SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Pronalazak najduljeg puta u grafu korištenjem linearnog programiranja

Heidi Sokolovski

Zagreb, Svibanj 2023.

Sadržaj

Uvod....1

1. Oblikovanje grafa....2

1.1. Pretraživanje u dubinu....2

1.1.1. Riječ predstavlja vrh....2

1.1.2. Grupirane riječi predstavljaju vrh....3

1.2. *Integer* linearno programiranje....4

1.2.1. Riječ predstavlja brid....4

2. *Integer* linearno programiranje....6

2.1. Kompleksnost....6

2.2. Forma problema ILP....7

2.2.1. Model za igru ulančavanja riječi....7

2.3. Pronalaženje najduljeg niza....10

3. Programska izvedba....12

3.1. DFS....12

3.1.1. *groups.py*....12

3.1.2. *kalodont_dfs.py*....12

3.2. ILP....13

3.2.1. *graph.py*....13

3.2.2. *solver.py*....13

3.2.3. *algorithms.py*....15

3.2.4. *words.py*....16

3.2.5. *kalodont.py*....17

3.2.6. *verify.py* 17

3.2.7. *playy.ipynb* 17

3.3. Rezultati 17

3.3.1. DFS 17

3.3.2. ILP 18

Zaključak 20

Literatura 21

Slike 21

Pisana djela 21

Sažetak 22

Summary 23

Uvod

Problem najduljeg puta u grafu se proučava u igri ulančavanja riječi poznatom kao *Kalodont*. Potrebno je pronaći najdulji niz ulančanih riječi. Riječi se ulančavaju na način da se posljednja dva slova trenutne riječi podudaraju s početna dva slova sljedeće riječi. Igra završava ako igrač ne može nastaviti niz [1].

Najdulji niz u igri ulančavanja riječi gdje je igra modelirana kao usmjereni graf je zapravo pronalazak najduljeg puta u tom grafu. Igra ulančavanja riječi jedan je od mnogih primjera gdje je problem pronalazak najduljeg puta primjenjiv. Još neki od primjera su pronalazak ruta među gradovima u prometnim mrežama i pronalazak najdulje zajedničke sekvence između dvije DNA molekule.

Problem se rješava na više različitih načina te se zatim njihova učinkovitost međusobno uspoređuje. Graf se kreira prema pravilima igre te mijenja obzirom na način rješavanja problema. Najprije se koristi način pretraživanja grafa u dubinu, a zatim *integer* linearno programiranje. Očekivanje je da će pronalazak najdulje sekvence *integer* linearnim programiranjem biti brže od rješavanja problema pretraživanjem grafa u dubinu.

U nastavku rada detaljnije se opisuju metode korištene u rješavanju problema te se proučavaju dobiveni rezultati.

1. Oblikovanje grafa

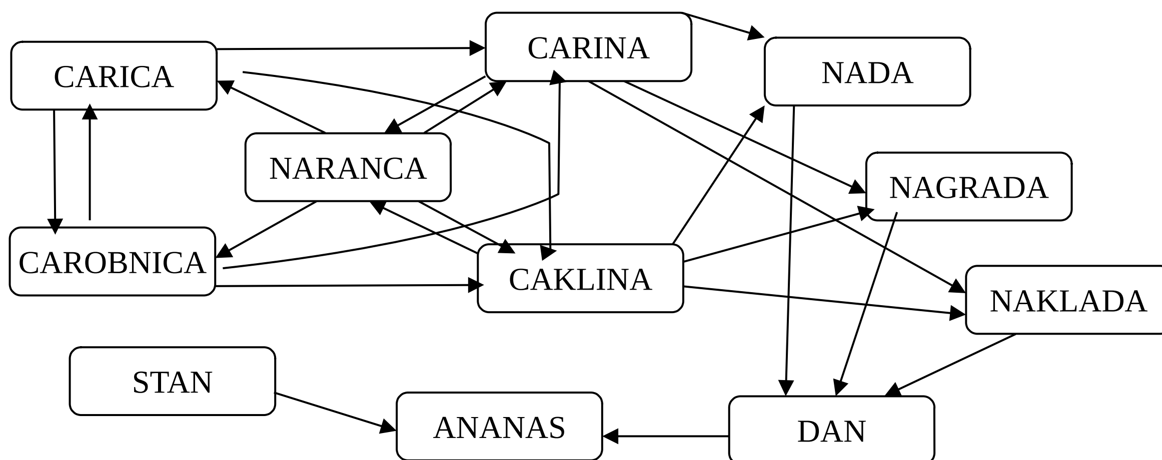
Najprije je potrebno pretvoriti riječi iz zadanog rječnika u usmjereni graf. Tek će tada biti moguće pronaći najdulji put. Put u grafu predstavlja konačan slijed bridova u kojem su svaka dva uzastopna brida susjedna, a svi vrhovi te stoga i svi bridovi su različiti [2].

1.1. Pretraživanje u dubinu

Pretraživanje u dubinu (engl. *depth-first search, DFS*) je algoritam za obilazak ili pretraživanje grafa. Algoritam počinje od početno zadanog čvora to jest vrha u grafu i istražuje što je dalje moguće prije vraćanja unazad. Potrebna je dodatna memorija za praćenje koji čvorovi su bili posjećeni.

1.1.1. Riječ predstavlja vrh

Direktan dizajn grafa iz igre ulančavanja riječi je onaj koji svaku riječ smatra posebnim vrhom koji je povezan s ostalim vrhovima pomoću usmjerenih bridova prema pravilima igre.

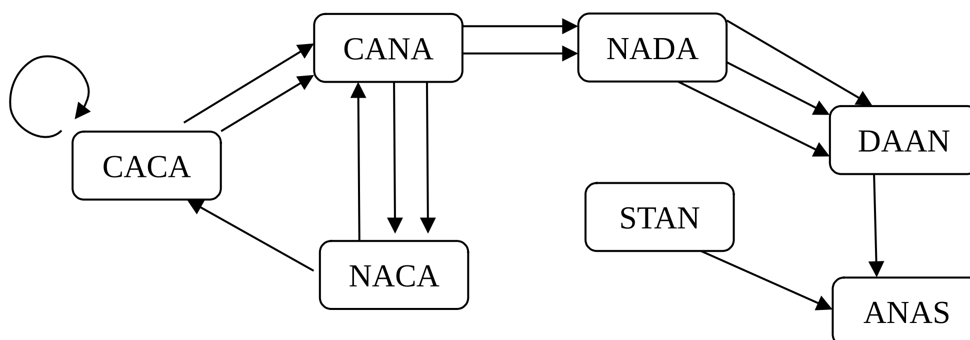


Sl. 1.1 Primjer grafa u kojemu je svaka riječ poseban vrh

Međutim, zato što se igra služi rječnikom hrvatskog jezika, taj način dizajna predstavlja prevelik broj vrhova i bridova. Potrebno je osmisliti drugačiji dizajn grafa.

1.1.2. Grupirane riječi predstavljaju vrh

Kako bi smanjili broj vrhova, riječi koje počinju i završavaju na ista dva slova se grupiraju. Bridovi u grafu predstavljaju prijelaze iz jedne u drugu grupu riječi.



Sl. 1.2 Primjer grafa u kojemu su riječi grupirane ovisno kojim slovima počinju odnosno završavaju

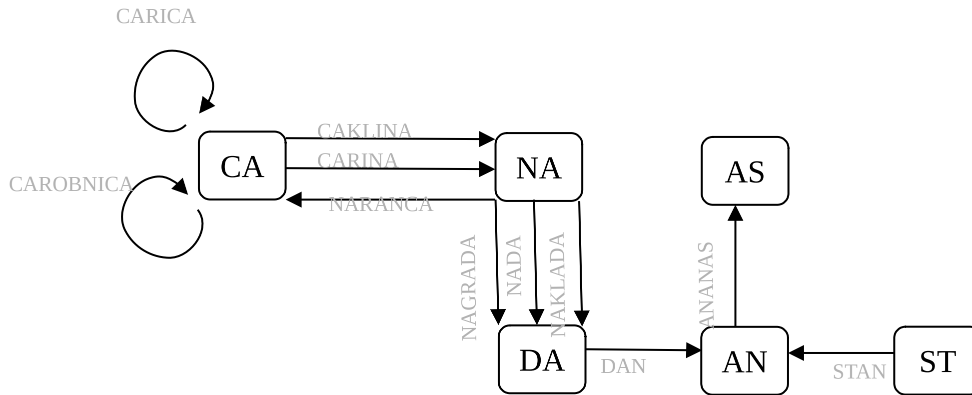
Zbog te promjene, više se ne traži najdulji put, već najdulja staza. Staza u grafu predstavlja konačan slijed bridova u kojem su svaka dva uzastopna brida susjedna, a svi bridovi su različiti [2].

1.2. *Integer* linearno programiranje

1.2.1. Riječ predstavlja brid

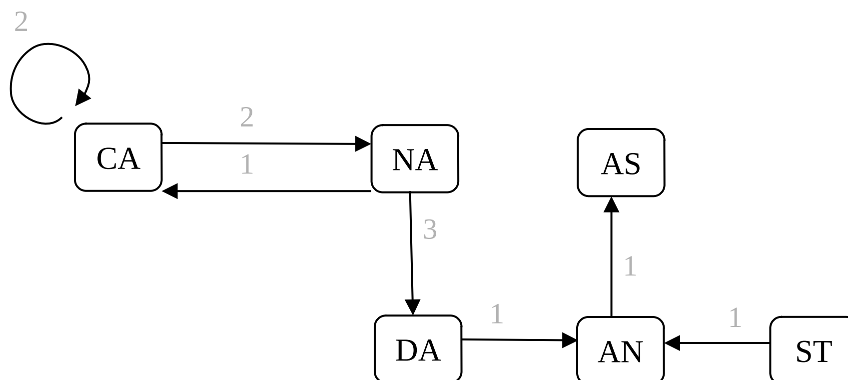
Graf potreban za *integer* programiranje je graf prikazan na Sl. 1.1 gdje pojedina riječ predstavlja vrh. Međutim, moguće ga je pojednostaviti. Početna odnosno posljednja dva

slova riječi se grupiraju. Nastale grupice predstavljaju vrhove grafa. Pojedine riječi su predstavljene bridovima.



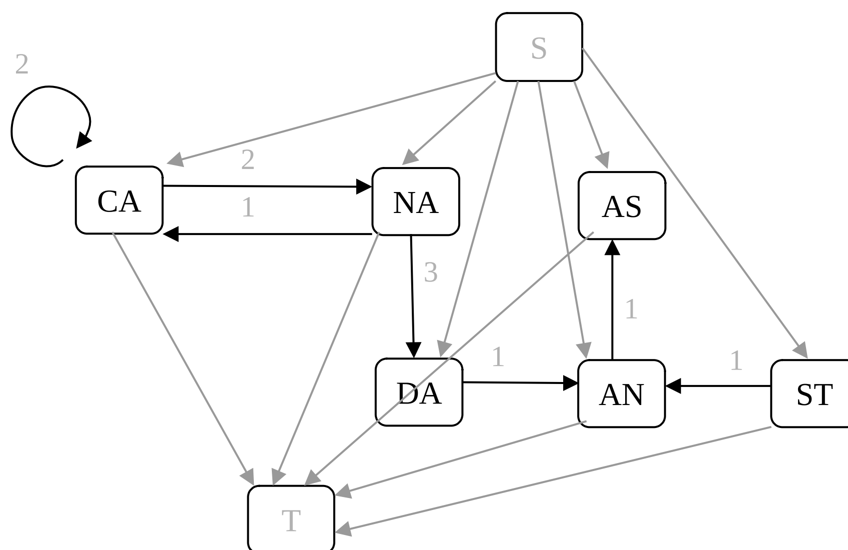
Sl. 1.3 Primjer grafa u kojemu je svaka riječ poseban brid

Daljnje pojednostavljivanje se postiže grupiranjem riječi čime graf postaje težinski. Broj bridova između vrhova predstavlja težinu na novostvorenom težinskom grafu.



Sl. 1.4 Primjer grafa u kojemu težina brida predstavlja broj riječi

Posljednji korak kod oblikovanja početnog grafa je dodavanje dva dodatna vrha koji će predstavljati početni i završni vrh staze koja će se kasnije koristiti u konstrukciji najduljeg niza riječi. Traži se najdulja staza, a ne put zbog mogućnosti prolaženja istim vrhom više puta. Početni vrh, S (engl. *super-source*) je usmjereno povezan sa svakim vrhom u grafu na način da ima sve izlazne bridove. Završni vrh, T (engl. *super-sink*) je slično povezan sa svakim vrhom, ali on ima sve ulazne bridove [3]. Bridovi *super* vrhova imaju težinu jedan.



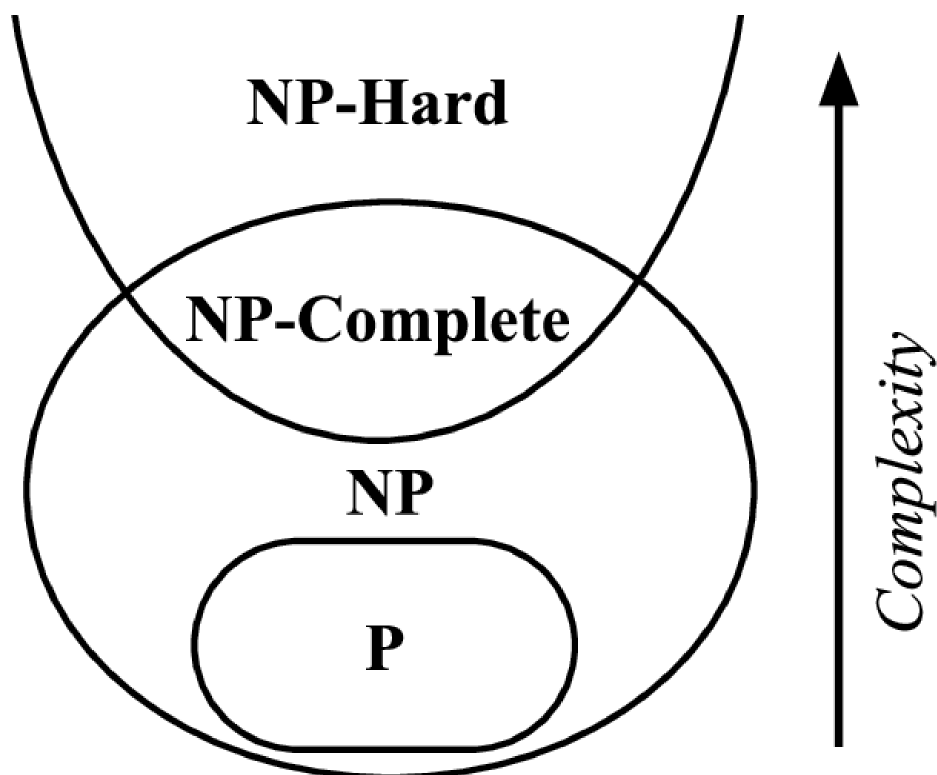
Sl. 1.5 Prikaz vrhova *super-source (S)* i *super-sink (T)* na primjeru grafa

2. *Integer* linearno programiranje

Integer linearno programiranje (ILP) je vrsta linearnog programiranja (LP). To je program matematičke optimizacije u kojemu su varijable ograničene da budu cijeli brojevi. Ciljna funkcija (engl. *objective function*) i ograničenja (engl. *constraints*) su linearna [4].

2.1. Kompleksnost

ILP je NP-potpun problem. N dolazi od izraza nedeterministički Turing strojevi, način da se *brute-force* algoritam pretraživanja matematički formalizira. P dolazi od izraza polinomno vrijeme, $O(n^k)$. Potpun označava da se sve može simulirati u istoj klasi složenosti. Točnost rješenja može se provjeriti u polinomnom vremenu. NP-potpun problemi su složeniji od NP problema [5].



Sl. 2.1 Prikaz dijagrama presjeka P, NP, *NP-complete* i *NP-hard* problema

2.2. Forma problema ILP

Forma ILP problema je prikazana sljedećom formulom:

$$\textit{Maksimiziraj} \quad \sum_{j=1}^{n} c_j x_j$$

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad (i=1,2,\ldots,m)$$

$$x_j \leq 0 \quad (j=1,2,\ldots,n)$$

$$x_j \in \mathbb{Z}$$

2.2.1. Model za igru ulančavanja riječi

Za problem opisan grafom iz poglavlja 1.2.1., forma je prikazana formulom [3]:

$$\textit{Maksimiziraj} \quad \sum_{i \in V \cup \{s\}} \sum_{j \in V \cup \{t\}} x_{ij}$$

$$\sum_{i \in V} x_{si} = 1$$

$$\sum_{i \in V} x_{ij} - \sum_{j \in V} x_{ji} = 0$$

$$\sum_{i \in V} x_{it} = 1$$

$$0 \leq x_{ij} \leq w_{ij}$$

$$x_{ij} \in \mathbb{Z}$$

Skup *V* je skup svih vrhova u grafu dok su *s* i *t* posebni vrhovi *super-source* i *super-sink*. Varijabla x_{ij} predstavlja usmjereni brid iz vrha *i* u vrh *j* između svaka dva povezana vrha skupa *V*. Vrijednost w_{ij} je težina brida x_{ij}.

Ciljna funkcija koja je prikazana kao suma svih bridova treba se maksimizirati uz navedena ograničenja. Ograničenja su sljedeća:

1. Zbroj svih bridova iz vrha *s* u vrhove skupa *V* treba biti jedan.
2. Zbroj svih ulaznih bridova vrha skupa *V* treba biti jednak zbroju izlaznih bridova.
3. Zbroj svih bridova iz vrhova skupa *V* u vrh *t* treba biti jedan.
4. Bridovi poprimaju vrijednosti od nula do svoje težine.
5. Svi bridovi su cijeli brojevi.

Ograničenja 1 i 3 označavaju da postoji samo jedan ulazni odnosno izlazni brid iz vrhova *s* i *t*.

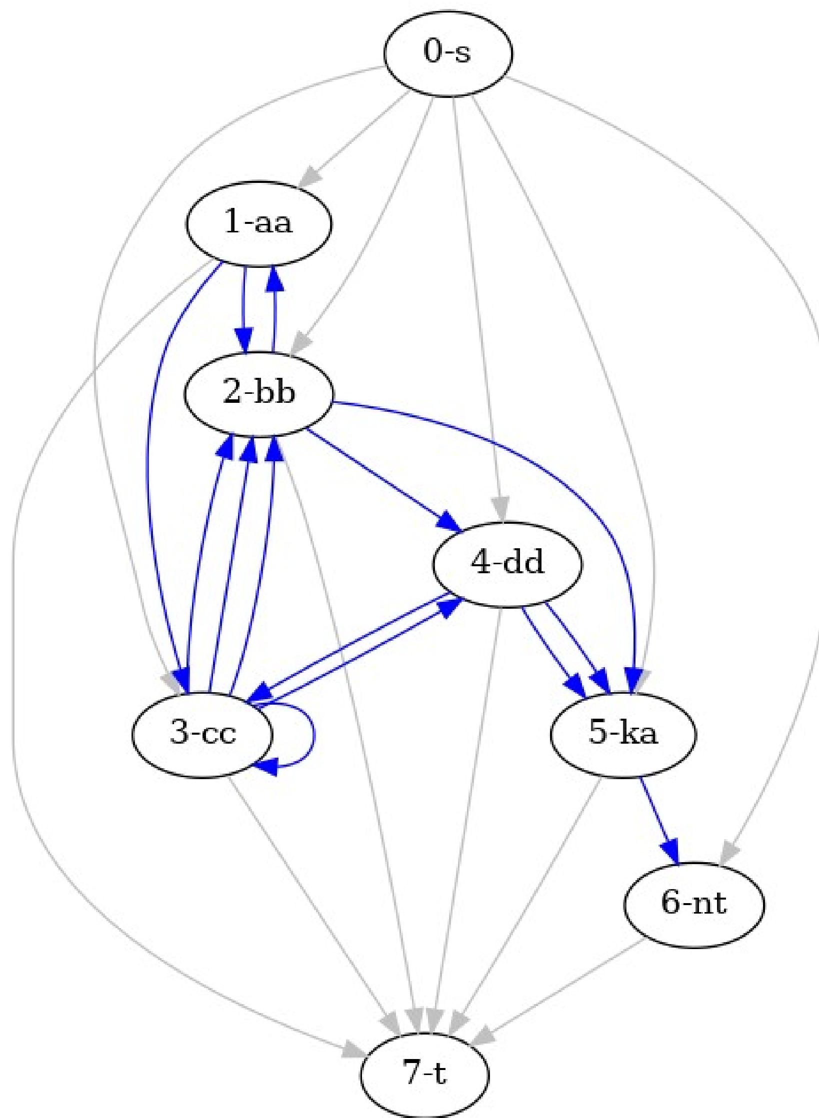
Ograničenje 2 označava da svaki posjećeni vrh skupa *V* ne smije biti niti početni niti završni vrh.

Ograničenje 4 označava da brid ne može imati negativnu težinu jer težina predstavlja stvarni broj bridova. Isto tako nije moguće dodavati nove nepostojeće bridove provođenjem optimizacije.

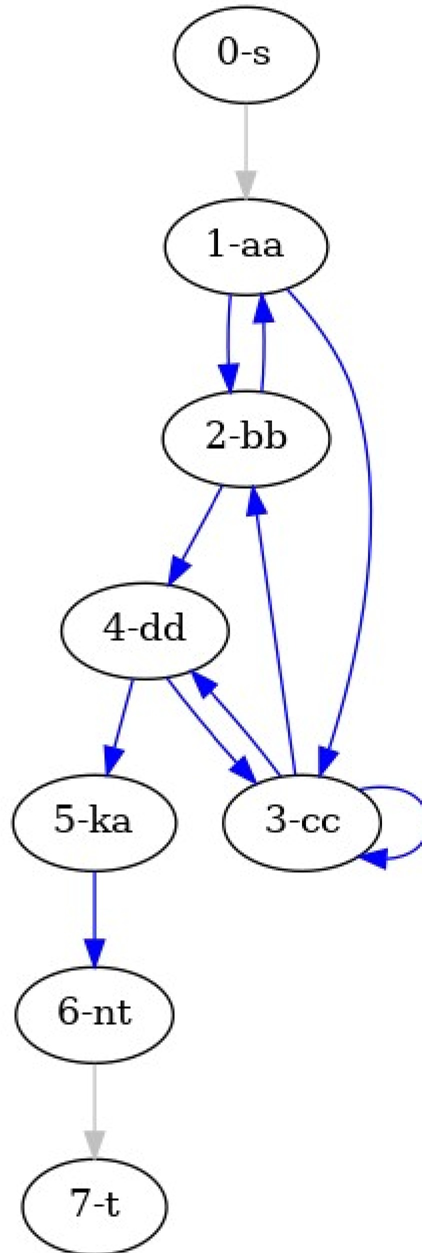
Ograničenje 5 je razlog zašto se koristi ILP, a ne LP. Pošto je broj bridova opisan težinama, moraju se koristiti cijeli brojevi.

Bridovi tipa x_{ii} takozvane petlje se izbacuju iz jednadžbe.

Maksimiziranjem ciljne funkcije dobit će se vrijednost varijable brida x_{ij} koja će predstavljati novu težinu. Ako je vrijednost nula, brid uklanja. Kreira se novi graf obzirom na dobiveno rješenje te se vraćaju petlje zanemarene kod postavljanja jednadžbi.



Sl. 2.2 Primjer grafa prije optimiziranja (nacrtani graf nije težinski)



Sl. 2.3 Primjer grafa nakon optimiziranja (nacrtani graf nije težinski)

2.3. Pronalaženje najduljeg niza

Nakon što se provede optimizacija nad grafom, rješenje je najveći povezan semi-Eulerski podgraf. Tada se može pronaći najdulja staza i izgraditi najdulji niz riječi.

U pisanom radu [3] se iz optimiziranog grafa vade najmanji ciklusi tako da na kraju ostane glavna staza iz vrha *s* u vrh *t*. Nakon ekstrakcije ciklusa prolazi se glavnom stazom i na nju se dodaju vrhovi izvađenih ciklusa ako odabrani ciklus sadrži trenutni vrh.

U ovom seminaru koristi se malo drugačija metoda pronalaženja najdulje staze.

Počinje se od vrha *s* i optimizirani graf se obilazi algoritmom pretraživanja u dubinu. Staza završava vrhom *t*. Nakon što se pronađe tražena staza, bridovi se zamijenjuju odgovarajućim riječima iz rječnika. Konačno rješenje je najdulji put u originalnom grafu gdje svaka riječ predstavlja vrh.

Najdulja staza grafa sa slike Sl. 2.3 je:

['s', 'aa', 'bb', 'aa', 'cc', 'cc', 'bb', 'dd', 'cc', 'dd', 'ka', 'nt', 't'].

Ako su riječi rječnika sljedeće:

aa1bb, bb1aa, aa1cc, cc1bb, cc2bb, cc3bb, cc1dd, dd1cc, bb1dd, bb1ka, kalodont, dd1ka, dd2ka, cc1cc,

najdulji put je:

['aa1bb', 'bb1aa', 'aa1cc', 'cc1cc', 'cc3bb', 'bb1dd', 'dd1cc', 'cc1dd', 'dd2ka', 'kalodont'].

3. Programska izvedba

Program za pronalazak najduljeg puta u grafu je napisan u programskom jeziku Python 3.8.10 te su korištene sljedeće biblioteke:

- NumPy
- SciPy (poziva HiGHS [6])
- JupyterLab
- PyGraphviz

Korišteno računalo ima procesor Intel® Core™ i7-6700HQ.

Čitav kod se može vidjeti na *Github* stranici u repozitoriju *kalosolver* (poveznica: <https://github.com/applepie-heidi/kalosolver>).

3.1. DFS

3.1.1. *groups.py*

Datoteka grupira riječi iz rječnik datoteke koje počinju te završavaju na ista dva ASCII znaka. Funkcija vraća rječnik koji kao ključ ima grupu od četiri ASCII znaka i vrijednost sve grupirane riječi za tu grupu.

3.1.2. kalodont_dfs.py

U datoteci se nalaze dvije klase: *Node* i *Graph*.

Node predstavlja čvorove grafa. Atributi *Node* klase su ime čvora, koliko puta se čvor smije posjetiti i lista čvorova koji se mogu posjetiti iz trenutnog čvora.

Graph predstavlja graf oblika sa slike Sl. 1.2. Klasa *Graph* ima metodu *kalodont()* koja poziva rekurzivnu funkciju koja obavlja modificirani DFS algoritam. Kako prolazi kroz čvorove, ažurira vrijednost *Node* objekta koliko puta se još čvor smije posjetiti. Pamti najdulji niz riječi iz igre te ga zamijenjuje duljim ako je dulji niz pronađen. Zapisuje ga u datoteku.

Bilo je potrebno povećati gornju granicu memorije stoga.

```
resource.setrlimit(resource.RLIMIT_STACK,
(2 ** 29, -1))
sys.setrecursionlimit(10 ** 6)
```

3.2. ILP

3.2.1. graph.py

Datoteka se sastoji od klase *Graph*. Njeni atributi su rječnik koji mapira imena čvorova na brojeve, lista čvorova i lista rječnika koji sadrže bridove za svaki čvor.

```
self._name_to_node = {}
self._nodes = []
self._edges = []
```

Klasa također ima metode za dodavanje čvorova i bridova, modificiranje bridova i dohvaćanje vrijednosti atributa i ostalih informacija.

3.2.2. solver.py

Datoteka se sastoji od klase *WordGameModel*. Njeni atributi su proslijeđeni graf, matrica koja predstavlja sustav jednadžbi ograničenja, težine bridova, lista *tuple*-ova u kojima se bilježe ulazni i izlazni čvorovi za pojedine varijable te rezultat koji će se dobiti optimiziranjem.

```
self.graph = graph
self.matrix = [[0] * edge_cnt for _ in range(node_cnt)]
self.costs = []
self.variables = []
self.res = None
```

Kod inicijaliziranja se gradi matrica i puni koeficijentima iz sustava jednadžbi prikazanom u izrazu 2. Ćelije koje predstavljaju petlje imaju vrijednost nula.

Zatim postoji metoda (kod 3.1) koja odrađuje optimizaciju odnosno maksimizira ciljnu funkciju uz zadana ograničenja iz izraza 1. Koristi se bibliotekom SciPy. Vraća rješenje provedene optimizacije.

```
def solve(self):
    edges_num = len(self.matrix[0])
    objective = np.array([1] * edges_num)
    integrality = np.array([1] * edges_num)

    bounds = optimize.Bounds(lb=np.array([0] * edges_num),
ub=np.array(self.costs))

    cons_s = optimize.LinearConstraint(self.matrix[0],-1,-1)
    cons_i = optimize.LinearConstraint(self.matrix[1:-1],0,0)
    cons_t = optimize.LinearConstraint(self.matrix[-1],1,1)
    constraints = [cons_s, cons_i, cons_t]

    self.res = optimize.milp(
        c=-objective,
        constraints=constraints,
        integrality=integrality,
        bounds=bounds,
    )
    return self.res
```

Kod 3.1 Optimiziranje uz SciPy

Dobiveno rješenje koristi se u izradi novog grafa. U klasi postoji metoda koja gradi i vraća novi optimizirani graf. Ona vraća u prethodnom koraku izbačene petlje.

3.2.3. algorithms.py

Datoteka sadrži jednu funkciju koja iz optimiziranog grafa vadi najdulju stazu pomoću DFS algoritma (kod 3.2).

Počinje se iz čvora *s*. Stvara se rječnik koji će pamtiti posjećene bridove, stog s početnom vrijednosti *tuple* čvora *s* i *iterator* izlaznih bridova čvora *s*. Dok stog nije prazan obavlja se pretraga u dubinu i ažurira varijabla za pohranu staze.

```
s = graph.node("s")
t = graph.node("t")

trail = []
visited_edges = {}
stack = [(s, iter(graph.edges(s)))]
while stack:
    node, edges = stack[-1]
    try:
        next_node, weight = next(edges)
        remaining_weight = visited_edges.setdefault((node,
next_node), weight)
        if remaining_weight > 0:
            visited_edges[(node, next_node)] -= 1
            stack.append((next_node,
iter(graph.edges(next_node))))
            trail.append((node, next_node))
    except StopIteration:
        stack.pop()
```

Kod 3.2 Izrada staze koristeći DFS algoritam

Kreirana staza će biti sljedećeg oblika (ovo je nasumični primjer):

[(s, aa), (aa, bb), (bb, cc), (cc, dd), (dd, aa), (aa, t),

(aa, aa),

(cc, ee), (ee, ff), (ff, gg), (gg, cc),

(ee, ee)].

Potrebno ju je pretvoriti u sljedeće segmente:

[[s, aa, bb, cc, dd, aa, t],

[aa],

[cc, ee, ff, gg],

[ee]].

To se postiže unutarnjom funkcijom za dodavanje segmenata.

```
def add_segment(seg):
    if seg[-1] == t:
        segments.append(seg)
    else:
        assert seg[0] == seg[-1]
        segments.append(seg[:-1])
```

Funkcija se poziva u nastavku originalne funkcije.

Na kraju se pomoću pronađenih segmenata gradi najdulja staza zadanog optimiziranog grafa.

3.2.4. words.py

Datoteka se sastoji od tri funkcija potrebnih za baratanje s riječima.

Prva funkcija učitava riječi iz zadane datoteke rječnika i vraća njihovu listu.

Druga funkcija služi za kreiranje grafa iz prethodno učitanih riječi. Razdvaja svaku riječ na dvije grupe od prva odnosno zadnja dva ASCII znaka te ih dodaje kao čvorove u grafu.

Povezanost riječi se bilježi pomoću bridova grafa. Također dodaju se *super-source* i *super-sink* čvorovi i povezuju s ostalim čvorovima grafa. Funkcija vraća stvoreni graf.

Zadnja funkcija koristi se za konstrukciju niza pomoću proslijeđene liste riječi, grafa i najdulje staze. Ona vraća listu koja predstavlja najdulji put u grafu, to jest najveći broj ulančanih riječi.

3.2.5. kalodont.py

U ovoj se datoteci pozivaju sve potrebne funkcije kako bi se iz zadanog rječnika pronašao najdulji niz po pravilima igre Kalodont. Također se u *log*-ovima ispisuju međurješenja. Konačni niz se upisuje u datoteku imena „*best_recursive_<trenutni timestamp>.txt*".

3.2.6. verify.py

Ova datoteka sadržava funkciju koja je potrebna za verifikaciju točnosti dobivenog rezultata. Ispisuje na ekran riječ „ *CORRECT*" ako je niz riječi korektno generiran. Točan niz ne sadrži duplikate i riječi su lančano povezane tako da su zadnja dva slova trenutne riječi jednaka kao prva dva slova sljedeće.

3.2.7. playy.ipynb

Služi za testiranje programa na manjim primjerima i crtanje grafova pomoću biblioteke PyGraphviz. Grafovi na slikama Sl. 2.2 i 2.3 su nacrtani pomoću koda iz ove datoteke.

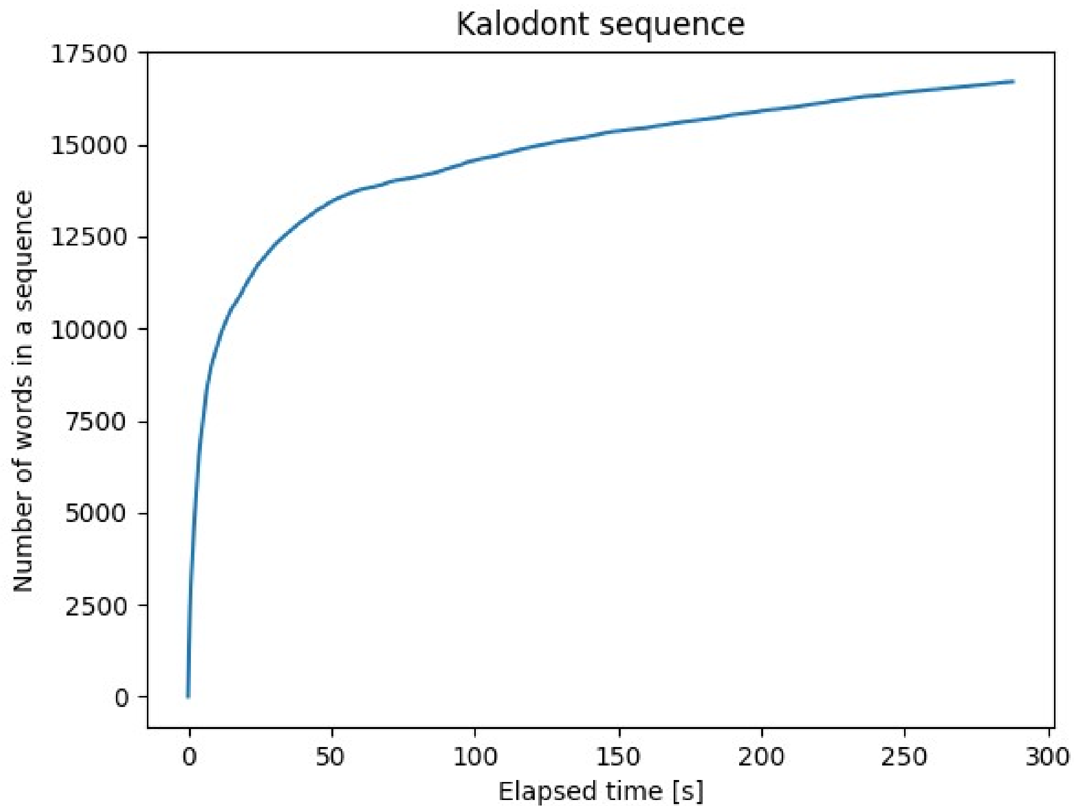
3.3. Rezultati

Zadani datoteka početnog rječnika koji se koristi u potrazi za najduljim putem sastoji se od 159 836 riječi.

3.3.1. DFS

Brute-force način se u nekoliko dana nije uspio pomaknuti s ~18 900 riječi.

To ukazuje na algoritam velike vremenske kompleksnosti. Brute-force metoda uobičajeno nije pravi način za rješavanje problema. Ovdje se lagano može vidjeti da slijepo pretraživanje grafa nije dobra ideja za problem ovolike složenosti bez obzira na to koliko se graf pojednostavni.



Sl. 3.1 Grafički prikaz pronalaženja duljeg niza riječi kroz proteklo vrijeme (~5 minuta)

3.3.2. ILP

Za razliku od prethodnog načina, optimizacija grafa pomoću ILP je začuđujuće brza.

U malo više od 2 sekunde uspije se pronaći niz od 26 552 riječi što je također i najdulji mogući niz odnosno put u grafu.

U sljedećoj tablici su prikazane prosječne vrijednosti trajanja pojedine radnje.

Stvaranje grafa	0.12 s
Maksimiziranje ciljne funkcije	1.19 s
Stvaranje optimiziranog grafa	0.02 s
Pronalazak najdulje staze	0.58 s
Stvaranje niza riječi	0.06 s
Cjelokupno izvođenje programa	2.15 s

Tablica 3.1 Prikaz trajanja dijelova programa u sekundama

Zaključak

U seminaru su bile isprobane dvije metode pronalaska najduljeg puta u grafu, DFS i ILP.

Nakon pojednostavljivanja grafova, pokazalo se da je ILP jako brz i točan način za pronalazak najduljeg puta optimiziranjem grafa. Kod DFS načina se usprkos dugom trajanju programa nije ni stiglo pronaći najdulje moguće rješenje.

Za složene grafove s velikim brojem vrhova i bridova koji se mogu izraziti u ILP formi se isplati koristiti ILP za pronalazak najduljeg puta. Pretpostavlja se da slično vrijedi i za standardno linearno programiranje gdje varijable nisu ograničene da budu cijeli brojevi.

Literatura

Slike

[2.1] Diagram of intersection among classes P, NP, NP-complete and NP-hard problems, Poveznica: <https://www.researchgate.net/figure/Diagram-of-intersection-among-classes-P-NP-NP-complete-and-NP-hard-problems_fig13_336890186>

Pisana djela

[1] OptimoRoute Inc., shaolin.dev, 2021., Poveznica: <https://shaolin.dev/>

[2] Bender, E. A., Williamson, S. G., *Lists, Decisions and Graphs. With an Introduction to Probability*, 2010.

[3] Inui, N., Shinano, Y., Kounoike, Y., Kotani, Y., *Solving the Longest Word-Chain Problem, Tokyo University of Agriculture and Technology*, 2004., str. 214-221

[4] Nemhauser, G. L., Wolsey L. A., *Integer and combinatorial optimization*, 1988.

[5] Garey, M. R., Johnson, D. S., *Computers and Intractability. A Guide to the Theory of NP-Completeness*, 1979.

[6] Parallelizing the dual revised simplex method, Q. Huangfu and J. A. J. Hall, *Mathematical Programming Computation*, **10** (1), 119-142, 2018. DOI: 10.1007/s12532-017-0130-5

Sažetak

Pronalazak najduljeg puta u grafu korištenjem linearnog programiranja

Rad se bavi uporabom *integer* linearnog programiranja (ILP) kod problema pronalaženja najduljeg puta u grafu. Proučavaju su metode pojednostavljivanja početnog grafa. Objašnjava se kako upotrijebiti ILP te dobiveni optimizirani graf pretvoriti u traženo rješenje. Uspoređuje se učinkovitost korištenja ILP i pretraživanja grafa algoritmom pretraživanja u dubinu (DFS).

Ključne riječi: *integer* linearno programiranje, najdulji put, najdulja staza, teorija grafa, optimizacija grafa, pretraživanje u dubinu, igra riječi, Kalodont

Summary

Finding the Longest Path in a Graph Using Linear Programming

This paper deals with the use of integer linear programming (ILP) in the problem of finding the longest path in a graph. Methods of simplifying the starting graph are studied. It is explained how to use ILP and convert the resulting optimized graph into the required solution. The efficiency of using ILP and searching the graph using depth-first search (DFS) is compared.

Keywords: integer linear programming, longest path, longest trail, graph theory, graph optimization, depth-first search, word chain game, Kalodont