Mid-term Review
Chapters 2-6

• Review Agents (2.1-2.3)
• Review State Space Search
  • Problem Formulation (3.1, 3.3)
  • Blind (Uninformed) Search (3.4)
  • Heuristic Search (3.5)
  • Local Search (4.1, 4.2)
• Review Adversarial (Game) Search (5.1-5.4)
• Review Constraint Satisfaction (6.1-6.4)
• Please review your quizzes and old CS-271 tests
  • At least one question from a prior quiz or old CS-271 test will appear on the mid-term (and all other tests)
Review Agents
Chapter 2.1-2.3

• Agent definition (2.1)

• Rational Agent definition (2.2)
  – Performance measure

• Task environment definition (2.3)
  – PEAS acronym
Agents

• An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators

  Human agent:
eyes, ears, and other organs for sensors; hands, legs, mouth, and other body parts for actuators

• Robotic agent:
cameras and infrared range finders for sensors; various motors for actuators
Rational agents

- **Rational Agent**: For each possible percept sequence, a rational agent should select an action that is *expected* to maximize its **performance measure**, based on the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

- **Performance measure**: An objective criterion for success of an agent's behavior.

- *E.g.*, performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.
Before we design an intelligent agent, we must specify its “task environment”:

**PEAS:**

- Performance measure
- Environment
- Actuators
- Sensors
PEAS

- Example: Agent = Part-picking robot

- **Performance measure:** Percentage of parts in correct bins

- **Environment:** Conveyor belt with parts, bins

- **Actuators:** Jointed arm and hand

- **Sensors:** Camera, joint angle sensors
Review State Space Search
Chapters 3-4

• Problem Formulation (3.1, 3.3)
• Blind (Uninformed) Search (3.4)
  • Depth-First, Breadth-First, Iterative Deepening
  • Uniform-Cost, Bidirectional (if applicable)
  • Time? Space? Complete? Optimal?
• Heuristic Search (3.5)
  • A*, Greedy-Best-First
• Local Search (4.1, 4.2)
  • Hill-climbing, Simulated Annealing, Genetic Algorithms
  • Gradient descent
Problem Formulation

A problem is defined by five items:

initial state e.g., "at Arad"

actions
  - Actions(X) = set of actions available in State X

transition model
  - Result(S,A) = state resulting from doing action A in state S

goal test, e.g., x = "at Bucharest", Checkmate(x)

path cost (additive, i.e., the sum of the step costs)
  - c(x,a,y) = step cost of action a in state x to reach state y
    - assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state
Vacuum world state space graph

- **states?** discrete: dirt and robot locations
- **initial state?** any
- **actions?** *Left, Right, Suck*
- **transition model?** as shown on graph
- **goal test?** no dirt at all locations
- **path cost?** 1 per action
Implementation: states vs. nodes

- A **state** is a (representation of) a physical configuration.

- A **node** is a data structure constituting part of a search tree.
- A node contains info such as:
  - state, parent node, action, path cost \( g(x) \), depth, etc.

- The **Expand** function creates new nodes, filling in the various fields using the **Actions** \( (S) \) and **Result** \( (S,A) \) functions associated with the problem.
Tree search vs. Graph search
Review Fig. 3.7, p. 77

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Test is often implemented as a hash table.
Search strategies

• A search strategy is defined by the order of node expansion

• Strategies are evaluated along the following dimensions:
  – completeness: does it always find a solution if one exists?
  – time complexity: number of nodes generated
  – space complexity: maximum number of nodes in memory
  – optimality: does it always find a least-cost solution?

• Time and space complexity are measured in terms of
  – $b$: maximum branching factor of the search tree
  – $d$: depth of the least-cost solution
  – $m$: maximum depth of the state space (may be $\infty$)
  – $l$: the depth limit (for Depth-limited complexity)
  – $C^*$: the cost of the optimal solution (for Uniform-cost complexity)
  – $\epsilon$: minimum step cost, a positive constant (for Uniform-cost complexity)
Blind Search Strategies (3.4)

- Depth-first: Add successors to front of queue
- Breadth-first: Add successors to back of queue
- Uniform-cost: Sort queue by path cost $g(n)$
- Depth-limited: Depth-first, cut off at limit
- Iterated-deepening: Depth-limited, increasing
- Bidirectional: Breadth-first from goal, too.
### Summary of algorithms

**Fig. 3.21, p. 91**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening DLS</th>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes[a]</td>
<td>Yes[a,b]</td>
<td>No</td>
<td>No</td>
<td>Yes[a]</td>
<td>Yes[a,d]</td>
</tr>
<tr>
<td>Time</td>
<td>(O(b^d))</td>
<td>(O(b^{\lceil 1+C*/\varepsilon \rceil}))</td>
<td>(O(b^m))</td>
<td>(O(b^l))</td>
<td>(O(b^d))</td>
<td>(O(b^{d/2}))</td>
</tr>
<tr>
<td>Space</td>
<td>(O(b^d))</td>
<td>(O(b^{\lceil 1+C*/\varepsilon \rceil}))</td>
<td>(O(bm))</td>
<td>(O(bl))</td>
<td>(O(bd))</td>
<td>(O(b^{d/2}))</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes[c]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes[c]</td>
<td>Yes[c,d]</td>
</tr>
</tbody>
</table>

There are a number of footnotes, caveats, and assumptions. See Fig. 3.21, p. 91.

[a] complete if \(b\) is finite

[b] complete if step costs \(\geq \varepsilon > 0\)

[c] optimal if step costs are all identical

(also if path cost non-decreasing function of depth only)

[d] if both directions use breadth-first search

(also if both directions use uniform-cost search with step costs \(\geq \varepsilon > 0\))

Generally the preferred uninformed search strategy
Heuristic function (3.5)

- **Heuristic:**
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - “using rules of thumb to find answers”

- **Heuristic function h(n)**
  - Estimate of (optimal) cost from n to goal
  - Defined using only the state of node n
  - $h(n) = 0$ if n is a goal node
  - Example: straight line distance from n to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate – actual state-space distance can be higher

- Provides problem-specific knowledge to the search algorithm
Greedy best-first search

- $h(n) = \text{estimate of cost from } n \text{ to goal}$
  - e.g., $h(n) = \text{straight-line distance from } n \text{ to Bucharest}$

- Greedy best-first search expands the node that appears to be closest to goal.
  - Sort queue by $h(n)$

- Not an optimal search strategy
  - May perform well in practice
A* search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n) = \text{cost so far to reach } n$
  - $h(n) = \text{estimated cost from } n \text{ to goal}$
  - $f(n) = \text{estimated total cost of path through } n \text{ to goal}$
- A* search sorts queue by $f(n)$
- Greedy Best First search sorts queue by $h(n)$
- Uniform Cost search sorts queue by $g(n)$
Admissible heuristics

- A heuristic $h(n)$ is **admissible** if for every node $n$, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from $n$.
- An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- **Theorem**: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal
Consistent heuristics
(consistent => admissible)

- A heuristic is **consistent** if for every node $n$, every successor $n'$ of $n$ generated by any action $a$,

$$h(n) \leq c(n,a,n') + h(n')$$

- If $h$ is consistent, we have

\[
\begin{align*}
    f(n') &= g(n') + h(n') \\
    &= g(n) + c(n,a,n') + h(n') \\
    &\geq g(n) + h(n) = f(n) \\
    &\geq f(n)
\end{align*}
\]

  - i.e., $f(n)$ is non-decreasing along any path.

- **Theorem:**
  If $h(n)$ is consistent, A* using **GRAPH-SEARCH** is optimal

  It's the triangle inequality!

keeps all checked nodes in memory to avoid repeated states
Local search algorithms (4.1, 4.2)

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it.
- Very memory efficient (only remember current state)
Local Search Difficulties

- Problem: depending on initial state, can get stuck in local maxima
Hill-climbing search

• "Like climbing Everest in thick fog with amnesia"

function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
  neighbor ← a highest-valued successor of current
  if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
  current ← neighbor
Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but **gradually decrease** their frequency

```python
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
             schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] − VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability $e^{ΔE/T}$
```

Properties of simulated annealing search

- One can prove: If $T$ decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1 (however, this may take VERY long)
  - However, in any finite search space RANDOM GUESSING also will find a global optimum with probability approaching 1.

- Widely used in VLSI layout, airline scheduling, etc.
Genetic algorithms

• A successor state is generated by combining two parent states

• Start with \( k \) randomly generated states (population)

• A state is represented as a string over a finite alphabet (often a string of 0s and 1s)

• Evaluation function (fitness function). Higher values for better states.

• Produce the next generation of states by selection, crossover, and mutation
- Fitness function: number of non-attacking pairs of queens (min = 0, max = \( \frac{8 \times 7}{2} = 28 \))
- \( P(\text{child}) = \frac{24}{24+23+20+11} = 31\% \)
- \( P(\text{child}) = \frac{23}{24+23+20+11} = 29\% \) etc
Gradient Descent

- Assume we have some cost-function: $C(x_1, \ldots, x_n)$ and we want minimize over continuous variables $X_1, X_2, \ldots, X_n$

1. Compute the gradient: $\frac{\partial}{\partial x_i} C(x_1, \ldots, x_n) \quad \forall i$

2. Take a small step downhill in the direction of the gradient:
   
   $$x_i \rightarrow x'_i = x_i - \lambda \frac{\partial}{\partial x_i} C(x_1, \ldots, x_n) \quad \forall i$$

3. Check if $C(x_1, \ldots, x'_i, \ldots, x_n) < C(x_1, \ldots, x_i, \ldots, x_n)$

4. If true then accept move, if not reject.

5. Repeat.
Review Adversarial (Game) Search
Chapter 5.1-5.4

• Minimax Search with Perfect Decisions (5.2)
  – Impractical in most cases, but theoretical basis for analysis
• Minimax Search with Cut-off (5.4)
  – Replace terminal leaf utility by heuristic evaluation function
• Alpha-Beta Pruning (5.3)
  – The fact of the adversary leads to an advantage in search!
• Practical Considerations (5.4)
  – Redundant path elimination, look-up tables, etc.
Game tree (2-player, deterministic, turns)

MAX (X)

MIN (O)

MAX (X)

MIN (O)

TERMINAL

Utility

How do we search this tree to find the optimal move?
Games as Search

• Two players: MAX and MIN

• MAX moves first and they take turns until the game is over
  – Winner gets reward, loser gets penalty.
  – “Zero sum” means the sum of the reward and the penalty is a constant.

• Formal definition as a search problem:
  – Initial state: Set-up specified by the rules, e.g., initial board configuration of chess.
  – Player(s): Defines which player has the move in a state.
  – Actions(s): Returns the set of legal moves in a state.
  – Result(s,a): Transition model defines the result of a move.
  – (2nd ed.: Successor function: list of (move, state) pairs specifying legal moves.)
  – Terminal-Test(s): Is the game finished? True if finished, false otherwise.
  – Utility function(s,p): Gives numerical value of terminal state s for player p.
    • E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    • E.g., win (+1), lose (0), and draw (1/2) in chess.

• MAX uses search tree to determine next move.
An optimal procedure: The Min-Max method

Designed to find the optimal strategy for Max and find best move:

• 1. Generate the whole game tree, down to the leaves.

• 2. Apply utility (payoff) function to each leaf.

• 3. Back-up values from leaves through branch nodes:
  – a Max node computes the Max of its child values
  – a Min node computes the Min of its child values

• 4. At root: choose the move leading to the child of highest value.
Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX’s best move is $A_1$, and MIN’s best reply is $A_{11}$. 
Two-Ply Game Tree
Two-Ply Game Tree

MAX

MIN

A_{11} \quad A_{12} \quad A_{13} \quad 3

A_{21} \quad A_{22} \quad A_{23} \quad 2

A_{31} \quad A_{32} \quad A_{33} \quad 2

3 \quad 12 \quad 8 \quad 2

4 \quad 6

14 \quad 5 \quad 2
Two-Ply Game Tree

Minimax maximizes the utility for the worst-case outcome for max.
Pseudocode for Minimax Algorithm

\textbf{function} MINIMAX-DECISION(state) \textbf{returns} an action
\textbf{inputs:} state, current state in game
\textbf{return} \text{arg max}_{a \in \text{ACTIONS}(state)} \text{MIN-VALUE(\text{Result}(state,a))}

\textbf{function} MIN-VALUE(state) \textbf{returns} a utility value
\textbf{if} TERMINAL-TEST(state) \textbf{then return} UTILITY(state)
\nu \leftarrow +\infty
\textbf{for} a \textbf{ in ACTIONS(state) do}
\quad \nu \leftarrow \text{MIN}(\nu,\text{MIN-VALUE(\text{Result}(state,a)))}
\textbf{return} \nu

\textbf{function} MAX-VALUE(state) \textbf{returns} a utility value
\textbf{if} TERMINAL-TEST(state) \textbf{then return} UTILITY(state)
\nu \leftarrow -\infty
\textbf{for} a \textbf{ in ACTIONS(state) do}
\quad \nu \leftarrow \text{MAX}(\nu,\text{MIN-VALUE(\text{Result}(state,a)))}
\textbf{return} \nu
Static (Heuristic) Evaluation Functions

• An Evaluation Function:
  – Estimates how good the current board configuration is for a player.
  – Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent’s score from the player’s.
  – Othello: Number of white pieces - Number of black pieces
  – Chess: Value of all white pieces - Value of all black pieces

• Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

• If the board evaluation is X for a player, it’s -X for the opponent
  – “Zero-sum game”
Evaluation functions

For chess, typically \textit{linear} weighted sum of features

\[ \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

\text{e.g., } w_1 = 9 \text{ with } \\
\[ f_1(s) = \text{(number of white queens)} - \text{(number of black queens)}, \text{ etc.} \]
Cutting off search

\textbf{MinimaxCutoff} is identical to \textbf{MinimaxValue} except

1. \texttt{Terminal?} is replaced by \texttt{Cutoff}?
2. \texttt{Utility} is replaced by \texttt{Eval}

Does it work in practice?

\[ b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4 \]

4-ply lookahead is a hopeless chess player!

4-ply \approx \text{human novice}
8-ply \approx \text{typical PC, human master}
12-ply \approx \text{Deep Blue, Kasparov}
General alpha-beta pruning

• Consider a node $n$ in the tree ---

• If player has a better choice at:
  – Parent node of $n$
  – Or any choice point further up

• Then $n$ will never be reached in play.

• Hence, when that much is known about $n$, it can be pruned.
Alpha-beta Algorithm

• Depth first search
  – only considers nodes along a single path from root at any time

\[ \alpha = \text{highest-value choice found at any choice point of path for MAX} \]
  (initially, \( \alpha = -\infty \))

\[ \beta = \text{lowest-value choice found at any choice point of path for MIN} \]
  (initially, \( \beta = +\infty \))

• Pass current values of \( \alpha \) and \( \beta \) down to child nodes during search.
• Update values of \( \alpha \) and \( \beta \) during search:
  – MAX updates \( \alpha \) at MAX nodes
  – MIN updates \( \beta \) at MIN nodes
• Prune remaining branches at a node when \( \alpha \geq \beta \)
When to Prune

• Prune whenever $\alpha \geq \beta$.

  – Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    • **Max nodes update alpha** based on children’s returned values.

  – Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.
    • **Min nodes update beta** based on children’s returned values.
Do DF-search until first leaf

\( \alpha, \beta, \text{initial values} \)

\( \alpha = -\infty \)
\( \beta = +\infty \)

\( \alpha, \beta, \text{passed to kids} \)

\( \alpha = -\infty \)
\( \beta = +\infty \)
MIN updates $\beta$, based on kids
MIN updates $\beta$, based on kids. No change.
MAX updates $\alpha$, based on kids.

$\alpha = 3$

$\beta = +\infty$

3 is returned as node value.
Alpha-Beta Example (continued)

\[
\alpha = 3 \\
\beta = +\infty
\]

\[
\alpha = 3 \\
\beta = +\infty
\]

\(\alpha, \beta, \text{passed to kids}\)

\[
\alpha = 3 \\
\beta = +\infty
\]
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

\[ MIN \text{ updates } \beta, \text{ based on kids.} \]
\[ \alpha = 3 \]
\[ \beta = 2 \]
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

\[ \alpha \geq \beta, \text{ so prune.} \]
Alpha-Beta Example (continued)

MAX updates $\alpha$, based on kids.
No change.

$\alpha = 3$
$\beta = +\infty$

2 is returned as node value.
Alpha-Beta Example (continued)

\[
\alpha = 3 \\
\beta = +\infty
\]

\(\alpha, \beta, \text{passed to kids}\)
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

MIN updates \( \beta \), based on kids.
\[ \alpha = 3 \]
\[ \beta = 14 \]
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

MIN updates \( \beta \), based on kids.
\[ \alpha = 3 \]
\[ \beta = 5 \]
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

2 is returned as node value.
Max calculates the same node value, and makes the same move!
Nondeterministic games in general

In nondeterministic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:

```
MAX

CHANCE

MIN
```

![Game Tree](image)
Algorithm for nondeterministic games

\textbf{EXPECTIMINIMAX} gives perfect play

Just like \textbf{MINIMAX}, except we must also handle chance nodes:

\ldots

\textbf{if} \ \texttt{state} \ \textbf{is a} \ \texttt{Max} \ \textbf{node} \ \textbf{then}
  \textbf{return} \ \text{the highest} \ \textbf{EXPECTIMINIMAX-Value} \ \text{of} \ \textbf{SUCCESSORS}(\texttt{state})
\textbf{if} \ \texttt{state} \ \textbf{is a} \ \texttt{Min} \ \textbf{node} \ \textbf{then}
  \textbf{return} \ \text{the lowest} \ \textbf{EXPECTIMINIMAX-Value} \ \text{of} \ \textbf{SUCCESSORS}(\texttt{state})
\textbf{if} \ \texttt{state} \ \textbf{is a} \ \texttt{chance} \ \textbf{node} \ \textbf{then}
  \textbf{return} \ \text{average of} \ \textbf{EXPECTIMINIMAX-Value} \ \text{of} \ \textbf{SUCCESSORS}(\texttt{state})
\ldots
Review Constraint Satisfaction
Chapter 6.1-6.4

• What is a CSP

• Backtracking for CSP

• Local search for CSPs
Constraint Satisfaction Problems

• What is a CSP?
  – Finite set of variables $X_1, X_2, \ldots, X_n$
  – Nonempty domain of possible values for each variable $D_1, D_2, \ldots, D_n$
  – Finite set of constraints $C_1, C_2, \ldots, C_m$
    • Each constraint $C_i$ limits the values that variables can take,
      • e.g., $X_1 \neq X_2$
    – Each constraint $C_i$ is a pair <scope, relation>
      • Scope = Tuple of variables that participate in the constraint.
      • Relation = List of allowed combinations of variable values.
        May be an explicit list of allowed combinations.
        May be an abstract relation allowing membership testing and listing.

• CSP benefits
  – Standard representation pattern
  – Generic goal and successor functions
  – Generic heuristics (no domain specific expertise).
CSPs --- what is a solution?

- A state is an assignment of values to some or all variables.
  - An assignment is complete when every variable has a value.
  - An assignment is partial when some variables have no values.

- **Consistent assignment**
  - assignment does not violate the constraints

- A **solution** to a CSP is a complete and consistent assignment.

- Some CSPs require a solution that maximizes an objective function.
CSP example: map coloring

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D_i=\{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors.
  - E.g. $WA \neq NT$
CSP example: map coloring

- Solutions are assignments satisfying all constraints, e.g.
  
  \{WA=\text{red},NT=\text{green},Q=\text{red},NSW=\text{green},V=\text{red},SA=\text{blue},T=\text{green}\}
Constraint graphs

- Constraint graph:
  - nodes are variables
  - arcs are binary constraints

- Graph can be used to simplify search
  e.g. Tasmania is an independent subproblem

  (will return to graph structure later)
Backtracking example
Minimum remaining values (MRV)

\[ var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE(}\text{VARIABLES}[csp],\text{assignment},csp) \]

- A.k.a. most constrained variable heuristic

- *Heuristic Rule*: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values
Degree heuristic for the initial variable

- **Heuristic Rule**: select variable that is involved in the largest number of constraints on other unassigned variables.

- Degree heuristic can be useful as a tie breaker.

- *In what order should a variable’s values be tried?*
Least constraining value for value-ordering

- Least constraining value heuristic

- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

[Diagram showing the least constraining value heuristic with maps of Australia demonstrating flexibility for subsequent variable assignments.]
Forward checking

- Can we detect inevitable failure early?
  - And avoid it later?

- Forward checking idea: keep track of remaining legal values for unassigned variables.

- Terminate search when any variable has no legal values.
• Assign \{WA=red\}

• Effects on other variables connected by constraints to WA
  – \textit{NT can no longer be red}
  – \textit{SA can no longer be red}
Forward checking

- Assign \(Q=green\)

- Effects on other variables connected by constraints with WA
  - \(NT\) can no longer be green
  - \(NSW\) can no longer be green
  - \(SA\) can no longer be green

- \textit{MRV heuristic} would automatically select \(NT\) or \(SA\) next
Forward checking

- If $V$ is assigned blue

- Effects on other variables connected by constraints with WA
  - NSW can no longer be blue
  - SA is empty

- FC has detected that partial assignment is inconsistent with the constraints and backtracking can occur.
Arc consistency

• An Arc $X \rightarrow Y$ is consistent if
  
  for every value $x$ of $X$ there is some value $y$ consistent with $x$

  (note that this is a directed property)

• Consider state of search after WA and Q are assigned:

  $SA \rightarrow NSW$ is consistent if
  
  $SA=blue$ and $NSW=red$
Arc consistency

- $X \rightarrow Y$ is consistent if
  for every value $x$ of $X$ there is some value $y$ consistent with $x$

- $NSW \rightarrow SA$ is consistent if
  $NSW=\text{red}$ and $SA=\text{blue}$
  $NSW=\text{blue}$ and $SA=???
Arc consistency

- Can enforce arc-consistency:
  Arc can be made consistent by removing blue from NSW

- Continue to propagate constraints....
  - Check $V \rightarrow NSW$
  - Not consistent for $V = \text{red}$
  - Remove red from $V$
Arc consistency

- Continue to propagate constraints....

- $SA \rightarrow NT$ is not consistent
  
  — and cannot be made consistent

- Arc consistency detects failure earlier than FC
Local search for CSPs

• Use complete-state representation
  – Initial state = all variables assigned values
  – Successor states = change 1 (or more) values

• For CSPs
  – allow states with unsatisfied constraints (unlike backtracking)
  – operators **reassign** variable values
  – hill-climbing with n-queens is an example

• Variable selection: randomly select any conflicted variable

• Value selection: *min-conflicts heuristic*
  – Select new value that results in a minimum number of conflicts with the other variables
Use of min-conflicts heuristic in hill-climbing.
Mid-term Review
Chapters 2-6

- Review Agents (2.1-2.3)
- Review State Space Search
  - Problem Formulation (3.1, 3.3)
  - Blind (Uninformed) Search (3.4)
  - Heuristic Search (3.5)
  - Local Search (4.1, 4.2)
- Review Adversarial (Game) Search (5.1-5.4)
- Review Constraint Satisfaction (6.1-6.4)
- Also, you should review your quizzes
  - At least one quiz question will appear on the mid-term