Introduction

Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many seemingly unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Constraint Programming (CP) is the field of research that specifically focuses on tackling these kinds of problems.[1][2] Additionally, boolean satisfiability problem (SAT), the satisfiability modulo theories (SMT), mixed integer programming (MIP) and answer set programming (ASP) are all fields of research focusing on the resolution of particular forms of the constraint satisfaction problem.

مقدمة:

مشاكل الرضا المقيّد (CSPs) هي أسئلة رياضية تُعرّف على أنها مجموعة من الكائنات التي يجب أن تلبي حالتها عددًا من القيود أو القيود. يمثل CSPs الكيانات في مشكلة ما كمجموعة متجانسة من القيود المحدودة على المتغيرات، والتي يتم حلها من خلال طرق إرضاء القيد. CSPs هي موضوع البحث في كل من الذكاء الاصطناعي وبحوث العمليات ، حيث أن الانتظام في صياغتها يوفر أساسًا مشتركًا لتحليل وحل مشاكل العديد من العائلات التي تبدو غير مرتبطة. غالبًا ما يُظهر CSPs تعقيدًا كبيرًا ، مما يتطلب مجموعة من الأساليب التجريبية وطرق البحث التجميعية ليتم حلها في وقت معقول. البرمجة القيدية (CP) هي مجال البحث الذي يركز بشكل خاص على معالجة هذه الأنواع من المشاكل. [١] [٢] بالإضافة إلى ذلك ، فإن مشكلة الرضا المنطقية (SAT) ، ونظريات نموذج الرضا (SMT) ، وبرمجة الأعداد الصحيحة المختلطة (MIP) وبرمجة مجموعة الإجابات (ASP) كلها مجالات بحث تركز على حل أشكال معينة من مشكلة رضا القيد.

Objectives:

In this paper we will write an interactive AI tutorial about Constraint Satisfaction Problems. we will write it as a Jupyter notebook which implements and explains some of the essential algorithms we have seen in class. The Jupyter notebook is to explain the algorithms and related concepts ,and most importantly it has to explain how to implement the algorithms from scratch. The notebook also contain appropriate formulated problem Arc Consistency AC-3, Backtracking with AC-3 and Forward Checking to which the algorithm is applied. Their performance evaluated and compared

الاهداف:

سنكتب في هذه الورقة درسًا تعليميًا تفاعليًا للذكاء الاصطناعي حول مشاكل الرضا المقيدة. سنكتبه كمفكرة Jupyter التي تنفذ وتشرح بعض الخوارزميات الأساسية التي رأيناها في الفصل. تهدف مفكرة Jupyter إلى شرح الخوارزميات والمفاهيم ذات الصلة ، والأهم من ذلك أنها يجب أن تشرح كيفية تنفيذ الخوارزميات من البداية. يحتوي الكمبيوتر الدفتري أيضًا على مشكلة مصاغة بشكل مناسب ، تناسق القوس AC-3 ، التراجع باستخدام AC-3 والتحقق الأمامي الذي يتم تطبيق الخوارزمية عليه. تقييم أدائهم ومقارنته

Main concepts

CSP model of the Problem

the CSP is defined by

• X = {X1, X2 ,..., Xn} set of n variables.

• D = {D1, D2 ,..., Dn} set of finite domains for each variable.

• C = {C1, C2 ,..., Cm} set of constraints which are restrictions on the values that one or more variables can take.

• R = (R1, R2 ,..., Rm) set of relationships.

• Solution: is an assignment of values to all the variables that satisfy all the constraints.

المفاهيم الرئيسية

مشكلة رضا القيد يمكن ان تتعرف عن طريق

{X1, X2 ,..., Xn} = X •مجموعة من المتغيرات.n

{D1, D2 ,..., Dn} = D •مجموعة من المجالات المحددة لكل متغير.

{C1, C2 ,..., Cm} = C •مجموعة من القيود التي هي قيود على القيم التي يمكن أن يتخذها متغير واحد أو أكثر.

(R1, R2 ,..., Rm) = R •مجموعة من العلاقات.

•الحل: هو تخصيص قيم لجميع المتغيرات التي تفي بكل القيود.

Solution

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search. These techniques are also often

combined, as in the VLNS method, and current research involves other technologies such as linear programming.[13]

Backtracking is a recursive algorithm. It maintains a partial assignment of the variables. Initially, all variables are unassigned. At each step, a variable is chosen, and all possible values are assigned to it in turn. For each value, the consistency of the partial assignment with the constraints is checked; in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned. Several variants of backtracking exist. Back marking improves the efficiency of checking consistency. Back jumping allows saving part of the search by backtracking "more than one variable" in some cases. Constraint learning infers and saves new constraints that can be later used to avoid part of the search. Look-ahead is also often used in backtracking to attempt to foresee the effects of choosing a variable or a value, thus sometimes determining in advance when a sub problem is satisfiable or unsatisfiable.

Constraint propagation techniques are methods used to modify a constraint satisfaction problem. More precisely, they are methods that enforce a form of local consistency, which are conditions related to the consistency of a group of variables and/or constraints. Constraint propagation has various uses. First, it turns a problem into one that is equivalent but is usually simpler to solve. Second, it may prove satisfiability or unsatisfiability of problems. This is not guaranteed to happen in general; however, it always happens for some forms of constraint propagation and/or for certain kinds of problems. The most known and used forms of local consistency are arc consistency, hyper-arc consistency, and path consistency. The most popular constraint propagation method is the AC-3 algorithm, which enforces arc consistency.

Local search methods are incomplete satisfiability algorithms. They may find a solution of a problem, but they may fail even if the problem is satisfiable. They work by iteratively improving a complete assignment over the variables. At each step, a small number of variables are changed in value, with the overall aim of increasing the number of constraints satisfied by this assignment. The min-conflicts algorithm is a local search algorithm specific for CSPs and is based on that principle. In practice, local search appears to work well when these changes are also affected by random choices. An

integration of search with local search has been developed, leading to hybrid algorithms.

الحل:

عادة ما يتم حل مشكلات الرضا المقيدة في المجالات المحدودة باستخدام شكل من أشكال البحث. الأساليب الأكثر استخدامًا هي متغيرات التراجع ، وانتشار القيد ، والبحث المحلي. غالبًا ما يتم الجمع بين هذه التقنيات ، كما هو الحال في طريقة VLNS ، ويتضمن البحث الحالي تقنيات أخرى مثل البرمجة الخطية.

التراجع هو خوارزمية عودية. يحافظ على تخصيص جزئي للمتغيرات. في البداية ، لم يتم تحديد جميع المتغيرات. في كل خطوة ، يتم اختيار متغير ، ويتم تخصيص جميع القيم الممكنة له بدوره. لكل قيمة ، يتم التحقق من اتساق التخصيص الجزئي مع القيود ؛ في حالة الاتساق ، يتم إجراء مكالمة متكررة. عندما تمت تجربة جميع القيم ، تتراجع الخوارزمية. في خوارزمية التراجع الأساسية هذه ، يتم تعريف الاتساق على أنه تلبية جميع القيود التي يتم تعيين جميع المتغيرات الخاصة بها. توجد عدة أنواع من التراجع. Backmarking يحسن كفاءة التحقق من الاتساق. يسمح Backjumping بحفظ جزء من البحث عن طريق التراجع عن "أكثر من متغير واحد" في بعض الحالات. يقيد التعلم القيد ويحفظ القيود الجديدة التي يمكن استخدامها لاحقًا لتجنب جزء من البحث. غالبًا ما يستخدم التطلع إلى الأمام في التراجع لمحاولة التنبؤ بآثار اختيار متغير أو قيمة ، وبالتالي في بعض الأحيان تحديد مسبقًا متى تكون المشكلة الفرعية مرضية أو غير مرضية.

تقنيات الانتشار المقيّد هي طرق مستخدمة لتعديل مشكلة رضا القيد. بتعبير أدق ، هي طرق تفرض شكلاً من أشكال الاتساق المحلي ، وهي شروط تتعلق باتساق مجموعة من المتغيرات و / أو القيود. انتشار القيد له استخدامات مختلفة. أولاً ، يحول المشكلة إلى مشكلة مكافئة ولكن عادةً ما يكون حلها أبسط. ثانيًا ، قد يثبت أنه مرضي أو غير مرضي للمشكلات. هذا ليس مضمونًا أن يحدث بشكل عام ؛ ومع ذلك ، فإنه يحدث دائمًا لبعض أشكال انتشار القيد و / أو لأنواع معينة من المشاكل. أكثر أشكال الاتساق المحلي شهرةً واستخدامًا هي اتساق القوس ، والاتساق الفائق للقوس ، واتساق المسار. أكثر طرق انتشار القيد شيوعًا هي خوارزمية AC-3 ، والتي تفرض اتساق القوس.

طرق البحث المحلية هي خوارزميات مرضية غير كاملة. قد يجدون حلاً لمشكلة ما ، لكنهم قد يفشلون حتى لو كانت المشكلة مرضية. إنهم يعملون عن طريق التحسين المتكرر لتعيين كامل على المتغيرات. في كل خطوة ، يتم تغيير عدد صغير من المتغيرات في القيمة ، بهدف عام هو زيادة عدد القيود التي تفي بهذه المهمة. تعد خوارزمية الحد الأدنى من التعارضات خوارزمية بحث محلية خاصة بـ CSPs وتستند إلى هذا المبدأ. من الناحية العملية ، يبدو أن البحث المحلي يعمل بشكل جيد عندما تتأثر هذه التغييرات أيضًا بخيارات عشوائية. تم تطوير تكامل البحث مع البحث المحلي ، مما أدى إلى خوارزميات هجينة.

The algorithms explanation:

1- AC3:

The AC3 algorithm is one of most popular algorithms for arc-consistency. The main algorithm is a simple loop that selects and revises the constraints stored in Q until either no change occurs (Q is empty) or the domain of a

variable becomes empty. The first case ensures that all values of domains are consistent with all constraints, and second case returns that the problem has no solution.

To avoid many useless calls to the Revise procedure, AC3 keeps all the constraints Rij that do not guarantee that Di is arc-consistent with the constraints in a queue Q. Also, Q is updated by adding constraints Rki, which were, previously assessed where Dk can be inconsistent because Di was pruned. AC3 achieves arc-consistency on binary networks in $O(md^3)$ time and O(m) space, where d is the domain size and m is the number of binary constraints in the problem.

تعد خوارزمية AC3 واحدة من أكثر الخوارزميات شيوعًا لاتساق القوس. الخوارزمية الرئيسية عبارة عن حلقة بسيطة تحدد القيود المخزنة في Q وتراجعها حتى لا يحدث أي تغيير (Q فارغ) أو يصبح مجال المتغير فارغًا. تضمن الحالة الأولى أن جميع قيم المجالات متوافقة مع جميع القيود ، وتعيد الحالة الثانية أن المشكلة ليس لها حل.

لتجنب العديد من المكالمات غير المجدية لإجراء المراجعة ، يحتفظ AC3 بجميع القيود Rij التي لا تضمن أن يكون Di متسقًا مع القيود الموجودة في قائمة الانتظار Q. أيضًا ، يتم تحديث Q بإضافة قيود Rki ، والتي تم تقييمها مسبقًا حيث يمكن أن يكون Dk غير متسق لأن Di تم تقليمه. يحقق AC3 اتساق القوس على الشبكات الثنائية في وقت $O(md^3)$ ومساحة O (m) ، حيث يمثل d حجم المجال و m هو عدد القيود الثنائية في المشكلة.

Algorithm 1 Procedure AC3

Input: A CSP, $P = \langle X, D, R \rangle$
Output: **true** and P' (which is arc-consistent) or **false** and P' (which is arc-inconsistent because some domain remains empty)

```
1: for every arc R_ij ∈ R do
2:     Append (Q, (R_ij)) and Append (Q, (R_ji))
3: end for
4: while Q ≠ φ do
5:     select and delete R_ij from queue Q
6:     if Revise(R_ij) = true then
7:         if D_i ≠ φ then
8:             Append (Q, (R_ki)) with k ≠ i, k ≠ j
9:         else
10:            return false /*empty domain*/
11:        end if
12:    end if
13: end while
14: return true
```

Algorithm 2 Procedure Revise

Input: A $CSP\ P'$ defined by two variables $X = (X_i, X_j)$, domains D_i and D_j, and constraint R_{ij}.
Output: D_i, such that X_i is arc-consistent relative X_j and the boolean variable $change$

```
1: change ← false
2: for each a ∈ D_i do
3:     if ∄b ∈ D_j such that (X_i = a, X_j = b) ∈ R_ij then
4:         remove a from D_i
5:         change ← true
6:     end if
7: end for
8: return change
```

2- Backtracking

As a skeleton we use the simple backtracking algorithm that incrementally instantiates variables and extends a partial solution that specifies consistent values for some of the variables, toward a complete solution, by repeatedly choosing a value for another variable. After assigning a value to the variable, some consistency technique is applied to the constraint graph. Depending on the degree of consistency technique we get various constraint satisfaction algorithms.

Even simple backtracking (BT) performs some kind of consistency technique and it can be seen as a combination of pure generate & test and a fraction of arc consistency. The BT algorithm tests arc consistency among already instantiated variables, i.e., the algorithm checks the validity of constraints considering the partial instantiation. Because the domains of instantiated variables contains just one value, it is possible to check only those constraints/arcs containing the last instantiated variable. If any domain is reduced then the corresponding constraint is not consistent and the algorithm backtracks to a new instantiation.

The following procedure AC3-BT is called each time a new value is assigned to some variable Vcv (cv is the consecutive number of the variable in the order of instantiating variables)

كهيكل عظمي ، نستخدم خوارزمية التراجع البسيطة التي تعمل على إنشاء المتغيرات بشكل تدريجي وتوسع حلًا جزئيًا يحدد قيمًا متسقة لبعض المتغيرات ، نحو حل كامل ، عن طريق اختيار قيمة متغير آخر بشكل متكرر. بعد تعيين قيمة للمتغير ، يتم تطبيق بعض تقنيات التناسق على الرسم البياني للقيد. اعتمادًا على درجة تقنية الاتساق ، نحصل على خوارزميات مختلفة لرضا القيد.

حتى التراجع البسيط (BT) يؤدي نوعًا من تقنية الاتساق ويمكن اعتباره مزيجًا من التوليد الخالص والاختبار وجزء من تناسق القوس. تختبر خوارزمية BT اتساق القوس بين المتغيرات التي تم إنشاء مثيل لها بالفعل ، أي أن الخوارزمية تتحقق من صحة القيود مع الأخذ في الاعتبار إنشاء مثيل جزئي. نظرًا لأن مجالات المتغيرات التي تم إنشاء مثيل لها تحتوي على قيمة واحدة فقط ، فمن الممكن التحقق فقط من تلك القيود / الأقواس التي تحتوي على آخر متغير تم إنشاء مثيل له. إذا تم تقليل أي مجال ، فلن يكون القيد المقابل متسقًا وتتراجع الخوارزمية إلى إنشاء مثيل جديد.

يتم استدعاء الإجراء التالي AC3-BT في كل مرة يتم فيها تعيين قيمة جديدة لبعض المتغيرات Vcv (cv هو الرقم المتتالي للمتغير بترتيب متغيرات إنشاء مثيل)

Algorithm AC-3 for Backtracking

```
procedure AC3-BT(cv)
  Q <- {(Vi,Vcv) in arcs(G),i<cv};
  consistent <- true;
  while not Q empty & consistent
    select and delete any arc (Vk,Vm) from Q;
    consistent <- REVISE(Vk,Vm)
  endwhile
  return consistent
end AC3-BT
```

3- Forward Checking

solution of 4 queen's by Forward Checking code in python

Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables. We speak about restricted arc consistency because forward checking checks only the constraints between the current variable and the future variables. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

يعد الفحص الأمامي أسهل طريقة لمنع التعارضات المستقبلية. بدلاً من أداء تناسق القوس للمتغيرات التي تم إنشاؤها ، فإنه يؤدي شكلًا مقيدًا من تناسق القوس إلى المتغيرات التي لم يتم إنشاء مثيل لها بعد. نتحدث عن تناسق القوس المقيد لأن الفحص الأمامي يتحقق فقط من القيود بين المتغير الحالي والمتغيرات المستقبلية. عندما يتم تعيين قيمة للمتغير الحالي ، فإن أي قيمة في مجال متغير "مستقبلي" تتعارض مع هذا التعيين يتم إزالتها (مؤقتًا) من المجال. ميزة هذا هو أنه إذا أصبح مجال المتغير المستقبلي فارغًا ، فمن المعروف على الفور أن الحل الجزئي الحالي غير متسق. وبالتالي ، فإن الفحص إلى الأمام يسمح بفروع شجرة البحث التي ستؤدي إلى عدم تقليمها في وقت أبكر من التراجع البسيط. لاحظ أنه عندما يتم النظر في متغير جديد ، فإن جميع قيمه المتبقية مضمونة لتكون متسقة مع المتغيرات السابقة ، لذلك لم يعد التحقق من التخصيص مقابل التخصيصات السابقة ضروريًا.

Algorithm AC-3 for Forward Checking

```
procedure AC3-FC(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      consistent <- not Dk empty
    endif
  endwhile
  return consistent
end AC3-FC
```

An example problem

Queen's problem

In 4- queens problem, we have 4 queens to be placed on a 4*4 chessboard, satisfying the constraint that no two queens should be in the same row, same column, or in same diagonal.

The solution space according to the external constraints consists of 4 to the power 4, 4 tuples i.e., Si = {1, 2, 3, 4} and 1<= I <=4, whereas according to the internal constraints they consist of 4! Solutions i.e., permutation of 4.

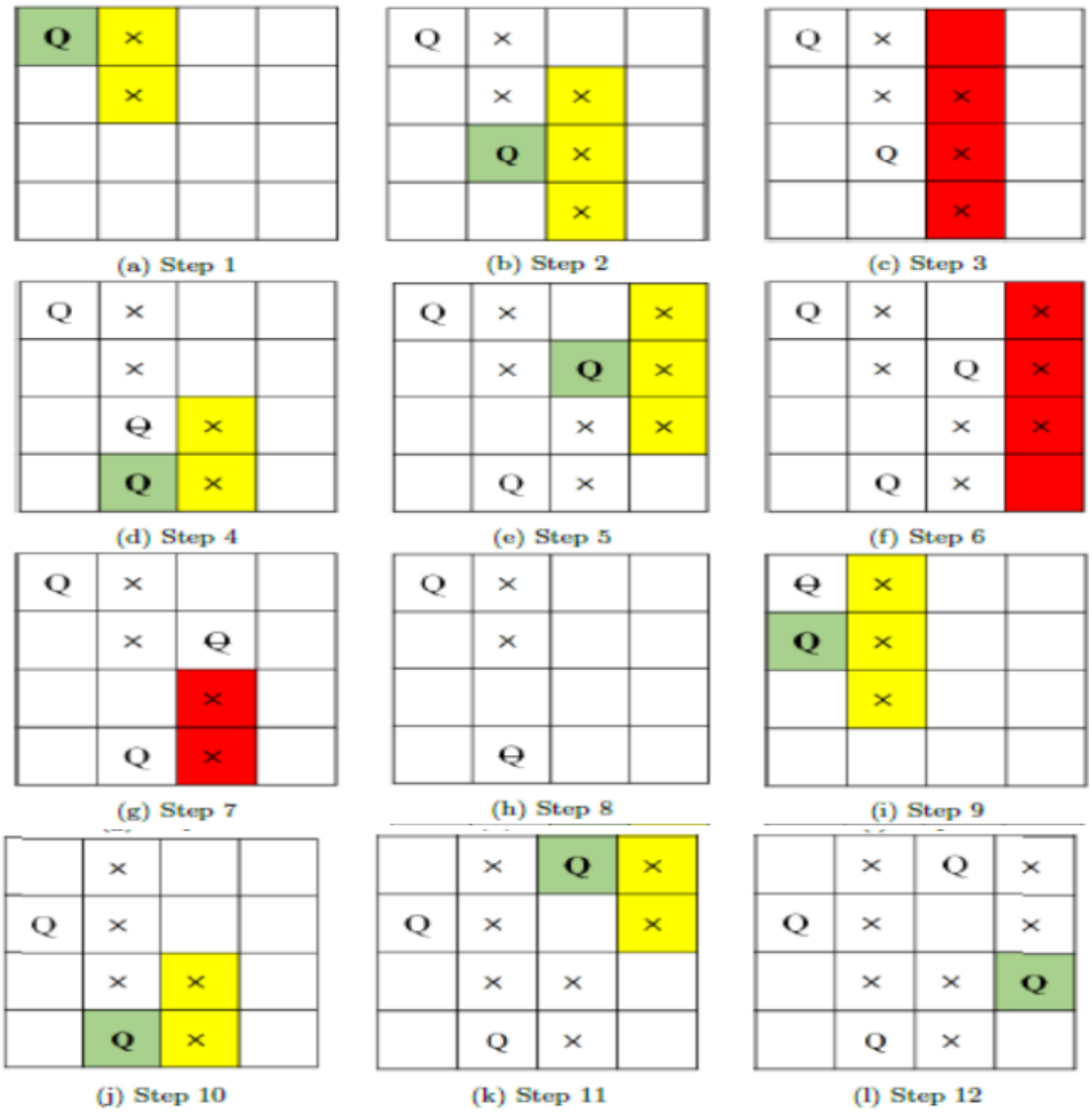
In this work, as stated, it has been possible to successfully implement essential CSP techniques based on N-Queens Problem. In order to find solution by legally placing Queens in a N × N board, the chess board has been buffered from row to row, and a Queen will be tried to place in a particular column's single cell at each iteration. To summarize, the goal is to achieve any solution while starting from buffering first row of the board till the last row, and for each buffering, column's cells will be checked to place a Queen by satisfying all the constraints given in N-Queens puzzle.

مشكلة الملكة

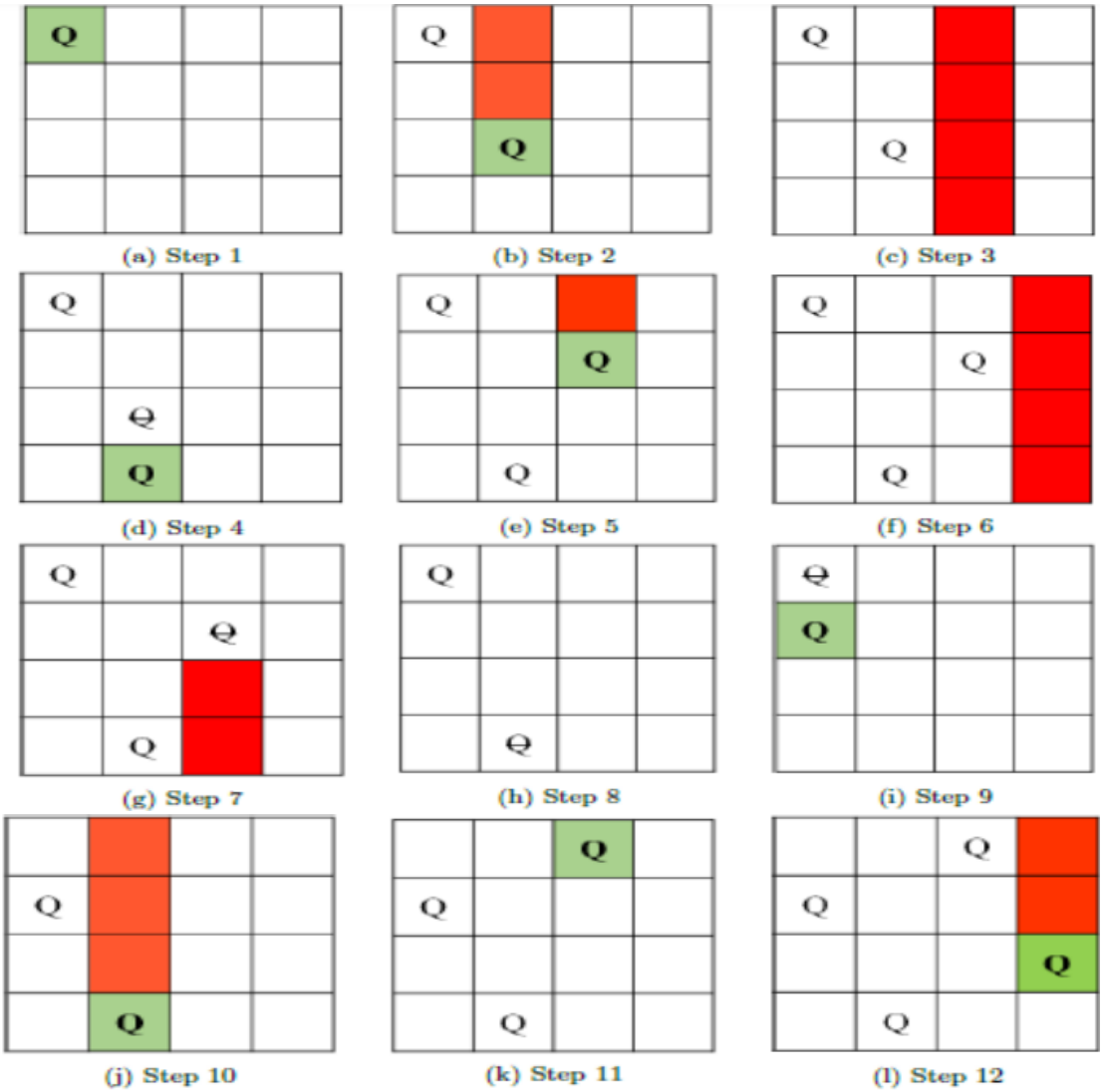
في مسألة ٤ - ملكات ، لدينا ٤ ملكات لوضعها على رقعة شطرنج ٤ * ٤ ، مما يرضي شرط عدم وجود ملكتين في نفس الصف ، أو نفس العمود ، أو في نفس القطر.

تتكون مساحة الحل وفقًا للقيود الخارجية من ٤ أس ٤ ، ٤ مجموعات ، أي Si = {١ ، ٢ ، ٣ ، ٤} و ١ <= I => 4 ، بينما وفقًا للقيود الداخلية ، تتكون من ٤! الحلول ، أي التقليب ٤.

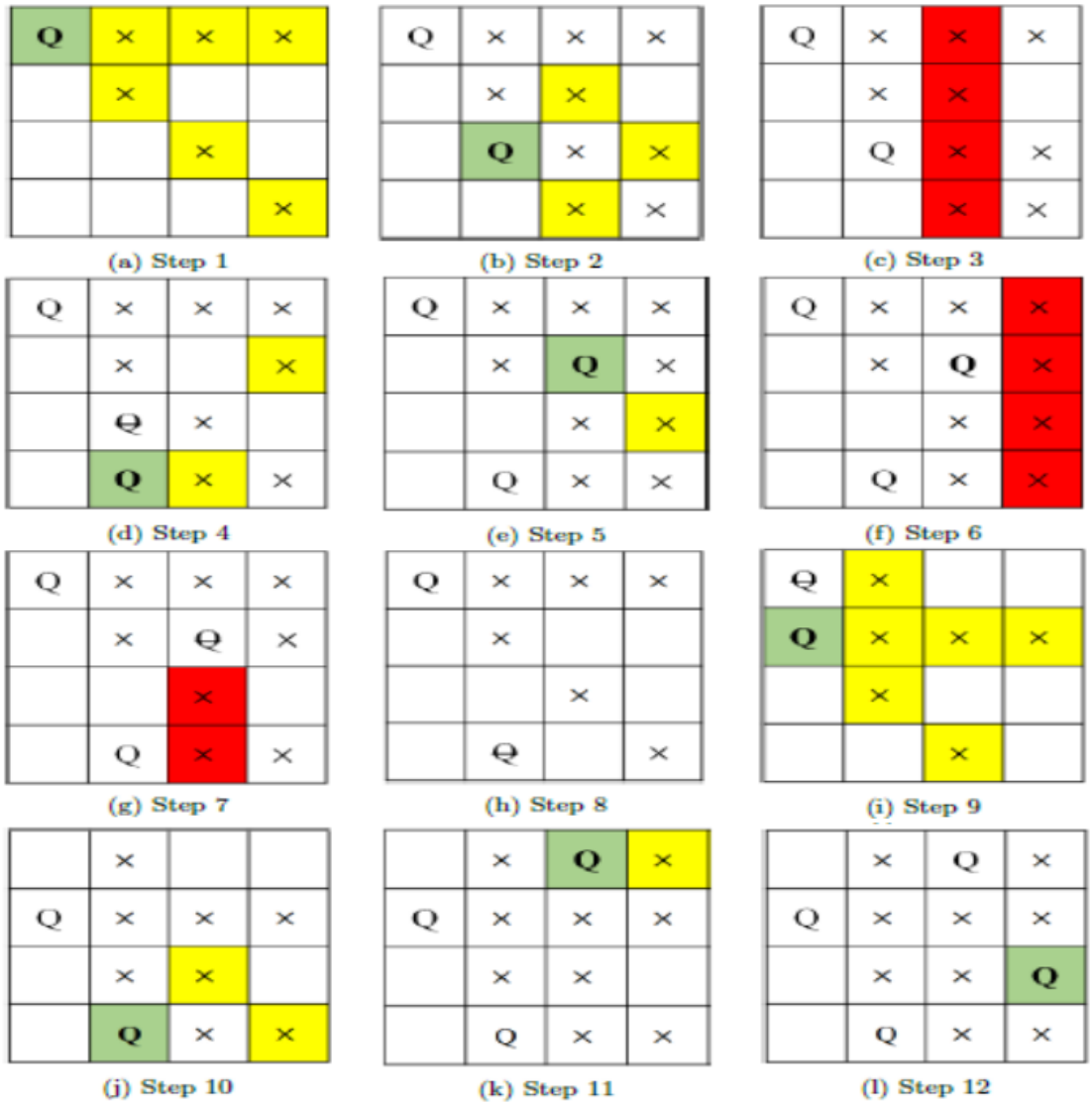
في هذا العمل ، كما هو مذكور ، كان من الممكن تنفيذ تقنيات CSP الأساسية بنجاح بناءً على مشكلة N-Queens. من أجل إيجاد حل عن طريق وضع الملكات بشكل قانوني في لوحة N × N ، تم تخزين رقعة الشطرنج مؤقتًا من صف إلى آخر ، وستتم محاولة وضع الملكة في خلية مفردة لعمود معين عند كل تكرار. للتلخيص ، الهدف هو تحقيق أي حل أثناء البدء من التخزين المؤقت للصف الأول من اللوحة حتى الصف الأخير ، ولكل تخزين مؤقت ، سيتم فحص خلايا العمود لوضع الملكة من خلال تلبية جميع القيود الواردة في لغز N-Queens.



Illustrative view to visualize AC-3 Algorithm for 4-Queens Problem



Illustrative view to visualize Backtrack Algorithm for 4-Queens Problem



Illustrative view to visualize Forward Checking Algorithm for 4-Queens Problem

Algorithm implementation:

```
1) Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column.  Do following
   for every tried row.
    a) If the queen can be placed safely in this row
       then mark this [row, column] as part of the
       solution and recursively check if placing
       queen here leads to a solution.
    b) If placing queen in [row, column] leads to a
       solution then return true.
    c) If placing queen doesn't lead to a solution
       then unmark this [row, column] (Backtrack)
       and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked,
   return false to trigger backtracking.
```

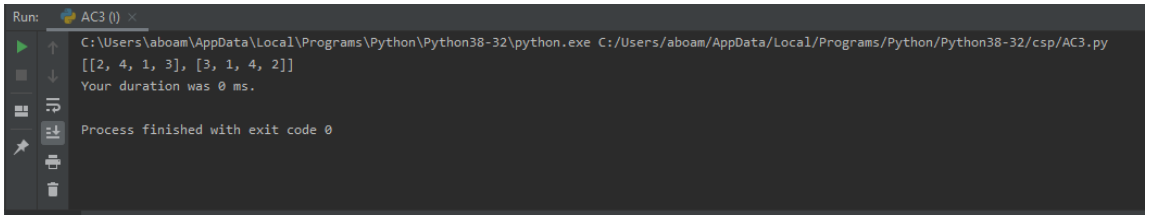
1- AC3

2-
```python
''' Python3 program to solve N Queen Problem using
backtracking '''

import time
result = []

# A utility function to print solution


''' A utility function to check if a queen can
be placed on board[row][col]. Note that this
function is called when "col" queens are
already placed in columns from 0 to col -1.
So we need to check only left side for
attacking queens '''


def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if (board[row][i]):
            return False

    # Check upper diagonal on left side
    i = row
    j = col
    while i >= 0 and j >= 0:
```

```
        if(board[i][j]):
            return False
        i -= 1
        j -= 1

    # Check lower diagonal on left side
    i = row
    j = col
    while j >= 0 and i < 4:
        if(board[i][j]):
            return False
        i = i + 1
        j = j - 1

    return True


''' A recursive utility function to solve N
Queen problem '''


def solveNQUtil(board, col):
    ''' base case: If all queens are placed
    then return true '''
    if (col == 4):
        v = []
        for i in board:
            for j in range(len(i)):
                if i[j] == 1:
                    v.append(j+1)
        result.append(v)
        return True

    ''' Consider this column and try placing
    this queen in all rows one by one '''
    res = False
    for i in range(4):

        ''' Check if queen can be placed on
        board[i][col] '''
        if (isSafe(board, i, col)):

            # Place this queen in board[i][col]
            board[i][col] = 1

            # Make result true if any placement
            # is possible
            res = solveNQUtil(board, col + 1) or res

            ''' If placing queen in board[i][col]
            doesn't lead to a solution, then
            remove queen from board[i][col] '''
            board[i][col] = 0 #

    ''' If queen can not be place in any row in
        this column col then return false '''
    return res
```

```
    def solveNQ(n):
        result.clear()
        board = [[0 for j in range(n)]
                 for i in range(n)]
        solveNQUtil(board, 0)
        result.sort()
        return result


    # Driver Code
    n = 4
    unix_time_ms_1 = int(time.time_ns() / 1000000)
    res = solveNQ(n)
    print(res)
    unix_time_ms_2 = int(time.time_ns() / 1000000)
    print(f"Your duration was {unix_time_ms_2-unix_time_ms_1} ms.")

    # This code is contributed by YatinGupta
```

Output:



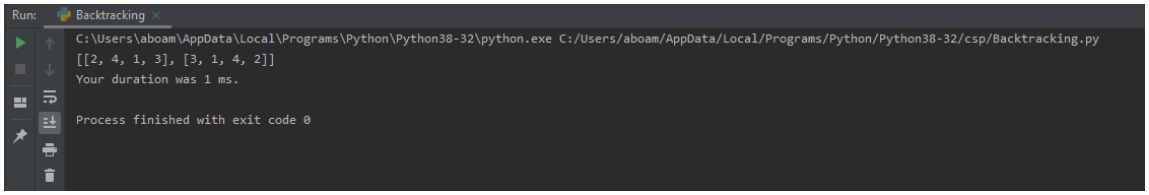
3- Backtracking:

```
4-  # Python program for above approach
    import math
    import time
    result = []

    # Program to solve N-Queens Problem


    def solveBoard(board, row, rowmask,
                   ldmask, rdmask):

        n = len(board)

        # All_rows_filled is a bit mask
        # having all N bits set
        all_rows_filled = (1 << n) - 1

        # If rowmask will have all bits set, means
        # queen has been placed successfully
        # in all rows and board is diplayed
        if (rowmask == all_rows_filled):
            v = []
            for i in board:
                for j in range(len(i)):
                    if i[j] == 'Q':
                        v.append(j+1)
            result.append(v)
```

```
    # We extract a bit mask(safe) by rowmask,
    # ldmask and rdmask. all set bits of 'safe'
    # indicates the safe column index for queen
    # placement of this iteration for row index(row).
    safe = all_rows_filled & (~(rowmask |
                        ldmask | rdmask))

    while (safe > 0):

        # Extracts the right-most set bit
        # (safe column index) where queen
        # can be placed for this row
        p = safe & (-safe)
        col = (int)(math.log(p)/math.log(2))
        board[row][col] = 'Q'

        # This is very important:
        # we need to update rowmask, ldmask and rdmask
        # for next row as safe index for queen placement
        # will be decided by these three bit masks.

        # We have all three rowmask, ldmask and
        # rdmask as 0 in beginning. Suppose, we are placing
        # queen at 1st column index at 0th row. rowmask, ldmask
        # and rdmask will change for next row as below:

        # rowmask's 1st bit will be set by OR operation
        # rowmask = 00000000000000000000000000000010

        # ldmask will change by setting 1st
        # bit by OR operation and left shifting
        # by 1 as it has to block the next column
        # of next row because that will fall on left diagonal.
        # ldmask = 00000000000000000000000000000100

        # rdmask will change by setting 1st bit
        # by OR operation and right shifting by 1
        # as it has to block the previous column
        # of next row because that will fall on right diagonal.
        # rdmask = 00000000000000000000000000000001

        # these bit masks will keep updated in each
        # iteration for next row
        solveBoard(board, row+1, rowmask | p,
              (ldmask | p) << 1, (rdmask | p) >> 1)

        # Reset right-most set bit to 0 so, next
        # iteration will continue by placing the queen
        # at another safe column index of this row
        safe = safe & (safe-1)

        # Backtracking, replace 'Q' by ' '
        board[row][col] = ' '

# Program to print board


def printBoard(board):
    for row in board:
        print("|" + "|".join(row) + "|")
```

```
# Driver Code


def main():

    n = 4 # board size
    board = []

    for i in range(n):
        row = []
        for j in range(n):
            row.append(' ')
        board.append(row)

    rowmask = 0
    ldmask = 0
    rdmask = 0
    row = 0

    # Function Call
    unix_time_ms_1 = int(time.time_ns() / 1000000)
    result.clear()
    solveBoard(board, row, rowmask, ldmask, rdmask)
    result.sort()
    print(result)
    unix_time_ms_2 = int(time.time_ns() / 1000000)
    print(f"Your duration was {unix_time_ms_2 - unix_time_ms_1} ms.")


if __name__ == "__main__":
    main()

# This code is contributed by Nikhil Vinay
```

Output:



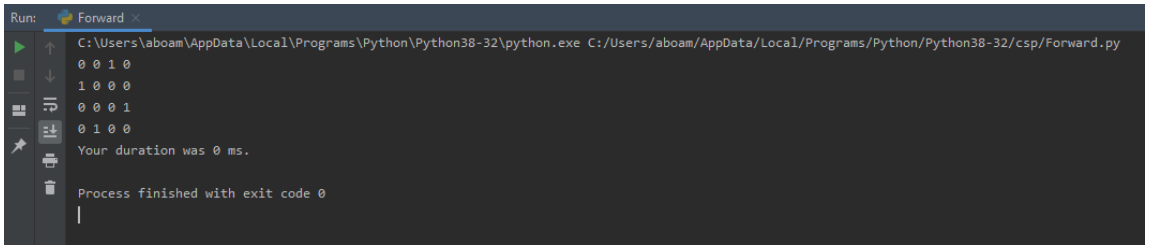
3- Forward:

```
# Python3 program to solve N Queen
# Problem using backtracking
import time
global N
N = 4


def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()
```

```
# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True


def solveNQUtil(board, col):
    # base case: If all queens are placed
    # then return true
    if col >= N:
        return True

    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):

        if isSafe(board, i, col):

            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True

            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this colum col then return false
    return False


# This function solves the N Queen problem using
# Forward Checking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
```

```
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]]

    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False

    printSolution(board)
    return True


# Driver Code
unix_time_ms_1 = int(time.time_ns() / 1000000)
solveNQ()
unix_time_ms_2 = int(time.time_ns() / 1000000)
print(f"Your duration was {unix_time_ms_2-unix_time_ms_1} ms.")

# This code is contributed by Divyanshu Mehta
```

Output:



Time complexity and run time:

	AC 3 algorithm	backtracking algorithm	Forward Checking algorithm
time complexity	$O(ed^3)$	1. Hamiltonian cycle : O(N!) in the worst case 2. WordBreak and StringSegment : O(2^N) 3. NQueens : O(N!)	$O(n^2d^2)$
Tim n=4	0 ms	1 ms	1 ms
Tim n=8	1 ms	2 ms	1 ms
Tim n=20	1 ms	2 ms	2 ms

References

[1] “Time complexity of a backtrack algorithm,” Computer Science Stack Exchange. https://cs.stackexchange.com/questions/13181/time-complexity-of-a-backtrack-algorithm (accessed Apr. 14, 2021).

[2] “How to measure time taken between lines of code in python?,” Stack Overflow. https://stackoverflow.com/questions/14452145/how-to-measure-time-taken-between-lines-of-code-in-python (accessed Apr. 14, 2021).

[3] A. Mejia, “8 time complexities that every programmer should know,” Adrian Mejia Blog. https://adrianmejia.com/most-popular-algorithms-time-complexity-every-programmer-should-know-free-online-tutorial-course/ (accessed Apr. 14, 2021).

[4] CSBreakdown, Graph Colouring Problem - Backtracking. 2015.

[5] Z. Liu, “Algorithms for Constraint Satisfaction Problems (CSPs),” p. 13

[6] Printing all solutions in N-Queen Problem, https://www.geeksforgeeks.org/printing-solutions-n-queen-problem/, 2021

[7] N Queen Problem | Backtracking-3, https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/ , 2021

[8] Exhaustive Study of Essential Constraint Satisfaction Problem Techniques based on N-Queens Problem, Md. Ahsan Ayub∗ , Kazi A Kalpoma† ∗Department of Computer Science ∗American International University-Bangladesh †Department of Computer Science and Engineering †Ahsanullah University of Science and Technology Dhaka, Bangladesh,

[9] Constraint satisfaction problem , https://en.wikipedia.org/wiki/Constraint_satisfaction_problem, 2021

[10] Constraint Propagation, https://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html, 2021