Solving Constraint Satisfaction Problems

Vincent A. Cicirello, Ph.D.

Professor of Computer Science

cicirelv@stockton.edu https://www.cicirello.org/

1

Lesson 1

WHAT IS A CONSTRAINT SATISFACTION PROBLEM?

2

A few example Constraint Satisfaction Problems

Minesweeper

Sudoku

Job-shop scheduling

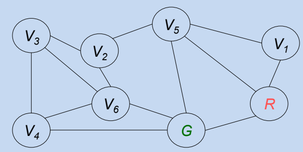
3

Another example Constraint Satisfaction Problem

Inside each node marked V_1 .. V_6 we must assign: *R*, *G* or *B*.

No two connected nodes may be assigned the same symbol.

Notice that two nodes have already been given an assignment.



Graph Coloring

4

Formal Constraint Satisfaction Problem

- A CSP is a triplet { *V* , *D* , *C* }.
- A CSP has a finite set of variables $V = \{ V_1 , V_2 .. V_N \}$.
- A set of domains, $D = \{ D_1 , D_2 .. D_N \}$, one for each variable. Each variable V_i may be assigned a value from domain D_i of values.
- *C* is a set of constraints, where each member of *C* is a pair. The first member of each pair is a tuple of variables. The second element is a set of legal values which that tuple may take.
- Example with graph coloring next slide….

5

An instance of Graph Coloring using the formal definition

$V = \{ V_1 , V_2 , V_3 , V_4 , V_5 , V_6 \}$

$D_i = \{ R , G , B \}$, for i=1..6

$C = \{ (V_1,V_5) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

$(V_5,V_2) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

$(V_2,V_6) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

$(V_2,V_3) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

$(V_3,V_6) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

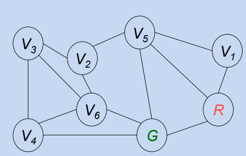
$(V_3,V_4) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

$(V_4,V_6) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

$(V_1) : \{ (\mathrm{G}), (\mathrm{B}) \},\ (V_5) : \{ (\mathrm{B}) \},\ (V_6) : \{ (\mathrm{R}), (\mathrm{B}) \},$

$(V_4) : \{ (\mathrm{R}), (\mathrm{B}) \} \}$

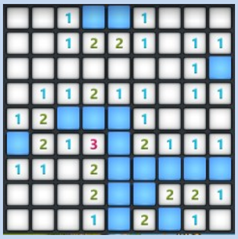
Obvious point: Usually *C* isn't represented explicitly, but by a function.



Graph Coloring

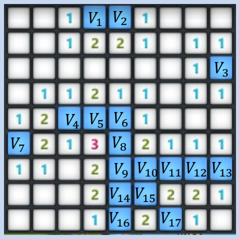
6

Minesweeper as a CSP



Each unknown cell is a variable

The domain of each variable is $D_i = \{B, E\}$, for Bomb or Empty.

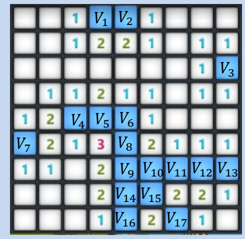


7

Minesweeper as a CSP

- Each cell with a number is one of the constraints in the constraint set C.
- Here is part of the constraint set C:

$$C = \{\, (V_{14}, V_{16}) : \{(B,E),(E,B)\},\\ (V_9, V_{14}, V_{16}) : \{(B,B,E),(B,E,B),(E,B,B)\},\\ (V_8, V_9, V_{14}) : \{(B,B,E),(B,E,B),(E,B,B)\},\\ (V_4, V_5, V_6, V_8, V_9) : \{(B,B,B,E,E),(B,B,E,B,E),\\ (B,E,B,B,E),(E,B,B,B,E),(B,B,E,E,B),\\ (B,E,B,E,B),(E,B,B,E,B),(B,E,E,B,B),\\ (E,B,E,B,B),(E,E,B,B,B)\},\\ \ldots.\}$$



8

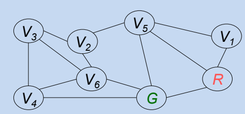
Lesson 2

DFS FOR CSP SOLVING

9

CSP as a Search Problem

- **Search State:** a search state has variables 1 … *k* assigned values from their domains. Variables *k+1* … *n*, not yet unassigned.
- **Start state:** All variables unassigned.
- **Goal states:** States where all variables are assigned values, and all constraints are satisfied.
- **Successors:** Successors of a state with $V_1 \ldots V_k$ assigned values and rest unassigned are all states (with $V_1 \ldots V_k$ unchanged) and with V_{k+1} assigned a value from its domain.
- **Cost on transitions:** 1 is fine. (We don't really care.)
  - –Note: the length of every path from the start state to a goal state is exactly equal to n.
  - –Nothing deeper than n steps.



10

CSP as a Search Problem

START =(V_1=? V_2=? V_3=? V_4=? V_5=? V_6=?)

succs(START) = {

(V_1=R V_2=? V_3=? V_4=? V_5=? V_6=?),

(V_1=G V_2=? V_3=? V_4=? V_5=? V_6=?),

(V_1=B V_2=? V_3=? V_4=? V_5=? V_6=?) }

succs((V_1=G V_2=? V_3=? V_4=? V_5=? V_6=?)) = {

(V_1=G V_2=R V_3=? V_4=? V_5=? V_6=?),

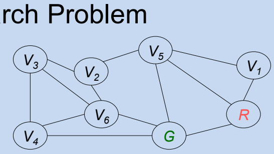
(V_1=G V_2=G V_3=? V_4=? V_5=? V_6=?),

(V_1=G V_2=B V_3=? V_4=? V_5=? V_6=?) }

And so forth….

What search algorithms could we use?

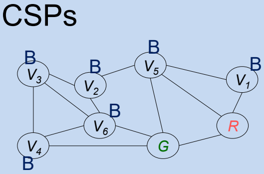
It turns out BFS is not a popular choice. Why not?



11

DFS for CSPs

- What about DFS?
- Much more popular. At least it has a chance of finding an easy answer quickly.
- What happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
- This makes DFS look very, very stupid!
- Play animation video: 9d.



We'll finally notice conflicts, and try G for V6, discover a conflict, try R for V6, discover conflicts, backup to V5 trying G there, then B at V6, discover conflicts, etc.…

12

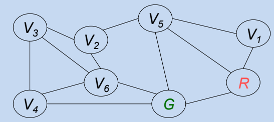
Lesson 3

CONSISTENCY CHECKING WITH BACKTRACKING

13

Consistency Checking with Backtracking

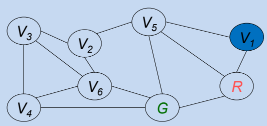
- Don't ever try successor which causes inconsistency with its neighbors.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
  - What's the computational overhead for this?
  - Backtracking still looks a little stupid!
- We'll step through this example.



14

Consistency Checking with Backtracking

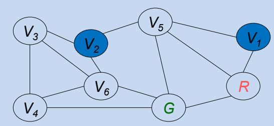
- Don't ever try successor which causes inconsistency with its neighbors.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
  - What's the computational overhead for this?
  - Backtracking still looks a little stupid!
- We'll step through this example.



15

Consistency Checking with Backtracking

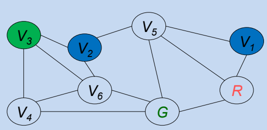
- Don't ever try successor which causes inconsistency with its neighbors.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
  - What's the computational overhead for this?
  - Backtracking still looks a little stupid!
- We'll step through this example.



16

Consistency Checking with Backtracking

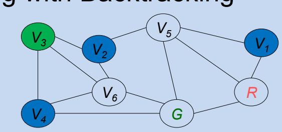
- Don't ever try successor which causes inconsistency with its neighbors.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
  - What's the computational overhead for this?
  - Backtracking still looks a little stupid!
- We'll step through this example.



17

Consistency Checking with Backtracking

- Don't ever try successor which causes inconsistency with its neighbors.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
  - What's the computational overhead for this?
  - Backtracking still looks a little stupid!
- We'll step through this example.



Here's where it starts to look a bit dumb.
- It will now discover nothing works for V5.
- It will backtrack and try red for V4.
- It discovers again nothing works for V5.
- It will backtrack, but no more options for V4.
- It will backtrack and change V3 to Red, and then V4 Blue, and then get stuck at V5 again
- Etc, etc…..

18

Consistency Checking with Backtracking

- More examples: play animation videos 9b, 27b

19

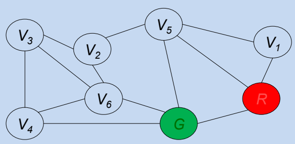
Lesson 4

FORWARD CHECKING

20

Forward Checking

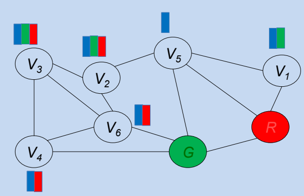
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



21

Forward Checking

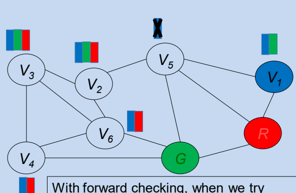
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



22

Forward Checking

- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?

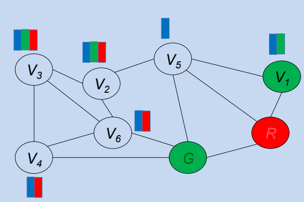


With forward checking, when we try assigning blue to V1, we remove blue from V5's availability list, emptying it, and immediately backtrack.

23

Forward Checking

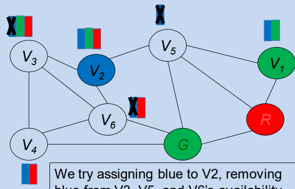
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



24

Forward Checking

- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?

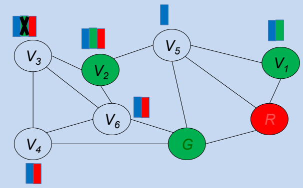


We try assigning blue to V2, removing blue from V3, V5, and V6's availability sets, emptying V5's, and immediately backtrack.

25

Forward Checking

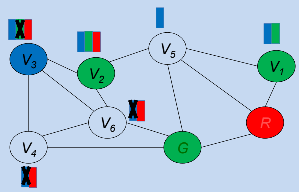
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



26

Forward Checking

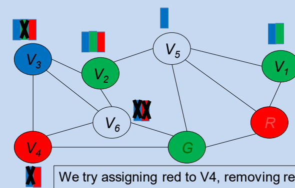
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



27

Forward Checking

- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?

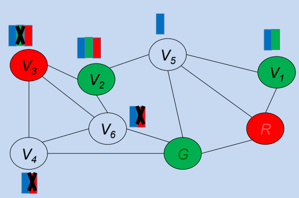


We try assigning red to V4, removing red from V6's availability set, emptying it, and immediately backtrack. Must backtrack to V3 because V4 has no options left.

28

Forward Checking

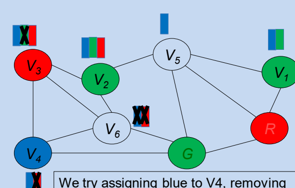
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



29

Forward Checking

- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?

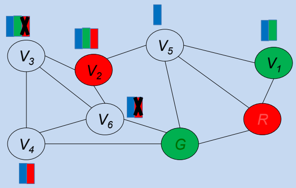


We try assigning blue to V4, removing blue from V6's availability set, emptying it, and immediately backtrack. Backtrack to V2 because V4, V3 have no options left.

30

Forward Checking

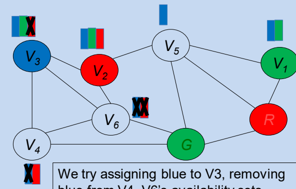
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



31

Forward Checking

- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?

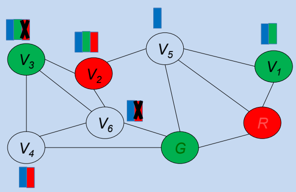


We try assigning blue to V3, removing blue from V4, V6's availability sets, emptying V6's, and immediately backtrack.

32

Forward Checking

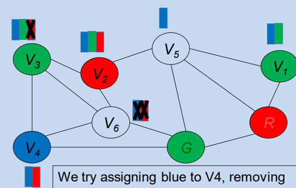
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



33

Forward Checking

- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?

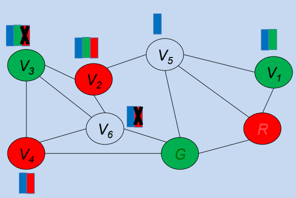


We try assigning blue to V4, removing blue from V6's availability set, emptying it, and immediately backtrack.

34

Forward Checking

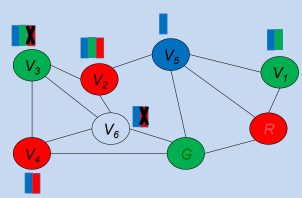
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



35

Forward Checking

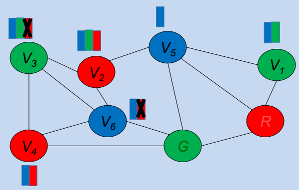
- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



36

Forward Checking

- At start, for each variable, record the current set of possible legal values for it (called an *availability set*).
- When you assign a value in the DFS search, update *availability set* for all variables. Backtrack immediately if you empty a variable's *availability set*.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?



37

Forward Checking

- More examples: play animation videos 27f, 49f

38

Lesson 5

CONSTRAINT PROPAGATION

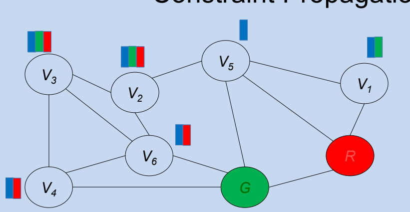
39

Constraint Propagation

- Forward checking computes the availability set of each variable independently at the start, and then only updates the availability sets when assignments are made in the DFS that are directly relevant to the current variable.
- Constraint Propagation carries this further. When you delete a value from your availability set, check all variables connected to you. If any of them change, delete all inconsistent values connected to them, etc…

40

Constraint Propagation

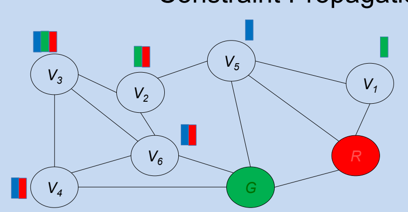


Constraint Propagation starts of like forward checking, but if any available sets are reduced from their initial domains, it attempts to carry this further.

In this case, V5 is left with only Blue, so blue will also be removed from V1 and V2.

41

Constraint Propagation

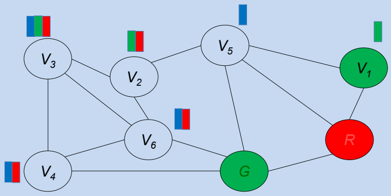


V1 only has green left. If it was connected to any variables with green in their available sets, then constraint propagation would continue to try to reduce those availability sets, etc.

It is not, so DFS begins.

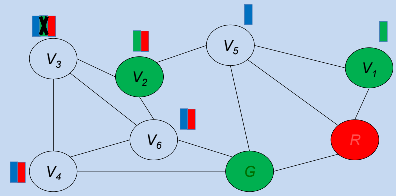
42

Constraint Propagation



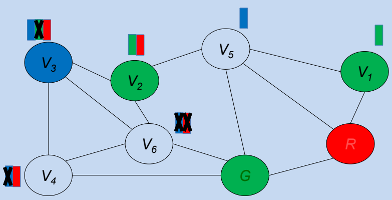
43

Constraint Propagation



44

Constraint Propagation

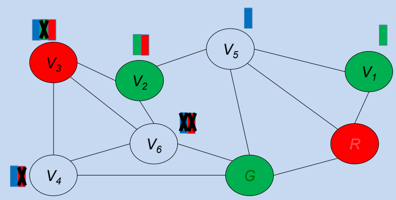


When we try assigning Blue to V3, constraint propagation removes blue from the available sets of V4 and V6, and then carries it further noticing that V4 only has red left, so removes red from V6, emptying it.

The empty V6 available set causes an immediate backtrack.

45

Constraint Propagation

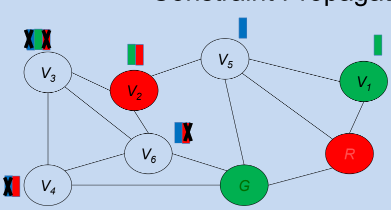


It then tries red for V3, but constraint propagation removes red from the available sets of V4 and V6, and then carries it further noticing that V4 only has blue left, so removes blue from V6, emptying it.

The empty V6 available set causes an immediate backtrack (this time to V2 since no more options at V3).

46

Constraint Propagation



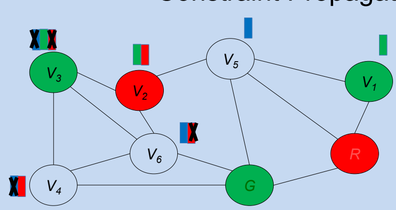
When we backtrack to change V2 to red begins by removing red from V3 and V6.

V6 only has 1 value left so constraint propagation continues removing blue from V3 and V4.

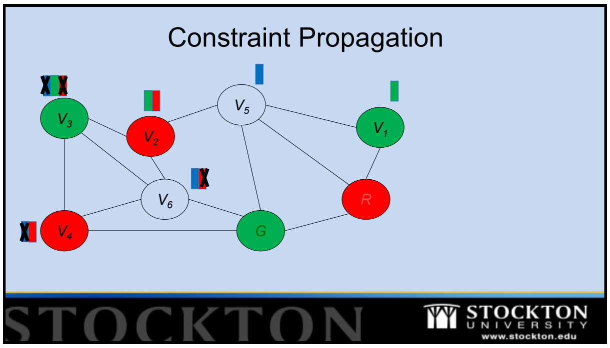
V3 and V4 only have 1 value left so constraint propagation checks if that can be propagated further.

47

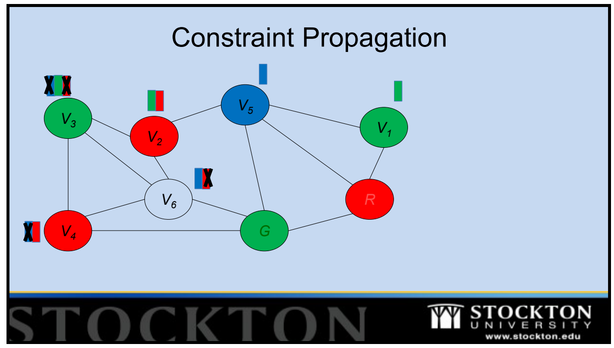
Constraint Propagation



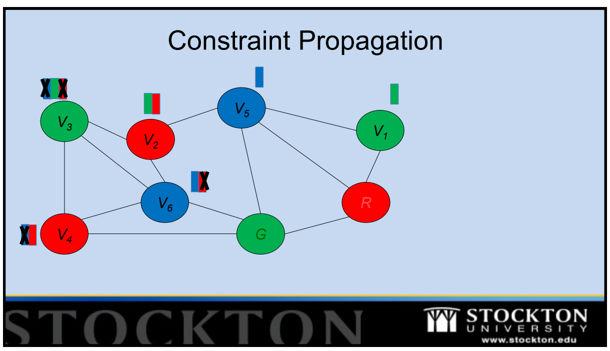
48



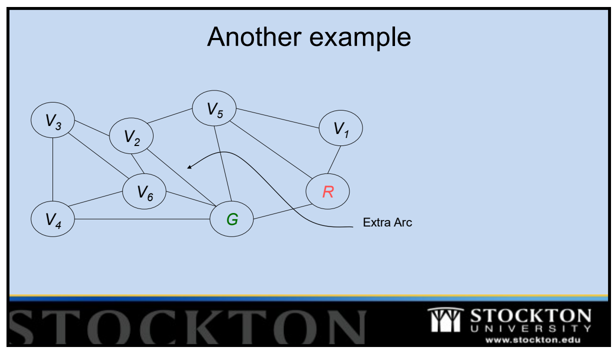
49



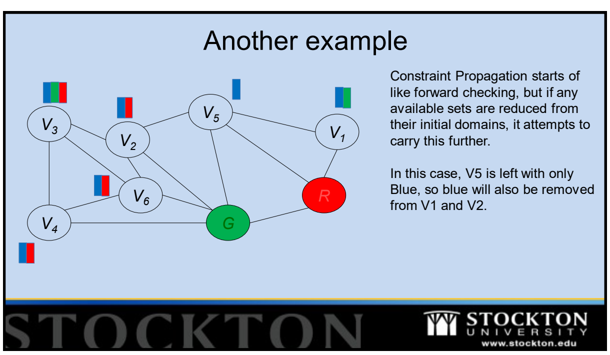
50



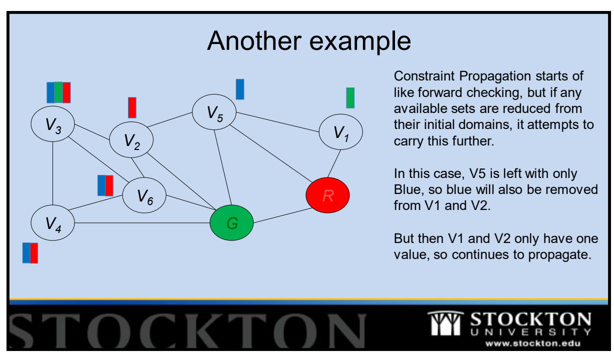
51



52

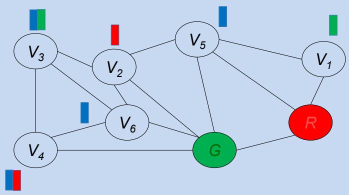


53



54

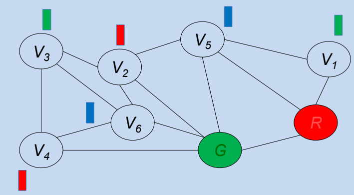
Another example



But then V6 only has one value, so continues to propagate.

55

Another example



But then V3 and V4 only have one value each, so will check if it can propagate further.

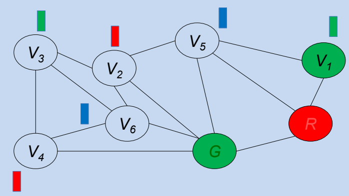
It cannot, so DFS can now begin.

Note that all of this was done before we ever started the DFS search.

We've eliminated many options, which has the effect of reducing the branching factor.

56

Another example



We can now run the DFS.

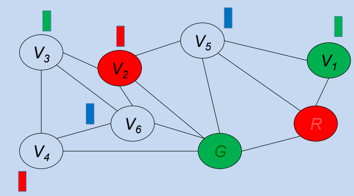
After each assignment we can run constraint propagation to try to further reduce available sets.

In this example no further reductions will occur.

Constraint propagation essentially solved the problem without search.

57

Another example



We can now run the DFS.

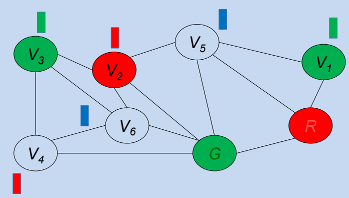
After each assignment we can run constraint propagation to try to further reduce available sets.

In this example no further reductions will occur.

Constraint propagation essentially solved the problem without search.

58

Another example



We can now run the DFS.

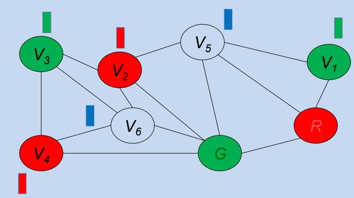
After each assignment we can run constraint propagation to try to further reduce available sets.

In this example no further reductions will occur.

Constraint propagation essentially solved the problem without search.

59

Another example



We can now run the DFS.

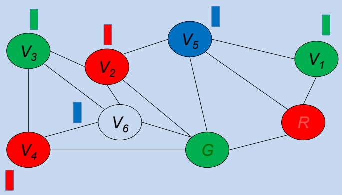
After each assignment we can run constraint propagation to try to further reduce available sets.

In this example no further reductions will occur.

Constraint propagation essentially solved the problem without search.

60

Another example



We can now run the DFS.

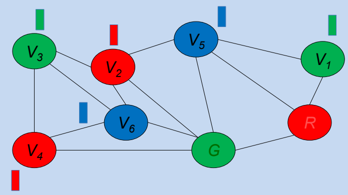
After each assignment we can run constraint propagation to try to further reduce available sets.

In this example no further reductions will occur.

Constraint propagation essentially solved the problem without search.

61

Another example



We can now run the DFS.

After each assignment we can run constraint propagation to try to further reduce available sets.

In this example no further reductions will occur.

Constraint propagation essentially solved the problem without search.

62

Constraint Propagation

- In the previous example, we saw that it is sometimes possible for constraint propagation to solve the problem before we even start the DFS search.
- Not usually that lucky (such as the example before that one).
- Constraint propagation is potentially costly.
  - Whenever you backtrack, you must undo work done by previous constraint propagation steps.
  - Must maintain a dynamic record of its changes in order to properly backtrack.
- As an alternative, you can run constraint propagation once as a preprocessing step, and then just use forward checking during the DFS.

63

Constraint Propagation

- More examples: play animation videos 27p, 49p

64

Lesson 6

GENERAL CONSTRAINT PROPAGATION

65

Graph-coloring-specific Constraint Propagation

In the case of Graph Coloring, CP looks simple: after we've made a search step (instantiated a node with a color), propagate the color at that node.

PropagateColorAtNode(node,color)

1. remove color from all of "available sets" of our uninstantiated neighbors.
2. If any of these neighbors gets the empty set, it's time to backtrack.
3. Foreach n in these neighbors: if n previously had two or more available colors but now has only one color c, run PropagateColorAtNode(n,c)

66

Graph-coloring-specific Constraint Propagation

In the case of Graph Coloring, CP looks simple: after we've made a search step (instantiated a node with a color), propagate the color at that node.

PropagateColorAtNode(node,color)

1. remove color fr... ...antiated neig...
2. If any... ...to backt...
3. Forea... ...these neighbors: if n previously had two or more available colors but now has only one color c, run PropagateColorAtNode(n,c)

But for General CSP problems, constraint propagation can do much more than only propagating when a node gets a unique value…

67

A New CSP (where fancier propagation is possible)

- The semi magic square
- Each variable can have value 1, 2 or 3

V_1	V_2	V_3	This row must sum to 6
V_4	V_5	V_6	This row must sum to 6
V_7	V_8	V_9	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

68

General Constraint Propagation

Propagate(A_1, A_2,… A_n)
  finished = FALSE
  while not finished
    finished = TRUE
    foreach constraint C
      Assume C concerns variables V_1, V_2,… V_k
      Set $NewA_{V1}$ = {} , $NewA_{V2}$ = {} , … $NewA_{Vk}$ = {}
      Foreach assignment (V_1=x_1, V_2=x_2, … V_k=x_k) in C
        If x_1 in A_{V1} **and** x_2 in A_{V2} **and …** x_k in A_{Vk}
          Add x_1 to $NewA_{V1}$, x_2 to $NewA_{V2}$,… x_k to $NewA_{Vk}$
      for i = 1 , 2 … k
        A_{Vi} := A_{Vi} intersection $NewA_{Vi}$
        If A_{Vi} was made smaller by that intersection
          finished = FALSE
        If A_{Vi} is empty, break out with "Backtrack" signal.

Specification: Takes a set of availability-sets for each and every node and uses all the constraints to filter out impossible values that are currently in availability sets

69

General Constraint Propagation

Propagate(A_1, A_2,… A_n)
  finished = FALSE
  while not finished
    fin...shed = TRUE
    fo...ach constraint C
      Assume C concerns variables V_1, V_2,… V_k
      Set $NewA_{V1}$ = {} , $NewA_{V2}$ = {} , … $NewA_{Vk}$ = {}
      ...oreach assignment (V_1=x_1, V_2=x_2, … V_k=x_k) in C
        If x_1 in A_{V1} **and** x_2 in A_{V2} **and …** x_k in A_{Vk}
          Add x_1 to $NewA_{V1}$, x_2 to $NewA_{V2}$,… x_k to $NewA_{Vk}$
      ...i = 1 , 2 … k
        ...n $NewA_{Vi}$
        ...aller by that intersection
          ...SE
        ...ak out with "Backtrack" signal.

A_i denotes the current set of possible values for variable i. This is call-by-reference. Some of the A_i sets may be changed by this call (they'll have one or more elements removed)

We'll keep iterating until we do a full iteration in which none of the availability sets change. The "finished" flag is just to record whether a change took place.

70

General Constraint Propagation

Propagate(A_1, A_2,… A_n)
  finished = FALSE
  while not finished
    finished = TRUE
    foreach constraint C
      Assume C conc...s variables V_1, V_2,… V_k
      Set $NewA_{V1}$ = {} , $NewA_{V2}$ = {} , … $NewA_{Vk}$ = {}
      Foreach assignment (V_1=x_1, V_2=x_2, … V_k=x_k) in C
        If x_1 in A_{V1} **and** x_2 ... **and …** x_k in A_{Vk}
          Add x_1 to $NewA_{V1}$, x_2 ... x_k t...
      for i = 1 , 2 … k
        A_{Vi} := A_{Vi} intersection $NewA_{Vi}$
        If A_{Vi} was made smaller by that intersection
          finished = FALSE
        If A_{Vi} is empty, break out with "Backtrack" s...

$NewA_i$ is going to be filled up with the possible values for variable V_i taking into account the effects of constraint C

After we've finished all the iterations of the foreach loop, *$NewA_i$* contains the full set of possible values of variable V_i taking into account the effects of constraint C.

71

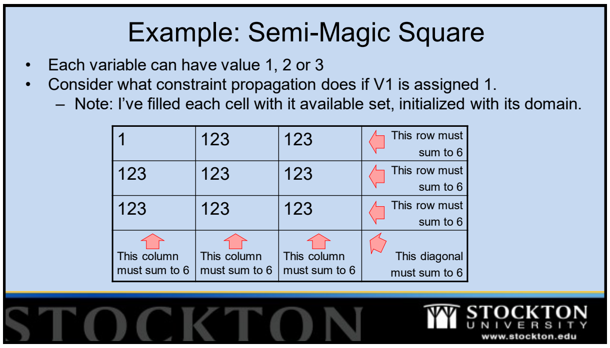
General Constraint Propagation

Propagate(A_1, A_2,… A_n)
  finished = FALSE
  while not finished
    finished = TRUE
    foreach constraint C
      Assume C concerns variabl... … V_k
      ...$NewA_{V1}$ = {} , $NewA_{V2}$ ... ...ewA_{Vk}$ = {}
      ...ach assignment (V_1=... ...x_2, … V_k=x_k) in C
        If x_1 in A_{V1} **and** x_2 in ... **...nd …** x_k in A_{Vk}
          Add x_1 to NewA... ...to $NewA_{V2}$,… x_k to $NewA_{Vk}$
      ...or i = 1 , 2 … k
        A_{Vi} := A_{Vi} intersec...on $NewA_{Vi}$
        ...f A_{Vi} was made smaller by that intersection
          finished = FALSE
        If A_{Vi} is empty, break out with "Backtrack" signal.

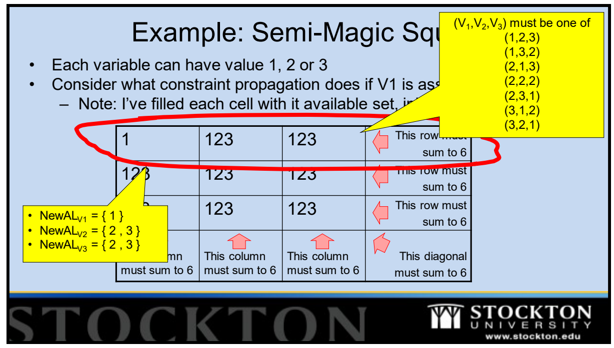
If this test is satisfied that means that there's at least one value q such that we originally thought q was an available value for V_i but we now know q is impossible.

If A_{Vi} is empty we've proved that there are no solutions for the availability-sets that we originally entered the function with.

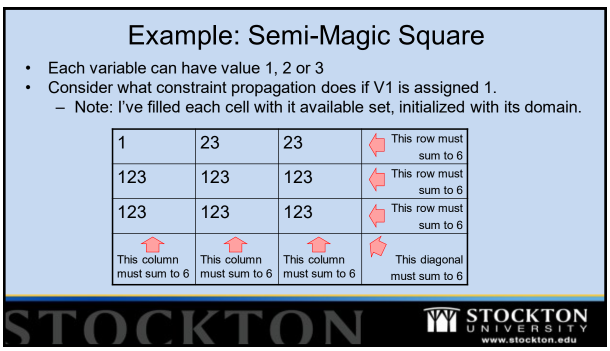
72



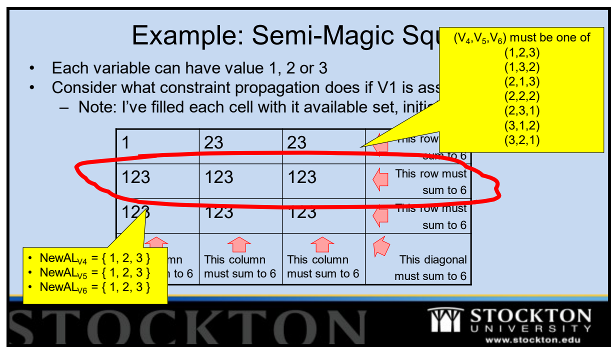
73



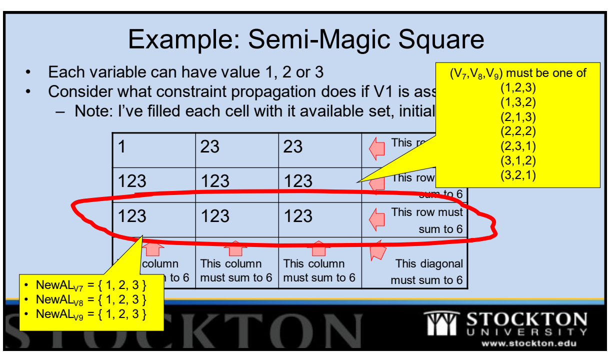
74



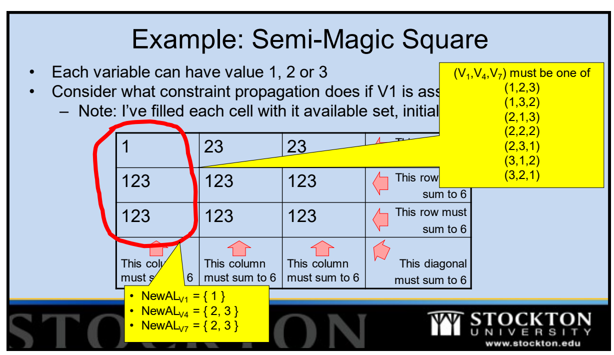
75



76



77



78

Example: Semi-Magic Square

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is assigned 1.
  - Note: I've filled each cell with it available set, initialized with its domain.

1	23	23	This row must sum to 6
23	123	123	This row must sum to 6
23	123	123	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

STOCKTON UNIVERSITY www.stockton.edu

79

Example: Semi-Magic Square

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is ass
  - Note: I've filled each cell with it available set, initial

(V_2,V_5,V_8) must be one of (1,2,3) (1,3,2) (2,1,3) (2,2,2) (2,3,1) (3,1,2) (3,2,1)

1	23	23	This
23	123	123	This row sum to 6
23	123	123	This row must sum to 6
This colu must	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

- NewAL$_{V2}$ = { 2, 3 }
- NewAL$_{V5}$ = { 1, 2, 3 }
- NewAL$_{V8}$ = { 1, 2, 3 }

STOCKTON UNIVERSITY www.stockton.edu

80

Example: Semi-Magic Square

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is ass
  - Note: I've filled each cell with it available set, initial

(V_3,V_6,V_9) must be one of (1,2,3) (1,3,2) (2,1,3) (2,2,2) (2,3,1) (3,1,2) (3,2,1)

1	23	23	This r
23	123	123	This row sum to 6
23	123	123	This row must sum to 6
This column must sum to 6	This colu must s	This column must sum to 6	This diagonal must sum to 6

- NewAL$_{V3}$ = { 2, 3 }
- NewAL$_{V6}$ = { 1, 2, 3 }
- NewAL$_{V9}$ = { 1, 2, 3 }

STOCKTON UNIVERSITY www.stockton.edu

81

Example: Semi-Magic Square

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is ass
  - Note: I've filled each cell with it available set, initial

(V_1,V_5,V_9) must be one of (1,2,3) (1,3,2) (2,1,3) (2,2,2) (2,3,1) (3,1,2) (3,2,1)

1	23	23	This r
23	123	123	This row sum to 6
23	123	123	This row must sum to 6
This column must sum to 6	This colu must s	This column must sum to 6	This diagonal must sum to 6

- NewAL$_{V1}$ = { 1 }
- NewAL$_{V5}$ = { 2, 3 }
- NewAL$_{V9}$ = { 2, 3 }

STOCKTON UNIVERSITY www.stockton.edu

82

After iterating over all constraints once

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is assigned 1.
  - Note: I've filled each cell with it available set, initialized with its domain.

1	23	23	This row must sum to 6
23	23	123	This row must sum to 6
23	123	23	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

STOCKTON UNIVERSITY www.stockton.edu

83

After iterating over all constraints once

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is assigned 1.
  - Note: I've filled each cell with it available set, initialized with its domain.

1	23	23	w must n to 6
23	23		must sum to 6
23			This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

Constraint propagation changed at least one available set during first pass over constraints, so it must iterate over them again.

STOCKTON UNIVERSITY www.stockton.edu

84

Iterate over constraints ag

(V$_4$,V$_5$,V$_6$) must be one of
(1,2,3)
(1,3,2)
(2,1,3)
(2,2,2)
(2,3,1)
(3,1,2)
(3,2,1)

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is ass
  - Note: I've filled each cell with it available set, initi

1	23	23	his row um to 6
23	23	123	This row must sum to 6
23	123	23	This row must sum to 6
mn to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

- NewAL$_{V4}$ = { 2, 3 }
- NewAL$_{V5}$ = { 2, 3 }
- NewAL$_{V6}$ = { 1, 2 }

85

Iterate over constraints again

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is assigned 1.
  - Note: I've filled each cell with it available set, initialized with its domain.

1	23	23	This row must sum to 6
23	23	12	This row must sum to 6
23	123	23	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

86

Iterate over constraints ag

(V$_7$,V$_8$,V$_9$) must be one of
(1,2,3)
(1,3,2)
(2,1,3)
(2,2,2)
(2,3,1)
(3,1,2)
(3,2,1)

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is ass
  - Note: I've filled each cell with it available set, initi

1	23	23	ow sum to 6
23	23	12	This row must um to 6
23	123	23	This row must sum to 6
Th column sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

- NewAL$_{V7}$ = { 2, 3 }
- NewAL$_{V8}$ = { 1, 2 }
- NewAL$_{V9}$ = { 2, 3 }

87

After iterating over all constraints twice

- Each variable can have value 1, 2 or 3
- Consider what constraint propagation does if V1 is assigned 1.
  - Note: I've filled each cell with it available set, initialized with its domain.

1	23	23	This row must sum to 6
23	23	12	This row must sum to 6
23	12	23	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

88

CSP Search with Constraint Propagation

CPSearch(A_1, A_2 ,… A_n)
  Let i = lowest index such that A_i has more than one value
  foreach available value x in A_i
    foreach k in 1, 2.. n
      Define $A'_k := A_k$
    $A'_i := \{ x \}$
    Call Propagate(A'_1, A'_2 ,… A'_n)
    If no "Backtrack" signal
      If A'_1, A'_2 ,… A'_n are all unique we're done!
      Recursively Call CPSearch(A'_1, A'_2 ,… A'_n)

89

CSP Search w ropagation

Specification: Find out if there's any combination of values in the given availability sets that satisfies all constraints.

CPSearch(A_1, A_2 ,… A_n)
  Let i = lowest index such that A_i has more than one value
  foreach available value x in A_i

At this point the A-primes are a copy of the original availability sets except A'_i has committed to value x.

    foreach k in 1, 2.. n
      Define $A'_k := A_k$
    $A'_i := \{ x \}$

This call may prune away some values in some of the copied availability sets

    Call Propagate(A'_1, A'_2 ,… A'_n)
    If no "Backtrack" signal
      If A'_1, A'_2 ,… A'_n are all unique we're done!
      Else recursively Call CPSearch(A'_1, A'_2 ,… A'_n)

Assuming that we terminate deep in the recursion if we find a solution, the CPSeach function only terminates normally if no solution is found.

90

CSP Search with Constraint Propagation

CPSearch($A_1, A_2, \ldots A_n$)

Let i = lowest index such that A_i has more than one value

foreach available value x in A_i

foreach k in 1, 2.. n

Define $A'_k := A_k$

$A'_i := \{ x \}$

Call Propagate($A'_1, A'_2, \ldots A'_n$)

If no "Backtrack" signal

If $A'_1, A'_2, \ldots A'_n$ are all unique we're done!

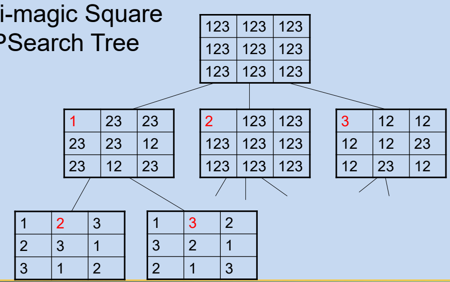
Recursively Call CPSearch($A'_1, A'_2, \ldots A'_n$)

What's the top-level call?

Call with each A_i = the complete domain, D_i, for V_i.

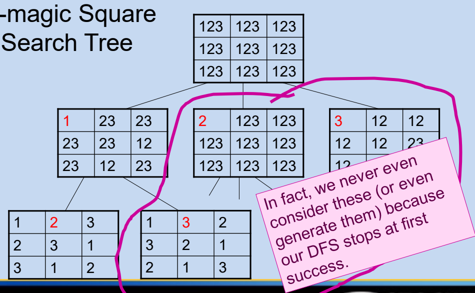
91

Semi-magic Square CPSearch Tree



92

Semi-magic Square CPSearch Tree



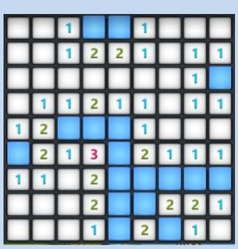
93

Lesson 7

MINESWEEPER AND CONSTRAINT PROPAGATION

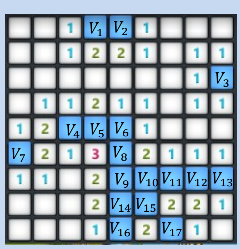
94

Minesweeper as a CSP



Each unknown cell is a variable

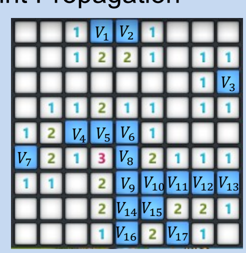
The domain of each variable is $D_i = \{B, E\}$, for Bomb or Empty.



95

Let's Simulate Constraint Propagation

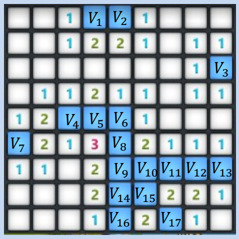
- Each cell with a number is a constraint on adjacent cells (including diagonals).
  - To simplify stepping through, I'm going to iterate over constraints in an order I choose.
  - Algorithm will iterate over constraints in an arbitrary order
  - Doesn't really matter though
    - Final result is the same regardless of order since we keep iterating until a full iteration results in no changes.



96

Let’s Simulate Constraint Propagation

Variable	Available	Variable	Available
V1	B, E	V10	B, E
V2	B, E	V11	B, E
V3	B, E	V12	B, E
V4	B, E	V13	B, E
V5	B, E	V14	B, E
V6	B, E	V15	B, E
V7	B, E	V16	B, E
V8	B, E	V17	B, E
V9	B, E		



97

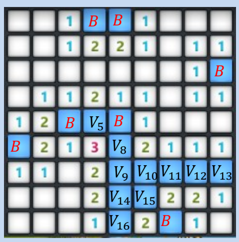
Let’s Simulate Constraint Propagation

Variable	Available	Variable	Available
V1	B, E	V10	B, E
V2	B, E	V11	B, E
V3	B, E	V12	B, E
V4	B, E	V13	B, E
V5	B, E	V14	B, E
V6	B, E	V15	B, E
V7	B, E	V16	B, E
V8	B, E	V17	B, E
V9	B, E		

98

Let’s Simulate Constraint Propagation

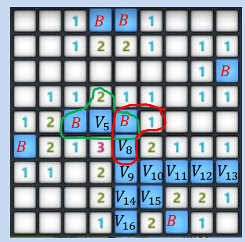
Variable	Available	Variable	Available
V1	B	V10	B, E
V2	B	V11	B, E
V3	B	V12	B, E
V4	B	V13	B, E
V5	B, E	V14	B, E
V6	B	V15	B, E
V7	B	V16	B, E
V8	B, E	V17	B
V9	B, E		



99

Let’s Simulate Constraint Propagation

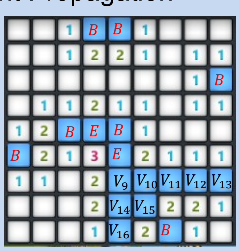
Variable	Available	Variable	Available
V1	B	V10	B, E
V2	B	V11	B, E
V3	B	V12	B, E
V4	B	V13	B, E
V5	B, E	V14	B, E
V6	B	V15	B, E
V7	B	V16	B, E
V8	B, E	V17	B
V9	B, E		



100

Let’s Simulate Constraint Propagation

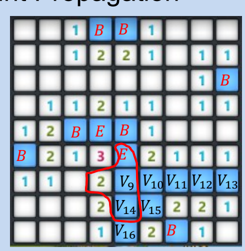
Variable	Available	Variable	Available
V1	B	V10	B, E
V2	B	V11	B, E
V3	B	V12	B, E
V4	B	V13	B, E
V5	E	V14	B, E
V6	B	V15	B, E
V7	B	V16	B, E
V8	E	V17	B
V9	B, E		



101

Let’s Simulate Constraint Propagation

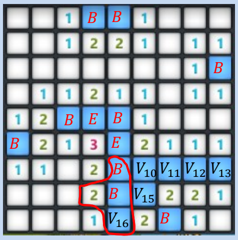
Variable	Available	Variable	Available
V1	B	V10	B, E
V2	B	V11	B, E
V3	B	V12	B, E
V4	B	V13	B, E
V5	E	V14	B, E
V6	B	V15	B, E
V7	B	V16	B, E
V8	E	V17	B
V9	B, E		



102

Let's Simulate Constraint Propagation

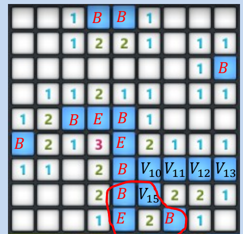
Variable	Available	Variable	Available
V1	B	V10	B, E
V2	B	V11	B, E
V3	B	V12	B, E
V4	B	V13	B, E
V5	E	V14	B
V6	B	V15	B, E
V7	B	V16	B, E
V8	E	V17	B
V9	B		



103

Let's Simulate Constraint Propagation

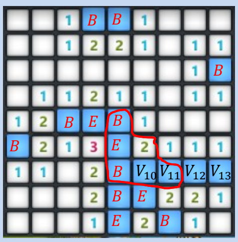
Variable	Available	Variable	Available
V1	B	V10	B, E
V2	B	V11	B, E
V3	B	V12	B, E
V4	B	V13	B, E
V5	E	V14	B
V6	B	V15	B, E
V7	B	V16	E
V8	E	V17	B
V9	B		



104

Let's Simulate Constraint Propagation

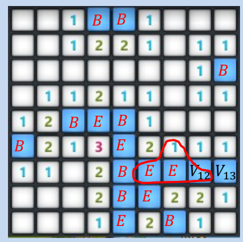
Variable	Available	Variable	Available
V1	B	V10	B, E
V2	B	V11	B, E
V3	B	V12	B, E
V4	B	V13	B, E
V5	E	V14	B
V6	B	V15	E
V7	B	V16	E
V8	E	V17	B
V9	B		



105

Let's Simulate Constraint Propagation

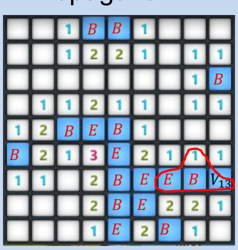
Variable	Available	Variable	Available
V1	B	V10	E
V2	B	V11	E
V3	B	V12	B, E
V4	B	V13	B, E
V5	E	V14	B
V6	B	V15	E
V7	B	V16	E
V8	E	V17	B
V9	B		



106

Let's Simulate Constraint Propagation

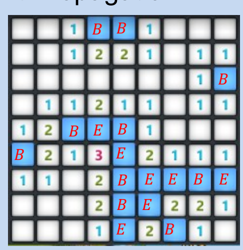
Variable	Available	Variable	Available
V1	B	V10	E
V2	B	V11	E
V3	B	V12	B
V4	B	V13	B, E
V5	E	V14	B
V6	B	V15	E
V7	B	V16	E
V8	E	V17	B
V9	B		



107

Let's Simulate Constraint Propagation

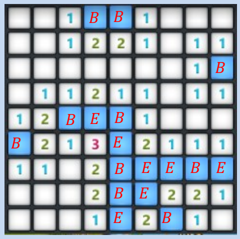
Variable	Available	Variable	Available
V1	B	V10	E
V2	B	V11	E
V3	B	V12	B
V4	B	V13	E
V5	E	V14	B
V6	B	V15	E
V7	B	V16	E
V8	E	V17	B
V9	B		



108

Let's Simulate Constraint Propagation

- In this example, constraint propagation solved the minesweeper instance without doing any DFS.
- Not usually this easy.
  - E.g., you start off with no information.



STOCKTON UNIVERSITY www.stockton.edu

109

Lesson 8

LIMITATIONS FOR MINESWEEPER

STOCKTON UNIVERSITY www.stockton.edu

110

Another Minesweeper Example

- Let's consider what our constraint propagation does with this instance.

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

STOCKTON UNIVERSITY www.stockton.edu

111

Another Minesweeper Example

Variable	Available
V1	B, E
V2	B, E
V3	B, E
V4	B, E
V5	B, E
V6	B, E

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

Constraint: (V1, V2) : { (B, E), (E, B) }

NewA1 = { B E }

NewA2 = { B E }

No reductions

STOCKTON UNIVERSITY www.stockton.edu

112

Another Minesweeper Example

Variable	Available
V1	B, E
V2	B, E
V3	B, E
V4	B, E
V5	B, E
V6	B, E

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

Constraint: (V1, V2, V3) : { (B, E, E), (E, B, E), (E, E, B }

NewA1 = { B E }

NewA2 = { B E }

NewA3 = { B E }

No reductions

STOCKTON UNIVERSITY www.stockton.edu

113

Another Minesweeper Example

Variable	Available
V1	B, E
V2	B, E
V3	B, E
V4	B, E
V5	B, E
V6	B, E

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

Constraint: (V2, V3, V4) : { (B, E, E), (E, B, E), (E, E, B }

NewA2 = { B E }

NewA3 = { B E }

NewA4 = { B E }

No reductions

STOCKTON UNIVERSITY www.stockton.edu

114

Another Minesweeper Example

Variable	Available
V1	B, E
V2	B, E
V3	B, E
V4	B, E
V5	B, E
V6	B, E

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

Constraint: (V3, V4, V5) : { (B, E, E), (E, B, E), (E, E, B }

NewA3 = { B E }

NewA4 = { B E }

NewA5 = { B E }

No reductions

115

Another Minesweeper Example

Variable	Available
V1	B, E
V2	B, E
V3	B, E
V4	B, E
V5	B, E
V6	B, E

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

Constraint: (V4, V5, V6) : { (B, E, E), (E, B, E), (E, E, B }

NewA4 = { B E }

NewA5 = { B E }

NewA6 = { B E }

No reductions

116

Another Minesweeper Example

Variable	Available
V1	B, E
V2	B, E
V3	B, E
V4	B, E
V5	B, E
V6	B, E

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

Constraint: (V5, V6) : { (B, E), (E, B) }

NewA5 = { B E }

NewA6 = { B E }

No reductions

117

The Limitation

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

- Constraint propagation (the form we've seen) doesn't reduce any of the available sets.
- Shouldn't it be able to though?
  - Exactly one of V1 or V2 must be a B (to satisfy the left-most constraint), which means V3 must be E.
  - Exactly one of V5 or V6 must be a B (to satisfy the right-most constraint), which means V4 must be E.
  - Since V3 and V4 must both be E, then V2 and V5 must both be B (to satisfy middle two constraints).
  - If V2 is B, then V1 must be E.
  - If V5 is B, then V6 must be E.

This logic is sound, and leads to the only solution.

But the form of consistency that our constraint propagation relies on doesn't do this.

Need to propagate k-consistency for this (see textbook).

118

Lesson 9

ORDERING HEURISTICS

119

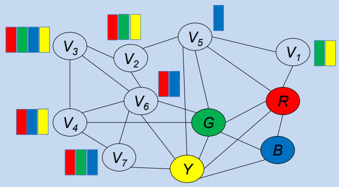
Ordering Heuristics

- Variable Ordering Heuristics:
  - We can often more easily solve constraint satisfaction problems if we intelligently choose the order that we consider variables (rather than considering them in whatever arbitrary order they are given).
  - Technically, it doesn't matter what order we work on variables, but some may be easier to assign than others.
- Value Ordering Heuristics:
  - Once we've decided which variable to work on next, we can often more easily solve constraint satisfaction problems if we intelligently choose the order that we try values its domain.

120

Ordering Heuristics

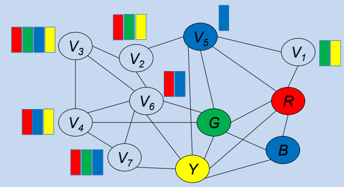
Here's another graph-coloring example (you're now allowed *R*, *G*, *B* and *Y*). The availability lists after running constraint propagation are shown.



121

Ordering Heuristics

For this example, let's assume that V5 is assigned Blue since it only has the one value remaining in its available list after constraint propagation.

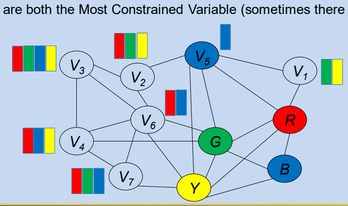


STOCKTON
STOCKTON UNIVERSITY www.stockton.edu

122

Most Constrained Variable

- The **Most Constrained Variable** heuristic chooses the (unassigned) variable with the fewest available values remaining. [This is a variable ordering heuristic.]
- V1 and V6 are both the Most Constrained Variable (sometimes there are ties).



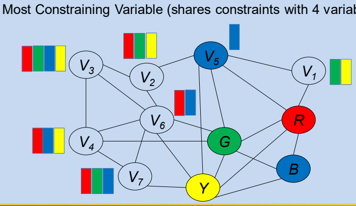
Rationale: The most constrained variable has the fewest options left, so will be harder to find a value. Therefore, work on it first.

STOCKTON
STOCKTON UNIVERSITY www.stockton.edu

123

Most Constraining Variable

- The **Most Constraining Variable** heuristic chooses the (unassigned) variable that shares the most constraints with other unsigned variables. [This is a variable ordering heuristic.]
- V6 is the Most Constraining Variable (shares constraints with 4 variables V2, V3, V4, V7).



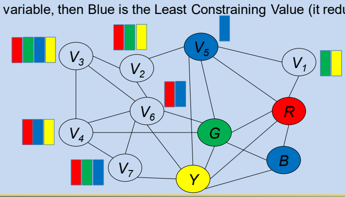
Rationale: The most constraining variable has the greatest impact on other variables, so try to get it right first.

STOCKTON
STOCKTON UNIVERSITY www.stockton.edu

124

Least Constraining Value

- Once a variable is chosen, the **Least Constraining Value** heuristic chooses the value from its available set that causes the least reduction in available sets of variables with which it shares constraints. [This is a value ordering heuristic.]
- If V6 is the variable, then Blue is the Least Constraining Value (it reduces V3, V4, V7).



Rationale: The least constraining value leaves the most flexibility.

STOCKTON
STOCKTON UNIVERSITY www.stockton.edu

125

Common Technique

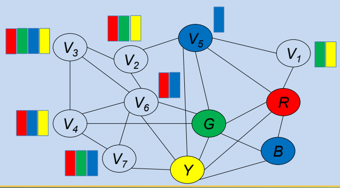
- Variable Ordering:
  1. Most Constrained Variable
  2. Break ties with Most Constraining Variable
- Value Ordering:
  - Once a variable is chosen, use Least Constraining Value to choose a value for it.

STOCKTON
STOCKTON UNIVERSITY www.stockton.edu

126

Example

- Most Constrained Variable
- Break ties with Most Constraining Variable
- Choose a value with Least Constraining Value.



V1 and V6 tied as most constrained variable.

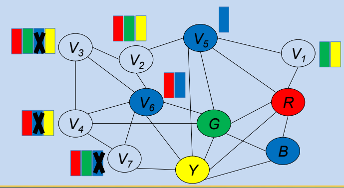
V6 is most constraining variable.

Blue is least constraining value.

127

Example

- Most Constrained Variable
- Break ties with Most Constraining Variable
- Choose a value with Least Constraining Value.



V1, V4, and V7 are tied as most constrained variable.

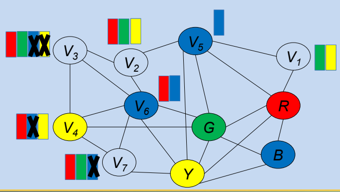
V4 is most constraining variable.

Yellow is least constraining value.

128

Example

- Most Constrained Variable
- Break ties with Most Constraining Variable
- Choose a value with Least Constraining Value.



V1, V3, and V7 are tied as most constrained variable.

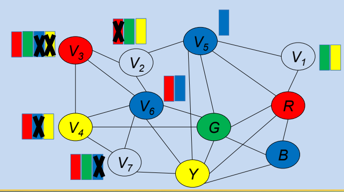
V3 is most constraining variable.

Red and green tied as least constraining value.

129

Example

- Most Constrained Variable
- Break ties with Most Constraining Variable
- Choose a value with Least Constraining Value.



V1, V2, and V7 are tied as most constrained variable.

Still tied as most constraining variable.

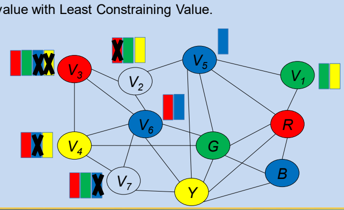
We'll pick V1 arbitrarily.

Green and yellow tied as least constraining value.

130

Example

- Most Constrained Variable
- Break ties with Most Constraining Variable
- Choose a value with Least Constraining Value.



V2, and V7 are tied as most constrained variable.

Still tied as most constraining variable.

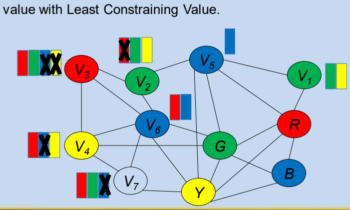
We'll pick V2 arbitrarily.

Green and yellow tied as least constraining value.

131

Example

- Most Constrained Variable
- Break ties with Most Constraining Variable
- Choose a value with Least Constraining Value.



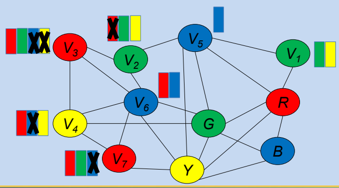
Only V7 left

Red and Green tied as least constraining value.

132

Example

- Most Constrained Variable
- Break ties with Most Constraining Variable
- Choose a value with Least Constraining Value.



With the help of variable ordering heuristics, and value ordering heuristics, constraint propagation helped solve the problem without the need for the DFS search to backtrack.

Not always this lucky but works well in practice.

133

Ordering Heuristics and Minesweeper

- Most constrained variable: V1,V2,V3,V4,V5,V6
  - They all have 2 values available.
  - Recall constraint propagation didn't reduce any to start with.
- Most Constraining Variable: V3, V4
  - V3 shares constraints with V1,V2,V4,V5
  - V4 shares constraints with V2,V3,V5,V6
  - V2 shares constraints with V1,V3,V4
  - V5 shares constraints with V3,V4,V6
  - V1 shares constraints with V2,V3
  - V6 shares constraints with V4,V5
- If we pick V3, then Least Constraining Value: E

1	1	1	1	1	1
V1	V2	V3	V4	V5	V6

Variable	Available
V1	B, E
V2	B, E
V3	B, E
V4	B, E
V5	B, E
V6	B, E

STOCKTON UNIVERSITY www.stockton.edu

134

Ordering Heuristics and Minesweeper

- Most constrained variable: V1,V2,V4,V5,V6
  - They all have 2 values available.
  - Constraint propagation doesn't reduce any in this case.
- Most Constraining Variable: V4
  - V4 shares constraints with V2,V5,V6
  - V2 shares constraints with V1,V4
  - V5 shares constraints with V4,V6
  - V6 shares constraints with V4,V5
  - V1 shares constraints with V2
- Least Constraining Value: E
  - E removes E from V2 and V5
  - B removes B from V2, V5, V6

1	1	1	1	1	1
V1	V2	E	V4	V5	V6

Variable	Available
V1	B, E
V2	B, E
V3	E
V4	B, E
V5	B, E
V6	B, E

STOCKTON UNIVERSITY www.stockton.edu

135

Ordering Heuristics and Minesweeper

- Constraint propagation now runs before next selection is made.
  - V4 = E causes V2=B and V5=B.

1	1	1	1	1	1
V1	V2	E	E	V5	V6

Variable	Available
V1	B, E
V2	B
V3	E
V4	E
V5	B
V6	B, E

STOCKTON UNIVERSITY www.stockton.edu

136

Ordering Heuristics and Minesweeper

- Constraint propagation now runs before next selection is made.
  - V4 = E causes V2=B and V5=B.
  - V2=B causes V1=E.
  - V5=B causes V6=E.

1	1	1	1	1	1
V1	B	E	E	B	V6

Variable	Available
V1	E
V2	B
V3	E
V4	E
V5	B
V6	E

STOCKTON UNIVERSITY www.stockton.edu

137

Ordering Heuristics and Minesweeper

- Constraint propagation now runs before next selection is made.
  - V4 = E causes V2=B and V5=B.
  - V2=B causes V1=E.
  - V5=B causes V6=E.

1	1	1	1	1	1
E	B	E	E	B	E

Variable	Available
V1	E
V2	B
V3	E
V4	E
V5	B
V6	E

STOCKTON UNIVERSITY www.stockton.edu

138