Trabalho Prático 1
Algoritmos de Busca no Pac-Man

Arthur Souto Lima - 2018055113

[1]Departamento de Ciência da Computação – Instituto de Ciências Exatas
Universidade Federal de Minas Gerais (UFMG)
Av. Pres. Antônio Carlos, 6627 - Pampulha, Belo Horizonte - MG, 31270-901

arthursl@ufmg.br

1. Introdução

O Pac-Man é um icônico expoente dos video-games, concebido na década de 80 para os arcades pela Namco. Nele, o jogador controla o personagem Pac-Man em diversas fases, nas quais ele deve comer todos os pontos do labirinto, enquanto evita os fantasmas, os inimigos. Há alguns pontos maiores que, se comidos, permitem que o jogador consiga comer os fantasmas para conseguir mais pontos.

Com essa mecânica simples de entender, um dos principais problemas a serem resolvidos pelo jogador é obter o caminho "ideal" para conseguir alcançar todos os pontos da fase, não necessariamente o mais curto, mas o que permite atingir seus objetivos. Assim, a meta deste trabalho prático é desenvolver algoritmos de busca que permitam encontrar esse caminho utilizando estratégias e heurísticas diferentes, além de comparar seu desempenho em vários casos.

A base de implementação é o a implementação do jogo Pac-Man em Python de um projeto do curso de Introdução à IA da Universidade de Berkley [1]. O trabalho consiste em modificar o arquivo *search.py* e implementar os seguintes algoritmos de busca: Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS, ou Algoritmo de Dijkstra), Busca Gulosa (Greedy Search) e A*, com duas heurísticas diferentes.

Há vários cenários de teste e labirintos disponíveis para avaliar a performance das diversas estratégias implementadas, algumas dessas situações serão compiladas aqui para fins de apresentação de experimentos e composição de análises acerca dos algoritmos e suas diferenças.

1.1. Visão Geral dos Algoritmos Implementados

1.1.1. Busca sem Informação

A ideia geral para a implementação dos algoritmos adveio do conteúdo visto em aula [Chaimowicz 2020b] [Chaimowicz 2020a], sendo uma adaptação dos algoritmos à realidade Python. No caso da DFS e BFS, a implementação das duas é muito semelhante, com a única diferença na escolha de qual nó explorar em seguida, a qual pode ser facilitada com uso de estruturas de dados específicas, respectivamente, uma pilha (LIFO) e uma fila (FIFO). Felizmente, ambas já estavam implementadas nos arquivos iniciais do projeto, bastou usá-las no código.

[1]http://ai.berkeley.edu/search.html

Uma outra questão a ser solucionada nesses dois casos era de como recuperar o caminho até o gol. Foi utilizada uma estratégia de manter um ”mapa do nó pai”, obtida de uma pergunta no StackOverflow [amit 2020]. Assim, ao encontrar (e expandir) o gol, configurando atingi-lo segundo a especificação, poderíamos seguir o mapa a partir desse nó, qual foi seu o nó-pa, qual foi o pai deste, e assim por diante, para recuperarmos o caminho obtido.

No UCS, o loop básico de expansão dos nós se mantém, porém agora usando uma outra estrutura de dados, também já implementada no projeto fornecido, o heap (fila de prioridades), seguindo o algoritmo visto em aula [Chaimowicz 2020b]. Contudo, dada a evolução do algoritmo, dever-se-ia encontrar uma outra solução para obter o caminho ideal para o gol. Nos casos anteriores, poderíamos manter simplesmente o mapa de ”qual nó é pai de quem” no caminho. Mas no caso do Algoritmo de Dijkstra, como pode ser o caso de explorarmos potenciais caminhos diferentes a cada vez, esse mapa não funcionava mais como antes.

A estratégia então adotada foi guardar no heap um par *(nó, caminho)* e seu custo, para composição da ordenação interna da estrutura. Assim, a cada iteração, se o nó não tivesse sido investigado ainda, ele era colocado junto com o caminho até ele até agora obtido. Poderia haver o caso também de o nó já estar no heap, mas com um caminho pior (mais caro), assim, deveríamos atualizá-lo para o caminho mais barato atual. A última situação era de o nó já estar na estrutura e o caminho atual ser pior do que o armazenado, ou seja, podemos ignorar essa linha de investigação e manter a estrutura como está.

Seguindo esse método, ao encontrarmos (expandirmos) o nó gol, o caminho até ele advinha junto, pois era guardado no heap também. Assim, não há mais a sem necessidade de backtracing posterior, como era o caso da DFS e da BFS implementadas.

1.1.2. Busca com Informação

Ainda seguindo os algoritmos fornecidos em aula [Chaimowicz 2020a], implementou-se a Busca Gulosa e o A*. A busca gulosa foi mais simples visto que usaremos a partir de agora sempre a estratégia de guardar o caminho no heap junto com o custo, que advinha de uma heurística fornecida, sendo que bastava chamar uma função para obter o valor para o nó atual. Outra facilidade foi o fato de que o método *update* da fila de prioridades funcionava como um *push* caso o nó não estivesse lá ainda, reduzindo um *if* na implementação.

O algoritmo A* foi implementado, tal como vimos em aula [Chaimowicz 2020a], amalgamando a Busca Gulosa com a UCS. Manteremos o caminho junto com o nó na Fila de Prioridades e o custo usado para manter a ordenação é composto tanto pelo custo real (UCS) quanto pelo custo de uma heurística (Greedy Search). Novamente, temos que considerar quando um potencial novo caminho é melhor ou pior do que o que já está armazenado e fazer eventuais modificações quando necessário.

Como pedido pela especificação, no passo 7, há de se implementar uma heurística para o algoritmo A*. Ela será apresentada e discutida na Seção 2.

2. Metodologia e Análise Experimental

2.1. Metodologia

Para comparar os algoritmos, vamos avaliar o tempo de busca, a quantidade de nós expandidos, a otimalidade da solução, o custo dela e a pontuação final obtida. Todos esses dados nos são fornecidos com a execução do programa-base, o que facilita a execução dos testes. Esses serão compilados em tabelas no Google Sheets para ulterior apresentação neste relatório seja sob forma de tabela ou de gráficos. Para evitar imprecisões por conta do acaso, para aferição dos tempos de execução, executa-se 5 vezes e toma-se a média de tempo dos valores obtidos.

A máquina utilizada nesses testes é uma munida de um processador de 6 núcleos, executando em ambiente Windows na plataforma WSL (Windows Subsystem for Linux) Ubuntu 20.04. Em termos da linguagem Python, usaremos o interpretador padrão, na sua versão 3.8.10.

Serão quatro cenários de comparação, o primeiro de uma busca simples, no *mediumMaze*, representando um cenário mais genérico. O segundo será também de busca simples, mas agora no *openMaze*, para observar como se comportam os algoritmos num ambiente com menos paredes. Já no contexto do *foodSearchProblem*, veremos o desempenho dos algoritmos no *smallSearch* e no *trickySearch*.

Nos dois primerios cenários, compararemos: DFS, BFS, UCS, Busca Gulosa (com *nullHeuristic*), Busca Gulosa (com *manhattanHeuristic*), Busca Gulosa (com *euclideanHeuristic*), A* (com *manhattanHeuristic*) e A* (com *euclideanHeuristic*). Nos dois últimos, os algoritmos testados são: UCS, Busca Gulosa (com *nullHeuristic*) e A* (com a heurística proposta).

2.2. Resultados

2.2.1. Busca Simples

Os resultados dos testes dos dois primeiros cenários podem ser observados nas tabelas 1 e 2. Considerou-se o caminho ótimo o caminho encontrado pela BFS (ou um que tenha o mesmo custo, se for o caso).

Tabela 1. Resultados no *mediumMaze*

Algoritmo	Tempo (s)	Nós expandidos	Otimalidade	Custo	Pontuação
DFS	0	146	Não	130	380
BFS	0	269	Sim	68	442
UCS	0	269	Sim	68	442
Greedy (nullHeuristic)	0	269	Sim	68	442
Greedy (manhattanHeuristic)	0	78	Não	74	436
Greedy (euclideanHeuristic)	0	159	Não	152	358
A* (manhattanHeuristic)	0	221	Sim	68	442
A* (euclideanHeuristic)	0	226	Sim	68	442

Fonte: o autor

Tabela 2. Resultados no *openMaze*

Algoritmo	Tempo (s)	Nós expandidos	Otimalidade	Custo	Pontuação
DFS	0	576	Não	298	212
BFS	0	682	Sim	54	456
UCS	0	682	Sim	54	456
Greedy (nullHeuristic)	0	682	Sim	54	456
Greedy (manhattanHeuristic)	0	89	Não	68	442
Greedy (euclideanHeuristic)	0	54	Sim*	54	456
A* (manhattanHeuristic)	0	682	Sim	54	456
A* (euclideanHeuristic)	0	54	Sim*	54	456

Fonte: o autor

Tabela 3. Resultados no *smallSearch*

Algoritmo	Tempo (s)	Nós expandidos	Custo	Pontuação
UCS	279,6	70726	34	636
Greedy (nullHeuristic)	1053,6	70726	34	636
A* (heurística própria)	2	4975	34	636

Fonte: o autor

2.2.2. *FoodSearchProblem*

No caso das buscas no contexto do *FoodSearchProblem*, os resultados estão nas tabelas 3 e 4. Nessas situações, decidiu-se não anotar o quesito ”Otimalidade”, visto que ele não é muito conveniente aqui, além de ser pouco claro o que ele representa. Nesse momento, quer-se comparar o desempenho em termos de nós expandidos, basicamente.

2.3. Discussão

2.3.1. Busca Simples

Nos dois primeiros cenários, o tempo gasto não foi grande o suficiente para que o registro feito pelo próprio programa capturasse as diferenças. Nem mesmo no cenário *bigMaze* esse tempo era diferente de zero para esses algoritmos. Olhando a quantidade de nós expandidos, vê-se que, à exceção da Busca Gulosa com a distância de Manhattan, todos os algoritmos expandiram da ordem de centenas de nós, o que leva a tempos semelhantes.

No *mediumMaze*, observando os nós expandidos, como já inicialmente comentado, os algoritmos BFS, UCS e Greedy Search (nullHeuristic) são os que mais expandem. Isso é esperado do BFS, que de fato faz uma exploração muito grande em cada nível dos caminhos. A DFS e a Busca Gulosa com as duas outras heurísticas procuraram em menos nós, a custo de encontrarem uma solução sub-ótima, isto é, com custo maior, o que diminuiu a pontuação do agente. Esse resultado vai de encontro com o exposto em [Chaimowicz 2020b] e [Chaimowicz 2020a].

Ainda nesse labirinto, o A* ficou no meio-termo entre nós expandidos e foi capaz de encontrar a solução ótima. Isso pode ser explicado, como vimos em aula [Chaimowicz 2020a], pelo fato de que ele combina os princípios da UCS com os da Busca Gulosa, no intuito de usufruir dos benefícios de cada um: a otimalidade do algoritmo de

Tabela 4. Resultados no *trickySearch*

Algoritmo	Tempo (s)	Nós expandidos	Custo	Pontuação
UCS	10,7	16688	60	570
Greedy (nullHeuristic)	20,4	16688	60	570
A* (heurística própria)	3,6	7203	60	570

Fonte: o autor

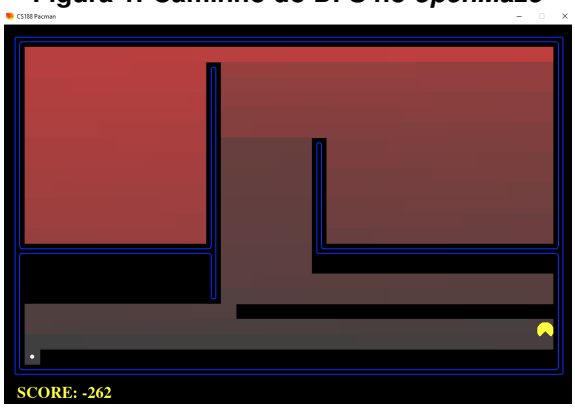
Dijkstra nesse ambiente e na eficiência de exploração da Busca Gulosa. Desse modo, o A* se mostra o candidato do custo-benefício, o que corrobora seu grande uso em IA's de pathfinding.

No caso do *openMaze*, as conclusões são as mesmas quanto aos nós expandidos. Mas convém ressaltar um detalhe da tabela. Na metodologia, adotou-se como o caminho ótimo o caminho obtido pela BFS. Contudo, no *openMaze*, os algoritmos que usaram a distância euclidiana encontraram um outro caminho, com exatamente o mesmo custo (e, portanto, pontuação). Assim, tanto a Busca Gulosa quanto o A* com essa heurística atingiram a otimalidade, mas foram assinaladas com um "*" para denotar esse outro caminho ótimo obtido.

Um último detalhe nesse âmbito é a diferença entre as heurísticas com distância de Manhattan e distância euclidiana. No *mediumMaze*, a distância euclidiana explorou mais nós. O efeito contrário acontece no *openMaze*, possivelmente por conta do fato de que, nesse segundo labirinto, pela falta de abundantes paredes, o movimento do agente é mais livre, o que permite um caminho mais próximo da distância em linha reta, computada pela distância euclidiana.

Esse outro labirinto, o *openMaze*, também expõe a falta que pode fazer uma garantia de otimalidade da solução. Enquanto os outros algoritmos obtiveram um caminho que seguia diretamente para a comida, o DFS encontrou um caminho que dava muitas voltas, como podemos ver na Figura 1. Isso penalizou o desempenho do algoritmo com o mais alto custo da solução e, por isso, menor pontuação dentre os analisados para o *openMaze*.

Figura 1. Caminho do DFS no *openMaze*



2.3.2. *FoodSearchProblem*

A escolha das fases para os testes do *FoodSearchProblem* se mostrou um desafio. Como é possível observar nos tempos experimentais, mesmo num dos menores labirintos, o *smallSearch*, o UCS demorava pouco mais de 4 minutos e o Greedy Search quase 20. Assim, decidiu-se analisar apenas esses três algoritmos, sem o DFS e o BFS, que eram demasiado ineficientes na expansão dos nós e, portanto, demorariam muito mais. Outro detalhe é que as heurísticas já implementadas da distância de Manhattan e distância euclidiana não funcionam com a interface da classe *FoodSearchProblem*, visto que ela não define um ponto de gol, como é o caso da busca simples. Por isso, usamos apenas a *nullHeuristic* para a Busca Gulosa e a heurística proposta para o A* nessas comparações.

Nos dois labirintos desse contexto, os algoritmos sempre encontraram o mesmo caminho ou um equivalente com o mesmo custo e pontuação. A grande diferença é na quantidade de nós expandidos e, consequentemente, no tempo gasto para tanto, portanto os atributos que serão o foco dessa análise.

Gráfico 2. Tempos no *smallSearch*

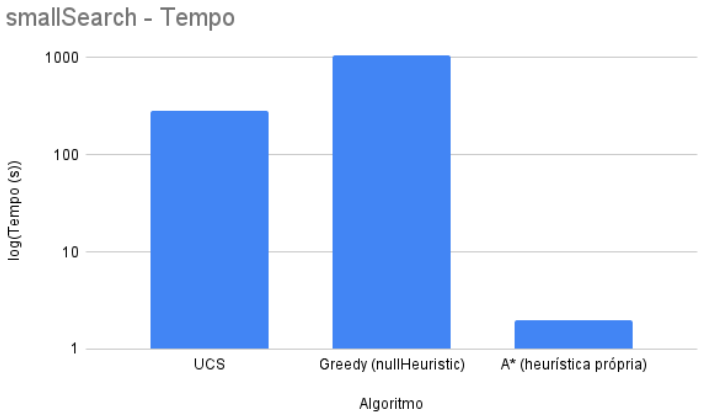
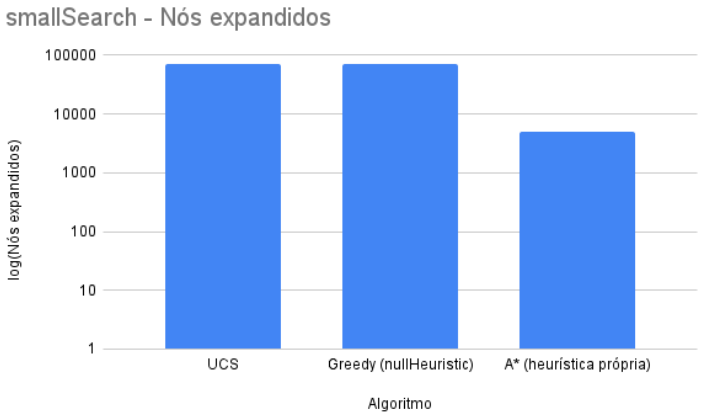


Gráfico 3. Nós expandidos no *smallSearch*



Compilados nos gráficos 2 e 3 para melhor contrastar os resultados, há uma diferença significativa entre o A* e os outros. Em termos de tempo o A* leva poucos

segundos enquanto os outros levam minutos. Em nós expandidos, o A* não chega a 5 mil, ao passo que os outros dois exploram pouco mais de 70 mil nós na busca. A diferença é tão abismal que foi necessário utilizar a escala logarítmica no eixo y para melhor visualização nos gráficos 2 e 3.

Gráfico 4. Tempos no *trickySearch*

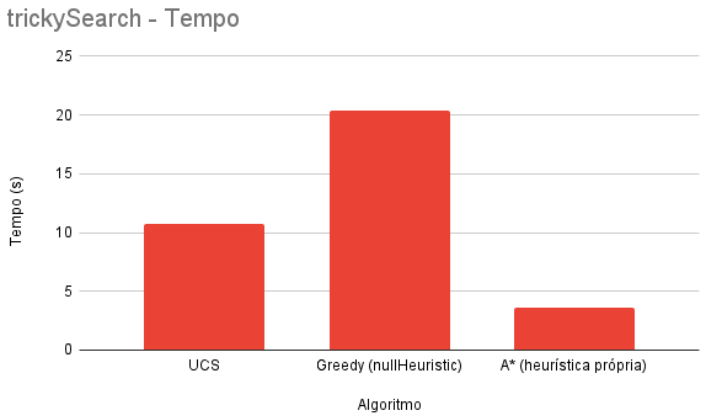
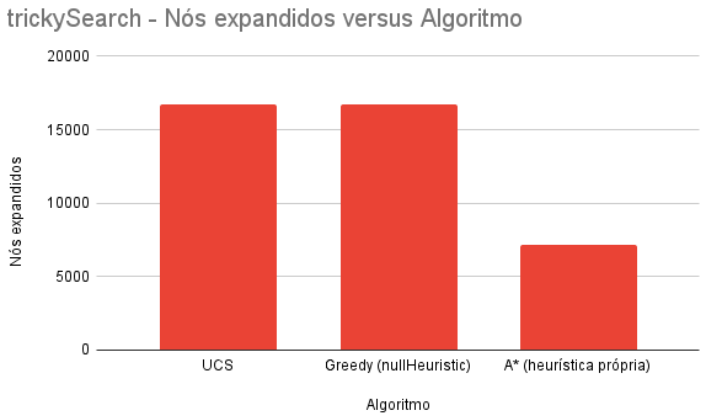


Gráfico 5. Nós expandidos no *trickySearch*



Os resultados no labirinto *trickySearch* corroboram a análise feita anteriormente, ou seja, o A* usando a heurística proposta explora menos nós e leva menos tempo que os outros dois. Contudo, a diferença não é tão dilatada quanto no labirinto anterior. Novamente, esses resultados ratificam a posição do A* como um dos algoritmos preferidos para tarefas de busca, devido a seu bom desempenho nas diversas situações.

2.4. Heurística do A*

Conceber uma heurística que atendesse aos requisitos de nós expandidos propostos na especificação e que ainda fosse admissível e consistente foi certamente uma das tarefas mais difíceis deste trabalho. Seguindo as sugestões obtidas em aula [Chaimowicz 2020a] e no livro [Russell and Norvig 2020], inicialmente tentou-se buscar heurísticas advindas de relaxamentos do problema. Uma primeira ideia inocente foi usar uma medida de distância (como a de Manhattan) até a comida mais próxima, mas a expansão dos nós

ainda era muito grande, cerca de 14 mil, melhor que a UCS simples, mas ainda bastante longe dos objetivos da especificação.

Em discussões com os colegas, levantou-se a ideia de penalizar considerando quantas comidas ainda restavam no labirinto. Incrivelmente, essa heurística sozinha foi considerada admissível e consistente pelo *autograder* e expandia cerca de 12 mil nós. Uma outra ideia foi computar a distância efetiva entre o agente e as comidas, considerando as paredes. Isso pode ser feito via uma BFS, já que temos acesso a essas informações do labirinto, partindo do agente em direção à comida em questão.

A heurística adotada conjuga essas três ideias:

1. Compute a distância de Manhattan para todas as comidas restantes
2. Escolha a mais próxima do agente
3. Compute a distância real (considerando as paredes)
4. Penalize essa distância real em 1 unidade para cada comida que não esteja na mesma coluna ou não esteja na mesma linha seja do agente seja da comida mais próxima

2.4.1. Admissibilidade e Consistência

Para argumentar que a heurística proposta é consistente, seguiremos o exposto no livro [Russell and Norvig 2020], que apresenta o conceito de heurística consistente ligado à desigualdade triangular. Diz-se que, para um estado n, a heurística para ele deve seguir, considerando o estado sucessor n' obtido através da ação a:

$$h(n) \leq c(n', a, n) + h(n')$$

O custo da heurística $h(n)$ de um certo estado é determinado por distância de Manhattan e, em seguida, pela distância efetiva do labirinto. Esses valores são positivos e admissíveis, pois não são maiores que a distância real que o agente deve percorrer até o gol (normalmente a comida a ser absorvida nessa etapa). Ainda nesse custo da heurística, é adicionada uma penalidade positiva para cada comida que não esteja na mesma linha ou na mesma coluna seja do agente seja da comida mais próxima.

O custo da ação $c(n', a, n)$ a ser tomada é basicamente a distância a ser percorrida, a qual foi levada em conta para o custo heurístico do estado atual. Tendo em vista que o custo heurístico do estado seguinte $h(n')$ é positivo, pela mesma argumentação que assim caracteriza o do estado atual $h(n)$, a desigualdade está atendida, denotando uma heurística consistente.

Ainda segundo [Russell and Norvig 2020], toda heurística consistente também é admissível. Logo, a heurística proposta também é admissível.

Finalmente, não mais como argumentos, mas como indícios de que, de fato, a heurística é consistente, pode-se citar alguns eventos. O primeiro está relacionado à situação de que, segundo a especificação, uma heurística que resolve a fase *mediumSearch* (com o parâmetro *AStarFoodSearchAgent*, naturalmente) rápido tende a ser inconsistente. A heurística proposta não consegue resolver essa fase de forma rápida. Para se ter uma

ideia, a *trickySearch* é resolvida em poucos segundos, enquanto a *mediumSearch* ficou minutos e não foi solucionada.

O outro fato relevante é que o *autograder* não repugnou a proposta, isto é, não a acusou de ser inadmissível ou de ser inconsistente. Isso ocorreu com outras heurísticas que foram concebidas durante a execução do trabalho, mas por serem objetadas por esse módulo, foram descartadas.

3. Conclusão

Neste trabalho, a tarefa era implementar diferentes algoritmos de busca para guiar um agente no contexto do jogo Pac Man. Como aprendido durante as aulas, são várias as estratégias adotadas por esses algoritmos, com suas vantagens e desvantagens, incluindo otimalidade da solução e quantidade de nós expandidos. Foram implementados os algoritmos DFS, BFS, UCS, Busca Gulosa e A*, sendo esses últimos com heurísticas diferentes para comparação. Além disso, foi proposta uma heurística consistente e admissível para o algoritmo A*.

Após comparação e análise do desempenho dos algoritmos, vê-se que o algoritmo A* é o que apresenta melhor custo-benefício entre quantidade de nós expandidos e otimalidade da solução. Há certamente aqueles que exploram menos nós, como é o caso, por vezes, da Busca Gulosa e do DFS, mas que não garantem a otimalidade. Por outro lado, também há aqueles que garantem esse atributo, mas a estratégia de busca é bastante ineficiente, como o BFS.

Essa atividade foi de farta importância para a consolidação do conteúdo visto em aula, sobretudo acerca da busca no contexto de IA. Implementar os algoritmos permitiu compreender as semelhanças e as diferenças entre eles, ademais de solidificar o entendimento acerca de seus respectivos funcionamentos, incluindo os vários detalhes.

Além disso, possibilitou a interação com elementos práticos que modelam não apenas o algoritmo em si, mas também suas estruturas auxiliares como os nós, o grafo e os caminhos, os quais eram abstratamente definidos e usados durante os exemplos e exercícios. Ao implementá-los, há de se conhecer e buscar soluções de como representá-los de forma eficiente e simples para uso no desenrolar do programa.

Em termos de percalços ao longo da realização, convém citar dois momentos. O primeiro foi a interação inicial com o sistema, de forma a compreender como usar as interfaces e as estruturas já implementadas e fornecidas. Felizmente, como o código era muito bem documentado e comentado, essa etapa foi prontamente vencida.

Outra dificuldade adveio na ocasião da concepção da heurística, que, além de atingir o objetivo de nós expandidos, deveria ser admissível e consistente. Esse trabalho mostrou como pode ser árduo criar uma heurística, ainda mais com características bem delimitadas. Esse obstáculo também foi vencido graças às sugestões obtidas nas aulas, nos materiais [Chaimowicz 2020a] e no livro [Russell and Norvig 2020], além das discussões com os demais colegas. Essa dificuldade trouxe bons frutos também, visto que, sobretudo na oportunidade de argumentar por sua admissibilidade e consistência esses conceitos foram consolidados.

Uma vantagem do sistema fornecido foi o *autograder*. Primeiro, pois ele permitiu um fluxo de trabalho da implementação aos moldes do TDD (Test-Driven Development),

permitindo incrementos graduais na concepção das soluções e rápida verificação de corretude. Além disso, ele dá mais segurança à nossa implementação, visto que temos um feedback, ainda que não necessariamente definitivo, de nossos algoritmos em vários casos de teste antes de entregar o trabalho.

Finalmente, esse trabalho foi bastante agradável de se realizar: havia um contexto, um problema claro, uma excelente especificação guiando e delimitando os objetivos, um sistema estável e consistente, pronto para receber e avaliar o desenvolvimento da tarefa. E, além disso, era possível ver o resultado dos algoritmos visualmente num ambiente conhecido: os labirintos do Pac Man, o que cativa ainda mais a continuar desenvolvendo a atividade.

References

amit (2020). Tracing and returning a path in depth first search. [2]. Acesso em: 09 dez 2021.

Chaimowicz, L. (2020a). Busca em espaço de estados: Busca com informação. Acesso em: 09 dez 2021.

Chaimowicz, L. (2020b). Busca em espaço de estados: Busca sem informação. Acesso em: 09 dez 2021.

Russell, S. J. and Norvig, P. (2020). Solving problems by searching. In *Artificial Intelligence: A Modern Approach*, chapter 3, pages 81–127. Pearson.

[2] `https://stackoverflow.com/a/12864196/`