Travelling Salesman Problem - Metode clasice si simulated annealing

Prepared for: Camelia Chira, PhD
Prepared by: Razvan-Stefan Grecu
10 April 2016
Group: TI 30643

REFERAT

Descrierea problemei

Problema comis-voiajorului este o problema de optimizare. Presupunem ca suntem un vanzator ambulant, dintr-un oras oarecare si vrem sa vizitam nu numar de orase dat, si apoi sa ne intoarcem acasa. Se doreste ca efortul depus de comis voiajor sa fie minim. Costul trecerii de la un oras la altul va fi dat de distanta dintre orase (distanta euclidiana in 2D). Deci, orasele vor fi caracterizate de 2 coordonate (x, y). O ruta va avea o lista de orase, astfel incat, distanta parcursa total sa fie minima.

Pentru rezolvarea acestei probleme, vom presupune ca fiecare oras are o legatura cu un alt oras. Initial vom transforma problema dintr-una de minimizare, intr-una de maximizare.

Prezentarea algoritmului

Avand specificatia problemei, cerinta a fost implementarea unor algoritmi clasici pentru rezolvarea acestei probleme dar si utilizarea metodei de racire simulata. In continuare, vor fi prezentati toti algoritmii utilizati pentru rezolvarea problemei. Va fi prezentat pseudocodul si descrieri ale unor metode folosite. Data fiind natura functionala a unei parti vaste a codului, pseudocodul poate sa nu fie foarte diferit de implementarea propriu-zisa.

Cautare exhaustiva

Primul algoritm prezentat este cel de cautare exhaustiva in spatiul problemei. Dupa nume, putem deduce principiul din spatele sau: cautam in mod enumerativ fiecare stare posibila - in functie de reprezentare! - si alegem solutia cu valoarea maximala ce respecta constrangerile problemei. Reprezentarea unei solutii se face sub forma unei liste de orase, fiecare avand o coordonata x si y. Pseudocod:

```
generatePermutations().sort( cities1, cities2 ->

    Route r1 = createRouteFrom(cities1)

    Route r2 = createRouteFrom(cities2)

    r2 <=> r1 }.head()
```

Metoda ***generatePermutations()*** genereaza toate permutarile ale modelului (listei cu orase). S-a folosit metoda implicita din Groovy, aceasta dovedindu-se a fi eficienta si fiabila chiar si la liste mari. Implementarea este bazata pe un model din “Discrete mathematics and its applications” de K. Rosen.

Cautare greedy

In practica vom avea input-uri mai mari decat 25-30 de obiecte si avem nevoie de o metoda care sa se descurce intr-un timp, cel putin rezonabil, cu acele instante. Una din acele metode este greedy. Folosindu-ne de anumite cunostinte despre obiecte se pot construi metrici de utilizat pentru rezolvarea problemei. Metrica propusa este una clasica: vom porni de la un oras aleator, si vom cauta mereu orasul cu

distanta minima de la acesta. O posibilitate de imbunatatire ar fi sa facem media coordonatelor, si sa luam orasul cu coordonatele cele mai apropiate de medie. Pseudocod:

```
var startingCity = cities.get(randomIndex)
route.add(startingCity)
cities.remove(startingCity)
while (!cities.empty) {
    var minDistanceCity = cities.min { city -> startingCity.distance(city) }
    route.add(minDistanceCity)
    cities.remove(minDistanceCity)
    startingCity = minDistanceCity
```

Cautare locala

Aceasta metoda genereaza o solutie candidat, iar pentru un numar de iteratii dat, genereaza un vecin aleator al solutiei curente. Daca acel vecin este mai bun decat solutia in urma evaluarii functiei de fitness, vecinul devine solutia si algoritmul continua. Pseudocod:

```
for i in params.restarts {
    var candidate = initRoute(cities)
    for (j in params.iterations) {
        var firstIndex = next-random-int(city.firstIndex, city.lastIndex-1)
        var secondIndex = next-random-int(firstIndex, city.lastIndex)
        var neighbor = create2SwapRoute(candidate, firstIndex, lastIndex)
        if (neighbor.betterThan(candidate)) {
            candidate = neighbor
        }
    }
}
```

Metodele de generare a unui vecin sunt metode simple, ce folosesc un bazate pe 2-opt si 3-opt. 2-opt se bazeaza pe interschimbari, iar 3-8 pe rotatii si shiftari. Am creat o metoda de experiment, 4-opt, in care indecsii incrucisati se interschimba iar un ~~bloc de lungime egala cu minimul diferentei a doi indecsi, va fi mutat k spatii.~~

```
Route create2SwapRoute(start, i, j) {
        var cities = start.cities
        return cities.sublist(start.cities.firstIndex, i+1) + cities.sublist(i,j).reverse() +

                cities.sublist(j, start.cities.lastIndex)
}
Route create3ShiftRoute(start, index, length, shift) {
        return temp.subList(firstIndex, i) +
                temp.subList(i + length, i + length + shift) +
                temp.subList(i, i + length) +
                temp.subList(i + length + shift, lastIndex + 1)
}
Route createQuadRoute(start, i,j,k,t, shift) {
        if (i + 1 != k - 1) {
          Collections.swap(temp, i + 1, k - 1)
     }
        if (j + 1 != t - 1) {
        Collections.swap(temp, j + 1, t - 1)
    }
        var len = k - j
        apply 3 move on current city list
        return result
}
```

Local greedy search

Am inceput de la o ruta initiala, aleasa in mod arbitrat. Apoi, am luat pe rand fiecare posibilitate de indecsi pentru a aplica 2-opt-swap sau ~~3-opt-move~~, alegand orasul cu distanta minima la fiecare pas (sau maxima cu semn negativ). Pseudocod:

```
initRoute(route, tempCities, this.maxNumber)
        for (int count in (0..<params.iterations)) {
           int startIndex = route.cities.indices.first()
           int lastIndex = route.cities.indices.last()
           List<Route> moves = []
           for (int i = startIndex; i < lastIndex - 1; i++) {
              List<Route> swaps = []
              for (int j = i + 1; j < lastIndex; j++) {
                 def otherRoute = create2SwapRoute(route, i, j)
                 if (otherRoute.isBetter(route)) {
                    swaps.add(otherRoute)
                    route = otherRoute
                 }
              }
              def otherRoute = create3MoveRoute(route, i, 3)
              if (otherRoute.isBetter(swaps.empty ? route : swaps.max { it.totalCost })) {
                 moves.add(otherRoute)
                 route = otherRoute
              }
              routes += moves
              routes += swaps
           }
```

Simulated annealing

O cautare locala cu anumiti parametri noi, precum temperatura, temperatura minima, factorul de racire…Daca avem o solutie mai rea, in functie de temperatura curenta, o putem accepta sau nu.
Pseudocod:

```
        while (temperature > minTemperature) {
```

```
            while (t < params.iterations) {
                int firstIndex = randomGenerator.nextInt(candidate.cities.indices.first(),
candidate.cities.indices.last() - 1)
                int lastIndex = randomGenerator.nextInt(firstIndex + 1, candidate.cities.indices.last())
                Route neighbor = create2SwapRoute(candidate, firstIndex, lastIndex)
                if (neighbor.isBetter(candidate)) {
                    candidate = neighbor
                } else {
                    Double acceptanceProbability =
computeAcceptanceProbability(candidate.totalCostMinimum.doubleValue(),
                            neighbor.totalCostMinimum.doubleValue())
                    if (acceptanceProbability > randomGenerator.nextUniform(0.0, 1.0)) {
                        candidate = neighbor
                    }
                }
                t++
            }
            t = 0
            coolTemperatureLinear()
            routes.add(candidate)
        }
    }
```

Rezultate experimentale si comparatii

In continuare vor fi prezentate comparatii intre solutii, atat din punctul de vedere al rezultatelor obtinute dar si a timpului de rulare.

COMPARISON OF SEARCH RESULTS (COST)

CITY COUNT	EXHAUSTIVE	GREEDY	LOCAL 2-8	LOCAL 3-8	LOCAL Q-8	LOCAL G-8	SIM. ANNEALING	OPTIMUM
5	***487.874***	487.874	***487.874***	***487.874***	***487.874***	***487.874***	***487.874***	**487.874**
6	***299.827***	322.263	***299.827***	***299.827***	***299.827***	***299.827***	***299.827***	**299.827**
7	***271.751***	290.608	***271.751***	***271.751***	***271.751***	***271.751***	***271.751***	**271.751**
8	***1411.414***	1459.162	***1411.414***	***1411.414***	***1411.414***	***1411.414***	***1411.414***	**1411.414**
9	***1719.506***	1903.025	***1719.506***	1789.091	1721.641	***1719.506***	1721.641	**1719.506**
10	***1548.71***	1597.086	***1548.71***	1550.968	1575.085	***1548.71***	1575.085	**1548.71**
15	N/A	4063.898	***3837.44***	4719.901	3952.441	***3837.44***	3952.441	*?*
20	N/A	2680.817	***2183.932***	3475.636	*2332.237*	2228.928	2332.237	?
51	N/A	551.523	***451.842***	*872.692*	512.176	496.939	512.176	**426**
76	N/A	661.726	***573.098***	1575.722	717.646	675.413	717.646	**538**
100	N/A	25279.656	***22213.735***	92350.566	33791.410	26594.061	33791.410	**20749**
101	N/A	808.463	***692.11***	2076.557	960.547	799.468	960.547	**629**
130	N/A	8262.002	***6569.871***	29329.052	11295.712	8105.293	11295.712	**6110**

COMPARISON OF SEARCH RESULTS (TIME)							
CITY COUNT	EXHAUSTIVE	GREEDY	LOCAL 2-8	LOCAL 3-8	LOCAL Q-8	LOCAL G-8	SIM. ANNEALING
ITERARTIONS/RESTARTS			50000/25	50000/25	75000/20	NA/1000	10000/1000
5	159 ms	48 ms	1.56 min	1.05 min	2 min	2 s	36 s
6	269 ms	3 ms	2.19 min	1.25 min	2.76 min	1 s	46 s
7	1197 ms	30 ms	2.48 min	0.84 min	3.08 min	1.1 s	55 s
8	1645 ms	3 ms	2.73 min	0.93 min	3.65 min	1 s	58 s
9	8328 ms	5 ms	2.94 min	1.21 min	2.95 min	1.2 s	1 min
10	2.36 min	17 ms	2.61 min	1.10 min	4.10 min	1.4 s	44 s
15	N/A	10 ms	3.10 min	1.64 min	3.89 min	5.5 s	1.73 min
20	N/A	12 ms	3.58 min	2.29 min	3.63 min	23 s	2.78 min
51	N/A	57 ms	10.64 min	5.45 min	17.78 min	9.98 min	5.78 min
76	N/A	103 ms	24.46 min	14.88 min	19.50 min	0.85 h	12.47 min
100	N/A	176 ms	17.18 min	26.13 min	23.40 min	2.36 h	14.42 min
101	N/A	179 ms	55.87 min	26.16 min	23.91 min	2.39 h	13.21 min
130	N/A	205 ms	54.83 min	38.89 min	35.46 min	12.1 h	17.76 min