IS-ZC444: ARTIFICIAL INTELLIGENCE

Lecture-10: ALPHA-BETA Pruning, Constraint Satisfaction

Dr. Kamlesh Tiwari
Assistant Professor
Department of Computer Science and Information Systems,
BITS Pilani, Pilani, Jhunjhunu-333031, Rajasthan, INDIA

Nov 01, 2020 FLIPPED (WILP @ BITS-Pilani Jul-Nov 2020)

Recap: Adversarial Search (game)

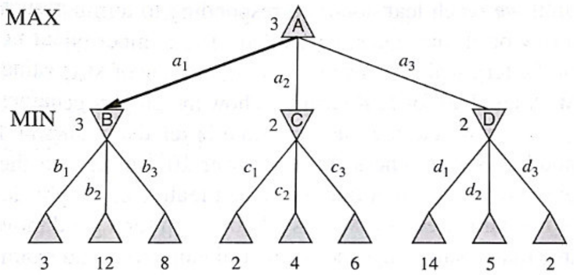
Agents having conflicting goals in competitive multiagent environment

- Deterministic, fully-observable, turn-taking, two-player, zero-sum
- Chess has roughly branching factor 35, moves 50 so tree search space is $35^{100} = 10^{154}$ however, graph has 10^{40} nodes
- Finding optimal move is infeasible but, needs an ability to decide

Game is between MAX and MIN (MAX moves first)

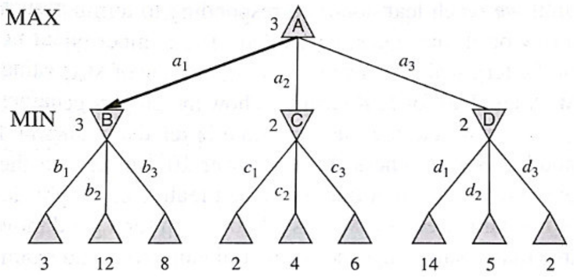
- S_0: the initial state
- PLAYER(s): defines which player has move to start
- ACTIONS(s): returns set of legal moves in a state
- RESULT(s, a):termination model defining result of a move
- TERMINAL_TEST(s): is true when game is over
- UTILITY(s, p): utility function defining reward (for chess +1,0,1/2)

Recap: Two half moves is one **ply**



[1] utility value for MAX of being in corresponding state (assuming then onwards both player play optimally)

Recap: Two half moves is one **ply**



Given the game tree, optimal strategy can be determined from **minimax value** of each node.

$$MINIMAX(s) = \begin{cases} UTILITY(s) & \text{if } TERMINAL\_TEST(s) \\ argmax_{a \in Actions(s)} MINIMAX(RESULT(s, a)) & \text{if } PLAYER(s) = MAX \\ argmin_{a \in Actions(s)} MINIMAX(RESULT(s, a)) & \text{if } PLAYER(s) = MIN \end{cases}$$

Action a_1 is the optimal choice [1]

(essentially optimizing worst-case outcome for MAX)

[1] utility value for MAX of being in corresponding state (assuming then onwards both player play optimally)

MINIMAX Algorithm

Returns the action corresponding to best move

function MINIMAX-DECISION(*state*) **returns** *an action*
  return arg max$_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

Recursion proceeds all the way down to the leaves. Time complexity $O(b^m)$ that is impractical but provides a basis of solution.

ALPHA-BETA Pruning

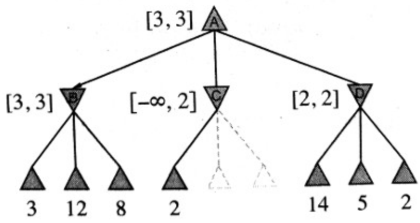
- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

ALPHA-BETA Pruning

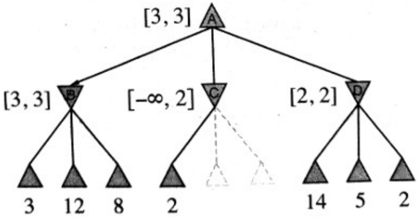
- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.



Consider two unevaluated successor of node C have value x and y

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

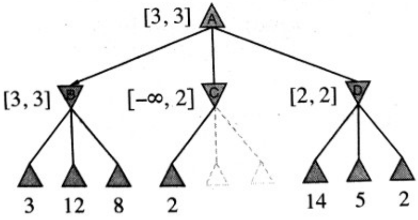


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

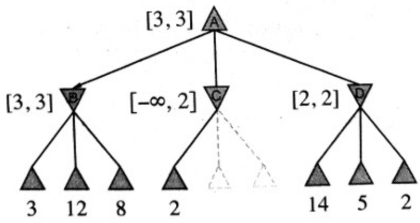


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))
= max(3, min(2,x,y), 2)

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

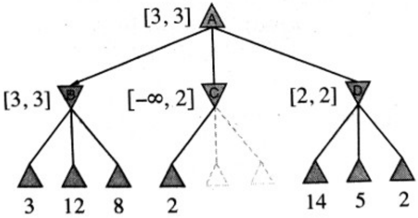


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))
= max(3, min(2,x,y), 2)
= max(3, z, 2) where z=min(2,x,y) $\leq$ 2

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

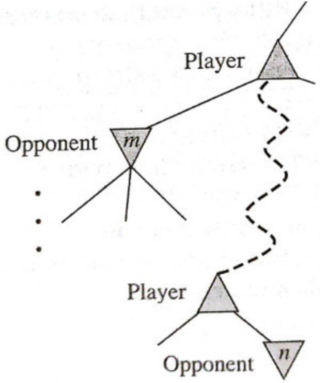


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))
= max(3, min(2,x,y), 2)
= max(3, z, 2) where z=min(2,x,y)$\leq$ 2
= 3

ALPHA-BETA Pruning

- Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtree rather than just leaves.



If *m* is better than *n* for player then we would never go to *n* in play

α	$=$	value of best choice (highest) found so far for MAX
β	$=$	value of best choice (lowest) found so far for MIN

ALPHA-BETA Pruning

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

ALPHA-BETA Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Order matters. So, examine likely to be best successor first.

ALPHA-BETA Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Order matters. So, examine likely to be best successor first.

Is it possible?

ALPHA-BETA Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

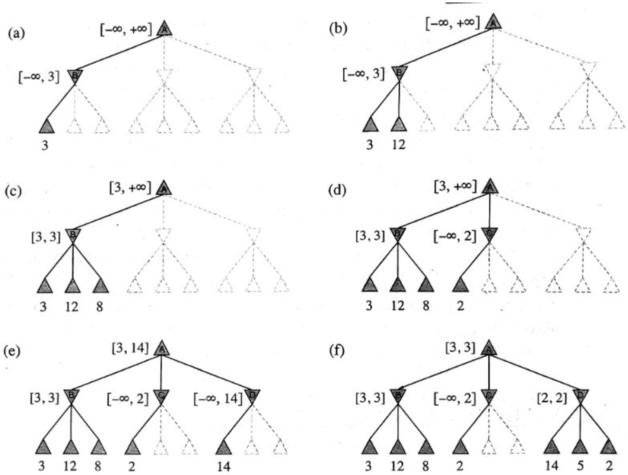
---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Order matters. So, examine likely to be best successor first.

Is it possible?
No

In-action: ALPHA-BETA Pruning



Move Ordering

- With perfect ordering we need to examine only $O(b^{m/2})$ nodes
- When successors are examined in random order it needs to examine $O(b^{3m/4})$ nodes
- In chess, a strategy that capture $\rightarrow$ threat $\rightarrow$ forward $\rightarrow$ backward, gets you to within about a factor of 2 of the best case $O(b^{m/2})$
- Try first the move that were found useful in past (killer move)
- Iterative deepening could help (adds constant fraction time)
- Hash table of previously seen positions (**transposition table**) can restrict re-computation of states

Imperfect Real-time Decision

- Generating entire search space is overkill
- Covering till large depth is not possible
- Idea is to cutoff the search earlier

$$H\text{-}MINIMAX(s,d) = \begin{cases} Eval(s) & \text{if } CUTOFF\_TEST(s,d) \\ argmax_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = MAX \\ argmin_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = MIN \end{cases}$$

Evaluation Function

- Returns the estimate of expected utility
- Performance would deeply depend on it
- Eval(win) $\geq$ Eval(draw) $\geq$ Eval(lose)
- Computation should be quick
- Value should relate to actual chance of winning
- Can use features (Number of pawns, Number of queens, ...)
- Can have various categories (all pawn vs one pawn, etc) Suppose experience suggests 72% of states encountered in the two-pawn vs. one pawn category lead to win, 20% lose, and 8% in draw. then eval could be

$$0.72 \times 1 + 0.20 \times 0 + 0.08 \times 0.5 = 0.76$$

- **Weighted linear function** are also possible

$$Eval(s) = w_1 \times f_1(s) + w_2 \times f_2(s) + w_3 \times f_3(s) + \ldots$$

Important Considerations

- **Cutting off search**: having fixed depth may not be a good idea. **Quiescent** states are unlikely to have wild swing so expand till it. **Horizon effect** is difficult to eliminate so **singular extension**[2] could be used
- **Forward Pruning**: cutting off the some search without further consideration. (we do not evaluate all possibilities **beam search**, we only use some in our mind) This may be fatal.
- **Search Vs Lookup** We have some policy on how to start and finish well. Use them as lookup

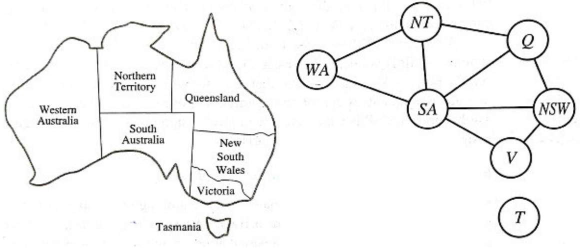
[2] can I take a previous good move

Constraint Satisfaction

- A Constraint Satisfaction Problem (CSP) has three components
  1. X as a set of variables $\{X_1, X_2, X_3, ..., X_n\}$
  2. D as a set of domains $\{D_1, D_2, D_3, ..., D_n\}$
  3. C set of constraints that specify allowable combinations of values
- Each D_i contains allowable set of values $\{v_1, v_2, v_3, ..., v_k\}$ for X_i
- Each C_i contains a pair $<$scope, relation$>$
  such as $<(X_1, X_3), X_1 \neq X_3>$
- To solve CSP it needs state space and a notion of solution
- An assignment of variables such as $(X_1 = v_1, X_2 = v_2, ..., X_n = v_n)$ that does not violates any constraints is called **consistent** or legal solution.
- Our target to have complete assignment that is consistent.

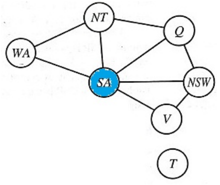
Example: Map Coloring

Given three colors {red,green,blue}, can you color following map such that no two neighboring region have same color.



- X = {WA, NT, Q, NSW, V, SA, T}
- C = {SA≠WA, SA≠NT, SA≠Q, SA≠NSW, SA≠V, WA≠NT, NT≠Q, Q≠NSW, NSW≠V}
- There are many solution to the problem
- It is helpful to visualize as a **constraint graph** (node: variable, edge: participate in constraint)

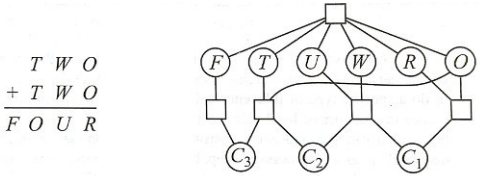
Constraint Propagation



- Once you choose (SA=blue)
- you can conclude that none of its five members could take the color blue
- Without taking advantage of constraint propagation, search needs to consider $3^5 = 243$ assignments for the five neighbors.
- With constraint propagation, it needs $2^5 = 32$ only
- 87% reduction

Example: Cryptarithmethc Puzzle

Consider following addition (we have to find digits)



- Each letter represents a different digit
- We need Alldiff{F,T,U,W,R,O} and
- Additional constraints are

$$\begin{aligned}
O + O &= R + 10 \times C_1 \\
C_1 + W + W &= U + 10 \times C_2 \\
C_2 + T + T &= O + 10 \times C_3 \\
C_3 &= F
\end{aligned}$$

Is it so 234+234=0468?

Preferential Constraints

- Many real world CSPs include **preference constraints**
- Indicating some solution are preferred over other
- Consider university class scheduling problem
  - Apart from absolute constraints such as no professor could simultaneously teach two classes
  - There are some preferential constraints such as Prof. A prefer teaching in morning whereas Prof. B prefer teaching on evening.
  - A solution that schedules Prof. A in evening and Prof. B in morning is still ok
  - But, we do not prefer it
- Such problems are sometimes called **constraint optimization problem** (COP)

Inference in CSP

- In CSP, an algorithm can either
  1. Search or
  2. Do inference: constraint propagation (that reduces number of legal values for another variable)

Enforcing **local consistency** in each part of the graph can cause inconsistency elimination throughout the graph.

- **Node consistency:** if all the values in variable's domain satisfy the variable's unary constraints [3]. It is always possible to eliminate all unary constraints by applying *Node consistency*
- **Arc consistency:** X_i is arc consistent with X_j if for every value in current domain D_i there is some value in D_j satisfying binary constraint on (X_i, X_j). Note [4]

[3] If SA do not like green color then use {red,blue} instead of {red,green,blue}

[4] Generalization is possible: using more than two variables in a constraint

AC-3 Algorithm

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Takes $O(cd^3)$ time in worst case

Inference in CSP: Path consistency

Arc consistency can help if some domain becomes empty, or size of every domain reduces to 1

- **Path consistency:** two variable set $\{X_i, X_j\}$ is path consistent wrt X_m if for every $\{X_i = a, X_j = b\}$ consistent with constraints on $\{X_i, X_j\}$ there is an assignment to X_m that satisfies constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$
- **K-consistency:** for any set of $k - 1$ variables and for any constraint assignment to those variables, a consistent value can always be assigned to any k^{th} variable.
  A CSP is strongly k-consistent is it is k-consistent, k-1-consistent, k-2-consistent, ..., 1-consistent [5]
- **Global Constraints:** like *alldiff*. If all m variable involved in alldiff have only n possible values where $m > n$ then there is no solution.

[5] Finding such considition is hard

Example: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

Example: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Example: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

$$alldiff\{A1, A2, A3, A4, A5, A6, A7, A8, A9\}$$
$$alldiff\{B1, B2, B3, B4, B5, B6, B7, B8, B9\}$$
$$\vdots$$
$$alldiff\{A1, A2, A3, B1, B2, B3, C1, C2, C3\}$$
$$\vdots$$

Backtracking Search for CSP

- When inference only do not work, use search
- CSP with n variable and d domain size can have branching factor nd at top level. Then $(n-1)d$ in next level and so on.
- Tree with $!n.d^n$ leaves get generated (however valid assignments are only d^n)
- It is why we have ignored **commutativity** [6]
- So consider single variable at a node.
- Now we need to **backtrack**, if no legal value is left for assignment

[6] No effect of assignment order

Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
           add inferences to assignment
           result ← BACKTRACK(assignment, csp)
           if result ≠ failure then
              return result
     remove {var = value} and inferences from assignment
  return failure
```

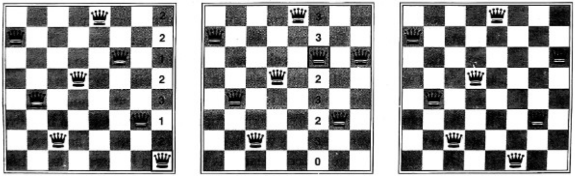
Backtracking Search

- Which variable to choose next? **minimum remaining value** [7] or "fail first"
- **Degree heuristic**, choose one which is involved in many constraints
- Which value to choose? **least constrained value**
- **Forward checking**: interleaving search and inference would help. Whenever a variable X is assigned, forward checking establishes arc consistency for it.
- **Intelligent Backtracking:** Some time it is needed to backtrack upward more than a single step to resolve the inconsistency. Conflicting set is used to find most suitable node.

[7] chose whose domain have fewer entries

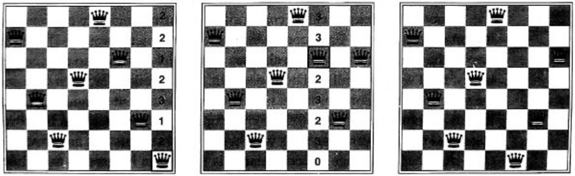
Local Search for CSP

Complete state space formulation can also be used for search



Local Search for CSP

Complete state space formulation can also be used for search



```
function MIN-CONFLICTS(csp, max_steps) returns a solution r failure
   inputs: csp, a constraint satisfaction problem
           max_steps, the number of steps allowed before giving up

   current ← an initial complete assignment for csp
   for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen conflicted variable from csp.VARIABLES
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
   return failure
```

Local Search for CSP

- **Min-conflict:** can solve Million Queen problem in 50 steps on an average
- **Tabu-search:** keeps a small list of recently visited states and forbidding algorithm to return to these states.
- **Constraint weighting:** starting from weight 1 for each variable; in each step, algorithm choses variable/value such that sum of weights is minimized. Weights of violated variables are incremented.
- Big advantage that the local search can be implemented in online environment.[8]

[8] Airline wants to repair its schedule with minimum changes.

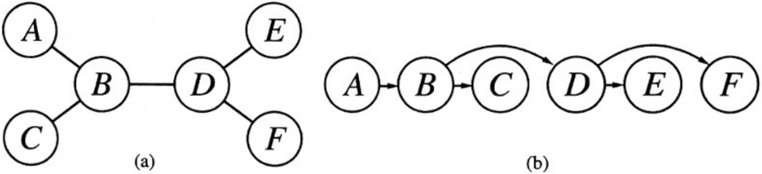
Exploit Problem Structure

Sometime structure of the problem could help to find solution

- **Independent Subproblems:** see that Tasmania is not connected to mainland in Australia map.

  $$\text{Compare } O(d^c n/c) \text{ with } O(d^n) \text{ it is linear}$$

  where each sub problem has c variables
- **Tree structured CSP** is solvable in linear time $O(nd^2)$ with topological sort

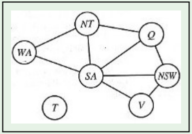


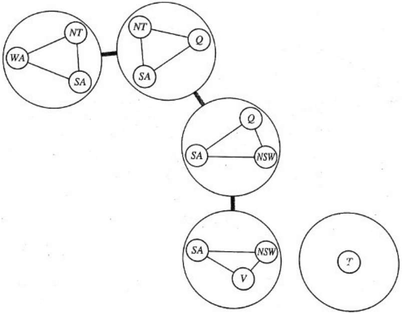
  - **Cut set conditioning:** Assign few, to get tree from remaining vars $O(d^c(n-c)d^2)$
  - Another approach is **tree decomposition**

Tree Decomposition for CSP

Divide the problem in sub-problems

- Every variable in original problem appears at least one of the subproblems
- If two variables are connected by a constraint in original problem, then they must appear together in at least one of the subproblem
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems





Thank You!

Thank you very much for your attention!

Queries ?

(Reference[9])

[9] 1) Book - *AIMA*, ch-06, Russell and Norvig.