loadLevel( levelNumber )
	state <- rootNode <- readGraphFromFile ( )
	shot <- null
	While gameState <> WON and gameState <> LOST{
		If shot <> null then {
```

```
                state <- calculateShotStats( shot );
                updateGraph( state, shot );
                writeGraphInFile( rootNode );
            }
            state.possibleShots <- findPossibleShots( state )
            shot <- chooseOneShot( learningMode, state.possibleShots )

            doShot( shot );
        }
}
```

Como pode ser visto muitas coisas são parecidas com o algoritmo de aprendizado por reforço citado anteriormente.

3.2.2 Métodos Importantes do Modo de aprendizagem

Alguns métodos importantes precisam ser destacados:

- calculateShotStats: Calcula estado inicial do jogo(pontos quantos passarinhos ainda restam, se foi uma jogada vencedora), além disso é tirado uma foto do jogo para uso na visualização de grafo.
- updateGraph: O grafo é atualizado(pontos, numero de vezes que esse tiro e estado foram alcançados entre outros), caso seja um novo estado ele é incluído no grafo junto com todas possíveis jogadas naquele estado.
- writeGraphInFile: A cada jogada o grafo é atualizado em arquivo permitindo parar o algoritmo e voltar quando for desejado.
- findPossibleShots: Esse método é o mais demorado dentro do modo aprendizado. Caso um novo estado seja alcançado, é calculado todas possíveis jogadas da seguinte forma:
    - São pegos todos os objetos em cenas;
    - Se o objeto tiver o tamanho maior que a metade do pássaro(20x20) é mapeado em cada objeto vários locais de tiro, onde cada local tem uma distancia euclidiana de 10 pontos para qualquer outro local já mapeado.

    - Para cada local de tiro mapeado, dois tiros são gerados, um tiro em formato de parábola e um tiro "reto".
    - Caso a cor do pássaro seja diferente de vermelho, cada jogada é clonada cinco vezes apenas adicionando um *tapInterval* diferente, o *tapInterval*, é o tempo em percentagem que após o pássaro sair do estilingue vai ser ativado o poder especial do pássaro.'
    - Após calcular os possíveis tiros esses tiros são guardados como filho do estado anterior, para não precisar ser caculado novamente.
- chooseOneShot: Escolhe o tiro entre os possíveis tiros de acordo com o TYPE_OF_LEARNING.

3.3 Visualizações do grafo

Além do Agente foi desenvolvido um programa utilizando apenas Html, Javascript e CSS para visualizar o grafo de tiro e estados de cada fase. Dentro do projeto do agente na pasta *reports* existe um arquivo chamado graphView.html acompanhando de um arquivo texto de instruções de como usar "Instruções Visualizador de Grafo.txt".

A imagem abaixo mostra o programa em execução mostrando o grafo da fase numero um.

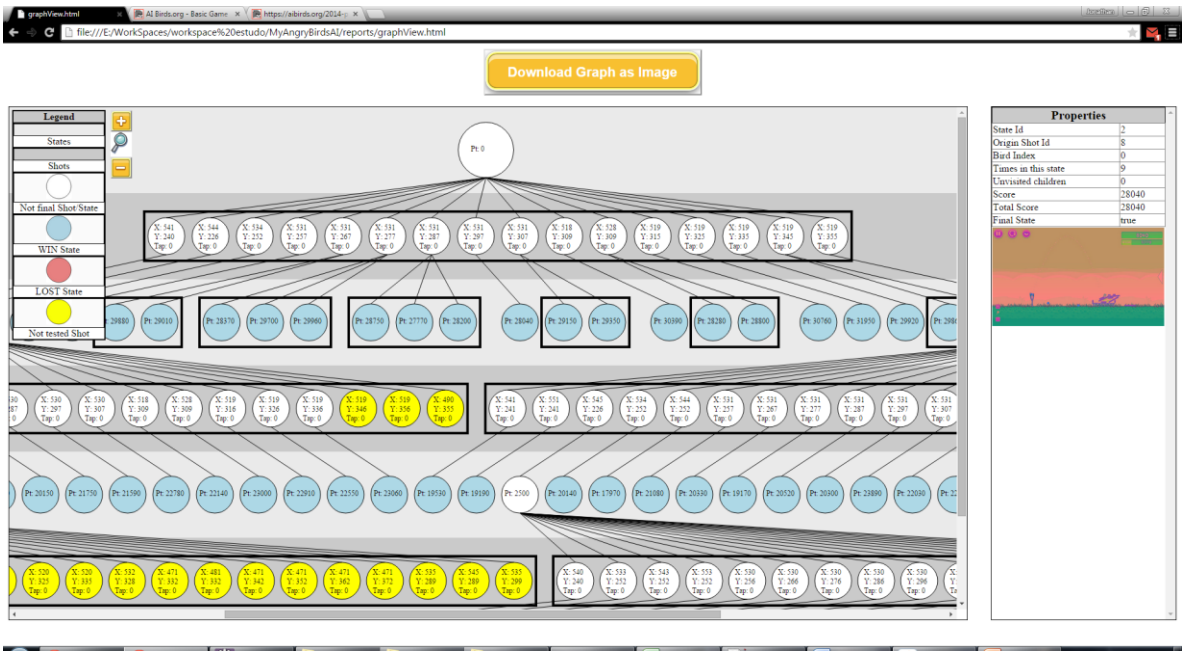

Figura 4: Visualizador de Grafos

3.4 Modo de Execução

Ao executar o agente sem o modo de aprendizagem o algoritmo tem duas grandes diferenças do modo aprendizagem.

A primeira delas é sutil, o grafo não é alterado, ou seja, ele apenas é utilizado e a sua navegação não alterará os arquivos físicos do grafo.

A segunda mudança é o método chooseOneShot que agora escolhe a melhor jogada baseado no algoritmo *expectMiniMax*.

O algoritmo foi proposto por Donald Michie em 1996, ele é baseado no algoritmo *miniMax* porém com influencias de probabilidade nos nós.

No modo aprendizado toda vez que um nó estado ou tiro é alcançado é adicionado uma unidade a um contador de quantas vezes o nó foi "usado". Logo quando se tem aquele problema anterior de um nó ter três possíveis resultados, é possível calcular a probabilidade de cada um acontecer.

Ou seja, a escolha de qual tiro executar é feita através de uma "média ponderada" de seus possíveis estados.

Veja o pseudocódigo:

```
expectMiniMax(node){
	float alpha = 0;
	if( node.isFinalState() ){
		return node.getScore();
	}

	If node is a State then{
		ForEach node.possibleShot{
			alpha = Math.max(alpha, expectMiniMax(shot) );

		}
	}else if node is a Shot{
		float totalTimes = 0;
		ForEach node.possibleStates {
			totalTimes += state.getTimes();
		}

		ForEach node.possibleStates {
			alpha = alpha + ( state.times / totalTimes *
expectMiniMax(state) );
		}


		return alpha;
	}
}
```

4. Resultados

Além da comparação com o algoritmo default, o agente foi comparado com outros cinco agentes. Esses cinco agentes foram das duas últimas edições da competição.

O agente padrão de execução utiliza de várias escolhas randômicas, logo é muito difícil pegar o real desempenho dele, foram feitas três execuções de todas as fases para obter uma média. Outra observação precisa ser feita sobre esse agente, foi capturado o resultado apenas de quando houve vitória. Por padrão o algoritmo reinicia automaticamente ao falhar na fase. Por isso o tempo de execução do agente padrão é bem elevado.

A primeira vantagem do agente desenvolvido é o tempo bem reduzido para completar as 21 fases.

Os resultados a seguir, mostram o resultado do agente (chamado de My Agent) comparado com os cinco melhores algoritmos da competição do ano passado e o agente padrão.

No gráfico a seguir vemos os resultados das nove primeiras fases, nessas fases apenas pássaros vermelhos são utilizados.

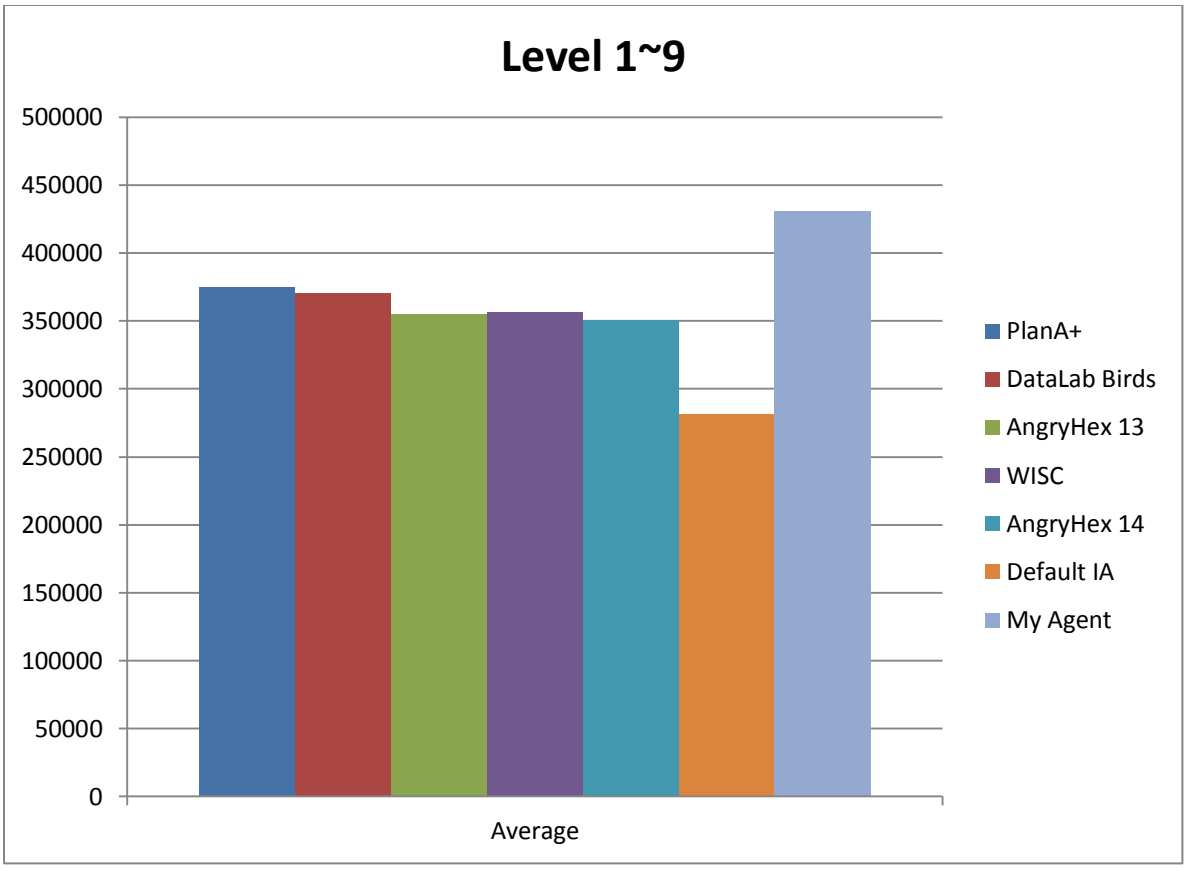

Figura 5: Resultados 9 primeiras fases

No gráfico a seguir é mostrado o resultado da fase 10 até a 14 onde é introduzido o pássaro de cor azul, onde sua habilidade é especial é se dividir em outros três.

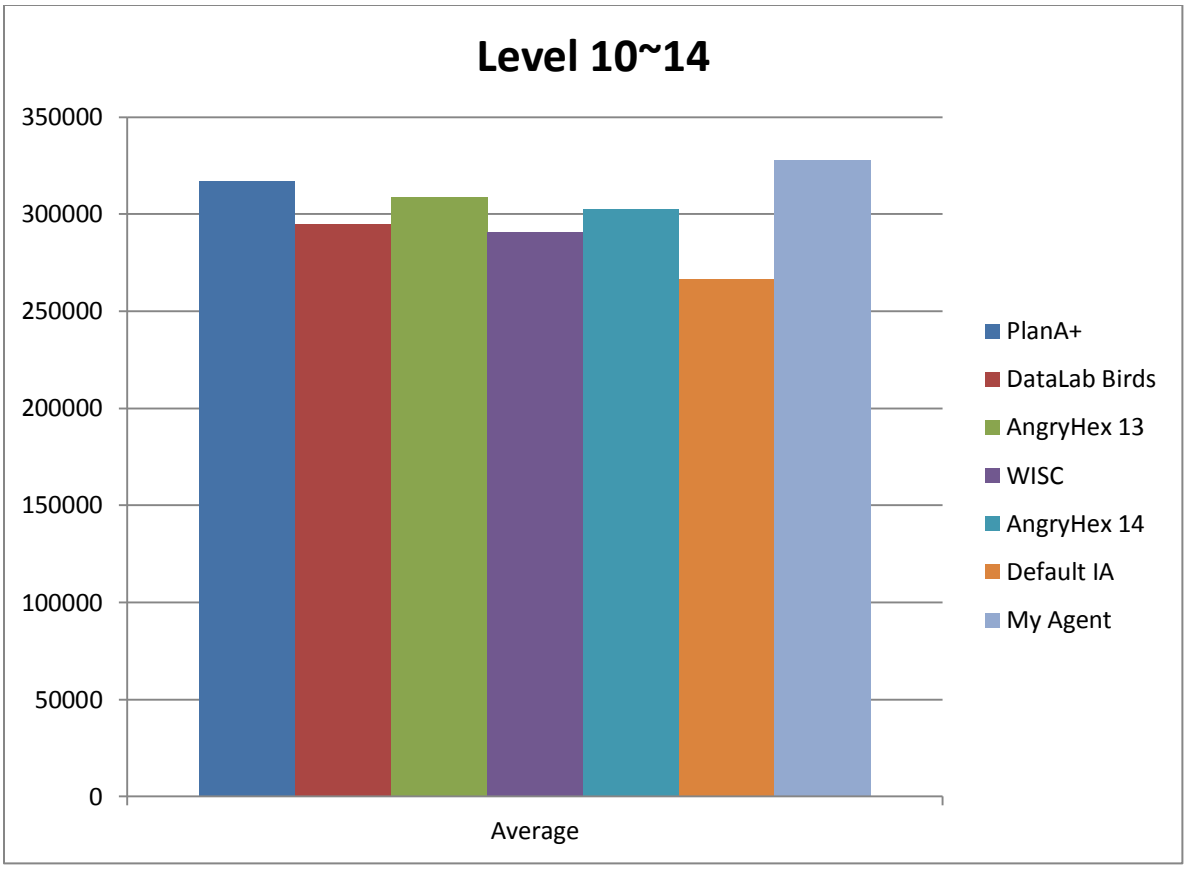

Figura 6: Gráficos 10~14.

Na fase 15 a 21 aparece um novo pássaro de cor amarela com a habilidade de mover mais rapidamente. O Gráfico a seguir mostra o resultado dessas fases em que aparecem os três tipos de pássaros.

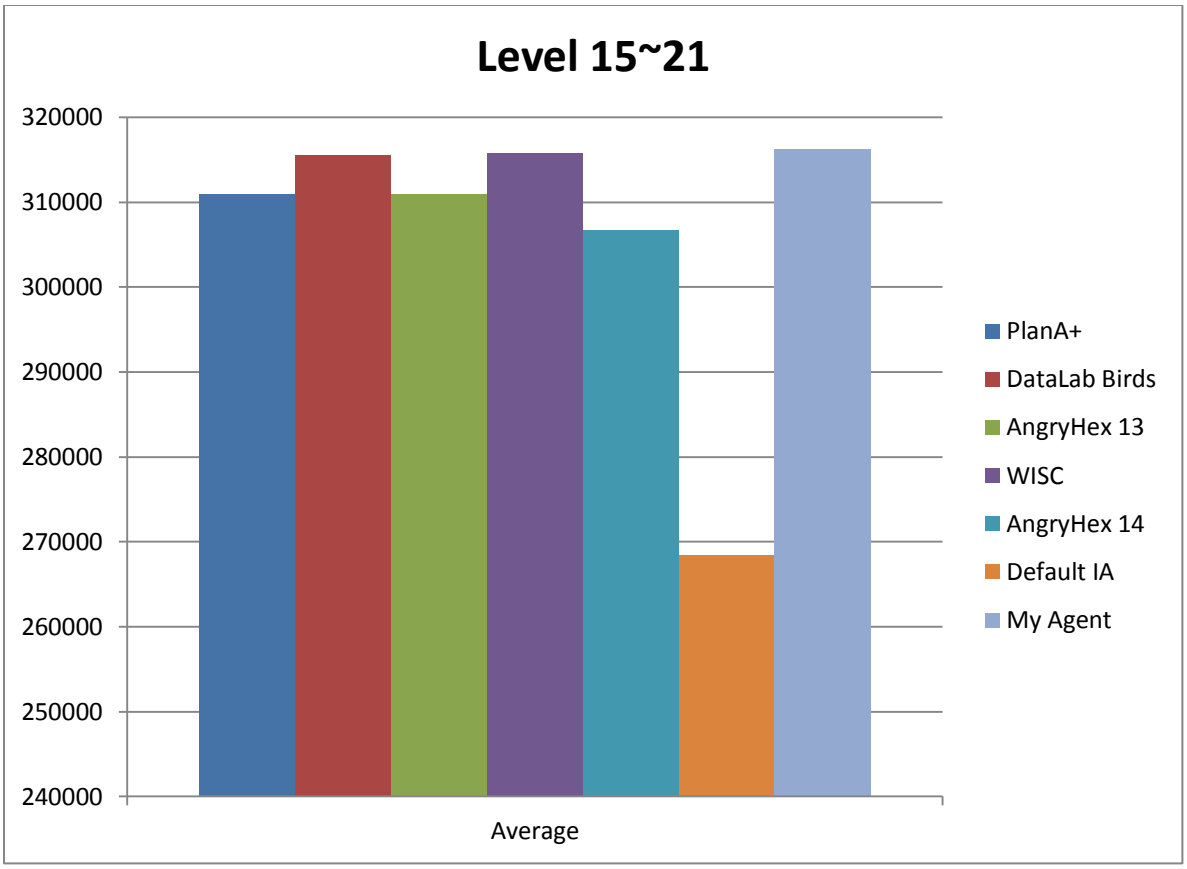

Figura 7: Gráfico 15~21.

O gráfico a seguir mostra o resultado somando todas as fases, onde mostra uma boa vantagem do agente desenvolvido.

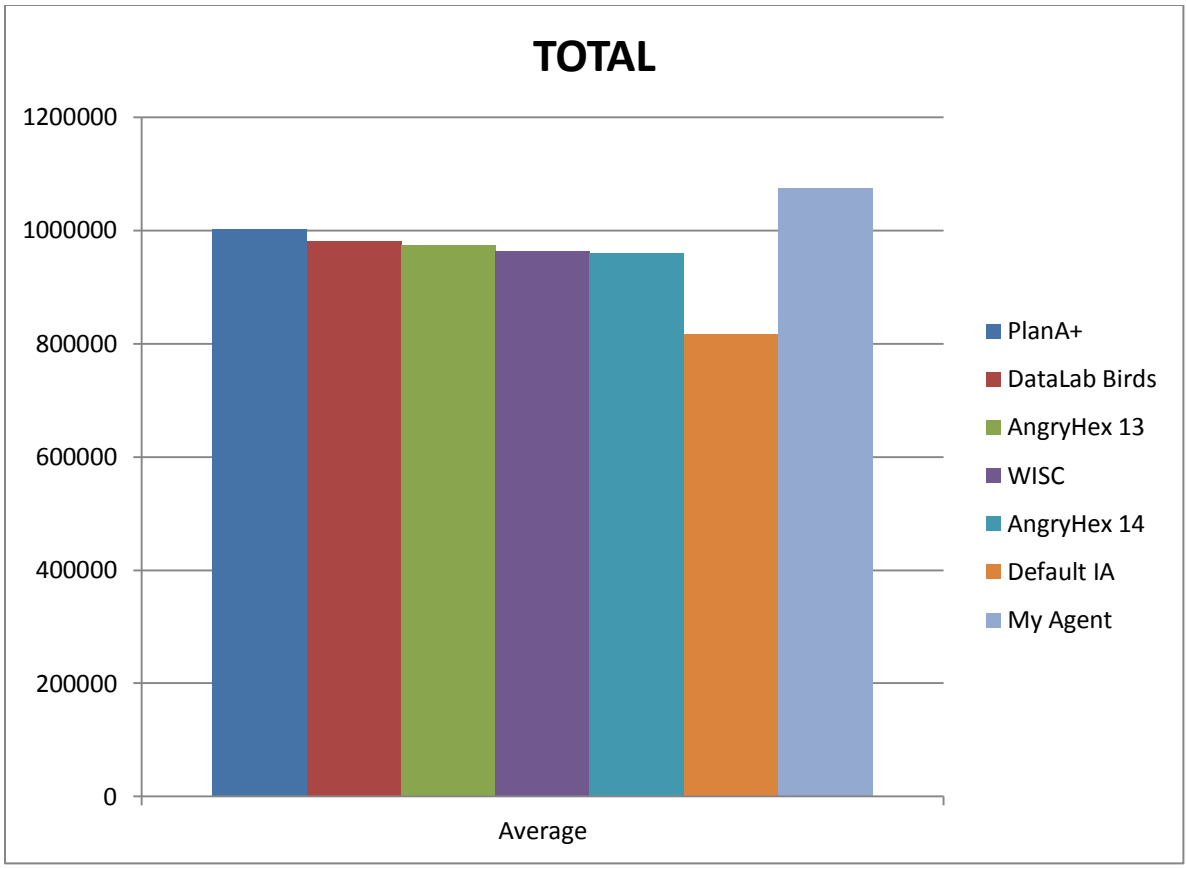

Figura 8: Gráfico Totais de pontos.

Os resultados completos pode ser encontrados na planilha IA Comparision results.xls.

5. Limitações e Problemas

Apesar de bem eficiente alguns problemas e limitações pode ser detectadas:

- Tempo de aprendizado elevado: Dependendo do número de objetos do cenário, o algoritmo precisa de algumas dezenas de execuções para começar a achar possíveis bons resultados.
- No modo execução pode-se chegar num estado não listado no grafo.

6. Futuras melhorias

Algumas melhorias podem ser enumeradas:

- A principal futura melhoria que se pretende implementar é a utilização de Perceptron para identificar cenários iguais ou muito parecidos. Diversos tiros podem resultar resultados iguais ou tão parecidos que podem ser unificados como o mesmo. Isso vai ser importante pra diminuir o tempo de aprendizagem, pois podemos verificar que em outro ramo do grafo ter uma situação bem parecida, sendo necessária apenas a cópia dos nós. No modo execução caso chegamos num cenário não listado podemos achar o cenário mais parecido e assim continuar jogando com ótimo desempenho;
- Mais treinos em busca de melhores resultados;
- Derivar jogadas, ou seja, a partir de duas jogadas boas e próximas entre si, achar outras jogadas.

7. Conclusão

O agente desenvolvido conseguiu um excelente resultado comparado com os demais agentes, porém ainda existem bastante no que melhorar alguns resultados obtidos em algumas vezes são instáveis. Porém, mesmo as piores pontuações do agente são tão bons ou melhores que os resultados das duas ultimas edições da competição.

8. Bibliografia

Rovio. Disponível em: http://www.rovio.com/. Acesso em abril de 2015.

Aibirds. Disponível em: https://aibirds.org/ . Acesso em abril de 2015.

D. Michie (1996). "Game-playing and game-learning automata".