Constraint Satisfaction Problem Part 2

Thursday, 23 September 2021 3:01 PM

Today's Menu

What will you learn today?

We still do not understand the meaning of lyrics of "Bohemian Rhapsody".

Inference is hard……..but, not so hard for CSP

Heuristics for variable picking and ordering domain values

Why does it matter?

Constraint-based reasoning is widespread (and scalability of it is considered the major challenge for today's AI systems)

Logistics

Eager vs non-eager
Eager: GoalTest before adding to the frontier

BFS was taught with eager goal-check for us to realize that eager check is not a good idea if we want to design an optimal algorithm.

Often algorithms are presented as things to remember but algorithms are designed by engineers/researchers and they don't come out of thin year; so it is important to understand why concepts are in a way than to memorize what concepts are.

The asymptotic time and space complexity of BFS/DFS does not change with eager vs non-eager.

Duplicate vs non-duplicate

Similarly, duplicate vs non-duplicate does not change asymptotic time and space complexity of BFS/DFS. (Hashing-based implementations for Queue/Stack)

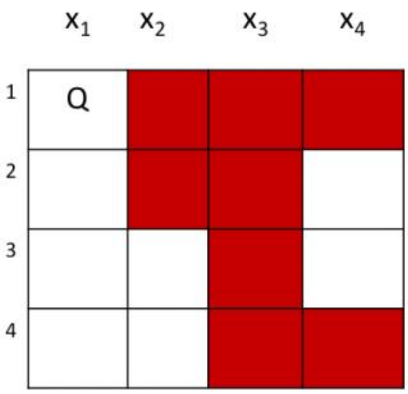
Backtracking Algorithm with Inference

BacktrackingSearch_with_Inference($prob, assign$)

```
if AllVariablesAssigned(prob, assign) then return assign
var ← PickUnassignedVar(prob, assign)
for value in OrderDomainValue(var, prob, assign) do
    if ValIsConsistentWithAssignment(value, assign) then
        assign ← assign ∪ (var = value)
        inference ← Infer(prob, var, assign)
        assign ← assign ∪ inference
        if inference != failure then
            result ← BacktrackingSearch.(prob, assign)
            if result != failure then return result
        assign ← assign \ {(var = value) ∪ inference}
return failure
```

4

Backtracking Algorithm with Inference



$x_1 = 1$

NoAttack(x_1, x_2):

$x_2 \notin \{1,2\}$

NoAttack(x_1, x_3):

$x_3 \notin \{1,3\}$

NoAttack(x_1, x_4):

$x_4 \notin \{1,4\}$

NoAttack(x_2, x_4):

....

NoAttack(x_2, x_3):

$x_2 \notin \{3\}$, $x_3 \notin \{3,4\}$

NoAttack(x_3, x_4):

$x_3 \notin \{2\}$

5

How to Implement INFER

Data structure for Inference:

(Unordered) list of tuples of the form: ($x \notin S$)

variable = value

Data structure for assign:

(Unordered) list of tuples, where every tuples is of the form (x=v) or ($x \notin S$)

↳ variable not part of S

ComputeDomain(x,assign,inference): e.g. $\{x_2 \notin \{2,3\}, x_3 \notin \{1,2,3,4\}\}$

e.g. $\{x_1 = 1\}$

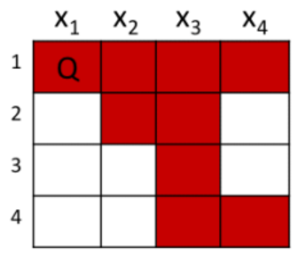
Returns S such that the effective domain of x is S

the spaces that x can take given the current assignment & inference

$CD(x_1, \ldots, \ldots) = \{1\}$

$CD(x_2, \ldots, \ldots) = \{1, 4\}$

6



x_1

NoAttack(x_1,x_2):

$x_2 \notin \{1,2\}$

NoAttack(x_1,x_3):

$x_3 \notin \{1,3\}$

NoAttack(x_1,x_4):

$x_4 \notin \{1,4\}$

x_2

NoAttack(x_2,x_4):

....

NoAttack(x_2,x_3):

$x_3 \notin \{3\}$

x_4

NoAttack(x_3,x_4):

$x_3 \notin \{2\}$

Ideas

1. Iterate over variables: Which variables?

   For which we infer *new information*

2. Iterate over constraints where the variable appears

3. We can infer $x \neq v$ from constraint C(x,y,z) if there is no value that y and z can take such that C(v,y,z) returns True → from the effective domain

   Returns True = Not falsified = Not violated

   Returns False = falsified = Violated

7

INFER

x_1

INFER(*prob*, *var*, *assign*)

1: $inference \leftarrow \emptyset$
2: $varQueue \leftarrow [var]$ x_1
3: **while** *varQueue* is not empty **do**
4: $y \leftarrow varQueue.\text{pop}()$ x_1
5: **for** each constraint C in *prob* where $y \in Vars(C)$ **do** ← all the constraints that y appears in
6: **for** all $x \in Vars(C) \setminus y$ **do** $\{x_2, x_3, x_4\}$
7: $S \leftarrow$ COMPUTEDOMAIN(x, *assign*, *inference*) $\{1, 2, 3, 4\}$
8: **for** each value v in S **do**
9: **if** no valid value exists for all var $\in Var(C) \setminus x$ s.t. $C[x \vdash v]$ is satisfied **then**
10: $inference \leftarrow inference \cup (x \notin \{v\})$ $x_2 \notin \{1, 2\}$
11: $T \leftarrow$ COMPUTEDOMAIN(x, *assign*, *inference*)
12: **if** $T = \emptyset$ **then return** *failure* ↳ returns failure no matter what the other vars are
13: **if** $S \neq T$ **then**
14: $varQueue.\text{add}(x)$ } new information ↳ domain smaller ↳ recompute
15: **return** *inference*

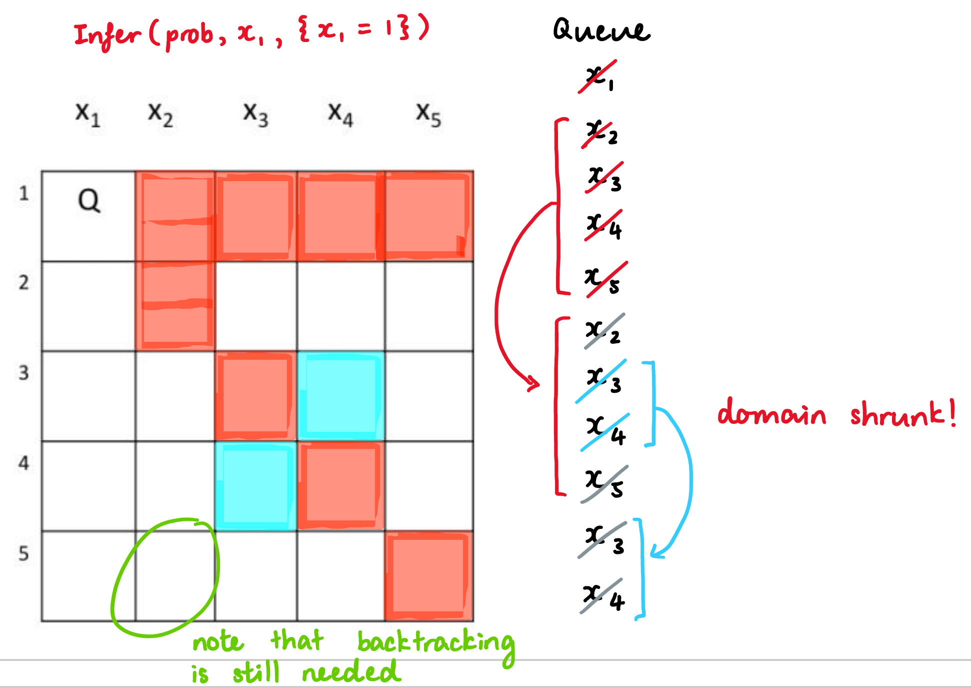
The above algorithm is conceptually identical to AC-3 algorithm

The order of constraints in line 5 only impacts runtimeefficiency but not the final answer

8

INFER

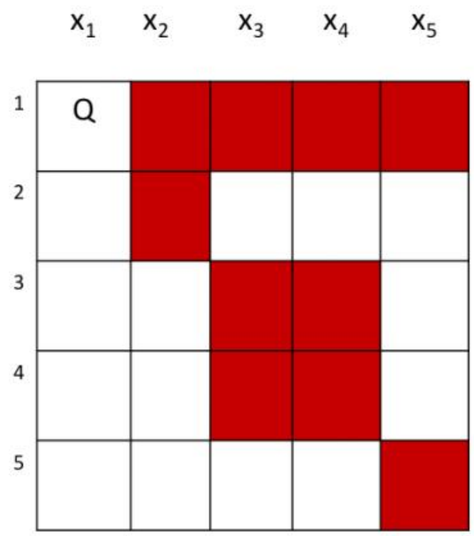
Infer(prob, x_1, $\{x_1 = 1\}$)



9

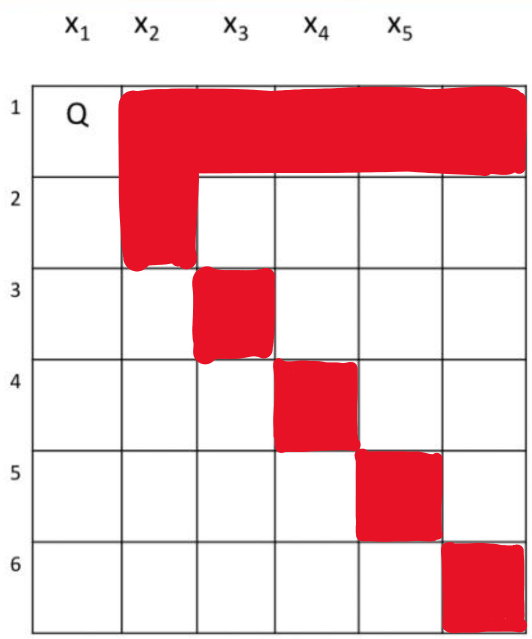
INFER

INFER



10

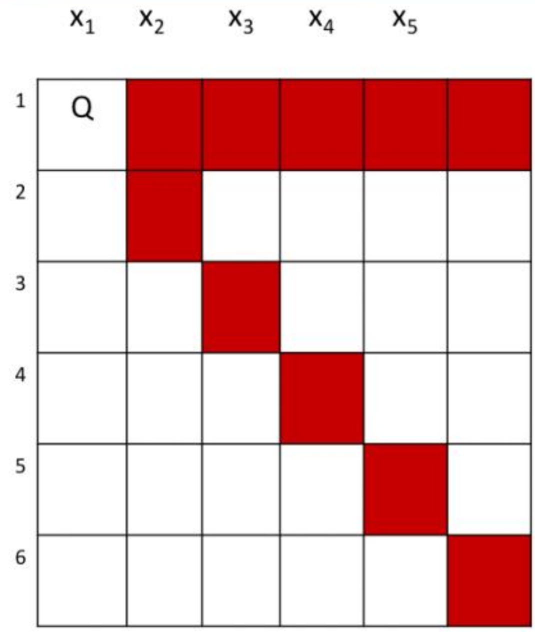
INFER



☆ nothing else to infer
↳ trade off

11

INFER



INFER

```
INFER(prob, var, assign)
inference ← ∅
varQueue ← [var]
while varQueue is not empty do
    y ← varQueue.pop()
    for each constraint C in prob where y ∈ Vars(C) do
        for all x ∈ Vars(C) \ y do
            S ← COMPUTEDOMAIN(x, assign, inference)
            for each value v in S do
                if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ⊢ v] is satisfied then
                    inference ← inference ∪ (x ∉ {v})
            T ← COMPUTEDOMAIN(x, assign, inference)
            if T = ∅ then return failure
            if S ≠ T then
                varQueue.add(x)
return inference
```

INFER function is expensive.

Limit the depth of the inference in order to reduce the computational cost

Forward Checking

INFER(*prob*, *var*, *assign*)

only infer 1 var

```
1:  inference ← ∅
2:  varQueue ← [var]
3:  while varQueue is not empty do
4:      y ← varQueue.pop()
5:      for each constraint C in prob where y ∈ Vars(C) do
6:          for all x ∈ Vars(C) \ y do
7:              S ← ComputeDomain(x, assign, inference)
8:              for each value v in S do
9:                  if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ⊢ v] is satisfied then
10:                     inference ← inference ∪ (x ∉ {v})
11:             T ← ComputeDomain(x, assign, inference)
12:             if T = ∅ then return failure
13:             if S ≠ T then
14:                 varQueue.add(x)
15: return inference
```

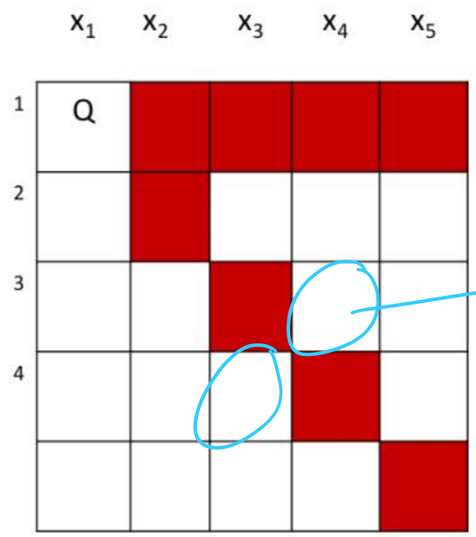
remove

Forward Checking: Don't add variable to varQueue at each iteration.
Remove lines 13 and 14.

Forward Checking

	X_1	X_2	X_3	X_4	X_5
1	Q				
2					
3					
4					

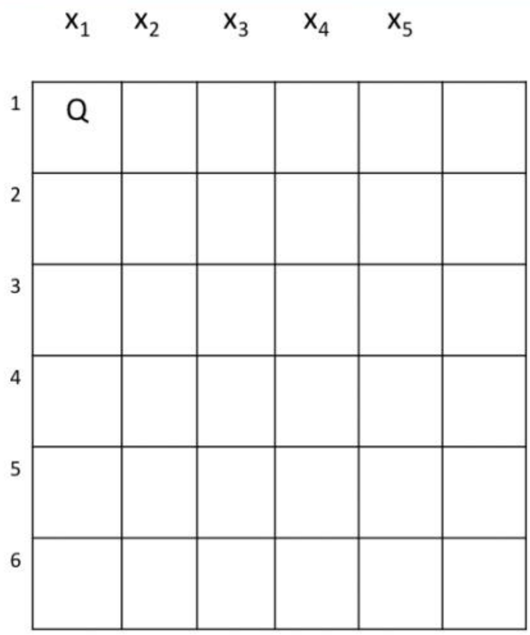
Forward Checking



lose out
↳ save time infering
↳ more time backtracking

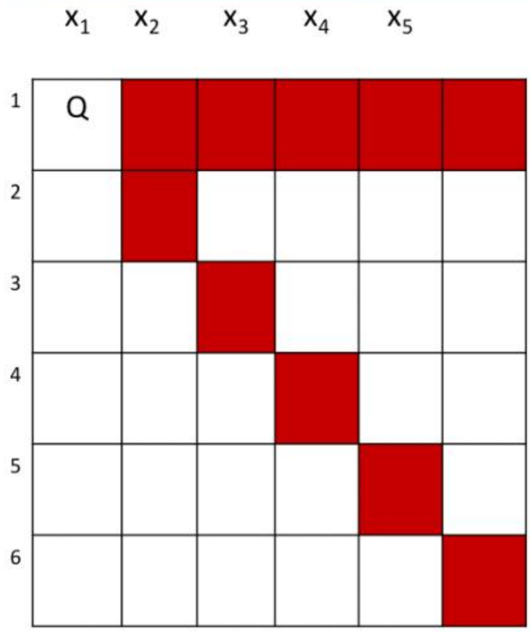
16

Forward Checking



17

Forward Checking



☆ the same!

INFER: Different Heuristics

```
INFER(prob, var, assign)
inference ← ∅
varQueue ← [var]
while varQueue is not empty do
    y ← varQueue.pop()
    for each constraint C in prob where y ∈ Vars(C) do
        for all x ∈ Vars(C) \ y do
            S ← COMPUTEDOMAIN(x, assign, inference)
            for each value v in S do
                if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ⊢ v] is satisfied then
                    inference ← inference ∪ (x ∉ {v})
            T ← COMPUTEDOMAIN(x, assign, inference)
            if T = ∅ then return failure
            if |T| = 1 then   → and |S| ≠ |T|
                varQueue.add(x)
return inference
```

more constraints for other vars

- Find inference for variables that have only ONE valid value in the domain, i.e., |T| = 1
- Can be set to any value, depending on the complexity of the application:
  |T| < 2 , |T| <= 3 ….

Backtracking Algorithm with Inference

BacktrackingSearch_with_Inference($prob, assign$)

```
if AllVariablesAssigned(prob, assign) then return assign
var ← PickUnassignedVar(prob, assign) ⟶ what order to pick
for value in OrderDomainValue(var, prob, assign) do ⟶ what order to try
    if ValIsConsistentWithAssignment(value, assign) then
        assign ← assign ∪ (var = value)
        inference ← Infer(prob, var, assign)
        assign ← assign ∪ inference
        if inference != failure then
            result ← BacktrackingSearch.(prob, assign)
            if result != failure then return result
        assign ← assign \ {(var = value) ∪ inference}
return failure
```

20

Backtracking Algorithm with Inference: Different Heuristics

- PickUnassignedVar
  - Minimum Remaining Value Heuristic: Choose the next unassigned variable with the smallest effective domain size
    - Can allow us to backtrack quickly

      ↳ exhaust the possible values more easily
- OrderDomainValue
  - Least Constraining Value Heuristic: Pick a value for a variable that rules out the least domain values for other variables.
    - Can allow us to find a solution faster

      ↳ more choices

21

How hard is Constraint Satisfaction Problem (CSP)?

CSP is Non-deterministic Polynomial time Complete (NP-complete)

Special Variants:

1. Binary CSP: Every constraint is defined over two variables.
   NP-complete

2. Boolean CSP (SAT): Domain of every variable is {0,1}
   NP-complete

3. 2-SAT: Binary CSP and Boolean CSP.
   PTIME (polynomial time solvable)

22

So many choices… What should I choose?

Short Answer: We do not know.

Long Answer: There is a CSP and SAT competition every year

- Tools from MeelGroup placed second
- You can use whatever is the best solver based on the last competition

But more importantly,

- There is still so much to discover when it comes to:
  - How to pick an unassigned variable?
  - How to pick an unassigned value?
  - How much to infer?
  - How to model problems effectively as CSP?

23