BİL 441 – Yapay Üs

Doç. Dr.
Osman Abul

TOBB
EKONOMİ VE TEKNOLOJİ
ÜNİVERSİTESİ

Group: 5

Project Final Report

Orhan Onar 131101063
Cihan Alma 151101063

4.12.2019

Introduction

In real life, crosswords [1] have been a highly popular puzzle across the world for many decades, particularly for entertainment. Recently, human-machine competitions on crosswords solving have been held as part of major artificial intelligence (AI) conferences such as ECAI-[4] and IJCAI [3]. Creating a crosswords puzzle includes two main components, which often interleave and impact each other. One component is to generate the grid. Given a grid size, a pattern of blocked cells and one or more lists of words (dictionary), all slots have to be filled with words from the dictionary.In this report, we will be discussing about Crossword puzzles, and our approach on how to tackle automatic crossword generation problem. A crossword is a word puzzle that normally takes the form of a square or rectangular grid of black and white squares. Generation of crossword puzzles [2] by hand is a very hard task, and automating this task through a computer program is also very complex. This complexity is a result of the fact that most crossword puzzle just not include simple vocabulary but also phrases, slang language, abbreviations etc. The goal of this project is to use intelligent techniques to create and solve crossword puzzles.

State Space search

State space search [6] is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or states of an instance are considered, with the intention of finding a goal state with a desired property.

Problems are often modelled as a state space, a set of states that a problem can be in. The set of states forms a graph where two states are connected if there is an operation that can be performed to transform the first state into the second.

State space search often differs from traditional computer science search methods because the state space is implicit: the typical state space graph is much too large to generate and store in memory. Instead, nodes are generated as they are explored, and typically discarded thereafter. A solution to a combinatorial search instance may consist of the goal state itself, or of a path from some initial state to the goal state.

A Word- Based Approach

The word-based approach is based on a simple implementation of the general constraint-satisfaction [7] framework (such as that suggested by Russel and Norvig), namely an assignment of values to a set of variables such that each meets all the specified constraints. For crossword puzzles, the variables are the words which need to be filled in, one variable for each of the words in the puzzle solution. The grid also provides the initial constraints for each word, such as the number of letters, what overlap relationships the word has with other words in the grid, etc. The basic rule that whenever two words overlap, they must share the same letter in the overlap position must also be encoded in the system. Finally, a dictionary provides the overall constraint that any valid value for a variable must be in the dictionary. (Of course, that limits the crossword puzzles which can be solved, since most puzzles use at least one two-word phrase or proper name; from a theoretical perspective, however, one could construct a sufficiently large dictionary to avoid this problem.)

The advantage to this approach is that as a direct implementation of the generic CSP model, there are algorithms which are known to work, and which would be extensible to other problems. However, I was unable to come up with a suitable representation for the constraints listed, and so began considering other models.

A basic crossword-solving algorithm

A large part of the research on this project centered around the creation and refinement of the following algorithm for filling in a crossword grid (under the character-based model described above). It takes as input a grid and a word list; with minor variations, the same algorithm can be used either to solve a puzzle (in which case the grid is input with "stops" already in place, and the word list is (possibly a subset of) the dictionary) or to generate a puzzle solution (in which case the grid is initially empty and the word list contains the words to fill into the grid).

Note that a subgrid, as used in the algorithm, is a section of the grid space whose upper left square is the first letter of both its horizontal and vertical words; the subgrid then extends to all squares which are part of words which intersect either the horizontal or vertical word starting from the initial position.

- Initialization: Set all the squares in the grid to their unconstrained state, that is all character values are possible. (For solving purposes, this is A-Z; for generation, A-Z plus a stop indicator, such as NUL.)
- Repeat the following for each subgrid:
    - For each of the possible character values of the initial (upper left) position, get all words from the word list which begin with the given character, and which satisfy the length constraints for the down and across words, respectively. If there is not at least one word which satisfies the length and initial letter constraints for each of the two words (that is, there must be at least one valid across word and one valid down word), move to the next character value. Otherwise:
        - "Write in" each of the words which were determined to meet the across criteria into the grid, moving righward from the initial cell as you go. For each character written, maintain a reference to what word caused it to be written in.
        - If at any point a letter cannot be written into a cell because it is no longer a possible value for the cell, remove the current word from the grid and proceed to the next word.
        - Repeat the above two steps for the down words, starting from the initial position and moving downward.
    - Move to the cell to the right of the initial position. For each character in the list of possible values, find all words which meet the length and initial letter constraints (i.e. start with the correct letter and are the proper length). Call this list *words.*
        - If *words* is empty, delete the current character from the list of possible values. Propagate the deletion backwards and forwards in the grid by removing the word which caused the deleted character to be written

        in. Repeat as necessary; if a cell ever loses all of its possible values, terminate the algorithm and return FAIL.

        - "Write in" each of the words, as above. If a letter must be written into a cell for which it is not a possible value, remove the current word and propagate the changes.

    - Repeat the above step for each row of the grid, for the length of the initial down word. This should completely traverse the subgrid.

- When all subgrids have been filled in, the grid should contain a representation of all possible solutions. Output either one or all possible solutions using one of any number of algorithms for enumerating the solutions. (The grid consists of a collapsed tree of possible solutions, with each cell representing all the possible values for that cell.)

Our main approach on solving the puzzle was to make use of state space searching. Riddler

State space search starts at this point. Space search methods are a method in which you check each dependency between the bag of candidate solutions, i.e first across answer's first letter with first down answer's first letter, and try to reduce the possible candidates even further. Our first approach on implementing constraint satisfaction was to create every single dependency. Although this approach would be the perfect solution to our problem that would without a doubt generate the correct puzzle and do it in a matter of second, it requires something extremely difficult to achieve which is for every correct answer to be in its

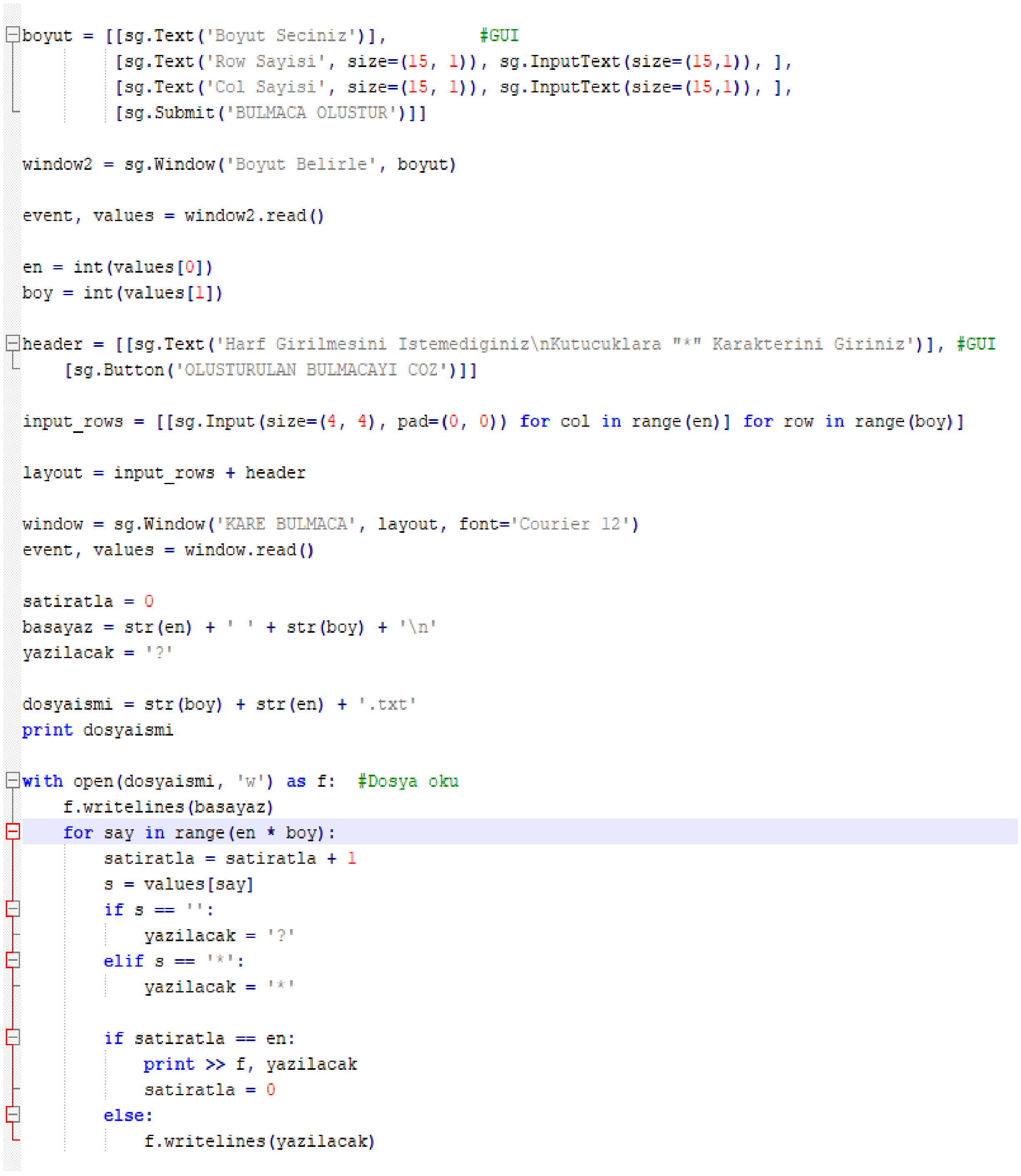
Implementation

Riddler will include of two main parts, one of them is a graphical user interface to visualize Black bars and generating the crossword.First, users can decide crossword size from keyboard after that users can easily select positions and numbers of black bars.
On the other hand backend code reaches to the dictionary JSON [5] file and use it as dictionary.We used python for implementation.All the information about the code is given next to the lines.

<u>Hardware Requirement:</u>
-i3 Processor Based Computer or higher
-Memory: 1 GB RAM
-Hard Drive: 50 GB
-Monitor

<u>Software Requirement:</u>
-Windows 7 or higher
-Python 2.0 or higher and pip library

GUI AND PREPARATION



```
boyut = [[sg.Text('Boyut Seciniz')],         #GUI
         [sg.Text('Row Sayisi', size=(15, 1)), sg.InputText(size=(15,1)), ],
         [sg.Text('Col Sayisi', size=(15, 1)), sg.InputText(size=(15,1)), ],
         [sg.Submit('BULMACA OLUSTUR')]]

window2 = sg.Window('Boyut Belirle', boyut)

event, values = window2.read()

en = int(values[0])
boy = int(values[1])

header = [[sg.Text('Harf Girilmesini Istemediginiz\nKutucuklara "*" Karakterini Giriniz')], #GUI
    [sg.Button('OLUSTURULAN BULMACAYI COZ')]]

input_rows = [[sg.Input(size=(4, 4), pad=(0, 0)) for col in range(en)] for row in range(boy)]

layout = input_rows + header

window = sg.Window('KARE BULMACA', layout, font='Courier 12')
event, values = window.read()

satiratla = 0
basayaz = str(en) + ' ' + str(boy) + '\n'
yazilacak = '?'

dosyaismi = str(boy) + str(en) + '.txt'
print dosyaismi

with open(dosyaismi, 'w') as f:  #Dosya oku
    f.writelines(basayaz)
    for say in range(en * boy):
        satiratla = satiratla + 1
        s = values[say]
        if s == '':
            yazilacak = '?'
        elif s == '*':
            yazilacak = '*'

        if satiratla == en:
            print >> f, yazilacak
            satiratla = 0
        else:
            f.writelines(yazilacak)
```

```
for i in range(len(yatayKelimeler)):
    order1.append(['a', i])
for i in range(len(dikeyKelimeler)):
    order2.append(['d', i])
order = []
while True:
    try:
        order.append(order1.pop(0))
        order.append(order2.pop(0))
    except IndexError:
        break
if len(order1) > 0:
    order.extend(order1)  #uyan yatay kelimeleri ekle
if len(order2) > 0:
    order.extend(order2)  #uyan dikey kelimeleri ekle
for init in range(100):   #100 cozum yeterli
    sols = sols + 1
    print  'Cozum', sols
    temp = time.time()
    bulmacaCoz()
    t += time.time() - temp
    sys.stdout = file
    print 'Cozum', sols
    ciktiVer()
    sys.stdout = sys.__stdout__
    file.flush()
    ciktiVer()
    if t > 15: #15 saniye arama yeterli
        break
print t
sys.stdout = file
print t
sys.stdout = sys.__stdout__

file.close()

filename = 'answers.txt'              #sonuclari GUI ile ekrana bas
root = Tk()
top = Frame(root);
top.pack(side='top')
text = Pmw.ScrolledText(top,
                        borderframe=5,
                        vscrollmode='dynamic',
                        hscrollmode='dynamic',
                        labelpos='n',
                        label_text='file %s' % filename,
                        text_width=60,
                        text_height=en+2,
                        text_wrap='none',
                        )
text.pack()

text.insert('end', open(filename, 'r').read())
Button(top, text='Quit', command=root.destroy).pack(pady=15)
root.mainloop()
```

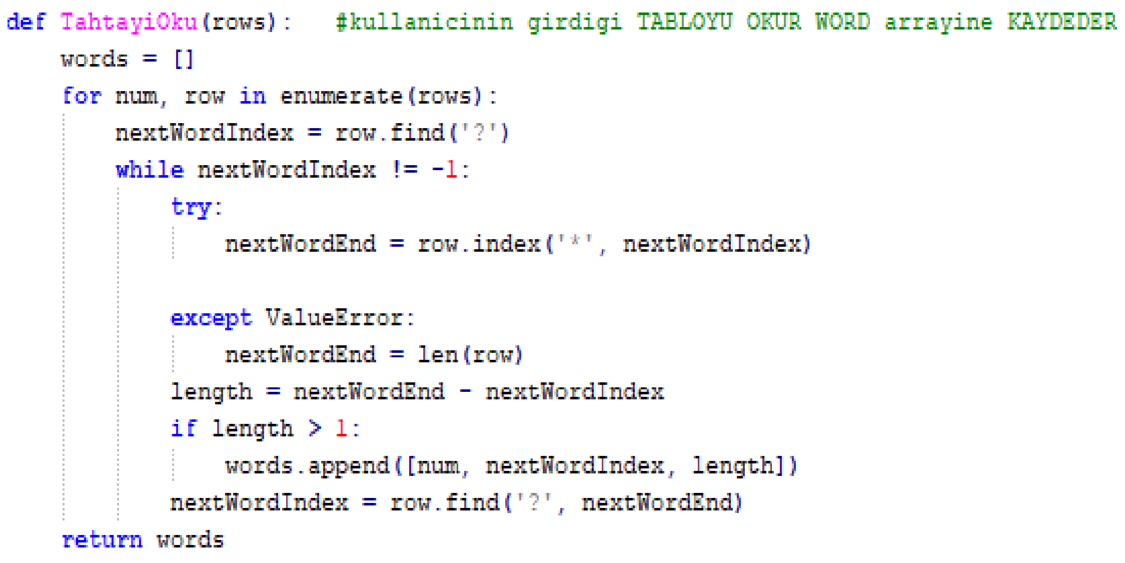
CONTRAINT FINDING

```python
def ConstraintsBul(pointer):   # BULMACANIN siyah NOKTALARINA D verir BEYAZ NOKTALARINA A VERIR
    ori, index = pointer[0], pointer[1]
    constraints = []
    if ori == 'a':
        constraints.append(yatayKelimeler[index][2])
        for lap in kesisenKelimeler:
            if lap[0] == index:
                hpos = lap[1]
                vind = lap[2]
                vpos = lap[3]
                for word in words:
                    if word[0] == 'd' and word[1] == vind:
                        constraints.append([hpos, word[2][vpos]])  #dikeyin yatay ile kesisme durumu
                        break
    elif ori == 'd':
        constraints.append(dikeyKelimeler[index][2])
        for lap in kesisenKelimeler:
            if lap[2] == index:
                hind = lap[0]
                hpos = lap[1]
                vpos = lap[3]
                for word in words:
                    if word[0] == 'a' and word[1] == hind:
                        constraints.append([vpos, word[2][hpos]]) #dikeyin yatayla kesisme durumu
                        break
    return constraints
```

SOLVE METHOD

```python
ef bulmacaCoz(init=0, offset=False):
    myOrder = order[:]
    delOrder = []
    delWords = {}
    while len(myOrder) > 0:
        if offset:
            prev = words.pop()
            kelime = eslestir(ConstraintsBul(myOrder[0]), prev[3] + 1) # prev[3] kelime tutulan yer,offset varsa  offsetten sonraki kelimeye gec ordan basla
            offset = False
        elif not vars().has_key('new'):
            new = True
            kelime = eslestir(ConstraintsBul(myOrder[0]), init)  #init durumu icin constaints check et
        else:
            if delWords.has_key((myOrder[0][0], myOrder[0][1])):  #kelime eslesmeyenler listesindeyse
                kelime = eslestir(ConstraintsBul(myOrder[0]), delWords[(myOrder[0][0], myOrder[0][1])] + 1)  #bir daha dene +1 offsetli
            else:
                kelime = eslestir(ConstraintsBul(myOrder[0])) # tekrar kelime bul
        if kelime == False:  #olmuyorsa pointeri ileri tasi yeni kelime gerekiyor
            pointer = 0
            const = myOrder[pointer]
            while kelime == False:  #uygun kelime bulunana kadar
                try:
                    move = words.pop()
                except IndexError:
                    continue
                delWords[(move[0], move[1])] = move[3]
                for tup in delOrder:
                    if tup[0] == move[0] and tup[1] == move[1]:
                        myOrder.append(tup)
                        break
                    if delWords.has_key((myOrder[0][0], myOrder[0][1])):  #kelime olmuyorsa
                        kelime = eslestir(ConstraintsBul(myOrder[0]), delWords[(myOrder[0][0], myOrder[0][1])] + 1) #tekar eslestir cagir
                    else:
                        kelime = eslestir(ConstraintsBul(myOrder[0]))        # tekrar kelime bul
            words.append([myOrder[pointer][0], myOrder[pointer][1], kelime[0], kelime[1]])  # bulunan kelimeyi ekle
            delOrder.append(myOrder.pop(pointer))  #eklenen kelimeyi delOrder'a ekle
            if disp:
                ciktiVer()
            continue
        words.append([myOrder[0][0], myOrder[0][1], kelime[0], kelime[1]]) # aramaya gerek kalmadan kelime bulundu, ekle
        delOrder.append(myOrder.pop(0))   # eklenen kelimeyi delOrder'a ekle
        if disp:
            ciktiVer()
```

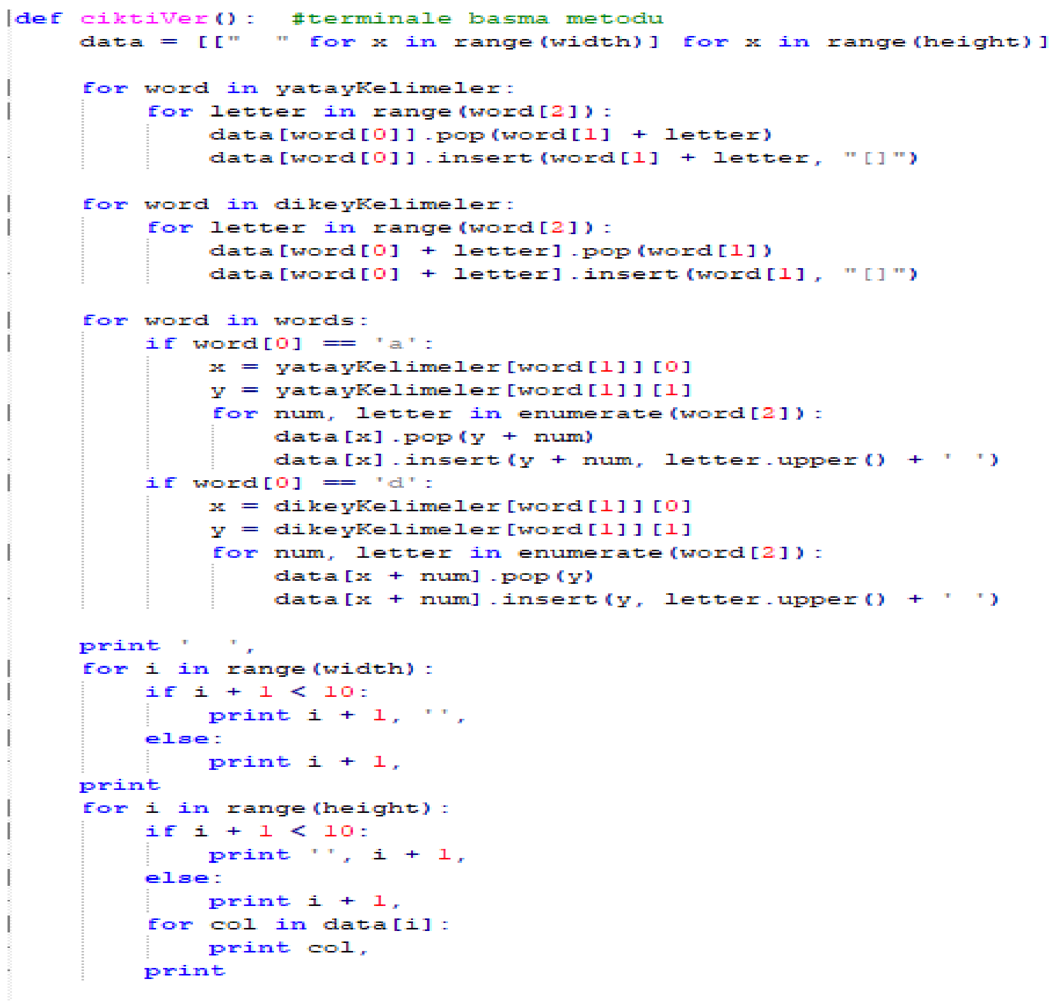
READING THE CROSSWORD BOARD



```
def TahtayiOku(rows):   #kullanicinin girdigi TABLOYU OKUR WORD arrayine KAYDEDER
    words = []
    for num, row in enumerate(rows):
        nextWordIndex = row.find('?')
        while nextWordIndex != -1:
            try:
                nextWordEnd = row.index('*', nextWordIndex)

            except ValueError:
                nextWordEnd = len(row)
            length = nextWordEnd - nextWordIndex
            if length > 1:
                words.append([num, nextWordIndex, length])
            nextWordIndex = row.find('?', nextWordEnd)
    return words
```

PRINT METHOD



```
def ciktiVer():  #terminale basma metodu
    data = [["  " for x in range(width)] for x in range(height)]

    for word in yatayKelimeler:
        for letter in range(word[2]):
            data[word[0]].pop(word[1] + letter)
            data[word[0]].insert(word[1] + letter, "[]")

    for word in dikeyKelimeler:
        for letter in range(word[2]):
            data[word[0] + letter].pop(word[1])
            data[word[0] + letter].insert(word[1], "[]")

    for word in words:
        if word[0] == 'a':
            x = yatayKelimeler[word[1]][0]
            y = yatayKelimeler[word[1]][1]
            for num, letter in enumerate(word[2]):
                data[x].pop(y + num)
                data[x].insert(y + num, letter.upper() + ' ')
        if word[0] == 'd':
            x = dikeyKelimeler[word[1]][0]
            y = dikeyKelimeler[word[1]][1]
            for num, letter in enumerate(word[2]):
                data[x + num].pop(y)
                data[x + num].insert(y, letter.upper() + ' ')

    print '  ',
    for i in range(width):
        if i + 1 < 10:
            print i + 1, '',
        else:
            print i + 1,
    print
    for i in range(height):
        if i + 1 < 10:
            print '', i + 1,
        else:
            print i + 1,
        for col in data[i]:
            print col,
        print
```

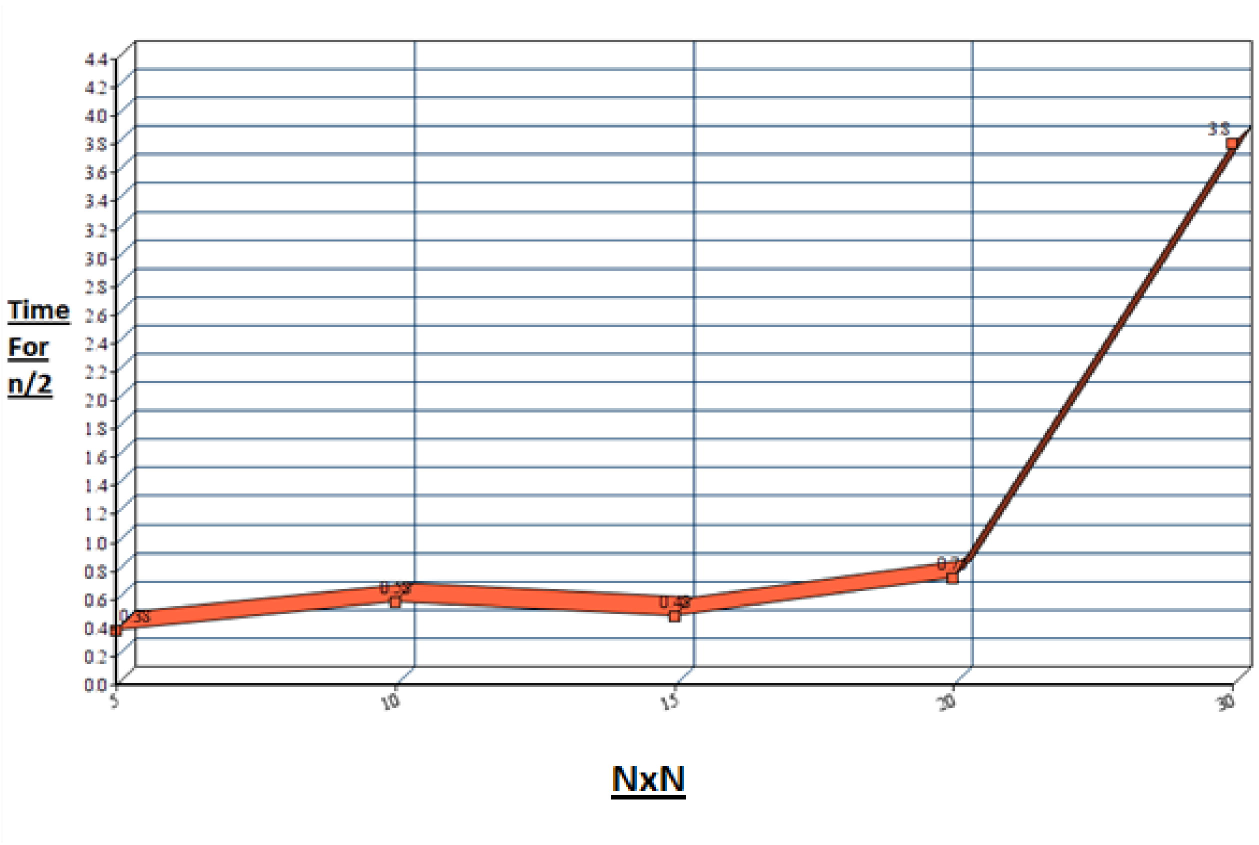
Test Runs

The below images are the outcomes of the puzzles in various times. Since our printing code first fills in the generated solutions for only the across clues in the 5 by 5 to 100 by 100 grid and then fills in the correctly found solutions. The test runs listed below are the randomly collected puzzles.

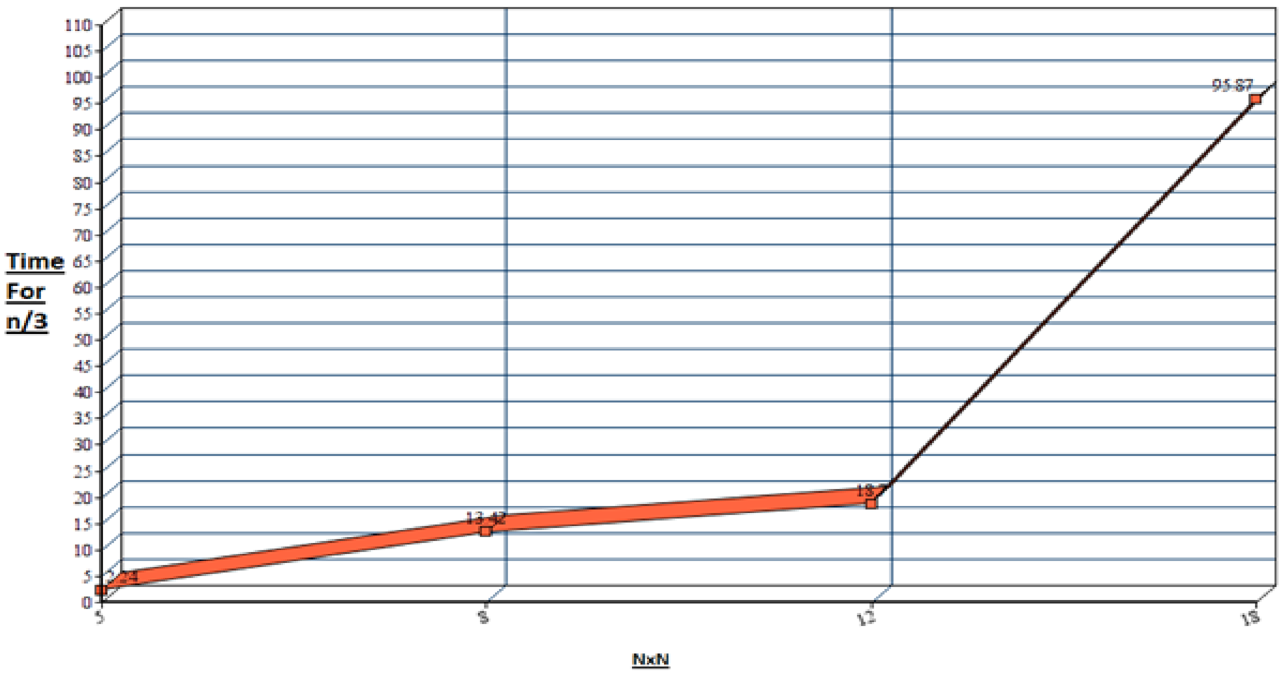
TEST VARIABLES

N= **Grid Size**
B= **Number Of Black Bars Per Line**
B1= **N/2**
B2= **N/3**
B3= **N/6**

B1

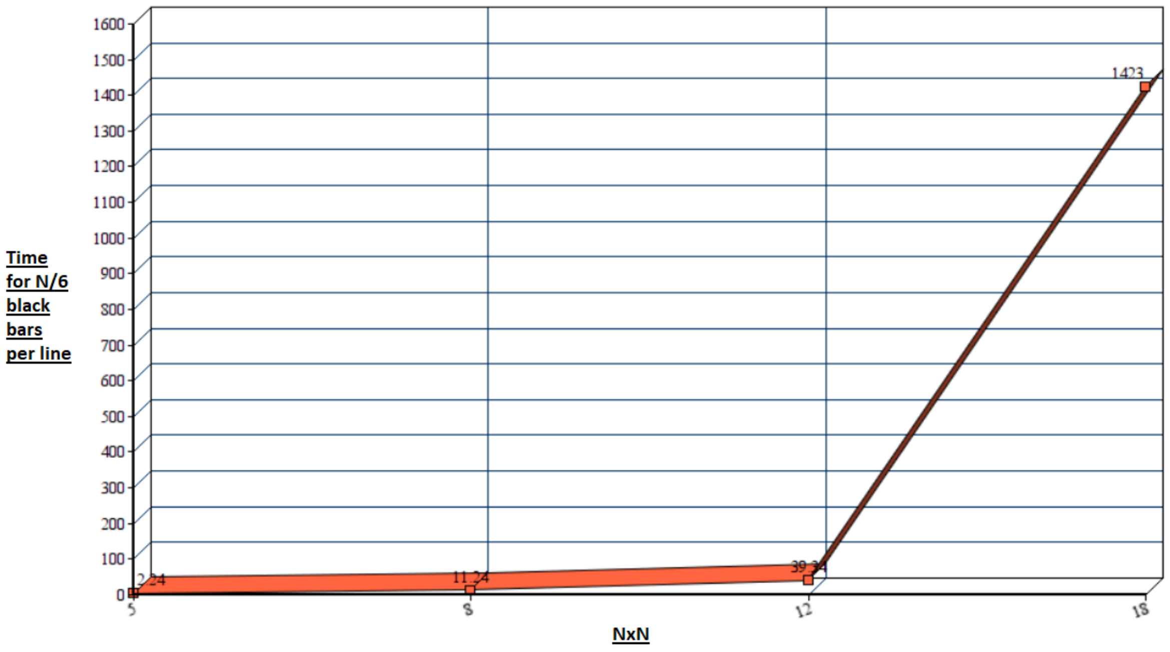


B2



B3

Riddler



For the test which is black bar numbers higher lower than n we   found 68 solutions from 100 tries.

Finding ratio= %68

[1]: L. J. Mazlack. Computer Construction of Crossword Puzzles Using Precedence Relationships Artiial Intelligence 7, pp 1-19, 1976.
[2]: M. L. Ginsberg et al. Search Lessons Learned From Crossword Puzzles Automated Reasoning, pp 210-215, 1990.
[3]:https://www.ijcai.org/
[4]: http://ecai.ro/
[5]: https://www.w3schools.com/js/js_json_intro.asp
[6]: https://sites.fas.harvard.edu/~cscie119/lectures/search.pdf
[7]: http://aima.cs.berkeley.edu/newchap05.pdf