Relatório - 2a Parte da Avaliação

Aryella Lacerda, Raul Andrade

[1] Universidade Federal de Sergipe
Departamento de Computação
Ciência da Computação

{aryella.lacerda, raul.andrade}@dcomp.ufs.br

Resumo. *Este relatório tem como objetivo mostrar o desempenho e fazer comparativos entre alguns tipos de algoritmos de buscas e algoritmos de CSP em alguns problemas, a fim de resolvê-los ou mostrar que não é possível chegar a uma solução. Os algoritmos utilizados neste relatório foram busca Gulosa (Greedy), A*, busca com Retrocesso (Backtracking), e Mínimos Conflitos (Min-Conflicts).*
Palavras-chave*: busca, problema, csp, labirinto, sudoku*

Abstract. *The objective of this paper is to show and compare the results of different search and constraint satisfaction algorithms as applied to certain problems, in order to solve them or show that it simply isn't possible to reach a solution. The algorithms discussed here are the Greedy, A*, Backtracking, and Min-Conflicts algorithms.*
Keywords*: search, csp, problem, labrinth, sudoku*

1. Busca Construtiva

1.1. Problema - Labirinto

Para testarmos os algoritmos de busca construtiva, escolhemos implementar o problema do Labirinto. As especificações do problema são as seguintes:

- **Estado inicial:** o ponto de início é demarcado pelo símbolo '@'.
- **Função sucessora:** a partir do estado atual, o estado sucessor somente poderá ser um estado que se encontra a um passo de distância do atual, na direção horizontal ou vertical (movimento na diagonal não é permitido). O estado em questão também deve ser permitido, representado por '.' ou '%' (espaços representados por '#' são paredes e portanto são restritos).
- **Teste de objetivo:** se o estado atual for representado por um '$', o objetivo foi alcançado.
- **Custo do caminho:** se o estado for representado por um '.', o custo do passo será 1. Se o estado for representado por um '%', o custo do passo será 2. O custo do caminho é o somatório do custo de todos os passos executados até o momento atual.

1.2. Análise

Implementamos 5 tipos de busca: em largura, em profundidade, de custo uniforme, utilizando técnica gulosa, e A*. Algumas buscas são variações de outras. Por exemplo, com a exceção da busca em profundidade, todas outras são variantes da busca em largura. Inclusive, a busca em profundidade e em largura diferem somente na estrutura de dados usada para guardar e ordenar os estados sucessores.

1.2.1. Instância 1: Labirinto 20 x 20

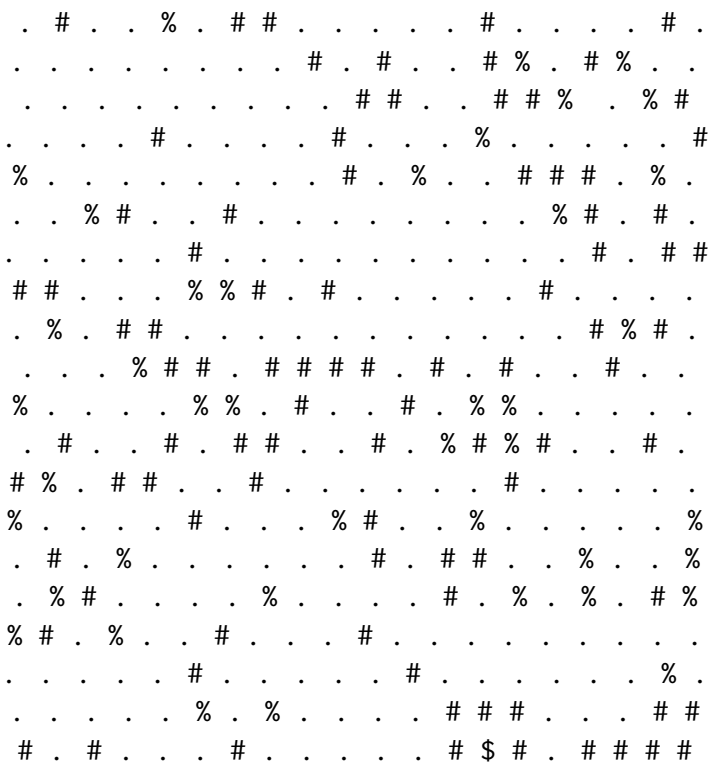
```
. # . . % . # # . . . . . # . . . . # .
. . . . . . . . . # . # . . # % . # % . .
. . . . . . . . . . # # . . # # %  . % #
. . . . # . . . . # . . . % . . . . . #
% . . . . . . . . . # . % . . # # # . % .
. . % # . . # . . . . . . . . % # . # .
. . . . . # . . . . . . . . . . # . # #
# # . . . % % # . # . . . . . # . . . .
. % . # # . . . . . . . . . . . # % # .
. . . % # # . # # # # . # . # . . # . .
% . . . . % % . # . . # . % % . . . . .
. # . . # . # # . . # . % # % # . . # .
# % . # # . . # . . . . . . # . . . . .
% . . . . # . . . % # . . % . . . . . %
. # . % . . . . . . # . # # . . % . . %
. % # . . . . % . . . . # . % . % . # %
% # . % . . # . . . # . . . . . . . . .
. . . . . # . . . . . # . . . . . . % .
. . . . . % . % . . . . . % . . . . # #
# . # . . . # . . . . . . $ . . # # # #
```

- **Largura**: ['v', '<', '<', '<', 'v', 'v', 'v', 'v', 'v', 'v', 'v', '<', 'v', 'v', 'v', '<', 'v', 'v', 'v', '>', 'v', 'v', 'v', '>']
  Total de expansões: 264
  Custo total: 27

- **Profundidade**: ['<', 'v', '<', '<', '<', '<', '<', 'v', 'v', '<', '<', '<', 'v', '<', 'v', '<', '<', '<', '<', 'v', '<', '<', 'v', 'v', '>', '>', 'v', 'v', '<', 'v', '<', 'v', 'v', 'v', 'v', 'v', '>', '>', '>', 'v', '>', '>', '^', '>', '>', 'v', '>', '>', '>', '>', '>', '>']
  Total de expansões: 54
  Custo total: 63

- **Custo uniforme**: ['>', 'v', 'v', 'v', 'v', 'v', '>', '>', 'v', 'v', '<', 'v', '<', '<', 'v', 'v', '<', 'v', 'v', 'v', 'v', 'v', 'v', 'v', '<', '<']
  Total de expansões: 257
  Custo total: 26

1.2.2. Instância 2: Labirinto 20 x 20 Saída inalcançável



- **Largura**: None
  Total de expansões: 309
  Custo total: 0

- **Profundidade**: None
  Total de expansões: 309
  Custo total: 0

- **Custo uniforme**: None
  Total de expansões: 309
  Custo total: 0

1.3. A* e Guloso

O objetivo dos algoritmos A* e Guloso é diminuir o custo de chegar ao objetivo o quanto possível, e fazer isso da maneira mais eficiente. Temos como comparativo a busca de custo uniforme, que nos deu o menor custo possível (26) em 257 expansões. Para ambos os algoritmos, utilizamos duas heurísticas, que devemos comparar:

- **Heurística 1: Distância Manhattan**

  A distância Manhattan é a que chega mais perto do custo real, pois ela calcula a soma da quantidade de passos na horizontal e na vertical que devemos tomar para chegar ao objetivo, que é tudo que nosso problema permite. Ela é admissível, logo que nunca superestima o custo real. Se o caminho entre o estado atual e o estado objetivo for composto inteiramente por espaços permitidos, a heurística

será exatamente igual ao custo real. Se o caminho for interrompido por uma parede, e for necessário escolher um caminho mais longo, a heurística será menor que o custo real.

- **Heurística 2: Distância Euclidiana**

  A distância Euclidiana, que representa distância em linha reta entre dois pontos, é um relaxamento do nosso problema. Aqui não é permitido movimento na diagonal, mas o valor calculado pela distância Euclidiana sempre será menor ou igual ao custo real. É, portanto, admissível. Porém não é uma heurística tão boa quanto a primeira, pois os casos em que ela é exatamente igual ao custo real são mais raros (somente em casos onde o estado atual e o estado objetivo se encontram na mesma linha ou coluna, e se o caminho entre eles for composto inteiramente por espaços permitidos) e a diferença entre a distância Euclidiana e o custo real em todos os outros casos geralmente é maior que a da distância Manhattan.

- **Instância 1**

  - **Guloso H1**: ['v', '<', '<', '<', 'v', 'v', 'v', 'v', 'v', 'v', 'v', '<', 'v', 'v', '>', 'v', '>', 'v', 'v', 'v', 'v', 'v', 'v', '<']
    Total de expansões: 27
    Custo total: 29

  - **Guloso H2**: ['v', '<', '<', '<', 'v', 'v', 'v', 'v', 'v', 'v', 'v', '<', 'v', 'v', 'v', '>', '>', 'v', 'v', 'v', 'v', 'v', 'v', '<']
    Total de expansões: 26
    Custo total: 29

  - **A* H1**: ['v', '>', 'v', 'v', 'v', 'v', '>', '>', 'v', 'v', 'v', 'v', 'v', '<', '<', 'v', '<', '<', 'v', 'v', 'v', 'v', 'v', 'v', '<', '<']
    Total de expansões: 104
    Custo total: 26

  - **A* H2**: ['v', '>', 'v', 'v', 'v', 'v', '>', '>', 'v', 'v', 'v', 'v', 'v', '<', 'v', 'v', '<', 'v', 'v', '<', 'v', '<', 'v', '<', 'v', '<']
    Total de expansões: 131
    Custo total: 26

- **Instância 2**

  - **Guloso H1**: None
    Total de expansões: 309
    Custo total: 0

- **Guloso H2**: None
  Total de expansões: 309
  Custo total: 0

- **A* H1**: None
  Total de expansões: 309
  Custo total: 0

- **A* H2**: None
  Total de expansões: 309
  Custo total: 0

1.4. Comparativo/Conclusão

Claramente, a busca em profundidade obteve o desempenho prior. Isso é devido às próprias propriedades da busca: ela é completa para *este problema* onde o espaço de busca é finito, mas não é ótima. Isso significa que ela chegará sim ao objetivo, se for possível alcançá-lo, mas não com custo mínimo. Veja a *instância 1*. Comparada à busca de custo uniforme, algoritmo completo e ótimo cujo custo foi 26, a busca em profundidade custou 63, uma diferença de 142%.

A busca em largura, por coincidência, chegou perto do custo mínimo (27). Ela é completa, mas não ótima para *esse problema* onde nem todo passo possui o mesmo custo. Implementamos ela por curiosidade, e também porque ela é a base dos algoritmos a seguir.

Como foi observado, o A* se sobressaiu em termos de **otimização**. Em relação ao único outro algoritmo ótimo dessa lista para esse caso, a busca de custo uniforme, o A* H1 foi bem mais eficiente: 104 expansões em relação a 257. Portanto, se o objetivo for alcançar sempre a solução ótima (e isso sempre implica uma certa perda de eficiência), a melhor opção seria A*.

Em contrapartida, na *instância 1*, a expansão de nós do A* foi muito maior que a do Guloso, que se mostrou ter mais **eficiência**. Como o próprio nome mostra, algoritmos baseados na técnica gulosa olham somente para heurística, ignorando todos os outros fatores, ou seja, são algoritmos míopes que só veem o que estão à sua frente; por isso o número de expansões é tão pouca. Não garantem, porém, uma solução ótima. Se isso for um resultado aceitável, o Guloso é a melhor opção.

A *instância 2* foi outra curiosidade nossa. Basicamente, queríamos saber como os algoritmos se comportavam em um labirinto em que a saída estava bloqueada, ou melhor dizendo, um labirinto sem saída. Note que para essa instância, todos os algoritmos fizeram o mesmo número de expansões. Isso, porque todo algoritmo teve que percorrer todos os espaços livres do labirinto (309) até descobrir que não tinha como alcançar a saída.

2. CSP

2.1. Problema - Sudoku

Para testarmos os algoritmos de CSP *(Constraint Satisfaction Problems)* Backtracking e o Mínimos Conflitos, escolhemos implementar o problema do Sudoku. As especificações do problema são as seguintes:

- **Conjunto de variáveis:** O 'tabuleiro' do Sudoku é formado por 9 linhas e 9 colunas. Logo, existem 9x9 = 81 células, e cada uma delas é uma variável.
- **Domínio dos valores:** Inicialmente é 1-9
- **Conjunto de restrições:** As variáveis da mesma linha, coluna, ou grade (3x3) não podem receber nenhum valor repetido. Ou seja, existem 27 restrições ALL-DIFF.

2.2. Análise

Alguns detalhes: no Backtracking, usamos a heurística do valor que causa menos conflito nas demais variáveis *(LCV - Least Constraining Value)*.

Abaixo temos dois Sudoku: um fácil e outro difícil, ambos com suas respectivas soluções. É importante observar que na solução/saída o **número** anterior à (seta)-> representa o `número` da solução e o número após representa o total de conflitos. Como estamos mostrando a solução, obviamente que o número de conflitos é zero. Usamos essa ideia da seta para debugar ou melhor acompanhar o desenvolvimento da solução; isso foi bastante útil para ver os dois algoritmos em ação.

2.2.1. Instância: Nível Fácil

```
0 0 9 | 2 1 8 | 0 0 6
3 0 0 | 0 4 0 | 0 0 0
6 2 8 | 0 3 0 | 4 0 0
---------------------
2 0 1 | 0 9 0 | 0 4 0
8 0 3 | 0 0 0 | 2 0 1
0 4 0 | 0 2 0 | 6 0 3
---------------------
0 0 2 | 0 5 0 | 3 7 9
0 0 0 | 0 7 0 | 0 0 8
5 0 0 | 9 8 3 | 1 0 0
```

Saída

```
4 -> 0 5 -> 0 9 -> 0 | 2 -> 0 1 -> 0 8 -> 0 | 7 -> 0 3 -> 0 6 -> 0
3 -> 0 1 -> 0 7 -> 0 | 5 -> 0 4 -> 0 6 -> 0 | 9 -> 0 8 -> 0 2 -> 0
6 -> 0 2 -> 0 8 -> 0 | 7 -> 0 3 -> 0 9 -> 0 | 4 -> 0 1 -> 0 5 -> 0
-------------------------------------------------------------------
2 -> 0 6 -> 0 1 -> 0 | 3 -> 0 9 -> 0 5 -> 0 | 8 -> 0 4 -> 0 7 -> 0
```

```
8 -> 0 9 -> 0 3 -> 0 | 4 -> 0 6 -> 0 7 -> 0 | 2 -> 0 5 -> 0 1 -> 0
7 -> 0 4 -> 0 5 -> 0 | 8 -> 0 2 -> 0 1 -> 0 | 6 -> 0 9 -> 0 3 -> 0
-----------------------------------------------------------------
1 -> 0 8 -> 0 2 -> 0 | 6 -> 0 5 -> 0 4 -> 0 | 3 -> 0 7 -> 0 9 -> 0
9 -> 0 3 -> 0 4 -> 0 | 1 -> 0 7 -> 0 2 -> 0 | 5 -> 0 6 -> 0 8 -> 0
5 -> 0 7 -> 0 6 -> 0 | 9 -> 0 8 -> 0 3 -> 0 | 1 -> 0 2 -> 0 4 -> 0
```

Expansões: 14 **Backtracks:** 0 **Iterações:** 79

2.2.2. Instância: Nível Difícil

```
1 0 7 | 0 0 0 | 9 0 5
0 0 0 | 0 8 0 | 0 0 0
0 6 0 | 2 0 0 | 0 8 0
---------------------
0 5 0 | 0 2 0 | 0 0 9
7 0 4 | 0 0 9 | 0 0 0
3 0 9 | 0 0 0 | 6 4 0
---------------------
0 0 0 | 7 0 3 | 0 0 4
0 0 8 | 5 0 0 | 0 1 0
0 0 0 | 0 0 0 | 0 0 6
```

Saída

```
1 -> 0 8 -> 0 7 -> 0 | 4 -> 0 3 -> 0 6 -> 0 | 9 -> 0 2 -> 0 5 -> 0
4 -> 0 3 -> 0 2 -> 0 | 9 -> 0 8 -> 0 5 -> 0 | 7 -> 0 6 -> 0 1 -> 0
9 -> 0 6 -> 0 5 -> 0 | 2 -> 0 7 -> 0 1 -> 0 | 4 -> 0 8 -> 0 3 -> 0
-----------------------------------------------------------------
8 -> 0 5 -> 0 6 -> 0 | 3 -> 0 2 -> 0 4 -> 0 | 1 -> 0 7 -> 0 9 -> 0
7 -> 0 2 -> 0 4 -> 0 | 6 -> 0 1 -> 0 9 -> 0 | 5 -> 0 3 -> 0 8 -> 0
3 -> 0 1 -> 0 9 -> 0 | 8 -> 0 5 -> 0 7 -> 0 | 6 -> 0 4 -> 0 2 -> 0
-----------------------------------------------------------------
2 -> 0 9 -> 0 1 -> 0 | 7 -> 0 6 -> 0 3 -> 0 | 8 -> 0 5 -> 0 4 -> 0
6 -> 0 4 -> 0 8 -> 0 | 5 -> 0 9 -> 0 2 -> 0 | 3 -> 0 1 -> 0 7 -> 0
5 -> 0 7 -> 0 3 -> 0 | 1 -> 0 4 -> 0 8 -> 0 | 2 -> 0 9 -> 0 6 -> 0
```

Expansões: 1565 **Backtracks:** 1509 **Iterações:** 4165438

- **Backtracking**: Como foi observado, o Backtracking resolveu a *instância fácil* sem fazer retrocesso algum e com poucas expansões (somente 14) por causa da propagação. Já para a *instância difícil*, houveram 1565 expansões e 1509 retrocesso. Em ambos as instâncias, o algoritmo BT resolveu relativamente rápido.

- **Mínimos Conflitos**: Agora chegamos à grande bronca. Observe que para a *instância fácil*, o Mínimos Conflitos resolveu muito rápido e com pouquíssimas iterações: somente 79. O grande problema é na *instância difícil*. Ele chegou a fazer 4165438 iterações para resolver o problema. Chegou também a rodar mais de uma hora; desistimos antes que uma solução fosse gerada.

2.2.3. Comparativo/Conclusão

Usamos a função time do Terminal Linux, função essa que retorna o tempo de execução de um programa. Ela foi bastante útil nos problemas de CSP, pois assim, conseguimos fazer algumas estimativas. Colocaremos aqui exatamente como foi o retorno no Terminal:

Backtracking em uma instância fácil demorou (`real 0m0.098s, user 0m0.088s, sys 0m0.008s`). Já o Mínimos Conflitos em uma instancia fácil (`real 0m0.199s, user 0m0.188s, sys 0m0.008s`). Para uma instância fácil, é muito complicado fazer um comparativo, pois ambos tiveram um tempo de resolução muito parecido. Poderíamos até dizer que tanto faz usar um ou o outro nesse tipo de problema.

Mas quando colocamos uma instância difícil, foi notória a diferença. Observe que usando Backtracking, o tempo de resposta foi (`real 0m0.276s, user 0m0.264s, sys 0m0.008s`). Usando o Mínimos Conflitos, foi (`real 9m13.702s, user 9m3.264s, sys 0m10.340s`). É impressionante a diferença de resposta dos dois algoritmos. Acreditamos que Mínimos Conflitos não seja tão recomendado em problemas no mesmo estilo do Sudoku. Para problemas assim, achamos mais recomendado usar Backtracking.

[Russell and Norvig 2010] [Foundation. 2018]

Referências

Foundation., P. S. (2018). Standard library. Disponível em: `https://docs.python.org/3.6/library/l`. Acesso em: 26 dez. 2017.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Michael Hirsch, 3th edition.