LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ALGORITMI VĀRDU SALIKŠANAI KRUSTVĀRDU MĪKLU REŽĢĪ

MAĢISTRA DARBS

Autors: Oskars Čaikovskis

Stud. apl. Nr. oc17004

Darba vadītājs: Dr. sc. comp. Kaspars Balodis

RĪGA 2019

ANOTĀCIJA

Šī maģistra darba mērķis ir apskatīt dažādu algoritmu implementēšanu krustvārdu mīklu veidošanas automatizācijai, veikt pētījumu par algoritmiem un izveidot datorprogrammu, kas veic krustvārdu mīklu režģu veidošanas (burtu salikšanu režģī tā, lai tie veidotu vārdus) automatizāciju.

Maģistra darbs sastāv no divām daļām – teorētiskās un praktiskās. Teorētiskajā daļā ir apkopota vispārīga informācija par krustvārdu mīklām un dažādiem algoritmiem, kurus var izmantot to veidošanā, kā arī to salīdzinājums, ņemot vērā latviešu valodas ortogrāfijas īpatnības. Praktiskajā daļā ir aprakstīta datorprogramma krustvārdu mīklu veidošanas automatizācijai un apkopoti no tās iegūtie dati.

Maģistra darbā tiek apskatīti dažādi algoritmi vārdu salikšanai krustvārdu mīklu režģī, kā arī krustvārdu mīklu režģu veidošana. Šie algoritmi tika implementēti datorprogrammā un tika salīdzināta to veiktspēja.

Maģistra darbā iekļauta kompilēta datoeprogramma *Windows* vidē, koda datne un datorprogrammas apraksts. Šī datorprogramma ir veidota kā konsoles lietotāja interfeisa (*CUI*) programmu.

Maģistra darbs sastāv no ievada, 5 nodaļām, secinājumiem, izmantotās literatūras saraksta un pielikumiem. Kopā maģistra darbā ir 61 lpp. un 15 attēli.

Atslēgvārdi

- Krustvārdu mīklas,
- Algoritmi,
- Optimizācija.

ABSTRACT

The title of the Master's Thesis is “Algorithms for word assembly in crossword puzzle grid”. The purpose of this thesis is to examine the implementation of various algorithms for automation of crossword puzzles, to carry out a research on algorithms and to create a program that performs the automation of crossword puzzle creation (compilation of letters in a grid to form words).

The Master's Thesis consists of two parts – theoretical and practical. The theoretical part summarizes general information about crossword puzzles and various algorithms that can be used in their creation, as well as their comparison, taking into account peculiarities of Latvian orthography. The second part describes the program and collect the data from it.

The Master's Thesis reviews various algorithms for assembling words in a crossword puzzle grid, as well as algorithms for creating crossword puzzle grids. These algorithms are be implemented in a computer program and their performance is compared.

Master's Thesis includes a compiled program in the *Windows* environment, a code file and a description of this work. The program is built as a console user interface (*CUI*).

The Master's Thesis consists of an introduction, 5 chapters, conclusions, a list of references and affix. Master's Thesis consists of 61 pages and 15 images.

Keywords

- Crossword puzzle,
- Agorithms,
- Optimisation.

AUTOREFERĀTS

Maģistra darbā ir izveidota datorprogramma C# valodā, kas ir spējīga izveidot krustvārdu mīklas no vārdu saraksta un dota režģa. Lielākā daļa iepriekš veikto pētījumu tika veikti angļu valodā. Tika salīdzināti algoritmi vārdu salikšanai krustvārdu mīklu režģī latviešu un angļu valodas.

Darbā ir veikta literatūras izpēte un apkopojums. Iepriekš veiktu pētījumu izpēti un apkopojumu ir veicis maģistra darba autors. Maģistra darbā tiek apkopoti pētījumi par krustvārdu mīklu veidošanas automatizāciju un izvirzīta hipotēze par krustvārdu mīklu automatizēšanu latviešu valodā. Maģistra darbā informācija tika iegūta gan no zinātniskajiem avotiem, gan interneta avotiem. Visas atsauces ir atzīmētas literatūras un avotu sarakstā.

Maģistra darbā tiek aplūkota problēma un to iespējamie risinājumi. Maģistra darba veikšanas gaitā šie risinājumi tika aplūkoti detalizētāk un tika implementēti iespējamie risinājumi datorprogrammā un iegūtie rezultāti salīdzināti ar līdzīgiem pētījumiem, kuri ir veikti angļu valodā.

Maģistra darbā autors veica literatūras izpēti un veica patstāvīgus pētījumus izmantojot paša programmētu datorprogrammu. Darba apjoms ir mērāms kā 20 kredītpunktu darbs.

Darbā iegūtie rezultāti ir salīdzināmi ar literatūrā aprakstītajiem. Rezultāti norāda uz vairākām tendencēm, kuri ir pielīdzināmi iepriekš veiktiem pētījumiem.

Maģistra darba teksts ir pārlasīts un ir izlabotas pareizrakstības un interpunkcijas kļūdas, kā arī ir lietota nozares terminoloģija. Darba autors ir centies ievērot fakultātes gala darba metodoloģijas norādījumiem.

Autors ir izpētījis publiski pieejamo zinātnisko rakstu krājumus. Visas idejas, formulējumi, attēli utt., kas aizgūti no citiem autoriem, ir atzīmēti ar attiecīgām literatūras atsaucēm. Visa izmantotā literatūra ir norādīta izmantotās literatūras sarakstā. Darba teksta gabali, kas ir burtisks tulkojums vai tuvs pārstāsts no kāda viena literatūras avota, ir atzīmēti kā teksta aizguvumi.

SATURA RĀDĪTĀJS

ANOTĀCIJA....2
Atslēgvārdi....2
ABSTRACT....3
Keywords....3
AUTOREFERĀTS....4
SATURA RĀDĪTĀJS....5
APZĪMĒJUMU SARAKSTS....6
IEVADS....7
1. KRUSTVĀRDU MĪKLAS....8
1.1. Ortogrāfija....10
1.2. Krustvārdu mīklu varianti....10
2. VĀRDNĪCA....12
3. KRUSTVĀRDU MĪKLU REŽĢU VEIDOŠANA....13
3.1. Krustvārdu mīklu veidošanas vispārīgs apraksts....13
3.2. "Vārdu pa vārdam" instancēšanas metode....13
3.3. "Burtu pa burtam" instancēšanas metode....14
3.4. Kontrolpunktu meklēšanas metode....14
3.5. *WordNet* implementācija....15
4. LATVIEŠU UN ANGĻU VALODU SALĪDZINĀJUMS....16
5. DATORPROGRAMMA....23
5.1. Datorprogrammas vispārīgs apraksts....23
5.2. Pielietotās vārdnīcas....27
5.3. Pielietotie algoritmi....29
5.4. Pielietotie režģi....31
REZULTĀTI UN DISKUSIJA....36
SECINĀJUMI....42
IZMANTOTĀ LITERATŪRA UN AVOTI....44
Interneta resursi....45
PIELIKUMI....46
1. pielikums. Vārdu garums latviešu un angļu valodās....47
2. pielikums. Burtu skaits vārdos latviešu un angļu valodās....48
3. pielikums. Pirmkods....49
DOKUMENTĀRĀ LAPA....61

APZĪMĒJUMU SARAKSTS

- Ailes – baltie kvadrāti krustvārdu mīklu režģī. Aile, kurā jāraksta vārda pirmais burts, tiek numurēta,
- Atkāpe (*backtrack*) – process, kurā tiek izdzēsts viens vai vairāki no iepriekš režģī ierakstītajiem burtiem,
- Krustvārdu mīklu režģis – spēles laukums, tas sastāv no baltiem un melniem kvadrātiem. Kvadrāti ir vienāda izmēra,
- Krustojošais burts – burts, kas kopīgs diviem vārdiem,
- Krustpunkts – aile, kurā tiek ierakstīts krustojošais burts,
- Norādījumi – parasti jautājumi, kuru atbildes jāieraksta krustvārdu mīklā pie attiecīgā numura horizontāli vai vertikāli,
- Vārdnīca – izmantojamo vārdu saraksts,
- Vārdu krustošanās – divi vai vairāk vārdi, kuriem ir kopīgi burti un tos iespējams pārklāt, sarakstot vertikālā un horizontālā virzienā,
- Vārdu likšana – krustvārdu mīklu tukšo aiļu aizpildīšana ar burtu kopu tā, lai tie secīgi, vertikāli no augšas uz leju, vai horizontāli no kreisās puses uz labo pusi, veidotu vārdu.

IEVADS

Krustvārdu mīkla ir spēle, kuru spēlē rakstiski uz drukāta izdevuma lapām vai datorā. Tā satur režģi ar baltiem un melniem kvadrātiem. Šī režģa baltās ailes spēlētājs aizpilda ar burtiem tā, lai režģī vertikāli un horizontāli veidotos vārdi. Ailes ir numurētas. Kopā ar režģi krustvārdu mīklām ir doti arī norādījumi – parasti jautājumi, kuru atbildes jāieraksta krustvārdu mīklā pie attiecīgā numura horizontāli vai vertikāli.

Šī maģistra darba mērķis ir apskatīt dažādu algoritmu implementēšanu krustvārdu mīklu veidošanas automatizācijai, veikt pētījumu par algoritmiem un izveidot datorprogrammu, kas veic krustvārdu mīklu režģu veidošanas automatizāciju.

Šajā maģistra darbā tiek aplūkoti standarta jeb britu/dienvidāfrikāņu krustvārdu mīklu veidošanas algoritmi un to pielietojums, ņemot vērā ortogrāfijas īpatnības latviešu valodā.

Šis maģistra darbs sastāv no maģistra darba teorētiskās daļas. Teorētiskajā daļā ir apkopota vispārīga informācija par krustvārdu mīklām un dažādiem algoritmiem, kurus var izmantot to veidošanā, kā arī to salīdzinājums, ņemot vērā latviešu valodas ortogrāfijas īpatnības. Praktiskajā maģistra darba daļā tiks aprakstīta programma un apkopoti no tās iegūtie dati.

1. KRUSTVĀRDU MĪKLAS

Krustvārdu mīkla ir populāra vienspēlētāja viktorīnas spēle, kuru spēlē uz drukāta izdevuma lapām, kā arī datorā. Latvijā tiek izdoti vairāki populāri krustvārdu mīklu žurnāli, piemēram, SIA "Žurnāls Santa" izdotais krustvārdu mīklu žurnāls "Mezgls", kas tiek izdots divas reizes mēnesī, un tā tirāža ir vairāk kā 14 tūkstoši eksemplāru. *[7]*

Pirmās liecības par krustvārdu mīklām līdzīgām spēlēm ir atklātas Pompejā, un tās tika veidotas 6.-3. gadsimtā pirms mūsu ēras. Pirmā mūsdienu krustvārdu mīkla tika izdota 1913. gada 21. decembrī laikrakstā *The New York World [8]* angļu valodā, savukārt pirmā krustvārdu mīkla latviešu valodā tika izdota 1933. gadā žurnālā "Krusta-Mīkla, Šachs, Bridžs". *[9]*

Krustvārdu mīklu parasti pilda viens spēlētājs. Spēles laukums satur režģi ar baltiem un melniem kvadrātiem. Šīs spēles mērķis ir režģa balto aiļu aizpildīšana ar burtiem tā, lai režģī vertikāli un horizontāli veidotos vārdi. Ailes ir numurētas. Kopā ar režģi krustvārdu mīklām ir doti arī norādījumi – parasti tie ir jautājumi, kuru atbildes jāieraksta krustvārdu mīklā pie attiecīgā numura horizontāli un vertikāli.

Latvijā izplatītajām mīklām bieži vien ir neregulāras malas, taču citviet pasaulē mīklas laukums parasti ir četrstūrains. Krustvārdu mīklu aiļu kopu saturs ir latviešu alfabēta burti tādā secībā, lai tie veidotu vārdus. Tā kā garākais literārais vārds latviešu valodā ir "pretpulksteņrādītājvirziens" *[10]* un tā garums ir 27 burti, bet garākais zinātniskais termins latviešu valodā ir "trimetilheksametilēndiizocianāts" un tā garums ir 32 burti, garākais iespējamais burtu kopas garums ir 32, īsākais 2.

Krustvārdu mīklu raksturojošā iezīme ir vārdu krustošanās, tas ir, pastāv divi vai vairāk vārdi, kuriem ir kopīgi burti un tos iespējams pārklāt, vārdus sarakstot vertikālā un horizontālā virzienā, piemēram, vārdos MAĢISTRS un DARBS ir trīs kopīgi burti – *a*, *s*, *t* un otrs *s*. šie vārdi var krustoties kādā no šīm četrām pozīcijām.

Bez krustvārdu mīklām pastāv arī spēle *Scrabble*, kuras mērķis ir aizpildīt spēles laukumu ar burtiem, līdzīgi kā krustvārdu mīklās. Algoritmi, kas paredzēti *Scrabble* spēlei, iespējams, var tikt izmantoti arī krustvārdu mīklu režģu veidošanai šo spēļu līdzību dēļ.

Citas līdzīgas spēles, kuru algoritmus iespējams izmantot ir ciparu mīkla, apļu mīkla, japāņu mīkla un vārdu meklēšanas burtu režģī spēle.

Tā kā datorprogrammatūru neierobežo lapas, iespējams veidot arī krustvārdu mīklas ar vairāk par divām dimensijām, kur vārdi var būt rakstīti režģī ne tikai vertikāli un horizontāli, bet arī tālumā.

Maģistra darbā apskatīta klasiskā, jeb britu/dienvidāfrikāņu krustvārdu mīkla un ar tās ģenerēšanu saistītie algoritmi.

1.1. Ortogrāfija

Krustvārdu mīklu veidošanā netiek ņemts vērā, vai vārds rakstāms ar lielu vai mazu burtu, kā arī saīsinājumos netiek likti punkti. Dažādās valodās, piemēram, angļu, franču, itāļu spāņu, un rumāņu valodās vienādi burti ar dažādām diaktriskajām zīmēm parasti tiek uzskatīti par vienu un to pašu burtu. Šādus burtus var rakstīt vienā rūtiņā, savukārt apostrofus krustvārdu mīklās neizmanto.

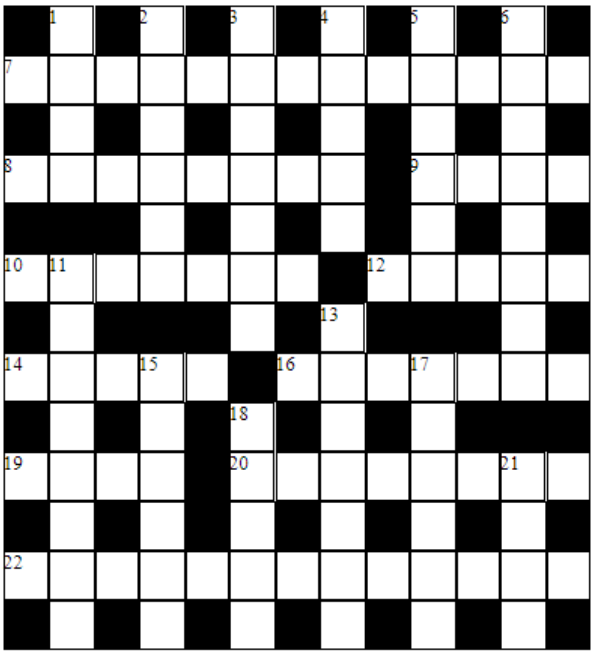
Krustvārdu mīklās nīderlandiešu un spāņu valodās digrāfi tiek rakstīti vienā rūtiņā. Krustvārdu mīklās vācu valodā burti *ä*, *ö*, *ü* un *ß* tiek aizstāti attiecīgi ar *ae*, *oe*, *ue* un *ss*.*[11]* Krievu valodā burti *e* un *ë* tiek uzskatīti par vienu un to pašu burtu un visi *ë* tiek aizstāti ar *e*, bet burti *u* un *ŭ* — par dažādiem.

Krustvārdu mīklas latviešu valodā parasti tiek ņemtas vērā garumzīmes, mīkstinājuma zīmes un jumtiņi, tas ir, burti *a* un *ā* vai *s* un *š* nevar aizstāt cits citu un tiek uzskatīti kā atšķirīgi burti. Latviešu valodā krustvārdu mīklās visi divskaņi, izņemot *o*, aizņem divas ailes un katrs burts tiek rakstīts atsevišķi. Britu/dienvidāfrikāņu krustvārdu mīklās vārdi parasti tiek uzskaitīti to pamatformā, savukārt zviedru krustvārdu mīklās iespējams vārdus rakstīt arī locījumos, bet tas jānorāda norādījumā.

1.2. Krustvārdu mīklu varianti

Ir dažādi krustvārdu mīklu tipi jeb varianti, kurām doti dažādu tautu nosaukumi, piemēram, britu/dienvidāfrikāņu, amerikāņu, japāņu un zviedru. Pēc papildus nosacījumiem atšķir arī bilžu mīklas un krustskaitļu mīklas.

Katram krustvārdu mīklu variantam ir savas īpatnības. Latvijā populārākie krustvārdu mīklu varianti ir standarta jeb britu/dienvidāfrikāņu krustvārdu mīklas un zviedru krustvārdu mīklas.



Attēls 1: Britu/dienvidāfrikāņu krustvārdu mīkla

Standarta jeb Britu/dienvidāfrikāņu krustvārdu mīklās vārdi nevar atrasties blakus, tas ir, krustvārdu mīklās baltās ailes veido vertikālas un horizontālas aiļu kopas, kuras var krustoties, bet divas vertikālas vai divas horizontālas aiļu kopas nevar atrasties blakus viena otrai (*skatīt 1. attēlu*).

Atšķirībā no britu/dienvidāfrikāņu krustvārdu mīklu varianta, zviedru krustvārdu mīklās vārdi var atrasties viens otram blakus. Tajās ailes netiek numurētas, bet tā vietā norādes tiek rakstītas tukšajās ailēs un virzienu nosaka bultiņa, kas atrodas pie norādes.

2. VĀRDNĪCA

Viens no svarīgākajiem faktoriem krustvārdu mīklu veidošanā, kas būtiski ietekmē automatizētas krustvārdu mīklu veidošanas efektivitāti, ir vārdnīcas struktūra. Vārdnīca ir to vārdu saraksts, kuru var izmantot krustvārdu mīklu veidošanā. Šis vārdu saraksts var būt gan ļoti ierobežots, izmantojot tikai tos vārdus, kurus krustvārdu mīklu veidotājs ir izvēlējies, gan ļoti plašs, un tas var saturēt visus vai daudzus attiecīgajā valodā esošos vārdus. Ja vārdu saraksts ir ierobežots, tad izveidotajā krustvārdu mīklā jābūt attēlotiem visiem vārdiem, bet ja vārdu saraksts ir plašs veidotājs izvēlas spēles laukuma izmēru, un tajā tiek ievietoti gadījuma vārdi tādā veidā, lai būtu pēc iespējas mazāk brīvo lauku.

Vārdnīcās var būt gan sugas vārdi, gan īpašvārdi. Britu/dienvidāfrikāņu krustvārdu mīklu vārdnīcās vārdi parasti tiek uzskaitīti to pamatformā (lietvārdiem – nominatīvā, darbības vārdiem – nonoteiksmē), savukārt zviedru krustvārdu mīklās vārdus iespējams rakstīt arī locījumos, bet tam jābūt minētam norādēs.

Pastāv dažādi algoritmi vārdu salikšanai krustvārdu mīklu režģī. Parasti algoritmi, gan no ierobežotas, gan no plašas vārdnīcas ieraksta režģī vārdus, sākot ar garākajiem, kā arī tos vārdus, kuros ir visbiežāk sastopamie burti. *[12]* Latviešu valodā biežāk sastopamie burti ir *s*, *t*, *i*, *a* un *r*, savukārt retāk izmantotie burti ir *ķ*, *ņ*, *ļ*, *h*, *ž*, *ģ* un *č* (*skatīt 4. nodaļu*).

Iespējams grupēt vārdus vārdnīcā ne tikai pēc to garuma un burtu biežuma, bet arī pēc to atminēšanas sarežģītības.

Vārdnīca tiek nolasīta programmatūras inicializēšanas laikā. Tipiski plaša vārdnīca ir lineāras formas (vārdi tiek rakstīti saraksta formā, nevis, piemēram, koka struktūrā) un satur vārdus alfabētiskā secībā katram vārda garumam. *[3]* Lai atbilstošu vārdu meklēšana būtu ātra, nepieciešams veikt indeksēšanu, kā arī nepieciešams pārtraukt meklēšanu, ja ir atrasts pietiekoši daudz atbilstošu vārdu – nav nepieciešams veikt pilnu meklēšanu.

Iespējams izveidot vairākas vārdnīcas, kurās atrodas tikai konkrēti vārdi ar atbilstošiem burtu kopu algoritmiem, piemēram, vārdi, kuri ir piecu burtu garumā un kuriem trešais burts pēc kārtas ir “a”.

3. KRUSTVĀRDU MĪKLU REŽĢU VEIDOŠANA

3.1. Krustvārdu mīklu veidošanas vispārīgs apraksts

Krustvārdu mīklu režģu manuāla veidošana ir laikietilpīgs process. Lai izveidotu krustvārdu mīklu, ir nepieciešams dot vairākus vārdus, kas krustojas, tas ir, tiem ir burti, kas pēc noteiktiem principiem atkārtojas dažādos vārdos. Šī iemesla dēļ nepieciešams izveidot krustvārdu mīklu veidošanas automatizāciju.

Krustvārdu mīklu režģu automatizācijas programmā nepieciešamas vismaz divas komponentes – tukšs režģis (ar vai bez robežām), kurš programmas darbības laikā tiek aizpildīts ar burtiem un vārdnīca – noteiktā secībā sarindots vai sagrupēts vārdu saraksts, no kuriem iespējams iegūt nepieciešamos burtus aizpildei krustvārdu mīklu režģī.

Krustvārdu mīklu režģi programmas sākumā var iedalīt divās kategorijās – režģī, kurā nav atzīmētas ailes, kurām jābūt tukšām, un režģī, kurā ir atzīmētas ailes, kurām jābūt tukšām. Ja nav atzīmēts, kurām ailēm jābūt tukšām, krustvārdu mīklu veidošanas programmai nepieciešams veikt vārdu likšanu režģī tā, lai tie būtu ar pēc iespējas lielāku blīvumu. Programma beidz darbu, ja režģī nav iespējams pievienot vairs nevienu vārdu. Savukārt, ja ir noteikts, kuras ailes būs tukšas, datorprogramma turpinās darbu līdz visas pārējās ailes būs aizpildītas vai būs notikusi noildze.

Krustvārdu mīklu veidošanas algoritmus iespējams arī kombinēt un pielāgot konkrētām situācijām.

3.2. “Vārdu pa vārdam” instancēšanas metode

Viena no vienkāršākajām krustvārdu mīklu veidošanas metodēm ir “Vārdu pa vārdam” instancēšanas metode (angļu; *Word-by-word instantiation method*). Šo metodi apraksta Gerijs Mīhanands un Pīters Grejs no Aberdīnas Universitātes.*[1]*

“Vārdu pa vārdam” instancēšanas metode ir mehāniska vārdu likšana krustvārdu mīklu režģī. Vārdi tiek izvēlēti no vārdnīcas, un tie tiek secīgi ievietoti krustvārdu mīklu režģī, kurā ir jau iezīmēts, kuras ailes būs tukšas. Ja nav neviens vārds, kas ietilpst krustvārdu mīklu režģī, notiek atkāpe – tiek izdzēsts kāds no iepriekš režģī ierakstītajiem vārdiem.

Programma, kura izmanto šo algoritmu turpina darbu līdz visas ailes, kuras nav atzīmētas kā tukšas, ir aizpildītas. Programmas efektivitāti ietekmē tas, kuras ailes ir atzīmētas kā tukšas, cik liela ir vārdnīca un cik daudzveidīgi ir vārdnīcā dotie vārdi, un kādā secībā lauki tiks aizpildīti vai nodzēsti. Metodi iespējams pielāgot, ka tā izvēlas vārdus kurus likt režģī nejauši. Šo algoritmu iespējams kombinēt ar citiem algoritmiem.

3.3. "Burtu pa burtam" instancēšanas metode

"Burtu pa burtam" instancēšanas metodē tiek atkārtoti izvēlēta kāda tukša režģa aile un aizpildīta ar burtu. Pirms katras ailes aizpildīšanas jāpārliecinās, ka šīs ailes aizpildīšana neizraisīs situāciju, ka nav iespējams kādu aiļu kopu aizpildīt ar vismaz vienu vārdu. *[1]* Programma beidz darbu, kad ir aizpildītas visas ailes, kuras nav atzīmētas kā tukšas.

Programmas efektivitāti ietekmē tas, kuras ailes ir atzīmētas kā tukšas, cik liela ir vārdnīca un cik daudzveidīgi ir vārdnīcā dotie vārdi, un kādā secībā lauki tiks aizpildīti vai nodzēsti. Efektīvi ir burtu ievietošanu sākt ar vārdiem, kuri ir garākie vai kuriem ir visvairāk krustpunktu.

3.4. Kontrolpunktu meklēšanas metode

Ariels Arbisers no Buenosairesas Universitātes apraksta kontrolpunktu meklēšanas krustvārdu mīklu veidošanas metodi. *[2]* Vārdnīca tiek grupēta pēc vārdu garuma un vārdu saraksti tiek turēti atsevišķās un neatkarīgās datnēs.

Krustvārdu mīklu veidošanas procesā tiek veidoti kontrolpunkti – krustvārdu mīklu režģu fiksēti stāvokļi tiek saglabāti kā atsauces nākotnes atkāpēm.

Tiek izveidota vērtēšanas skala, piemēram, izveidojot mērvienību "veidošanas progress". Šī mērvienība nosaka, cik režģa ailes tiek aizpildītas noteiktā laika vienībā. Kad kontrolpunktam ir augsts veidošanas progress, tas tiek atzīmēts kā kvalitatīvs.

Kontrolpunktos tiek iezīmēti stāvokļi, kad krustvārdu mīklu veidošanas laikā ir konstatēta liela izvēle, kādu vārdu likt kā nākamo krustvārdu mīklu režģī. Viena no iespējām, kā pieņemt lēmumu, cik lielai izvēlei jābūt, lai tiktu izveidots kontrolpunkts, var tikt implementēta ar mašīnmācīšanos vai metameklēšanu vārdnīcā *[4]*.

Katru reizi, kad tiek konstatēts, ka ir zems veidošanas progress, nepieciešams izdzēst visus burtus, kas ir tikuši ierakstīti pēc pēdējā kontrolpunkta izveides. Tad turpinās krustvārdu mīklu veidošanas process.

3.5. *WordNet* implementācija

WordnNet ir datubāze, kurā angļu valodas vārdi tiek grupēti sinonīmu kopās (*synsets*). Šajā datubāzē vārdiem ir dotas definīcijas, un tā saglabā sinonīmu skaitu. *[7] WordNet* implementāciju apraksta *Aoife Aherne* un Karls Vogels no Dublinas Universitātes. *[5,6]* Tā kā *WordNet* datubāze ir veidota kā semantisks tīmeklis, kura struktūra atspoguļo pašreizējos psiholingvistiskos modeļus, līdzīgi kā leksiskā informācija tiek glabāta cilvēka smadzenēs.

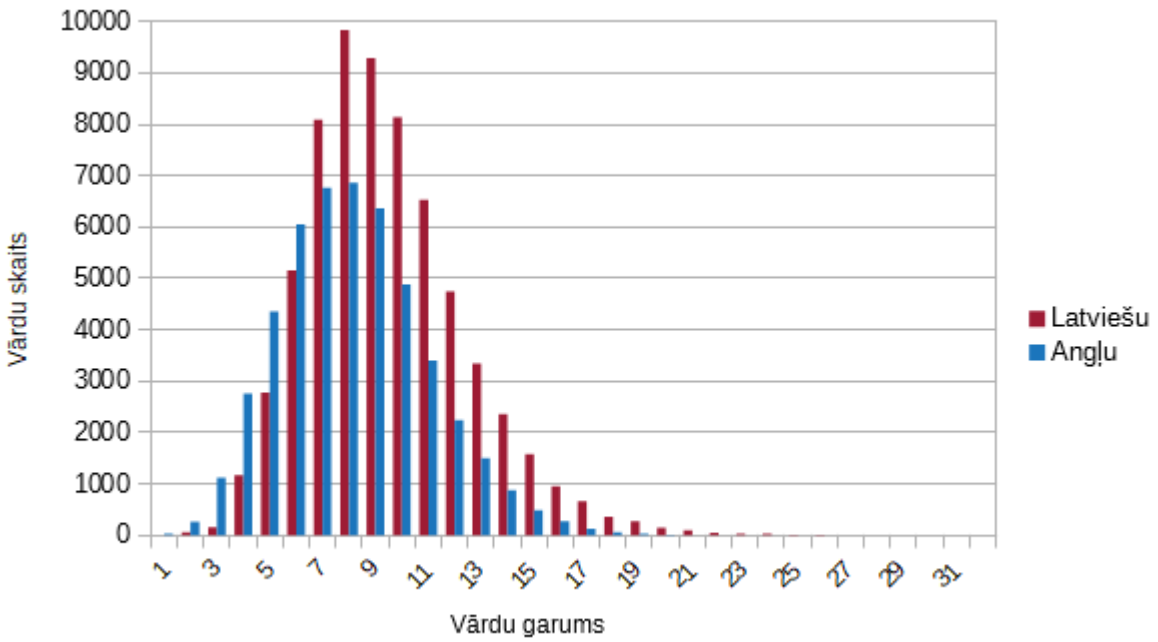
WordNet struktūra ir tuvāka sinonīmu vārdnīcai nekā skaidrojošās vārdnīcas struktūrai. Tādējādi *WordNet* tiek veikti vārdu būtības meklējumi, nevis parasta vārdu meklēšana. Tas ļauj iegūt vārdus, kas saistīti ar konkrētu tēmu. Šī valodas instrumenta avota kods ir brīvi pieejams. Turklāt ir izstrādātas dažādas saskarnes ar *WordNet*, ieskaitot *MySQL* datubāzes versiju, kas ģenerēta no *Prolog* pirmkoda.

4. LATVIEŠU UN ANGĻU VALODU SALĪDZINĀJUMS

Latviešu valodai un angļu valodai ir vairākas atšķirības. Viena no galvenajām – latviešu valodas alfabētam ir 33 burti, savukārt angļu valodas alfabētam ir tikai 26, no tiem 22 burti ir kopīgi abām valodām.

Tā kā starp latviešu un angļu valodām ir nozīmīgas atšķirības un lielākā daļa pētījumu krustvārdu mīklu veidošanas optimizācijai ir angļu valodā, tas var nozīmīgi ietekmēt pētījumu rezultātus latviešu valodā.

Elasticsearch GitHub vietnē *[14]* ir publicētas *Hunspell* gramatikas pārbaudes vārdnīcas vairākās valodās, ieskaitot latviešu un britu angļu valodās. Tās maģistra darbā tiks izmantotas kā krustvārdu mīklu vārdnīcas, kā arī latviešu un angļu valodu salīdzinājumam, lai noskaidrotu, vai atšķirības starp latviešu un angļu valodu ir pietiekoši nozīmīgas, lai krustvārdu mīklu veidošanas algoritmu efektivitāte atšķirtos starp šīm valodām. No interneta portāla *GitHub* tika lejupielādētas vārdnīcas latviešu un britu angļu valodās. Izmantojot izklājlapu programmatūru *LibreOffice Calc* tika analizēti 65 905 vārdi latviešu valodā, kā arī 48 505 vārdi britu angļu valodā.

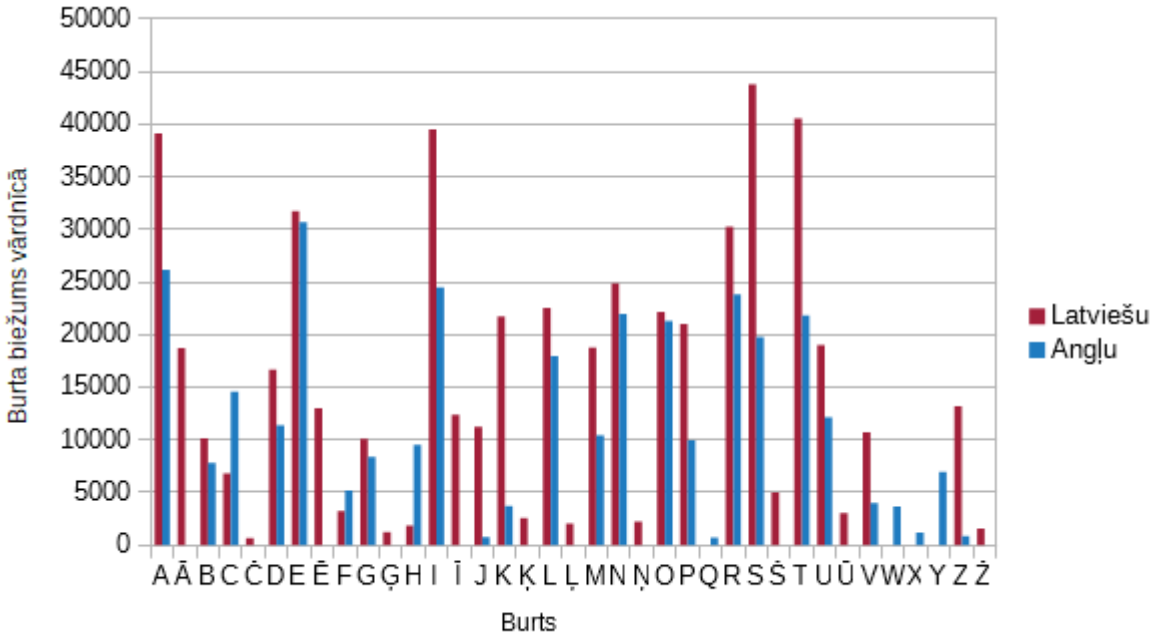


Attēls 2: Vārdu garumu latviešu un angļu valodās salīdzināšanas histogramma. Tabula ar vārdu garumiem pieejama 1. pielikumā.

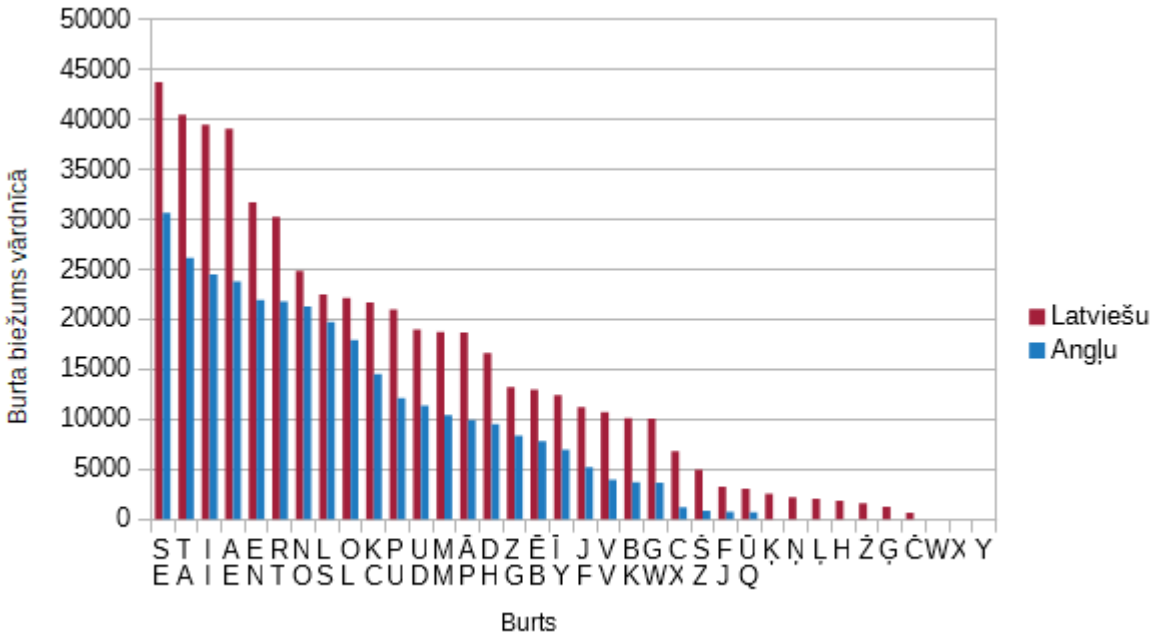
Valoda	Minimālais	Maksimālais	Vidējais	Mediāna	Moda	Bez 10%
Latviešu	1	32	9.5	9	8	6 → 13
Angļu	1	45	8.11	8	8	5 → 12

Tabula 1: Vārdu garumu latviešu un angļu valodās analīze – ir noteikts minimālais, maksimālais, vidējais, mediāna un moda vārdu garums. Tāpat tabulas pēdējā kolonnā ir aprakstīts minimālais un maksimālais burtu skaits vārdos, ja netiek aplūkoti 10% no īsākajiem un 10% no garākajiem vārdiem.

Kaut arī šajās vārdnīcās gan latviešu, gan angļu valodā biežākais burtu skaits (moda) ir vienāds – 8, vidēji šajās vārdnīcās latviešu valodā ir vairāk vārdu un tie ir vidēji garāki nekā angļu valodā (*skatīt 2. attēlu un 1. tabulu*). Tāpat latviešu valodā ir vairāk burtu un to izmantošanas biežums atšķiras no angļu valodas.

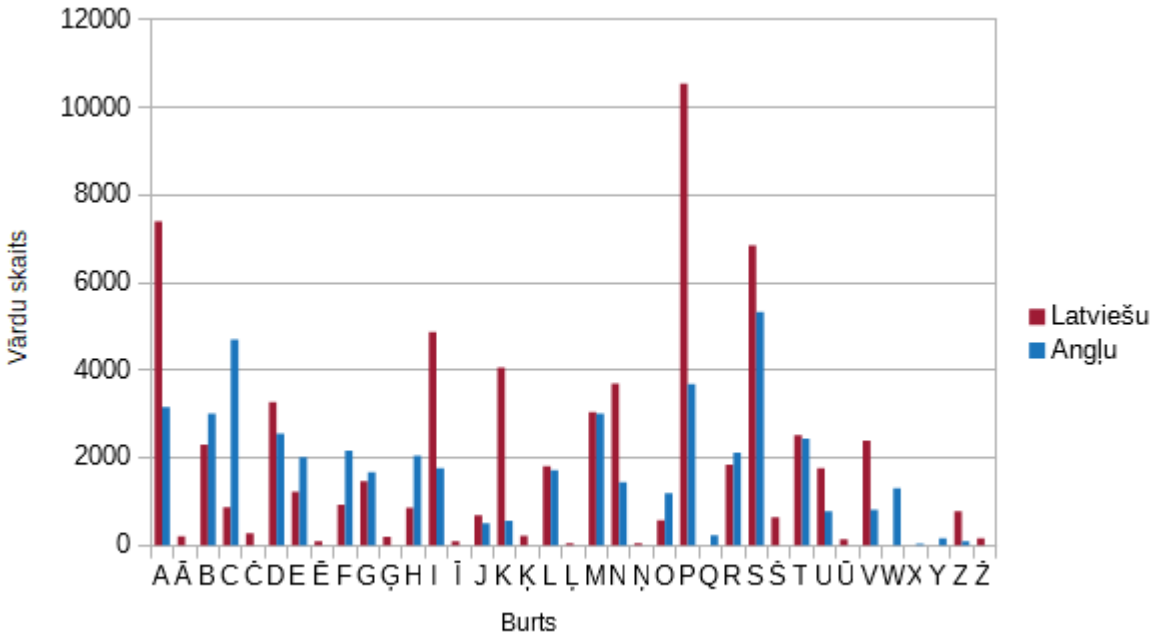


Attēls 3: Burtu biežuma latviešu un angļu valodās histogramma. Tabula ar burtu biežumu pieejama 2. pielikumā. Vārdnīcā ir vārdi ar nestandarta latviešu alfabēta burtiem: GWh, kWh, MWh, Slackware, TWh, Windows, Linux un yōs. Histogrammā burti ir sakārtoti pēc alfabēta.

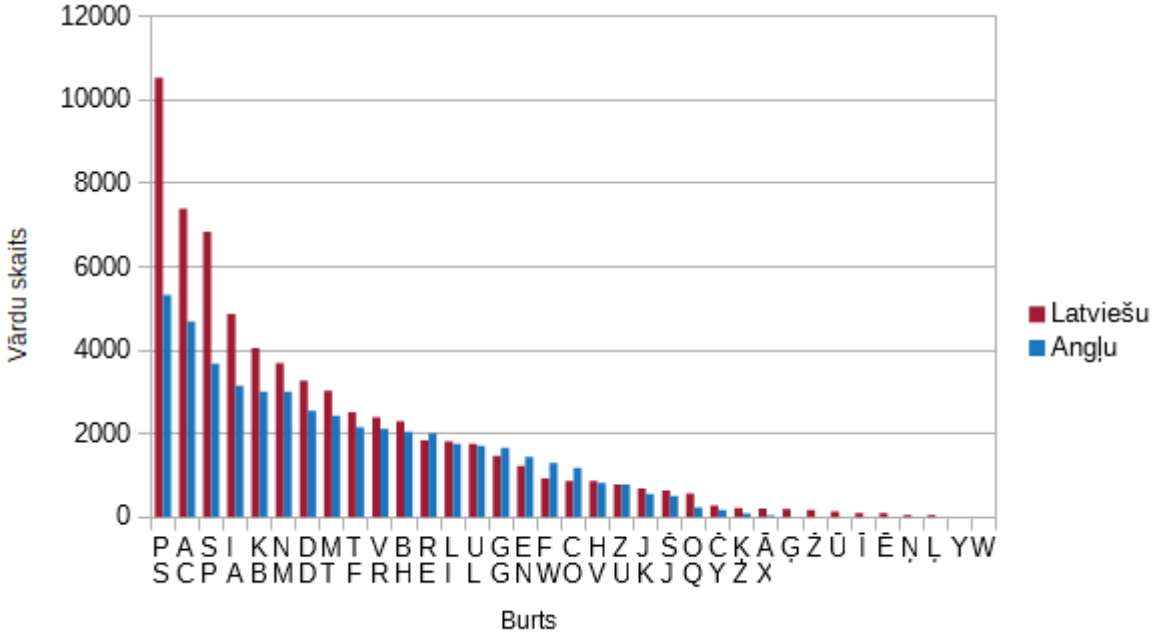


Attēls 4: Burtu biežuma latviešu un angļu valodās histogramma. Burti sakārtoti pēc to biežuma abās valodās. Pirmajā rindā norādīta burtu secība latviešu valodā, bet otrajā – angļu valodā.

Latviešu valodā burti bez mīkstinājuma zīmēm vai jumtiņiem parasti ir biežāk sastopami nekā burti ar mīkstinājuma zīmēm vai jumtiņiem (*skatīt 3. attēlu*). Visbiežāk sastopamie burti latviešu valodā ir *s*, *t*, *i* un *a*, bet visretāk sastopamie – *ū*, *ķ*, *ņ*, *ļ*, *h*, *ž*, *ģ* un *č*. Angļu valodā burtu biežums ir līdzīgs, bet to secība ir nedaudz citādāka (*skatīt 4. attēlu*).

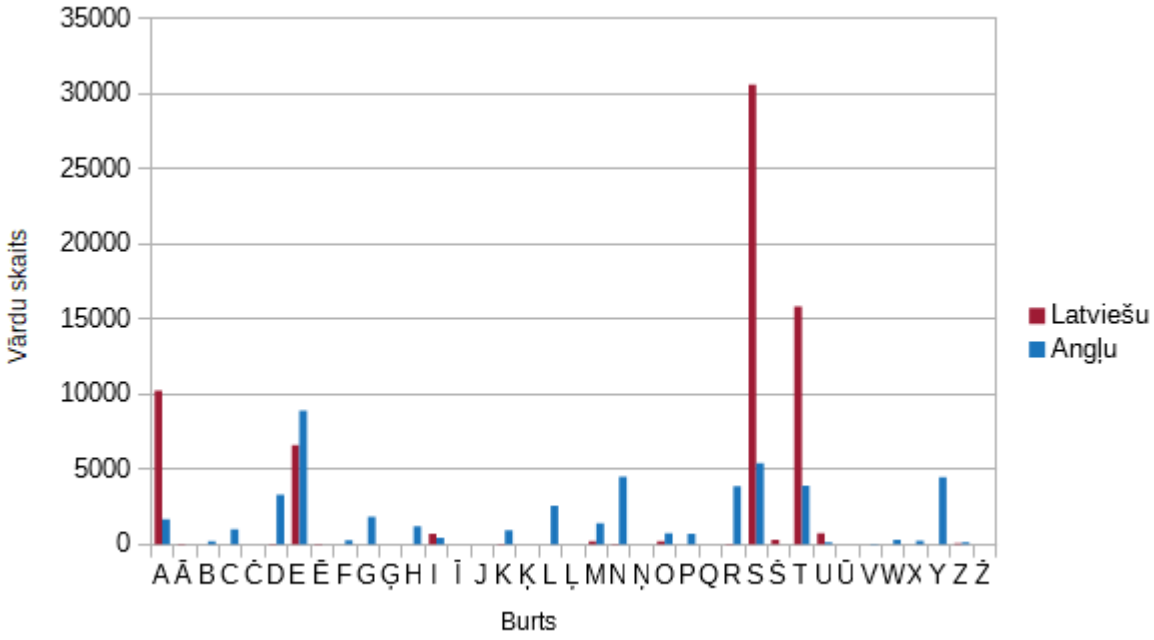


Attēls 5: Vārda pirmā burta biežuma latviešu un angļu valodās histogramma. Burti ir sakārtoti pēc alfabēta.

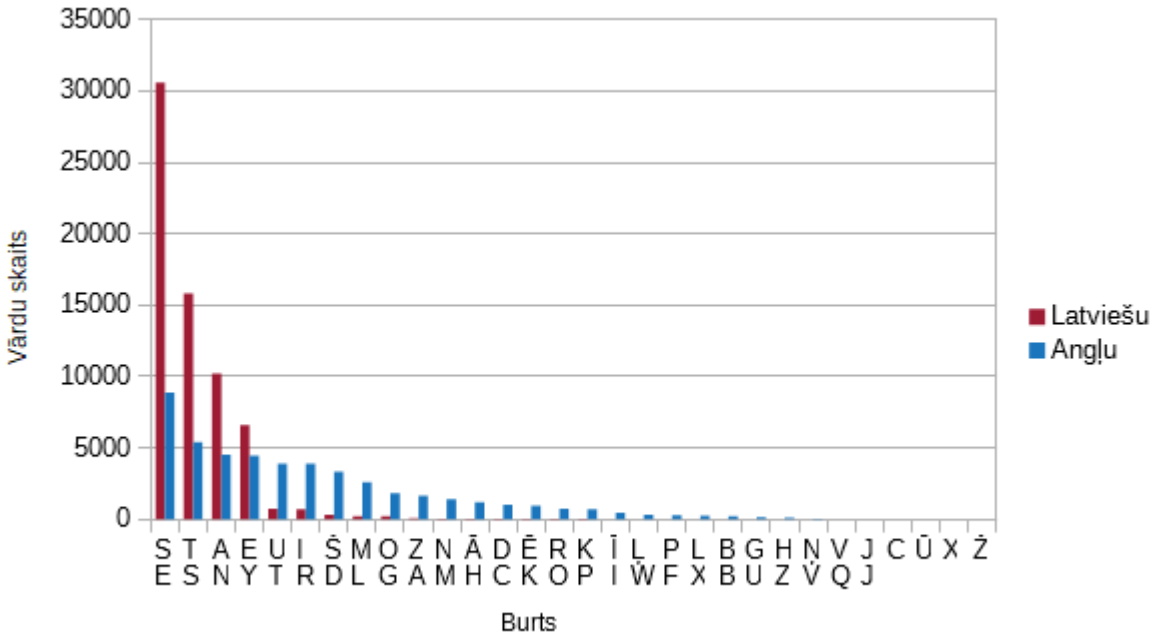


Attēls 6: Vārda pirmā burta biežuma latviešu un angļu valodās histogramma. Burti sakārtoti pēc to biežuma abās valodās. Pirmajā rindā norādīta burtu secība latviešu valodā, bet otrajā – angļu valodā.

Latviešu valodā vārdi, kuru pirmais burts ir bez garumzīmes, mīkstinājuma zīmes vai jumtiņa parasti ir biežāk sastopami nekā tie vārdi, kuru pirmais burts ir ar garumzīmi mīkstinājuma zīmi vai jumtiņu (*skatīt 5. attēlu*). Visbiežāk sastopamais vārdu pirmais burts latviešu valodā ir "p", savukārt angļu valodā – "s" (*skatīt 6. attēlu*). Angļu valodā ir lielāka daudzveidība vārdu pirmajiem burtiem.



Attēls 7: Vārda pēdējā burta biežuma latviešu un angļu valodās histogramma. Burti ir sakārtoti pēc alfabēta.



Attēls 8: Vārda pēdējā burta biežuma latviešu un angļu valodās histogramma. Burti sakārtoti pēc to biežuma abās valodās. Pirmajā rindā norādīta burtu secība latviešu valodā, bet otrajā – angļu valodā.

	Procenti latviešu valodā			**Procenti angļu valodā**
S	46,47%		E	18,40%
T	24,06%		S	11,18%
A	15,53%		N	9,35%
E	10,02%		Y	9,23%
Citi	3,91%		*Citi*	51,84%

Tabula 2: Tabulā attēlots cik procentu no visiem vārdnīcas vārdiem beidzas ar visbiežāk lietotajiem burtiem.

Latviešu valodā 96,09% visu vārdu nenoteiksmē beidzas ar "s", "t", "a" vai "e". Tikai 3,91% vārdu latviešu valodā beidzas ar kādu citu burtu.

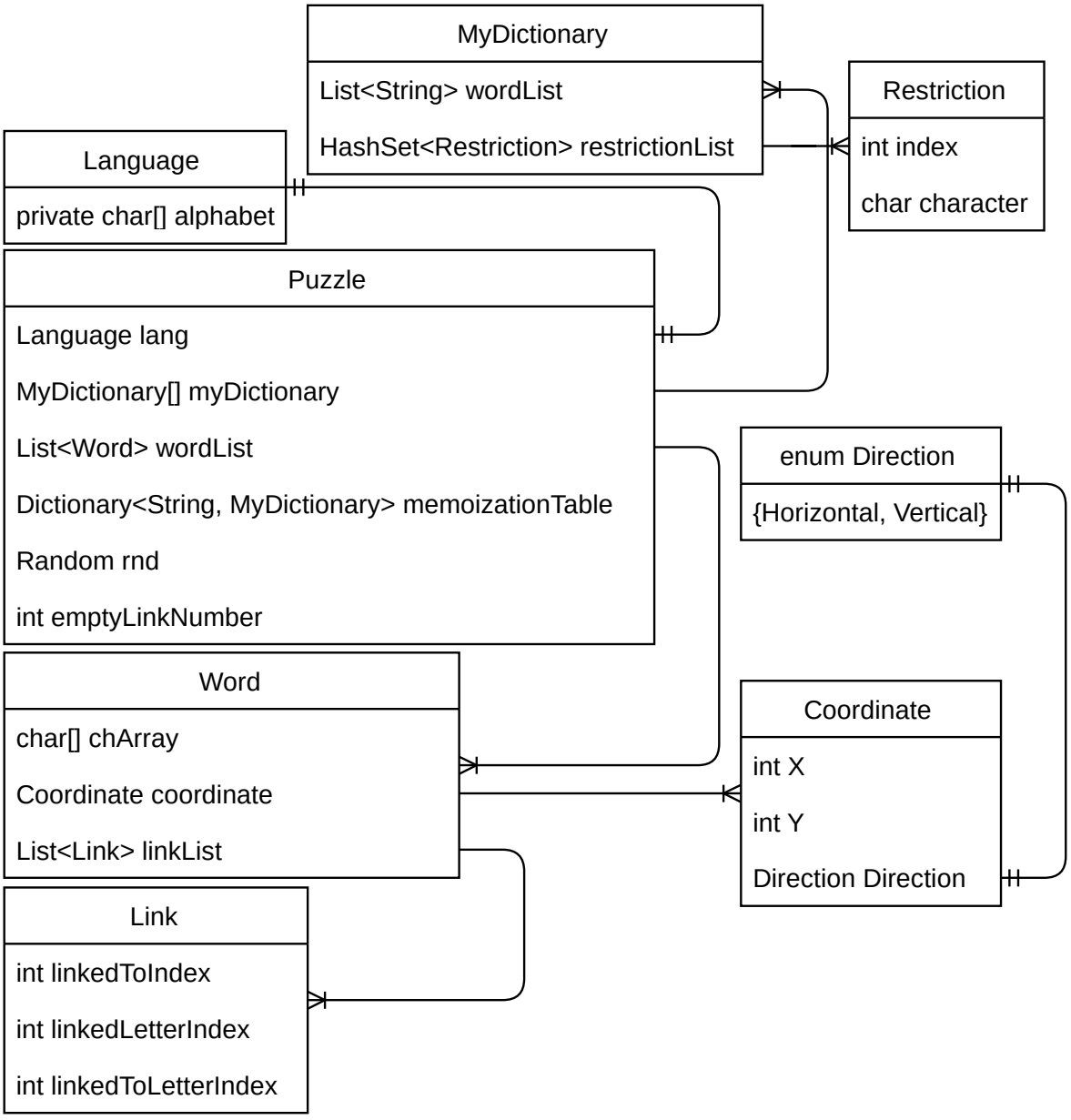
Latviešu valodai ir raksturīga vārdu uzbūve, kas satur vārdu priedēkli, sakni, piedēkli un galotni. Vārdu galotnes lielākajai daļai latviešu valodas lietvārdu un darbības vārdu nenoteiksmē ir *-s*, *-š*, *-us*, *-is*, *-a*, *-e*, *-t* vai *-ties*. Šī iemesla dēļ, nepārprotami, visbiežāk ir sastopami vardi, kuru pēdējais burts ir "s" (*skatīt 7. un 8. attēlu*).

Angļu valodā, savukārt, nav vārdu struktūras un vārda galotņu. Šī iemesla dēļ angļu valodā ir daudz lielāka daudzveidība vārdu pēdējiem burtiem (*skatīt 2. tabulu*). Kaut arī burti "e", "s", "n" un "y" ir biežākie vārda pēdējie burti, to īpatsvars ir tikai 48,16% no visiem vārdiem angļu valodā.

5. DATORPROGRAMMA

5.1. Datorprogrammas vispārīgs apraksts

Tika izstrādāta datorprogramma, kurā tika pielietota “Vārdu pa vārdam” instancēšanas metode, kā arī kombinētā metode. Kombinētajā metodē tiek izmantota “Burtu pa burtam” instancēšanas metode visos krustpunktos, savukārt pēc tam, kad tiek ierakstīti visi burti krustpunktos, režģī tiek ierakstīti visi vārdi pēc gadījuma principa.



Attēls 9: Programmas klašu diagramma.

Programma rakstīta C# programmēšanas valodā. Tā satur mīklas (*Puzzle*), vārdnīcas (*MyDictionary*), valodas (*Language*), koordinātu (ieskaitot virziena, *Coordinate*), vārda (*Word*) un krustpunktu saišu (*Link*) klases, kā arī burtu ierobežojumu (aizpildīto burtu saraksta, *Restriction*) struktūru un virziena (Direction) numurēts tips (*skatīt 9. attēlu*).

```
static void mainFunction(String lang, int gridNumber, String type, int TIMEOUT) {
  Puzzle p = new Puzzle("..\\..\\"+lang+"dic.txt", lang);
  if (gridNumber == 1) p.grid1();
  else if (gridNumber == 2) p.grid2();
  else if (gridNumber == 3) p.grid3();
  else p.grid4();

  int timer = 0;
  if (type == "WordByWord") {
    while (p.EmptyFieldExists() && timer < TIMEOUT) {
      p.BacktrackWords();
      for (int i = 0; i < p.Count(); i++) p.insertRandom(i);
      timer++;
    }
  }
  else {
    while (p.EmptyLinkExists() && timer < TIMEOUT) {
      p.BacktrackWords();
      p.insertRandomCrosspoint(p.GetWordIndexWithMostLinksNotComplete(), type);
      timer++;
    }
    for (int i = 0; i < p.Count(); i++) p.insertRandom(i);
  }
  bool checkWords = true;
  for (int i = 0; i < p.Count(); i++) {
    if (!p.CheckWord(p.GetWord(i).GetString())) {
      checkWords = false;
      break;
    }
  }
  p.print(19, 19);
}
```

Pirmkods 1: Algoritma izsaukšanas funkcijas pirmkods.

Programmas sākumā tiek izveidots jauns *Puzzle* klases objekts "p" un objekta vārdnīcu *myDictionary* masīvā tiek ievietoti vārdi no teksta faila. Pēc tam tiek izveidots režģis izmantojot *Add* funkciju vai jau iepriekš sagatavotu režģi. Tiek izveidots skaitītājs *timer*, kas skaita cik reizes algoritms ticis izmantots. Ja *timer* lielums pārsniedz *TIMEOUT* skaitu vai režģī nav nevienas neaizpildītas ailes, programma beidz darbu (*skatīt 1. pirmkodu*).

Tiek izveidots režģis. Vārdi tiek izveidoti kā Word klases objekti. Šiem vārdiem vajadzīgi trīs mainīgie (vārda garums, x-koordināta, y-koordināta) un virziens (horizontāls vai vertikāls).

Izmantojot *Puzzle* klases funkciju *Add*, vārds tiek saglabāts *Puzzle* klasē, *Word* klases sarakstā *wordList*. Šajā momentā tiek pārbaudīts, vai vārds krustojas ar kādu citu vārdu un, ja tas krustojas, tad tas tiek atzīmēts abu vārdu *Link* sarakstos, izmantojot funkciju *Checkforlinks*.

```
public bool insertRandomCrosspoint(int index, String type) {
  int crosspoint = wordList[index].selectRandomIncompletCrosspoint(rnd);
  if (crosspoint == -1) return false;
  int w1possibleLetters;
  int w2possibleLetters;
  int[] intArray = new int[lang.Length()];
  HashSet<Restriction> res1, res2;
  for (int j = 0; j < lang.Length(); j++) {
    res1 = GetRestrictions(index);
    res1.Add(
      new Restriction(
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(j)
      )
    );
    w1possibleLetters = newDictionaryWithMemory(
      wordList[index].GetL(),
      res1
    ).Count();
    res2 = GetRestrictions(wordList[index].GetLink(crosspoint).linkedToIndex);
    res2.Add(
      new Restriction(
        wordList[index].GetLink(crosspoint).linkedToLetterIndex,
        lang.GetLetter(j)
      )
    );
    w2possibleLetters = newDictionaryWithMemory(
      wordList[wordList[index].GetLink(crosspoint).linkedToIndex].GetL(),
      res2
    ).Count();
    intArray[j] = Math.Min(w1possibleLetters, w2possibleLetters);
  }
  if (intArray.Sum() > 0) {
    if (type == "popular") {
      setLetter(
        index,
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(mostPopularFromintArray(intArray))
      );
    }
    else if (type == "weighted") {
      setLetter(
        index,
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(weightedFromArray(intArray))
      );
    }
    else if (type == "random") {
      setLetter(
        index,
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(randomFromArray(intArray))
      );
    }
```

```
    else return false;
    emptyLinkNumber--;
    return true;
  }
  BacktrackCrosspoint(index, crosspoint);
  return false;
}
```

Pirmkods 2: Burta ievietošanas funkcijas pirmkods.

Programmā tiek izvēlēts gadījuma vārds režģī ar vislielāko neaizpildīto krustpunktu skaitu. Ja režģī ir vairāki neaizpildīti krustpunkti, tad apstrādājamais vārds tiek izvēlēts nejauši no šiem vārdiem. No šī vārda tālāk tiek atlasīts gadījuma krustpunkts. Šajā krustpunktā tiek ierakstīts burts atkarībā no algoritma (*skatīt 5.3. nodaļu un 2. pirmkodu*), izmantojot *mostPopularFromintArray* (pēc burtu beižuma), *weightedFromArray* (pēc svērta gadījuma butiem) un *randomFromArray* (pēc gadījuma burtu) funkcijas.

Tiek aizpildīti visi krustpunkti un pārbaudīts, vai ir iespējams ievietot vārdus. Ja nav, tad attiecīgie burti tiek izdzēsti attiecīgajās ailēs un algoritms turpina likt burtus krustpunktos. Ja ir iespējams ievietot vārdus, tad šie vārdi tiek ievietoti (*skatīt 1. pirmkodu*).

Lai programma neieciklētos, ir nepieciešams izveidot skaitītāju *timer*, kas skaita to, cik reizes ir mēģināts aizpildīt krustvārdu mīklas režģī burtu vai vārdu, izmantojot kādu no algoritmiem. Ja skaitītājs *timer* pārsniedz kādu noteiktu noildzes lielumu, piemēram 500, tad programma pārtrauc darbību un tiek izmests kļūdas paziņojums. Ja programma ir spējīga izveidot krustvārdu mīklu, tad tā tiek izvadīta konsoles logā.

```
public void BacktrackWords() {
  for (int i=0; i<wordList.Count; i++) {
    if (newDictionaryWithMemory(wordList[i].GetL(), GetRestrictions(i)).Count()==0) {
      ClearWord(i);
    }
  }
  setEmptyLinkNumber();
}
```

```
public void BacktrackCrosspoint(int wordindex, int linkIndex) {
  ClearWord(wordindex);
  ClearWord(wordList[wordindex].GetLink(linkIndex).linkedToIndex);
  setEmptyLinkNumber();
}
```

Pirmkods 3: Atkāpšanās funkciju pirmkodi.

Programma satur divas atkāpšanās funkcijas. Pirmā atkāpšanās funkcija *BacktrackCrosspoint* tiek izsaukta, ja krustpuntā nav iespējams ievietot vārdu. Šādā gadījumā tiek izdzēsti abi vārdi, kuri atrodas krustpunktos.

Otro funkciju *BacktrackWords* pielieto pēc visu vārdu ielases. Tiek pārbaudīts, vai visi vārdi, kuri ievietoti režģī, ir atbilstoši. Ja nav, tad vārds tiek izdzēsts un turpinās algoritma darbība, ja visi ievietotie vārdi ir atbilstoši, tad programma beidz darbu (*skatīt 3. pirmkodu*).

5.2. Pielietotās vārdnīcas

Programmā tika pielietota *Hunspell* gramatikas pārbaudes vārdnīcas angļu un latviešu valodās. Tās ir plašas vārdnīcas. No šīm vārdnīcām programmā netika ielasīti tie vārdi, kuriem ir nestandarta rakstu zīmes, piemēram, vārdi "café" un "don't". Visi burti vārdnīcā tika kapitalizēti – pārveidoti no mazajiem uz lielajiem burtiem.

Vārdnīca tiek implementēta klasē *MyDictionary,* kā arī klases *Puzzle* funkcijā *newDictionaryWithMemory*.

Programmas sākumā vārdi tiek nolasīti no vārdu saraksta teksta datnē *lv.txt* vai *en.txt*. Šie vārdu saraksti tika veidoti balstoties uz *Hunspell* gramatikas pārbaudes vārdnīcām latviešu un britu angļu valodās *[14]*. Atkarībā no burtu skaita vārdā tie tiek ierakstīti *Puzzle* klases masīvā m*yDictionary. MyDictionary* masīva indeksa lielums atbilst visu tajā atrodošos vārdu garumam, tas ir – nultajā nav neviena vārda, pirmajā ir vārdi, kuriem ir viens burts (angļu *a* un *I*), otrajā vārdi ar garumu divi un tā līdz maksimālajam noteiktajam vārda garumam, kas programmā ir noteikts 30.

MyDictionary klasē pastāv arī ierobežojumu (*Restriction*) klases saraksts *restrictionList*. Kopa satur sarakstu ar visiem zināmajiem burtiem un to koordinātēm vārdā.

```
public MyDictionary newDictionaryWithMemory(int wordLength, HashSet<Restriction> res)
{
  String pattern = wordLength + " ";
  List<Restriction> list = new List<Restriction>(res);
  if (res.Count == 0)
  {
    return new MyDictionary(myDictionary[wordLength].GetWords(), res);
  }
  for (int i = 0; i < list.Count; i++) pattern += list[i].toString();
  if (memoizationTable.ContainsKey(pattern)) return memoizationTable[pattern];
  else {
    MyDictionary tmpLeft = new MyDictionary(
      myDictionary[wordLength].GetWords(),
      res.First()
    );
    HashSet<Restriction> tmpRes = res;
    tmpRes.Remove(tmpRes.First());
    MyDictionary tmpRight = newDictionaryWithMemory(wordLength, tmpRes);
    List<String> tmpStringList = new List<String>();
    for (int i = 0; i < tmpLeft.GetWords().Count; i++) {
      if (tmpRight.GetWords().Contains(tmpLeft.GetWords()[i])) {
        tmpStringList.Add(tmpLeft.GetWords()[i]);
      }
    }
    MyDictionary retval = new MyDictionary(tmpStringList, res);
    memoizationTable[pattern] = retval;
    return retval;
  }
}
```

Pirmkods 4: newDictionaryWithMemory funkcijas pirmkods.

No ievaddatiem tiek izveidots unikāls identifikators *pattern*. Šis identifikators ir burtu kopa (*String*), kas sastāv no apskatītā vārda garuma, atstarpes un ierobežojumu sarakstu, kas satur ierobežojuma indeksu, atstarpi un ierobežojuma burtu.

Lai uzlabotu programmatūras veiktspēju, funkcijas *newDictionaryWithMemory* sākumā tiek pārbaudīts, vai jau iepriekš šī funkcija nav pie dotajiem ievaddatiem atgriezusi rezultātu izmantojot iegūto identifikatoru un to salīdzinot ar *memoizationTable* esošajiem identifikatoriem. Ja identifikators sakrīt ar kādu no tabulā *memoizationTable* jau esošajiem, tad rezultāts tiek atgriezts atkārtoti no tabulā saglabātā.

Ja funkcija nav pie dotajiem ievaddatiem iepriekš atgriezusi rezultātus, tad tiek rekursīvi izveidotas divas vārdnīcas – ar tikai pirmo ierobežojumu un ar pārējajiem ierobežojumiem. Ja vārds pastāv abās vārdnīcās, tad beigu vārdnīcā jāpastāv tikai tiem vārdiem, kuri ir abās izveidotajās vārdnīcās. Funkcija pirms rezultāta izsniegšanas to saglabā *memoizationTable* tabulā (*skatīt 4. pirmkodu*).

5.3. Pielietotie algoritmi

Programmā tika apskatīti "Burtu pa burtam" variācijas, kurās kā pirmie burti tiek ievietoti nejaušos krustpunktos secībā no tādiem vārdiem, kurus ir visvairāk neaizpildītu krustojumu, uz tādiem, kuros ir mazāk neaizpildītu vārdu. Pēc visu krustpunktu aizpildes, atlikušajām neaizpildītajām ailēm tiek pielietots "Vārdu pa vārdam" algoritms. Programmā tika apskatīti arī gadījumi, kuros šie algoritmi netika izmantoti, bet tika izmantots parasts "Vārdu pa vārdam" algoritms.

Tiek izvēlēts krustpunkts, kuru ir nepieciešams aizpildīt. Tiek izveidots skaitļu masīvs *intArray* burtu skaita alfabēta garumā. Masīvā tiek ierakstīts potenciālais vārdu skaits, kādu var ierakstīt vertikāli un horizontāli, ja krustpunktā tiek ierakstīts konkrēts burts (nultajā masīva ailē – cik vārdus var ierakstīt, ja krustpunktā atrodas pirmais alfabēta burts, pirmajā – otrais alfabēta burts un tā pēc kārtas līdz pēdējam alfabēta burtam, *skatīt 1. pirmkodu*).

```
int  mostPopularFromintArray(int[] intArray) {
  List<int> intist = new List<int>();
  int max = 0;
  for (int i = 0; i < intArray.Length; i++) {
    if (intArray[i] > max) {
      max = intArray[i];
      intist.Clear();
      intist.Add(i);
    }
    else if (intArray[i] == max) intist.Add(i);
  }
  return intist[rnd.Next(intist.Count)];
}
```

Pirmkods 5: Algoritma "Burtu pa burtam" krustpunktos pēc vārdu biežuma burtu izvēles pirmkods.

Algoritmā "Burtu pa burtam" krustpunktos pēc vārdu biežuma tiek izvēlēts burts, kas atstātu vislielāko potenciālo vārdu daudzumu. Ja ir vairāki burti, kuri atstātu vienādu potenciālo vārdu skaitu, tad burts tiek izvēlēts nejauši starp tiem (*skatīt 5. pirmkodu*).

```
int randomFromArray(int[] intArray) {
  List<int> intist = new List<int>();
  for (int i = 0; i < intArray.Length; i++) {
    if (intArray[i] > 0) intist.Add(i);
  }
  return intist[rnd.Next(intist.Count)];
}
```

Pirmkods 6: Algoritma "Burtu pa burtam" krustpunktos pēc gadījuma burtiem burtu izvēles pirmkods.

Algoritmā "Burtu pa burtam" krustpunktos pēc gadījuma burtiem tiek izvēlēti burti, kas atstātu vismaz vienu potenciālo vārdu daudzumu. No šiem burtiem ierakstāmais burts tiek izvēlēts nejauši (*skatīt 6. pirmkodu*).

```
int weightedFromArray(int[] intArray) {
  int random = rnd.Next(intArray.Sum());
  int sum = intArray.Sum();
  for (int i=0; i<intArray.Length; i++) {
    sum -= intArray[i];
    if (sum <= random) return i;
  }
  return -1;
}
```

Pirmkods 7: Algoritma "Burtu pa burtam" krustpunktos pēc svērta gadījuma burtiem burtu izvēles pirmkods.

Algoritmā "Burtu pa burtam" krustpunktos pēc svērta gadījuma burtiem tiek izvēlēts gadījuma burts, masīva *intArray* skaitļus izmantojot kā atsvarus. Tas tiek darīts sākotnēji iegūstot gadījuma skaitli no 0 līdz visu skaitļu *intArray* masīvā summai. Atņemot no summas vienu pēc otra masīva elementus līdz summa ir vienāda vai mazāka gadījuma skaitlim, iespējams iegūt gadījuma burtu (*skatīt 3. tabulu*).

Burts	Potenciālo vārdu kaits	Aprēķins
		Gadījuma skaitlis = 20 Summa = 55
A	10	55 – 10 = 45 > 20 ⬇
Ā	5	45 – 5 = 40 > 20 ⬇
B	15	40 – 15 = 25 > 20 ⬇
C	**5**	**25 – 5 = 20 =< 20**
Č	20	
...	0	

Tabula 3: Aprēķina tabulas algoritmam "Burtu pa burtam" krustpunktos pēc svērta gadījuma burtiem piemērs. Izvēlētais burts piemērā ir "C".

Statistiski pastāv proporcionāli lielāka varbūtība, ka ierakstītais burts ir burts, kurš satur vislielāko potenciālo vārdu daudzumu (*skatīt 7. pirmkodu*).

```
public void insertRandom(int index) {
  MyDictionary tmpDic = newDictionaryWithMemory(
    wordList[index].GetL(),
    GetRestrictions(index)
  );
  if (tmpDic.Count() > 0) {
    wordList[index].setWord(tmpDic.GetWord(rnd.Next(tmpDic.Count())));
    linksToLetters(index);
  }
}
```

Pirmkods 8: Algoritma "Vārdu pa vārdam" pirmkods.

Algoritmā "Vārdu pa vārdam" tiek izveidota speciāla *MyDictionary* klases vārdnīca *tmpDic* izmantojot *newDictionaryWithMemory* funkciju, kas satur tikai tos vārdus, kurus iespējams ievietot mīklā. Tiek nejauši izvēlēts kāds no šīs vārdnīcas vārdiem un ierakstīts režģī (*skatīt 8. pirmkodu*).

5.4. Pielietotie režģi

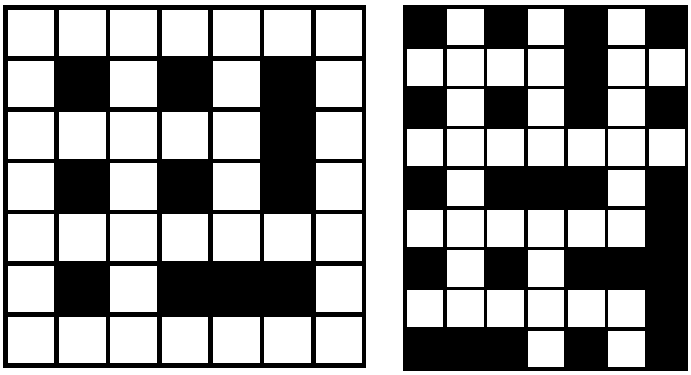
Lai testētu programmu, tika izveidoti četri dažādas sarežģītības krustvārdu mīklu režģi. Šie režģi bija ar atšķirīgu sarežģītības pakāpi – no vienkārša režģa (režģis Nr. 1), kuru iespējams aizpildīt visai viegli uz papīra ar zīmuli, līdz ļoti sarežģītam (režģis Nr. 2 un Nr. 3) un praktiski neiespējami aizpildāmam režģim (režģis Nr. 4).

Lai veidotu režģi nepieciešams izmantot *Puzzle* klases funkciju *Add*. Ar šo funkciju pievieno vārdus vārdu sarakstam režģī. Vārdu pievienošanai nepieciešami četri ievaddati – vārda garums, x-koordināta, y-koordināta, un vārda virzienu.

Pēc vārda izveidošanas, funkcija *Add* apskata visus vārdus, kuri ir pretēja virziena un kuru koordinātas ir apskatāmā vārda robežās. Tiek pārbaudīts vai šie vārdi krustojas un ja tie krustojas, tad tiem krustpunktu saišu (*Link*) sarakstā *linkList* pievieno saiti, kurā atrodas trīs skaitļu mainīgie – vārda ar kuru apskatāmais vārds krustojas indekss (*linkedToIndex*), krustpunkta burta indekss vārdā (linkedLetterIndex) un krustpunkta burta indekss vārdā ar kuru apskatāmais vārds krustojas (*linkedToLetterIndex*).

Attēls 10: Režģis Nr. 1. Vienkārša režģa piemērs.

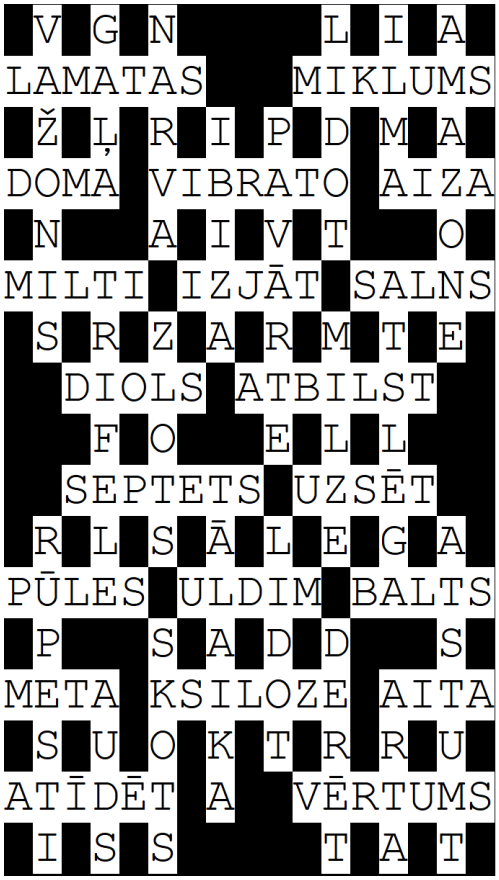
Režģis Nr. 1 *(skatīt 10. attēlu)* ir vienkārša režģa piemērs. Režģa aizpildīšanu iespējams veikt arī neizmantojot algoritmus uz papīra ar zīmuli. Režģis ir 8x7 izmēra. Režģī atrodas 6 vārdi un 7 krustpunkti. Vidēji katram vārdam ir 2,33 krustpunkti.



Attēls 11: Pa kreisi – krustvārdu mīkla ar taisnām malām, pa labi – krustvārdu mīkla ar robotu malu. [15]

Krustvārdu mīklu režģus iespējams vērtēt pēc to cik režģim ir taisnas vai robotas malas. Krustvārdu mīklu režģi ar taisnu malu ir tādi režģi, kuros visi vai liela daļa aiļu pirmajā un pēdējā rindā, kā arī pirmajā un pēdējā kolonnā ir balti. Krustvārdu mīklu režģi ar robotu malu ir tādi režģi, kuros lielākā daļa aiļu pirmajā un pēdējā rindā, kā arī pirmajā un pēdējā kolonnā ir melni.

Programmā tika apskatīti arī režģi gan ar izteikti robotu malu, gan ar izteikti taisnu malu.

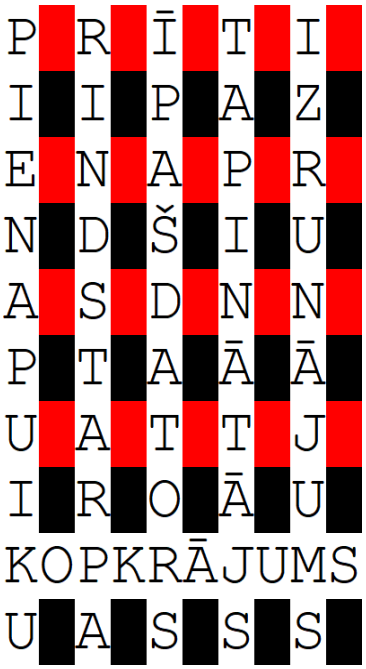


Attēls 12: Režģis Nr. 2. Krustvārdu mīkla ar robotu malu.

Režģis Nr. 2 *(skatīt 12. attēlu)* ir krustvārdu mīklas ar robotu malu piemērs. Režģa aizpildīšana bez programmatūras izmantošanas ir laikietilpīga. Režģis ir 17x17 izmēra. Režģī atrodas 40 vārdi un 50 krustpunkti. Vidēji katram vārdam ir 2,5 krustpunkti.

Attēls 13: Režģis Nr. 3. Krustvārdu mīkla ar taisnu malu.

Režģis Nr. 3 *(skatīt 13. attēlu)* ir krustvārdu mīklas ar taisnu malu piemērs. Režģa aizpildīšana bez programmatūras izmantošanas ir ļoti laikietilpīga. Tajā ir daudz krustpunktu un vārdi, kuri satur tikai krustpunktus. Režģis ir 13x13 izmēra. Režģī atrodas 36 vārdi un 60 krustpunkti. Vidēji katram vārdam ir 3,33 krustpunkti.



Attēls 14: Režģis Nr. 4. Praktiski neiespējama krustvārdu mīkla.

Režģis Nr. 4 *(skatīt 14. attēlu)* ir tāds režģis, kura aizpilde ir praktiski neiespējama. Režģis tika izveidots, lai testētu programmatūras veiktspēju gadījumā, ja atrisinājums nav iegūts. Sarkanie lauki apzīmē laukus, kurus nav iespējams aizpildīt. Režģis ir 10x10 izmēra. Režģī atrodas 10 vārdi un 25 krustpunkti. Katram vārdam ir 5 krustpunkti. Režģi neviens no algoritmiem nebija spējīgs aizpildīt.

REZULTĀTI UN DISKUSIJA

Maģistra darbā tika aprakstītas vairākas pieejas krustvārdu mīklu veidošanas automatizācijai kā arī tika izpētīta literatūra. Lai noskaidrotu, kuras no pieejām ir labākās latviešu valodā veidotajām krustvārdu mīklām, ir nepieciešama tālāka izpēte. Izstrādājot maģistra darbu, šajā darbā apskatītās pieejas un to algoritmi tika implementēti datorprogrammā, kā arī tika aplūkota to veiktspēja un pielietojums.

Pēc programmas izveides katrs no pārbaudītajiem algoritmiem tika pārbaudīts 100 reizes katram režģim gan latviešu, gan angļu valodās. Katras izpildes rezultāts tika ierakstīts teksta datnē un pēc tam rezultāti tika apkopoti izmantojot izklājlapu programmatūru *Microsoft Office Excel*. Šajā nodaļā aplūkotas programmu veiktspējas tabulas.

Katram no implementētajiem algoritmam (*skatīt 5.3. nodaļu*) un katram no krustvārdu mīklu režģiem (*skatīt 5.4. nodaļu*) tika aprēķināts veiksmīgu iznākumu īpatsvars, vidējais laiks, kādu patērēja veiksmīga iznākuma gadījumā, visīsākais, visgarākais un vidējais laiks, kādu programma patērēja, lai darbotos. Šo eksperimentu veica, izmantojot plašu vārdnīcu gan angļu, gan latviešu valodās.

Tabulās ar vārdu "Veiksmīgi iznākumi" apzīmēts gadījums, kad programma ir izpildījusi savu darbu un ir iegūts aizpildīts krustvārdu mīklu režģis. Veiksmīgu iznākumu īpatsvars tiek aprēķināts izdalot to cik reižu programma ir ieguvusi aizpildītu krustvārdu mīklu režģi ar visu mēģinājumu aizpildīt krustvārdu mīklu režģi skaitu.

Tabulās ar vārdu "Skaitītājs" apzīmēts tas, cik reizes ir tikusi izsaukta burta vai vārda ievietošanas funkcija. Lai novērstu programmas ieciklēšanos, skaitītājs nevar būt lielāks par noteiktu skaitli, šajā gadījumā 500. Ja tas notiek, tad programma pārtrauc darbību un tiek izsaukts kļūdas paziņojums. Šāds gadījums netiek uzskatīts par veiksmīgu iznākumu.

Valoda	Režģis	Veiksmīgi iznākumi			Visi iznākumi		
		Īpatsvars	Skaitītājs	Vidējais laiks	Minimālais laiks	Maksimālais laiks	Vidējais laiks
Latv.	1	67%	1,37	00:00,131	00:00,078	00:00,529	00:00,181
	2	0%			00:00,638	00:01,437	00:01,003
	3	0%			00:00,375	00:00,839	00:00,512
	4	0%			00:00,422	00:01,142	00:00,658
Angļu	1	89%	1,09	00:00,090	00:00,047	00:00,342	00:00,104
	2	0%			00:00,562	00:01,151	00:00,746
	3	0%			00:00,437	00:01,019	00:00,631
	4	0%			00:00,172	00:00,365	00:00,216

Tabula 4: Algoritma "Vārdu pa vārdam" veiksmīgu iznākumu īpatsvars un laiks kādu algoritma darbība patērēja.

Kaut arī algoritms "Vārdu pa vārdam" ir ātrs, tas nav precīzs un ir pielietojams tikai ļoti vienkāršās krustvārdu mīklās (*skatīt 4. tabulu*). Algoritmu bez advancētas atkāpšanās funkcijas vai bez mācīšanās algoritma implementācijas nav lietderīgi izmantot.

Valoda	Režģis	Veiksmīgi iznākumi		Visi iznākumi		
		Īpatsvars	Skaitītājs	Minimālais laiks	Maksimālais laiks	Vidējais laiks
Latv.	1	100%	7	00:00,859	00:01,600	00:01,074
	2	100%	50,4	00:04,672	00:08,000	00:05,740
	3	100%	108,85	00:01,484	00:02,808	00:01,926
	4	0%		00:15,999	00:27,560	00:18,239
Angļu	1	100%	7	00:01,141	00:01,880	00:01,310
	2	100%	50	00:03,797	00:06,219	00:04,331
	3	100%	82,13	00:02,203	00:05,015	00:02,713
	4	0%		00:05,547	00:09,141	00:06,373

Tabula 5: Algoritma "Burtu pa burtam" krustpunktos pēc vārdu biežuma, kam seko "Vārdu pa vārdam" pārējās ailēs veiksmīgu iznākumu īpatsvars un laiks kādu algoritma darbība patērēja.

Algoritms ļoti ir ļoti efektīvs un ātrs. Izmantojot šo algoritmu iespējams visefektīvāk iegūt rezultātu salīdzinot ar citiem no pārbaudītajiem algoritmiem. Tomēr jāpiemin, ka vārdi, kurus šis algoritms ir spējīgs atrast nav daudzveidīgi un vienam un tam pašam režģim algoritmu pielietojot vairākkārtīgi, iegūtie rezultāti atkārtojas.

Algoritms ir spējīgs aizpildīt krustvārdu mīklu režģi ātrāk angļu valodā nekā latviešu valodā, kā arī ātrāk režģim ar taisnu malu, nekā režģim ar robotu malu. Algoritms izpildās ļoti ātri un ar minimālu soļu skaitu vienkāršam režģim.Algoritmam nav nepieciešama sarežģīta atkāpšanās funkcija (*skatīt 5. tabulu*).

Valoda	Režģis	Veiksmīgi iznākumi			Visi iznākumi		
		Īpatsvars	Skaitītājs	Vidējais laiks	Minimālais laiks	Maksimālais laiks	Vidējais laiks
Latv.	1	100%	10,2	00:01,084	00:00,826	00:02,043	00:01,084
	2	100%	76,85	00:06,253	00:04,583	00:10,893	00:06,253
	3	14%	360,29	00:03,410	00:02,653	00:05,495	00:03,859
	4	0%			00:16,757	00:27,274	00:19,629
Angļu	1	100%	7,98	00:01,366	00:01,170	00:03,221	00:01,366
	2	100%	59,89	00:04,270	00:03,441	00:06,558	00:04,270
	3	71%	332,11	00:05,164	00:03,49	00:07,617	00:05,477
	4	0%			00:05,939	00:10,126	00:06,796

Tabula 6: Algoritma "Burtu pa burtam" krustpunktos pēc gadījuma burtiem, kam seko "Vārdu pa vārdam" pārējās ailēs veiksmīgu iznākumu īpatsvars un laiks kādu algoritma darbība patērēja.

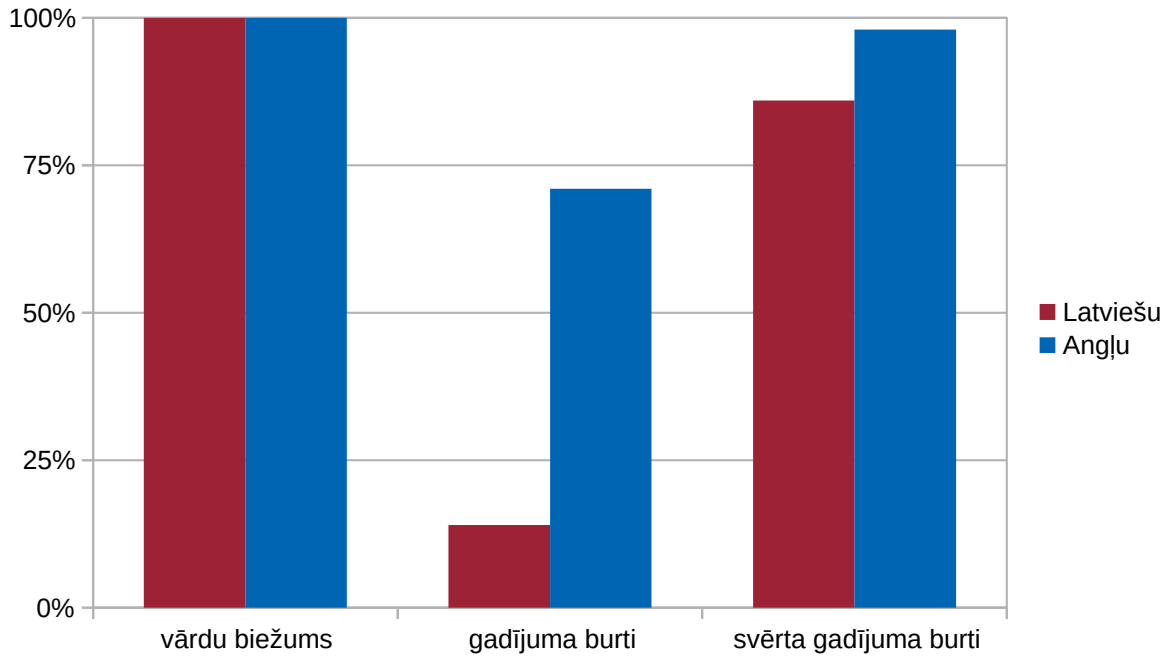
Kaut gan algoritms ir lēnāks par "Burtu pa burtam" krustpunktos pēc vārdu biežuma un tas ir mazāk efektīvs, algoritms ir spējīgs atrast daudzveidīgus risinājumus krustvārdu mīklām (*skatīt 6. tabulu*). Pie sarežģītiem krustvārdu mīklu režģiem (*skatīt 3. režģi*) algoritms nav spējīgs atrast atrisinājumus. Algoritms ir efektīvāks angļu valodas vārdiem nekā latviešu valodas vārdiem.

Valoda	Režģis	Veiksmīgi iznākumi			Visi iznākumi		
		Īpatsvars	Skaitītājs	Vidējais laiks	Minimālais laiks	Maksimālais laiks	Vidējais laiks
Latv.	1	100%	7,25	00:01,022	00:00,803	00:02,369	00:01,022
	2	100%	56,07	00:05,499	00:04,156	00:09,119	00:05,499
	3	86%	269,47	00:02,748	00:01,734	00:04,989	00:02,912
	4	0%			00:16,015	00:26,220	00:18,164
Angļu	1	100%	7,05	00:01,255	00:01,062	00:02,105	00:01,255
	2	100%	51,87	00:04,099	00:03,521	00:05,126	00:04,099
	3	98%	227,88	00:04,064	00:02,500	00:08,540	00:04,125
	4	0%			00:05,765	00:08,975	00:06,373

Tabula 7: Algoritma "Burtu pa burtam" krustpunktos pēc svērta gadījuma burtiem, kam seko "Vārdu pa vārdam" pārējās ailēs veiksmīgu iznākumu īpatsvars un laiks kādu algoritma darbība patērēja.

Kaut gan algoritms "Burtu pa burtam" krustpunktos pēc gadījuma burtiem ir nedaudz lēnāks par algoritmu "Burtu pa burtam" krustpunktos pēc vārdu biežuma un tas ir mazāk efektīvs par šo algoritmu, algoritms ir spējīgs atrast daudzveidīgus risinājumus krustvārdu mīklām salīdzinoši īsā laika posmā (*skatīt 7. tabulu*). Algoritms ir spējīgs atrast atrisinājumus sarežģītiem režģiem biežāk nekā algoritms "Burtu pa burtam" krustpunktos pēc gadījuma burtiem.

Algoritmu ir iespējams optimizēt, izveidojot efektīvāku atkāpšanās funkciju vai pielietojot mācīšanās algoritmu. Algoritms ir efektīvāks angļu valodas vārdiem nekā latviešu valodas vārdiem.

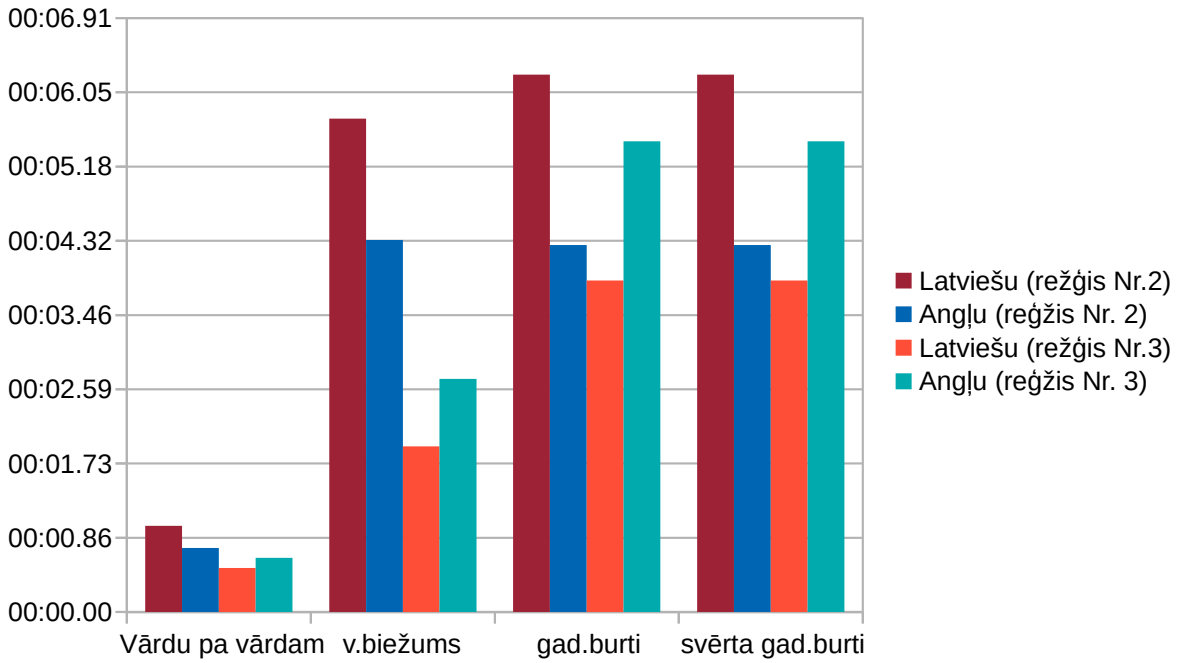


Attēls 15: Algoritmu veiksmīgu iznākumu īpatsvars angļu un latviešu valodās režģim Nr. 3.

Režģis Nr. 1 un Nr. 2 izpildās 100% gadījumu, savukārt režģis Nr. 4 izpildās 0% gadījumu. 15. attēlā redzams veiksmīgu iznākumu īpatsvaru latviešu un angļu valodās "Burtu pa burtam" krustpunktos pēc vārda biežuma, gadījuma burtiem un svērta gadījuma burtiem algoritmiem.

Režģis Nr. 3 ir vissarežģītāk aizpildāmais režģis. Ne visi algoritmi ir spējīgi to aizpildīt. Algoritms "Burtu pa burtam" krustpunktos pēc vārdu biežuma ir visefektīvākais un tas spēj to aizpildīt 100% gadījumu. Citi algoritmi nav spējīgi to aizpildīt. Algoritms "Burtu pa burtam" krustpunktos pēc gadījuma burtiem ir visneefektīvākais no apskatītajiem algoritmiem.

Angļu valodā visi no algoritmiem ir spējīgi atrast atrisinājumu ar lielāku veiksmīgu iznākumu īpatsvaru. Algoritms "Burtu pa burtam" krustpunktos pēc gadījuma burtiem ir ar vislielāko veiktspējas atšķirību starp latviešu un angļu valodām (*skatīt 15. attēlu*).



16. attēls: Vidējais algoritma izpildes laiks

16. attēlā ir attēlots vidējais algoritma izpildes laiks. Jo sarežģītāks izmantotais algoritms, jo ilgāks programmas izpildes laiks.

"Vārdu pa vārdam" algoritms ir ātrs, bet tas nav spējīgs atrast vajadzīgo rezultātu. Kaut arī algoritmi veic vairāk darbību režģī ar taisnu malu nekā ar robotu, latviešu valodā šīs darbības izpildās ātrāk režģī ar taisnu malu. Iespējams tas notiek tādēļ, ka latviešu valodā ir daudz mazāka daudzveidība vārdu pēdējos burtos (*skatīt 2. tabulu*) un līdz ar to ir nepieciešams apskatīt daudz mazāku vārdu grupu labajai un apakšējai režģa malai.

Algoritms "Burtu pa burtam" pēc vārdu biežuma ir ātrāks par citiem algoritmiem, bet tas nav spējīgs atrast risinājumus. Angļu valodā režģim ar robotu malu visi pārbaudītie "Burtu pa burtam" algoritmi strādā ar aptuveni vienādu veiktspēju un līdzvērtīgā laikā.

"Burtu pa burtam" pēc gadījuma burtiem un pēc svērta gadījuma burtiem algoritmi darbojas aptuveni vienlīdzīgos laikos, bet "Burtu pa burtam" pēc svērta gadījuma burtiem algoritmam ir daudz augstāks veiksmīgu iznākumu īpatsvars nekā "Burtu pa burtam" pēc gadījuma burtiem algoritmam.

SECINĀJUMI

Krustvārdu mīklu manuāla veidošana ir laikietilpīgs process, ir nepieciešams izveidot datorprogrammu, kas veic krustvārdu mīklu veidošanas automatizāciju.

Tā kā latviešu un angļu valodā atšķiras alfabēta burtu skaits, burtu biežums, pirmo burtu un pēdējo burtu biežums, tiek izvirzīta hipotēze – krustvārdu mīklu veidošanas automatizācijas algoritmu ātrums un noderīgums atšķiras dažādām valodām. Sarežģītu krustvārdu mīklu piemēri norāda uz šīs hipotēzes ticamību. Visi algoritmi bija efektīvāki angļu valodā nekā latviešu valodā (*skatīt 15. attēlu*).

Ņemot vērā, ka latviešu valodā ir ļoti limitējošs vārdu pēdējo burtu daudzums – 96.08% no vārdiem to pamatformā beidzas ar burtiem *s*, *t*, *a* vai *e*, un, lai gan šie burti tiek bieži izmantoti, no šiem burtiem ir grūti izveidot vārdus krustvārdu mīklās tā, lai ar tiem krustojošie vārdi varētu krustoties arī ar citiem vārdiem. Šī iemesla dēļ latviešu valodā, atšķirībā no angļu valodas, daudz biežāk krustvārdu mīklām jābūt ar robotām malām, nevis taisnām (*skatīt 9. attēlu*). Darba rezultāti norāda uz faktu, ka latviešu valodā izveidot krustvārdu mīklas ar roboto malu ir vienkāršāk nekā ar taisnu malu, bet šī darbība aizņem vairāk laika. Visi pārbaudītie "Burtu pa burtam" algoritmi bija spējīgi izveidot krustvārdu mīklu režģī ar robotu malu.

Kaut arī efektīvākais algoritms, kas tika pārbaudīts ir "Burtu pa burtam" krustpunktos pēc vārdu biežuma, tā kā tas nedod daudzveidīgus atrisinājumus, tad lietderīgāk ir izmantot mazāk efektīvo algoritmu "Burtu pa burtam" krustpunktos pēc svērta gadījuma burtiem.

Kaut arī algoritmi veic vairāk darbību režģī ar taisnu malu nekā ar robotu, latviešu valodā šīs darbības izpildās ātrāk režģī ar taisnu malu. Iespējams tas notiek tādēļ, ka latviešu valodā ir daudz mazāka daudzveidība vārdu pēdējos burtos (*skatīt 2. tabulu*) un līdz ar to ir nepieciešams apskatīt daudz mazāku vārdu grupu labajai un apakšējai režģa malai. Ir nepieciešami tālāki pētījumi šīs hipotēzes pārbaudei.

Algoritms "Burtu pa burtam" krustpunktos pēc gadījuma burtiem ir efektīvs tikai vienkāršās krustvārdu mīklās, bet neefektīvs sarežģītās.

Kaut arī krustvārdu mīklu veidošanas automatizācija ir sarežģīts un laikietilpīgs process, tas ir noderīgs, jo šo laiku nepieciešams patērēt vienreiz un nevis katru reizi, kad nepieciešams veidot krustvārdu mīklu.

Ir nepieciešami turpmāki pētījumi, kā arī nepieciešams izveidot algoritmu krustvārdu mīklu režģu veidošanas automatizācijai. Tāpat nepieciešams izveidot režģu rediģēšanas funkciju un iespēju automātiski ievietot mīklu avīzē vai mājaslapā. Ar *WordNet* vārdnīcas implementāciju var vienkāršot jautājumu sastādīšanu.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] Constructing Crossword Grids: Use of Heuristics vs Constraints - Gary Meehanand un Peter Gray – University of Aberdeen. – Aberdīna, 1997. Pieejams: http://gtoal.com/scrabble/meehan/cross.pdf

[2] PRACTICAL CROSSWORD GENERATION WITH CHECKPOINT SEARCH – Ariel Arbiser – University of Buenos Aires – Buenosairesa, 2005. Pieejams: https://pdfs.semanticscholar.org/a45e/6a1ec33a9e8eee1c70b907aa08cb55ef1a1d.pdf

[3] Automated Generation of Unconstrained Crossword Puzzles and an Estimate of Their Solution Space – Zafer Barutçuoğlu – Boğaziçi University – Stambula. Pieejams: https://pdfs.semanticscholar.org/714a/fe3a94f684f828047f0818899006652efcb7.pdf

[4] CROSSWORD PUZZLES: EXPERIMENTS WITH META-SEARCH IN PROPOSITIONAL REASONING – JAMES J. LU, JEFFREY S. ROSENTHAL, ANDREW E. SHAFFE – Bucknell University, University of Toronto, Cornell University – Levisburga, Toronto, Itaka. Pieejams: https://www.cs.cornell.edu/boom/2002sp/extproj/www.people.cornell.edu/pages/aes47/research/nd.pdf

[5] Wordnet Enhanced Automatic Crossword Generation Aoife Aherne, Carl Vogel – University of Dublin – Dublina. Pieejams: https://www.scss.tcd.ie/disciplines/intelligent_systems/clg/clg_web/ahernevogel06.pdf

[6] Crossing WordNet with Crosswords, Netting Enhanced Automatic Crossword Generation – Aoife Aherne, Carl Vogel – Trinity College – Dublina. Pieejams: https://pdfs.semanticscholar.org/4e9b/de3a44e33a3154cff02a336888178b9acc00.pdf

Interneta resursi

[7] http://www.lpia.lv/?id=191&izd=2&izdid=25

[8] https://www.crosswordtournament.com/more/wynne.html

[9] https://apollo.tvnet.lv/5842788/esam-krustvardu-miklu-lielvalsts

[10] https://skaties.lv/izklaide/raibumi/donaudampfschiffahrtselektrizitatenhauptbetriebswerk-bauunterbeamtengesellschaft-kas-ir-eiropas-valodu-garakie-vardi/

[11] https://lv.wikipedia.org/wiki/Krustvārdu_mīkla

[12] https://stackoverflow.com/questions/943113/algorithm-to-generate-a-crossword

[13] https://en.wikipedia.org/wiki/WordNet

[14] https://github.com/elastic/hunspell/tree/master/dicts

[15] https://www.crosswordsite.com

PIELIKUMI

1. pielikums. Vārdu garums latviešu un angļu valodās

Garums	**Latviešu**	**Angļu**
1	2	32
2	54	267
3	160	1125
4	1174	2761
5	2775	4357
6	5162	6054
7	8092	6765
8	9842	6870
9	9298	6372
10	8141	4883
11	6535	3405
12	4752	2244
13	3342	1497
14	2360	881
15	1580	490
16	955	274
17	665	124
18	356	58
19	275	29
20	146	12
21	96	3
22	52	1
23	31	0
24	27	0
25	11	0
26	9	0
27	4	0
28	3	1
29	4	0
30	1	0
31	0	0
32	1	0
45	0	1
Kopā:	**65905**	**48505**

2. pielikums. Burtu skaits vārdos latviešu un angļu valodās

Burts	**Latviešu**			**Angļu**		
	Kopā	Vārda sākumā	Vārda beigās	Kopā	Vārda sākumā	Vārda beigās
A	39125	7402	10235	26164	3155	1666
Ā	18711	206	46	0	0	0
B	10148	2300	8	7846	3012	205
C	6838	873	4	14567	4698	1008
Č	670	278	0	0	0	0
D	16673	3278	39	11413	2557	3326
E	31738	1228	6600	30690	2012	8904
Ē	13015	94	33	0	0	0
F	3272	925	0	5227	2161	274
G	10098	1470	6	8375	1667	1844
Ģ	1251	196	0	0	0	0
H	1863	864	6	9524	2051	1198
I	39500	4876	697	24511	1763	457
Ī	12420	96	18	0	0	0
J	11244	691	5	783	511	7
K	21726	4060	31	3709	559	939
Ķ	2584	217	0	0	0	0
L	22530	1814	12	17979	1713	2589
Ļ	2064	46	18	0	0	0
M	18789	3040	211	10450	3007	1423
N	24857	3693	57	21971	1441	4524
Ņ	2236	54	6	0	0	0
O	22153	569	204	21323	1186	756
P	21011	10548	15	9963	3684	702
Q	0	0	0	720	236	13
R	30271	1842	32	23836	2117	3889
S	43777	6847	30617	19772	5338	5410
Š	4999	640	310	0	0	0
T	40539	2514	15852	21834	2435	3893
U	19037	1764	740	12164	784	156
Ū	3067	135	4	0	0	0
V	10715	2393	6	3997	818	50
W	6	1	0	3691	1302	326
X	1	0	1	1198	35	242
Y	1	1	0	6986	168	4466
Z	13224	784	66	889	90	130
Ž	1595	166	1	0	0	0

3. pielikums. Pirmkods

Program.cs

```
static void Main(string[] args) {
  Console.BackgroundColor = ConsoleColor.Black;
  Console.ForegroundColor = ConsoleColor.White;
  Console.OutputEncoding = System.Text.Encoding.Unicode;

  mainFunction("lv", 3, "weighted", 500);
  Console.ReadLine();
}

static void mainFunction(String lang, int gridNumber, String type, int TIMEOUT) {
  DateTime start = DateTime.Now;
  Puzzle p = new Puzzle("..\\..\\"+lang+"dic.txt", lang);
  if (gridNumber == 1) p.grid1();
  else if (gridNumber == 2) p.grid2();
  else if (gridNumber == 3) p.grid3();
  else p.grid4();

  int timer = 0;
  if (type == "WordByWord") {
    while (p.EmptyFieldExists() && timer < TIMEOUT) {
      p.BacktrackWords();
      for (int i = 0; i < p.Count(); i++) p.insertRandom(i);
      timer++;
    }
  }
  else {
    while (p.EmptyLinkExists() && timer < TIMEOUT) {
      p.BacktrackWords();
      p.insertRandomCrosspoint(p.GetWordIndexWithMostLinksNotComplete(), type);
      timer++;
    }
    for (int i = 0; i < p.Count(); i++) p.insertRandom(i);
  }
  bool checkWords = true;
  for (int i = 0; i < p.Count(); i++) {
    if (!p.CheckWord(p.GetWord(i).GetString())) {
      checkWords = false;
      break;
    }
  }
  p.print(19, 19);
}
```

Puzzle.cs

```
readonly int MAX_LEN = 30;
Language lang;
MyDictionary[] myDictionary;
List<Word> wordList;
private Dictionary<String, MyDictionary> memoizationTable;
Random rnd = new Random();
int emptyLinkNumber;

public Puzzle(string file, String str) {
  lang = new Language();
  if (str == "en") lang.setEnglish();
```

```
  memoizationTable = new Dictionary < String, MyDictionary > ();
  try {
    System.IO.StreamReader reader = new System.IO.StreamReader(
      file,
      System.Text.Encoding.UTF8
    );
    myDictionary = new MyDictionary[MAX_LEN];
    String line;
    for (int i = 0; i < MAX_LEN; i++) myDictionary[i] = new MyDictionary();
    while ((line = reader.ReadLine()) != null) {
    if (line.Length < MAX_LEN && lang.isAcceptable(line.ToUpper())) {
      myDictionary[line.Length].Add(line.ToUpper());
    }
  }
  catch (System.IO.IOException e) {
    Console.WriteLine("The file could not be read:");
    Console.WriteLine(e.Message);
  }
  wordList = new List<Word>();
  emptyLinkNumber = 0;
}
public void Add(Word word) {
  wordList.Add(word);
  Checkforlinks(wordList.Count - 1);
}
public int Count() { return wordList.Count; }
public void insertRandom(int index) {
  MyDictionary tmpDic = newDictionaryWithMemory(
    wordList[index].GetL(),
    GetRestrictions(index)
  );
  if (tmpDic.Count() > 0) {
    wordList[index].setWord(tmpDic.GetWord(rnd.Next(tmpDic.Count())));
    linksToLetters(index);
  }
}
public HashSet<Restriction> GetRestrictions(int index) {
  HashSet<Restriction> res = new HashSet<Restriction>();
  for (int j = 0; j < wordList[index].GetL(); j++) {
    if (wordList[index].GetLetter(j) != '_') {
      res.Add(new Restriction(j, wordList[index].GetLetter(j)));
    }
  }
  return res;
}
public bool insertRandomCrosspoint(int index, String type) {
  int crosspoint = wordList[index].selectRandomIncompletCrosspoint(rnd);
  if (crosspoint == -1) return false;
  int w1possibleLetters;
  int w2possibleLetters;
  int[] intArray = new int[lang.Length()];
  HashSet<Restriction> res1, res2;
  for (int j = 0; j < lang.Length(); j++) {
    res1 = GetRestrictions(index);
    res1.Add(
      new Restriction(
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(j)
      )
    );
    w1possibleLetters = newDictionaryWithMemory(
      wordList[index].GetL(),
      res1
```

```
    ).Count();
    res2 = GetRestrictions(wordList[index].GetLink(crosspoint).linkedToIndex);
    res2.Add(
      new Restriction(
        wordList[index].GetLink(crosspoint).linkedToLetterIndex,
        lang.GetLetter(j)
      )
    );
    w2possibleLetters = newDictionaryWithMemory(
      wordList[wordList[index].GetLink(crosspoint).linkedToIndex].GetL(),
      res2
    ).Count();
    intArray[j] = Math.Min(w1possibleLetters, w2possibleLetters);
  }
  if (intArray.Sum() > 0) {
    if (type == "popular") {
      setLetter(
        index,
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(mostPopularFromintArray(intArray))
      );
    }
    else if (type == "weighted") {
      setLetter(
        index,
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(weightedFromArray(intArray))
      );
    }
    else if (type == "random") {
      setLetter(
        index,
        wordList[index].GetLink(crosspoint).linkedLetterIndex,
        lang.GetLetter(randomFromArray(intArray))
      );
    }
    else return false;
    emptyLinkNumber--;
    return true;
  }
  BacktrackCrosspoint(index, crosspoint);
  return false;
}
int weightedFromArray(int[] intArray) {
  int random = rnd.Next(intArray.Sum());
  int sum = intArray.Sum();
  for (int i=0; i<intArray.Length; i++) {
    sum -= intArray[i];
    if (sum <= random) return i;
  }
  return -1;
}
int randomFromArray(int[] intArray) {
  List<int> intist = new List<int>();
  for (int i = 0; i < intArray.Length; i++) {
    if (intArray[i] > 0) intist.Add(i);
  }
  return intist[rnd.Next(intist.Count)];
}
int mostPopularFromintArray(int[] intArray) {
  List<int> intist = new List<int>();
  int max = 0;
  for (int i = 0; i < intArray.Length; i++) {
```

```
    if (intArray[i] > max) {
      max = intArray[i];
      intist.Clear();
      intist.Add(i);
    }
    else if (intArray[i] == max) intist.Add(i);
  }
  return intist[rnd.Next(intist.Count)];
}
public void setLetter(int index, int charindex, char ch) {
  Link tmp;
  if (index < wordList.Count && charindex < wordList[index].GetL()) {
    wordList[index].setLetter(charindex, ch);
    for (int i=0; i< wordList[index].linkCount(); i++) {
      tmp = wordList[index].GetLink(i);
        if (tmp.linkedLetterIndex == charindex) {
          wordList[tmp.linkedToIndex].setLetter(tmp.linkedToLetterIndex, ch);
          return;
      }
    }
  }
}
public void ClearWord(int index) {
  for (int i=0; i<wordList[index].GetL(); i++) setLetter(index, i, '_');
}
public MyDictionary newDictionaryWithMemory(int wordLength, HashSet<Restriction> res)
{
  String pattern = wordLength + " ";
  List<Restriction> list = new List<Restriction>(res);
  if (res.Count == 0)
  {
    return new MyDictionary(myDictionary[wordLength].GetWords(), res);
  }
  for (int i = 0; i < list.Count; i++) pattern += list[i].toString();
  if (memoizationTable.ContainsKey(pattern)) return memoizationTable[pattern];
  else {
    MyDictionary tmpLeft = new MyDictionary(
      myDictionary[wordLength].GetWords(),
      res.First()
    );
    HashSet<Restriction> tmpRes = res;
    tmpRes.Remove(tmpRes.First());
    MyDictionary tmpRight = newDictionaryWithMemory(wordLength, tmpRes);
    List<String> tmpStringList = new List<String>();
    for (int i = 0; i < tmpLeft.GetWords().Count; i++) {
      if (tmpRight.GetWords().Contains(tmpLeft.GetWords()[i])) {
        tmpStringList.Add(tmpLeft.GetWords()[i]);
      }
    }
    MyDictionary retval = new MyDictionary(tmpStringList, res);
    memoizationTable[pattern] = retval;
    return retval;
  }
}
public void linksToLetters(int index) {
  Link tmp;
  for (int j = 0; j < wordList[index].linkCount(); j++) {
    tmp = wordList[index].GetLink(j);
    wordList[tmp.linkedToIndex].setLetter(
      tmp.linkedToLetterIndex,
      wordList[index].GetLetter(tmp.linkedLetterIndex)
    );
  }
```

```
}
public char[,] toGrid(int x, int y) {
  char[,] tmp = new char[x,y];
  for (int i=0; i<x; i++) {
    for (int j = 0; j < y; j++) tmp[i, j] = ' ';
  }
  Coordinate c;
  char[] word;
  for (int i=0; i<wordList.Count; i++) {
    c = wordList[i].GetCoordinate();
    word = wordList[i].GetChArray();
    if (c.Direction == Direction.Horizontal) {
      for (int j = 0; j < word.Length; j++) {
        try { tmp[c.Y, j + c.X] = word[j]; }
        catch { new IndexOutOfRangeException(); }
      }
    }
    else {
      for (int j = 0; j < word.Length; j++) {
        try { tmp[j + c.Y, c.X] = word[j]; }
        catch { new IndexOutOfRangeException(); }
      }
    }
  }
  return tmp;
}
public char[,] toXGrid(int x, int y) {
  char[,] tmp = new char[x, y];
  for (int i = 0; i < x; i++) {
    for (int j = 0; j < y; j++) tmp[i, j] = ' ';
  }
  Coordinate c;
  char[] word;
  for (int i = 0; i < wordList.Count; i++) {
    c = wordList[i].GetCoordinate();
    word = wordList[i].GetChArray();
    if (c.Direction == Direction.Horizontal && c.Y < y) {
      for (int j = 0; j < word.Length  && j + c.X < x; j++) tmp[c.Y, j + c.X] = 'O';
      for (int j = 0; j < wordList[i].linkCount()
        && wordList[i].GetLink(j).linkedLetterIndex + c.X < x; j++
      ) tmp[c.Y, wordList[i].GetLink(j).linkedLetterIndex + c.X] = 'X';
    }
    else if (c.Direction == Direction.Vertical && c.X < x) {
      for (int j = 0; j < word.Length && j + c.Y < x; j++) tmp[j + c.Y, c.X] = 'O';
      for (int j = 0; j < wordList[i].linkCount()
        && wordList[i].GetLink(j).linkedLetterIndex + c.Y < x; j++
      ) tmp[wordList[i].GetLink(j).linkedLetterIndex + c.Y, c.X] = 'X';
    }
  }
  return tmp;
}
public void print(int x, int y) {
  char[,] tmp = toGrid(x, y);
  for (int i=0; i<x; i++) {
    for (int j = 0; j < y; j++) {
      if (tmp[i, j] == ' ') Console.BackgroundColor = ConsoleColor.Black;
      else if (tmp[i, j] == '_') {
        Console.BackgroundColor = ConsoleColor.Red;
        Console.ForegroundColor = ConsoleColor.Red;
      }
      else {
        Console.BackgroundColor = ConsoleColor.White;
        Console.ForegroundColor = ConsoleColor.Black;
```

```
      }
      Console.Write(tmp[i, j]);
    }
    Console.BackgroundColor = ConsoleColor.Black;
    Console.Write('\n');
  }
  Console.BackgroundColor = ConsoleColor.Black;
  Console.ForegroundColor = ConsoleColor.White;
}
public void Checkforlinks(int index) {
  for (int i = 0; i < wordList.Count; i++) {
    if (i != index && DoTheyCross(wordList[index], wordList[i])) {
      if (wordList[index].GetD() == Direction.Horizontal) {
        wordList[index].Add(
          new Link(
            i,
            wordList[i].GetX()-wordList[index].GetX(),
            wordList[index].GetY() - wordList[i].GetY()
          )
        );
        wordList[i].Add(
          new Link(
            index,
            wordList[index].GetY() - wordList[i].GetY(),
            wordList[i].GetX() - wordList[index].GetX()
          )
        );
      }
      else {
        wordList[i].Add(
          new Link(
            index,
            wordList[index].GetX() - wordList[i].GetX(),
            wordList[i].GetY() - wordList[index].GetY()
          )
       );
       wordList[index].Add(
          new Link(
            i,
            wordList[i].GetY() - wordList[index].GetY(),
            wordList[index].GetX() - wordList[i].GetX()
          )
       );
      }
    }
  }
}
public bool DoTheyCross(Word w1, Word w2) {
  Word wh, wv;
  if (w1.GetD() == Direction.Horizontal && w2.GetD() == Direction.Vertical) {
    wh = w1;
    wv = w2;
  }
  else if (w1.GetD() == Direction.Vertical && w2.GetD() == Direction.Horizontal) {
    wv = w1;
    wh = w2;
  }
  else return false;
  return (
    wh.GetY() >= wv.GetY() && wh.GetY() < wv.GetY() + wv.GetL()
    && wv.GetX() >= wh.GetX() && wv.GetX() < wh.GetX() + wh.GetL()
  );
}
```

```
public Word GetWord(int index) { return wordList[index]; }
public bool CheckWord(string str) { return myDictionary[str.Length].CheckWord(str); }
public int GetWordIndexWithMostLinksNotComplete() {
  int maxLinkCount = 0;
  List<int> tmpList = new List<int>();
  for (int i=0; i<wordList.Count; i++) {
    if (wordList[i].incompleteLinkCount() > maxLinkCount) {
      maxLinkCount = wordList[i].incompleteLinkCount();
      tmpList.Clear();
      tmpList.Add(i);
    }
    else if (wordList[i].incompleteLinkCount() == maxLinkCount) tmpList.Add(i);
  }
  return tmpList[rnd.Next(tmpList.Count)];
}
public void setEmptyLinkNumber() {
  emptyLinkNumber = 0;
  for (int i = 0; i < wordList.Count; i++) {
    emptyLinkNumber += wordList[i].incompleteLinkCount();
  }
  emptyLinkNumber /= 2;
}
public bool EmptyLinkExists() { return emptyLinkNumber > 0; }
public bool EmptyFieldExists() {
  for (int i=0; i<wordList.Count; i++) {
    if (wordList[i].hasEmptyField()) return true;
  }
  return false;
}
public void BacktrackWords() {
  for (int i=0; i<wordList.Count; i++) {
    if (newDictionaryWithMemory(wordList[i].GetL(), GetRestrictions(i)).Count()==0) {
      ClearWord(i);
    }
  }
  setEmptyLinkNumber();
}
public void BacktrackCrosspoint(int wordindex, int linkIndex) {
  ClearWord(wordindex);
  ClearWord(wordList[wordindex].GetLink(linkIndex).linkedToIndex);
  setEmptyLinkNumber();
}
public void grid1() {
  if (wordList.Count() == 0) {
    Add(new Word(5, 0, 0, Direction.Vertical));
    Add(new Word(6, 0, 1, Direction.Horizontal));
    Add(new Word(6, 2, 1, Direction.Vertical));
    Add(new Word(6, 5, 1, Direction.Vertical));
    Add(new Word(4, 0, 4, Direction.Horizontal));
    Add(new Word(6, 2, 6, Direction.Horizontal));
    setEmptyLinkNumber();
  }
}
public void grid2() {
  if (wordList.Count() == 0) {
    Add(new Word(7, 0, 1, Direction.Horizontal));
    Add(new Word(4, 0, 3, Direction.Horizontal));
    Add(new Word(5, 0, 5, Direction.Horizontal));
    Add(new Word(5, 0, 11, Direction.Horizontal));
    Add(new Word(4, 0, 13, Direction.Horizontal));
    Add(new Word(6, 0, 15, Direction.Horizontal));
    Add(new Word(7, 1, 0, Direction.Vertical));
    Add(new Word(7, 1, 10, Direction.Vertical));
```

```
    Add(new Word(5, 2, 7, Direction.Horizontal));
    Add(new Word(7, 2, 9, Direction.Horizontal));
    Add(new Word(4, 3, 0, Direction.Vertical));
    Add(new Word(7, 3, 5, Direction.Vertical));
    Add(new Word(4, 3, 13, Direction.Vertical));
    Add(new Word(5, 5, 0, Direction.Vertical));
    Add(new Word(7, 5, 3, Direction.Horizontal));
    Add(new Word(5, 5, 6, Direction.Vertical));
    Add(new Word(5, 5, 12, Direction.Vertical));
    Add(new Word(7, 5, 13, Direction.Horizontal));
    Add(new Word(5, 6, 5, Direction.Horizontal));
    Add(new Word(5, 6, 11, Direction.Horizontal));
    Add(new Word(5, 7, 2, Direction.Vertical));
    Add(new Word(7, 7, 9, Direction.Vertical));
    Add(new Word(7, 8, 7, Direction.Horizontal));
    Add(new Word(7, 9, 2, Direction.Vertical));
    Add(new Word(5, 9, 10, Direction.Vertical));
    Add(new Word(7, 10, 1, Direction.Horizontal));
    Add(new Word(5, 10, 9, Direction.Horizontal));
    Add(new Word(7, 10, 15, Direction.Horizontal));
    Add(new Word(5, 11, 0, Direction.Vertical));
    Add(new Word(5, 11, 6, Direction.Vertical));
    Add(new Word(5, 11, 12, Direction.Vertical));
    Add(new Word(5, 12, 5, Direction.Horizontal));
    Add(new Word(5, 12, 11, Direction.Horizontal));
    Add(new Word(4, 13, 0, Direction.Vertical));
    Add(new Word(4, 13, 3, Direction.Horizontal));
    Add(new Word(7, 13, 5, Direction.Vertical));
    Add(new Word(4, 13, 13, Direction.Horizontal));
    Add(new Word(4, 13, 13, Direction.Vertical));
    Add(new Word(7, 15, 0, Direction.Vertical));
    Add(new Word(7, 15, 10, Direction.Vertical));
    setEmptyLinkNumber();
  }
}
public void grid3() {
  if (wordList.Count()==0) {
    Add(new Word(5, 0, 0, Direction.Horizontal));
    Add(new Word(5, 0, 0, Direction.Vertical));
    Add(new Word(5, 2, 0, Direction.Vertical));
    Add(new Word(5, 4, 0, Direction.Vertical));
    Add(new Word(6, 6, 0, Direction.Vertical));
    Add(new Word(5, 8, 0, Direction.Horizontal));
    Add(new Word(5, 8, 0, Direction.Vertical));
    Add(new Word(5, 10, 0, Direction.Vertical));
    Add(new Word(5, 12, 0, Direction.Vertical));
    Add(new Word(5, 4, 1, Direction.Horizontal));
    Add(new Word(5, 0, 2, Direction.Horizontal));
    Add(new Word(5, 8, 2, Direction.Horizontal));
    Add(new Word(5, 4, 3, Direction.Horizontal));
    Add(new Word(5, 0, 4, Direction.Horizontal));
    Add(new Word(5, 1, 4, Direction.Vertical));
    Add(new Word(5, 3, 4, Direction.Vertical));
    Add(new Word(5, 8, 4, Direction.Horizontal));
    Add(new Word(5, 9, 4, Direction.Vertical));
    Add(new Word(5, 11, 4, Direction.Vertical));
    Add(new Word(6, 0, 6, Direction.Horizontal));
    Add(new Word(6, 7, 6, Direction.Horizontal));
    Add(new Word(6, 6, 7, Direction.Vertical));
    Add(new Word(5, 0, 8, Direction.Horizontal));
    Add(new Word(5, 0, 8, Direction.Vertical));
    Add(new Word(5, 2, 8, Direction.Vertical));
    Add(new Word(5, 4, 8, Direction.Vertical));
```

```
    Add(new Word(5, 8, 8, Direction.Horizontal));
    Add(new Word(5, 8, 8, Direction.Vertical));
    Add(new Word(5, 10, 8, Direction.Vertical));
    Add(new Word(5, 12, 8, Direction.Vertical));
    Add(new Word(5, 4, 9, Direction.Horizontal));
    Add(new Word(5, 0, 10, Direction.Horizontal));
    Add(new Word(5, 8, 10, Direction.Horizontal));
    Add(new Word(5, 4, 11, Direction.Horizontal));
    Add(new Word(5, 0, 12, Direction.Horizontal));
    Add(new Word(5, 8, 12, Direction.Horizontal));
    setEmptyLinkNumber();
  }
}
public void grid4() {
  if (wordList.Count() == 0) {
    Add(new Word(10, 0, 0, Direction.Vertical));
    Add(new Word(10, 2, 0, Direction.Vertical));
    Add(new Word(10, 4, 0, Direction.Vertical));
    Add(new Word(10, 6, 0, Direction.Vertical));
    Add(new Word(10, 8, 0, Direction.Vertical));
    Add(new Word(10, 0, 0, Direction.Horizontal));
    Add(new Word(10, 0, 2, Direction.Horizontal));
    Add(new Word(10, 0, 4, Direction.Horizontal));
    Add(new Word(10, 0, 6, Direction.Horizontal));
    Add(new Word(10, 0, 8, Direction.Horizontal));
    setEmptyLinkNumber();
  }
}
```

MyDictionary.cs

```
private List<String> wordList;
private HashSet<Restriction> restrictionList;

public MyDictionary() {
  wordList = new List<string>();
  restrictionList = new HashSet<Restriction>();
}
public MyDictionary(List<String> wordList, HashSet<Restriction> restrictionList) {
  this.wordList = wordList;
  this.restrictionList = restrictionList;
}
public MyDictionary(List<String> wordList, Restriction restriction) {
  this.wordList = new List<String>();
  this.restrictionList = new HashSet<Restriction>();
  restrictionList.Add(restriction);
  for (int i = 0; i < wordList.Count; i++) {
    if (wordList[i].Length > restriction.index) {
      if (wordList[i][restriction.index] == restriction.character) {
        this.wordList.Add(wordList[i]);
      }
    }
  }
}
public void Add(string str) { wordList.Add(str); }
public int Count() { return wordList.Count(); }
public string GetWord(int index) { return wordList[index]; }
public List<String> GetWords() { return wordList; }
public bool CheckWord(String str) { return wordList.Contains(str); }
```

Restriction.cs

```
public int index;
public char character;
```

```
public Restriction(int index, char character) {
  this.index = index;
  this.character = character;
}
public String toString() { return index + "" + character + " "; }
```

Language.cs

```
private char[] alphabet;

public Language() { setLatvian(); }
public void setLatvian() {
  alphabet = new char[33] {
    'A', 'Ā', 'B', 'C', 'Č', 'D', 'E', 'Ē', 'F', 'G', 'Ģ', 'H', 'I', 'Ī', 'J', 'K',
    'Ķ', 'L', 'Ļ', 'M', 'N', 'Ņ', 'O', 'P', 'R', 'S', 'Š', 'T', 'U', 'Ū', 'V', 'Z',
    'Ž'
  };
}
public void setEnglish() {
  alphabet = new char[26] {
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
    'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
  };
}
public bool isAcceptable(String str) {
  for (int i=0; i<str.Length; i++) {
    if (!alphabet.Contains(str[i])) return false;
  }
  return true;
}
public char GetLetter(int index) {
  if (index >= 0 && index < alphabet.Length) return alphabet[index];
  else return '_';
}
public int Length() { return alphabet.Length; }
```

Word.cs

```
private char[] chArray;
private Coordinate coordinate;
private List<Link> linkList;

public Word(int size, int x, int y, Direction d) {
  if (size > 0 && size < 100) {
    chArray = new char[size];
    for (int i = 0; i < size; i++) chArray[i] = '_';
    coordinate = new Coordinate {
      X = x,
      Y = y,
      Direction = d
    };
    linkList = new List<Link>();
  }
  else throw new InvalidOperationException();
}
internal Link GetLink(int i) { return linkList[i]; }
public String GetString() { return new String(chArray); }
public int GetL() { return chArray.Length; }
public char GetLetter(int index) {
  if (index >= 0 && index < GetL()) return chArray[index];
  else throw new IndexOutOfRangeException();
}
public Coordinate GetCoordinate() { return coordinate; }
public char[] GetChArray() { return chArray; }
public int GetX() { return GetCoordinate().X; }
```

```
public int GetY() { return GetCoordinate().Y; }
public Direction GetD() { return GetCoordinate().Direction; }
public void setLetter(int index, char letter) {
  if (index >= 0 && index < GetL()) chArray[index] = letter;
  else throw new IndexOutOfRangeException();
}
public void setWord(string word) {
  if (word.Length == GetL()) {
    for (int i = 0; i < word.Length; i++) chArray[i] = word[i];
  }
  else throw new IndexOutOfRangeException();
}
public void Add(Link l) { linkList.Add(l); }
public bool hasEmptyField() { return chArray.Contains('_'); }
public int linkCount() { return linkList.Count(); }
public int incompleteLinkCount() {
  int linkCount = 0;
  for (int i = 0; i < linkList.Count(); i++) {
    if (chArray[linkList[i].linkedLetterIndex] == '_') linkCount++;
  }
  return linkCount;
}
public bool CompleteLink(int index) {
  if (index < linkCount()) {
    return (chArray[linkList[index].linkedLetterIndex] != '_');
  }
  else throw new IndexOutOfRangeException();
}
public String toString() {
  String str = "{ " + coordinate.X + "; " + coordinate.Y + "; " +
  coordinate.Direction + " - ";
  for (int i = 0; i < linkList.Count(); i++) str += linkList[i].toString() + " - ";
  str += "} ";
  for (int i = 0; i < GetL(); i++) str += chArray[i];
  return str;
}
public int selectRandomIncompletCrosspoint(Random rnd) {
  List<int> intList = new List<int>();
  for (int i=0; i<linkCount(); i++) {
    if (!CompleteLink(i)) intList.Add(i);
  }
  if (intList.Count > 0) return intList[rnd.Next(intList.Count)];
  else return -1;
}
```

Link.cs

```
public int linkedToIndex;
public int linkedLetterIndex;
ublic int linkedToLetterIndex;

public Link(int linkedToIndex, int linkedLetterIndex, int linkedToLetterIndex) {
  this.linkedToIndex = linkedToIndex;
  this.linkedLetterIndex = linkedLetterIndex;
  this.linkedToLetterIndex = linkedToLetterIndex;
}
internal string toString() {
  return linkedToIndex + "," + linkedLetterIndex + "," + linkedToLetterIndex;
}
```

Coordinate.cs

```
public struct Coordinate {
  public int X { get; set; }
  public int Y { get; set; }
```

```
  public Direction Direction { get; set; }
}
```

Direction.cs

```
public enum Direction { Horizontal, Vertical }
```

DOKUMENTĀRĀ LAPA

Maģistra darbs "Algoritmi vārdu salikšanai krustvārdu mīklu režģī" izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 20.01.2019.

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

Autors: _____

(Autora paraksts un datums)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par p i e m ē r o t u / n e p i e m ē r o t u (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____

(Vadītāja paraksts un datums)

Darbs iesniegts maģistratūras sekretariātā _____.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: _____.

(Metodiķes paraksts)

Recenzents: _____

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)