

דוח מעבדה 3:

נא לוודא שתיקית הINPUT וקובץ הEXE
נמצאים באותו תיקייה.
ניתן לעורר זמן ריצה וקובץ קלט על ידי שימוש
בקובץ GLOBAL_ARGS.TXT.
ניתן לעורר ארגומנטים שונים בשני הקבצים
האחרים שנמצאים ביחד בתיקיית INPUT.

1. עלייכם לקרוא את הקלט ולהדפיס את מאפייניו בכלים מספר הצלמתים הקשנות וכן לחשב את צפיפות הגרף

את הצפיפות- מצורפת תמונה של הפונקציה שמחשבת את זה.

$$D = \frac{|E|}{\binom{|V|}{2}} = \frac{2|E|}{|V|(|V| - 1)}$$

לגביה תכונות הגרף: מצורפת דוגמא לאופן ההדפסה של הפרטיהם.

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 23
Number of edges: 71
Graph Density = 0.28063241106719367
```

```
def printGraphStats(v, e):
    density = float((2 * e) / (v * (v - 1))) # which is also E / (V choose 2)
    print('Number of nodes: ', v)
    print('Number of edges: ', e)
    print('Graph Density = ', density)
```

ב. בחלק הראשון של העבודה תשוו בין ביצועי אלגוריתם חיפוש לאחרו לבין אלגוריתם חיפוש קדימה

- a. אלגוריתם חיפוש לאחרו עם BACKTRACKING עם BACKJUMPING
 - b. אלגוריתם חיפוש קדימה FORWARD CHECKING עם ARC CONSISTENCY
- במיום של אלגוריתמים אלה עלייכם להשתמש בהיריסטיקות HD ו-MRV LCV

FORWARD CHECKING:

מימוש וריעון האלגוריתם: כמו שראינו בהרצאת המבוא של הקורס, אנחנו מתחילהים לצבוע את הצלמתים לפי היוריסטיות תוך כדי שמירה

על הסתכלות קדימה כדי לזהות בעיות בכמה שיוור מוקדם כדי לחסוך בכניסה לעומק העץ. את זה עשינו לפי הסתכלות על AC של הגרפ', וcmbwn להסתכל את עברו צביעה כלשהי לצומת קיבלנו שאחד השכנים אין לא עוד צבעים אז זאת צביעה לא טובה ונחזור. וגם כן מתיחסים למקומות של "סינגלטון".

בו אנחנו בודקים ARC, אם כן איז נמשיך.

```
csp = CSP(nodes, constraintsWrapper, domains, [i + 1 for i in range(domainNumber)])
if not csp.arcConsistency():
    print("No solution found!")
else:
    print('AC-3 success')

solution = csp.forwardChecking()

if solution is None:
    print("No solution found!")
else:
    if checkSol(solution):
        printSolution(solution)
    else:
        print('Solution Not Consistent !!!!!!!!!!!!!')
print()
```

פונקציית הבדיקה והיפוך הגרף ל ARC

```
def arcConsistency(self):
    while len(self.constraintsWrapper.allConstraints) != 0:
        constraint = self.constraintsWrapper.allConstraints.pop(0)
        self.constraintsWrapper.constraintsDict[constraint] = False
        firstNode = constraint.node1
        secondNode = constraint.node2

        firstDomains = self.domains[firstNode]
        secondDomains = self.domains[secondNode]

        addOppositeArcs = False
        removeValues = []
        noSatisfactionFound = True
        for val1 in firstDomains:
            found = False
            for val2 in secondDomains:
                if val1 != val2:
                    found = True
                    noSatisfactionFound = False
            if not found:
                removeValues.append(val1)
        if noSatisfactionFound:
            return False
        for val in removeValues:
            self.domains[firstNode].remove(val)
        addOppositeArcs = True
        if addOppositeArcs:
            self.constraintsWrapper.addArcs(firstNode)
    return True
```

המימוש העיקרי של האלגוריתם:

```
def forwardChecking(self, assignment=None, domains=None):
    # assignment is complete if every variable is assigned (our base case)
    if assignment is None:
        assignment = {}
    if domains is None:
        domains = self.domains
    if len(assignment) == len(self.nodes):
        return assignment

    # get all Variables in the CSP but not in the assignment
    unassigned = [node for node in self.nodes if node not in assignment]

    # get the every possible domain value of the first unassigned variable
    node = self.getNode(unassigned, domains)
    # domains = self.LCV(node, assignment, domains)
    for value in domains[node]:
        toBreak = False
        localAssignment = {key: value for key, value in assignment.items()} # deep copy
        localAssignment[node] = value
        localDomains = {key: value[:] for key, value in domains.items()} # deep copy
        # if we're still consistent, we recurse (continue)
        if self.isConsistent(node, localAssignment):
            if len(localAssignment) == len(self.nodes):
                return localAssignment
            neighbors = node.neighbors
            for neighbor in neighbors:
                if neighbor not in assignment:
                    if value in localDomains[neighbor]:
                        localDomains[neighbor].remove(value)
                        if len(localDomains[neighbor]) == 0:
                            toBreak = True
                            break
            if toBreak:
                continue
            result = self.forwardChecking(localAssignment, localDomains)
            # if we didn't find the result, we will end up backtracking
            if result is not None:
                return result
    return None
```

התיחסות ל SINGLETON בצורה ה затה:

```
def MRV(self, unassigned, domains): # choose node with minimum domains left
    minLen = float('inf')
    retNode = None
    for node in unassigned:
        length = len(domains[node])
        # check singleton
        if length == 1:
            return node
        if length < minLen:
            minLen = length
            retNode = node
        elif length == minLen: # solves equality
            if len(self.constraintsWrapper.constraintsByNode[node]) > \
                len(self.constraintsWrapper.constraintsByNode[retNode]):
                minLen = length
                retNode = node
    return retNode
```

BACKTRACKING + BACKJUMPING:

שימוש האלגוריתם: כמו שראינו בקורס המבוא, אנחנו מנסים למצוא צביעה חוקית ב-K צבעים, אנחנו עושים זאת זה תוך כדי שימוש בהיוריסטיות שמיישנו (מצורפות למטה).

לכל NODE אנחנו נשמר זמן בקרה כדי שנצליח למיין את ה-conflictSet לפי הזמן הזה. כשנכנסים לצומת אנחנו מעדכנים את ה-CONFLICTSET שלה לפि השכינים, ואם לצומת אין צבעים לבחירה אנחנו חוזרים לצומת הראשונה - conflict של הצומת הנוכחי וمعدכנים בהתאם.

```
def backtracking(self, time=0, assignment=None, domains=None):
    # assignment is complete if every variable is assigned (our base case)
    if assignment is None:
        assignment = {}
    if domains is None:
        domains = self.domains
    if len(assignment) == len(self.nodes):
        return assignment

    # get all variables in the CSP but not in the assignment
    unassigned = [node for node in self.nodes if node not in assignment]

    # get the every possible domain value of the first unassigned variable
    node = self.getNode(unassigned, domains)
    node.time = time
    self.conflictNode = None
    self.LCV(node, assignment)
    for value in domains[node]:
        localAssignment = {key: value for key, value in assignment.items()}  # deep copy
        localAssignment[node] = value
        localDomains = {key: value[:] for key, value in domains.items()}  # deep copy
        # if we're still consistent, we recurse (continue)
        if self.isConsistent(node, localAssignment):
            localDomains[node].remove(value)
            for neighbor in node.neighbors:
                if value in localDomains[neighbor]:
                    localDomains[neighbor].remove(value)

            self.updateConflictSet(node)
            result = self.backtracking(time + 1, localAssignment, localDomains)
            # if we didn't find the result, we will end up backtracking
            if result is not None:
                return result

        self.updateAllConflictSets(node)
        if self.conflictNode is not None:
            result = self.backtracking(time + 1, assignment, domains)
            # if we didn't find the result, we will end up backtracking
            if result is not None:
                return result
    return None
```

LCV-1:

```
def LCV(self, node, assignment, domains):
    neighbors = node.neighbors
    futureOptions = {}
    for value in domains[node]:
        counter = 0
        assignment[node] = value
        # if it is not consistent
        if not self.isConsistent(node, assignment):
            futureOptions[value] = 0
            continue
        for neighbor in neighbors:
            if neighbor not in assignment: # if isn't colored
                for tempVal in domains[neighbor]:
                    if tempVal != value:
                        counter += 1
        if counter == 0:
            break
        futureOptions[value] = counter
    assignment.pop(node, None)
    sorted(futureOptions.items(), key=lambda kv: (kv[1], kv[0]))
    d = [k for k, v in sorted(futureOptions.items(), key=lambda item: item[1])]
    d.reverse()
    domains[node] = d
    return domains
```

LCV 2:

```
def LCV_1(self, node):
    bestColor = -1
    color = 0
    for value in node.domains:
        classColor = self.numberOfDomains[value - 1]
        if classColor > bestColor:
            bestColor = classColor
            color = value
    return color
```

עבור היוריסטייקת LCV מימשנו שתי יוריסטייקות שכל אחת מהן מגדירה את "יותר גמישות" הוצאה אחרת.

ה-1-LCV: מגדירה את הgmישות כך: שהצבע שאנו נבחר הוא הצבע שמשאיר לשכנים מספר הci גדול של צבעים.

ה-2-LCV: מגדירה את gmישות כך: נבחר את הצבע שצבענו בו הci הרבה צמתים.

MRV + HD:

שתי היוריסטייקות עובדות ביחד, מנסים לנבחר צומת שנשאר לו הci פחות צבעים, ואם יש כמה כאלה, נבחר את זה על דרגה הci גדולה.

```
def MRV(self, unassigned, domains):    # choose node with minimum domains left
    minLen = float('inf')
    retNode = None
    for node in unassigned:
        length = len(domains[node])
        if length < minLen:
            minLen = length
            retNode = node
        elif length == minLen:    # solves equality
            if len(self.constraintsWrapper.constraintsByNode[node]) > \
                len(self.constraintsWrapper.constraintsByNode[retNode]):
                minLen = length
                retNode = node
    return retNode
```

השוואה בין שני האלגוריתמים:

backtracking:

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 450
Number of edges: 8169
Graph Density = 0.08086117297698589

Trying to color in 18 colors ... Success !!
Solution: {32 : 18, 115 : 17, 399 : 16, 270 : 15, 5 : 14, 403 : 13, 269 : 12, 244 : 11, 441 : 15, 267 : 10, 39

Trying to color in 17 colors ... Success !!
Solution: {32 : 17, 115 : 16, 399 : 15, 270 : 14, 5 : 13, 403 : 12, 269 : 11, 244 : 10, 441 : 14, 267 : 9, 39

Trying to color in 16 colors ...
```

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 450
Number of edges: 9803
Graph Density = 0.097035538728037614

Trying to color in 12 colors ... Success !!
Solution: {388 : 12, 319 : 11, 86 : 10, 49 : 11, 285 : 9, 17 : 12, 53 : 8, 168 : 8, 439

Trying to color in 11 colors ... Success !!
Solution: {388 : 11, 319 : 10, 86 : 9, 49 : 10, 285 : 8, 17 : 11, 53 : 7, 168 : 7, 439 :

Trying to color in 10 colors ... Success !!
Solution: {388 : 10, 319 : 9, 86 : 8, 49 : 9, 285 : 7, 17 : 10, 53 : 6, 168 : 6, 439 : 9

Trying to color in 9 colors ...
```

```
main ✘
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 23
Number of edges: 71
Graph Density = 0.28063241106719367

Trying to color in 5 colors ... Success !!
Solution: {23 : 5, 22 : 4, 6 : 5, 11 : 4, 17 : 3, 2 : 4, 4 : 4, 12 : 3, 7 : 5, 1 : 3, 18

Trying to color in 4 colors ... No solution found with 4 colors.

Minimum number of colors needed is 5 colors.

time = 0.9373292922973633

Process finished with exit code 0
```

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 49
Number of edges: 952
Graph Density = 0.8095238095238095

Trying to color in 12 colors ... Success !!
Solution: {25 : 12, 17 : 11, 18 : 10, 19 : 9, 11 : 8, 24 : 9, 32 : 11, 26 : 8, 33 : 7, 10 : 6, 16 : 5, 21 : 4, 27 : 3, 31 : 2, 3 : 1}

Trying to color in 11 colors ... Success !!
Solution: {25 : 11, 17 : 10, 18 : 9, 19 : 8, 11 : 7, 24 : 8, 32 : 10, 26 : 7, 33 : 6, 10 : 5, 16 : 4, 21 : 3, 27 : 2, 31 : 1, 3 : 1}

Trying to color in 10 colors ... Success !!
Solution: {25 : 10, 17 : 9, 18 : 8, 19 : 7, 11 : 6, 24 : 7, 32 : 9, 26 : 6, 33 : 5, 10 : 4, 16 : 3, 21 : 2, 27 : 1, 31 : 1, 3 : 1}

Trying to color in 9 colors ... Success !!
Solution: {25 : 9, 17 : 8, 18 : 7, 19 : 6, 11 : 5, 24 : 6, 32 : 8, 26 : 5, 33 : 4, 10 : 3, 16 : 2, 21 : 1, 27 : 1, 31 : 1, 3 : 1}

Trying to color in 8 colors ... Success !!
Solution: {25 : 8, 17 : 7, 18 : 6, 19 : 5, 11 : 4, 24 : 5, 32 : 7, 26 : 4, 33 : 3, 10 : 2, 16 : 1, 21 : 1, 27 : 1, 31 : 1, 3 : 1}

Trying to color in 7 colors ... Success !!
Solution: {25 : 7, 17 : 6, 18 : 5, 19 : 4, 11 : 3, 24 : 4, 32 : 2, 26 : 1, 33 : 1, 10 : 1, 16 : 1, 21 : 1, 27 : 1, 31 : 1, 3 : 1}

Trying to color in 6 colors ...

```

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 138
Number of edges: 986
Graph Density = 0.10430551147783772

Trying to color in 11 colors ... Success !!
Solution: {18 : 11, 36 : 10, 95 : 9, 74 : 8, 72 : 7, 138 : 6, 116 : 5, 120 : 4, 102 : 3, 128 : 2, 144 : 1, 152 : 1, 156 : 1, 160 : 1, 164 : 1, 168 : 1, 172 : 1, 176 : 1, 180 : 1, 184 : 1, 188 : 1, 192 : 1, 196 : 1, 200 : 1, 204 : 1, 208 : 1, 212 : 1, 216 : 1, 220 : 1, 224 : 1, 228 : 1, 232 : 1, 236 : 1, 240 : 1, 244 : 1, 248 : 1, 252 : 1, 256 : 1, 260 : 1, 264 : 1, 268 : 1, 272 : 1, 276 : 1, 280 : 1, 284 : 1, 288 : 1, 292 : 1, 296 : 1, 300 : 1, 304 : 1, 308 : 1, 312 : 1, 316 : 1, 320 : 1, 324 : 1, 328 : 1, 332 : 1, 336 : 1, 340 : 1, 344 : 1, 348 : 1, 352 : 1, 356 : 1, 360 : 1, 364 : 1, 368 : 1, 372 : 1, 376 : 1, 380 : 1, 384 : 1, 388 : 1, 392 : 1, 396 : 1, 400 : 1, 404 : 1, 408 : 1, 412 : 1, 416 : 1, 420 : 1, 424 : 1, 428 : 1, 432 : 1, 436 : 1, 440 : 1, 444 : 1, 448 : 1, 452 : 1, 456 : 1, 460 : 1, 464 : 1, 468 : 1, 472 : 1, 476 : 1, 480 : 1, 484 : 1, 488 : 1, 492 : 1, 496 : 1, 500 : 1, 504 : 1, 508 : 1, 512 : 1, 516 : 1, 520 : 1, 524 : 1, 528 : 1, 532 : 1, 536 : 1, 540 : 1, 544 : 1, 548 : 1, 552 : 1, 556 : 1, 560 : 1, 564 : 1, 568 : 1, 572 : 1, 576 : 1, 580 : 1, 584 : 1, 588 : 1, 592 : 1, 596 : 1, 600 : 1, 604 : 1, 608 : 1, 612 : 1, 616 : 1, 620 : 1, 624 : 1, 628 : 1, 632 : 1, 636 : 1, 640 : 1, 644 : 1, 648 : 1, 652 : 1, 656 : 1, 660 : 1, 664 : 1, 668 : 1, 672 : 1, 676 : 1, 680 : 1, 684 : 1, 688 : 1, 692 : 1, 696 : 1, 700 : 1, 704 : 1, 708 : 1, 712 : 1, 716 : 1, 720 : 1, 724 : 1, 728 : 1, 732 : 1, 736 : 1, 740 : 1, 744 : 1, 748 : 1, 752 : 1, 756 : 1, 760 : 1, 764 : 1, 768 : 1, 772 : 1, 776 : 1, 780 : 1, 784 : 1, 788 : 1, 792 : 1, 796 : 1, 800 : 1, 804 : 1, 808 : 1, 812 : 1, 816 : 1, 820 : 1, 824 : 1, 828 : 1, 832 : 1, 836 : 1, 840 : 1, 844 : 1, 848 : 1, 852 : 1, 856 : 1, 860 : 1, 864 : 1, 868 : 1, 872 : 1, 876 : 1, 880 : 1, 884 : 1, 888 : 1, 892 : 1, 896 : 1, 900 : 1, 904 : 1, 908 : 1, 912 : 1, 916 : 1, 920 : 1, 924 : 1, 928 : 1, 932 : 1, 936 : 1, 940 : 1, 944 : 1, 948 : 1, 952 : 1, 956 : 1, 960 : 1, 964 : 1, 968 : 1, 972 : 1, 976 : 1, 980 : 1, 984 : 1, 988 : 1, 992 : 1, 996 : 1, 1000 : 1}
```

ForwardChecking -ARC:

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 450
Number of edges: 8169
Graph Density = 0.08086117297698589

Trying to color in 18 colors ... Success !!
Solution: {32 : 18, 115 : 17, 399 : 16, 270 : 15, 5 : 14, 403 : 13, 269

Trying to color in 17 colors ... Success !!
Solution: {32 : 17, 115 : 16, 399 : 15, 270 : 14, 5 : 13, 403 : 12, 269

Trying to color in 16 colors ...
```

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 450
Number of edges: 9803
Graph Density = 0.097035538728037614

Trying to color in 12 colors ... Success !!
Solution: {388 : 12, 319 : 11, 86 : 10, 49 : 11, 285 : 9, 17 : 12, 53 : 8,

Trying to color in 11 colors ... Success !!
Solution: {388 : 11, 319 : 10, 86 : 9, 49 : 10, 285 : 8, 17 : 11, 53 : 7,

Trying to color in 10 colors ... Success !!
Solution: {388 : 10, 319 : 9, 86 : 8, 49 : 9, 285 : 7, 17 : 10, 53 : 6, 16

Trying to color in 9 colors ...
```

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes:  23
Number of edges:  71
Graph Density =  0.28063241106719367

Trying to color in  5  colors ... Success !!
Solution: {23 : 5, 22 : 4, 6 : 5, 11 : 4, 17 : 3, 2 : 4, 4 : 4, 12 : 3, 7

Trying to color in  4  colors ... No solution found with  4  colors.

Minimum number of colors needed is  5  colors.

time =  0.6003096103668213

Process finished with exit code 0
```

```
Solution: {25 : 12, 17 : 11, 18 : 10, 19 : 9, 11 : 8, 24 : 9, 32 : 11, 26 : 8, 33 : 10, 31 : 8, 9 : 9, 10 : 12, 45

Trying to color in  11  colors ... Success !!
Solution: {25 : 11, 17 : 10, 18 : 9, 19 : 8, 11 : 7, 24 : 8, 32 : 10, 26 : 7, 33 : 9, 31 : 7, 9 : 8, 10 : 11, 45 :

Trying to color in  10  colors ... Success !!
Solution: {25 : 10, 17 : 9, 18 : 8, 19 : 7, 11 : 6, 24 : 7, 32 : 9, 26 : 6, 33 : 8, 31 : 6, 9 : 7, 10 : 10, 45 : 5,

Trying to color in  9  colors ... Success !!
Solution: {25 : 9, 17 : 8, 18 : 7, 19 : 6, 11 : 5, 24 : 6, 32 : 8, 26 : 5, 33 : 7, 31 : 5, 9 : 6, 10 : 9, 45 : 4, 2

Trying to color in  8  colors ... Success !!
Solution: {25 : 8, 17 : 7, 18 : 6, 19 : 5, 11 : 4, 24 : 5, 32 : 7, 26 : 4, 33 : 6, 31 : 3, 10 : 8, 23 : 6, 34 : 5,

Trying to color in  7  colors ... Success !!
Solution: {25 : 7, 17 : 6, 18 : 5, 19 : 4, 11 : 3, 24 : 4, 32 : 2, 26 : 1, 33 : 3, 31 : 1, 10 : 2, 12 : 6, 13 : 5,

Trying to color in  6  colors ... No solution found with  6  colors.

Minimum number of colors needed is  7  colors.

time =  22.17302417755127

Process finished with exit code 0
```

```
C:\python381\python.exe S:/onedrive/sync/AILab/lab3/main.py
Number of nodes: 138
Number of edges: 986
Graph Density = 0.10430551147783772

Trying to color in 11 colors ... Success !!
Solution: {18 : 11, 36 : 10, 95 : 9, 74 : 8, 72 : 7, 138 : 6, 116 : 5,
Trying to color in 10 colors ...
```

לסייעם הרצות עם שמות הקבצים:

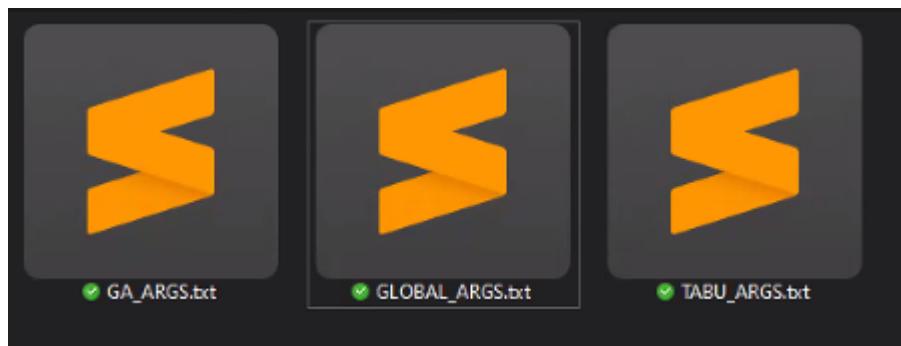
	<u>le450_1 5b.col</u>	<u>le450_5c .col</u>	<u>myciel4. col</u>	<u>queen7 7.col</u>	<u>anna.col</u>
<u>backtracking:</u>	17	10	5	7	11
<u>forward-c</u>	17	10	5	7	11

חשיבות לציון:

בכל שימוש של כל אלגוריתם אנחנו מתחילה מצביעה GREEDY ובה אנו מוצאים את המספר המקסימלי של הצבעים, ומשם אנחנו מתחילה בכל אלגוריתם ומנסים להקטין את המספר זהה.

החלק השני:

הנחיות ריצת האלגוריתמים: כל קובץ מכיל את המידע המתאים.



לפי התמונה מטה עברו TABU: עבור שלושת הגישות צריך להיות "1" מול הגישה הרצiosa ו "0" מול האחרות.

אותו הדבר עבר בבחירה היוריסטיקה: "1" מול היוריסטיקה הרצiosa ו "0" ושניתה.

```
1 MAX_ITER = 3000
2 LOCAL_OPTIMUM_STOP = 25
3 MAX_TABU = 20
4 NEIGHBORS_NUM = 2048
5 fitness_type_feasible = 1
6 fitness_type_objective = 0
7 fitness_type_hybrid = 0
8 neighborhoodType_regular = 1
9 neighborhoodType_kempe = 0
```

זה עבר בבחירה הקובץ המתאים והגבול זמן: (מתיחס להם כולם)

```
1 max_time(sec) = 60
2 input_file_path = instances\le450_15b.col
```

עבור הנתונים של GA:

משתמש יכול לבחור ולשנות רק את המספרים.

```
1 population_size = 200
2 max_iterations = 16384
3 elite_rate = 0.1
4 mutation_rate = 0.5
5 K_variable_(tournament_selection) = 10
6 min_age = 2
7 max_age = 30
8 linear_scaling_A = 1
9 linear_scaling_B = 10
10 local_optimum_iterations_limit = 30
11 selection_Type_RWS = 1
12 selection_Type_SUS = 0
13 selection_Type_TOURNAMENT = 0
14 mutation_Type_EXCHANGE = 1
15 mutation_Type_SIMPLE_INVERSION = 0
```

הערה לגבי מימושים: מימושנו של אלגוריתם TABU ודרך התייחסנו לסייע ג', כך שהוא TABU מתייחס לשלווה הגישות הללו, וההבדל הוא בבחירה השכינים ואיך אנחנו נחשב את הNESS.

- ג. בחלק השני של העבודה עלייכם להשוות בין שלושת גישות של **החיפוש הлокאלי** קרי:
- הגישה שasma את **הפייזibilities** במרכזזה
 - הגישה שasma את **פונקציית המטריה** במרכזזה (שרשראות KEMPE)
 - הגישה **היברידית**

פייזibilities:

שכנים: מכיוון שאנחנו מחפשים צבעה עם 1-K צבעים, אנחנו משתמשים על הצמתים שבكونפליקט ובוחרים אחד רנדומלי, ומנסים את הצבע שלו לצבע שמקטין את מספר ההתנגשויות עם השכנים. (מתחלים מצבעה עם K צבעים, ומוחקם את הצבע שנמצא הכי פחות)

```
def regularNeighborhood(origAssignment, numNeighbors, conflictNodes, domains):
    neighbors = []
    for _ in range(numNeighbors):
        assignment = {key: value for key, value in origAssignment.items()} # deep copy
        minConflictsColor = [0 for _ in range(len(domains))]
        node = conflictNodes[random.randint(0, len(conflictNodes) - 1)]
        for neighbor in node.neighbors:
            minConflictsColor[origAssignment[neighbor] - 1] += 1
        minimum = float('inf')
        minimumColor = 0
        for i in range(len(minConflictsColor)):
            if minConflictsColor[i] < minimum:
                minimum = minConflictsColor[i]
                minimumColor = i + 1
        if random.random() < 0.5:
            assignment[node] = minimumColor
        else:
            assignment[node] = domains[random.randint(0, len(domains) - 1)]
        neighbors.append(assignment)
    return neighbors
```

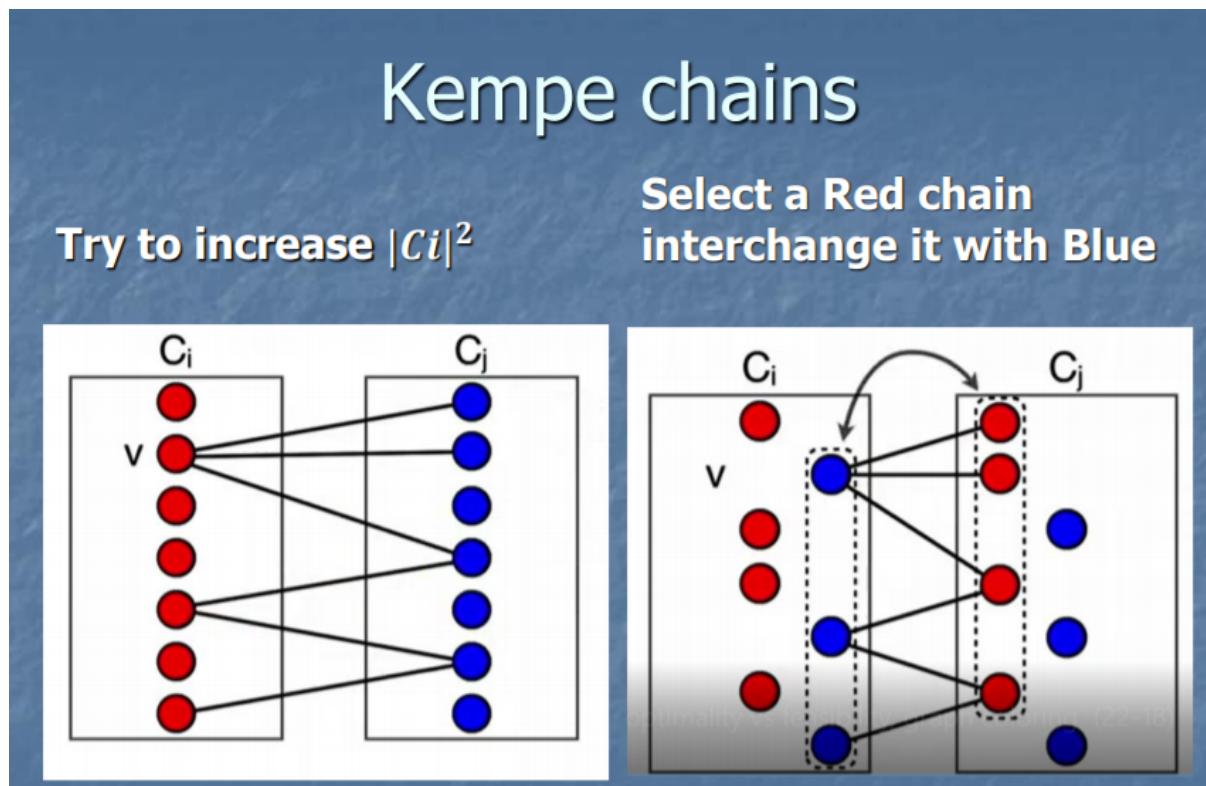
FITNESS:

מחשבים את זה כמספר ההתנגשויות של כל הצמתים, ומובן שהזה יהיה אפס אז זו היא צביעה חוקית.

```
def numberOfConflicts(assignment): # fitness function
    conflictsNum = 0
    conflictNodes = []
    for node in assignment:
        currentNodeColor = assignment[node]
        for neighbor in node.neighbors:
            if assignment[neighbor] == currentNodeColor and neighbor not in conflictNodes:
                conflictsNum += 1
                conflictNodes.append(neighbor)
    return conflictNodes, conflictsNum
```

פונקציית המטריה:

שכנים: אנחנו רוצים להקטיין את מספר הצבעים כדי להגיע ל COLOR CLASS בגודל אפס, ואני עושים את זה לפני ההערכתה.
נבחר צומת עם צבע שניצב פחות בgraf ופועלים בהתאם לפני ההערכתה:



```
def kempeNeighborhood(origAssignment, numNeighbors, domains):
    colorClasses = {}
    for value in domains:
        colorClasses[value] = []
    for node in origAssignment:
        colorClasses[origAssignment[node]].append(node)
    minValue = None
    minNumber = float('inf')
    for value in domains:
        if len(colorClasses[value]) < minNumber:
            minNumber = len(colorClasses[value])
            minValue = value
    nodesByColor = colorClasses[minValue]

    domains.remove(minValue)

    neighbors = []
    for _ in range(numNeighbors):
        assignment = {key: value for key, value in origAssignment.items()} # deep copy
        node = nodesByColor[random.randint(0, len(nodesByColor) - 1)]
        oldColor = assignment[node]
        assignment[node] = domains[random.randint(0, len(domains) - 1)]
        newColor = assignment[node]
        kempeChain(assignment, node, oldColor, newColor)
        neighbors.append(assignment)
    return neighbors
```

FITNESS:

מחשבים לפיה הרצאה, סכום הריבועים מקיים.

```
def objectiveFitness(assignment, domains):
    colorClasses = {}
    for value in domains:
        colorClasses[value] = 0
    for node in assignment:
        colorClasses[assignment[node]] += 1
    fitness = 0
    flag = False
    for value in colorClasses:
        if colorClasses[value] == 0:
            flag = True
            fitness += (colorClasses[value] ** 2)
    return flag, fitness
```

ההיברידיות:

שכנים: אחת משתי השיטות למעלה.

FITNESS:

Combined objective function

- Minimize the bad edges while maximizing the color classes:
- Minimize $\sum_{i=1}^n 2|Bi||Ci| - \sum_{i=1}^n |Ci|^2$
- This objective function guarantees that the local minima will be a legal coloring ("feasible") by design
- **Left term** – feasibility constraint
- **Right term** - objective

```
def hybridFitness(assignment, domains):  
    colorClasses = {}  
    edgeClasses = {}  
    for value in domains:  
        colorClasses[value] = 0  
        edgeClasses[value] = 0  
    for node in assignment:  
        colorClasses[assignment[node]] += 1  
    for node in assignment:  
        currentNodeColor = assignment[node]  
        for neighbor in node.neighbors:  
            if assignment[neighbor] == currentNodeColor:  
                edgeClasses[currentNodeColor] += 1  
    fitness = 0  
    for value in domains:  
        bi = edgeClasses[value]  
        ci = colorClasses[value]  
        fitness += (2 * bi * ci) - (ci ** 2)  
    return None, fitness
```

ד. בIMPLEMENTATION יש לבחור אלגוריתם חיפוש לوكאלי (אפשר יותר) וכן אלגוריתם גנדי (עם אופרטורים המותאימים לבעה)

LOCAL ALGORITHM IS TABU-SEARCH:

כמו שאנו מכירים את האלגוריתם מהשיעור הקודם, התאמנו אותו לבעה הנוכחית והשינו עבור כל גישה הוא רק באופן בחירת השכנים ובחישוב **FITNESS**.
פואודו קוד לאלגוריתם:

```
1 sBest ← s0
2 bestCandidate ← s0
3 tabuList ← []
4 tabuList.push(s0)
5 while (not stoppingCondition())
6     sNeighborhood ← getNeighbors(bestCandidate)
7     bestCandidate ← sNeighborhood[0]
8     for (sCandidate in sNeighborhood)
9         if ( (not tabuList.contains(sCandidate)) and (fitness(sCandidate) > fitness(bestCandidate)) )
10            bestCandidate ← sCandidate
11        end
12    end
13    if (fitness(bestCandidate) > fitness(sBest))
14      sBest ← bestCandidate
15    end
16    tabuList.push(bestCandidate)
17    if (tabuList.size > maxTabuSize)
18      tabuList.removeFirst()
19    end
20 end
21 return sBest
```

```
def tabuSearch(constraints, domains, args, assignment, fitnessType, neighborhoodType):
    startTime = time.time()
    best = assignment
    conflictNodes, bestFitness = getFitness(best, domains, fitnessType)
    bestCandidate = best
    globalBest = best
    globalFitness = bestFitness
    tabuDict = {str(best): True}
    tabu = [best]
    local_counter = 0
    for _ in range(args.maxIter):
        if bestFitness == 0:
            print('Time elapsed: ', time.time() - startTime)
            return globalBest, globalFitness
        iterTime = time.time()
        neighborhood = getNeighborhood(bestCandidate, args.numNeighbors, conflictNodes,
                                         domains[:, neighborhoodType]) # get neighborhood of current solution
        _, minimum = getFitness(best, domains, fitnessType)
        bestCandidate = neighborhood[0]
        for neighbor in neighborhood: # get the best neighbor and save it
            tempConflictNodes, cost = getFitness(neighbor, domains, fitnessType)
            if cost < minimum and not tabuDict.get(str(neighbor), False):
                minimum = cost
                bestCandidate = neighbor
                conflictNodes = tempConflictNodes
        if minimum < bestFitness: # update best (take a step towards the better neighbor)
            bestFitness = minimum
            best = bestCandidate
            local_counter = 0
        elif minimum == bestFitness: # to detect local optimum
            local_counter += 1
        if bestFitness < globalFitness: # update the best solution found until now
            globalBest = best
            globalFitness = bestFitness
        tabu.append(bestCandidate)
        tabuDict[str(bestCandidate)] = True
        if len(tabu) > args.maxTabu:
            tabuDict[str(tabu[0])] = False
            tabu.pop(0)
        if local_counter == args.localOptStop: # if fallen into local optimum, reset and continue with the algorithm
            if bestFitness < globalFitness:
                globalBest = best
                globalFitness = bestFitness
            bestCandidate = assignment
            best = bestCandidate
            conflictNodes, bestFitness = getFitness(assignment, domains, fitnessType)
            local_counter = 0
            tabuDict = {str(bestCandidate): True}
            print('Generation time: ', time.time() - iterTime)
            printSolution(best)
            print('cost = ', bestFitness)
            print()
        print('Time elapsed: ', time.time() - startTime)
    return globalBest, globalFitness
```

GA ALGORITHM:

כמו המעבדות הקודמות, אנחנו מסתכלים על הבעיה כפרמוטציה בגודל ח, (מספר הצמתים בגרף) וכל קודקוד יהיה לו צבע רנדומלי בין 1...K, ואת פונקציית FITNESS- תהיה ביחס למספר התנטשוויות, וכמוון אנחנו נחפש פרמוטציה עם הכי פחות התנטשוויות. כשתגיע לאפס אז צביעה חוקית.

```
def run(self):
    startTime = time.time()
    self.initPopulation()

    repeat = 0
    bestFitness = float('inf')

    best = []
    genBestFit = 0

    for _ in range(self.args.GA_MAXITER):
        iterTime = time.time()
        self.calcFitness()
        self.sortByFitness()
        self.printBest()
        self.calcAvgSd()

        best = self.population[0].getString()[:]
        genBestFit = self.population[0].getFitness()

        if bestFitness == genBestFit:
            repeat += 1
        elif genBestFit < bestFitness:
            bestFitness = genBestFit
            repeat = 1
        # this checks if we have reached a local optimum or found the goal
        if repeat == self.args.LOCAL_STOP_ITER or self.population[0].getFitness() == 0:
            print('Generation time: ', time.time() - iterTime)
            print()
            break

        self.mate()
        try:
            pass
        except:
            print('Generation time: ', time.time() - iterTime)
            break
        self.swap()
        self.aging()
        print('Generation time: ', time.time() - iterTime)
        print()
    print('Time elapsed: ', time.time() - startTime)
    return best, genBestFit
```

ה. עלייכם להשוות בין האלגוריתמים וההיוריסטיות עפי הクリיטריונים הבאים:

- a. זמן ריצה
- b. כמות STATES שנsparkים
- c. איקות הפתרון
- d. ולنمך את בחירתכם בבחירה האלגוריתמים וההיוריסטיות שהשתמשתם בהם

לגביו זמן ריצה:

השאינו את זה לשלית המשתמש.(התוצאות המופיעות מטה זה עברו SEC 30)
כמות STATES

	Back.	Forw.	FESA	KEM	HYB	GA
le450_5a.col	459	450	98	37	30	11
le450_5b.col	452	450	93	35	31	12
le450_15a.col	450	450	65	25	21	8
le450_15b.col	463	450	69	26	28	8
le450_25a.col	450	450	64	28	24	8
le450_25b.col	474	450	65	22	20	8
myciel3.col	11	11	3000	3000	2859	30
myciel4.col	21	23	3000	2900	3000	37
queen7_7.col	37	49	552	75	50	84
queen8_8.col	68	64	340	60	42	55

aicot hafturon: (חשוב לנו לציין שלא צירפנו תמונות להרצות כי יש המון הרצות ולא רוצים למלא את הדוח בתמונות)

	Back.	Forw.	FESA	KEM	HYB	GA	OPT
le450_5a.col	8	8	10	11	10	10	5
le450_5b.col	7	7	10	12	11	11	5
le450_15a.col	17	17	18	18	16	17	15
le450_15b.col	17	17	18	17	18	17	15
le450_25a.col	25	25	26	26	26	25	25
le450_25b.col	25	25	25	25	25	25	25
myciel3.col	4	4	4	4	4	4	4
myciel4.col	5	5	5	5	5	5	5
queen7_7.col	7	7	9	9	11	10	7
queen8_8.col	10	10	11	12	12	11	8

לגביו בחירת אלגוריתם ה-TABU: בחרנו אותו מכיוון שבמקרה השני היה האלגוריתם הטוב ביותר מביון כל השאר ולכן צפינו שיתן גם בבעיה הנוכחית פתרונות טובים.

עבור היוריסטייקות שהשתמשנו בהן: הסבכנו את הסיבה לבחירתם במקומות המתאים. בכל זאת נDIGISH שהשתמשנו ביוריסטייקות לפי הרצאה, שכן אחת מתייחס לגישה משילה לפתרון הבעיה.