Dinamičko programiranje

Strukture podataka i algoritmi 2

Algoritamski postupak

- Algoritamski postupak – ideja ili šablon primenljiv na širi spektar algoritamskih problema
- Dinamičko programiranje je algoritamski postupak primenjiv na široku klasu problema optimizacije
- Tehnika koju je osmislio Richard E. Bellman 40-ih godina prošlog veka
  - Reč programiranje u sintagmi “dinamičko programiranje” ne označava **pisanje izvornog koda računarskih programa** nego **tabelarni pristup rešavanju optimizacionih problema**
  - Reč dinamičko u sintagmi “dinamičko programiranje” ne označava **rad sa dinamičkim strukturama podataka**, nego nekakvu **promenu**
  - **dinamičko programiranje = popunjavanje tabele u procesu rešavanja optimizacionog problema**

Belmanov princip optimalnosti

- Tehnikom dinamičkog programiranja možemo rešavati probleme koji imaju **optimalnu strukturu,** odnosno zadovoljavaju **Belmanov princip optimalnosti**
- Problem P ima optimalnu strukturu ako se optimalno rešenje P može dobiti iz optimalnih rešenja **manjih instanci** problema P
  - instance istog problema koje su manje dimenzije
- Neka je $P(a_1, a_2, ..., a_n)$ optimizacioni problem, gde su a_1 ... a_n parametri (prirodni brojevi) koji odslikavaju veličinu (dimenziju) problema
  - Problem ruksaka: P(broj predmeta, kapacitet ranca)
- Problem $P(b_1, b_2, ..., b_n)$ je manja instanca problema $P(a_1, a_2, ..., a_n)$ ako važi

$$b_1 <= a_1,\ b_2 <= a_2,\ \ldots,\ \mathbf{b_i < a_i},\ \ldots,\ \mathbf{b_n < a_n}$$

Dinamičko programiranje

- Dinamičko programiranje = popunjavanje tabele optimalnih rešenja **svih** instanci problema $P(a_1, a_2, \ldots, a_n)$
  - Tabela optimalnih rešenja: *n*-dimenzionalni niz veličine $(a_1 + 1) \times (a_2 + 1) \times \ldots \times (a_n + 1)$
- Optimalno rešenje P dobijamo iz optimalnih rešenja manjih instanci P

```
for (int i1 = 0; i1 <= a1; i1++)
  for (int i2 = 0; i2 <= a2; i2++)
   …
     for (int in = 0; in <= an; in++)
       T[i1][i2]…[in] = optimalno rešenje P(i1, i2, ..., in)
```

Optimalno rešenje P(a1, ..., an) je u T[a1][a2]…[an]

Dinamičko programiranje

- Optimalno rešenje P dobijamo iz optimalnih rešenja manjih instanci P
  - Moramo postaviti relaciju između optimalnog rešenja P i optimalnih rešenja manjih instanci P → dobijamo neku rekuretnu relaciju
  - Stoga moramo identifikovati i **trivijalne instance problema P**

- Trivijalne instance problema ruksaka
  - Imamo samo jedan predmet za ruksak proizvoljnog kapaciteta
  - Imamo proizvoljan broj predmeta za ruksak kapaciteta 0

Problem ruksaka

- MP – tabela maksimalnog profita svih instanci problema ruksaka
  - MP[*i*][*j*] – maksimalni profit koji se može ostvariti selekcijom prvih *i* predmeta niza *S* u ranac kapaciteta *j*
  - MP je matrica celih brojeva dimenzija n X (W + 1)
  - **Maksimalni profit je u poslednjem elemetu matrice, MP[n-1][W]**
- Trivijalna instanca 1: ruksak kapaciteta 0
  - MP[i][0] = 0 za svako *i* u [0 .. n – 1]
- Trivijalna instanca 2: jedan (prvi) predmet za ruksak proizvoljnog kapaciteta
  - MP[0][j] = 0, ako je S[0].w > j (predmet ne može da stane)
  - MP[0][j] = S[0].p, ako je S[0].w <= j (predmet može da stane)

Ne-trivijalne instance

- Za MP[i][j] imamo dve mogućnosti
  - S[i].w > j: i-ti predmet ne možemo dodati u ruksak kapaciteta j → MP[i][j] = MP[i – 1][j]
  - S[i].w <= j: i-ti predmet možemo dodati u ruksak kapaciteta j → MP[i][j] = max{Add, Skip}
    - Add = S[i].p + MP[i -1][j – S[i].w]
      i-ti predmet dodajemo u ruksak
    - Skip = MP[i -1][j]
      i-ti predmet ne dodajemo ruksak

```java
public class Item  {

	private int weight;
	private int profit;

	public Item(int weight, int profit) {
		this.weight = weight;
		this.profit = profit;
	}

	public int getWeight() {
		return weight;
	}

	public int getProfit() {
		return profit;
	}

	public String toString() {
		return "(" + weight + ", " + profit + ")";
	}
}
```

Glavna klasa

```
public class KnapsackDP {

	private int knapsackWeight;
	private Item[] items;
	private int[][] mp; // tabela maksimalnog profita

	public KnapsackDP(int knapsackWeight, Item[] items) {
		this.knapsackWeight = knapsackWeight;
		this.items = items;
		mp = new int[items.length][knapsackWeight + 1];
	}

	//... to be continued
}
```

Trivijalne instance problema

```
public int findMaxProfit() {
	// trivijalna instanca (1) - ruksak kapaciteta 0
	for (int i = 0; i < items.length; i++) {
		mp[i][0] = 0;
	}

	// trivijalna instanca (2) - imamo samo jedan (prvi) predmet
	for (int j = 0; j <= knapsackWeight; j++) {
		// da li prvi element moze da stane u ruksak kapaciteta j?
		if (items[0].getWeight() <= j) {
			mp[0][j] = items[0].getProfit();
		} else {
			mp[0][j] = 0;
		}
	}

	//... to be continued
}
```

Netrivijalne instance problema

```
public int findMaxProfit() {
      // trivijalne instance -- vidi prethodni slajd

      // ne-trivijalne instance
      for (int i = 1; i < items.length; i++) {
            for (int j = 1; j <= knapsackWeight; j++) {
                  // koliki je profit kada i-ti predmet
                  // dodamo u ruksak kapaciteta j?
                  int profitAdd = 0;
                  if (items[i].getWeight() <= j) {
                        profitAdd = items[i].getProfit() +
                                    mp[i - 1][j - items[i].getWeight()];
                  }

                  // koliki je profit kada i-ti predmet preskocimo?
                  int profitSkip = mp[i - 1][j];

                  mp[i][j] = Math.max(profitAdd, profitSkip);
            }
      }

      return mp[items.length - 1][knapsackWeight];
}
```

Rekonstrukcija strukture optimalnog rešenja

- Koji predmeti su ubačeni u ruksak?
- Odgovor na prethodno pitanje možemo dobiti inspekcijom matrice profita.
  - **Ako je MP[i, j] = MP[i – 1, j] tada i-ti predmet nije ubačen u ruksak kapaciteta j**

#	W	P
0	7	28
1	6	22
2	5	18
3	2	6
4	1	1

i\W	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	28	28	28	28	28
1	0	0	0	0	0	0	22	28	28	28	28	28
2	0	0	0	0	0	18	22	28	28	28	28	40
3	0	0	6	6	6	18	22	28	28	34	34	**40**
4	0	1	6	7	7	18	22	28	29	34	35	**40**

Predmet 4. nije ubačen u ruksak: MP[4, 11] = MP[3, 11]

Rekonstrukcija strukture optimalnog rešenja

- Koji predmeti su ubačeni u ruksak?

#	W	P
0	7	28
1	6	22
2	5	18
3	2	6
4	1	1

i\W	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	28	28	28	28	28
1	0	0	0	0	0	0	22	28	28	28	28	28
2	0	0	0	0	0	18	22	28	28	28	28	**40**
3	0	0	6	6	6	18	22	28	28	34	34	**40**
4	0	1	6	7	7	18	22	28	29	34	35	**40**

Predmet 3. nije ubačen u ruksak: MP[3, 11] = MP[2, 11]

Rekonstrukcija strukture optimalnog rešenja

- Koji predmeti su ubačeni u ruksak?

#	W	P
0	7	28
1	6	22
2	5	18
3	2	6
4	1	1

i\W	0	1	2	3	4	5	6	7	8	9	10	11
0	**0**	0	0	0	0	0	**0**	28	28	28	28	28
1	0	0	0	0	0	0	**22**	28	28	28	28	**28**
2	0	0	0	0	0	18	22	28	28	28	28	**40**
3	0	0	6	6	6	18	22	28	28	34	34	**40**
4	0	1	6	7	7	18	22	28	29	34	35	**40**

Predmet 2. **jeste** ubačen u ruksak: MP[2, 11] != MP[1, 11]

→ MP[2, 11] sam dobio iz MP[1, 6] (6 = 11 – w[2])

Predmet 1. **jeste** ubačen u ruksak: MP[1, 6] != MP[0, 6]

→ MP[1, 6] sam dobio iz MP[0, 0] (0 = 6 – w[1])

Rekonstrukcija strukture optimalnog rešenja

- **Sumirano:**
  - Neka je (i, j) trenutna instanca problema čiju strukturu rekonstruišemo.
    - Na početku (i, j) = (n – 1, W)
  - MP[i, j] = MP[i – 1, j]
    - Nisam dodao i-ti predmet u ruksak
    - (i, j) → (i – 1, j)
  - MP[i, j] != MP[i – 1, j]
    - Dodao sam i-ti predmet u ruksak
    - (i, j) → (i – 1, j – w[i])
  - (i, j) = (0, 0) → kraj rekonstrukcije

Rekonstrukcija optimalnog rešenja

```
public LinkedList<Item> getAddedItems() {
    LinkedList<Item> l = new LinkedList<Item>();
    int i = items.length - 1;
    int j = knapsackWeight;

    while (i > 0) {
        if (mp[i][j] != mp[i - 1][j]) {
            l.addFirst(items[i]);
            j -= items[i].getWeight();
        }
        i--;
    }

    if (j > 0 && items[0].getWeight() <= j)
        l.addFirst(items[0]);

    return l;
}
```

Primer korišćenja klase

```
public static void main(String[] args) {
      int[] weight = {7, 6, 5, 2, 1};
      int[] profit = {28, 22, 18, 6, 1};

      Item[] items = new Item[weight.length];
      for (int i = 0; i < items.length; i++) {
            items[i] = new Item(weight[i], profit[i]);
      }

      KnapsackDP k = new KnapsackDP(11, items);
      System.out.println("Maks profit = " + k.findMaxProfit());
      LinkedList<Item> addedItems = k.getAddedItems();
      for (int i = 0; i < addedItems.size(); i++) {
            System.out.println(addedItems.get(i));
      }
}
```

String distance

- U procesorima za editovanje teksta i sistemima za pretraživanje često se dešava da korisnik pogreši prilikom unosa
  - Spell check opcije u procesorima teksta
  - "Did you mean?" kod Google pretraživača
- Sistemi sa mogućnostima korekcije korisničkog ulaza poseduju rečnik – skup stringova koji su validne reči u nekom jeziku.
- Za svaki token iz korisničkog unosa se proverava da li postoji u rečniku.
- Ako ne postoji tada se korisniku sugeriše *k* **najsličnijih (najbližih)** reči iz rečnika.
- **Kako meriti sličnost (blizinu, rastojanje, distancu) između dva stringa?**

String distanca

- String distanca je funkcija **D(s1, s2), D: String X String → Real** za koju važe sledeći uslovi:
  - D(S1, S2) = 0 → s1 i s2 su identični
  - D(S1, S2) = D(S2, S1)
  - D(S1, S2) < D(S1, S3) → string S1 je sličniji stringu S2 nego što je sličniji stringu S3
- String distance možemo klasifikovati u
  - **edit distance:** računamo stepen transformacije jednog stringa u drugi string
    - Veći stepen transformacije → veća distanca (manja sličnost)
  - **set distance:** računamo stepen preklapanja stringova
    - Veći stepen preklapanja → manja distanca (veća sličnost)

n-gram distance

- n-gram distance su klase jednostavnih set string distanci
- Od stringa formiramo skup **N-grama**
- N-gram: n susednih karaktera u stringu
  - S = “algoritam”
  - 3-gram = {“alg”, “lgo”, “gor”, “ori”, “rit”, “ita”, “tam”}
- Sličnost dva stringa sada možemo meriti kao stepen preklapanja njihovih n-gram skupova za neko fiksirano n.
- Jaccardova n-gram distanca

$$J(S_1, S_2) = 1 - \frac{|Ngram(S_1) \cap Ngram(S_2)|}{|Ngram(S_1) \cup Ngram(S_2)|}$$

Edit string distanca

- Najjednostavnija edit string distanca meri broj elementarnih transformacija koje su potrebne kako bi se jedan string transformisao u drugi

- Elementarne transformacije
  - Dodavanje karaktera
  - Brisanje karaktera
  - Supstitucija (zamena) karaktera nekim drugim karakterom

- Računanje edit distance je optimizacioni problem: tražimo minimalni broj elementarnih transformacija

Primer

- edit("petar", "patrik") = 4
- Možemo imati više najkraćih elementarnih transformacija
- Prvi način transformacije
  - p**e**tar → p**a**tar (supstitucija e u a)
  - patar → pat**r**ar (dodavanje r)
  - patr**a**r → patr**i**r (supstitucija a u i)
  - patri**r** → patri**k** (supstitucija r u k)
- Drugi način transformacije
  - p**e**tar → p**a**tar (supstitucija e u a)
  - pat**a**r → patr (brisanje a)
  - patr → patr**i** (dodavanje i)
  - patri → patri**k** (dodavanje k)

DP pristup edit distanci

- Neka su A i B dva stringa reprezentovani nizom karaktera indeksiranim od 1: $A = a_1a_2\ldots a_n$, $B = b_1b_2\ldots b_m$
- String A je prazan ako je n = 0 (m = 0 za B)
  - n = 0 → edit(A, B) = m (imamo m dodavanja karaktera)
  - m = 0 → edit(A, B) = n (imamo n brisanja karaktera)
- Ako je $a_n = b_m$ tada je edit(A, B) = edit($a_1a_2\ldots a_{n-1}$, $b_1b_2\ldots b_{m-1}$)
- Ako je a_n != b_m tada imamo jednu od tri moguće transformacije
  - Supstitucija a_n u b_m : editS(A, B) = 1 + edit($a_1a_2\ldots a_{n-1}$, $b_1b_2\ldots b_{m-1}$)
  - Brisanje a_n : editD(A, B) = 1 + edit($a_1a_2\ldots a_{n-1}$, $b_1b_2\ldots b_m$)
  - Dodavanje b_m : editA(A, B) = 1 + edit($a_1a_2\ldots a_n$, $b_1b_2\ldots b_{m-1}$)
  - Biramo transformaciju koja ima minimalni edit:

    edit(A, B) = min{editS, editD, editA}

DP pristup edit distanci

- Neka su A i B dva stringa reprezentovani nizom karaktera indeksiranim od 1: $A = a_1a_2\ldots a_{n,}$ $B = b_1b_2\ldots b_m$
- Definišemo matricu distanci d[0 .. n, 0 .. m]
  - d[i, j] = edit distanca za stringove $(a_1a_2\ldots a_i)$ i $(b_1b_2\ldots b_j)$

a\b	**#**	**p**	**e**	**t**	**a**	**r**
#	0	1	2	3	4	5
p	1					
a	2					
t	3					
r	4			d[i][j]		
i	5					
k	6					**EDITD**

a\b	#	p	e	t	a	r
#	0	1	2	3	4	5
p	1					
a	2					
t	3		d[i-1][j-1]	d[i-1][j]		
r	4		d[i][j-1]	d[i][j]		
i	5					
k	6					**EDITD**

- d[i][j] = edit("patr", "pet")
- Supstitucija "r" u "t"

  d[i][j] = 1 + edit("pat", "pe") = 1 + d[i – 1, j – 1]
- Brisanje "r" iz "patr"

  d[i][j] = 1 + edit("pat", "pet") = 1 + d[i – 1][j]
- Dodavanje "t" u "patr"

  d[i][j] = 1 + edit("patr", "pe") = 1 + d[i][j – 1]

a\b	#	p	e	t	a	r
#	0	1	2	3	4	5
p	1					
a	2					
t	3				d[i-1][j-1]	d[i-1][j]
r	4				d[i][j-1]	d[i][j]
i	5					
k	6					**EDITD**

- d[i][j] = edit("patr", "petar")
- d[i][j] = edit("pat", "peta") = d[i – 1][j - 1]

a\b	#	p	e	t	a	r
#	0	1	2	3	4	5
p	1					
a	2					
t	3		d[i-1][j-1]	d[i-1][j]		
r	4		d[i][j-1]	d[i][j]		
i	5					
k	6					**EDITD**

- Trivijalne instance:
  - d[i][j] = i za j = 0
  - d[i][j] = j za i = 0
- Netrivijalne instance
  - d[i][j] = d[i-1][j-1], ako je A[i] = B[j]
  - d[i][j] = 1 + min{d[i-1][j-1], d[i][j-1], d[i – 1][j]}, inače

Rekonstrukcija strukture rešenja

- Tražimo jedan minimalni put od (n, m) do (0, 0)

	#	p	e	t	a	r
#	**0**	1	2	3	4	5
p	1	**0**	1	2	3	4
a	2	1	**1**	2	2	3
t	3	2	2	**1**	2	3
r	4	3	3	**2**	2	2
i	5	4	4	3	**3**	3
k	6	5	5	4	4	**4**

- 4 → 3 (supstitucija “k” u “r”)
- 3 → 2 (supstitucija “i” u “a”)
- 2 → 1 (brisanje “r”)
- 1 → 1 (ništa, nije bilo transformacije)
- 1 → 0 (supstitucija “a” u “e”)

Rekonstrukcija rešenja

- i = m, j = n
- Poruke o transformacijama treba štampati unatraške → koristimo stek
- Za trenutno (i, j) nađemo minumum od S = d[i – 1][j – 1], D = d[i – 1][j] i A = d[i][j – 1]
- min = S i d[i][j] != d[i - 1][j - 1]
  - **push** supstitucija A[i] u B[j]; (i, j) → (i – 1, j – 1)
- min = D
  - **push** obrisan je karakter A[i]; (i, j) → (i – 1, j)
- min = A
  - **push** dodat je karakter B[j]; (i, j) → (i, j – 1)
- Ponavljam prethodni korak dokle god d[i][j] > 0
- Isprazni stek sa porukama

```java
public class EditDistance {
	private String s1, s2;  // ulazni stringovi
	private int[][] d;      // matrica distanci

	public EditDistance(String s1, String s2) {
		this.s1 = s1;
		this.s2 = s2;
		d = new int[s1.length() + 1][s2.length() + 1];
		computeDistance();
	}

	private void computeDistance() {
		for (int i = 0; i <= s1.length(); i++) d[i][0] = i;
		for (int i = 0; i <= s2.length(); i++) d[0][i] = i;

		for (int i = 1; i <= s1.length(); i++) {
			for (int j = 1; j <= s2.length(); j++) {
				char c1 = s1.charAt(i - 1);
				char c2 = s2.charAt(j - 1);
				if (c1 == c2) {
					d[i][j] = d[i - 1][j - 1];
				} else {
					int dDel = d[i - 1][j];
					int dAdd = d[i][j - 1];
					int dSub = d[i - 1][j - 1];
					d[i][j] = 1 + min3(dDel, dAdd, dSub);
				}
			}
		}
	}
}
```

```
private int min3(int a, int b, int c) {
        int min = a;
        if (b < min) min = b;
        if (c < min) min = c;
        return min;
}

public int getDistance() {
        return d[s1.length()][s2.length()];
}

public void printDistanceMatrix() {
        System.out.print("     ");
        for (int j = 0; j < s2.length(); j++) {
                System.out.print(s2.charAt(j) + "  ");
        }
        System.out.println();

        for (int i = 0; i <= s1.length(); i++) {
                if (i > 0)
                        System.out.print(s1.charAt(i - 1) + "  ");
                else
                        System.out.print("   ");
                for (int j = 0; j <= s2.length(); j++) {
                        System.out.print(d[i][j] + "  ");
                }
                System.out.println();
        }
}
```

```java
public void getExplanation() {
	int i = s1.length();
	int j = s2.length();
	Stack<String> messages = new Stack<String>();
	int[] di = {-1, -1, 0};
	int[] dj = {-1, 0, -1};

	while (d[i][j] > 0) {
		int min = Integer.MAX_VALUE;
		int minIndex = -1;

		for (int k = 0; k < di.length; k++) {
			if (i + di[k] >= 0 && j + dj[k] >= 0) {
				if (d[i + di[k]][j + dj[k]] < min) {
					min = d[i + di[k]][j + dj[k]];
					minIndex = k;
				}
			}
		}

		// to be continued
```

```java
            if (minIndex == 0) {
                if (d[i][j] != min) {
                    messages.push(s1.charAt(i - 1) + " --> " +
                                s2.charAt(j - 1));
                }
                i--; j--;
            } else if (minIndex == 1) {
                i--; messages.push(s1.charAt(i) + " deleted");
            } else  {
                j--; messages.push(s2.charAt(j) + " inserted");
            }
        } // END OF while(d[i][j] > 0)

        if (messages.isEmpty()) {
            System.out.println("Identical strings... ");
        } else {
            System.out.println("Transformations: ");
            while (!messages.isEmpty()) {
                System.out.println(messages.pop());
            }
        }
} // END OF getExplanation()
```

```java
public static void main(String[] args) {
      Scanner reader = new Scanner(System.in);
      System.out.println("First string: ");
      String s1 = reader.nextLine();
      System.out.println("Second string: ");
      String s2 = reader.nextLine();

      EditDistance ed = new EditDistance(s1, s2);
      System.out.println("Distanca: " + ed.getDistance());
      ed.printDistanceMatrix();
      ed.getExplanation();

      reader.close();
}
```

```
First string:
ana
Second string:
anci
Distanca: 2
Distance matrix
    a n c i
  0 1 2 3 4
a 1 0 1 2 3
n 2 1 0 1 2
a 3 2 1 1 2
Transformations:
c inserted
a --> i
```