

Comparando planejamento e aprendizado por reforço em ambientes com obstáculos

Daniel Orlandi and Maurício Pires

Instituto de Computação

Universidade Federal Fluminense - Niterói

Abstract

Uma das principais características dos agentes inteligentes é a capacidade de tomar decisões de maneira autônoma. A determinação de trajetórias em agentes robóticos é um problema relevante e possui diferentes abordagens de solução. Este trabalho busca comparar a técnica de planejamento e aprendizado por reforço para a movimentação de um agente robótico em um ambiente com obstáculos.

Introdução

Atualmente no campo da robótica, há uma ênfase no aumento da autonomia de robôs. Um aspecto fundamental dessa autonomia, é a habilidade de um robô em planejar seus movimentos, visando executar, com sucesso, uma tarefa ou objetivo a ele designado. As aplicações de agentes robóticos autônomos cobre uma vasta quantidade de áreas, desde problemas de entregas a aplicações médicas.

Dentre os principais desafios da área, a capacidade de um robô determinar a melhor trajetória para a execução de uma tarefa é um ponto importante. Dentro de inteligência artificial, a clássica estratégia de planejamento apresenta resultados interessantes, embora sejam necessárias abordagens mais elaboradas quando encaram-se ambientes parcialmente observáveis e dinâmicos. O desenvolvimento das técnicas de aprendizado de máquina também tem obtido resultados significativos, porém as técnicas possuem um custo de tempo e computacional demasiadamente grande.

Este trabalho focou-se em experimentar a descoberta de trajetórias em ambientes fechados e totalmente observáveis utilizando alguns algoritmos de *path planning* e aprendizado por reforço.

Preliminares

Antes de mencionar as ferramentas utilizadas nesse trabalho experimental, faz-se necessário abordar os principais conceitos teóricos sobre os quais se baseiam as ferramentas e experimentos detalhados neste relatório.

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

Planejamento baseado em amostras

Tarefas de planejamento são tipicamente descritas como sendo compostas por uma configuração (estado) inicial e uma configuração final e a tarefa do agente é encontrar a melhor sequência de ações que permita atingir o objetivo.

Path/motion planning são duas categorias de planejamento que estão associadas a aspectos geométricos, onde o objetivo do agente é encontrar uma trajetória que obedeça a geometria do robô e do ambiente. Apesar de sua descrição simples, esta tarefa é complexa e, muitas vezes, exige um certo nível de conhecimento do ambiente em ordem de obter uma melhor sequência de ações.

Muitos problemas de planejamento em três dimensões podem ser interpretados por meio do clássico problema de mover um piano. Este problema assume que o piano deve ser movido de uma posição inicial para uma outra dentro de um ambiente com obstáculos, de modo a não existirem colisões.

Este problema é conhecidamente *PSPACE-hard*, o que demanda que sejam desenvolvidas técnicas que permitam ao agente computar uma solução, ainda que aproximada, em um tempo de processamento viável, visto que muitas tarefas são executadas em ambientes com restrições de tempo. Nesse contexto, surgem as técnicas de planejamento baseadas em amostras, que visam reduzir a complexidade do espaço a ser explorado pelo agente.

Dentre as abordagens conhecidas baseadas em amostras, a mais comum é a baseada em grafos. De maneira bem resumida, são geradas amostras aleatórias do espaço de estado baseado em não-colisões. Após esta etapa, as amostras são conectadas formando um grafo, cujos caminhos representam possíveis soluções para o problema. Esta abordagem utiliza um planejador local de modo que, entre dois vértices desse grafo de amostras, o caminho pode ou não ser interpolado pelo planejador para computar uma pequena trajetória livre de colisões.

Rapidly-exploring random tree

Um *rapidly-exploring random tree* (*RRT*) é um algoritmo projetado para pesquisar, de forma eficiente,

espaços de estados não convexos de alta dimensão. O algoritmo utiliza uma técnica incremental para a construção de uma árvore de solução, de modo que a distância entre uma amostra aleatória e a árvore de solução seja reduzida. Esta técnica é ideal para problemas de planejamento de caminho que envolvem obstáculos e restrições diferenciais (não-holonômicas ou kinodinâmicas). Uma descrição do algoritmo pode ser encontrada na Figura 1.

Figure 1: Pseudocódigo do RRT.

```

 $\mathcal{T} \leftarrow \text{InitializeTree}()$ 
 $\mathcal{T} \leftarrow \text{InsertNode}(\emptyset, z_{init}, \mathcal{T})$ 
for i = 0 to i = N do
     $z_{rand} \leftarrow \text{Sample}(i)$ 
     $z_{nearest} \leftarrow \text{Nearest}(\mathcal{T}, z_{rand})$ 
     $(z_{new}, U_{new}) \leftarrow \text{Steer}(z_{nearest}, z_{rand})$ 
    if  $\text{ObstacleFree}(z_{new})$  then
         $\mathcal{T} \leftarrow \text{InsertNode}(z_{min}, z_{new}, \mathcal{T})$ 
return  $\mathcal{T}$ 
```

O algoritmo recebe uma amostra do espaço de estados como entrada. A partir dele, uma árvore será enraizada e irá de crescer de maneira aleatória. Para cada iteração do algoritmo, uma amostra será selecionada. Após selecionar a amostra, o algoritmo calcula qual nó da árvore está mais próximo da amostra obtida. Ambos os nós são submetidos a um teste de distância máxima. Caso eles não atendam essa restrição, um nó é pego no intervalo entre esses dois nós. Se o novo nó for considerado livre de colisões, então ele é adicionado à árvore.

O algoritmo também possui variações que permitem cálculos mais sofisticados, como o RRTStar, que faz busca por caminhos com menor custo, e o RRTConnect, que cresce duas árvores RRT enraizadas na origem e destino da tarefa de planejamento.

Aprendizado por reforço

De modo muito simplista, aprendizado por reforço poderia ser visto como uma técnica de tentativa e erro. Assim como uma criança aprende algumas tarefas por meio das suas falhas, o aprendizado por reforço (reinforcement learning - RL) visa “replicar” esse comportamento em sistemas computacionais.

No RL padrão, o agente computacional recebe uma percepção de como o mundo parece estar. Tendo em vista o estado em que se encontra o mundo, ele deve escolher uma ação (de um conjunto possível) e atuar no mundo. A atuação dele gera um *sinal de reforço* que o auxilia a perceber se a ação foi boa ou não. O trabalho do agente é descobrir uma política π , mapeando estados em ações, a fim de atingir o maior valor acumulado de sinais de reforço. Alguns autores preferem chamar o sinal de reforço de recompensa.

O aprendizado por reforço é uma técnica muito interessante para lidar com problemas onde há interação. Nesse tipo de problemas, é quase impraticável a existência de um supervisor que possa apontar ao

agente os comportamentos esperados dadas situações específicas. É como se o agente estivesse em um ambiente inexplorado. Ele deve aprender a melhor forma de interagir com ele a fim de atingir seus objetivos.

Esta técnica sofre do chamado “*exploration-exploitation dilemma*”. Para conseguir o máximo de recompensa, o agente tende a tentar ações que ele tomou antes e obtiveram um bom resultado. Porém, para saber quais ações são melhores recompensadas, ele precisa tentar ações que ele não tentou antes. Esse tipo de problema requer que os algoritmos criem um certo balanceamento entre quando tentar algo novo e quando repetir algo já feito.

Processo de Decisão de Markov

Os problemas de aprendizado por reforço podem ser interpretados como um Processo de Decisão de Markov. As ações realizadas pelo agente transformam o ambiente levando-o a um novo estado, onde esse ente pode realizar uma nova ação, dado um conjunto de ações possíveis. O conjunto de ações passíveis de execução, dado um novo estado, respeita um conjunto de regras, também chamado de políticas. Onde as políticas são responsáveis por controlar a maneira como as ações são tomadas.

O conjunto formado pelas ações e estados, junto com as regras de transição de um estado para outro, formam um processo de decisão de Markov. Cada época (episódio) desse processo (por exemplo, um robô explorando um ambiente) forma uma sequência finita de estados, ações e recompensas:

$$e_0, a_0, r_0 + \dots + e_{n-1}, a_{n-1}, r_n, e_n$$

Na equação acima, e_i representa o estado, a_i a ação e r_i a recompensa. O episódio termina com no estado terminal e_n (por exemplo, o fim da simulação). Um processo Markoviano de decisão se baseia na tese de que a probabilidade de e_{i+1} acontecer, depende apenas de e_i e a_i e não dos estados e ações anteriores.

Discounted Future Reward

Para que a *performance* do modelo não decaia com o tempo, é preciso levar em consideração não apenas a recompensa atual, mas também as recompensas provenientes de estados futuros. Para isso, calcula-se a recompensa total para uma época:

$$R = r_0, r_1, + \dots + r_n$$

A recompensa dos estados futuros é da forma:

$$R_t = r_t, r_{t+1}, + \dots + r_{t+n}$$

Como o ambiente é estocástico, nada garante que uma mesma ação, repetida em um estado futuro, levará a mesma recompensa. Na verdade, é mais provável que quanto mais se avance no futuro mais divergentes serão as recompensas. Por isso adiciona-se um fator de desconto γ , entre 0 e 1. Ao incluir esse fator na equação de

recompensa futura, estamos penalizando as recompensas provenientes de estados localizados muito a frente no eixo do tempo. Portanto, após a inclusão do γ a equação de recompensa dos estados futuros pode ser representada como:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \dots)) = r_t + \gamma(R_{t+1})$$

Quanto maior o valor de γ mais estamos levando em consideração o futuro. Se quisermos um equilíbrio entre presente e futuro, devemos utilizar $\gamma = 0,9$.

Quality-Learning

Uma estratégia coerente para um agente, seria escolher ações que maximizem $R_t = r_t + \gamma(R_{t+1})$.

No *Quality-Learning* define-se uma função $Q(e_t, a_t)$ que representa o *discounted future reward* máximo ao se executar uma ação a no estado e . E , partir dessa função, escolhe-se a próxima ação a ser executada em e_{t+1} . Portanto, a função Q é da forma:

$$Q(e_t, a_t) = \max(R_{t+1})$$

A função $Q(e_t, a_t)$ representa a qualidade de uma certa ação dado um certo estado. Em posse da função Q é trivial encontrar a política π responsável por levar a melhor escolha de ações (ações que maximizam Q). π é da forma:

$$\pi(e) = \text{argmax}_a(Q(e, a))$$

Olhando para apenas uma transição de estados (e, a, r, e') , como na equação do *discounted reward*, é possível escrever:

$$Q(e, a) = r + \gamma \max_{a'}(Q(e', a'))$$

Essa equação, também conhecida como equação de Bellman é a principal ideia por trás do *Q-Learning*. Onde, através de um método interativo, utiliza-se a equação de Bellman para aproximar-se a função Q .

Implementação Q-Learning

O pseudocódigo, simplificado na Figura 2, mostra a implementação da função de Bellman.

Figure 2: Pseudocódigo mostrando uma implementação simplificada de função de Bellman, onde s corresponde a um estado e .

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
    s = s'
until terminated
```

A letra grega α , no algoritmo, representa a taxa de aprendizado e controla o quanto do novo valor de Q será

levado em consideração, na interação atual. O termo $\max_{a'}(Q(s', a'))$, utilizado para atualizar $Q(s, a)$, é uma aproximação, e nos estágio iniciais do treinamento está completamente errado. Contudo, após um número suficiente de interações a função Q converge e representa o Q correto.

Estimar os valores da função Q pode ser muito custoso computacionalmente, fazendo com que a arquitetura demore muito tempo para convergir. Para contornar esse problema, introduziu-se o *experience-replay*, onde todos os conjuntos (e, a, r, e') são armazenados em uma "memória". Durante o processo de treinamento da rede, *mini-batches* aleatórios dessa memória são utilizados ao invés da interação atual. Isso insere algumas descontinuidades na rede, durante o treinamento, diminuindo a similaridade das entradas, que de outro modo, levariam à um *overfitting*.

Outro ponto de implementação também importante, do *Q-learning*, é a forma como o paradigma *Exploration-Exploitation* é tratado. Ao inserir-se um termo ϵ , permite-se que durante o treinamento seja possível, com uma probabilidade ϵ , escolher uma ação aleatoriamente. Isso faz com que a exploração seja aleatória no começo do treinamento, mas com o passar dos episódios, o agente aprende mais e a aleatoriedade de seus movimentos é cada vez menor. Isso implica que o agente passa a conhecer melhor o ambiente que explora e toma suas decisões baseadas no que aprendeu.

A inclusão desses tratamentos ao *Q-learning*, leva portanto a implementação que pode ser observada na Figura 3.

Figure 3: Pseudocódigo mostrando uma implementação do *Q-learning*, onde s corresponde a um estado e .

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
    select an action a
        with probability ε select a random action
        otherwise select a = argmax_a Q(s,a)
    carry out action a
    observe reward r and new state s'
    store experience <s, a, r, s'> in replay memory D

    sample random transitions <ss, aa, rr, ss'> from replay memory D
    calculate target for each minibatch transition
        if ss' is terminal state then tt = rr
        otherwise tt = rr + γ max_a' Q(ss', aa')
    train the Q network using (tt - Q(ss, aa))^2 as loss

    s = s'
until terminated
```

Ferramentas

Para a implementação do trabalho, a ferramenta *Virtual Robot Experimentation Platform* foi utilizada em conjunto com os *plug-ins* para aprendizado por reforço e algoritmos de planejamento.

Virtual Robot Experimentation Platform O *Virtual Robot Experimentation Platform* (V-Rep) é

um ambiente multiplataforma para design e simulação de robôs. No V-Rep é possível controlar cada objeto/modelo via script embarcado, plugin, API Remoto ou através de uma solução customizada. O que faz com que o V-Rep seja extremamente versátil e ideal para aplicações envolvendo diversos robôs. Vale ressaltar que os controladores podem ser escritos em: C/C++, Python, Java, Lua, Matlab ou Octave. Mais informações podem ser encontradas em <http://coppeliarobotics.com>.

Open Motion Planning Library A *Open Motion Planning Library* (OMPL) é uma biblioteca especializada em algoritmos de planejamento baseados em amostras. Ela foi projetada sem estar amarrada a um modelo específico de tratamento de colisões ou visualização, de modo que é genérica e bastante para ser utilizada por diferentes sistemas de simulação de robôs, como o V-Rep por exemplo.

A biblioteca reúne vários algoritmos que compõem o estado da arte dos algoritmos de planejamento baseado em amostras para *path/motion planning* além de estar integrada com diversas plataformas de simulação, como o V-Rep. Mais informações podem ser encontradas em <http://ompl.kavrakilab.org>.

Experimentos

Os experimentos realizados nesse trabalho tiveram como objetivo a construção de uma trajetória para um robô em um labirinto. Os scripts para aprendizado por reforço foram feitos utilizando a API remota do V-Rep, permitindo que eles fossem escritos em Python. Os scripts para planejamento estão embutidos nas cenas testadas no ambiente. Estes scripts foram escritos em Lua, utilizando a biblioteca OMPL.

Os testes foram realizados em um computador equipado com um processador I7 4790k, 11gB de memória RAM e uma GPU Nvidia GTX-980ti.

Todos os arquivos gerados nesse trabalho podem ser encontrados em <https://github.com/SPires/RRTCon-RL>.

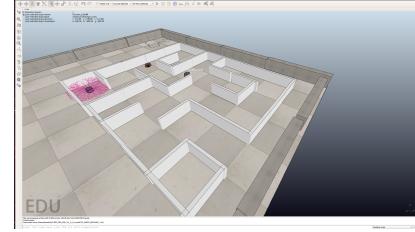
Experimentos com Aprendizado por Reforço

Para os experimentos utilizando aprendizado por reforço, foram criadas 2 cenas. A primeira cena é composta por um labirinto quadrangular com uma saída, como pode ser observado na Figura 4. Inicialmente, utilizou-se essa cena, dotada de um labirinto mais complexo, e com 2 outros agentes (robôs) que se moviam dentro do labirinto.

O objetivo era que o robô fosse capaz de encontrar a saída, sem colidir com o labirinto ou com os outros robôs. Porém, a rede não foi capaz de treinar um modelo que conseguisse executar essa tarefa.

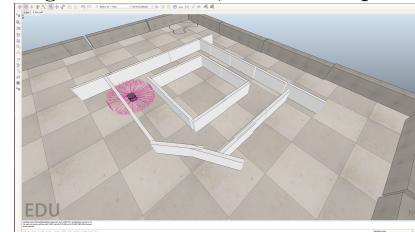
Para tentar contornar esse problema utilizou-se uma cena mais simples, com um labirinto quadrangular fechado (figura 5). O objetivo do agente, nesse cenário,

Figure 4: Cena1, mais complexa.



era conseguir dar uma volta no labirinto sem colidir com nenhuma parede.

Figure 5: Cena 2, menos complexa



O agente utilizado para o treinamento foi um modelo virtual do robô Pioneer. O Pioneer possui 16 sensores de proximidade, cujos sinais foram utilizados como dados entrada para a Q-Net.

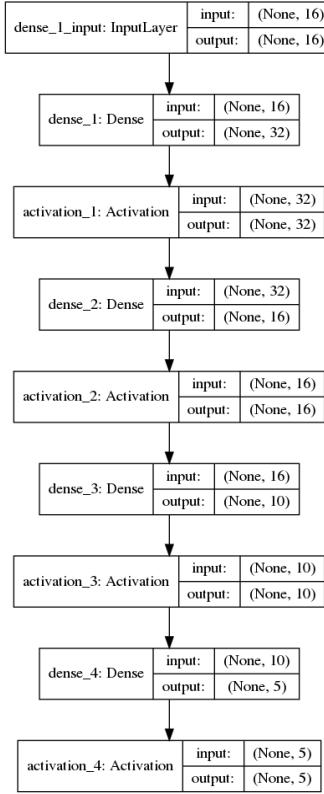
Para construir a rede neural, utilizou-se o Keras. O Keras é uma API de alto nível que tem como motor o Tensorflow e é utilizada na implementação de redes neurais. Mais informações podem ser encontradas em <https://keras.io/>. O Tensorflow é uma biblioteca *open-source*, disponibilizada pelo Google e possui aplicabilidade na solução de diversos problemas, incluindo aprendizado por reforço. Mais informações em <https://www.tensorflow.org/>. O código fonte utilizado para o treinamento do agente foi adaptado de: <https://github.com/akashdeepjassal/VREP-RL-bot>.

A Q-net mapeia os valores retornados pelos 16 sensores do Pioneer e retorna uma matriz de probabilidades com formato 1x4 (matriz de ações). Cada coluna dessa matriz de saída representa uma de 4 ações possíveis que podem ser executadas pelo agente, são elas: virar para a esquerda, virar para a direita, parar, seguir em frente e marcha à ré. Nessa matriz é então aplicado uma função *argmax*(matriz de ações), responsável por selecionar a próxima ação tomada pelo robô. A esquematização da Q-net pode ser encontrada na Figura 6.

Experimentos com Planejamento

A fim de manter a consistência do trabalho, a primeira tentativa de execução do planejamento foi feita utilizando o robô Pioneer. Como os testes não obtiveram resultados considerando o robô como um elemento estático ou dinâmico, foi adotada uma simplificação do problema, onde o robô passou a ser representado por

Figure 6: Figura mostrando o esquema da rede Q-net utilizada. A rede é do tipo *Feedforward*, possui 3 camadas *fully connected*, utiliza ReLUs como função de ativação, Adam como otimizador e entropia cruzada como função de custo.



um bloco. Uma explicação plausível para o ocorrido é o uso inadequado de algum elemento da biblioteca OMPL para a definição do problema de planejamento.

Esses experimentos serão feitos tendo em vista três situações. A primeira situação será considerando o labirinto apresentado na cena da Figura 4. A segunda situação será uma leve adaptação da cena da Figura 5, visto que para fazer o “robô” dar uma volta no labirinto é preciso impedir que ele alcance o final do trajeto de uma forma mais simples. A terceira situação será um novo labirinto, o qual é apresentado na Figura 7.

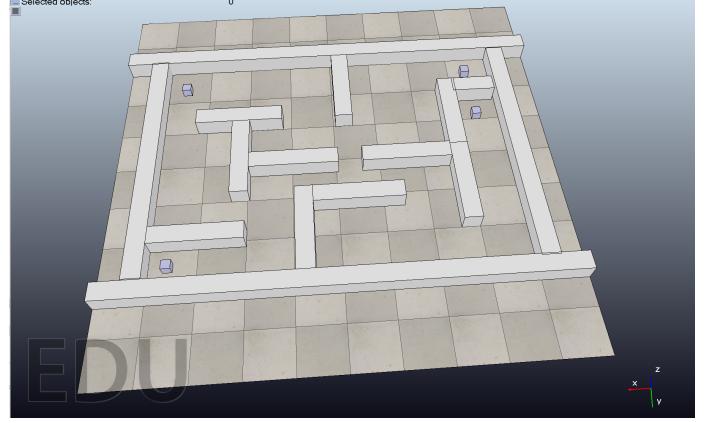
Desses experimentos foram contabilizados o total de colisões e o tempo total da execução (desde a construção do caminho até o bloco atingir a configuração final).

Resultados

Experimentos com Aprendizado por Reforço

Os experimentos foram realizados, inicialmente, no ambiente da Figura 4. Os hiper-parâmetros utilizados para o treinamento foram os seguintes: $\gamma = 0.99$, $\epsilon = 0.1$, Desconto de $\epsilon = 0.001$, $\text{Épocas} = 1500$, Passos por época = 300. Inicialmente percebe-se que a exploração

Figure 7: Cenário de *path planning* com mais de um elemento móvel.



feita pelo agente é completamente aleatória, porém ao longo que o número de interações com o ambiente aumenta, a aleatoriedade das decisões diminui.

O treinamento durou aproximadamente 4 dias e apesar de o robô parecer ser capaz de evitar colisões, o mesmo não conseguiu sair do labirinto e também não foi capaz de se afastar muito de sua localização inicial. Vale ressaltar que o agente apenas conseguia evitar colisões, com as paredes do labirinto e/ou com o outro Pioneer, se as mesmas ocorressem perto de sua localização inicial. Assim que o robô saía dessa área, seus movimentos pareciam aleatórios o que resultava em uma colisão (Figura 8).

Figure 8: Posição final após primeiro treinamento.



Como o problema pareceu ser causado pela complexidade da cena, uma nova cena, mais simples, foi configurada (Figura 5). Levando em consideração que ao final do primeiro treinamento o robô pareceu ter aprendido algo, os pesos do primeiro treinamento foram reutilizados, como configuração inicial, nessa nova cena. Os hiper-parâmetros foram configurados da seguinte forma: $\gamma = 0.99$, $\epsilon = 0.1$, Desconto de $\epsilon = -0.001$, $\text{Épocas} = 100$, Passos por época = 500.

Este treinamento durou um total de 7 horas e, no fim, apresentou resultados parecidos com o primeiro experimento: o agente era apenas capaz de não colidir com os obstáculos (paredes do labirinto), se estivesse próximo

ao ponto de partida. Se o mesmo, por ventura, se afastasse de sua posição inicial, colidia com alguma parede e não era mais capaz de prosseguir (Figura 9).

Figure 9: Posição final após segundo treinamento.



Outras tentativas foram feitas e o resumo dos hiperparâmetros utilizados pode ser vista na Tabela 1.

Table 1: Resultados dos experimentos com planejamento.

Treinamento	Cena	Tempo	Épocas	Passos por época	γ	ϵ	Desconto de ϵ
3	Cena 2	9h	150	300	0.99	0.1	-0.001
4	Cena 2	9h	150	300	0.99	0.3	-0.001
5	Cena 2	9h	150	300	0.99	0.2	-0.001
6	Cena 2	9h	150	300	0.99	0.2	-0.01
7	Cena 2	9h	150	300	0.99	0.2	-0.02

Na tentativa 8, uma nova estratégia foi utilizada. O valor que era descontado de ϵ , a partir da época 70, mudava de Desconto de $\epsilon = -0.001$ para Desconto de $\epsilon = -0.002$, e o número de Passos por época, aumentava gradativamente até 500, durante o treinamento.

Essas estratégias foram utilizadas, primeiro, tentando aumentar mais rapidamente a taxa com que o agente aprendia sobre o ambiente. E segundo, tentando fazer com que o robô tivesse menos tempo para explorar o mundo, no início do treinamento, diminuindo assim o tempo em que o mesmo explora o mundo aleatoriamente.

Contudo, o comportamento observado no primeiro e segundo treinamentos (Figuras 8 e 9), apesar de levemente atenuados (o agente conseguia avançar um pouco mais no ambiente), ainda persistiram.

Todavia, apesar do *Q-Learning* não ter funcionado corretamente junto *framework* utilizado, sabe-se pela literatura que esta técnica funciona. Diversas implementações da mesma podem ser encontradas, em diferentes bibliotecas, e em implementações que variam desde simples às mais complexas.

É notória também, a dificuldade de se re-implementar pesquisas já feitas, mesmo com o código fonte disponível, de modo que os exemplos oferecidos, em sua maioria são muito básicos, o que dificulta usá-los como ponto de partida para uma implementação mais robusta.

Portanto, os resultados obtidos podem acabar sendo influenciados por alguma funcionalidade não óbvia do framework, ou até mesmo uma incompatibilidade não documentada das bibliotecas usadas.

Experimentos com Planejamento

As execuções realizadas no V-Rep para as cenas apresentadas nas Figuras 4, 5 e 7 produziu os resultados apresentados na Tabela 2. As duas primeiras colunas referentes a Cena 1 estão preenchidas com um “X” pois a simulação apresentou um erro que não foi possível corrigir. Mesmo marcando as paredes como objetos colidíveis, o algoritmo de planning movia os blocos da origem para o destino, mas atravessando as paredes. Não foi possível identificar o problema.

Table 2: Resultados dos experimentos com planejamento.

Cena	Tempo	Colisões	Objetos
Cena 1	X	X	2
Cena 2	45s	0	1
Cena 3	46s	4	3

É de suma importância ressaltar que algumas colisões apresentadas obviamente teriam efeitos maiores em casos onde o bloco utilizado para representar o robô fosse de fato um robô. Como o ideal seria utilizar um robô Pioneer, uma colisão seria capaz de impulsioná-lo para fora da rota originalmente traçada no ponto de partida. Os resultados também evidenciam a necessidade de um tratamento mais cuidadoso quando há comportamento dinâmico no ambiente. Quando há mais de um robô tentando chegar a um objetivo em comum, eventualmente eles irão colidir, pois a amostragem feita previamente não representa mais a realidade do ambiente.

A criação dos ambientes, apesar de simplificada pela interface *drag-and-drop* da aplicação, se torna complexa uma vez que há uma série de pequenos detalhes que devem ser atribuídos aos elementos de cena por meio de menus que não estão facilmente acessíveis, por exemplo, elementos estáticos que são naturalmente criados como dinâmicos.

Como o algoritmo de planejamento utilizado (RRT-Connect) é baseado em amostras, diferentes execuções podem gerar diferentes caminhos e, consequentemente, um diferente número de colisões.

O uso do OMPL reduz o trabalho de programar as melhores estruturas de dados e tratar funções básicas, como a geração aleatória da amostra. Todavia, leigos não são capazes de aproveitar com propriedade as funcionalidades da ferramenta em sua totalidade, uma vez que os exemplos oferecidos não são intuitivos e o manual não provê informações necessárias a compreensão dos parâmetros das funções.

Conclusão

Do ponto de vista ferramental, fica evidente o quanto os sistemas de software podem oferecer funcionalidades importantes para a pesquisa e desenvolvimento no campo da robótica. A própria criação da OMPL demonstra o interesse na criação de ambientes que possuam portabilidade para atender aos mais variados tipos de usuários.

Apesar de o trabalho não ter atingido um resultado comparável proposto, foi extremamente útil para fins didáticos ao nos colocar em contato com ferramentas que possuem ampla utilização acadêmica e industrial.

Ao comparar as técnicas de planejamento e aprendizado por reforço, pudemos perceber não só as vantagens e desvantagens destacadas na literatura, mas também as dificuldades operacionais para a implementação dessas ideias em *frameworks* específicos.

References

- [Iram Noreen2016] Iram Noreen, Amna Khan, Z. H. 2016. A comparison of rrt, rrt* and rrt*-smart path planning algorithms. *International Journal of Computer Science and Network Security* 16:20–27.
- [J. Kuner, Steven, and Lavalle2003] J. Kuner, J.; Steven, J.; and Lavalle, M. 2003. Rrt-connect: An efficient approach to single-query path planning.
- [Kaelbling, Littman, and Moore1996] Kaelbling, L. P.; Littman, M. L.; and Moore, A. P. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.
- [Şucan, Moll, and Kavraki2012] Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine* 19(4):72–82. <http://ompl.kavrakilab.org>.
- [Sutton and Barto1998] Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. MIT Press.