KAUNO TECHNOLOGIJOS UNIVERSITETAS

ALGORITMŲ SUDARYMAS IR ANALIZĖ (P170B400)

Laboratorinis darbas Nr. 2

Užduoties variantas 15 – 3

Atliko: IF – 4/13 gr. Studentas Šarūnas Šarakojis

Priėmė: dėst: Vytautas Pilkauskas

KAUNAS

2016

Užduoties aprašymas

Bitoninė keliaujančio pirklio problema:

In the ***euclidean traveling-salesman problem***, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).
J. L. Bentley has suggested that we simplify the problem by restricting our attention to ***bitonic tours***, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible. Describe an O.n2/-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. (*Hint:* Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

Dinaminio programavimo eiga

Tarkime turime funkciją, kuri aprašoma taip: $b(i,j)$. Aprašytoji funkcija turi rasti trumpiausią bitoninį atstumą tarp dviejų mazgų $i\ ir\ j$. Galutinis atsakymas bus: $b(n,n)$.

Taip pat norint gauti reikiamą rezultatą visus turimus mazgus reikia surikiuoti pagal x koordinates didėjančiai. Taip pat egzistuoja bitoninė savybė, kuri teigia, kad: $b(i,j) = b(j,i)$. Tai reiškia, kad, rezultatas yra simetriškas jei eisime iš kitos pusės. Taip reikia apsibrėžti dar vieną funkciją, kuri apskaičiuoja atstumą tarp dviejų mazgų: $ats(i,j)$.

Uždavinio skaidymas į mažesnius uždavinius:

1) $Kai\ i = 1\ ir\ j = 2, tada\ b(i,j) = ats(1,2)$

Šiuo atveju dviejų gretimų taškų atstumas, pirmo ir antro mazgo, apskaičiuojamas pasinaudojus turima funkcija: $ats(i,j)$. Kadangi taškai yra surikiuoti pagal x koordinatę, taigi nėra jokio kito taško tarp pirmo ir antro mazgo, taigi šitas atstumas yra optimalus.

2) $Kai\ i < j-1, tada\ b(i,j) = b(i,j-1) + ats(j-1,j)$

Šiuo atveju reikia nukeliauti iki $j-1$ mazgo, tačiau nėra kol kas žinomo atstumo tarp $j-1\ ir\ j$ mazgų. Pasinaudojus funkcija: $ats(i,j)$, atstumą galime apskaičiuoti ir pridėti gautą rezultatą prie prieš tai apskaičiuoto.

3) $Kai\ i = j\ arba\ i = j-1, tada\ randame\ briauną\ su\ minimaliu\ atstumu\ k; 1 \leq k \leq i;\ ir\ apskaičiuojame: b(i,j) = b(k,i) + ats(k,j).$

Šiuo atveju reikia apskaičiuoti atstumą einant atgal, tačiau problema yra tame, kad galimas optimalus mazgas gali būti $nuo\ 1\ iki\ j-1$. Todėl reikia išrinkti mazgą iki kurio atstumas yra optimalus, tą mazgą pavadinkime k. Tuomet nuo gautojo mazgo galime apskaičiuoti atstumą iki esamos briaunos, tai yra: $ats(k,j)$ ir rezultatą pridėti prie anksčiau apskaičiuoto.

Programos išeities kodo įvertinimas

Žemiau pateikta, kaip parašytas programinis kodas buvo vykdomas, tai yra pirma surikiuojant visas koordinates, tada apskaičiuojant minimalų bitoninį atstumą tarp duotų taškų:

```
Collections.sort(points);
double bitonicPath = BitonicPathCalculator.calculateBitonicPath(points);
```

Žemiau pateikiamas metodo, kuris randa minimalų atstumą tarp esamų surikiuotų mazgų, išeities kodas su apskaičiuotais sudėtingumais:

```
public static double calculateBitonicPath(List<Point> points) {
    double[][] bitonicLengths = new double[points.size()][points.size()];    | 1
    bitonicLengths[0][0] = 0;                                                | 1
    bitonicLengths[0][1] = points.get(0).getDistanceToPoint(points.get(1)); | 1

    for (int j = 2, size = points.size(); j < size; j++) {                   | n -2
        bitonicLengths[0][j] = bitonicLengths[0][j-1]+points.get(j -
1).getDistanceToPoint(points.get(j));                                        | ∑_{j=2}^{n} t_j
    }

    for (int row = 1, size = points.size(); row < size; row++) {             | n -1
        for (int column = row; column < size; column++) {                    | ∑_{j=i}^{n} t_j
            bitonicLengths[row][column] = UNDEFINED;                         | ∑_{j=i}^{n} t_j

            if (row == column || row == column - 1) {       | ∑_{j=i}^{n}   * ∑_{j=2}^{n}(t_j − 1)
                double temp;
                double min = Double.MAX_VALUE;              | ∑_{j=i}^{n}   * ∑_{j=2}^{n}(t_j − 1)

                for (int counter = 0; counter < row; counter++) {∑_{j=i}^{n}   * ∑_{j=2}^{n}(t_j − 1)
                    temp = bitonicLengths[counter][row] +
```

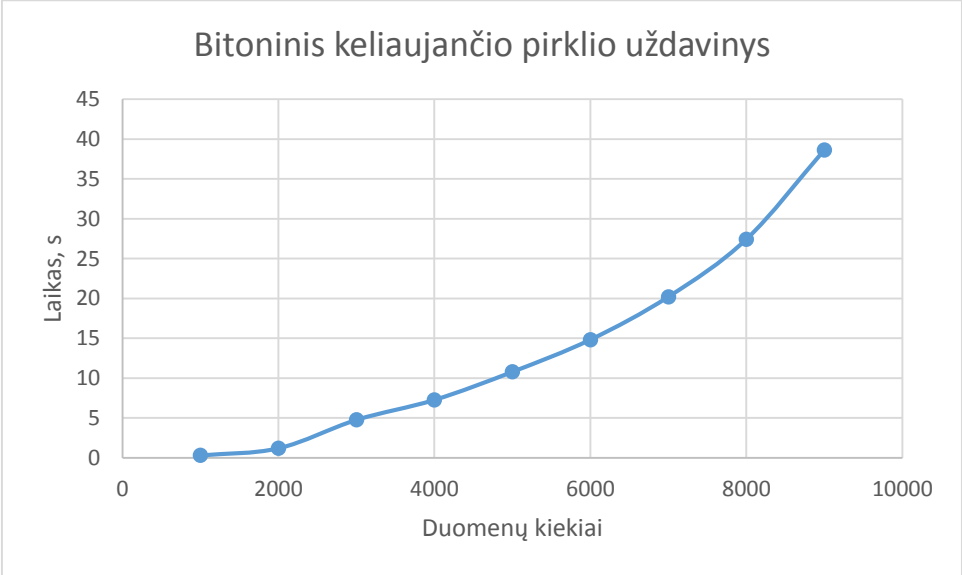
```
points.get(counter).getDistanceToPoint(points.get(column)); | ∑_{j=i}^{n}   * ∑_{j=2}^{n}(t_j − 1)
                    min = temp < min ? temp : min;          | ∑_{j=i}^{n}   * ∑_{j=2}^{n}(t_j − 1)
                }

                bitonicLengths[row][column] = min;          | ∑_{j=i}^{n}   * ∑_{j=2}^{n}(t_j − 1)
            } else {
                bitonicLengths[row][column] = bitonicLengths[row][column - 1] +
                        points.get(column -
1).getDistanceToPoint(points.get(column)); ∑_{j=i}^{n}   * ∑_{j=2}^{n}(t_j − 1)
            }
        }
    }

    return bitonicLengths[points.size() - 1][points.size() - 1];            | 1
}
```

Apskaičiuotas sudėtingumas: $3 * 1 + (n-2) + 3 * \sum_{j=i}^{n} t_j + 6 * \sum_{j=i}^{n} * \sum_{j=2}^{n}(t_j - 1) = O(n^2)$

Greitaveikos eksperimentai

Žemiau pateikiama rezultatų lentelė ir grafikas, kuriose aiškiai matomi tikslūs algoritmo veikimo laikai su tam tikromis duomenų apimtimis (duomenys buvo automatiškai sugeneruojami):

Kiekis	Laikas, s
1000	0.304
2000	1.201
3000	4.764
4000	7.271
5000	10.792
6000	14.826
7000	20.202
8000	27.413
9000	38.61



Išvados

Apskritai keliaujančio pirklio problemą spręsti galima naudojantis daugeliu būdų, tačiau sprendimas naudojant bitoninius maršrutus ir dinaminį programavimą (griežtai iš kairės į dešinę ir po to griežtai iš dešinės į kairę) galima gauti optimalų (good enough) ir greitą rezultatą. Sprendžiant šį uždavinį dinaminio programavimo metodu, susidarant bent 3 smulkesnius uždavinius ir apskaičiavus didesnio uždavinio atsakymą pasinaudojus smulkesnių uždaviniu atsakymais buvo gautas pakankamai greitas algoritmo veikimo laikas. Tačiau dinaminio programavimo metodo netaupoma atmintis, čia yra vienas iš svarbesnių aspektų, jog naudojant dinaminį programavimą išlošiama laiko, tačiau prarandama atminties (performance over memory).