Minimalno K-klasterovanje

(eng. *Minimum k-clustering*)
Projekat u okviru kursa Računarska inteligencija
Matematički fakultet

Igor Milošević, Nenad Perišić
igormilosevic96@gmail.com, perisicnenad96@gmail.com

22. maj 2019

Sadržaj

1 Uvod

1.1 Opis problema

Problem minimalnog k-klasterovanja se može opisati na sledeći način: ulazni podaci predstavljaju skup od n tačaka u ravni T = {t1, t2, ... tn} koje zadovoljavaju nejednakost trougla i k koji predstavja broj skupova (klastera) u koje treba rasporediti tih n tačaka. Klasteri predstavljaju disjunktnu uniju skupova K1 U K2 U ... U Kk (tj. presek svaka dva klastera je prazan skup), pri čemu je potrebno minimizovati maksimalno rastojanje između bilo koja dve tačke u jednom skupu. Na primer, ako je d1 maksimalno rastojanje prvog skupa, d2 maksimalno rastojanje drugog, ..., dk maksimalno rastojanje k-tog skupa vrednost d koju tražimo d = max{d1, d2, ... dk} treba da bude minimalna. Ovaj problem je NP-težak i za njegovo rešavanje smo koristili algoritam grube sile (eng. *brute force*) koji smo implementirali u programskom jeziku Python3 (eng. *Python3*) i algoritam simuliranog kaljenja kao optimizacionu tehniku implementiran u Javi (eng. *Java*).

1.2 Uopšteno o klasterovanju

Klasterovanje je tehnika istraživanja podataka koja detektuje objekte sličnih osobina i deli ih u grupe (klastere), čineći ih preglednijim i jasnijim za dalju analizu. Klaster analiza zapravo predstavlja pronalaženje grupa objekata takvih da su objekti u grupi medusobno slični, a objekti u različitim grupama medusobno raličiti.[1]

2 Opis rešenja problema

2.1 Algoritam grube sile

Dati problem ćemo prvo rešiti pomoću algoritma grube sile koji će nam kasnije koristiti za proveru ispravnosti rešenja za manji skup tačaka koje bude dao optimizacioni algoritam. U ovom algoritmu smo prvo nasumično odabrali određen broj tačaka i sačuvali u datoteku points.txt. Potom čitamo datoteku i te tačke čuvamo u listu tačaka.

```
    n = int(input("Enter the number of points to generate: "))
2   random_numbers = (random.sample(range(100), 2*n))
    pointsForFile = []
4   counter = 0
    numOfPointsToGenerate = n
6
    for x in range(numOfPointsToGenerate):
8       pointsForFile.append(str(random_numbers[counter]) + "," + str(random_numbers[counter
    +1]))
        counter = counter + 2
10
    i = 0
12  with open("points.txt", "w") as f1:
        for e in pointsForFile:
14          if i == n-1:
                f1.write(str(e))
16          else:
                f1.write(str(e) + "\n")
18          i+= 1
```

Program 1: nasumično biranje tačaka i smeštanje u datoteku

Nakon toga pravimo matricu rastojanja između svih tačaka i generišemo sve mogućnosti različitog particionisanja datog skupa tačaka.

```
    matrix = np.zeros((numOfPoints,numOfPoints))
    for i in range(numOfPoints):
        for j in range(numOfPoints):
            matrix[i][j] = distance(points[i], points[j])

    k = int(input("Unesite zeljeni broj klastera: \n"))
    disjointSets = list()

    for n, p in enumerate(partition(points), start=1):
        if len(p) != k:
            continue

        ind = 1
        for i in range(k):
            if len(p[i]) < 2:
                ind = 0
                break

        if  ind == 1:
            disjointSets.append(p)
```

Program 2: Generisanje matrice rastojanja i partitivnih skupova

Zatim smo prolazili kroz sve generisane particije i nalazili maksimalno rastojanje u svakoj i na kraju od svih izračunatih vrednosti uzeli najmanju.

```
    listOfMax = list()
    for e in disjointSets:
        listOfMaxDistInE = list()
        for s in e:
            maxDistInSet = 0
            distance1 = 0
            for i in range(len(s)):
                for j in range(i, len(s)):
                    distance1 = matrix[s[i].field_init][s[j].field_init]
                    if distance1 > maxDistInSet:
                        maxDistInSet = distance1
            listOfMaxDistInE.append(maxDistInSet)

        maxInE = max(listOfMaxDistInE)
        listOfMax.append(maxInE)

     print(min(listOfMax))
```

Program 3: Nalaženje minimalnog rastojanja

Kao što se može primetiti, u ovom koraku postoje četiri ugnježdene for petlje što nam ukazuje na veliku složenost algoritma ($O(n^4)$). Zbog velike složenosti, pri većem broju tačaka ovaj algoritam će biti praktično neupotrebljiv. Iz tog razloga prelazimo na optimizacionu tehniku simuliranog kaljenja (eng. *simulated annealing*).

2.2 Simulirano kaljenje

Na početku ovog algoritma se proizvoljno ili na neki drugi način generiše početno rešenje i izračuna vrednost njegove funkcije cilja. Vrednost najboljeg rešenja se najpre inicijalizuje na vrednost početnog. Zatim se algoritam ponavlja kroz nekoliko iteracija. U svakom koraku se razmatra rešenje u okolini trenutnog. Ukoliko je vrednost njegove funkcije cilja bolja od vrednosti funkcije cilja trenutnog rešenja, ažurira se trenutno rešenje. Ukoliko vrednost funkcije cilja novog rešenja nije bolja od vrednosti funkcije cilja trenutnog, upoređuje se vrednosti unapred definisane funkcije p i proizvoljno izabrane vrednosti q iz intervala (0, 1). Ako je p $>$ q, trenutno rešenje se ažurira novoizabranim. Takođe se, po potrebi, ažurira i vrednost najboljeg dostignutog rešenja. Algoritam se ponavlja dok nije ispunjen kriterijum zaustavljanja. Kriterijum

zaustavljanja može biti, na primer, dostignut maksimalan broj iteracija, dostignut maksimalan broj ponavljanja najboljeg rešenja, ukupno vreme izvršavanja, itd. Pod određenim uslovima, kod simuliranog kaljenja se može prihvatiti novo rešenje, čak i ako njegova vrednost nije bolja od vrednosti trenutnog. Na taj način se smanjuje verovatnoća konvergencije algoritma ka lokalnom optimumu koje nije i globalno.[3]

Algoritam simuliranog kaljenja je zasnovan na procesu kaljenja čelika, čiji je cilj oplemenjivanje metala tako da on postane čvršći. Prvi korak u kaljenju čelika je zagrevanje do određene temperature, a zatim, nakon kratkog zadržavanja na toj temperaturi, počinje postepeno hladenje. Pritom treba voditi o brzini hlađenja, jer brzo hladenje može da uzrokuje pucanje metala.[3]

Algoritam simuliranog kaljenja se ne pokazuje najbolje u svim situacijama. Na primer, za prost lokacijski problem koji pripada klasi NP-kompletnih problema se pokazuje da ovaj algoritam daje i višestruko sporije rešenje od ostalih algoritama zasnovanih na heuristikama, ali to ne utiče na konačno rešenje. [2]

Jedan primer kako radi algoritam simuliranog kaljenja:

Opis klase Point

```
  private int x;
  private int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
```

Atributi i konstruktor

Metode kojima ova klasa raspolaže su:

```
  public static Point createPoint(int x, int y) {
    return new Point(x,y);
  }

  public static List<Point> createPoints(long seed) {
    List<Point> points = new ArrayList<Point>(5);
    Random generator = new Random(seed);

    for(int i = 0; i < 12; i++) {
      int x = generator.nextInt(101);
      int y = generator.nextInt(101);
      points.add(createPoint(x,y));
    }
    return points;
  }
```

Program 3: Klasa Point i njene metode

Opis klase Cluster

```
  private List<Point> points;
  private int pointCount;
  private int clusterID;

  public Cluster(int c) {
    this.clusterID = c;
    this.points = new ArrayList<Point>();
    this.pointCount = 0;
  }
```

Atributi i konstruktor

Metode kojima ova klasa raspolaže su:

```
public void addPoint(Point x) {
  this.points.add(x);
  this.pointCount = this.pointCount + 1;
}

public static Cluster createCluster(int c) {
  return new Cluster(c);
}

public void removePoint(int x) {
  this.points.remove(x);
  this.pointCount = this.pointCount - 1;
}

public int getCount() {
  return this.pointCount;
}

public static double euclideanDistance(Point a, Point b) {
  return Math.sqrt(Math.pow(a.getCoordX() - b.getCoordX(), 2) +
  Math.pow(a.getCoordY() - b.getCoordY(), 2));
}

public double intraClusterDistance() {
  double max = 0;
  for(int i = 1; i < this.getCount(); i++) {
    for (int j = 1; j < this.getCount(); j++){
      Point point1 = (Point) this.getPoint(i - 1);
      Point point2 = (Point) this.getPoint(j);
      double max1 = euclideanDistance(point1, point2);
      if(max1 > max){
        max = max1;
      }
    }
  }
  return max;
}
```

Program 4: Klasa Cluster i njene metode

Metoda *intraClusterDistance*: za dati klaster prolazimo kroz sve njegove tačke i tražimo maksimalnu distancu u njemu. Za računanje distance između tačaka koristili smo euklidsko rastojanje.

Opis klase Main

Metode kojima ova klasa raspolaže su:

```
private static double acceptanceProbability(double neighborCost, double solutionCost,
   double temp) {
  double a = (solutionCost - neighborCost) / (temp);
  double ap = Math.exp(a);
  return ap;
}

private static void generateInitialSolution(List<Cluster> solution, List<Point> points,
   long seed, int x, int y) {
  Cluster cluster = (Cluster)solution.get(x);
  Point point = (Point)points.get(y);
  cluster.addPoint(point);
}

private static void generateNeighbor(List<Cluster> neighbor, List<Point> points, long seed,
    int x, int y, int yy) {
```

```
    Point point = (Point) points.get(x);
    int cluster = 0;
    int pointNum = 0;

    for(int i = 0; i < neighbor.size(); i++) {
      Cluster cluster1 = (Cluster) neighbor.get(i);

      for(int j = 0; j < cluster1.getCount(); j++) {
        Point point1 = (Point) cluster1.getPoint(j);
        if(point1.equals(point)) {
          cluster = i;
          pointNum = j;
        }
      }
    }

    Cluster cluster2 = (Cluster) neighbor.get(cluster);
    cluster2.removePoint(pointNum);

    Cluster cluster3 = (Cluster) neighbor.get(yy);
    cluster3.addPoint(point);
  }
```

Program 5: Klasa Main i njene metode

Metod *generateInitialSolution* na nasumičan način ravnomerno raspoređuje tačke po klasterima i na taj način se inicijalizuje početno rešenje problema. Metoda *generateNeighbor* omogućava traženje rešenja u okolini trenutnog, dok nam *acceptanceProbability* služi ukoliko vrednost funkcije cilja novog rešenja nije bolje od vrednosti funkcije cilja trenutnog da vidimo da li ipak treba da razmotrimo to rešenje (da bismo izbegli zaglavljivanje na lokalnom ekstremumu).

Bitnije koje smo koristili u Main-u:

```
    List<Point> points = new ArrayList<Point>();
    ArrayList<Cluster> solution = new ArrayList<Cluster>();
    ArrayList<Cluster> neighbor = new ArrayList<Cluster>();
    double temp = points.size()*40;
    double coolRate = 0.003;
    double solutionCost = 0;
    double neighborCost = solutionCost;
```

Prvo smo inicijalizovali početno rešenje i za njega izračunali *solutionCost*.

```
    for(int i = 0; i < size; i++) {
      x = generator.nextInt(numOfClusters);
      generateInitialSolution(solution, points, seed, x, i);
    }
    double max = 0;
    for (int i = 0; i < numOfClusters; i++) {
      double tmp = solution.get(i).intraClusterDistance();
      if (tmp >=max) {
        max = tmp;
      }
    }
    solutionCost = max;
```

Program 6: Inicijalizacija početnog rešenja

Sada sledi glavni deo koda u kome se primenjuje optimizacioni algoritam simuliranog kaljenja.

```
    while (temp > 1) {
      x = generator.nextInt(points.size());
      int y = generator.nextInt(numOfClusters);
      int yy = generator.nextInt(numOfClusters);

      while(yy == y)
        yy = generator.nextInt(numOfClusters);

```

```
      generateNeighbor(neighbor, points, seed, x, y, yy);
      max = 0;
      for (int i = 0; i < numOfClusters; i++) {
        double tmp = neighbor.get(i).intraClusterDistance();
        if (tmp >=max) {
          max = tmp;
        }
      }
      neighborCost = max;

      if(neighborCost < solutionCost) {
        solutionCost = neighborCost;
        solution.clear();
        solution = copyNeighbor(solution, neighbor);

      }

      if(neighborCost > solutionCost) {
        double acceptanceProbability = acceptanceProbability(neighborCost, solutionCost, temp
    );

        if( acceptanceProbability > 0.9999997) {
          solutionCost = neighborCost;
          solution.clear();
          solution = copyNeighbor(solution, neighbor);
        }
      }

      temp = temp - coolRate;
      numOfIterations++;
    }
```

Program 7: Simulirano kaljenje

U *while* petlji pozivom funkcije *generateNeighbor* razmatra se rešenje u okolini trenutnog. Računamo *intraClusterDistance* za datu okolinu i ukoliko je bolje od trenutnog ažuriramo trenutno rešenje datom okolinom, a ukoliko nije onda upoređujemo povratnu vrednost funkcije p = *acceptanceProbability* i proizvoljno izabrane vrednosti q iz intervala (0,1). Ako je p > q, trenutno rešenje se ažurira novoizabranim. Algoritam se ponavlja dok nije ispunjen zadat broj iteracija.

3 Rezultati

U ovoj sekciji analiziraćemo rezultate dobijene metodom grube sile i metodom simuliranog kaljenja. Takođe, razmotrićemo i performanse, odnosno brzinu izvršavanja jednog i drugog algoritma. Primetili smo da algoritam grube sile nije praktično primenljiv za više od 15 tačaka. Iz tog razloga, testirali smo rezultate na 12, 13 i 15 istih tačaka za 4 klastera kako bismo proverili ispravnost algoritma simuliranog kaljenja.

```
Solution cost: 40.52159917870962
Execution time: 17.201806783676147 seconds
```

Izlaz za algoritam grube sile za 12 tačaka i 4 klastera

```
Solution cost: 40.52159917870962
Execution time: 0.177 seconds
```

Izlaz za algoritam simuliranog kaljenja za 12 tačaka i 4 klastera

```
Solution cost: 43.32435804486894
Execution time: 82.74097323417664 seconds
```

Izlaz za algoritam grube sile za 13 tačaka i 4 klastera

```
Solution cost: 43.32435804486894
Execution time: 0.354 seconds
```

Izlaz za algoritam simuliranog kaljenja za 13 tačaka i 4 klastera

```
Solution cost: 41.036569057366385
Execution time: 2037.343002319336 seconds
```

Izlaz za algoritam grube sile za 15 tačaka i 4 klastera

```
Solution cost: 41.036569057366385
Execution time: 3.3230 seconds
```

Izlaz za algoritam simuliranog kaljenja za 15 tačaka i 4 klastera

Kao što možemo videti rezultati rešenja su identični, dok se vreme izvršavanja znatno razlikuje, to postaje jako upadljivo već sa 13 tačaka. Sa 15 tačaka se vidi da algoritam simuliranog kaljenja radi čak 617 puta brže, a daje identične rezultate. Naravno za veće instance očekuje se da simulirano kaljenje ne daje optimalna rešenja, ali se očekuje da budu približna optimalnom. Menjanjem semena (eng. *seed*) za nasumične vrednosti (što utiče na kreiranje inicijalnog rasporeda tačaka po klasterima) dobijali smo rezultate koji su bili približni rezultatima grube sile, najčešće se razlikovalo za 1 konačno rešenje, što i ne odskače mnogo od optimalnog rešenja. Ako nam je bitan kvalitet rešenja možemo povećati broj iteracija u algoritmu simuliranog kaljenja i u zavisnosti od tog povećanja i vreme izvršavanja će se povećati.

3.1 Specifikacija hardvera

Specifikacije računara nad kojim su izvršavani algoritmi:
CPU: Intel Core i3-3220 CPU 3.30GHz x 4
RAM: 11,7 GiB
OS: Ubuntu 18.04.2 LTS
IDE: Eclipse

4 Zaključak

U ovom radu, razmatrali smo rešavanje problema minimalnog k-klasterovanja pomoću algoritma grube sile i simuliranog kaljenja. Kao što smo videli, rešenja ovog problema daju slične ili identične rezultate za manje instance, dok se vreme izvršavanja znatno razlikuje. Problem minimalnog k-klasterovanja ima veliku primenu u oblastima istraživanja podataka (eng. *data mining*) tako da je preciznost rezultata od velikog značaja za njegov kvalitet, ali takođe bitna stavka je brzina izvršavanja za obimne podatke.

Jedna od mogućnosti unapređivanja algoritma je paralelizacija rada programa, koju planiramo da uradimo u bliskoj budućnosti.

Literatura

[1] C. C. Aggarwal. *Data Mining: The Textbook*. Springer International Publishing Switzerland, 2015.

[2] Jozef J. Kratica. *Paralelizacija genetskih algoritama za rešavanje nekih NP-kompletnih problema*. 2009.

[3] Stefan Mišković. Simulirano kaljenje, 2019. on-line at: http://poincare.matf.bg.ac.rs/~stefan/ri/sa_uflp.pdf.