Relatório Detalhado sobre Algoritmos para o MWC

Origem: Gerado pelo NotebookLM

Assunto: Problema da Clique de Peso Máximo (MWC)

Este relatório fornece uma descrição completa e, sempre que possível, o pseudocódigo para cada algoritmo destinado a resolver o **Problema da Clique de Peso Máximo (MWC)**.

O MWC é um problema de otimização combinatória classificado como *NP-Hard*. A abordagem para resolvê-lo depende da prioridade: garantir a solução ótima (algoritmos exatos) ou encontrar uma solução de alta qualidade rapidamente (algoritmos heurísticos).

Secção 1: Algoritmos Exatos

Algoritmos exatos exploram o espaço de busca exaustivamente, geralmente usando o esquema *Branch-and-Bound* (BnB), e garantem que a solução encontrada é a ótima.

1.1 WLMC (Weighted Large Maximum Clique)

O WLMC é um algoritmo exato de BnB introduzido em 2017 e projetado para grafos grandes e esparsos.

Descrição Completa:
O WLMC exige que os pesos dos vértices sejam inteiros. As suas características principais são as estratégias especiais de pré-processamento e de divisão de vértices (vertex splitting).

- **Pré-processamento (InitializeWLMC):** Esta fase recebe o grafo $G = (V, E, w)$ e um limite inferior lb (*Lower Bound*). Calcula uma ordenação inicial dos vértices (O_0) e uma clique inicial (C_0). O WLMC usa uma ordenação baseada no grau dos vértices (o vértice com o menor grau é removido primeiro), que tem provado ser eficaz para o MC em grafos grandes. O pré-processamento também reduz o grafo, removendo todos os vértices v para os quais o peso da sua vizinhança inclusiva $w(N[v])$ não é maior do que o lb.
- **Estratégia BnB:** A busca recursiva (*SearchMaxWCliqueWLMC*) utiliza a ordenação O_0 e um limite superior UB dinâmico para podar a árvore de busca.
- **Divisão de Vértices (UP&SplitWLMC):** No nó de ramificação, o algoritmo divide os vértices restantes em dois conjuntos disjuntos, A (não ramificados) e B (ramificados). O WLMC utiliza uma técnica de *splitting* (divisão) e propagação de Conjuntos Independentes Unitários (ISs) para determinar um limite superior mais preciso ub para o peso da clique que pode ser construída a partir dos ISs. Se este limite ub for inferior ou igual ao limite de peso $t = w(\hat{C}) - w(C)$, o vértice de ramificação é

removido de B.

Pseudocódigo (Função de Inicialização):

```
Algoritmo 2: InitializeWLMC(G, lb)
Input: Grafo G = (V, E, w), limite inferior lb de w_v(G)
Output: C_0, O_0, G' (Grafo reduzido)

1:  H <- Cópia de G
2:  O_0 <- Vazio
3:  while H != Vazio do
4:      v_i <- Vértice em H com o menor grau deg(v_i)
5:      O_0 <- O_0 U {v_i}
6:      if deg(v_i) == |V(H)| - 1 then
7:          C_0 <- H; Ordem arbitrária em C_0
8:          break
9:      end if
10:     H <- H \ {v_i}; Atualiza graus dos vizinhos
11: end while
12: if w(C_0) > lb then lb <- w(C_0)
13: G' <- Cópia de G
14: V' <- {v in V | w(N[v]) > lb}
15: G' <- G[V']
16: return C_0, O_0, G'
```

1.2 TSM-MWC (Two-Stage MaxSAT MWC)

O TSM-MWC é um solucionador exato que utiliza o *MaxSAT Reasoning* em duas fases para calcular limites superiores mais rigorosos (UB).

Descrição Completa:
O TSM-MWC é comparável ao WLMC em muitas partes, mas difere significativamente no processo de determinação dos ramos. A função principal (GetBranchesTSM-MWC) chama duas subfunções que realizam as etapas de raciocínio MaxSAT.

- **Primeira Fase (MaxSAT Binário - BinaryMaxSATTSM-MWC):** Calcula um conjunto inicial de vértices de ramificação B, um conjunto de ISs Π, e um limite superior inicial ub.
- **Segunda Fase (MaxSAT Ordenado - OrderedMaxSATTSM-MWC):** Se a primeira fase não indicar que a busca é fútil, esta fase é aplicada. Ela refina o conjunto B olhando para diferentes subconjuntos de ISs em conflito com o vértice de ramificação b_i. O objetivo é integrar uma fração suficientemente grande do peso de b_i em Π tal que o limite de peso t seja satisfeito, permitindo remover b_i de B.

Pseudocódigo (Fase 2 de Ramificação):

Algoritmo 10: OrderedMaxSATTSM-MWC(G, t, O, B, Pi, ub)
Input: G, Limite t, Ordem O, Ramificação B, ISs Pi, ub

```
1:  B = {b_1, ..., b_|B|}
2:  for i := |B| downto 1 do
3:      Pi' := Pi; delta := w(b_i)
4:      // 1. Seja S_1, S_2, ..., S_k os ISs em Pi que não contêm vizinho de b_i
5:      for j := 1 to k do
6:          // 1.1 Adiciona b_i com peso máximo de S_j
            S_j := S_j U {b_i^(w(S_j))}
7:          // 1.2
            delta := delta - w(S_j)
8:      end for
9:      // 2. Seja U_1, U_2, ..., U_r os ISs contendo exatamente um vizinho de b_i cada
10:     for j := 1 to r do
11:         ... (Processa subconjuntos conflitantes de 3 membros, usando estratégia
UP&SplitWLMC)
12:         // 3.2
            delta := delta - beta;
            if ub + delta <= t then break
13:     end for
14:     // 4.
        if ub + delta <= t then
15:         B := B \ {b_i}; ub := ub + delta
16:     else
            Pi := Pi'
17: end for
18: return B
```

1.3 MWCRedu (Maximum Weight Clique Reduction)

O MWCRedu é um algoritmo exato recente (2023) que utiliza várias estratégias de redução de grafo, incluindo regras de redução que exploram estruturas locais.

Descrição Completa:
O MWCRedu funciona em dois estágios:

1. **Inicialização:** Determina a melhor clique inicial $\hat{C}$ usando a estratégia de geração de clique inicial de *InitializeWLMC*.
2. **Redução de Grafo (ReduceMWCRedu):** Aplica exaustivamente um conjunto de regras de redução exatas. Estas regras incluem: *Twin Reduction* (contração de vértices adjacentes com a mesma vizinhança fechada), *Domination Reduction* (remoção ou

modificação de arestas/pesos baseada em vizinhanças), e *Simplicial Vertex Removal*. A aplicação das regras pode ser limitada dinamicamente pelo grau dos vértices (o *vertexDegreeLimit* é aumentado iterativamente) para priorizar reduções mais eficientes em vértices de menor grau.

3. **Resolução Exata:** O grafo reduzido G' é passado ao solucionador exato TSM-MWC para calcular a solução final.

O MWCRedu demonstrou ser significativamente mais rápido que o TSM-MWC em grafos de redes de ruas (OSM) e grafos hiperbólicos aleatórios (RHG).

Pseudocódigo (Procedimento Principal e Redução):

```
Algoritmo 11: MWCRedu
1:  C_hat := InitializeWLMC(G, 0);
2:  ReduceMWCRedu(G, C_hat, true);
3:  return TSM-MWC(G, C_hat);

Algoritmo 12: ReduceMWCRedu(G, C_hat, lim)
1:  if lim then vertexDegreeLimit := 0.1;
2:  foreach r_i do initialize D_i;
3:  repeat
4:      // 1.
        foreach reduction rule r_i do
5:          if (r_i not paused) AND (D_i != Vazio) then
6:              apply r_i on every v in D_i; update D_i;
7:              if reduction rate not achieved then pause r_i;
8:              update C_hat via local search;
9:          else if (r_i paused) AND (G reduced enough) then unpause r_i;
10:     // 2.
        if lim AND (all reductions paused) then
11:         vertexDegreeLimit := vertexDegreeLimit + 0.1;
12:         foreach r_i do update D_i; unpause r_i;
13: until (all reductions paused) AND (vertexDegreeLimit > 1.0);
14: return G, C_hat; (Implícito no contexto)
```

1.4 WC-MWC (Weight Cover MWC)

Descrição Completa:
O WC-MWC é outro algoritmo exato baseado em BnB. Diferencia-se pela forma como implementa a sua estratégia de ramificação e poda (PartitionWC-MWC). Baseia-se no conceito de weight cover (cobertura de peso). O objetivo é dividir os vértices V do grafo em dois conjuntos disjuntos, A (não ramificados) e B (ramificados), garantindo que a melhor clique no conjunto A não é melhor do que a melhor clique global atual $\hat{C}$.

A função *PartitionWC-MWC* itera sobre os vértices. Para cada vértice v_i, tenta cobrir o seu peso total $w(v_i)$ usando o conjunto de ISs Π. Se o peso de v_i puder ser coberto por Π sem que o limite superior UB_{WC} exceda o limite t, então v_i é adicionado ao conjunto não ramificado A. Caso contrário, v_i é adicionado ao conjunto de vértices de ramificação B.

Pseudocódigo (Função de Particionamento, simplificada):

Algoritmo 7: PartitionWC-MWC(G, t, O)
Input: Grafo G, Limite t, Ordem O
Output: Conjuntos A (Não Ramificados), B (Ramificados)

```
1:  A := Vazio, B := Vazio, Pi := Vazio (Inicializa Cobertura de Peso)
2:  for each v_i in V in order O do
3:      Tenta integrar o peso de v_i no Weight Cover Pi
4:      if (A integração é possível sem exceder t) then
5:          Atualiza Pi (cobrindo w(v_i))
6:          A := A U {v_i}
7:      else (v_i não pode ser coberto)
8:          B := B U {v_i}
9:      end if
10: end for
11: return (A, B)
```

Secção 2: Algoritmos Focados em Redução (MWCRedu e MWCPeel)

Estes algoritmos utilizam intensamente regras de redução de grafo para simplificar a instância do MWC antes ou durante o processo de busca.

2.1 MWCRedu

(Ver Secção 1.3 - Algoritmo Exato). O MWCRedu é primariamente um algoritmo exato focado em redução.

2.2 MWCPeel

O MWCPeel é um algoritmo heurístico que compartilha o framework de redução com o MWCRedu.

Descrição Completa:
O MWCPeel é uma abordagem híbrida que combina reduções exatas com uma técnica de redução heurística chamada peeling (descascamento).

1. **Inicialização:** Determina a clique inicial $\hat{C}$ usando *InitializeWLMC*.
2. **Redução Exata:** Aplica as regras de redução exatas através de *ReduceMWCRedu*. Na primeira iteração, é usado um limite no grau dos vértices (lim = true).
3. **Peeling Heurístico:** Segue-se uma redução heurística (*peeling*), onde um certo número (*num*) de vértices com o menor score $w(N[v])$ (peso da vizinhança inclusiva) é removido. O número de vértices a remover é geralmente 10% de $|V|$ para grafos muito grandes ($|V| > 50.000$) ou calculado com base numa fórmula que garante uma remoção mínima para grafos menores.
4. **Resolução Final:** O processo de redução e *peeling* repete-se até que os critérios de paragem sejam satisfeitos. O grafo reduzido é então passado para o solucionador exato TSM-MWC para calcular o resultado final.

Pseudocódigo (MWCPeel):

Algoritmo 23: MWCPeel

```
1:  C_hat := InitializeWLMC(G, 0);
2:  isFirstIteration := true;
3:  repeat
4:      G, C_hat := ReduceMWCRedu(G, C_hat, isFirstIteration);
5:      isFirstIteration := false;
6:      num := (0.1 * |V| se |V| > 50000, caso contrário fórmula específica)
7:      remove num vertices with the lowest score w(N[v]);
8:  until stopping criteria met;
9:  return TSM-MWC(G, C_hat);
```

Secção 3: Algoritmos Baseados em Estocasticidade, Heurísticas de Busca Local (Monte Carlo) e Aproximação

Os algoritmos heurísticos e estocásticos são frequentemente classificados como algoritmos Monte Carlo, pois são rápidos (tempo de execução garantido), mas a qualidade da solução (otimalidade) não é garantida.

3.1 FastWClq

O FastWClq é um solucionador heurístico que intercala a construção de cliques com a redução de grafo. Pode provar a otimalidade das suas soluções em alguns casos, sendo por vezes descrito como semi-exato.

Descrição Completa:
O FastWClq utiliza:

- **Busca Local:** Baseada em heurísticas, a construção da clique é guiada pela estratégia *Best from Multiple Selection* (BMS), onde k candidatos diferentes são amostrados aleatoriamente, e o melhor vértice é escolhido com base numa função de estimativa de benefício.
- **Redução de Grafo (ReduceGraphFastWClq):** Após encontrar uma nova melhor clique $\hat{C}$, o grafo é reduzido. O algoritmo remove vértices v para os quais o limite superior $UB_0(v)$ ou $UB_1(v)$ for menor ou igual ao peso da clique atual $w(\hat{C})$. Se o grafo se tornar vazio após a redução, o $\hat{C}$ encontrado é provado ser o ótimo, e o algoritmo termina.

Pseudocódigo (Procedimento Principal):

Algoritmo 13: FastWClq

```
C_hat := Vazio;
G' := G;
repeat
   C_hat_old := C_hat;
   C := GenerateCliqueFastWClq(G');
   if w(C) > w(C_hat) then
      C_hat := C;
      G' := ReduceGraphFastWClq(G', C_hat);
   else if w(C) = w(C_hat) AND some tie-breaking condition then
      G' := ReduceGraphFastWClq(G', C_hat);
   else if w(C) <= w(C_hat) AND no improvement then
      (Diversificação ou reinício, implícito)
until termination criteria met or G' = Vazio;
return C_hat;
```

3.2 SCCWalk (Strong Configuration Checking Walk)

O SCCWalk é um algoritmo heurístico de busca local introduzido em 2020.

Descrição Completa:
O SCCWalk modifica iterativamente várias cliques iniciais para procurar soluções melhores. As estratégias inovadoras são:

- **Strong Configuration Checking (SCC):** Uma regra de proibição mais rigorosa do que a *Configuration Checking* (CC), usada para minimizar a ciclagem na busca local. O SCC impõe restrições mais apertadas sobre os operadores Adicionar (*Add*), Remover (*Drop*) e Trocar (*Swap*), limitando a re-adição de vértices.
- **Walk Perturbation:** Estratégia utilizada quando o algoritmo fica preso numa área limitada do espaço de busca (*unimproveStep* atinge *maxUStep*).

Pseudocódigo (SCCWalk - Outline):

Algoritmo 17: SCCWalk

```
1:  C_hat <- Vazio; C <- InitGreedySCCWalk(G);
2:  repeat
3:      Initialize Penalties and SCC rules;
4:      repeat (Inner search loop)
5:          Select an operation (Add, Drop, or Swap);
6:          Apply operation to C;
7:          Update SCC rules and configuration values;
8:          if C improved C_hat then C_hat <- C; reset unimproveStep;
9:          if unimproveStep > maxUStep then
10:             C <- WalkPerturbationSCCWalk(C);
11:             Reset unimproveStep and SCC values;
12:         end if
13:     until termination criteria for local search met;
14: until overall termination criteria met;
15: return C_hat;
```

3.3 SCCWalk4L (SCCWalk for Large Graphs)

O SCCWalk4L é uma variação do SCCWalk.

Descrição Completa:
Embora muito semelhante ao SCCWalk original, o SCCWalk4L incorpora duas diferenças principais:

- **Seleção de Swap Refinada:** A seleção do par de troca (*swap pair*) é feita por uma heurística especial implementada em *GetSwapPairSCCWalk4L()*.
- **Redução de Grafo Dinâmica:** Quando uma nova melhor clique $\hat{C}$ é encontrada, o grafo é reduzido através de *ReduceGraphSCCWalk4L*. Esta função elimina vértices que nunca poderiam fazer parte de uma clique melhor do que a $\hat{C}$ atual, usando limites UB_0 e UB_1.

Pseudocódigo (Redução de Grafo no SCCWalk4L):

Algoritmo 22: ReduceGraphSCCWalk4L(G, C_hat, C)

```
1:  RemovalQueue := Vazio;
2:  foreach v in V do
3:      if (V < Criterio) then ... (Simplificado)
4:          RemovalQueue := RemovalQueue U {v};
5:  end for
```

```
6:  while RemovalQueue != Vazio do
7:      u := pop a vertex from RemovalQueue;
8:      remove u and its incident edges from G;
9:      foreach v in N(u) do
10:         if (UB0(v) <= w(C_hat)) OR (UB1(v) <= w(C_hat)) then
11:             RemovalQueue := RemovalQueue U {v};
12:         end if
13:     end for
14: end while
15: return G, C;
```

3.4 UEW (Unexplored Weight)

O UEW é um algoritmo heurístico guloso (*one-shot greedy*) que se espera que seja muito rápido, mas sem garantia de otimalidade.

Descrição Completa:
O UEW constrói apenas uma clique. Baseia-se na heurística de selecionar em cada passo o vértice candidato que maximiza o peso inexplorado ($w_{UE}(v)$). O algoritmo é rápido, sendo a melhor escolha em testes para grafos pequenos a médios onde a velocidade é a prioridade máxima.
Pseudocódigo (UEW):

Algoritmo 24: UEW

```
1:  Candidates := V;
2:  C_hat := Vazio;
3:  while Candidates != Vazio do
4:      c := argmax {v in Candidates} w_UE(v);
5:      C_hat := C_hat U {c};
6:      Candidates := Candidates \ {c};
7:      Candidates := Candidates INTERSECTION N(c);
8:  end while
9:  return C_hat;
```

3.5 UEW-R (Unexplored Weight - Reduction)

O UEW-R é uma extensão do UEW que incorpora conceitos de redução de grafo do FastWClq.

Descrição Completa:
Este algoritmo cria cliques iterativamente usando a estratégia gulosa do UEW e aplica reduções de grafo entre as iterações.

- **Iteração:** Em cada iteração, $C_{current}$ é construída usando a heurística UEW.
- **Redução Condicional:** Se $w(C_{current}) > w(\hat{C})$, $\hat{C}$ é atualizada, e o grafo G é reduzido usando a função *ReduceGraphFastWClq*. Esta redução remove vértices que não podem estar na melhor clique atual.
- **Critério de Paragem:** Se não houver melhoria, o algoritmo termina (para evitar ciclos infinitos). Se o grafo se tornar vazio após a redução, o UEW-R pode garantir a otimalidade da solução.

Pseudocódigo (UEW-R):

Algoritmo 25: UEW-R

```
1:  C_hat := Vazio;
2:  repeat
3:      // 1.
        C_current := Vazio; Candidates := V;
4:      while Candidates != Vazio do
5:          c := argmax {v in Candidates} w_UE(v);
6:          C_current := C_current U {c};
7:          Candidates := Candidates \ {c};
8:          Candidates := Candidates INTERSECTION N(c);
9:      end while
10:     // 2.
        if w(C_current) > w(C_hat) then
11:         C_hat := C_current;
12:         G := ReduceGraphFastWClq(G, C_hat);
13:     else
14:         return C_hat;
15:     end if
16: until G = Vazio;
17: return C_hat;
```

3.6 Algoritmos Estocásticos via Relaxação (Bregman-Sinkhorn)

Descrição Completa:
Algoritmos de aproximação que utilizam Programação Semidefinida (SDP) e Programação Linear (LP) são cruciais para fornecer garantias teóricas e soluções escaláveis.
O **Algoritmo Bregman-Sinkhorn** (aplicado ao problema dual MWIS) é um algoritmo aproximado escalável para o *Maximum-Weight Independent Set* (MWIS), que é equivalente ao MWC. O seu componente central é um *dual coordinate descent* aplicado a uma relaxação LP suavizada do problema, também conhecida como método de Bregman ou algoritmo de Sinkhorn.

Esta abordagem é notável por:

- **Otimização Dual:** Aplica a descida de coordenadas dual para resolver a relaxação LP suavizada, controlando o nível de suavização com base numa estimativa da *relaxed duality gap*.
- **Heurística Primal:** Após a otimização dual, uma heurística primal simples e muito eficiente é usada para obter soluções inteiras viáveis. Esta heurística combina geração gulosa aleatória e recombinação ideal.

(Nota: O pseudocódigo explícito para o algoritmo Bregman-Sinkhorn não foi detalhado nos excertos originais).