**TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**KHOA/VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

**---🙠**🕮**🙢---**



**PROJECT 1**

**Đề tài: Thuật toán TabuSearch để giải bài toán TSP và VRPTW**

***Giáo viên hướng dẫn: Ts.Nguyễn Khánh Phương***

***Sinh viên thực hiện: Lê Duy Anh – 20204629***

***Trần Xuân Bách - 20204634***

***Năm học 2022-1***

Mục lục

[I) Tabu search 3](#_Toc124930280)

[II) Tabu search để giải bài toán TSP( Traveling salesman problem) 6](#_Toc124930281)

[Phát biểu bài toán : 6](#_Toc124930282)

[+Áp dụng Tabu search để giải bài toán TSP 7](#_Toc124930283)

[Minh họa thuật toán : 8](#_Toc124930284)

[Đồ thị thể hiện giá trị hàm mục tiêu và giá trị tốt nhất qua mỗi bước. 10](#_Toc124930285)

[CODE (C++): 11](#_Toc124930286)

[Bảng so sánh thời gian: 13](#_Toc124930287)

[III) Thuật toán Tabu Search để giải bài toán Vehicle Routing Problem with Time Window ( VRPTW ). 14](#_Toc124930288)

[Phát biểu bài toán : 15](#_Toc124930289)

[Thuật toán : 16](#_Toc124930290)

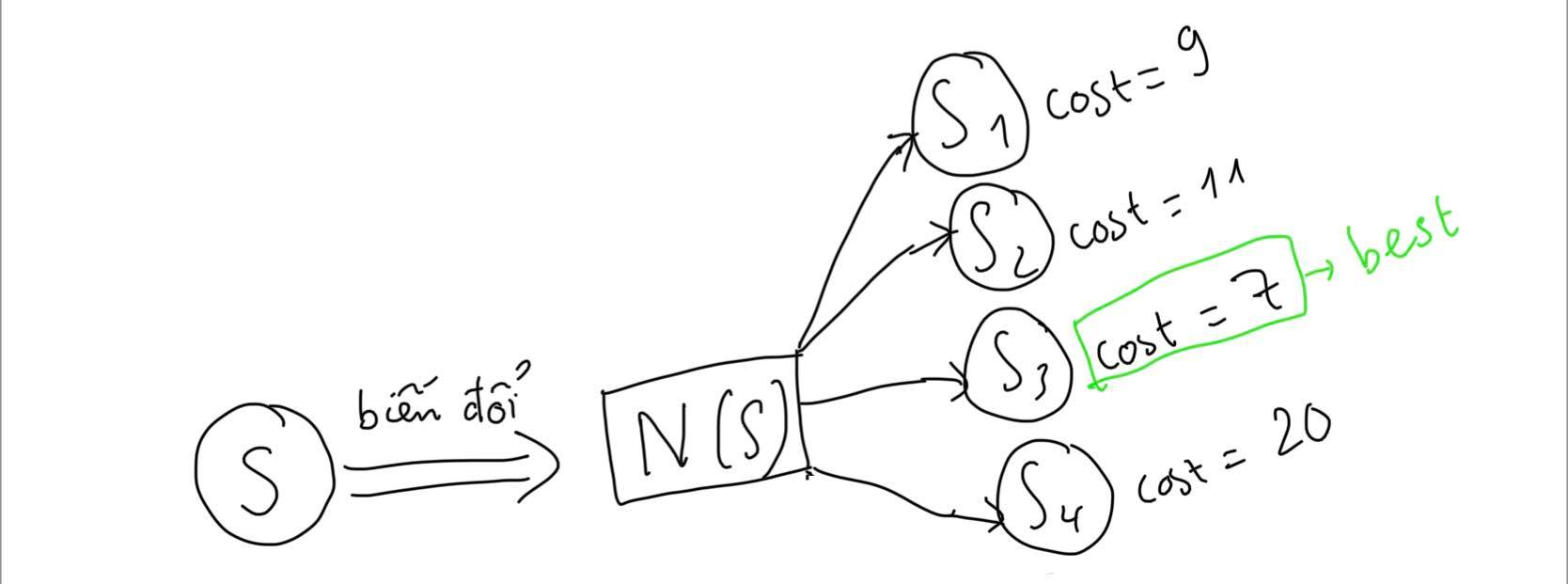
[CODE: (C++) 20](#_Toc124930291)

[KẾT LUẬN 27](#_Toc124930292)

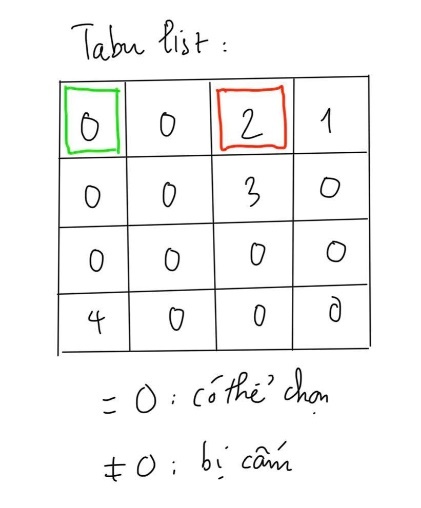
1. Tabu search

-“Tabu” có nghĩa là cấm kỵ trong tiếng Tongan. Tabu seach là một phương pháp tìm kiếm cấp cao để giải quyết một lớp các bài toán tối ưu, được đề xuất bởi Fred Glover vào năm 1986.

-Phương pháp tìm kiếm Tabu search xuất phát từ một nghiệm ban đầu S (initial solution) , sau đó biến đổi để tìm các nghiệm lân cận N(S) (neighborhoods), từ đó chọn ra nghiệm tốt nhất trong N(S). Khác với các phương pháp cổ điển, ta chấp nhận giá trị của hàm mục tiêu có thể xấu đi với hi vọng tìm được nghiệm tối ưu toàn cục (global maximum).



-Tuy nhiên để tránh tình trạng quay vòng (cycling), ta sử dụng danh sách cấm kỵ (tabu list) để lưu trữ các nghiệm mà ta sẽ không thăm ở lượt tiếp theo, vì điều kiện máy móc có hạn nên ta không thể lưu trữ tất cả các nghiệm đã thăm. Ta chỉ lưu trong thời gian t (Tabu-tenure) là thời gian bị cấm kỵ. (Thường ta lấy t= sqrt(N) ).



-Aspiration conditions cung cấp cho chúng ta ngoại lệ của Tabu: Nếu một nghiệm cấm kỵ có giá trị tốt hơn tất cả các giá trị mà ta biết tới trước đây thì ta vẫn lấy nghiệm đó (overridden) và lại cho vào danh sách tabu 1 lần nữa.

-Điều kiện dừng của thuật toán Tabu: (Stopping condition) có nhiều loại điều kiện dừng:

+Khi N(S) rỗng (Không còn nghiệm lân cận khả thi).

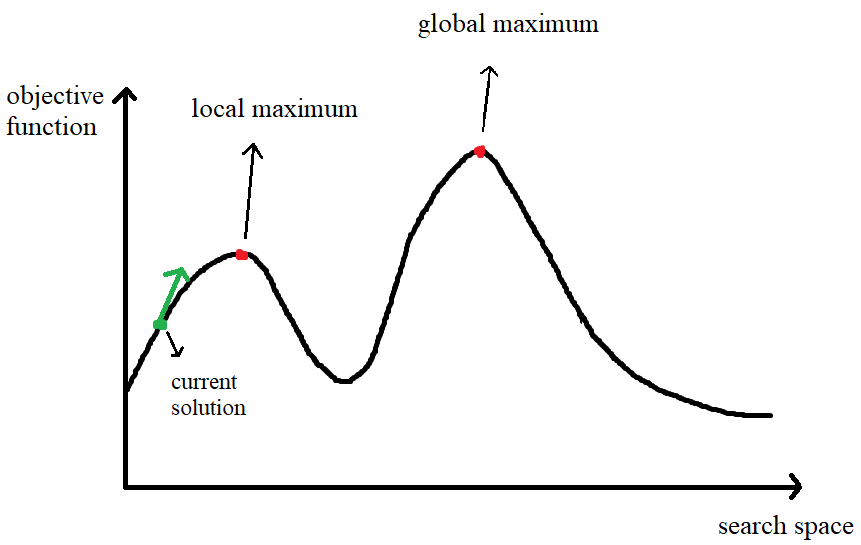
+Số bước thực hiện bị giới hạn bởi 1 số. (K<=Q).

+Số bước thực hiện từ lần cập nhật i\* gần nhất vượt quá 1 số nào đó.(VD: Sau 100 bước ta không tìm thấy giá trị tốt hơn thì sẽ dừng).

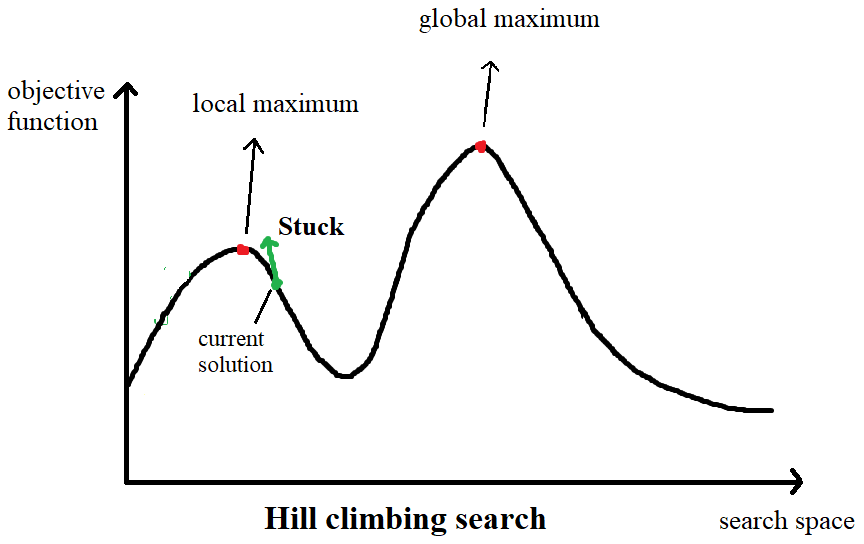
+Nhận định rằng nghiệm tối ưu đã được tìm thấy(VD:

Ta chứng minh được nghiệm <=75 , và ta đã tìm thấy 1 nghiệm = 75 thì đó là nghiệm tối ưu và dừng thuật toán).

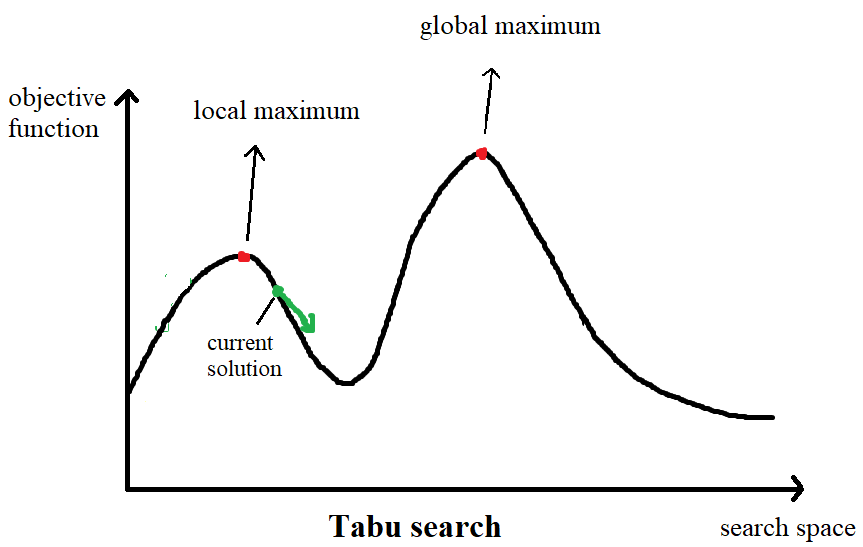
-Vậy Tabu search hoàn chỉnh hơn Hill climbing search (tìm kiếm leo đồi) ở điểm gì?



Đây là đồ thị không gian tìm kiếm và giá trị hàm mục tiêu, với giải thuật tìm kiếm leo đồi thì khi tìm được local maximum (cực đại địa phương), nghiệm S sẽ bị kẹt lại tại local maximum này và không tìm được global maximum (nghiệm tối ưu toàn cục).



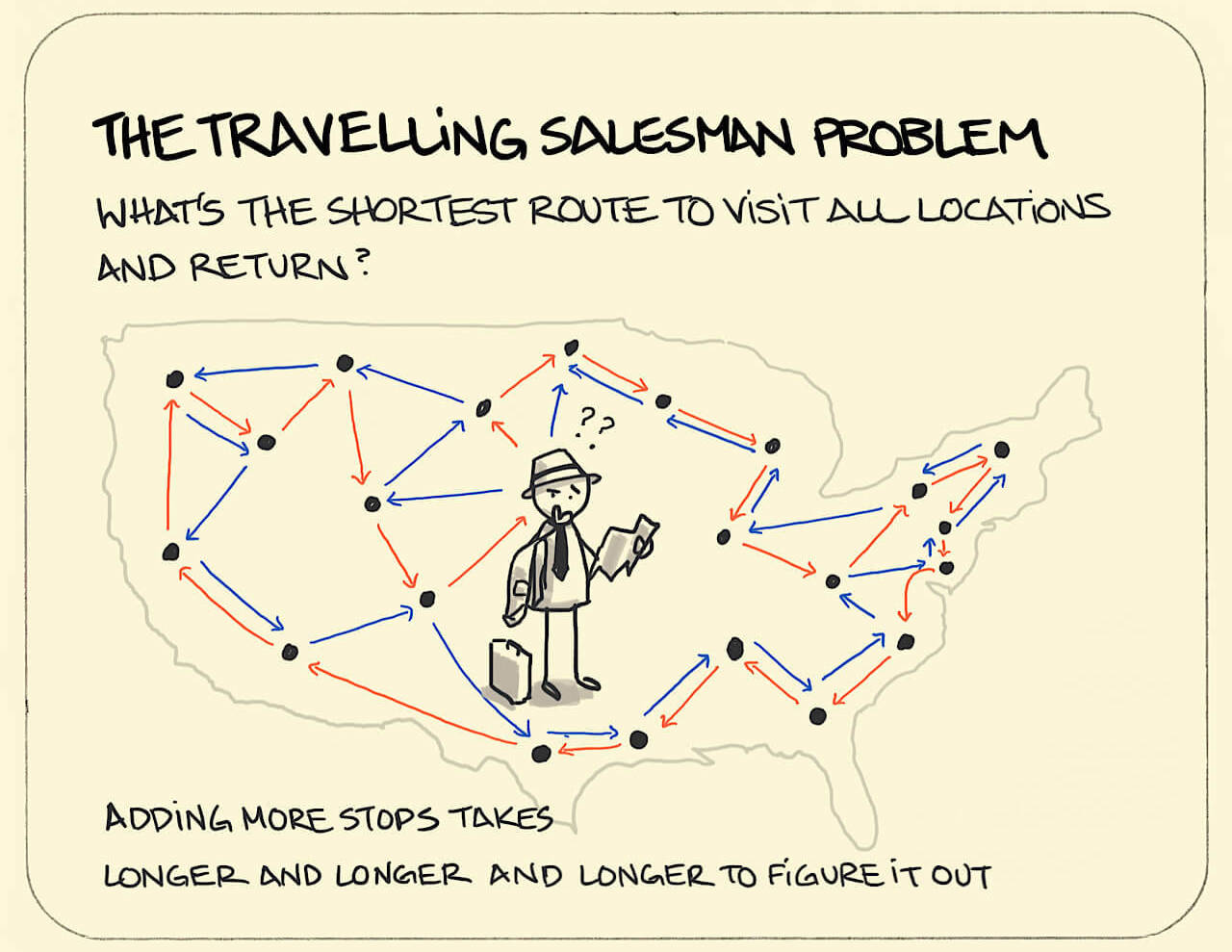
Trong khi đó, Tabu search sẽ có 1 Tabu list để ngăn chặn việc nghiệm bị kẹt tại local maximum, chấp nhận giá trị hàm mục bị giảm để có thể tìm ra global maximum.



1. Tabu search để giải bài toán TSP( Traveling salesman problem)

## Phát biểu bài toán :

Một người du lịch muốn đi từ thành phố 1 qua tất cả n-1 thành phố còn lại ( mỗi thành phố chỉ đi qua 1 lần) và quay lại thành phố 1 sao cho chi phí là nhỏ nhất. Biết tồn tại đường đi từ một thành phố đến mọi thành phố, chi phí khi đi từ TP a sang TP b và ngược lại là bằng nhau.



Ảnh có chứa dây điện

Mô tả được tạo tự động

Nghiệm ban đầu : 1->2->3->4->5->1 : Cost=23

Nghiệm tối ưu : 1->3->5->2->4->1 : Cost=10

## +Áp dụng Tabu search để giải bài toán TSP

-Ta có Tabu list ban đầu = 0

**B1** : Chọn trạng thái bắt đầu là 1->2->3->4->5->1

**B2** : Bắt đầu một iteration, ta sẽ giảm tất cả các giá trị khác 0 trong tabu list đi 1 đơn vị.

Ta chuyển đổi trạng thái bằng cách hoán đổi 2 city bất kỳ trong n-1 city, như vậy sẽ có (n-1)\*(n-2)/2=6 cách chọn. Sau đó ta tính toán giá trị hàm mục tiêu với mỗi lân cận đó. Và chọn ra cặp 2 thành phố tốt nhất mà không vi phạm tabu list ( giá trị của cặp thành phố đó trong tabu list = 0).

Sau khi chọn 2 thành phố u, v để hoán đổi thì ta sẽ thay đổi giá trị tenure[u][v]=t.

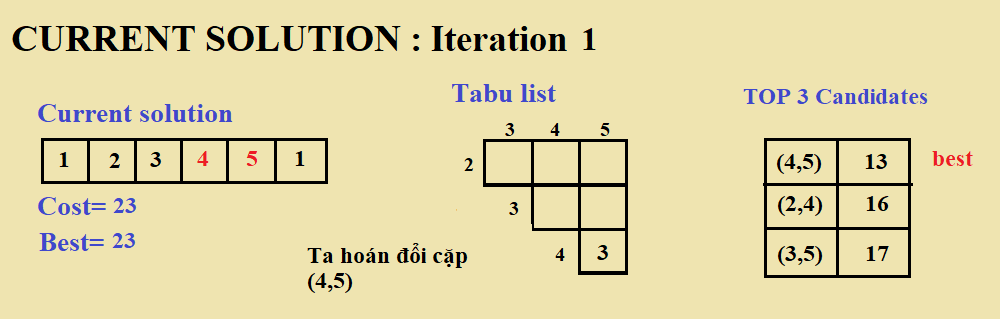
Lưu ý: Nếu một cách hoán đổi u, v mà vi phạm tabu list nhưng tạo ra giá trị hàm mục tiêu lớn chưa từng có thì ta sẽ chấp nhận hoán đổi này (overridden).

**B3** : Kiểm tra điều kiện dừng :

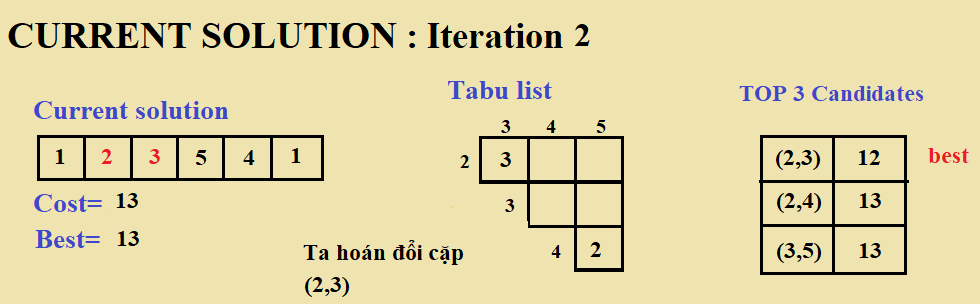
Sau 7 iterations thì ta sẽ dừng thuật toán.

Nếu chưa dừng thì quay lại B2, nếu dừng thì dừng thuật toán và in ra kết quả.

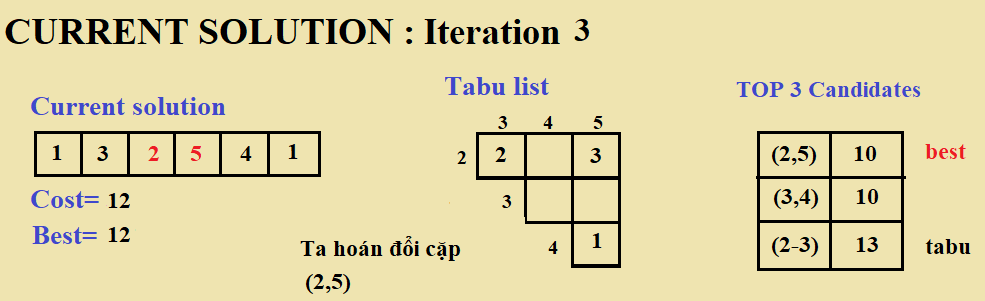
## Minh họa thuật toán :



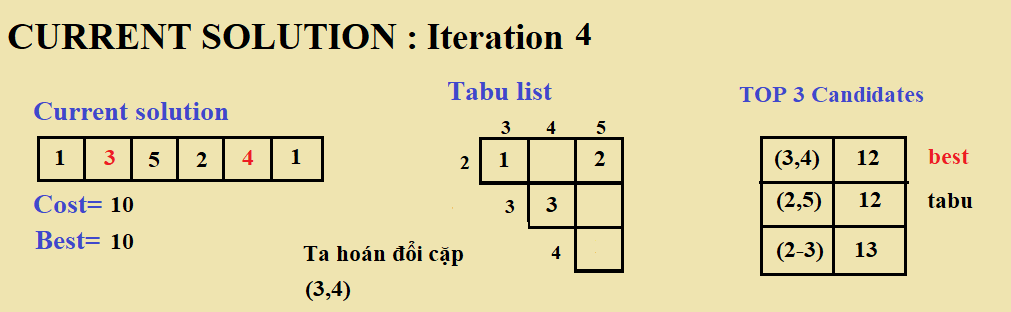
Bước 1: Đổi chỗ cặp (4,5) sẽ cho ta kết quả tốt nhất =13, đồng thời ta đặt tabu-tenure của (4,5) = 3.

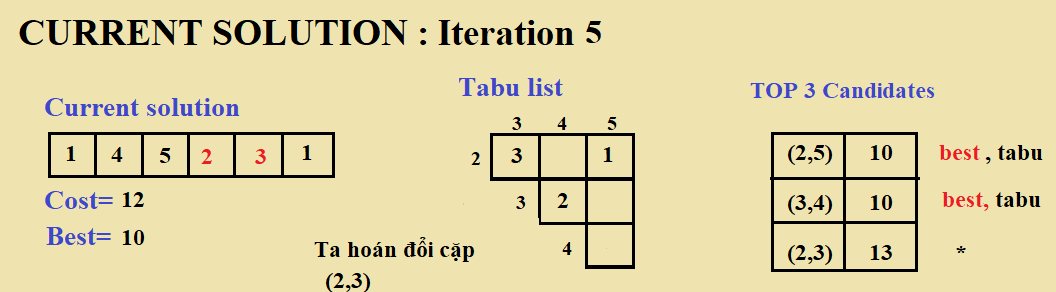


Bước 2: Giảm thời gian cấm của các cặp trong tabu list đi 1 đơn vị. Đổi chỗ cặp (2,3) sẽ cho ta kết quả tốt nhất =12, đồng thời ta đặt tabu-tenure của (2,3) = 3.

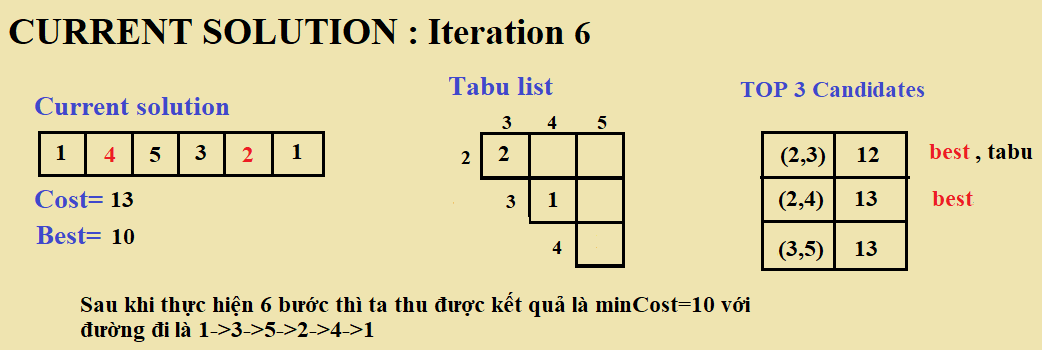


Bước 3: Tương tự bước 2 và ta thu được Best = 10.

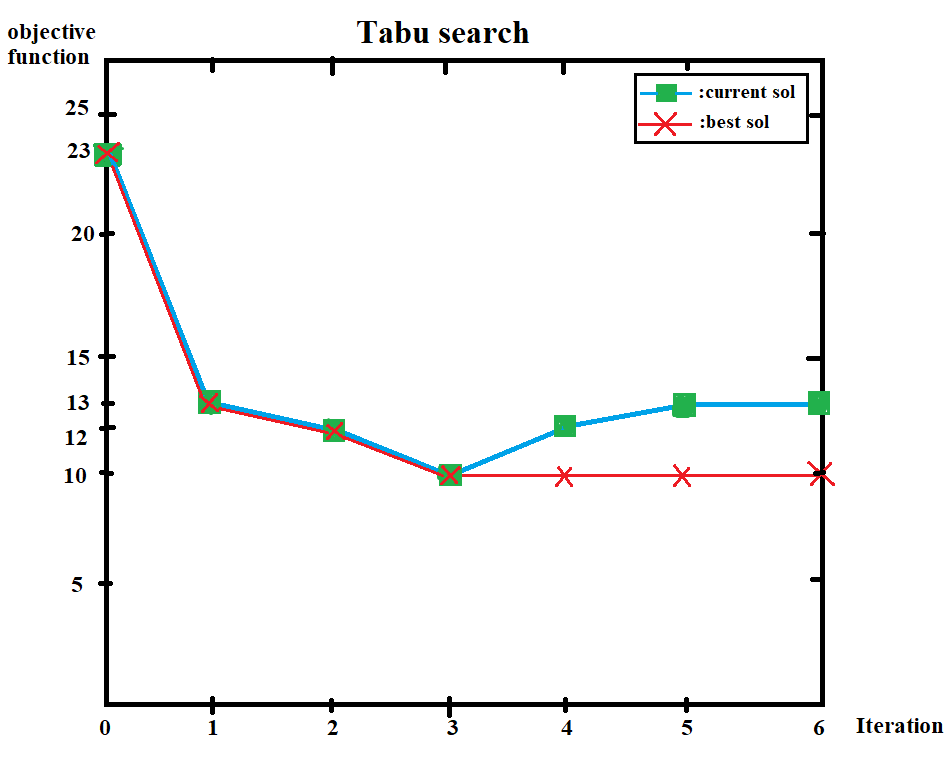


Bước 4: Ở bước này ta không thu được kết quả tốt hơn nhưng vẫn hoán đổi (3,4) với hy vọng tìm được kết quả tốt hơn nếu có. 

Bước 5: Trường hợp này ta hoán đổi cặp (2,3) vì 2 cặp tốt hơn kia đều đang bị tabu và kết quả không tốt hơn Best.



Bước 6: Kết thúc thuật toán sau 6 iterations và ta thu được minCost=10 với đường đi là 1->3->5->2->4->1.



## Đồ thị thể hiện giá trị hàm mục tiêu và giá trị tốt nhất qua mỗi bước.

## CODE (C++):

|  |
| --- |
| #include <bits/stdc++.h>  #define ll long long  #define pi 3.141592653589793238462  #define big 1000000007  #define fi first  #define se second  #define FOR(i,a,b) for(int i=a; i<=b; ++i)  using namespace std;  int n, c[105][105], x[105], xOPT[105], fOPT, f, k, tabu[105][105], tenure;  struct Edge{  int u, v, cost;  };  bool operator < (Edge a, Edge b) {  if(a.cost==b.cost) return a.u<b.u;  return a.cost > b.cost;  }  int countf(){  int sum=0;  FOR(i,1,n-1) sum+=c[x[i]][x[i+1]];  sum+=c[x[n]][1];  return sum;  }  void TabusearchTSP(){  FOR(i,1,n) x[i]=i;  FOR(i,1,n) xOPT[i]=i;  f=countf();  fOPT=f;  while(k--){  /\*cout<<f<<'\n';  FOR(i,1,n) cout<<x[i]<<"->"; cout<<1<<'\n';\*/  FOR(i,2,n) FOR(j,i+1,n) if(tabu[i][j]) tabu[i][j]--;  priority\_queue<Edge> q;  for(int i=2; i<n; ++i){  for(int j=i+1; j<=n; ++j){  swap(x[i],x[j]);  q.push({i,j,countf()});  swap(x[i],x[j]);  }  }  while(!q.empty()){  Edge tmp = q.top();  q.pop();  int a1=x[tmp.u], a2=x[tmp.v];  if(a1>a2) swap(a1,a2);  if(tabu[a1][a2]){  swap(x[tmp.u],x[tmp.v]);  f=countf();  if(f<fOPT){  FOR(i,1,n) xOPT[i]=x[i];  fOPT=f;  tabu[a1][a2]=tenure;  break;  }  else swap(x[tmp.u],x[tmp.v]);  }  else if(!tabu[a1][a2]){  swap(x[tmp.u],x[tmp.v]);  f=countf();  if(f<fOPT){  FOR(i,1,n) xOPT[i]=x[i];  fOPT=f;  }  tabu[a1][a2]=tenure;  break;  }  }  while(!q.empty()) q.pop();  }  }  int main(){  cin >> n >> k >> tenure;  FOR(i,1,n) FOR(j,1,n) cin >> c[i][j];  TabusearchTSP();  cout<<fOPT<<'\n';  FOR(i,1,n) cout<<xOPT[i]<<"->"; cout<<1;  return 0;  } |

**Độ phức tạp** : O(k\*n2) với k là số bước.

Việc chọn k = bao nhiêu rất quan trọng, ảnh hưởng đến kết quả cuối cùng. Nếu chọn k nhỏ thì có thể ta sẽ không tìm được đáp án tối ưu, chọn k càng lớn thì càng tăng xác suất tìm được đáp án tối ưu, tuy nhiên chọn k quá lớn thì tốc độ thực hiện sẽ rất chậm => Ta phải trade-off giữa thời gian chạy chương trình với độ chính xác của đáp án.

## Bảng so sánh thời gian:

Ta có bảng so sánh thời gian chạy và đáp án tìm được với trường hợp n=10, tenure=5 và K tăng dần:

|  |  |  |
| --- | --- | --- |
| K | fOPT | Thời gian chạy |
| 101 | 74 | 0.013(s) |
| 102 | 60 | 0.017(s) |
| 103 | 60 | 0.035(s) |
| 104 | 60 | 0.214(s) |
| 105 | 60 | 1.889(s) |

Nhận xét : Khi thử lại bằng thuật toán TSP nhánh cận thì đáp án = 60 là chính xác !

Trường hợp n=20, tenure=10 và K tăng dần:

|  |  |  |
| --- | --- | --- |
| K | fOPT | Thời gian chạy |
| 101 | 76 | 0.015(s) |
| 102 | 57 | 0.030(s) |
| 103 | 41 | 0.126(s) |
| 104 | 41 | 1.036(s) |
| 105 | 41 | 10.087(s) |

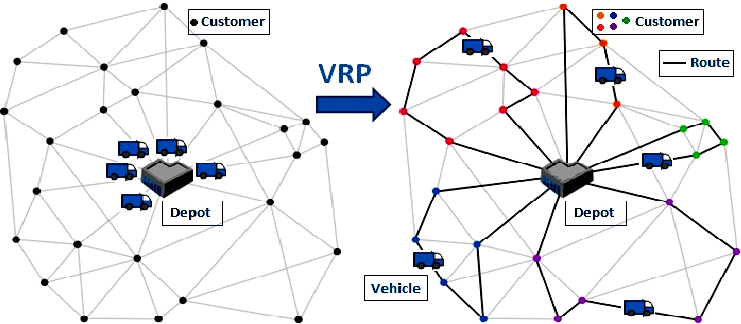
Nhận xét : Khi thử lại bằng thuật toán TSP nhánh cận thì đáp án = 37.

Như vậy là đáp án do thuật toán Tabu search chỉ gần tối ưu chứ chưa tối ưu. Do ta chưa chọn K đủ lớn và tenure đủ chuẩn!

1. Thuật toán Tabu Search để giải bài toán Vehicle Routing Problem with Time Window ( VRPTW ).

VRP là bài toán xác định các lộ trình tối ưu cho đội xe vận chuyển nhằm phục vụ các khách hàng ở các vị trí khác nhau. Đây là bài toán có nhiều ứng dụng trong thực tế : vận chuyển hàng hóa của các nhà máy sản xuất, vận chuyển hành khách… . Nếu tìm được lộ trình tối ưu cho các xe thì ta sẽ giảm được chi phí. Không chỉ vậy, nhu cầu của khách hàng về thời gian đáp ứng, thời gian hoạt động của công ty cũng phải được tuân thủ, …

Vì vậy, bài toán VRPTW là một trong những bài toán được quan tâm nhất. VRPTW là bài toán NP-hard với mục tiêu là tối thiểu số xe vận chuyển và tổng thời gian di chuyển khi phục vụ các khách trong thời gian ràng buộc và không vi phạm khả năng tải tối đa của xe. Ở đây ta sẽ sử dụng thuật toán metaheuristic TabuSearch để tiếp cận cũng như giải bài toán này.



## Phát biểu bài toán :

Tại một kho hàng (Depot) có M xe vận chuyển {1,2,3…,M}, mỗi xe có tải tối đa là Q và có N điểm, mỗi điểm là một khách hàng với nhu cầu là xi và thời gian có thể phục vụ là [ai,bi], biết thời gian di chuyển giữa 2 điểm i, j là cij.

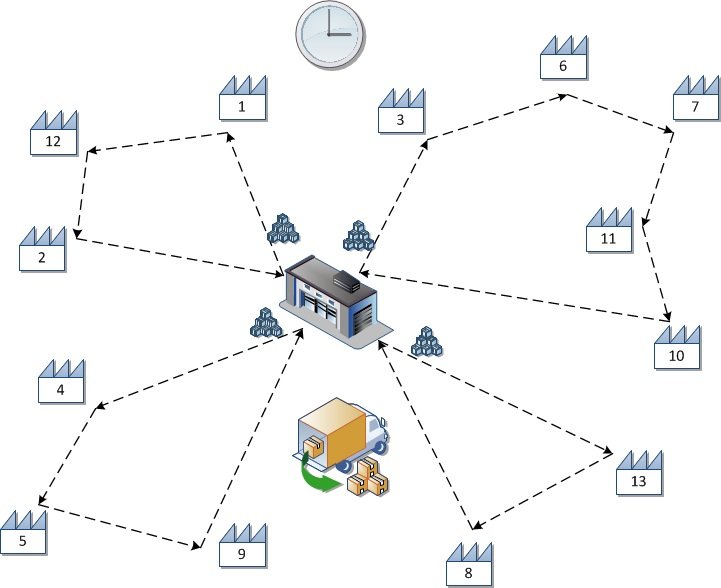
Mục tiêu : Xây dựng nhiều nhất M tuyến đường xuất phát từ 0 (depot) qua 1 số điểm và quay lại 0 sao cho :

1/ Tất cả các khách đều thuộc 1 tuyến đường duy nhất.

2/ Tổng nhu cầu của các khách trên 1 tuyến không vượt quá Q : .

3/ Xe phải phục vụ khách i trước bi, nếu xe đến trước thời điểm ai thì sẽ phải đợi.

Tổng thời gian lộ trình của tất cả các xe là nhỏ nhất.



## Thuật toán :

1) Tạo solution ban đầu:

Có nhiều cách để chọn ra solution ban đầu:

+Thuật toán tham lam:

-Tại mỗi bước ta sẽ chọn ra khách hàng có vị trí gần nhất so với khách hàng hiện tại và thêm vào solution.

+Thuật toán Clarke-Wright savings

+Thuật toán chèn ngẫu nhiên

+…

2) Sinh hàng xóm :

Từ lời giải hiện tại => sinh ra các lời giải hàng xóm (tối đa num\_neighbors hàng xóm) bằng cách :

+ Hoán đổi khách hàng trong cùng 1 tuyến:

-Đổi 2 khách bất kỳ trong cùng 1 tuyến xe: d3, p3.

Ảnh có chứa văn bản, đồng hồ

Mô tả được tạo tự động

-Lấy 1 khách bất kỳ ra khỏi tuyến rồi thêm lại vào vị trí khác:

+Lấy d­3 ra rồi thêm vào giữa p1, p2.

Ảnh có chứa văn bản, đồng hồ

Mô tả được tạo tự động

* Sử dụng 2-opt exchange:

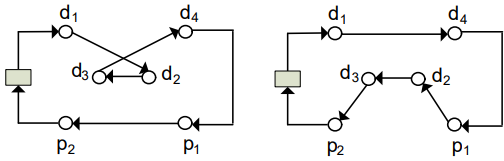
+ Nối d1 với p2 và p1 với d2

troẢnh có chứa văn bản, đồng hồ

Mô tả được tạo tự động

-Or-opt:

+Lấy đoạn d1->d2 ra rồi cho vào giữa p1->p2



+ Hoán đổi khách hàng trong 2 tuyến khác nhau:

-Đổi 2 khách bất kỳ trong 2 tuyến xe: p2, d2

Ảnh có chứa đồng hồ

Mô tả được tạo tự động

-Sử dụng 2-opt\* exchange:

+Chọn 2 cạnh thuộc 2 tuyến khác nhau : d1->d2 và p1->p­2

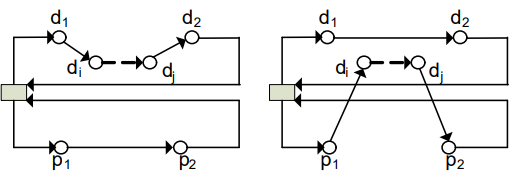
Sau đó nối d1->p2 và p1->d2.

Ảnh có chứa văn bản, đồng hồ, ảnh chụp màn hình

Mô tả được tạo tự động

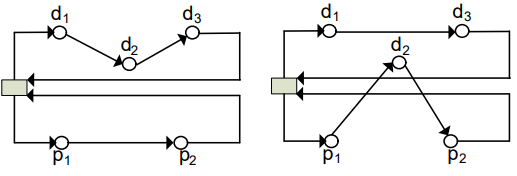
-Sử dụng Or-opt exchange:

+Chọn 1 đoạn (2 đến 3 khách liên tiếp) di->dj trong 1 tuyến bất kỳ sau đó chèn đoạn đó vào giữa 2 khách p1, p2.



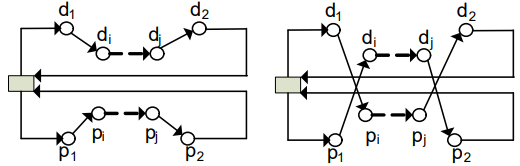
-Định tuyến lại cho 1 khách:

+Lấy khách d2 ra khỏi tuyến rồi thêm vào giữa p1 và p2



-Sử dụng CROSS-exchange:

+Chọn 1 đoạn khách liên tiếp d­i->dj và pi->pj từ 2 tuyến khác nhau sau đó hoán đổi vị trí 2 đoạn này:



\*Lưu ý: hàng xóm sinh ra không được vi phạm tabu\_list.

3) Evaluate Solution ( Đánh giá solution)

+Với mỗi hàng xóm sinh được thì ta sẽ tính giá trị của hàm xóm đó bằng cách : tính tổng chi phí từ depot qua các điểm rồi quay lại depot, tính cả thời gian xe chờ (=total\_distance) và tổng phạt (=total\_penalty).

+Trong đó, total\_penalty đưọc tính bằng cách : tính tổng thời gian khách phải đợi nếu xe đến sau thời gian có thể phục vụ ([a,b], xe đến sau b), và nếu dùng quá M xe thì ta sẽ tăng penalty thêm 1 lượng là (Số xe dùng – M) \* BigNum (BigNum là 1 số rất lớn).

+Ta tính giá trị của hàng xóm để chọn ra hàng xóm khả thi tốt nhất hoặc hàng xóm tốt nhất để chọn làm lời giải hiện tại rồi tiếp tục các bước.

\* Nếu hàng xóm mới khả thi thì ta sẽ cập nhật solution hiện tại, nếu giá trị tốt hơn best solution thì ta cập nhật thêm best solution.

Nếu không có hàng xóm nào khả thi thì ta sẽ chọn hàng xóm có giá trị tốt nhất (total\_distance+total\_penalty)

Ví dụ: Hàng xóm A: total\_distance = 22, total\_penalty=0

Hàng xóm B: total\_distance = 15, total\_penalty=5

Thì ta sẽ chọn hàng xóm A dù tổng giá trị (Evaluate) tệ hơn.

4) Tabusearch:

Mỗi khi chọn được một lời giải mới thì ta sẽ thêm từng phần tử vào danh sách tabu tại vị trí đó.

Ví dụ: 3->5->2->4->1

Thì 3 sẽ bị cấm tại vị trí 1 trong tabu\_tenure lượt.

5 sẽ bị cấm tại vị trí 2 trong tabu\_tenure lượt.

…..

5) Đưa ra lời giải cuối cùng:

Sau khi thực hiện T iterations, ta sẽ dừng thuật toán và đưa ra lời giải cuối cùng (nếu có).

CODE: (C++)

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  struct Customer {  int id;  int demand;  int a, b;  };  vector<Customer>customers;  int m, n, vehicle\_capacity, cost[105][105], best=INT\_MAX;  bool check;  vector<vector<int>> routes, tabu\_list;  vector<int>current\_solution, route\_demand, best\_sol;  // các tham số đầu vào  const int tabu\_tenure = 2;  const int max\_iterations = 1000;  const int num\_neighbors = 50;  //  vector<int> create\_greedy\_initial\_solution(){  // tham lam lời giải ban đầu  vector<int> solution;  vector<bool> visited(customers.size(),false);  int cur\_customer = 0;  solution.push\_back(cur\_customer);  while(solution.size() <= n){  int nearest\_customer = -1;  int min\_dis = INT\_MAX;  for(int i=1; i <= n; ++i){  if(!visited[i]){  if(cost[cur\_customer][i] < min\_dis){  min\_dis = cost[cur\_customer][i];  nearest\_customer = i;  }  }  }  visited[nearest\_customer] = true; //thăm vị khách gần nhất  cur\_customer = nearest\_customer;  solution.push\_back(cur\_customer);  }  return solution;  }  int evaluate\_solution(){  // hàm đánh giá solution  int total\_dis = 0; //tổng chi phí  int total\_penalty = 0; // tổng phạt  routes.clear();  route\_demand.clear();  vector<int>route;  int cur\_dem = 0; //trọng tải hiện tại  int cur\_time = 0; //thời gian hiện tại  // lưu các routes cho lời giải hiện tại (current solution)  for(int i=1; i<=n; ++i){  int cus=current\_solution[i];  // nếu nhận khách 'cus' vào mà vượt quá trọng tải tối đa  // thì ta sẽ lưu route vào và tạo route mới  if(cur\_dem + customers[cus].demand > vehicle\_capacity){  total\_dis+=cost[route[route.size()-1]][0];  routes.push\_back(route);  route\_demand.push\_back(cur\_dem);  route.clear();  cur\_dem = 0;  }  route.push\_back(cus); //nhận khách hàng 'cus' vào route  cur\_dem += customers[cus].demand;  if(route.size()==1){ // nếu đây là route mới  cur\_time = max(cost[0][cus],customers[cus].a);  total\_dis += cost[0][cus]; // chi phí di chuyển  }  else{ // route cũ  cur\_time += cost[route[route.size()-2]][cus];  total\_dis += cost[route[route.size()-2]][cus]; // chi phí di chuyển  }  cur\_time = max(cur\_time, customers[cus].a); // xét trường hợp phải đợi để đón khách  if(cur\_time > customers[cus].b){ //nếu thời gian hiện tại vượt quá thời gian có thể chờ của khách  total\_penalty += (cur\_time - customers[cus].b);  }  }  if(route.size()>0){  routes.push\_back(route);  route\_demand.push\_back(cur\_dem);  total\_dis+=cost[route[route.size()-1]][0];  }  int x = routes.size();  total\_penalty+=max(x-m,0)\*10000; // nếu solution sử dụng quá m xe thì sẽ bị phạt  if(!total\_penalty) check=true; //kiểm tra xem solution có bị vi phạm time window không  return total\_dis + total\_penalty;  //cout<<total\_dis<<" "<<total\_penalty;  }  void generate\_neighbor(){  // sinh hàng xóm bằng cách đổi chỗ 2 khách i j bất kỳ  int i = rand() % n + 1;  int j = rand() % n + 1;  while(j==i) j = rand() % n + 1;  swap(current\_solution[i], current\_solution[j]);  }  int main(){  // input  cin >> m >> n >> vehicle\_capacity;  customers.push\_back({0,0,0,0});  for(int i=1; i<=n; ++i){  int demand, a, b;  cin >> demand >> a >> b;  customers.push\_back({i,demand,a,b});  }  for(int i=0; i<=n; ++i)  for(int j=0; j<=n; ++j)  cin >> cost[i][j]; // thời gian đi từ khách i->j  // tạo solution ban đầu = phương pháp tham lam  current\_solution = create\_greedy\_initial\_solution();  //cout<<evaluate\_solution()<<'\n';  tabu\_list.resize(n+1);  // tabu search  int best\_obj = evaluate\_solution();  int best\_feasible\_obj = INT\_MAX;  for(int i=0; i<max\_iterations; ++i){  vector<int>best\_neighbor = current\_solution;  vector<int>best\_feasible\_neighbor;  //sinh num\_neighbors hàng xóm  for(int j=1; j<=num\_neighbors; ++j){  generate\_neighbor(); // sinh hàng xóm  vector<int>neighbor = current\_solution;  bool is\_tabu = false;  for(int k=1; k<=n; ++k){  // kiểm tra từng phần tử thứ k xem có thuộc danh sách tabu\_list[k] hay không  if(find(tabu\_list[k].begin(),tabu\_list[k].end(), neighbor[k])!=tabu\_list[k].end()){  is\_tabu = true;  break;  }  }  if(!is\_tabu){  check=false;  int obj = evaluate\_solution(); // tính giá trị của solution  if(check==true){  if(obj<best) best=obj, best\_sol=neighbor; //nếu penalty = 0 và tốt hơn best thì cập nhật best  if(obj<best\_feasible\_obj){  best\_feasible\_obj = obj; // tìm hàng xóm tốt nhất (penalty = 0)  best\_feasible\_neighbor = neighbor;  }  }  if(obj < best\_obj){  best\_neighbor = neighbor; // tìm hàng xóm tốt nhất (penalty có thể !=0)  best\_obj = obj;  }  }  }  if(best\_feasible\_neighbor.size()){ // nếu tồn tại hàng xóm có penalty = 0 thì lấy hàng xóm đó  best\_neighbor = best\_feasible\_neighbor;  }  for(int j=1; j<=n; ++j){  tabu\_list[j].push\_back(best\_neighbor[j]);  if(tabu\_list[j].size()>tabu\_tenure){  tabu\_list[j].erase(tabu\_list[j].begin());  }  }  current\_solution = best\_neighbor;  }  // in ra final solution  current\_solution=best\_sol;  if(best\_sol.size()==0){  cout<<"NO SOLUTION!";  return 0;  }  cout << "Final solution: ";  for (int i = 1; i <= n; i++) {  cout << current\_solution[i] << " ";  }  cout << '\n';  cout << "Total distance: " << evaluate\_solution() << '\n';  // in ra các routes  cout << "Routes: " << '\n';  for (int r = 0; r < routes.size(); r++) {  cout << "Route " << r+1 << ": ";  for (int c = 0; c < routes[r].size(); c++) {  cout << routes[r][c] << " ";  }  cout << " (Demand: " << route\_demand[r] << ")" << '\n';  }  return 0;  } |

**ĐỘ PHỨC TẠP:** O(T.Q.N) : T là số iterations, Q là số hàng xóm sinh ra trong mỗi bước, N là số khách.

Test:

Ảnh có chứa văn bản

Mô tả được tạo tự động

**Kiểm thử Thuật toán TSP:**

**Ảnh có chứa bàn

Mô tả được tạo tự động**

# 

# KẾT LUẬN

Thuật toán Tabu Search là một giải thuật tối ưu metaheuristic được sử dụng đề tìm lời giải gần đúng cho các bài toán tối ưu hóa tổ hợp. Tabu search sử dụng bộ nhớ để lưu các thay đổi gần đây, cấm lặp lại thay đổi đó trong 1 số bước tiếp theo để tránh thăm lại các lời giải đã thăm (cycles) , tránh bị kẹt ở cực đại địa phương (local optima) trong không gian tìm kiếm. Giải thuật Tabu search có thể được áp dụng rộng rãi cho các bài toán tối ưu hóa tổ hợp như Traveling Salesman Problem (TSP), Vehicle Routing Prolem with Time Windows (VRPTW), Job-Shop Scheduling Problem (JSSP), … .Vì Tabusearch là một phương pháp heuristic nên ta chỉ có thể tìm được lời giải gần tối ưu chứ không chắc chắn tìm được lời giải tối ưu, với các tham số đầu vào khác nhau (max iterations, tabu tenure) và cách sinh hàng xóm khác nhau thì kết quả có thể tốt hơn hoặc tệ hơn tùy trường hợp. Nhìn chung, TabuSearch là một thuật toán đơn giản, có thể cài đặt trên mọi máy tính, yêu cầu ít tài nguyên và có thể tìm được một lời giải khá tốt cho bài toán trong thời gian ngắn.

**-THE END-**