Универзитет у Београду
Математички факултет

Далибор Станисављевић

Алгоритми за сређивање клизне слагалице

Београд 2018.

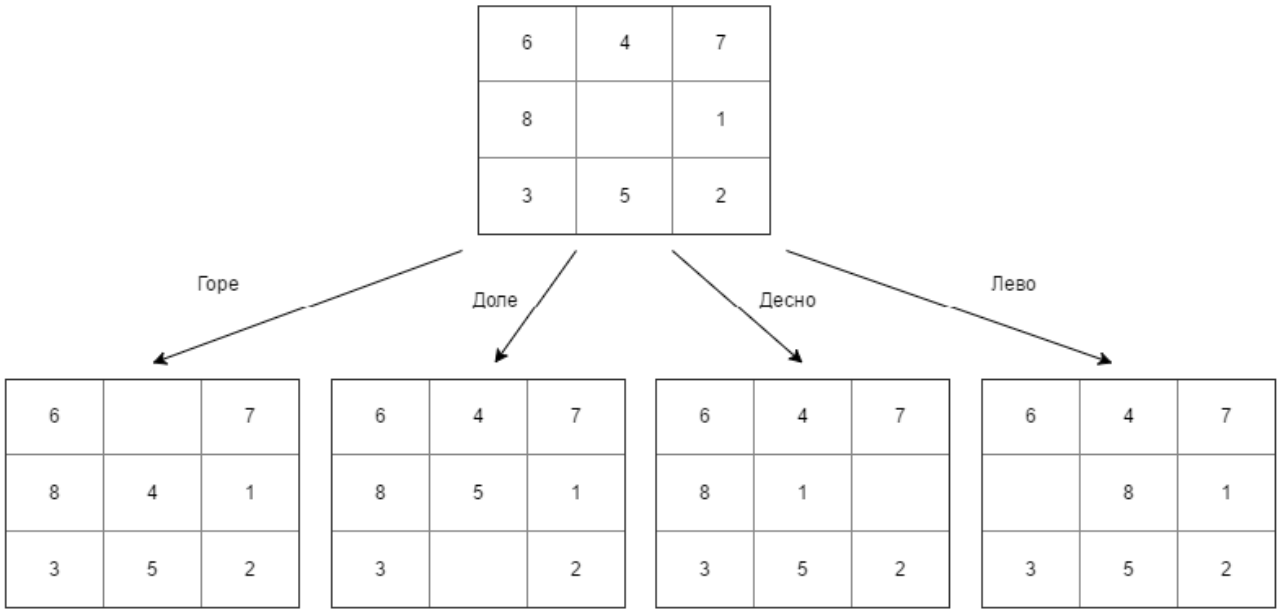
Универзитет у Београду
Математички факултет

Аутор:	Далибор Станисављевић
Наслов:	Алгоритми за сређивање клизне слагалице
Ментор:	др Миодраг Живковић Универзитет у Београду, Математички факултет
Чланови комисије:	др Филип Марић, ванр. проф. др Весна Маринковић, доцент
Датум:	2018

Садржај

1 Увод

Клизна слагалица је игра која се састоји од оквира у коме се налазе случајно распоређене квадратне плочице од којих једна недостаје. Често се ова слагалица означава као слагалица са 8 или 9 плочица, у зависности да ли се рачуна празно поље или не. Циљ слагалице је поређати плочице у растућем поретку, клизним померањем једне по једне плочице на празно место.



Слика 1. Пример четири могућа преласка у наредну конфигурацију

На слици 1 приказане су могуће конфигурације слагалице, које се могу добити у зависности од померања празне плочице. Ова игра постоји у различитим величинама, на пример клизна слагалица са 15 плочица. Број плочица може бити различит, тако да се ова игра може посматрати у општем случају као клизна слагалица са n плочица.

Игру је измислио Нојс Палмер Чампан (енг. Noyes Palmer Chapman) 1874. у Канастоти, Њујорк и показао својим пријатељима. Игра је постала популарна широм Америке, Канаде и Европе 1880. године. Чампан је покушао да је патентира 1880. године али је одбијен због велике сличности са игром коју је, 1878 године, направио и заштитио Ернест Кинси (енг. Ernest U. Kinsey). Настанак ове слагалице, ипак, се приписује Сему Лојду (енг. Sam Loyd), који се цео живот трудио да патентира ову игру [12]. Лојдов први чланак о слагалицама је био објављен 1886, а понудио је и награду од 1000 долара ономе ко реши ону на којој су плочице 14 и 15 замениле места. Касније се показало да је таква конфигурација нерешива [1].

Рад је подељен у пет поглавља. Прво поглавље је уводно [12]. Друго поглавље бави се условима које треба испунити да би слагалица била решива, потребним и довољним условима за решавање слагалице као и доказима када је слагалица решива [11]. У трећем поглављу су описани алгоритми који су коришћени за сређивање клизне слагалице. Ово поглавље је подељено у пет делова. У првом делу су описани тачни алгоритми који немају никакву информацију приликом претраге [4,6] и који сигурно проналазе решење, у другом су објашњени алгоритми који користе различите хеуристике ради ефикасније претраге [2,3,5,7,13,14] и који увек проналазе решење. У трећем делу је описан начин рада генетског алгоритма за решавање различитих проблема, а детаљно је описано како се користи за сређивање клизне слагалице [8,9]. У четвртом делу објашњен је алгоритам симулираног каљења док се пети бави рекурзивним алгоритмом који ефикасно проналази решења за слагалице великих димензија [10]. Опис конкретног софтверског решења, као и резултати који су добијени детаљно су изложени у поглављу 4, а поглавље 5 садржи закључак. Сви описани алгоритми раде само за слагалице димензија три и четри, осим последњег описаног рекурзивног алгоритма за решавање слагалица који ради на моделима димензија до 100.

2 Решивост слагалице

У овом поглављу излажу се потребни и довољни услови да уопштена слагалица димензија $n\times n$ буде решива [11]. Клизна слагалица представља оквир у коме се налазе нумерисане плочице поређане с лева на десно, нижући врсту једну за другом, при чему је последња плочица празна. Број плочица у врсти и колони је исти; изузетак су врста и колона са празном плочицом. Стабло претраживања представља различите конфигурације слагалице које се могу добити померањем плочица. Решивост слагалице представља могућност да се померањем плочица добије циљна конфигурација. Ширина мреже је максимални број плочица које се налазе у врсти или колони. Пермутација бројева $(1,2,..,n)$ је свака уређена n-торка $(i_1,i_2,..,i_n)$ у којој се сваки од бројева $(1,2,\ldots,n)$ јавља тачно једном. Бројеви i_p и i_q су у инверзији ако је $p<q$ и $i_p>i_q$. На пример, пермутација $1,2,4,3$ има једну инверзију $(4>3)$. Пермутација $3,1,2,4$ има две инверзије $(3>1,3>2)$. Пермутација $1,2,3,4$ има 0 инверзија. У случају клизне слагалице инверзија је пар плочица нумерисаних бројевима $a,b,$ при чему је $a>b,$ а плочица a се појављује пре b када се врсте матрице нижу једна за другом. Ако су дате почетна конфигурација слагалице $P,$ односно циљна конфигурација $Q,$ матрице реда $n\times n,$ онда се за израчунавање укупног броја инверзија $r,$ може искористити алгоритам описан следећим кодом. Претпоставка је да су у почетној конфигурацији $P,$ плочице поређане редом према величини.

Алгоритам: Број инверзија почетне слагалице P у односу на жељену слагалицу Q
Улаз: почетна конфигурација $P,$ циљна конфигурација Q
Излаз: број инверзија r

```
1  r = 0
2  for  i = 0 to n^2-1
3       for j = i + 1 to n^2-1
```

```
4            if (indeks(Q[i]) > indeks(Q[j]) && P[j] != 0) { r++; }
5  return r
```

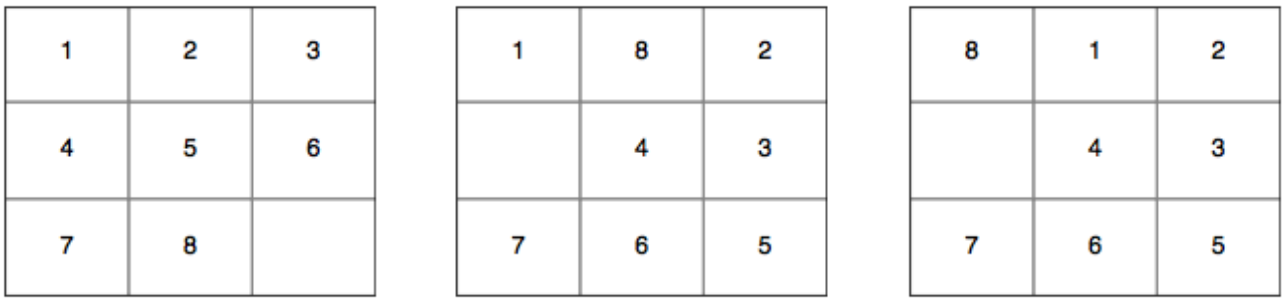
Нека је k редни број врсте у којој је празна плочица. Испоставља се да је клизна слагалица решива ако и само ако је испуњен један од следећих услова (ова тврдња доказује се у наредном поглављу):

1 n је непарно, r је парно
2 n је парно и
  2.1 парности k и r су различите

Према томе, провера решивости слагалице на основу n, k и r може се извршити алгоритмом са следећим кодом:

Алгоритам: Решивост слагалице
Улаз: n ширина мреже, k врста празне плочице рачунајући одоздо (при чему је редни број прве врсте једнак 1), r број инверзија
Излаз: да ли је слагалица решива или не

```
1  if (n % 2 == 0)
2       { return (k+r) % 2 == 1;}
3  else { return r % 2 == 0; }
```



r = 0 r = 24 r = 25

Слика 2. Циљна конфигурација слагалице, пример решиве и нерешиве слагалице

На слици 2 приказана је циљна конфигурација слагалице за $n=3$, пример решиве слагалице која се померањем празне плочице може решити, то јест може се добити циљна конфигурација. На крају се налази пример нерешиве конфигурације, од које се померањем плочица не може добити циљна конфигурација.

2.1 Потребан услов решивости слагалице

Доказује се тврђење да је за решивост слагалице неопходно да буде испуњен услов који се односи на парност укупног броја инверзија [11].

У доказу се разматра општији случај када слагалица има m врста и n колона. Нека је $A[i,j]$ број на плочици у врсти i и колони j, где је $1 \le i \le m$ и $1 \le j \le n$. Ако је плочица празна онда $A[i,j]=0$. Важи услов $1 \le A[i,j] < n \times m$. Пермутација $A[1,1], A[1,2], \ldots, A[1,n], A[2,1], A[2,2], \ldots, A[2,n], \ldots, A[m,n]$ једнозначно одређује конфигурацију. Другим речима, скуп конфигурација може бити мапиран 1 на 1 са овим скупом пермутација. Пермутацији одговара низ $B[1,..,nm]$ такав да је $B[k]=A[1+((k-1)/n), 1+((k-1) mod n)]$, $k=1,2,..,nm$. При томе / означава целобројно дељење. На овај начин $B[1]=A[1,1]$, $B[2]=A[1,2], \ldots, B[n]=A[1,n]$, $B[n+1]=A[2,1], \ldots, B[n \times m]=A[m,n]$. Празна плочица се може игнорисати, па дефинишимо низ $C[1, \ldots, n \times m-1]$ који се од низа B добија избацивањем нуле, односно празне плочице. На пример за $n=3$ и $m=2$ ако је $B[]=1,2,4,3,0,5$, онда је $C[]=1,2,4,3,5$.

Почевши од полазне конфигурације $C[]$, можемо достићи било коју конфигурацију која је решива. Постоји само 4 померања која можемо разматрати:

- Померање на лево - Мења $B[]$ померањем празне плочице на десно, али не мења $C[]$, а самим тим не мења број инверзија.
- Померање на десно - Мења $B[]$ померајући празну плочицу на лево, али не мења $C[]$, а самим тим не мења број инверзија.
- Померањем плочице нагоре (или на доле) - Мења C на следећи начин: једна од вредности $C[]$ сада долази пре $n-1$ различитих вредности који су били пре ње. На пример, конфигурацији

	7	2	1
4	3	6	5

одговара низ $C[]=7,2,1,4,3,6,5$. Померање 4 нагоре одговара премештању 4 у низу $C[]$ преко $3=4-1$ елемената тог низа, после чега се добија $C[]=4,7,2,1,3,6,5$. Нека је T број на плочици коју померамо (у овом случају, T је 4). Испитајмо $n-1$ плочице које смо померили. Нека је r плочица од њих мање од T. Осталих $(n-1)-r$ плочица су веће од T. Ако је V број инверзија пре померања и W је број инверзија после померања, онда $W=V-(n-1-r)+r=V+2 \times r-n+1$. Видимо да је парност промене броја инверзија једнака парности броја $n-1$. Ово важи и у случају када се плочица помера на доле.

Теорема 1:

1.

a) Ако је n непарно, онда свака решива конфигурација одговара пермутацији $C[]$ са парним бројем инверзија.

b) Ако је n парно, онда свака решива конфигурација са празном плочицом у врсти i, где је $m-i$ парно одговара пермутацији $C[]$, са парним бројем инверзија.

c) Ако је n парно, онда свака решива конфигурација са празном плочицом у врсти i где је $m-i$ непарно одговара пермутацији $C[]$, са непарним бројем инверзија.

2. Има $n!/2$ пермутација величине n које имају паран број инверзија.

3. Од свих $(n\times m)!$ могућих конфигурација само њих пола (које имају паран број инверзија) могу бити решиве.

Доказ:

1.

a) Ако је n непарно, онда је W увек парно. Ако је n парно, онда померање нагоре или надоле доводи до $W=V+1 mod 2$. Нека је U број померања нагоре и D број померања надоле да би достигли циљну конфигурацију.

b) Ако је празна плочица у врсти i и $m-i$ је парно, онда $D-U=0 mod 2$. Због тога је број инверзија $D x 1-U x 1=0 mod 2$. Конфигурација мора имати парни број инверзија.

c) Ако је празна плочица у врсти i и $m-i$ је непарна, онда $D-U=1 mod 2$. Због тога је број инверзија $D x 1-U x 1=1 mod 2$. Конфигурација мора имати непарни број инверзија.

2.

Скуп пермутација са парним бројем инверзија може се пресликати 1 на 1 на скуп пермутација са непарним бројем инверзија на следећи начин:

Претпоставимо да је $C[1,...,n]$ пермутација са парним бројем инверзија. Онда можемо да креирамо пермутацију са непарним бројем инверзија мењајући $C[n-1]$ са $C[n]$, тако што повећавамо број инверзија за 1, ако је $C[n-1]<C[n]$, а ако није смањујемо број инверзија за 1.

Како је мапирање 1 на 1 и има $n!$ различитих пермутација, има $n!/2$ парних пермутација и $n!/2$ непарних пермутација.

3.

Претпоставимо да је n непарно. На основу тврђења 1.а) и 2. теореме, нерешивим конфигурацијама одговара бар $(n x m-1)!/2$ пермутација са непарним бројем инверзија. Свака

од ових пермутација одговара $n \times m$ различитим конфигурацијама (одговара постављању празне плочице на $n \times m$ различитих позиција). То значи да има најмање $((n x m-1)!/2) x (n x m)=(n x m)!/2$ нерешивих конфигурација.

Претпоставимо да је n парно. На основу тврђења 2, има $(n x m-1)!/2$ пермутација са непарним бројем инверзија. То одговара $(n x m)/2$ конфигурацијама у којима је празна плочица у i-тој врсти, где је i парно. На основу теореме 1.1b, чини укупно $((n x m-1)!/2) x ((n x m)/2)=(n x m)!/4$ нерешивих конфигурација, а на основу теореме 2, има $(n x m-1)!/2$ пермутација са парнм бројем инверзија. То одговара $(n x m)/2$ конфигурацијама у којима је празна плочица у врсти i где је i непарно, а по теореми 1.c), то одговара укупно $((n \times m-1)!/2) \times ((n \times m)/2)=(n \times m)!/4$ нерешивим конфигурацијама. Укупно то чини $(n x m)!/2$ нерешивих конфигурација.

2.2 Довољан услов решивости конфигурације

У овом одељку доказује се да су решиве све конфигурације које се од коначне разликују за паран број инверзија. У доказу се користи чињеница да се свака пермутација исте парности као полазна, може добити од полазне пермутације низом тројних замена $(a,b,c) \rightarrow (c,a,b)$ суседних елемената пермутације. Треба нагласити да примена тројне замене не мења парност пермутације [11].

Лема 1: Полазећи од парне пермутације низом тројних замена може се добити пермутација $(1,2,\ldots,n)$. Полазећи од непарне пермутације низом тројних замена такође се може добити непарна пермутација $(1,2,\ldots,n-2,n,n-1)$.

Доказ:

Доказ се изводи индукцијом по n. Посматрајмо пермутацију $1,2,3$. Тројним заменама могу се добити све три пермутације са парним бројем инверзија:

$1,2,3$

$3,1,2$

$2,3,1$

Полазећи од непарне пермутације, могу се добити све непарне пермутације:

$1,3,2$

$2,1,3$

$3,2,1$

Индукцијски корак: Претпоставимо да можемо да достигнемо све непарне пермутације са n бројева из једне од непарних пермутација са n бројева и да можемо да достигнемо све парне пермутације из једне од парних пермутација. Онда можемо да достигнемо све непарне

пермутације са $n+1$ бројева из једне непарне пермутације, и можемо да достигнемо све парне пермутације из једне парне пермутације.

Заиста, посматрајмо пермутацију $1,2,3,\ldots,n+1$. Тројним заменама можемо да достигнемо све парне пермутација за n бројева. Постоје две фазе којим се то постиже:

- довођење 1 на прво место низом тројних замена
- преуређење осталих n на основу индуктивне хипотезе

Тако добијамо све пермутације са парним бројем инверзија које почињу бројем 1.

Низом инверзија може се било који број довести на прво место у пермутацији. Преосталих n бројева могу се преуредити низом тројних замена до свих пермутација исте парности. На овај начин можемо доћи до свих пермутација $n+1$ бројева са парним бројем инверзија. На аналогни начин може се доказати да се може доћи и до свих пермутација $n+1$ бројева са непарним бројем инверзија.

Теорема 2: У клизној слагалици на табли $n x m$ може се остварити тројна замена било које три узастопне плочице у истој врсти.

Доказ:

Тројна замена било које три суседне плочице представља $(a,b,c)\rightarrow(c,a,b)$.
Применом два пара инверзија можемо превести једну конфигурацију у другу на следећи начин $(a,b,c)\rightarrow(b,c)(a,c)$. Парност пермутације остаје исти јер смо применили два пара инверзија за добијање нове конфигурације. Применом прве инверзија (b,c) добија се $(a,b,c)\rightarrow(a,c,b)$, а затим применом друге (a,c) добија се $(a,c,b)\rightarrow(c,a,b)$. На сличан начин добија се $(c,a,b)\rightarrow(a,b,c)$.

Теорема 3: Клизна слагалица на табли, $n x m$ где је $n,m>1$ има бар $(n x m)!/2$ решивих конфигурација.

Доказ:

Доказ ове теореме произилази из претходних теорема, 1 и 2 и гласи - све пермутације неке парности могу се добити тројним заменама од произвољне пермутације исте парности; а свака тројна замена може се остварити низом одређених потеза на клизној слагалици.

Можемо формирати другу конфигурацију у којој је једна од плочица померена кроз две друге плочице, укључујући померање около.

Можемо то приказати следећим примером:

	7	2	1	8
4	6	3	5	9

Можемо померити 8 преко 1 и 2 и добити:

	7	8	2	1
4	6	3	5	9

Или померити 8 преко 4 и 6 и добити:

	7	2	1	4
6	8	3	5	9

На основу теореме 1 и 3 проистиче да клизна слагалица са $n x m$ где је $n, m > 1$ има тачно $(n x m)!/2$ легалних конфигурација.

3 Алгоритми за решавање слагалице

Постоји доста истраживања у овој области. У овом раду ће бити анализирани неки од тих алгоритама, за које мислим да најбоље одговарају решавању овог проблема. Такође ће бити описане предности и мане алгоритама, а биће упоређени и резултати извршавања ових алгоритама.

Стабло варијанти може се дефинисати, имајући на уму пример на слици 1. Чворови стабла су конфигурације, синови чвора су конфигурације које се могу од њега добити једним потезом, а корен стабла је почетна конфигурација. Задатак који треба решити је пронаћи у овом стаблу пут од корена до чвора који одговара циљној конфигурацији. У даљем тексту уместо „стабло варијанти“ каже се једноставније „стабло“.

У раду се приказује више алгоритама за решавање клизне слагалице. Алгоритми се могу разврстати у односу на испуњеност неколико критеријума, као што су: потпуност, временска и просторна сложеност, оптималност. Оптимално решење је решење које има најмањи број потеза из полазне позиције.

Потпуност:
Да ли алгоритам враћа тачно решење ако постоји. Неки алгоритми претраге нису потпуни и не проналазе увек решење. Генетски алгоритми у неким случајевима не враћају решење иако постоји.

Временска сложеност - Време потребно да алгоритам реши слагалицу. Брзина зависи колико је почетна конфигурација удаљена од циљне конфигурације слагалице, то јест колико је померања плочице потребно за њено решавање. У случају алгоритама који користе хеуристике, зависи од избора хеуристичке функције.

Просторна сложеност - Количина меморије потребна да би се пронашло решење.

Оптималност - Да ли алгоритам гарантује враћање решења које се састоји од најмањег могућег

броја потеза.

На почетку су описани алгоритми који претражују стабло без информација приликом претраге. Неки од описаних алгоритама су претрага у ширину, дубину и претрага у дубину итеративним продубљивањем. Хеуристички алгоритми имају додатне информације приликом претраге стабла, што омогућава краће време претраге и број корака потребних за решавање слагалице. Описана је и примена генетских алгоритама приликом решавања клизне слагалице, као и њихове предности и мане. Симулирано каљење је један од алгоритама који је примењен на решавање клизне слагалице. Рекурзивни алгоритам за решавање клизне слагалице представља један од најефикаснијих алгоритама у погледу димензије слагалице које може решити $n=100$.

3.1 Тачни алгоритми

Овакви алгоритми претражују чворове стабла док не дођу до решења. Они траже решења са јединим знањем који је потез могућ. Они претражују путање где се можда налази решење, не узимајући у обзир колико је та конфигурација која се претражује удаљена од решења.
У овој категорији су алгоритми као што су претага у ширину, претрага у дубину, ограничена претрага у дубину, претрага у дубину итеративним продубљивањем, униформна претрага. Алгоритми неинформисане претраге користе отворену и затворену листу, где отворена листа представља конфигурације које тек треба обрадити док затворена листа представља конфигурације које су обрађене. Листа затворених конфигурација служи како се не би обрађивали исте конфигурације више пута.

Алгоритам: Неинформисана претрага
Улаз: почетна конфигурација P , циљна конфигурација C
Излаз: низ корака који воде до решења solution

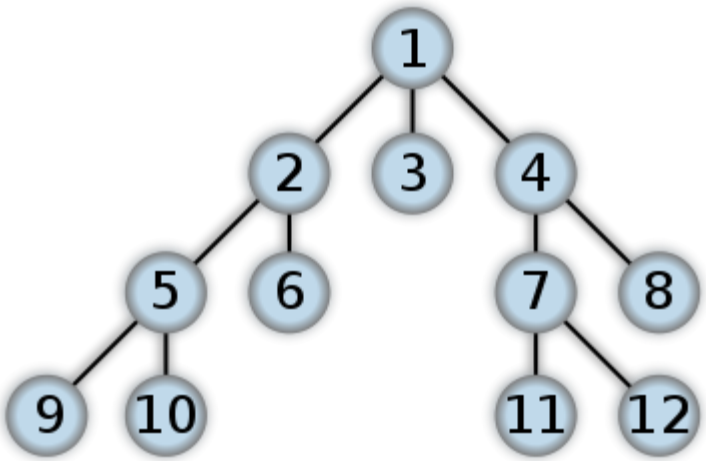
```
// отворена листа open_set
// затворена листа closed_set
// решење solution
// функција push() ставља елемент у отворену листу
// функција top() скида елемент са врха листе
// функција expand() одређује све могуће конфигурације које се могу
// добити из тренутне конфигурације temp.

1  open_set->push(P);
2  while (open_set->size())
3       temp = open_set->top()
4       if (temp == C)
5            solution = temp;
6       if (temp not in closed_set)
7            children = temp->expand();
```

```
8            closed_set->insert(temp);
9            open_set→push(children);
```

3.1.1 Претрага у ширину

Код претраге у ширину (BFS) листа отворених конфигурација је ред, то јест у кораку 3 општег алгоритма из реда се вади конфигурација која је најдуже у реду, а у линији 9 у ред се додају синови те конфигурације.

Пошто претрага у ширину претражује стабло у целини, ниво по ниво, најкраће решење се увек проналази [2, 14]. Такође алгоритам гарантује да ће решење са најкраћом путањом бити пре нађено него било које решење са дужом путањом. Псеудокод алгоритма је дат у уводном делу поглавља 3.1 и једина разлика је што је отворена листа open_set имплементирана као ред.
На слици 3 приказан је пример претраге у ширину стабла варијанти; бројеви у чворовима показују редослед њихове обраде.

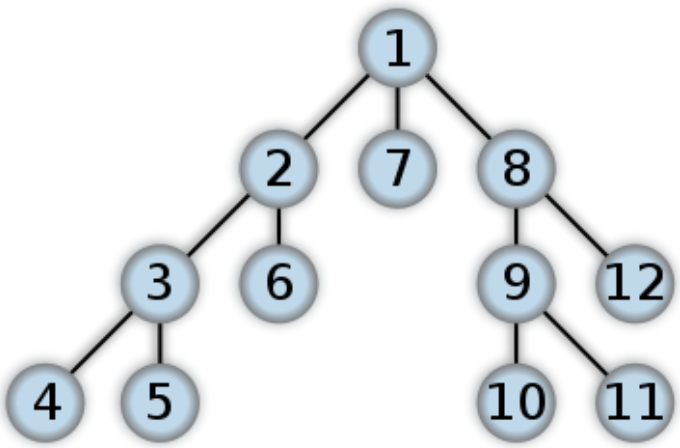


Слика 3. Редослед претраге чворова коришћењем претраге у ширину

3.1.2 Претрага у дубину

Претрага у дубину (DFS) ради тако што претражује једну путању у једном тренутку. Претрага у дубину креће из почетног чвора r . Затим се узима непосећени чвор $r1$ суседан r, па се из чвора $r1$ одређују све конфигурације које се могу добити и посећује први од тих чворова. Претрага се завршава када се дође до чвора за који се не могу генерисати нове конфигурације. То значи да ће алгоритам променити путању коју обрађује само ако дође до

краја. У крајње стање се долази или ако нема више стања која се могу проширити или када знамо да се решење не може наћи па одбацимо ту грану [2, 14].
На слици 4 бројевима је означен редослед претраге, а чворови представљају конфигурације слагалице коришћењем претаге у дубину.



Слика 4. Редослед претраге чворова коришћењем претраге у дубину

Ограничена претрага у дубину

Постоји варијанта алгоритма DFS где се као улаз може навести дубина до које треба обрађивати стабло:

Алгоритам: Ограничена претрага у дубину
Улаз: почетна конфигурација P, циљна конфигурација C
Излаз: низ корака који воде до решења solution

```
// отворена листа open_set
// решење solution
// функција push() ставља елемент у отворену листу
// top() скида елемент са врха листе
// функција expand() одређује све могуће конфигурације које се могу
     добити из тренутне конфигурације temp
// get_path_cost() број корака од почетне до тренутне конфигурације
     слагалице
// limit ограничење до које дубине треба претрага да иде

  1  open_set->push(P);
  2  while (open_set->size())
  3         temp = open_set->top()
```

```
4         if (temp == C)
5             solution = temp;
6         if (temp->get_path_cost() == limit)
7            continue;
8        children = temp->expand();
9        open_set->push(children);
```

3.1.3 Претрага у дубину итеративним продубљивањем

Претрага у дубину итеративним продубљивањем је стратегија претраге у којој се алгоритам ограничене претраге у дубину узастопно употребљава повећавајући границу дубине са сваком наредном итерацијом док се не достигне решење или нека максимална дубина.
С обзиром да почетне итерације користе мале вредности за дубину, извршавају се изузетно брзо. Ово омогућава алгоритму да достави ране претпоставке о резултату скоро тренутно, а претпоставке постају све прецизније како се дубина повећава. Када се користи у итеративној поставци, као на пример у програму за шах, потез са тренутно најбољом проценом омогућава програму да у било ком тренутку одигра тренутно најбољи до сада пронађени потез [14].

Алгоритам: Претрага у дубину итеративним продубљивањем
Улаз: почетна конфигурација P, циљна конфигурација C
Излаз: низ корака који воде до решења solution

```
1  int depth = 0; //  Иницијално дубина је 0
2  while(!solution)
3       depth_limited_search(P, C, solution);
4       depth++;
```

Алгоритам: Ограничена претрага у дубину
Улаз: почетна конфигурација P, циљна конфигурација C
Излаз: низ корака који воде до решења solution

```
// отворена листа open_set
// решење solution
// функција push() ставља елемент у отворену листу
// top() скида елемент са врха листе
// функција expand() одређује све могуће конфигурације које се могу
добити из тренутне конфигурације temp
// get_path_cost() број корака од почетне до тренутне конфигурације
слагалице
```

```
// limit ограничење до које дубине треба претрага да иде

  1 depth_limited_search(P, C, solution)
  2 open_set->push(P);
  3 while (open_set→size())
  4       temp = open_set->top()
  5       if (temp == C)
  6             solution = temp;
  7       if (temp->get_path_cost() == limit)
  8             continue;
  9       children = temp->expand();
  10      open_set->push(children);
```

3.2 Хеуристички алгоритми

За разлику од неинформисане, информисана претрага не бира наредне чворове за претрагу насумично. Информисана претрага процењује колико је сваки чвор удаљен од решења, што омогућава да се као наредни чвор одабере онај који највише обећава.

Хеуристички алгоритми као и алгоритми неинформисане претраге користе отворену и затворену листу. Листа отворених конфигурација такође служи како се иста конфигурација не би обрађивала више пута. У сваком кораку, најбољи чвор из отворене листе се помера у затворену листу, потом се проширује и његови следбеници се додају у отворену листу. Та процедура се наставља све док се не дође до решења. За отворену листу као структура података користи се модификовани ред. Ред је сортиран на основу вредности хеуристичке функције. Уз конзистентну хеуристичку функцију, чвор који је проширен и премештен у затворену листу, никада не би требао да буде поново враћен у отворену листу. Сам алгоритам има следећу структуру:

Алгоритам: Хеуристички алгоритми
Улаз: почетна конфигурација P , циљна конфигурација C
Излаз: низ корака који воде до решења solution

```
// отворена листа open_set
// затворена листа closed_set
// решење solution
// функција push() ставља елемент у отворену листу
// top() скида елемент са врха листе
// функција expand() одређује све могуће конфигурације које се могу
добити из temp
```

```
// функција get_heur_value() одређује вредност хеуристике за слагалицу
// функција set_heur_value() додељује вредност хеуристике за слагалицу

1  open_set->push(P);
2  while(open_set->size())
3       temp = open_set->top() // тренутно најбоља отворена
                                  конфигурација
4       if (temp == C)
5            solution = temp;
6       if(temp not in closed_set)
7            children = temp->expand();
8            closed_set→insert(temp);
9            children->set_heur_value(get_heuristic(children));
10           open_set→push(children);
```

3.2.1 Похлепни алгоритам (eng. Greedy algoritam)

За сортирање елемената у отвореној листи користи се само вредност погодне хеуристичке функције. У зависности од вредности хеуристике алгоритам прво обрађује чворове са најмањом вредношћу. На овај начин алгоритам проналази решења веома брзо, али она нису увек оптимална по броју корака потребних за решавање слагалице. Најбоље перформансе се постижу приликом решавања проблема малих димензија и уколико је потребан брз одговор. За многе друге проблеме, похлепни алгоритми могу бити погрешан избор стратегије. Рецимо, у партији шаха, похлепни алгоритам ће увек одиграти онај потез који изгледа најбоље у датом тренутку, не обазирући се на његове последице. На пример, уколико највећу вредност за играча у датој ситуацији има потез у коме он узима противничког ловца, он ће то и учинити, све и ако тај потез отвара противнику могућност да победи матом. Похлепни алгоритам не гарантује проналажење оптималног решења [3,14].

3.2.2 A*

Алгоритам A* спада у најпопуларније алгоритме за проблеме налажења пута јер је флексибилан и може се користити у великом броју ситуација, укључујући и рад са огромним просторима стања. Први пут је представљен 1968. године као надоградња Дијкстриног алгоритма из 1959. године. Добре перформансе су постигнуте коришћењем хеуристичке функције за процену минималних трошкова обиласка графа [2].

A*, као један од хеуристичких алгоритама, користи две листе, отворену и затворену, да би управљао претрагом за најјефтинијим путем од почетног до циљног чвора. На почетку, отворена

листа садржи почетни чвор и затворена листа је празна. У сваком циклусу алгоритма, чвор са најбољом проценом ће бити проширен, померен у затворену листу, а наследници чвора се убаце у отворену листу. Због тога затворена листа садржи чворове који су већ проширени, генерисањем њихових потомака, а отворена листа садржи чворове који су већ генерисани, а нису проширени. Претрага се завршава када је циљни чвор одабран за проширење. Путања решења се може добити коришћењем показивача уназад од циљног чвора до почетног.

Редослед у ком се чворови проширују се добија на основу вредности ограничавајуће функције $f(n)=g(n)+h(n)$, где је $g(n)$ цена путање од почетног чвора до чвора n и $h(n)$ представља вредност хеуристике, процену цене пута од чвора n до циљног чвора. Понашање A* зависи највише од избора хеуристичке функције $h(n)$, која одређује ток претраге. Ако је $h(n)$ прихватљива, то јест ако никад не прецењује стварну цену, и ако су чворови проширени на основу функције $f(n)$, онда се гарантује да ће први чвор који ће бити одабран за проширење бити оптималан [3]. Друга важна карактеристика хеуристике да мора бити конзистентна. Хеуристика $h(n)$ је конзистентна ако важи да за сваки чвор n и сваког сина n' чвора n, цена достизања циља из n, није већа од цене пута од n до n' плус цена достизања циља из n' $h(n)\leq c(n,n')+h(n')$. То је облик неједнакости троугла која подразумева да свака страна троугла не може бити већа од збира друге две странице. У овом случају троугао је формиран од чворова n, n' и циљног чвора. Уколико је хеуристика конзистентна то значи и да је прихватљива [6,14].

Ако хеуристика $h(n)$ није конзистентна, постоји могућност да A* нађе бољу путању до чвора након што је чвор проширен. У овом случају побољшана цена функције g неког чвора треба да се проследи даље ка његовим потомцима.

Идеална хеуристичка функција ће веома брзо наћи најкраћу путању. Уколико није изабрана најпогоднија хеуристичка функција, алгоритам ће и даље налазити најкраће путање, али ће му требати више времена или ће радити веома брзо, али неће увек налази најбоља решења. Дакле, добра хеуристичка функција мора направити компромис између брзине и прецизности рада.

3.2.3 IDA*

IDA* је варијанта алгоритма A*. Користи идеју хеуристике од алгоритма A* за одређивање цене да би се дошло до циљног чвора. Пошто користи претрагу у дубину итеративним продубљивањем, због чега користи мање меморије него A*, алгоритам IDA* претражује чворове који много више обећавају. IDA* не користи динамичко програмирање, тако да се више пута претражују исти чворови. Приликом одсецања користи информацију $f(n)=g(n)+h(n)$, где је $g(n)$ цена од почетног чвора до чвора n, $h(n)$ хеуристичка функција која даје цену од чвора n до циљног чвора [3,14].

На почетку алгоритма као вредност горње границе (function_limit_cost) додељује се вредност хеуристике почетног чвора. Затим се позива алгоритам претраге у ширину (run_search_dfs) све док није пронађено решење или уколико function_limit_cost не достигне горњи праг (infinty_value). Приликом сваког повратка из рекурзивног позива run_search_dfs, вредност горње границе је повећана и претрага поново креће од почетка са новом вредношћу

нове границе.

Постоје два услова изласка из алгоритма run_search_dfs: уколико је хеуристика тренутног чвора који се обрађује већа од тренутне вредности горње границе (function_limit_cost) враћа се хеуристика тренутног чвора или уколико је пронађено решење, претрага се прекида. Током сваког рекурзивног позива када се достигне чвор чија је вредност хеуристике већа од горње границе, такви чворови се не проширују и минимум хеуристика таквих чворова представља нову вредност горње границе function_limit_cost са којом се поново започиње претрага.

```
Алгоритам: IDA
Улаз: почетна конфигурација P, циљна конфигурација C
Излаз: solution

\\ P.h() одређује вредност хеуристике
\\ решење solution,
\\ function_limit_cost горња граница за број корака
   1
   2  is_solved = false
   3  solution = null // у функцији  run_search_dfs, додаје се низ
                       корака које воде до решења
   4  function_limit_cost = P.h()
   5  while (!is_solved && function_limit_cost < infinty_value)
   6       function_limit_cost = run_search_dfs(P, 0)
   7  return solution

Алгоритам: run_search_dfs(P, move_cost)
Улаз: конфигурација слагалице P,  цена померања move_cost
Излаз: минимална вредност функције fun_min_value

\\ цена померања приликом добијања наредне конфигурације move_cost
\\ moves садржи све могуће преласке у наредну конфигурације слагалице

   1  f = move_cost + P.h()
   2  if (f > function_limit_cost) // горња граница за број корака у
                                   // решењу слагалице
   3       return f
   4  solution.push_back (P)
   5  if (P == C)
   6       is_solved = true
   7       return f
   8  fun_min_value = infinty_value //  додели  fun_min_value  горњу
                                    //  границу
   9  moves = puzzle_successors (solution);
   10 for it in moves
   11       f = run_search_dfs(it, move_cost + it->cost())
            // позива се претрага са наредним чвором као тренутним
```

```
12      if (is_solved) return f;
13      if (f < fun_min_value) fun_min_value = f;
        // пронаћи минимум од свих ново насталих чворова у току
        // једне итерације
14 solution.pop_back();
15 return fun_min_value // минимална вредност функције свих чворова
                        // текуће итерације се узима као вредност
                        // function_limit_cost за следећу итерацију
```

3.2.4 Хеуристике за решавање клизне слагалице

Хеуристика је битна компонента алгоритама са информисаном претрагом. Што је тачнија хеуристика то је потребно претражити мањи простор и алгоритам се брже извршава. Када би на располагању била савршена хеуристика (она која даје увек тачну минималну вредност између два чвора), алгоритам се усмерава право према циљу. Нажалост да би се одредила тачна удаљеност између два чвора, потребно је одредити најкраћу удаљеност између њих а то је управо што тражимо. За несавршену хеуристику алгоритам се понаша другачије, у зависности од тога да ли је вредност хеуристике мања или већа у односту на реални број корака потребних за достизања решења.

Хеуристичка функција је прихватљива уколико је цена достизања циљног чвора из чвора n мања од стварног растојања. Ако је хеуристика мања у односу на реалан број корака потребних за решавање проблема, постоји могућнст да се алгоритам извршава нешто дуже. То повећава време претраживања до циља али омогућава проналажење оптималног решења. Цена израчунавања функције процене може да буде већа од уштеде у времену претраживања. Мора се водити рачуна о компромису између добитка и губитка који прозилази из једне такве примене. Ако је хеуристика већа од реалне, тако да прецењује стварну удаљеност између два чвора, постоји могућност да алгоритам не врати најбољи пут. Алгоритам даје предност чворовима који имају мању удаљеност до циља, што усмерава потрагу према циљу брже, али с изгледом да не нађе најбољи пут до циља. То значи да укупна дужина пута може бити већа од оптималног пута. Срећом, то не имплицира да даје лоше путеве. Може се показати да ако хеуристика прецењује за највише x (x је највећа вредност којом прецењује чвор), тада је коначни пронађени пут за највише x дужи од најбољег. Прецењена хеуристика се понекад назива и „недопустива хеуристика“, дакле њу не можемо користити, а односи се на то да алгоритам више не враћа најкраћи пут [2]. Добро одмерена хеуристичка функција може да послужи као добар водич у процесу претраживања чинећи га ефикаснијим. Како хеуристика прецењује све више, перформансе алгоритма рапидно постају све лошије.

У наставку следи списак хеуристика које се могу користити за решавање клизне слагалице, видети [5].

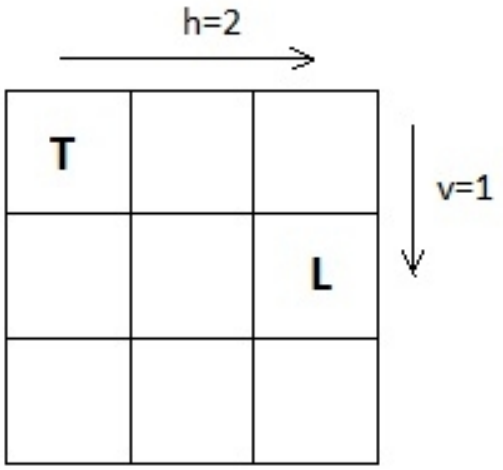
Недостајуће плочице

Хеуристика претпостаља да се плочице могу померати директно на циљну позицију без

обзира да ли је позиција слободна или не. Број корака у овом случају потребних да би решили било коју конфигурацију применом ове хеуристике јесте број плочица које нису на својој циљној позицији.

Менхетн растојање

Менхетн растојање представља суму хоризонталних и вертикалних корака свих плочица да би дошле до своје циљне позиције. Сврстава се у прихватљиве хеуристике.

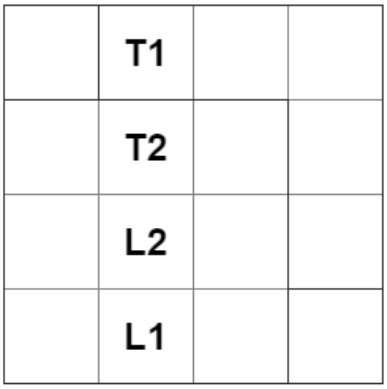


Слика 5. Пример Менхетн растојања једне плочице

У примеру на слици 5 ознаком T означена је почетна позиција плочице, а L представља циљну позицију плочице. Растојање од почетне до циљне позиције плочице представља збир хоризонталних и вертикалних корака. У примеру на слици вредност хеуристике је $h+v=3$.

Менхетн растојање и линеарни конфликт

Ово је корекција Менхетн растојања, начин на који можемо побољшати њену тачност, одржавајући њену строгу прихватљивост. Када се плочице налазе у истој врсти, а такође и њихове циљне позиције су у истој тој врсти, само је погрешан редослед, онда се овај феномен назива лиенарни конфликт. Исто важи ако су плочице у истој колони, видети пример на слици 6. Да би плочице замениле места, једна од плочица се мора уклонити из те колоне, пре него што се врати у ту колону (на другој страни друге конфликтне плочице). Сваки пут када дође до ове ситуације додаје се најмање два потеза на Менхетн растојање за сваки конфликт



Слика 6. Пример плочица у линеаром конфликту

У примеру на слици 6 ознакама T1 и T2 означене су почетне позиције плочица, а L1 и L2 представљају њихове циљне позиције респективно. Плочице T1 и T2 су у тачној колони, али у погрешној врсти према њиховим циљним позицијама. Менхетн растојање плочице T1 и T2 су 3 и 1 респективно. Плочица T2 може бити померена право на своју позицију L2 без компликација. Да би плочица T1 достигла своју циљну позицију L1, плочица T1 или T2 мора бити померена ван тренутне колоне да омогући пролаз другој, пре него што се врати назад у колону. Сходно томе у случају линеарног конфликта, приказаног на слици 6, најмање два додатна потеза ће бити додата Менхетн растојању за две конфликтне плочице. Веома је битно додати 2 додатна потеза Менхетн растојању за само једну конфликтну плочицу овог линеарног конфликта. Ако се додају 2 додатна потеза за обе конфликтне плочице хеуристика више неће бити прихватљива. У овој ситуацији Менхетн растојање плус линеарни конфликт је 3 + 1 + 2 + 6 .

Шаблони

Шаблони представљају хеуристику која је први пут била примењена приликом решавања клизних слагалица. Клизна слагалица са n плочица има n!/2 решивих конфигурација. Уколико би било довољно меморијског простора, за сваку решиву конфигурацију може се сачувати минимални број потеза потребних за њено решавање. У меморији се чува као пар, конфигурација слагалице и број потеза за њено решавање. Оваква хеуристика би била најпрецизнија, јер даје тачан број корака за решавање сваке конфигурације, што би током претраге довело до мањег броје претражених чворова и временски бржег налажења решења. Главни проблем је што је овакав начин меморијски скуп и непрактичан.

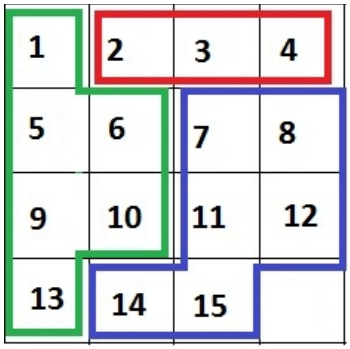
Решење је да уместо чувања свих n!/2 различитих конфигурација и броја потеза за

довођење свих плочица на циљну позицију, чува $\frac{n!}{(n-r)!}$ (где је r број посматраних плочица) различитих конфигурација и број потеза за довођење r плочица на циљну позицију. У табели се чувају пар: конфигурација, број корака за решавање, при чему је свака конфигурација јединствено одређена распоредом плочица. Не разматра се кретање плочица које не припадају посматраној групи плочица. Група плочица се често означава као шаблон. Један пример шаблона је група плочица означена црвеном бојом на слици 7.

За одређивање свих позиција у коме могу да се нађу плочице из шаблона, потребно је да се изврши претрага у дубину почевши од решене конфигурације. Приликом претраге памти се број корака приликом померања плочица које припадају шаблону, а остале плочице се сматрају неозначене то јест оне које не припадају шаблону, тако да је конфигурација јединствено одређена распоредом плочица које припадају шаблону. Приликом тога добијамо конфигурације које садрже све различите позиције плочице шаблона и број корака за њихово довођење на циљну позицију.

За сваку конфигурацију слагалице, збир броја потеза потребних за довођење сваке плочице једног шаблона на своје циљне позиције, представља доњу границу за решавање клизне слагалице што чини ову хеуристику прихватљивом [13].

Слагалица може да се подели у одређени број шаблона, као на примеру приказаном на слици 7. Број шаблона и њихов распоред може бити различит. За сваки од шаблона имамо унапред израчунати број потеза потребних за довођење плочица на циљну позицију. Као вредност хеуристике, можемо узети максималну вредност броја потеза датих шаблона, њихов збир или узети вредност само једног од шаблона.



Слика 7. Подела слагалице у три различита подскупа

Можемо, унапред, израчунати све ове вредности, сачувати у меморији и током претраге искористити. Број пермутација клизне слагалице димензије n је n!, а број могућих пермутација у случају подскупа величине r је $\frac{n!}{(n-r)!}$. Уколико посматрамо шаблон означен зеленом бојом на слици 7. онда је број могућих пермутација плочица из посматраног шаблона $\frac{16!}{(16-6)!}=5765760$. За сваку пермутацију, сачувамо број потребних потеза за довођење плочица из шаблона на циљну позицију. То урадимо за сваки од шаблона [3].

У зависности од броја и позиције плочица које се одаберу као шаблон, као и начина како се рачуна коначна вредност хеуристике разликују се неадитивни и адитивни шаблони.

Неадитивни шаблони

Фелнер [7] описује неадитивне базе података као базе које једноставно чувају за сваки шаблон у бази број потребних потеза да би поставили плочице из шаблона на циљне позиције рачунајући и померања плочица које не припадају посматраном шаблону. Зато што узима у обзир кретања плочица које не припадају шаблону, кретања плочица у шаблону могу ометати кретања плочица из другог шаблона. Да би комбиновали два одвојена шаблона у хеуристику, морамо узети максималну вредност те две вредности јер се у шаблонима налазе плочице које се преклапају.

Уколико шаблон садржи велики број плочица, процена цена ће бити ближа стварним ценама. Цена тога је већа меморија базе и више времена потребног за добијање свих пермутација. Све пермутације могу бити произведене у релативном кратком временском периоду за шаблоне мањих димензија, које садрже 7 или 8 плочица.

Адитивни шаблони (Additive Patterns database heuristics)

Корф (енг. Richard E. Korf) [7] описује једну врсту шаблона базе података која се може користити као ефективна алетернитива која се зове „раздвојени шаблони базе података“. За разлику од неадитивних шаблона, овде се рачунају само кретања плочица које се налазе у оквиру шаблона. Као резултат тога изабере се одређени број шаблона тако да покрију целу клизну слагалицу и да се не преклапају. Слагалица се решава тако што се за сваку комбинацију плочица обухваћену шаблоном пронађе у бази података потребан број корака за њено решавање. Када се добије број корака за сваки од шаблона, њихов збир представља вредност хеуристике.

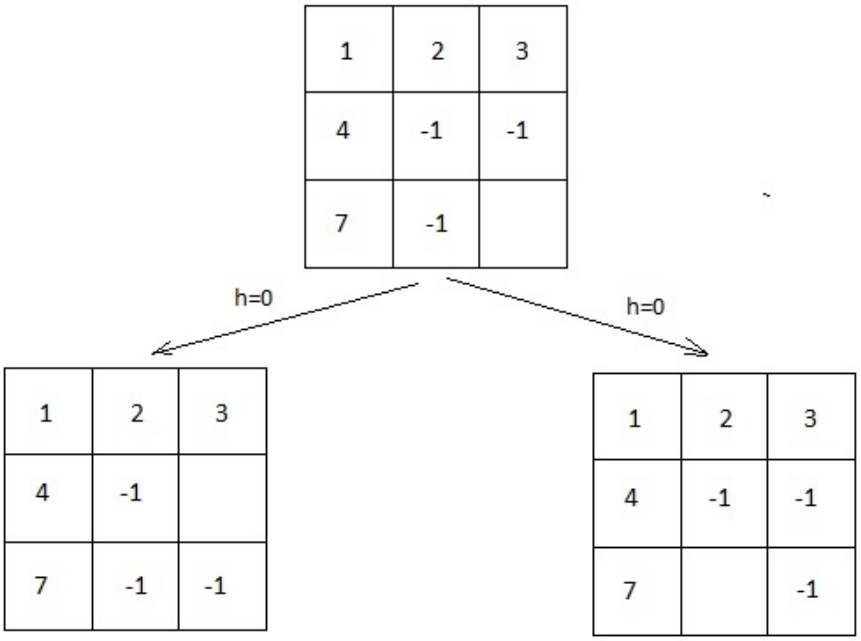
Алгоритам за прављење хеуристике база података шаблона

Потребно је одредити шаблон који посматрамо. Код слагалице где је n једнако 9, као један од шаблона узмемо следеће вредности „1,2,3,4,7“ за први шаблон и „5,6,8“ за други шаблон. На остала поља ставимо -1, број који се не налази у слагалици и њега не узимамо у обзир, као на примеру на слици 8.

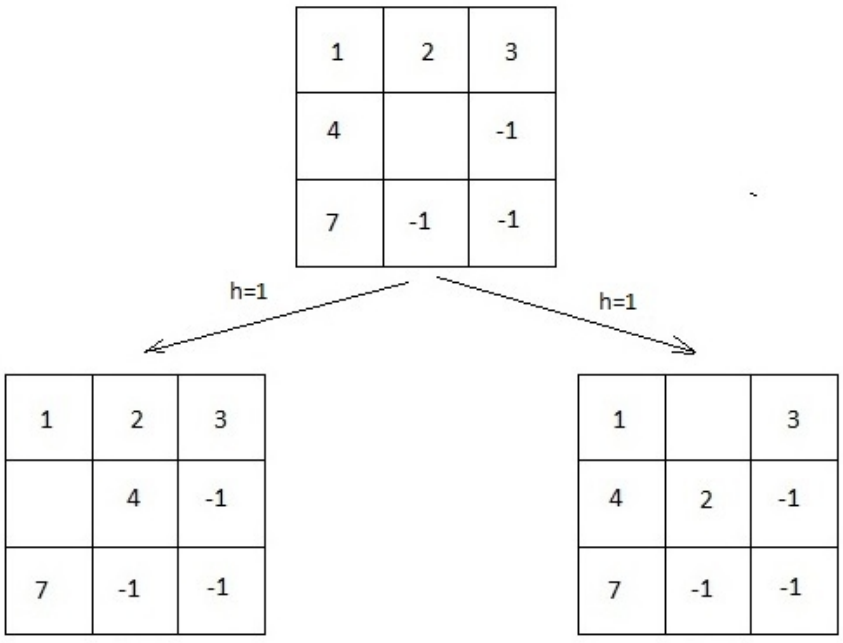
1	2	3
4	-1	-1
7	-1	

Слика 8. Шаблон „1,2,3,4,7“ клизне слагалице

Претрагом у ширину почевши од циљне конфигурације, генеришемо све могуће конфигурације из којих можемо добити ову посматрану конфигурацију једног од шаблона. За све те конфигурације одредимо током претраживања колико нам је корака потребно да би дошли до ње.Уколико приликом померања празна плочица замени место са плочицом ван посматаног шаблона као у примеру на слици 9, која је означена са -1, не повећавамо вредност хеуристике за 1 јер је дошло до померања плочица ван шаблона. Уколико празна плочица замени место са плочицом која се налази у оквиру шаблона као на примеру на слици 10, у овом случају са 1,2,3,4,7 повећава се вредност хеуристике за 1.



Слика 9. Померање плочица које не припадају шаблону



Слика 10. Померање плочица у оквиру шаблона

У бази података чувамо ту конфигурацију као и број потеза који је потребан да би дошли до посматраног шаблона. Када се покрене алгоритам претраживања и када добије слагалицу, приступимо базама, прочитамо број потеза потребних за решавања одговарајућих шаблона и њихов збир представља хеуристику. У случају $n=16$ шаблони представљају 4 врсте слагалице без празне плочице. Величина шаблона је веома важна јер одређује време и меморију потребну за чување у бази, као и брзину одређивања тачног решења. Што је шаблон већи, више меморије и времена је потребно за генерисање базе и брзина проналажења решења је већа.

Број плочица у погрешној врсти и колони - Рачуна број плочица које се налазе у погрешној врсти и погрешној колони.

Све горе наведене хеуристике представљају прихватљиве хеуристике које показују добре резултате на слагалицама димензије 3 и 4.

3.3 Генетски алгоритам

Еволуција је процес прилагођавања живих бића на услове у околини. Да би нека врста током еволуције опстала, она се мора прилагођавати различитим условима околине која се стално мењају. Добра својства јединки као што су отпорност на различите болести, способност

трчања итд. помажу јединкама у непрестаној борби за опстанак и очување популације.

Претпоставља се да су сва својства јединки записана у хромозомима, које представљају ланчане структуре које се налазе у језгру ћелије. Свака ћелија садржи све информације о својствима јединки. Скуп информација које карактеришу једно својство записано је у једном делићу хромозома који се назива ген. Хромозоми долазе увек у паровима, један од оца а један од мајке, што значи да за свако својство постоје два гена, а доминантност гена одређује који ће се испољити.

Генетика (грчки γεννώ - гено, значи дати род, родити) је наука која проучава гене, наследност и варијацију између организама. Реч генетика је први пут употребљена када је Енглески научник Вилијам Бејтсон послао писмо Адаму Седвику, 1905. године у којем је описао науку која за циљ има проучавање процеса наслеђивања и варијације између организама.

3.3.1 Како раде генетски алгоритми

Генетски алгоритми предложени су од стране Џона Х. Холанда (енг. John Henry Holland) још у раним седамдесетим. Током нешто више од 20 година показали су се врло моћним и, у исто време, општим алатом за решавање низа различитих проблема. То се може објаснити њиховом једноставношћу, почевши од саме идеје како су настали и њиховој примени и прилагођавању великом броју различитих проблема. По начину рада убрајају се у методе случајног претраживања простора решења у потрази за глобалним максимумом. Генетски алгоритми могу одредити положај глобалног максимума у простору с више локалних екстремума. Нису зависни од почетне тачке претраживања и могу својим поступком претраживања са одређеном вероватноћом лоцирати глобални максимум. За разлику од класичних алгоритама за које са сигурношћу можемо рећи да је постигнут глобални максимум, за генетске алгоритме не можемо са сигурношћу рећи да ли је постигнут глобални или локални максимум. Сигурност добијених резултата значајно се повећава поступком понављања процеса решавања, што код класичних метода нема смисла [8].

Основна јединица генетског алгоритма јесте јединка, а скуп јединки представља популацију. Јединка је кодирана хромозомима, који могу бити представљени различитим структурама података (матрица, низ). За сваки хромозом се проверава колико је близу решења, што се изражава вредношћу циљне функције. Из старе популације се ствара нова, тако што се поступком којим се имитира природна селекција бирају јединке које ће преживети, то јест које имају највећу вредност циљне функције. На припадницима нове генерације примењују се генетски оператори, који се деле на унарне, који делују на један хромозом, и операторе вишег реда који стварају нове хромозоме комбинацијом хромозома неколико старих хромозома родитеља. Цео поступак одређивања вредности циљне функције, селекције и примена генетских алгоритама се примењује док не буде испуњен одређени услов заустављања. Пример рада генетског алгоритма састоји се од неколико корака. На почетку алгоритма се генерише почетна популација потенцијалних решења. Све док није задовољен услов завршетка еволуцијског процеса врши се селекција, укрштање и мутација бољих јединки.

3.3.2 Иницијализација популације

Јединка представља једну клизну слагалицу, а популација представља низ јединки то јест низ слагалица. Униформно генерисање почетне популације је поступак којим се генерише почетна популација у којој су све јединке исте конфигурације. Касније се захваљујући мутацији појављују нова својства. Популација створена на овај начин у почетку се споро приближава решењу.

Неуниформна генерација се може створити на више начина, случајним генерисањем јединки или стварањем популације од неких раније добијених добрих јединки, које се онда алгоритмом усавршавају, или неком комбинацијом ових начина.

Величина популације је овде константа и садржи 100 решења које чува у вектору и који представљају исту почетну конфигурацију слагалице. Покушава се једино примењивањем оператора мутације да се дође до решења пре почетка претраживања. Понекад је могуће још у почетној фази да се стигне до решења, али то је у случају када је почетна конфигурација веома близу циљног решења.

3.3.3 Селекција

Селекција је процес којим генетски алгоритам чува добре, а одбацује лоше јединке из популације решења. Селекцијом се бирају јединке које ће учествовати у репродукцији и тако пренети свој генетски материјал на следећу генерацију. Како и лоше јединке могу садржати добре и корисне гене, потребно је и њима омогућити барем минималну вероватноћу за размножавање. Боље јединке, наравно, треба да добију већу вероватноћу размножавања.

Према начину преношења генетског материјала бољих јединки у следећу итерацију поступци селекције се деле на:

- генерацијске селекције - селекцијом се директно бирају боље јединке чији ће се генетски материјал пренети у следећу итерацију и
- елиминацијске селекције - бирају се лоше јединке за елиминацију, а боље јединке преживе поступак селекције.

Пре примене селекције врши се сортирање на основу функције вредновања. Најбоље јединке се касније користе приликом укрштања [9].

3.3.4 Укрштање

Укрштање је генетски оператор који имитира природни процес укрштања (crossover). У њему учествују две јединке из популације, па се укрштањем њиховог генетског материјала добијају једна или две нове јединке. Новонастала јединка настаје из две добре јединке, које су прошле селекцију, па би њихова својства требало да буду добра. Према томе, нова јединка би такође требало да буде добра. Поступком укрштања се брзо постиже приближавање оптимуму.

Метода која се користи за укрштање је на једној тачки. У елитном хромозому се итерира од тренутног ка почетном стању. За сваку позицију претрага се врши у слабијем хромозому са поклапајућом позицијом у елитном. Лева половина новог хромозома биће из слабијег а десна из елитног [9].

3.3.5 Мутација

У алгоритму који је имплементиран постоје 4 различите врсте мутација. Случајним избором се одређује која ће од четири наведених мутација бити примењена [9].

1. Додаје се још један нови потез како би се побољшала вредност функције вредновања. Бира се конфигурација са најбољом функцијом вредновања.
2. Додаје се још један нови потез тако што се изабере конфигурација са одређеном вероватноћом.
3. Замени се последњи елемент новом конфигурацијом која се добија од случајно конфигурација добијених из претпоследњег елемента.
4. Уколико је листа дужа од неке одређене вредности врши се одсецање.

Функција вредновања

Као функција вредновања користи се хеуристика Менхетн растојање и недостајуће плочице. Менхетн растојање као и недостајуће плочице објашњене су у поглављу о хеуристикама 3.2.4 . Вредност се рачуна на основу израза: Fitness = 1 - get_manhattan_distance() * .01 - getMisPlaced() * .01. Уколико је вредност функције вредновања једнака 1 значи да се се свака плочица налази на циљној позицији. Обе хеуристике множе се са истим фактором 0.01 с тим да ће већи утицај имати менхетн растојање јер има већу вредност. Фактор може бити промењен у циљу добијања бољих резултата или уколико желимо да нека хеуристика има већи или мањи утицај [8].

Мерење различитости

Свака индивидуална јединка у популацији се анализира и броји да би се одредио број јединствених решења у популацији. Јединствена решења представљају она чије су конфигурације слагалице исте. Ово се ради једноставном методом коришћењем мапе. За популацију у свакој генерацији ова вредност се израчуна унапред [8].

Контролисање различитости

За контролу различитости користе се два параметра: максималан број дупликата и хипер мутација. Пре него што се примене оператори мутације и укрштања на популацију одреди се број јединствених решења.
Ако је број решења у популацији већи од максимално дозвољених, на ове хромозоме се примењује хипер мутација. Хипер мутација представља позивање оператора мутације n пута, како не би постојало дупликата решења у оквиру популације [9].

3.3.6 Предности генетских алгоритама

Генетски алгоритми су примењиви на велики број различитих проблема [8,9].

1. Структура алгоритма нуди велике могућности надоградње и повећања ефикасности алгоритама једноставним захватима (већи степен слободе).
2. Једноставним понављањем поступака се може повећати поузданост резултата.
3. Ако већ не нађе решење (глобални оптимум), даје неко добро решење којим се може задовољити
4. Као резултат може дати скуп решења, а не једно решење
5. Решава све проблеме који се могу представити као оптимизација, без обзира да ли функција коју треба оптимизовати има за аргументе реалне бројеве или битове, знакове или било коју врсту информације
6. Врло једноставно је применљив на вишемодалним проблемима
7. Једноставност идеје и доступност програмске подршке

3.4 Симулирано каљење

Алгоритам симулираног каљења је настао по аналогији са металуршким каљењем, чији је циљ оплемењивање метала како би постао чвршћи. Како би метал био што чвршћи потребно је да кристална решетка метала има минималну потенцијалну енергију. У процесу металуршког каљења метал се прво загрева до високе температуре а потом се након краћег задржавања на тој температури, полагано хлади до собне температуре. Пребрзо хлађење може изазвати пуцање метала. Последица каљења је да атоми метала након процеса каљења формирају правилну решетку, чиме се постиже енергетски минимум решетке.

Алгоритам симулираног каљења као параметре садржи тренутну температуру, минималну температуру, фактор хлађења као и функцију којој желимо одредити глобални максимум. Алгоритам изабере на почетку неко почетно решење, а почетна температура има велику вредност. Постојеће решење се замењује бољим уз одређену вероватноћу. Вероватноћа се одређује избором случајног броја *random* из интервала [0,100) и уколико важи да је услов тачан $random < \exp(E/T) * 100$, где је E функција за коју се тражи глобални максимум а T температура, решење се прихвата. Вероватноћа да ће бити лошије решење је веће када је већа температура, што значи да је простор претраге на почетку велики и да се смањује са падом температуре и на крају процеса је уско локализован. Температура експоненацијално опада са параметром *random* из интервала [0,100) када је близу максималној вредности. За премало *random* температура брзо опада и упада у локални минумум, а за превелико *random* температура преспоро опада и претражује велики простор решења, што успорава рад и зато треба одабрати оптималну величину вредности а да температура при крају процеса буде ниска. Треба пажљиво одабрати вредност променљиве *random* .

Постоје два критеријума за заустављање алгоритма. Први је да температура достигне нулу, а други да решење буде пронађено. Хеуристика која се користи код алгоритма симулираног

кађења је Менхетн растојање. Уколико је вредност хеуристике новог чвора successor = generateRandomPuzzle(p) већа од хеуристике родитељског чвора обрађујемо нови чвор. Уколико је вредност хеуристике новог чвора већи од хеуристике родитељског чвора онда се одређује да ли ће вредост бити узета на основу формуле $random < \exp(E/T) \times 100$. У сваком кораку вредност тренутне температуре се смањује тако да се не додаје у листу решења ново добијени чвор [3].

Алгоритам: Симулирано кађење
Улаз: клизна слагалица P
Излаз: нема

```
// тренутна температура curTemperature
// фактор хлађења coolingRate
// минимална температура minTemperature
// generateRandomPuzzle Генерише се једна конфигурација слагалице на основу улазне
// izlaz низ слагалица које воде до решења
// meetResult да ли је решење пронађено

  1 curTemperature = 100.0
  2 coolingRate = 0.9999
  3 minTemperature = 0.0001
  4
  5 while (curTemperature > minTemperature)
  6      if (p.getHeuristic() == 0) { meetResult = true; } крај
    алгоритма
  7      successor = generateRandomPuzzle(p)
  8      if (successor.getHeuristic() < p.getHeuristic())
  9           p = successor
 10           izlaz.push_back(toString(p))
 11      else
 12           E = p.getHeuristic() - successor.getHeuristic()
 13           T = curTemperature
 14           random = rand() % 100;
 15           if (random < exp(E / T) * 100)
 16                p = successor
 17                izlaz.push_back(toString(p))
 18      curTemperature *= coolingRate
 19 meetResult = false;
```

3.5 Рекурзивни алгоритам за решавање великих слагалица

Алгоритам о коме је реч [10] је заснован на техникама подели па владај и алгоритмима похлепне

претраге. Гарантује да је време извршавања $O(n^3)$ где n представља ширину слагалице. Основна идеја алгоритма је да се прво реши прва врста, онда прва колона и да се алгоритам рекурзивно позове за остатак слагалице ширине $n-1$.
Овај алгоритам има следеће особине:

- Време извршавања потребно за сваки корак је $O(1)$, као и време израчунавања за сваки наредни потез
- Прави највише 5 пута више корака него што је потребно у случају најгоре конфигурације (конфигурација која захтева највећи број потеза за решавање)
- Прави највише 7.5 пута више корака него што је потребно у случају просечне конфигурације

3.5.1 Опис алгоритма

Означимо плочицу која се налази у i-тој врсти и j-тој колони као (i, j). Похлепни алгоритам се заснива на томе да поставља плочице $(1,k)$ за $1 \le k \le n$ на место, а онда плочице $(k,1)$ за $2 \le k \le n$ на место, једну по једну. На тај начин је решена прва врста и колона. Алгоритам се рекурзивно позива за слагалицу мање димензије $n-1$, решавајући увек прву врсту и колону, све док не буде преостала слагалица димензије 2x2 чијим се решавањем излази из рекурзије. Приликом решавања слагалице користећи потезе за померање плочице дијагонално, померамо изабрану плочицу у одговарајућу врсту или колону.

Постоји низ од 6 корака потребних да померимо плочицу једно место дијагонално као што је приказано на слици 11. Плочица означена - је празна, а плочица означена са x циљна плочица коју желимо да померимо. Прво доведемо празну плочицу изнад циљне плочице, како би циљну плочицу довели у тачну врсту њиховом заменом места. На слици 12 приказано је како у пет корака померити плочицу једно место вертикално или хоризонтално у зависности од позиције плочице. Помери се празна плочица на позицији изнад циљне плочице и затим се изврши њихова замена места. На крају се празна плочица премести десно од циљне позиције и наставља се са премештањем осталих плочица на циљно место. Уколико је потребно померити плочицу за неколико позиција потребно је само поновити претходно описане кораке.

abc	a-b	axb	axb	-xb	x-b
dx-	dxc	d-c	-dc	adc	adc

Слика 11. Померање празне плочице за једну позицију дијагонално у 6 потеза

ab	a-	-a	xa	xa	x-
x-	xb	xb	-b	b-	ba

Слика 12. Померање празне плочице за једну позицију вертикално у 5 потеза

Алгоритам: `Solve()`

Улаз: нема
Излаз: број корака

```
// Remap смањује се димензија слагалице наког решавања прве врсте и колоне
// PuzzleSize ширина слагалице
// emptyPosition позиција празне плочице
// moveTile померање празне плочице, L-лево, R-десно, U-горе, D-доле

1  for(int i=0; i<PuzzleSize * PuzzleSize; i++)
2       Blocked[i] = false; // плочице које су на својој позицији
    не могу бити померене, на                          //  почетку све
    плочице нису блокиране и могу се померати
3  for(int i=PuzzleSize; i>2; i--){
4       SolveFirstRow(i);
5       SolveFirstCol(i);
6       Remap(i-1);
7  }
8
9  // решавање слагалице 2x2
10 if(emptyPosition == 0){
11      moveTile("RD");
12 }
13 if(emptyPosition == 1){
14      moveTile("D");
15 }
16 if(emptyPosition == 2){
17      moveTile("R");
18 }
19
```

Алгоритам: SolveFirstRow
Улаз: Ширина слагалице n
Излаз: нема

```
// SolveFirstRow, Решавање прве врсте
// MoveTileUpTo(i,i) коришћењем покрета са слике 6. плочица се помера у одговарајућу врсту

1  for(int i=0; i<n-2; i++){
2       if(m_nPositionToTile[i] != i)
3            MoveTileUpTo(i, i);
4       m_bLocked[i] = true;
5  }
```

Алгоритам: Решавање прве колоне, SolveFirstCol
Улаз: Ширина слагалице n
Излаз: нема

```
1  for(int i=1; i<n-2; i++){
2       const int tile = i*n;
3       if(m_nPositionToTile[tile] != tile){
4            MoveTileLeftTo(tile, tile);
5       }
6       m_bLocked[tile] = true;
7  }
```

4 Програмска реализација и добијени резултати

У овом делу изложени су детаљи имплементације алгоритама, као и експериментални резултати добијени њиховом применом.

4.1 Имплементација алгоритама

У овом делу представљене су програмске реализације алгоритама. Алгоритми су имплементирани у различитим програмима. Као посебни програми имплементирани су рекурзивни алгоритам за решавање слагалица, генетски алгоритам, IDA* алгоритам, симулирано каљење, док су остали алгоритми имплементирани у оквиру једног програма.

У оквиру програма где је имплементирана већина алгоритама структура је следећа. Класа search_factory представља основну класу коју наслеђују друге класе, у којој су имлементиране основне методе за покретање претаге, као и операција са отвореним и затвореним листама, бројање итерација. Отворена листа је имплементирана као ред одакле се узимају елементи. Затворена листа реализована је као скуп тако да се слагалице са одређеном конфигурацијом обрађују једном. У функцији search као аргументи прослеђују се почетна конфигурација слагалице и циљна, а као резултат претраге низ конфигурација које воде до решења.

```
#pragma once
#include "Puzzle.h"
#include "basic_queue.h"
#include <set>
using namespace std;
class search_factory {
public:
    search_factory();
    virtual ~search_factory(){ delete this->closed_set; };
    virtual void search(Puzzle *, Puzzle *, Puzzle *&)=0;
    int get_open_set_size() { return this->open_set->size(); }
    int get_closed_set_size() { return this->closed_set->size(); }
    int get_iterations();
```

```
protected:
    basic_queue<Puzzle *> *open_set;
    set<hash_id> *closed_set;
    int iterations;
};
```

Информисана informed_search_factory и неинформисана uninformed_search_factory претрага наслеђују претходну search_factory класу као основну и имплементирају одговарајуће методе. Алгоритми информисане претраге отворену листу имплементирају као приоритетни ред. Од компаратора који му је прослеђен зависи како ће елементи реда бита сортирани. Функција како ће елементи реда бити сортирани изабрана је тако да слагалица са највећом вредношћу хеуристичке функције буде на врху.

```
template <class T, class C>
modified_queue<T, C>::modified_queue() {
     this->s = new std::priority_queue<T, vector<T>, C>();
}
```

Информисана претрага садржи показивач на одговарајућу хеуристику. Помоћу методе set_heuristic можемо поставити било коју од имплементираних хеуристика. Класу неинформисане претраге наслеђују касније претрага у ширину, дубину, претрага у дубину итеративним продубљивањем, униформна претрага. Информисану претрагу насеђују A*, похлепна и униформна претрага.

Класа heuristic представља основну класу која садржи метод get_heuristic који имплементирају све хеуристике које насеђују ову класу као што су Менхетн, линеарни конфилкти и други.

```
#ifndef HEURISTIC_H
#define HEURISTIC_H
#include "Puzzle.h"
class heuristic {
protected:
    Puzzle *goal_s;
public:
    heuristic(Puzzle *goal_data);
    virtual ~heuristic() {};
    virtual int get_heuristic(Puzzle *init_data) = 0;
};

#endif
```

Представљен је само најважнији део како је имплементирана слагалица, то јест само њене променљиве. Слагалица се чува као низ бројева puzzle_data. Показивач на родитеља puzzle_parent предстаља показивач на слагалицу чијим је померањем празне плочице настала

дата слагалица, што помаже приликом реконструисања низа потеза који воде од почетне до циљне слагалице.

```
hash_id state_identifier;     // позитиван 32 битни јединствени број
unsigned int *puzzle_data;              // низ плочица слагалице
unsigned int gap_position;              // позиција нуле
int path_cost;                          // цена путање
int heur_value;                         // вредност хеуристике
Puzzle *puzzle_parent;                  // показивач на родитеља
state_action_direction move_action;     // правац померања
```

Када се програм покрене понуди се који алгоритам ће бити покренут, а на основу тога ако је алгоритам информисане претраге биће понуђен избор хеуристика.

Генетски алгоритам

Параметри који се користе у генетском алгоритму су следећи:

величина популације: 500
број генерација: 500
вероватноћа мутације: 1.0
вероватноћа укрштања: 0.10
случајна вредност: rand(0)
почетна величина решења: 30

Класа Population чува решења. Омогућава позивање метода укрштања и мутације над решењима. Вектор solutions чува низ решења чији број сами дефинишемо и величина популације може бити 500.

```
class Population
{
private:
  int id;
    vector < Solution > solutions;
    map < string, int >uniques;
  double average_dist_value;
  double average_mean_value;
  double average_fitness_value;

Пример метода мутације у класи population.
Population::mutate_puzzle (int mrate)
{
  for (int i = 0; i < solutions.size (); i++)
    {
      int r = rand () % mrate;
```

```
      if (r == 0)
     {
       r = rand () % 4;
       if (r == 0)
         solutions[i].grow_new_puzzleBestNoCycle ();
       if (r == 1)
         solutions[i].grow_new_puzzleNoCycle ();
       if (r == 2)
         solutions[i].mutate_puzzle ();
       if (r == 3)
         solutions[i].truncate ();
     }
    }
}
```

Опис мутације дат је у претходном делу.

Класа solution чува решења. Садржи основне методе као што су мутација и укрштање, као и методе за добијање и проверавање детаља решења. Листа се користи за чување решења, као и чување новодобијених конфигурација слагалица приликом позива мутације или неког другог метода. Мапа служи за проверу да ли је дата конфигурација већ генерисана.

```
class Solution
{
private:
  double average_dist_value;
  double average_mean_value;

public:
    multimap < string, int >map;
    deque < string > list;
```

У класи board налази се структура података којом се описује слагалица, као и методи за померање плочица и израчунавање вредности функције прилагођавања.

IDA*

За представљање слагалице користи се typedef std::vector < int >Puzzle_data. Основна метода која се позива за решавање слагалице је std::list < Puzzle > start_solving (const Puzzle & puzzle) која враћа листу слагалица које представљају слагалице то јест кораке које воде до решења.

```
    std::list < Puzzle > start_solving (const Puzzle & puzzle)
    {
      puzzle_solution.clear ();
```

```
      is_solved = false;
      function_limit_cost = puzzle.h ();
      while (!is_solved && function_limit_cost < infinty_value)
     {
       function_limit_cost = run_search_dfs (puzzle, 0);
     }
      return puzzle_solution;
    }
```

Унутар функције solving позива се run_search_dfs који представља најважније методу која DFS претрагом тражи решење.

Рекурзивни алгоритам за решавање слагалица

У класи којом се дефинише структура слагалице, конфигурација слагалице се чува као низ бројева. У наставку текста представља се део класе. Чува се позиција тренутне плочице која се помера, циљна позиција на којој треба да се налази плочица као и позиција празне плочице. Дефинисане су различите методе које омогућавају померање плочица.

```
    int puzzleValues[MAX_SIZE] // Мапирање са броја плочице на њену
позицију

    int processed; // број плочице која се помера
    int processedPosition; // индекс плочице која се помера
    int processedRow;  // врста у којој се налази плочица
    int processedCol;  // колона у којој се налази плочица

    int targetPosition; // циљна позиција плочице
    int targetRow; // циљна врста плочице
    int targetCol;  // циљна колона плочице

    int emptyPosition; // позиција празне плочице
    int emptyRow; // врста празне плочице
    int emptyCol; // колона празне плочице
```

Позивом методе StartPuzzleSolver() покреће се решавање слагалице, решавање прве врсте и колоне, затим се смањује димензија слагалице и реурзивно позива решавање слагалице чија је димензија мања за 1. Када се стигне до слагалице величине два, решење се састоји од унапред познатих потеза за сваку од три различите конфигурације који омогућавају њено решавање.

4.2 Резултати

Приликом тестирања алгоритама коришћен је Linux као оперативни системи. Процесор је i3-

40050U са фреквенцијом 1.70GHz и радном меморијом од 4GB.

Као тестни примери користе се конфигурације које су генерисане на случајан начин. Помоћу претраге у ширину одређено је на којој се дубини од решења налазе конфигурације. Изабране су конфигурације које су на различитим дубинама, то јест, конфигурације за које је потребан оптимални број корака да би се добила циљна конфигурација.

Редни број алгоритма	Назив алгоритма	Скраћенице
1	Претрага у ширину	ПШ
2	Претрага у дубину	ПД
3	Претрага итеративним продубљивањем	ПИП
4	A* недостајуће плочице	A*H
5	A* Менхетн растојање	A*M
6	A* линеарни конфилкт	A*Л
7	A* плочице ван врсте и колоне	A*П
8	A* статички	A*A
9	IDA недостајуће плочице	ИДАН
10	IDA Менхетн растојање	ИДАМ
11	IDA линеарни конфилкт	ИДАЛ
12	IDA плочице ван врсте и колоне	ИДАП
13	IDA статички	ИДАС
14	Похлепни алгоритам недостајуће плочице	ПАН
15	Похлепни алгоритам Менхетн растојање	ПАМ
16	Похлепни алгоритам	ПАЛ

	линеарни конфилкт	
17	Похлепни алгоритам плочице ван врсте и колоне	ПАП
18	Похлепни алгоритам статички	ПАС
19	Генетски алгоритам	ГА
20	Симулирано каљење	СК

Табела 1. Редни број, имена и скраћенице алгоритама који су тестирани

Скраћенице алгоритама	Просечно време за решавање слагалице		Просечан број корака за решавање слагалице	
	n=3	n=4	n=3	n=4
ПШ	1.261	124.4	**20.6**	**31**
ПД	5.537	480.43	*637.2.*	1734
ПИП	3.041	184.4	**11.5**	**31**
А*Н	0.093	72.7	**20.6**	**31**
А*М	0.060	0.239	**20.6**	**31**
А*Л	0.028	0.225	**20.6**	**31**
А*П	2.979	5.639	*33.1*	*101*
А*А	2.958	181.2	**21**	73
ИДАН	0.142	43.058	**20.6**	**31**
ИДАМ	0.079	0.464	**20.6**	**31**
ИДАЛ	0.040	0.137	**20.6**	**31**
ИДАП	3.21&	3.135	*40.4*	*272*
ИДАС	0.009	43.7	23	**35**
ПАН	**0.006**	**0.085**	*88.6*	*313*
ПАМ	**0.002**	**0.063**	*57.6*	*143*

ПАЛ	**0.016**	**0.020**	*41.5*	*101*
ПАП	3.74	0.384	*89.4*	*435*
ПАС	1.157	52.7	22	**35**
ГА	0.43	73.424	*38.7*	**74**
СК	1.261	43.135	*431.6*	*1734*

Табела 2. Просечно време и број корака потребних за решавање слагалица

Резултати у табели број 2 означени подебљаним фонтом представљају најбоље резултате у колони, а са подвученом цртом означени су најлошији.
Резултати показују да се најбрже извршава похлепни, а онда и алгоритам A*. Главни разлог је што су то хеуристички алгоритми са усмереном претрагом приликом претраживања. Уколико се посматрају хеуристике, најбоље резултате дају линеарни конфилкт са Менхетн растојањем као и хеуристика Менхетн растојање и недостајуће плочице. Најлошије резултате према времену извршавања имају алгоритми претрага у дубини, похлепни алгоритам и IDA* које користе хеуристику плочице ван врсте и колоне. Уколико се упоређује просечан број корака потребних за решавање слагалице, најбоље резултате имају A* и IDA*, а најлошије претрага у дубину, похлепни алгоритам са хеуристиком плочице ван врсте и колоне. Оптималан број корака потребних за решавање слагалице димензије 3 је од 0 до 31 корак, а слагалице димензије 4 је од 0 до 80 корака [1]. Косим фонтом у табели означени су резултати чији број корака није оптималан. Генетски алгоритам и симулирано каљење успешно реше око 60-70% слагалица. Ради прегледнијег изгледа табела треба узети ову информацију у обзир. Само ти резултати су узети у обзир.

n	Број корака	Време у секундама
3	0.04	0.000018
4	0.1	0.000031
5	0.2	0.000069
6	0.4	0.000158
7	0.8	0.000281
8	1.1	0.000471
9	1.7	0.000794
10	2.4	0.001266
11	3.3	0.002064

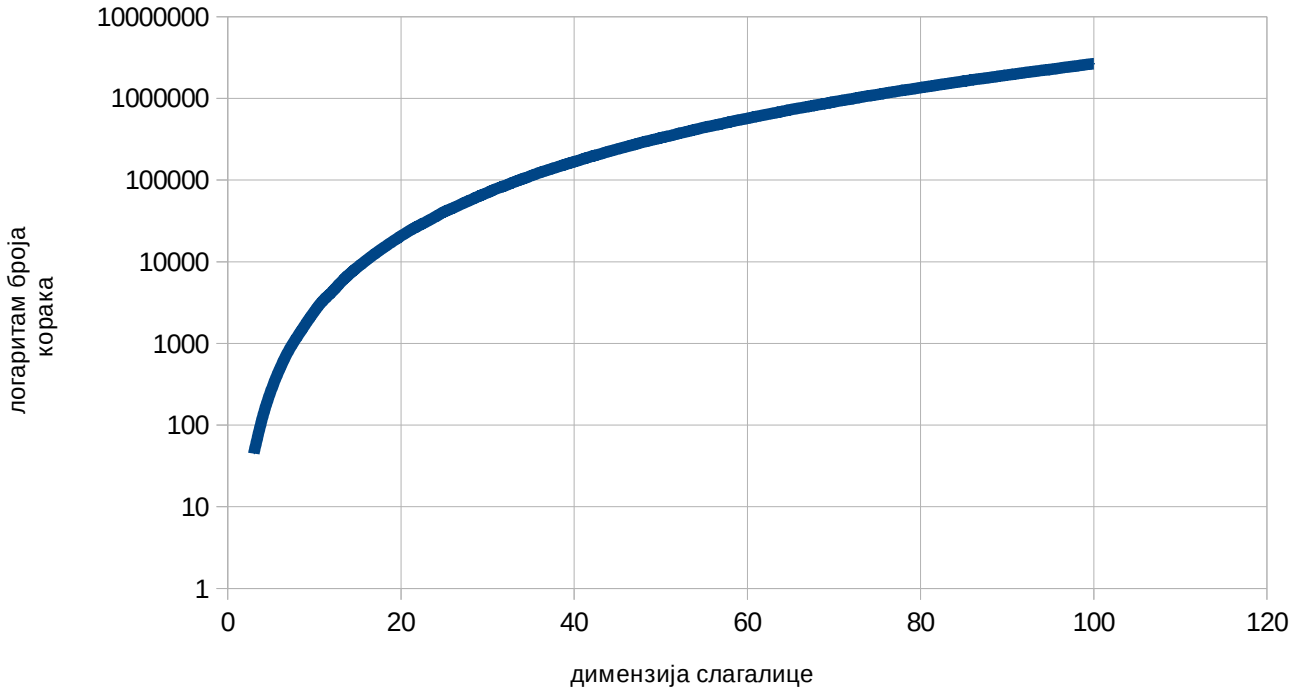
n	Број корака	Време у секундама
52	370.1	3.123615
53	391.5	3.449867
54	415.4	3.843371
55	440.6	4.268095
56	464.1	4.667817
57	488.0	5.085066
58	516.0	5.68039
59	542.8	5.916576
60	569.1	6.371785

12	4.2	0.002867
13	5.5	0.004701
14	6.9	0.006191
15	8.5	0.008359
16	10.4	0.010767
17	12.5	0.014028
18	14.9	0.017883
19	17.5	0.022636
20	20.6	0.029308
21	23.98	0.037051
22	27.4	0.045604
23	31.1	0.056453
24	35.6	0.069176
25	40.8	0.08404
26	45.4	0.101819
27	50.9	0.127528
28	56.8	0.144286
29	63.4	0.168321
30	70.2	0.202253
31	78.1	0.24555
32	85.2	0.280987
33	93.9	0.326078
34	102.9	0.369995
35	112.4	0.438666
36	123.1	0.496453
37	133.0	0.569928
38	144.1	0.663632
39	155.5	0.744731
40	167.6	0.84286
41	180.6	0.925519
42	195.0	1.062822

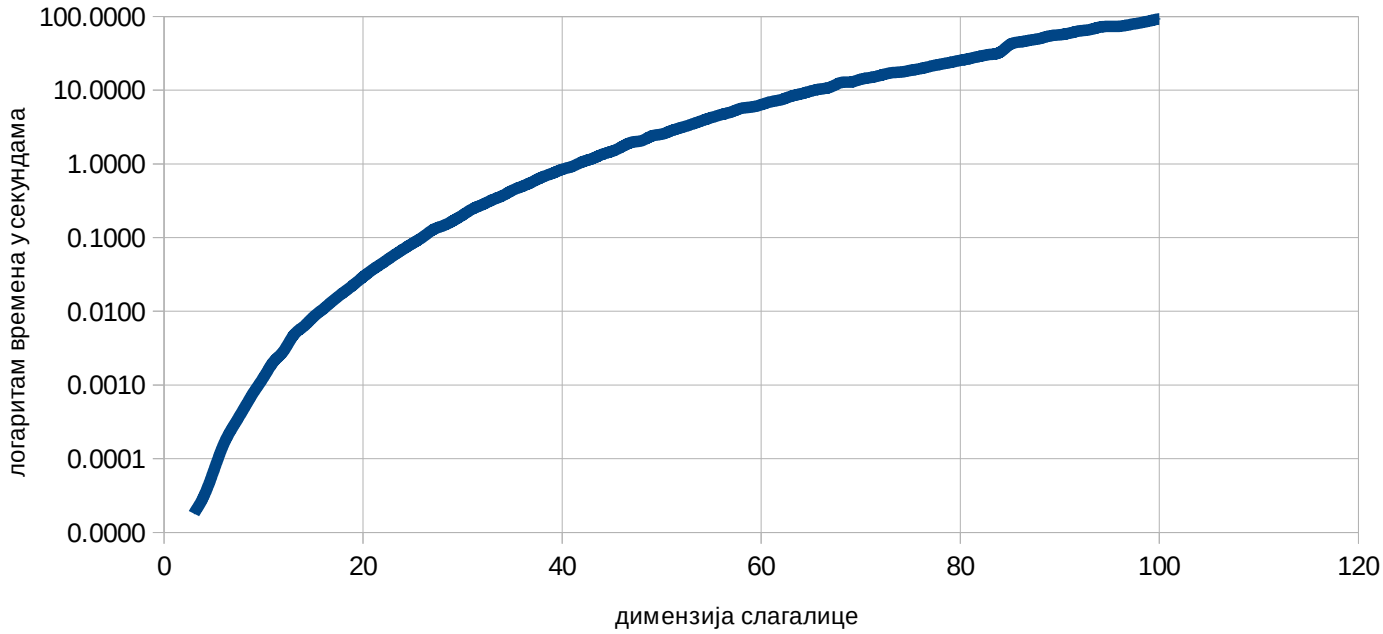
61	599.6	7.007982
62	628.1	7.430666
63	658.9	8.301054
64	691.8	8.938372
65	726.8	9.755607
66	760.8	10.401307
67	792.3	11.085966
68	829.6	12.711702
69	865.5	12.902316
70	907.1	14.111187
71	946.3	14.90842
72	984.1	15.95376
73	1029.3	17.226748
74	1073.5	17.663688
75	1112.4	18.609354
76	1162.6	19.664182
77	1207.8	21.157749
78	1258.9	22.510823
79	1300.0	23.877404
80	1355.2	25.428788
81	1403.2	27.0173
82	1460.8	28.992622
83	1513.6	30.63147
84	1568.6	32.80739
85	1626.1	41.846832
86	1691.0	45.212233
87	1744.5	47.760237
88	1801.2	50.427171
89	1866.0	54.740952
90	1933.4	56.609868
91	1994.4	59.649246

43	208.1	1.177584		92	2066.3	63.98106
44	224.1	1.342793		93	2132.9	66.340042
45	239.5	1.483808		94	2202.1	71.868979
46	255.9	1.708542		95	2263.5	73.641452
47	272.8	1.970535		96	2343.7	73.788454
48	291.9	2.067782		97	2423.0	77.611234
49	309.5	2.393424		98	2496.9	81.71355
50	328.6	2.534679		99	2585.1	87.159151
51	348.0	2.838008		100	2656.9	93.371816

Табела 3. Резултати добијени рекурзивним алгоритмом за решавање великих слагалица



Слика 13. Просечан број корака коришћењем рекурзивног алгортима [10]



Слика 14. Просечно време извршавања коришћењем рекурзивног алгортима [10]

Рекурзивни алгоритам за решавање слагалице показује одличне резултате у односу на остале алгоритме. Време извршавања је приметно боље у односу на друге, али је број корака већи него код осталих алгоритама, уколико поредимо за слагалице димензије $n=3,4$. За разлику од других, има могућност решавања слагалица великих димензија у реалном времену (видети табелу број 3). За $n=100$ просечно време извршавања је око 90 секунди. На слици 13 и 14 види се раст у времену извршавања и броју корака са порастом ширине слагалице. Уколико је на апциси уместо броја корака, логаритам броја корака, нагиб праве је у том случају приближно једнак три.

5 Закључак

Програмски су реализовани различити алгоритми за решавање клизне слагалице. С обзиром на недовољну ефикасност, сви алгоритми осим последњег примењени су само на слагалице димензије највише 4x4. То је последица чињенице да је одређивање најбољег решења слагалице тежак проблем. С друге стране, реализован је рекурзивни алгоритам из рада [10], који без проблема решава слагалице великих димензија, налазећи при томе неоптимална решења.

Клизна слагалица се може решити коришћењем различитих алгоритама и хеуристика. Унапређења која се могу применити су примена других алгоритама и хеуристка које постоје. Додатно убрзање се може постићи коришћењем ефикаснијих структура података као и паралелизацијом самих алгоритама.

Циљ рада је истраживање најефикасније методе за решавање клизне слагалице. У овом раду имплеметирани су алгортми за које сам сматрао да су најефикаснији за решавање овог проблема. A* и IDA* представљају алгоритме које увек проналазе решења са оптималним бројем корака потребних за решавање.

Још један врло важан концепт који сам открио је да када се користи информисана претрага, као што су на пример A* и IDA*, квалитет хеуристике је веома важан за претрагу. Менхетн растојање и линеарни конфликт су две хеуристике које дају добре резултате. Такође на брзину рада утиче и сама кофигурација слагалице, колико је удаљена од решења. Као добар алгоритам показао се A* коришћењем хеуристике линеарни конфликт са Менхетн растојањем.

У раду је изложен и генетски алгоритам који не гарантује налажење оптималног решења али га карактерише велика брзина приликом претраге великог простора решења и показује добре особине у случају када желимо да пронађемо неко решење у кратком временском интервалу.

Описан је и алгоритам симулираног каљења који успешно реши око 60% слагалица, у кратком временском интервалу при чему решења нису оптимална.

Најбољи алгоритам у погледу броја корака за решавање слагалице и времена извршавања јесте рекурзивни алгоритам из рада [10].

Литература

[1] https://en.wikipedia.org/wiki/15_puzzle
[2] Judea Pearl, Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley Publishing Company, 1984.
[3] Stefan Edelkamp, Stefan Schrodl, Heuristic Search Theory and Applications, 2002.
[4] K. Mathew, M. Tabassum, M. Ramakrishnan, Experimental Comparison of Uninformed and Heuristic AI Algorithms for N Puzzle Solution, The Second International Conference on Informatics Engineering & Information Science (ICIEIS2013) - Malaysia, 2013.
[5] Othar Hansson, Andrew E. Mayer, Mordechai M. Yung, Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models, Technical Report 219-85, Dept. of Computer Science, Columbia University, 1985.
[6] George F. Luger, William A. Stubblefield, Artificial Intelligence, Structures and Strategies for Complex Problem Solving, Third Edition, Addison Wesley Longman, 1998.
[7] Ariel Felner, Richard E. Korf, Sarit Hanan, Additive Pattern Database Heuristics, Journal of Artificial Intelligence Research, 22:279–318, 2004.
[8] Mitchell Melanie, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press 1996.
[9] Nicholas Freitag McPhee, Riccardo Poli, William B. Langdon, *A Field Guide to Genetic Programming, 2008.*
[10] Ian Parberry, A Real-Time Algorithm for the $(n^2 - 1)$-Puzzle, Information Processing Letters, 56(1):23–28, 1995
[11] http://kevingong.com/Math/SixteenPuzzle.html
[12] Jerry Slocum and Dic Sonneveld, The 15 Puzzle: How it Drove the World Crazy, The Slocum Puzzle Foundation, 2006.
[13] Culberson, J., and Schaeffer, J. 1998. Pattern databases. Computational Intelligence 14(3):318–334.
[14] Peter Russell, Stuart J.; Norvig. Artificial Intelligence: A Modern Approach (3nd ed.). Upper Saddle River, New Jersey: Prentice Hall, 2009.