Universidade Federal do Rio Grande do Sul
Inteligência Artificial — INF01048)
Prof. Bruno Castro da Silva
Trabalho 1 — 2016/1

Relatório de Implementação do Controlador de IA para o Segway

Bhárbara S. S. Amaro Cechin — 00240430
Gabriel Marangoni Moita — 00242248

1. Introdução

Este documento apresenta um relatório sobre as atividades feitas durante o processo de desenvolvimento do controlador de IA para o Segway. O objetivo do trabalho constituía na utilização de técnicas de Inteligência Artificial para a elaboração de um controlador capaz de equilibrar um Segway simulado. Para esse desenvolvimento, o grupo analisou o problema e analisou quais seriam os melhores algoritmos e *features* a serem utilizados, chegando a conclusão que testariam os algoritmos de Simulated Annealing e Busca de Feixe Local, ambos trabalhando sobre o conjunto de *features* básicas $s_t = \{x, vel, ang\}$, sendo a posição na tela, a velocidade angular do cabo e o ângulo do cabo, respectivamente. O grupo decidiu não adicionar *features* extras por não achar necessário para que os algoritmos pudessem aprender os parâmetros necessários e atingissem o objetivo de equilíbrio. Esse conjunto está presente na função *compute_features*. Sendo assim, o conjunto final de equações da função *take_action* é:

$$Q_\Theta(s_t, D) \equiv Q_\Theta(x, vel, ang, D) \equiv (\Theta_1 \times x) + (\Theta_2 \times vel) + (\Theta_3 \times ang)$$
$$Q_\Theta(s_t, E) \equiv Q_\Theta(x, vel, ang, E) \equiv (\Theta_4 \times x) + (\Theta_5 \times vel) + (\Theta_6 \times ang)$$
$$Q_\Theta(s_t, N) \equiv Q_\Theta(x, vel, ang, N) \equiv (\Theta_7 \times x) + (\Theta_8 \times vel) + (\Theta_9 \times ang)$$

Todos os testes foram executados em um computador com Windows 10 64-Bit com processador Intel Core i7-4510U com 4 núcleos e frequência de 2.6GHz, utilizando Python 2.7.11 Uma série de atributos foram adicionados na classe *Controller* para permitir a implementação dos algoritmos descritos, podendo ser vistos no código fonte.

2. Algoritmos de Busca Avaliados

a. Simulated Annealing (Têmpera Simulada)

O primeiro algoritmo pensado, logo que o grupo foi apresentado ao problema, foi o Simulated Annealing. A escolha foi feita pensando nos potenciais máximos locais em que um algoritmo de Hill Climbing poderia acabar caindo. O algoritmo implementado corresponde ao seguinte trecho de código:

```
#SIMULATED ANNEALING
        temperature = (10000/(episode+1))
        if performance > self.last_performance: #se for melhor, salva o novo, senão, depende
da probabilidade/temperatura
                self.last_parameters = self.parameters
                self.last_performance = performance
        else:
            if temperature > 0:
                diff = performance - self.last_performance
                prob = math.e**(diff/temperature)
                if (round(random.uniform(0,1),5) <= prob): #será considerado, mesmo sendo
pior; senão, não será atualizado
                    self.last_parameters = self.parameters
                    self.last_performance = performance
        disturb = numpy.random.normal(0, 1, 3*len(self.compute_features()))
        disturb *= 0.1
        self.parameters += disturb
```

Em resumo, o algoritmo verifica se a performance do episódio foi melhor que a do episódio anterior. Se for melhor, os parâmetros são salvos. Se for pior, com probabilidade $e^{diferença/temperatura}$ os parâmetros são salvos mesmo assim. O parâmetro de temperatura utilizado foi $temperature = (10000/\#episódio + 1)$, somando-se um no divisor para evitar divisão por zero. Caso contrário, os parâmetros do último é mantido. Para a próxima execução, os parâmetros da próxima execução são gerados a partir da distorção dos parâmetros escolhidos, utilizando-se uma distribuição normal (e multiplicados por 0.1 para a distorção não ser exagerada). Sendo assim, era esperado que próximo ao 10000º episódio o algoritmo estabilizasse e tivesse aprendido os parâmetros, entretanto pode ser interrompido a qualquer momento através de uma interrupção do usuário.

b. Busca em Feixe Local

O segundo algoritmo escolhido para implementação foi o de Busca em Feixe Local. A escolha foi feita após fracasso da implementação do Simulated Annealing (vide item 3a do relatório). Notou-se que uma das grandes dificuldades para o algoritmo anterior era o possível *spawn* do Segway muito perto dos limites da tela, ocasionando em uma performance ruim e abandono pelo algoritmo dessa possibilidade. Pensou-se que, rodando 10 distorções para cada um dos 10 conjuntos de parâmetros e escolhendo os 10 melhores, tal dificuldade seria minimizada, além de continuar escapando de máximos locais. O algoritmo implementado corresponde ao seguinte trecho de código:

```
#LOCAL BEAM SEARCH
        if(self.itparents == 0):
            self.keys.append(self.parameters)
            self.ten_parents[self.itparents] = performance #save in parents hash
            for i in range(9): #generate other 9 parents (the 1st is the generated by
initialize_parameters in __init__)
                self.next_parents.append(numpy.random.normal(0, 1,
3*len(self.compute_features())))
            self.parameters = self.next_parents[0] #pick first next parent
            self.itparents += 1;

        elif(self.itparents < 9): #processing first 9 parents
            self.keys.append(self.parameters)
            self.ten_parents[self.itparents] = performance #save in parents hash
            self.parameters = self.next_parents[self.itparents] #pick next parent
            self.itparents += 1;

        elif (self.itparents == 9): #end of processing first parents, generate first
generation (each parent has 10 "children")
            self.keys.append(self.parameters)
            self.ten_parents[self.itparents] = performance #save in parents hash
            for parent in self.ten_parents.keys():
                parent_params = self.keys[parent]
                for i in range(10):
                    disturb = numpy.random.normal(0, 1, 3*len(self.compute_features()))
                    #disturb *= 0.1
                    self.next_hundred.append(parent_params+disturb) #append 10 disturbed
version of each father
            self.itparents = 11 #to never enter in the first three if's again
            del self.keys[:]
            self.parameters = self.next_hundred[self.ithundred] #pick first child
            self.ithundred +=1;
            print "End of processing fathers, starting children processing..."

        elif (self.ithundred > 0 and (self.ithundred % 100) != 0): #processing 100 children
            self.keys.append(self.parameters)
            self.last_hundred[(self.ithundred%100)-1] = performance #save in children hash
            self.parameters = self.next_hundred[self.ithundred%100] #pick next child
            self.ithundred +=1;

        elif (self.ithundred > 0 and (self.ithundred % 100) == 0): #end of processing
children, select Top 10 and generate next generation
            self.keys.append(self.parameters)
            self.last_hundred[(self.ithundred%100)-1] = performance #save in children hash
            self.ten_parents =
dict(sorted(dict(self.last_hundred.items()).iteritems(),key=operator.itemgetter(1),reverse=Tru
e)[:10])#select 10 best children
            del self.next_hundred[:]
            for parent in self.ten_parents.keys():
                parent_params = self.keys[parent]
                self.parents_keys.append(parent_params) #for saving parameters
                for i in range(10):
                    disturb = numpy.random.normal(0, 1, 3*len(self.compute_features()))
                    #disturb *= 0.1
                    self.next_hundred.append(parent_params+disturb) #append 10 disturbed
version of each father
            del self.keys[:]
            self.parameters = self.next_hundred[self.ithundred%100] #pick first child
            self.ithundred +=1;
```

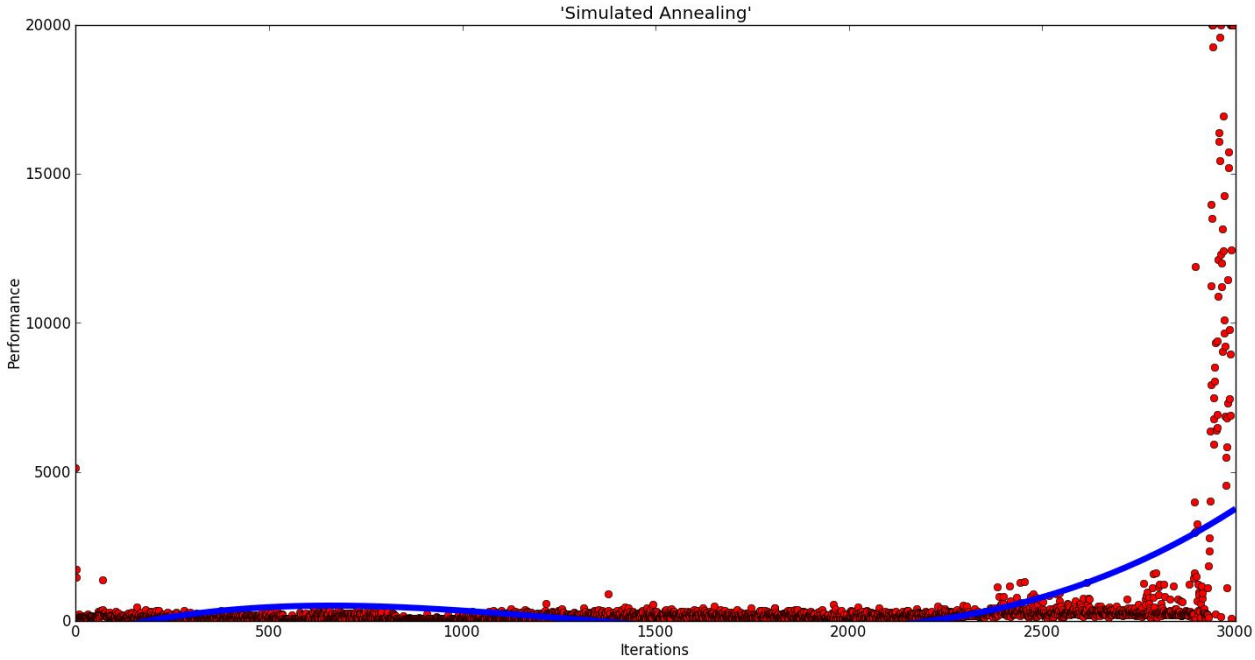
Em resumo, o algoritmo, após a geração pelo __init__ do primeiro conjunto de parâmetros, gera outros 9 conjuntos de parâmetros aleatórios e testa suas performances, constituindo essa a geração 0. Após isso, a partir de cada conjunto, gera 10 distorções

deles, da mesma forma que o Simulated Annealing, totalizando em 100 candidatos da geração 1, que tem suas performances testadas. Desses 100, os 10 melhores são selecionados para serem pais da próxima geração, que é gerada da mesma forma que a geração 1. O algoritmo continua rodando até ser interrompido, sendo que ao fim de cada geração um dos 10 melhores (selecionado aleatoriamente) tem seus parâmetros salvos em um arquivo. O algoritmo se assemelha a uma reprodução assexuada, como dito em aula.

3. Performance dos Algoritmos

a. Simulated Annealing (Têmpera Simulada)

O algoritmo de Simulated Annealing teve um desempenho muito aquém do imaginado. Seu aprendizado era muito lento e sensível a execuções ruins por causa de *spawn* muito próximo das bordas da tela, muitas vezes abandonando soluções promissoras. O problema não era na temperatura, pois testaram-se várias fórmulas diferentes, todas sem sucesso. Um exemplo de execução é o seguinte:

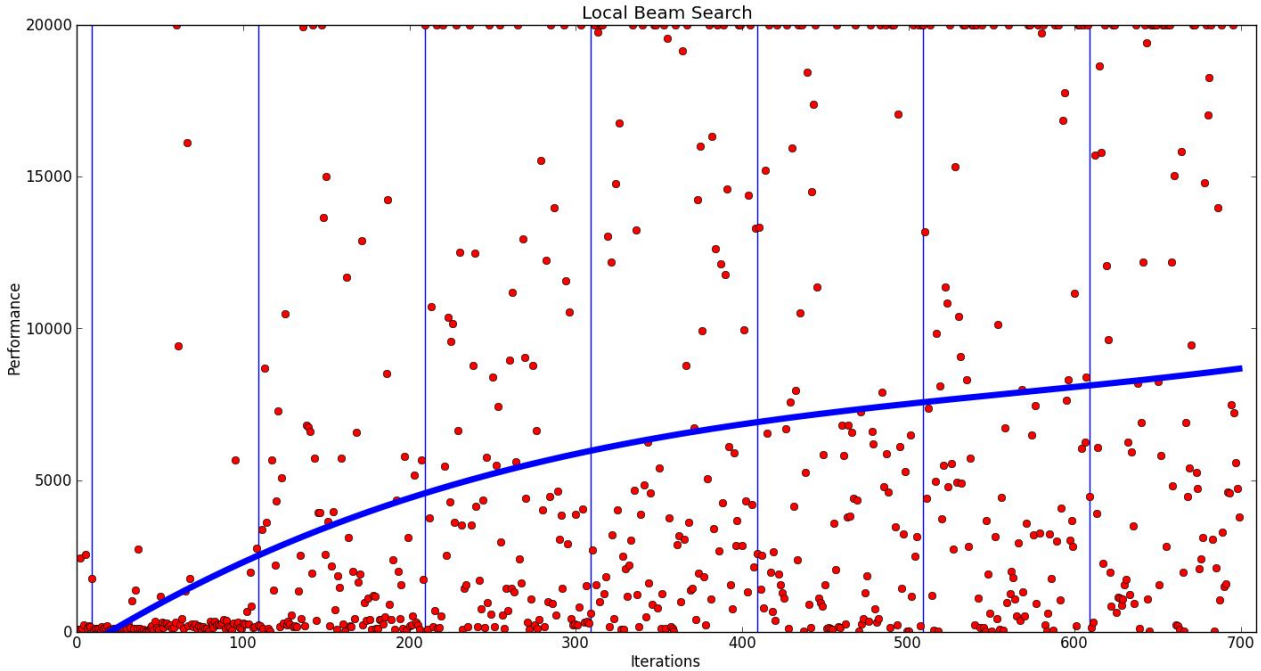


Cada ponto vermelho é a performance de um episódio, enquanto a linha azul é uma curva de ajuste de grau 3, pois não conseguiu-se fazer a curva de ajuste logarítmica. Nota-se que demoraram aproximadamente 3000 episódios para que a performance melhorasse, a curva de aprendizado não sendo da forma esperada: ao invés de aprender rápido e depois chegar a um limite superior, o algoritmo fica muito tempo com performances baixas e, de repente, explode em performance boas.

b. Busca em Feixe Local

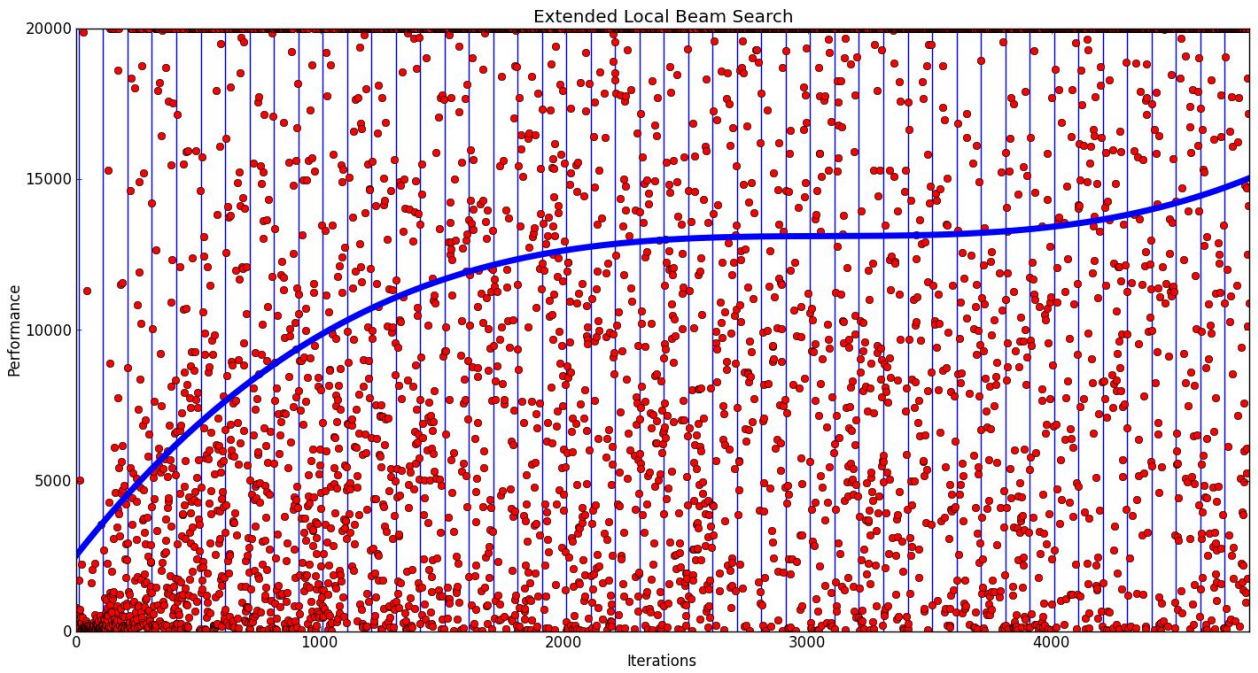
O algoritmo de Busca em Feixe Local, por outro lado, apresentou um desempenho muito bom, tendo um aprendizado rápido, eficiente e menos sensível a performances ruins. A isso se atribuiu ao fato de o algoritmo executar 100 episódios em cada geração e, a partir deles, escolher os 10 melhores para gerar os próximos 100. Sendo assim, uma combinação boa de parâmetros tem 10 chances de boas execuções a suas distorções, ao

invés de apenas uma, reduzindo a chance de ser descartado por *spawn* desfavorável. Um exemplo de execução é:



Cada ponto vermelho é a performance de um episódio, enquanto a linha azul é uma curva de ajuste de grau 3, pois não conseguiu-se fazer a curva de ajuste logarítmica. O intervalo entre duas linhas verticais azuis é uma geração de 100 execuções (com exceção da geração 0, que é de 10 execuções). Nota-se que poucas gerações foram necessárias para encontrar-se uma boa combinação de parâmetros. O número de execuções é consideravelmente menor do necessário pelo algoritmo de Simulated Annealing. A curva de aprendizagem é da forma esperada.

Outra execução, dessa vez propositalmente mais longa, mostra o quão resistente o algoritmo é:



Nesse gráfico, é possível perceber também a saturação do algoritmo após um determinado número de gerações. É importante notar que a curva apenas cresce após um tempo por ser uma curva de ajuste linear de grau 3 ao invés de uma logarítmica. Isso se deve ao fato de, ao ter mais de 10 conjuntos de parâmetros com performance de 20000, dez aleatórios serem escolhidos, provavelmente os 10 primeiros de performance 20000 a serem executados.

É importante frisar que todos os gráficos são relativos a execuções com o grupo de *features* básicas. Ao acrescentar *features*, apenas percebeu-se um maior número de episódios necessários para se encontrar um bom conjunto de parâmetros.

4. Dificuldade Encontradas

Várias dificuldades foram encontradas no desenvolvimento do controlador. O primeiro foi perceber como o código funcionava, visto que existe quase nenhuma documentação a respeito da execução do simulador. Após essa etapa, a primeira dificuldade encontrada foi como ter uma memória dos parâmetros da execução anterior, para isso sendo adicionadas novos atributos a classe *Controller*. Outra dificuldade enfrentada durante todo o projeto foi a recuperação dos parâmetros após a execução para vê-los na execução em modo *evaluate*, visto que o programa apenas salvava a cada 10 episódios de aprendizado.

Uma dificuldade também existente foi recuperar as performances durante o aprendizado, visto que o programa apenas salvava as performances no modo *evaluate*, sendo necessário encontrar e alterar para que salvasse no modo *learn*. Outra dificuldade incômoda foi o *spawn* do segway poder acontecer próximo dos ou nos limites da tela, inclusive no modo *evaluate*, o que muitas vezes atrapalhava os algoritmos e incomodava na visualização do controlador funcionando.

Mais uma dificuldade enfrentada foi a resistência ao vento. Não conseguiu-se que os algoritmos aprendessem que, além de ser importante se equilibrar, era ainda mais importante manter-se próximo ao centro da tela, o que faria com que ele acabasse vencendo o vento. Isso é bastante dificultado pela função que avalia a performance do segway ser apenas o tempo que ele ficou vivo, além de o vento não ser ligado no modo *learn*. Na continuidade do trabalho, tentar-se-á modificar a função de avaliação de performance para que ela também leve em consideração a distância média em relação ao centro da tela e que o modo *learn* tenha também um vento aleatório ligado.

5. Conclusão

A implementação do controlador de Inteligência Artificial do segway ajudou a reforçar os conceitos vistos em aula, assim como a testar o que os integrantes do grupo pensavam sobre os algoritmos testados, aprendendo melhor como eles funcionam e suas limitações de uso, principalmente quando não se tem acesso aos valores de todos os vizinhos para tomar a decisão de qual próximo conjunto de parâmetros escolher. O algoritmo de Busca em Feixe Local obteve um desempenho melhor comparado ao Simulated Annealing, por considerar os episódios anteriores, tornando-o menos sensível a situações desfavoráveis de testes e tendo uma curva de aprendizado conforme o esperado de um algoritmo de aprendizagem. Conseguiu-se alcançar o objetivo de criar um controlador que, com Inteligência Artificial, aprendesse por conta própria a equilibrar o segway simulado.