Prirodoslovno-matematički fakultet Sveučilišta u Zagrebu

Problem izabira aktivnosti: dinamičko programiranje i pohlepni pristup

Seminarski rad

Zagreb, 20.11.2023. Petra Salaj

1. UVOD

Problem izabira aktivnosti je problem optimizacije koji se koristi za određivanje maksimalnog broja kompatibilnih aktivnosti koje koriste zajednički resurs. Postoje razne interpretacije i upotrebe ovog problema. U svima imamo pretpostavku da se aktivnosti mogu obavljati jedna po jedna, a ne više njih istovremeno. Neke od njih su:

- Određivanje maksimalnog broja aktivnosti koje osoba može obavljati u određenom periodu, ako znamo vrijeme početka i završetka pojedine aktivnosti
- Broj različitih autobusa na jednom peronu ako imamo njihovo vrijeme dolaska na peron i vrijeme odlaska s perona
- Rezerviranje 1 konferencijske sobe za maksimalni broj sastanaka u danu ako imamo vrijeme početka i završetka svakog sastanka

i mnoge druge. Od sada nadalje, u svim primjerima koristit ćemo 1.interpretaciju, odnosno pronalazit ćemo maksimalan broj kompatibilnih aktivnosti (onih koje se ne preklapaju) u jednom danu koje osoba može obaviti. Dakle, pretpostavimo da smo odgovorni za sastavljanje rasporeda osobe u jednom danu. Neka su nam sa skupom $S = \{a_1, a_2, \dots, a_n\}$ dane predložene aktivnosti, njih n. Svaka od aktivnosti ima početno vrijeme s_i, i završno vrijeme f_i gdje je $0 \leq s_i < f_i < \infty$ (u našem primjeru: $f_i \leq 24$). Ako je odabrana aktivnosti a_i se obavlja tijekom poluotvorenog intervala $[s_i, f_i\rangle$. Aktivnosti a_i i a_j su kompatibilne tj. ne preklapanju se ako i samo ako $[s_i, f_i\rangle \cap [s_j, f_j\rangle = \emptyset \Leftrightarrow s_i \geq f_j \; ILI \; s_j \geq f_i$ (1), odnosno ako aktivnost a_i počinje nakon završetka aktivnosti a_j ili obratno. Za primjer, pretpostavimo da imamo 11 različitih aktivnosti s danim početnim i završnim vremenima kao u tablici 1. Podskup $\{a_2, a_6, a_{11}\}$ sastoji se od 3 međusobno kompatibilnih aktivnosti, međutim to nije najveći mogući podskup budući da je $\{a_2, a_4, a_9, a_{11}\}$ veći. Još jedan takav najveći je i $\{a_1, a_4, a_8, a_{11}\}$.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Tablica 1: aktivnosti $\{a_1, a_2, \dots, a_{11}\}$ sa njihovim početnim i završnim vremenima

U ovom seminaru, predstavit ću rješenje problema na više načina. Najprije ćemo pokazati da je problem moguće riješiti pomoću dinamičkog programiranja, ali da zahvaljujući posebnoj karakteristici problema izabira aktivnosti postoji bolji (u smislu vremenske složenosti) pohlepni algoritam koji nam i u teorijskoj i u empirijskoj analizi daje manju vremensku složenost.

2. DINAMIČKO PROGRAMIRANJE

2.1. Princip optimalnosti

Kako bi problem izabira aktivnosti mogli riješiti dinamičkim programiranjem, moramo naći princip optimalnosti koji njemu odgovara i dokazati da on vrijedi. Prisjetimo se, princip optimalnosti glasi da optimalni niz odluka ima svojstvo da za bilo koje početno stanje i početnu odluku u tom stanju, preostale odluke moraju činiti optimalan niz gledano iz stanja nakon prve odluke. Također važno je napomenuti da ukoliko je prva odluka bila pogrešna, optimalnost nadalje, neće dovesti do globalnog optimalnog rješenja. Ostaje pitanje optimalnosti prve odluke, ali samo po optimalnim nizovima nakon te odluke.

Dokaz principa optimalnosti:

Označimo sa $S_{i,j}$ skup aktivnosti koje počinju nakon što aktivnost a_i završi i završavaju prije nego aktivnost a_j započne, preciznije $S_{i,j} = \{a_k \in S: s_k \geq f_i \ \wedge \ f_k \leq s_j\}$ te neka je $A_{i,j}$ maksimalan podskup kompatibilnih aktivnosti iz $S_{i,j}$. Primijetimo da je maksimalni broj kompatibilnih aktivnosti (naše optimalno rješenje) jednako kardinalitetu skupa $A_{i,j}$.

Pretpostavimo da je $a_k \in S_{i,j}$ dio rješenja, odnosno da $a_k \in A_{i,j}$ (1.odluka).

Uključivanjem a_k u optimalno rješenje ostaje nam riješiti 2 podproblema;

1. Pronaći $A_{i,k}$ = maksimalni podskup kompatibilnih aktivnosti od $S_{i,k}$ (aktivnosti koje počinju nakon što aktivnost a_i završi i završavaju prije nego aktivnost a_k započne)
2. Pronaći $A_{k,j}$ = maksimalni podskup kompatibilnih aktivnosti od $S_{k,j}$ (aktivnosti koje počinju nakon što aktivnost a_k završi i završavaju prije nego aktivnost a_j započne)

Neka je $\boldsymbol{A_{i,k} = A_{i,j} \cap S_{i,k}}$ **i** $\boldsymbol{A_{k,j} = A_{i,j} \cap S_{k,j}}$ (2) kako bi imali da $A_{i,k}$ sadrži sve aktivnosti iz $A_{i,j}$ koje završavaju prije nego što a_k započne i $A_{k,j}$ sadržava sve aktivnosti iz $A_{i,j}$ koje započnu nakon što a_k završi. Prema tome, imamo $A_{i,j} = A_{i,k} \sqcup \{a_k\} \sqcup A_{k,j}$ pa je kardinalitet skupa $A_{i,j}$ (maksimalni broj kompatibilnih aktivnosti) jednak zbroju kardinaliteta tih disjunktnih skupova odnosno $|A_{i,j}| = |A_{i,k}| + 1 + |A_{k,j}|$.

Tvrdimo: $A_{i,j}$ mora sadržavati optimalna rješenja oba podproblema $S_{i,k}$ *i* $S_{k,j}$ tj. vrijedi (2). Pokazat ćemo tvrdnju za $S_{k,j}$, analogno se pokaže i za $S_{i,k}$.

Naime, kada bi postojao skup $A'_{k,j} \subset S_{k,j}$ takav da je $|A'_{k,j}| > |A_{k,j}|$ $\Rightarrow |A_{i,k}| + 1 + |A'_{k,j}| > |A_{i,k}| + 1 + |A_{k,j}| = |A_{i,j}|$ što bi dovelo do kontradikcije s pretpostavkom da je $A_{i,j}$ optimalno rješenje za podproblem $S_{i,j}$. Međutim, ako bi u tom slučaju uzeli $A'_{k,j}$ umjesto $A_{k,j}$ kao dio optimalnog rješenja za podproblem $S_{i,j}$ i dalje bi vrijedilo $A_{i,j} = A_{i,k} \sqcup \{a_k\} \sqcup A'_{k,j}$ pa bi optimalno rješenje bilo jednako $|A_{i,k}| + 1 + |A'_{k,j}|$. Analogno, se pokaže ako postoji $A'_{i,k} \subset S_{i,k}$ takav da je $|A'_{i,k}| > |A_{i,k}|$, možemo uzeti taj $A'_{i,k}$ kao dio optimalnog rješenja za $S_{i,j}$.

Time smo dokazali da se optimalno rješenje podproblema $S_{i,j}$ može dobiti iz optimalnih rješenja za podprobleme $S_{i,k}$ i $S_{k,j}$. Dakle, princip optimalnosti vrijedi pa se problem izabira aktivnosti može riješiti dinamičkim programiranjem.

Označimo sada sa $c[i,j] = |A_{i,j}|$ maksimalan broj aktivnosti (optimalno rješenje) za podproblem $S_{i,j}$. Pokazali smo da je $c[i,j] = c[i,k] + c[k,j] + 1$.

Međutim, ako ne znamo koja aktivnost $a_k \in S_{i,j}$ je dio rješenja $A_{i,j}$ moramo isprobavati sve aktivnosti iz skupa $S_{i,j}$ da bi dobili maksimalan broj kompatibilnih aktivnosti tj. najveći kardinalitet skupa $A_{i,j}$. Dakle,

$$c[i,j] = \begin{cases} 0, \; ako\; je\; S_{i,j} = \emptyset \\ \max\{c[i,k] + c[k,j] + 1 : a_k \in S_{i,j}\}, \; ako\; je\; S_{i,j} \neq \emptyset. \end{cases}$$

Možemo napraviti rekurzivni algoritam (top-down pristup dinamičkog programiranja) koji kada jednom izračuna rješenje nekog podproblema, to rješenje zapamti (spremi ga u memoriju) i kad mu idući put zatreba, jednostavno ga pročita s mjesta gdje ga je spremio. Ovakav način rješavanja naziva se memoizacija. S druge strane ako se odlučimo na bottom-up pristup, možemo jednostavno spremati međurezultate u tablicu (dvodimenzionalno polje) i popunjavati redom kako ih određujemo. Međutim, idući teorem daje nam prvu odluku, aktivnost a_k koja pripada optimalnom rješenju. Time dobivamo i bolje načine implementacije jer ne moramo najprije rješavati sve podprobleme kako bi došli do nje.

2.2. Odabir prve aktivnosti a_k

Kad bi željeli dodati neku aktivnost a_k optimalnom rješenju bez računanja svih mogućih rješenja podproblema i spremanja istih u memoriju, intuitivno bismo htjeli odabrati aktivnost koja ostavlja najviše dostupnog resursa, u ovom slučaju vremena za što više ostalih aktivnosti. Možda bi nam prva ideja bila sortirati aktivnosti uzlazno po trajanju i odabrati najprije aktivnost koja traje najkraće. Međutim, ako odaberemo aktivnosti kao u Tablici 2 i sortiramo ih po trajanju kao u Tablici 3, vidimo da bi nam taj algoritam dao rješenje $\{a_2\}$, što nije optimalno jer postoji veći podskup kompatibilnih aktivnosti $\{a_1, a_3\}$.

i	**1**	**2**	**3**
s_i	8	8.90	9
f_i	9	9.50	10
Trajanje/h $(f_i - s_i)$	1	0.60	1

Tablica 2: aktivnosti $\{a_1, a_2, a_3\}$ sa njihovim početnim i završnim vremenima

i	**2**	**1**	**3**
s_i	8.90	8	9
f_i	9.50	9	10
Trajanje/h $(f_i - s_i)$	0.60	1	1

Tablica 3: aktivnosti sortirane uzlazno po trajanju

Postoji ipak pohlepna (greedy) odluka koja daje optimalno rješenje, a to je da najprije odaberemo aktivnost iz S koja prva završava (ima najranije vrijeme završetka). Ako ima više aktivnosti sa istim najranijim vremenom završetka, tada biramo bilo koju od njih. Pokažimo da je to dobra odluka.

Pretpostavimo da su aktivnosti u S sortirane uzlazno po vremenima završetka aktivnosti, tada je aktivnost koja ima najranije vrijeme završetka upravo prva aktivnost a_1. U tom slučaju, ne ostaje nam za riješiti 2 podproblema $S_{i,k}$ i $S_{k,j}$ kao ranije, nego samo jednog S_1 – pronalaženje maksimalnog podskupa kompatibilnih aktivnosti u skupu aktivnosti koje počinju nakon što a_1 završi. Ne moramo provjeravati aktivnosti koje završavaju prije nego a_1 započne jer imamo uvjete da je $s_1 < f_1$ i f_1 najranije vrijeme završetka pa ne postoji aktivnost koja ima završno vrijeme manje ili jednako s_1 (kad bi to vrijedilo bile bi kompatibilne po (1)).

Ranije smo pokazali da za problem izabira aktivnosti vrijedi princip optimalnosti, a naš idući cilj je pokazati da je pohlepni odabir u kojem biramo aktivnost koja najranije završava uvijek dio optimalnog rješenja. Idući teorem nam daje potvrdan odgovor.
Neka je sada $S_k = \{a_i \in S: s_i \geq f_k\}$ skup aktivnosti koje započinju nakon što aktivnost a_k završi.

TEOREM: Neka je S_k neki neprazan podproblem i $a_m \in S_k$ aktivnost koja ima najranije vrijeme završetka. Tada je aktivnost a_m dio nekog mogućeg maksimalnog podskupa kompatibilnih aktivnosti od S_k.

Dokaz: Neka je A_k maksimalni podskup kompatibilnih aktivnosti od S_k i neka je $a_j \in A_k$ aktivnost s najranijim vremenom završetka. Imamo 2 slučaja:

1. $a_j = a_m$, onda smo GOTOVI jer smo pokazali da a_m pripada nekom maksimalnom podskupu kompatibilnih aktivnosti od S_k.
2. $a_j \neq a_m$ Neka je $A'_k = (A_k \backslash \{a_j\}) \cup \{a_m\}$

   A'_k zapravo skup A_k u kojem a_m zamjenjuje a_j.

   $a_j \in A_k \subseteq S_k$ je aktivnost u A_k koja najranije završava i $a_m \in S_k$ je aktivnost u S_k koja najranije završava $\Rightarrow f_m \leq f_j$.

   Iz $f_m \leq f_j$ i činjenice da su aktivnosti u A_k kompatibilne $\Rightarrow$ aktivnosti iz skupa A'_k su kompatibilne. Budući da je $|A_k| = |A'_k|$, možemo zaključiti da je A'_k maksimalan podskup kompatibilnih aktivnosti od S_k koji sadrži a_m. Čime je tvrdnja teorema dokazana.

Dakle, ovim teoremom pokazali smo da iako se problem izabira aktivnosti može riješiti dinamičkim programiranje da to nije potrebno. Umjesto toga, možemo ponavljati postupak uzimanja najprije aktivnosti koja prva završava a_m, uzeti samo aktivnosti a_k kompatibilne s njom (takve da je $s_k \geq f_m$) i ponavljati postupak dok više nemamo aktivnosti. Važno je primijetiti da postoji još jedan mogući dobar pohlepni odabir. Kad bi aktivnosti sortirali po njihovim vremenima početka uzlazno tada bi mogli uvijek izabrati aktivnost koja zadnja počinje. U tom slučaju nam isto ostaje riješiti samo 1 podproblem $S_n = \{a_i \in S: f_i \leq s_n\}$, a to je pronaći maksimalan podskup kompatibilnih aktivnosti u skupu aktivnosti koje završavaju prije nego a_n počne. Nadalje, ako je a_m aktivnost koja najkasnije počinje traženje njoj kompatibilnih svodi se na provjeravanje uvjeta $f_k \leq s_m$. Prošli teorem bi se analogno dokazao za ovakvu aktivnost a_m i $S_k = \{a_i \in S: f_i \leq s_k\}$.

PRIMJER: (za pohlepni pristup)

Aktivnosti sortirane po vremenima završetka uzlazno ($S_k = \{a_i \in S : s_i \geq f_k\}$)

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

START: S = $\{a_1, a_2, \ldots, a_{11}\}$, A = $\cup A_k = \emptyset$ maksimalni podskup kompatibilnih aktivnosti iz S

Po pohlepnom algoritmu u kojem biramo aktivnost koja završava prva uzmemo najprije a_1 ($A = \{a_1\}$) i rješavamo podproblem S_1 tako što provjeravamo je li a_2 kompatibilna s njom, odn. vrijedi li $3 = s_2 \geq f_1 = 4$, to očito ne vrijedi pa idemo dalje. Za a_3 također ne vrijedi ($0 < 4$). Nadalje, za a_4 vrijedi jer $5 = s_4 \geq f_1 = 4$ ($A = \{a_1, a_4\}$), pa sada rješavamo podproblem S_4. Aktivnosti a_5, a_6, a_7 nisu kompatibilne s a_4 jer ne vrijedi uvjet, pa a_8 je jer $7 = s_8 \geq f_4 = 7$ ($A = \{a_1, a_4, a_8\}$). Sada rješavamo podproblem S_8, opet nam aktivnosti a_9 i a_{10} ne zadovoljavaju uvjet kompatibilnosti, a aktivnosti a_{11} zadovoljava jer $12 = s_{11} \geq f_8 = 11$. Ostali smo bez aktivnosti, pa smo GOTOVI. Dobili smo maksimalni podskup kompatibilnih aktivnosti A= $\{a_1, a_4, a_8, a_{11}\}$.

3. TEORIJSKE SLOŽENOSTI IMPLEMENTACIJA

U obje implementacije imamo vrijedi pretpostavka da su aktivnosti sortirane po vremenima završetka uzlazno.

3.1. Dinamičko programiranje – implementacija

```
struct Aktivnost
{
    double pocetak;
    double kraj;
};
```

Slika 1: struktura Aktivnost

```
int activitySelection_Dynamic(vector<Aktivnost> aktivnosti, int n)
{
  int i, j, max_element = 1;
  vector<int> dtable(n, 1);
  for (i = 1; i < n; i++)
  {
    for (j = 0; j < i; j++)
    {
      if (aktivnosti[i].pocetak >= aktivnosti[j].kraj && dtable[i] < dtable[j] + 1)
      {
        dtable[i] = dtable[j] + 1;
      }
    }
  }

  for (i = 0; i < n; i++)
  {
    if (dtable[i] > max_element)
    {
      max_element = dtable[i];
    }
  }
  return max_element;
}
```

Slika 2: funkcija activitySelection_Dynamic

```
brojOdabranihAktivnosti = activitySelection_Dynamic(aktivnosti, n);
```

Slika 3: Poziv funkcije activitySelection_Dynamic u glavnom programu

OBJAŠNJENJE ALGORITMA: Funkcija activitySelection_Dynamic prima 2 argumenta; vector s n različitih aktivnosti tipa podataka Aktivnost (struktura koja se sastoji od 2 varijable početak i kraj tipa double) i broj aktivnosti (veličinu vectora). Koristi pomoćni vector dtable duljine n tipa podataka int u kojem će na kraju funkcije activitySelection_Dynamic na k-tom mjestu biti spremljen najveći kardinalitet maksimalnog podskupa kompatibilnih aktivnosti ako je a_k dio tog rješenja. Optimalno rješenje, kardinalitet maksimalnog podskupa kompatibilnih aktivnosti od svih danih dobivamo tako da pronađemo maksimalni element u vectoru dtable.

Po zaključku nakon dokaza principa optimalnosti spremali bi rješenja subproblema $S_{i,j}$ u $n \times n$ matricu, ali budući da su nam aktivnosti sortirane po vremenu završetka uzlazno to bi rezultiralo time da nam je većina podskupova $S_{i,j} = \emptyset$ i samim time njihove pripadne vrijednosti u matrici jednake 0. Ali zahvaljujući upravo tom načinu sortiranja možemo iskoristiti „trik" koji koristi sljedeću karakteristiku problema.

Karakteristika: Neka je a_k dio optimalnog rješenja. Ako je $a_j \in S_k = \{a'_j \in S: f'_j \leq s_k\}$ kompatibilna s a_k tada su i sve aktivnosti iz skupa A_j (maksimalnog podskupa kompatibilnih aktivnosti koje završavaju prije nego što a_j započne) kompatibilne s a_k. Dokaz (kako bi se uvjerili da ovo stvarno vrijedi): Pretpostavimo da su a_k *i* a_j *kompatibilne* $\Leftrightarrow s_k \geq f_j$ *ILI* $s_j \geq f_k$ (1). Za $a_j \in S_k \Rightarrow f_j \leq s_k$ tj. $s_k \geq f_j$ odnosno za $a_j \in S_k$ kompatibilnu s a_k uvijek vrijedi 1.uvjet iz (1). Drugi ne vrijedi jer $f_k > s_k \geq f_j > s_j \Rightarrow f_k < s_j$. Analogno, za proizvoljnu aktivnost $a_i \in A_j$ (kompatibilnu s a_j iz skupa aktivnosti koje završavaju prije nego a_j započne) vrijedi $s_j \geq f_i$. Pa imamo $s_k \geq f_j > s_j \geq f_i$ pa iz tranzitivnosti uređaja $s_k > f_i$ odnosno aktivnosti a_k *i* a_i su kompatibilne, čime je tvrdnja dokazana.
Ova tvrdnja nam daje da za svaku aktivnost a_k možemo tražiti kompatibilne s njom $a_j \in S_k = \{a'_j \in S: f'_j \leq s_k\}$ i za optimalno rješenje problema koje sadrži aktivnost a_k uzimati $|A_j|$+1 (maksimalan broj kompatibilnih s a_j iz S_j i upravu tu a_j). U funkciji activitySelection_Dynamic to je upravo ono što rade ove 2 for petlje, a budući nam je cilj pronaći maksimalan prije nego promijeni vrijednost u vectoru dtable provjerava je li zapisano optimalno rješenje manje od nove mogućnosti.

3.2. Dinamičko programiranje - teorijska složenost algoritma

Promotrimo funkciju activitySelection_Dynamic sa Slike 1, i poprima vrijednosti od 1,…, n-1 dok j u unutarnjoj for petlji za fiksni i poprima vrijednosti j = 0,…, i-1, odnosno za fiksni i unutarnja petlja se izvrši točno i puta. Čime dobivamo ukupan broj iteracija $\frac{1}{2}n^2 - \frac{1}{2}n$ u ugniježđenim for petljama. U 3.for petlji u kojoj prolazimo vectorom dtable od n elemenata imamo još n interacija, što nam daje $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n + n = \frac{1}{2}n^2 + \frac{1}{2}n \in O(n^2)$

i	**Broj iteracija**
1	1
2	2
3	3
…	…
n-1	$n-1$

Tablica 4: Broj iteracija u ugniježđenim for petljama

If unutar ugniježdenih for petlja nam doprinosi sa složenosti O(1) pa ga zato prilikom analize možemo zanemariti. (https://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html) Prostorna složenost je O(n) jer koristimo tablicu dinamičkog programiranja (vector dtable) veličine n.

3.3. Pohlepni pristup – implementacija

Većina pohlepnih algoritama se implementira rekurzivnim algoritmom, međutim u ovom slučaju ta rekurzija bi bila skoro rekurzija repa(„tail recursive“). Rekurzija repa je rekurzija koja završavala s rekurzivnom pozivom na samu sebe. U praksi, većina ovakvih rekurzija se mijenja iterativnim funkcijama i to obično nije zahtjevan zadatak, jedan takav primjer je pretvaranje rekurzivnog algoritma za računanje faktorijela u iterativan (https://www.baeldung.com/cs/convert-recursion-to-iteration). Zapravo, neki današnji kompajleri naprave ovu optimizaciju koda automatski. (https://helloacm.com/understanding-tail-recursion-visual-studio-c-assembly-view/) U našem slučaju, kada želimo odrediti vrijeme izvršavanja za velike brojeve aktivnosti rekurzivni algoritam uzrokovao bi prepunjenje stoga (stack overflow), pa je bolje koristiti iterativnu funkciju.

```
void greedy_activity_selector(vector<double> &s, vector<double> &f, int n, vector<Aktivnost> &odabraneAktivnosti)
{
  int m, k = 0;
  Aktivnost a;
  for (m = 1; m < n; m++)
  {
    if (s[m] >= f[k])
    {
      a.pocetak = s[m];
      a.kraj = f[m];
      odabraneAktivnosti.push_back(a);
      k = m;
    }
  }
  return;
}
```

Slika 4: funkcija greedy_activity_selector

```
odabraneAktivnosti.push_back(aktivnosti[0]);
greedy_activity_selector(s, f, n, odabraneAktivnosti);
```

Slika 5: Poziv funkcije greedy_activity_selector u glavnom programu

OBJAŠNJENJE: funkcija greedy_activity_selector prima 4 argumenta; pointer na vector s sa početnim vremenima svih n različitih aktivnosti, pointer na vector f sa završnim vremenima svih n aktivnosti, broj aktivnosti n i pointer na vector odabraneAktivnosti s kojim ćemo dobiti neki maksimalni, optimalni niz kompatibilnih aktivnosti. Algoritam radi kao u PRIMJERU (za pohlepni pristup), samo što imamo skup aktivnosti S = $\{a_0, a_1, \ldots, a_{n-1}\}$ koje su spremljene u

vectoru aktivnosti. Na početku prije prvog poziva funkcije u vector odabraneAktivnosti (skup A iz primjera) stavimo prvu aktivnost a_0 jer to je aktivnost koja ima najranije vrijeme završetka te nam ostaje riješiti podproblem S_0. Varijabla k nam ustvari predstavlja indeks zadnje dodane aktivnosti optimalnom rješenju (A=$\{a_0, \ldots, a_k\}$) i indeks podproblema koji trenutno rješavamo. Znamo da je $f_k = \max\{f_i : a_i \in A\}$ odn. f_k ima maksimalno vrijeme završetka od svih odabranih aktivnosti do sada zbog toga što su aktivnosti sortirane uzlazno po vremenima završetka. For petlja pronalazi aktivnost $a_m \in S_k$ koja zadovoljava uvjet $s_m \geq f_k$ jer je tada ona kompatibilna sa svim aktivnostima iz A, dodaje ju u skup A i traži dalje njoj kompatibilne aktivnosti jer je zbog $f_m > s_m \geq f_k$ sada $f_m = \max\{f_i : a_i \in A\}$. Primijetimo da funkcija prolazi skupom aktivnosti točno jednom kako bi odabrala skup A tj. svaku aktivnost provjerava točno jednom.

3.4. Pohlepni pristup - teorijska složenost

Budući da funkcija greedy_activity_selector prolazi skupom aktivnosti točno jednom, u for petlji varijabla m poprimi vrijednosti od m = 1, …, n-1 pa imamo n iteracija. Iz čega slijedi:

$$\mathrm{T(n)} = \mathrm{n} \in O(n)$$

Prostorna složenost funkcije je O(1), budući da ne koristi dodatni prostor koji raste s veličinom ulaza. Jedini dodatni prostor koji se koristi je za izlazni vector odabraneAktivnosti, ali je njegova veličina ograničena na broj odabranih aktivnosti, koji je najviše n. Stoga je prostorna složenost konstantna.

4. EMPIRIJSKA ANALIZA

Empirijska analiza napravljena je na prijenosnom računalu sa sljedećim specifikacijama:

Procesor: Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz

Instalirani RAM: 16,0 GB Vrsta sustava: 64-bitni operacijski sustav, procesor x64

Programski jezik: C++ Program: Visual studio Code 1.84

Operacijski sustav: Linux Mint 20.2 x86_64 Kompajler: gcc version 9.4.0

Aktivnosti sam generirala pomoću funkcije na slici ispod koja vraća vector slučajno generiranih aktivnosti. Ovaj način generiranja slučajnih vremena početka i kraja aktivnosti nam garantira da ćemo svakim pokretanjem programa imati različiti niz aktivnosti S za dani broj aktivnosti n.

```
vector<Aktivnost> generirajRandomAktivnosti(int n)
{
    vector<Aktivnost> aktivnosti;
    int i;

    mt19937 rng(static_cast<unsigned int>(time(nullptr)));
    uniform_real_distribution<double> pocetnoVrijemeDist(0.0, 23.0); // gornja
i donja granica za početno vrijeme 0.0 i 23.0 uključeno

    for (int i = 0; i < n; ++i)
    {
        Aktivnost aktivnost;
        aktivnost.pocetak = pocetnoVrijemeDist(rng);
                            aktivnost.kraj      =      aktivnost.pocetak      +
uniform_real_distribution<double>(0.01, 24.0 - aktivnost.pocetak)(rng);
        aktivnosti.push_back(aktivnost);
    }

    return aktivnosti;
}
```

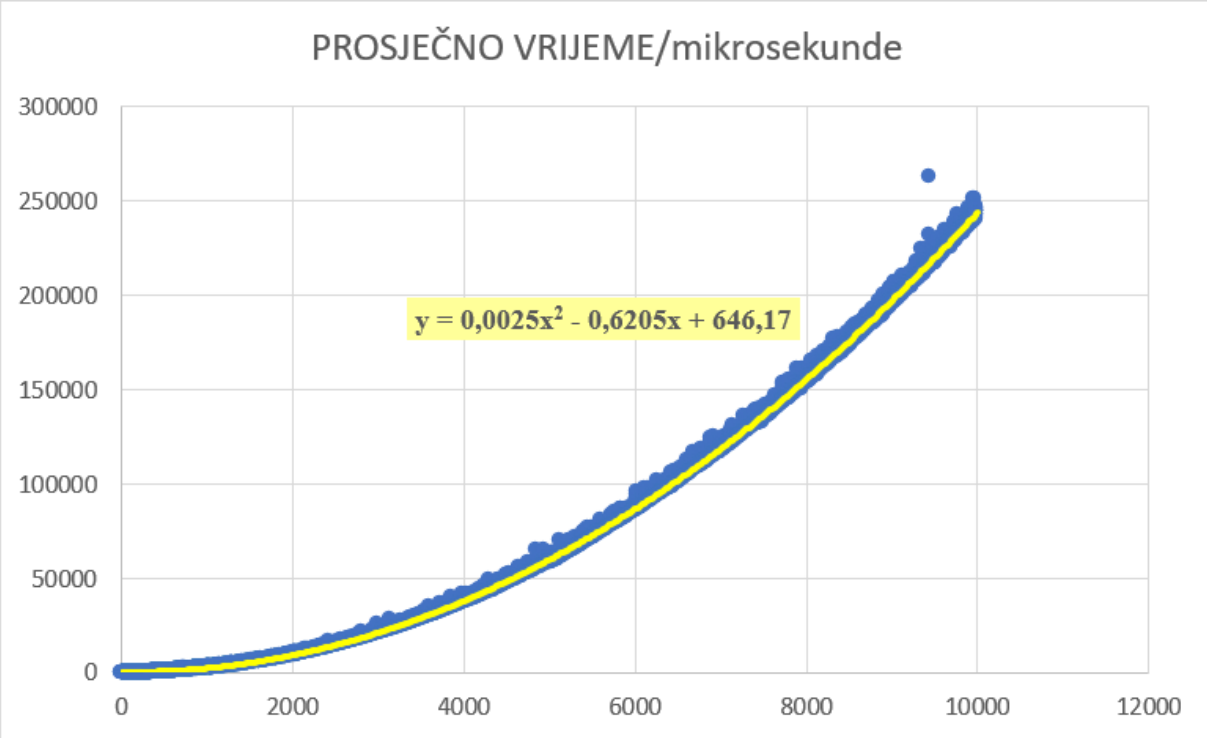
Slika 6: funkcija generirajRandomAktivnosti

mt19937 je generator brojeva pseudoslučajnih (slučajnih brojeva koje generiraju računalni algoritmi) Mersenne Twister koji pruža c++ Standardna biblioteka. (https://cplusplus.com/reference/random/mt19937/) **rng** je instanca generatora **mt19937** koja će se koristiti za generiranje slučajnih brojeva, a **static_cast<unsigned int>(time(nullptr))** postavlja sjeme generatora na trenutno vrijeme u sekundama od epohe time(nullptr) i castano je u oblik unsigned int. **uniform_real_distribution<double>** je klasa iz C++ Standardne biblioteke koja generira slučajne realne brojeve (tipa double) iz uniformne distribucije. (https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution) Odabrala sam ovu distribuciju jer se najčešće koristi kada želimo generirati slučajno vrijeme u satima.

Dakle, **aktivnost.pocetak = pocetnoVrijemeDist(rng);** generira slučajno vrijeme početka aktivnosti. To se postiže korištenjem distribucije **pocetnoVrijemeDist** koja je prethodno definirana i generira realne brojeve u rasponu od 0.0 do 23.0 pomoću generatora **rng**. Linija **aktivnost.kraj = aktivnost.pocetak + uniform_real_distribution<double>(0.01, 24.0 - aktivnost.pocetak)(rng);** generira slučajno vrijeme završetka aktivnosti. Prvo se koristi **uniform_real_distribution<double>** za stvaranje distribucije slučajnih realnih brojeva u rasponu od 0.01 do razlike između 24.0 i početnog vremena aktivnosti. Zatim se koristi **rng** za generiranje stvarnog slučajnog broja iz ove distribucije, a rezultat se dodaje početnom vremenu aktivnosti kako bi se odredilo završno vrijeme aktivnosti.

4.1. Dinamičko programiranje – empirijska analiza

Pokretala sam program 10 puta na brojevima aktivnosti od 10 do 10 000 i određivala vrijeme izvršavanja za funkciju activitySelection_Dynamic. Izračunala sam njihove prosječne vrijednosti u programu Excel te u istom nacrtala graf. Na grafu 1 prikazana su prosječna vremena izvršavanja funkcije u mikrosekundama i linija trenda odnosno polinom 2.stupnja koji najbolje opisuje podatke. Iz grafa, a i iz funkcije koja najbolje opisuje podatke jasno vidimo da je naša teorijska analiza složenosti O(n^2) jednaka i u empirijskoj analizi.



Graf 1: Prosječno vrijeme funkcije activitySelection_Dynamic na n = 10,…., 10000

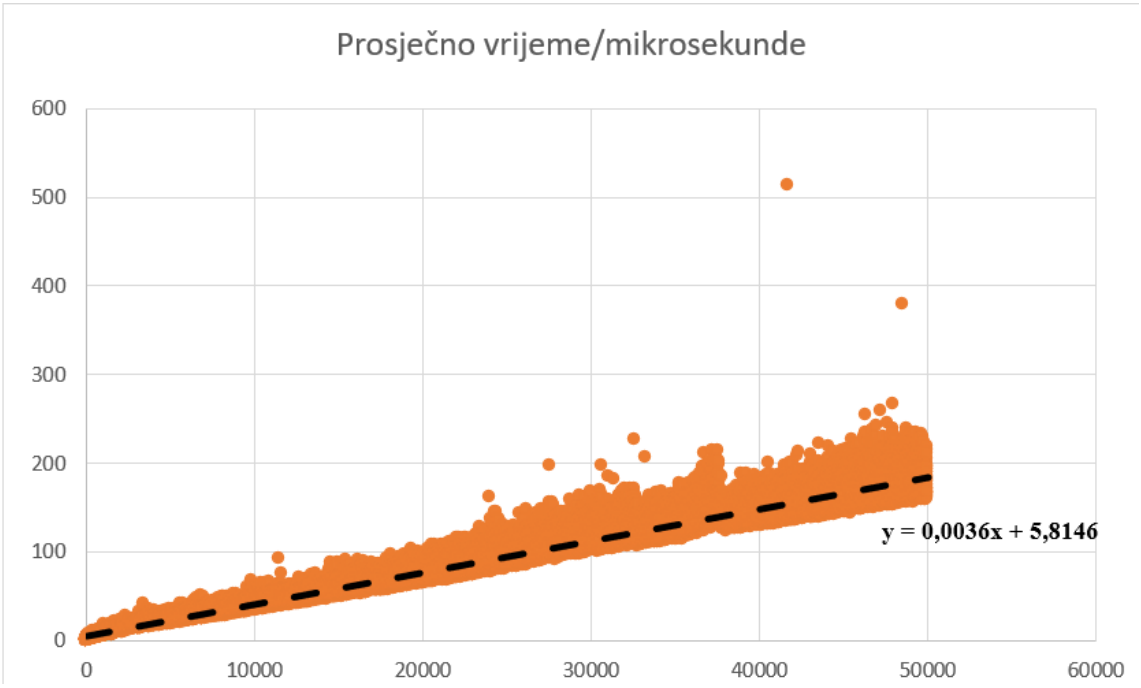
```
 Generirane sortirane aktivnosti:
1.aktivnost, pocetak: 4.95 kraj: 6.70
2.aktivnost, pocetak: 4.57 kraj: 8.45
3.aktivnost, pocetak: 2.86 kraj: 11.62
4.aktivnost, pocetak: 12.46 kraj: 14.64
5.aktivnost, pocetak: 5.39 kraj: 15.58
6.aktivnost, pocetak: 11.30 kraj: 15.62
7.aktivnost, pocetak: 19.57 kraj: 21.41
8.aktivnost, pocetak: 21.01 kraj: 22.24
9.aktivnost, pocetak: 16.06 kraj: 22.51
10.aktivnost, pocetak: 22.72 kraj: 23.13

 broj odabranih aktivnosti = 4
Vrijeme izvrsavanja: 2 microseconds.
Vrijeme izvrsavanja zapisano u 'DY_vrijeme_izvrsavanja_n10.csv'
```

Slika 7: primjer ispisa programa activity_dynamicit.cpp za n = 10

4.2. Pohlepni pristup – empirijska analiza

Pokretala sam program 10 puta na brojevima aktivnosti od 10 do 50 000 i određivala vrijeme izvršavanja za funkciju greedy_activity_selector. Također sam kao i za prošli graf izračunala njihove prosječne vrijednosti vremena izvršavanja u Excelu te u istom nacrtala graf. Na grafu 2 prikazan su prosječna vremena izvršavanja u mikrosekundama i linija trenda, odnosno pravac koji najbolje opisuje podatke. Iz grafa, vidimo da ima linearni rast pa je time jasno da je prosječna složenost algoritma O(n) što je jednako dobivenoj složenosti u teorijskoj analizi.

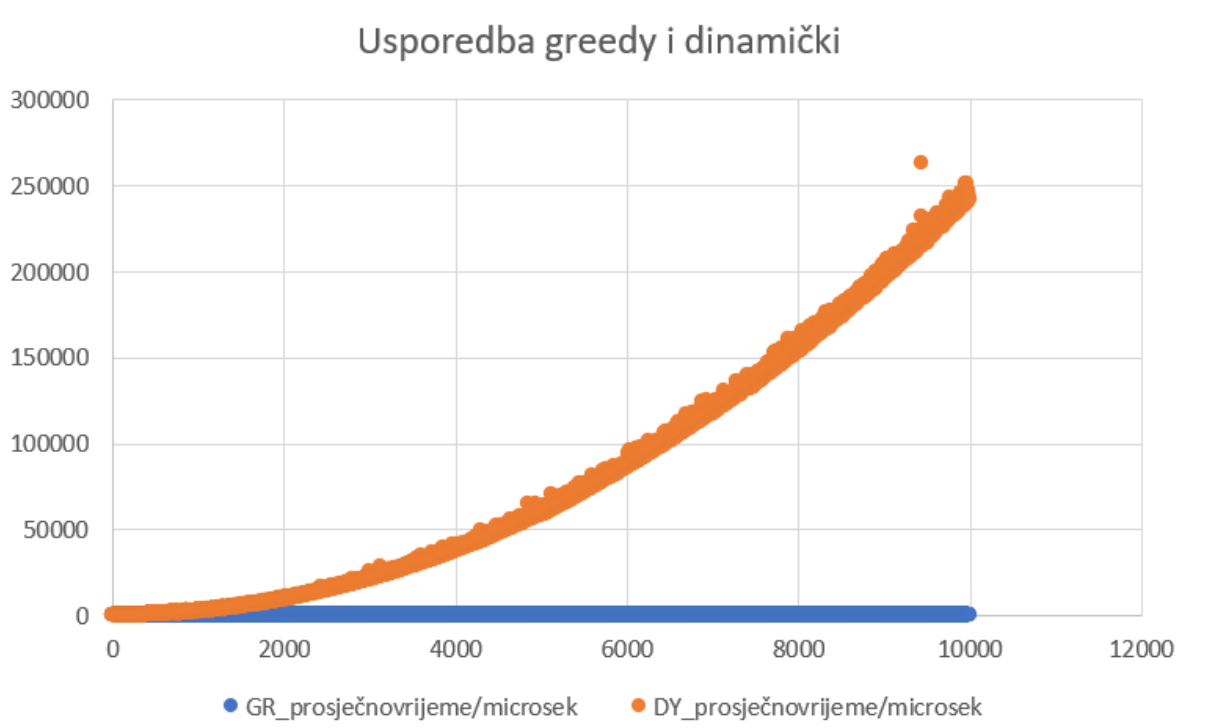


Graf 2: Prosječno vrijeme funkcije greedy_activity_selector na n = 10,…., 10000

```
 Generirane sortirane aktivnosti:
1.aktivnost, pocetak: 7.00 kraj: 10.79
2.aktivnost, pocetak: 9.63 kraj: 12.02
3.aktivnost, pocetak: 5.82 kraj: 13.86
4.aktivnost, pocetak: 14.59 kraj: 15.36
5.aktivnost, pocetak: 4.27 kraj: 15.98
6.aktivnost, pocetak: 3.65 kraj: 16.84
7.aktivnost, pocetak: 12.63 kraj: 19.78
8.aktivnost, pocetak: 18.57 kraj: 22.57
9.aktivnost, pocetak: 22.70 kraj: 23.69
10.aktivnost, pocetak: 17.48 kraj: 23.77

 broj odabranih aktivnosti = 4
Odabrane aktivnosti:
1.aktivnost, pocetak: 7.00 kraj: 10.79
2.aktivnost, pocetak: 14.59 kraj: 15.36
3.aktivnost, pocetak: 18.57 kraj: 22.57
4.aktivnost, pocetak: 22.70 kraj: 23.69
Vrijeme izvrsavanja: 2 microseconds.
Vrijeme izvrsavanja zapisano u 'vrijeme_izvrsavanja_n10.csv'
```

Slika 8: primjer ispisa programa activity_iterative.cpp za n = 10



Graf 3: Pokazuje prosječna vremena složenosti na istom grafu za n = 10,…, 10000 aktivnosti

5. ZAKLJUČAK:

Pokazali smo da za problem izabira aktivnosti vrijedi princip optimalnosti pa se on može riješiti dinamičkim programiranjem sa algoritmom vremenske složenosti O(n^2). Zatim smo pokazali da postoji bolji pohlepni pristup koji govori da možemo uzastopno odabrati aktivnost koja prva završava, uzeti u optimalno rješenje samo aktivnosti kompatibilne s njom (s kojima se ne preklapa) i ponavljati postupak dok nam ne ponestane aktivnosti i da taj algoritam ima puno bolju vremensku složenost O(n). Također u implementaciji pohlepni algoritam daje i optimalni niz aktivnosti tj. neki maksimalni podskup kompatibilnih aktivnosti od početnog skupa aktivnosti, dok dinamički algoritam daje samo kardinalitet tog maksimalnog podskupa odnosno maksimalan broj aktivnosti koje se mogu izvoditi bez preklapanja.

6. LITERATURA:

Cormen, T. H., Leiserson C. E., Rivest, R. L., Stein, C. Introduciton to algorithms 4th edition
https://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html
https://www.baeldung.com/cs/convert-recursion-to-iteration
https://helloacm.com/understanding-tail-recursion-visual-studio-c-assembly-view/
https://cplusplus.com/reference/random/mt19937/
https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution
Mario Skočić: „Generiranje pseudoslučajnih brojeva i testovi slučajnosti“ (Diplomski rad)