Математичка гимназија

МАТУРСКИ РАД

Из предмета Програмирање и програмски језици

Алгоритми за решавање проблема из теорије графова

Ученик:

Магдалина Јелић, IVд

Ментор:

Јелена Хаџи-Пурић

У Београду, мај 2020

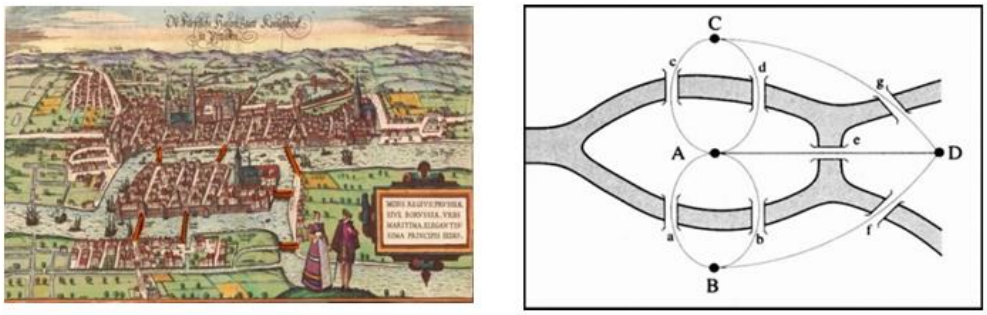
С А Д Р Ж А Ј

1 Увод

Графове као моделе објеката и веза међу њима често препознајемо у свакодневном животу. Могу да представе различите ситуације, а користе се у разним областима од математике до социјалне психологије. Мрежа путева, односи међу људима (познанства, пријатељства…), али и многи други појмови могу бити представљени уз помоћ графова. Структурне формуле молекула у хемији и шеме електричних кола су само неки од примера како су ови математички објекти леп приказ реалног света.

1.1 Историјат

Кенигзбершки мостови. Први проблем и његово решење које представља претечу теорије графова изнесено је у једном раду швајцарског математичара Леонарда Ојлера под називом Седам мостова Кенигзберга који је објављен 1736. године. Кроз Кенигзберг, данашњи Калињинград протиче река Прегел која у самом средишту града обилази око два велика острва. Како би се река повезала са левом и десном обалом, али и острва међусобно пре много векова изграђено је седам мостова као на слици 1.



Слика 1: Кенигзбергшки мостови.

Проблем који је био занимљив мештанима дошао је до Ојлера једном када је он посетио овај град. Питање је било да ли је могуће кренути из било које тачке на обали и проћи све мостове тачно једном.

Леонард Ојлер је представио обале као чворове графа, а мостове као гране и доказао да није могуће обићи све гране тачно једном. Решавањем овог наизглед једноставног проблема започета је нова математичка дисциплина, теорија графова, а доказ да је немогуће наћи решење узима се и као први доказ из математичке области топологије.

Проблем четири боје. Проблем четири боје тврди да је могуће сваку географску карту обојити са четири боје тако да свака држава буде обојена једном од ових боја, а да суседне државе буду обојене различитим бојама (под суседним подразумевамо државе које имају заједничку граничну линију).

Дуго је овај проблем био један од најпознатијих нерешених проблема из теорије графова зато што је заинтересовао многе научнике да покушају да га докажу. Кенет Апел и Волфганг Хекел су 1976. године уз значајану помоћ рачунара и претходне

теоријске резултате решили проблем, односно доказали да је могуће обојити сваку карту на описани начин. Занимљиво је да је тада уложено око 1200 часова рачунарског времена. Данас је неизвесно да ли је могуће доказати проблем четири боје без употребе рачунара.

1.2. Графови у информационим технологијама

Гугл мапе су представљене гранама које означавају путеве и чворовима који су у пресеку свака два пута. На тај начин систем за навигацију користи алгоритме за проналазак најкраћих путева и обезбеђује кориснику најоптималнију руту.

Друштвене мреже Графови представљају и основу на којој раде друштвене мраже. На Фејсбуку су корисници, али и постови, места и фотографије представљени различитим врстама чворовима, а разни односи између корисника од пријатељстава до коментара на сликама представљени су различитим гранама које су и саме комплексне структуре у којима се чува много информација. Тако је теорија графова и овде неопходна (нпр. постоји алгоритам који проналази особе које се вероватно познају и предлаже пријатељства).

Светска комуникациона мрежа (world wide web скраћено www) може се представити тако што странице третирамо као чворове, а усмерена грана између чворова *u* и *v* постоји ако постоји линк од странице *u* до странице *v*. Рангирање веб страница приликом претраге користи графовске алгоритме. Претраживање веб страница (енгл. Page rang algorithm) је алгоритам који разматра колико је страница важна на основу линкова од других страница ка њој. Дакле важније странице добијају више линкова од других страница. Ово није једини, али је први и најпознатији алгоритам који користи претрага на вебу.

2 Основни појмови и представљање графова

2.1 Основни појмови и дефиниције

- Граф је уређен пар $G=(V,E)$ где је V скуп чворова или темена, а E скуп грана или ивица при чему гране представљају релације између чорова.

- Граф може бити орјентисан и неорјентисан. У неорјентисаном графу гране су неуређени парови чворова (грана $(v->u)$ је исто што и грана $(u->v)$). У орјентисаном графу гране су уређени парови чворова ($(v->u)$ и $(u->v)$ представљају различите гране).

- Подграф $G1$ графа G је граф чији скуп чворова је подскуп скупа чворова графа G и чији скуп грана је подскуп скупа грана графа G.

- Степен чвора у ознаци *d(v)* је број грана суседних чвору *v* (односно број грана које тај чвор повезују са неким другим).

- У орјентисаним графовима разликујемо улазни и излазни степен. Улазни степен чвора *v* представља број грана за које је *v* крај, док је излазни степен број грана за које је *v* почетак.

- Пут је низ чворова $v_1,v_2,....,v_k$ повезаних гранама (v_1,v_2), $(v_2,v_3),...,(v_{k-1},v_k)$ таквим да су сви чворови и све гране међусобно различити.

- Циклус је пут на коме се први и последњи чвор поклапају.

- Тежински граф је граф код кога је свакој грани придружен неки реалан број, односно вредност.

- Комплетан граф је онај код којег између свака два чвора постоји грана.

- Повезан неорјентисан граф је онај у коме између свака два чвора постоји пут.
  Компонентом повезаности називамо скуп чворова таквих да за свака два чвора *v* и *u* из скупа важи да постоји пут од *v* до *u*.

- Стабло је повезан граф без циклуса.
  Особине стабла:

  -Стабло са *n* чворова има тачно *n-1* грану.

  -Између било која два чвора у стаблу постоји јединствен пут који их спаја.

- Шума је граф који се састоји од више стабала, то јест свака компонента повезаности шуме је стабло.

- Коренско стабло је усмерено стабло са једним посебно издвојеним чвором (кореном) и све гране су усмерене од корена.

- Бипартитни граф је граф чији се чворови могу поделити на два дисјунктна подскупа тако да у графу постоје само гране између чворова из различитих подскупова.

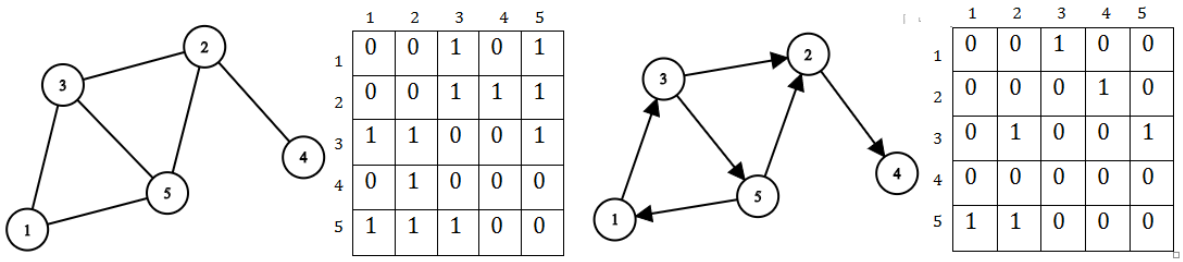
2.2 Представљање графова

Граф можемо представити на више начина. Описаћемо два најпознатија.

Матрица суседства. Најчешћа имплементација подразумева матрицу суседства (повезаности). Уколико имамо *n* чворова нумерисаних од *1* до *n* у матрици димензија *n x n* на пољу *(i,j)* налази се информација о постојању гране од чвора *i* до чвора *j* (*1* ако

постоји грана, иначе *0*). У случају тежинског графа у матрици суседства можемо чувати и информацију о тежини гране или ознаку да не постоји.

На слици 2 су приказане матрице суседства у неорјентисаном и орјентисаном графу.



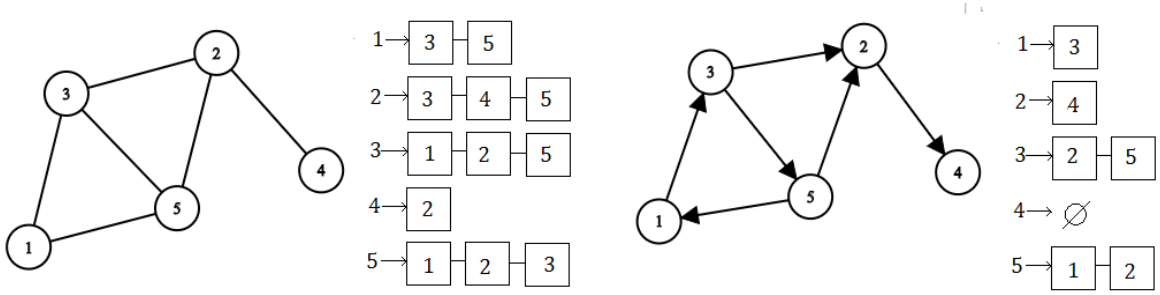
Слика 2: Матрице суседства неорјентисаног и орјентисаног графа.

Уочавамо да је у случају неорјентисаних графова матрица симетрична у односу на главну дијагоналу.

Предности и мане. Предност овог типа представљања графа је брз приступ информацији о повезаности два чвора, у сложености *O(1)*, док је слабост велика меморијска сложеност, чак $O(|V|^2)$ која чини овај тип представљања неефиканим у случају графова са великим бројем чворова и релативно малим бројем грана.

Листа суседства. Граф можемо представити набрајањем суседа сваког чвора, односно за сваки чвор можемо чувати само списак његових суседа. Најлакше се имплементира коришћењем структуре вектор (нпр. структура *vector* STL C++). Пошто нам за сваки чвор треба вектор користићемо низ вектора. На слици 3 приказане су листе суседства за неорјентисан и орјентисан граф.

Предности и мане. Предност овог типа представљања графа је мала меморијска сложеност *O(|V|+|E|)*, док је мана што у сложености *O(d(v))* добијамо информацију о постојању гране између два чвора.



Слика 3: Листе суседства неорјентисаног и орјентисаног графа.

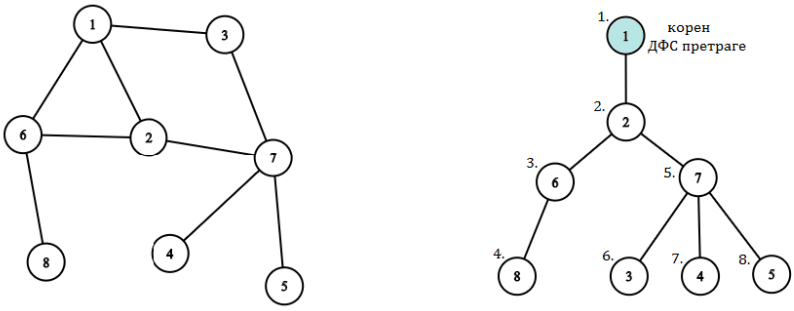
3 Алгоритми претраге графа

Уопштено о претрази графа. Претрага графа представља обилазак свих чворова тако што се крећемо гранама графа а да притом сваки чвор обиђемо тачно једном. Чворове обилазимо по одређеном правилу у зависности од ког разликујемо више типова претраге. Два основна алгоритма за обилазак графа су претрага у дубину и претрага у ширину.

3.1 Претрага у дубину (Depth first search)

Као скица за разумевање овог алгоритма претраге може послужити тражење излаза из лавиринта. Напредујемо даље остављајући траг све док имамо где да се крећемо, а онда се враћамо назад.

Овај тип обиласка креће од полазног чвора *v*, корена претраге у дубину, обележава да је посећен и за њега позива функцију за обилазак. У функцији за обилазак произвољно се бира један од чворова суседних са *v* који још није посећен рецимо *u*. Чвор *u* се даље маркира као посећен и за њега се рекурзивно позива функција за обилазак. По извршењу тог позива уколико постоји још неки непосећен чвор повезан са *v* за њега се понавља наведени поступак, све док сви суседи чвора *v* нису посећени. На крају, након свих рекурзивних позива посећена је цела једна компонента повезаности. Обиласком у дубину повезаног графа са *n* чворова добија се стабло које називамо ДФС-стабло које има значајну примену у разним алгоритмима и задацима. На слици 4 дат је граф са његовим ДФС-стаблом и означеним редоследом посећивања чворова.



Слика 4: Граф и његово ДФС стабло.

Реализација. Претрага у дубину је по природи рекурзиван процес па ћемо овде објаснити такву имплементацију, али је потребно напоменути да постоји и нерекурзивна верзија овог алгоритма.

Прво се полазни чвор обележава као посећен, и ставља на стек. Даље, све док се стек не испразни ради се једна од следеће две акције:

-уколико чвор који тренутно анализирамо има непосећених суседа један од њих се маркира као посећен и убацује у стек.
-уколико су сви суседи чвора посећени он се скида са стека.

Када се стек испразни значи да не постоји више ниједан непосећен чвор у датој компоненти повезаности и алгоритам се завршава.

Сложеност. Увиђамо да сваку грану прегледамо два пута, по једном са оба краја. Нека је *d(u)* ознака за степен чвора *u*. Укупно време за све позиве је:

$$\sum_{u\in V} O(1+d) = O(\sum_{u\in V}\big(1+d(u)\big) = O(\sum_{u\in V} 1 + \sum_{u\in V} d(u)) = O(|V| + 2|E|) = O(|V| + |E|)$$

Дакле временска сложеност ДФС алгоритма је *O(|V|+|E|)*. Меморијска сложеност је (у случају чувања графа уз помоћ листе суседства) такође *O(|V|+|E|)*.

Примена

Пример1. Наћи компоненте повезности датог графа.

Задатак решавамо вишеструким позивањем ДФС функције. Редом пролазимо све чворове од *1* до *n* и сваки пут када наиђемо на још непосећен чвор започињемо ДФС обилазак. Уочавамо да скуп чворова које на тај начин посетимо сваким обиласком представља по једну компоненту повезаности.

Пример2. Пронаћи циклус у графу, уколико постоји или детектовати да не постоји.

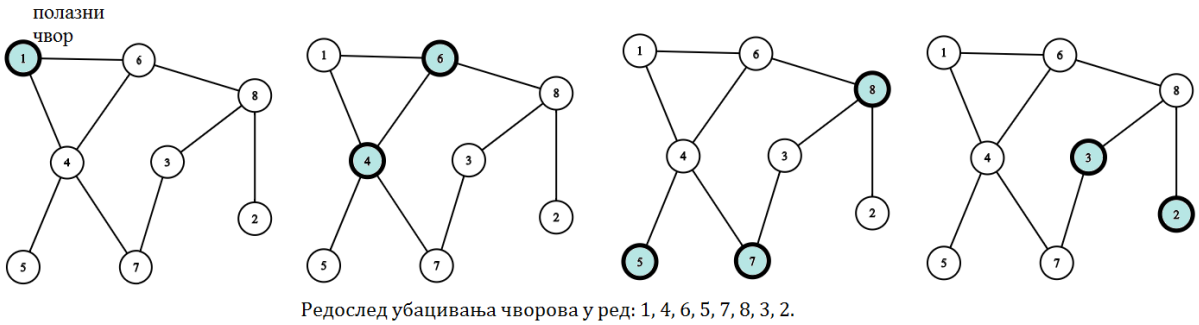
Уколико у графу постоји циклус он ће бити откривен када у ДФС обиласку наиђемо на грану која води од чвора *v* који тренутно анализирамо до неког већ посећеног чвора *u* (различитог од чвора који је родитељ чвору *v* у дфс-стаблу). Уколико желимо да испишемо редом чворове који чине циклус морамо се вратити уназад крећући се (преко родитеља) од чвора *v* до чвора *u*.

Пример 3. Испитати да ли је дати граф бипартитиван.

Подсетимо се да је граф бипартитиван уколико је могуће поделити скуп чворова у два подскупа тако да свака грана спаја чворове из различитих подскупова. У почетку потребно је да произвољни полазни чвор означимо као припадника првог подскупа. Даље обиласком у дубину сваки нови чвор *u* убацујемо у онај скуп из ког није чвор *v* из ког се позива обилазак за *u*. Граф није бипатритиван ако у неком тренутку наиђемо на грану која повезује два чвора из истог подскупа, у сопротном је бипартитиван.

3.2 Претрага у ширину (Breadth first search)

Претрагом у ширину обилазимо граф ниво по ниво, то јест прво се обилазе чворови који су најближи полазном чвору, затим њима најближи… Полазећи од једног чвора најпре обилазимо све његове директне суседе који још нису посећени, а након тога настављамо од тих суседа обилазак на исти начин. На слици 5 дати су нивои БФС претраге графа.



Слика 5: Нивои БФС претраге графа.

Реализација. За разлику од обиласка графа у дубину БФС алгоритам није по природи рекурзиван, и укратко се описује следећим корацима:

1. Почињемо претрагу из полазног чвора који обележавамо као посећен и стављамо у ред.

2. Избацујемо први чвор из реда, пролазимо кроз све његове суседе који још нису посећени, затим их маркирамо као посећене и додајемо их у ред.

3. Понављамо корак 2 све док се ред не испразни.

Коначно када је ред празан то значи да су сви чворови (до којих је могуће доћи поласком из почетног чвора) обиђени и алгоритам се завршава. Напоменимо да описани процес користи структуру података која се зове ред (структура queue STL C++) али да се на сличан начин може имплементирати и уз коришћење низа.

Сложеност. Временска и меморијска сложеност (слично као код ДФС алгоритма) су $O(|V|+|E|)$.

Примена

Пример 1. За дати граф потребно је пронаћи колико је најкраће растојање (одређено бројем грана) између чворова v и u.

Задатак решавамо БФС обиласком графа са почетним чвором v и притом у помоћном низу пратимо растојање између чвора v и чвора који тренутно анализирамо. Када наиђемо на чвор u можемо обуставити претрагу и исписати тражено растојање.

4 Тополошко сортирање

Претпоставимо да постоји неки скуп послова које треба завршити такав да једни зависе од других, односно да неке послове можемо да започнемо тек када завршимо неке друге. Потребно је пронаћи такав редослед којим ћемо радити послове тако да за сваки посао важи да је започет тек када су сви послови од којих он зависи завршени. Налажење тог редоследа назива се тополошко сортирање графа у коме сваки чвор представља један посао, а усмерена грана од *v* ка *u* значи да је посао представљен чвором *u* могуће радити тек након завршеног посла који је представљен са *v*.

Прво потребно је наћи услове када граф има тополошки редослед. Јасно је да је неопходан услов да граф не садржи циклусе (у супротном послове који чине циклус никад не би било могуће урадити). Покажимо следећу лему.

Лема. Усмерен ацикличан граф увек има чвор са улазним степеном нула.

Доказ. Претпоставимо супротно, да сви чворови у произвољном усмереном ацикличном графу имају улазни степен већи или једнак *1*. Тада бисмо увек могли да идемо даље „уназад“ (кретање супротно од смера у коме су гране) и пошто је број чворова у графу коначан у неком тренутку бисмо (оваквим проласком) дошли у неки од чворова који смо већ прошли. То би значило да у графу постоји циклус што је у контрадикцији са условом да је посматрани граф ацикличан.

Даље можемо показати индукцијом по броју чворова да је сваки ацикличан усмерен граф могуће тополошки сортирати.

Постоје два основна начина како наћи тополошки редослед чворова графа.

4.1 Канов алгоритам

Овај алгоритам први је описао Артур Кан 1962. године.

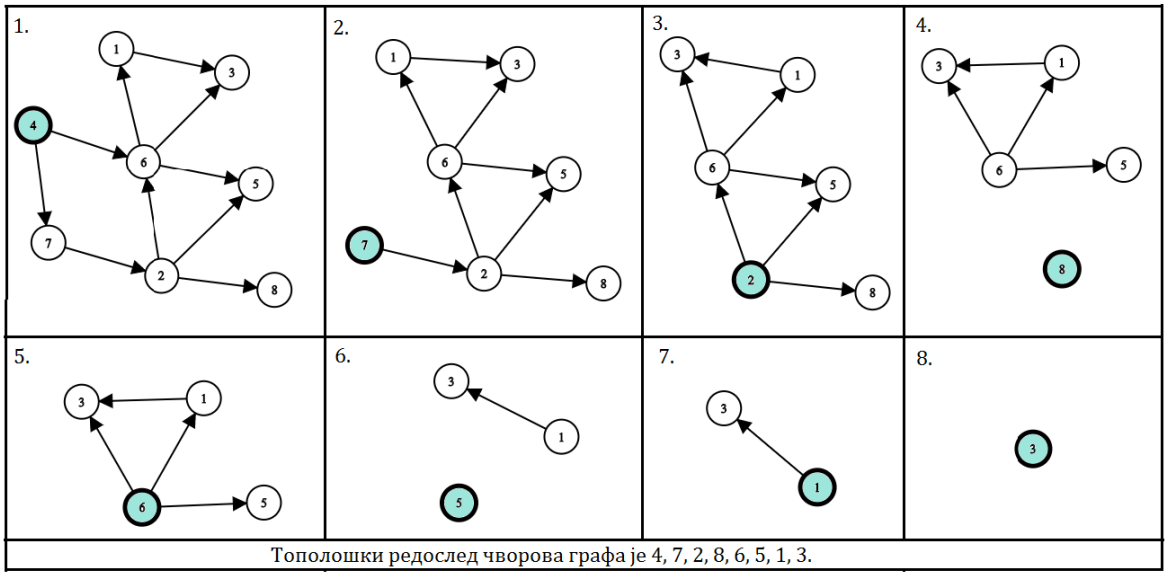
Реализација. Нека је дат усмерен ацикличан граф.

1. Израчунамо улазни степен за све чворове у графу (у низу *us[]* чувамо улазни степен за све чворове графа) и убацимо у ред све чворове са улазним степеном *0*.
2. Избацујемо чвор из реда, а затим свим његовим суседима смањујемо улазни степен за *1* и у ред се уписују они чворови којима улазни степен постаје *0*.
3. Поновља се корак *1* све док се ред не испразни.

Тополошки редослед графа је редослед убацивања чворова у ред.

Уочимо да за неке графове тополошки редослед није јединствен, и тада зависи од редоследа убацивања чворова у ред у тренутку када постоји више чворова улазног степена нула. Такође, на овај начин можемо да откријемо да граф нема тополошки редослед (уколико на почетку није сигурно да ли је дат ацикличан граф), и то тако што

након завршетка алгоритма, у том случају, постојаће чворови који никад нису били уписани у ред. На слици 6 дат је Канов алгоритам у корацима.



Слика 6: Канов алгоритам за тополошко сортирање графа.

Сложеност. Сложеност је $O(|V|+|E|)$ јер се сваки чвор једном убацује и једном избацује из реда, и још се свака грана пролази по једном када се чвор избацује из реда и тада се смањује степен свим његовим суседима.

4.2 Алгоритам заснован на ДФС-у

Други начин представља још једну примену већ описаног алгоритма претраге у дубину. Укратко, покрећемо обилазак у дубину из произвољног чвора и уписујемо у низ чворове по редоследу завршавања. Напоменимо да је потребно позвати више пута ДФС-функцију јер не морају сви чворови да буду уписани први пут због тога што је граф усмерен. Када се сви чворови упишу у низ тражени редослед је обрнут од оног којим су дати чворови у низу.

Сложеност. Сложеност овог начина ако граф памтимо као листу суседства је $O(|V|+|E|)$, због већ описане сложености ДФС алгоритма.

5 Алгоритми за налажење минималног повезујућег стабла

Минимално повезујуће стабло (minimum cost spanning tree) дефинише се на повезаном тежинском графу и то је било које стабло које је подграф датог графа са минималним могућим укупним збиром грана.

Размотримо два алгоритма за проналазак минималног покривајућег стабла.

5.1 Прим-Јарников алгоритам

Чешки математичар Војтех Јарник је 1930. године смислио овај алгоритам, а касније су до њега независно дошли и информатичари Роберт Прим и Едсгер Дајкстра. Најчешће се назива Примов алгоритам.

Следећим поступком добија се минимално повезујуће стабло повезаног тежинског графа. Напоменимо да се на сличан начин долази до минималне покривајуће шуме тако што се описани алгоритам понови за сваку компоненту повезаности.

Опис:

1. Бира се произвољан чвор v_1.

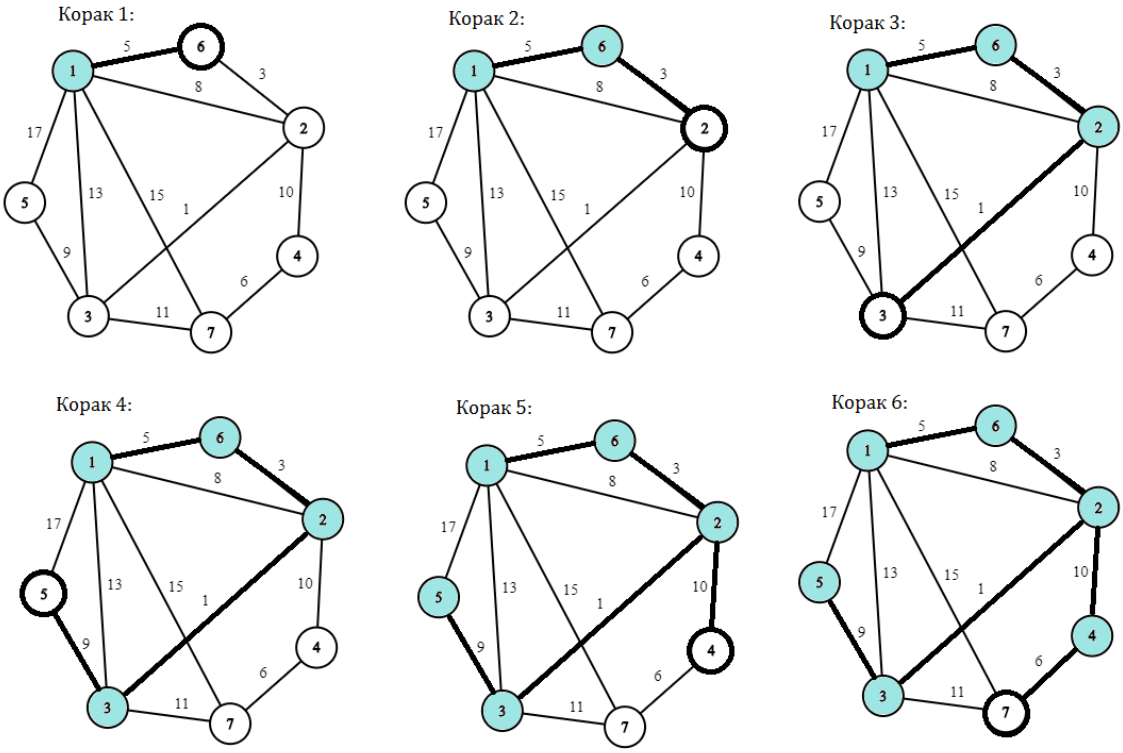
2.Од свих грана које полазе из чвора v_1 бира се она која има најмању тежину.

3.Ако су досад изабрани чворови $v_1,v_2,....,v_{k+1}$ и гране $e_1,e_2....e_k$ онда се нова грана e_{k+1} која спаја чворове v_i и v_j бира на следећи начин: v_i припада скупу $\{v_1,v_2....v_{k+1}\}$, а v_j не припада скупу $\{v_1,v_2,....,v_{k+1}\}$ и e_{k+1} је минималне тежине од свих таквих грана.

4.Уколико је изабрано $n-1$ грана алгоритам је завршен, у супротном понавља се корак 3.

На слици 7 је приказан пример налажења минималног покривајућег стабла Примовим алгоримом у корацима.

Реализација. За имплементацију користићемо приоритетни ред (priority queue STL C++), а могуће је користити и другу стуктуру података сличних особина на пример хип. У приоритетном реду налазе се гране такве да им је један крај у скупу изабраних чворова, поређане тако да на врху буде грана најмање тежине. На почетку ћемо убацити све гране са једним крајем у првом изабраном чвору. Осим тога у додатном низу логичког типа (bool) чуваћемо ознаку да ли је одређени чвор већ изабран или није. Редом ћемо разматрати грану по грану са врха приоритетног реда, и уколико испуњава услове корака 3 додавати чвор који још није у изабраном скупу. Описани низ је потребан јер у међувремену оба чвора (крајеви разматране гране) могу бити изабрани. У том случају само избацујемо грану из приоритетног реда и настављамо процес.



Слика 7: Примов алгоритам за налажење минималног покривајућег стабла.

Доказ коректности.

Нека је дат тежински повезан граф G са n чворова и T граф добијен описаним поступком. У сваком кораку алгоритма додаје се грана чији један крај припада подграфу (графу који чине изабране гране и чворови), а други не, па на тај начин никад не можемо добити циклус.У сваком тренутку подграф је повезан, па самим тим и када садржи свих n чворова. Дакле добијени граф је повезан и не садржи циклус. Тиме смо доказали да је граф добијен у Примовом алгоритму стабло.

Докажимо да је T минимално повезујуће стабло.

Нека је P следећа претпоставка: Ако је F подграф добијен у било ком стадијуму алгоритма онда постоји неко минимално покривајуће стабло целог графа G које садржи F.

Докажимо индукцијом да важи P.

База: Претпоставка P је тачна на почетку јер је тада F само један чвор, па постоји минимално покривајуће стабло јер сваки повезан тежински граф има минимално покривајуће стабло.

Индуктивна хипотеза: Претпоставимо да P важи након k корака, односно када F садржи k грана. Дакле за подграф F постоји неко минимално покривајуће стабло S, такво да је F подграф графа S. Докажимо да P важи након $k+1$ корака. Нека је након k корака добијен подграф F и e грана изабрана у последњем $k+1$ кораку.

Ако *S* садржи *e*, онда важи претпоставка *P*, јер због индуктивне хипотезе *S* садржи *F*, дакле *S* садржи *F+e*. Ако *F* не садржи *e*, онда у графу *S+e* постоји циклус. (*S* је минимално покривајуће стабло па садржи *n-1* грану, *S+e* садржи *n* грана, а повезан граф са *n* чворова и *n* грана садржи циклус). Један крај гране *e* припада *F*, други не припада, одакле закључујемо да у том циклусу постоји још једна грана чији један крај припада *F*, а други не припада. Обележимо је са *e1*. Грана *e1* не може имати мању вредност од *e* (тада би била изабрана раније у алгоритму), а не може бити ни већа од *e* јер тада *S* не би било минимално покривајуће стабло. Закључујемо да *S+e-e1* јесте минимално покривајуће стабло одакле следи да постоји минимално покривајуће стабло које садржи *F+e* па и у овом случају важи *P*.

Индукцијом смо доказали да претпоставка *P* важи у сваком стадијуму, па важи и након што се изабере *n-1* грана, што значи да је граф добијен Примовим алгоритмом заиста минимално покривајуће стабло задатог графа *G*.

Сложеност. У случају представљања графа матрицом суседства и линеарне претраге када се додаје нови чвор свеукупна сложеност је $O(|V|^2)$. За случај када је граф представљен листом сусуседства, а користи се бинарни хип сложеност је $O((|V|+|E|)log|E|)$. Ако користимо Фибоначијев хип тада се добија најбоља сложеност $O(|E|+|V|log|V|)$.

5.2 Структура дисјунктних скупова

Пре него што пређемо на опис Крускаловог алгоритма потребно је да упознамо структуру која ће чувати информацију о припадности чвора одређеном скупу.

Структура дисјунктних скупова (disjoint set data structure) је корисна техника која се примењује не само у графовским проблемима, већ и у многим другим областима и задацима из алгоритмике, зато ћемо је објаснити уопштено, а касније ће бити објашњена примена у Крускаловом алгоритму.

Проблем који се поставља је следећи:

Посматра се скуп који се састоји од више дисјунктних делова односно подскупова таквих да је пресек свака два подскупа празан скуп.

Потребно је обезбедити ефикасно извршавање два типа операција:
1. *nadji(x)*- проналази се подскуп којем припада конкретни елемент *x*
2. *spoj(x,y)*- обједињује два подскупа задата са по једним елементом, први са *x*, а други са *y*.

Реализација. Представимо сваки елемент једним чвором. Сваки од ових подскупова представљаће по једно усмерено коренско стабло. Корен тог стабла биће представник одговарајућег подскупа. За сваки чвор чува се његов родитељ у тако уређеном стаблу или ознака да је сам чвор корен то јест представник подскупа.

nadji(x) - Када је потребно наћи подскуп којем припада конкретан елемент иде се уназад преко родитеља све до корена стабла односно представника траженог подскупа. Имплементира се уз помоћ рекурзивне функције.

spoj(x,y) - Ако је потребно спојити два подскупа на почетку се морају наћи њихови представници, а затим се за родитеља једног од два представника поставља други који постаје корен новонасталог стабла у коме су сада сви чворови из оба подскупа.

Спајање по рангу. Уочавамо да није свеједно који ће од два представника бити нови корен јер се не добијају идентична стабла, а потребно је да добијена стабла имају што мању дубину због ефикасног извршавања операције *nadji*. Наиме што је већа дубина стабла то је време потребно за одређивање којем подскупу припада елемент веће. Једна врста оптимизације користи информацију о дубини стабла (за сваки чвор у низу *h[]* чувамо растојање до најудаљенијег потомка). Када спајамо скупове за корен се поставља онај чвор који има већу максималну дубину и на тај начин се дубина новонасталог стабла не повећава осим ако је вредност најудаљенијег потомка једнака за оба стабла.

Сажимање пута. Још једна корисна оптимизација је сажимање пута сваки пут када се позива функција *nadji(x).* Тада се за родитеља оног чвора за који се тражи корен (*x*) поставља сам корен. На тај начин се скраћује пут до корена свих будућих позива за оне чворове који су потомци траженог чвора *x*.

Сложеност. Роберт Тарјан први је доказао да је се сложеност ове методе (за једну операцију) са свим описаним оптимизацијама $O(\alpha(n))$ где је $\alpha(n)$ инверз Акерманове функиције која расте веома споро, па се може узети да је сложеност константа за сваку операцију.

Пример 1. Описан метод даје други начин за проналажење броја компоненти повезаности, као и броја циклуса у графу, а такође се користи у многим проблемима у којима се не користе графови.

Даље ћемо описати како се користи у Крускаловом алгоритму.

5.3 Крускалов алгоритам

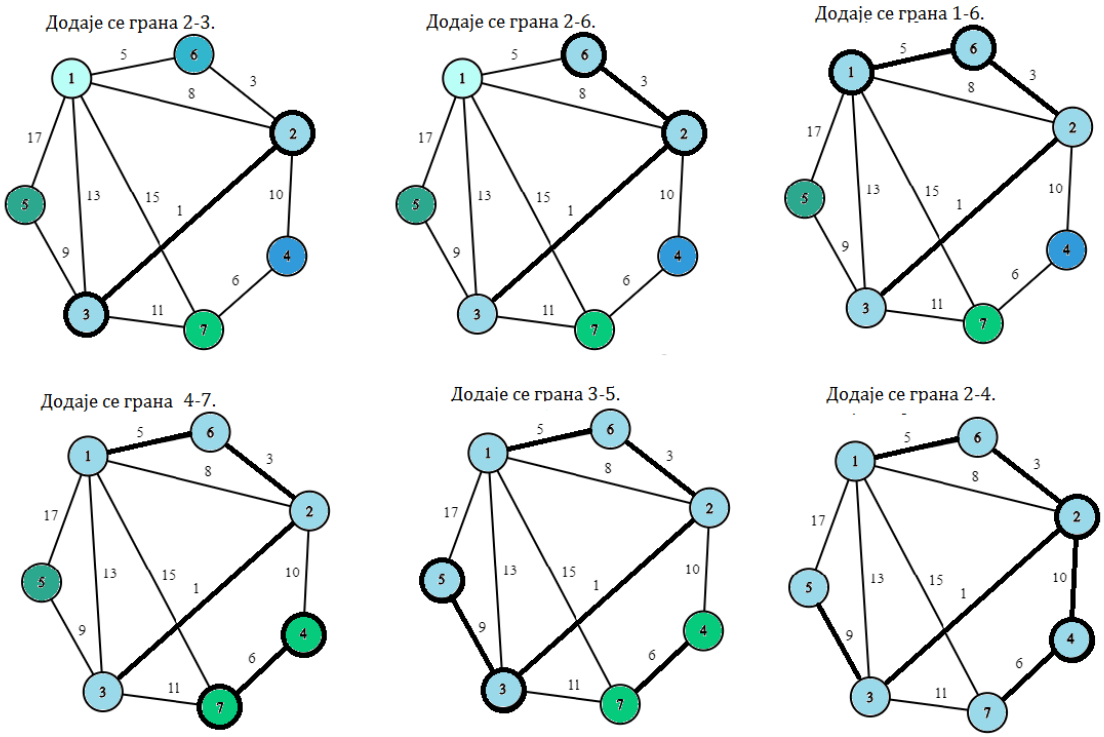
Џозеф Крускал је творац овог алгоритма који је 1956. године објављен у једном његовом раду.

За разлику од Примовог алгоритма, Крускалов алгоритам налази минималну покривајућу шуму графа уколико је он неповезан, односно стабло у случају повезаног графа.

Овај алгоритам сврстава се у похлепне алгоритме, јер он у сваком кораку додаје грану минималне тежине, али такву да добијени граф нема циклуса.

Опис:

1. Нека сваки чвор у почетку представља засебно стабло. Означимо сваки чвор као представника стабла које садржи само њега. Скуп грана *E1* је у почетку празан скуп.
2. Сортирамо гране растући по тежини.
3. Крећемо се по сортираном низу грана. Ако грана коју тренутно посматрамо (између чворова *v* и *u*) повезује два различита подстабла додајемо је у скуп грана *E1*, а спајамо подстабла одређена чворовима *v* и *u*, у супротном је игноришемо и настављамо даље.
4. Понављамо корак 3 све док не изаберемо *n-1* грану.
   Скуп грана *E1* и свих чворова у графу представља тражену минималну повезујућу шуму.



Слика 8: Крускалов алгоритам за налажење минималног покривајућег стабла.

Реализација. У почетку сваки чвор представља корен свог стабла, односно важи да је за сваки чвор графа *roditelj[v]=v*. Даље се крећемо по сортираном низу грана. За проверу да ли су чворови у истом подстаблу, као и спајање потребно је користити структуру раздвојених скупова где смо детаљно објаснили имплементацију наведених функција. На слици 8 дати су кораци Крускаловог алгоритма. У сваком тренутку различити скупови су представљени различитим бојама, а чворови између којих је грана која се додаје су подебљани.

Доказ коректности:

Нека је дат тежински повезан граф *G* са *n* чворова. Доказ исправности Крускаловог алгоритма састоји се из два дела. Нека је *T* граф добијен описаним поступком.
Најпре ћемо доказати да је добијено стабло повезаности. Довољно је дакле доказати да је добијени граф повезан и да нема циклуса. Претпоставимо супротно, да постоји циклус у графу. Ако постоји циклус постоји и грана чијим додавањем он настаје, међутим та грана би морала онда да повезује два чвора из истог подстабла што је у супротности са описаним алгоритмом. Такође граф *T* мора бити повезан јер се у сваком тренутку спајају две повезане компоненте, то јест број компоненти повезаности смањује се за један са сваком додатом граном. У почетку је тај број *n*, дакле након што се дода *n-1* грана граф постаје повезан. Тимо смо доказали да је *T* повезујуће стабло.

Докажимо да је *T* минимално повезујуће стабло.
Нека је *P* следећа претпоставка: Ако је *F* скуп грана које су изабране у било ком стадијуму алгоритма онда постоји неко минимално покривајуће стабло целог графа *G* које садржи *F*.
Докажимо индукцијом да *P* важи.
База: Претпоставка *P* је тачна на почетку јер је тада *F* празан скуп, па постоји минимално покривајуће стабло јер сваки повезан тежински граф има минимално покривајуће стабло.

Индуктивна хипотеза: Претпоставимо да *P* важи након *k* корака, односно када *F* садржи *k* грана. Дакле за скуп изабраних грана *F* постоји неко минимално покривајуће стабло графа *G*, нека је то стабло *S*.

Докажимо да *P* важи након *k+1* корака.
Нека је *e* грана изабрана у последњем кораку.
Ако *S* садржи *e*, онда важи претпоставка *P*.
Ако *S* не садржи *e*, онда у графу *S+e* постоји циклус. (*S* је минимално покривајуће стабло па садржи *n-1* грану, *S+e* садржи *n* грана, а повезан граф са *n* чворова и *n* грана садржи циклус). Дакле постоји једна грана тог циклуса која није у *F* јер у супротном грана *e* не би била додата јер не би повезивала две одвојене компоненте. Обележимо ту грану са *e1*. Посматрајмо сада граф *S+e-e1* који је стабло чија је тежина једнака тежини минималног покривајућег стабла *S* јер *e1* не може имати мању вредност од *e* (тада би била изабрана раније у алгоритму), а не може бити ни већа од *e* јер тада *S* не би било минимално покривајуће стабло. Закључујемо да *S+e-e1* јесте минимално покривајуће стабло које садржи *F* па и у овом случају важи *P*. Индукцијом смо доказали да претпоставка *P* важи у сваком стадијуму, па важи и након што се изабере *n-1* грана, што значи да је граф добијен Крускаловим алгоритмом заиста минимално покривајуће стабло задатог графа *G*.
Овим смо доказали да је алгоритам тачан за повезан граф. У случају неповезаног графа на описан начин се доказује да је добијено МПС за сваку компоненту задатог графа, одакле је јасно да се добија минимална покривајућа шума целог графа.

Сложеност. Уочавамо да сложеност највише зависи од тога како сортирамо гране. У случају да користимо уграђену функцију *sort* сложеност је *O(ElogE)*. Некад је могуће користити неку другу врсту сортирања која ради у линеарном времену на пример сортирање пребројавањем (*counting sort*) и уз коришћење структуре дисјунктих скупова сложеност је *O(Eα(V))*.

Примене алгоритама за налажење минималног повезујућег стабла.

Дизајн мрежа. Овај алгоритам налази примену у разним проблемима где је потребно наћи најоптималнију мрежу некаквих веза било да су у питању телефонске линије међу градовима, повезивање компјутера или чак мрежа путева у некој области. На пример ако је у некој компанији потребно повезати рачунаре кабловима и зна се цена повезивања за сваки пар рачунара онда се наведеним алгоритмом добија најисплативија мрежа каблова.

Апроксимација проблема трговачког путника. (traveling salesman problem) Проблем трговачког путника је НП тежак проблем, упркос једноставној формулацији. Потребно је за дати скуп градова и дата растојања између свака два града пронаћи најкраћу могућу руту која пролази кроз сваки град и враћа се у полазни град. Могуће је доказати да је таква рута (вредност те минималне руте) већа од вредности минималног покривајућег стабла, а мања или једнака од вредности минималног покривајућег стабла помножене са два. То није увек најбоље решење, али је у случају великог броја градова веома корисна апроксимација.

6 Дајкстрин алгоритам за налажење најкраћих путева

Алгоритам носи назив по холандском научнику Едгстеру Дајкстри који га је смислио 1956. године. Проблем који се разматра јесте како пронаћи најкраћи пут од задатог чвора до свих осталих чворова у тежинском графу. Алгоритам захтева да вредности свих грана буду ненегативни бројеви.

Опис:

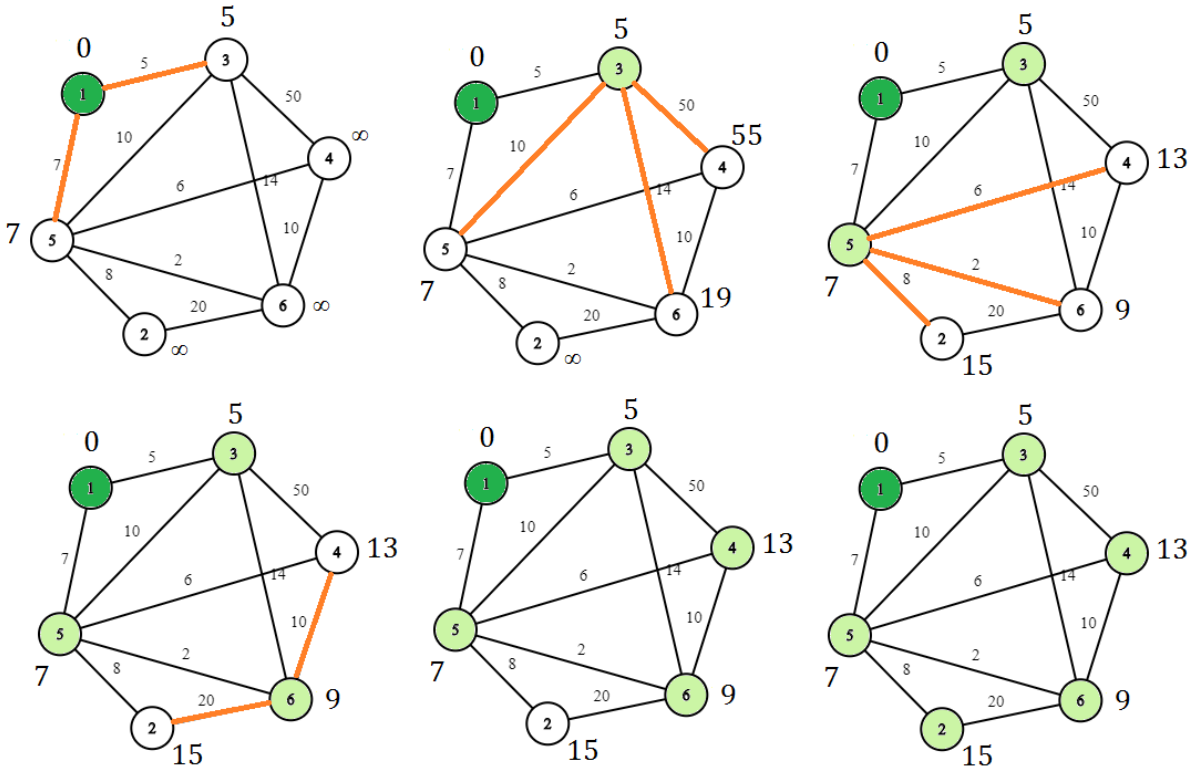
1. Нека је дат повезан тежински граф и нека је *s* полазни чвор. У низу *rast[]* чувамо дужину најкраћег пута од чвора *s* до одговарајућег чвора. У почетку постављамо *rast[s]=0* и *rast[v]* $= \infty$ за све чворове *v* различите од *s*. У низу *marka[]* чувамо информацију о томе да ли смо већ израчунали вредност траженог минималног растојања. Дакле у почетку је *marka[v]=0* за све чворове графа.
2. Ако су сви чворови означени алгоритам је завршен и израчуната су сва најмања растојања, у супротном бирамо неозначен чвор *v* са најмањом вредношћу *rast[v]* и маркирамо га као завршен (ако постоји више чворова са минималном вредношћу бирамо произвољан). Сада пролазимо кроз све гране *(v,u)* такве да

чвор *u* још није посећен и ажурирамо вредност низа *rast[]*. Дакле уколико је *rast[v]+l(v,u)* <*rast[u]* тада *rast[u]* постаје *rast[v]+l(v,u)*, (*l(v,u)* је тежина посматране гране).

3. Понављамо корак 2 све док сви чворови не буду маркирани као завршени.

Напомена: Описаним алгоритмом добијамо дужине најкраћих путева у повезаном графу. Уколико је граф неповезан истим поступком добићемо најкраће путеве од полазног чвора до свих из његове компоненте повезаности, а алгоритам треба прекинути када за сваки чвор *v* који није маркиран важи *rast[v]* =∞.

На слици 9 налази се приказ по корацима Дајкстриног алгоритма. Поред сваког чвора налази се тренутна вредност дотад добијеног минималног пута односно ∞, ако још није пронађен ниједан пут. Тамнозелено је представљен полазни чвор, а светлозелено чворови који су маркирани закључно са тим кораком. Посебно су истакнуте оне гране које се у датом кораку разматрају (то су само оне које воде до још немаркираних суседа).



Слика 9: Дајкстрин алгоритам за налажење најкраћег пута од почетног чвора до свих осталих.

Доказ коректности.

Нека је дат повезан тежински граф *G* са *n* чворова такав да су тежине грана позитивни бројеви.

Нека је *P* следећа претпоставка: Вредност *rast[v]* за сваки маркирани чвор *v* у сваком стадијуму алгоритма је дужина најкраћег пута од *s* до *v*.
Докажимо индукцијом да важи *P*.

База: Када скуп маркираних чворова садржи само полазни чвор *s* тада је *rast[s]=0*, дакле важи *P*.

Индуктивна хипотеза: Претпоставимо да *P* важи након *n-1* корака, односно када скуп маркираних чворова садржи *n-1* чвор.
Докажимо да *P* важи након *n* корака.
Нека је *v* чвор који је маркиран у последњем *n*-том кораку. Претпоставимо супротно, односно да постоји пут мање дужине од *rast[v]*. Нека је тада претпоследњи чвор тог минималног пута *w* (последња грана тог пута је она која повезује *w* и *v*). Ако чвор *w* није маркиран тада је дужина минималног пута од *s* до *w* сигурно већа или једнака *rast[v]* (кад би била мања тада би чвор *w* био маркиран пре *v*, јер по индуктивној хипотези у већ реализованих *n-1* корака пронађено је *n-1* чворова који имају најмање минимално растојање од *s*) тако да није могуће да је пут до *v* преко *w* бољи од нађеног. У другом случају, ако је *w* маркиран, тада је по индуктивној хипотези *rast[w]* заиста дужина минималног пута до *w*, а по опису алгоритма грана од *w* до *v* је већ разматрана, одакле следи да би вредност *rast[v]* већ била измењена и да је тај пут већ пронађен.
Дошли смо до контрадикције, дакле добијена вредност *rast[v]* јесте дужина минималног пута од *s* до *v*.
Индукцијом смо доказали да претпоставка *P* важи у сваком стадијуму, па важи и након што се маркира свих *n* чворова, што значи да су тражене дужине путева заиста минималне.

Реализација и сложеност.

Описаћемо два начина за имплементацију Дајкстриног алгоритма и њихову сложеност.
Зарад једноставности описа подразумеваћемо да је граф представљен листом суседства.

I начин: Први начин (познатији као Дајкста без хипа) једноставнији је за имплементацију, међутим његова временска сложеност је релативно велика. Сваки пут када бирамо чвор који има најмању вредност у низу *rast[]* проћићемо кроз све чворове графа и тако пронаћи неозначен чвор *v* са минималном вредношћу *rast[v]*. Након што га маркирамо, потребно је проћи кроз све гране које полазе из тог чвора, па је укупна временска сложеност је $O(|V|^2+|E|)=O(|V|^2)$.

II начин: Други начин (познатији као Дајкста са хипом) захтева коришћење неке структуре података уз помоћ које можемо у логаритамској сложености добити информацију о чвору који има најмању вредност низа *rast[]*. То могу бити хип или приоритетни ред (редом *heap, priority queue* STL C++). У логаритамској сложености врши се и ажурирање вредности у низу *rast[]* када је то потребно. У приоритетном реду чувамо који је чвор и која је његова вредност у низу *rast[]*. Дакле по већ описаним корацима у алгоритму када бирамо нови чвор, не пролазимо све чворове графа, већ узимамо са „врха“ приоритетног реда чвор са најмањом вредношћу *rast[]*. Када ажурирамо вредности чворова убацујемо у приоритетни ред нови пар чвор и ажурирана вредност минималног растојања. Због тога што $|V|$ пута узимамо чвор и још уз додавање нових парова у приоритетни ред, а сваки пут добијамо информацију у сложености *log(величина приоритетног реда)* свеукупна временска сложеност овог начина је $O(log|V|(|V|+|E|))$.

Примене.

На почетку рада, наводећи примене графова у информационим технологијама, споменуто је коришћење графова у Гугл мапама. Описани Дајкстрин алгоритам користи се за проналажење минималних путева од полазне тачке до одредишта.

Дајкстра је такође део већ споменутог алгоритма који предлаже пријатељства на дрштвеним мрежама.

Користи се и у телефонским мрежама, где се помоћу овог алгоритма одређује најкраћи пут протока информације, где су чворовима представљене телефонске централе и уређаји. Гране у графу су линије које спајају телефоне и централе, а њихова вредност је опсег сигнала који се може пренети преко одговарајућег проводника.

Модификација овог алгорима налази примену у процесу прављења распореда летова на аеродромима. Аеродроми су чворови, а усмерене гране су летови, са вредностима полазака и долазака. На тај начин се добија најбољи распоред за што ранији долазак авиона на одредиште.

7 Закључак

Од тренутка када сам ушла у свет програмирања и научила прве графовске алгоритме била сам одушевљена овом облашћу којом се преко наизглед једноставних цртежа и мапа градова долази до прелепе области математике и ефикасних решења за разне занимљиве проблеме.

Помоћу графова се могу сагледати разни проблеми из света науке, али и они из свакодневног живота. Због тога ће алгоритми за рад са графовима имати све већу примену у будућности и све више нових алгоритама ће се конструисати.

Током израде матурског детаљније сам проучила приказане алгоритме и научила о њиховој примени у савременим информационим технологијама, као и могућности како да се развијају даље.

Циљ овог матурског рада био је опис и имплементација неких од графовских алгоритама. Надам се да ће рад бити од помоћи онима које интересује теорија графова и њена примена у информатици као и ученицима који се припремају за такмичење.

Надам се да ћу у будућности имати прилике да истражујем и бавим се оптимизацијом различитих проблема који се могу представити графовима.

На крају бих волела да се захвалим свима који су ме подржавали у току средњошколског образовања, а посебно професорки и ментору Јелени Хаџи-Пурић која ме је увела у свет алгоритмике, мотивисала да почнем да се бавим информатиком и од које сам много научила.

8 Литература

[1] Миодраг Живковић, Алгоритми, http://poincare.matf.bg.ac.rs/~ezivkovm/, 21.5.2020.

[2] Cs Academy, https://csacademy.com/, 18.5.2020.

[3] Antti Laaksonen ,Competitive Programmer's Handbook, https://cses.fi/book/book.pdf, 21.5.2020.

[4] Geeks for geeks, https://www.geeksforgeeks.org/, 18.5.2020.

[5] Wikipedia, Graph algorithms, https://en.wikipedia.org/wiki/Category:Graph_algorithms, 18.5.2020.