KARINCA KOLONİSİ OPTİMİZASYONU

Buse BALTACIOĞLU ~ 2019900540

20 01 2021

Contents

1 Karınca Kolonisi Optimizasyonu Tarihi

Karınca koloni optimizasyonu ilk olarak 1992 senesinde Marco Dorigo tarafından doktora tezi olarak önerilmiştir. İlk geliştirilen algoritma iki nokta arasındaki optimal yolu bulmayı hedefleyen sezgisel bir algoritmadır. Dorigo, ilk algoritmayı karınca sistemi olarak adlandırmıştır. 1999 yılında Hoos ve Stützle, algoritmayı geliştirmişler ve maksimum-minimum karınca sistemi adını vermişlerdir.

---

2 Karınca Kolonisi Algoritması Felsefesi

Birçok karınca türünde, gıda kaynağına gidip gelen karıncalar feromon adı verilen bir maddeyi yere bırakırlar. Diğer karıncalar feromonun varlığını algılar ve feromon yoğunluğunun daha yüksek olduğu yolları takip etme eğilimi gösterirler.

Karıncalar yüksek feromon seviyesine sahip yolu daha yüksek olasılıkla seçme eğilimi gösterdikleri için, zaman içerisinde yol başka karıncalar tarafından da kullanılmaya başlar. Bunun sonucunda yol üzerinde bulunan feromon seviyesi daha belirgin hale gelir ve karınca sürüsünün başarısı artar. Diğer bir ifadeyle, karıncalar hedeflerine ulaşmak için en verimli yolu bulmuş ve kullanmaya başlamış olurlar.

3 Karınca Kolonisi Algoritması

Adım 1. KKO parametrelerini belirle.

(İterasyon sayısı, karınca sayısı, başlangıç feromon seviyesi, rho, alpha, beta)

Adım 2. Tüm Karıncalar hedefe ulaşmadığı sürece aşağıdaki adımları tekrarla.

Adım 2.1. Olasılık dağılımını kullanarak çözüm oluştur.
Adım 2.2. Yerel feromonları güncelle.

Adım 3. En uygun yolun uzunluğunu hesapla ve yalnızca en uygun yolun feromon seviyesini güncelle.

Adım 4. Sona erme koşulları sağlanmıyorsa Adım 2'ye dön.

Adım 5. En yüksek feromon seviyesini içeren çözümü göster.

4 Karınca Kolonisi Algoritmasının İşleyişi

Feromon Seviyesinin Matematiksel Modeli

$$\Delta\tau_{i,j}^{k} = \begin{cases} \frac{1}{L_k} & k.karıncanın\ i\ ve\ j\ noktaları\ arasındaki\ yolu\ alma\ maliyeti \\ 0\ Diğer\ Durumlar \end{cases}$$

- Denklemde delta her bir karıncanın yola bıraktığını feromon miktarını temsil etmektedir. Karınca bir noktadan bir noktaya hareket ettiği için modelde bulunan ve j noktalar arasındaki mesafeyi temsil etmektedir.
- Karıncanın hareket etmemesi durumunda salgılanan feromon seviyesi 0'dır.
- 1/L ise iki nokta arasındaki uzaklığı ifade eder. Bu algoritmanın en kısa yolu aramasından kaynaklanır. Bu formül sayesinde daha kısa yol için daha fazla feromon seviyesi salgılanacaktır.

Birden fazla karınca olması durumunda noktalar arasındaki feromon seviyesi

$$\Delta\tau_{i,j}^{k} = \sum_{k=1}^{m} \Delta\tau_{i,j}^{k} \ (buharlaşma\ olmadan)$$

- Yukarıda denklemde m toplam karınca sayısını temsil etmektedir. Her bir karınca tarafından yüzeye salgılanan feromon seviyesi zaman içerisinde buharlaşacaktır. Yukarıdaki denklemde buharlaşma olmadan model kurulmuştur.

Modele buharlaşma ekleyerek, karıncalardan esinlendiğimiz bu algoritmayı daha iyi simüle edebiliriz

$$\Delta\tau_{i,j}^{k} = \underbrace{(1-\rho)\tau_{i,j}^{k}}_{1} \underbrace{\sum_{k=1}^{m} \Delta\tau_{i,j}^{k}}_{2} \ (buharlaşma\ ile\ birlikte)$$

- 1 numaralı kısım mevcut feromon seviyesini, 2 numaralı kısım ise diğer tüm karıncalar tarafından salgılanan feromon seviyesini temsil etmektedir.
- Modelde görüldüğü üzere Rho sabit bir sayıdır ve buharlaşma oranını temsil eder.
- Rho = 0 olduğu durumda herhangi bir buharlaşma söz konusu değildir.
- Rho = 1 olduğu durumda ise karıncanın bıraktığı tüm feromon buharlaşacaktır.

Olasılıkların hesaplanması

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\sum\left((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}\right)}, \quad \eta_{i,j} = \frac{1}{L_{i,j}}$$

$\eta\ (eta)$: *iki nokta arasındaki maliyet*
$\tau\ (tau)$: *iki nokta arasındaki fermon seviyesi*
$\alpha\ (alfa)$: *fermon seviyesinin formüle etkisini kontrol eder.*
$\beta\ (beta)$: *maliyetin formüle etkisini kontrol eder.*

- Formülde eta etkisi 0'a indirilerek yani beta = 1 alınarak olasılıklar yalnızca feromon seviyesine göre hesaplanabilir.

---

5 Karınca Kolonisi Optimizasyonun Avantajları

- Eşzamanlı olarak büyük bir popülasyon üzerinde arama yapılabilir.
- İyi çözümleri hızlı bir şekilde keşfedebilir.
- Yeni hedeflere hızlıca uyum sağlayabilir, dinamiktir.
- Hedefe yakınsama garantisi sunar.
- Gezgin satıcı problemi ve benzeri problemlerde etkili çözümler sunar

6 Karınca Kolonisi Optimizasyonun Dezavantajları

- Olasılık dağılımı her bir yineleme için değişebilir.
- Teorik olarak analizi zordur.
- Teorik araştırma yerine daha deneyseldir.
- Hedefe yakınsamanın ne zaman olacağı belirsizdir, çok fazla iterasyon gerekebilir.
- Rastgele karar dizilerinden oluşur.

---

7 Karınca Kolonisi Optimizasyonun Kullanıldığı Uygulamalar

- Bulut tabanlı sistemlerde görev zamanlaması ve yük dengelemesini optimal hala getirmek için kullanılabilir.
- Su dağıtım sistemleri malzeme, inşaat, bakım ve enerji gereksinimleri açısından maliyetleri en aza indirmek için kullanılabilir.
- Veri tabanlarında yapılan sorguların optimizasyonunda kullanılabilir.
- Sağlık alanında yapılan çalışmalarda virüsün modellenmesinde kullanılabilir. 2010 yılında yapılan bir çalışmada anti-HIV-1 aktivitelerinin tahmin edilmesinde kullanılmıştır.
- Gezgin satıcı problemlerinin çözümünde kullanılabilir.

---

8 Karınca Kolonisi Optimizasyonu ile Gezgin Satıcı Problemi

Bir şehir listesi ve her bir şehir çifti arasındaki mesafeler göz önüne alındığında, her şehri tam olarak bir kez ziyaret edip başlangıç şehrine geri dönen mümkün olan en kısa rota nedir?

```
library(knitr)
library(dplyr)
library(ggplot2)
```

Şehir sayısı

```
n<-10
```

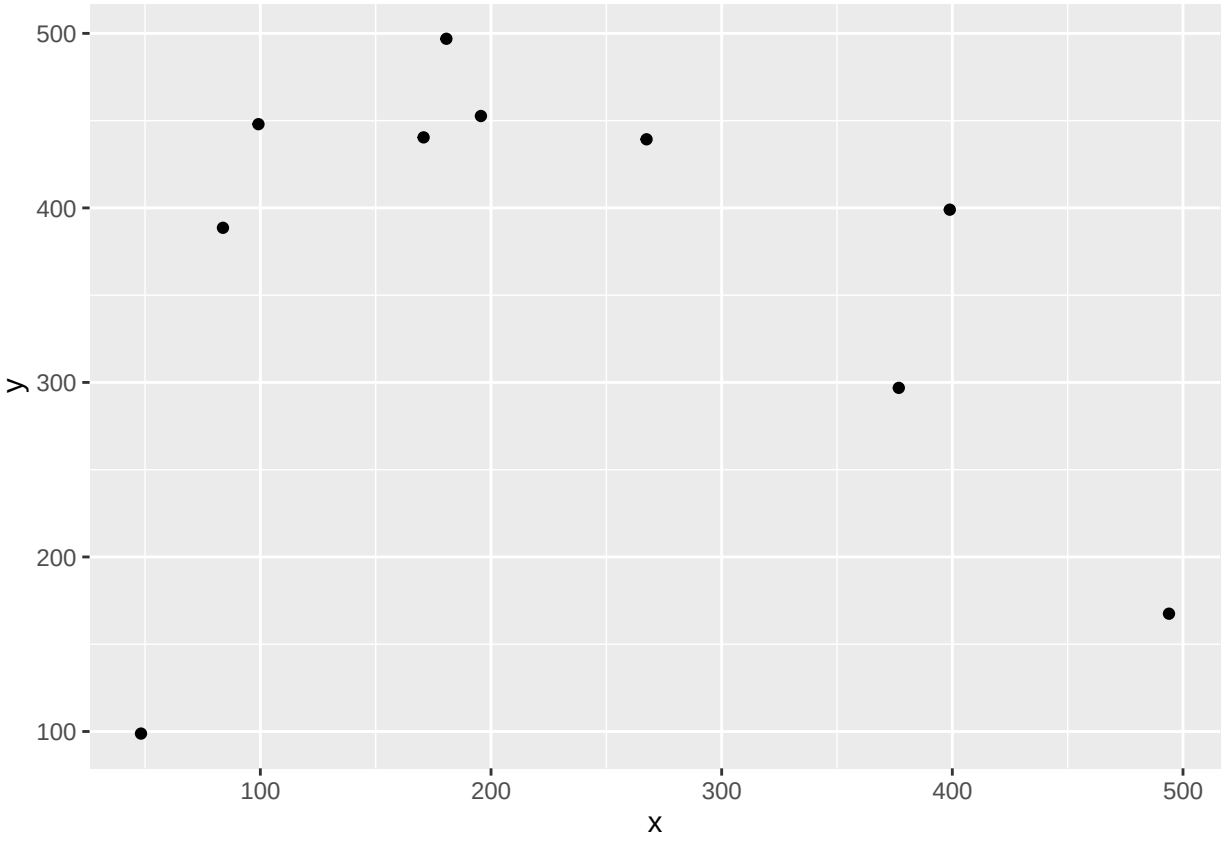
Öklid uzayının sınırı

```
max_x<-500
max_y<-500
```

Bazı rastgele şehirler

```
set.seed(123456)
cities <- data.frame(id = 1:n, x = runif(n, max = max_x), y = runif(n, max = max_y))
```

```
ggplot(cities, aes(x,y)) + geom_point()
```



Mesafe matrisi

```
distance <- as.matrix(stats::dist(select(cities, x, y), diag = TRUE, upper = TRUE))
dist_fun <- function(i, j) {
  vapply(seq_along(i), function(k) distance[i[k], j[k]], numeric(1L))
}
```

```
library(ompr)
```

```
model <- MIPModel() %>%
  # we create a variable that is 1 iff we travel from city i to j
  add_variable(x[i, j], i = 1:n, j = 1:n,
               type = "integer", lb = 0, ub = 1) %>%

  # a helper variable for the MTZ formulation of the tsp
  add_variable(u[i], i = 1:n, lb = 1, ub = n) %>%

  # minimize travel distance
  set_objective(sum_expr(dist_fun(i, j) * x[i, j], i = 1:n, j = 1:n), "min") %>%
```

```r
  # you cannot go to the same city
  set_bounds(x[i, i], ub = 0, i = 1:n) %>%

  # leave each city
  add_constraint(sum_expr(x[i, j], j = 1:n) == 1, i = 1:n) %>%
  #
  # visit each city
  add_constraint(sum_expr(x[i, j], i = 1:n) == 1, j = 1:n) %>%

  # ensure no subtours (arc constraints)
  add_constraint(u[i] >= 2, i = 2:n) %>%
  add_constraint(u[i] - u[j] + 1 <= (n - 1) * (1 - x[i, j]), i = 2:n, j = 2:n)
```

```r
model
```

```
## Mixed integer linear optimization problem
## Variables:
##   Continuous: 10
##   Integer: 100
##   Binary: 0
## Model sense: minimize
## Constraints: 110
```

```r
library(ompr.roi)
library(ROI.plugin.glpk)
```

```r
result <- solve_model(model, with_ROI(solver = "glpk", verbose = TRUE))
```

```
## <SOLVER MSG>  ----
## GLPK Simplex Optimizer, v4.47
## 110 rows, 110 columns, 434 non-zeros
##       0: obj =  0.000000000e+000  infeas = 2.900e+001 (20)
## *    33: obj =  2.101879949e+003  infeas = 0.000e+000 (1)
## *    57: obj =  1.450925576e+003  infeas = 0.000e+000 (1)
## OPTIMAL SOLUTION FOUND
## GLPK Integer Optimizer, v4.47
## 110 rows, 110 columns, 434 non-zeros
## 100 integer variables, 90 of which are binary
## Integer optimization begins...
## +    57: mip =     not found yet >=              -inf        (1; 0)
## +    83: >>>>>  1.469860058e+003 >=  1.450925576e+003   1.3% (3; 0)
## +   118: mip =  1.469860058e+003 >=     tree is empty   0.0% (0; 5)
## INTEGER OPTIMAL SOLUTION FOUND
## <!SOLVER MSG> ----
```

```r
result
```

```
## Status: optimal
## Objective value: 1469.86
```

```
head(result$solution, 10)
```

```
##  u[1]  u[2]  u[3]  u[4]  u[5]  u[6]  u[7]  u[8]  u[9] u[10]
##     1     2     9     7     8     6    10     4     3     5
```

Çözümü çıkarmak için get_solution, tidyverse paketlerle daha fazla kullanabileceğimiz bir data.frame döndüren yöntemi kullanabiliriz .

```
solution <- get_solution(result, x[i, j]) %>%
  filter(value > 0)
kable(head(solution, 3))
```

variable	i	j	value
x	7	1	1
x	1	2	1
x	5	3	1

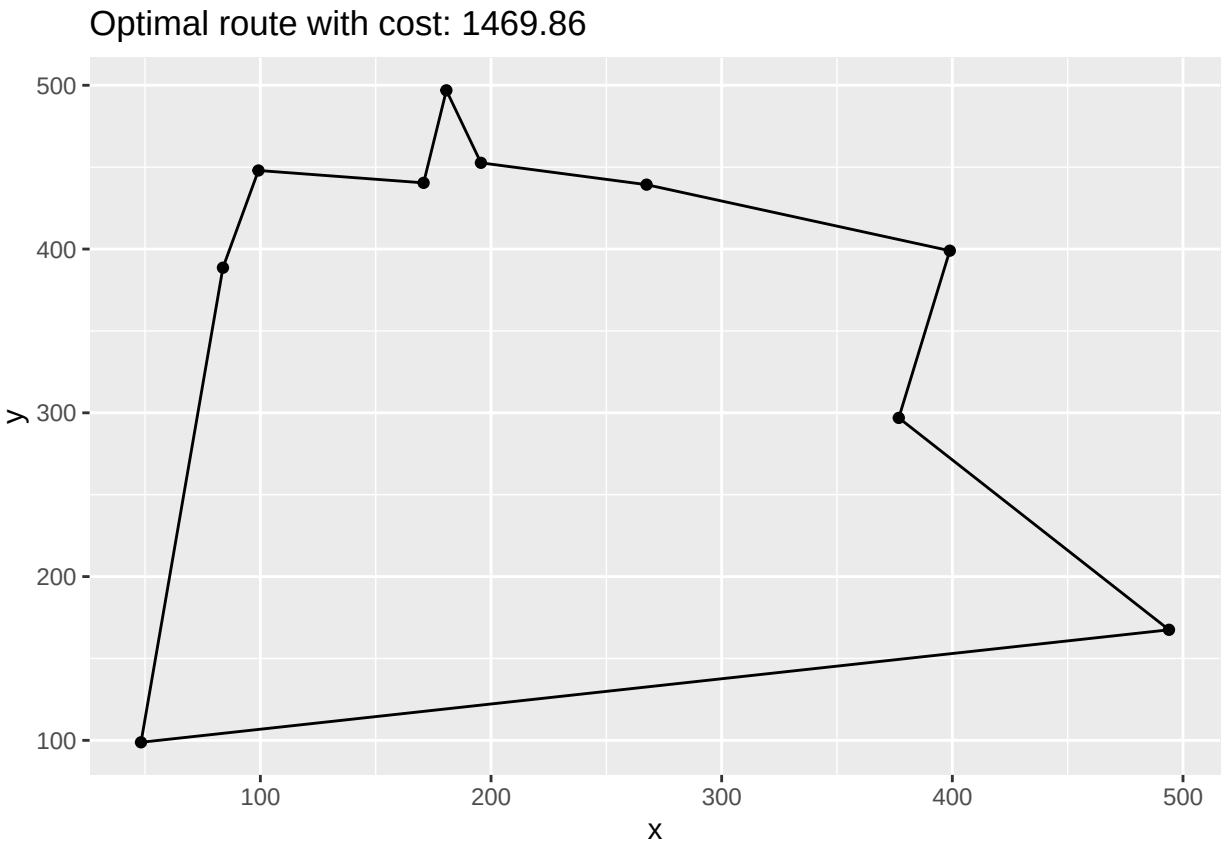
Şimdi modelimizdeki indeksleri gerçek şehirlerle tekrar ilişkilendirmemiz gerekiyor.

```
paths <- select(solution, i, j) %>%
  rename(from = i, to = j) %>%
  mutate(trip_id = row_number()) %>%
  tidyr::gather(property, idx_val, from:to) %>%
  mutate(idx_val = as.integer(idx_val)) %>%
  inner_join(cities, by = c("idx_val" = "id"))
```

```
kable(head(arrange(paths, trip_id), 4))
```

trip_id	property	idx_val	x	y
1	from	7	267.4290	439.3217
1	to	1	398.8922	398.9946
2	from	1	398.8922	398.9946
2	to	2	376.7825	296.8970

```
ggplot(cities, aes(x, y)) +
  geom_point() +
  geom_line(data = paths, aes(group = trip_id)) +
  ggtitle(paste0("Optimal route with cost: ", round(objective_value(result), 2)))
```



Gezgin satıcı problemimiz için ürettiğimiz şehirler arasında minimum yolu bulmayı hedeflemiştik ve optimal değerimizi 1469.86 bulmuş olduk.