**Tóm tắt**

Báo cáo này trình bày tổng quan các ứng dụng và thuật giải tiêu biểu cho bài toán Vertex Cover, từ những phương pháp giải truyền thống cho đến các thuật giải hiện đại. Đồng thời, tiến hành so sánh thực nghiệm giữa các thuật giả trên nhiều tập dữ liệu đồ thị khác nhau nhằm đánh giá hiệu quả về mặt thời gian, chi phí tính toán, chất lượng lời giải. Mục tiêu là cung cấp cái nhìn hệ thống về các thuật giải và và giúp định hướng lựa chọn phương pháp phù hợp cho từng bối cảnh ứng dụng cụ thể. Trong báo cáo này chúng em sử dụng danh sách kề để biểu diễn đồ thị.

Trong khuôn khổ bài báo cáo, chúng em phân loại các giải thuật ra làm ba nhóm chính:

1. Thuật giải chính xác.
2. Thuật giải xấp xỉ.
3. Các thuật giải heuristic và metaheuristic.

Các phần tiếp theo sẽ giới thiệu bài toán Vertex Cover và các ứng dụng của nó, sau đó mô tả chi tiết các thuật giải được khảo sát, trước khi đi vào phần thiết kế thực nghiệm và phân tích kết quả.

1. **Giới thiệu**

Vertex Cover là một bài toán cơ bản trong lý thuyết đồ thị, yêu cầu tìm tập con nhỏ nhất các đỉnh sao cho mỗi cạnh trong đồ thị đều kề với ít nhất một đỉnh trong tập này. Đây là một bài toán NP-Hard, có quan hệ bổ sung với bài toán tập độc lập tối đa, và có thể được mô hình hóa dưới dạng bài toán nguyên tuyến tính (ILP). Dù không thể giải được trong thời gian đa thức cho mọi trường hợp tổng quát, bài toán có thể giải hiệu quả trên đồ thị lưỡng phân nhờ định lý Kőnig, và có thuật toán giải nhanh khi kích thước tập cover là tham số cố định (FPT theo k).

**1.1 Định nghĩa bài toán Vertex Cover**

Cho một đồ thị vô hướng G=(V,E), trong đó V  là tập đỉnh và E là tập cạnh, một tập con các đỉnh C⊆V  được gọi là một vertex cover nếu mỗi cạnh (u,v)∈E  đều có ít nhất một đầu mút nằm trong C. Mục tiêu của bài toán *Vertex Cover* là tìm một tập đỉnh C nhỏ nhất thỏa mãn điều kiện trên. Kích thước nhỏ nhất của một vertex cover được gọi là *kích thước vertex cover tối ưu*.

**1.2 Mô hình toán học.**

Vertex Cover tiêu chuẩn (Unweighted Vertex Cover):

Cho đồ thị vô hướng, không trọng số .

Biến quyết định:

* + Với mỗi đỉnh , gán biến nhị phân
    - nếu đỉnh v nằm trong Vertex Cover
    - nếu không

Mô hình Integer Linear Programming (ILP):

Ràng buộc:

**1.3 Các ứng dụng của Vertex Cover.**

Vertex Cover trong mạng giao thông. Giả sử đồ thị vô hướng G biểu diễn một mạng giao thông với mỗi đỉnh là một nút giao và mỗi cạnh là một cạnh là một con đường nối các nút giao. Nếu lắp đặt các camera giám sát tại các đỉnh thuộc tập Vertex Cover tối thiểu, thì mọi đoạn đường đều sẽ được theo dõi bởi ít nhất một camera, giúp **giảm chi phí lắp đặt** camera đến mức thấp nhất.

Trong mạng máy tính, giả sử đồ thị G biểu diễn cho một mạng lưới máy chủ với mỗi đỉnh là một máy chủ và có các cạnh chính là kết nối giữa hai máy chủ với nhau. Theo nghiên cứu trong [1], sự lan truyền mã độc phụ thuộc vào tô-pô (topology) của mạng. Nếu ta có thể xử lý các máy chủ trong vập Vertex Cover tối thiểu, thì có thể **ngăn chặn sự lan truyền của mã độc** [2].

Trong quy trình kinh doanh, giải sử đồ thị G có mỗi đỉnh chính là một trạng thái trong quy trình làm việc và mỗi cạnh chính là quá trình chuyển từ trạng thái này sang trạng thái khác. Một phần công việc được thực hiện khi có sự chuyển trạng thái và khi đó có thể phát sinh lỗi. Do đó, phải kiểm tra lỗi tại các trạng thái; nếu các lần kiểm tra chỉ được thực hiện  tại các đỉnh trong Vertex Cover tối thiểu thì giảm được số lần kiểm tra, với điều kiện bất kỳ cặp trạng thái nào diễn ra cần kiểm tra ít nhất một lần.

Ví dụ, giả sử ta có một quy trình kinh doanh như sau:

A: Nhận đơn hàng

B: Kiểm tra kho

C: Xử lý thanh toán

D: Đóng gói hàng

E: Giao hàng

Quy trình làm việc có thể chuyển qua lại như sau: A -> B -> C -> D -> E. Nhưng cũng có thể đi vòng như: C -> B, nếu kho không đủ; D -> C, nếu đóng gói sai. Do đó cần kiểm tra lỗi tại một số trạng thái; nếu các Nếu các lần kiểm tra chỉ được thực hiện tại các đỉnh trong Vertex Cover tối thiểu, thì giảm số lần cần phải kiểm tra.

Trong lý thuyết trò chơi, giả sử mỗi đỉnh trong đồ thị biểu diễn một chiến lược, và mỗi cạnh đại diện cho một xung đột lợi ích giữa hai chiến lược không thể cùng tồn tại hoặc gây ảnh hưởng lẫn nhau. Khi đó, việc tìm một tập phủ đỉnh (vertex cover) tối thiểu tương ứng với việc lựa chọn một tập con nhỏ các chiến lược sao cho mọi xung đột đều được kiểm soát hoặc giám sát.

Trong bài toán lập lịch, trong đồ thị G có các đỉnh là lịch thi các môn học và cạnh nối hai đỉnh bất kỳ biểu trưng cho có ít nhất 1 sinh viên học cả hai môn đó. Cần tìm tập hợp môn thi nhỏ nhất sao cho mỗi cạnh trong đồ thị đều có ít nhất một đầu mút thuộc tập này. Chỉ cần lập lịch thi cho Vertex Cover tối thiểu còn các đỉnh còn lại có thể chia sẻ khung giờ với nhau vì không còn xung đột.

Trong khai phá dữ liệu, cho đồ thị G có các đỉnh là các khách hàng, cạnh nối hai đỉnh là hai khách hàng có hành vi tương tự. Tìm Vertex Cover tối thiểu để chọn ra một tập khách hàng ‘đại diện’ sao cho mọi mối hành vi đều được xếp vào ít nhất một khách hàng đại diện. Làm giảm kích thước mẫu và vẫn giữ được các cấu trúc quan trọng của dữ liệu.

Trong việc vẽ đồ thị, khi vẽ một đồ thị, cần giảm tối đa số đường cắt nhau để bản vẽ dễ đọc. Ví dụ, cho một đồ thị G có V đỉnh và E cạnh, gọi mỗi cạnh của G là một đỉnh của đồ thị G’, nếu hai cạnh trong G cắt nhau khi vẽ, thì trong G’ sẽ có một cạnh nối hai đỉnh tương ứng. Sau khi tìn Vertex Cover tối thiểu trong G’, ta có được danh sách các cạnh trong G cần phải được chỉnh lại để đồ thị G có ít nút giao nhất.

1. **Các thuật giải bài toán Vertex Cover**

Vertex Cover là bài toán NP-Complete, trong báo cáo này chỉ xét đến Vertex Cover trên đồ thị vô hướng, không trọng số. Do đó, không có thuật giải đa thức nào có thể giải chính xác cho mọi trường hợp. Tuy nhiên, vẫn có nhiều phương pháp khác nhau đã được đề xuất - từ thuật giải chính xác cho đến xấp xỉ và heuristic - để lý hiệu quả trường hợp cụ thể hoặc đồ thị lớn.

**2.1 Phương pháp vét cạn**

Phương pháp vét cạn kiểm tra mọi tập con có thể để tìm ra tập đỉnh để tìm vập cover nhỏ nhất. Vét cạn là phương pháp cổ điển đơn giản nhưng kém hiệu quả đối với đồ thị lớn vì nó có độ phức tạp tăng theo cấp số mũ, trong bài toán Vertex Cover ta có thể triển khai thuật toán theo các bước như sau:

1. Duyệt qua tất cả tập con của tập đỉnh V, có tổng cộng 2^n với n là số đỉnh.
2. Với mỗi tập con S thuộc V, kiểm tra xem nó có phải là một Vertex Cover không.
3. Lưu lại tập cover nhỏ nhất.

Mã giả:

|  |
| --- |
| Input:    n: số lượng đỉnh trong đồ thị (V = {0, 1, ..., n - 1})    edges: danh sách các cạnh (mỗi cạnh là một cặp (u, v))  Output:   best\_cover: tập đỉnh nhỏ nhất che phủ tất cả các cạnh  1: Function isCover(set, edges):  2: For each edge (u, v) in edges:  3: If neither u nor v is in set:  4: Return false  5: Return true  6: Function BruteForceVertexCover(n, edges):  7: best\_cover ← ∅  8: For mask from 0 to (2^n - 1):  9: current\_set ← ∅  10: For i from 0 to n - 1:  11: If bit i of mask is 1:  12: Add i to current\_set  13: If isCover(current\_set, edges):  14: If best\_cover is empty or size(current\_set) < size(best\_cover):  15: best\_cover ← current\_set  16: Return best\_cover |

Với mask: là một số nguyên biểu diễn tập con bằng nhị phân. Bit thứ i bằng 1 nghĩa là đỉnh i được chọn.

Độ phức tạp thuật toán:

* Tập đỉnh có phần tử, số tập con là:
* Với mỗi tập con , ta duyệt toàn bộ danh sách kề (gồm cạnh) để kiểm tra xem có cạnh nào mà cả hai đỉnh của nó không nằm trong . Chi phí kiểm tra một tập con: .

Kết hợp hai bước:

Vậy với độ phức tạp thuật toán tăng theo cấp số mũ, đây là một thuật toán chậm, kém hiệu quả đối với các đồ thị lớn. Vì thế, phương pháp vét cạn chỉ thích hợp với các đồ thị nhỏ, chủ yếu được dùng như một cơ sở để đánh giá độ chính xác của các thuật toán khác, hoặc trong trường hợp cần lời giải tối ưu tuyệt đối cho các bài toán quy mô nhỏ.

**2.2 Phương pháp quy hoạch động (DP)**

Phương pháp quy hoạch động là một kĩ thuật thường được áp dụng để tối ưu các bài toán có thể chia ra thành các bài toán nhỏ hơn, giải quyết từng bài toán con một lần duy nhất,  lưu trữ kết quả trung gian để tái sử dụng.

Trong bài toán Vertex Cover, phương pháp quy hoạch động được áp dụng hiệu quả nhất trên cây hoặc đồ thị có cấu trúc gần như cây (tree decomposition), bởi vì trên đồ thị tổng quát, phương pháp này thường không khả thi do số trạng thái lớn và có thể bị vòng lặp.

Giải sử ta có một cây T với gốc là r, phương pháp quy hoạch động hoạt động bằng cách xét mỗi đỉnh theo hai trạng thái:

* : Kích thước nhỏ nhất của vertex cover cho cây con gốc tại , *khi không chọn*  vào cover.
* : Kích thước nhỏ nhất của vertex cover cho cây con gốc tại , *khi chọn*  vào cover.

Công thức truy hồi:

* Nếu không chọn : ta chọn tất cả các con của .
* Nếu chọn : ta có thể chọn các con của hoặc không.

Mã giả:

|  |
| --- |
| Input:    - G: cây có n đỉnh, biểu diễn dưới dạng danh sách kề    - root: đỉnh gốc để bắt đầu DFS  Output:    - Giá trị nhỏ nhất của vertex cover  01 Function SolveVertexCover(T, r):  02 // T: adjacency list of a tree with n nodes, r: root  03 for v in 0..n−1 do  04 visited[v] ← false  05 dp[v][0] ← 0 // dp[v][0]: min cover size if v is NOT chosen  06 dp[v][1] ← 0 // dp[v][1]: min cover size if v IS chosen  07 end for  08  09 DFS(r)  10  11 return min(dp[r][0], dp[r][1])  12  13 Procedure DFS(u):  14 visited[u] ← true  15 dp[u][0] ← 0 // if u not chosen  16 dp[u][1] ← 1 // if u chosen  17  18 for each v in T[u] do  19 if not visited[v] then  20 DFS(v)  21 // if u not chosen, v must be chosen  22 dp[u][0] ← dp[u][0] + dp[v][1]  23 // if u chosen, v may or may not be chosen  24 dp[u][1] ← dp[u][1] + min(dp[v][0], dp[v][1])  25 end if  26 end for |

Độ phức tạp thuật toán:

* Với cây có n đỉnh (n node), hàm DFS(u) được gọi đúng một lần cho mỗi đỉnh, có n lần gọi.
* Mỗi đỉnh u duyệt tất cả các đỉnh kề v, nhưng do cây có n-1 cạnh, tổng số lần duyệt là

Vậy thuật toán có độ phức tạp thời gian là , tuy có độ phức tạp đa thức nhưng DP chỉ hoạt động trên đồ thị dạng cây – nơi không có chu trình. Làm cho DP chỉ làm một trường hợp đặc biệt, không chứng minh được Vertex Cover có thể giải trong thời gian đa thức.

**2.3 Thuật toán FPT, kỹ thuật Kernelization và Nhánh-Ràng buộc**

Bài toán **Vertex Cover** là một trong những bài toán kinh điển trong lý thuyết độ phức tạp và tối ưu tổ hợp. Với một đồ thị vô hướng và một số nguyên , bài toán yêu cầu xác định xem có tồn tại tập đỉnh với , sao cho mọi cạnh đều có ít nhất một đầu mút thuộc .

Bài toán này là NP-Complete, tuy nhiên, nó có thể được giải hiệu quả trong thực tế với các kỹ thuật thuộc lớp thuật toán *Fixed-Parameter Tractable (FPT)* theo tham số . Hai kỹ thuật nổi bật trong hướng tiếp cận này là *kernelization (rút gọn bài toán)* và *branch-and-bound (nhánh và ràng buộc)*.

Một thuật toán được gọi là khi thời gian chạy của nó có dạng:

Trong đó:

* là một hàm phụ thuộc vào tham số , nhưng không phụ thuộc vào .
* là một hàm đa thức theo kích thước đầu vào .

**Kỹ thuật Kernelization:**

Kernelization là bước tiền xử lý nhằm rút gọn thể hiện bài toán về kích thước nhỏ hơn, mà vẫn bảo toàn tương đương logic với bài toán ban đầu. Đối với Vertex Cover trong bài báo cáo này, các quy tắc kernel hóa bao gồm:

* 1. Đỉnh cô lập: nếu không kề cạnh nào, xóa khỏi đồ thị.
  2. Đỉnh bậc một: nếu chỉ kề đúng một đỉnh , thị buộc phải nằm trong cover, xóa giảm
  3. Đỉnh độ cao: nếu , buộc , xóa , giảm .
  4. Đỉnh thống trị: nếu , tức là mọi neighbor của đều là neighbor của , thì có thể loại bỏ (vì đã bao gồm các cạnh nối với ).

Sau cùng, ta thu được đồ thị còn rất nhỏ (thường đỉnh) và đảm bảo:

**Ta chứng minh (\*):**

Sau khi áp dụng kernel ta thu được đồ thị thỏa mãn: (A) mọi đỉnh đều có Để chứng minh , ta sử dụng lập luận về số cacnhj và khả năng phủ cạnh của một vertex cover kích thước .

Bước 1. Giới hạn số cạnh : Một vertex cover có kích thước nhiều nhất có thể phủ được đỉnh, và mỗi đỉnh trong lại kề tối đa cạnh (do A). Do đó, mỗi cover size phải phủ tối đa cạnh hay . Nếu , thì không tồn tại vertex cover size , vì muốn kernel cho bài toán “có cover size hay không”, ta chỉ tập trung vào trường hợp .

Bước 2. Từ đến : với một đồ thị vô hướng, tổng bậc của đồ thị bằng . Mỗi đỉnh góp tối đa vào tổng bậc (theo (A)), nên

Mặt khác:

Kết hợp cả hai bước trên ta có:

Mã giả:

|  |
| --- |
| 1. Function Kernelize\_Param (G, k):  2. coverForced ← ∅  3. changed ← true  4.  5. while changed do  6. changed ← false  7.  8. // 1) Đỉnh cô lập  9. for each v ∈ V(G) with degree(v) = 0 do  10. RemoveVertex(G, v)  11. changed ← true  12. end for  13.  14. // 2) Đỉnh bậc một  15. for each v ∈ V(G) with degree(v) = 1 do  16. let u be the unique neighbor of v  17. coverForced ← coverForced ∪ {u}  18. RemoveVertex(G, v); RemoveVertex(G, u)  19. k ← k - 1  20. changed ← true  21. end for  22.  23. // 3) Đỉnh độ cao (High-degree)  24. for each v ∈ V(G) with degree(v) > k do  25. coverForced ← coverForced ∪ {v}  26. RemoveVertex(G, v)  27. k ← k - 1  28. changed ← true  29. end for  30.  31. // 4) Đỉnh thống trị (Domination)  32. for each pair u, v ∈ V(G) do  33. if N(v) ⊆ N(u) then  34. RemoveVertex(G, u)  35. changed ← true  36. end if  37. end for  38. end while  39.  40. return G, coverForced, k |

**Thuật toán Nhánh-Ràng buộc (Branch and Bound) trên Kernel**

Kỹ thuật **nhánh và ràng buộc** là một phương pháp giải thuật tổng quát thường được áp dụng để giải các bài toán tổ hợp khó, đặc biệt là các bài toán NP-Complete. Phương pháp này dựa trên nguyên lý chia để trị, trong đó không gian lời giải được duyệt theo cấu trúc cây tìm kiếm, với mỗi nút đại diện cho một trạng thái bán phần của lời giải. Tại mỗi bước, thuật toán sẽ phân nhánh không gian tìm kiếm thành các trường hợp con và sử dụng các tiêu chí ràng buộc để loại bỏ các nhánh không thể dẫn tới lời giải tối ưu.

Trong bối cảnh của thuật toán tham số hóa (FPT), nhánh và ràng buộc không chỉ giúp kiểm soát không gian tìm kiếm theo tham số , mà còn kết hợp hiệu quả với các kỹ thuật như *kernelization* để rút gọn kích thước bài toán trước khi tiến hành phân nhánh. Phần này sẽ trình bày nguyên lý hoạt động, cấu trúc thuật toán và cách áp dụng kỹ thuật nhánh và ràng buộc trong giải bài toán Vertex Cover theo hướng tiếp cận FPT.

Mã giả: ta giả sử chạy kernel trước khi thực hiện branch and bound.

|  |
| --- |
| 1. // ===== Branch-and-Bound =====  2. Function BranchAndBound(G\_curr, coverCurr, kCurr):  3. // 1) Nếu không còn cạnh, cập nhật giải tốt nhất  4. if E(G\_curr) = ∅ then  5. if |coverCurr| < bestSize then  6. bestCover ← coverCurr  7. bestSize ← |coverCurr|  8. end if  9. return  10. end if  11. // 2) Cắt tỉa nếu vượt ngân sách hoặc không thể tốt hơn  12. if kCurr ≤ 0 or |coverCurr| ≥ bestSize then  13. return  14. end if  15. // 3) Chọn đỉnh độ lớn nhất để phân nhánh  16. v ← argmax\_{v ∈ V(G\_curr)} degree(v, G\_curr)  17. // 4) Nhánh include v  18. G1 ← G\_curr without v and its incident edges  19. BranchAndBound(G1, coverCurr ∪ {v}, kCurr - 1)  20. // 5) Nhánh exclude v: include toàn bộ N(v)  21. N\_v ← neighbors of v in G\_curr  22. G2 ← G\_curr without N\_v and their incident edges  23. BranchAndBound(G2, coverCurr ∪ N\_v, kCurr - |N\_v|)  24. end Function |

Độ phức tạp thuật toán :

* 1. Kernelization:
* Số vòng while changed .
* Phần quét “degree-0”, “degree-1”, “high-degree” mỗi vòng khoảng
* Phần domination mỗi vòng , suy ra (tất cả dựa trên (\*)):
* Tổng chi phí xóa đỉnh trên toàn hàm: .

Vậy mỗi lần kernel lặp tốn:

* Không gian: lưu đồ thị, tập .

Kernelization chỉ thực hiên một lần trước Branch and Bound.

* 1. Branch and Bound

Gọi là số node của cây tìm kiếm khi còn ‘ngân sách’ . Với chiến lược phân nhánh:

* Include một đỉnh giảm đi 1: gọi
* Exclude buộc toàn bộ vào cover, giảm ít nhất bằng . Trong trường hợp (kịch bản tệ nhất sau kernel), gọi

Do đó nghiệm xấp xỉ của đệ quy:

Mà giải cho ta được:

* Thời gian tổng thể: , hay viết gọn:
* Không gian: lưu cây đệ quy cỡ , cùng đồ thị kernel kích thước mà và nên tổng .

Với tham số không quá lớn, thuật toán này rất hiệu quả, vì kernel giúp giảm đáng kể kích thước đồ thị và branch and bound chỉ chạy bước.

**2.3 Phương pháp xấp xỉ**

Vì Vertex Cover là bài toán NP-Complete, việc tìm lời giải chính xác cho đồ thị lớn là không khả thi trong thời gian đa thức. Do đó, các thuật giải xấp xỉ được sử dụng để tìm lời giải gần tối ưu, nhưng chạy nhanh hơn nhiều. Một thuật giải xấp xỉ có thể đảm bảo lời giải nằm trong một khoảng gần tối ưu — gọi là tỷ số xấp xỉ (approximation ratio).

Tỷ số xấp xỉ của một thuật toán là:

* Nếu , nghĩa là lời giả xấp xỉ không bao giờ lớn hơn 2 lần lời giả tối ưu.
* Với Vertex Cover, thuật giải xấp xỉ hệ số có tỷ số 2, nghĩa là luôn có lời giải không quá gấp đôi lời giải tối ưu, bất kể đầu vào.

**2.3.1 Thuật giải xấp xỉ với hệ số 2**

**2.3.1.1 Maximal Matching Algorithm**

Thuật giải này dựa trên matching cực đại – một tập hợp các cạnh mà không có cạnh nào chung đỉnh, và không thể thêm bất kì cạnh nào mà giữ nguyên tính chất đó.

Mỗi khi chọn một cạnh trong matching, ta đưa ra hai đỉnh của vào tập Vertex Cover, như vậy đảm bảo mỗi cạnh đều được phủ bởi ít nhất một đỉnh trong cover.

Tỷ lệ xấp xỉ: ta có là tập cạnh được chọn trong quá trình maching, là tập cover trả về, nên (vì ta thêm cả hai đầu mút mỗi cạnh vào) và là Vertex Cover tối ưu. Các cạnh trong là matching--tức là không có hai cạnh nào trong chia sẻ một đỉnh. Do đó để phủ toàn bộ các cạnh trong , một vertex cover cần chứa ít nhất một đỉnh từ mỗi cạnh, mà các cạnh không đụng nhau nên không thể dung chung đỉnh để phủ nhiều cạnh Vertex Cover tối ưu cần ít nhất đỉnh để phủ . Vậy

Tỷ lệ xấp xỉ .

Mã giả:

|  |
| --- |
| 1: C ← ∅  2: // Khởi tạo: đánh dấu tất cả các cạnh là “chưa xét”  3: for each e ∈ E do  4: mark e as unvisited  5: end for  6:  7: // Vòng lặp chính  8: while ∃ e = (u,v) ∈ E unvisited do  9: if u ∉ C ∧ v ∉ C then  10: C ← C ∪ {u, v}  11: end if  12: // Đánh dấu tất cả các cạnh kề u hoặc v là “đã xét”  13: for each e′ ∈ E with u ∈ e′ or v ∈ e′ do  14: mark e′ as visited  15: end for  16: end while  17:  18: return C |

Độ phức tạp thuật toán , với là số cạnh vì:

* Duyệt qua m cạnh:
* Đánh dấu các cạnh kề hoặc :

**2.3.1.2 Linear Programming Rounding (LP)**

Thuật toán này sử dụng kĩ thuật làm tròn các nghiệm liên tục của bài toán LP thành nghiệm nguyên, tìm ra nghiệm xấp xỉ gần đúng của bài toán tối ưu hóa nguyên.  Mỗi đỉnh v trong đồ thị có thể có một giá trị x\_v biểu diễn "xác suất" đỉnh này được chọn vào vertex cover.

Mô hình liên tục cho bài toán Vertex Cover:

Với các ràng buộc:

Tỷ số xấp xỉ:

Mã giả:

|  |
| --- |
| Input:  G = (V, E) // Đồ thị vô hướng, V là tập đỉnh, E là tập cạnh  Output:  C ⊆ V // Tập vertex cover tìm được  1. // Bước 1: Giải bài toán LP thư giãn  2. Solve the following linear program:  minimize ∑\_{v ∈ V} x\_v  subject to x\_u + x\_v ≥ 1 ∀ (u, v) ∈ E  0 ≤ x\_v ≤ 1 ∀ v ∈ V  Let x\* = (x\*\_v : v ∈ V) be an optimal solution.  3. // Bước 2: Randomized Rounding  4. C ← ∅  5. for each v ∈ V do  6. r ← DrawUniform(0, 1)  7. if r ≤ x\*\_v then  8. C ← C ∪ {v}  9. end if  10. end for  11. // Bước 3: Trả về kết quả  12. return C |

Độ phức tạp thuật toán:

* Giải bài toán LP bằng phương pháp Interior Point: O(n^3)
* Sau khi giải LP ta cần làm tròn n đỉnh: O(n)

Vậy độ phức tạp tổng thể của LP Rounding là O(n^3), nếu sử dụng phương pháp giải LP với Interior Point Methods.

**2.3.1.3 Primal-Dual**

Thuật toán Primal-Dual giải quyết bài toán tối ưu bằng cách xử lý đồng thời bài toán Primal và Dual qua các bước lặp.

Primal:

Dual:

Ý tưởng như sau:

* Bắt đầu với với mọi đỉnh và cạnh.
* Duyệt qua từng cạnh . Nếu cả (chưa cover), thì: tăng lên 1, đặt đưa vào cover.

Thuật toán khởi tạo một lời giải khả thi cho cả Primal và Dual (tức 0, ), sau đó tăng dần các biến dual cho đến khi mọi ràng buộc primal được thoả mãn, đồng thời xây dựng lời giải Primal (tập cover) từ đó.

**Chứng minh tỷ lệ xấp xỉ:**

Bước 1. lời giải Dual khả thi: với mỗi mà ta tăng , đồng thời đặt . Do đó mỗi đỉnh có tổng vì ta đặt ngay lần đầu gặp cạnh chứa Các ràng buộc Dual được thỏa mãn Nghiệm là khả thi.

Bước 2. Lời giải Primal khả thi: với mỗi cạnh , nếu ta đặt cả hia lên 1 mọi cạnh được cover Nghiệm là khả thi.

Bước 3. So sánh giá trị nghiệm Primal-Dual: gọi là tập đỉnh đầu ra (primal solution), là nghiệm Dual. Ta có, mỗi lần tăng một , ta thêm cả hai đầu mút vào Mỗi đơn vị trong tạo ra tối đa 2 đỉnh trong .

Nhưng do weak duality, ta có:

Với là kích thước nhỏ nhất của tập đỉnh cover:

Nên:

Tỷ lệ xấp xỉ .

Mã giả:

|  |
| --- |
| 1: Function PrimalDualVertexCover(G = (V, E)):  2: // 1. Khởi tạo  3: For each v in V:  4: x[v] ← 0 // biến primal: 0 = chưa chọn v  5: For each e in E:  6: y[e] ← 0 // biến dual  7: C ← ∅ // tập vertex cover  8: // 2. Lặp cho đến khi mọi cạnh được cover  9: For each edge e = (u, v) in E do  10: If x[u] == 0 and x[v] == 0 then  11: // Cạnh (u,v) chưa được cover  12: y[e] ← y[e] + 1  13: // Kéo biến primal sao cho ràng buộc của e: x[u] + x[v] ≥ 1  14: x[u] ← 1  15: x[v] ← 1  16: C ← C ∪ {u, v}  17: End If  18: End For  19: // 3. Trả về kết quả  20: Return C |

Độ phức tạp thuật toán:

* Khởi tạo: gán mảng x có n phần tử, gán mảng y có m phần tử. Tổng là O(n+m)
* Lặp qua các cạnh và gán các giá trị: O(m)

Vậy tổng hợp độ phức tạp là O(n+m).

**2.3.1.4 Massively Parallel Algorithm (MPA)**

**2.5 Phương pháp heuristic**

Phương pháp heuristic không đảm bảo tìm ra nghiệm tối ưu tuyệt đối, nhưng thường cho kết quả tốt trong thời gian tính toán ngắn. Ý tưởng chính của heuristic là sử dụng các tiêu chí gần đúng để hướng dẫn quá trình tìm kiếm lời giải, giúp giảm đáng kể không gian tìm kiếm. Với bài toán Vertex Cover, các chiến lược heuristic phổ biến bao gồm: chọn đỉnh có bậc cao nhất, chọn đỉnh bao phủ nhiều cạnh nhất, hay kết hợp các chiến lược điểm số (scoring) để ưu tiên chọn đỉnh vào cover.

Các heuristic nâng cao, lấy ý tưởng từ tự nhiên hoặc quá trình tiến hóa, đã được phát triển nhằm cải thiện chất lượng nghiệm, bao gồm:

**2.5.1 Tối ưu đàn kiến**

Phương pháp Tối ưu đàn kiến (Ant Colony Optimization - ACO) là một thuật giải heuristic lấy cảm hứng từ hành vi tìm thức ăn của bầy kiến.

Trong tự nhiên, kiến tìm đường đi đến nguồn thức ăn bằng cách để lại dấu vết pheromone trên đường đi. Các kiến khác có xu hướng đi theo những đường đi có nồng độ pheromone cao, dẫn đến sự hình thành của đường đi tối ưu. Tương tự, trong ACO, các ‘kiến ảo’ xây dựng lời giải cho bài toán bằng cách di chuyển trên đồ thị và để lại dấu pheromone, giúp các ‘kiến’ khác tìm lời giải tối ưu cho các vòng lặp tiếp theo.

Quy trình chung của thuật toán ACO như sau:

* 1. Khởi tạo
     + Cho số kiến , số vòng lặp tối đa , hệ số pheromone ban đầu , tốc độ bay hơi , trọng số bồi đắp pheromone Q, và trọng số .
     + Khởi tạo mảng pheromone cho mối đối tượng quyết định (là đỉnh trong bài toán Vertex Cover).
     + Tính heuristic theo độ lớn bậc của đỉnh.
  2. Lặp cho đến khi đạt điều kiện dừng:
     1. Mỗi kiến xây dựng lời giải
     + Bắt đầu với lời giải rỗng.
     + Cho đến khi lời giải hợp lệ:
  + Tính xác suất chọn mỗi đỉnh chưa được chọn:
  + Ngấu nhiên chọn một đỉnh theo phân phối , them vào lời giải, loại bỏ các cạnh kề đỉnh vừa chọn.
    - Lưu lại lời giải và độ ‘tốt’ (kích thước cover).
    1. Cập nhật pheromone
    - Bay hơi: với mỗi đỉnh ,
    - Bồi đắp: cho mỗi kiến, với lời giải cover có kích thước , tăng pheromone trên mỗi đỉnh thêm
    - (Tùy chọn) Bồi đắp pheromone dựa trên lời giải tốt (bestCover)
    1. Local search (tùy chọn)
  1. Trả về lời giải tốt nhất tìm được trong suốt các vòng.

Mã giả:

|  |
| --- |
| 1. // Khởi tạo  2. For each v in V:  3. τ[v] ← τ0 // khởi tạo pheromone ban đầu (tau0 > 0)  4. η[v] ← degree(v) // hoặc số cạnh chưa được cover kề với v  5.  6. bestCover ← V // giả sử tệ nhất ban đầu là chọn tất cả đỉnh  7.  8. // Vòng lặp ACO  9. For iter from 1 to maxIter:  10. solutions ← [] // lưu các cover do kiến xây dựng  11. costs ← [] // lưu kích thước tương ứng  12.  13. // Mỗi con kiến xây dựng giải pháp  14. For k from 1 to m:  15. uncoveredEdges ← E  16. cover ← ∅  17.  18. While uncoveredEdges ≠ ∅:  19. // Tính xác suất chọn mỗi đỉnh chưa được chọn  20. For each v in V \ cover:  21. f[v] ← (τ[v]^α) \* (η[v]^β)  22. sumF ← Σ f[v] over all v ∉ cover  23.  24. // Chọn ngẫu nhiên v theo phân phối f[v]/sumF  25. select v\* with probability f[v\*] / sumF  26.  27. // Thêm v\* vào cover và loại bỏ các cạnh kề nó  28. cover ← cover ∪ {v\*}  29. Remove from uncoveredEdges all edges incident to v\*  30.  31. // Cập nhật heuristic:  32. η[v] = số cạnh kề v trong uncoveredEdges  33. End While  34.  35. solutions[k] ← cover  36. costs[k] ← |cover|  37.  38. If |cover| < |bestCover|:  39. bestCover ← cover  40. End If  41. End For  42.  43. // Bay hơi pheromone  44. For each v in V:  45. τ[v] ← (1 - ρ) \* τ[v]  46.  47. // Cập nhật pheromone theo các kiến  48. For k from 1 to m:  49. For each v in solutions[k]:  50. τ[v] ← τ[v] + Q / costs[k]  51. End For  52. End For  53.  54. // (Tùy chọn) Cập nhật pheromone trên bestCover  55. For each v in bestCover:  56. τ[v] ← τ[v] + Q / |bestCover|  57. End For  58. End For  59.  60. Return bestCover |

Độ phức tạp của thuật toán ACO metaheuristic cho bài toán Vertex Cover nhỏ nhất có thể mô tả như sau:

Chi phí xây dựng lời giải cho mỗi kiến:

* Khởi tạo tập cạnh chưa được cover:
* Mỗi kiến có thể chọn đến đỉnh, và với mỗi đỉnh được chọn:
  + - Tính trọng số pheromone–heuristic cho tất cả đỉnh còn lại: .
    - Lựa chọn ngẫu nhiên một đỉnh theo phân phối này: .
    - Loại bỏ cạnh kề với đỉnh vừa chọn: tích cho toàn bộ quá trình.  
       ⇒ Tổng: cho mỗi kiến.

Chi phí trên toàn bộ kiến và vòng lặp: với kiến và vòng lặp thì:

Chi phí cập nhật pheromone mỗi vòng: bay hơi pheromone cho mọi đỉnh cập nhật pheromone từ mỗi lời giải (mỗi lời giả có độ lớn ≤ n), có tổng cập nhật là *.* Tổng mỗi vòng: *,* và với vòng lặp thì .

Vậy với độ phức tạp thời gian là:

Việc ứng dụng thuật toán ACO người ta thường chọn I và ở mức vừa phải (vài chục đến vài trăm) để cân bằng giữa việc thời gian thực thi và chất lượng đáp án.

**2.5.2 Thuật toán di truyền**

Thuật toán di truyền (GA) là một phương pháp tìm kiếm và tối ưu hóa dựa trên quá trình chọn lọc tự nhiên, nơi các cá thể tốt hơn có cơ hội sinh sản và truyền lại đặc điểm của mình cho thế hệ sau.

Trong GA, một tập hợp các lời giải ứng viên (quần thể) được tiến hóa qua nhiều thế hệ thông qua các thao tác di truyền như chọn lọc (selection), lai ghép (crossover), và đột biến (mutation). Mục tiêu của thuật toán là tìm kiếm lời giải tối ưu hoặc gần tối ưu cho một bài toán nhất định bằng cách khai thác và khám phá không gian lời giải.

Quy trình họat động của thuật toán di truyền:

1. Khởi tạo

* Đặt:
* Kích thước quần thể
* Số thể hệ tối đa
* Xác suất lai ghép
* Xác suất đột biến
* Kích thước
* Khởi tạo quần thể gồm ca thể ngẫu nhiên, mỗi cá thể là một chuỗi nhị phân (0/1), biểu diễn tập đỉnh đã chọn.
* Tính toán độ thích nghi cho từng cá thể
* Ghi nhân cá thể tốt nhất ban đầu ()

1. Vong lặp chính

* Tạo thế hệ mới
* Khởi tạo quần thể mới
* Lăp cho tới khi đủ cá thể:
* Chọn lựa:
* Chọn ngẫu nhiên cá thể từ
* Chọn cá thể có độ thích nghi tốt nhất trong làm
* Lặp lại để chọn
* Lai ghép:
* Với xác xuất , thực hiện lai ghép giữa và để tạo và .
* Nếu không lai ghép, sao chép nguyên và .
* Đột biến, với mỗi đứa con, tại mỗi gene, xác xuất , lật bit (0 thành 1, 1 thành 0).
* Sửa lỗi: nếu cá thể không là Vertex Cover thì thực hiện thêm đỉnh cần thiết.
* Thêm và vào .
* Đánh giá: tính độ thích nghi cho tất cả cá thể trong , dựa trên số lượng đỉnh trong Vertex Cover.
* Cập nhật lời giải tốt nhất.
* Thay thế quần thể: .

1. Trả về .

Mã giả:

|  |
| --- |
| 1. // 1. Khởi tạo quần thể ban đầu  2. P ← GenerateInitialPopulation(P\_size)  3. for each individual ∈ P do  4. fitness[individual] ← EvaluateFitness(individual)  5. end for  6. BestSolution ← argmax\_{individual ∈ P} fitness[individual]  7. // 2. Vòng lặp tiến hóa  8. for gen = 1 to G\_max do  9. P\_new ← ∅  10. while |P\_new| < P\_size do  11. // 2.1 Tournament Selection  12. Parent1 ← TournamentSelect(P, k)  13. Parent2 ← TournamentSelect(P, k)  14. // 2.2 Lai ghép  15. r ← DrawUniform(0, 1)  16. if r ≤ p\_crossover then  17. (Child1, Child2) ← Crossover(Parent1, Parent2)  18. else  19. Child1 ← Copy(Parent1)  20. Child2 ← Copy(Parent2)  21. end if  22. // 2.3 Đột biến  23. Mutate(Child1, p\_mutation)  24. Mutate(Child2, p\_mutation)  25. // 2.4 Sửa lỗi để đảm bảo tính hợp lệ  26. Repair(Child1, G)  27. Repair(Child2, G)  28. // 2.5 Local Search: Greedy Remove  29. GreedyRemoveLS(Child1, G)  30. GreedyRemoveLS(Child2, G)  31. // 2.6 Local Search: Memetic Bit-Flip  32. MemeticBitFlipLS(Child1, G)  33. MemeticBitFlipLS(Child2, G)  34. P\_new ← P\_new ∪ {Child1, Child2}  35. end while  36. // 2.7 Đánh giá quần thể mới  37. for each individual ∈ P\_new do  38. fitness[individual] ← EvaluateFitness(individual)  39. end for  40. // 2.8 Cập nhật BestSolution  41. candidate ← argmax\_{individual ∈ P\_new} fitness[individual]  42. if fitness[candidate] > fitness[BestSolution] then  43. BestSolution ← candidate  44. end if  45. // 2.9 Cập nhật quần thể  46. P ← P\_new  47. end for  48. // 3. Trả về kết quả  49. return BestSolution |

Độ phức tạp thuật toán:

* Khởi tạo quần thể, mỗi cá thể là một số:
* Đánh giá fitness: cần kiểm tra mỗi cạnh có được cover hay không, với mỗi cá thể duyệt toàn bộ cạnh.
* Vòng lặp tiến hóa chạy lần và mỗi thế hệ bao gồm:
* Chuẩn bị quần thể mới: while lặp khoảng lần, vì mỗi lần sinh 2 cá thể.
* Tournament Selection (dòng 7–9): có hai lần gọi mỗi lần chọn ngẫu nhiên cá thể:
* Lai ghép (dòng 11–18): sao chép và chia các bit
* Đột biến (dòng 20–22): lật bit nhưng vẫn duyệt toàm bộ
* Repair (dòng 24–26): đảm bảo cover hết cạnh nhưng cần phải duyệt hết cạnh .
* Thêm vào quần thể mới: .

Tổng cho mỗi cặp con:

Với lần lặp thì mỗi thế hệ:

* Đánh giá fitness quần thể mới (dòng 31–33):
* Cập nhật BestSolution và thay quần thể (dòng 34–41): tìm best trong .

Tổng cho 1 thế hệ:

Tổng độ phức tạp:

Nếu coi là một số nhỏ thì:

**2.5.3 Leo đồi**

Thuật toán Leo đồi (Hill Climbing) là một phương pháp tìm kiếm cục bộ đơn giản nhưng hiệu quả trong việc tối ưu hóa lời giải. Ý tưởng chính của thuật toán là bắt đầu từ một lời giải ban đầu, sau đó liên tục di chuyển tới một lời giải lân cận tốt hơn cho đến khi không còn khả năng cải thiện.

Trong bối cảnh bài toán Vertex Cover, *Hill Climbing* kết hợp với chiến lược *Stochastic Local Search* và *chọn lọc moveType thích ứng* (Adaptive Move Selection) cho phép khai thác không gian lời giải cục bộ một cách linh hoạt và có kiểm soát, tránh xét toàn bộ neighbor nhưng vẫn giữ khả năng cải thiện giải pháp.

Bản thuật toán khởi tạo một lời giải hợp lệ, sau đó lặp lại tối đa bước (hoặc đến khi không còn cải thiện) bằng cách sinh ra một số lượng move ngẫu nhiên có giới hạn mỗi bước. Tùy theo số cạnh chưa được phủ và độ lớn hiện tại của cover, thuật toán ưu tiên các thao tác *add, remove* hoặc *swap*, đồng thời áp dụng kiểm tra nhanh chỉ trên các cạnh liên quan để đánh giá tính hợp lệ. Cuối cùng, lời giải tốt nhất quan sát được được trả về.

Quy trình hoạt động của thuật toán:

**Input/Output:**

* Input: đồ thị phương pháp khởi tạo lời giải ban đầu , số bước tối đa (hoặc bất kì điều kiện dừng nào khác), giới hạn số move mỗi vòng , kích thước (ước lượng) của cover:
* Output: tập Vertex Cover tối ưu tìm được.

**Mã giả:**

|  |
| --- |
| 1. // 1. Khởi tạo  2. currentCover ← InitialSolution(G)  3. bestCover ← currentCover  4. iter ← 0  5. // 2. Vòng lặp tìm kiếm  6. while iter < maxIter do  7. N ← ∅  8. moveCount ← 0  9. uncoveredEdges ← CountUncoveredEdges(currentCover, G)  10. // 2.1 Sinh move có hướng ưu tiên  11. while moveCount < MaxMovePerIter do  12. if uncoveredEdges > 0 then  13. moveType ← RandomChoice(["add", "swap"], [0.7, 0.3])  14. else if uncoveredEdges = 0 and |currentCover| > targetSize then  15. moveType ← RandomChoice(["remove", "swap"], [0.7, 0.3])  16. else  17. moveType ← RandomChoice(["add", "remove", "swap"], [0.33, 0.33, 0.34])  18. end if  19. if moveType = "add" then  20. u ← RandomSelect(V \ currentCover)  21. C' ← currentCover ∪ {u}  22. else if moveType = "remove" then  23. v ← RandomSelect(currentCover)  24. C' ← currentCover \ {v}  25. else if moveType = "swap" then  26. v ← RandomSelect(currentCover)  27. u ← RandomSelect(V \ currentCover)  28. C' ← (currentCover \ {v}) ∪ {u}  29. end if  30. // 2.2 Kiểm tra nhanh  31. if IsValidFast(C', G) then  32. N ← N ∪ {C'}  33. end if  34. moveCount ← moveCount + 1  35. end while  36. // 2.3 Đánh giá và chọn neighbor tốt nhất  37. if N ≠ ∅ then  38. bestNeighbor ← argmin\_{C ∈ N} |C|  39. // 2.4 Cải thiện  40. if |bestNeighbor| < |bestCover| then  41. currentCover ← bestNeighbor  42. bestCover ← bestNeighbor  43. iter ← iter + 1  44. else  45. break // Không cải thiện → dừng  46. end if  47. else  48. break // Không sinh được move hợp lệ → dừng  49. end if  50. end while  51. // 3. Trả về kết quả  52. return bestCover |

Độ phức tạp thuật toán:

* Khởi tạo lời giải hợp lệ ban đầu và gán biến (2-4):
* Mỗi vòng lặp chính (tối đa lần) (6-50):
* Sinh không gian lân cận (11-35):
* Mỗi bước sinh một move, giới hạn moves.
* Add/remove/swap move: duyệt qua đỉnh. Mỗi đỉnh lại cần kiểm tra hợp lệ . (với D là bậc đỉnh cao nhất của đồ thị)
* Duyệt tập neighbor so sánh kích thức bitmask và chọn : O(MaxMovePerIter).
* Cập nhật cover: .

Tổng độ phức tạp thuật toán là:

Nếu và thì:

**2.3.2 Giải thuật tham lam**

Giải thuật tham lam là một phương pháp xấp xỉ đơn giản nhưng hiệu quả cho bài toán Vertex Cover. Ý tưởng cơ bản của phương pháp này là lặp lại quá trình lựa chọn các đỉnh vào tập phủ sao cho thỏa mãn một tiêu chí cục bộ nhất định – phổ biến nhất là lựa chọn đỉnh có bậc cao nhất tại thời điểm hiện tại. Việc lựa chọn này nhằm tối đa hóa số lượng cạnh được phủ tại mỗi bước, từ đó nhanh chóng giảm kích thước bài toán.

Mã giả:

|  |
| --- |
| 1: Input: G = (V, E)  2: cover ← ∅  3:  4: while E ≠ ∅ do  5: chọn đỉnh v có bậc cao nhất  6: thêm v vào cover  7: xóa v và tất cả các cạnh kề khỏi đồ thị  8:  9: return cover |

Độ phức tạp thuật toán:

* Khởi tạo: duyệt qua danh sách kề để tính bậc của đỉnh: .
* Vòng lặp while: duyệt qua tối đa đỉnh: , duyệt lại danh sách cạnh để xóa tất cả các cạnh có chứa đỉnh vừa chọn, tối đa: .

Vậy độ phức tạp tổng hợp là: .

Độ phức tạp bộ nhớ là .S

1. Thử nghiệm và so sánh hiệu suất.

Trogn phạm vi bài báo cáo này, chúng em sử dụng nguồn đồ thi theo chẩn DIMACS ở trang [DIMACS Networks | Network Data Repository](https://networkrepository.com/dimacs.php)

6. Tài liệu tham khảo