Fakultet elektrotehnike i računarstva
Zavod za primjenjeno računarstvo

Napredni algoritmi i strukture podataka

2. laboratorijska vježba

Josip Milić 0036456339

Zagreb, 07.12.2015.

1. Zadatak

Zadaci za 11 (17) bodova

2) Napisati program (poželjno C, C++, C# ili Java) koji će **dinamičkom strategijom** (dinamičkim programiranjem) riješiti problem u kojem svaka kategorija ima konačni broj podkategorija, a može se uzeti samo po jedan član tih podskupova. U primjeru s predavanja, to bi značilo da npr. postoje iste stvari od različitih materijala pa su im različite i vrijednosti i težine, a lopov može (želi) uzeti najviše jednu stvar „svake vrste“, pri čemu je ograničenje ukupna težina, a ne volumen.

- za kolokviranje vježbe važno je pregledno i jasno opisati problem i njegove karakteristike koje ga čine prikladnim za primjenu dinamičkog programiranja
- naglasak rada treba biti na kvalitetnom programu pa u dokumentaciji treba jasno (ali sažeto!) prikazati ideju rješenja i organizaciju programa (strukture podataka, pomoćne funkcije, itd.). Programski kod ne upisivati u dokumentaciju, osim kratkih izvadaka ako se radi o posebno važnim blokovima.

2. Rješenje zadatka

2.1. Teorijski uvod

2.1.1. Dinamičko programiranje

Dinamičko programiranje je strategija kojoj je osnovno načelo postupno graditi rješenje složenog problema koristeći rješenja istovrsnih manje složenih problema (bottom-up pristup).
Za razliku od podijeli pa vladaj (divide and conquer) strategije koja problem rješava top-down pristupom, ne ponavlja već obavljeni posao jer maksimalno iskorištava rezultate prethodnih koraka.

Tipična primjena je u optimizacijskim problemima u kojima se do konačnog rješenja dolazi tek nakon niza odluka, pri čemu nakon svake odluke problem ostaje istovrstan, samo manje složenosti,
a konačno rješenje se dobiva na temelju optimalnih rješenja podproblema koji nastaju nakon svake pojedine odluke i čija su rješenja najbolja moguća s obzirom na do tada postignuto stanje.

Da bi problem bio rješiv po načelu dinamičkog programiranja, problem mora zadovoljavati sljedeća dva uvjeta:

1. optimalna podstruktura (optimal substructure)
    - svojstvo problema da optimalno rješenje sadrži u sebi optimalna rješenja nezavisnih podproblema (sastoji se od njih)
        - to samo po sebi nije dovoljno jer je inače i svojstvo koje upućuje na primjenu "lakome" (greedy) strategije
        - dobar primjer je problem traženja najkraćeg puta (Dijkstrin algoritam, lakoma strategija); najkraći put između dva vrha sastoji se od najkraćeg puta od polaznog vrha do nekog međuvrha i najkraćeg puta od međuvrha do završnog vrha $\Leftrightarrow$ optimalna rješenja dvaju nezavisnih podproblema.

2. preklopljenost podproblema (overlapping subproblems)
    - svojstvo problema da njegovo rješavanje zahtijeva (vodi u) višekratno rješavanje identičnih (pod...)podproblema pa se prethodni rezultati mogu iskoristiti za brže rješavanje kasnijih koraka
    - (pod...)podproblemi i dalje moraju biti nezavisni

Tipičan primjer dinamičkog programiranja je rješenje problema naprtnjače (Knapsack problem):

Dostupne su mnogobrojne stvari i za svaku pojedinu stvar poznaju se dva svojstva: **vrijednost** i **cijena**. Ta dva svojstva mogu podrazumijevati različita stvarna svojstva stvari. Uobičajeni primjer je lopov koji pokušava ukrasti što više stvari koje mu stanu u vreću, a da njihova ukupna vrijednost bude što veća. U tom slučaju **vrijednost** predstavlja stvarnu vrijednost stvari u npr. količini novca koju lopov može dobiti za tu stvar, a **cijena** predstavlja volumen pojedine stvari. Zbroj cijena stvari koje lopov može uzeti mora biti ispod nekog određenog broja (volumen vreće), a cilj je odabrati stvari čiji je zbroj vrijednost što veći.

U primjeru koji se obrađuje u ovom dokumentu stvari su osobe ili svojstva (npr. žanr filma) koje pripadaju filmskoj industriji. **Vrijednost** stvari predstavlja faktor množenja uloženog novca odnosno koliko se isplati zaposliti baš tu osobu (npr. neki poznati glumac koji je glumio u vrlo lukrativnim filmovima će vrlo vjerojatno privući više publike i više će se zaraditi nego ako se zaposli neki glumac koji nije baš poznat) ili odabrati baš to svojstvo (npr. Sci-Fi film će vrlo vjerojatno više profitirati od neke ljubavne drame). **Cijenu** osobe predstavlja količina novca koja je potrebna kako bi se zaposlilo tu osobu ili odabralo to svojstvo. Ako osoba više vrijedi, logično je i da traži veću količinu novca za svoj angažman na filmu no ta cijena može biti precijenjena ili podcijenjena. Identično vrijedi i za neko svojstvo filma (npr. za Sci-Fi film je potrebno uložiti više novca zbog specijalnih efekata). Potrebno je odabrati osobe i svojstva čija ukupna cijena ne prelazi odobreni budžet filma, a da je njihova ukupna vrijednost što veća.

U ovom dokumentu problem koji se obrađuje je nešto složeniji jer su stvari podijeljene u grupe i moguće je odabrati maksimalno jednu stvar iz pojedine grupe no prije opisa tog problema odnosno njegovog rješenja potrebno je razumjeti rješenje osnovnog problema jer je ono vrlo slično rješenju složenijeg problema.

Prigodno je stvari prikazati tablično:

	Stvar 1	Stvar 2	Stvar n
Vrijednost	V1	V2	Vn
Cijena	C1	C2	Cn

Gdje je Vn vrijednost stvari n, a Cn cijena stvari n.

Potrebno je uočiti strukturu optimalnog rješenja:

- Recimo da znamo najbolje moguće rješenje kad promatramo *k* stvari i cijeli raspoloživi kapacitet (zbroj cijena). Označimo skup stvari koje čine najbolji izbor s Ω.
- Ključno je primijetiti da ako iz skupa Ω uklonimo samo jednu (bilo koju) stvar, preostale stvari sigurno čine najbolji mogući izbor iz skupa od *k–1* stvari, ali za ukupni kapacitet umanjen za cijenu izdvojene stvari.

Zato se najbolje moguće rješenje većeg problema sastoji od najboljih mogućih rješenja manjih istovrsnih problema → optimalna podstruktura. Rješenje za *k* stvari se dobiva koristeći rješenja za *k-1* stvari, ono za *k-1* se dobiva za *k-2* itd.

Zbog prethodnog se zaključuje da nam je potrebna rekurzivna formula za izračunavanje konačnog rješenja:

- Promatranjem bilo koje stvari, recimo *k*-te, raspoloživog skupa S, uočavamo da konačno rješenje može biti samo dvojako – ono ili uključuje ili ne uključuje promatranu stvar.
- Ako ju ne uključuje, onda je rješenje problema za zadanu cijenu c i cijeli skup S jednako optimalnom rješenju za cijenu c i skup bez k-te stvari S\{*k*}. Simbolički, $v_k(c) = v_{-k}(c)$, gdje je $v_{-k}(c)$ označava najveću moguću vrijednost za cijenu c kada promatramo skup bez *k*-te stvari S\{*k*}.
- Ako ju uključuje, onda je najveća ostvariva vrijednost za cijenu c i skup s *k*-tom stvari jednaka optimalnom rješenju za skup bez *k*-te stvari i najveću dozvoljenu cijenu umanjenu za cijenu *k*-te stvari, tj. za cijenu $c - cost(k)$, uvećana za vrijednost *k*-te stvari. Simbolički, $v_k(c) = v_{k-1}\big(c - cost(k)\big) + value(k)$.
- Iz prethodnih razmatranja slijedi da k-ta stvar ulazi u najbolji izbor ako je

$$v_{k-1}\big(c - cost(k)\big) + value(k) > v_{k-1}(c)$$

Tražena rekurzivna formula glasi:

$$v_k(c) = \max\big\{v_{k-1}(c),\ \ v_{k-1}\big(c - cost(k)\big) + value(k)\big\};\ v_0(c) = 0$$

Koristeći gornju rekurzivnu formulu može se izgraditi tablica koja ima stupaca koliko i svih stvari, a broj redaka jednak ciljanom zbroju cijena + 1 (jer nas zanima i slučaj kad je kapacitet nula, taj slučaj je bitan jer je moguće da postoji stvar koja je 'besplatna' odnosno kojoj je cijena jednaka nula).
Svaki element te tablice je izračunata vrijednost $v_k(c)$ gdje *k* stupac, a *c* redak.

U krajnjem retku i stupcu nalazit će se maksimalni zbroj vrijednosti za traženi zbroj cijena stvari (treba napomenuti da to ne znači da će se zbog toga uvijek odabrati i zadnje gledana stvar, njezina izračunata vrijednost može biti jednaka prethodnoj pa zbog toga odabiremo ili gledamo prethodnu stvar). Osim maksimalnog zbroja vrijednosti zanima nas i koje su stvari odabrane. Za to nam je potrebna informacija svakog elementa: je li njegova izračunata vrijednost jednaka vrijednosti prethodnog elementa (isti redak, stupac prije) ili je jednaka zbroju elementa iznad i vrijednosti te stvari.

- Ako je ista kao vrijednost prethodnog elementa onda se zanemaruje taj element i gleda se taj element prije odnosno ne uzima se trenutno gledani element.
- Ako nije ista onda se uzima i gleda se gornji element koji je 'uzrokovao' povećanje vrijednosti trenutno gledanog elementa.
- Ponavlja se sve dok nema više elemenata za promatranje.

Krajnji rezultat je niz stvari čiji indeksi odgovaraju stupcima uzimanih elemenata.
Ukupan zbroj vrijednosti je jednak vrijednosti krajnjem elementu, a ukupan zbroj cijena je manji ili jednaki traženom zbroju cijena.

2.1.2. Dinamičko programiranje s grupama

Ideja dinamičkog programiranja s grupama je vrlo slična ideji dinamičkog programiranja s nezavisnim stvarima.
Umjesto cjelovitog skupa stvari S, u ovom problemu postoje podskupovi stvari Si i vrijedi sljedeće:

$$S_i \cap S_j = \emptyset, \qquad za\ svaki\ i \neq j$$
$$\sum_i^n S_i = S$$

Cilj je isti, odabrati što veću ukupnu vrijednost za dani dozvoljeni zbroj cijena, no u ovom problemu je dozvoljeno odabrati najviše jednu (0 ili 1) stvar iz pojedinog podskupa (grupe) S_i.
Zato se algoritam modificira na način da se ne gleda (gornji) element prethodnog stupca tablice nego se gledaju svi (gornji) elementi prethodne grupe odnosno maximum njihovih izračunatih vrijednosti.
Kao i u prethodnom algoritmu, ukupna vrijednost je u zadnjem retku i stupcu tablice.
Odabir stvari je također identičan osim što se gleda element s maksimalnom izračunatom vrijednosti u tom stupcu i toj grupi koji je 'uzrokovao' povećanje.
Krajnji rezultat su odabrane stvari.
Ukupan zbroj vrijednosti je jednak vrijednosti krajnjem elementu, a ukupan zbroj cijena je manji ili jednaki traženom zbroju cijena.

2.1.3. Dinamičko programiranje s grupama smanjene složenosti

Može se uočiti da ako bi se slijepo pratio algoritam i promatrao za svaki element tablice sve elemente istog reda prethodne grupe i sve elemente gornjeg ($c - cost(k)$) reda prethodne grupe da bi složenost bila velika i primjerice podskupovi koji sadrže mnogo stvari bi uzrokovali dugotrajni izračun vrijednosti samo jednog elementa tablice (od velikog broja elemenata jer broj elemenata može biti u najboljem slučaju jednak broju svih stvari).
Zato je potrebno imati imati strukturu koja će umjesto pamćenja svih izračunatih vrijednosti elemenata sadržavati samo maximume izračunatih vrijednosti elemenata za grupu u tom redu jer nas zanima za neki red i grupu samo najveća izračunata vrijednost.
Tako bi za n stvari neke grupe potrebno bilo zapamtiti umjesto n vrijednosti samo **1** vrijednost. Tako se zapravo tablica stvari pretvara u tablicu grupa gdje svaki element ima maksimalnu vrijednost za taj red i stupac (grupu) u kojem se nalazi.
Sljedeći element odnosno stvar ne treba gledati sve vrijednosti prethodne grupe već samo jednu vrijednost.
To uzrokuje da nam za odabir stvari nije potrebna tablica u kojoj su stupci stvari, a retci tražene ukupne cijene. Dovoljna nam je tablica u kojoj su stupci grupe, a retci tražene ukupne cijene. Elementi su maksimumi grupa za pojedini redak (traženu cijenu).

2.2. Implementacija

Implementacija rješenja problema dinamičkog programiranja s grupama je izvršena pomoću C#-a koristeći standardne biblioteke. Unutar *solutiona* nalaze se tri projekta:

1. `KnapsackGroupsProject`: glavni projekt koji sadrži razrede koje koriste ostala dva projekta: `DynamicTable`, `DynamicTableObject` i `KnapsackGroupsCore`.
2. `KnapsackGroupsGUI`: projekt koji sadrži verziju programa koji sadrži GUI napravljen pomoću *WinFormsa*. Ova verzija sadrži preglednik tablice i označenih ključnih elemenata. Podržava i mijenjanje vrijednosti i cijena stvari, kao i traženi zbroj cijena i omogućava novo računanje na temelju tih promjena.
3. `KnapsackGroupsConsole`: projekt koji sadrži konzolnu verziju programa. Ova verzija za danu datoteku samo ispisuje odabrane stvari i zbrojeve vrijednosti i cijena. Ovom verzijom se može testirati brzina programskog rješenja jer se ne troši vrijeme na iscrtavanje tablice.

2.2.1. Opis rješenja osnovnog problema

U teorijskom uvodu objašnjen ja način dobivanja rješenja osnovnog (sve stvari se nalaze u jedinstvenom skupu S) problema.
Pseudokod koji opisuje upisivanje svih elemenata tablice:

```
for c = 0 to targetCost do
     for k = 0 to count(S) do
          if cost(k) <= c then
               table[c][k] = calculateV(table, c, k)
          else
               table[c][k] = table[c][k-1]
          end if
     end do
end do


calculateV(table, c, k)
     if (k < 0) then
          return 0
     end if
     first = table[c][k-1]
     second = table[c – cost(k)][k-1] + value(k)
     return max(first, second)
```

`cost(k)` – cijena za stvar k (indeks stvari kreće od 0 za razliku od gornje formule gdje kreće od 1).
`value(k)` – vrijednost za stvar k.

2.2.2. Opis rješenja problema s grupama

Napomenuto je kako rješenje problema s grupama slično rješenju osnovnog problema. Pseudokod koji opisuje upisivanje svih elemenata tablice:

```
for c = 0 to targetCost do
     for k = 0 to count(S) do
          if cost(k) <= c then
               table[c][k] = max(calculateV(table, c,
table[c][E[previousGroup-1][0]]), calculateV(table, c,
table[c][E[previousGroup-1][1]]),…, calculateV(table, c,
table[c][E[previousGroup-1][n]]))
          else

               table[c][k] = max(table[c][E[previousGroup-1][0]],
table[c][E[previousGroup-1][1]],…, table[c][E[previousGroup-1][n]])
          end if
     end do
end do
```

`E[previousGroup-1][n]` – indeks n-tog elementa prethodne grupe.

2.2.3. Opis rješenja problema s grupama smanjene složenosti

Pogledom na prethodni pseudokod može se uočiti gdje bi takvim načinom mogao nastati problem i zašto je potrebna dodatna struktura (ili zašto je moguće zamijeniti cijelu tablicu takvom strukturom) koja će umjesto čuvanja svih vrijednosti neke grupe za neki redak čuvati samo najveću izračunatu vrijednost za pojedinu grupu i redak. Time se se omogućava sljedeći način nalaženja potrebnih vrijednosti:

```
for c = 0 to targetCost do
     for k = 0 to count(S) do
           if cost(k) <= c then
                 structure[c][k] = calculateVmax(table, c,
                 previousGroup-1)
           else

                 structure [c][k] = structure [c][previousGroup-1]
           end if
     end do
end do
```

`calculateVmax(table, c, previousGroup-1)` – prilagođena funkcija `calculateV` koja izračunava vrijednosti i odabire maksimalnu za trenutno gledanu grupu.

Može se primijetiti da se usporedno s popunjavanjem te strukture može popunjavati i tablica s vrijednostima koje su nepotrebne za izračun najboljeg odabira, ali nam poslužuju kao grafička interpretacija rješenja problema što je i napravljeno u verziji koja sadrži GUI. Konzolna verzija izostavlja taj dio.

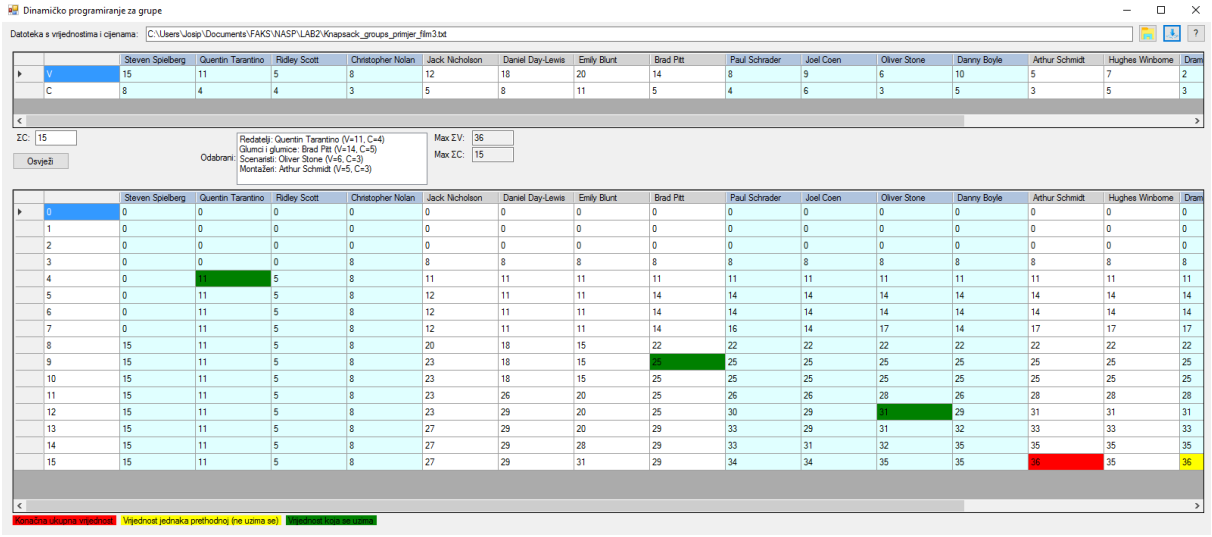
Prethodnim rješenjem složenost se smanjuje na linearnu složenost $O(n)$.
To pokazuje i stvarno mjerenje izvršenja konzolne verzije programa koristeći random generirane primjere (vrijednosti i cijene stvari su randomizirane, broj stvari u pojedinoj grupi također). Pomoću skripte napravljeni su redom sljedeći primjeri, a uz njih je izračunato potrebno vrijeme za izvršenje:

1. nasp_lab2_gen_C1000_G100_S10000.txt - 0.185 s
2. nasp_lab2_gen_C10000_G100_S10000.txt – 1.556 s
3. nasp_lab2_gen_C100000_G100_S10000.txt – 14.260 s
4. nasp_lab2_gen_C1000000_G100_S10000.txt – 148.233 s

Imena primjera su deskriptivna: C predstavlja traženi ukupni zbroj cijena, G ukupni broj grupa i S ukupni broj stvari.
Preporučljivo je ove primjere koristiti samo u konzolnoj verziji jer je iscrtavanje u tablici ograničeno i vremenski zahtjevno.

2.3. Grafičko sučelje programa

Izgled verzije programa koji sadrži grafičko sučelje (GUI):



Na slici je prikazan izgled tablice za učitan primjer *Knapsack_groups_primjer_film2.txt*

Pomoću gumbiju se odabire i učitava datoteka s vrijednostima i cijena stvari. Pomoću trećeg gumba može se saznati kako bi ta datoteka trebala izgledati (u ovom dokumentu je opisan primjer filmske industrije, ali se program može koristiti za bilo koji sličan problem koji sadrži stvari i grupe precizirane formatom datoteke). Uz program su priložene datoteke primjeri.

Učitavanjem datoteke se odmah pokreće rješavanje danog problema i zatim se rješenje grafički prezentira pomoću tablice. **Crvenom** bojom se označava element stvari koji se prvi uzima (gleda se od zadnjeg retka prema prvom). **Žutom** bojom se označavaju elementi tablice koji su jednaki najboljem elementu prethodne grupe za taj red. Primjerice na gornjoj slici, uzima se lijeva stvar jer se za istu cijenu dobiva veća vrijednost. **Zelenom** bojom se označavaju elementi koji su najbolji u redu i grupi koja se gleda i koji su nastali zbrajanjem vrijednosti stvari i vrijednosti gornjeg elementa prethodne grupe. Stvari su tablicama označene različitom bojom (parne grupe svjetlo plavom, a neparne bijelom bojom).

U listi su prezentirane uzimane stvari kao i imena grupa kojima pripadaju. Uz nju su napisani maksimalno postignuti zbrojevi cijena za traženi zbroj cijena i ukupan zbroj cijena odabranih stvari.

U gornjoj tablici su prikazane stvari i njihove vrijednosti i cijene. Svaka vrijednost i cijena se mogu **mijenjati**, kao i traženi zbroj cijena. Potrebno je pritisnuti gumbe Osvježi kako bi se prihvatili novi podaci čime se pokreće novo računanje i prezentacija rješenja.

3. Zaključak

Rješenje problema dinamičkog programiranja s grupama je zanimljivo jer se za rješavanje mogu koristiti jednostavne strukture i izračun je jednostavan i brz. Umjesto pregledavanja svih mogućih kombinacija stvari i grupa moguće je na (relativno) brz način odabrati stvari bez obzira na broj stvari, grupa i traženi ukupni zbroj cijena. To jasno pokazuje moć i praktičnu svrhu dinamičkog programiranja. Kao osnova koristilo se dinamičko programiranje za cjeloviti skup stvari objašnjeno u prezentaciji [1].

4. Literatura

[1] *Nikica Hlupić, Damir Kalpić, Dinamičko programiranje (prezentacija predavanja), FER, Zagreb, 2009.*