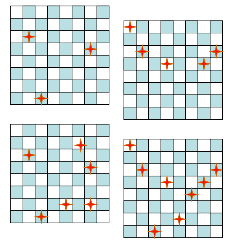
Constraint Satisfaction Problems (CSP)

(Where we delay difficult decisions until they become easier)

R&N: Chap. 6

(These slides are primarily from a course at Stanford University – any mistakes were undoubtedly added by me.)

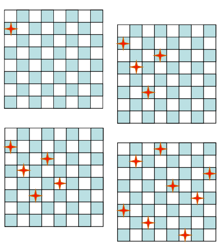
8-Queens: Search Formulation #1



- **States:** all arrangements of 0, 1, 2, ..., or 8 queens on the board
- **Initial state:** 0 queen on the board
- **Successor function:** each of the successors is obtained by adding one queen in a non-empty square
- **Arc cost:** irrelevant
- **Goal test:** 8 queens are on the board, with no two of them attacking each other

→ 64x63x...x53 ~ 3×10^{14} states

8-Queens: Search Formulation #2



- **States:** all arrangements of k = 0, 1, 2, ..., or 8 queens in the k leftmost columns with no two queens attacking each other
- **Initial state:** 0 queen on the board
- **Successor function:** each successor is obtained by adding one queen in any square that is not attacked by any queen already in the board, in the leftmost empty column
- **Arc cost:** irrelevant
- **Goal test:** 8 queens are on the board

→ 2,057 states

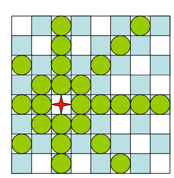
Issue

- Previous search techniques make choices in an often arbitrary order, even if there is still little information explicitly available to choose well.
- There are some problems (called constraint satisfaction problems) whose states and goal test conform to a standard, structured, and very simple representation.
- This representation views the problem as consisting of a set of variables in need of values that conform to certain constraint.

Issue

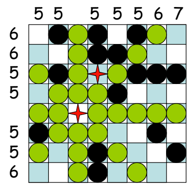
- In such problems, the same states can be reached independently of the order in which choices are made (commutative actions)
- These problems lend themselves to *general-purpose* rather than *problem-specific* heuristics to enable the solution of large problems
- Can we solve such problems more efficiently by picking the order appropriately? Can we even avoid having to make choices?

Constraint Propagation



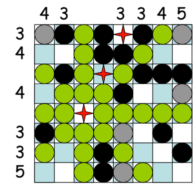
- Place a queen in a square
- Remove the attacked squares from future consideration

Constraint Propagation



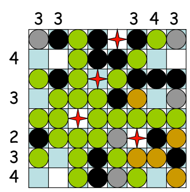
- Count the number of non-attacked squares in every row and column
- Place a queen in a row or column with minimum number
- Remove the attacked squares from future consideration

Constraint Propagation

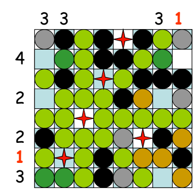


- Repeat

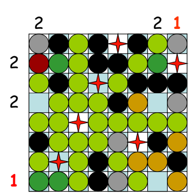
Constraint Propagation



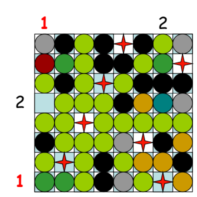
Constraint Propagation



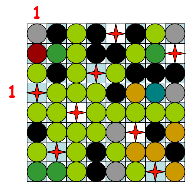
Constraint Propagation



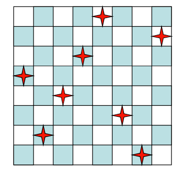
Constraint Propagation



Constraint Propagation



Constraint Propagation



What do we need?

- More than just a successor function and a goal test
- We also need:
  - A means to propagate the constraints imposed by one queen's position on the the positions of the other queens
  - An early failure test

→ Explicit representation of constraints
→ Constraint propagation algorithms

Constraint Satisfaction Problem (CSP)

- Set of variables $\{X_1, X_2, \ldots, X_n\}$
- Each variable X_i has a domain D_i of possible values. Usually, D_i is finite
- Set of constraints $\{C_1, C_2, \ldots, C_p\}$
- Each constraint relates a subset of variables by specifying the valid combinations of their values
- Goal: Assign a value to every variable such that all constraints are satisfied

8-Queens: Formulation #1

- 64 variables X_{ij}, i = 1 to 8, j = 1 to 8
- The domain of each variable is: {1,0}
- Constraints are of the forms:
  - $X_{ij} = 1 \Rightarrow X_{ik} = 0$ for all k = 1 to 8, k≠j
  - $X_{ij} = 1 \Rightarrow X_{kj} = 0$ for all k = 1 to 8, k≠i
  - Similar constraints for diagonals
  - $\Sigma_{i,j\in[1,8]} X_{ij} = 8$

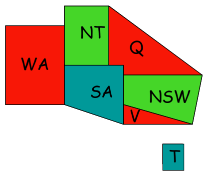
Binary constraints
(each constraint relates only 2 variables)

8-Queens: Formulation #2

- 8 variables X_i, i = 1 to 8
- The domain of each variable is: {1,2,…,8}
- Constraints are of the forms:
  - $X_i = k \Rightarrow X_j \neq k$ for all j = 1 to 8, j≠i
  - Similar constraints for diagonals

All constraints are binary

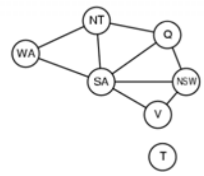
Map Coloring



- 7 variables {WA,NT,SA,Q,NSW,V,T}
- Each variable has the same domain: {red, green, blue}
- No two adjacent variables have the same value:
  WA≠NT, WA≠SA, NT≠SA, NT≠Q, SA≠Q, SA≠NSW, SA≠V,Q≠NSW, NSW≠V

Constraint graph

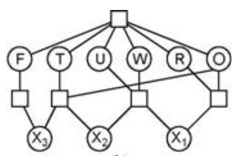
- Constraint graph: nodes are variables, arcs are constraints



A Cryptarithmetic Problem

```
  T W O
+ T W O
F O U R
```

(a)



(b)

- Here each constraint is a square box connected to the variables it constrains
- allDiff; O + O = R + 10 * $X_{1;...}$

Street Puzzle

1 2 3 4 5

N_i = {English, Spaniard, Japanese, Italian, Norwegian}
C_i = {Red, Green, White, Yellow, Blue}
D_i = {Tea, Coffee, Milk, Fruit-juice, Water}
J_i = {Painter, Sculptor, Diplomat, Violinist, Doctor}
A_i = {Dog, Snails, Fox, Horse, Zebra}

The Englishman lives in the Red house
The Spaniard has a Dog
The Japanese is a Painter
The Italian drinks Tea
The Norwegian lives in the first house on the left
The owner of the Green house drinks Coffee
The Green house is on the right of the White house
The Sculptor breeds Snails
The Diplomat lives in the Yellow house
The owner of the middle house drinks Milk
The Norwegian lives next door to the Blue house
The Violinist drinks Fruit juice
The Fox is in the house next to the Doctor's
The Horse is next to the Diplomat's

Who owns the Zebra?
Who drinks Water?

Street Puzzle

1 2 3 4 5

N_i = {English, Spaniard, Japanese, Italian, Norwegian}
C_i = {Red, Green, White, Yellow, Blue}
D_i = {Tea, Coffee, Milk, Fruit-juice, Water}
J_i = {Painter, Sculptor, Diplomat, Violinist, Doctor}
A_i = {Dog, Snails, Fox, Horse, Zebra}

The Englishman lives in the Red house → (N_i = English) ⇔ (C_i = Red)
The Spaniard has a Dog
The Japanese is a Painter → (N_i = Japanese) ⇔ (J_i = Painter)
The Italian drinks Tea
The Norwegian lives in the first house on the left → (N_1 = Norwegian)
The owner of the Green house drinks Coffee
The Green house is on the right of the White house → (C_i = White) ⇔ (C_{i+1} = Green)
The Sculptor breeds Snails
The Diplomat lives in the Yellow house
The owner of the middle house drinks Milk
The Norwegian lives next door to the Blue house → (C_5 ≠ White), (C_1 ≠ Green)
The Violinist drinks Fruit juice
The Fox is in the house next to the Doctor's → left as an exercise
The Horse is next to the Diplomat's

Street Puzzle

1 2 3 4 5

N_i = {English, Spaniard, Japanese, Italian, Norwegian}
C_i = {Red, Green, White, Yellow, Blue}
D_i = {Tea, Coffee, Milk, Fruit-juice, Water}
J_i = {Painter, Sculptor, Diplomat, Violinist, Doctor}
A_i = {Dog, Snails, Fox, Horse, Zebra}

The Englishman lives in the Red house → (N_i = English) ⇔ (C_i = Red)
The Spaniard has a Dog
The Japanese is a Painter → (N_i = Japanese) ⇔ (J_i = Painter)
The Italian drinks Tea
The Norwegian lives in the first house on the left → (N_1 = Norwegian)
The owner of the Green house drinks Coffee
The Green house is on the right of the White house → (C_i = White) ⇔ (C_{i+1} = Green)
The Sculptor breeds Snails
The Diplomat lives in the Yellow house
The owner of the middle house drinks Milk
The Norwegian lives next door to the Blue house → (C_5 ≠ White), (C_1 ≠ Green)
The Violinist drinks Fruit juice
The Fox is in the house next to the Doctor's
The Horse is next to the Diplomat's

unary constraints

Street Puzzle

1 2 3 4 5

N_i = {English, Spaniard, Japanese, Italian, Norwegian}
C_i = {Red, Green, White, Yellow, Blue}
D_i = {Tea, Coffee, Milk, Fruit-juice, Water}
J_i = {Painter, Sculptor, Diplomat, Violinist, Doctor}
A_i = {Dog, Snails, Fox, Horse, Zebra}

$\forall i,j \in [1,5], i \neq j, N_i \neq N_j$
$\forall i,j \in [1,5], i \neq j, C_i \neq C_j$
...

The Englishman lives in the Red house
The Spaniard has a Dog
The Japanese is a Painter
The Italian drinks Tea
The Norwegian lives in the first house on the left
The owner of the Green house drinks Coffee
The Green house is on the right of the White house
The Sculptor breeds Snails
The Diplomat lives in the Yellow house
The owner of the middle house drinks Milk
The Norwegian lives next door to the Blue house
The Violinist drinks Fruit juice
The Fox is in the house next to the Doctor's
The Horse is next to the Diplomat's

Street Puzzle

1 2 3 4 5

N_i = {English, Spaniard, Japanese, Italian, Norwegian}
C_i = {Red, Green, White, Yellow, Blue}
D_i = {Tea, Coffee, Milk, Fruit-juice, Water}
J_i = {Painter, Sculptor, Diplomat, Violinist, Doctor}
A_i = {Dog, Snails, Fox, Horse, Zebra}

The Englishman lives in the Red house
The Spaniard has a Dog
The Japanese is a Painter
The Italian drinks Tea
The Norwegian lives in the first house on the left → N_1 = Norwegian
The owner of the Green house drinks Coffee
The Green house is on the right of the White house
The Sculptor breeds Snails
The Diplomat lives in the Yellow house
The owner of the middle house drinks Milk → D_3 = Milk
The Norwegian lives next door to the Blue house
The Violinist drinks Fruit juice
The Fox is in the house next to the Doctor's
The Horse is next to the Diplomat's

Street Puzzle

1 2 3 4 5

N_i = {English, Spaniard, Japanese, Italian, Norwegian}
C_i = {Red, Green, White, Yellow, Blue}
D_i = {Tea, Coffee, Milk, Fruit-juice, Water}
J_i = {Painter, Sculptor, Diplomat, Violinist, Doctor}
A_i = {Dog, Snails, Fox, Horse, Zebra}

The Englishman lives in the Red house → $C_1 \neq$ Red
The Spaniard has a Dog → $A_1 \neq$ Dog
The Japanese is a Painter
The Italian drinks Tea
The Norwegian lives in the first house on the left → N_1 = Norwegian
The owner of the Green house drinks Coffee
The Green house is on the right of the White house
The Sculptor breeds Snails
The Diplomat lives in the Yellow house
The owner of the middle house drinks Milk → D_3 = Milk
The Norwegian lives next door to the Blue house
The Violinist drinks Fruit juice → $J_3 \neq$ Violinist
The Fox is in the house next to the Doctor's
The Horse is next to the Diplomat's

Finite vs. Infinite CSP

- Finite CSP: each variable has a finite domain of values
- Infinite CSP: some or all variables have an infinite domain
  E.g., linear programming problems over the reals:
  for i = 1, 2, ..., p : $a_{i,1}x_1+a_{i,2}x_2+\ldots+a_{i,n}x_n = a_{i,0}$
  for j = 1, 2, ..., q : $b_{j,1}x_1+b_{j,2}x_2+\ldots+b_{j,n}x_n \leq b_{j,0}$
- We will only consider finite CSP

What does CSP Buy You?

- Each of these problems has a standard pattern – a set of variables that need to be assigned values that conform to a set of constraints.
- Successors function and a Goal test predicate can be written that works for any such problem.
- Generic Heuristics can be used for solving that require NO DOMAIN-SPECIFIC EXPERTISE
- The constraint graph structure can be used to simplify the search process.

CSP as a Search Problem

- n variables $X_1, \ldots, X_n$
- Valid assignment:
  $\{X_{i1} \leftarrow v_{i1}, \ldots, X_{ik} \leftarrow v_{ik}\}$, $0 \leq k \leq n$,
  such that the values $v_{i1}, \ldots, v_{ik}$ satisfy all constraints relating the variables $X_{i1}, \ldots, X_{ik}$
- States: valid assignments
- Initial state: empty assignment (k = 0)
- Successor of a state:
  $\{X_{i1}\leftarrow v_{i1}, \ldots, X_{ik}\leftarrow v_{ik}\} \rightarrow \{X_{i1}\leftarrow v_{i1}, \ldots, X_{ik}\leftarrow v_{ik}, X_{ik+1}\leftarrow v_{ik+1}\}$
- Goal test: complete assignment (k = n)

How to solve?

- States: valid assignments
- Initial state: empty assignment (k = 0)
- Successor of a state:

  $\{X_{i1} \leftarrow v_{i1}, \ldots, X_{ik} \leftarrow v_{ik}\} \rightarrow \{X_{i1} \leftarrow v_{i1}, \ldots, X_{ik} \leftarrow v_{ik}, X_{ik+1} \leftarrow v_{ik+1}\}$
- Goal test: complete assignment (k = n)

- NOTE: If "regular" search algorithm is used, the branching factor is quite large since the successor function must try (1) all unassigned variables, and (2) for each of those variables, try all possible values

A Key property of CSP: Commutativity

The order in which variables are assigned values has no impact on the assignment reached

Hence:

1) One can generate the successors of a node by first selecting **one** variable and then assigning every value in the domain of this variable
[→ big reduction in branching factor]

- 4 variables $X_1, \ldots, X_4$
- Let the current assignment be:
  $A = \{X_1 \leftarrow v_1, X_3 \leftarrow v_3\}$
- (For example) pick variable X_4
- Let the domain of X_4 be $\{v_{4,1}, v_{4,2}, v_{4,3}\}$
- The successors of A are

  $\{X_1 \leftarrow v_1, X_3 \leftarrow v_3, X_4 \leftarrow v_{4,1}\}$
  $\{X_1 \leftarrow v_1, X_3 \leftarrow v_3, X_4 \leftarrow v_{4,2}\}$
  $\{X_1 \leftarrow v_1, X_3 \leftarrow v_3, X_4 \leftarrow v_{4,3}\}$

A Key property of CSP: Commutativity

The order in which variables are assigned values has no impact on the assignment reached

Hence:

1) One can generate the successors of a node by first selecting **one** variable and then assigning every value in the domain of this variable
[→ big reduction in branching factor]

2) One need not store the path to a node

→ Backtracking search algorithm

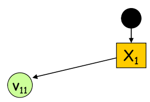
Backtracking Search

Essentially a simplified depth-first algorithm using recursion

Backtracking Search
(3 variables)

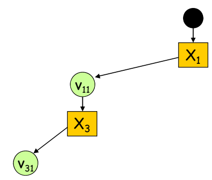
Assignment = {}

Backtracking Search
(3 variables)



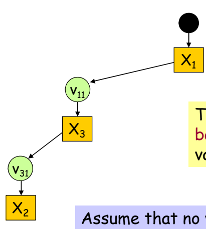
Assignment = {(X_1,v_{11})}

Backtracking Search
(3 variables)



Assignment = {(X_1,v_{11}), (X_3,v_{31})}

Backtracking Search
(3 variables)

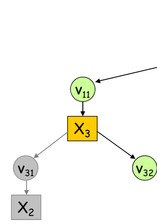


Then, the search algorithm backtracks to the previous variable and tries another value

Assume that no value of X_2 leads to a valid assignment

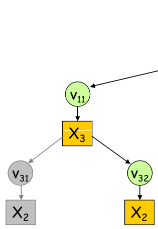
Assignment = {(X_1,v_{11}), (X_3,v_{31})}

Backtracking Search
(3 variables)



Assignment = {(X_1,v_{11}), (X_3,v_{32})}

Backtracking Search
(3 variables)

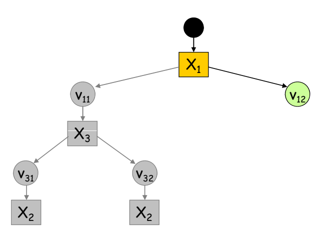


The search algorithm backtracks to the previous variable (X_3) and tries another value. But assume that X_3 has only two possible values. The algorithm backtracks to X_1

Assume again that no value of X_2 leads to a valid assignment

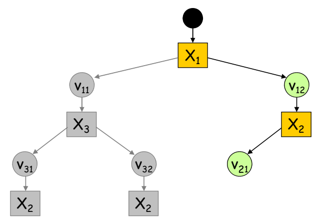
Assignment = {(X_1,v_{11}), (X_3,v_{32})}

Backtracking Search
(3 variables)



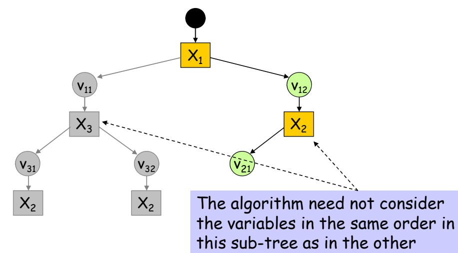
Assignment = {(X_1,v_{12})}

Backtracking Search
(3 variables)



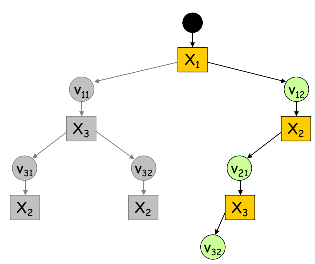
Assignment = {(X_1,v_{12}), (X_2,v_{21})}

Backtracking Search
(3 variables)



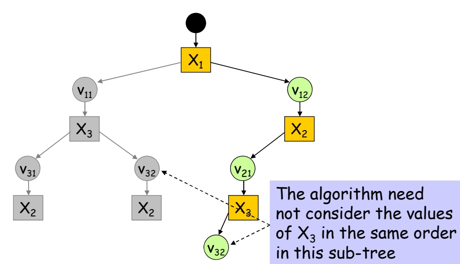
Assignment = {(X_1,v_{12}), (X_2,v_{21})}

Backtracking Search
(3 variables)



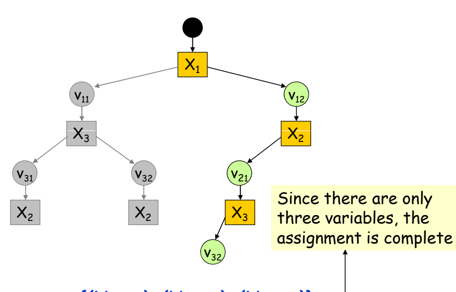
Assignment = {(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})}

Backtracking Search
(3 variables)



Assignment = {(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})}

Backtracking Search
(3 variables)



Assignment = {(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})}

Backtracking Algorithm

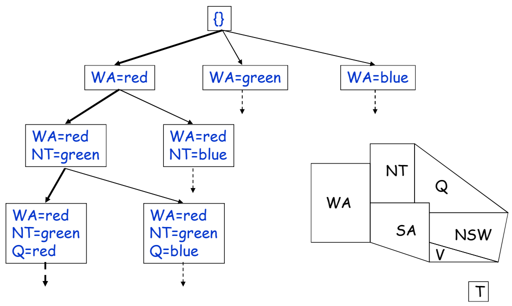
CSP-BACKTRACKING(A)

1. If assignment A is complete then return A
2. X ← select a variable not in A
3. D ← select an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to A
   b. If A is valid then
      i. result ← CSP-BACKTRACKING(A)
      ii. If result ≠ failure then return result
5. Return failure

Call CSP-BACKTRACKING({})

[This recursive algorithm keeps too much data in memory.
An iterative version could save memory (left as an exercise)]

Map Coloring



Critical Questions for the Efficiency of CSP-Backtracking

CSP-BACKTRACKING(a)

1. If assignment A is complete then return A
2. X ← **select** a variable not in A
3. D ← **select** an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to A
   b. If a is valid then
      i. result ← CSP-BACKTRACKING(A)
      ii. If result ≠ failure then return result
5. Return failure

Critical Questions for the Efficiency of CSP-Backtracking

1) Which variable X should be assigned a value next?

2) In which order should X's values be assigned?

Critical Questions for the Efficiency of CSP-Backtracking

1) Which variable X should be assigned a value next?
   The current assignment may not lead to any solution, but the algorithm still does not know it. Selecting the right variable to which to assign a value may help discover the contradiction more quickly.

2) In which order should X's values be assigned?

Critical Questions for the Efficiency of CSP-Backtracking

1) Which variable X should be assigned a value next?
   The current assignment may not lead to any solution, but the algorithm still does not know it. Selecting the right variable to which to assign a value may help discover the contradiction more quickly.

2) In which order should X's values be assigned?
   The current assignment may be part of a solution. Selecting the right value to assign to X may help discover this solution more quickly.

Critical Questions for the Efficiency of CSP-Backtracking

1) Which variable X should be assigned a value next?
   The current assignment may not lead to any solution, but the algorithm still does not know it. Selecting the right variable to which to assign a value may help discover the contradiction more quickly.

2) In which order should X's values be assigned?
   The current assignment may be part of a solution. Selecting the right value to assign to X may help discover this solution more quickly.

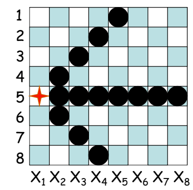
More on these questions in a short while ...

Propagating Information Through Constraints

- Our search algorithm considers the constraints on a variable only at the time that the variable is chosen to be given a value.
- If we can we look at constraints earlier, we might be able to drastically reduce the search space.

Forward Checking

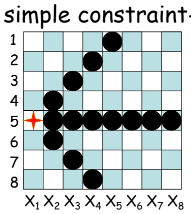
A simple constraint-propagation technique:



Assigning the value 5 to X_1 leads to removing values from the domains of X_2, X_3, ..., X_8

Forward Checking

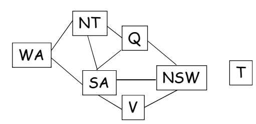
A simple constraint-propagation technique:



Assigning the value 5 to X_1 leads to removing values from the domains of X_2, X_3, ..., X_8

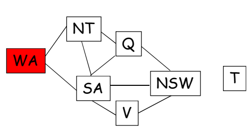
Whenever a variable X is assigned, forward checking looks at each unassigned variable Y that is connected to X by a constraint, and removes from Y's domain any value that is inconsistent with the value chosen for x.

Forward Checking in Map Coloring



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB

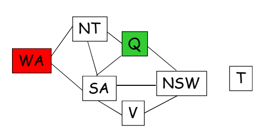
Forward Checking in Map Coloring



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	~~R~~GB	RGB	RGB	RGB	~~R~~GB	RGB

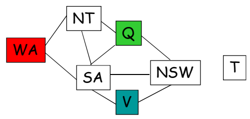
Forward checking removes the value Red of NT and of SA

Forward Checking in Map Coloring



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	~~G~~B	G	R~~G~~B	RGB	~~G~~B	RGB

Forward Checking in Map Coloring



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

Forward Checking in Map Coloring

Empty set: the current assignment {(WA ← R), (Q ← G), (V ← B)} does not lead to a solution

WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

Forward Checking (General Form)

When a pair (X←v) is added to assignment *A* do:

For each variable Y not in *A* do:

For every constraint *C* relating Y to X do:

Remove all values from Y's domain that do not satisfy *C*

Modified Backtracking Algorithm

CSP-BACKTRACKING(A, var-domains)

1. If assignment *A* is complete then return *A*
2. X ← select a variable not in *A*
3. D ← select an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to *A*
   b. var-domains ← forward checking(var-domains, X, v, *A*)
   c. If a variable has an empty domain then return failure
   d. result ← CSP-BACKTRACKING(*A*, var-domains)
   e. If result ≠ failure then return result
5. Return failure

Modified Backtracking Algorithm

CSP-BACKTRACKING(A, var-domains)

1. If assignment *A* is complete then return *A*
2. X ← select a variable not in *A*
3. D ← select an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to *A*
   b. var-domains ← forward checking(var-domains, X, v, *A*)
   c. If a variable has an empty domain then return failure
   d. result ← CSP-BACKTRACKING(*A*, var-domains)
   e. If result ≠ failure then return result
5. Return failure

No need any more to verify that *A* is valid

Modified Backtracking Algorithm

CSP-BACKTRACKING(A, var-domains)

1. If assignment *A* is complete then return *A*
2. X ← select a variable not in *A*
3. D ← select an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to *A*
   b. var-domains ← forward checking(var-domains, X, v, *A*)
   c. If a variable has an empty domain then return failure
   d. result ← CSP-BACKTRACKING(*A*, var-domains)
   e. If result ≠ failure then return result
5. Return failure

Need to pass down the updated variable domains

1) Which variable X_i should be assigned a value next?
→Most-constrained-variable heuristic
(also called minimum remaining values heuristic)
→ Most-constraining-variable heuristic

2) In which order should its values be assigned?
→ Least-constraining-value heuristic

Keep in mind that **all** variables must eventually get a value, while only **one** value from a domain must be assigned to each variable.

The general idea with 1) is, if you are going to fail, do so as quickly as possible. With 2) it is give yourself the best chance for success.

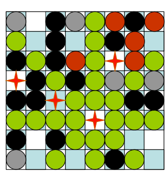
Most-Constrained-Variable Heuristic

1) Which variable X_i should be assigned a value next?

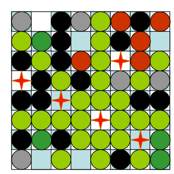
Select the variable with the smallest remaining domain

[Rationale: Minimize the branching factor]

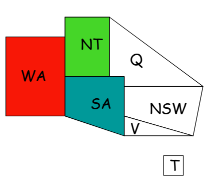
8-Queens



8-Queens



Map Coloring



- SA's remaining domain has size 1 (value Blue remaining)
- Q's remaining domain has size 2
- NSW's, V's, and T's remaining domains have size 3

→ Select SA

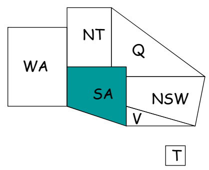
Most-Constraining-Variable Heuristic

1) Which variable X_i should be assigned a value next?

Among the variables with the smallest remaining domains (ties with respect to the most-constrained variable heuristic), select the one that appears in the largest number of constraints on variables not in the current assignment

[Rationale: Increase future elimination of values, to reduce branching factors]

Map Coloring



- Before any value has been assigned, all variables have a domain of size 3, but SA is involved in more constraints (5) than any other variable

→ Select SA and assign a value to it (e.g., Blue)

Least-Constraining-Value Heuristic

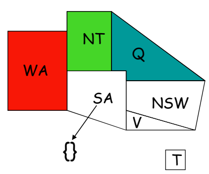
2) In which order should X's values be assigned?

Select the value of X that removes the smallest number of values from the domains of those variables which are not in the current assignment

[Rationale: Since only one value will eventually be assigned to X, pick the least-constraining value first, since it is the most likely one not to lead to an invalid assignment]

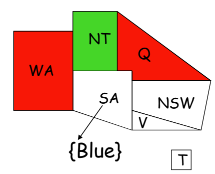
[Note: Using this heuristic requires performing a forward-checking step for every value, not just for the selected value]

Map Coloring



- Q's domain has two remaining values: Blue and Red
- Assigning Blue to Q would leave 0 value for SA, while assigning Red would leave 1 value

Map Coloring



- Q's domain has two remaining values: Blue and Red
- Assigning Blue to Q would leave 0 value for SA, while assigning Red would leave 1 value

→ So, assign Red to Q

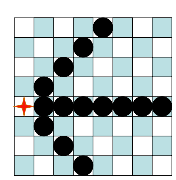
Constraint Propagation …

… is the process of determining how the constraints and the possible values of one variable affect the possible values of other variables

It is an important form of "least-commitment" reasoning

Forward checking is only one simple form of constraint propagation

When a pair (X←v) is added to assignment A do:
  For each variable Y not in A do:
    For every constraint C relating Y to variables in A do:
      Remove all values from Y's domain that do not satisfy C

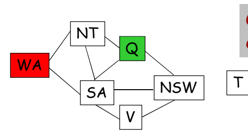


Forward Checking in Map Coloring

Empty set: the current assignment {(WA ← R), (Q ← G), (V ← B)} does not lead to a solution

WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

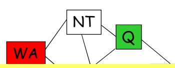
Forward Checking in Map Coloring



Contradiction that forward checking did not detect

WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

Forward Checking in Map Coloring



Contradiction that forward checking did not detect

Detecting this contradiction requires a more powerful constraint propagation technique

WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

Constraint Propagation for Binary Constraints

REMOVE-VALUES(X,Y)

1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (x,y) is satisfied then
     a. Remove v from Y's domain
     b. removed ← true
3. Return removed

Constraint Propagation for Binary Constraints

AC3

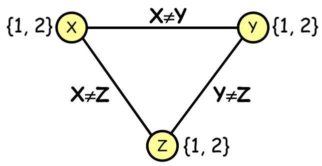
1. contradiction ← false
2. Initialize queue Q with all variables
3. While Q ≠ ∅ and ¬contradiction do
   a. X ← Remove(Q)
   b. For every variable Y related to X by a constraint do
      - If REMOVE-VALUES(X,Y) then
        i. If Y's domain = ∅ then contradiction ← true
        ii. Insert(Y,Q)

Complexity Analysis of AC3

- n = number of variables
- d = size of initial domains
- s = maximum number of constraints involving a given variable (s ≤ n-1)
- Each variables is inserted in Q up to d times
- REMOVE-VALUES takes $O(d^2)$ time
- AC3 takes $O(n\ s\ d^3)$ time
- Usually more expensive than forward checking

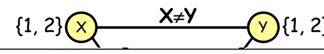
Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among binary constraints



Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among binary constraints



REMOVE-VALUES(X,Y)

1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (X,Y) is satisfied then
     a. Remove v from Y's domain
     b. removed ← true
3. Return removed

Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among

RE
1.
2.
3.

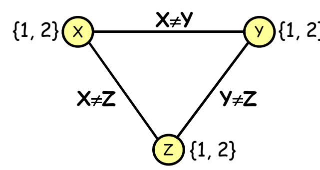
REMOVE-VALUES(X,Y,Z)

1. removed ← false
2. For every value w in the domain of Z do
   - If there is no pair (u,v) of values in the domains of X and Y verifying the constraint on (X,Y) such that the constraints on (X,Z) and (Y,Z) are satisfied then
     a. Remove w from Z's domain
     b. removed ← true
3. Return removed

31

Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among binary constraints



- Not all constraints are binary

Tradeoff

Generalizing the constraint propagation algorithm increases its time complexity

→Tradeoff between backtracking and constraint propagation

A good tradeoff is often to combine backtracking with forward checking and/or AC3

Modified Backtracking Algorithm with AC3

CSP-BACKTRACKING(A, var-domains)

1. If assignment A is complete then return A
2. var-domains ← AC3(var-domains)
3. If a variable has an empty domain then return failure
4. X ← select a variable not in A
5. D ← select an ordering on the domain of X
6. For each value v in D do
   a. Add (X←v) to A
   b. var-domains ← forward checking(var-domains, X, v, A)
   c. If a variable has an empty domain then return failure
   d. result ← CSP-BACKTRACKING(A, var-domains)
   e. If result ≠ failure then return result
7. Return failure

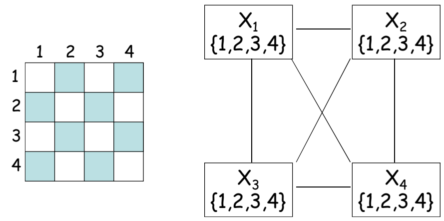
Modified Backtracking Algorithm with AC3

CSP-BACKTRACKING(A, var-domains)

1. If assignment A is complete then return A
2. var-domains ← AC3(var-domains)
3. If a variable has an empty domain then return failure
4. X ← select a variable not in A
5. D ← select an ordering on the domain of X
6. For each value v in D do
   a. Add (X←v) to A
   b. var-domains ← forward checking(var-domains, X, v, A)
   c. If a variable has an empty domain then return failure
   d. result ← CSP-BACKTRACKING(A, var-domains)

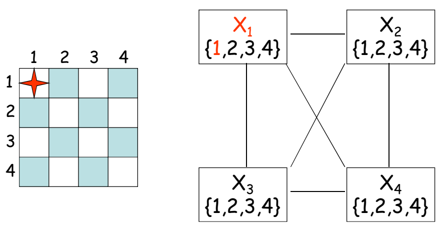
AC3 and forward checking prevent the backtracking algorithm from committing early to some values

A Complete Example: 4-Queens Problem



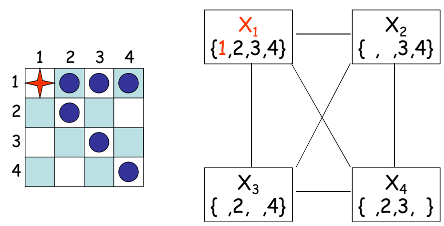
1) The modified backtracking algorithm starts by calling AC3, which removes no value

4-Queens Problem



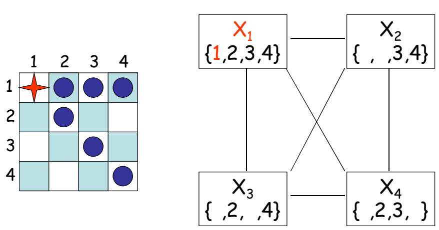
2) The backtracking algorithm then selects a variable and a value for this variable. No heuristic helps in this selection. X_1 and the value 1 are arbitrarily selected

4-Queens Problem



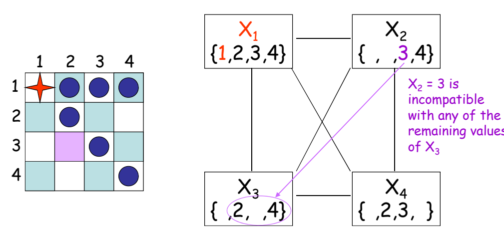
3) The algorithm performs forward checking, which eliminates 2 values in each other variable's domain

4-Queens Problem



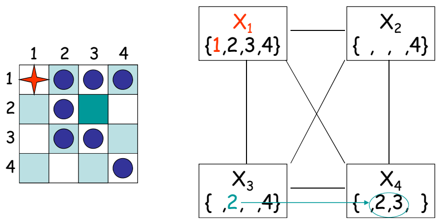
4) The algorithm calls AC3

4-Queens Problem



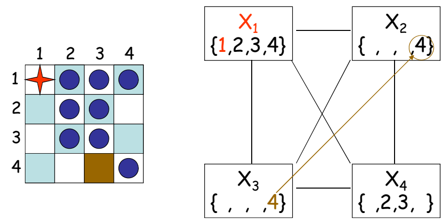
4) The algorithm calls AC3, which eliminates 3 from the domain of X_2

4-Queens Problem



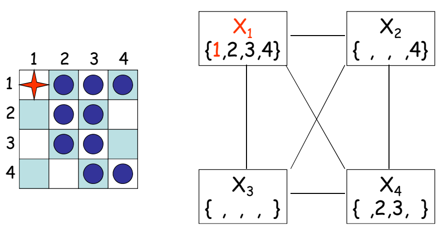
4) The algorithm calls *AC3*, which eliminates 3 from the domain of X_2, and 2 from the domain of X_3

4-Queens Problem



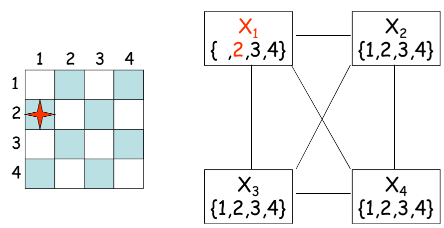
4) The algorithm calls *AC3*, which eliminates 3 from the domain of X_2, and 2 from the domain of X_3, and 4 from the domain of X_3

4-Queens Problem



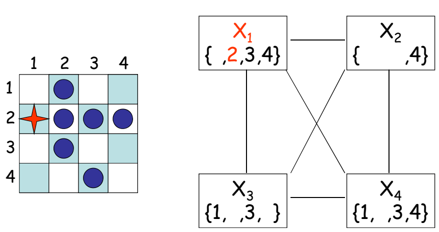
5) The domain of X_3 is **empty** → backtracking

4-Queens Problem



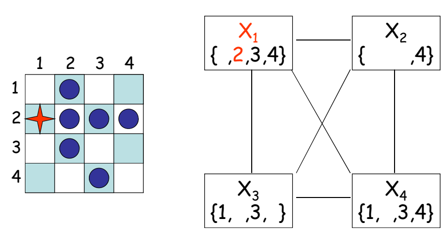
6) The algorithm removes 1 from X_1's domain and assign 2 to X_1

4-Queens Problem



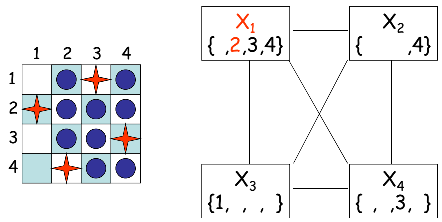
7) The algorithm performs forward checking

4-Queens Problem



8) The algorithm calls *AC3*

4-Queens Problem



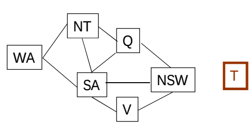
8) The algorithm calls AC3, which reduces the domains of X_3 and X_4 to a single variable

Dependency-Directed Backtracking

- Assume that CSP-BACTRACKING has successively picked values for k-1 variables: X_1, then X_2, …, then X_{k-1}
- It then tries to assign a value to X_k, but each remaining value in X_k's domain leads to a contradiction, that is, an empty domain for another variable
- Chronological backtracking consists of returning to X_{k-1} (called the "most recent" variable) and picking another value for it
- Instead, dependency-directed backtracking consists of:
  1. Computing the conflict set made of all the variables involved in the constraints that have led either to removing values from X_k's domain or to the empty domains which have caused the algorithm to reject each remaining value of X_k
  2. Returning to the most recent variable in the conflict set

Exploiting the Structure of CSP

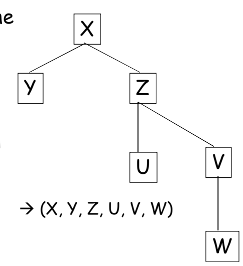
If the constraint graph contains several components, then solve one independent CSP per component



Exploiting the Structure of CSP

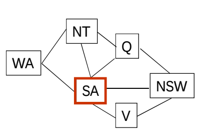
If the constraint graph is a tree, then :

1. Order the variables from the root to the leaves
   → (X_1, X_2, …, X_n)
2. For j = n, n-1, …, 2 call REMOVE-VALUES(X_j, X_i) where Xi is the parent of X_j
3. Assign any valid value to X_1
4. For j = 2, …, n do
   Assign any value to X_j consistent with the value assigned to X_i, where X_i is the parent of X_j



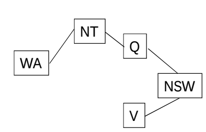
Exploiting the Structure of CSP

Whenever a variable is assigned a value by the backtracking algorithm, propagate this value and remove the variable from the constraint graph



Exploiting the Structure of CSP

Whenever a variable is assigned a value by the backtracking algorithm, propagate this value and remove the variable from the constraint graph



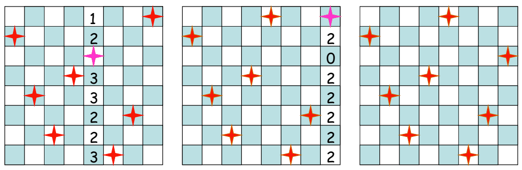
If the graph becomes a tree, then proceed as shown in previous slide

Finally, don't forget local search

(see slides on Heuristic Search)

Repeat n times:

1) Pick an initial state S at random with one queen in each column
2) Repeat k times:
   a) If GOAL?(S) then return S
   b) Pick an attacked queen Q at random
   c) Move Q it in its column to minimize the number of attacking queens is minimum → new S [min-conflicts heuristic]
3) Return failure



Applications of CSP

- CSP techniques are widely used
- Applications include:
  - Crew assignments to flights
  - Management of transportation fleet
  - Flight/rail schedules
  - Job shop scheduling
  - Task scheduling in port operations
  - Design, including spatial layout design
  - Radiosurgical procedures

Constraint Propagation

- The following shows how a more complicated problem (with constraints among 3 variables) can be solved by constraint satisfaction.
- It is merely an example from some old Stanford Slides just to see how it works…

Semi-Magic Square

- 9 variables X_1, …, X_9, each with domain {1, 2, 3}
- 7 constraints

X_1	X_2	X_3	This row must sum to 6
X_4	X_5	X_6	This row must sum to 6
X_7	X_8	X_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

Semi-Magic Square

1, 2, 3	1, 2, 3	1, 2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	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

Semi-Magic Square

- We select the value 1 for X_1
- Forward checking can't eliminate any value
  [only one variable has been assigned a value and every constraint involves 3 variables]

1	1, 2, 3	1, 2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	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

C.P. in Semi-Magic Square

- But the only remaining valid triplets for X_1, X_2, and X_3 are (1, 2, 3) and (1, 3, 2)

1	1, 2, 3	1, 2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	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

C.P. in Semi-Magic Square

- But the only remaining valid triplets for X_1, X_2, and X_3 are (1, 2, 3) and (1, 3, 2)
- So, X_2 and X_3 can no longer take the value 1

1	2, 3	2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	This row must sum to 6
1, 2, 3	1, 2, 3	1, 2, 3	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

C.P. Semi-Magic Square

- In the same way, X_4 and X_7 can no longer take the value 1

1	2, 3	2, 3	This row must sum to 6
2, 3	1, 2, 3	1, 2, 3	This row must sum to 6
2, 3	1, 2, 3	1, 2, 3	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

C.P. Semi-Magic Square

- In the same way, X_4 and X_7 can no longer take the value 1
- ... nor can X_5 and X_9

1	2, 3	2, 3	This row must sum to 6
2, 3	2, 3	1, 2, 3	This row must sum to 6
2, 3	1, 2, 3	2, 3	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

C.P. Semi-Magic Square

- Consider now a constraint that involves variables whose domains have been reduced

1	2, 3	2, 3	This row must sum to 6
2, 3	2, 3	1, 2, 3	This row must sum to 6
2, 3	1, 2, 3	2, 3	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

C.P. Semi-Magic Square

- For instance, take the 2nd column: the only remaining valid triplets are (2, 3, 1) and (3, 2, 1)

1	2, 3	2, 3	This row must sum to 6
2, 3	2, 3	1, 2, 3	This row must sum to 6
2, 3	1, 2, 3	2, 3	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

Semi-Magic Square

- For instance, take the 2nd column: the only remaining valid triplets are (2, 3, 1) and (3, 2, 1)
- So, the remaining domain of X_8 is {1}

1	2, 3	2, 3	This row must sum to 6
2, 3	2, 3	1, 2, 3	This row must sum to 6
2, 3	1	2, 3	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

C.P. Semi-Magic Square

- In the same way, we can reduce the domain of X_6 to {1}

1	2, 3	2, 3	This row must sum to 6
2, 3	2, 3	1	This row must sum to 6
2, 3	1	2, 3	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

C.P. Semi-Magic Square

- We can't eliminate more values
- Let us pick $X_2 = 2$

1	2, 3	2, 3	This row must sum to 6
2, 3	2, 3	1	This row must sum to 6
2, 3	1	2, 3	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

C.P. Semi-Magic Square

- Constraint propagation reduces the domains of $X_3, \ldots, X_9$ to a single value
- Hence, we have a solution

1	2	3	This row must sum to 6
2	3	1	This row must sum to 6
3	1	2	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