2º Projeto de Desenho de Algoritmos

Turma 2, Grupo 5
Elementos: David Cordeiro, Diogo Viana, Gonçalo Martins

Algoritmo de backtracking

```
void Algorithms::backtrackingTsp(const vector<vector<float>>& distances, vector<bool>& visited, vector<int>& solution, vector<int>& currTour,
                                 const int& currPos, const int& n, const int& count, const float& currCost, float& solCost) {
    if ((count == n) && (distances[currPos][0] != INFINITY_FLOAT)) {
        float smallest = min(solCost, currCost + distances[currPos][0]);
        if (smallest < solCost) {
            solCost = smallest;
            solution = currTour;
        }
        return;
    }

    for (int i = 0; i < n; i++) {
        if (!visited[i] && distances[currPos][i] != INFINITY_FLOAT) {
            visited[i] = true;
            currTour.push_back(i);

            backtrackingTsp(distances, &: visited, &: solution, &: currTour, currPos: i, n, count: count + 1, currCost: currCost + distances[currPos][i], &: solCost);

            currTour.pop_back();
            visited[i] = false;
        }
    }
}
```

Percorre recursivamente todos os vértices da tsp, mantendo as informações necessárias (o caminho percorrido até então e o seu custo, número de vértices visitados) na stack. Ao percorrer todos os vértices, verifica primeiramente se um ciclo hamiltoniano é formado. Se sim, então verifica se o custo total do percurso percorrido é o melhor até então, atualizando o respectivo custo e percurso da solução. Após isso, ocorre o momento de backtracking. Vértices são removidos do tour atual, o array de vértices visitados marcado a falso e a função continua assim a procurar tours com o menor possível custo total

Algoritmo de aproximação triangular

Usa o algoritmo de prim para encontrar uma minimum spanning tree dos vértices do problema.

```
vector<vector<float>> Graph::primFullyConnected(int s, vector<bool> &visited) {
    vector<vector<float>> newAdjMatrix;

    newAdjMatrix.resize( new_size: getNumOfNodes());

    for (vector<float>& row : newAdjMatrix) {
        row.resize( new_size: getNumOfNodes(), x: INFINITY_FLOAT);
    }


    visited[s] = true;
    priority_queue<Edge> pq;
    for (int i = 0; i < adjMatrix.size(); i++) {
        if (i != s) {
            if(adjMatrix[s][i] == INFINITY_FLOAT){continue;}
            else{pq.push( x: Edge( u: s,  v: i,  w: adjMatrix[s][i]));}
        }
    }

    double mst_cost = 0;
    while (!pq.empty()) {
        Edge e = pq.top(); pq.pop();
        if (visited[e.v]) continue;
        visited[e.v] = true;
        mst_cost += e.w;
        newAdjMatrix[e.u][e.v] = e.w; newAdjMatrix[e.v][e.u] = e.w;
        for (int i = 0; i < adjMatrix.size(); i++) {
            if (!visited[i]){
                if(adjMatrix[e.v][i] == INFINITY_FLOAT){continue;}
                else{pq.push( x: Edge( u: e.v,  v: i,  w: adjMatrix[e.v][i]));}
            }
        }
    }
    return newAdjMatrix;
}
```

Usa dfs para executar um preorder traversal da mst encontrada, de forma a encontrar uma solução aproximada para o tsp

```
void Algorithms::dfs(int u, vector<vector<float>> &adjMatrix, vector<bool> &visited, vector<int> &preorder) {
    visited[u] = true;
    preorder.push_back(u);
    for(int v = 1; v < adjMatrix[u].size(); v++){
        if(!visited[v] && (adjMatrix[u][v] != INFINITY_FLOAT)){
            dfs( u: v,  &: adjMatrix,  &: visited,  &: preorder);
        }
    }
    return;
}
```

A nossa heurística

5 Passos:

- Uso de clustering para juntar vértices relativamente próximos e reduzir o tamanho do problema
- Nearest neighbor + 2 opt para encontrar uma solução razoável para a tsp dentro de cada cluster
- Calcular centróides de cada cluster com base na soma das distâncias aos outros vértices
- Determinar uma solução para a tsp entre os centróides e outliers
- Reconstruir o caminho global a partir da solução encontrada.

Clustering

Percorre todos os vértices até encontrar o primeiro core point. Marca o core point como visitado, forma um cluster e expande-o. Depois de uma primeira passagem , volta a iterar pelos vértices para verificar aqueles que não foram visitados (não fazem parte de nenhum cluster), e marca-os como outliers.

```cpp
vector<vector<int>> Algorithms::clusteringDBSCAN(const vector<vector<float>>& distances, const float& epsilon, const int& minPts, vector<int>& outliers) {
    vector<vector<int>> closePointsList = getClosePointsList(distances, epsilon);

    int numCities = static_cast<int>(closePointsList.size());
    vector<bool> visited( n: numCities);
    vector<vector<int>> clusters;

    for (int cityIndex = 0; cityIndex < numCities; cityIndex++) {
        bool isCorePoint = (closePointsList[cityIndex].size() >= minPts);
        if (!visited[cityIndex] && isCorePoint) {
            visited[cityIndex] = true;
            vector<int> cluster = {cityIndex};
            expandCluster(closePointsList,  &: cluster, cityIndex, minPts,  &: visited);
            clusters.push_back(cluster);
        }
    }

    for (int cityIndex = 0; cityIndex < numCities; cityIndex++) {
        if (!visited[cityIndex]) {
            outliers.push_back(cityIndex);
        }
    }

    return clusters;
}
```

```cpp
void Algorithms::expandCluster(const vector<vector<int>>& closePointsList, vector<int>& cluster, const int& cityIndex, const int& minPts, vector<bool>& visited) {
    queue<int> neighbors;
    neighbors.push( x: cityIndex);

    while (!neighbors.empty()) {
        int currentCity = neighbors.front();
        neighbors.pop();

        vector<int> neighborCities = closePointsList[currentCity];
        bool isCorePoint = (neighborCities.size() >= minPts);

        if (isCorePoint) {
            for (const int& neighbor : neighborCities) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    cluster.push_back(neighbor);
                    neighbors.push( x: neighbor);
                }
            }
        }
    }
}
```

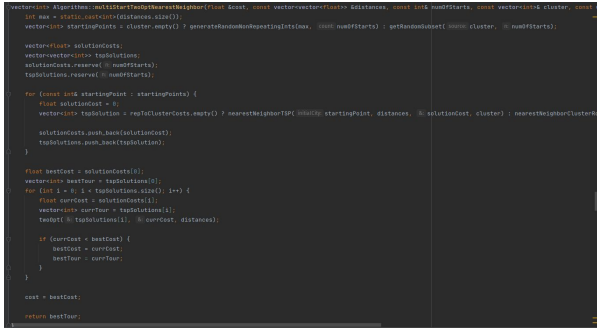
Adiciona data points a uma queue. Se o data point atual for um core point então expande o cluster (os seus vizinhos são adicionados à queue), se não é apenas adicionado ao cluster

```cpp
vector<vector<int>> Algorithms::getClosePointsList(const vector<vector<float>>& distances, const float& epsilon) {
    int n = static_cast<int>(distances.size());
    vector<vector<int>> neighborsList(n);

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (distances[i][j] <= epsilon) {
                neighborsList[i].push_back(j);
                neighborsList[j].push_back(i);
            }
        }
    }

    return neighborsList;
}
```

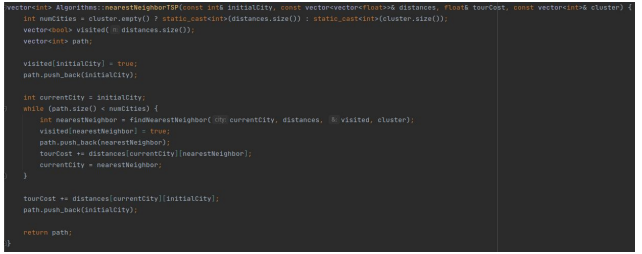
Usada para diminuir o número de cálculos na fase de clustering ao determinar core points. Retorna uma matriz com todos os vértices a uma distância igual ou menor que épsilon, para cada vértice.

Multistart nearest neighbor + 2 opt

Determina um certo número de random starting points. Usa nearest neighbor para gerar uma solução para o tsp, com cada starting point que gerou. Otimiza cada solução encontrada com 2-opt e retorna a melhor solução



Constrói um percurso cíclico através de um algoritmo greedy que escolhe sempre o vértice mais próximo do atual.



Compara quaisquer 2 edges não consecutivas num percurso cíclico e determina se a única troca possível que mantém um percurso válido, resulta num melhoramento do custo do percurso. Se sim, então executa a troca e continua à procura de melhorias.

Representantes dos clusters

Encontra o vértice representante de cada cluster.

```cpp
vector<int> Algorithms::findClustersRepresentatives(const vector<vector<int>> &clusters, const vector<vector<float>> &distances) {
    vector<int> clusterRepresentatives;
    for (const vector<int>& cluster : clusters) {
        int representative = getClusterRepresentative(cluster, distances);
        clusterRepresentatives.push_back(representative);
    }
    return clusterRepresentatives;
}
```

Determina a maior soma das distâncias de um vértice a qualquer outro do mesmo cluster, e escolhe esse vértice como o representante do vértice.

```cpp
int Algorithms::getClusterRepresentative(const vector<int>& cluster, const vector<vector<float>>& distances) {
    int clusterSize = static_cast<int>(cluster.size());
    int representativeIndex = -1;
    float minAvgDistance = INFINITY_FLOAT;

    for (int i = 0; i < clusterSize; i++) {
        float totalDistance = 0.0;

        for (int j = 0; j < clusterSize; j++) {
            if (i != j) {
                totalDistance += distances[cluster[i]][cluster[j]];
            }
        }

        if (totalDistance < minAvgDistance) {
            minAvgDistance = totalDistance;
            representativeIndex = cluster[i];
        }
    }

    return representativeIndex;
}
```

Solução reduzida

Encontra uma solução para o tsp considerando apenas representantes de clusters e outliers. Faz uso do multi start nearest neighbor + 2 opt descrito acima, mas também tendo em consideração os custos das soluções para o tsp de cada cluster.

```
float reducedTspInstanceCost = 0;
vector<int> reducedTspInstanceSolution = Algorithms::multiStartTwoOptNearestNeighbor( & reducedTspInstanceCost, distances,  numOfStarts: static_cast<int>(reducedTspInstanc
```

Reconstrução da solução global

Reconstrói um percurso com todos os vértices a partir da solução encontrada usando os representantes dos clusters.

```
int n = static_cast<int>(reducedTspInstanceSolution.size());
vector<int> finalSolution;
finalSolution.reserve( n: distances.size());
finalSolution.push_back(reducedTspInstanceSolution[0]);

for (int i = 1; i < n; i++) {
    int currVertice = reducedTspInstanceSolution[i];
    bool isCluster = repToClusterSol.contains( x: currVertice);
    if (isCluster) {
        vector<int> clusterSol = repToClusterSol[currVertice];
        finalSolution.insert( position: finalSolution.end(),  first: make_move_iterator( i: clusterSol.begin()),  last: make_move_iterator( i: clusterSol.end() - 1));
    } else {
        finalSolution.push_back(currVertice);
    }
}

bool firstWasCluster = repToClusterSol.contains( x: reducedTspInstanceSolution[0]);
if (firstWasCluster) {
    resultCost = reducedTspInstanceCost - distances[finalSolution[0]][finalSolution[1]] + distances[finalSolution.back()][finalSolution[1]];
    finalSolution[0] = finalSolution.back();
} else {
    resultCost = reducedTspInstanceCost;
}
```