Uso de heurísticas em problemas de biclustering

Bernardo de Almeida Abreu
bernardoabreu@dcc.ufmg.br

Cristiano Guimarães Pimenta
cgpimenta@dcc.ufmg.br

I. Introdução

O termo *biclustering* foi inicialmente utilizado por Mirkin [1] para descrever uma técnica de *clustering* introduzida por Hartigan [2] em 1972. Um *cluster* é um subconjunto de dados cujas entidades constituintes possuem alta similaridade entre si, em contraste com o restante dos dados do conjunto [1]. Assim, *clustering* é uma técnica matemática para agrupar os dados coletados sobre fenômenos do mundo real, de forma a revelar essas similaridades.

Um conjunto de dados pode ser representado por uma matriz, cujas linhas representam objetos que se deseja caracterizar e cujas colunas representam suas características. O termo *biclustering* se refere ao *clustering* simultâneo de linhas e colunas da matriz de dados, evidenciando, assim, a interrelação entre linhas e colunas expressadas pelos dados [1].

Os objetivos de diferentes algoritmos de *biclustering* podem se referir a encontrar diferentes tipos de *bicluster*. Nesse trabalho, serão considerados os algoritmos que buscam encontrar *biclusters* de valor constante, ou seja, aqueles algoritmos que buscam agrupar linhas e colunas similares, de forma a encontrar um *bicluster* cujos valores representados no mesmo são constantes em todo o *bicluster*. No mundo real, valores exatamente idênticos são muito difíceis de serem encontrados, de forma que se faz necessário permitir a introdução de ruídos nos *biclusters*. Por esse motivo, a qualidade de um *bicluster* é avaliada através de sua variância.

Uma das principais aplicações de *biclustering* é na expressão gênica. Uma matriz é utilizada para representar os dados, de forma que cada linha representa um gene e cada coluna, uma condição. Assim, cada elemento da matriz representa o nível de expressão de um gene sob uma condição específica. Objetivos comuns ao se analisar dados de expressão gênica incluem agrupar os genes de acordo com sua expressão sob várias condições, classificar um novo gene a partir da expressão de outros genes, agrupar as condições baseado na expressão de uma certa quantidade de genes e classificar uma nova amostra a partir da expressão de genes sob aquela condição experimental. Técnicas de *clustering* tradicionais podem se tornar deficientes, pois muitos padrões de ativação são comuns a subgrupos de genes sob certas condições, de modo que, sob outras condições, eles se comportem de maneira diferente. Essas características resultam na necessidade de se utilizar *biclustering* [3].

II. Definição e formulação do problema

Uma formulação simples do problema de *biclustering* é dada por Madeira e Oliveira [3] e será apresentada a seguir.

A. Bicluster

Seja $A \in \mathbb{R}^{n \times m}$ uma matriz de n linhas por m colunas contendo os dados do problema. A é definida pelos seus conjuntos de linhas $X = \{x_1, \cdots, x_n\}$ e de colunas $Y = \{y_1, \cdots, y_m\}$. O elemento na interseção entre a linha $x_i \in X$ e a coluna $y_j \in Y$ é denotado por a_{ij}. Uma submatriz de A composta por um subconjunto $I \subseteq X$ de linhas e um subconjunto $J \subseteq Y$ de colunas pode ser representada por A_{IJ}.

Um *cluster* de linhas A_{IY} é um subconjunto de linhas que apresentam comportamento similar ao longo de todas as colunas, podendo ser definido como uma submatriz de A de $k \leq n$ linhas por m colunas. Similarmente, um *cluster* de colunas A_{XJ} é um subconjunto de colunas que apresentam comportamento similar ao longo de todas as linhas. A_{XJ}, portanto, é definido como uma submatriz de A de n linhas por $s \leq m$ colunas.

Por sua vez, um *bicluster* A_{IJ} engloba um *cluster* de linhas e um de colunas, sendo, assim, um subconjunto $I = \{i_1, \cdots, i_k\}$ de linhas e um subconjunto $J = \{i = j_1, \cdots, j_s\}$ de colunas. A_{IJ} é uma submatriz de A de $k \leq n$ linhas por $s \leq m$ colunas.

B. Algoritmos de biclustering

Dada uma matriz de dados A, um algoritmo de *biclustering* busca um conjunto de *biclusters* $B_k = (I_k, J_k)$, conforme definição apresentada na Seção II-A, tal que cada *bicluster* B_k satisfaz a algum critério de qualidade, descrito na seção a seguir.

C. Avaliação da qualidade

Um *bicluster* constante perfeito é uma submatriz (I, J), onde todos os valores são iguais. Ou seja, para todo $i \in I$ e todo $j \in J$:

$$a_{ij} = \mu \tag{1}$$

No mundo real porém, *biclusters* perfeitos são muito difíceis de serem encontrados. Dessa forma, se faz necessário permitir a introdução de ruídos nos mesmos, de modo que que os valores a_{ij} dos *biclusters* passam a ser representados por $\eta_{ij} + \mu$, onde η_{ij} é o ruído associado ao valor μ do *bicluster* a_{ij}.

O algoritmo introduzido por Hartigan [2] utiliza a variância para avaliar a qualidade de cada *bicluster* B_k, como mostrado na Equação 3. Em geral, os diferentes algoritmos de *biclustering* introduzidos posteriormente definem suas próprias métricas de qualidade. Apesar disso, muitas dessas medidas de qualidade também utilizam uma função da variância do

bicluster. O número de *bliclusters* encontrados também é variável.

As heurísticas consideradas no escopo deste trabalho retornam K *biclusters* B_k, com $k \in 1, 2, \cdots, K$. Entretanto, avaliaremos apenas o melhor *bicluster* B encontrado por cada uma delas. Uma vez que cada heurística utiliza uma métrica própria de avaliação de qualidade dos *biblusters* durante a busca, foi definido um índice de qualidade que permite a comparação dos resultados, levando em consideração a variância e a área de B. Tal índice, que desejamos maximizar, é definido conforme a Equação 2. A variância do *bicluster* é dada na Equação 3.

$$Q(B) = \frac{\log(|I| \cdot |J|)}{VAR(B)} \tag{2}$$

$$VAR(B) = \sum_{i \in I, j \in J} (a_{ij} - a_{IJ})^2 \tag{3}$$

O termo a_{IJ} utilizado na Equação 3 corresponde à média de todos os elementos do *bicluster* e é definido conforme a Equação 4.

$$a_{IJ} = \frac{1}{|I||J|} \sum_{i \in I, j \in J} a_{ij} \tag{4}$$

D. Complexidade

A complexidade do problema de *biclustering* varia de acordo com a formulação e com a função utilizada para avaliar a qualidade das soluções. Entretanto, é possível mostrar que, na formulação utilizada no presente trabalho, o problema é NP-completo. A prova, conforme enunciada por Madeira e Oliveira [3], será apresentada a seguir.

Considere o grafo $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas. G é dito bipartido se for possível particionar o conjunto de vértices em subconjuntos V_1 e V_2 tais que $V = V_1 \bigcup V_2$ e cada aresta em E possuir uma extremidade em V_1 e a outra em V_2. A matriz de dados A, definida na Seção II-A, pode ser transformada em um grafo bipartido completo. Cada linha x_i de A corresponde a um vértice $v_i \in V_1$, enquanto cada coluna y_j de A corresponde a um vértice $v_j \in V_2$. A aresta entre v_i e v_j possui peso a_{ij}. A Figura 1 ilustra um exemplo de matriz de dados (Figura 1a) e o grafo bipartido correspondente (Figura 1b).

Dada a transformação da matriz de dados A em um grafo bipartido G, encontrar o maior *bicluster* em A é equivalente a achar a *biclique* de peso máximo em G. Um algoritmo que resolve o problema de *biclique* retorna um subconjunto $V_1' \in V_1$ e um subconjunto $V_2' \in V_2$, que correspondem às linhas e colunas de A, respectivamente, que fazem parte do maior *bicluster* de A.

Dawande e colaboradores [4] provaram que o problema de encontrar a *biclique* de peso máximo em um grafo bipartido é NP-completo. Dessa forma, mostramos que o problema de *biclustering* também faz parte dessa classe.

Como *biclustering* é NP-completo, é inviável utilizar algoritmos exatos exponenciais para encontrar a melhor solução para o problema. Assim, podemos utilizar heurísticas para encontrar soluções viáveis em tempo polinomial. Heurísticas bem projetadas podem encontrar soluções satisfatoriamente próximas da ótima, gastando tempo consideravelmente menor que algoritmos exatos.

III. Trabalhos relacionados

Existe uma grande diversidade de algoritmos para *biclustering* na literatura, principalmente aplicados a dados de expressão gênica. Revisões de algoritmos para tal aplicação podem ser encontradas em [3], [5]–[7]. A seguir, são apresentadas quatro heurísticas construtivas e três metaheurísticas para o problema, além de uma metaheurística baseada em busca tabu que resolve o problema de biclique balanceada máxima.

A. Heurísticas construtivas

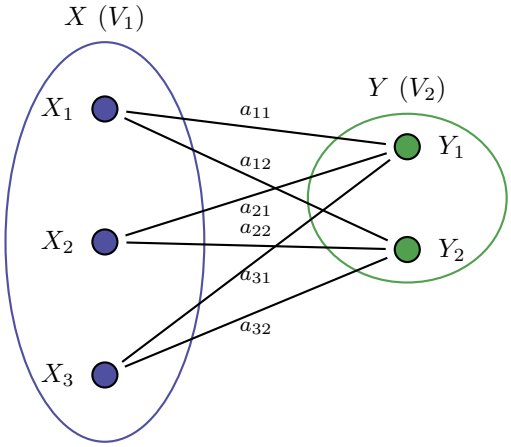
a) Hartigan, 1972 [2]: Hartigan introduziu o conceito de *biclustering* no início dos anos 1970. O algoritmo foi proposto com o objetivo que agrupar dados sobre as eleições presidenciais dos Estados Unidos da América durante um período de tempo. As linhas da tabela representavam os estados do sul do país, as colunas representavam os anos e os valores representavam a porcentagem de votos para o candidato republicano daquela eleição. Foi utilizada uma estratégia gulosa de dividir-para-conquistar, na qual a matriz de entrada é iterativamente particionada em um conjunto de submatrizes, até que um determinado número de matrizes, que correspondem aos *biclusters* (sem sobreposição), seja obtido. O particionamento é avaliado com base na variância de cada *bicluster* em relação ao modelo ideal.

b) Cheng e Church, 2000 [8]: Este foi o primeiro algoritmo de *biclustering* aplicado especificamente a dados de expressão gênica. Trata-se de um algoritmo guloso que encontra *biclusters* (com sobreposição) por meio da inserção e remoção de linhas e colunas da matriz de dados. A qualidade dos *biclusters* é avaliada por meio de seu resíduo quadrático médio (MSR, do inglês *mean squared residue*), uma função que considera a variância do *bicluster* e as médias das variâncias das linhas e colunas. A cada iteração, o algoritmo remove todas as linhas e colunas cujo MSR está acima de um determinado limite. No caso de matrizes pequenas (dimensão inferior a 100), apenas a linha e coluna com maior MSR são removidas. Após remoção, as linhas e colunas que não aumentam o MSR total da matriz são adicionadas à solução. Dessa forma, o algoritmo busca *biclusters* grandes e com MSR abaixo de um determinado limite.

c) Yip et al., 2004 [9]: Incialmente, este algoritmo cria *clusters* pequenos que são iterativamente mesclados, de forma a produzir um *cluster* maior. São utilizados dois parâmetros que definem a dimensão mínima dos *clusters* criados e a relevância mínima de uma dimensão (definida em termos da variância da dimensão e do *cluster* inteiro). No início do processo, cada linha e coluna, também chamadas de objetos, é considerada como um *cluster* separado. Para cada cluster, as dimensões que satisfazem as restrições impostas sobre os parâmetros descritos acima são selecionadas e mescladas. Esse

$$\begin{array}{c} \\ X_1 \\ X_2 \\ X_3 \end{array} \begin{array}{c} \begin{array}{cc} Y_1 & Y_2 \end{array} \\ \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \end{array}$$

(a) Matriz de dados



(b) Grafo bipartido completo. Os pesos das arestas correspondem aos elementos da matriz de dados.

Figura 1: Transformação da matriz de dados em um grafo bipartido completo.

processo é repetido enquanto existirem *clusters* que podem ser mesclados.

d) Teng e Chan, 2008 [10]: Na abordagem proposta, os dados de expressão gênica são alternadamente ordenados e transpostos, utilizando correlações ponderadas para medir a similaridade de genes e condições. O índice de correlação proposto é uma variação do coeficiente de correlação de Pearson e o algoritmo é baseado na abordagem do conjunto dominante de Pavan e Pelillo [11]. No algoritmo, vetores ponderados são utilizados para ordenar os genes ou condições, e a cada iteração a matriz é transposta de forma a se alternar entre genes e condições. Ao final do processo, o *bicluster* se encontra no canto da matriz reordenada, sendo definido através de um limiar definido para a correlação entre linhas e colunas adjacentes. Para que se encontre mais de um *bicluster*, assim que um *bicluster* é encontrado, os pesos dos atributos que não foram utilizados no mesmo são aumentados, ao mesmo tempo que o peso dos atributos utilizados é diminuído. Essa abordagem permite *biclusters* sobrepostos, porém eles são controlados e penalizados.

B. Metaheurísticas

a) Bryan et al., 2006 [12]: O trabalho utiliza a técnica estocástica de *simulated annealing* (SA), que define a qualidade das soluções de acordo com uma função de *fitness*. Neste caso, a métrica utilizada foi o MSR. SA explora o espaço de soluções de forma a aceitar, probabilisticamente, soluções com *fitness* pior que a melhor solução encontrada. Durante a busca, a probabilidade de se escolher uma solução pior decresce, seguindo uma estratégia chamada de "resfriamento", que permite uma eventual convergência da solução. A profundidade da busca é determinada pelo número de sucessos (soluções que levam a uma melhoria da *fitness*) em cada temperatura. Cada solução é perturbada por meio da inserção ou remoção de linhas ou colunas, mantendo-se um tamanho mínimo da solução. O algoritmo deve ser rodado k vezes para se obter k *biclusters*, substituindo a solução na matriz original por valores aleatórios, de forma a evitar sobreposição.

b) Dharan e Nair, 2009 [13]: A solução proposta para o problema de *biclustering* utiliza a técnica de GRASP reativo. GRASP consiste de uma fase de construção, na qual uma solução é construída por uma estratégia gulosa aleatória a partir de uma solução inicial, e de uma fase de busca local, que parte da solução encontrada na fase anterior e caminha em direção a um ótimo local. Ambas as fases são executadas iterativamente, até que uma condição de parada seja alcançada. Na versão reativa da técnica, o parâmetro que controla a lista de candidatos a serem adicionados à solução na fase de construção é ajustado automaticamente, de acordo com a qualidade das últimas soluções obtidas. A qualidade é avaliada de acordo com o MSR. Na fase de construção, as linhas e colunas que aumentam o MSR da solução até o limite máximo definido são adicionadas à lista de candidatos. Um elemento é selecionado aleatoriamente dessa lista e inserido na solução. Para se obter k *biclusters*, é necessário fornecer k soluções iniciais à fase de construção.

c) França e Zuben, 2011 [14]: França e Zuben utilizam uma abordagem baseada em otimização de colônia de formigas (ACO, do inglês *ant colony optimization*), que encontra um conjunto de *biclusters* que maximiza a cobertura da base de dados, com MSR abaixo de um limite máximo definido e com tamanho médio suficientemente bom. O algoritmo é composto por três heurísticas principais. Uma delas constrói dinamicamente uma lista de candidatos que são mais prováveis de levar a *biclusters* com maior cobertura da base. Tal lista é passada para uma heurística construtiva, que começa com um *bicluster* inicial formado por apenas uma linha. Iterativamente, a heurística construtiva remove colunas e insere linhas de forma a controlar o MSR da solução. A metaheurística ACO fornece à heurística construtiva a melhor ordem na qual as linhas devem ser inseridas com o objetivo de maximizar o tamanho do *bicluster*.

d) Zhou e Hao, 2017 [15]: O trabalho apresenta uma heurística baseada em busca tabu para resolver o problema de biclique balanceada máxima. Conforme descrito na Seção II-D, é possível transformar uma instância de *biclustering* em uma instância de biclique em tempo polinomial. A heurística proposta por Zhou e Hao inclui duas partes principais: uma busca tabu baseada em restrição (CBTS, do inglês *constraint-based tabu search*) e um procedimento de redução de grafo. CBTS é usada para encontrar bicliques de alta qualidade em um espaço de busca que contém bicliques levemente desbalanceadas. A busca começa a partir de uma solução inicial, não necessariamente balanceada, e vai sendo melhorada pela CBTS. A etapa de redução do grafo remove todos os vértices (e arestas incidentes a eles) cujo grau é menor ou igual ao tamanho da maior biclique conhecida, já que tais vértices não podem melhorar a melhor solução encontrada, aumentando assim a eficiência da heurística.

IV. Contribuições

Nas Seções V e VI apresentamos duas novas heurísticas construtivas para o problema de *biclustering*, uma baseada na enumeração gulosa de subconjuntos e uma baseada na ordenação das linhas e colunas da matriz de dados. Também foram propostas duas metaheurísticas, uma baseada em GRASP (Seção VII) e uma baseada em busca tabu (Seção VIII).

A Seção IX descreve os experimentos realizados. Os algoritmos propostos foram comparados aos de Cheng e Church [8], Teng e Chan [10], França e Zuben [14] e Zhou e Hao [15]. Os resultados obtidos estão descritos na Seção IX-E.

V. Heurística de enumeração gulosa de subconjuntos

O algoritmo de enumeração gulosa de subconjuntos busca encontrar, para cada linha, os conjuntos de colunas que apresentam os maiores valores na matriz. Quando os conjuntos de colunas se repetem em mais de uma linha, as duas linhas são agrupadas em um único conjunto. O algoritmo é composto dos seguintes passos.

1) Construir um DAG, grafo acíclico direcionado, no qual cada vértice representa um conjunto de K até $\mathcal{C} - K$ colunas, onde $\mathcal{C}$ é o conjunto de colunas da matriz que possuem os maiores valores em pelo menos uma linha. Cada aresta do grafo liga um vértice que representa um conjunto de n colunas que têm os n maiores valores em uma linha até o vértice que representa o conjunto de $n+1$ colunas que têm os $n+1$ maiores valores para esta mesma linha.
2) Utilizar programação dinâmica para enumerar $|\mathcal{R}|$ *biclusters* candidatos, onde $\mathcal{R}$ é o conjunto de linhas da matriz.

A. Primeira etapa: Construção do DAG

A primeira etapa do algoritmo consiste em construir um grafo acíclico direcionado que liga cada conjunto de n colunas ao seus super-conjuntos de $n+1$ colunas. Cada vértice do grafo corresponde a um subconjunto das colunas da matriz e possui um subconjunto de linhas associado ao mesmo. O Algoritmo 1 descreve essa etapa.

Algoritmo 1 Construção do DAG

Input
$\mathcal{C}$, o conjunto de todas as colunas,
$\mathcal{R}$, o conjunto de todos os linhas,
D, a matriz de valores, $D \in \mathbb{R}^{(\mathcal{R}\times\mathcal{C})}$,
k, o número mínimo de colunas, $k \in \{1, ..., \lfloor \frac{|\mathcal{C}|}{2} \rfloor\}$
Output
vertices $V \subset 2^{\mathcal{C}}$
sucessores, $succ \in (2^{\mathcal{C}})^V$
biclusters em cada nível do grafo, $L, \{v \mid v \subseteq V\}$ $tal\ que\ |L| = (|\mathcal{R}| - 2 \times (k-1))$
qualidade de cada vértice, $q \in \mathbb{R}_+^V$
linhas de cada vértice, $R = \{r \mid r \subseteq \mathcal{R}\}$

1: **function** BUILDDAG($\mathcal{C}, \mathcal{R}, D, k$)
2: $(V, succ, L, q, R) \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
3: **for all** $r \in \mathcal{R}$ **do**
4: $C \leftarrow \emptyset$
5: $(c_1, \ldots, c_{|\mathcal{C}|}) \leftarrow$ REVERSESORT($\mathcal{C}_r$)
6: $M \leftarrow \{c_1, \ldots, c_k\}$
7: UPDATEVERTICES($V, C, L, 0, q, |D_{r,c_k} - D_{r,c_{k+1}}|$)
8: $L_0 \leftarrow L_0 \cup C$
9: **for** $i \leftarrow k+1, (|\mathcal{C}| - k - 1)$ **do**
10: $C \leftarrow C \cup S(i)$
11: $succ(C) \leftarrow succ(C) \cup \{m_i\}$
12: $(V, L, q, R) \leftarrow$ UPDATEVERTICES($V, C, L, i - k, q, |D_{r,c_k} - D_{r,c_{k+1}}|$)
13: **end for**
14: **end for**
15: **return** $(V, succ, L, q, R)$
16: **end function**

O primeiro passo da construção do DAG consiste em, para cada linha, ordenar de forma decrescente as colunas em função de seu valor. Na linha 1.6, as primeiras k colunas, onde k é um parâmetro passado para o algoritmo, são utilizadas para criar um subconjunto. Esse subconjunto é utilizado para inserir um novo vértice no grafo, caso ele não tenha sido inserido, ou atualizar o vértice existente, adicionando uma nova linha ao subconjunto de linhas do mesmo.

O primeiro subconjunto de colunas para a linha atual é inserido, e, para cada nível restante para essa linha, um subconjunto de colunas é criado e, a partir dele, um vértice é inserido ou atualizado no grafo. Cada vértice do grafo possui um conjunto de sucessores, que são os vértices com os quais eles se ligam. A relação entre sucessores e vértices é tal que o subconjunto de colunas relacionadas aos sucessores corresponde à união entre o subconjunto de colunas associadas ao vértice e uma nova coluna. O conjunto de sucessores é atualizado a cada novo vértice inserido ou atualizado no grafo. A qualidade do vértice passada para a função de atualização dos vértices é calculada como a diferença do valor da coluna

com menor valor que pertence ao vértice e o valor da próxima coluna, que será a coluna fora do vértice de maior valor.

A atualização do grafo com o novo vértice é descrita no Algoritmo 2. É necessário verificar se já existe um vértice que contém o subconjunto de colunas. Caso ele tenha sido inserido previamente, a qualidade do vértice inserido e o conjunto de linhas do mesmo são atualizados. No pseudocódigo mostrado, a qualidade é atualizada utilizando a norma 1, mas qualquer norma pode ser utilizada. Caso o subconjunto de colunas ainda não tenha sido visto, ele é adicionado ao conjunto de subconjuntos vistos e então um novo vértice é inserido no grafo. A função `insert` insere um vértice no grafo, definindo seu subconjunto de colunas, a linha inicial do seu conjunto de linhas e sua qualidade inicial. O conjunto L é utilizado para guardar quais vértices estão inseridos em cada nível do grafo.

Algoritmo 2 Atualiza vértices do grafo

1: **function** UPDATEVERTICES(V, C, L, k, q, δ)
2: **if** $C \in V$ **then**
3: $q \leftarrow q \cup \{C \mapsto q(C) + \delta\} \setminus \{C \mapsto q(C)\}$
4: $R \leftarrow R \cup \{C \mapsto \ r \cup R(C)\} \setminus \{C \mapsto R(C)\}$
5: **return** (V, L, q, R)
6: **else**
7: $L_k \leftarrow L_k \cup C$
8: **return** $(V \cup C, L, q \cup \{C \mapsto \delta\}, R \cup \{C \mapsto \ r\})$
9: **end if**
10: **end function**

B. Segunda etapa: Enumeração de padrões

A segunda etapa do algoritmo consiste em enumerar, por meio de programação dinâmica, os *biclusters*. Cada vértice no grafo corresponde a um *bicluster*, uma vez que contém um subconjunto de colunas e um subconjunto de linhas da matriz. Essa função seleciona os *biclusters* com as maiores qualidades, de forma que não sejam escolhidos *biclusters* cujo subconjunto de colunas associadas não seja um subconjunto ou um superconjunto de algum *bicluster* já selecionado.

A enumeração de padrões, descrita no Algoritmo 3, é dividida em duas partes, que consistem em descer no grafo marcando os melhores *biclusters* levando em consideração subconjuntos, e depois subir atualizando os melhores *biclusters* levando em consideração os superconjuntos. As melhores qualidades vistas até o momento são guardadas em b e são inicializados com os valores da qualidade de cada vértice, como mostrado na linha 3.3 do Algoritmo 3.

No primeiro passo, mostrado a partir da linha 3.4, para cada vértice do grafo, a melhor qualidade já vista até o momento é atualizada em cada um dos vértices de superconjuntos ligados a ele. No segundo passo, mostrado a partir da linha 3.12, uma *flag* é marcada para indicar se aquele vértice foi o melhor encontrado, uma vez que o melhor valor deve ser reiniciado para fazer o caminho inverso. Nesse passo, os níveis do grafo são percorridos do último para o primeiro, salvando os melhores valores de superconjunto vistos até o momento. Quando se descobre que um padrão apresenta o valor do melhor superconjunto visto até agora e o mesmo já havia sido marcado como o melhor subconjunto, ele é adicionado ao conjunto B de *biclusters* encontrados.

Algoritmo 3 Enumera padrões candidatos

Input
vértices, $V \subset 2^{\mathcal{C}}$
sucessores, $succ \in (2^{\mathcal{C}})^V$
$L, \{v \mid v \subseteq V\}$ *tal que* $|L| = (|\mathcal{R}| - 2 \times (k-1))$
qualidade de cada vértice, $q \in \mathbb{R}_+^V$
Output
biclusters enumerados, $B \subset 2^{\mathcal{C}}$

1: **function** ENUMERATEPATTERNS($V, succ, L, q$))
2: $E \leftarrow \emptyset$
3: $b \leftarrow q$
4: **for** $i \leftarrow 0, |L|$ **do** ▷ Descendo pelo grafo
5: **for all** $C \in L_i$ **do**
6: **for all** $N \in succ(C)$ **do**
7: $b(N) \leftarrow max\{b(C), b(N)\}$
8: **end for**
9: **end for**
10: **end for**
11:
12: **for** $i \leftarrow |L| - 1, 0$ **do** ▷ Subindo pelo grafo
13: **for all** $C \in L_i$ **do**
14: $possible \leftarrow q(C) \geq b(C)$
15: $b(C) \leftarrow q(C)$
16: **for all** $N \in succ(C)$ **do**
17: $b(C) \leftarrow max\{b(C), b(N)\}$
18: **end for**
19: **if** $possible \wedge (q(C) = b(C)$ **then**
20: $B \leftarrow B \cup \{C\}$
21: **end if**
22: **end for**
23: **end for**
24: **return** B
25: **end function**

VI. HEURÍSTICA GULOSA DE ORDENAÇÃO DE LINHAS E COLUNAS

A heurística gulosa de ordenação de linhas e colunas busca agrupar *biclusters* começando a partir dos maiores valores da matriz. As linhas e colunas são ordenadas em função do valor máximo localizado nas mesmas, de forma a tentar agrupar os maiores valores no canto da matriz. O *bicluster* inicial é definido como o elemento no canto da matriz. A partir disso, linhas e colunas adjacentes são adicionadas ao *bicluster*, utilizando um critério de escolha que leva em consideração qual nova adição aumentará menos a variância do *bicluster*. Quando não é possível adicionar uma nova linha ou coluna sem ultrapassar um certo limite de variância, ou quando não há mais linhas ou colunas para retornar, o *bicluster* é retornado.

O Algoritmo 4 obtém no máximo max *biclusters*. Ele primeiramente ordena as linhas, com o peso de cada linha

sendo o valor do maior elemento da linha, e depois as colunas. Caso o maior elemento da matriz seja $-\infty$, uma constante atribuída para os elementos que já foram analisados pelo algoritmo, todos os *biclusters* já foram encontrados.

O Algoritmo 5 mostra como os *biclusters* são encontrados. Primeiramente, a variância do *bicluster* ao se adicionar uma nova linha e a variância ao se adicionar uma nova coluna são calculadas. Caso as duas excedam um um determinado limite, o algoritmo retorna. Caso contrário, deve-se decidir se uma nova linha ou uma nova coluna serão adicionadas ao *bicluster*. Deseja-se adicionar a linha ou coluna que menos aumenta a variância do *bicluster*, porém, para evitar que um *bicluster* seja composto somente por uma linha ou por uma coluna, existe uma margem extra de variância aceitável para adicionar uma nova linha caso o *bicluster* possua menos linhas do que colunas, e o contrário também é verdade.

Ao se adicionar uma nova linha ou coluna, os novos elementos adicionados são marcados com $-\infty$ na matriz D.

Algoritmo 4 Heurística gulosa de ordenação de linhas e colunas

Input

$\mathcal{C}$, o conjunto de todas as colunas,

$\mathcal{R}$, o conjunto de todos os linhas,

D, a matriz de valores, $D \in \mathbb{R}^{(\mathcal{R}\times\mathcal{C})}$,

t, o *threshold* da variância para adicionar uma nova linha ou coluna ao *bicluster*, $t \in \mathbb{R}^+$

max, o número máximo de *biclusters* a serem encontrados

Output

B_R, conjunto dos conjuntos de linhas dos *biclusters* encontrado

B_C, conjunto dos conjuntos de colunas dos *biclusters* encontrado

```
1:  function GreedySortHeuristic(C, R, D, t, max)
2:      B_R, B_C ← (∅, ∅)
3:      while |B| < max do
4:          for i ← 1, 2 do
5:              S ← getSortingVector(D)
6:              R ← weightedRowSort(R, S)
7:              D ← D^T
8:              swap(R, C)
9:          end for
10:         if D_{|R|,|C|} = −∞ then
11:             return B_R, B_C
12:         end if
13:         br, bc ← findBicluster(R, C, D, t)
14:         B_R ← B_R ∪ {br}
15:         B_C ← B_C ∪ {bc}
16:     end while
17:     return B_R, B_C
18: end function
```

Algoritmo 5 Busca de *biclusters*

Input

$\mathcal{C}$, o conjunto de todas as colunas,

$\mathcal{R}$, o conjunto de todos os linhas,

D, a matriz de valores, $D \in \mathbb{R}^{(\mathcal{R}\times\mathcal{C})}$,

t, o *threshold* da variância para adicionar uma nova linha ou coluna ao *bicluster*, $t \in \mathbb{R}^+$

max, o número máximo de *biclusters* a serem encontrados

Output

b_R, conjunto das linhas do *bicluster* encontrado

b_C, conjunto das colunas do *bicluster* encontrado

```
1:  function findBicluster(C, R, D, t)
2:      I, J ← |R| − 1, |C| − 1
3:      b_R, b_C ← {R_I}, {C_J}
4:      D_{I,J} ← −∞
5:      size ← 1
6:      while I > 0 & J > 0 do
7:          row, col ← R_I, C_J
8:          row_Var ← Variance(D, b_R, b_C, row − 1)
9:          col_Var ← Variance(D, b_R, b_C, col − 1)
10:         if row_Var > t & col_Var > t then
11:             if size > MAX_SIZE then
12:                 return b_R, b_C
13:             else
14:                 return ∅, ∅
15:             end if
16:         end if
17:         if |b_c| > |b_r| & (row_Var − col_Var) < LIMIT then
18:             choice ← ROWS
19:         else if |b_r| > |b_c| & (col_Var − row_Var) < LIMIT then
20:             choice ← COLS
21:         else
22:             if row_Var < col_Var then
23:                 choice ← ROWS
24:             else
25:                 choice ← COLS
26:             end if
27:         end if
28:         if choice = ROWS then
29:             b_R ← b_R ∪ {row − 1}
30:             I ← I − 1
31:             D_{row−1,b_C} ← {−∞, ..., −∞}
32:         else if choice = COLS then
33:             b_C ← b_C ∪ {col − 1}
34:             J ← J − 1
35:             D_{b_R,col−1} ← {−∞, ..., −∞}
36:         end if
37:         size ← size + 1
38:     end while
39:     return b_R, b_C
40: end function
```

VII. Metaheurística baseada em GRASP

A metaheurística GRASP (do inglês *greedy randomized search procedure*) foi baseada na descrita por Dharan e Nair [13] e está ilustrada no Algoritmo 6.

Algoritmo 6 Metaheurística baseada em GRASP

Input
D, a matriz de valores,
I, a solução inicial,
t, o limite superior para a variância do *bicluster*,
n, o número de iterações por rodada da busca,
max, o número máximo de *biclusters* a serem encontrados
Output
B_R, conjuntos de linhas dos *biclusters* encontrados,
B_C, conjuntos de colunas dos *biclusters* encontrados

```
1: function GraspHeuristic(D,I,t,n,max)
2:     B_R, B_C ← (∅, ∅)
3:     while |B| < max do
4:         bestSol ← I
5:         for i ← 0, n do
6:             initSol ← randGreedyHeuristic(D,t)
7:             newSol ← localSearch(D,[initSol])[0]
8:             newSol_Var ← Variance(newSol)
9:             bestSol_Var ← Variance(bestSol)
10:            if newSol_Var < bestSol_Var then
11:                bestSol ← newSol
12:            end if
13:        end for
14:        B_R ← B_R ∪ {bestSol_R}
15:        B_C ← B_C ∪ {bestSol_C}
16:    end while
17:    return B_R, B_C
18: end function
```

A. Heurística construtiva aleatória

A fase de construção, ilustrada no Algoritmo 7, utiliza uma versão modificada do procedimento `GreedySortHeuristic`, utilizado na heurística gulosa descrita na Seção VI. Neste caso, o procedimento ordena as linhas e colunas da matriz de forma aleatória, mas o restante do algoritmo permanece inalterado. O resultado encontrado é utilizado como solução inicial para a etapa de busca local, descrita a seguir.

B. Busca local

O algoritmo de busca local é utilizado para melhorar uma solução inicial obtida de outra forma. Essa busca local, apresentada no Algoritmo 8, utiliza uma função de vizinhança chamada aqui de `1diff`. Essa vizinhança consiste nas soluções que diferem em uma linha ou em uma coluna da solução inicial. Cada solução que difere em um elemento da inicial é testada até que se encontre a primeira solução com variância inferior à original. A nova solução é então definida como a solução vigente, e o processo se repete até que não exista nenhuma solução melhor do que a vigente. O Algoritmo 9 apresenta a função de vizinhança.

Algoritmo 7 Heurística gulosa aleatória

Input
D, a matriz de valores,
t, o limite superior para a variância do *bicluster*
Output
B_R, conjunto dos conjuntos de linhas dos *biclusters* encontrado,
B_C, conjunto dos conjuntos de colunas dos *biclusters* encontrado

```
1: function randGreedyHeuristic(D,t)
2:     return greedySortHeuristic(D_C,D_R,D,t,1)
3: end function
```

Algoritmo 8 Busca local

Input
D, a matriz de valores,
I_C, conjunto de colunas de uma solução inicial
I_R, conjunto de linhas de uma solução inicial
$\mathcal{C}$, o conjunto de todas as colunas,
$\mathcal{R}$, o conjunto de todos os linhas,

Output
B_R, conjunto dos conjuntos de linhas dos *biclusters* encontrado,
B_C, conjunto dos conjuntos de colunas dos *biclusters* encontrado

```
1: function LocalSearch(D, I_C, I_R, C, R)
2:     B_R, B_R ← I_R, I_C
3:     while true do
4:         b_R, b_C ←get1DiffNeighbor(D, B_C, B_R, C, R)
5:         if b_R = ∅ & b_C = ∅ then
6:             return B_R, B_C
7:         end if
8:         B_R ← b_R
9:         B_C ← b_C
10:    end while
11: end function
```

VIII. Metaheurística baseada em busca tabu

A metaheurística baseada em busca Tabu, apresentada no Algoritmo 10, utiliza uma lista Tabu de no máximo max_size elementos, que guarda as últimas soluções encontradas. A melhor solução encontrada durante todas as iterações é retornada no final. Uma versão alterada da busca local utilizada com a metaheurística baseada em GRASP foi aplicada para a busca Tabu. Essa busca local alterada, apresentada no Algoritmo 11, utiliza o melhor vizinho encontrado que não está na lista tabu, ao contrário do primeiro vizinho que melhora a solução atual.

Algoritmo 9 1-diff Neighborhood

Input
D, a matriz de valores,
I_C, conjunto de colunas de uma solução inicial
I_R, conjunto de linhas de uma solução inicial
$\mathcal{C}$, o conjunto de todas as colunas,
$\mathcal{R}$, o conjunto de todos os linhas,

function GET1DIFFNEIGHBOR(D, I_C, I_R, C, R)
  $orig\_var \leftarrow$ VARIANCE(D, I_R, I_C)
  for all $new\_r \in (R \setminus I_R)$ **do**
    for all $old\_r \in I_R$ **do**
      $b_R \leftarrow (B_R \setminus \{old\_R\}) \cup \{new\_r\}$
      $var \leftarrow$ VARIANCE(D, b_R, I_C)
      if $var < orig\_var$ **then**
        return b_R, I_C
      end if
    end for
  end for
  for all $new\_c \in (R \setminus I_C)$ **do**
    for all $old\_c \in I_C$ **do**
      $b_R \leftarrow (B_C \setminus \{old\_c\}) \cup \{new\_c\}$
      $var \leftarrow$ VARIANCE(D, I_R, b_C)
      if $var < orig\_var$ **then**
        return I_R, b_C
      end if
    end for
  end for
  return $\emptyset, \emptyset$
end function

IX. EXPERIMENTAÇÃO

As heurísticas propostas foram avaliadas utilizando-se quatro bases de dados disponíveis no Kaggle[1] ou no repositório de aprendizado de máquina da Universidade da Califórnia, Irvine (UCI)[2].

Expressão de proteínas em camundongos (BD1)[3]: A base contém dados sobre os níveis de expressão de 77 proteínas medidas no córtex cerebral de camundongos de diferentes grupos experimentais. A matriz é formada por 1080 linhas (medições) e 77 colunas (proteínas).

Proteomas de câncer de mama (BD2)[4]: A base de dados contém o perfil de proteoma de 77 amostras de câncer de mama. A matriz é formada por 12553 linhas (níveis de expressão de proteínas) e 83 colunas (proteínas).

Transações comerciais (BD3)[5]: A base contém dados semanais sobre as vendas de uma empresa. Foram catalogados 811 produtos (linhas) e 106 valores (52 semanas, valores normalizados de cada semana e valores máximos e mínimos).

[1]https://www.kaggle.com
[2]https://archive.ics.uci.edu/ml
[3]https://www.kaggle.com/ruslankl/mice-protein-expression/home
[4]https://www.kaggle.com/piotrgrabo/breastcancerproteomes/home
[5]https://archive.ics.uci.edu/ml/datasets/Sales_Transactions_Dataset_Weekly

Algoritmo 10 Busca Tabu

Input
D, a matriz de valores,
I_C, conjunto de colunas de uma solução inicial
I_R, conjunto de linhas de uma solução inicial
$\mathcal{C}$, o conjunto de todas as colunas,
$\mathcal{R}$, o conjunto de todos os linhas,
max_it, o número de iterações a serem executadas pelo algoritmo,
max_size, o tamanho máximo da lista Tabu.
Output
B_R, conjunto dos conjuntos de linhas dos *biclusters* encontrado,
B_C, conjunto dos conjuntos de colunas dos *biclusters* encontrado

function TABUSEARCH(D, I_C, I_R, C, R)
  $tabuList \leftarrow$ PUSH((I_R, I_C))
  $b_R, b_R \leftarrow I_R, I_C$
  $best\_var \leftarrow$ VARIANCE(D, b_R, b_C)
  for $i \leftarrow 1, max\_it$ **do**
    $b_R, b_C \leftarrow$GETDIFFTABU($D, b_C, b_R, C, R, tabuList$)
    $new\_var \leftarrow$ VARIANCE(D, b_R, b_C)
    if $new\_var < best\_var$ **then**
      $B_R, B_C \leftarrow b_R, b_C$
      $best\_var \leftarrow new\_var$
    end if
    $tabuList \leftarrow$ PUSH((b_R, b_C))
    if $|tabuList| > max\_size$ **then**
      $tabuList \leftarrow$ REMOVEFIRST($tabuList$)
    end if
  end for
end function

Análise química de vinhos (BD4)[6]: A base contém dados sobre a análise química de 178 amostras de vinho (linhas). Foram consideradas 13 substâncias (colunas) de cada uma dessas amostras.

A distribuição dos valores das bases de dados está representada na Tabela I.

Tabela I: Distribuição dos valores das bases de dados.

Base	Mínimo	Máximo	Média	Variância	Linhas	Colunas
BD1	-0,06	8,50	0,72	0,73	1080	77
BD2	-24,55	17,62	-2,05	32,19	12553	83
BD3	0	73	4,71	93,21	811	106
BD4	0,13	1680	69,13	46546,42	178	13

A. Comparação com heurísticas da literatura

Os resultados das heurísticas propostas foram comparados com os obtidos pelas heurísticas de Cheng e Church [8], Teng e Chan [10], de França e Zuben [14] e Zhou e Hao [15].

[6]https://archive.ics.uci.edu/ml/datasets/Wine

Algoritmo 11 1-diff Neighborhood Tabu Search

Input
D, a matriz de valores,
I_C, conjunto de colunas de uma solução inicial
I_R, conjunto de linhas de uma solução inicial
$\mathcal{C}$, o conjunto de todas as colunas,
$\mathcal{R}$, o conjunto de todos os linhas,
$tabuList$, a listaTabu,

1: **function** GETDIFFTABU($D, I_C, I_R, C, R, tabuList$)
2: $B_R, B_C \leftarrow \emptyset, \emptyset$
3: $min\_var \leftarrow$ VARIANCE(D, I_R, I_C)
4: **for all** $new\_r \in (R \setminus I_R)$ **do**
5: **for all** $old\_r \in I_R$ **do**
6: $b_R \leftarrow (B_R \setminus \{old\_R\}) \cup \{new\_r\}$
7: **if** $(b_R, I_C) \notin tabuList$ **then**
8: $var \leftarrow$ VARIANCE(D, b_R, I_C)
9: **if** $var < min\_var$ **then**
10: $B_R, B_C \leftarrow b_R, I_C$
11: $min\_var \leftarrow var$
12: **end if**
13: **end if**
14: **end for**
15: **end for**
16: **return** B_R, B_C
17: **end function**

Como esta última metaheurística foi proposta com o objetivo de resolver um problema de biclique sem peso nas arestas, modificamos as bases de dados de forma que os valores acima da média foram substituídos por 1, correspondendo à presença de uma aresta entre a linha e coluna correspondentes. Por outro lado, valores abaixo da média foram substituídos por 0, significando que não existe aresta.

B. Preprocessamento de dados

Algumas bases continham dados faltando, que foram substituídos por valores selecionados aleatoriamente a partir de uma distribuição uniforme entre 2 vezes o valor mínimo e o máximo da base, conforme sugerido por [8]. Além disso, as colunas não numéricas foram removidas.

C. Hiperparâmetros

Os hiperparâmetros de cada heurística foram ajustados para cada base de dados individualmente, comparando-se a qualidade dos *biclusters* obtidos. Apenas os melhores resultados de cada base serão apresentados.

D. Implementação e execução dos experimentos

A implementação das heurísticas foi feita utilizando-se a linguagem C++11. Todos os experimentos foram executados em um Intel® Core™ i5-3450 CPU @ 3.10GHz x 4, com 4GB de memória RAM. O tempo de execução de cada heurística foi medido utilizando-se a função `clock()` da biblioteca `ctime` de C++.

E. Resultados

O tempo de execução e o índice de qualidade do melhor *bicluster* encontrado por cada uma das heurísticas testadas estão representados na Tabela II. Já a Tabela III sumariza o número de linhas e colunas e a variâncias de tais *biclusters*.

Como podemos observar, os resultados encontrados pelas heurísticas e metaheurísticas implementadas variaram muito. A primeira heurística proposta, de enumeração de subconjuntos, foi mais rápida que ambas as da literatura, apesar de ter sido mais lenta que a segunda heurística proposta. Para a base de dados BD1, a qualidade só foi inferior à obtida pela heurística de Cheng e Church. Já para as bases BD2, BD3 e BD4, esta heurística encontrou um *bicluster* com índice de qualidade superior às outras. Entretanto, observamos que tais índices de qualidade elevados refletem *biclusters* pequenos, frequentemente formados por apenas uma linha, o que leva a variâncias muito pequenas. Os *biclusters* maiores apresentaram alta variância, de forma que o índice de qualidade foi baixo. Esse comportamento também foi observado para a heurística de Teng e Chan. As bases BD1 e BD2 contêm dados de medições de níveis de proteínas, o que está diretamente relacionado a expressão gênica. Dessa forma, *biclusters* formados por apenas uma linha (ou proteína, ou gene) têm baixa qualidade biológica, uma vez que é esperada a existência de grandes *clusters* de genes com funções relacionadas e que exibem padrões de expressão semelhantes sob várias condições experimentais [16].

A segunda heurística proposta, de ordenação de linhas e colunas, foi a mais rápida dentre as testadas. No caso da base BD1, o melhor *bicluster* encontrado é cerca de 7 vezes maior que o da enumeração de subconjuntos, mas ainda pode ser considerado pequeno em relação aos encontrados pelas duas heurísticas da literatura. O pequeno tamanho deste *bicluster* justifica sua pequena variância e índice de qualidade relativamente elevado, tendo superado a heurística de Teng e Chan. Já no caso da base BD2, o melhor *bicluster* encontrado foi extremamente grande, incluindo todas as colunas e cerca de 99% das linhas, o que levou a uma variância alta e qualidade baixa. Tal comportamento também não é esperado de dados relacionados a expressão gênica, uma vez que apenas pequenos conjuntos de genes relacionados exibem padrões de expressão semelhantes [16]. O *bicluster* encontrado para a base BD3 é consideravelmente maior do que os *biclusters* encontrados para outras bases, porém sua qualidade não supera a qualidade das da heurística de Cheng e Church, que encontrou um *bicluster* grande, mas com menor variância. O *bicluster* encontrado para a base BD4 possui um tamanho pequeno, que é refletido na sua qualidade baixa.

A baixa qualidade, em termos biológicos, dos *biclusters* encontrados pela heurística de enumeração de subconjuntos pode ser justificada pelo fato de que ela tenta apenas maximizar a distância entre os elementos do *bicluster* e os demais, não levando em consideração a variância do *bicluster*. Dessa forma, podem ser obtidos *biclusters* ruins. Por sua vez, a heurística de ordenação de linhas e colunas insere linhas e

Tabela II: Comparação de qualidade dos *biclusters*.

	BD1		BD2		BD3		BD4	
	Qualidade	Tempo (s)	Qualidade	Tempo (s)	Qualidade	Tempo (s)	Qualidade	Tempo (s)
Cheng e Church	265,238	2,367	2,901	365,889	169,819	3,506	0,554	0,086
Teng e Chan	8,504	20,871	368,116	3831,62	0,216	41,658	7,978	0,081
França e Zuben	28,845	429,942	-	-	2,159	137,473	0,198	12,352
Zhou e Hao	4,979	40,016	13,160	40,013	0,064	40,023	$9{,}5E^{-6}$	40,008
Enum. de subconj.	160,36	1,971	**500,815**	94,239	196,097	7,483	10,337	0,039
Ord. linhas e cols.	30,585	0,032	0,436	3,265	60,592	0,019	4,024	0,023
Busca tabu	77,408	232,858	36,248	3891,69	44,599	1278,62	3,225	0,736
GRASP	**13086,7**	0,001	29,492	95,758	**209,522**	1,559	**59,109**	0,018

Tabela III: Comparação de tamanho e variância dos *biclusters*.

	BD1 (1080 x 77)			BD2 (12553 x 83)			BD3 (811 x 106)			BD4 (178 x 13)		
	Linhas	Colunas	Var.	Linhas	Colunas	Var.	Linhas	Colunas	Var.	Linhas	Colunas	Var.
Cheng e Church	601	42	0,039	8101	42	4,391	271	40	0,055	172	9	13,264
Teng e Chan	403	4	0,868	1	9	$1{,}2E^{-29}$	46	54	36,189	177	1	0,649
França e Zuben	1080	70	0,389	-	-	-	807	52	4,929	75	11	33,772
Zhou e Hao	22	22	1,267	76	76	0,658	54	54	124,032	3	2	$18{,}7E^{4}$
Enum. de subconj.	3	45	0,031	2	12	0,006	1	86	0.023	178	1	0.501
Ord. linhas e cols.	13	77	0,226	12470	83	31,767	105	106	0,154	5	1	0,400
Busca tabu	17	2	0,046	9	83	0,183	105	106	0,209	6	1	0,556
GRASP	10	10	$3{,}5E^{-4}$	1	83	0,150	3	106	0,028	4	3	0,042

colunas no *bicluster* de forma gulosa e não há a opção de remover um elemento que já foi inserido. Assim, o tamanho do *bicluster* aumenta enquanto houver linha ou coluna cuja inserção não aumente a variância além do limite tolerado. Isso faz com que não seja possível controlar o tamanho e a variância dos *biclusters* encontrados, levando qualidades muito diferentes para bases de dados diferentes.

A primeira metaheurística proposta, que consiste em uma busca tabu, foi mais lenta que todas as outras heurísticas e metaheurísticas, com exceção da metaheurística de França e Zuben, que apresentou um tempo de execução maior para as bases BD1 e BD4 e não conseguiu encontrar um resultado na base BD2. O *bicluster* encontrado para a base BD1 foi pequeno, com somente 2 colunas, e foi de uma qualidade inferior aos resultados obtidos pelas heurísticas de Cheng e Church, enumeração de subconjuntos e pela metaheurística proposta de GRASP. Os resultados obtidos para a base BD2 tiveram uma qualidade inferior às heurísticas de Cheng e Church e enumeração de subconjuntos, e, apesar de ser de um tamanho pequeno, o *bicluster* obtido foi maior do que o *bicluster* obtido pela enumeração de subconjuntos. O *bicluster* obtido para a base BD3 possui uma grande área, porém foi de uma qualidade inferior às outras heurísticas e metaheurística propostas e à heurística de Cheng e Church. Já o resultado obtido para a base BD4, foi de um *bicluster* pequeno, que resultou em uma qualidade ruim.

A segunda metaheurística proposta, GRASP, obteve as melhores qualidades para as bases BD1, BD3 e BD4, porém, os *biclusters* obtidos para essas bases foram pequenos, de modo que sua pequena variância justifica a alta qualidade, mas não implica em uma significância biológica relevante para as bases de expressão gênica. Em relação ao tempo, essa metaheurística teve a segunda execução mais rápida para quase todas as bases, ficando atrás da ordenação de linhas e colunas em todas, e atrás da enumeração de subconjuntos na base BD2.

A busca tabu utiliza uma busca local cuja vizinhança não possui *biclusters* de tamanhos diferentes. Assim, uma solução inicial é gerada a partir da heurística de ordenação de linhas e colunas, e melhorada através da busca local proposta. O tamanho do *bicluster* não altera durante esse processo, de modo que só é possível melhorar a variância. Isso justifica a baixa qualidade dos *biclusters* em termos biológicos. Além disso, como a solução inicial é gerada pela heurística de ordenação de linhas e colunas, que possui o pior desempenho em questão de qualidade, a qualidade final é inferior a muitas das outras heurística e metaheurísticas analisadas.

A metaheurística de GRASP utiliza como solução inicial uma versão modificada da heurística de ordenação de linhas e colunas. Essa heurística modificada seleciona linhas e colunas de maneira aleatória, sem que exista a opção de remover um elemento já inserido. Além disso, a heurística é gulosa, de forma que, mesmo que exista um elemento melhor a ser inserido, a heurística para quando um novo elemento faz o *bicluster* ultrapassar o *threshold* permitido. Dessa forma, os *biclusters* resultantes podem tem um tamanho pequeno.

X. Conclusão

Os resultados apresentados demostram que heurísticas para *biclustering* são capazes de encontrar um grande variedade de soluções para o mesmo conjunto de dados. A avaliação da qualidade dos *biclusters* é um ponto crucial na análise de dados, não sendo possível encontrar uma métrica adequada a todas as bases de dados e possíveis aplicações da técnica.

No caso de dados de expressão gênica, o tamanho ideal de um *bicluster* deve ser definido por um especialista, que é capaz de associar às linhas e colunas encontradas com os genes e condições relacionadas. O mesmo ocorre para outros tipos de bases de dados, de forma que uma métrica que leva em consideração apenas o tamanho e variância dos *biclusters* pode não ser adequada em algumas situações.

REFERÊNCIAS

[1] B. Mirkin, *Mathematical Classification and Clustering*, ser. Nonconvex Optimization and Its Applications. Springer US, 1996. [Online]. Available: https://books.google.com.br/books?id=brzLe4X4ypEC

[2] J. A. Hartigan, "Direct Clustering of a Data Matrix," *Journal of the American Statistical Association*, vol. 67, no. 337, pp. 123–129, 1972. [Online]. Available: http://dx.doi.org/10.2307/2284710

[3] S. C. Madeira and A. L. Oliveira, "Biclustering algorithms for biological data analysis: A survey," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 1, no. 1, pp. 24–45, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1109/TCBB.2004.2

[4] M. Dawande, P. Keskinocak, and S. Tayur, "On the biclique problem in bipartite graphs," *GSIA Working Paper*, 1996.

[5] A. Prelić, S. Bleuler, P. Zimmermann, A. Wille, P. Bühlmann, W. Gruissem, L. Hennig, L. Thiele, and E. Zitzler, "A systematic comparison and evaluation of biclustering methods for gene expression data," *Bioinformatics*, vol. 22, no. 9, pp. 1122–1129, 2006.

[6] K. Eren, M. Deveci, O. Küçüktunç, and Ü. V. Çatalyürek, "A comparative analysis of biclustering algorithms for gene expression data," *Briefings in Bioinformatics*, vol. 14, no. 3, pp. 279–292, 2012.

[7] B. Pontes, R. Giráldez, and J. S. Aguilar-Ruiz, "Biclustering on expression data: A review," *Journal of Biomedical Informatics*, vol. 57, pp. 163–180, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jbi.2015.06.028

[8] Y. Cheng and G. M. Church, "Biclustering of expression data." in *Ismb*, vol. 8, no. 2000, 2000, pp. 93–103.

[9] K. Y. Yip, D. W. Cheung, and M. K. Ng, "Harp: A practical projected clustering algorithm," *IEEE Transactions on knowledge and data engineering*, vol. 16, no. 11, pp. 1387–1397, 2004.

[10] L. Teng and L. Chan, "Discovering biclusters by iteratively sorting with weighted correlation coefficient in gene expression data," *Journal of Signal Processing Systems*, vol. 50, no. 3, pp. 267–280, Mar 2008. [Online]. Available: https://doi.org/10.1007/s11265-007-0121-2

[11] M. Pavan and M. Pelillo, "A new graph-theoretic approach to clustering and segmentation," in *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, ser. CVPR'03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 145–152. [Online]. Available: http://dl.acm.org/citation.cfm?id=1965841.1965859

[12] K. Bryan, P. Cunningham, and N. Bolshakova, "Application of simulated annealing to the biclustering of gene expression data," *IEEE transactions on information technology in biomedicine*, vol. 10, no. 3, pp. 519–525, 2006.

[13] S. Dharan and A. S. Nair, "Biclustering of gene expression data using reactive greedy randomized adaptive search procedure," *BMC bioinformatics*, vol. 10, no. 1, p. S27, 2009.

[14] F. O. de França and F. J. Von Zuben, "Extracting additive and multiplicative coherent biclusters with swarm intelligence," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE, 2011, pp. 632–638.

[15] Y. Zhou and J.-K. Hao, "Combining tabu search and graph reduction to solve the maximum balanced biclique problem," *arXiv preprint arXiv:1705.07339*, 2017.

[16] J. M. Stuart, E. Segal, D. Koller, and S. K. Kim, "A gene-coexpression network for global discovery of conserved genetic modules," *Science*, vol. 302, no. 5643, pp. 249–255, 2003. [Online]. Available: http://science.sciencemag.org/content/302/5643/249