Cobertura exata de conjuntos: modelagem alternativa do problema das N rainhas

João Mateus de Freitas Veneroso

Departamento de Ciência da Computação da Universidade Federal de Minas Gerais

June 24, 2017

1 Introdução

Este artigo discute o problema NP-completo de cobertura exata de conjuntos e, em particular, a solução para o problema das N rainhas por meio de três abordagens diferentes: o algoritmo de backtracking de Dijkstra [1], o algoritmo Dancing Links (DLX) de Knuth e o algoritmo DLX com a utilização da heurística S [2]. Embora o algoritmo Dancing Links também seja baseado no paradigma backtracking, ele utiliza uma estrutura de dados otimizada que reduz significativamente o tempo de execução. Além disso, a heurística S é uma estratégia de poda que visa reduzir o número de subproblemas, reduzindo o espaço de busca por soluções. Este artigo vai investigar e comparar as estratégias listadas em termos do número de subproblemas e do tempo de execução.

2 Cobertura exata de conjuntos

O problema de cobertura exata por conjuntos pode ser descrito da seguinte forma: dada uma coleção S de subconjuntos de um conjunto C, $S^* \subseteq S$ é uma cobertura exata de C se todo subconjunto $S_i \in S^*$ é disjunto em relação aos demais subconjuntos de S^* e $S_1 \cup S_2 \cup ... \cup S_n = C$. Ou seja, todo elemento $x \in C$ deve aparecer em um único subconjunto de S^*. O problema da cobertura exata é o problema número 14 dos 21 problemas NP-completos descritos por Karp em 1972 [3].

Perceba que o problema de cobertura exata de conjuntos é um caso particular do problema de cobertura de conjuntos, também NP-completo. No problema de cobertura de conjuntos não existe a necessidade dos subconjuntos da solução serem disjuntos. Portanto, dado um conjunto $\mathcal{S} = \{A, B, C, D, E\}$ e subconjuntos $\mathcal{X} = \{A, B, C\}$ e $\mathcal{Y} = \{C, D, E\}$, a subcoleção $\{\mathcal{X}, \mathcal{Y}\}$ representa uma solução para o problema de cobertura de conjuntos, porém não representa uma

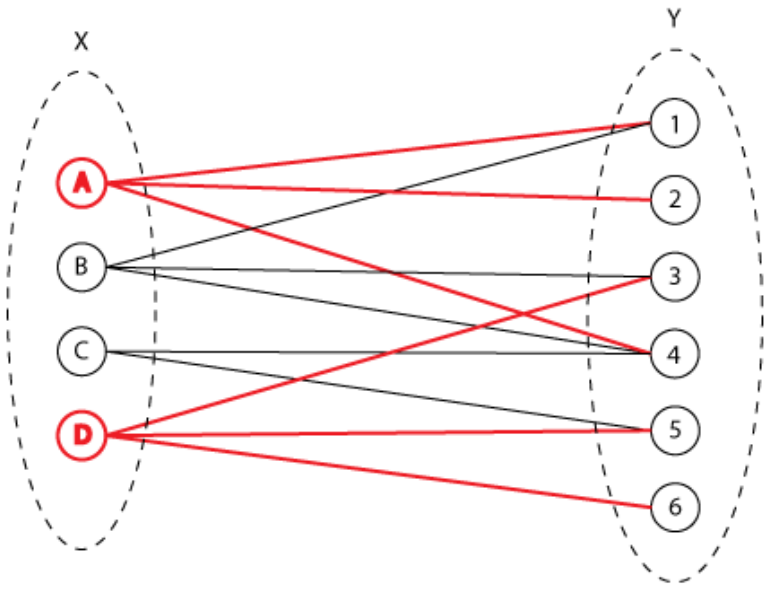

Figure 1: Exemplo de representação em grafo de uma cobertura exata

solução válida para o problema de cobertura exata. A limitação adicional no problema de cobertura exata torna muito difícil encontrar algoritmos que encontrem soluções aproximadas para o problema, uma vez que se torna complicado até mesmo definir o que seria uma solução aproximada. Por conta disto, o que nos resta é pensar em estratégias inteligentes de backtracking que reduzam o espaço de busca. Estas estratégias serão exploradas nas próximas seções.

2.1 Representação em grafo

O problema de cobertura exata de conjuntos pode ser representado por um grafo não direcionado bipartido onde os vértices são divididos em dois conjuntos disjuntos X e Y. Dado um conjunto $\mathcal{S}$ para o qual se procura uma cobertura exata, o conjunto X contém os subconjuntos de $\mathcal{S}$ e o conjunto Y contém os elementos de $\mathcal{S}$. Se um subconjunto de X contiver um elemento de Y haverá uma aresta no grafo ligando o subconjunto ao elemento. A figura 1 apresenta um exemplo de cobertura exata em grafos no qual $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ e os subconjuntos de $\mathcal{S}$ são:

- $A = \{1, 2, 4\}$
- $B = \{1, 3, 4\}$
- $C = \{4, 5\}$
- $D = \{3, 5, 6\}$

Nesse caso, A e D são disjuntos e $A \cup B = \mathcal{S}$. Portanto, $\{A, D\}$ é uma cobertura exata de $\mathcal{S}$.

2.2 Representação em matriz

Uma modelagem alternativa do problema de cobertura exata se dá por meio da utilização de uma matriz booleana onde as colunas representam elementos e as linhas representam subconjuntos. Para o exemplo descrito na figura 1, construiríamos a matriz equivalente:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Nesse caso, o problema se torna encontrar conjuntos de linhas que contenham exatamente um 1 em cada coluna. Para o exemplo em questão a única solução se dá pelas linhas 1 e 4, que representam os subconjuntos A e D, respectivamente. Esta abordagem para o problema de cobertura exata é muito útil no contexto do algoritmo DLX.

2.3 Exemplos concretos do problema

Alguns exemplos concretos do problema de cobertura exata de conjuntos são o quebra-cabeça Sudoku, o problema de Polyomino tiling, o problema de construção de quadrados latinos e o problema das N-rainhas, que será explorado nas próximas seções.

3 N-rainhas

O problema das N-rainhas ou N-damas consiste em dispor N rainhas em um tabuleiro de xadrez com $N \times N$ casas para qualquer $N \geq 4$ de forma que nenhuma rainha seja atacada por outra. Portanto, nenhuma rainha pode ocupar a mesma linha, coluna ou diagonal do tabuleiro. O caso clássico se dá para $N = 8$, mas aqui exploraremos como os algoritmos se comportam com diversos valores de N.

3.1 Solução de Dijkstra para o problema das N-rainhas

Se tivéssemos de testar todas as combinações com N rainhas no tabuleiro, o número de possibilidades seria tão grande que tornaria a solução do problema impraticável. Tomando o problema das 8 rainhas por exemplo, o número de combinações seria $64!/56!$ ou aproximadamente 2^{47}. A solução de Dijkstra [1] utiliza cinco propriedades do problema para reduzir o número de combinações a um número praticável. As propriedades são:

- Toda linha deve conter precisamente uma rainha

- Toda coluna deve conter precisamente uma rainha
- Existem 2N - 1 diagonais no sentido da diagonal principal, das quais N contém uma rainha
- Existem 2N - 1 diagonais no sentido da diagonal secundária, das quais N contém uma rainha
- Dada uma configuração onde nenhuma rainha seja atacada por outra, a remoção de uma das rainhas preserva esta propriedade

Tendo em vista as propriedades descritas, podemos construir um algoritmo de backtracking eficiente para encontrar todas as soluções do problema. Para guardar quais posições do tabuleiro estão livres poderíamos utilizar uma representação em matriz $N \times N$, no entanto, cada atualização poderia ter de alterar o valor de até $3N - 1$ quadrados. Portanto, utilizaremos uma representação otimizada com três arranjos representando as colunas ocupadas, as diagonais primárias ocupadas e as diagonais secundárias ocupadas. Dessa forma, apenas três valores terão de ser alterados para cada rainha posicionada. Além disso, é útil lembrar que todos os quadrados em uma diagonal primária tem a mesma diferença de índices de linha e coluna. E todos os quadrados em diagonais secundárias tem a mesma soma entre os índices de linhas e das colunas. Dessa forma, podemos indexar o arranjo das diagonais por meio destas somas e diferenças para acelerar a atualização da estrutura. No problema das 8 rainhas por exemplo, teríamos a seguinte estrutura de arranjos booleanos:

- Colunas: cols[0:7]
- Diagonais principais: up[-7:7]
- Diagonais secundárias: down[0:14]

Logo, para verificar se uma posição (i,j) do tabuleiro está livre basta checar os três arranjos com a seguinte expressão:

col[j] and up[i - j] and down[i + j]

Adicionalmente, é necessário manter um arranjo de inteiros com N elementos que guardará os índices das rainhas em cada linha e servirá de estrutura base para a iteração. Ao colocar a rainha número N no tabuleiro sabemos que chegamos a uma solução válida e podemos imprimir a posição de cada uma das rainhas pelo índice da sua linha. O pseudocódigo está descrito no algoritmo 1.

3.2 Algoritmo DLX

O algoritmo X implementado em termos de Dancing Links (DLX) descrito por Knuth [2] encontra todas as soluções para um problema de cobertura exata representado no formato de uma matriz de 0s e 1s, conforme descrito na seção

Algorithm 1 Backtracking N-queens

```
1: integer N;
2: integer array x[0 : N − 1];
3: boolean array col[0 : N − 1];
4: boolean array up[−(N − 1) : (N − 1)];
5: boolean array down[0 : 2N − 2];
6: procedure BacktrackingNQueensAux(n)
7:     if n = N then
8:         Imprime Solução return
9:     for h = 0; h < N; h ← h + 1 do
10:        if col[h] and up[n − h] and down[n + h] then
11:            x[n] ← h;
12:            col[h] ← false; up[n − h] ← false; down[n + h] ← false;
13:            BacktrackingNQueensAux(n + 1)
14:            col[h] ← true; up[n − h] ← true; down[n + h] ← true;
15: procedure BacktrackingNQueens(N)
16:    for k = 1; k ≤ N; k ← k + 1 do
17:        col[k] ← true
18:    for k = 0; k < 2N − 1; k ← k + 1 do
19:        up[k − N + 1] ← true
20:        down[k] ← true
21:    BacktrackingNQueensAux(1)
```

Índice	i	j	Up	Down
1	0	0	0	0
2	0	1	-1	1
3	0	2	-2	2
4	0	3	-3	3
5	1	0	1	1
6	1	1	0	2
7	1	2	-1	3
8	1	3	-2	4
9	2	0	2	2
10	2	1	1	3
11	2	2	0	4
12	2	3	-1	5
13	3	0	3	3
14	3	1	2	4
15	3	2	1	5
16	3	3	0	6

Table 1: Problema das 4 rainhas na forma de matriz

2.2. No caso do problema das N-rainhas, podemos modelar a solução com base em uma matriz onde as linhas correspondem aos N^2 diferentes quadrados do tabuleiro de xadrez e as colunas correspondem à quatro grupos:

- N colunas correspondentes às linhas do tabuleiro de xadrez
- N colunas correspondentes às colunas do tabuleiro de xadrez
- 2N - 3 colunas correspondentes às diagonais principais do tabuleiro de xadrez
- 2N - 3 colunas correspondentes às diagonais secundárias do tabuleiro de xadrez

A tarefa do algoritmo se torna encontrar uma cobertura exata para as colunas correspondentes às linhas e colunas do tabuleiro de xadrez, enquanto utilizamos no máximo uma vez cada elemento das colunas correspondentes às diagonais do tabuleiro. Perceba que o número de colunas para as diagonais é $2N-3$ e não $2N-1$ como seria esperado. Isso acontece pois os quadrados dos cantos superior esquerdo e inferior direito são os únicos por onde passam as suas diagonais secundárias e os quadrados dos cantos superior direito e inferior esquerdo são os únicos por onde passam suas diagonais principais, portanto não há possibilidade de existir conflito nestas diagonais e se torna desnecessário incluir as colunas correspondentes na matriz. Um exemplo de matriz para o problema das 4 rainhas se dá pela tabela 1. No caso desta tabela, ao invés de incluir todas as

colunas da matriz foi incluído apenas o índice da coluna referente ao seu grupo com a notação utilizada no algoritmo de Backtracking de Dijkstra. Note que uma solução possível para o problema é o conjunto de linhas 1, 8, 10, 15 que cobre todos os valores dos índices i e j sem repetir os valores das colunas Up e Down.

O algoritmo DLX consiste basicamente em construir a matriz descrita acima por meio de listas duplamente encadeadas. As linhas da matriz são listas onde cada elemento está conectado por ponteiros aos elementos à sua esquerda e à sua direita de forma toroidal, ou seja, o último elemento da linha está conectado ao primeiro pelo seu ponteiro direito e o primeiro elemento está conectado ao último pelo seu ponteiro esquerdo. Da mesma forma, as colunas tem o mesmo tipo de ligação, onde cada elemento está conectado aos elementos superior e inferior de forma toroidal. Além disso, cada coluna possui uma lista de cabeçalho, aos quais todos os elementos da coluna estão conectados. Cabe ressaltar neste momento que, no caso das N-rainhas por exemplo, apenas as colunas primárias referentes aos índices das linhas e das colunas do tabuleiro estarão ligadas entre si. Os ponteiros L e R das colunas secundárias referentes às diagonais serão nulos. O motivo disso ficará aparente no pseudo código do algoritmo.

Cada elemento x = 1 da matriz pode ser representado por um objeto com os seguintes campos: U[x], D[x], L[x], R[x], C[x], onde U aponta para o elemento de cima, D aponta para o elemento de baixo, L aponta para o elemento da esquerda, R aponta para o elemento da direita e C aponta para o cabeçalho da coluna. Além dos campos descritos, o cabeçalho possui os campos size, que indicam o número de 1s na coluna, e o campo name, que indica o nome da coluna. Por fim, existe um objeto root que simplesmente aponta para o primeiro item do cabeçalho.

A cada iteração do algoritmo, escolhemos uma coluna da matriz e, para cada linha naquela coluna (lembrando que a nossa lista encadeada só contém as linhas cujo valor naquela coluna é 1), adicionamos aquela linha à resposta e retiramos da matriz todas as linhas e colunas conflitantes. Em posse da nova matriz diminuída repetimos o processo até que tenhamos coberto todas as colunas encontrando uma resposta válida ou descobrindo que não há resposta naquele ramo da árvore de busca se ainda houverem colunas descobertas e não existirem mais linhas na matriz. Fazemos então o processo de backtracking recolocando as linhas e colunas conflitantes na matriz e repetindo o processo para a próxima linha.

O processo de remoção e reinserção de colunas na matriz é muito eficiente porque os ponteiros dos elementos removidos nunca são apagados. Por exemplo, digamos que o elemento x foi removido. Para reinserí-lo na sua antiga linha basta realizar a operação:

$$R[L[x]] = x \text{ e } L[R[x]] = x$$

A operação de reinserção na coluna é análoga.

O pseudocódigo do algoritmo DLX para o problema das N-rainhas está descrito no algoritmo 2. O procedimento InicializarMatriz cria as listas encadeadas

Algorithm 2 DLX N-queens

```
1: Object Array Root ▷ Guarda a representação em listas encadeadas da matriz
2: Object Array O ▷ Arranjo de linhas que constituem uma solução
3: procedure ImprimirSolução
4:     for O_k ∈ O do
5:         for i := R[O_k], i := R[R[O_k]] ... do
6:             Print Name[C[i]]
7: procedure RemoverColuna(col)
8:     L[R[col]] := L[col]
9:     R[L[col]] := R[col]
10:    for i := D[col], i := D[D[col]] ... do
11:        for j := R[i], j := R[R[i]] ... do
12:            U[D[j]] := U[j]
13:            D[U[j]] := D[j]
14:            Size[C[j]] := Size[C[j]] - 1
15: procedure ReinserirColuna(col)
16:    for i := U[col], i := U[U[col]] ... do
17:        for j := L[i], j := L[L[i]] ... do
18:            U[D[j]] := j
19:            D[U[j]] := j
20:            Size[C[j]] := Size[C[j]] + 1
21:    L[R[col]] := col
22:    R[L[col]] := col
23: procedure DlxNQueensAux(k)
24:    if R[Root] = Root then
25:        ImprimirSolução() return
26:    col := EscolherColuna()
27:    RemoverColuna(col)
28:    for linha := D[col], linha := D[D[col]] ... do
29:        O_k := linha
30:        for el := R[linha], el := R[R[linha]] ...  do
31:            RemoverColuna(el)
32:        DLXNQueensAux(k + 1)
33:        for el := L[linha], el := L[L[linha]] ...  do
34:            ReinserirColuna(el)
35:    ReinserirColuna(col)
36: procedure DLXNQueens(N)
37:    InicializarMatriz()
38:    DLXNQueenAux(0)
```

correspondentes à matriz para o problema das N-rainhas e inicializa o elemento Root. As soluções para o problema constituem em conjuntos de linhas que façam uma cobertura exata da matriz. A variável O guarda as linhas à medida que o algoritmo progride e a função ImprimirSolução imprime o identificador das colunas destas linhas. A função EscolherColuna, a princípio pode escolher sempre a primeira coluna da matriz, mas na próxima seção, exploraremos estratégias de escolha mais inteligentes. Por último, o atributo Size do cabeçalho parece não fazer nada no algoritmo descrito. No entanto, a sua importância ficará clara na próxima seção, que explora uma heurística para ampliar a eficácia do algoritmo DLX.

3.3 Heurística S

A heurística S consiste em uma estratégia inteligente de escolha das colunas no algoritmo DLX. Knuth observou que ao escolhermos a coluna que contém o menor número de 1s estaremos minimizando o número de subproblemas a serem resolvidos, uma vez que, para cada 1 na coluna, um novo galho da árvore de subproblemas é gerado. A implementação desta heurística é muito simples levando em conta que no algoritmo 2 já salvamos no cabeçalho o número de 1s de cada coluna. Portanto, basta que a função EscolherColuna retorne a coluna com o menor atributo Size.

Além da heurística S, utilizaremos uma outra heurística para acelerar a solução do problema das N-rainhas. É melhor colocar as rainhas nas casas do meio do tabuleiro antes das extremidades, pois estas casas normalmente limitam mais a colocação das próximas rainhas, diminuindo a quantidade de subproblemas. Por exemplo, considerando o problema das 4 rainhas, a ordem de prioridade para colocação das peças nas linhas (L1, L2, L3, L4) e colunas (C1, C2, C3, C4) seria: L3, C3, L2, C2, L4, C4, L1, C1. Dessa forma, caso duas colunas possuam o mesmo Size pela heurística S, o desempate será feito por meio desta ordem de prioridade.

A próxima seção apresenta resultados experimentais para a resolução do problema das N-rainhas utilizando cada um dos algoritmos e heurísticas apresentados até agora.

3.4 Heurística Alternativa (Heurística A)

A heurística S realiza uma poda que pode ou não reduzir o número de sub problemas, no entanto, ela ainda produz todos os resultados exatos para o problema. Nesta seção, nós propomos uma estratégia de poda que reduz o espaço de busca relaxando a restrição de ter que produzir todas as respostas válidas. O intuito dessa heurística é reduzir o número de subproblemas que precisam ser resolvidos descartando o mínimo possível de respostas válidas. Para tanto, vamos chamar de eficiência a razão:

$$\frac{\textit{número de respostas}}{\textit{número de subproblemas}} \tag{1}$$

O nosso objetivo é reduzir o espaço de busca ao máximo preservando a eficiência em relação ao algoritmo exato. A heurística alternativa acrescenta um passo no algoritmo DLX durante a etapa de recursão na linha 32 que limita o número de recursões. A recursão só acontece se, ao posicionar a n-ésima rainha, o tabuleiro resultante da remoção das colunas e linhas escolhidas possui pelo menos lim_n quadrados vazios. A intuição por trás disso é que, configurações de rainhas que deixam mais quadrados vazios tem mais chance de produzir respostas válidas para o problema. Para definir o limiar lim_n nós utilizamos a distribuição de quadrados vazios para cada n rainhas posicionadas, de forma que:

$$lim_n = \mu(n) + k\sigma(n) \tag{2}$$

onde $\mu(n)$ representa a média de quadrados vazios entre as configurações com n rainhas e $\sigma(n)$ representa o desvio padrão entre as configurações com n rainhas. O valor $k = 1$ se mostrou satisfatório nos experimentos. Os valores $\mu(k)$ e $\sigma(k)$ podem ser estimados com base em um conjunto pequeno de dados. Para os experimentos realizados, um conjunto de 1000 configurações se mostrou suficiente.

A heurística apresentada consegue reduzir o número de subproblemas e aumentar a eficiência em relação à solução exata. Quanto maior o valor do parâmetro k, mais o número de soluções encontradas se aproxima do número de soluções exatas, no entanto, mais subproblemas também são resolvidos. No caso das 8 rainhas, para $k = 1$, a heurística alternativa encontra 40 das 92 respostas, no entanto, ela resolve apenas 356 subproblemas, enquanto o algoritmo DLX com a heurística S resolve 980. Portanto, a eficiência da heurística alternativa é 11.24% e a eficiência do algoritmo DLX+S é 9.39%.

Cabe ressaltar que a heurística alternativa pode ser aplicada junto ao algoritmo DLX+S e, de fato, esse foi o caso nos experimentos realizados. Como a heurística S já reduz consideravelmente o espaço de busca, o impacto da heurística alternativa é atenuado em relação à sua aplicação no algoritmo DLX sem outras heurísticas.

4 Experimentos

O primeiro experimento testou o desempenho de 3 algoritmos na resolução do problema das N-rainhas: Backtracking N-Queens (BT), DLX e DLX com a heurística S (DLX+S). O número de rainhas nos experimentos variou entre 4 e 16. Foi medido o tempo de execução e o número de nodos da árvore de subproblemas em cada um dos casos de teste. O resultado dos experimentos

Rainhas	Número de Soluções	BT	DLX	DLX+S
4	2	0,053	0,24	0,23
5	10	0,16	0,74	0,71
6	4	0,48	2,47	2,09
7	40	1,98	10,02	5,80
8	92	9,27	31,08	19,30
9	352	31,19	134,92	65,92
10	724	160,50	563,74	255,95
11	2680	739,95	2.615,23	988,45
12	14200	4.067,38	13.477,93	4.673,26
13	73712	24.093,38	79.421,48	21.941,21
14	365596	149.156,43	455.339,30	122.343,63
15	2279184	905.307,84	3.234.598,81	775.524,83
16	14772512	6.364.506,68	20.468.219,73	3.852.383,21

Table 2: Tempo de execução (milisegundos)

Rainhas	Número de Soluções	BT	DLX	DLX+S
4	2	17	17	13
5	10	54	54	46
6	4	153	153	91
7	40	552	552	317
8	92	2057	2057	980
9	352	8394	8394	3504
10	724	35539	35539	11959
11	2680	166926	166926	45879
12	14200	856189	856189	213835
13	73712	4674890	4674890	1052203
14	365596	27358553	27358553	5555787
15	2279184	171129072	171129072	31208159
16	14772512	1141190303	1141190303	194103108

Table 3: Número de subproblemas (nodos da árvore de possibilidades)

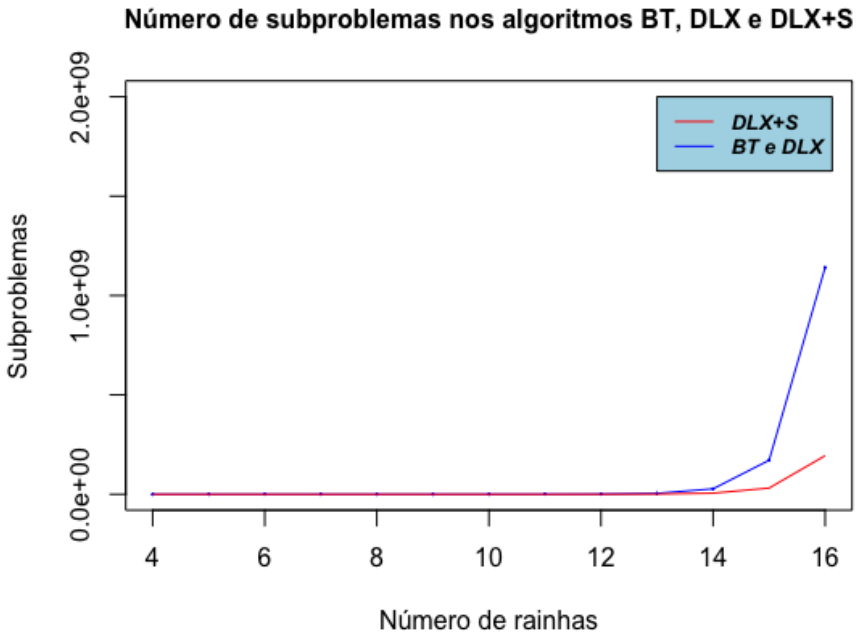

Figure 2: Número de subproblemas para os algoritmos BT, DLX e DLX+S

Rainhas	Soluções		Subproblemas	
	DLX+S	DLX+S+A	DLX+S	DLX+S+A
4	2	2	17	13
5	10	6	54	29
6	4	3	153	68
7	40	17	552	143
8	92	40	2057	356
9	352	172	8394	1398
10	724	264	35539	3484
11	2680	1093	166926	15137
12	14200	3978	856189	48460
13	73712	17676	4674890	201218
14	365596	82177	27358553	961543
15	2279184	330187	171129072	3386786
16	14772512	2266861	1141190303	23676987

Table 4: Número de soluções e subproblemas para os algoritmos DLX+S e DLX+S+A

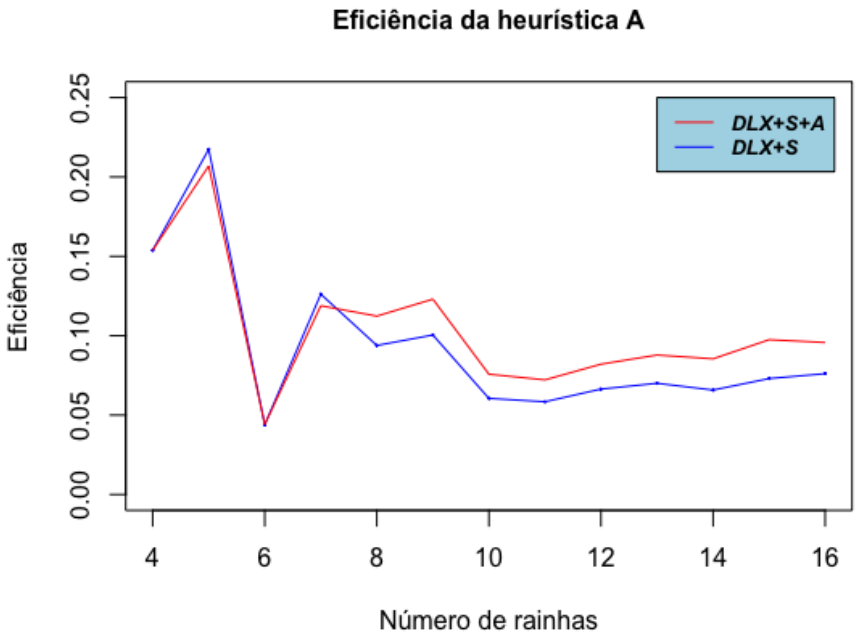

Figure 3: Eficiência da heurística A

está descrito nas tabelas 2 e 3. A figura 2 mostra o resultado em formato de gráfico. Todos os algoritmos foram implementados em Python e rodados em um MacBook Pro 2.6 GHz Intel Core i5.

O segundo experimento comparou a eficiência (conforme definida na seção anterior) do melhor algoritmo obtido no experimento 1 (DLX+S) com o algoritmo (DLX+S+A) que consiste no algoritmo DLX com o uso da heurística S e da heurística A. O resultado está descrito na figura 3. Por fim, os resultados estão descritos mais detalhadamente na tabela 4.

5 Conclusão

Os experimentos demonstraram que o algoritmo DLX com a utilização da heurística S diminui consideravelmente o espaço de busca por soluções quando o número N de rainhas fica grande, acelerando o tempo de execução. No entanto, sem a utilização da heurística S, o algoritmo DLX apresenta desempenho pior do que o Backtracking de Dijkstra, porque resolve o mesmo número de subproblemas, mas possui um custo fixo de atualização maior. A heurísitica A consegue uma eficiência maior na busca por resultados do que o algoritmo DLX com a heurística S, no entanto, ela não consegue produzir todos os resultados válidos.

O problema das N-rainhas é um caso particular de cobertura exata, mas as conclusões deste trabalho possivelmente podem ser generalizadas para outros problemas deste tipo, uma vez que a heurística S e a heurística A podem ser aplicadas em qualquer problema de cobertura exata.

References

[1] C. A. R. H. Ole-Johan Dahl Edsger W. Dijkstra, *Structured Programming.* Academic Press, London, pages 72-82, 1972.

[2] D. Knuth, *Dancing links.* Millennial Perspectives in Computer Science, Palgrave, pages 187-214, 2000.

[3] R. M. Karp, *Reducibility Among Combinatorial Problems.* Plenum, New York, pages 85–103, 1972.