Artificial Intelligence Lab – CSP

Submission date :11/5/2021

Shadi Halloun – 313552309

Noor Khamaisi – 207076076

קישור ל GitHub שלנו , Source קוד וקובץ הPDF מופיעים בצורה מסודרת.

https://github.com/shadihalloun35/ArtificialIntelligenceLab.git

הקדמה:

התרגיל לא היה קל בכלל , הוא דרש הרבה השקעה והרבה זמן ובמיוחד שהאלגוריתמים שדרושים בתרגיל הם לא קלים יחסית ואפילו אין עליהם הרבה חומר באינטרנט , אבל אחרי הרבה השקעה ואינסוף חיפושים ב GOOGLE הצלחנו סוף סוף להתגבר על התרגיל ולאתגר את עצמנו.

אנחנו מצפים אחרי הניסוי למצוא קשר בין דרגת צפיפות הגרף לבין הזמן שבו מוצאת התוכנית את צביעה מינימלית.

הבעיה היא בעיה NP קשה.

אין פתרון דטרמיניסטי ואופטימלי שפותר את הבעיה הזאת בזמן פחות מאקפוננציאלי מאורך הקלט.

ייצגנו את הגרף כמטריצת שכיוניות דו ממדית (Adjacency Matrix).

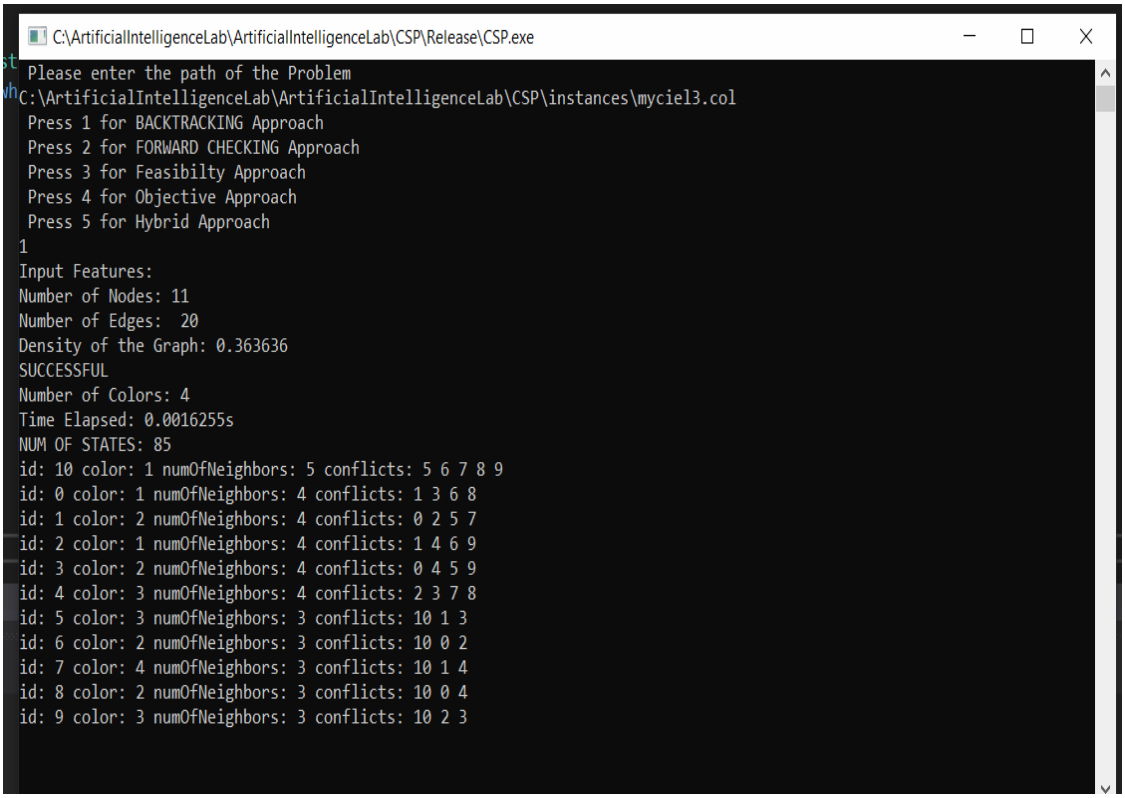
עבור החלק הראשון כדי להפעיל את התוכנה יש להריץ את ה EXE FILE , להכניס את ה-PATH של הקלט ואז לבחור את אחת הגישות שהיה צריך לממש.

חלק א':

1- עבור ה PATH הזה לדוגמא :

C:\ArtificialIntelligenceLab\ArtificialIntelligenceLab\CSP\instances\myciel3.col

המאפיינים מופיעים תחת הכותרת INPUT FEATURES :



קריאת הקלט :

הקטע הזה מטפל בצמתים

```
void Objective::ActivateObjective(string filePath,int timeAllowed)
{
    using clock = std::chrono::system_clock;
    using sec = std::chrono::duration<double>;
    const auto before = clock::now();               // for elapsed time
    Matrix mtx;
    int* nodes = NULL;
    Node* sorted = NULL;

    srand(unsigned(time(NULL)));
    counter3 = 0;
    ifstream input(filePath);
    if (!input.is_open())
        cout << "Error Opening a file.\n";
    else {
        string x;
        while (getline(input, x)) {
            if (x.size() != 0) {
                if (x[0] == 'c')                //ignore
                    continue;
                if (x[0] == 'p') {              //build the adjacency matrix and nodes
                    int i = 7;                  //to get the dimensions
                    while (x[i] != ' ') {
                        i++;
                    }
                    string line = x;
                    x = x.substr(7, i);
                    cout << "Input Features:" << std::endl;
                    cout << "Number of Nodes: " << stoi(x) << std::endl;
                    cout << "Number of Edges: " << line.substr(i, line.size() - 1) << std::endl;
                    int nodesNum = stoi(x);
                    int edgesNum = stoi(line.substr(i, line.size() - 1));
                    cout << "Density of the Graph: " << edgesNum * nodesNum*(nodesNum - 1) << std::endl;
                    createMatrix(&mtx, stoi(x));
                    nodes = new int[mtx.dimension];
                    sorted = new Node[mtx.dimension];
                    CreateObjectiveNodes(sorted, mtx.dimension);
```

הקטע הזה מטפל ב EDGES:

```
if (x[0] == 'e') {
    int firstNode, secondNode;
    string tmp;
    int i = 2;
    while (x[i] != ' ')
        i++;
    tmp = x.substr(2, i);
    firstNode = stoi(tmp);
    tmp = x.substr(i + 1, x.size());
    secondNode = stoi(tmp);
    mtx.adj[firstNode - 1][secondNode - 1] = 1;
    mtx.adj[secondNode - 1][firstNode - 1] = 1;
    sorted[firstNode - 1].numOfNeighbors++;
    sorted[secondNode - 1].numOfNeighbors++;
}
```

חלק ב':

a.

Backtracking with Back jumping

- התחלנו לצבוע צמתים לפי מספר השכנים (סדר יורד)

Largest Degree Ordering (LDO): It chooses a vertex with the highest number of neighbors

```
void BackJumping::ordering(Matrix mtx, BackJumpingNode* sorted, int* nodes) { //Order neighbors by numOfNeighbors
    sort(sorted, sorted + (mtx.dimension), sortByNeighbors);
    for (int i = 0; i < mtx.dimension; i++) {
        nodes[sorted[i].id] = i;
    }
}
```

- האלגוריתם בכלל:

```cpp
void BackJumping::backtracking(Matrix mtx, BackJumpingNode* nodes, int* org, int index) { //Backtracking with Back jumping
    int Index = index;
    while (1) {
        if (Index >= mtx.dimension) {
            cout << "SUCCESSFUL" << endl;
            cout << "Number of Colors: " << mtx.colors << endl;
            return;
        }
        counter++;
        if (nodes[0].color > 1) {
            resetBackJumpingNodes(mtx, nodes, mtx.dimension); //reset the graph
            mtx.colors++; //increase number of colors available
            Index = 0;
            continue;
        }
        if (!tryColor(mtx, nodes, org, Index)) { //try to color current node in the next color available
            int tmp;
            if (!nodes[Index].conflicts.empty()) {
                tmp = nodes[Index].conflicts.back();
                tmp = org[tmp];
            }
            else {
                tmp = Index - 1;
            }

            backjump(mtx, nodes, org, Index, tmp); //backtrack to the last node that is in the coflict set

            fixNeighbors(mtx, nodes, org, nodes[tmp].id); //fix conflict set and colors of the nodes that has been effected

            Index = tmp;
```

Output

```cpp
BackJumping                                  backtracking(Matrix mtx,
            continue;
        }
        if (!tryColor(mtx, nodes, org, Index)) { //try to color current node in the next color available
            int tmp;
            if (!nodes[Index].conflicts.empty()) {
                tmp = nodes[Index].conflicts.back();
                tmp = org[tmp];
            }
            else {
                tmp = Index - 1;
            }

            backjump(mtx, nodes, org, Index, tmp); //backtrack to the last node that is in the coflict set

            fixNeighbors(mtx, nodes, org, nodes[tmp].id); //fix conflict set and colors of the nodes that has been effected

            Index = tmp;
            if (tmp == 0) {
                resetBackJumpingNodes(mtx, nodes, mtx.dimension);
                mtx.colors++;
                Index = 0;
                continue;
            }
            continue;
        }
        else {
            updateNeighbors(mtx, nodes, org, nodes[Index].id); //update conflict set
            Index++;
            continue;
        }
    }
}
```

- הפונקציה המנסה לצבוע צומת בודד.

```
bool BackJumping::tryColor(Matrix mtx, BackJumpingNode * nodes, int * org, int index)
{
    for (int i = (nodes[index].color + 1); i <= mtx.colors; i++) { //tries all colors
        int flag = 0;
        for (int j = 0; j < mtx.dimension; j++) { //checks availablity
            if (mtx.adj[nodes[index].id][j] == 1) {
                if (nodes[org[j]].color == i) {
                    flag = 1;
                    break;
                }
            }
        }
        if (flag)
            continue;
        else {
            nodes[index].color = i;
            return true;
        }
    }
    return false;
}
```

- fixNeighbors: בשלב ה-backtracking מוחקים את הצומת שאנחנו חוזרים ממנו מה-conflict set של שכניו.

```
void BackJumping::fixNeighbors(Matrix mtx, BackJumpingNode* nodes, int* org, int index) {
    for (int i = 0; i < mtx.dimension; i++) {
        if (mtx.adj[index][i] == 1) {
            for (size_t j = 0; j < nodes[org[i]].conflicts.size(); j++) {
                if (nodes[org[i]].conflicts[j] == index) {
                    nodes[org[i]].conflicts.erase(nodes[org[i]].conflicts.begin() + j);
                    break;
                }
            }
        }
    }
```

- backjump: הפונקציה החוזרת למצב קודם וקוראת לfixNeighbor כדי לשחזר את מצב הגרף קודם.

```
void BackJumping::backjump(Matrix mtx, BackJumpingNode* nodes, int* org, int index, int des) {
    int x = index;
    while (x >= 0 && x > des) {
        nodes[x].color = 0;
        fixNeighbors(mtx, nodes, org, nodes[x].id);
        x--;
    }
}
```

b.

Forward Checking & Arc Consistency

- האלגוריתם :

coloredNodes מכילה את כל הצמתים שכבר צבענו.

deleted הוא vector שמכיל Deleted שמייצג את כל השינויים שנעשו בגרף. Deleted מכל הצומת שצובעים אותו עכשיו, צומת שה-domain שלו השתנה ואת הצבע.

```
void ForwardChecking::FCandAC(Matrix mtx, ForwardCheckingNode* nodes, int index) { //Forward Checking & Arc Consistency
    stack<int> coloredForwardCheckingNodes;
    vector<Deleted> deleted;
    int Index = index;
    while (1)
    {
        counter1++;
        if (Index == 0 && nodes[0].color != 0)
        {
            mtx.colors++;
            resetForwardCheckingNodes(mtx, nodes, mtx.dimension);
            while (!coloredForwardCheckingNodes.empty())
                coloredForwardCheckingNodes.pop();
            deleted.clear();
            continue;
        }
        fixNeighbors(mtx, nodes, Index, &deleted);
        if (!tryColor(mtx, nodes, Index, &deleted))
        {
            if (Index == 0) {
                mtx.colors++;
                resetForwardCheckingNodes(mtx, nodes, mtx.dimension);
                while (!coloredForwardCheckingNodes.empty())
                    coloredForwardCheckingNodes.pop();
                deleted.clear();
                continue;
            }
            int tmp = coloredForwardCheckingNodes.top();
            coloredForwardCheckingNodes.pop();
```

```
ForwardChecking                                  FCandAC(Matrix mtx, ForwardCheckingNode * n
            mtx.colors++;
            resetForwardCheckingNodes(mtx, nodes, mtx.dimension);
            while (!coloredForwardCheckingNodes.empty())
                coloredForwardCheckingNodes.pop();
            deleted.clear();
            continue;
        }
        int tmp = coloredForwardCheckingNodes.top();
        coloredForwardCheckingNodes.pop();
        nodes[Index].color = 0;
        Index = tmp;
        continue;
    }
    else
    { //succeeded coloring
        coloredForwardCheckingNodes.push(Index);
        Index = nextForwardCheckingNode(mtx, nodes, Index);
        if (Index == -1)
        {
            printForwardCheckingNodes(mtx, nodes);
            cout << endl;
            cout << "SUCCESSFUL" << endl;
            cout << "Number of Colors: " << mtx.colors << endl;
            return;
        }
        continue;
    }
}
```

```
}

void ForwardChecking::forwardChecking(Matrix mtx, ForwardCheckingNode* nodes, int index, int color, vector<Deleted>* deleted) {
    for (int i = 0; i < mtx.dimension; i++) {
        if (mtx.adj[index][i] == 1 && nodes[i].color == 0) {
            Deleted tmp;
            tmp.node1 = nodes[index];
            tmp.node2 = nodes[i];
            tmp.color = color;
            (*deleted).push_back(tmp);
            deleteColorFromDomain(nodes, i, color);
        }
    }
}

bool ForwardChecking::tryColor(Matrix mtx, ForwardCheckingNode* nodes, int index, vector<Deleted>* deleted) {
    for (int i = (nodes[index].color + 1); i <= mtx.colors; i++) { //tries all colors
```

- בחירת הצומת הבא לצביעה:

השתמשנו בהיוריסטיקת הבחירה MRV (Most Restricted Variable)

```
int ForwardChecking::nextForwardCheckingNode(Matrix mtx, ForwardCheckingNode* nodes, int index) {
    int cand = -1;
    size_t val = mtx.dimension + 1;
    for (int i = 0; i < mtx.dimension; i++) { //MRV
        if (mtx.adj[index][i] == 1 && nodes[i].color == 0) {
            if (nodes[i].domain.size() < val) {
                val = nodes[i].domain.size();
                cand = i;
            }
        }
    }
    if (cand == -1) { // all neighbors are colored
        for (int i = 0; i < mtx.dimension; i++) {
            if (nodes[i].color == 0)
                return i;
        }
        return -1;
    }
    else {
        return cand;
    }
}
```

- Arc Consistency:

אם consistent=0 אז קיבלנו גרף קונסיסטנטי

אם consistent=2 אז קיבלנו גרף שה-domain שלו ריק לכן לא קונסיסטנטי (וצריך לבצע backtracking)

```
bool ForwardChecking::arcConsistency(Matrix mtx, ForwardCheckingNode* nodes, int source, vector<Deleted>* deleted) {
    int consistent = 1; // 0 -> consistent
    while (consistent == 1) {
        for (int i = 0; i < mtx.dimension; i++) {
            consistent = checkConsistency(mtx, nodes, source, i, deleted);
            if (consistent == 0)
                return true;
            if (consistent == 2)
                return false;
        }
    }
    return true;
}
```

אנחנו מצפים אחרי הניסוי למצוא קשר בין דרגת צפיפות הגרף לבין הזמן שבו מוצאת התוכנית את הפתרון.

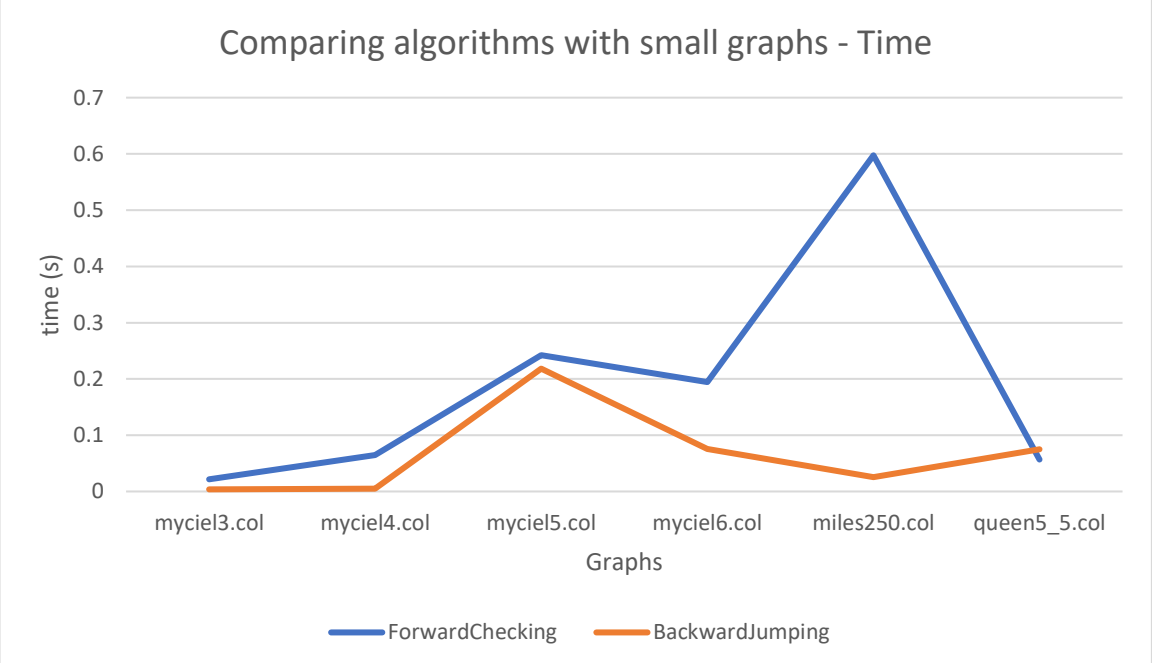
ביצוע האלגוריתם Backtracking with Back jumping:

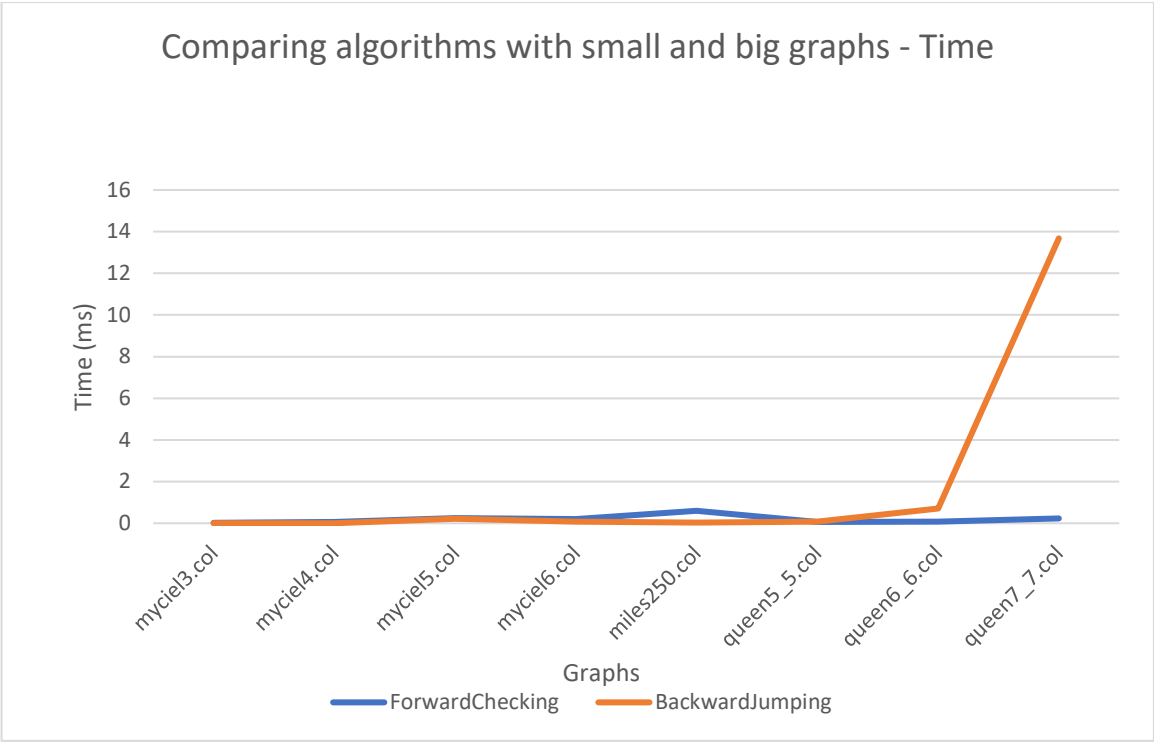
Test name	|E|	|V|	|States|	Elapsed time(s)	#Colors
myciel3.col	20	11	85	0.00374	4
myciel4.col	71	23	1424	0.0048218	5
myciel5.col	236	47	707525	0.218316	6
myciel6.col	755	95	35694	0.075347	7
miles250.col	774	128	12582	0.0256729	8
queen5_5.col	320	25	110	0.0748707	5
queen6_6.col	580	36	463420	0.700396	7
queen7_7.col	952	49	8373327	13.678	8

ביצוע האלגוריתם Backtracking with Forward checking

Test name	|E|	|V|	|States|	Elapsed Time(s)	#Colors
myciel3.col	20	11	34	0.0216522	4
myciel4.col	71	23	79	0.064918	5
myciel5.col	236	47	144	0.242285	6
myciel6.col	755	95	247	0.194444	7
miles250.col	774	128	271	0.597428	8
queen5_5.col	320	25	41	0.0565123	5
queen6_6.col	580	36	214	0.0841219	9
queen7_7.col	952	49	282	0.230837	10

גרף השוואה בזמני ביצוע בין שני האלגורתמים:



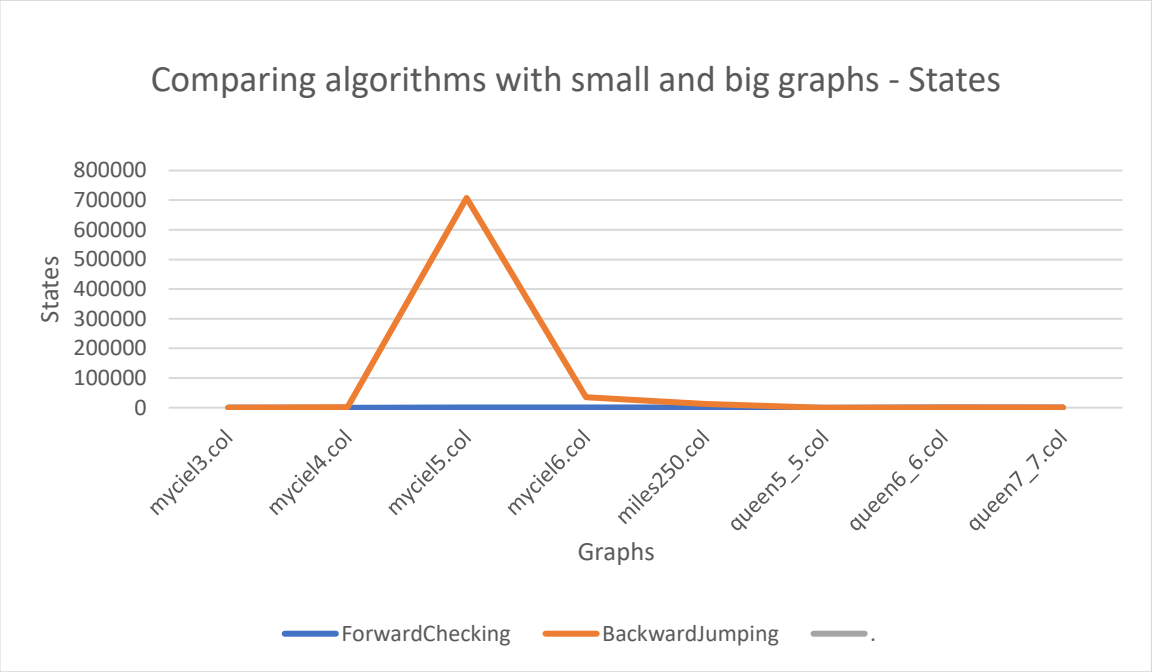


אפשר לראות משני הגרפים הנ"ל שעבור קלטים קטנים Back Jumping מוציא פתרון בזמן פחות יותר מזמן מציאת הפתרון כשמשתמשים ב Forward Checking & Arc Consistency.

זה נובע מהסיבה שבגרפים קטנים יחסית מספר ה-Backtracking סביר והוא לא "מבזבז" זמן על בדיקות קדימה.

אך לעומת זאת בגרפים גדולים יש הפרש עצום בין זמן ביצוע אלגוריתם Back jumping לבין Forward Checking, וזה נובע מהעובדה ש-Backjumping עובד בלי בדיקות קדימה ובמקרה של תקלה חוזר ומנסה לסדר את הבעיה ובכך הוא מגדיל את זמן העבודה שלו, אלה ה Forward Checking & Arc Consistency פוסל וחוסך אפשרויות.

גרף השוואה במספר STATES שנסרקו בשני האלגורתמים:



אלגוריתם Back jumping מסתמך על זה שהוא בודק כל האפשריות ובמקרה של בעיה חוזר אחורה לעומת אלגוריתם Forward checking & Arc Consistency בודק אחרי כל מצב איזה מצבים לא חוקיים נוספו ופוסל אותם ואז ממשיך בדרך הצביעה, ומפה רואים שפעולתו העיקרית לפני ההתקדמות היא לפסול מקרים לא חוקיים, כל הנ"ל מתבטא בגרף שיצרנו ובכך שבכל הטסטים שנבדקו מספר הצמתים שעוברים עליהם ב Back jumping הרבה יותר גדול ממספר הצמתים שעוברים עליהם ב Forward checking.

עבור החלק השני כדי להפעיל את התוכנה יש להריץ את ה EXE FILE , להכניס את ה-PATH של הקלט ,לבחור את אחת הגישות שהיה צריך לממש ואז להכניס את הזמן המוקצה להרצה.

חלק ג' + ד':

בחלק זה נעבוד על השוואה בין ביצועי 3 גישות לפתירת בעיית צביעת גרף שלושת הגישות בחלק זה הן מתאימות לשיטת החיפוש הלוקאלי, הגישה הראשונה היא **הגישה הפיזבילית** שמימשנו בעזרת אלגוריתם החיפוש הלוקאלי **MIN CONFLICT** ,הגישה השנייה היא **פונקציית המטרה** שבעזרת אלגוריתם Simulated Annealing ו-Kempe Chain דאגנו למקסם $\sum_{i=1}^{n}|C_i|^2$ כך ש n הוא מספר הצבעים ו- $|C_i|$ הוא מספר הצמתים הצבועים ב- i, הגישה השלישית היא **הגישה ההיברידית** שבעזרת אלגוריתם החיפוש **GENETIC ALGORITHM** דאגנו להקטין

$$\sum_{i=1}^{n} 2|C_i||B_i| - \sum_{i=1}^{n}|C_i|^2$$

כך ש n הוא מספר הצבעים, $|C_i|$ הוא מספר הצמתים הצבועים ב- i ו- $|B_i|$ הוא מספר ה-Bad Edges שקצותיו צבועים ב-i.

a.

Min Conflicts Algorithm

- האלגוריתם:

```
void Feasibility::MinimalConflict(Matrix & mtx, Node* nodes, std::chrono::duration<double> timeAllowed) {
    using clock = std::chrono::system_clock;
    using sec = std::chrono::duration<double>;
    const auto before = clock::now();
    sec duration = clock::now() - before;
    greedyAlgorithm(&mtx, nodes);                          //colors the graph greedly
    int maxColors = mtx.colors;
    while (mtx.colors > 1 && duration < timeAllowed) {              //try to reduce number of colors
        counter2++;
        if (isLegal(mtx, nodes)) {
            mtx.colors = maxColors;
            maxColors--;
            fixGraph(mtx, nodes, maxColors);               //change color randomly
        }
        else {
            tryFix(mtx, nodes, maxColors);                 //try finding a coloring that makes the graph feasibile
        }

        duration = clock::now() - before;
    }
    cout << "SUCCESSFUL" << endl;
    cout << "Number of Colors: " << mtx.colors << endl;
}
```

מתחילים בצביעת הגרף בצורה חמדנית (greedy) שזמן ריצתו פולינומי באורך הקלט וכמובן הגרף שנוצר הוא גרף חוקי.

בוחרים צבע אקראי שרוצים למחוק אותו וצובעים את כל הצמתים שצבועים בצבע זה בצבע אחר.

המטרה שלנו להפוך את הגרף שלנו לחוקי וזאת הגישה הפיזבילית.

- הצביעה החמדני:

```
void Feasibility::greedyAlgorithm(Matrix* mtx, Node* nodes) {
    for (int i = 0; i < (*mtx).dimension; i++) {
        int color = 1, j = 0;
        while (j < i) {
            if ((*mtx).adj[i][j] == 1 && nodes[j].color == color) {
                color++;
                j = 0;
            }
            else
                j++;
        }
        nodes[i].color = color;
        if (color > (*mtx).colors)
            (*mtx).colors = color;
    }
    cout << (*mtx).colors;
}
```

- הפונקציה המנסה להפוך את הגרף לפיזבילי:

על פעם היא בוחרת צומת אקראי ובודקת מהו הצבע "הטוב" ביותר כלומר הצבע שעבורו מספר השכנים הצבועים באותו צבע הוא קטן ביותר

(min-conflict) וצובעת אותו בצבע זה.

```
void Feasibility::tryFix(Matrix & mtx, Node* nodes, int maxColors) {
    int randNode = (rand() % mtx.dimension);                    //choose a node randomly
    int tmp = 1;
    int minConflict = mtx.dimension + 1;
    minConflict = calcConflict(mtx, nodes, randNode, tmp);
    for (int i = 2; i <= maxColors; i++) {
        int tmp2;
        tmp2 = calcConflict(mtx, nodes, randNode, i);
        if (tmp2 < minConflict) {
            minConflict = tmp2;
            tmp = i;
        }

    }
    nodes[randNode].color = tmp;
}
```

Simulated Annealing

Kempe Chain

הגישה ששמה פונקציית המטרה במרכזה

הגישה הזאת מטרתה למקסם $\sum_{i=1}^{n}|C_i|^2$ כך ש n הוא מספר הצבעים ו- $|C_i|$ הוא מספר הצמתים הצבועים ב- i.

- Simualted Annealing:

מתחילים עם טמפרטורה גבוהה שמוגדרת מראש ובכל שלב הטמפרטורה יורדת בהדרגה. כאשר הטמפרטורה גבוהה אז כל שינוי בגרף נקבל אותו. כאשר הטמפרטורה פוחתת הסיכוי שגרף "לא טוב" (כלומר קיבלנו $\sum_{i=1}^{n}|C_i|^2$ יותר גרוע) שנקבל אותו הולכת ופוחתת. אלגוריתם זה עוזר לנו לא להיתקע ב-Local Maxima

```
void Objective::SimulatedAnnealing(Matrix mtx, Node* nodes, std::chrono::duration < double> timeAllowed) {

    using clock = std::chrono::system_clock;
    using sec = std::chrono::duration<double>;
    const auto before = clock::now();
    sec duration = clock::now() - before;

    int startTemp = STARTING_TEMP;
    double endTemp = double(ENDING_TEMP);
    double temp = double(startTemp);
    int Ci = GreedyAlgorithm(&mtx, nodes);
    while (mtx.colors > 1 && duration < timeAllowed) {
        counter3++;
        int* oldColors = new int[mtx.dimension];
        CopyColors(nodes, oldColors, mtx.dimension);
        int tmp = KempeChain(&mtx, nodes);
        if (tmp > Ci) {
            Ci = tmp;
        }
        else {
            if ((rand() % startTemp) > temp) {
                Retreat(nodes, oldColors, mtx.dimension);
            }
            else
            {
                Ci = tmp;
            }
        }
        temp *= COOLING_RATE;
        duration = clock::now() - before;

    }
    cout << "SUCCESSFUL" << endl;
    cout << "Number of Colors: " << mtx.colors << endl;
}
```

- Kempe Chain: מנסים למקסם $\sum_{i=1}^{n}|C_i|^2$ בעזרת Kempe Chain ע"י בחירת שתי מחלקות של צבעים והחליף צבעים בין שתי המחלקות אבל חייבים לשמור על הפיזביליות כלומר שיישאר הגרף חוקי.

```
int Objective::KempeChain(Matrix* mtx, Node* nodes) {
    int randNode = (rand() % (*mtx).dimension);
    int color1 = nodes[randNode].color;
    int color2 = ((rand() % (*mtx).colors) + 1);
    vector<int> cand1;
    vector<int> cand2;
    cand1.clear();
    cand2.clear();
    cand1.push_back(randNode);
    while (!cand1.empty()) {
        ChangeColor(&cand1, nodes, color2);
        PushConflicted(mtx, nodes, &cand2, &cand1, color2);
        int tmp = color1;
        color1 = color2;
        color2 = tmp;
        SwapCand(&cand1, &cand2);

    }
    return CalcCi(mtx, nodes, mtx->dimension, mtx->colors);
}
```

Genetic Algorithm

הגישה ההיברידית

בעזרת אלגוריתם החיפוש **GENETIC ALGORITHM** דואגים להקטין

$\sum_{i=1}^{n} 2|C_i||B_i| - \sum_{i=1}^{n}|C_i|^2$ (פונקציית המטרה)

כך ש n הוא מספר הצבעים, $|C_i|$ הוא מספר הצמתים הצבועים ב- i ו-

$|B_i|$ הוא מספר ה-Bad Edges שקצותיו צבועים ב-i.

בעזרת אלגוריתם greedy נקבל מספר צביעות חוקי מסוים k ומתחילים

כל פעם להקטין את k ולקבל גרף חוקי (הגישה הפיזבילית)

- חישוב ה-Fitness:

$$\sum_{i=1}^{n} 2|C_i||B_i| - \sum_{i=1}^{n}|C_i|^2$$

```
nt Hybrid::calc_fitness(ga_vector &population, Matrix* mtx, int maxColors)

    int flag = 1;
    int flag2 = 0;
    int tsize = mtx->dimension;
    int fitness;
    avg = 0;
    dev = 0;

    for (int i = 0; i < GA_POPSIZE; i++) {
        flag = 1;
        fitness = 0;
        int CiS = 0;
        for (int j = 1; j <= maxColors; j++) {
            int Bj = calcBj(population[i], mtx, j);
            if (Bj != 0)
                flag = 0;
            int Cj = calcCj(population[i], mtx, j);
            CiS += pow(Cj, 2);
            fitness += (2 * Bj*Cj);
        }
        fitness = fitness - CiS;
        population[i].fitness = fitness;
        avg += fitness;
        if (flag == 1) {
            flag2 = 1;
        }

    }
    avg /= GA_POPSIZE;

    for (int i = 0; i < GA_POPSIZE; i++) {
        dev += (pow(population[i].fitness - avg, 2));
    }
    dev /= GA_POPSIZE;
    dev = sqrt(dev);
    return flag2;

utput
```

- שיטת מוטציה:

Order Mutation (OM) changing the order of some (few) vertices.

```
void Hybrid::orderMutate(ga_struct &member, int tsize)
{
    int num = ((rand() % MAX_OM) + 2);
    vector<int> cand1;
    vector<int> cand2;
    for (int i = 0; i < tsize; i++) {
        if (cand1.size() >= num)
            break;
        if (rand() < OM_RATE) {
            cand1.push_back(member.colors[i]);
            cand2.push_back(i);
        }
    }
    random_shuffle(cand1.begin(), cand1.end());
    while (!cand2.empty()) {
        member.colors[cand2.size() - 1] = cand1[cand1.size() - 1];
        cand2.pop_back();
        cand1.pop_back();
    }
}
```

חלק ה':

ביצועים הגישה הפיזבילית עם אלגוריתם MINCONFLICT:

Test name	|E|	|V|	|States|	Elapsed Time	#Colors
myciel3.col	20	11	28163480	15.0545	4
myciel4.col	71	23	9105431	15.2131	5
myciel5.col	236	47	5444599	15.1385	6
myciel6.col	755	95	3711061	15.7799	7
miles250.col	774	128	1012471	15.5869	9
queen5_5.col	320	25	12713987	15.0615	7
queen6_6.col	580	36	7898449	15.0922	9
queen7_7.col	952	49	6883177	15.1069	9

אפשר לראות שהמימוש עם אלגוריתם MINIMAL CONFLICT,

יכול למצוא צביעה מינימלית של גרפים צפופים בזמן סביר .

Simulated Annealing

Kempe Chain

הגישה ששמה פונקציית המטרה במרכזה:

Test name	|E|	|V|	|States|	Elapsed Time	#Colors
myciel3.col	20	11	8968243	10.0104	4
myciel4.col	71	23	3935773	10.0276	5
myciel5.col	236	47	1731896	10.0592	6
myciel6.col	755	95	1033012	10.189	7
miles250.col	774	128	2956805	10.2417	8
queen5_5.col	320	25	2844988	10.044	5
queen6_6.col	580	36	2346187	10.0635	7
queen7_7.col	952	49	1811136	10.0777	7

אנחנו ניסינו לעבוד עם ה MINIMAL CONFLICT, וזה נתן לנו תוצאות רחוקות יחסית לכן ניסנו את ה SIMULATED וראינו שהמימוש עם אלגוריתם SIMULATED ANNEALING, נותן לנו תוצאות יותר טובות .

ביצועים הגישה ההיברידית עם אלגוריתם GENETIC:

Test name	|E|	|V|	|Generations|	Elapsed Time	#Colors
myciel3.col	20	11	1000	2.84535	4
myciel4.col	71	23	1000	6.33858	5
myciel5.col	236	47	482	10.1301	6
myciel6.col	755	95	273	20.207	7
miles250.col	774	128	248	20.3205	9
queen5_5.col	320	25	1000	6.56133	6
queen6_6.col	580	36	1000	12.5221	9
queen7_7.col	952	49	940	20.1067	10

לא היה לנו ברירה רק לממש את ה GENETIC כי המרצה ביקש לממש את GENETIC ALGORITM עם אחת הגישות