BÁO CÁO ĐỒ ÁN CUỐI KÌ

Môn: Cấu trúc dữ liệu và giải thuật

I. Các thuật toán sử dụng:

+ **Tìm đường đi ngắn nhất: Dijkstra** - thuật toán mang tên của nhà khoa học máy tính người Hà Lan Edsger W. Dijkstra, là một thuật toán giải quyết bài toán đường đi ngắn nhất trong một đồ thị có hướng không có cạnh trọng số âm.

- Thuật toán được xây dựng trên cơ sở gán cho mỗi đỉnh các nhãn tạm thời. Nhãn tạm thời của các đỉnh cho biết cận trên của chiều dài đường đi ngắn nhất từ s đến đỉnh đó. Nhãn của các đỉnh sẽ biến đổi trong các bước lặp, mà ở mỗi bước lặp sẽ có một nhãn tạm thời trở thành chính thức. Nếu nhãn của một đỉnh nào đó trở thành chính thức thì đó cũng chính là chiều dài ngắn nhất của đường đi từ s đến đỉnh đó.
- Thuật toán:

- Khởi tạo các mảng n phần tử: label, length, prev. Gán label[k] = 1, length[k] = -1 (inf), prev[k] = -1 với k chạy từ 0 -> n – 1. Gán length[first] = 0
- Chọn đỉnh v trong mảng sao cho length[k] là nhỏ nhất. Sau đó gán label[k] = 0 (Đã đánh dấu)
- Tạo vòng lặp với biến chạy k, xét nếu label[k] = 1 (Chưa đánh dấu) và có đường đi từ v -> k: Nếu length[k] > length[v] + trọng số từ v -> k hoặc length[k] = inf, có nghĩa là nếu ta tìm được 1 đường từ v -> k là nhỏ nhất, hoặc là chưa tìm được đường nào ngắn nhất (inf) => Gán length[k] = length[v] + trọng số v -> k, prev[k] = v (Tạo vết chân đỉnh trước đó).
- Nếu label[last] = 0 (Đã đánh dấu đỉnh đến), kết thúc vòng lặp. Nếu không thì quay lại bước 2.

- Thuật toán Dijkstra bình thường sẽ có độ phức tạp là $O(n^2 + m)$ với n là số đỉnh và m là số cạnh của đồ thị

+ **Tìm k đường đi tốt nhất: Yen** là thuật toán tìm kiếm K đường đi ngắn nhất không lặp lại của đồ thị không tồn tại cạnh có trọng số âm. Thuật toán được phát mình bởi nhà khoa học Jin Y.Yen vào năm 1971.

- Thuật toán lấy nền tảng từ đường đi ngắn nhất, từ đó tìm kiếm K – 1 đường đi còn lại với độ chênh lệch thấp nhất so với đường đi ngắn nhất được tìm thấy.

- Thuật toán sử dụng:
- Tập hợp A lưu trữ kết quả K đường đi ngắn nhất
- Tập hợp B lưu trữ những kết quả có khả năng trở thành 1 trong những đường đi ngắn nhất thuộc tập A
- Để tìm đường đi thứ Ak, thuật toán giả định đã tìm được K – 1 đường đi trước đó. Quá trình được thực hiện gồm 2 giai đoạn: giai đoạn 1, tìm kiếm tất cả cách đi có thể; giai đoạn 2, lựa chọn kết quả tốt nhất trở thành Ak. Giai đoạn một của quá trình tìm kiếm Ak được chia thành 3 bước cụ thể như sau:
- Chọn Rk-I từ Ak trong đó Rk-I là đường đi từ s đến i trong đường đi A(k-1). Nếu Rk-I được tìm thấy, cạnh i->i+1 sẽ được đánh dấu không thể đi qua.
- Từ i đã được tìm thấy, thực hiện thuật toán đường đi ngắn nhất để xác định Sk-I (là cách di chuyển từ i đến d) với cạnh i->i+1 đã được đánh dấu xóa khỏi đồ thị để đảm bảo rằng đường đi Ak-I = Rk-I + Sk-I chưa từng tồn tại trước đó trong A và B).
- Nếu tìm thấy Sk-I lưu kết quả Rk-I + Sk-I vào B. Trả lại giá trị ban đầu của i>i+1, quay lại bước 1. Đến khi i = K -1 thì dừng lại.
- Giai đoạn hai của quá trình tìm kiếm Ak. Lựa chọn kết quả tốt nhất hiện có của B và chuyển kết quả trở thành Ak, nếu K đã đạt được K ban đầu thì thuật toán kết thúc. Hoặc B rỗng thì thuật toán kết thúc do không thể tìm đủ K kết quả như mong muốn.
- Mã giả thực hiện thuật toán Yen:

```
function YenKSP(Graph, source, sink, K):
  // Tính toán đường đi ngắn nhất đầu tiên nhờ vào thuật toán Dijkstra. Từ node bắt đầu đến node đích
  A[0] = Dijkstra(Graph, source, sink);
  // Khởi tạo tập hợp B lưu trữ những đường đi có khả năng trở thành K đường đi ngắn nhất
  B = [];

  for k from 1 to K:
    // Giả định node-I thỏa mãn Ak-I = Rk-I + Sk-I, với node-I thuộc Ak-1
    for i from 0 to size(A[k − 1]) − 1:

      // Xác định node-I trung gian là node bắt đầu để tính Sk-i
      spurNode = A[k-1].node(i);
      // Xác định Rk-I là đường đi từ node-1 đến node-I bằng cách đi Ak-1
      rootPath = A[k-1].nodes(0, i);

      for each path p in A:
```

```
            if rootPath == p.nodes(0, i):
               // Xác định cạnh i->i+1 trong cách đi Ak-1 là cờ cấm, từ đó cho phép xây dựng Ak = Rk-1 + Sk-1 khác với Ak-1
                remove p.edge(i, i + 1) from Graph;

         for each node rootPathNode in rootPath except spurNode:
            remove rootPathNode from Graph;

         // Tính toán Sk-I với điểm bắt đầu là I và điểm kết thúc là node-n bằng thuật toán Dijkstra
         spurPath = Dijkstra(Graph, spurNode, sink);

         // Kết quả đường đi với node-I trung gian được tính từ kết quả Rk-I + Sk-i
         totalPath = rootPath + spurPath;
         // Lưu kết quả đường đi thành một kết quả có khả năng trở thành một đường đi ngắng nhất
         B.append(totalPath);

         // Trả về các cạnh đã bị đánh dấu cấm
         restore edges to Graph;
         restore nodes in rootPath to Graph;

      if B is empty:
         // Nếu tập hợp B là rỗng, xác định thuật toán kết thúc, không thể tìm đủ k đường đi
         break;
      // Sắp xếp tập hợp B theo giá trị độ dài những đường đi khả dĩ tăng dần
      B.sort();
      // Lưu giá trị của đường đi thứ k bằng đường đi ngắn nhất có trong B
      A[k] = B[0];
      // Loại kết quả Ak ra khỏi B
      B.pop();

   return A;
```

- Độ phức tạp của thuật toán Yen là O(kn (m + n log n)) với k là số đường đi cần tìm, m là số cạnh và n là số đỉnh của đồ thị.

II. <u>Cấu trúc chương trình:</u>

- Tổ chức lưu dữ liệu của file input:

- Dùng mảng Cities có kiểu vector<string> để lưu danh sách tên các thành phố.
  => Ta sẽ xử lí đồ thị với mỗi đỉnh là chỉ số của mỗi thành phố, do đó sẽ nhanh chóng và đơn giản hơn trong việc so sánh

- Tạo 2 đồ thị ứng với mỗi loại đồ thị ta sẽ có trọng số khác nhau: “1” (đồ thị CostGraph) trọng số giữa 2 đỉnh sẽ là chi phí của chuyến bay và “2’ (đồ thị DurationGraph) trọng số giữa 2 đỉnh sẽ là thời gian bay.
- Tổ chức chương trình xử lí: 2 quá trình
- Đầu tiên tìm đường đi ngắn nhất từ thành phố xuất phát (s) đến thành phố đích (d) bằng thuật toán Dijkstra.

```
void DijkstraShortestPathAlg::improve2vertex(BaseVertex* cur_vertex_pt, bool is_source2sink)
{
	Lấy các đỉnh lân cận
	set<BaseVertex*>* neighbor_vertex_list_pt = new set<BaseVertex*>();

	if (is_source2sink)
	{
		m_pDirectGraph->get_adjacent_vertices(cur_vertex_pt, *neighbor_vertex_list_pt);
	}
	else
	{
		m_pDirectGraph->get_precedent_vertices(cur_vertex_pt, *neighbor_vertex_list_pt);
	}

	Cập nhật lại khoảng cách
	for (set<BaseVertex*>::iterator cur_neighbor_pos = neighbor_vertex_list_pt->begin();
		cur_neighbor_pos != neighbor_vertex_list_pt->end(); ++cur_neighbor_pos)
	{
		Bỏ qua nếu đỉnh đã được truy cập trước đó
		if (m_stDeterminedVertices.find((*cur_neighbor_pos)->getID()) != m_stDeterminedVertices.end())
		{
			continue;
		}

		Tính khoảng cách
		map<BaseVertex*, double>::const_iterator cur_pos = m_mpStartDistanceIndex.find(cur_vertex_pt);
		double distance = cur_pos != m_mpStartDistanceIndex.end() ? cur_pos->second :
Graph::DISCONNECT;

		distance += is_source2sink ? m_pDirectGraph->get_edge_weight(cur_vertex_pt, *cur_neighbor_pos) :
			m_pDirectGraph->get_edge_weight(*cur_neighbor_pos, cur_vertex_pt);

		Cập nhật lại khoảng cách (nếu cần)
		cur_pos = m_mpStartDistanceIndex.find(*cur_neighbor_pos);
		if (cur_pos == m_mpStartDistanceIndex.end() || cur_pos->second > distance)
		{
			m_mpStartDistanceIndex[*cur_neighbor_pos] = distance;
			m_mpPredecessorVertex[*cur_neighbor_pos] = cur_vertex_pt;

			(*cur_neighbor_pos)->Weight(distance);

			multiset<BaseVertex*, WeightLess<BaseVertex> >::const_iterator pos =
m_quCandidateVertices.begin();
			for (; pos != m_quCandidateVertices.end(); ++pos)
			{
				if ((*pos)->getID() == (*cur_neighbor_pos)->getID())
				{
					break;
				}
```

```
                        }
                        if (pos != m_quCandidateVertices.end())
                        {
                                m_quCandidateVertices.erase(pos);
                        }
                        m_quCandidateVertices.insert(*cur_neighbor_pos);
                }
        }
}

void DijkstraShortestPathAlg::determine_shortest_paths(BaseVertex* source, BaseVertex* sink, bool is_source2sink)
{
        Xóa các biến trung gian
        clear();

        Khai báo các biến cục bộ
        BaseVertex* end_vertex = is_source2sink ? sink : source;
        BaseVertex* start_vertex = is_source2sink ? source : sink;
        m_mpStartDistanceIndex[start_vertex] = 0;
        start_vertex->Weight(0);
        m_quCandidateVertices.insert(start_vertex);

        Tìm đường đi ngắn nhất
        while (!m_quCandidateVertices.empty())
        {
                multiset<BaseVertex*, WeightLess<BaseVertex> >::const_iterator pos =
m_quCandidateVertices.begin();

                BaseVertex* cur_vertex_pt = *pos;
                m_quCandidateVertices.erase(pos);

                if (cur_vertex_pt == end_vertex) break;

                if (cur_vertex_pt->getID() == 31)
                {
                        int i = 1;
                        i = 100;
                }
                m_stDeterminedVertices.insert(cur_vertex_pt->getID());

                improve2vertex(cur_vertex_pt, is_source2sink);
        }
}
```

- Sau đó sử dụng thuật toán Yen:

```
BasePath* YenTopKShortestPathsAlg::next()
{
        Chuẩn bị cho việc loại bỏ các đỉnh và cung
        BasePath* cur_path = *(m_quPathCandidates.begin());
        m_quPathCandidates.erase(m_quPathCandidates.begin());
        m_vResultList.push_back(cur_path);

        int count = m_vResultList.size();

        BaseVertex* cur_derivation_pt = m_mpDerivationVertexIndex.find(cur_path)->second;
        vector<BaseVertex*> sub_path_of_derivation_pt;
        cur_path->SubPath(sub_path_of_derivation_pt, cur_derivation_pt);
```

```
int sub_path_length = sub_path_of_derivation_pt.size();

Xóa các đỉnh và cung trong đồ thị
for (int i = 0; i<count - 1; ++i)
{
	BasePath* cur_result_path = m_vResultList.at(i);
	vector<BaseVertex*> cur_result_sub_path_of_derivation_pt;

	if (!cur_result_path->SubPath(cur_result_sub_path_of_derivation_pt, cur_derivation_pt)) continue;

	if (sub_path_length != cur_result_sub_path_of_derivation_pt.size()) continue;

	bool is_equal = true;
	for (int i = 0; i<sub_path_length; ++i)
	{
		if (sub_path_of_derivation_pt.at(i) != cur_result_sub_path_of_derivation_pt.at(i))
		{
			is_equal = false;
			break;
		}
	}
	if (!is_equal) continue;

	BaseVertex* cur_succ_vertex = cur_result_path->GetVertex(sub_path_length + 1);
	m_pGraph->remove_edge(make_pair(cur_derivation_pt->getID(), cur_succ_vertex->getID()));
}

Loại bỏ các đỉnh và cạnh theo kết quả hiện tại
int path_length = cur_path->length();
for (int i = 0; i<path_length - 1; ++i)
{
	m_pGraph->remove_vertex(cur_path->GetVertex(i)->getID());
	m_pGraph->remove_edge(make_pair(
		cur_path->GetVertex(i)->getID(), cur_path->GetVertex(i + 1)->getID()));
}

Tìm đường đi ngắn nhất từ đỉnh nguồn đến đỉnh đích
DijkstraShortestPathAlg reverse_tree(m_pGraph);
reverse_tree.get_shortest_path_flower(m_pTargetVertex);

Khôi phục các đỉnh đã xóa và cập nhật chi phí và xác định kết quả mới
bool is_done = false;
for (int i = path_length - 2; i >= 0 && !is_done; --i)
{
	Lấy đỉnh cần khôi phục
	BaseVertex* cur_recover_vertex = cur_path->GetVertex(i);
	m_pGraph->recover_removed_vertex(cur_recover_vertex->getID());

	Kiểm tra xem có nên tiếp tục không
	if (cur_recover_vertex->getID() == cur_derivation_pt->getID())
	{
		is_done = true;
	}

	Tính toán chi phí
	BasePath* sub_path = reverse_tree.update_cost_forward(cur_recover_vertex);

	Thêm đường đi mới (nếu có thể)
	if (sub_path != NULL)
	{
```

```
                        ++m_nGeneratedPathNum;
                        double cost = 0;
                        reverse_tree.correct_cost_backward(cur_recover_vertex);

                        vector<BaseVertex*> pre_path_list;
                        for (int j = 0; j<path_length; ++j)
                        {
                                BaseVertex* cur_vertex = cur_path->GetVertex(j);
                                if (cur_vertex->getID() == cur_recover_vertex->getID())
                                {
                                        break;
                                }
                                else
                                {
                                        cost += m_pGraph->get_original_edge_weight(cur_path->GetVertex(j),
cur_path->GetVertex(1 + j));
                                        pre_path_list.push_back(cur_vertex);
                                }
                        }
                        for (int j = 0; j<sub_path->length(); ++j)
                        {
                                pre_path_list.push_back(sub_path->GetVertex(j));
                        }

                        Tạo đường đi mới
                        sub_path = new Path(pre_path_list, cost + sub_path->Weight());

                        Thêm nó vào tập đường đi đã có (nếu nó là đường đi mới)
                        if (m_mpDerivationVertexIndex.find(sub_path) == m_mpDerivationVertexIndex.end())
                        {
                                m_quPathCandidates.insert(sub_path);
                                m_mpDerivationVertexIndex[sub_path] = cur_recover_vertex;
                        }
                }

                Khôi phục lại cạnh
                BaseVertex* succ_vertex = cur_path->GetVertex(i + 1);
                m_pGraph->recover_removed_edge(make_pair(cur_recover_vertex->getID(), succ_vertex->getID()));

                Cập nhật lại chi phí (nếu cần thiết)
                double cost_1 = m_pGraph->get_edge_weight(cur_recover_vertex, succ_vertex) +
reverse_tree.get_start_distance_at(succ_vertex);

                if (reverse_tree.get_start_distance_at(cur_recover_vertex) > cost_1)
                {
                        reverse_tree.set_start_distance_at(cur_recover_vertex, cost_1);
                        reverse_tree.set_predecessor_vertex(cur_recover_vertex, succ_vertex);
                        reverse_tree.correct_cost_backward(cur_recover_vertex);
                }
        }
        m_pGraph->recover_removed_edges();
        m_pGraph->recover_removed_vertices();

        return cur_path;
}

void YenTopKShortestPathsAlg::get_shortest_paths(BaseVertex* pSource, BaseVertex* pTarget, int top_k,
vector<BasePath*>& result_list)
{
        m_pSourceVertex = pSource;
```

```
        m_pTargetVertex = pTarget;

        _init();
        int count = 0;
        while (has_next() && count < top_k)
        {
                next();
                ++count;
        }

        result_list.assign(m_vResultList.begin(), m_vResultList.end());
}
```

III. Các file test:

Test 1:

- CityFile:

Albuquerque

Chicago

San Diego

- FlightFile:

Chicago, San Diego 703 325 3

Chicago, Albuquerque 111 250 5

Albuquerque, Chicago 178 250 2

- RequestFile:

2

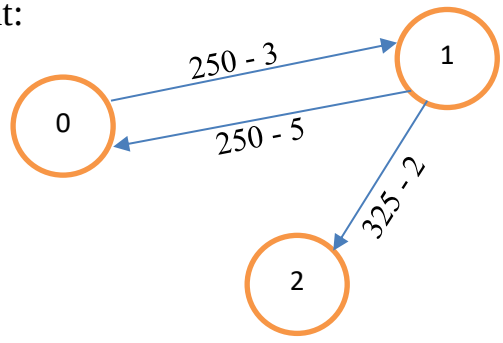
3

Albuquerque, San Diego 1

Albuquerque, Paris 2

San Diego, Chicago 1

- Output:



0: Albuquerque

1: Chicago

2: San Diego

Edge: a – b

(a: Cost, b: Duration)

```
Request is to fly from Albuquerque to San Diego.
Route 1: Albuquerque - Chicago - San Diego
Flight #178 from Albuquerque to Chicago Cost : $250 Duration : 2 hours
Flight #703 from Chicago to San Diego Cost : $325 Duration : 3 hours
Total Cost .............$575
Total Duration ......5 hours

Request is to fly from Albuquerque to Paris.
Sorry.HPAir does not serve Paris.

Request is to fly from San Diego to Chicago.
Sorry.HPAir does not fly from San Diego to Chicago.
```

Test 2:

- CityFile:

Albuquerque

Chicago

San Diego

Bangkok

Shanghai

Tan Son Nhat

- FlightFile:

Chicago, San Diego 703 325 1

Chicago, Bangkok 111 250 4

Albuquerque, Chicago 178 250 1

Albuquerque, Tan Son Nhat 132 234 1

San Diego, Tan Son Nhat 121 322 2

Bangkok, Tan Son Nhat 221 123 3

Shanghai, Albuquerque 222 125 1

Shanghai, Chicago 122 231 2

Shanghai, Bangkok 101 122 5

Shanghai, San Diego 112 240 3

- RequestFile:

3

5

Shanghai, Tan Son Nhat 1

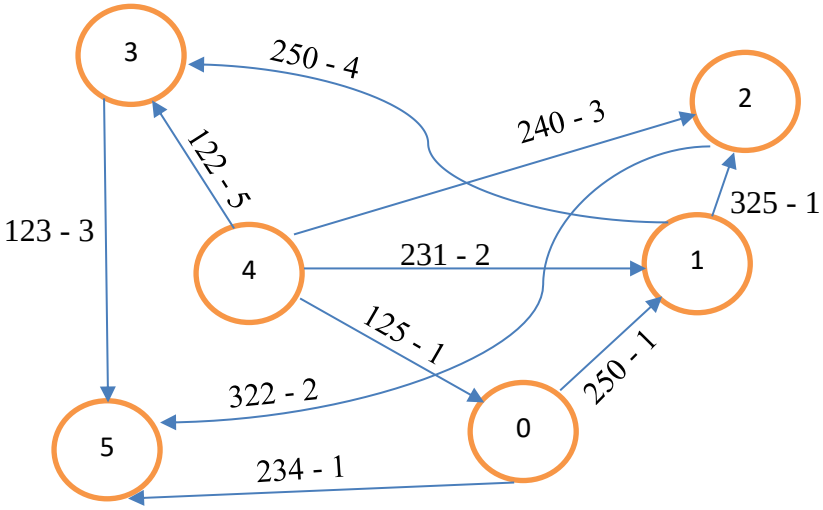
Shanghai, Tan Son Nhat 2

Albuquerque, Bangkok 2

Tan Son Nhat, Albuquerque 1

Chicago, Paris 2

- Output:



0: Albuquerque

1: Chicago

2: San Diego

3: Bangkok

4: Shanghai

5: Tan Son Nhat

Edge: a – b

(a: Cost, b: Duration)

```
Request is to fly from Shanghai to Tan Son Nhat.
Route 1: Shanghai - Bangkok - Tan Son Nhat
Flight #101 from Shanghai to Bangkok Cost : $122 Duration : 5 hours
Flight #221 from Bangkok to Tan Son Nhat Cost : $123 Duration : 3 hours
Total Cost .............$245
Total Duration ......8 hours
Route 2: Shanghai - Albuquerque - Tan Son Nhat
Flight #222 from Shanghai to Albuquerque Cost : $125 Duration : 1 hours
Flight #132 from Albuquerque to Tan Son Nhat Cost : $234 Duration : 1 hours
Total Cost .............$359
Total Duration ......2 hours
Route 3: Shanghai - San Diego - Tan Son Nhat
Flight #112 from Shanghai to San Diego Cost : $240 Duration : 3 hours
Flight #121 from San Diego to Tan Son Nhat Cost : $322 Duration : 2 hours
Total Cost .............$562
Total Duration ......5 hours

Request is to fly from Shanghai to Tan Son Nhat.
Route 1: Shanghai - Albuquerque - Tan Son Nhat
Flight #222 from Shanghai to Albuquerque Cost : $125 Duration : 1 hours
Flight #132 from Albuquerque to Tan Son Nhat Cost : $234 Duration : 1 hours
Total Cost .............$359
Total Duration ......2 hours
Route 2: Shanghai - Albuquerque - Chicago - San Diego - Tan Son Nhat
Flight #222 from Shanghai to Albuquerque Cost : $125 Duration : 1 hours
Flight #178 from Albuquerque to Chicago Cost : $250 Duration : 1 hours
Flight #703 from Chicago to San Diego Cost : $325 Duration : 1 hours
Flight #121 from San Diego to Tan Son Nhat Cost : $322 Duration : 2 hours
Total Cost .............$1022
Total Duration ......5 hours
Route 3: Shanghai - Chicago - San Diego - Tan Son Nhat
Flight #122 from Shanghai to Chicago Cost : $231 Duration : 2 hours
Flight #703 from Chicago to San Diego Cost : $325 Duration : 1 hours
Flight #121 from San Diego to Tan Son Nhat Cost : $322 Duration : 2 hours
Total Cost .............$878
Total Duration ......5 hours

Request is to fly from Albuquerque to Bangkok.
Route 1: Albuquerque - Chicago - Bangkok
Flight #178 from Albuquerque to Chicago Cost : $250 Duration : 1 hours
Flight #111 from Chicago to Bangkok Cost : $250 Duration : 4 hours
Total Cost .............$500
Total Duration ......5 hours

Request is to fly from Tan Son Nhat to Albuquerque.
Sorry.HPAir does not fly from Tan Son Nhat to Albuquerque.

Request is to fly from Chicago to Paris.
Sorry.HPAir does not serve Paris.
```

Test 3:

- CityFile:

Chicago

San Diego

Bangkok

Shanghai

Tan Son Nhat

Korea

Singapore

Taipei

California

Seattle

- FlightFile:

Chicago, California 001 120 1

San Diego, Singapore 002 245 5

Bangkok, Shanghai 003 125 3

Shanghai, Singapore 004 111 2

Shanghai, Taipei 005 111 1

Tan Son Nhat, Shanghai 006 240 1

Korea, San Diego 007 120 3

Chicago, Singapore 008 156 2

Singapore, San Diego 009 156 2

Bangkok, Singapore 010 126 4

Taipei, Korea 011 220 2

California, Tan Son Nhat 012 163 5

Seattle, Chicago 013 121 2

Seattle, Korea 014 123 2

Korea, Seattle 015 112 6

Chicago, Bangkok 016 120 1

- RequestFile:

4

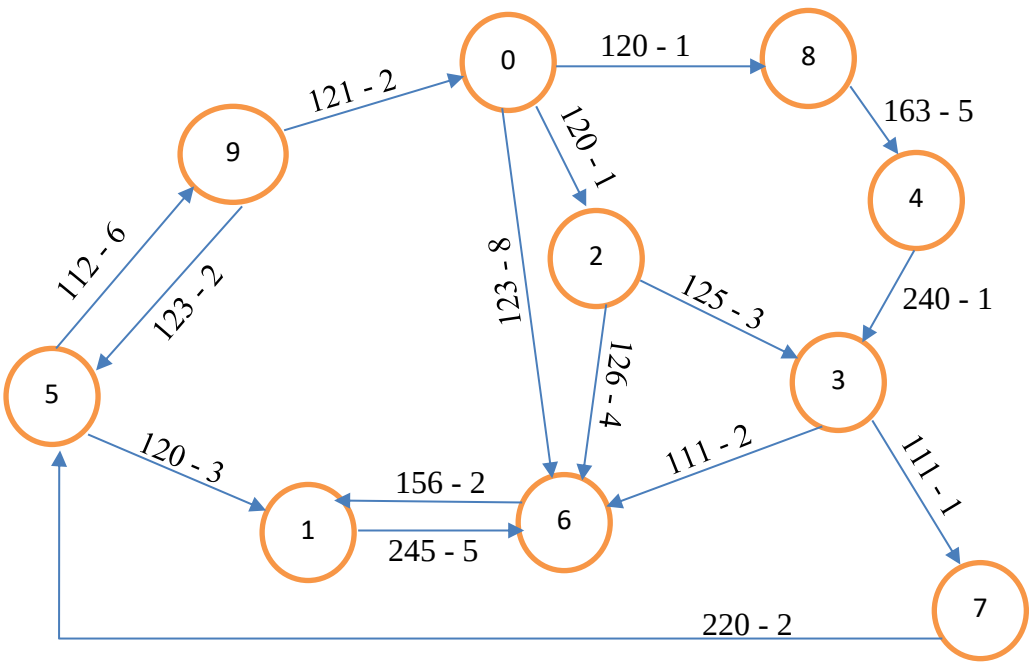
4

Shanghai, San Diego 1

Taipei, Chicago 2

Singapore, Shanghai 1

California, Seattle 2



0: Chicago	5: Korea
1: San Diego	6: Singapore
2: Bangkok	7: Taipei
3: Shanghai	8: California
4: Tan Son Nhat	9: Seattle

Edge: a – b (a: Cost, b: Duraion)

Nguồn tài liệu tham khảo

1. Thuật toán Dijkstra – Wiki
2. K lộ trình đường đi ngắn nhất – Wiki
3. Tìm kiếm k đường đi tốt nhất – blogspot
4. K-th shortest path algorithm - geeksforgeeks