Temă pentru acasă - PCC - Problema Comerciantului Călător

STAN ADRIAN MARIAN

Aprilie 2024

1 Introducere

Link către GitHub

2 Enunțul problemei

TSP - Problema Comerciantului Călător
Dată fiind o listă de orașe și distanțele dintre fiecare pereche de orașe, care este ruta de cost minim care vizitează fiecare oraș exact o dată și se întoarce la orașul de origine? Funcția de cost a soluției trebuie să fie minimizarea celei mai lungi distanțe dintre două orașe consecutive.

Trebuie să furnizați implementările pentru următoarele strategii:

- Strategii exhaustive:
  - Căutare în lățime (BFS)
  - Căutare de cost minim (UCS)
- Strategie euristică:
  - Căutare A*

3 Pseudocodul algoritmilor

3.1 Căutare în lățime (BFS)

```
function TSP_BFS(dist) returns the minimum cost and path
persistent: queue, an empty queue of Nodes

min_cost, initially INF
best_path, an empty list

initialize:
INITIAL_NODE      Node(city: 0, path: [0], path_size: 1, cost: 0, max_edge: 0)

queue.push(INITIAL_NODE)

while NOT queue.is_empty() do
    curr <- queue.pop()

    if curr.path_size = number_of_cities + 1 AND curr.path[curr.path_size - 1] = 0 then
        max_edge <- CALCULATE_MAX_EDGE(curr.path, dist)
        if max_edge < min_cost then
            min_cost <- max_edge
            best_path <- curr.path
        continue

    for each city i in range(0 to number_of_cities) do
```

```
        if city i NOT in curr.path OR (curr.path_size = number_of_cities AND i = 0) then
            next <- COPY curr
            next.path.append(i)
            next.path_size <- next.path_size + 1
            next.cost <- next.cost + dist[curr.city][i]
            next.max_edge <- MAX(curr.max_edge, dist[curr.city][i])
            queue.push(next)

PRINT "BFS TSP Path:", best_path
PRINT "Maximum Edge:", min_cost

function CALCULATE_MAX_EDGE(path, dist) returns the maximum edge
    max_edge      0
    for i in range(1 to LENGTH(path)) do
        u <- path[i-1]
        v <- path[i]
        max_edge      MAX(max_edge, dist[u][v])
    return max_edge
```

3.2 Căutare de cost minim (UCS)

```
function TSP_UCS(dist) returns the minimum cost and path
persistent: priority_queue, an empty priority queue of Nodes

min_cost, initially INF
best_path, an empty list

initialize:
INITIAL_NODE      Node(city: 0, cost: 0, max_edge: 0, count: 1, visited: ARRAY of FALSE v
INITIAL_NODE.visited[0] = TRUE
priority_queue.push(INITIAL_NODE)

while NOT priority_queue.is_empty() do
    curr <- priority_queue.pop()

    if curr.count = number_of_cities AND dist[curr.city][0] is not INF then
        min_cost <- MIN(min_cost, MAX(curr.max_edge, dist[curr.city][0]))
        best_path <- curr.path
        continue

    for each city i in range(0 to number_of_cities) do
        if NOT curr.visited[i] AND dist[curr.city][i] is not INF then
            next <- COPY curr
            next.city <- i
            next.cost <- next.cost + dist[curr.city][i]
            next.max_edge <- MAX(curr.max_edge, dist[curr.city][i])
            next.count <- next.count + 1
            next.visited[i] <- TRUE
            priority_queue.push(next)

return (min_cost, best_path)
```

3.3 Căutare A*

```
function TSP_A_STAR(dist) returns the minimum cost and path
persistent: priority_queue, an empty priority queue of Nodes
```

```
min_cost, initially INF
best_path, an empty list

initialize:
INITIAL_NODE <- Node(city: 0, path: [0], path_size: 1, cost: 0, max_edge: 0)
priority_queue.push(INITIAL_NODE)

while NOT priority_queue.is_empty() do
    curr <- priority_queue.pop()

    if curr.path_size = number_of_cities + 1 AND curr.path[curr.path_size - 1] = 0 then
        max_edge <- CALCULATE_MAX_EDGE(curr.path, dist)
        if max_edge < min_cost then
            min_cost <- max_edge
            best_path <- curr.path
        continue

    for each city i in range(0 to number_of_cities) do
        if city i NOT in curr.path OR (curr.path_size = number_of_cities AND i = 0) then
            next <- COPY curr
            next.path.append(i)
            next.path_size      next.path_size + 1
            next.cost <- next.cost + dist[curr.city][i] + HEURISTIC(next.path, dist, nu
            next.max_edge <- MAX(curr.max_edge, dist[curr.city][i])
            priority_queue.push(next)

PRINT "A* TSP Path:", best_path
PRINT "Maximum Edge:", min_cost

function CALCULATE_MAX_EDGE(path, dist) returns the maximum edge
    max_edge      0
    for i in range(1 to LENGTH(path)) do
        u <- path[i-1]
        v <- path[i]
        max_edge      MAX(max_edge, dist[u][v])
    return max_edge

function HEURISTIC(path, dist, num_cities) returns a heuristic value
    total <- 0
    count <- 0
    for i in range(0 to num_cities) do
        for j in range(0 to num_cities) do
            if dist[i][j] is not INF then
                total <- total + dist[i][j]
                count <- count + 1
    return (count > 0) ? total / count : 0
```

4 Schița aplicației

4.1 Prezentare generală arhitecturală de nivel înalt a aplicației

Aplicația este concepută pentru a rezolva Problema Comerciantului Călător (TSP) utilizând trei strategii de căutare diferite: Căutare în lățime (BFS), Căutare de cost minim (UCS) și Căutare A*. Scopul este de a găsi ruta de cost minim care vizitează fiecare oraș exact o dată și se întoarce la orașul de origine, cu funcția de cost fiind minimizarea celei mai lungi distanțe dintre două orașe consecutive.

Arhitectura constă din patru module principale:

1. Modulul de structuri de date: Definește structurile de date utilizate pentru noduri și muchii.

2. Modulul de funcții ajutătoare: Conține funcții utilitare pentru calcule și euristici.

3. Modulul de algoritmi de căutare: Implementează algoritmii BFS, UCS și A*.

4. Modulul principal: Inițializează matricea de distanțe și invocă algoritmii de căutare.

4.2 Specificația formatului de date de intrare

Formatul de date de intrare constă dintr-o matrice de distanțe care reprezintă distanțele dintre perechile de orașe. Matricea de distanțe este un array 2D unde elementul la indexul [i][j] reprezintă distanța dintre orașul i și orașul j. Dacă nu există o cale directă între două orașe, distanța este reprezentată ca INF (infinit).

Exemplu:

```
[0, 10, 15, 20],
[10, 0, 35, 25],
[15, 35, 0, 30],
[20, 25, 30, 0]
```

4.3 Specificația formatului de date de ieșire

Formatul de date de ieșire include:

1. Costul minim găsit utilizând fiecare algoritm de căutare.

2. Drumul parcurs pentru a obține acest cost minim.

Exemplu:

```
Cost minim folosind UCS: 35
Drum: [0, 1, 3, 2, 0]

Cost minim folosind BFS: 35
Drum: [0, 1, 3, 2, 0]

Cost minim folosind Căutare A*: 35
Drum: [0, 1, 3, 2, 0]
```

5 Experimente și evaluare

Exemplu 1:

```
[0, 10, 25, 15]
[10, 0, 30, 20]
[25, 30, 0, 35]
[15, 20, 35, 0]
```

5.1 Rezultate

- Cost minim folosind UCS: 30
- Drum: [0, 1, 3, 2, 0]
- Cost minim folosind BFS: 30
- Drum: [0, 1, 3, 2, 0]
- Cost minim folosind Căutare A*: 30
- Drum: [0, 1, 3, 2, 0]

6 Concluzie

În această lucrare, am implementat și evaluat trei algoritmi diferiți pentru a rezolva problema TSP: Căutare în lățime (BFS), Căutare de cost minim (UCS) și Căutare A*. Toți cei trei algoritmi au fost capabili să găsească soluția optimă în exemplele furnizate, iar costurile și drumurile găsite au fost aceleași pentru fiecare algoritm, deși timpul de execuție și resursele utilizate pot varia.

7 Referințe

1 Problema Comerciantului Călător, Wikipedia, `https://en.wikipedia.org/wiki/Travelling_salesman_problem`

2 A* Search Algorithm, Wikipedia, `https://en.wikipedia.org/wiki/A*_search_algorithm`