Routecraft

Routing Algorithms for Ocean Shipping and Urban Deliveries

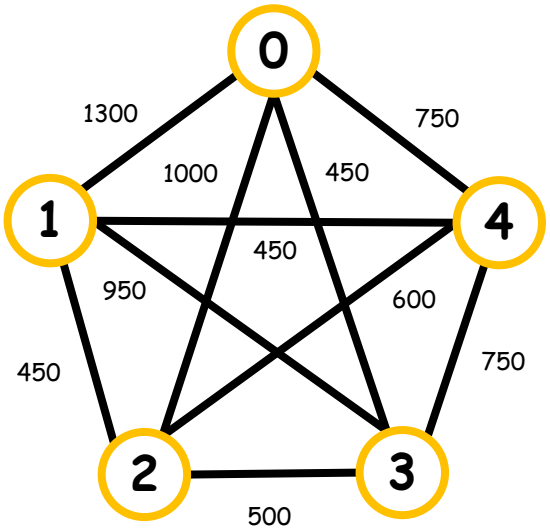
Programming Project II

Desenho de Algortimos

2022/2023

Developed by: José Santos | Luís Du | Madalena Ye | G13_4

01 Backtracking using Brute Force



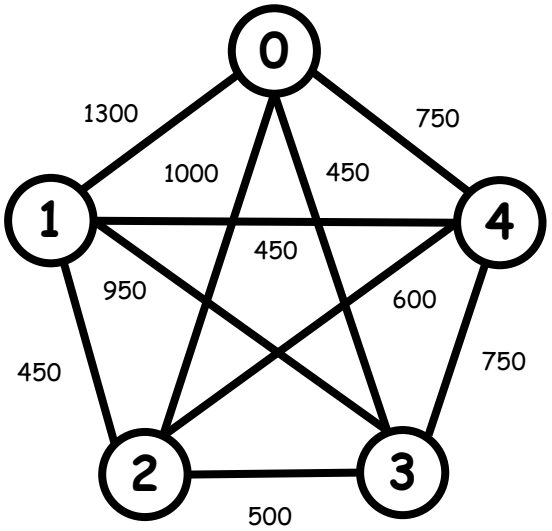

```
void Graph::tspBT(std::stack<Vertex*> &bestPath, double &minDist) {
    for(Vertex* v: vertexSet)
        v->setVisited(false);

    Vertex* startingNode = vertexSet[0];

    startingNode->setVisited(true);
    startingNode->setPathCost(0);
    startingNode->setPath(nullptr);

    minDist = INF;
    tspBTRec(0, 0, minDist, bestPath);
}
```

01 Backtracking using Brute Force

```
void Graph::tspBTRec(int curVertex, int curIndex,
                     double &minDist, std::stack<Vertex*> &bestPath) {
    Vertex* v1 = findVertex(curVertex);

    if (curIndex == vertexSet.size() - 1){
        double cost = v1->getPathCost();
        bool hasCon = false;
        for (auto e: v1->getAdj()) {
            if (e->getDest()->getId() == 0) {
                hasCon = true;
                cost += e->getDistance();
                break;
            }
        }
        if (hasCon && cost < minDist) {
            minDist = cost;
            bestPath = savePath(v1);
        }
        return;
    }
```

```
    for(Edge* edge : v1->getAdj()) {
        Vertex* v2 = edge->getDest();
        double distance = edge->getDistance();
        if(!v2->isVisited() && v1->getPathCost() + distance < minDist) {
            v2->setPath(edge);
            v2->setPathCost(v1->getPathCost() + distance);
            v2->setVisited(true);
            tspBTRec(v2->getId(), curIndex+1,minDist,bestPath);
            v2->setVisited(false);
        }
    }
}
```

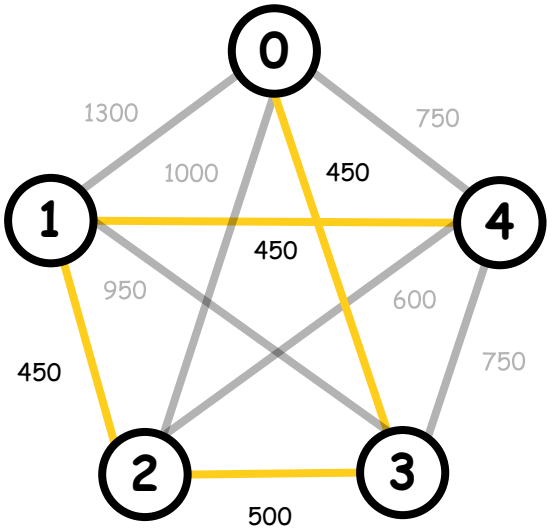
02 Triangular Approximation




```
void Graph::triangularApproximation
 (std::queue<Vertex*> &tour, double &dist) {

    prim();

    tour = preOrderTraversal();
    tour.push(vertexSet[0]);

    std::queue<Vertex*> aux = tour;
    Vertex* cur = aux.front();
    aux.pop();
    Vertex* next = aux.front();
    while (!aux.empty()){
        dist += distance(cur, next);
        cur = next;
        aux.pop();
        next = aux.front();
    }
}
```

1. Compute MST at root (id = 0) using Prim's algorithm
2. Define a pre-order walk of the MST
3. Tour the graph using the path induced by the pre-order walk.

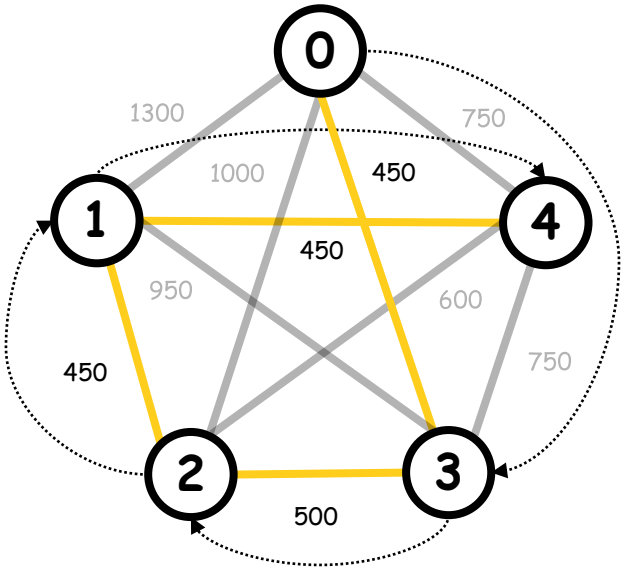
02 Triangular Approximation




```
void Graph::prim() {
    if (vertexSet.empty()) return;
    for (auto v: vertexSet){
        v->setVisited(false);
        v->setPath(nullptr);
        v->setPathCost(INF);
        v->setMstAdj({});
        v->setDegree(0);
    }
    PriorityQueue q;

    vertexSet[0]->setPathCost(0);
    q.insert(vertexSet[0]);

    while(!q.empty()){
        auto u = q.extractMin();
        u->setVisited(true);
        if (u->getPath() != nullptr) updateMst(u);
        for (auto e : u->getAdj()){
            auto w = e->getDest();
            if (!w->isVisited()){
                auto oldDist = e->getDest()->getPathCost();
                if (e->getDistance() < oldDist){
                    w->setPathCost(e->getDistance());
                    w->setPath(e);
                    if (oldDist == INF) q.insert(w);
                    else q.decreaseKey(w);
                }
            }
        }
    }
}
```

1. Compute MST at root (id = 0) using Prim's algorithm
2. Define a pre-order walk of the MST
3. Tour the graph using the path induced by the pre-order walk.

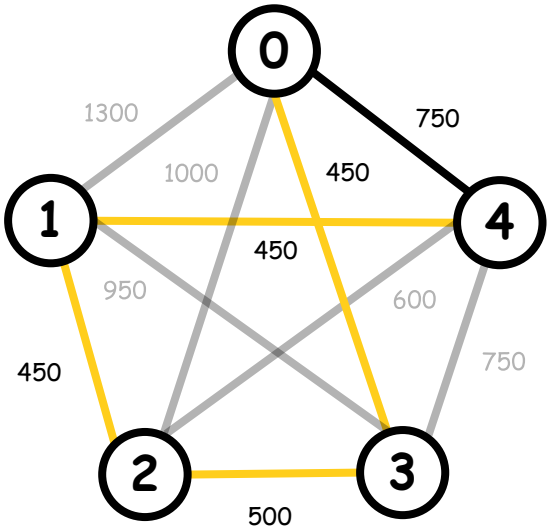
02 Triangular Approximation




```
void Graph::preOrder
 (Vertex* vertex, std::queue<Vertex *> &l) {
    l.push(vertex);
    vertex->setVisited(true);
    for (Edge* edge : vertex->getMstAdj()) {
        Vertex* w = edge->getDest();
        if (!w->isVisited())
            preOrder(w,l);
    }
}

std::queue<Vertex *> Graph::preOrderTraversal() {
    std::queue<Vertex *> l;

    for (auto v : vertexSet)
        v->setVisited(false);

    Vertex* startingNode = vertexSet[0];

    preOrder(startingNode,l);

    return l;
}
```

1. Compute MST at root (id = 0) using Prim's algorithm
2. Define a pre-order walk of the MST
3. Tour the graph using the path induced by the pre-order walk.

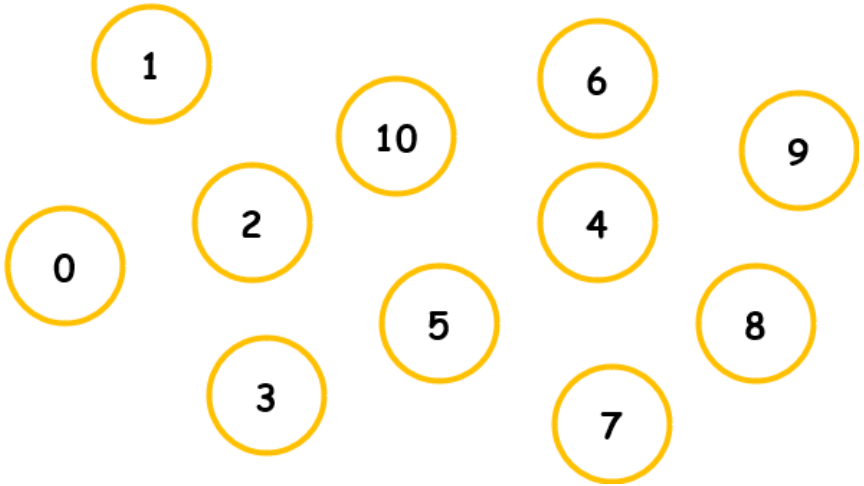
02 Triangular Approximation




```
Path: 0 3 2 1 4 0
Distance: 2600 meters
Execution Time: 0.024375 milliseconds
```

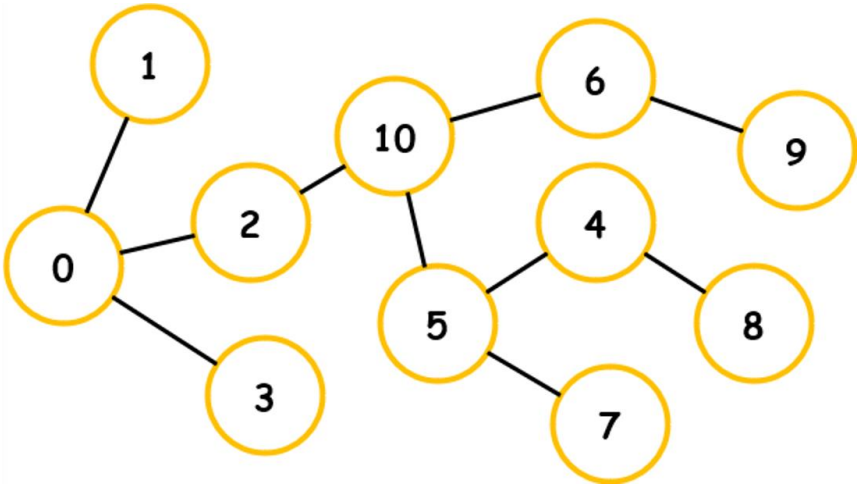
1. Compute MST at root (id = 0) using Prim's algorithm
2. Define a pre-order walk of the MST
3. Tour the graph using the path induced by the pre-order walk.

03 Christofides with 2-opt



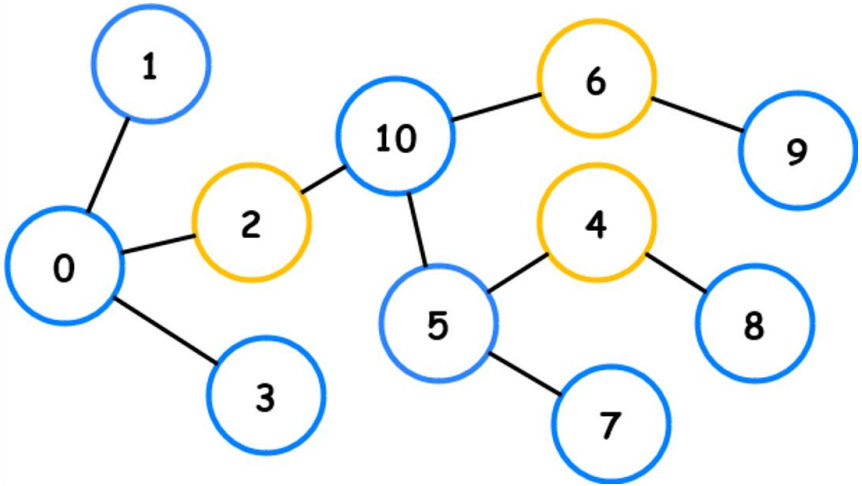
1. Compute MST
2. Add a minimum-weight perfect matching M of the odd vertices in T
3. Find an Eulerian Circuit
4. Transform the Circuit into a Hamiltonian Cycle

03 Christofides with 2-opt



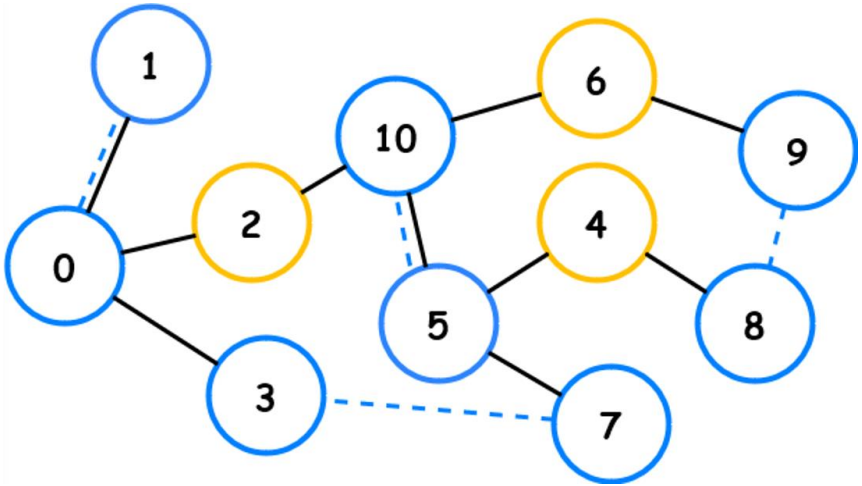
1. Compute MST
2. Add a minimum-weight perfect matching M of the odd vertices in T
3. Find an Eulerian Circuit
4. Transform the Circuit into a Hamiltonian Cycle

03 Christofides with 2-opt



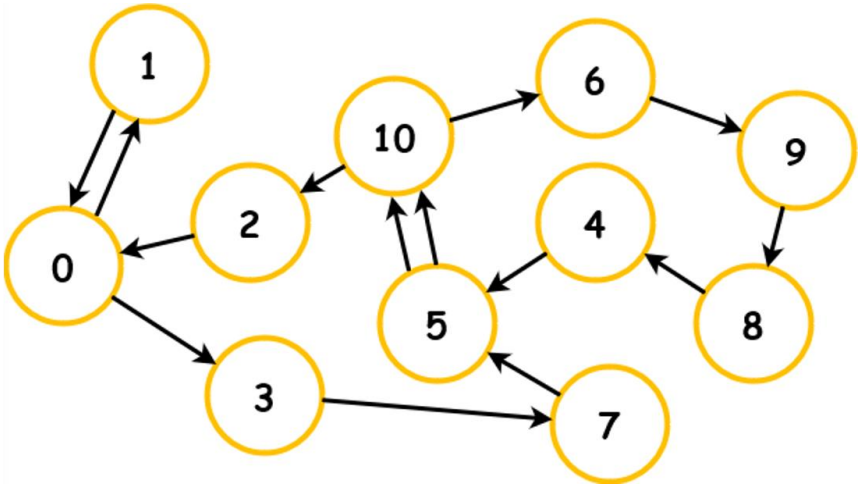
1. Compute MST
2. Add a minimum-weight perfect matching M of the odd vertices in T
3. Find an Eulerian Circuit
4. Transform the Circuit into a Hamiltonian Cycle

03 Christofides with 2-opt



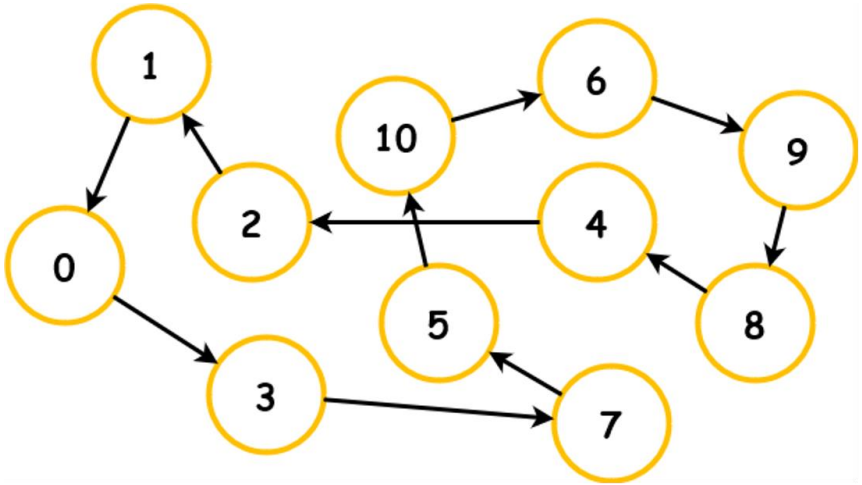
1. Compute MST
2. Add a minimum-weight perfect matching M of the odd vertices in T
3. Find an Eulerian Circuit
4. Transform the Circuit into a Hamiltonian Cycle

03 Christofides with 2-opt



1. Compute MST
2. Add a minimum-weight perfect matching M of the odd vertices in T
3. Find an Eulerian Circuit
4. Transform the Circuit into a Hamiltonian Cycle

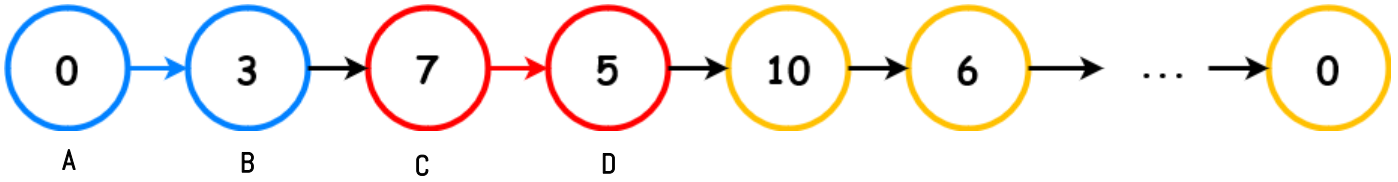
03 Christofides with 2-opt



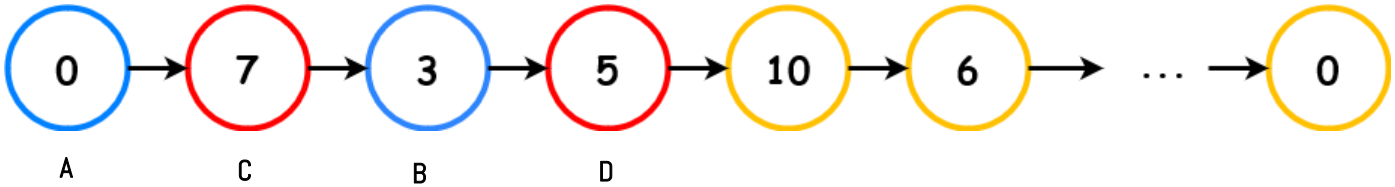
1. Compute MST
2. Add a minimum-weight perfect matching M of the odd vertices in T
3. Find an Eulerian Circuit
4. Transform the Circuit into a Hamiltonian Cycle

0 -> 3 -> 7 -> 5 -> 10 -> 6 -> 9 -> 8 -> 4 -> ~~5~~ -> ~~10~~ -> 2 -> ~~0~~ -> 1 -> 0

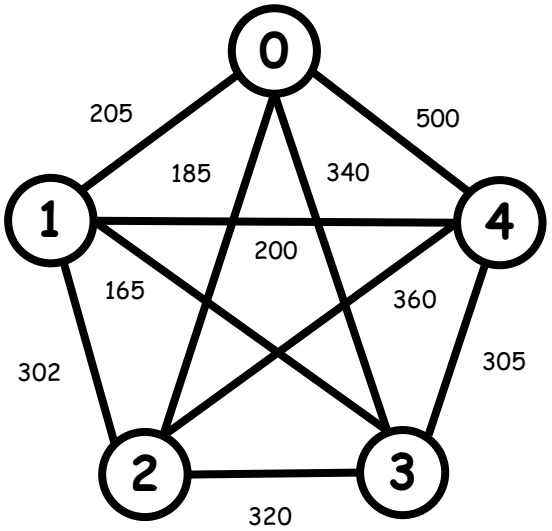
03 2-opt algorithm



If distance(A,C) + distance(B,D) < distance(A,B) + distance(C,D)



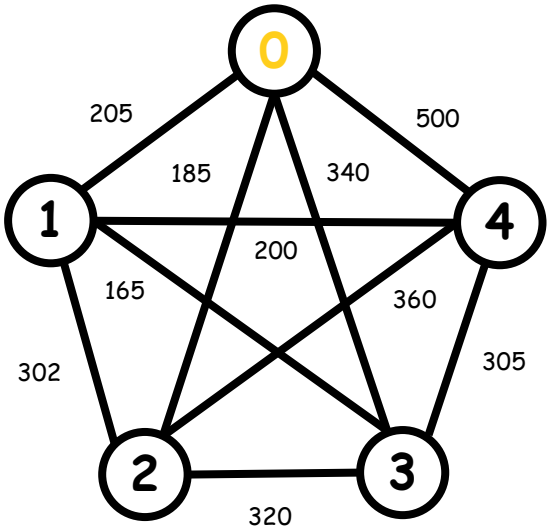
04 Nearest neighbour




```
void Graph::nearestNeighborTSP
 (std::vector<Vertex *> &tour, double &distance) {
    for (auto v : vertexSet)
        v->setVisited(false);

    Vertex* currentVertex = vertexSet[0];
    currentVertex->setVisited(true);

    while(true){
        tour.push_back(currentVertex);
        double minDist = INT_MAX;
        Vertex* nextVertex = nullptr;
        for (auto e : currentVertex->getAdj()){
            Vertex* neighbor = e->getDest();
            if (!neighbor->isVisited()){
                double dist = e->getDistance();
                if (dist < minDist){
                    minDist = dist;
                    nextVertex = neighbor;
                }
            }
        }
        if (nextVertex == nullptr) break;
        nextVertex->setVisited(true);
        distance += minDist;
        currentVertex = nextVertex;
    }

    distance += currentVertex->getAdj()[0]->getDistance();
    tour.push_back(vertexSet[0]);
}
```

1. Start at the root
2. Find out the shortest edge connecting the current vertex and an unvisited neighbour
3. Repeat the process until all the vertices have been visited
4. Return to starting point to obtain the tour

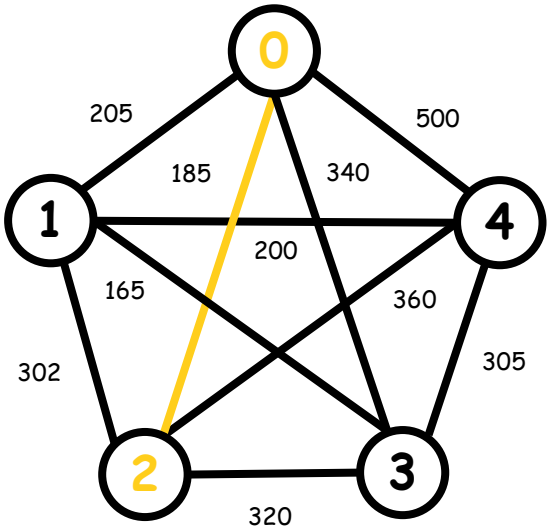
04 Nearest neighbour




```
void Graph::nearestNeighborTSP
 (std::vector<Vertex *> &tour, double &distance) {
    for (auto v : vertexSet)
        v->setVisited(false);

    Vertex* currentVertex = vertexSet[0];
    currentVertex->setVisited(true);

    while(true){
        tour.push_back(currentVertex);
        double minDist = INT_MAX;
        Vertex* nextVertex = nullptr;
        for (auto e : currentVertex->getAdj()){
            Vertex* neighbor = e->getDest();
            if (!neighbor->isVisited()){
                double dist = e->getDistance();
                if (dist < minDist){
                    minDist = dist;
                    nextVertex = neighbor;
                }
            }
        }
        if (nextVertex == nullptr) break;
        nextVertex->setVisited(true);
        distance += minDist;
        currentVertex = nextVertex;
    }

    distance += currentVertex->getAdj()[0]->getDistance();
    tour.push_back(vertexSet[0]);
}
```

1. Start at the root
2. Find out the shortest edge connecting the current vertex and an unvisited neighbour
3. Repeat the process until all the vertices have been visited
4. Return to starting point to obtain the tour

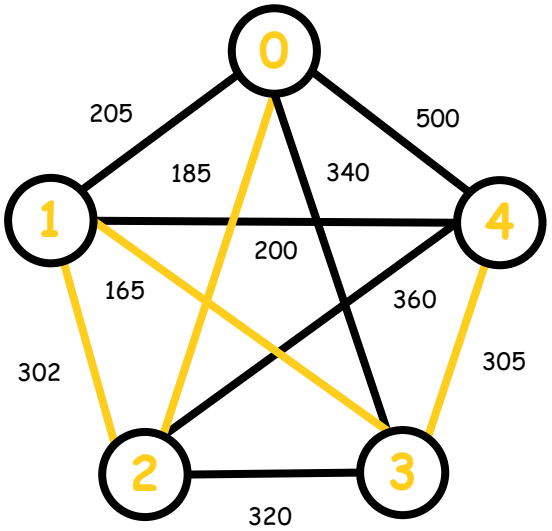
04 Nearest neighbour




```
void Graph::nearestNeighborTSP
 (std::vector<Vertex *> &tour, double &distance) {
    for (auto v : vertexSet)
        v->setVisited(false);

    Vertex* currentVertex = vertexSet[0];
    currentVertex->setVisited(true);

    while(true){
        tour.push_back(currentVertex);
        double minDist = INT_MAX;
        Vertex* nextVertex = nullptr;
        for (auto e : currentVertex->getAdj()){
            Vertex* neighbor = e->getDest();
            if (!neighbor->isVisited()){
                double dist = e->getDistance();
                if (dist < minDist){
                    minDist = dist;
                    nextVertex = neighbor;
                }
            }
        }
        if (nextVertex == nullptr) break;
        nextVertex->setVisited(true);
        distance += minDist;
        currentVertex = nextVertex;
    }

    distance += currentVertex->getAdj()[0]->getDistance();
    tour.push_back(vertexSet[0]);
}
```

1. Start at the root
2. Find out the shortest edge connecting the current vertex and an unvisited neighbour
3. Repeat the process until all the vertices have been visited
4. Return to starting point to obtain the tour

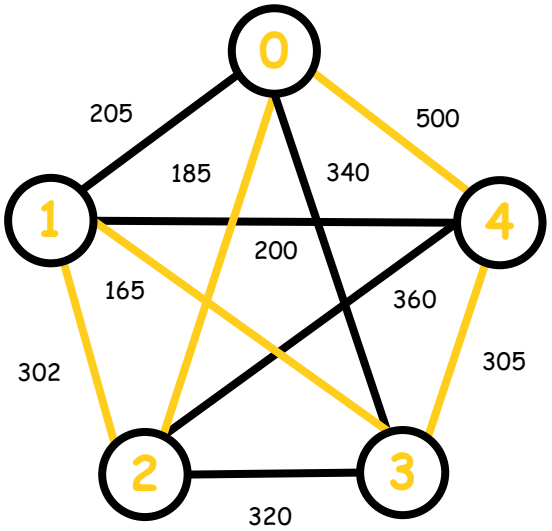
04 Nearest neighbour




```
void Graph::nearestNeighborTSP
 (std::vector<Vertex *> &tour, double &distance) {
    for (auto v : vertexSet)
        v->setVisited(false);

    Vertex* currentVertex = vertexSet[0];
    currentVertex->setVisited(true);

    while(true){
        tour.push_back(currentVertex);
        double minDist = INT_MAX;
        Vertex* nextVertex = nullptr;
        for (auto e : currentVertex->getAdj()){
            Vertex* neighbor = e->getDest();
            if (!neighbor->isVisited()){
                double dist = e->getDistance();
                if (dist < minDist){
                    minDist = dist;
                    nextVertex = neighbor;
                }
            }
        }
        if (nextVertex == nullptr) break;
        nextVertex->setVisited(true);
        distance += minDist;
        currentVertex = nextVertex;
    }

    distance += currentVertex->getAdj()[0]->getDistance();
    tour.push_back(vertexSet[0]);
}
```

1. Start at the root
2. Find out the shortest edge connecting the current vertex and an unvisited neighbour
3. Repeat the process until all the vertices have been visited
4. Return to starting point to obtain the tour

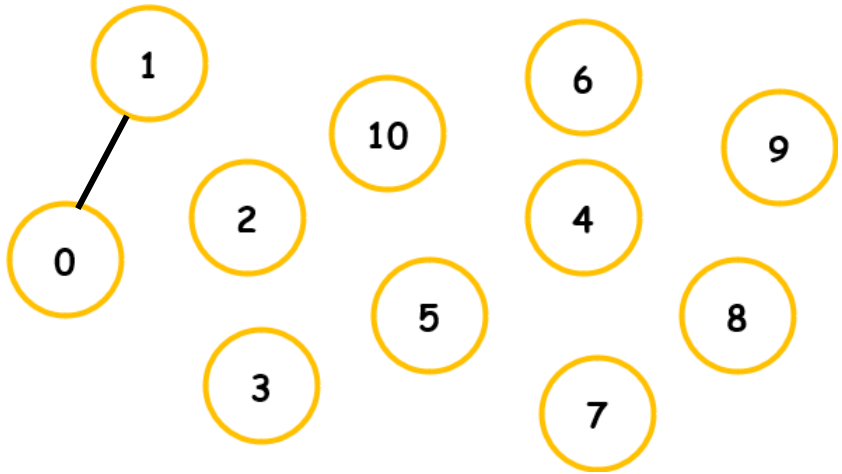
04 Nearest neighbour




```
void Graph::nearestNeighborTSP
 (std::vector<Vertex *> &tour, double &distance) {
    for (auto v : vertexSet)
        v->setVisited(false);

    Vertex* currentVertex = vertexSet[0];
    currentVertex->setVisited(true);

    while(true){
        tour.push_back(currentVertex);
        double minDist = INT_MAX;
        Vertex* nextVertex = nullptr;
        for (auto e : currentVertex->getAdj()){
            Vertex* neighbor = e->getDest();
            if (!neighbor->isVisited()){
                double dist = e->getDistance();
                if (dist < minDist){
                    minDist = dist;
                    nextVertex = neighbor;
                }
            }
        }
        if (nextVertex == nullptr) break;
        nextVertex->setVisited(true);
        distance += minDist;
        currentVertex = nextVertex;
    }

    distance += currentVertex->getAdj()[0]->getDistance();
    tour.push_back(vertexSet[0]);
}
```

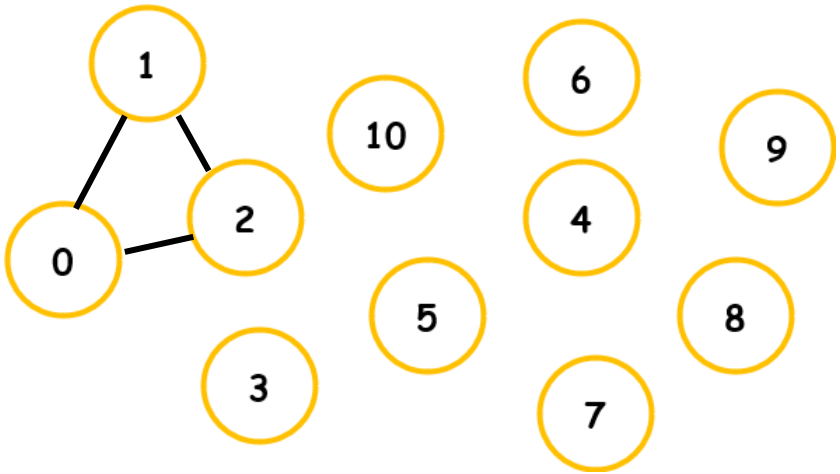
1. Start at the root
2. Find out the shortest edge connecting the current vertex and an unvisited neighbour
3. Repeat the process until all the vertices have been visited
4. Return to starting point to obtain the tour

05 Nearest insertion with 2-opt



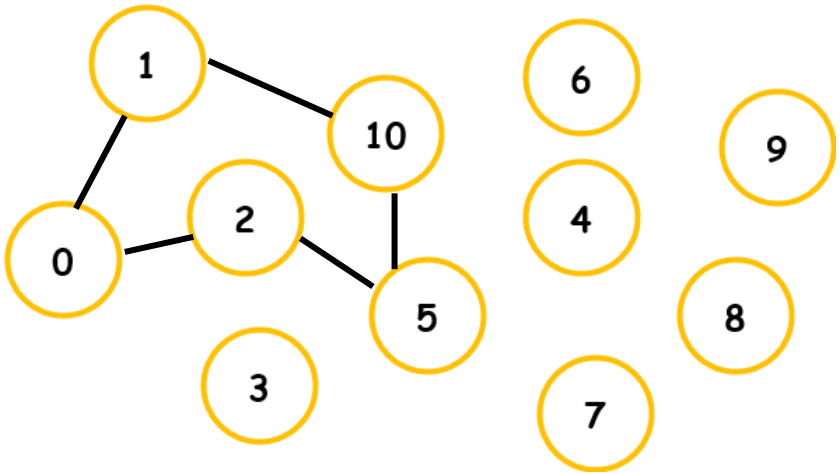
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



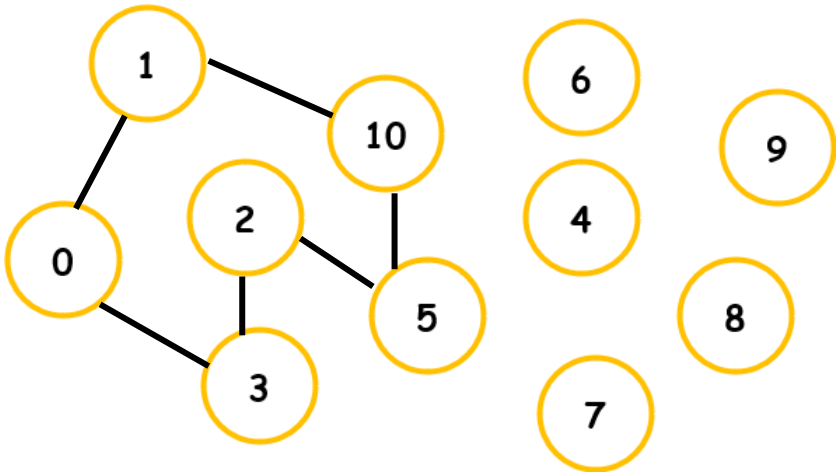
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



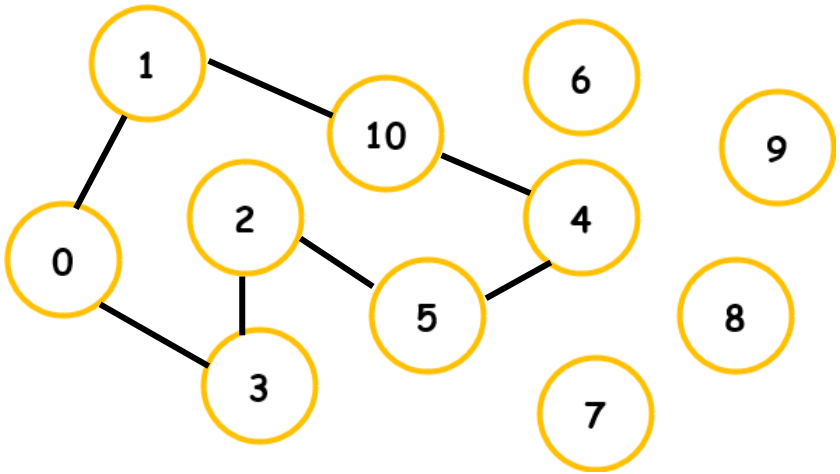
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



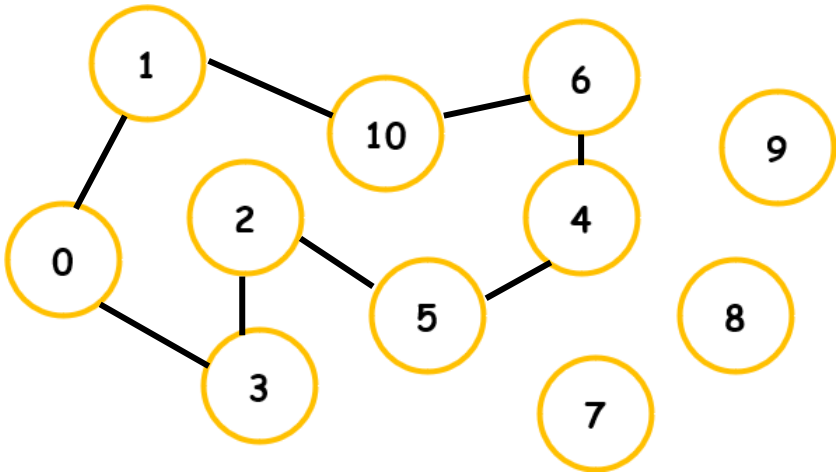
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



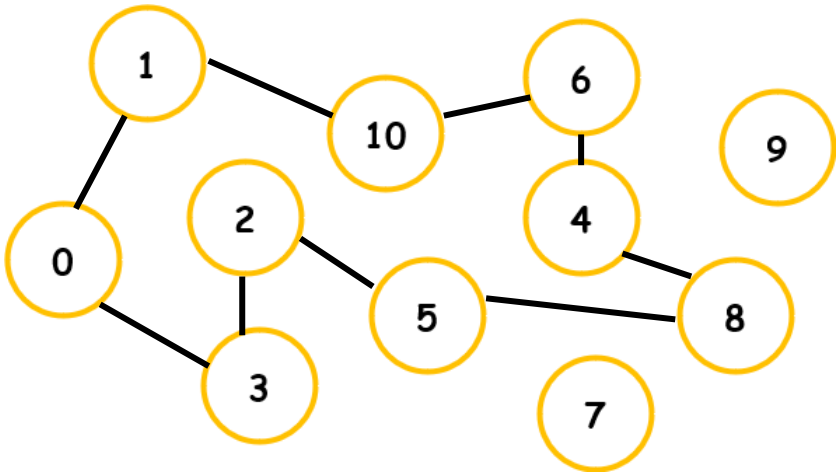
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



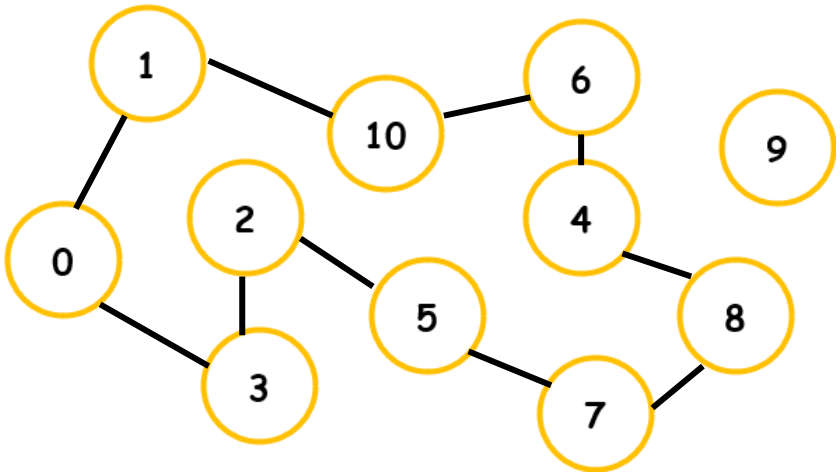
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



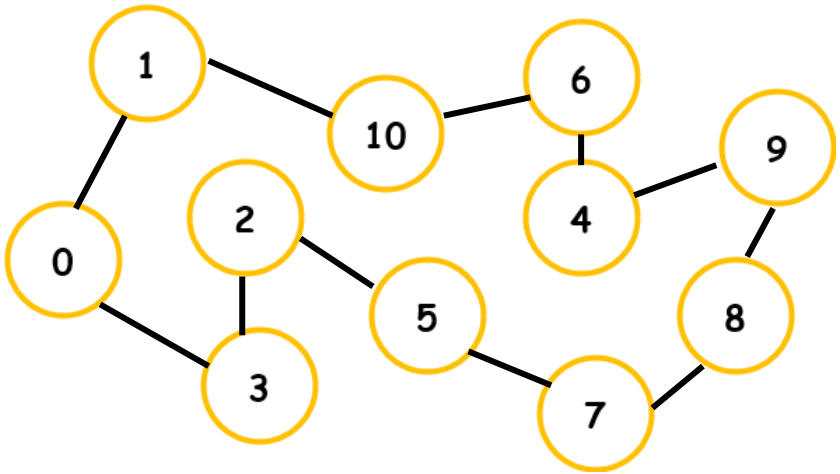
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



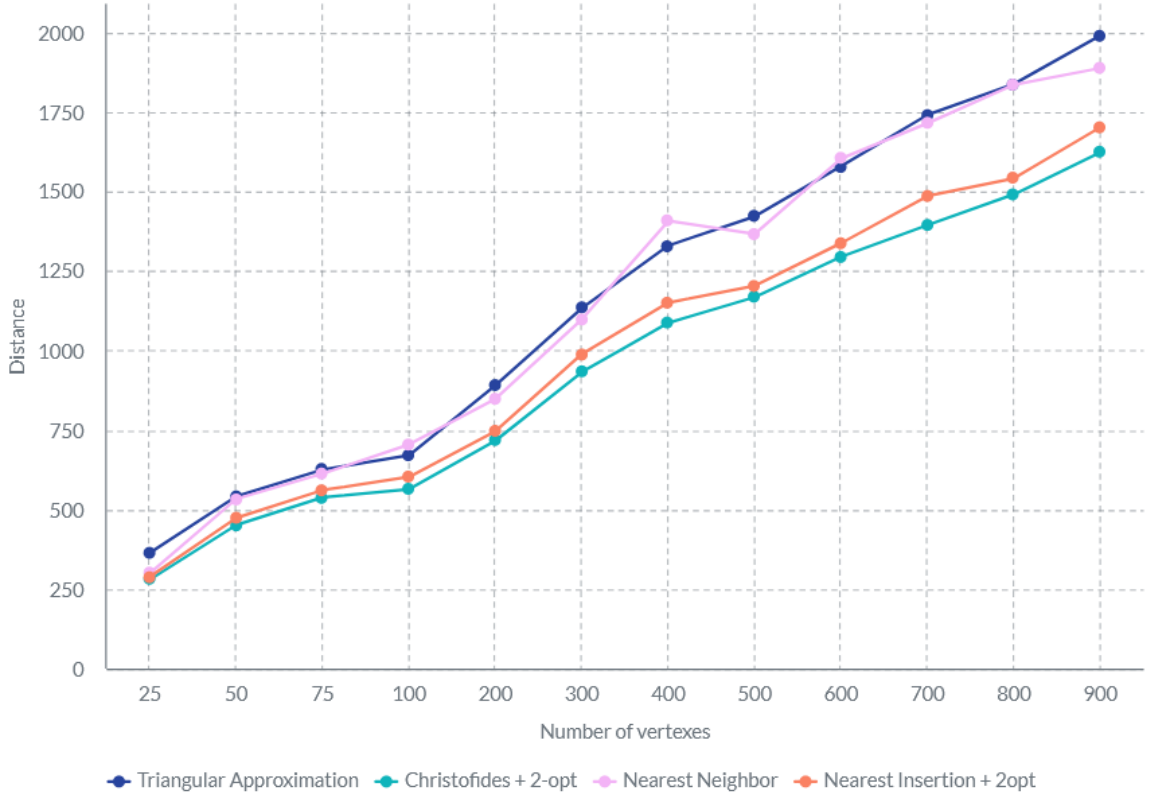
1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

05 Nearest insertion with 2-opt



1. Choose smallest edge from tour visited vertexes
2. Repeat until there are no more unvisited vertexes
3. Apply the 2-opt algorithm

Distance comparison



Time comparison