**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**KHOA CÔNG NGHỆ THÔNG TIN**



**ĐỒ ÁN LẬP TRÌNH TÍNH TOÁN**

**BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT**

Người hướng dẫn**: TS. NGUYỄN VĂN HIỆU**

Sinh viên thực hiện**:**

**Trương Công Hoàng Long LỚP: 21TCLC\_Nhat1 NHÓM: 11**

**Trần Lê Minh LỚP: 21TCLC\_Nhat1 NHÓM: 11**

**Đà Nẵng, 07/2022**

MỤC LỤC

[MỤC LỤC 1](#_Toc105580243)

[DANH MỤC HÌNH VẼ 2](#_Toc105580244)

[MỞ ĐẦU 2](#_Toc105580245)

[1. TỔNG QUAN ĐỀ TÀI 3](#_Toc105580246)

[2. CƠ SỞ LÝ THUYẾT 3](#_Toc105580247)

[2.1. Ý tưởng 3](#_Toc105580248)

[2.2. Cơ sở lý thuyết 3](#_Toc105580249)

[3. TỔ CHỨC CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN 5](#_Toc105580250)

[3.1. Phát biểu bài toán 5](#_Toc105580251)

[3.2. Cấu trúc dữ liệu 5](#_Toc105580252)

[3.3. Thuật toán 5](#_Toc105580253)

[4. CHƯƠNG TRÌNH VÀ KẾT QUẢ 7](#_Toc105580254)

[4.1. Tổ chức chương trình 7](#_Toc105580255)

[4.2. Ngôn ngữ cài đặt: C 7](#_Toc105580256)

[4.3. Kết quả 7](#_Toc105580257)

[4.3.1. Giao điện chính của chương trình 7](#_Toc105580258)

[4.3.2. Kết quả thực thi của chương trình 9](#_Toc105580259)

[4.3.3. Nhận xét đánh giá 9](#_Toc105580260)

[5. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN 10](#_Toc105580261)

[5.1. Kết luận 10](#_Toc105580262)

[5.2. Hướng phát triển 10](#_Toc105580263)

[TÀI LIỆU THAM KHẢO 11](#_Toc105580264)

[PHỤ LỤC 11](#_Toc105580265)

DANH MỤC HÌNH VẼ

Hình 2.2a. Danh sách kề

Hình 2.2b Circular Queue

Hình 3.3a

Các hình ở mục 4.3.1 thể hiện giao diện của chương trình.

Hình 4.3.3a Test Dijkstra

Hình 4.3.3b Test Ford-Bellman

Hình 4.3.3c Chu trình âm với Ford-Bellman thuần

Hình 4.3.3d Chu trình âm với Ford-Bellman kết hợp Circular Queue

MỞ ĐẦU

Mục đích thực hiện đề tài: Trình bày một số kiến thức cơ bản của lí thuyết đồ thị liên quan đến bài toán đường đi ngắn nhất, giới thiệu bài toán và áp dụng một số thuật toán để giải quyết vấn đề.

Phạm vi và đối tượng nghiên cứu: Tập trung trình bày hai thuật toán Dijkstra và Ford-Bellman, đồng thời tìm hiểu và trình bày một số ứng dụng của bài toán tìm đường đi ngắn nhất trong thực tế.

Phương pháp nghiên cứu: Tìm hiểu sách tham khảo, video, bài giảng, các mẫu báo cáo trên mạng.

# TỔNG QUAN ĐỀ TÀI

Tìm đường đi ngắn nhất giữa các điểm trên bản đồ nhằm tối ưu chi phí nhất có thể đã trở nên quen thuộc trong thời đại ngày nay, khi mà việc giao thông đi lại toàn cầu phát triển mạnh mẽ. Có nhiều hướng kết nối hai vị trí lại với nhau nhưng dựa vào các yếu tố chẳng hạn quãng đường, địa hình, phương tiện, giao thông … mà ta quyết định đường đi ngắn nhất, tối ưu chi phí nhất. Chẳng hạn tìm đi ngắn nhất giữa hai thành phố, tìm tuyến xe buýt…

Mở rộng ra có thể kể đến ứng dụng trong tìm kiếm và kết nối bạn bè trên mạng xã hội, định tuyến trong hệ thống mạng …

# CƠ SỞ LÝ THUYẾT

## Ý tưởng

Thuật toán Ford-Bellman: Với đỉnh xuất phát S. Gọi d[v] là khoảng cách từ S tới v. Ban đầu d[v] được khởi gán bằng c[S, v]. Sau đó ta tối ưu hoá dần các d[v] như sau: Xét mọi cặp đỉnh u, v của đồ thị, nếu có một cặp đỉnh u, v mà d[v] > d[u]+ c[u, v] thì ta đặt lại d[v] := d[u] + c[u, v]. Tức là nếu độ dài đường đi từ S tới v lại lớn hơn tổng độ dài đuờng đi từ S tới u cộng với chi phí đi từ u tới v thì ta sẽ huỷ bỏ đuờng đi từ S tới v đang có và coi đuờng đi từ S tới v chính là đường đi từ S tới u sau đó đi tiếp từ u tới v. Chú ý rằng ta đặt c[u, v] = +∞ nếu (u, v) không là cung. Thuật toán sẽ kết thúc khi không thể tối ưu thêm bất kỳ một nhãn d[v] nào nữa.

Thuật toán Dijkstra: Việc tối ưu các nhãn tương tự thuật toán Ford-Bellman nhưng cách chọn đỉnh để tối ưu nhãn sẽ khác. Cụ thể tại mỗi buớc ta sẽ chọn đỉnh u là đỉnh mà không thể tối ưu nhãn d[u] thêm được nữa. Cụ thể là nhãn có giá trị nhỏ nhất trong các nhãn tự do, giả sử đó là d[u], ta chứng minh không thể tối ưu d[u] thêm nữa. Giả sử có thể tối ưu d[u]= d[v] + c[v, u], mà c[v,u] dương nên d[v] phải nhỏ hơn d[u], trái với cách chọn d[u] nhỏ nhất. Thuật toán dừng khi chọn đến nhãn d[F] hoặc không thể chọn được nhãn tiếp theo (không có đường đi ngắn nhất).

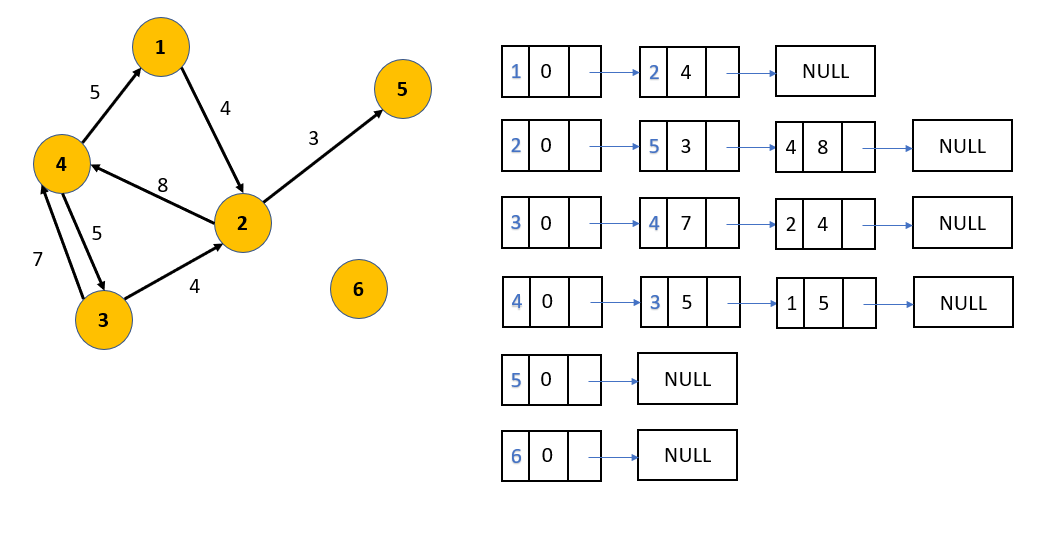
## Cơ sở lý thuyết

Đồ thị G=(V,E): V là các đỉnh, E là các cung có trọng số

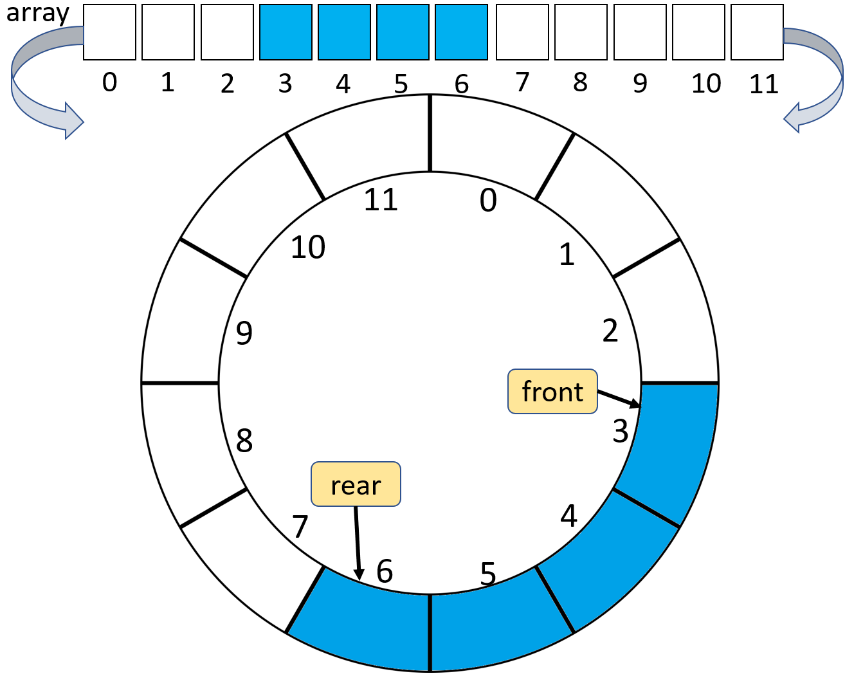
Đồ thị trọng số (weighted graph) là đồ thị có gắn một số (số nguyên hay số thực) cho mỗi cạnh hoặc mỗi cung.

Số nguyên hay số thực cho mỗi cạnh đại điện cho: cự ly, thời gian, chi phí, tốc độ…

Danh sách kề: Đồ thị G = (V,E) V={v1,...,vn} có thể được biểu điễn bằng n danh sách liên kết, trong đó mỗi danh sách liên kết thứ i sẽ biểu điễn các đỉnh kề với đỉnh vi. Mỗi phần tử trong danh sách liên kết thứ i chứa dữ liệu là đỉnh có liên kết với đỉnh vi và trọng số của liên kết đó.



*Hình 2.2a. Danh sách kề*

Hàng đợi vòng tròn (Circular Queue) là dạng nâng cao của hàng đợi tuyến tính bình thường. Hàng đợi vòng sử dụng phương thức lấy phần dư khi thực hiện EnQueue và DeQueue giúp tối ưu không gian bộ nhớ. Cụ thể việc EnQueue và DeQueue sẽ liên tục đến chỉ số MaxLength-1 rồi quay trở lại vị trí đầu tiên-chỉ số 0.

*Hình 2.2b Circular Queue*

# TỔ CHỨC CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

## Phát biểu bài toán

Mô tả đầu vào (Input) và đầu ra (Output)

Input: Tệp chứa ma trận trọng số của đồ thị, dữ liệu có thể tạo ngẫu nhiên hoặc nhập từ bàn phím. Dòng đầu của tệp chứa số đỉnh đồ thị, các dòng còn lại là của ma trận trọng số.

Output: Xuất ra màn hình kết quả của 2 truy vấn: tìm đường đi ngắn nhất giữa hai đỉnh trên đồ thị và tìm các đỉnh có đường đi ngắn nhất đến một đỉnh cố định không lớn hơn một giá trị cụ thể.

## Cấu trúc dữ liệu

Sử dụng danh sách kề để lưu đồ thị đọc từ ma trận trọng số. Đối với thuật toán Ford-Bellman, sử dụng hàng đợi vòng tròn (Circular Queue) trong quá trình tính toán.

## Thuật toán

Trình bày các thuật toán và phân tích độ phức tạp của các thuật toán.

**a. Thuật toán Ford-Bellman:**

**Bước 1**

* Khởi tạo hàng đợi vòng (sau đây gọi là hàng đợi) với 1 phần tử là S: Queue[0]=S. Hàng đợi này có chỉ số ban đầu front= rear=0.
* Đánh dấu S có trong hàng đợi: InQueue[S]=1;
* Gán nhãn d[S]=0, d[v]= +∞ và InQueue[v]=0 ∀v ∈ V, v≠S.
* Khởi tạo mảng kiểm tra số bước tối ưu nhãn của từng đỉnh v: check[v]=0 ∀v ∈ V.

**Bước 2**

* Lấy phần tử đầu hàng đợi (DeQueue) u=Queue[front]. Xoá đánh dấu u có trong hàng đợi InQueue[u]=0.
* Duyệt danh sách liên kết đơn của đỉnh u: Với mỗi node v thuộc danh sách liên kết của đỉnh u mang key là vertex (tên đỉnh) và cost (chi phí ban đầu từ u đến v biểu diễn bởi ma trận trọng số), ta thực hiện phép so sánh d[v]>d[u]+cost ? Nếu điều kiện đúng, chuyển sang bước 3.

**Bước 3**

* Cập nhật lại nhãn d[v]= d[u]+cost và lưu vết trace[v]=u.
* Kiểm tra check[v] có bằng số đỉnh (n) hay không.
* Nếu check[v] == n, tức là đã sửa nhãn d[v] n lần, mà trong khi đó tối đa chỉ có thể sửa n-1 lần. Vậy là đồ thị có chu trình âm, không tồn tại đường đi ngắn nhất, trả về 0. Nếu không, tăng check[v] thêm 1.
* Nếu v chưa có trong hàng đợi (InQueue[v]==0): thêm v vào hàng đợi rear=(rear+1)%n; Queue[rear]=v; InQueue[v]=1;

**Bước 4** Kiểm tra hàng đợi có rỗng hay không, nếu hàng đợi rỗng (front==(rear+1)%n) thì có nghĩa là không còn nhãn nào dùng để sửa các nhãn khác, thuật toán kết thúc thành công, return 1, lúc này các giá trị d[v] sẽ là đáp án cần tìm. Nếu không, quay lại bước 2.

**Độ phức tạp tính toán**: Tại thao tác DeQueue thì ta duyệt lần lượt các phần tử trong hàng đợi vòng có tối đa có n phần tử, mà mỗi đỉnh trong đồ thị có tối đa n-1 lần được lấy ra để thực hiện sửa nhãn, thao tác sửa nhãn duyệt danh sách liên kết đơn của đỉnh có vừa lấy ra, giả sử một danh sách liên kết đơn có số liên kết lớn nhất là m. Vậy độ phức tạp tính toán là O(n.(n-1).m).

**b. Thuật toán Dijkstra:**

**Bước 1**

* Khởi tạo d[S]=0, d[v]= +∞ ∀v ∈ V, v≠S.
* Đánh dấu nhãn tự do Free[v]=1 ∀v ∈ V.

**Bước 2**

* Tìm nhãn chưa được cố định có giá trị nhỏ nhất (Free[u]==1 và d[u] min).
* Nếu không tìm được hoặc tìm thấy nhãn của đỉnh đích đến thì chuyển đến bước 3.
* Cố định nhãn vừa tìm được Free[u]=0.
* Duyệt danh sách liên kết đơn của đỉnh u: Với mỗi node v thuộc danh sách liên kết của đỉnh u mang key là vertex (tên đỉnh) và cost (chi phí ban đầu từ u đến v biểu diễn bởi ma trận trọng số), ta kiểm tra nếu nhãn d[v] tự do và phép so sánh d[v]>d[u]+cost đúng thì cập nhật lại nhãn d[v]= d[u]+cost và lưu vết trace[v]=u.
* Thực hiện lại từ đầu bước 2.

**Bước 3** Trả về giá trị cần tìm và kết thúc thuật toán.

**Độ phức tạp tính toán:** Mỗi bước lặp sẽ chọn ra đỉnh có nhãn nhỏ nhất trong n đỉnh để cố định, thực hiện n lần so sánh như vậy, thao tác sửa nhãn duyệt danh sách liên kết đơn của đỉnh có nhãn dùng để sửa nhãn, giả sử một danh sách liên kết đơn có số liên kết lớn nhất là m. Vậy độ phức tạp tính toán là O(n.m).

# CHƯƠNG TRÌNH VÀ KẾT QUẢ

## Tổ chức chương trình

Đầu tiên là chọn dữ liệu để tải lên chương trình. Có 3 hướng:

* Tạo ngẫu nhiên ma trận trọng số các số nguyên.
* Tạo ma trận trọng số với dữ liệu nhập từ bàn phím.
* Tải lên ma trận đã có sẵn.

Sau đó là các truy vấn:

* Tìm đường đi ngắn nhất giữa 2 đỉnh trên đồ thị.
* Tìm tất cả các đỉnh cách một đỉnh nào đó trong một phạm vi yêu cầu.

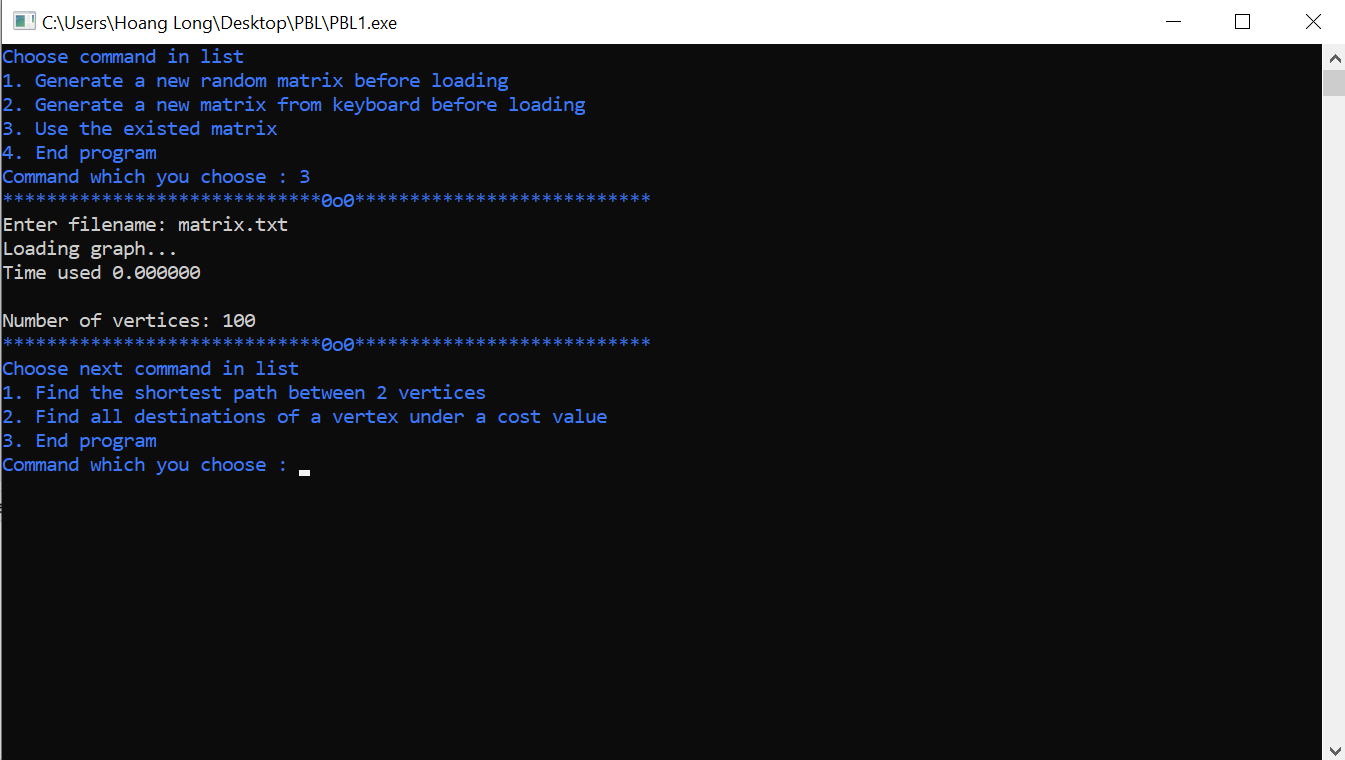
## Ngôn ngữ cài đặt: C

## Kết quả

### Giao điện chính của chương trình

Text

Description automatically generated



Text

Description automatically generated

Text

Description automatically generated

### Kết quả thực thi của chương trình

Mô tả kết quả thực hiện chương trình.

Bước 1, thì có 3 lựa chọn là tạo một ma trận ngẫu nhiên mới, hoặc ma trận với dữ liệu nhập từ bàn phím, hoặc là tải lên ma trận sẵn có, ví dụ trên thì chọn yêu cầu 3 : tải lên ma trận đã có với 100 đỉnh.

Bước 2 là thực hiện các truy vấn:

* Ban đầu là tìm đường đi có chi phí nhỏ nhất, ta thử với đường đi từ 1 đến 5 thì có chi phí là -9.00 và đường đi là 1->5. Lần thử tiếp theo là đường đi từ 5 đến 1 nhỏ nhất có chi phí là 308.00 và đường đi là 5->83->57->1
* Truy vấn tiếp theo là tìm các đỉnh có đường đi ngắn nhất với chi phí ≤50 và tìm được các đỉnh xếp tăng dần theo chi phí là 5 (-9.00), 83(-3.00), 86(30.00). Thử lại với truy vấn tìm đường đi ngắn nhất từ 1 đến 86 thì kết quả đúng là 30.00 với đường đi là 1->5->86.

### Nhận xét đánh giá

Dưới đây là so sánh thời gian thực hiện thuật toán Dijkstra và Ford-Bellman trên đồ thị 5000 đỉnh và không có trọng số âm.

Graphical user interface, text, application, chat or text message

Description automatically generated

*Hình 4.3.3a* *Test Dijkstra*

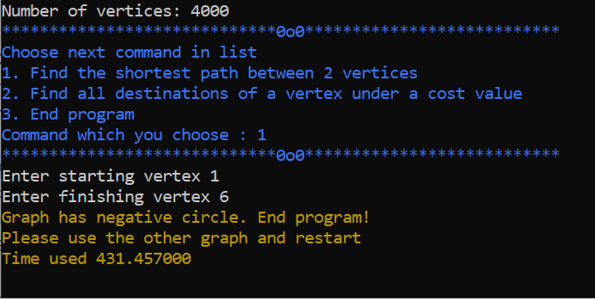
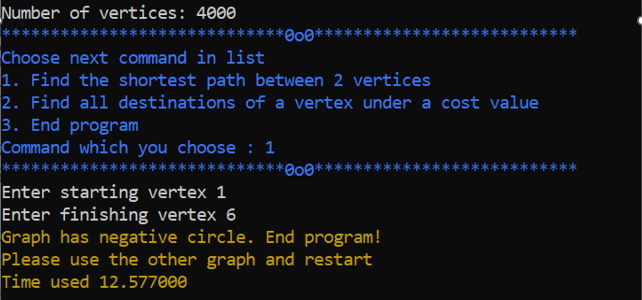
Graphical user interface, text, chat or text message

Description automatically generated

*Hình 4.3.3b* *Test Ford-Bellman*

Rõ ràng thuật toán Dijkstra hoạt động tốt hơn Ford-Bellman với trường hợp đồ thị không có trọng số âm.

Tiếp sau đây là so sánh tốc độ thực thi khi đồ thị có chu trình âm của thuật toán Ford-Bellman thuần (hình 4.3.3c) và Ford-Bellman kết hợp hàng đợi vòng tròn (Circular Queue) (hình 4.3.3d).

*Hình 4.3.3c Hình 4.3.3d*

Dễ thấy Ford-Bellman dùng Circular Queue sẽ nhanh hơn Ford-Bellman thuần rất nhiều, thể hiện rõ nhất khi phát hiện chu trình âm, bởi Ford-Bellman thuần sẽ phải chạy hết n-1 vòng lặp Ford-Bellman và chạy thêm 1 vòng nữa mới phát hiện chu trình âm. Còn khi dùng Circular Queue để lưu các nhãn vừa được sửa thì nếu phát hiện nhãn nào sửa quá n-1 lần thì phát hiện chu trình âm ngay.

# KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

## Kết luận

Với bài toán đường đi ngắn nhất không có trọng số âm thì hiện nay thuật toán Dijkstra vẫn được ứng dụng nhiều nhất. Với độ phức tạp O(n2) thì thuật toán Dijkstra nhanh hơn nhiều so với thuật toán Ford-Bellman, rõ rệt ở những đồ thị lớn.

## Hướng phát triển

Với đồ thị có số cạnh không nhiều có thể sử dụng thuật toán Dijkstra với cấu trúc dữ liệu HEAP thì độ phức tạp chỉ là O(nlogn).

Với ứng dụng tìm các điểm lân cận thì trong trường hợp tổng quát sẽ dùng thuật toán Ford-Bellman như đã trình bày, nếu đồ thị không có trọng số âm thì áp dụng thuật toán Dijkstra với tất cả các đỉnh sẽ nhanh hơn vì độ phức tạp sẽ là O(n2)

TÀI LIỆU THAM KHẢO

1. Lê Minh Hoàng. *Lí thuyết đồ thị*, trang 58-62
2. Thuật toán Selection Sort, https://www.geeksforgeeks.org/selection-sort/
3. Và Quick Sort, https://www.geeksforgeeks.org/quick-sort/
4. Phan Công Minh và cộng sự. *Một số vấn đề đáng chú ý trong môn tin học,* trang 82, 83: *Thuật toán Ford Bellman kết hợp Queue vòng.*

PHỤ LỤC

|  |
| --- |
| #pragma once  “constant.h”  #define MAXC 1e108 //no direct path  #define MAX 10000 //max number of vertices  #define LINK "matrix.txt"  #define ERR "Warning! Please enter valid data\n\n"  #define LINE "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*0o0\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"  typedef struct node  {  int vertex;  double cost;  struct node \*next;  } Node;  typedef Node \*pNode;  typedef struct  {  int numVertices;  pNode \*lists;  } Graph;  typedef Graph \*pGraph; |
| #pragma once  “AllMethod.h”  #include <stdio.h>  #include "constant.h"  void Dijkstra(double \*ShD, int trace[], pGraph graph, int S, int F);  //int Ford\_Bellman(double d[], int trace[], pGraph graph, int S);  int Ford\_Bellman2(double d[], int trace[], pGraph graph, int S);  void initMat(FILE \*f);  void EnterMatrix(FILE \*f);  pGraph MatrixToList(FILE \*f, int \*sign);  int checkinput1(char \*s); |
| #include <stdio.h>  “randomMatrix.cpp”  #include <conio.h>  #include <time.h>  #include <stdlib.h>  #include "constant.h"  //#include <float.h>  void initMat(FILE \*f)  {  char check;  printf("Are you sure to generate a new matrix?\nThe exist one will be deleted!\nPress 'y'(if yes) or 'o'(if no)\n");  do  {  fflush(stdin);  check=getch();  } while (check != 'y' && check != 'o');  if (check=='o') return;  f=fopen(LINK,"w");  srand(time(NULL));  int n,tp;  char str[101];  char temp;  do  {  do  {  fflush(stdin);  printf("Enter number of vertices: (1<n<10000) ");  if (scanf("%4[0-9]s",str)==0) printf(ERR);  else  {  scanf("%c",&temp);  if (temp!='\n') printf(ERR); else break;  }  } while (1);  n=atoi(str);  if (n<2) printf(ERR); else break;  } while (1);  fprintf(f,"%d\n",n);  printf("Gererating Matrix...\n");  clock\_t start,end;  start=clock();  for (int i=1; i<=n; i++)  {  for (int j=1; j<=n; j++)  {  if (i==j) fprintf(f,"%d ",0);  else  {  tp=1+rand()%10000;  if (tp>5000) fprintf(f,"%.0e ",MAXC);  else fprintf(f,"%d ",tp);  }  }  fprintf(f,"\n");  }  end=clock();  double time\_use = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("Time used %lf\n\n", time\_use);  fclose(f);  } |
| #include <stdio.h>  “readFromKeyboard.cpp”  #include <conio.h>  #include "constant.h"  #include <stdlib.h>  int checkinput1(char \*s)  {  int t=0,k[2]={1,1};  while (s[t]!='\0')  {  if (s[t]=='-') k[0]--;  if (s[t]=='.') k[1]--;  if (k[0]<0 || k[1]<0) return 0;  t++;  }  if (t==0) return 0;  if (t==1 && k[0]\*k[1]==0) return 0;  return 1;  }  void EnterMatrix(FILE \*f)  {  char check;  printf("Are you sure to generate a new matrix?\nThe exist one will be deleted!\nPress 'y'(if yes) or 'o'(if no)\n");  do  {  fflush(stdin);  check=getch();  } while (check != 'y' && check != 'o');  if (check=='o') return;  f=fopen(LINK,"w");  int n;  char temp;  char str[101];  double cost;  do  {  do  {  fflush(stdin);  printf("Enter number of vertices: (1<n<10000) ");  if (scanf("%4[0-9]s",str)==0) printf(ERR);  else  {  scanf("%c",&temp);  if (temp!='\n') printf(ERR); else break;  }  } while (1);  n=atoi(str);  if (n<2) printf(ERR); else break;  } while (1);    fprintf(f,"%d\n",n);    for (int i=1; i<=n; i++)  {  for (int j=1; j<=n; j++)  {  if (i==j) fprintf(f,"%d ",0);  else  {  printf("Is there a direct path from %d to %d? Press 'y'(yes) or 'o'(no)",i,j);  fflush(stdin);  do  {  check=getch();  } while (check!='y' && check!='o');    if (check=='y')  {  printf("\r");  printf(" ");  printf("\r");  do  {  printf("Enter direct cost from %d to %d: ",i,j);  fflush(stdin);  if (scanf("%100[-.0-9]s",str)==0) printf(ERR);  else  {  scanf("%c",&temp);  if (temp!='\n' || checkinput1(str)==0) printf(ERR);  else break;  }  } while (1);  cost = atof(str);  fprintf(f,"%.2lf ",cost);  } else  {  printf("\r");  printf(" ");  printf("\r");  printf("No direct path from %d to %d\n",i,j);  fprintf(f,"%.0e ",MAXC);  }  }  }  fprintf(f,"\n");  }  fclose(f);  } |
| “adjacency\_list.cpp”  #include <stdio.h>  #include <stdlib.h>  #include "constant.h"  pNode CreateNode(int d, double val)  {  pNode newNode;  newNode = (pNode)malloc(sizeof(Node));  newNode->vertex = d;  newNode->cost = val;  newNode->next = NULL;  return newNode;  }  pGraph CreateGraph(int vertices)  {  pGraph newGraph;  newGraph = (pGraph)malloc(sizeof(Graph));  newGraph->numVertices = vertices;  newGraph->lists = (pNode \*)malloc((vertices+1) \* sizeof(pNode));  int i;  pNode temp;  for (i = 1; i <= vertices; i++)  {  temp = CreateNode(i, 0);  newGraph->lists[i] = temp;  }    return newGraph;  }  void AddEdge(pGraph graph, int s, int d, double val)  {    pNode newNode = CreateNode(d, val);    newNode->next = graph->lists[s]->next;    graph->lists[s]->next = newNode;    }  pGraph MatrixToList(FILE \*f, int \*sign)  {  int n,tp;  f=fopen(LINK,"r");  if (fscanf(f,"%d",&n)==EOF || n<=0)  {  printf("Data file is not valid. End program!");  exit(0);  }  pGraph graph = CreateGraph(n);  double val;  pNode temp;  for (int i = 1; i <= n; i++)  {  for (int j=1;j<=n;j++)  {  if (i==j)  {  if (fscanf(f,"%lf", &val)==0 || val!=0)  {  printf("Data file is not valid. End program!");  system("pause");  exit(0);  }  continue;  }  if (fscanf(f,"%lf", &val)==0)  {  printf("Data file is not valid. End program!");  system("pause");  exit(0);  }  if(val<0) \*sign=1;  if (val<MAXC) AddEdge(graph,i,j,val);  }  }  char last[256];  fscanf(f,"%[ \n]s",last);  if (!feof(f))  {  printf("Data file is not valid. End program!");  system("pause");  exit(0);  }  fclose(f);  return graph;  } |
| #include <stdio.h>  “dijkstra.cpp”  #include "constant.h"  static double d[MAX];//distance  static int Free[MAX];  void init\_d(int trace[], int n, pGraph graph, int S)  {  for (int i=1; i<=n; i++) d[i]=MAXC;  d[S]=0;  for (int i=1; i<=n; i++) Free[i]=1;//all vertices is free  }  void Dijkstra(double \*ShD, int trace[], pGraph graph, int S, int F)  {  int n=graph->numVertices;  int u,v;  double min;  pNode temp;  init\_d(trace,n,graph,S);  do  {  u=0; min=MAXC;  for (v=1; v<=n; ++v)  if (Free[v] && d[v]<min)  {  min=d[v];  u=v;  }  if (u==0 || u==F) break;  Free[u]=0;//fix vertex  temp=graph->lists[u]->next;  while (temp!=NULL)  {  v=temp->vertex;  if (Free[v] && (d[v]>d[u]+temp->cost))  {  d[v]=d[u]+temp->cost;  trace[v]=u;  }  temp=temp->next;  }  }  while (1);  \*ShD=d[F];  } |
| #include <stdio.h>  “ford\_bellman.cpp”  #include "constant.h"  /\*  void init\_d(double d[], int trace[], int n, pGraph graph, int S)  {  pNode temp=graph->lists[S]->next;  for (int i=1; i<=n; i++) d[i]=MAXC;  d[S]=0;  while (temp!=NULL)  {  d[temp->vertex]=temp->cost;//distance firstly is cost  temp=temp->next;  }  for (int i=1; i<=n; i++)  {  trace[i]=S;//initial path to other vertices is from starting vertex  }  }  int Ford\_Bellman(double d[], int trace[], pGraph graph, int S)  {  int n=graph->numVertices;  int CountLoop = 0;  int i, j, Stop;  pNode temp;  init\_d(d,trace,n,graph,S);  do {  Stop = 1;  for (i = 1; i <= n; i++)  {  temp=graph->lists[i]->next;  while (temp!=NULL)  {  j=temp->vertex;  if (d[j] > (d[i] + temp->cost))  {  d[j] = d[i] + temp->cost;  trace[j] = i;  Stop = 0;  }  temp=temp->next;  }  }  CountLoop++;  } while (!Stop && CountLoop < n - 2);  //check for negative circle  for (i = 1; i <= n; i++)  {  temp=graph->lists[i]->next;  while (temp!=NULL)  {  j=temp->vertex;  if (d[j] > (d[i] + temp->cost)) return 0;  temp=temp->next;  }  }  return 1;  }  \*/  int Ford\_Bellman2(double d[], int trace[], pGraph graph, int S)  {  int n=graph->numVertices;  int u,v,front=0,rear=0,count=0;  int Queue[n];  int InQueue[n+1]={0};//check if u exists in Queue  int check[n+1]={0};//check how many times a path is upgraded  pNode temp;  for (int i=1; i<=n; i++) d[i]=MAXC;  d[S]=0;  Queue[0]=S;//Fristly add S to Queue  InQueue[S]=1;  do  {  u=Queue[front];//DeQueue  InQueue[u]=0;  front=(front+1)%n;  temp=graph->lists[u]->next;  while (temp!=NULL)  {  v=temp->vertex;  if (d[v] > (d[u] + temp->cost))  {  d[v] = d[u] + temp->cost;  trace[v] = u;  if (check[v]==n) return 0;//check for negative circle  else ++check[v];  if (InQueue[v]==0)  {  //EnQueue  rear=(rear+1)%n;  Queue[rear]=v;  InQueue[v]=1;  }  }  temp=temp->next;  }  } while (front != (rear+1)%n);  return 1;  } |
| //#include <stdio.h>  “main.cpp”  #include <time.h>  #include <conio.h>  //#include <unistd.h>  #include "AllMethod.h"  #include "constant.h"  #include <Windows.h>  /\* run this program using the console pauser or add your own getch, system("pause") or input loop \*/  int trace[MAX];  int n,S,F,command,tp,sign=0,stop=0,endpr=0,rewrite=0;  double dA[MAX];  FILE \*f;  pGraph graph;  clock\_t start,end;  double time\_use;  void out(double cost);  void load();  void FindPath();  void FindLoc();  void intro();  void thanks();  char checkend();  void swap(int\* a, int\* b);  int partition(int Map[], int l, int h);  void quickSortIterative(int Map[], int l, int h);  void selectionSort(int Map[], int n);  int main(int argc, char\*\* argv)  {  intro();  system("cls");  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 9);  printf("Choose command in list\n");  printf("1. Generate a new random matrix before loading\n");  printf("2. Generate a new matrix from keyboard before loading\n");  printf("3. Use the exist matrix\n");  printf("4. End program\n");  do  {  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 9);  printf("Command which you choose : ");  fflush(stdin);  do  {  command=getch();  } while (command<'1' || command>'4');  printf("%c\n",command);  printf(LINE);  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 7);  switch (command)  {  case '1': initMat(f); load(); stop=1; break;  case '2': EnterMatrix(f); load(); stop=1; break;  case '3': load(); stop=1; break;  case '4': if (checkend()=='y') thanks();  }  } while(stop==0);  stop=0;  printf("Number of vertices: %d\n",n);  do  {  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 9);  printf(LINE);  printf("Choose next command in list\n");  printf("1. Find the shortest path between 2 vertices\n");  printf("2. Find all destinations of a vertex under a cost value\n");  printf("3. End program\n");  fflush(stdin);  printf("Command which you choose : ");  do  {  command=getch();  } while (command<'1' || command>'3');  printf("%c\n",command);  printf(LINE);  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 7);  switch (command)  {  case '1': FindPath(); break;  case '2': FindLoc(); break;  case '3': if (checkend()=='y') thanks();  }  } while(stop==0);  return 0;  }  void out(double cost)  {  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 6);  if (cost==MAXC) printf("Path from %d to %d not found",S,F);  else  {  printf("Distance from %d to %d is %.2lf\n",S,F,cost);  while (F!=S)  {  printf("%d <- ",F);  F=trace[F];  }  printf("%d",S);  }  }  void load()  {  printf("Loading graph...\n");  start=clock();  graph = MatrixToList(f,&sign);  end=clock();  time\_use = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("Time used %lf\n\n", time\_use);  n=graph->numVertices;  }  void FindPath()  {  char temp;  do  {  fflush(stdin);  printf("Enter starting vertex ");  if (scanf("%d%c",&S,&temp)!=2 || temp!='\n' || S<1 || S>n) printf(ERR);  else break;  } while (1);  do  {  fflush(stdin);  printf("Enter finishing vertex ");  if (scanf("%d%c",&F,&temp)!=2 || temp!='\n' || F<1 || F>n) printf(ERR);  else break;  } while (1);  if (sign==0)  {  start=clock();  Dijkstra(&dA[F],trace,graph,S,F);  out(dA[F]);  end=clock();  time\_use = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("\nTime used %lf\n", time\_use);  }  else  {  start=clock();  if (Ford\_Bellman2(dA,trace,graph,S)==0)  {  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 6);  printf("Graph has negative circle. End program!\nPlease use the other graph and restart");  end=clock();  time\_use = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("\nTime used %lf\n", time\_use);  getch();  thanks();    }  else  {  out(dA[F]);  }  end=clock();  time\_use = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("\nTime used %lf\n", time\_use);  }  }  void FindLoc()  {  char temp;  char str[101];  double cost;  do  {  fflush(stdin);  printf("Enter starting vertex ");  if (scanf("%d%c",&S,&temp)!=2 || temp!='\n' || S<1 || S>n) printf(ERR);  else break;  } while (1);    do  {  fflush(stdin);  printf("Enter cost value: ");  if (scanf("%100[-.0-9]s",str)==0) printf(ERR);  else  {  scanf("%c",&temp);  if (temp!='\n' || checkinput1(str)==0) printf(ERR);  else break;  }  } while (1);  cost=atof(str);  start=clock();  if (Ford\_Bellman2(dA,trace,graph,S)==0)  {    SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), 6);  printf("Graph has negative circle. End program!\nPlease use another graph and restart");  end=clock();  time\_use = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("\nTime used %lf\n", time\_use);  getch();  thanks();  }  else  {  int Map[n+1];  int countMap=0;  for (int i=1; i<=n; i++)  {  if (i!=S && dA[i]<=cost)  {  countMap++;  Map[countMap]=i;  }  }  printf("Number of destinations found: %d\n",countMap);  if (countMap<1000) selectionSort(Map,countMap);  else quickSortIterative(Map,1,countMap);  printf("|\tDestination\t|\t Cost \t|");  for (int i=1; i<=countMap; i++)  {  printf("\n");  printf("|\t%11d\t|\t%11.2lf\t|",Map[i],dA[Map[i]]);  }  }  end=clock();  time\_use = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("\nTime used %lf\n", time\_use);  }  void intro()  {  int cc=1,icolor=1;  f=fopen("intro.txt","r");  char s[1000];  while (!feof(f))  {  fgets(s,1000,f);  if (cc%4==0)  {  icolor++;  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), icolor);  }  cc++;  printf("%s",s);  }  fclose(f);  Sleep(1000);  printf("GETTING STARTED ...\n");  /\* for (int i=1; i<=99; i++) printf(" ");  printf("%c\r",178);  for (int i=0; i<=9; i++)  {  for (int j=1; j<=10; j++)  {  printf("%c",220);  if (10\*i+j==100) printf("100%%");  Sleep(1.5\*(i\*4+j));  }    }  Sleep(1000);  \*/  getch();  }  void thanks()  {  system("cls");  int cc=1,icolor=1;  f=fopen("thanks.txt","r");  char s[1000];  while (!feof(f))  {  fgets(s,1000,f);  if (cc%4==0)  {  icolor++;  SetConsoleTextAttribute(GetStdHandle( STD\_OUTPUT\_HANDLE ), icolor);  }  cc++;  printf("%s",s);  }  fclose(f);  system("pause");  exit(0);  }  char checkend()  {  char check;  printf("Sure to end ? Press 'y'(yes) or 'o'(no)");  do  {  check=getch();  }while (check!='y'&&check!='o');  printf("\n");  return check;  }  void swap(int\* a, int\* b)  {  int t = \*a;  \*a = \*b;  \*b = t;  }    void selectionSort(int Map[], int n)  {  int i, j, min\_idx;  for (i = 1; i < n; i++)  {  min\_idx = i;  for (j = i+1; j <= n; j++) if (dA[Map[j]] < dA[Map[min\_idx]]) min\_idx = j;  swap(&Map[i], &Map[min\_idx]);  }    }  int partition(int Map[], int l, int h)  {  int x = dA[Map[h]];  int i = (l - 1);    for (int j = l; j <= h - 1; j++) {  if (dA[Map[j]] <= x) {  i++;  swap(&Map[i], &Map[j]);  }  }  swap(&Map[i+1], &Map[h]);  return (i + 1);  }    void quickSortIterative(int Map[], int l, int h)  {  // Create an auxiliary stack  int stack[h - l + 1];    // initialize top of stack  int top = -1;    // push initial values of l and h to stack  stack[++top] = l;  stack[++top] = h;    // Keep popping from stack while is not empty  while (top >= 0) {  // Pop h and l  h = stack[top--];  l = stack[top--];    // Set pivot element at its correct position  // in sorted array  int p = partition(Map, l, h);    // If there are elements on left side of pivot,  // then push left side to stack  if (p - 1 > l) {  stack[++top] = l;  stack[++top] = p - 1;  }    // If there are elements on right side of pivot,  // then push right side to stack  if (p + 1 < h) {  stack[++top] = p + 1;  stack[++top] = h;  }  }  } |