Trabalho Prático 1: Busca em Mapas

Bernardo de Almeida Abreu[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Disciplina: DCC028 - Inteligência Artificial 2018/1

bernardoabreu@dcc.ufmg.br

1. Introdução

Esse trabalho possui como objetivo analisar diferentes algoritmos de busca em espaços de estados. Os algoritmos analisados são:

- Busca de aprofundamento iterativa (IDS)
- Busca de custo uniforme (UCS)
- Busca gulosa de melhor escolha (*Best First Search*)
- Busca A*

Os mapas nos quais os algoritmos de busca serão executados são bidimensionais, representados por matrizes de caracteres. As células marcadas com um "." estão livres e células marcadas com um "@" estão bloqueadas.

2. Implementação

Os algoritmos de busca foram implementados utilizando a linguagem *Python 3*. Foi criada uma classe *GraphSearch* para implementar o algoritmo de busca em grafo básico, apresentado no algoritmo 1. Cada algoritmo a ser implementado no trabalho, herda da classe *GraphSearch* e implementa as funções necessárias para executar o algoritmo de busca.

Algorithm 1 Função de busca em grafo

```
1:  function GRAPH_SEARCH(problem)
2:      node ← problem.INITIAL_STATE
3:      frontier ← INIT_FRONTIER()
4:      frontier ← INSERT(frontier, node)
5:      explored ← ∅
6:      loop
7:          if EMPTY?(frontier) then
8:              return FAILURE
9:          frontier, node ← REMOVE(frontier)
10:         if GOAL?(problem, node) then
11:             return SOLUTION(node)
12:         explored ← explored ∪ {node}
13:         for all action ∈ GET_ACTIONS(problem, node) do
14:             child ← CHILD_NODE(problem, node, action)
15:             frontier ← UPDATE_FRONTIER(child, frontier, explored)
```

A função *GRAPH_SEARCH* implementa o comportamento básico de todas os algoritmos de busca. As particularidades da implementação de cada algoritmo se referem ao modo como a fronteira será gerenciada em cada algoritmo. Algumas das funções chamadas nesse algoritmo são as seguintes:

INIT_FRONTIER Inicializa a fronteira com a estrutura apropriada para o algoritmo a ser utilizado.
GOAL? Verifica se o nó corresponde ao objetivo.
GET_ACTIONS Retorna uma lista de ações válidas para o nó.
CHILD_NODE Cria um novo nó a partir de uma ação.
UPDATE_FRONTIER Atualiza a fronteira de acordo com o tratamento necessário para o algoritmo.

2.1. Estruturas

2.1.1. Conjunto de estados explorados

O algoritmo implementado é o de busca em grafos, dessa forma é preciso manter um conjuntos de nós já visitados para que eles não sejam repetidos. Esse conjunto é representado pelo conjunto *explorado*. Uma vez que cada estado do problema é representado por uma posição no mapa, ou seja, um par de coordenadas (X,Y), é possível criar uma tabela *hash* com acesso O(1) para guardar quais estados foram visitados. Essa tabela é implementada como uma matriz booleana, onde *true* indica que o estado já foi visitado e *false* o contrário.

2.1.2. Classe *Problem*

Foi implementada uma classe *Problem* para representar o problema. Essa classe guarda as informações sobre o estado inicial, o estado objetivo e o mapa onde a busca ocorre. Além disso, ela possui funções para retornar as ações válidas a partir de um estado, para gerar um novo estado a partir de uma ação e para testar se um estado é a solução.

2.1.3. Classe *Node*

Um nó de busca também é implementado através de uma classe. Essa classe guarda informações sobre o estado do nó, o nó que gerou esse estado, o custo total do caminho encontrado até o momento e a ação que levou a esse estado. A partir dessa classe é possível reconstruir o caminho encontrado para a solução.

2.2. Fronteira

Foram criadas duas classes para a fronteira. A primeira classe, *Frontier*, implementa a fronteira utilizando uma pilha ou uma lista, secionável por parâmetro. Existem funções para inserir e retirar elementos da fronteira, verificar se a mesma está vazia e verificar se um certo elemento se encontra nela. Além disso, existe uma função para substituir um nó presente na fronteira por um nó com o mesmo estado caso o novo possua um custo de caminho menor. Para verificar se um elemento já se encontra na fronteira, foi utilizada a classe *ExploredSet*, implementando essa verificação em O(1).

A segunda classe é a *PriorityFrontier*. Ela reimplementa as funções de inserir, remover e substituir para utilizar uma fila de prioridades. As funções de inserir e substituir agora recebem uma tupla de (prioridade, nó).

2.3. Busca de aprofundamento iterativa - IDS

A classe *IDS* implementa a busca de custo uniforme, e herda da classe *GraphSearch*. Essa nova classe reimplementa os métodos de inicializar o conjunto de explorados, inicializar a fronteira, atualizar a fronteira e executar. Além disso ela apresenta novos campos de variáveis. Esses novos membros são um variável do tipo *float* indicando o limite de profundidade e uma variável booleana indicando se esse limite foi atingido.

O método de execução é alterado para chamar o método de busca diversas vezes, aumentado iterativamente o limite de profundidade para a nova execução, até uma solução ser encontrada ou o grafo inteiro for percorrido. A profundidade considerada para esse algoritmo foi o custo do caminho, e não a aresta do grafo. Dessa forma o limite é aumentado de 0.5 em 0.5, e não de 1 em 1.

A fronteira é inicializada com a classe *Frontier* utilizando uma pilha. O algoritmo 2 mostra o pseudocódigo para atualizar a fronteira. Caso o custo do caminho do nó seja maior do que o limite, é indicado que o limite foi atingido e mais nada é feito. Caso o estado do nó não esteja presente na fronteira ou no conjunto de explorados, o mesmo é adicionado a fronteira.

A terceira condição da função *UPDATE_FRONTIER*, indicada na linha 6, existe pois esse algoritmo consiste em uma busca em profundidade com um conjunto de nós visitados. Dessa forma, um nó pode ser visitado primeiro por um caminho mais longo, e depois por um caminho mais curto, o que deve ser levado em consideração pelo algoritmo para se obter o caminho correto. Portanto, o conjunto de explorados utilizado nesse algoritmo é um pouco diferente do tradicional, a classe *ExploredCosts* foi criada, herdando de de *ExploredSet*, de forma a guardar o custo dos caminhos explorados ao invés de um valor booleano. Assim, na terceira condição é verificado se o nó já foi explorado com um caminho maior do que o atual, para que se possa repetir a busca caso o novo caminho seja melhor.

Algorithm 2 Atualiza a fronteira do IDS

1: **function** UPDATE_FRONTIER($child, frontier, explored, limit$)
2: **if** $child$.PATH_COST $> limit$ **then**
3: $limit\_reached \leftarrow true$
4: **else if** $child$.STATE $\notin frontier$ **and** $child$.STATE $\notin explored$ **then**
5: $frontier \leftarrow$ INSERT($frontier, child$)
6: **else if** GET_COST($explored, child$.STATE) $> child$.PATH_COST **then**
7: $frontier \leftarrow$ INSERT($frontier, child$)

2.4. Busca de custo uniforme - UCS

A classe *UCS* implementa a busca de custo uniforme, e herda da classe *GraphSearch*. Essa nova classe reimplementa os métodos de inicializar a fronteira e atualizar a fronteira.

A fronteira é inicializada com uma *PriorityFrontier*. O algoritmo 3 mostra o pseudocódigo para atualizar a fronteira. Caso um nó com o mesmo estado do novo nó já esteja presente na fronteira, o novo substitui o antigo caso o segundo tenha um custo de caminho maior. Caso o novo nó não esteja na fronteira ou no conjunto de explorados, ele é inserido na fronteira.

Algorithm 3 Atualiza a fronteira do UCS

```
1: function UPDATE_FRONTIER(child, frontier, explored)
2:     if child.STATE ∈ frontier then
3:         frontier ← REPLACE(frontier, (child.PATH_COST, child))
4:     else if child.STATE ∉ explored then
5:         frontier ← INSERT(frontier, (child.PATH_COST, child))
```

2.5. Algoritmos de busca com informação

Para os algoritmos de busca com informação, a busca gulosa e A*, foi implementada uma classe *HeuristicSearch* que herda de *GraphSearch*. Essa classe adiciona a possibilidade de se escolher uma heurística e inicializa a fronteira com a classe *PriorityFrontier*.

2.5.1. Best first Search

A classe *BestFirstSearch* implementa a busca gulosa, e herda da classe *HeuristicSearch*. Essa nova classe reimplementa o método de atualizar a fronteira. Nesse método, um novo nó é inserido na fronteira caso ele já não se encontre na fronteira ou no conjunto de estados explorados. O valor utilizado como prioridade na fila de prioridades é o valor da heurística.

2.5.2. A*

A classe *AStar* implementa a busca A*, e herda da classe *HeuristicSearch*. Essa nova classe reimplementa o método de atualizar a fronteira. Esse método é igual ao método de atualização apresentado no algoritmo 3, porém o valor de prioridade é a soma do valor da heurística com o custo do caminho, ao invés de somente o custo do caminho.

2.5.3. Heurísticas

Duas heurísticas possíveis foram implementadas, a *distância Manhattan* (1) e a *distância octile* (2). Entre essas heurísticas, a única admissível e consistente é a *distância octile*.

$$\begin{aligned} dx &= abs(node.x - goal.x) \\ dy &= abs(node.y - goal.y) \\ h(n) &= (dx + dy) \end{aligned} \tag{1}$$

$$\begin{aligned} dx &= abs(node.x - goal.x) \\ dy &= abs(node.y - goal.y) \\ h(n) &= max(dx, dy) + 0.5 \times min(dx, dy) \end{aligned} \tag{2}$$

A *distância Manhattan* não é capaz de diferenciar o custo de se movimentar na diagonal dos outros custos, assim, o valor de custo retornado pela heurística para se andar para um quadrado diretamente na diagonal é 2, enquanto o valor real é 1.5. Para uma heurística ser admissível, é necessário que seu valor seja menor ou igual ao valor real, o que não acontece nesse caso.

A *distância octile* leva em consideração os custos das diagonais. O número de passos diagonais em um caminho livre é igual a $min(dx, dy)$. O termo $(0.5 \times min(dx, dy))$ pode ser reescrito como $(1.5 \times min(dx, dy) -$

$1 \times min(dx, dy)$), que corresponde ao números de passos na diagonal que se deve dar, multiplicado pelo custo da diagonal, menos o número de passos que não se está dando na direção de $max(dx, dy)$, por se estar usando a diagonal.

Com um caminho livre, o custo dado para se caminha na horizontal ou vertical é igual ao custo ótimo, uma vez que o valor de min(dx,dy) é 0, e o valor de max(dx,dy) é o próprio valor da distância. Ao se caminhar para o quadrado exatamente na diagonal, o custo total será 1.5, que corresponde à distância ótima.

Uma heurística é consistente se, para um nó *n* e os seus sucessores *n'* gerados por uma ação *a*, o custo estimado de atingir o objetivo a partir de *n* não é maior que o custo de chegar a *n'* somado ao custo estimado de *n'* para o objetivo [Russell and Norvig 2010]. O custo estimado de se atingir um nó com a heurística de *distância octile* corresponde ao custo ótimo possível dentro de um mapa com caminho livre. Logo o custo real não é menor do que o custo estimado.

3. Experimentos

Para realizar o experimento foram gerados 100 pontos válidos para cada mapa. Os pontos foram então ordenados em função do custo dado pelo algoritmo A* com heurística *octile*. Dessa forma, foi possível executar o algoritmo IDS nos pontos mais próximos, algo necessário devido ao alto tempo de execução, como será mostrado mais a frente. Será feita uma comparação entre cada um dos algoritmos, utilizando somente a heurística *octile* para o A*. Em seguida, a *distância octile* é comparada com a *distância Manhattan* no A*.

3.1. Custo total

Os pontos avaliados são mostrados na tabela 1 para cada mapa.

É possível observar que o custo dos algoritmos A* com *distância octile*, UCS e IDS são iguais, uma vez que esses algoritmos são ótimos. Já o custo do algoritmo guloso é, na maioria das vezes, maior do que o ótimo.

Foram selecionados alguns outros pontos para se comparar as heurísticas para o algoritmo A*, para demonstrar a otimalidade do algoritmo. Esses pontos são mostrados na tabela 5. A heurística de *distância octile* é admissível e consistente, portanto a sua utilização torna o algoritmo ótimo. Já a heurística da *distância Manhattan* não é admissível e consistente, de forma que o resultado obtido nem sempre é ótimo, como se pode observar.

3.2. Estados expandidos

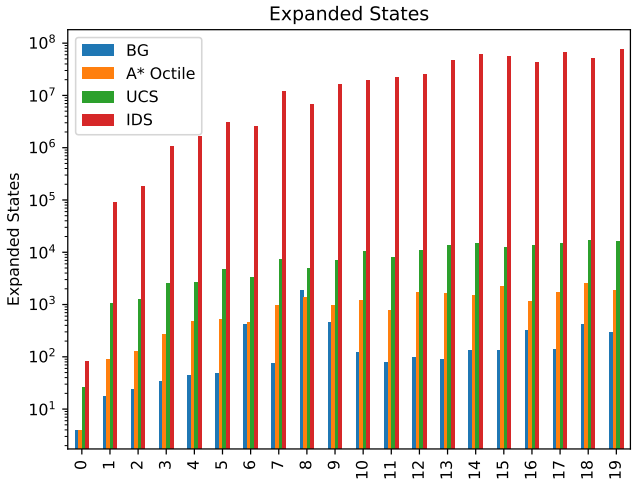

Figure 1. Estados expandidos para o mapa 1

A quantidade de estados expandidos no algoritmo IDS é muito grande, come é possível notar nas figuras 1, 2 e 3, assim como nas tabelas 9, 10 e 11. Isso se deve em parte ao fato de o algoritmo executar por diversas iterações, repetindo a expansão de vários estados.

O algoritmo de *Best first Search* expande o menor número de estados, devido à sua caraterística gulosa. Isso faz com que o algoritmo muitas vezes não expanda os estados pertencentes à solução ótima. O algoritmo

Table 1. Custos totais para cada algoritmo

Table 2. Mapa 1

		Total Cost			
Initial Point	Final Point	A* Octile	BG	UCS	IDS
186,111	186,108	3.0	3.0	3.0	3.0
179,98	170,115	21.5	21.5	21.5	21.5
235,107	229,84	26.0	26.0	26.0	26.0
231,93	211,126	43.0	43.0	43.0	43.0
19,44	35,1	51.0	51.0	51.0	51.0
206,178	221,130	55.5	55.5	55.5	55.5
151,240	163,194	57.0	70.5	57.0	57.0
155,182	224,150	85.0	90.0	85.0	85.0
81,233	59,192	86.5	88.5	86.5	86.5
119,61	180,61	92.0	104.5	92.0	92.0
89,208	93,129	106.0	106.0	106.0	106.0
245,107	181,184	109.0	109.0	109.0	109.0
92,218	184,181	110.5	111.5	110.5	110.5
74,96	158,39	115.5	115.5	115.5	115.5
149,201	68,130	128.0	138.0	128.0	128.0
121,45	72,142	129.5	131.5	129.5	129.5
68,194	173,246	131.0	139.0	131.0	131.0
92,96	68,194	135.0	135.5	135.0	135.0
138,133	26,160	140.5	150.0	140.5	140.5
110,58	230,52	143.0	155.5	143.0	143.0

Table 3. Mapa 2

		Total Cost			
Initial Point	Final Point	A* Octile	BG	UCS	IDS
128,199	129,189	11.5	11.5	11.5	11.5
102,158	95,125	36.5	36.5	36.5	36.5
231,93	211,126	47.5	47.5	47.5	47.5
130,11	83,27	55.0	59.0	55.0	55.0
251,26	239,79	64.0	65.5	64.0	64.0
88,254	66,218	77.5	80.5	77.5	77.5
27,217	87,227	86.0	106.5	86.0	86.0
167,173	232,224	93.0	94.0	93.0	93.0
12,228	97,201	98.5	98.5	98.5	98.5
143,218	52,203	98.5	98.5	98.5	98.5
109,199	12,208	101.5	105.0	101.5	101.5
92,218	184,181	115.0	117.0	115.0	115.0
174,14	47,13	127.5	127.5	127.5	127.5
210,14	245,126	129.5	131.5	129.5	129.5
111,225	209,247	136.5	160.0	136.5	136.5
58,215	197,210	146.5	147.5	146.5	146.5
174,14	53,61	154.5	238.0	154.5	154.5
175,196	25,217	162.0	178.5	162.0	162.0
48,107	147,205	167.0	168.0	167.0	167.0
85,17	164,127	169.0	489.5	169.0	169.0

Table 4. Mapa 3

		Total Cost			
Initial Point	Final Point	A* Octile	BG	UCS	IDS
186,111	186,111	3.0	3.0	3.0	3.0
15,68	15,68	42.0	43.0	42.0	42.0
211,187	211,187	53.5	54.5	53.5	53.5
194,216	194,216	70.5	115.5	70.5	70.5
97,165	97,165	96.5	112.0	96.5	96.5
109,199	109,199	101.5	101.5	101.5	101.5
58,18	58,18	104.0	128.0	104.0	104.0
105,38	105,38	114.5	126.0	114.5	114.5
129,252	129,252	116.0	128.0	116.0	116.0
138,133	138,133	125.5	142.0	125.5	125.5
92,218	92,218	129.0	142.0	129.0	129.0
121,45	121,45	134.0	187.0	134.0	134.0
161,18	161,18	144.0	144.0	144.0	144.0
212,163	212,163	145.5	152.5	145.5	145.5
85,17	85,17	150.0	153.0	150.0	150.0
144,100	144,100	154.5	187.5	154.5	154.5
91,170	91,170	166.0	225.5	166.0	166.0
79,119	79,119	168.0	188.0	168.0	168.0
193,73	193,73	171.5	174.5	171.5	171.5
108,44	108,44	172.5	330.5	172.5	172.5

A* com *distância octile* expande o menor número de estados entre os algoritmos ótimos. Já o UCS, expande um número grande comparado com o A*, por não ser uma busca com informação. Dessa forma ele precisa buscar muito mais estados pelo objetivo, enquanto o A* possui um certo direcionamento.

O número de estados expandidos pelo A* com *distância Manhattan* é menor do que o A* com *distância octile*, como pode ser visto nas figuras 4, 5 e 6. Dessa forma, o A* com *distância Manhattan* pode não expandir estados que fazem parte do caminho ótimo. Isso é mais uma comprovação de que essa heurística não é admissível e consistente, uma vez que uma heurística consistente expande todos os estados cujo custo é menor do que o ótimo.

Table 5. Custos totais para cada heurísticas do A*

Table 6. Mapa 1

		Total Cost	
Initial Point	Final Point	A* Manhattan	A* Octile
58,112	205,70	198.0	184.0
91,170	144,41	197.0	192.0
163,247	215,93	225.0	217.0
45,70	241,106	262.5	253.0
33,218	26,98	259.5	254.5
212,2	69,180	262.5	259.5
189,12	113,184	267.5	264.5
57,238	4,11	334.5	329.5

Table 7. Mapa 2

		Total Cost	
Initial Point	Final Point	A* Manhattan	A* Octile
92,218	184,181	115.5	115.0
213,79	104,12	157.0	153.0
251,25	92,13	166.5	166.0
236,125	65,162	269.0	267.5
200,249	147,16	325.5	320.5
225,229	32,55	325.0	322.5
249,134	34,130	372.5	371.0
244,213	54,68	384.0	380.5

Table 8. Mapa 3

		Total Costs	
Initial Point	Final Point	A* Manhattan	Expanded States
97,165	47,236	99.5	96.5
153,229	69,195	104.5	102.5
58,18	149,22	105.5	104.0
107,145	38,205	124.0	115.5
138,133	26,160	126.5	125.5
212,163	135,232	146.0	145.5
85,17	164,127	151.0	150.0
30,93	152,2	200.5	196.5

Table 9. Estados expandidos para cada algoritmo no mapa 1

				Expanded States		
Initial Point	Final Point	A* Octile	A* Manhattan	BG	UCS	IDS
186,111	186,108	4	4	4	26	83
179,98	170,115	90	18	18	1069	92736
235,107	229,84	126	24	24	1284	186447
231,93	211,126	277	34	34	2543	1066778
19,44	35,1	476	44	44	2707	1693766
206,178	221,130	538	49	49	4749	3073754
151,240	163,194	469	291	429	3395	2565796
155,182	224,150	974	105	77	7497	12227326
81,233	59,192	1392	1403	1901	4887	6899864
119,61	180,61	965	774	462	7028	16341646
89,208	93,129	1228	872	123	10350	19834773
245,107	181,184	797	78	78	8060	22677851
92,218	184,181	1743	102	100	11006	25162759
74,96	158,39	1641	91	91	13814	47312849
149,201	68,130	1504	232	136	15283	60832569
121,45	72,142	2273	750	132	12804	55797562
68,194	173,246	1147	211	331	13756	44103719
92,96	68,194	1694	1598	139	14904	67165513
138,133	26,160	2570	1579	427	16956	51952503
110,58	230,52	1868	1685	297	15997	78211492

3.3. Tempo de execução

É possível observar que o algoritmo *Best first Search* é o algoritmo mais rápido dentre os examinados. Isso acontece pois ele é um algoritmo guloso, e expande o menor número de estados. O A* é o segundo mais rápido, sendo consideravelmente mais rápido que o UCS. A diferença entre essas três velocidades pode ser vista claramente nas imagens 7, 8 e 9. O tempo do UCS, o mais lento entre os três, ainda é muito mais rápido do que o tempo do algoritmo IDS, como é observado na tabela 12, 13 e 14. Essas velocidades são um reflexo do número de estados

Table 10. Estados expandidos para cada algoritmo no mapa 2

		Expanded States				
Initial Point	Final Point	A* Octile	A* Manhattan	BG	UCS	IDS
128,199	129,189	23	17	13	297	6426
102,158	95,125	216	34	34	1282	414877
231,93	211,126	194	47	42	1375	529645
130,11	83,27	434	81	94	3467	2180651
251,26	239,79	325	277	76	1968	2007591
88,254	66,218	1077	929	214	3270	4156214
27,217	87,227	1836	1456	340	4148	10917703
167,173	232,224	848	74	72	8398	15029554
12,228	97,201	1434	86	86	3799	10722782
143,218	52,203	929	92	92	10390	23697540
109,199	12,208	838	175	105	10408	22711387
92,218	184,181	2240	487	113	11309	34295156
174,14	47,13	254	128	128	9451	78926624
210,14	245,126	832	167	177	8451	35478455
111,225	209,247	3377	2241	884	14368	68924917
58,215	197,210	890	331	153	11325	71565466
174,14	53,61	3345	2650	2463	12635	78926624
175,196	25,217	2341	1478	282	16032	1.27E+08
48,107	147,205	1667	171	139	12078	84551870
85,17	164,127	3649	2172	10527	13974	1.1E+08

Table 11. Estados expandidos para cada algoritmo no mapa 3

		Expanded States				
Initial Point	Final Point	A* Octile	A* Manhattan	BG	UCS	IDS
186,111	186,111	4	4	4	27	82
15,68	15,68	230	45	39	1861	602532
211,187	211,187	264	52	45	3134	1459635
194,216	194,216	1371	1070	742	6031	5963349
97,165	97,165	609	290	316	12109	22316958
109,199	109,199	176	98	98	13998	33204967
58,18	58,18	893	960	326	9843	23784138
105,38	105,38	1228	245	129	15398	49647688
129,252	129,252	1307	223	632	11183	23684339
138,133	138,133	1576	301	532	15049	32222058
92,218	92,218	3738	2614	1392	17071	75312598
121,45	121,45	3682	2047	1144	19495	85341346
161,18	161,18	1724	107	107	17049	64638883
212,163	212,163	2016	1473	170	11209	53804824
85,17	85,17	1220	135	3290	17488	91230642
144,100	144,100	2622	2473	3263	18289	71820660
91,170	91,170	2465	1123	433	25180	1.7E+08
79,119	79,119	5166	4155	2824	25758	1.55E+08
193,73	193,73	3809	1856	1289	15678	1.14E+08
108,44	108,44	7312	6069	4138	25905	2.23E+08

expandidos por cada algoritmo.

Uma comparação melhor entre o tempo de execução dos algoritmos de busca com informação pode ser feita nas imagens 10, 11 e 12. O algoritmo guloso se mostra mais rápido que o A* em quase todos os exemplos testados. Isso é esperado, devido a sua característica gulosa. O A* utilizando *distância Manhattan* é mais rápido do que utilizando *distância octile* na maioria das vezes. Porém, existem algumas ocasiões em que a segunda heurística é mais rápida, esses casos porém não representam uma melhora muito significativa.

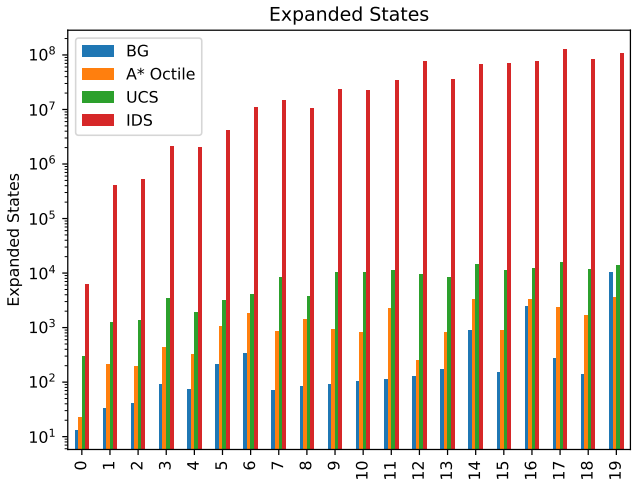

Figure 2. Estados expandidos para o mapa 2

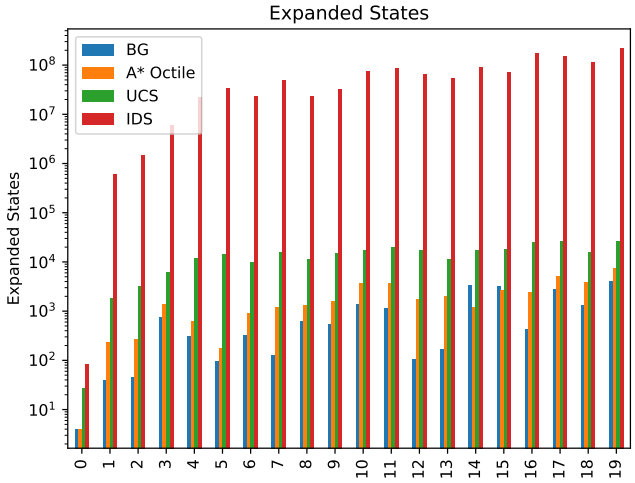

Figure 3. Estados expandidos para o mapa 3

4. Conclusão

Entre os algoritmos analisados, foi possível perceber claramente uma superioridade entre os algoritmos de busca com informação nos aspectos de tempo e estados expandidos. Isso demonstra a clara vantagem de se possuir algum tipo de conhecimento sobre o espaço em que se está fazendo a busca. Também foi possível observar como a otimalidade do algoritmo A* depende da heurística utilizada. Com uma heurística apropriada, o A* se mostra o algoritmo superior.

Os algoritmos de busca sem informação tiveram um desempenho pior em número de estados expandidos e tempo de execução. O UCS se manteve relativamente próximo ao desempenho dos outros algoritmos, porém o desempenho do algoritmo IDS foi drasticamente inferior até mesmo ao UCS. Seu tempo de execução tornou inviável o teste para distâncias muito grandes. A complexidade de tempo de um DFS normal é $O(b^m)$, onde m é o tamanho máximo de um caminho, o que comparado a complexidade do UCS, $O(b^d)$, onde d é a profundidade da solução mais rasa, é pior. O IDS ainda deve executar um número de iterações igual à profundidade da solução, ou no caso da implementação nesse trabalho, ao custo da solução. Essas diversas iterações tornam o IDS ainda mais lento. Sua vantagem está somente na melhor complexidade de espaço.

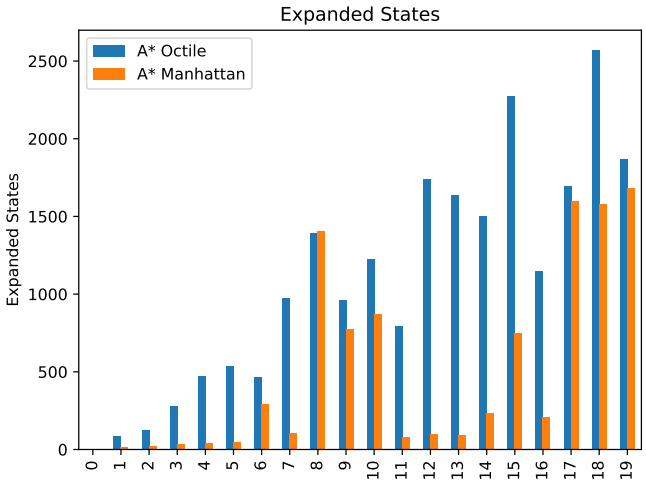

Figure 4. Estados expandidos para o mapa 1

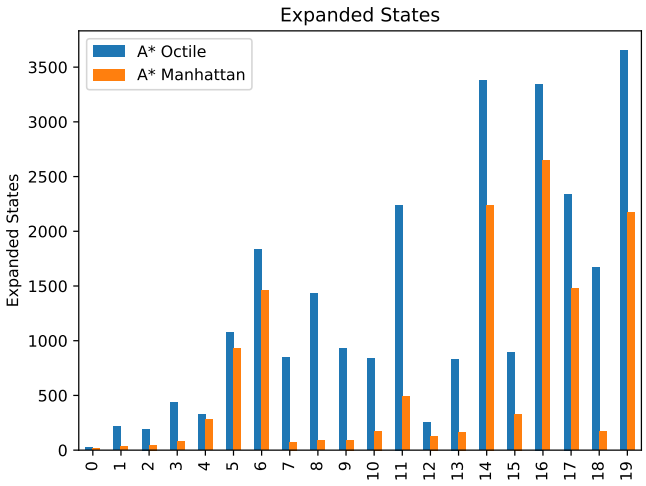

Figure 5. Estados expandidos para o mapa 2

5. Bibliografia

References

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

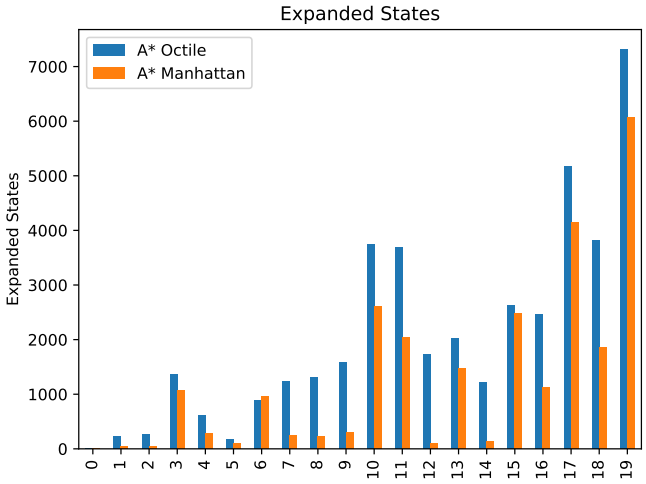

Figure 6. Estados expandidos para o mapa 3

Table 12. Tempo de execução para cada algoritmo no mapa 1

		Execution Time (s)				
Initial Point	Final Point	A* Octile	A* Manhattan	BG	UCS	IDS
186,111	186,108	0.00022912	0.000215	0.000176907	0.00123167	0.003684998
179,98	170,115	0.005465031	0.001057	0.000766039	0.077837229	3.380535364
235,107	229,84	0.007772446	0.002243	0.001454353	0.091156006	6.778869152
231,93	211,126	0.01644659	0.002351	0.001851559	0.190565109	38.18325949
19,44	35,1	0.03013587	0.004137	0.002331257	0.192735195	62.67043543
206,178	221,130	0.034475803	0.004678	0.002551317	0.448458672	107.2497153
151,240	163,194	0.031044483	0.019964	0.017571211	0.254716396	91.81169033
155,182	224,150	0.067073822	0.007468	0.003612995	0.692895889	443.6037984
81,233	59,192	0.105770111	0.107919	0.069516897	0.363559008	249.6795256
119,61	180,61	0.076903105	0.061347	0.017851114	0.605227947	585.268539
89,208	93,129	0.109166622	0.073444	0.004621267	1.109109163	692.6941526
245,107	181,184	0.052963495	0.005388	0.003852844	0.663764	841.5725067
92,218	184,181	0.129849911	0.009158	0.00447011	1.154006481	878.0102181
74,96	158,39	0.153810024	0.008676	0.00438118	1.587245464	1729.284992
149,201	68,130	0.153713703	0.016836	0.005400658	1.734092474	2235.185043
121,45	72,142	0.222425222	0.106038	0.005651712	1.325466871	2068.670755
68,194	173,246	0.080482721	0.016682	0.012129784	1.434723854	1547.722389
92,96	68,194	0.17039609	0.174343	0.00539732	1.61458993	2482.593381
138,133	26,160	0.257503986	0.166004	0.016361713	2.384456396	1867.119252
110,58	230,52	0.233627081	0.201377	0.010693312	1.806649923	2832.220277

Table 13. Tempo de execução para cada algoritmo no mapa 2

		Execution Time (s)				
Initial Point	Final Point	A* Octile	A* Manhattan	BG	UCS	IDS
128,199	129,189	0.001057386	0.000772	0.00045	0.017038584	0.216660261
102,158	95,125	0.012551069	0.00332	0.001963	0.071397781	14.47288847
231,93	211,126	0.010560751	0.002771	0.002047	0.074293137	17.86253524
130,11	83,27	0.026477814	0.005358	0.003844	0.272413731	76.84994078
251,26	239,79	0.020436764	0.020624	0.003156	0.105737209	70.23968339
88,254	66,218	0.073720932	0.070755	0.008171	0.206737757	149.5253656
27,217	87,227	0.157820463	0.155375	0.012994	0.271953821	411.1455958
167,173	232,224	0.06012392	0.005354	0.003436	0.806951284	540.720645
12,228	97,201	0.105855942	0.009255	0.003945	0.220592737	391.0644498
143,218	52,203	0.071871042	0.012527	0.004191	1.066217661	865.2234561
109,199	12,208	0.053186655	0.017768	0.004434	1.052721739	820.2954609
92,218	184,181	0.232818365	0.059996	0.004777	1.091066122	1245.447398
174,14	47,13	0.031002998	0.023714	0.005718	0.760714054	2905.215796
210,14	245,126	0.045234919	0.01556	0.006346	0.618350744	1292.072212
111,225	209,247	0.455786228	0.303797	0.031269	1.450247288	2514.61786
58,215	197,210	0.073244095	0.035646	0.006349	0.919107437	2623.47488
174,14	53,61	0.455956459	0.409636	0.090393	1.055133343	2905.215796
175,196	25,217	0.187964201	0.350165	0.011611	1.511779785	4833.390216
48,107	147,205	0.121622562	0.012377	0.005502	1.013915062	3041.408662
85,17	164,127	0.316787958	0.375743	0.383448	1.15297246	3970.978921

Table 14. Tempo de execução para cada algoritmo no mapa 3

		Execution Time (s)				
Initial Point	Final Point	A* Octile	A* Manhattan	BG	UCS	IDS
186,111	186,111	0.000219	0.000214	0.000183	0.001164	0.00343
15,68	15,68	0.013657	0.003328	0.002026	0.124224	20.15578
211,187	211,187	0.014713	0.003211	0.00242	0.252593	48.50437
194,216	194,216	0.132944	0.104225	0.027695	0.551317	202.4842
97,165	97,165	0.047384	0.027515	0.012239	1.420377	782.2352
109,199	109,199	0.01736	0.014132	0.004478	1.705806	1206.474
58,18	58,18	0.062685	0.0832	0.022181	0.906925	873.4582
105,38	105,38	0.084614	0.025141	0.005295	1.851654	1722.161
129,252	129,252	0.092695	0.021729	0.022822	1.291064	811.4382
138,133	138,133	0.114413	0.031309	0.018551	2.023453	1152.86
92,218	92,218	0.476952	0.442947	0.049721	1.95339	2601.962
121,45	121,45	0.57254	0.302336	0.044307	2.390218	2935.372
161,18	161,18	0.183484	0.007826	0.00484	2.062245	2216.831
212,163	212,163	0.171368	0.199795	0.006381	0.957157	1918.716
85,17	85,17	0.110935	0.009797	0.122043	1.886439	3152.716
144,100	144,100	0.174845	0.173401	0.116663	2.349741	2404.193
91,170	91,170	0.261915	0.134624	0.015531	3.449323	6050.029
79,119	79,119	0.515221	0.463435	0.103451	3.528434	5268.019
193,73	193,73	0.30427	0.206841	0.04659	1.475206	3991.63
108,44	108,44	1.240918	1.120419	0.148668	3.194556	7631.316

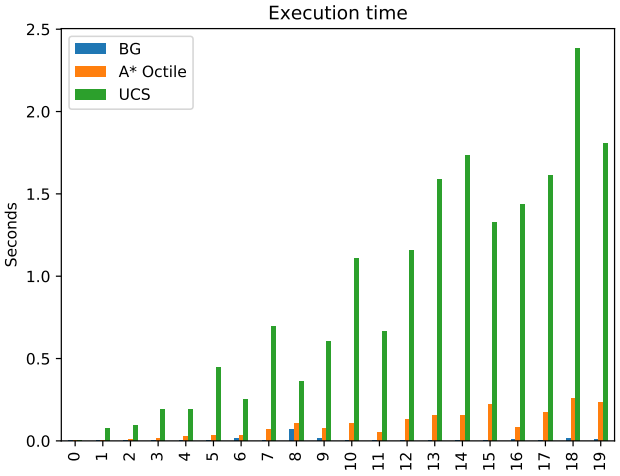

Figure 7. Tempo de execução para o mapa 1

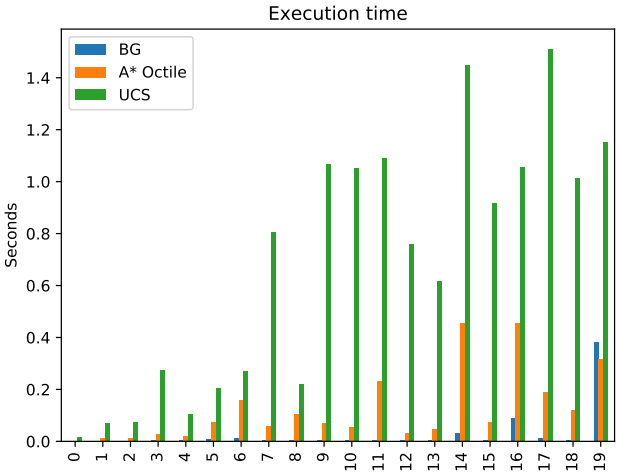

Figure 8. Tempo de execução para o mapa 2

Figure 9. Tempo de execução para o mapa 3

Figure 10. Tempo de execução para o mapa 1

Figure 11. Tempo de execução para o mapa 2

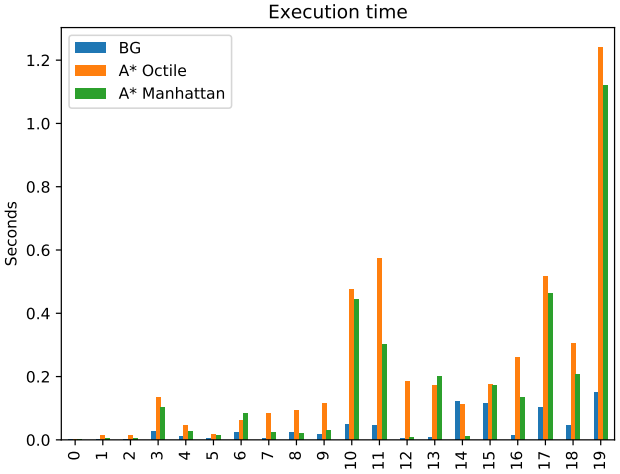

Figure 12. Tempo de execução para o mapa 3