LATVIJAS UNIVERSITĀTE
FIZIKAS, MATEMĀTIKAS UN OPTOMETRIJAS FAKULTĀTE
MATEMĀTIKAS NODAĻA

GRAFU NEIRONA TĪKLU IZMANTOŠANA OPTIMĀLO MARŠRUTU NOTEIKŠANĀ

BAKALAURA DARBS

Autors: Rolands Aleksandrs Rudenko
Studenta apliecības nr.: rr19044
Darba vadītājs: Mg. math. Artis Alksnis

RĪGA 2023

ANOTĀCIJA

Bakalaurs ”Grafu neirona tīklu izmantošana optimālo maršrutu noteikšanā” ir darbs, kas veltīts ceļojošā pārdevēja problēmai, kas ir populāra kombinatorikas optimizācijas problēma ar NP-sarežǵītību. Šī problēma tiek atrisināts ar grafu neirona tīklu un šis risinājums salīdzināts ar klasiskajām metodēm, kā: dinamiskā programmēšana un skudru kolonijas optimizācijas algoritms, kas arī ir šī darba mērķis.

Risinājums ar grafu neirona tīkliem balstās uz Jošhi piedāvāto pieeju, kas tika publicēts 2019. gadā. Jošhi pieeja sastāv no pāris būtiskiem soļiem. Grafu konvolūciju tīklā tiek ievadīts divdimensiju grafs. GCN veic darbības, izdot varbūtības ar kādu loks piederēs optimālā ceļa atrisinājumam, kas vēlāk tiek izmantots, lai, izmantojot alkatīgo pieeju, atrastu GCN optimālo ceļu.

Darba uzdevumi ir implementēt iepriekš minētos algoritmus TSP atrisināšanai, izmantojot python, un sniegt atzinumu par dažādo algoritmu veiktspēju. Darba rezultāti uzrāda, ka lielām instancēm GCN performē gan ātrāk, gan arī labāk, nekā klasiskās metodes, bet mazām instancēm, iespējams, nav vērts tērēt visus resursus, lai implementētu GCN, toties akadēmiskiem un zinātniskiem mērķiem, GCN ir plaša un interesanta tēma, ko apgūt un attīstīt tā lielā potenciāla dēļ.

Atslēgas vārdi: Ceļojošā pārdevēja problēma, Neirona tīkli, Grafu neirona tīkli, mašīnmācīšanās, optimizācija, dinamiskā programmēšana.

ABSTRACT

”Optimal root determination using Graph neural network” bachelors main focus is on solving traveling salesman problem. Traveling sales man problem is popular NP-hard combinatorial optimization problem and in this work TSP is solved used graph neural network and alternative methods such as dynamic programming and ant colony optimization. Bachelors aim is to explore graph neural network and compere its efficiency to alternative methods.

Graph neural network solutions is based on Joshi published paper on 2019. Joshi solution is based on few important steps. As an graph convolution network input is two-dimensional graph witch is prpceded by GCN and as an output model gives probability that edge is part of the solution. After that greedy approach is used to find GCN solution for TSP.

Bachelors aim was achieved by implementing all methods using python and analysing results. Thus results shows that GCN is useful and better for big instances, but not for the small instances since problem could be solved using traditional methods with better accuracy and traditional methods does not requires as much resources as GCN does, but even tho GCN structure is complex, there is a lot of room to explore and discoverer new possibilities that GNN and GCN provides with.

Keywords: Traveling salesman problem, Neural networks, Graph neural networks, Machine learning, optimization, dynamic programming.

SATURA RĀDĪTĀJS

Apzīmējumu saraksts 5

Ievads 6

1. TSP problēmas apraksts 8

2. Neirona tīkli 10

2.1. Neirona tīklu bioloģiskā motivācija 11

2.2. Neirona tīkla uzbūve un darbība 12

2.2.1. Summēšanas funkcija 13

2.2.2. Aktivizācijas funkcijas 14

2.2.3. Izejas funkcija, izejas slānis 17

3. Grafu teorijas pamatjēdzieni 18

4. Grafu neirona tīkli 23

4.1. Grafu neirona tīkla darbības pamatprincipi 23

4.2. Grafu neirona tīklu pielietojums TSP 24

5. Alternatīvās metodes 27

5.1. Naivā metode 27

5.2. Dinamiska programmēšana un Held-Karp algoritms 28

5.3. Skudru kolonijas optimizācijas metode 29

5.3.1. TSP risinājums ar skudru kolonijas optimizācijas palīdzību 30

6. Praktiskā daļa 32

6.1. Datu kopa un GCN modelis 32

6.2. Simulācija 32

6.3. Algoritma novērtēšana 33

7. Rezultāti 34

Nobeigums 38

Izmantotā literatūra un avoti 40

Pateicības 42

Pielikumi 43

A. Python programmas kods 43

APZĪMĒJUMU SARAKSTS

ANN – Mākslīgais neirona tīkls (*Artifical neural network*)

GNN – Grafu neironu tīkli ir māklsīgā neirona tīkla paveids, kas ir specifiski veltīts, lai apstrādātu grafus, meklējot savienojumus un sakarības grafa iekšienē (*Graph neural network*)

GCN – Grafu konvulūciju tīkli ir mākslīgā neirona tīkla paveids, kas ir spcifiski veltīts grafiem. GCN ir atvasinājums no konvulūciju neirona tīkliem (*Graph convolutional network*)

TSP – Ceļojošā pārdevēja problēma, kas ir viena no populārākajām kombinatorikas optimizācijas problēmām (*Travelling salesman problem*)

ACO – Skudru kolonijas optimizācija, kas ir algoritms iedvesmots no dabas un risina dažāda veida kombinatorikas optimizācijas problēmas (*Ant colonny optimization*)

DP – Dinamiskā programmēšana ir algoritms, kas sadala problēmu sīkākās apakšapgabalos, kas savstarpēji pārklājas (*Dynamic programming*)

HK – Held-Karp algoritms, kas ir balstīts uz dinamisko programmēšanu, bet ir specifiski pielāgots TSP problēmas risināšanai

BR – TSP naivā metode, kas apskata visus iespējamos ceļus, nodrošinot optimālo ceļu *brute force*

Python – Objektorientētā programmēšanas valoda

$\mathscr{G}$ – Grafs

$\mathscr{V}$ – Mezglu/virsotņu kopa (*vertex*)

$\mathscr{E}$ – Loku kopa (*edge*)

l – apzīmējums ar kuru apzīmē slāņa kārtu

p_{ij}^{TSP} – Varbūtība, ka loks pieder optimālajam atrisinājumam

$\tau_{ij}(t)$ – Feromona daudzums (i,j) lokā t laika brīdī

$p_{ij}^{k}(t)$ – Varbūtība k-tajai skudrai iet pa loku/ceļu (i,j) t laika brīdī

IEVADS

Nu jau ik vienā industrijā neatņemama sastāvdaļa ir mašīnmācīšanās (*ML*) algoritmi vai mākslīgais intelekts, jo ML ir spējīgi izmantot datora datu apstrādes veiktspēju un tai pašā brīdī spēj mācīties tā pat kā cilvēks, tādējādi pārspējot gan datora pasīvo atmiņu un aprēķinu veiktspēju, gan arī cilvēka fleksibilitāte interpretēt dažāda veida informāciju, kas sevī iekļauj attēlus, sakarības, skaņas un arī grafus.

Šo fleksibilitāti sniedz tieši neirona tīkli, kuri, atšķirībā no klasiskajām ML metodēm, spēj mācīties, paverot plašas iespējas cilvēcei automizēt un pieņemt lēmumus, balstoties uz skaitļiem un plašo datu apjomu nevis uz ierobežoto cilvēka pieredzi. Turklāt, palielinoties cilvēku daudzumam, palielinās arī tādas lietas, kā pārtikas patēriņš, kas izraisa nepieciešamību pēc efektīvākas un lētākas transportēšanas infrastruktūras.

Šī darba ietvaros tiks sīkāk apskatīta viena no loģistikas problēmām, kas ir zināms izaicinājums datorzinātņu jomā, lai gan pati problēma ir ļoti vienkārša pēc tās būtības, bet ne no matemātiskā un/vai datorzinātņu puses. ceļojošā pārdevēja problēmas risināšanai ir daudzi algoritmi, kas risina šo problēmu, bet izaicinājums ir spēja atrisināt uzdevumu ātri un tai pašā brīdī spēt atrast īsāko ceļu, jo, palielinoties mezglu skaitam, apskatāmo ceļu skaits skaits palielinās eksponenciāli.

Problēmas atrisināšanai darbā tiks izmantoti četri algoritmi: dinamiskā programmēšana, dinamiskās programmēšanas paveids Held-Karp algoritms, skudru kolonijas optimizāciju un grafa neirona tīklus. DP un HK ir operāciju pētīšanas metodes, ACO ir mašīnmācīšanās algoritms, bet GNN ir dziļāsmašīnmācīšanās algoritms. Darbā tiks implementēta arī naivā metode ar mērķi izpētīt un saprast, kāpēc mūsdienu problēmām šī metode nav gana efektīva un gana laba.

Darbs tiks balstīts primāri uz pāris grāmatām, kā piemēram: "Īss ievads Neirona tīklos", ko sarakstīja Deivids Krisels [13], "Grafu neirona tīkli", kur ir vairāki līdzautori [20], mācību materiāliem no Latvijas Universitātes Datorikas fakultātes lektora J. Zutera par neirona tīkliem [21] un P. Dauguļa par grafu teoriju [5] un raksta "Efektīva grafu konvolūcijas tīkla tehnikas ceļojošā pārdevēja problēmai" kuras autori ir Čaitanja Džoši, Tomass Lorāns un Ksavjers Bresons [11] un citi saistoši raksti par ACO, DP un GNN. Darbs sastāvēs no TSP problēmas apraksta, teorētiskās daļas, kur liels uzsvars tiks likts uz neirona tīkliem un grafu neirona tīkliem un pavisam nedaudz tiks izklāstīts par alternatīvajām metodēm. Pēc teorētiskās daļas, būs arī izklāstīta praktiskā daļā un tās rezultāti, uz kuriem balstīsies arī secinājumi.

Darba galvenais mērķis ir izpētīt, kā TSP problēmu var atrisināt izmantojot GNN un salīdzi-

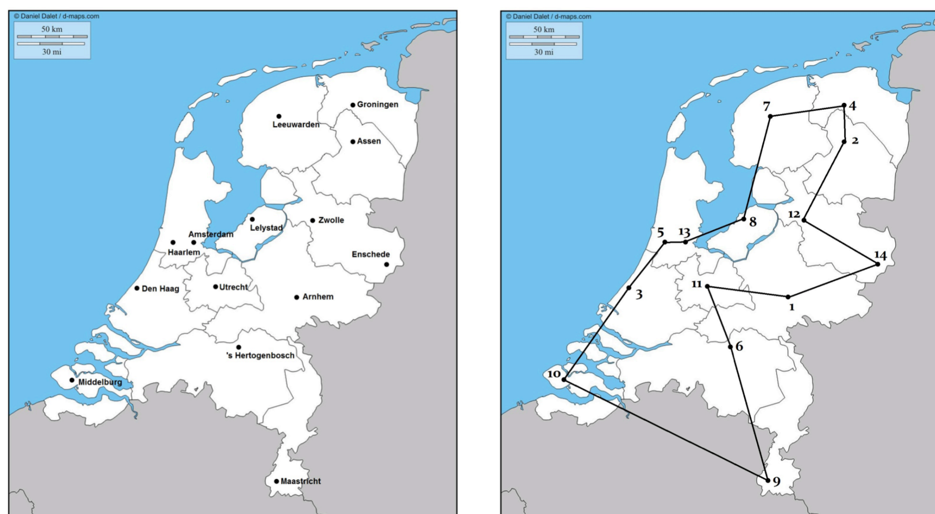
nāt GNN veiktspēju ar citām klasiskajām metodēm. Lai izpildītu darba mērķi, tika izmantota *python* programmēšanas valoda un tika izvirzīti sekojoši darba uzdevumi:

- ģenerēt dažāda lieluma grafus;
- iepazīties ar visam metodēm un tās implementēt darbā;
- salīdzināt dažādos algoritmus;
- veikt secinājumus par GNN un citu algoritmu veiktspēju.

1. TSP PROBLĒMAS APRAKSTS

Ceļojošā pārdevēja problēma (*Travelling salesman problem (TSP)*) ir viena no populārākajam NP-sarežģītības kombinatoriskas optimizācijas problēmām datorzinībās pie lielu virsotņu daudzuma, kuras mērķis ir atrast īsāko ceļo, apmeklējot visas pilsētas. Šajā problēmā, atkarībā no tās sarežģītības un vēlamā rezultāta, ko vēlas iegūt, primāri tiek dots pilsētu skaits, ko matemātiski apzīmēsim par mezgliem/punktiem (*nodes*) un attālums no vienas pilsētas līdz otrai, ko apzīmēsim kā savienojumu no viena mezgla/punkta uz otru (*edge*).

Vizuāli TSP risinājumu reālās dzīves gadījumā var redzēt 1. attēlā . Kreisajā attēlā tiek dota karte ar 14 Nīderlandes pilsētām, kuras ceļotājam ir jāapceļo, un labajā pusē ir risinājums ar ”naivo metodi,” kas arī darbā tiks aprakstīta. Neatkarīgi no pilsētas un virziena, kuras ceļotājs uzsāks, tas nemaina īsāko ceļu.



(a) **Nīderlandes karte ar 14 pilsētām** *(b)* **TSP risinājums/īsākais ceļa garums**

1. att. **TSP piemērs [8]**

TSP pirmo reizi tika matemātiski aprakstīts 19. gadsimtā, ko aprakstīja Viljams Hamiltons, bet tobrīd šī problēma neguva lielu popularitāti tīri tāpēc, ka datorzinības un matemātiskie algoritmi vēl nebija tik populāras tajā laikā. Ap 20. gadsimta vidu TSP ieguva lielu popularitāti, ko izraisīja datorzinību attīstība. 1954. gada rakstos Merrills Floods un Melvins Dreshers TSP aprakstīja kā matemātisku problēmu, bet Džordžs Dancigs bija pirmais, kas izveidoja atrisinājumu TSP problēmai.

Lai atrisinātu TSP, ir jāizmanto algoritmi, kas spēj optimizēt ceļu starp visiem punktiem. Var izmantot ”Brute Force” metodi, kas ir ”naivā” metode šīs problēmas atrisināšanai, jo šī metode apskata pilnībā visus iespējamos ceļus un atrod īsāko ceļu. Šī metode sniedz 100% precizitāti,

bet tiklīdz pilsētu skaits pārsniedz 20, tad šī metode prasa daudz laika, lai izrēķinātu, jo ceļu skaitu aprēķina pēc $(n-1)!$ un ceļu skaits pieaug faktoriāli. Protams, jāņem vērā, ka realitātē dažreiz nepastāv tiešais ceļš no pilsētas A uz pilsētu B, tāpēc apskatīto ceļu daudzums varētu nedaudz samazināties.

Tā kā naivā metode nav diži ko efektīva no praktiskā viedokļa, tad ir nepieciešami citi algoritmi, kas spēs atrisināt šo problēmu. TSP var atrisināt gan izmantojot mašīnmācīšanās algoritmus, gan arī klasiskās operāciju pētīšanas metodes.

Mūsdienās TSP ir plaši izmantota tādās sfērās, kā kravas pārvadāšana, loģistika, ražošana, telekomunikācijas un nerunājot par citu problēmu pārveidojumiem, kas izriet no TSP. TSP risinājums ir aktuāls ne tikai praktiskajam pielietojam, bet TSP arī ir veicinājis plašu izpēti tādās matemātikas un datorzinību zinātniskās optimizācijas nozarēs, kā: kombinatorikas optimizācija, grafu teorija un aproksimācijas algoritmi.

2. NEIRONA TĪKLI

Mākslīgais neirona tīkls (*Artificial neural network (ANN))* ir mašīnmācīšanās (ML) tehnoloģija, ko pirmo reizi izveidoja F. Rozenbalts 1958. gadā. ANN ir iedvesmots no cilvēka smadzeņu darbības un tā ir programmējama sistēma, kas cenšas pēc iespējas vairāk imitēt cilvēku smadzeņu darbību, lai apstrādātu un analizētu datus un veikt prognozes. ANN ir spējīgs apstrādāt lielu datu daudzumus, atpazīt un mācīties no sarežģītām struktūrām.

Neirona tīklu popularitāte ar katru gadu pieaug, jo neironu tīklu pielietojums ir ļoti plašs tīri tās fleksibilitātes dēļ, kas spēj atrisināt gandrīz jebkuru problēmu, ko spēj klasiskās metodes atrisināt un dažreiz pat sasniegt labākus rezultātus.

Toties, lai arī cik daudz pētījumi nav uzrādījuši ANN efektivitāti pār klasiskajām metodēm, ir arī ierobežojumi ar ko saskarās ANN. Lai ANN sasniegtu precizitāti, kas ir nepieciešama, ir nepieciešams liels datu apjoms, lai uztrenētu modeli. Turklāt šiem datiem ir jābūt ar pareizajiem risinājumiem. TSP problēmā tas izpaužas, ka ir nepieciešams optimālais ceļš. Bet TSP lielu datu ieguve ar pareizajām problēmām nav sarežģīti dabūt, bet reālas dzīves problēmas atrisināju iegūt būtu daudz sarežģītāk, jo nāk klāt vēl daudz un dažādi parametri. Šis ir tikai viens no iespējamajiem mīnusiem un tādu ir vēl daudz. Pat tādi, kas skar likumu un cilvēka morālās robežas.

Ar neirona tīklu palīdzību, var atrisināt sekojošas problēmas:

1. **Klasifikācija**: klasifikācijas problēma ir ļoti bieži sastopama problēma dažādās industrijās, kā piemēram medicīnā, psiholoģijā vai finansēs, kur ar klasifikācijas metodi cenšamies atbildēt uz jautājumu par to vai pacientam nākotnē būs depresija vai nebūs. Klasiski industrijas izmanto loģistisko regresiju, lai atbildētu uz šo jautājumu, bet ir daži pētījumi, kur neirona tīkli ir spējīgi sniegt ievērojami labākus rezultātus salīdzinājumā ar klasiskajām metodēm. Arī attēlu un objektu atpazīšana ir klasifikācijas problēma;

2. **Optimizācija**: optimizācija neirona tīklos varētu izmantot, piemēram, lai atrastu īsāko ceļo no punkta A līdz punktam B. Arī šī darba apskatītā problēma, kur ir jāatrod īsākais ceļš no punkta A līdz punktam B izbraucot cauri pilnībā visiem punktiem jeb darba kontekstā - visām pilsētām, var atrisināt ar neirona tīkliem. Darba ietvaros TSP tiks atrisināts ar grafu neirona tīkliem, kas ir ANN paveids veltīts specifiski grafiem. GNN darbība tiks salīdzināta ar klasiskajām metodēm;

3. **Regresija**: regresija ir problēma, kur prognozē nepārtrauktu lielumu jeb regresija problē-

mu Šeit tiek arī ietverti sarežģīti regresijas modeli, kuri ir gan vien dimensionāli un vairāk dimensionāli. Lineāri un arī nelineāri.

2.1. Neirona tīklu bioloģiskā motivācija

Palielinoties cilvēku daudzumam, palielinās arī lielais datu apjoms, kas ir nepieciešams apstrādāt, bet, neskatoties uz to, ka datu apjoms un pieejamība palielinās ar katru gadu, ir problēmas, ko tomēr nevar algoritms atrisināt, jo tas ir atkarīgs no ļoti daudziem faktoriem. Labs piemērs algoritmu nespējai ir psiholoģija, kur var ļoti vispārēji novērtēt zināmus parametrus, bet mūsdienu standarta algoritmi pagaidām nav spējīgi veiksmīgi novērtēt visus nepieciešamos parametrus.

Cilvēku smadzenes ir apveltītas ar lielisku spēju - mācīties. Tā ir arī viena būtiska atšķirība, kas klasiskajiem mašīnmācīšanās algoritmiem nepiemīt, bet cilvēku smadzenēm piemīt. Datoram piemīt liela atmiņa un tas spēj veikt sarežģītus aprēķinus krietni ātrāk un precīzāk nekā cilvēka smadzenes, kas ir liels ieguvums tieši datoriem, bet nav cilvēku smadzenēm.

Un teorētiski, ja salīdzina cilvēku smadzenes ar datoru, tad dators ir krietni jaudīgāks nekā smadzenes, jo dators sevī ietver 10^9 tranzistorus un tos apstrādāt jeb pārslēgt ir spējīgs 10^{-9} sekundēs, kamēr smadzenēm ir 10^{11} neironi, bet to pārslēgšanās spēja ir tikai 10^{-3} sekundes, jo lielākā daļa smadzeņu aktīvi strādā, kamēr datoram ir ļoti daudz pasīvās atmiņas, kas netiek pielietota, kas arī ir viena no motivācijām, lai pētītu un attīstītu neirona tīklus.

Tā kā ANN darbības princips ir pielīdzināms smadzeņu darbībai, tad tas arī veicina plašo zinātnieka interesi un izpēti par ANN. 1. tabulā ir attēlots to cik līdzīga ir neirona darbības princips un smadzeņu neirona darbība. Turpmāk darbā arī tiks aprakstīts, ka ANN, tā pat kā bioloģiskajam neironam ir ieejas vērtības un apstrādājot to ir izejas vērtības.

1. tabula

Neirona bioloģiskie termini ANN interpretācijā.

Bioloģiskā terminoloģija	ANN terminoloģija
Neirons	Mezgls/vienība/šūna/neirons
Sinapse	Savienojums
Sinaptiskā efektivitāte	Svari
Šaušanas biežums	Izvade

Tūrklāt arī ANN lielumi, kas palīdz apstrādāt ieejas vērtības, ir pielīdzinamas bioloģiskajam

neironam. Spilgts piemēras tam ir sinaptiskā efektivitāte, kas ANN gadījumā ir svari, kas turmāk darbā arī tiks paskaidrots.

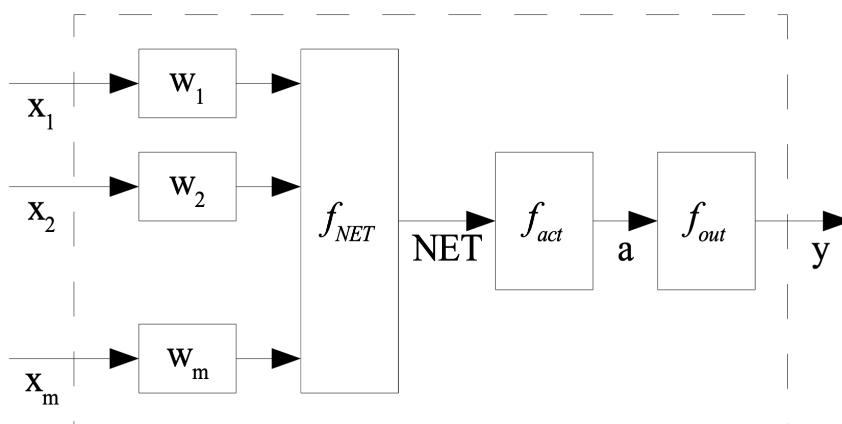
2.2. Neirona tīkla uzbūve un darbība

2.1. Definīcija. Neirona tīklu var definēt kā trīs elementu kopumu (N,V,w), kas sastāv no divām kopām N, V un funkcijas w, kur N ir neirona kopa un V ir kopa no $\{(i,j)|i,j \in \mathbb{Z}\}$, kuras elementi ir savienojami starp neironu i un neironu j. Funkciju $w : V \to \mathbb{R}$ piešķir svaru katram savienojumam starp neironu i un j. Saīsinājumā šī funkcija tiek pierakstīta sekojoši $w_{i,j}$.

Neirona tīkls sevī iekļauj sekojošas konstruktīvus elementus:

- (Sinaptiskie) svari ([*Synaptic*] *weights*),
- Summēšanas funkcija (*Propagation function* vai *Summation function*),
- Aktivizācijas funkcija (*Activation function*),
- Apmācības likums (*Learning rule*)

Šī darba ietvaros netiks sīki aprakstīts apmācības likumu, bet šis likums nosaka cik ļoti un kuri svari tiek izmainīti neirona tīklā, kas tiek aprēķināti, lai minimizētu kļūdu, izmantoju zaudējuma funkciju (*loss function*), par ko tiks vairāk runāts nodaļā par grafu neirona tīkliem.

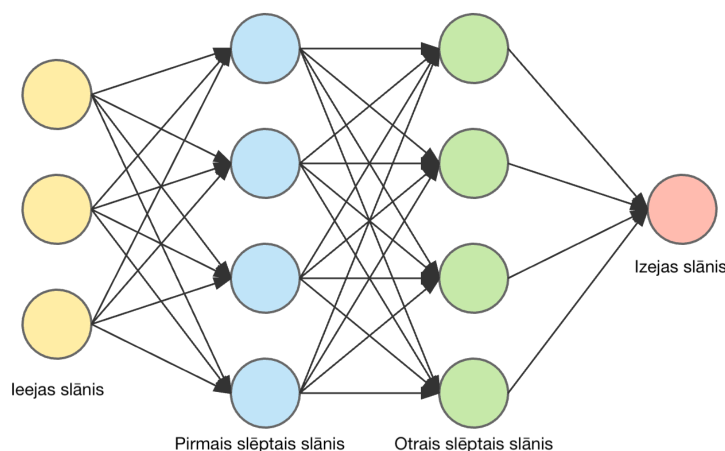


2. att. **Neirona darbības princips ar m ieejām**

Vienkāršākai izpratnei par neirona tīkla uzbūvi var skatīt 2. attēlu, kur x_m reprezentē m ieejas vērtības, w_m ir svari, f_{NET} ir summēšanas funkcija, kur iegūto vērtību NET tālāk ievieto aktivizācijas funkcijā f_{act}. Aktivizācijas funkcijas iznākumu a vēlāk izmanto izejas funkcijā f_{out}, bet vairumā neironu modeļu izejas funkcija nav atsevišķa no aktivizācijas funkcijas. Tas

nozīmē, ka aktivizācijas funkcijas rezultāts ir ne tikai aktivitātes stāvoklis, bet arī gala vērtība, kas ir neirona izejas vērtība.

Kā arī nelielam ieskatam par dziļomašīnmācīšanos, jo GNN modelī tiks izmantota dziļāmašīnmācīšanās. 3. attēlā var redzēt vienkāršu ANN modeli ar vairākiem slēptajiem slāņiem. Šie slēptie slāņi arī ir tie, kas veido dziļomašīnmācīšanos, padarot mācīšanās procesu sarežģītāku, bet krietni precīzāku, paverot iespējas arī jaunām iespējām modeļu izveidei un problēmu risināšanai.



3. att. **Vienkāršs dziļāsmašīnmācīšanās piemērs ar diviem slēptajiem slāņiem**

3. attēlā ir parādīta ļoti vienkārša ANN topoloģija, bet ir ļoti daudz un dažādu iespējamo variāciju skaitu, kas mūsdienās tika izdomātas. Mēdz būt arī situācijas, kur neirons nodot informāciju arī pats sev vai neirons pilda kādu pasīvās atmiņas funkciju.

2.2.1. Summēšanas funkcija

Viena no neironu tīkla svarīgām sastāvdaļām ir summēšanas funkcija (*Propagation function*), kura, apvienojot ieejas vērtības (*Input*) un to svarus, aprēķina vienu vērtību. Iegūto rezultātu pēc tam izmanto, lai noteiktu neirona tīkla izejas vērtību (*Output*). Literatūrā parasti šo vērtību apzīmē ar NET, bet pati funkcija tiek apzīmēta kā f_{NET} vai NET. Vienkāršības labad arī šajā darbā tiks izmantots līdzīgs apzīmējums.

Tā pat arī literatūrā visbiežāk izmantotā summēšana funkcija ir parādīta 2.1. formulā, kur i-tā neirona ieejas vērtību reizina ar i-tā neirona svariem un galā visus reizinājumus saskaita kopā. Matemātiski tas izskatītos sekojoši:

$$NET = WX = \sum_{i=1}^{m} w_i x_i, \tag{2.1}$$

kur NET - neirona summēšanas vērtība, $W = [w_1, w_2, \ldots, w_m]$ svaru vektors, kur w_i - neirona $i-$tais svars, $X = [x_1, x_2, \ldots, x_m]^T$ ieejas, vērtību vektors, kur x_i - neirona $i-$ā ieeja un m ir neirona tīkla ieeju skaits.

Literatūrā piedāvā arī citas, līdzīgas summēšanas funkcijas.

$$NET = \prod_{i=1}^{m} w_i x_i$$

$$NET = \max(w_i x_i); i = 1, \ldots, m$$

$$NET = \min(w_i x_i); i = 1, \ldots, m$$

Un dažos modeļos summēšanas funkciju izmanto arī Eiklīda attālumu vai tā kvadrātu.

$$NET = ||W - X|| = \sqrt{\sum_{i=1}^{m} (w_i - x_i)^2}$$

Iepriekšējās formulās un darbā jau iepriekš tika minēts, ka tiek izmantoti **svari** (*weight*) un ir svarīgi arī izprast tos, jo tie būtiski ietekmē rezultātu.

Katra neirona un visa neironu tīkla galvenā atmiņas daļa ir svari, kas ir skaitliskas vērtības. To galvenā funkcija ir nodrošināt, lai neironu tīkls darbotos kā paredzēts, un to vērtības tiek pielāgotas apmācības procesa laikā. Svari, kopā ar citām vērtībām, piemēram, sliekšņiem, tiek uzskatīti par neironu tīkla parametriem. Parasti katram ieejas neironam ir viens svars, un ienākošie signāli tiek izmainīti un kombinēti, pamatojoties uz atbilstošajām svaru vērtībām, kas tiek apstrādātas ar **summēšanas funkciju**. Parasti literatūrā svarus apzīmē ar w_i, bet ieejas vērtību literatūrā bieži apzīmē ar x_i vai o_i.

2.2.2. Aktivizācijas funkcijas

Pēc tā, kad tiek veikta summēšana, iegūtās summas rezultāts NET tiek pēc tam ievietot **aktivizācijas funkcijā** (*Activation function*). Literatūrā bieži vien aktivizēto stāvokli apzīmē ar burtu a, bet pašu aktivitātes funkciju apzīmē kā f_{act}. Aktivizācijas funkcijai ir ļoti liela nozīme neirona tīkla skaitļošanai, jo aktivizācijas funkcija palīdz noteikt rezultātu jeb izeju dotajai ieejai. Summēšanas funkcijas saņem tikai vienu vērtību, kuru vēlāk ievietot iepriekš definētā matemātiskā funkcijā. Funkcijas izvēle var ietekmēt neironu tīkla darbības principus, piemēram, cik sarežģītas problēmas tas spēj risināt vai to, cik viegli ir neironu tīklu apmācīt. Aktivizācijas funkcijas izvēle bieži ir arī atkarīga no problēmas, ko vēlas atrisināt, un no neirona tīkla arhitektūras. Aktivizācijas funkcijas palīdz normalizēt neirona izejas datus. Tā kā aktivizācijas

funkcija tiek izsaukta katrā neironā, tad tai ir jābūt vienkārši lietojamai, citādi radīsies skaitļošanas problēmas.

Neirona tīkls bez aktivizācijas funkcijas strādā ļoti līdzīgi kā lineārās regresijas modelis, kurai ir ļoti ierobežota darbība un jauda. Ir vēlams, lai neironu tīkls ne tikai iemācītos un aprēķinātu lineāro funkciju, bet arī veiktu sarežģītākus uzdevumus, kas palīdz analizēt un apstrādāt attēlus, skaņu, tekstu, grafus, valodu un citas darbības, ko spēj cilvēku smadzenes.

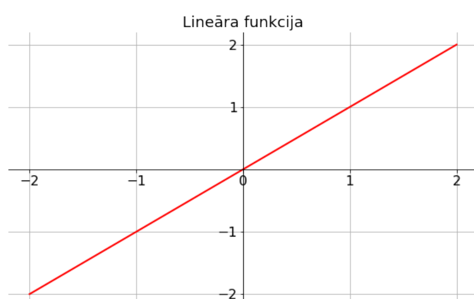
Ir pieejamas dažādas funkcijas, gan lineāras, gan nelineāras, lai aprēķinātu neirona aktivācijas stāvokli.

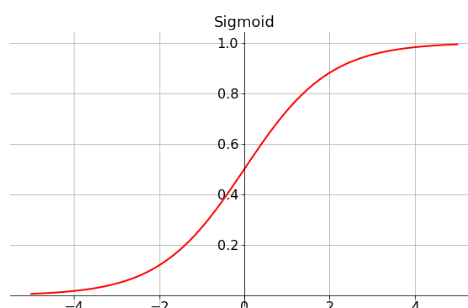
Lineāra funkcija – Lineāra funkcija ir viena no vienkāršākajām aktivizācijas funkcijām. Tiklīdz tiek aprēķināts f_{NET} rezultāts, tas tiek ievietots f_{act} un tiek iegūts tās rezultāts. Tā kā šī funkcijas uzbūve ir tik triviāla, tad tas liedz izmanto šo aktivizācijas funkciju komplicētos uzdevumos. Neatkarīgi no tā, cik daudz slāņu ir neirona tīklam, neirona tīkla pēdējais slānis būs tieša reprezentācija ieejas slānim tīri lineārās kombinācijas dēļ. Līdz ar to neirona tīkls ar lineārās aktivizācijas funkciju ir tikai viens slānis. Lineāra funkcija ir apskatāma 4. attēlā un definē ar sekojošu formulu:

$$f_{lin}(NET) = NET, \tag{2.2}$$

Sigmoīda funkcija – Sigmoīda funkcija, kuras attēlojumu var apskatīt 4. attēlā, sniedz rezultātu skalā no 0 līdz 1 un sigmoid funkciju bieži izmantot izejas slānim bināro klasifikācijas problēmās, sniedzot, piemēram, varbūtību ar kādu pacientam nākotnē varētu būt augsts risks iekrist depresijā. Tā kā sigmoīda funkcija ir ne-lineāra funkcija, tad tā ir viena no visplašāk izmantotākajām aktivizācijas funkcijām, kuru matemātiski apraksta ar sekojošu formulu:

$$f_{sig}(NET) = \frac{1}{1 + e^{-NET}}, \tag{2.3}$$





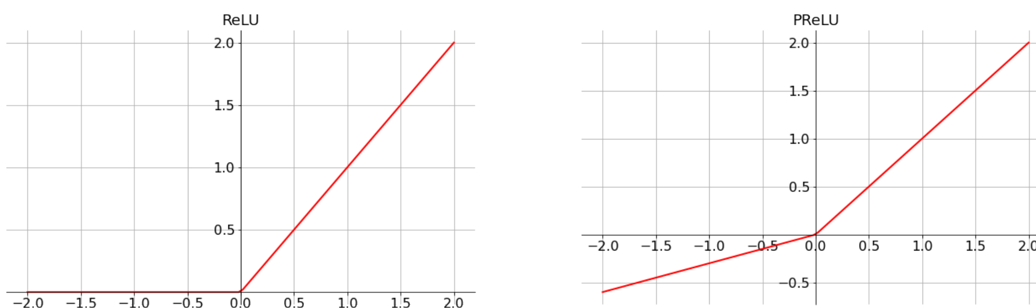
4. att. **Lineāras un Sigmoīdas aktivizācijas funkciju vizuālā reprezentācija**

ReLU – ReLu aktivizācija funkcijai viena milzīga priekšrocība, kas nav citām aktivizācijas funkcijām. ReLU ir visefektīvākā aktivizācijas funkcija, jo ReLU neaktivizē visus neironus, bet tikai dažus no tiem. Pēc 2.4. formulas varam redzēt, ka, ja summēšanas funkcijas rezultātā vērtība ir negatīva, tad tai tiek piešķirta 0 vērtība, bet, tiklīdz ir vairāk par 0, tad tā tiek aktivizēta. ReLU bieži pielieto dziļajāmašīnmācīšanā. ReLU aktivizācijas funkcijas vizuālo reprezentāciju var redzēt 5. attēlā.

$$f_{ReLU}(NET) = \begin{cases} 0, \text{ ja } NET < 0 \\ NET, \text{ ja } NET \geq 0 \end{cases} \tag{2.4}$$

PReLU – PReLU ir ReLU paveids, bet tai vietā, lai visas negatīvās vērtības automātiski pārveidotu par 0, PReLU izmanto α koeficientu. Tas palīdz un performē labāk situācijās, kur ir ļoti daudz negatīvas vērtības. Šī ļoti mazā izmaiņa ļauj labāk perfomēt tādās problēmās, kā: bilžu klasifikācijā vai objektu atpazīšanā. 5. attēlā var redzēt PReLU, kur $\alpha = 0.3$. 2.5. formulā var redzēt, kā izskatās šī funkcija matemātiski.

$$f_{PReLU}(NET) = \begin{cases} \alpha NET, \text{ ja } NET < 0 \\ NET, \text{ ja } NET \geq 0 \end{cases} \tag{2.5}$$



5. att. **ReLU un PReLU aktivizācijas funkciju vizuālā reprezentācija**

Kad runa ir par neirona tīkla darbību un rezultātiem, tad aktivizācijas funkcija ir ļoti svarīgs aspekts, ko ir vērts ņemt vērā, veidojot neirona tīkla modeli, tāpēc, ka dažas aktivizācijas funkcijas citās problēmas performē krietni sliktā kā kāda cita aktivizācijas funkcija. Piemēram, klasifikācijas problēmās sigmoīda funkcija sniedz labākus rezultātus. Ir pētījumi, kur ReLU izmantot neirona tīkla slēptajos slāņos, kamēr citas aktivizācijas funkcijas tam varētu būt pilnīgi nepiemērotas, kā piemēram: sigmoīda vai nepieminētā hiperboliskā tangensa funkcija. Šīs funkcijas nav piemērotas, jo funkcijas slīpums kļūst ļoti mazs, bet ievade kļūst ļoti liela vai ļoti maza, kas savukārt palēnina gradienta metodi, ko izmanto, lai optimizētu svarus.

Literatūra, nepiedāvā vienu pareizu atbildi, kā izvēlēties aktivizācijas funkciju, kas nozīmē, ka aktivizācijas funkcijas izpēte var būt pavisam cita pētāmā problēma, kam pievērsties. Tā kā, šī bakalaura darba mērķis nav pētīt aktivizācijas funkcijas, tad sīkāk par aktivizācijas funkcijām šī darba ietvaros netiks apskatīts.

2.2.3. Izejas funkcija, izejas slānis

Pēdējā darbība neirona tīklā ir izejas funkcijas (*Output function*), kas veido izejas slāni (*Output layer*). Izejas funkcija tiks apzīmēta ar f_{out} un izejas slāni jeb rezultātu tiks apzīmēts ar y vai y_i atkarībā no tā cik izejas neironu ir neirona tīklam. Arī izejas slāni var pierakstīt kā vektoru $Y = [y_1, y_2, \ldots, y_n]$, bet 2. attēlā mums ir dots neirons tikai ar vienu izeju, bet lielākoties neironam ir vairākas izejas.

Definēt izejas funkciju vispārīgā formā var sekojoši:

2.2. Definīcija. Pieņemsim, ka tiek dots j-otais neirons, tad izejas funkcija

$$f_{out}(a_j) = y_j$$

aprēķina rezultātu y_j j-otā neirona aktivizācijas stāvoklī a_j

Bieži literatūrā izejas funkcija ir tā pati aktivizācijas funkcija un izejas funkcijā bieži izmanto tās pašas funkcijas, ko pielieto aktivizācijas funkcijas gadījumā.

$$f_{out}(a_j) = a_j \text{ tātad } y_j = a_j$$

Nodaļa ir uzrakstīta pateicoties [13], [21], [17], [7] avotiem

3. GRAFU TEORIJAS PAMATJĒDZIENI

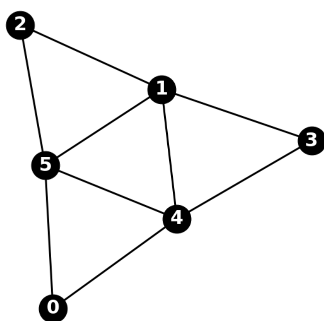
Iepriekš jau tika teikts, ka šī darba problēma ir atrast īsāko ceļu apceļojot pilnībā visas pilsētās. Lai šo problēmu efektīvi varētu atrisināt, tad šo problēmu var aprakstīt izmantojot grafu teoriju, jo pamatā tiek dots grafs ar noteiktu skaitu pilsētām un ceļu garumiem no vienas pilsētas dodoties uz otru pilsētu. Tāpēc šajā nodaļā tiks izklāstīts pavisam virspusēji par grafu teorijas pamatjēdzieniem, kas tiek pielietoti TSP un grafu neirona tīkla topoloģijā. Turklāt, vērts pieminēt, ka neirona tīkla arhitektūras pamatā arī ir grafs un grafu teorija.

3.1. Definīcija. Par grafu tiks saukts pāris $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, kur $\mathscr{V} \subseteq N$ ir virsotņu kopa jeb grafu neironu tīklu gadījumā tos varētu saukt arī par mezgliem/punktiem un TSP kontekstā $\mathscr{V} = \{v_1, v_2, \ldots, v_n\}$ reprezentēs pilsētas, bet $\mathscr{E} = \{(v_i, v_j) | 1 \leq i,j \leq |\mathscr{V}| \text{ un } i \neq j\}$ ir kopas $\mathscr{V}$ elementu pāru kopa jeb TSP kontekstā $\mathscr{E} = \{e_1, e_2, \ldots, e_m\}$ apzīmē to vai starp divām pilsētām ir ceļš.

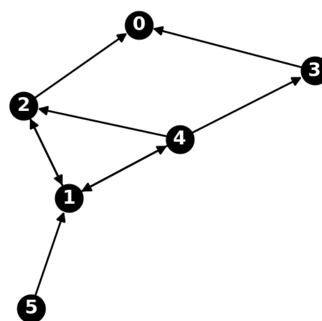
Kopa $\mathscr{E}$ sastāv no elementiem $e = (i,j) \in \mathscr{V} \times \mathscr{V}$, kas tiks saukti par lokiem. Grafa virsotnes skaits darbā tiks apzīmētas ar $|\mathscr{V}|$, bet loku skaits $|\mathscr{E}|$

3.2. Definīcija. Par neorientētu grafu tiek saukts pāris $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, kur $\mathscr{V}$ ir virsotņu kopa, bet $\mathscr{E}$ ir neorientētu loku kopa.

3.3. Definīcija. Par orientētu grafu tiek saukts pāris $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, kur $\mathscr{V}$ ir virsotņu kopa, bet $\mathscr{E}$ ir orientētu loku kopa. Šajā situācijā loku (i,j) sauks par virsotnes i izejošo loku, bet loku (j,i) par virsotnes i ieejošo loku.



(a) **Neorientēts grafs**



(b) **Orientēts grafs**

6. att. **Neorientētu un orientētu grafu vizuālais piemērs**

6. attēlā var vizuāli redzēt abu grafu atšķirības. Galvenā atšķirība starp neorientētu un orientētu grafu ir tā, ka neorientētu grafu gadījumā no vienas virsotnes uz otru virsotni var pārvietoties neatkarīgi no secības, bet orientētu grafu gadījumā secībai ir nozīme. Šī darba ietvaros tiks risināta problēma, kur tiek dots neorientēts grafs, jo nav svarīgi kādā secībā pārvietojamies caur visiem punktiem, bet svarīgākais ir saprast kurš punkts būs nākamais, lai veidotu īsāko ceļu.

Tā kā TSP problēmai nav nozīmes kādā secībā tiek pāriets no vienas virsotnes uz otru, tad turpmāk tiks apskatīts tikai neorientēts grafs un saistošie lielumi būs attiecināmi uz to.

3.4. Definīcija. Par nosvarotu grafu (grafu ar indeksētiem lokiem) $(\mathscr{V}, \mathscr{E}, S)$ sauksim grafu ar papildus struktūru - "svaru" funkciju $S : \mathscr{E} \to R$, kas katram lokam $e \in \mathscr{E}$ piekārto kādu skaitli (svaru) $S(e) \geq 0$.

Nosvarotu grafu $(\mathscr{V}, \mathscr{E}, S)$ sauc par neorientētu, ja grafs $(\mathscr{V}, \mathscr{E})$ ir neorientēts un $\forall (i,j) \in \mathscr{E}$ $S(i,j) = S(j,i)$. Ja grafs ir nosvarots, tad ceļa $C : (i_1, j_1), (i_2, j_2), \dots (i_{|\mathscr{V}|}, j_{|\mathscr{V}|})$ garums $L(C)$ definē kā svaru summu

$$L(C) = \sum_{i=1}^{|\mathscr{V}|} S(i_i, i_i). \tag{3.1}$$

Svaru koeficientiem vienmēr ir konkrēta interpretācija - tie var būt riska varbūtība, izmaksas u.c. TSP kontekstā svara koeficienti būs attālums no vienas pilsētas uz otru.

3.5. Definīcija. Par ceļu grafā $\mathscr{G}$ sauc virsotņu virkni $(v_k)_{k=1}^{m}$, kurai izpildās, ka $v_k - v_{k+1} \in \mathscr{E}$ $\forall k = 1, \dots, m-1$. Ceļu darbā apzīmēsim ar C.

3.6. Definīcija. Par grafa $\mathscr{G}$ ceļa garumu sauc tā loku skaitu $|\mathscr{E}|$.

Grafu var aprakstīt, izmantojot matricu, un otrādi – ja ir dots grafu, to var attēlot arī kā matricu. Šāda veida matricas ir pazīstamas kā kaimiņattiecību matricas, un to rindu un kolonnu skaits vienmēr ir vienāds ar grafa virsotņu skaitu $|\mathscr{V}|$. Katrai matricas rindai i un kolonnai j atbilst tikai viena grafa virsotne, un katru virsotni apzīmē ar naturālu skaitli. Matricas rindu un kolonnu secība vienmēr ir vienāda.

3.7. Definīcija. Tiek dots grafs $\mathscr{G}$ un loku sadalījumu apzīmēsim, izmantojot kaimiņattiecību matricu, kas tiks pierakstīta sekojoši $A \in \{0,1\}^{|\mathscr{V}| \times |\mathscr{V}|}$ $(i,j)-$ tais ieraksts kaimiņattiecību matricā tiek apzīmēts kā a_{ij}, kas reprezentē savienojumu starp divām virsotnēm v_i un v_j.

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1j} \\ \vdots & \ddots & \vdots \\ a_{i1} & \cdots & a_{ij} \end{pmatrix}$$

Neorientētiem grafiem matricas elementi ir sekojoši:

$$a_{ij} = \begin{cases} 1, \text{ja virsotnes } i \text{ un } j \text{ ir blakus virsotnes} \\ 0, \text{ja nav loka } i - j \end{cases}$$

Vienkāršiem vārdot sakot, ja starp divām virsotnēm pastāv savienojums, tad kaimiņattiecību matricā vērtība ir 1, bet citādi 0. Neorientētiem grafiem virsotne v_i ir blakus virsotne jeb kaimiņvirsotne v_j tad un tikai tad, ja v_j ir blakus virsotne v_i, turklāt $\forall(v_i, v_j)\ a_{ij} = a_{ji}$. Tātad, neorientētiem grafiem kaimiņattiecību matrica ir simetriska.

3.8. Definīcija. Kaimiņu kopa virsotnei v_i grafā $\mathscr{G}$ tiek apzīmēts kā $N(v_i) = \{u : (v,u) \in \mathscr{E}\}$, kas sevī iekļauj visas virsotnes, kas ir blakus virsotnei v_i. [18]

Iepriekš tika sniegts īss ieskats par grafiem un grafu teorēmu līdz ar to tagad var definēt ceļojošā pārdevēja problēmu matemātiski.

3.9. Definīcija. (*Travelling Saleman Problem*) Pieņemsim, ka $\mathscr{G} = (\mathscr{V}, \mathscr{E}, S)$ ir nosvarots Hamiltona grafs, kur $\mathscr{V} = \{v_1, v_2, \ldots, v_n\}$, $i : \mathscr{E} \to \mathscr{V} \times \mathscr{V}$ nosaka lokus un $S : \mathscr{E} \to \mathbb{R}_+$ ir nosvarotie loka garumi. Problēmu, kurā jāatrod īsākais Hamiltona cikls, tiek saukts par ceļojošā pārdevēja problēmu.

Attiecīgi šai problēmai būtisks parametrs, kam ir liela praktiska nozīmē, ir distanču matrica, ko definē sekojoši:

3.10. Definīcija. (*Distance matrix*) $|\mathscr{V}| \times |\mathscr{V}|$ matrica D ar sekojošiem elementiem

$$d_{ij} := \begin{cases} \min\{S(e) | e \in \mathscr{E}, i(e) = (v_i, v_j)\}, \text{ja eksistē tāds } e \\ \infty, \text{ja nepastāv tāds loks } e \end{cases}$$

tiek saukta par distanču matricu nosvarotam grafam $\mathscr{G}$.

Distance starp divām pilsētām tiek aprēķināta, izmantojot Eiklīda distances formula, jo problēma matemātiski tiek definēta 2D eiklīda telpā. Specifiski šīs formulas definēšanai apzīmēsim vienu pilsētu ar a un otru ar b, tad distance starp pilsētām tiek aprēķināta sekojoši:

$$S(e) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

Vienkāršiem vārdot sakot, elements d_{ij} apzīmē īsāko loku. Līdzīgi kā ar kaimiņattiecību matricu, tad arī distanču matrica D neorientētu grafu gadījumā būs simetriska. Ja nav loka no v_i uz v_j, tad $d_{ij} = \infty$. Gadījumā, ja ir vairāki loki no v_i uz v_j, tad tiek ņemts vērā īsākais, jo mums interesē īsākais Hamiltona cikls.

Attiecīgi arī tiks definēts Hamiltona cikls.

3.11. Definīcija. (*Hamilton cycle*) Tiek dots grafs $\mathscr{G}$. Cikls grafā $\mathscr{G}$ ir aizvērts ceļš, kas sākas un beidzas vienā un tajā pašā virsotnē. Hamiltona cikls (var arī teikt Hamiltona ceļš) grafā $\mathscr{G}$ ir cikls, kas krusto visas grafa $\mathscr{G}$ virsotnes.

Ja grafā ir vismaz viens Hamiltona cikls, tad grafu sauc par *Hamiltona grafu*. Vērts piebilsts, ka ne visiem grafiem piemīt Hamiltona ceļš, piemēram, grafs, kas ir pārtraukts, jo nepastāvēs savienojums starp daļām, kas nav savienotas. Tā pat arī grafiem, kas ir pilnībā savienoti, var būt situācija, ka nepastāv Hamiltona cikls un to ļoti labi var redzēt 6b. attēlā, kur, ja cikls sākas no piektās virsotnes, tad tā apstāsies pirmajā. Ja pat nesāksies cikls no piektās virsotnes, tad ceļš uz piekto virsotni nepastāv, tātad savienojums uz piekto virsotni arī nepastāvēs, kas izriet, ka Hamiltona cikls šim piemēram nav.

Visbeidzot TSP var formulēt arī kā lineārās programmēšanas problēmu, ko pirmo reizi lietoja Danzigs, Fulkersons un Džonsons [1]. Turpmāk darbā šo formulējumu apzīmēsim kā DFJ formulējums, kas tiks definēts sekojoši:

$$\min \sum_{i=1}^{|\mathscr{V}|} \sum_{j \neq i, j=1}^{|\mathscr{V}|} d_{ij} a_{ij} : \tag{3.2}$$

$$\sum_{i=1, i \neq j}^{|\mathscr{V}|} a_{ij} = 1, \qquad j = 1, \ldots, |\mathscr{V}|$$

$$\sum_{j=1, j \neq i}^{|\mathscr{V}|} a_{ij} = 1, \qquad i = 1, \ldots, |\mathscr{V}|$$

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} a_{ij} \leq |Q| - 1 \qquad \forall Q \subsetneq \{1, \ldots, |\mathscr{V}|\}, |Q| \geq 2 \tag{3.3}$$

Kur Q ir grafu apakškopa un 3.3. ierobežojums nodrošina, ka TSP atrisinājums nav mazāks par doto grafu, turklāt 3.3. sauc par *apakškopu izslēgšanas* ierobežojumu. Apakškopu ierobežojums pārbauda $O(|\mathscr{V}|^2 \cdot 2^{|\mathscr{V}|})$ mezglu apakškopas. Lai izvairītos no eksponenciāla pieauguma, apakškopas ierobežojumu var aizvietot ar heiristikas algoritmu, kas apskata tikai daļu no apakškopām. Apskatītās apakškopas ar heiristisko algoritmu visdrīzāk ir tie, kas varētu pārkāpt 3.3. ierobežojumu. Ar heiristiskā algoritma palīdzību laika sarežģītību ir iespējams samazināt no $O(|\mathscr{V}|^2 \cdot 2^{|\mathscr{V}|})$ līdz $O(|\mathscr{V}|^2)$. Ja atkārtoti lieto izvēlēto heiristiku un katru reizi, kad tiek konstatēti ierobežojumu pārkāpumi, tiek veikti labojumi un tiks sasniegts risinājums bez pārkāpumiem.

Tā pat kā DFJ formulējumā, tiek izmantots 3.2. izteiksme, bet apakškopas izslēgšanas ierobežojums heiristiskā algoritma gadījumā tiks aizvietots sekojoši:

$$cut(Q, \mathscr{V} - Q) \geq 1 \qquad \forall Q \subsetneq \{1, \ldots, |\mathscr{V}|\}, |Q| \geq 2, \tag{3.4}$$

kur $cut(S,T) = \sum_{i \in S} \sum_{j \in T} a_{ij}$. Tā pat kā DFJ formulējumā, 3.4. ierobežojums nodrošina, ka TSP nav atrisinājumi, kas neiekļauj visus mezglus. Un $|Q| \geq 2$ nodrošina, ka apakškopai ceļu daudzums nav mazāks par 2, jo, ja ir pilsēta, kurai ir tikai viens savienojums, tad pie tās pilsetas apstāsies risinājums, kas nozīmē, ka tā vairs nav TSP problēma. Katrai pilsētai ir jābūt ceļš no kuras ceļotājs ieceļo pilsētā un ir jābūt ceļš, kur ceļotājs dodas tālāk, jo, ja pilsētai ir tikai viens ceļš, tas nozīmē, ka ceļotājam ir jāatgriežas iepriekšējā pilsētā un tas ir apmeklējis vienu pilsētu divreiz, kas maina visu TSP būtību.

Tā kā darbā tiks apskatīta arī pavisam nedaudz TSP naivās metodes risinājums, tad vērts piebilsts, ka naivās metodes laika sarežģītība ir $O(|\mathscr{V}|!)$, kas nozīmē, ka izmantojot GNN un ACO laika sarežģītība tiks samazināta no $O(|\mathscr{V}|!)$ līdz $O(|\mathscr{V}|^2)$. Turklāt, laika sarežģītība ir viens no veidiem, kā salīdzināt dažādu algoritmu efektivitāti. Jo ātrāk algoritms performē, jo efektīvāks tas ir. Un šis ir arī viens no galvenajiem rādītājiem, kas parāda to cik sarežģīta tomēr ir TSP risināšana, kam jānodrošina īsākā ceļa atrašana pēc iespējas mazākā laikā.

Nodaļa tika izveidota pateicoties [1], [15], [14], [16], [5] literatūras avotiem.

4. GRAFU NEIRONA TĪKLI

Grafu neirona tīkli (GNN) ir vispārinājums no dziļāsmašīnmācīšanās neironu tīkliem (DNN). Kamēr DNN spēj apstrādāt vienkāršus režģus, kā: attēlus, video un tekstu, tad GNN spēj risināt jau neregulāras datu struktūras, kā piemēram, transportu tīklus, bioloģiskos tīklus un sociālus tīklus, kas bieži vien tiek apzīmēti ar grafiem.

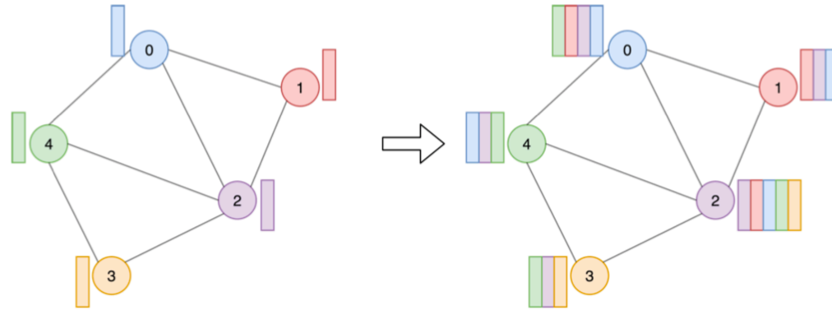
GNN vēsture un tās nozīme un veikspēja jau tika pētīta ap 1940. gadu, kad tika arī veidoti pirmie ANN modeļi. 1997. gados tika uzrakstīti raksti par rekursīvajiem neirona tīkliem, kuri tika izmantoti uz virzītajiem acikliskajiem grafikiem. Šo darbu autori ir Sperduti un Starita. Savukārt grafu konvulūciju tīkli pirmo reizi sāka pētīti 2010 gados, kad tika pētīts konvulūciju neirona tīklu pielietojums uz grafiem. Tā kā rezultāti bija ievērojami labi, tad tas arī veicināja padziļinātāku izpēti tieši GNN izpētē un 2014. gadā T. N. Kipfs un M. Vellings pirmo reizi piedāvā grafu konvulūciju tīkla modeli, kas ieviesa lokalizētu uz spektru balstītu konvolūcijas modeli uz grafa strukturētiem datiem. Tikai pēdējos gados GNN ir guvusi ievērojami lielu popularitāti un tās pētniecība aktīvi norisinās dažādās cilvēka dzīves jomās.

Neskatoties uz to, ka GNN ir ļoti specifisks ANN paveids, GNN tā pat spēj tikt galā ar daudz problēmām un brīžiem pat aizvietot citus ANN veidus, sniedzot labākus rezultātus, bet pārsvarā GNN tiek izmantotu, lai risinātu ar grafiem saistītas problēmas, kā piemēram: spēt atrast savienojumu starp moliem ķīmiskajā vielā un risināt dažāda veida problēmas, kas skar sociālas struktūras, prognozējot vai divi cilvēki ir pazīstami vai nav. Arī anomāliju noteikšanā GNN var būt ļoti noderīgi, meklējot krāpniekus, kas uzdarbojas online vidē, bet šie ir tikai dazi no piemēriem, ko GNN spēj atrisināt un to pielietojums ir ļoti plašs it īpaši medicīnā.

4.1. Grafu neirona tīkla darbības pamatprincipi

Grafu neironu tīklu galvenā ideja ir iteratīvi atjaunināt mezglu sūtijumus, apvienojot savu kaimiņu un savu pārstāvniecību sūtījumi.

GNN darbības vienkāršu principu var apskatīt 7. attēlā. No attēla var redzēt, ka sākumā katram mezglam ir informācija tikai par sevi pašu, bet GNN mērķis ir iegūt informāciju par katru no tās kaimiņu mezgliem jeb, ja nultais mezgls ir savienots ar pirmo, otro un ceturto, tad pēc pirmās iterācijas nultajam mezglam ir nepieciešamā informācija par šiem mezgliem, lai pieņemtu kādu lēmumu.



7. att. **Vienkārša vizualizācija GNN darbības principam [10].**

Šajā nodaļā tiks stāstīts par vispārīgu grafu neironu tīkla sistēmas darbību. Sākot no sākotnējā mezgla, katram slānim ir divas svarīgas funkcijas:

- **AGREGĒT** - funkcija, kas mēģina savākt/apvienot informāciju no visiem blakus esošajiem mezgliem;
- **KOMBINĒT** - funkcija, kas mēģina atjaunināt mezglu attēlojumus, apvienojot apkopoto informāciju no kaimiņiem ar pašreizējiem mezglu attēlojumiem.

Matemātiski var definēt vispārīgu grafa neirona tīkla struktūru sekojoši:

$$a_v^l = \textbf{AGREGĒT}^l\{H_u^{l-1} : u \in N(i)\} \tag{4.1}$$

$$H_v^l = \textbf{KOMBINĒT}^l\{H_v^{l-1}, a_i^k\}, \tag{4.2}$$

kur $N(v_i)$ ir kaimiņu kopa i-tajam mezglam, $k = 1, 2, \cdots, K$ un $H^0 = X$

4.2. Grafu neirona tīklu pielietojums TSP

Iepriekšējā nodaļā tika ļoti vispārīgi aprakstīts GNN darbības princips un, tā kā ir dažādas problēmas, ko var atrisināt ar GNN palīdzību, tad GNN ir jāpielāgo specifiski TSP situācijai. Daudz un dažādi darbi, kas ir orientēti uz GNN un TSP atrisināšana ar GNN parādās tikai pēdējos gadus, līdz ar to TSP risinājums, izmantojot GNN, ir diezgan jauna un tā ir vēl neizpētīta joma, kuru var vēl daudz pētīt.

Literatūra, kas ir jau pieejama, risina TSP ar grafu konvulūciju tīklu (GCN). Šī darba ietvaros arī tiks izmantoti GCN, lai spriestu vairāk par tās efektivitāti salīdzinājumā ar citām metodēm. Tūrklāt, struktūra, kas tiek ņemta šajā darbā pirmo reizi tika iepazīstināta 2017. gadā, kuras autori ir B. Ksavjers un T. Lorāns [4].

Ieejas vērtības: Kā ievades mezgla iezīme ir dotas divdimensiju koordinātas $x_i \in [0,1]^2$, kas tiek ievietota h-dimensiju elementos:

$$\alpha_i = A_1 x_i + b_1, \tag{4.3}$$

kur $A_1 \in \mathbb{R}^{h\times 2}$ un d_{ij} ir kā $\frac{h}{2}$ dimensiju vērtību vektors. δ_{ij}^{k-NN} ir indikatora funkcija TSP lokiem, kas ir 1, ja virsotne i un j ir k-tuvākie kaimiņi, 2, ja ir savienota pati ar sevi, un 0 pretējā gadījumā.

Ievada vērtība lokam β_{ij} ir aprakstīta sekojoši:

$$\beta_{ij} = A_2 d_{ij} + b_2 || A_3 \delta_{ij}^{k-NN} \tag{4.4}$$

kur $A_2 \in \mathbb{R}^{\frac{h}{2}\times 1}$, $A_3 \in \mathbb{R}^{\frac{h}{2}\times 3}$ un $\cdot||\cdot$ konkatenācijas operators. To, ka tiek izmantots k-tuvāko kaimiņu indikatora funkcija, paātrina mācīšanos procesu, jo TSP atrisinājumā virsotnes, kas atrodas tuvāk viena otrai, palielina iespējamību būt atrisinājumā.

Grafu konvolūcijas tīkla slānis: x_i^l tiks apzīmēts kā mezglu pazīme un e_{ij}^l būs loku pazīmes vektors l-tajā slānī i-tajā mezglā pie (i,j) loka. Līdz ar to x_i^l un e_{ij}^l pazīmes nākamajā slāni tiek definētas sekojoši:

$$x_i^{l+1} = x_i^l + \text{ReLU}\big(\text{BN}\big(W_1^l x_i^l + \sum_{j\sim i} \eta_{ij}^l \odot W_2^l x_j^l\big)\big), \tag{4.5}$$

$$e_{ij}^{l+1} = e_{ij}^l + \text{ReLU}\big(\text{BN}\big(W_3^l e_{ij}^l + W_4^l x_i^l + W_5^l x_j^l\big)\big), \tag{4.6}$$

kur $W \in \mathbb{R}^{h\times h}$, $\eta_{ij}^l = \frac{\sigma(e_{ij}^l)}{\sum_{j'\sim i}\sigma(e_{ij'}^l)+\epsilon}$, σ ir sigmoīda funkcija, ϵ ir maza vērtība un ar BN apzīmē grupas normalizāciju. Kad $l = 0$, tad x_i^l un e_{ij}^l ir vērtības no ieejas slāņa.

GCN izejas slānī ir sagaidāms, ka katram lokam/ceļam no pilsētas i līdz pilsētai j ir dota zināma varbūtība, ka konkrētais ceļš piederēs kopēja optimālā ceļa garumam. Lai iegūtu šo varbūtību, katram lokam, tam tiks izmantots pēdējais slānis, kur katra varbūtība ceļam (i,j) $p_{ij}^{TSP} \in [0,1]^2$ tiek dota izmantojot Daudzslāņu perceptronus (*Multi-layer Perceptron*(MLP)):

$$p_{ij}^{TSP} = \text{MLP}(e_{ij}^L)$$

Neirona tīkli sastāv no divām daļām. Pirmā daļa ir *Forward Propagation* un otrā daļa ir *Backward Propagation*. Pirmajā daļā svari tiek reizināti ar ieejas vērtībām un pēc tam izmanto summēšanas funkciju, lai visu sasummētu. Par šo procesu ir izklāsts 2. nodaļā. Bet otrā daļa ir process, kur tiek atjaunoti modeļa svari. *Backward Propagation* ir nozīmīga loma, kad runa ir par modeļa precizitāti, jo šajā procesā svari tiek atjaunoti tā, lai mazinātu kļūdu.

Svari tiek atjaunoti, izmantojot zaudējuma funkciju (*loss function*). Katrai problēmai un katrai situācijai zaudējuma funkcija var atšķirties, bet pamata ideja tiek ņemta no tā, ka mums ir zināms risinājums. Darbā modelis tiks uztrenēts uz TPS problēmām, kur ir doti punkti un pareizais risinājums. Tā kā rezultātā tiek dota kaimiņattiecību matrica ar p_{ij}^{TSP} elementiem, tad pareizo risinājumu pierakstīs kā $\hat{p}_{ij}^{TSP}$, apzīmējot vai starp pilsētu i un j pastāv ceļš vai nepastāv TSP optimālajā ceļā. Tiek minimizēts svērtais binārais krusteniskais entropijas zudumus, kas ir vidējais rādītājs no *mini-batch*. Problēmai palielinoties, klasifikācijas problēma kļūst ļoti nelīdzsvarota attiecībā pret negatīvo klasi, kas veicina nepieciešamību pret attiecīgu klašu svariem, lai šo efektu līdzsvarotu. Šie svari tiks aprēķināti sekojoši:

$$w_0 = \frac{|\mathscr{V}|^2}{(|\mathscr{V}|^2 - 2|\mathscr{V}|^2) \cdot c}$$
$$w_1 = \frac{|\mathscr{V}|^2}{2|\mathscr{V}|^2 \cdot c},$$

kur $c = 2$, kas apzīme klašu skaitu.

Kad GCN modelis tiek palaists, tad rezultātā tiek dota kaimiņattiecību matrica, kuras vērtība ir varbūtība, ka starp pilsētām ir ceļš. Lai aprēķinātu problēmas risinājumu, tad pēc varbūtību ķēdes likuma iespējamā ceļa $\hat{C}$ varbūtību aprēķina pēc sekojošās formulas:

$$p(\hat{C}) = \prod_{j' \sim i' \in \hat{C}} p_{i'j'}^{TSP}$$

Šajā brīdī rodas neliela problēma, jo dažreiz ir situācija, kad no vienas virsotnes ir vairāki ceļi ar augstām varbūtībām, kas nozīmē, ka GCN nesniedz rezultātā vienu no hamiltona grafu atrisinājumiem. Lai atrastu vienu TSP risinājumu, varētu izmantot arg max funkciju, kas izvēlas nākamo ceļu pēc lielākās varbūtības, bet tas izraisa situācija, ka atrisinājumam varētu būt, ka trūkst pilsētas, kas tika apskatītas, tādējādi neveidojot hamiltona ceļu.

[11] darbā un arī šajā darbā tiks izmantota alkatīgā pieeja (*greedy aproache*), lai atrast GCN TSP atrisinājumu. Alkatīgā metode balstās uz to, ka tiek izvēlēta pirmā virsotne un izvēlas nākamo pilsētu pēc maksimālās varbūtības, bet atšķirība no arg max funkcijas ir tāda, ka jau apmeklētā virsotne tiek fiksēta, tādējādi novēršot iespējamību atgriezties tajā virsotnē. Algoritms apstājas, kad tiek iziet cauri visām virsotnēm.

Nodaļa tika sarakstīta pateicoties [20], [11] avotiem.

5. ALTERNATĪVĀS METODES

Lai saprastu GNN darbību un to vai GNN ir vērts implementēt TSP problēmas risināšanai, tad TSP tiks atrisināts arī ar citām klasiskām metodēm un tās rezultāti salīdzināti ar alternatīvajām metodēm.

TSP tiks atrisināts, izmantojot naivo metodi, skudru kolonijas optimizācijas metode, dinamisko programmēšanu, held-kerp algoritmu un iepriekšējā nodaļā aprakstītos GNN.

Naivā metode tiks darbā implementēta tikai, lai saņemtu pareizo optimizācijas ceļu un šo ceļu salīdzināt ar citiem algoritmiem, lai saprastu cik precīzs ir aprēķinātais risinājums.

5.1. Naivā metode

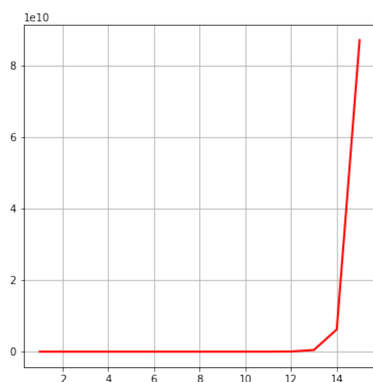
Kā jau sākumā tika minēts, tad naivā metode ir viena no klasiskajām metodēm, ko var izmantot, lai atrastu risinājumu esošajai problēmai. Naivās metodes darbības princips vairāk balstās uz datorzinību un programmēšanas loģiku un mazāk paskaidrots ar matemātiskiem principiem, jo tās algoritms ir pavisam vienkāršs un rada izaicinājumu tikai uzprogrammējot šo algoritmu.

Pieņemsim, ka mums tiek dots TSP problēma, lai atrisinātu risinājumu problēmai, izmantojot naivo metodi, tad izpilda sekojošu principu/algoritmu, kas tiek īstenots python programmēšanas valodā:

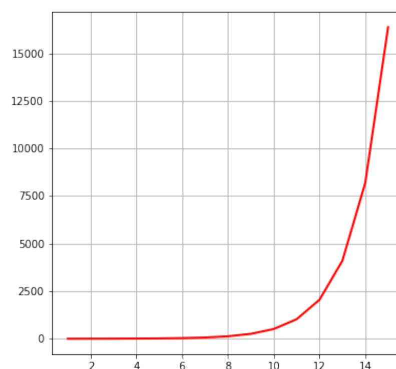
- Saskaita visus iespējamos ceļus;
- Sastāda sarakstu ar visiem iespējamajiem ceļiem. Šim nolūkam šī darba ietvaros tika izmantota funkcija *premuntation* no *iteration* bibliotēkas;
- Tiek iziets pilnībā visiem ceļiem un tiek fiksēts katra ceļa garums;
- Tiek izvēlēts ceļš ar īsāko garumu.

Šai metodei, kā jau sākumā arī tika minēts, ir viens milzīgs plus, kas nav pārējām metodēm. Tā sniedz 100% precizitāti jeb tā atrod īsāko ceļu, jo iziet pilnībā visiem iespējamajiem ceļiem, bet naivā metode pie liela pilsētu daudzuma īsti vairs nav praktiski pielietojama, jo izpildes laiks ļoti strauji pieaug katrai nākamajai pilsētai, kas tiek pievienots. Izpildes laiks noteikti korelē ar ceļu daudzumu grafā.

Ja tiek pieņemts, ka katrai virsotnei ir savienojums ar citām virsotnēm, tad naivās metodes gadījumā pēc formulas $(n-1)!$ var aprēķināt ceļu daudzumu, kur n šeit reprezentē pilsētu jeb virsotņu skaitu. -1 formulā ir, jo konkrētajai virsotnei nav savienojums pašai ar sevi.



(a) **Ceļu skaits naivajai metodei**



(b) **Ceļu skaits DP un/vai HK**

8. att. **Naivās metodes ceļu skaits salīdzinājums ar alternatīviem risinājumiem**

8. attēlā tiek vizuāli parādīts kā pieaug ceļu skaits (uz vertikālās ass) atkarībā no pilsētas daudzuma (uz horizontālās ass). Naivajai metodei ceļu skaits, kas ir redzams 8a. attēlā, tiek atbilstoši pielāgots jeb skaitļi uz asīm ir jāpareizina ar 10^{10}. Pēc attēla var ļoti labi redzēt, ka pie 15 pilsētām naivajai metodei ceļu skaits ir vairāk par $8 \cdot 10^{10}$, kamēr alternatīvajām metodēm, kas redzamas 8b., kuras tiks izklāstītas nedaudz vēlāk darbā, ir nedaudz virs $1{,}5 \cdot 10^4$, kas ir krietni mazāk līdz ar to paātrina risinājumu.

5.2. Dinamiska programmēšana un Held-Karp algoritms

Pirmais algoritms, kas tiks izmantots, lai atrisinātu TSP būs dinamiskā programmēšana un viena no tās modifikācijām Held-Karp algoritms. Kamēr DP ir spējīgs atrisināt dažāda rakstura uzdevumus, kas nav specifiski saistīts ar šo darbu, tad HK ir specifiski veltīts tieši TSP risināšanai. HK pirmo reizi tika izmantots 1962. gadā un Bellman, Helds un Karps [2] bija tie, kas pirmo reizi piedāvāja šo algoritmu. DP un HK laika sarežģītība ir $O(|\mathscr{V}|^2 \cdot 2^{|\mathscr{V}|})$, ko ļoti labi varēs arī redzēt praktiskajā daļā, bet tā kā DP pati par sevi ir vispārīga rakstura algoritms, tad konkrētās situācijās vai problēmās var perfomēt nedaudz lēnāk, kas arī būs spilgti redzams praktiskajā daļā.

Būtiskākā atšķirība DP un HK no naivās metodes ir tāda, ka DP un HK izmanto apakškopas, tādējādi ievērojami uzlabojot algoritma risinājumu, samazinot ceļu skaitu, kas tiek apskatīti.

Katrai virsotnei i un j un katrai virsotņu apakškopai X, kas izslēdz i un j, pieņemsim, ka $L_{HK}(i, X, j)$ apzīmē īsāko Hamiltona ceļu no i uz j inducētajā apakšgrafā $\mathscr{G}(X \cup \{i,j\})$. HK algoritmu ir balstīts uz sekojošu rekursiju:

$$L_{HK}(i,X,j) = \begin{cases} d_{ij} & \text{ja } X = \varnothing \\ \min_{v \in X}(L(i,X \setminus \{v\}, v) + d_{ij} & \text{pretējā gadījumā} \end{cases}$$

Šāds algoritms ir efektīvs, kad runa ir par vidēja izmēra TSP problēmu, kas ir ap 20 vai 50 pilsētām, lai gan praktiskajā daļā varēs redzēt, ka algoritmam jau pie 30 pilsētām ir sarežģījumi efektīvi izpildīt un sniegt risinājumu. Tā kā, ne naivā metode, ne arī DP ir spējīga atrisināt TSP lielām datu kopām, tad ir nepieciešami citi algoritmi. Algoritmi, kas nesniegts pilnībā precīzu atbildi, bet vismaz ir iespēja krietni optimizēt, tādējādi optimizējot arī izmaksas, kad runa ir par reālas dzīves problēmu.

5.3. Skudru kolonijas optimizācijas metode

Viens no pēdējiem algoritmiem, kas darba ietvaros tiek izmantots un salīdzināts ar GNN, ir Skudru kolonijas optimizācijas metode (*Ant colony optimization method* (ACO)), kas ir heiristiskā meklēšanas algoritms.

Heiristisku meklēšanu algoritmu būtība ir balstīta uz mašīnmācīšanās principa, kur tiek izmantoti kādi vēsturiska informācija, lai atrisinātu problēmu. Bieži vien heiristiskie algoritmi precizitāti maina pret ātrumu, ko simulācijas un rezultātu nodaļā varēs arī ļoti labi novērot.

ACO tika ieviesta ar Dorigo palīdzību, kur pirmo reizi viņš risināja TSP izmantojot ACO savā disertācijā. ACO ir bieži izmantojama tieši TSP risināšanai un citu diskrētu kombinatorikas optimizācijas problēmu risināšanai.

Šī metode atšķiras no darbā jau iepriekš apskatītajām metodēm vairāku iemeslu dēļ. Šī metode tika iedvesmota ar reālu dzīves piemēru no dabas un ātrās performances dēļ. ACO metodes iedvesma nāk tieši no skudrām, kur skudras izmanto feromonu, kas palīdz viņām atrast īsāko ceļu starp skudru pūzni un pārtikas avotu.

Šī metode sniedz tuvinātu risinājumu lielām datu kopām. Ar šo metodi viennozīmīgi var samazināt izmaksas un pāris sekunžu laikā ir spējīga arī atrast TSP risinājumu pat vairāk kā 100 pilsētām, ko savukārt ne naivā metode, ne arī DP vai HK nespēj paveikt.

ACO ir ļoti daudzi un dažādi veidi kā atrisināt TSP problēmu un būtiskākā atšķirība starp visām metodēm ir tieši feromona τ_{ij} daudzuma aprēķināšana katram ceļam. Tā kā šī darba mērķis nav specifiski izpētīt ACO, bet gan saprast vai ar neirona tīkla palīdzību TSP problēmu var atrisināt veiksmīgāk nekā standarta metodes, tad darba ietvaros tiek apskatīta tikai viens no ACO iespējamajiem risinājumiem.

5.3.1. TSP risinājums ar skudru kolonijas optimizācijas palīdzību

ACO mērķis: Visas skudras atrodas skudru pūznī un viņām ir jāatrod ēdiens un jāatnes atpakaļ uz skudru pūzni. ACO mērķis tiek izpildīts pēc sekojošiem soļiem:

1. Vispirms skudras sāk kustēties randomizēti;
2. skudrām kustoties un meklējot ēdienu, tiek izdalīts feromons;
3. feromons piesaista citas skudras. Tiklīdz skudras atrod feromonu, tad tās sāk sekot feromonam nevis pārvietojas randomizēti;
4. ar laiku feromona daudzums palielinās, kas piesaista vairāk skudru;
5. pēc kāda laika, ceļš ar labākiem rezultātiem, paliks ar vien izteiktās par pārējiem, bet ceļi ar sliktāku performanci ar laiku izzudīs, kas noved līdz tam, ka skudras atrod īsāko ceļu līdz ēdienam.

Iepriekš aprakstītā skudru darbība ir attiecināma arī uz ACO, tāpēc turpmāk definēsim ACO specifiski TSP problēmai.

Katrā solī ACO skudrām ir jāizvēlas nākamā pilsēta uz kuru doties. Lai skudras zinātu, kur doties, tad tiek izmantots varbūtības funkcija, jo pirmā pilsēta tiek izvēlēta randomizēti. Kad skudra atrodas i-tajā pilsētā, k-tajai skudrai jāizvēlas nākamā pilsēta j kur doties un to aprēķina izmantojot sekojošu formulu:

$$p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}^\alpha(t)(\eta_{ij})^\beta}{\sum_{c_{il} \in N(s_k^p)} \tau_{il}^\alpha(t)(\eta_{ij})^\beta}, \text{ ja } j \in N(s_k^p) \\ 0 \text{ pretējā gadījumā} \end{cases},$$

kur $\tau_{ij}-$ feromona daudzums ejot no pilsētas i uz pilsētu j, ko skudras atstāj uz ceļa, $\eta_{ij}-$ heiristiskā redzamība, kas saistīts ar pieejamām risinājuma komponentēm c_{ij}. α un β ir konstantes un $N(s_k^p)$ ir komponenšu kopa, kura vēl nepieder pie daļveida risinājumiem s_k^p.

α tieši ietekmē feromonu, jo, ja $\alpha = 0$, tad tas nozīmē, ka lielāko svari tiek doti kaimiņa īsākajam ceļam, kas nozīmē, ka ACO risinājums ir pielīdzināms *greedy approach* algoritmiem. Savukārt β ietekmē heiristisko redzamību un, ja $\beta = 0$, tad tikai feromonu daudzums tiek ņemts vērā. Dažās situācijās, atkarībā no pilsētu izvietojuma, ir izdevīgi palielināt vai samazināt kādu no šīm konstantēm, dodot lielākus svarus kādam no parametriem.

Kā jau iepriekš darbā tika minēts, tad būtiskākā atšķirība starp dažādajiem ACO algoritmiem ir tieši feromonu aprēķināšana. Feromonu atstāšana uz ceļa notiek divos līmeņos: lokāli un globāli.

- **Lokālais feromonu atjaunošana** - Ikreiz, kad skudra ir izvēlējusies pilsētu, kuru iet, feromonu daudzums, ko skudra atstās uz ceļa, tiks atjaunots pēc sekojošas formulas:

  $$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \rho\tau_0,$$

  kur $0 < \rho \leqslant 1$ iztvaikošanas parametrs, $\tau_0 = \frac{1}{|\mathscr{V}| \cdot L_{nn}}$, kur L_{nn} ir kopējais attālums starp visiem mezgliem, kuru parāda viena no konstruēšanas heiristikām.

- **Globālais feromonu atjaunošana** - Pēc tā, kad visas skudras ir apceļoja visus mezglus īsāka ceļa atrašanai, tad tiek atjaunots feromona koncentrāts un to dara pēc sekojošas formulas:

  $$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}^{gl}(t),$$

  kur $\tau_{ij}^{gl}(t) = Q/L_{ij}^{gl}$ un L_{ij}^{gl} ir īsākais kopējais ceļš, ko kāda skudra k, ir atradusi, kādā laika solī t un Q - konstante.

Feromonu iztvaikošanas parametram ρ ir nozīme algoritma pārāk ātrai konverģencei jeb tās palēnināšana, jo, ja nebūtu iztvaikošanas parametra, tad ir lielāka varbūtība, ka skudras atradīs lokālo minimumu. Iztvaikošana veicina ceļu aizmiršanu, kas palīdz skudrām meklēt jaunus ceļus/apgabalus, tādējādi atrodot iespējams labāku ceļu.

Feromonu vadītas, skudru kolonija pakāpeniski uzlabo risinājumu un algoritms izbeidz darbību, kad tiek izpildīts konkrēts iterāciju skaits, kur iterāciju skaitu nosaka lietotājs.

Nodaļa sarakstīta pateicoties [19], [3], [9] avotiem.

6. PRAKTISKĀ DAĻA

Visbeidzot, lai saprastu GNN efektivitāti salīdzinājumā ar klasiskajām metodēm, tad tika veikta simulācija, kur visām izmantotajām metodēm tiek ielaisti vieni un tie paši dati, lai varētu godīgi salīdzināt katru metodi savā starpā. Praktiskajā daļā ļoti nedaudz tiks parādīta naivās metodes darbības efektivitāte. DP un HK algoritms salīdzinājumā savā starpā un ar mašīnmācīšanās algoritmiem, kā GNN un ACO.

6.1. Datu kopa un GCN modelis

Darba ietvaros tiks izveidotas divas datu kopas: TSP20 un TSP50. TSP20 apzīmē grafu ar 20 pilsētām, bet TSP50 apzīmē grafu ar 50 pilsētām. Arī citur literatūrā šīs ir bieži izmantotas datu kopas. Vērts piebilsts, ka algoritma ātrums ir atkarīgs no vairākiem faktoriem un, kas nav tieši saistīts ar pašu algoritmu, bet gan ar tehniskiem ierobežojumiem un datora veiktspējas.

Lai uztrenētu GCN, tad tika ņemts 30% no tā, ko izmantoja Joshi 2020. gada rakstā, bet pārējie lielumi tiks atstāti nemainīgi jeb *learning rate* ir 10^{-4} un *batch size* ir 20. Zaudējuma funkcija tiek optimizēta, izmantojot *Adam* gradienta metodi [12]. Un grafu atšifrēšanai tiks izmantota alkatīgā pieeja (*greedy aproatch*), ko Joshi izmantoja arī savā darbā [11].

6.2. Simulācija

Lielākām datu kopām, izmantojot naivo metodi, ir grūti dabūt optimālo ceļu, tāpēc, lai aprēķinātu īsāko ceļu ātrā laikā, tiks izmantota izveidots rīks, kas ir tieši domāts, lai risinātu simetrisku TSP. Šo programmu sauc *concorde* [6]. Šis rīks ir paredzēts tikai akadēmiskiem nolūkiem un ir brīvi pieejams. Concorde darba nolūkos tika implementēts python vidē, kur arī tika veikts viss praktiskais darbs.

Darba ietvaros tika uzsimulēti divi x un y koordinātes pēc nejaušības principa un, izmantojot divdimensiju eiklīda distances formulu iegūstot attālumu no punkta i līdz punktam j, un šī distance tiks apzīmēta ar d_{ij}.

Lai aprēķinātu īsāko ceļa garumu, tad tiks izmantota 3.1., bet nedaudz pielāgosim specifiski darba problēmas kontekstam, jo ceļotājs sāk un beidz ceļu no vienas un tās pašas pilsētas.

Pieņemsim, ka $C = \{c_1, c_2, c_3, \ldots, c_{|\mathscr{V}|}, c_1\}$ ir atrisinājums TSP problēmai ar $|\mathscr{V}|$ pilsētām, kur $c_i \in \mathscr{V}, 1 \leq i \leq |\mathscr{V}|$. Šajā kontekstā C ir apceļoto pilsētu secība jeb ideja ir tāda, ka

cēļš sākas un beidzas vienā un tajā pašā pilsētā. Lai aprēķinātu šī ceļa garumu specifiski TSP situācijai izmantosim sekojošu formulu:

$$L(C) = \sum_{i=1}^{|\mathscr{V}|-1} d_{c_i c_{i+1}} + d_{c_{|\mathscr{V}|} c_1},$$

kur mērķis ir atrast īsāko ceļu $min(L(C))$. Tā kā praktiskās daļās laikā tika veikts novērojums, ka ACO performē krietni sliktāk, ja datu punkti x un y tiek doti robežās no $[0,1]$. Šādi varētu notikt, jo skaitļi atrodas ļoti tuvu viens otram, tāpēc ACO ieejas punkti tiks pareizinātas ar konstanti. Darba ietvaros ACO tika izvēlēta $\alpha = 0.1$ un $\beta = 5$, liekot lielāku uzsvaru uz feromonu daudzuma, mazinot kaimiņa mezglu ietekmi. Kā arī, ACO performances laiks ir atkarīgs no tā, cik daudz skudras tiek izmantotas vienas iterācijas laikā, tāpēc visi rezultāti tiks attiecināmi ar $k = 30$ un iterāciju daudzums ir 100. Šādi lielumi tika izvēlēti, jo praktiskās daļas ietvaros, uzrādīja labākus/precīzākus rezultātus.

6.3. Algoritma novērtēšana

Lai varētu spriest par algoritma atbilstību reālās problēmas implementācijai, tad tiks izmantoti sekojošie rādītāji:

- Algoritma izpildes laiks. Jo ātrāk atrisina, jo labāk, jo TSP tiek klasificēts, kā NP-sarežģītības uzdevums;
- Optimalitātes tuvums (*optimality gap*), ko rēķina pēc sekojošas formulas: $\frac{l - l_{opt}}{l_{opt}} \cdot 100\%$, kur l ir algoritma izrēķinātais optimālais ceļa garums, bet l_{opt} īstenībā īsākais ceļš;
- Algoritma sarežģītība, kas ir autora subjektīvs mērījums, kur tiks ņēma vērā to, cik viegli ir implementēt no biznesa perspektīvas un to, cik viegli ir interpretēt konkrētā algoritma rezultātus.

Utopiskā situācijā vēlamākais rezultāts būtu, ja algoritms risina problēmu pāris sekundēs, optimalitātes tuvums ir 0% vai mazāks par 5% un ir ļoti viegli izpildīt un implementēt, bet šīs nodaļas ietvaros tiks attaisnoti visi trīs novērtēšanas parametri, kur būs redzami, ka algoritmiem lielākoties tikai viens no rādītājiem būs ļoti labs, kamēr pārējie ir slikti, kas parāda, ka TSP problēmas atrisināšanai vēl ir tāls ceļš ejams no tā, kas mūsdienās jau ir pieejami.

7. REZULTĀTI

2. tabulā var redzēt visu metožu rezultātus un tās rādītājus. TSP20 gadījumā varam redzēt, ka BR, DP un HK sniedz 100 % precizitāti, bet BR prasīja vairāk kā 15 minūtes, lai šo problēmu atrisinātu. Teorētiskajā daļā jau tika izklāstīts, ka lielām instancēm BR ir ļoti neefektīva, jo tiek apskatītu visu iespējamās ceļu variācijas. TSP50 pagāja vairāk kā dienakts līdz BR sāka sniegt rezultātus. BR bija gana efektīva problēmai ar 11 pilsētām, atrisinot to mazāk kā minūtes laikā. Ja bija vairāk kā 11 pilsētas, tad algoritms bija ievērojami lēnāks, jeb ilgāk kā 10 minūtes.

DP un HK performē vidēji ātri, atrisinot TSP20 vienu līdz divu minūšu ātrāk, turklāt var arī redzēt, ka HK atrisina ātrāk. Tā kā divas minūtes nav ilgs laiks un tas sniedz optimālo ceļu, tad TSP20 gadījumā ir viennozīmīgi labāk izmantot šo algoritmu vienkāršības pēc.

	TSP20			TSP50 [1]		
Metode	Īsākais ceļš	Opt. tuv.	Laiks	Īsākais ceļš	Opt. tuv.	Laiks
BR	4.16	0.00 %	>15 min	5.57	0.00 %	> 1 dienu
DP	4.16	0.00 %	2 min , 30 sec	3.52	0.00 %	1 st, 2 min
HK	4.16	0.00 %	1 min, 34 sec	3.85	0.00 %	25 min
ACO	4.29	0.33 %	10 sec	6.97	7.46 %	14 sec
GCN	4.33	0.58 %	8 sec	5.74	3.29 %	1 min, 5 sec

2. tabula

TSP risinājums ar GCN un alternatīvajām metodēm

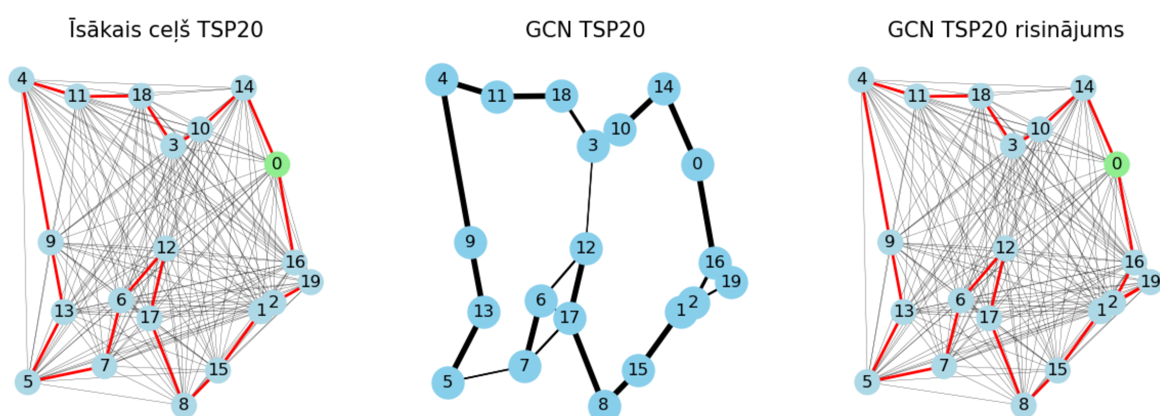
TSP20 gadījumā ACO ātruma ziņā ir līdzīgs GCN, bet ACO minimāli ir labāks par GCN. Zinot to, cik sarežģīti ir implementēt GCN, tad TSP20 gadījumā noteikti ir labāk implementēt ACO, lai gan DP un HK sniedz precīzu rezultātu, tad varbūt ACO šajā situācijā arī nebūtu izvēle. Savukārt pavisam cits rezultāts ir TSP50 gadījumā, kur BR, DP un HK aizņem ļoti ilgi, lai sniegtu rezultātu, ACO rezultātu sniedz ļoti ātri, bet sniedz rezultātus ar 7.5 % neprecizitāti. GCN sniedz rezultātus krietni precīzāk nekā ACO, bet nedaudz ilgāk, kas nav tik būtiski, jo tas ir tikai minūšu intervālā.

TSP50 gadījumā BR, ACO un GCN metodēm ir rezultāti no 50 pilsētām, jo salīdzinājumā BR, ACO un GCN ar DP un HK, DP un HK algoritms pieprasa saglabāt atrastās vērtības, kas apgrūtina datora skaitļošanas procesus. Lai DP un HP iegūtu rezultātu, šīm metodēm ir jāap-

[1] DP algoritmiem salīdzinājumam tiek izmantotas 23 pilsētas, bet HK 24 pilsētas nevis 50. Kas nozīmē, ka tiek izmantotas citas datu kopas.

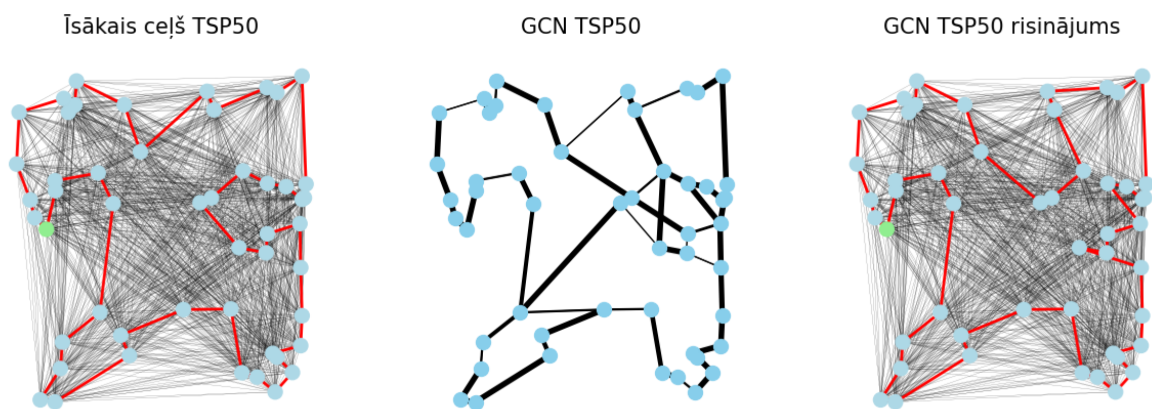
skata $2^{|\mathscr{V}|}$ apakškopas. Tas nozīmē, ka TSP20 gadījumā tās būtu $1 \cdot 10^6$ apskatītās apakškopas. Savukārt TSP50 gadījumā DP un HK jāapskata $1 \cdot 10^9$ reizes vairāk apakškopu nekā TSP20 un ir jāsaglabā krietni vairāk datu. Ņemot vērā šo lielo skaitļošanu daudzumu, salīdzinājumam 2. tabulā pie TSP50, DP gadījumā tiek ņemtas 23 pilsētas un HK tiek ņemtas 24 pilsētas nevis 50 pilsētas. Ņemot mazāk pilsētu skaitu gan HK un DP, redzam, ka izpildes laiks ir stunda, kas liek ticēt, ka ar 50 pilsētām DP un HK izpildīs šo uzdevumu vairākas dienas, kas nozīmē, ka šādām problēmām šie algoritmi ir bezspēcīgi, lai gan tie sniedz precīzus rezultātus.

9. attēlā var redzēt TSP20 risinājumu ar GCN. Attēlā ir redzama īsākā ceļa atrisinājums, otrajā attēlā var redzēt, ka ir arī citi ceļi, bet lielākās varbūtības tomēr ir tiem ceļiem, kas ir redzams pirmajā attēlā. Vidējā bildē var redzēt, ka, jo biezāka līnija, jo lielāka varbūtība, ka ceļš pieder optimālajam ceļam. Visbeidzot, izmantojot alkatīgo algoritmu (GS), lai atrastu GCN īsāko ceļu. Trešajā attēlā risinājums ir ļoti līdzīgs pareizajam, bet vienā vietā GCN GS izvēlējies nedaudz citādāku ceļu.



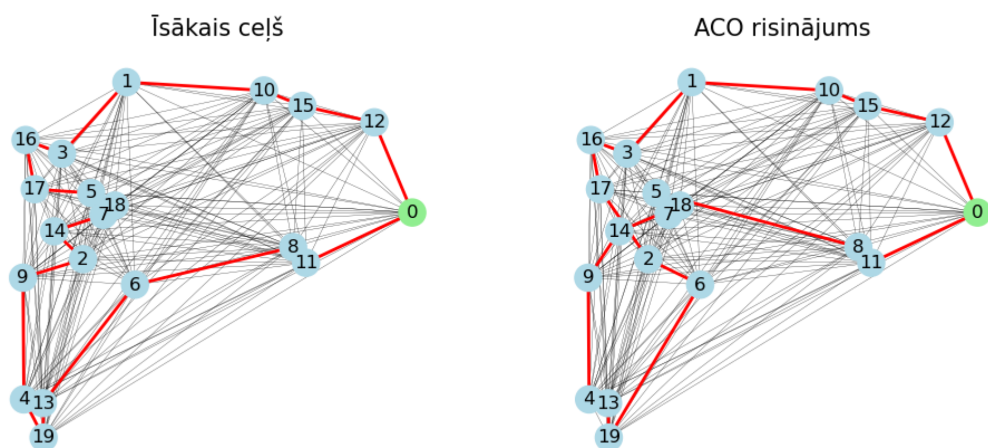
9. att. **GCN risinājuma piemērs TSP20**

10. attēlā var redzēt GCN risinājumu 50 pilsētām. Var redzēt, ka lielākam pilsētu daudzumam, ir vairāk pilsētas ar varbūtību piederēt optimālajam ceļam. Turklāt, atrisinājumā redzam nedaudz citu ceļu, nekā tas ir optimālajā atrisinājumā. Lasot citus literatūras avotus, varēja arī ieraudzīt, ka GCN atrod īsāku ceļu nekā optimālā ceļa programma, kas nozīmē, ka concorde nesniedz īsāko ceļu, bet tā kā tas tomēr sniedz labus rezultātus, tad šī darba ietvaros mēs pieņemam, ka concorde risinājums ir optimālais risinājums.



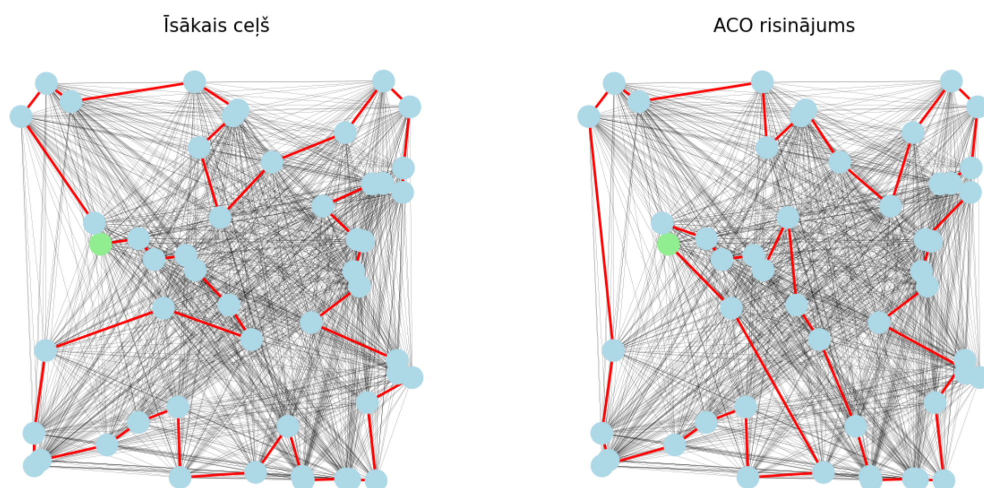
10. att. **GCN risinājuma piemērs TSP50**

11. attēlā var redzēt ACO risinājumu TSP20 datu kopai. ACO rezultāti ļoti ietekmējas no tās konstantēm. Tajā brīdī, kad $\alpha = 1$ un $\beta = 1$, tad risinājums vairāk ņem blakus pilsētās, kas noved līdz tam, ka ceļi kaut kādā brīdi krustosies. Uzskatāmi to var redzēt 11. attēlā, kur abas konstantes ir ir vienādas ar 1 un skudru daudzums vienā iterācijā ir 1.



11. att. **ACO risinājuma piemērs TSP20**

Tieši tā pat kā GCN gadījumā, ka modelis kļūst nestabils brīdī, ka pilsētu daudzums palielinās, tad ACO ir arī tas pats, tikai tās optimalitātes tuvums pieaug ātrāk nekā tas ir GCN. 12. attēlā va redzēt, ka tīri vizuāli ACO risinājums atšķiras no optimālā risinājuma, lai gan dažās vietās šie ceļi sakrīt. Šajā attēlā ACO parametri tika izvēlēti, kas tiek aprakstīts sadaļā pie simulācijas un var redzēt, ka lielām datu kopā tiek novērsta krustošanās, bet atstājot konstantes 1, šim atrisinājumam būtu vairāk kā viens risinājums.



12. att. **ACO risinājuma piemērs TSP50**

Un runājot par katras metodes vienkāršību un tās iespējām implementēt potenciāli reālās dzīves situācijās un biznesā, tad BR būtu pati vienkāršākā no visām. Arī DP un HK nav sarežģītas. ACO jau prasa vairāk zināšanas un spēju veikt aprēķinus, bet ir labs rīks, lai ātri atrisinātu arī problēmas ar lielu pilsētu skaitu. Savukārt GCN no visām metodēm ir vissarežģītākā un visdrīzāk prasītu daudz resursus implementēt biznesā tīri to sarežģītās uzbūves dēļ, toties, lielām problēmām, kas ir virs 50 pilsētām, varētu būt noderīgi implementēt un veltīt resursus.

NOBEIGUMS

Darba ietvaros tika izpētīta TSP problēma un noskaidrots, kāpēc šī problēma ir izaicinājums matemātiķiem un datorzinātņu speciālistiem. TSP tika atrisināts, izmantojot primāri 4 metodes un implementējot naivo metodi kopā ar pārējām metodēm. Tika izpētīts GNN, tā pielietojums un attīstība un kādas ir iespējas risināt TSP izmantojot GNN un izpētīts, cik efektīvi GNN perfomē attiecībā pret alternatīvajām metodēm.

Lai atrisinātu TSP izmantojot GNN, tiek izmantots GCN, kas ir viens no GNN veidiem kā atrisināt problēmas par grafiem un tās lokiem. TSP20 datu kopas gadījumā var redzēt, ka precizitāte GNN ir vissliktākā, lai gan tā atšķirība ir ļoti minimāla, ka nevar apgalvot, ka GNN ir slikts variants, lai pielietotu GNN, toties GNN implementācija ir diezgan sarežģīta un, raugoties no biznesa puses, būtu kārtīgi jāizvērtē cik daudz laika un līdzekļus ir iespējams veltīt GNN implementācijai tās sarežģītās struktūras dēļ. Vēljovairāk ir vērts piebilsts, ka, neskatoties uz to, ka ANN uzrāda labākus rezultātus nekā klasiskās metodes, tāpat uzņēmumi izvēlas klasiskās metodes, jo tās ir vienkāršas un viegli interpretējamas, neprasos daudz papildus resursu, bet ANN un vēljovairāk GNN ir jaunas un sarežģītas metodes, kas ir jāturpina izpētīt, lai varētu vispārināt un padarīt tos pieejamākus plašākam sabiedrības lokam.

Bet, turpinot par rezultātu apkopojumu, TSP gadījumā var redzēt, ka DP un HK algoritmi prasa daudz laika, lai izpildītu. DP algoritms ar 23 pilsētām sniedz rezultātu stundas laikā. No 20 pilsētām līdz 23 pilsētām DP izpildes laiks ir palielinājies par 58 minūtēm, kas liek ticēt, ka pie 50 pilsētām DP risinās problēmu vairākas dienas. Līdzīgs novērojums ir redzams arī HK, tikai tas performē krietni labāk. 23 pilsētam HK algoritms atrisināja problēmu 10 minūtēs. Ar 24 pilsētām HK atrisināja 25 minūšu laikā un ap stundu HK sasniedz pie 25. pilsētas, kur var arī labi redzēt, ka pavisam nedaudz HK ir efektīvāks nekā DP.

Un pēc rezultātiem var arī redzēt, ka mašīnmācīšanās metodes ar lielām instancēm spēj strādāt diezgan efektīvi, bet efektivitāte/ātrums tiek samainīts ar precizitāti. Pēc rezultātiem var arī redzēt, ka GCN ir spējīgs sniegt labākus rezultātus kā ACO.

Pati problēma un GNN ir ļoti plašas tēmas, ko apskatīt. Šo darbu var paplašināt un turpināt pētīt daudz un dažādos virzienos. TSP problēmu varētu pielīdzināt vairāk reālajai dzīvei jeb ņemt vērā to, ka bieži vien no vienas pilsētas līdz otrai īsti nav tiešā ceļa un to, ka no vienas pilsētas līdz otrai nav tāds pats ceļa garums, kā braucot atpakaļ no pilsētas uz kuru tika aizbraukts. Kā arī GCN nav vienīgais risinājums, ko pašlaik pieejamā literatūra piedāvā, tāpēc darbu var turpināt, pētot dažādas GNN metodes ar citiem staru meklēšanas algoritmiem. Skudru kolonijas

optimizācijai arī uzrāda diezgan labus rezultātus, līdz ar to, būtu arī interesanti atrast un izpētīt veidus kā novērtēt konstantes tā, lai būtu pēc iespējas labāka precizitāte, bet tad visdrīzāk tas kļūs par ANN modeli, kas ir spējīgs pielāgot zināmas vērtības konkrētai situācijai. Šī darba ietvaros netika smalki pētītas ACO konstantes, bet noteikti būtu tās vērts apskatīt detalizētāk.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Implementing the dantzig-fulkerson-johnson algorithm for large traveling salesman problems. *Mathematical programming*, 97:91–153, 2003.

[2] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

[3] Paul Bouman, Niels Agatz, and Marie Schmidt. Dynamic programming approaches for the traveling salesman problem with drone. *Networks*, 72(4):528–542, 2018.

[4] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[5] P. Daugulis. Grafu teorija. 2005.

[6] Vašek Chvátal David Applegate, Robert E. Bixby and William J. Cook. Concoder tsp solver.

[7] AD Dongare, RR Kharde, Amit D Kachare, et al. Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*, 2(1):189–194, 2012.

[8] IEAC Droste. Algorithms for the travelling salesman problem. B.S. thesis, 2017.

[9] Pengzhen Du, Ning Liu, Haofeng Zhang, and Jianfeng Lu. An improved ant colony optimization based on an adaptive heuristic factor for the traveling salesman problem. *Journal of Advanced Transportation*, 2021:1–16, 2021.

[10] Chaitanya K. Joshi. Graph neural networks for the travelling salesman problem.

[11] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

[12] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] David Kriesel. *A Brief Introduction to Neural Networks*. 2007.

[14] Līva Elizabete Liepiņa. Dažādas pieejas grafa virsotņa centralitātes noteikšanai. B.S. thesis, 2022.

[15] Yao Ma and Jiliang Tang. *Deep learning on graphs*. Cambridge University Press, 2021.

[16] Quang Nhat Nguyen. Traveling salesman problem and bellman-held-karp algorithm, 2020.

[17] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[18] Chuan Shi, Xiao Wang, and Cheng Yang. *Advances in Graph Neural Networks*. Springer Nature, 2022.

[19] Thomas Stützle, Marco Dorigo, et al. Aco algorithms for the traveling salesman problem. *Evolutionary algorithms in engineering and computer science*, 4:163–183, 1999.

[20] Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, and Le Song. *Graph neural networks*. Springer, 2022.

[21] Janis Zuters. *Neironu tīkli. Mācību materiāli*.

PATEICĪBAS

Bakalaura darba autors izsaka pateicību Latvijas Universitātes Datorikas fakultātes lektoram un Elektronikas un Datorzinību institūta vadošajam pētniekam Dr. sc. comp. Kārlim Freivaldim par konsultāciju sniegšanu par grafu neirona tīkliem.

Autors pateicību vēlas izteikt arī saviem kursabiedriem un ģimenei par emocionāla atbalsta sniegšanu darba izstrādes laikā un paldies arī bakalaura darba vadītājam Mg. math. Artim Alksnim par veltīto laiku, lai šo darbu varētu īstenot.

PIELIKUMI

A. PYTHON PROGRAMMAS KODS

```
import inspect
import networkx as nx #packacge for graphs and graph visualization
import pandas as pd
from scipy.spatial import distance_matrix
import numpy as np
import matplotlib.pyplot as plt #libary that helps with vizualization
import itertools
import math
import os #libary that helps with paths
import sys #system values. Used to get time
from sys import maxsize
from itertools import permutations #library who shows all the possible
    answers of a list
import time #sets time and helps to count how many seconds does cell work
import warnings
warnings.filterwarnings("ignore") #line that helps to ignore unsecery
    warnings
from main import AntColony #library where is ant colony code
```

1. pirmkods. Izmantotās bibliotēkas

```
cities = 20  # number of cities to visit

xb = np.round(np.random.rand(cities,2)*100, 0) # generate data
chosen_example = 13
answer_path= corect_paths[chosen_example]

#Calculates distance betwen between two points
dist_matrix = distance_matrix(np.column_stack((xb[:,0], xb[:,1])), np.
    column_stack((xb[:,0], xb[:,1])))
distant_matrix = dist_matrix

Graph = nx.Graph()
for i in range(cities):
    for j in range(i + 1, cities):
        Graph.add_edge(i, j, weight=distant_matrix[i, j])

```

```
16 pos = xb
17
18 #first node is colored in different color
19 node_colors_adj = ['lightgreen' if node==0 else 'lightblue' for node in
       Graph.nodes()]
20 Graph = Graph.to_undirected()
21 #optimality gap calculation
22 (shortest_path[1]−shortes_distance)/(shortes_distance)*100
23
24 #code that creats every edges lenght
25 #put shortes path
26 path = corect_paths[chosen_example]
27 # creats a list of edges (i,j)
28 edges = [(path[i], path[i+1]) for i in range(len(path)−1)]
29
30 #add edges distance
31 edge_labels = {(path[i], path[i+1]): str(distant_matrix[path[i]][path[i
       +1]]) for i in range(len(path)−1)}
32
33 #summs all edges distance to get the shortes path distance
34 shortes_distance = sum(distant_matrix[path[i]][path[i+1]] for i in range(
       len(path)−1))
35
36 print("Total weight:", shortes_distance)
37 print("Shortes path edges and distances:", edge_labels)
38
39 #code that changes path format
40 my_nodes = []
41 for i in range(len(dist_matrix)):
42     my_nodes.append(i)
43 print(my_nodes)
44
45 init_graph = {}
46 for node in my_nodes:
47     init_graph[node] = {}
48
49 for i in range(len(dist_matrix)):
50     for j in range(len(dist_matrix)):
51         init_graph[i][j] = dist_matrix[i][j]
```

2. pirmkods. Simulācija

```
#brute force method implementations
def brute_force_tsp(graph, s):
    virsotne = []
    for i in range(V):
        if i != s:
            virsotne.append(i)

    min_path = maxsize
    next_permutation = permutations(virsotne)
    shortest_path = []
    for i in next_permutation:
        current_pathweight = 0
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k=j
        current_pathweight += graph[k][s]
        if current_pathweight < min_path:
            min_path = current_pathweight
            shortest_path = list(i)
            shortest_path.insert(0, s)
            shortest_path.append(s)
    return min_path, shortest_path

#results
start_time = time.time()
graph = distant_matrix

V = cities
s = 0

opt_path_BF, path_BR = brute_force_tsp(graph,s)

end_time = time.time()
total_time_bruce = end_time - start_time

print(f"Total time taken: {total_time_bruce} seconds")
print("shortes paths lenght:", opt_path_BF)
print("Shortes path:", path_BR)

```

```
41 #graph visualization
42 nx.draw(Graph, pos=pos, node_color=node_colors_adj, with_labels=True ,width
       =0.3)
43 nx.draw_networkx_edges(Graph, pos=pos,
44                        edgelist=[(path_BR[i], path_BR[i+1]) for i in range(
                              len(path_BR)-1)],
45                        edge_color='red',width=2)
46 plt.show()
```

3. pirmkods. Naivās metodes implementācija

```
1  big_val = 99999999
2  #dynamic programming
3  def dp_tsp(mask, p, graph, dp, n, visited, parent):
4      if mask == visited:
5          return graph[p][0], [0]
6      if dp[mask][p] != -1:
7          return dp[mask][p], parent[mask][p]
8      answer = big_val
9      shortest_path = []
10     for city in range(0, n):
11         if ((mask & (1 << city)) == 0):
12             new, path = dp_tsp(mask|(1<<city), city, graph, dp, n, visited,
                     parent)
13             new += graph[p][city]
14             if new < answer:
15                 answer = new
16                 shortest_path = path + [city]
17     dp[mask][p] = answer
18     parent[mask][p] = shortest_path
19     return dp[mask][p], parent[mask][p]
20
21 #results
22 start_time = time.time()
23 graph = distant_matrix
24 n = cities
25 visited = (1 << n) - 1
26 r, c = (1 << n), n
27 dp = [[-1 for j in range(c)] for i in range(r)]
28 parent = [[[] for j in range(c)] for i in range(r)]
29
```

```
30 for i in range(0, (1<<n)):
31     for j in range(0, n):
32         dp[i][j] = -1
33
34 dist, path = dp_tsp(1, 0 ,graph, dp, n, visited, parent)
35 print("Shortest distance:", dist)
36 print("Shortest path:", path)
37
38 end_time = time.time()
39 total_time_DP = end_time - start_time
40
41 print(f"Total time taken: {total_time_DP} seconds")
42
43 # visualize the solution using networkx
44 path = np.concatenate(([0], path, [0]))
45 nx.draw(Graph, pos=pos, node_color=node_colors_adj, with_labels=True ,width
     =0.3)
46 nx.draw_networkx_edges(Graph, pos=pos,
47                        edgelist=[(path[i], path[i+1]) for i in range(len(
                              path)-1)],
48                        edge_color='red',width=2)
49 plt.show()
```

4. pirmkods. Dinamiskās programmēšanas implementācija

```
1 #HK implementation
2 def hk_tsp(dists):
3     n = len(dists)
4     C = {}
5     for k in range(1, n):
6         C[(1 << k, k)] = (dists[0][k], 0)
7     for subset_size in range(2, n):
8         for subset in itertools.combinations(range(1, n), subset_size):
9             b = 0
10            for bit in subset:
11                b |= 1 << bit
12            for k in subset:
13                prev = b & ~(1 << k)
14
15                result = []
16                for m in subset:
```

```
17                            if m == 0 or m == k:
18                                continue
19                            result.append((C[(prev, m)][0] + dists[m][k], m))
20                        C[(b, k)] = min(result)
21      b = (2**n - 1) - 1
22      result = []
23      for k in range(1, n):
24          result.append((C[(b, k)][0] + dists[k][0], k))
25      opt, parent = min(result)
26      path = []
27      for i in range(n - 1):
28          path.append(parent)
29          new_b = b & ~(1 << parent)
30          _, parent = C[(b, parent)]
31          b = new_b
32      path.append(0)
33      return opt, list(reversed(path))
34
35  #result tracking
36  start_time = time.time()
37  opt_HK, path_HK = hk_tsp(distant_matrix)
38  print(opt_HK, path_HK)
39  end_time = time.time()
40  total_time_HK = end_time - start_time
41  print(f"Total time taken: {total_time_HK} seconds")
42
43  #visualize the solution using networkx
44  fig, axs = plt.subplots(1, 2, figsize=(30, 15))
45  axs[0].set_title('Held-Karp')
46  nx.draw(Graph, pos=pos, node_color='lightblue', with_labels=True ,width
        =0.3, ax = axs[0])
47  nx.draw_networkx_edges(Graph, pos=pos,
48                              edgelist=[(path_HK1[i], path_HK1[i+1]) for i in
                                    range(len(path_HK1)-1)],
49                              edge_color='red',width=2, ax = axs[0])
```

5. pirmkods. Held-Karp algoritma implementācija

```
1  #ACO implementations
2  class ACO(object):
3      def __init__(self, distances, k_ants, n_best, iterations, decay, alpha
```

```
        =1, beta=1):
4           self.distances = distances
5           self.feromons = np.ones(self.distances.shape) / len(distances)
6           self.all_inds = range(len(distances))
7           self.k_ants = k_ants
8           self.n_best = n_best
9           self.iterations = iterations
10          self.decay = decay
11          self.alpha = alpha
12          self.beta = beta
13
14      def main(self):
15          shortest_path = None
16          all_time_shortest_path = ("placeholder", np.inf)
17          for i in range(self.iterations):
18              all_paths = self.gen_all_paths()
19              self.spread_pheronome(all_paths, self.n_best, shortest_path=
                    shortest_path)
20              shortest_path = min(all_paths, key=lambda x: x[1])
21              print(shortest_path)
22              if shortest_path[1] < all_time_shortest_path[1]:
23                  all_time_shortest_path = shortest_path
24              self.feromons = self.feromons * self.decay
25          return all_time_shortest_path
26
27      def spread_pheronome(self, all_paths, n_best, shortest_path):
28          sorted_paths = sorted(all_paths, key=lambda x: x[1])
29          for path, dist in sorted_paths[:n_best]:
30              for move in path:
31                  self.feromons[move] += 1.0 / self.distances[move]
32
33      def gen_path_dist(self, path):
34          total_dist = 0
35          for ele in path:
36              total_dist += self.distances[ele]
37          return total_dist
38
39      def best_path(self, start):
40          path = []
41          visited = set()
```

```
42          visited.add(start)
43          prev = start
44          for i in range(len(self.distances) - 1):
45              move = self.pick_move(self.feromons[prev], self.distances[prev
                    ], visited)
46              path.append((prev, move))
47              prev = move
48              visited.add(move)
49          path.append((prev, start))
50          return path
51
52      def pick_move(self, feromons, dist, visited):
53          feromons = np.copy(feromons)
54          feromons[list(visited)] = 0
55
56          row = feromons ** self.alpha * ((1.0 / dist) ** self.beta)
57
58          norm_row = row / row.sum()
59          move = np_choice(self.all_inds, 1, p=norm_row)[0]
60          return move
61
62  # getting results for ACO
63
64  xb1 = xb*100
65  dist_matrix = distance_matrix(xb1, xb1)
66  array = np.array([[round(num, 2) for num in sublist] for sublist in dist_
        matrix])
67  array[array == 0] = np.inf
68
69  distances = array
70  ant_colony = ACO(distances, 30, 1, 100, 0.95, alpha=0.5, beta=5)
71  shortest_path = ant_colony.run()
72  print("shorted_path: {}".format(shortest_path[0]))
73  print("shorted_paths_distance: {}".format(shortest_path[1]))
74  points = xb1
75
76  fig, axs= plt.subplots(1, 2, figsize=(12, 5))
77
78  nx.draw(H, pos=pos, with_labels=True, node_color = node_colors_adj, ax=axs
        [1],width=0.2)
```

```
79 nx.draw_networkx_edges(H, pos=pos,
80                         edgelist=shortest_path[0],
81                         edge_color='red',width=2, ax=axs[1])
82
83 #visualize the solution using networkx
84 Graph.add_edges_from(edges)
85 nx.draw(Graph, pos=pos, node_color='lightblue', with_labels=True ,width
      =0.3, ax = axs[1])
86 nx.draw_networkx_edges(Graph, pos=pos,
87                         edgelist=[(path_AOC[i], path_AOC[i+1]) for i in
                              range(len(path_AOC)-1)],
88                         edge_color='red',width=2, ax = axs[1])
89 nx.draw_networkx_edge_labels(Graph, pos, edge_labels=edge_labels)
```

6. pirmkods. Skudru kolonijas optimizācijas implementācija

```
1 import torch
2 import torch.nn.functional as F
3 import torch.nn as nn
4
5 import numpy as np
6
7
8 class BatchNormNode(nn.Module):
9     def __init__(self, hidden_dim):
10         super(BatchNormNode, self).__init__()
11         self.batch_norm = nn.BatchNorm1d(hidden_dim, track_running_stats=
              False)
12
13     def forward(self, x):
14         x_trans = x.transpose(1, 2).contiguous()  # Reshape input: (batch_
              size, hidden_dim, num_nodes)
15         x_trans_bn = self.batch_norm(x_trans)
16         x_bn = x_trans_bn.transpose(1, 2).contiguous()  # Reshape to
              original shape
17         return x_bn
18
19
20 class BatchNormEdge(nn.Module):
21     def __init__(self, hidden_dim):
22         super(BatchNormEdge, self).__init__()
```

```
23         self.batch_norm = nn.BatchNorm2d(hidden_dim, track_running_stats=
               False)
24 
25     def forward(self, e):
26         e_trans = e.transpose(1, 3).contiguous()  # Reshape input: (batch_
               size, num_nodes, num_nodes, hidden_dim)
27         e_trans_bn = self.batch_norm(e_trans)
28         e_bn = e_trans_bn.transpose(1, 3).contiguous()  # Reshape to
               original
29         return e_bn
30 
31 
32 class NodeFeatures(nn.Module):
33     def __init__(self, hidden_dim, aggregation="mean"):
34         super(NodeFeatures, self).__init__()
35         self.aggregation = aggregation
36         self.U = nn.Linear(hidden_dim, hidden_dim, True)
37         self.V = nn.Linear(hidden_dim, hidden_dim, True)
38 
39     def forward(self, x, edge_gate):
40         Ux = self.U(x)  # B x V x H
41         Vx = self.V(x)  # B x V x H
42         Vx = Vx.unsqueeze(1)  # extend Vx from "B x V x H" to "B x 1 x V x
               H"
43         gateVx = edge_gate * Vx  # B x V x V x H
44         if self.aggregation=="mean":
45             x_new = Ux + torch.sum(gateVx, dim=2) / (1e-20 + torch.sum(edge
                   _gate, dim=2))  # B x V x H
46         elif self.aggregation=="sum":
47             x_new = Ux + torch.sum(gateVx, dim=2)  # B x V x H
48         return x_new
49 
50 
51 class EdgeFeatures(nn.Module):
52     def __init__(self, hidden_dim):
53         super(EdgeFeatures, self).__init__()
54         self.U = nn.Linear(hidden_dim, hidden_dim, True)
55         self.V = nn.Linear(hidden_dim, hidden_dim, True)
56 
57     def forward(self, x, e):
```

```
58            Ue = self.U(e)
59            Vx = self.V(x)
60            Wx = Vx.unsqueeze(1)  # Extend Vx from "B x V x H" to "B x V x 1 x
                  H"
61            Vx = Vx.unsqueeze(2)  # extend Vx from "B x V x H" to "B x 1 x V x
                  H"
62            e_new = Ue + Vx + Wx
63            return e_new
64
65
66  class ResidualGatedGCNLayer(nn.Module):
67      def __init__(self, hidden_dim, aggregation="sum"):
68          super(ResidualGatedGCNLayer, self).__init__()
69          self.node_feat = NodeFeatures(hidden_dim, aggregation)
70          self.edge_feat = EdgeFeatures(hidden_dim)
71          self.bn_node = BatchNormNode(hidden_dim)
72          self.bn_edge = BatchNormEdge(hidden_dim)
73
74      def forward(self, x, e):
75          e_in = e
76          x_in = x
77          # Edge convolution
78          e_tmp = self.edge_feat(x_in, e_in)  # B x V x V x H
79          # Compute edge gates
80          edge_gate = F.sigmoid(e_tmp)
81          # Node convolution
82          x_tmp = self.node_feat(x_in, edge_gate)
83          # Batch normalization
84          e_tmp = self.bn_edge(e_tmp)
85          x_tmp = self.bn_node(x_tmp)
86          # ReLU Activation
87          e = F.relu(e_tmp)
88          x = F.relu(x_tmp)
89          # Residual connection
90          x_new = x_in + x
91          e_new = e_in + e
92          return x_new, e_new
93
94
95  class MLP(nn.Module):
```

```
    def __init__(self, hidden_dim, output_dim, L=2):
        super(MLP, self).__init__()
        self.L = L
        U = []
        for layer in range(self.L - 1):
            U.append(nn.Linear(hidden_dim, hidden_dim, True))
        self.U = nn.ModuleList(U)
        self.V = nn.Linear(hidden_dim, output_dim, True)

    def forward(self, x):
        Ux = x
        for U_i in self.U:
            Ux = U_i(Ux)  # B x H
            Ux = F.relu(Ux)  # B x H
        y = self.V(Ux)  # B x O
        return y
```

7. pirmkods. GCN implementācija: GCN slāņi

```
import torch
import torch.nn.functional as F
import torch.nn as nn

from models.gcn_layers import ResidualGatedGCNLayer, MLP
from utils.model_utils import *

class ResidualGatedGCNModel(nn.Module):
    def __init__(self, config, dtypeFloat, dtypeLong):
        super(ResidualGatedGCNModel, self).__init__()
        self.dtypeFloat = dtypeFloat
        self.dtypeLong = dtypeLong
        self.num_nodes = config.num_nodes
        self.node_dim = config.node_dim
        self.voc_nodes_in = config['voc_nodes_in']
        self.voc_nodes_out = config['num_nodes']  # config['voc_nodes_out']
        self.voc_edges_in = config['voc_edges_in']
        self.voc_edges_out = config['voc_edges_out']
        self.hidden_dim = config['hidden_dim']
        self.num_layers = config['num_layers']
        self.mlp_layers = config['mlp_layers']
        self.aggregation = config['aggregation']
```

```
23          self.nodes_coord_embedding = nn.Linear(self.node_dim, self.hidden_
                dim, bias=False)
24          self.edges_values_embedding = nn.Linear(1, self.hidden_dim//2, bias
                =False)
25          self.edges_embedding = nn.Embedding(self.voc_edges_in, self.hidden_
                dim//2)
26          gcn_layers = []
27          for layer in range(self.num_layers):
28              gcn_layers.append(ResidualGatedGCNLayer(self.hidden_dim, self.
                    aggregation))
29          self.gcn_layers = nn.ModuleList(gcn_layers)
30          self.mlp_edges = MLP(self.hidden_dim, self.voc_edges_out, self.mlp_
                layers)
31
32      def forward(self, x_edges, x_edges_values, x_nodes, x_nodes_coord, y_
            edges, edge_cw):
33          # Node and edge embedding
34          x = self.nodes_coord_embedding(x_nodes_coord)
35          e_vals = self.edges_values_embedding(x_edges_values.unsqueeze(3))
36          e_tags = self.edges_embedding(x_edges)
37          e = torch.cat((e_vals, e_tags), dim=3)
38          # GCN layers
39          for layer in range(self.num_layers):
40              x, e = self.gcn_layers[layer](x, e)
41          y_pred_edges = self.mlp_edges(e)
42          edge_cw = torch.Tensor(edge_cw).type(self.dtypeFloat)
43          loss = loss_edges(y_pred_edges, y_edges, edge_cw)
44
45          return y_pred_edges,
```

8. pirmkods. GCN implementācija: GCN modelis

```
1 import numpy as np
2 import torch
3
4 class Beamsearch(object):
5
6     def __init__(self, beam_size, batch_size, num_nodes,
7                  dtypeFloat=torch.FloatTensor, dtypeLong=torch.LongTensor,
8                  probs_type='raw', random_start=False):
9
```

```
10         self.batch_size = batch_size
11         self.beam_size = beam_size
12         self.num_nodes = num_nodes
13         self.probs_type = probs_type
14         self.dtypeFloat = dtypeFloat
15         self.dtypeLong = dtypeLong
16         self.start_nodes = torch.zeros(batch_size, beam_size).type(self.
               dtypeLong)
17         if random_start == True:
18             self.start_nodes = torch.randint(0, num_nodes, (batch_size,
                   beam_size)).type(self.dtypeLong)
19         self.mask = torch.ones(batch_size, beam_size, num_nodes).type(self.
               dtypeFloat)
20         self.update_mask(self.start_nodes)
21         self.scores = torch.zeros(batch_size, beam_size).type(self.
               dtypeFloat)
22         self.all_scores = []
23         self.prev_Ks = []
24         self.next_nodes = [self.start_nodes]
25 
26     def get_current_state(self):
27 
28         current_state = (self.next_nodes[-1].unsqueeze(2)
29                          .expand(self.batch_size, self.beam_size, self.num_
                             nodes))
30         return current_state
31 
32     def get_current_origin(self):
33 
34         return self.prev_Ks[-1]
35 
36     def advance(self, trans_probs):
37 
38         if len(self.prev_Ks) > 0:
39             if self.probs_type == 'raw':
40                 beam_lk = trans_probs * self.scores.unsqueeze(2).expand_as(
                       trans_probs)
41             elif self.probs_type == 'logits':
42                 beam_lk = trans_probs + self.scores.unsqueeze(2).expand_as(
                       trans_probs)
```

```
43            else:
44                beam_lk = trans_probs
45                if self.probs_type == 'raw':
46                    beam_lk[:, 1:] = torch.zeros(beam_lk[:, 1:].size()).type(
                          self.dtypeFloat)
47                elif self.probs_type == 'logits':
48                    beam_lk[:, 1:] = -1e20 * torch.ones(beam_lk[:, 1:].size()).
                          type(self.dtypeFloat)
49            # Multiply by mask
50            beam_lk = beam_lk * self.mask
51            beam_lk = beam_lk.view(self.batch_size, -1)  # (batch_size, beam_
                  size * num_nodes)
52            bestScores, bestScoresId = beam_lk.topk(self.beam_size, 1, True,
                  True)
53            self.scores = bestScores
54            prev_k = bestScoresId / self.num_nodes
55            self.prev_Ks.append(prev_k)
56            new_nodes = bestScoresId - prev_k * self.num_nodes
57            self.next_nodes.append(new_nodes)
58            perm_mask = prev_k.unsqueeze(2).expand_as(self.mask)  # (batch_size
                  , beam_size, num_nodes)
59            self.mask = self.mask.gather(1, perm_mask)
60            self.update_mask(new_nodes)
61
62        def update_mask(self, new_nodes):
63            arr = (torch.arange(0, self.num_nodes).unsqueeze(0).unsqueeze(1)
64                   .expand_as(self.mask).type(self.dtypeLong))
65            new_nodes = new_nodes.unsqueeze(2).expand_as(self.mask)
66            update_mask = 1 - torch.eq(arr, new_nodes).type(self.dtypeFloat)
67            self.mask = self.mask * update_mask
68            if self.probs_type == 'logits':
69                # Convert 0s in mask to inf
70                self.mask[self.mask == 0] = 1e20
71
72        def sort_best(self):
73            return torch.sort(self.scores, 0, True)
74
75        def get_best(self):
76            scores, ids = self.sort_best()
77            return scores[1], ids[1]
```

```
78
79      def get_hypothesis(self, k):
80          assert self.num_nodes == len(self.prev_Ks) + 1
81
82          hyp = -1 * torch.ones(self.batch_size, self.num_nodes).type(self.
                dtypeLong)
83          for j in range(len(self.prev_Ks) - 1, -2, -1):
84              hyp[:, j + 1] = self.next_nodes[j + 1].gather(1, k).view(1,
                    self.batch_size)
85              k = self.prev_Ks[j].gather(1, k)
86          return hyp
```

9. pirmkods. GCN implementācija: staru meklēšana

```
 1  import torch
 2  import torch.nn.functional as F
 3  import numpy as np
 4
 5  def tour_nodes_to_W(nodes):
 6      W = np.zeros((len(nodes), len(nodes)))
 7      for idx in range(len(nodes) - 1):
 8          i = int(nodes[idx])
 9          j = int(nodes[idx + 1])
10          W[i][j] = 1
11          W[j][i] = 1
12      W[j][int(nodes[0])] = 1
13      W[int(nodes[0])][j] = 1
14      return W
15
16  def tour_nodes_to_tour_len(nodes, W_values):
17
18      tour_len = 0
19      for idx in range(len(nodes) - 1):
20          i = nodes[idx]
21          j = nodes[idx + 1]
22          tour_len += W_values[i][j]
23      tour_len += W_values[j][nodes[0]]
24      return tour_len
25
26  def W_to_tour_len(W, W_values):
27      tour_len = 0
```

```
28      for i in range(W.shape[0]):
29          for j in range(W.shape[1]):
30              if W[i][j] == 1:
31                  tour_len += W_values[i][j]
32      tour_len /= 2  # Divide by 2 because adjacency matrices are symmetric
33      return tour_len
34 
35 
36  def is_valid_tour(nodes, num_nodes):
37      return sorted(nodes) == [i for i in range(num_nodes)]
38 
39 
40  def mean_tour_len_edges(x_edges_values, y_pred_edges):
41 
42      y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
43      y = y.argmax(dim=3)  # B x V x V
44      # Divide by 2 because edges_values is symmetric
45      tour_lens = (y.float() * x_edges_values.float()).sum(dim=1).sum(dim=1)
        / 2
46      mean_tour_len = tour_lens.sum().to(dtype=torch.float).item() / tour_
        lens.numel()
47      return mean_tour_len
48 
49 
50  def mean_tour_len_nodes(x_edges_values, bs_nodes):
51 
52      y = bs_nodes.cpu().numpy()
53      W_val = x_edges_values.cpu().numpy()
54      running_tour_len = 0
55      for batch_idx in range(y.shape[0]):
56          for y_idx in range(y[batch_idx].shape[0] - 1):
57              i = y[batch_idx][y_idx]
58              j = y[batch_idx][y_idx + 1]
59              running_tour_len += W_val[batch_idx][i][j]
60          running_tour_len += W_val[batch_idx][j][0]  # Add final connection
            to tour/cycle
61      return running_tour_len / y.shape[0]
62 
63 
64  def get_max_k(dataset, max_iter=1000):
```

```
65
66      ks = []
67      for _ in range(max_iter):
68          batch = next(iter(dataset))
69          for idx in range(batch.edges.shape[0]):
70              for row in range(dataset.num_nodes):
71                  connections = np.where(batch.edges_target[idx][row]==1)[0]
72
73                  sorted_neighbors = np.argsort(batch.edges_values[idx][row],
                        axis=-1)
74                  for conn_idx in connections:
75                      ks.append(np.where(sorted_neighbors==conn_idx)[0][0])
76      return int(np.max(ks))
77
78  import torch
79  import torch.nn.functional as F
80  import torch.nn as nn
81
82  from utils.beamsearch import *
83  from utils.graph_utils import *
84
85
86  def loss_nodes(y_pred_nodes, y_nodes, node_cw):
87      y = F.log_softmax(y_pred_nodes, dim=2)  # B x V x voc_nodes_out
88      y = y.permute(0, 2, 1)  # B x voc_nodes x V
89      loss_nodes = nn.NLLLoss(node_cw)(y, y_nodes)
90      return loss_nodes
91
92
93  def loss_edges(y_pred_edges, y_edges, edge_cw):
94
95      # Edge loss
96      y = F.log_softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
97      y = y.permute(0, 3, 1, 2)  # B x voc_edges x V x V
98      loss_edges = nn.NLLLoss(edge_cw)(y, y_edges)
99      return loss_edges
100
101
102 def beamsearch_tour_nodes(y_pred_edges, beam_size, batch_size, num_nodes,
        dtypeFloat, dtypeLong, probs_type='raw', random_start=False):
```

```
103
104     if probs_type == 'raw':
105         y = F.softmax(y_pred_edges, dim=3)
106         y = y[:, :, :, 1]
107     elif probs_type == 'logits':
108         y = F.log_softmax(y_pred_edges, dim=3)
109         y = y[:, :, :, 1]
110         y[y == 0] = -1e-20
111     beamsearch = Beamsearch(beam_size, batch_size, num_nodes, dtypeFloat,
            dtypeLong, probs_type, random_start)
112     trans_probs = y.gather(1, beamsearch.get_current_state())
113     for step in range(num_nodes - 1):
114         beamsearch.advance(trans_probs)
115         trans_probs = y.gather(1, beamsearch.get_current_state())
116     ends = torch.zeros(batch_size, 1).type(dtypeLong)
117     return beamsearch.get_hypothesis(ends)
118
119
120 def beamsearch_tour_nodes_shortest(y_pred_edges, x_edges_values, beam_size,
        batch_size, num_nodes,
121                                    dtypeFloat, dtypeLong, probs_type='raw',
                                        random_start=False):
122
123     if probs_type == 'raw':
124         y = F.softmax(y_pred_edges, dim=3)
125         y = y[:, :, :, 1]  # B x V x V
126     elif probs_type == 'logits':
127         y = F.log_softmax(y_pred_edges, dim=3)
128         y = y[:, :, :, 1]
129         y[y == 0] = -1e-20
130     beamsearch = Beamsearch(beam_size, batch_size, num_nodes, dtypeFloat,
            dtypeLong, probs_type, random_start)
131     trans_probs = y.gather(1, beamsearch.get_current_state())
132     for step in range(num_nodes - 1):
133         beamsearch.advance(trans_probs)
134         trans_probs = y.gather(1, beamsearch.get_current_state())
135     ends = torch.zeros(batch_size, 1).type(dtypeLong)
136     shortest_tours = beamsearch.get_hypothesis(ends)
137     shortest_lens = [1e6] * len(shortest_tours)
138     for idx in range(len(shortest_tours)):
```

```
139            shortest_lens[idx] = tour_nodes_to_tour_len(shortest_tours[idx].cpu
                   ().numpy(),
140                                                     x_edges_values[idx].cpu
                                                          ().numpy())
141        for pos in range(1, beam_size):
142            ends = pos * torch.ones(batch_size, 1).type(dtypeLong)  # New
                   positions
143            hyp_tours = beamsearch.get_hypothesis(ends)
144            for idx in range(len(hyp_tours)):
145                hyp_nodes = hyp_tours[idx].cpu().numpy()
146                hyp_len = tour_nodes_to_tour_len(hyp_nodes, x_edges_values[idx
                       ].cpu().numpy())
147                if hyp_len < shortest_lens[idx] and is_valid_tour(hyp_nodes,
                       num_nodes):
148                    shortest_tours[idx] = hyp_tours[idx]
149                    shortest_lens[idx] = hyp_len
150        return shortest_tours
151
152
153    def update_learning_rate(optimizer, lr):
154
155        for param_group in optimizer.param_groups:
156            param_group['lr'] = lr
157        return optimizer
158
159
160    def edge_error(y_pred, y_target, x_edges):
161
162        y = F.softmax(y_pred, dim=3)  # B x V x V x voc_edges
163        y = y.argmax(dim=3)  # B x V x V
164
165        # Edge error: Mask out edges which are not connected
166        mask_no_edges = x_edges.long()
167        err_edges, _ = _edge_error(y, y_target, mask_no_edges)
168
169
170        mask_no_tour = y_target
171        err_tour, err_idx_tour = _edge_error(y, y_target, mask_no_tour)
172
173        mask_no_tsp = ((y_target + y) > 0).long()
```

```
    err_tsp, err_idx_tsp = _edge_error(y, y_target, mask_no_tsp)

    return 100 * err_edges, 100 * err_tour, 100 * err_tsp, err_idx_tour, err_idx_tsp

def _edge_error(y, y_target, mask):
    acc = (y == y_target).long()
    acc = (acc * mask)
    acc = acc.sum(dim=1).sum(dim=1).to(dtype=torch.float) / mask.sum(dim=1).sum(dim=1).to(dtype=torch.float)
    err_idx = (acc < 1.0)
    acc = acc.sum().to(dtype=torch.float).item() / acc.numel()
    err = 1.0 - acc
    return err, err_idx


import time
import numpy as np
from scipy.spatial.distance import pdist, squareform
from sklearn.utils import shuffle


class DotDict(dict):
    def __init__(self, **kwds):
        self.update(kwds)
        self.__dict__ = self

class GoogleTSPReader(object):
    def __init__(self, num_nodes, num_neighbors, batch_size, filepath):

        self.num_nodes = num_nodes
        self.num_neighbors = num_neighbors
        self.batch_size = batch_size
        self.filepath = filepath
        self.filedata = shuffle(open(filepath, "r").readlines())  # Always shuffle upon reading data
        self.max_iter = (len(self.filedata) // batch_size)

    def __iter__(self):
        for batch in range(self.max_iter):
```

```
211             start_idx = batch * self.batch_size
212             end_idx = (batch + 1) * self.batch_size
213             yield self.process_batch(self.filedata[start_idx:end_idx])
214 
215     def process_batch(self, lines):
216 
217         batch_edges = []
218         batch_edges_values = []
219         batch_edges_target = []
220         batch_nodes = []
221         batch_nodes_target = []
222         batch_nodes_coord = []
223         batch_tour_nodes = []
224         batch_tour_len = []
225 
226         for line_num, line in enumerate(lines):
227             line = line.split(" ")  # Split into list
228 
229             nodes = np.ones(self.num_nodes)  # All 1s for TSP...
230 
231             nodes_coord = []
232             for idx in range(0, 2 * self.num_nodes, 2):
233                 nodes_coord.append([float(line[idx]), float(line[idx + 1])
                        ])
234 
235             W_val = squareform(pdist(nodes_coord, metric='euclidean'))
236 
237             if self.num_neighbors == -1:
238                 W = np.ones((self.num_nodes, self.num_nodes))  # Graph is
                        fully connected
239             else:
240                 W = np.zeros((self.num_nodes, self.num_nodes))
241                 knns = np.argpartition(W_val, kth=self.num_neighbors, axis
                        =-1)[:, self.num_neighbors::-1]
242                 for idx in range(self.num_nodes):
243                     W[idx][knns[idx]] = 1
244             np.fill_diagonal(W, 2)  # Special token for self-connections
245 
246             tour_nodes = [int(node) - 1 for node in line[line.index('output
                    ') + 1:-1]][:-1]
```

```
247
248                 tour_len = 0
249                 nodes_target = np.zeros(self.num_nodes)
250                 edges_target = np.zeros((self.num_nodes, self.num_nodes))
251                 for idx in range(len(tour_nodes) - 1):
252                     i = tour_nodes[idx]
253                     j = tour_nodes[idx + 1]
254                     nodes_target[i] = idx
255                     edges_target[i][j] = 1
256                     edges_target[j][i] = 1
257                     tour_len += W_val[i][j]
258
259                 nodes_target[j] = len(tour_nodes) - 1
260                 edges_target[j][tour_nodes[0]] = 1
261                 edges_target[tour_nodes[0]][j] = 1
262                 tour_len += W_val[j][tour_nodes[0]]
263
264                 batch_edges.append(W)
265                 batch_edges_values.append(W_val)
266                 batch_edges_target.append(edges_target)
267                 batch_nodes.append(nodes)
268                 batch_nodes_target.append(nodes_target)
269                 batch_nodes_coord.append(nodes_coord)
270                 batch_tour_nodes.append(tour_nodes)
271                 batch_tour_len.append(tour_len)
272
273             batch = DotDict()
274             batch.edges = np.stack(batch_edges, axis=0)
275             batch.edges_values = np.stack(batch_edges_values, axis=0)
276             batch.edges_target = np.stack(batch_edges_target, axis=0)
277             batch.nodes = np.stack(batch_nodes, axis=0)
278             batch.nodes_target = np.stack(batch_nodes_target, axis=0)
279             batch.nodes_coord = np.stack(batch_nodes_coord, axis=0)
280             batch.tour_nodes = np.stack(batch_tour_nodes, axis=0)
281             batch.tour_len = np.stack(batch_tour_len, axis=0)
282             return batch
283
284
285 import torch
286 import torch.nn.functional as F
```

```
287
288 import matplotlib
289 import matplotlib.pyplot as plt
290 import networkx as nx
291
292 from utils.graph_utils import *
293
294 def plot_tsp(p, x_coord, W, W_val, W_target, title="default"):
295     def _edges_to_node_pairs(W):
296         pairs = []
297         for r in range(len(W)):
298             for c in range(len(W)):
299                 if W[r][c] == 1:
300                     pairs.append((r, c))
301         return pairs
302
303     G = nx.from_numpy_matrix(W_val)
304     pos = dict(zip(range(len(x_coord)), x_coord.tolist()))
305     adj_pairs = _edges_to_node_pairs(W)
306     target_pairs = _edges_to_node_pairs(W_target)
307     colors = ['g'] + ['b'] * (len(x_coord) - 1)  # Green for 0th node, blue
          for others
308     nx.draw_networkx_nodes(G, pos, node_color=colors, node_size=50)
309     nx.draw_networkx_edges(G, pos, edgelist=adj_pairs, alpha=0.3, width
        =0.5)
310     nx.draw_networkx_edges(G, pos, edgelist=target_pairs, alpha=1, width=1,
          edge_color='r')
311     p.set_title(title)
312     return p
313
314
315 def plot_tsp_heatmap(p, x_coord, W_val, W_pred, title="default"):
316
317     def _edges_to_node_pairs(W):
318         pairs = []
319         edge_preds = []
320         for r in range(len(W)):
321             for c in range(len(W)):
322                 if W[r][c] > 0.25:
323                     pairs.append((r, c))
```

```
324                         edge_preds.append(W[r][c])
325             return pairs, edge_preds
326 
327         G = nx.from_numpy_matrix(W_val)
328         pos = dict(zip(range(len(x_coord)), x_coord.tolist()))
329         node_pairs, edge_color = _edges_to_node_pairs(W_pred)
330         node_color = ['g'] + ['b'] * (len(x_coord) - 1)  # Green for 0th node, blue for others
331         nx.draw_networkx_nodes(G, pos, node_color=node_color, node_size=50)
332         nx.draw_networkx_edges(G, pos, edgelist=node_pairs, edge_color=edge_color, edge_cmap=plt.cm.Reds, width=0.75)
333         p.set_title(title)
334         return p
335 
336 
337 def plot_predictions(x_nodes_coord, x_edges, x_edges_values, y_edges, y_pred_edges, num_plots=3):
338     y = F.softmax(y_pred_edges, dim=3)
339     y_bins = y.argmax(dim=3)
340     y_probs = y[:,:,:,1]
341     for f_idx, idx in enumerate(np.random.choice(len(y), num_plots, replace=False)):
342         f = plt.figure(f_idx, figsize=(10, 5))
343         x_coord = x_nodes_coord[idx].cpu().numpy()
344         W = x_edges[idx].cpu().numpy()
345         W_val = x_edges_values[idx].cpu().numpy()
346         W_target = y_edges[idx].cpu().numpy()
347         W_sol_bins = y_bins[idx].cpu().numpy()
348         W_sol_probs = y_probs[idx].cpu().numpy()
349         plt1 = f.add_subplot(121)
350         plot_tsp(plt1, x_coord, W, W_val, W_target, 'Groundtruth: {:.3f}'.format(W_to_tour_len(W_target, W_val)))
351         plt2 = f.add_subplot(122)
352         plot_tsp_heatmap(plt2, x_coord, W_val, W_sol_probs, 'Prediction Heatmap')
353         plt.show()
354 
355 
356 def plot_predictions_beamsearch(x_nodes_coord, x_edges, x_edges_values, y_edges, y_pred_edges, bs_nodes, num_plots=3):
```

```
357
358     y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
359     y_bins = y.argmax(dim=3)
360     y_probs = y[:,:,:,1]
361     for f_idx, idx in enumerate(np.random.choice(len(y), num_plots, replace
            =False)):
362         f = plt.figure(f_idx, figsize=(15, 5))
363         x_coord = x_nodes_coord[idx].cpu().numpy()
364         W = x_edges[idx].cpu().numpy()
365         W_val = x_edges_values[idx].cpu().numpy()
366         W_target = y_edges[idx].cpu().numpy()
367         W_sol_bins = y_bins[idx].cpu().numpy()
368         W_sol_probs = y_probs[idx].cpu().numpy()
369         W_bs = tour_nodes_to_W(bs_nodes[idx].cpu().numpy())
370         plt1 = f.add_subplot(131)
371         plot_tsp(plt1, x_coord, W, W_val, W_target, 'Groundtruth: {:.3f}'.
                format(W_to_tour_len(W_target, W_val)))
372         plt2 = f.add_subplot(132)
373         plot_tsp_heatmap(plt2, x_coord, W_val, W_sol_probs, 'Prediction
                Heatmap')
374         plt3 = f.add_subplot(133)
375         plot_tsp(plt3, x_coord, W, W_val, W_bs, 'Beamsearch: {:.3f}'.format
                (W_to_tour_len(W_bs, W_val)))
376         plt.show()
377
378 #model training
379 def train_one_epoch(net, optimizer, config, master_bar):
380     net.train()
381
382     num_nodes = config.num_nodes
383     num_neighbors = config.num_neighbors
384     batch_size = config.batch_size
385     batches_per_epoch = config.batches_per_epoch
386     accumulation_steps = config.accumulation_steps
387     train_filepath = config.train_filepath
388
389     dataset = GoogleTSPReader(num_nodes, num_neighbors, batch_size, train_
            filepath)
390     if batches_per_epoch != -1:
391         batches_per_epoch = min(batches_per_epoch, dataset.max_iter)
```

```
392     else:
393         batches_per_epoch = dataset.max_iter
394 
395     dataset = iter(dataset)
396 
397     edge_cw = None
398 
399     running_loss = 0.0
400     running_pred_tour_len = 0.0
401     running_gt_tour_len = 0.0
402     running_nb_data = 0
403     running_nb_batch = 0
404 
405     start_epoch = time.time()
406     for batch_num in progress_bar(range(batches_per_epoch), parent=master_
            bar):
407         try:
408             batch = next(dataset)
409         except StopIteration:
410             break
411 
412         x_edges = Variable(torch.LongTensor(batch.edges).type(dtypeLong),
                requires_grad=False)
413         x_edges_values = Variable(torch.FloatTensor(batch.edges_values).
                type(dtypeFloat), requires_grad=False)
414         x_nodes = Variable(torch.LongTensor(batch.nodes).type(dtypeLong),
                requires_grad=False)
415         x_nodes_coord = Variable(torch.FloatTensor(batch.nodes_coord).type(
                dtypeFloat), requires_grad=False)
416         y_edges = Variable(torch.LongTensor(batch.edges_target).type(
                dtypeLong), requires_grad=False)
417         y_nodes = Variable(torch.LongTensor(batch.nodes_target).type(
                dtypeLong), requires_grad=False)
418 
419         if type(edge_cw) != torch.Tensor:
420             edge_labels = y_edges.cpu().numpy().flatten()
421             edge_cw = compute_class_weight("balanced", classes=np.unique(
                    edge_labels), y=edge_labels)
422 
423         y_preds, loss = net.forward(x_edges, x_edges_values, x_nodes, x_
```

```
                nodes_coord, y_edges, edge_cw)
424             loss = loss.mean()  # Take mean of loss across multiple GPUs
425             loss = loss / accumulation_steps  # Scale loss by accumulation
                    steps
426             loss.backward()
427 
428 
429             if (batch_num+1) % accumulation_steps == 0:
430                 optimizer.step()
431                 optimizer.zero_grad()
432 
433 
434             pred_tour_len = mean_tour_len_edges(x_edges_values, y_preds)
435             gt_tour_len = np.mean(batch.tour_len)
436 
437 
438             running_nb_data += batch_size
439             running_loss += batch_size* loss.data.item()* accumulation_steps  #
                    Re-scale loss
440             # running_err_edges += batch_size* err_edges
441             # running_err_tour += batch_size* err_tour
442             # running_err_tsp += batch_size* err_tsp
443             running_pred_tour_len += batch_size* pred_tour_len
444             running_gt_tour_len += batch_size* gt_tour_len
445             running_nb_batch += 1
446 
447             result = ('loss:{loss:.4f} pred_tour_len:{pred_tour_len:.3f} gt_
                    tour_len:{gt_tour_len:.3f}'.format(
448                 loss=running_loss/running_nb_data,
449                 pred_tour_len=running_pred_tour_len/running_nb_data,
450                 gt_tour_len=running_gt_tour_len/running_nb_data))
451             master_bar.child.comment = result
452 
453         loss = running_loss/ running_nb_data
454         err_edges = 0 # running_err_edges/ running_nb_data
455         err_tour = 0 # running_err_tour/ running_nb_data
456         err_tsp = 0 # running_err_tsp/ running_nb_data
457         pred_tour_len = running_pred_tour_len/ running_nb_data
458         gt_tour_len = running_gt_tour_len/ running_nb_data
459 
```

```
460     return time.time()-start_epoch, loss, err_edges, err_tour, err_tsp,
            pred_tour_len, gt_tour_len
461
462
463 def metrics_to_str(epoch, time, learning_rate, loss, err_edges, err_tour,
        err_tsp, pred_tour_len, gt_tour_len):
464     result = ( 'epoch:{epoch:0>2d}\t'
465                'time:{time:.1f}h\t'
466                'lr:{learning_rate:.2e}\t'
467                'loss:{loss:.4f}\t'
468                'pred_tour_len:{pred_tour_len:.3f}\t'
469                'gt_tour_len:{gt_tour_len:.3f}'.format(
470                    epoch=epoch,
471                    time=time/3600,
472                    learning_rate=learning_rate,
473                    loss=loss,
474                    pred_tour_len=pred_tour_len,
475                    gt_tour_len=gt_tour_len))
476     return result
477
478
479
480 #validation
481 def test(net, config, master_bar, mode='test'):
482     net.eval()
483
484     num_nodes = config.num_nodes
485     num_neighbors = config.num_neighbors
486     batch_size = config.batch_size
487     batches_per_epoch = config.batches_per_epoch
488     beam_size = config.beam_size
489     val_filepath = config.val_filepath
490     test_filepath = config.test_filepath
491
492     if mode == 'val':
493         dataset = GoogleTSPReader(num_nodes, num_neighbors, batch_size=
                batch_size, filepath=val_filepath)
494     elif mode == 'test':
495         dataset = GoogleTSPReader(num_nodes, num_neighbors, batch_size=
                batch_size, filepath=test_filepath)
```

```
496     batches_per_epoch = dataset.max_iter
497
498     dataset = iter(dataset)
499
500     edge_cw = None
501
502     running_loss = 0.0
503     running_pred_tour_len = 0.0
504     running_gt_tour_len = 0.0
505     running_nb_data = 0
506     running_nb_batch = 0
507
508     with torch.no_grad():
509         start_test = time.time()
510         for batch_num in progress_bar(range(batches_per_epoch), parent=
                master_bar):
511             try:
512                 batch = next(dataset)
513             except StopIteration:
514                 break
515
516             x_edges = Variable(torch.LongTensor(batch.edges).type(dtypeLong
                    ), requires_grad=False)
517             x_edges_values = Variable(torch.FloatTensor(batch.edges_values)
                    .type(dtypeFloat), requires_grad=False)
518             x_nodes = Variable(torch.LongTensor(batch.nodes).type(dtypeLong
                    ), requires_grad=False)
519             x_nodes_coord = Variable(torch.FloatTensor(batch.nodes_coord).
                    type(dtypeFloat), requires_grad=False)
520             y_edges = Variable(torch.LongTensor(batch.edges_target).type(
                    dtypeLong), requires_grad=False)
521             y_nodes = Variable(torch.LongTensor(batch.nodes_target).type(
                    dtypeLong), requires_grad=False)
522
523             if type(edge_cw) != torch.Tensor:
524                 edge_labels = y_edges.cpu().numpy().flatten()
525                 edge_cw = compute_class_weight("balanced", classes=np.
                        unique(edge_labels), y=edge_labels)
526
527             y_preds, loss = net.forward(x_edges, x_edges_values, x_nodes, x
```

```
                    _nodes_coord, y_edges, edge_cw)
528             loss = loss.mean()  # Take mean of loss across multiple GPUs
529
530             if mode == 'val':  # Validation: faster 'vanilla' beamsearch
531                 bs_nodes = beamsearch_tour_nodes(
532                     y_preds, beam_size, batch_size, num_nodes, dtypeFloat,
                        dtypeLong, probs_type='logits')
533             elif mode == 'test':  # Testing: beamsearch with shortest tour
                    heuristic
534                 bs_nodes = beamsearch_tour_nodes_shortest(
535                     y_preds, x_edges_values, beam_size, batch_size, num_
                        nodes, dtypeFloat, dtypeLong, probs_type='logits')
536
537             pred_tour_len = mean_tour_len_nodes(x_edges_values, bs_nodes)
538             gt_tour_len = np.mean(batch.tour_len)
539
540             running_nb_data += batch_size
541             running_loss += batch_size* loss.data.item()
542             running_pred_tour_len += batch_size* pred_tour_len
543             running_gt_tour_len += batch_size* gt_tour_len
544             running_nb_batch += 1
545
546             result = ('loss:{loss:.4f} pred_tour_len:{pred_tour_len:.3f} gt
                    _tour_len:{gt_tour_len:.3f}'.format(
547                 loss=running_loss/running_nb_data,
548                 pred_tour_len=running_pred_tour_len/running_nb_data,
549                 gt_tour_len=running_gt_tour_len/running_nb_data))
550             master_bar.child.comment = result
551
552     loss = running_loss/ running_nb_data
553     err_edges = 0 # running_err_edges/ running_nb_data
554     err_tour = 0 # running_err_tour/ running_nb_data
555     err_tsp = 0 # running_err_tsp/ running_nb_data
556     pred_tour_len = running_pred_tour_len/ running_nb_data
557     gt_tour_len = running_gt_tour_len/ running_nb_data
558
559     return time.time()-start_test, loss, err_edges, err_tour, err_tsp, pred
            _tour_len, gt_tour_len
560
561
```

```
562 #whole piplenie at once
563 def main(config):
564     net = nn.DataParallel(ResidualGatedGCNModel(config, dtypeFloat,
        dtypeLong))
565     if torch.cuda.is_available():
566         net.cuda()
567     print(net)
568 
569     nb_param = 0
570     for param in net.parameters():
571         nb_param += np.prod(list(param.data.size()))
572     print('Number of parameters:', nb_param)
573 
574     log_dir = f"./logs/{config.expt_name}/"
575     os.makedirs(log_dir, exist_ok=True)
576     json.dump(config, open(f"{log_dir}/config.json", "w"), indent=4)
577     writer = SummaryWriter(log_dir)  # Define Tensorboard writer
578 
579     num_nodes = config.num_nodes
580     num_neighbors = config.num_neighbors
581     max_epochs = config.max_epochs
582     val_every = config.val_every
583     test_every = config.test_every
584     batch_size = config.batch_size
585     batches_per_epoch = config.batches_per_epoch
586     accumulation_steps = config.accumulation_steps
587     learning_rate = config.learning_rate
588     decay_rate = config.decay_rate
589     val_loss_old = 1e6  # For decaying LR based on validation loss
590     best_pred_tour_len = 1e6  # For saving checkpoints
591 
592     optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
593     print(optimizer)
594 
595     epoch_bar = master_bar(range(max_epochs))
596     for epoch in epoch_bar:
597         writer.add_scalar('learning_rate', learning_rate, epoch)
598 
599         train_time, train_loss, train_err_edges, train_err_tour, train_err_
            tsp, train_pred_tour_len, train_gt_tour_len = train_one_epoch(
```

```
                net, optimizer, config, epoch_bar)
600         epoch_bar.write('t: ' + metrics_to_str(epoch, train_time, learning_
                rate, train_loss, train_err_edges, train_err_tour, train_err_tsp
                , train_pred_tour_len, train_gt_tour_len))
601         writer.add_scalar('loss/train_loss', train_loss, epoch)
602         writer.add_scalar('pred_tour_len/train_pred_tour_len', train_pred_
                tour_len, epoch)
603         writer.add_scalar('optimality_gap/train_opt_gap', train_pred_tour_
                len/train_gt_tour_len - 1, epoch)
604 
605         if epoch % val_every == 0 or epoch == max_epochs-1:
606             val_time, val_loss, val_err_edges, val_err_tour, val_err_tsp,
                    val_pred_tour_len, val_gt_tour_len = test(net, config, epoch
                    _bar, mode='val')
607             epoch_bar.write('v: ' + metrics_to_str(epoch, val_time,
                    learning_rate, val_loss, val_err_edges, val_err_tour, val_
                    err_tsp, val_pred_tour_len, val_gt_tour_len))
608             writer.add_scalar('loss/val_loss', val_loss, epoch)
609             writer.add_scalar('pred_tour_len/val_pred_tour_len', val_pred_
                    tour_len, epoch)
610             writer.add_scalar('optimality_gap/val_opt_gap', val_pred_tour_
                    len/val_gt_tour_len - 1, epoch)
611 
612 
613             if val_pred_tour_len < best_pred_tour_len:
614                 best_pred_tour_len = val_pred_tour_len  # Update best
                        prediction
615                 torch.save({
616                     'epoch': epoch,
617                     'model_state_dict': net.state_dict(),
618                     'optimizer_state_dict': optimizer.state_dict(),
619                     'train_loss': train_loss,
620                     'val_loss': val_loss,
621                 }, log_dir+"best_val_checkpoint.tar")
622 
623             # Update learning rate
624             if val_loss > 0.99 * val_loss_old:
625                 learning_rate /= decay_rate
626                 optimizer = update_learning_rate(optimizer, learning_rate)
627 
```

```
628                 val_loss_old = val_loss  # Update old validation loss
629 
630             if epoch % test_every == 0 or epoch == max_epochs-1:
631                 test_time, test_loss, test_err_edges, test_err_tour, test_err_
                    tsp, test_pred_tour_len, test_gt_tour_len = test(net, config
                    , epoch_bar, mode='test')
632                 epoch_bar.write('T: ' + metrics_to_str(epoch, test_time,
                    learning_rate, test_loss, test_err_edges, test_err_tour,
                    test_err_tsp, test_pred_tour_len, test_gt_tour_len))
633                 writer.add_scalar('loss/test_loss', test_loss, epoch)
634                 writer.add_scalar('pred_tour_len/test_pred_tour_len', test_pred
                    _tour_len, epoch)
635                 writer.add_scalar('optimality_gap/test_opt_gap', test_pred_tour
                    _len/test_gt_tour_len - 1, epoch)
636 
637             torch.save({
638                 'epoch': epoch,
639                 'model_state_dict': net.state_dict(),
640                 'optimizer_state_dict': optimizer.state_dict(),
641                 'train_loss': train_loss,
642                 'val_loss': val_loss,
643             }, log_dir+"last_train_checkpoint.tar")
644 
645             if epoch != 0 and (epoch % 250 == 0 or epoch == max_epochs-1):
646                 torch.save({
647                     'epoch': epoch,
648                     'model_state_dict': net.state_dict(),
649                     'optimizer_state_dict': optimizer.state_dict(),
650                     'train_loss': train_loss,
651                     'val_loss': val_loss,
652                 }, log_dir+f"checkpoint_epoch{epoch}.tar")
653 
654         return net
655 
656 #visualization
657 
658 if notebook_mode==True:
659     net.eval()
660     batch_size = 10
661     num_nodes = config.num_nodes
```

```
662     num_neighbors = config.num_neighbors
663     beam_size = config.beam_size
664     test_filepath = config.test_filepath
665     dataset = iter(GoogleTSPReader(num_nodes, num_neighbors, batch_size,
            test_filepath))
666     batch = next(dataset)
667 
668     with torch.no_grad():
669         x_edges = Variable(torch.LongTensor(batch.edges).type(dtypeLong),
                requires_grad=False)
670         x_edges_values = Variable(torch.FloatTensor(batch.edges_values).
                type(dtypeFloat), requires_grad=False)
671         x_nodes = Variable(torch.LongTensor(batch.nodes).type(dtypeLong),
                requires_grad=False)
672         x_nodes_coord = Variable(torch.FloatTensor(batch.nodes_coord).type(
                dtypeFloat), requires_grad=False)
673         y_edges = Variable(torch.LongTensor(batch.edges_target).type(
                dtypeLong), requires_grad=False)
674         y_nodes = Variable(torch.LongTensor(batch.nodes_target).type(
                dtypeLong), requires_grad=False)
675 
676         edge_labels = y_edges.cpu().numpy().flatten()
677         edge_cw = compute_class_weight("balanced", classes=np.unique(edge_
                labels), y=edge_labels)
678         print("Class weights: {}".format(edge_cw))
679 
680         y_preds, loss = net.forward(x_edges, x_edges_values, x_nodes, x_
                nodes_coord, y_edges, edge_cw)
681         loss = loss.mean()
682 
683         bs_nodes = beamsearch_tour_nodes_shortest(
684             y_preds, x_edges_values, beam_size, batch_size, num_nodes,
                    dtypeFloat, dtypeLong, probs_type='logits')
685 
686         pred_tour_len = mean_tour_len_nodes(x_edges_values, bs_nodes)
687         gt_tour_len = np.mean(batch.tour_len)
688         print("Predicted tour length: {:.3f} (mean)\nGroundtruth tour
                length: {:.3f} (mean)".format(pred_tour_len, gt_tour_len))
689 
690         for idx, nodes in enumerate(bs_nodes):
```

```
691             if not is_valid_tour(nodes, num_nodes):
692                 print(idx, " Invalid tour: ", nodes)
693 
694         plot_predictions_beamsearch(x_nodes_coord, x_edges, x_edges_values,
                y_edges, y_preds, bs_nodes, num_plots=batch_size)
```

10. pirmkods. GCN implementācija: atlikušais, kas nepieciešams GCN darbībai

Bakalaura darbs "Grafu neirona tīklu izmantošana optimālo maršrutu noteikšanā" izstrādāts LU Fizikas, matemātikas un optometrijas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: Rolands Aleksandrs Rudenko

(paraksts) (datums)

Rekomendēju darbu aizstāvēšanai:

Vadītājs: Mg. math. Artis Alksnis

(paraksts) (datums)

Recenzents: Mg. math. Jānis Gredzens

(paraksts) (datums)

Darbs iesniegts Matemātikas nodaļā 2023. gada ____. jūnijā.

(Dekāna pilnvarotā persona: vecākā metodiķe Inita Šneidere)

Darbs aizstāvēts bakalaura gala parbaudījuma komisijas sēdē

2023. gada ____. jūnijā.