Exam Topics

• Search
• Game
• Constraint Satisfaction Problem
• Logic / First-order Logic
• Markov Decision Process
Exam Topics

- Search
- Game
- Constraint Satisfaction Problem
- Logic / First-order Logic
- Markov Decision Process
State Space Graph

- A state space graph is a mathematical representation of a search problem
  - Nodes are world configurations (abstracted)
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a search graph, each state occurs only once.
Search Tree

- Each node is corresponding to a state in the state space
- The start state is the root node
- Each edge is corresponding to an action
- Children nodes correspond to successors
- Each node encodes an entire path and correspond to plans to achieve that state
State Space Graphs vs. Search Trees

State Space Graph

Search Tree
Search Algorithms

- Input: Search problem
- Output: solution or an indication of failure
- Superimpose a search tree over the state space graph
How do we decide which node from the frontier to expand next?

Priority queue

Select node with minimum f(n) value

A node that represents a path to a goal

**Best-first Search**

The function `BEST-FIRST-SEARCH(problem, f)` returns a solution node or failure.

```python
node ← NODE(State=problem.INITIAL)
frontier ← a priority queue ordered by f, with node as an element
reached ← a lookup table, with one entry with key problem.INITIAL and value node

while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node

    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
            reached[s] ← child
            add child to frontier
    return failure
```

**f(n): evaluation function**

Select node with minimum f(n) value

A node that represents a path to a goal

**Expand**

The function `EXPAND(problem, node)` yields nodes.

```python
s ← node.STATE
for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
yield NODE(State=s', Parent=node, Action=action, Path-COST=cost)
```
Graph Search

fringe, frontier or border

function GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)

    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, STATE[node]) then return node

        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
        end
    end
Uninformed Search Strategies

No clue about how close a state is to the goal(s)

• Breath-First Search
• Depth-First Search
• Uniform-Cost Search
Breath-First Search

- All actions have the same cost
- Frontier is FIFO (First-In-First-Out) queue
- Completeness: If a solution exists, Yes
- Cost optimality: only if costs are all the same
- Time complexity: $O(b^d)$ \((1+b+b^2+b^3+\ldots +b^d)\)
- Space complexity: $O(b^d)$ (keeps every node in memory)
Depth-First Search

- Funds the “leftmost” solution in the search tree
- Frontier is LIFO (Last-In-First-Out) queue
- Completeness: No
- Cost optimality: Doesn’t care about costs, No
- Time complexity: $O(b^m)$
- Space complexity: $O(bm)$
Depth-limited Search

- DFS with a depth limit $L$
- Depth limit can be chosen based on knowledge of the problem
- Completeness: No
- Cost optimality: No
- Time complexity: $O(b^L)$
- Space complexity: $O(bL)$
Iterative Deepening Search

- Depth-limited search with depth $L=1$, then $L=2$, ...
- Until goal state found
- Completeness: Yes
- Cost optimality: Only if costs are all the same
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
Uniform-cost Search

- Dijkstra’s algorithm
- Actions have different costs
- Expand nodes in order of cost from the initial state
- Completeness: Yes
- Cost optimality: Yes
- Time complexity: $O(b^{1+[C^*/\epsilon]})$
- Space complexity: $O(b^{1+[C^*/\epsilon]})$
- $C^*$ is the cost of optimal path
- $\epsilon$ a lower bound on the cost of each action, with $\epsilon > 0$
Uniform-cost Search

- S->R: 80
- S->R->P: 177
- S->F: 99
- S->F->B: 310
- S->R->P->B: 278
Informed Search

• Uniformed search: can both complete and optimal, but can be very slow
• Informed search:
  – Uses domain-specific knowledge (hints)
  – Can find solutions more efficiently
  – Heuristic: Lead the search algorithm faster towards a goal state.
    • Implemented via Heuristic function: $h(n)$
Heuristics

- Heuristic function $h(n)$:
  - Estimated **cost** of the cheapest path from the state at node $n$ to a goal state
- Designed for a particular search problem
- **Examples**:
  - Manhattan distance: $|x_1 - x_2| + |y_1 - y_2|$
  - Euclidean distance: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Greedy Best-first Search

- Strategy: Expand a node with the lowest \( h(n) \) value
- Operates like UCS with a priority queue
  - Difference: use estimated forward cost, not computed backward cost
- Not optimal
- Could not find the best solution
- Could not find the solution (if use a very bad \( h(n) \))
- Worse-case: Just like a badly-guided DFS and exploring all the wrong areas
A* Search

- Combing UCS and Greedy
- \( f(n) = g(n) + h(n) \): lowest estimated cost of the path from \( n \) to \( G \)
- \( g(n) \) is the path cost from the initial state to node \( n \)
- Uses a priority queue
- A* combines the total backward cost and estimated forward cost (heuristic value)
- Effectively yielding an estimated total cost from start to goal
A* Search (Continued.)

- Completeness: Yes
- Cost optimality: **Depends on the properties of the heuristic**
- Properties of the heuristic:
  - **Admissibility**: admissible (optimistic) heuristic never overestimates the cost. \( 0 \leq h(n) \leq h^*(n) \)
  - **Consistency**: \( h(n) \leq h(n') + \text{cost}(n, a, n') \)
  - **Consistent heuristic is admissible**

- A* may or may not cost-optimal with an **inadmissible** heuristic
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - **Admissibility:** heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from } A \text{ to } G \]
  - **Consistency:** heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) \leq \text{cost}(A \text{ to } C) + h(C) \]
Admissible Heuristics

• admissible (optimistic) heuristic never overestimates the cost.

\[ 0 \leq h(n) \leq h^*(n) \]

• where \( h^*(n) \) is the true cost to a nearest goal

• Coming up with admissible heuristics is most of what’s involved in using A* in practice.
A* is cost-optimal with an admissible heuristic

- admissible (optimistic) heuristic never overestimates the cost.
  \[ 0 \leq h(n) \leq h^*(n) \]
- \( C^* \) is the optimal path cost
- If A* is not cost-optimal, \( f(n) > C^* \)
  - (There must be some node \( n \) is on the optimal path and is unexpanded)
- \( f(n) = g(n) + h(n) \)
- \( f(n) = g^*(n) + h(n) \)
  - \( n \) is on the optimal path
- \( f(n) \leq g^*(n) + h^*(n) \)
- \( f(n) \leq C^* \), Contradiction! So, A* is cost-optimal
Example: Consistency

- $h(B)$ is consistent
  - $h(A) \leq c(A, a, B) + h(B)$
  - $h(B) \leq c(B, a, A) + h(A)$
  - $h(C) \leq c(C, a, B) + h(B)$
  - $h(B) \leq c(B, a, C) + h(C)$
  - $h(A) = 7$, $h(B) = 5$, $h(C) = 5$
Inadmissible Heuristics and Weighted A*

- Weighted A* search: (Suboptimal Search)
  - \( f(n) = g(n) + W \times h(n), \ W > 1 \)
  - With a slightly costlier, but could search faster
  - Optimal path would not be found

- Cost of Weighted A* search: between \( C^* \) and \( W \times C^* \)

<table>
<thead>
<tr>
<th>A* search</th>
<th>( g(n) + h(n) )</th>
<th>( W = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS search</td>
<td>( g(n) )</td>
<td>( W = 0 )</td>
</tr>
<tr>
<td>Greedy search</td>
<td>( h(n) )</td>
<td></td>
</tr>
<tr>
<td>Weighted A* search</td>
<td>( g(n) + W \times h(n) )</td>
<td>( W &gt; 1 )</td>
</tr>
</tbody>
</table>
Exam Topics

• Search
• Game
• Constraint Satisfaction Problem
• Logic / First-order Logic
• Markov Decision Process
Definition of Two-player Zero-Sum Game

- **States:** S. Start at $S_0$ (initial state)
- **To-Move(s):** Move in state $s$
- **Action(s):** Set of legal moves in state $s$
- **Result(s, a):** The *transition model*. The state resulting from taking action $a$ in state $s$
- **Is-Terminal(s):** A *terminal test*. Game over?
  - Terminal states: States where the game has ended
- **Utility(s, p):** A *utility function* which defines the final number value to player $p$ when the game ends in terminal state $s$
Adversarial Game Tree

• Deterministic, zero-sum games:
  – Tic-tac-toe, chess, go, etc.
  – One player maximizes result
  – The other player minimizes result

• Example: Two-ply game tree:
  – A state-space search tree
  – Players (MAX, MIN) alternate turns (ply)
  – Compute minimax value of each node in the tree and select best (achievable) utility against adversary
Two-ply Game Tree

MAX

MIN

A

B

C

D

a_1

a_2

a_3

b_1

b_2

b_3

c_1

c_2

c_3

3

3

2

2

3

12

8

2

4

6

14

5

2

VT

VIRGINIA TECH.
**Minimax**

\[ \text{MINIMAX}(s) = \begin{cases} 
\text{UTILITY}(s) & \text{if } \text{THERMINAL-TEST}(s) \\
\max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} 
\end{cases} \]
Game Tree Pruning

- Chess: branching factor $b \approx 35$, ply $\approx 80$. $35^{80} \approx 10^{123}$
- We don’t need to explore the whole game tree!
Alpha-Beta Pruning

• General principle:
  – A player has a choice of moving to n
  – However, m’ is a better choice for player. Player choice m’
  – Or m is a better choice for player. Player choice m
  – Player will never move to n. So, we can prune it.
Alpha-Beta Pruning

- Can be applied to trees of any depth
- Prune entire subtrees or leaves
- $\alpha$: MAX’s best choice we have found along the path
- $\beta$: MIN’s best choice we have found along the path
Alpha-Beta Pruning Properties

- State can be pruned because it makes no difference to the outcome (minimax value for the root)
- Effectiveness is highly dependent on the child ordering
- With **perfect** ordering:
  - Time complexity drop from $O(b^m)$ to $O(b^{m/2})$
  - Double solvable depth
- Random move ordering is about $O(b^{3m/4})$
Resource Limits

• Resource is Limit, game tree is way too big!
• Solution:
  – Use a heuristic evaluation function
  – Replace the UTILITY function with EVAL
  – Terminal test ➔ Cutoff test
  – Search only a limited depth in the tree (Depth-limited search)
• Cutoff test
  – Return true for terminal states
  – Decide to cut off the search
Heuristic Alpha-Beta Tree Search

\[
\text{MINIMAX}(s) = \begin{cases} 
\text{UTILITY}(s) & \text{if}\ \text{TERMINAL-TEST}(s) \\
\max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if}\ \text{PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if}\ \text{PLAYER}(s) = \text{MIN}
\end{cases}
\]

\[
\text{H-MINIMAX}(s, d) = \begin{cases} 
\text{EVAL}(s) & \text{if}\ \text{CUTOFF-TEST}(s, d) \\
\max_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if}\ \text{PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if}\ \text{PLAYER}(s) = \text{MIN}.
\end{cases}
\]
Expectiminimax Search

- Explicit randomness
- Expectiminimax value for game with chance nodes
- Chance node is an expected value. Sum of the value over all outcomes, weighted by the probability of each chance action
- Compute the average score under optimal play
**Expectiminimax**

\[
\text{EXPECTIMINIMAX}(s) = \begin{cases} 
\text{UTILITY}(s) & \text{if Terminal-Test}(s) \\
\max_a \text{EXPECTIMINIMAX}((\text{RESULT}(s, a)) & \text{if Player}(s) = \text{Max} \\
\min_a \text{EXPECTIMINIMAX}((\text{RESULT}(s, a)) & \text{if Player}(s) = \text{Min} \\
\sum_r \Pr(r)\text{EXPECTIMINIMAX}((\text{RESULT}(s, r)) & \text{if Player}(s) = \text{Chance}
\end{cases}
\]

\[
\text{MINIMAX}(s) = \begin{cases} 
\text{UTILITY}(s) & \text{if Terminal-Test}(s) \\
\max_{a \in \text{Actions}(s)} \text{MINIMAX}((\text{RESULT}(s, a)) & \text{if Player}(s) = \text{Max} \\
\min_{a \in \text{Actions}(s)} \text{MINIMAX}((\text{RESULT}(s, a)) & \text{if Player}(s) = \text{Min}
\end{cases}
\]
Expectimax Search

- Max nodes as in minimax search
- Chance nodes are like min nodes, but the outcome is uncertain
- Calculate their *expected utilities*
A maximizer, a minimizer, and an equalizer. The maximizer chooses the highest score, the minimizer chooses the lowest score, and the equalizer tries to minimize the absolute value.

```
Fair Play

Max
  / \  
Equal
  / \  
Min  -1  2
     / \  
    /   \  
   /     \  
  -3     3
     / \  
    /   \  
   /     \  
  -5     3
     / \  
    /   \  
   /     \  
  2     4
```
Probabilities Basic

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes
- Probabilities are always positive
- Probabilities over all possible outcome sum to one
- Expectations: The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes
  - Example: How long to get to the ROA airport?
  - 45(25%) + 55(55%) + 60(20%) = 53.5 (mins)
Utilities

- Utilities are functions from outcomes (state of the world) to real numbers that describe an agent’s preferences
- Utilities summarize the agent’s goals
- Theorem: any “rational” preferences can be summarized as a utility function
- Example: Game (+1 / -1)
Decisions

- Agent make actions based on the desirability of their immediate outcomes (state)
  - Buy a lottery
  - Pick a Movie to watch
  - Take a course
  - ...

- Agent choose the action based on its preferences
- Agent’s preference are captured by a utility function, $U(s)$
- $U(s)$ assigns a value to express the desirability of a state
Maximum Expected Utility (MEU)

- $EU(a)$: Expected utility of an action $a$ given the evidence
- $P(\text{Result}(a)=s')$: The probability of reaching $s'$ by doing action $a$ in the current state
- $EU(a) = \sum_{s'} P(\text{Result}(a) = s')U(s')$
- Principle: A rational agent should choose the action that maximizes the agent’s expected utility

\[
\text{action} = \arg\max_a EU(a)
\]
Exam Topics

• Search
• Game
• **Constraint Satisfaction Problem**
• Logic / First-order Logic
• Markov Decision Process
Constraint Satisfaction Problem (CSP)

- A special subset of search problems
- State is represented by **factored representation**
  - A vector of attribute values
  - Boolean, real-valued, etc.
- Goal test: When each variable has a value that satisfies all the constraints on the variable
- A type of **identification problem**
Components

• X is a set of **Variables**

• D is a set of **Domains**, one for each variable and consist of a set of allowable **values**

• C is a set of **Constraints** that specify allowable combinations of values
Assignments

- Variable $X_1$ has domain $\{1,2,3\}$
- Variable $X_2$ has domain $\{1,2,3\}$
- Constraint $C <(X_1, X_2), X_1 > X_2>$
- An assignment that fulfills all constrains called a consistent or legal assignment
- A partial assignment has some variables unassigned.
- A complete assignment is that every variable is assigned a value.
- A solution to a CSP is a consistent, complete assignment
  \( \rightarrow \{(2,1),(3,2)\} \)
- A partial solution is a partial assignment that is consistent
- CSP is an NP-complete problem, which means that exists no known algorithm for finding solutions in polynomial time
Constraint Satisfaction Problem (CSP)

- A special subset of search problems
- State is represented by **factored representation**
  - A vector of attribute values
  - Boolean, real-valued, etc.
- Goal test: When each variable has a value that satisfies all the constraints on the variable
- A type of **identification problem**
Components

- X is a set of **Variables**
- D is a set of **Domains**, one for each variable and consist of a set of allowable **values**
- C is a set of **Constraints** that specify allowable combinations of values
Assignments

- Variable $X_1$ has domain $\{1,2,3\}$
- Variable $X_2$ has domain $\{1,2,3\}$
- Constraint $C \prec (X_1, X_2), X_1 > X_2$
- An assignment that fulfills all constraints called a consistent or legal assignment
- A partial assignment has some variables unassigned.
- A complete assignment is that every variable is assigned a value
- A solution to a CSP is a consistent, complete assignment
  - $\{(2,1),(3,2)\}$
- A partial solution is a partial assignment that is consistent
- CSP is an NP-complete problem, which means that exists no known algorithm for finding solutions in polynomial time
CSPs - Varieties of Domains

• Simplest: discrete and finite domains
  – Map coloring, Scheduling with time limits, 8-queens problem

• Discrete with infinite domains (integers, strings, etc.)
  – E.g. Scheduling with no deadline. (Infinite start time)
  – Linear constraints is solvable
  – Nonlinear constraints is undecidable

• Continuous domains
  – E.g. Start/End times for Hubble Telescope observations
  – Linear constraints is solvable in polynomial time by Linear programming
CSPs - Varieties of Constraints

• Simplest: unary constraint
  – Restricts the value of a single variable. E.g., SA != green

• Binary constraints:
  – Relates two variables. E.g., SA != NT

• Higher-order constraints:
  – Involves three or more variables. E.g., <(X,Y,Z), X<Y<Z>

• Global constraint:
  – Arbitrary number of variables
  – Alldiff: All the variables in the constraint must have different values. (Sudoku)
Standard Search Formulation

• States defined by the values assigned so far (partial assignments)
  – Initial State: the empty assignment {} 
  – Successor function: assign a value to an unassigned variable 
  – Goal test: The current assignment is competed and satisfies all constraints
Constraint Propagation

• Atomic state-space search algorithm: Expands a node to visit the successors
• CSP algorithm:
  – Generate successors by choosing a new variable assignment
  – Constraint propagation: a specific type of inference
• Constraint propagation:
  – Uses the constraints to reduce the number of legal values for variable
  – Have fewer choices to consider for the next choice of a variable assignment
  – May be done as a preprocessing step, before search starts
  – Local consistency
Node consistency

• If all the values in the variable’s domain satisfy the variable’s unary constraints
  – \(<(SA), \text{SA} \neq \text{green}>\) \(\rightarrow\) SA \{red, blue\}

• Each single node’s domain has a value which meets that node’s unary constraints

• A graph is node-consistent if every variable in the graph is node-consistent
  – \(<(SA), \text{SA} \neq \text{green}>, <(WA), \text{WA} \neq \text{red}>, <(NT), \text{NT} \neq \text{red}>, <(Q), \text{Q} \neq \text{red}>, <(NSW), \text{NSW} \neq \text{red}>, <(V), \text{V} \neq \text{red}>, <(T), \text{T} \neq \text{green}>\)
Arc consistency

• If every value in the variable’s domain satisfies the variable’s binary constraints
• For each pair of nodes, any consistent assignment to one can be extended to the other
  – Variable $X_i$ is arc-consistent with another variable $X_j$
  – For every value in $D_i$, you can find a value in $D_j$ that can satisfy the binary constraint on the arc $(X_i, X_j)$
  – E.g., $X \{0, 1, 2, 3\}$, $Y \{0, 1, 4, 9\}$, Binary constraint $Y = X^2$
• A graph is arc-consistency if every variable is arc-consistency with every other variable.
• **Arc consistency Demo**
Path consistency

- Tightens the binary constrains by looking at triples of variables.
- A set \( \{X_i, X_j\} \) is path-consistent with respect to a third variable \( X_m \) if every assignment \( \{X_i=a, X_j=b\} \) consistent with the constraints on \( \{X_i, X_j\} \), there is an assignment to \( X_m \) that satisfies the constraints on \( \{X_i, X_m\} \) and \( \{X_m, X_j\} \).
K-consistency

• Increasing degrees of consistency
  – 1-consistency: Node consistency
  – 2-consistency: Arc consistency
  – 3-consistency: Path consistency

• K-Consistency: For each k nodes, any consistent assignment to k-1 node can be extended to the kth node.

• K-consistent CSP: any set of k-1 variable and any consistent assignment

• Strong k-consistent CSP: It is k-consistent, (k-1)-consistent, (k-2)-consistent, …1-consistent
Global Constraint

• Occur frequency in real problems
• Can be handled by special-purpose algorithms that are more efficient than the general-purpose algorithms
• Alldiff constraint
• Sudoku: [Demo]
AC-3 Algorithm

• First stores all arcs in the CSP in a queue \( Q \).
  – Each binary constraint becomes two arcs, one in each direction.\n    \( Q = [\text{SA} \rightarrow \text{V}, \text{V} \rightarrow \text{SA}, \ldots] \)
• Remove arc \((X_i, X_j)\) from the \( Q \) and make \( X_i \) arc-consistent with \( X_j \)
• Continue remove values from the domains of variables until queue is empty
• Output: an arc-consistent CSP that have smaller domains, or no solution exists
• Time complexity: \( O(cd^3) \). \( c \) is the number of arcs and \( d \) is the size of the largest domain.
Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- Arc consistency still runs inside a backtracking search
Recap: Forward Checking

• When a variable X is assigned, for each unassigned variable Y that is connected to X by a constraint, delete any value in Y’s domain that is inconsistent with the value chosen for X.
Backtracking Search

• Basic uninformed algorithm for solving CSPs

• **One variable at a time**
  – Variable assignments are commutative (WA = red and NT = blue is the same as NT = blue, WA = red)
  – The order of any given set of actions does not matter
  – Only need to consider assignments to a single variable at each step

• **Check constraints as you go**
  – Select values that don’t conflict with any previously assigned values (assignments)
  – If no such value exist, **backtrack** and return to the previous variable and change its value
  – Might need computation to check the constraints

• **DFS** with above two improvements is called **backtracking search**
Inference

• AC-3 can reduce the domains of variables before the search
• Inference: Infer new domain reductions on the neighboring variables every time when making a choice of a value for a variable
• Forward checking:
  – One of the simplest forms of inference
  – When a variable X is assigned, for each unassigned variable Y that is connected to X by a constraint, delete any value in Y’s domain that is inconsistent with the value chosen for X
Ordering

- **Minimum Remaining Values (MRV)** heuristic: Choose the variable with the fewest legal left values in its domain
  - “Most constrained variable” heuristic
  - “Fail-first” heuristic
- **Degree** heuristic: Select the variable that is involved in the largest # of constraints
- **Least Constraining Value (LCV)** heuristic: Choose the value that rules out the fewest choices for the neighboring variables in the constraint graph
  - “Fail-last” heuristic
Forward checking

• Forward checking + MRV: After assigned \{WA = red\}, what next? NT or SA

• Maintaining Arc Consistency (MAC):
  – Variable $X_i$ is assigned a value
  – Inference procedure calls AC-3
  – Only the arcs $(X_j, X_i)$ for all $X_j$ that are unassigned variables that are neighbors of $X_i$ are stored in the queue
  – More strictly than forward checking
Local Search

• Find a good state without worrying about the path to get there
• Iterative improvement
  – Start with some random assignment to values
  – Iteratively select a random conflicted variable and reassign its value to the one that violates the fewest constraints until no more constraint violations exist
• A policy known as the **min-conflicts** heuristic
• Generally, much faster and more memory efficient (but incomplete and suboptimal)
Hill Climbing Search

- Most basic local search technique
- At each step, the current node is replaced by the best neighbor
- Greedy local search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← problem.INITIAL
    while true do
        neighbor ← a highest-valued successor state of current
        if VALUE(neighbor) ≤ VALUE(current) then return current
        current ← neighbor
```
Exam Topics

• Search
• Game
• Constraint Satisfaction Problem
• Logic / First-order Logic
• Markov Decision Process
Logic

• Sentence:
  – Represents some assertion about the world using knowledge representation language
  – **Axiom**: a *sentence* that is taken to be *true*, to serve as a premise or starting point for further reasoning and arguments
  – Every *sentence* must be either *true* or *false* in each possible world (model)
  – New *sentence* is derived from old *sentence* through inference

• Knowledge representation language has syntax
  – \( X + Y = 4 \) is a well-formed sentence. \( 4xy+ \) is not.
Logic

- Logic has **syntax** and **semantics**
  - **Syntax:** What sentences are allowed?
  - **Semantics:**
    - What are the possible worlds?
    - Which sentences are true in which worlds? (i.e., definition of truth)

- Propositional symbol: P, Q, R
- Syntax: \( P \lor (Q \land R) \)
- **Semantics** defines the truth of each sentence with respect to each possible **world** (model)
  - \( X + Y = 4 \) is true if and only if (iff) in the **world** that \( X = 2 \) and \( Y = 2 \)

- **Possible world**, a.k.a., **Model**
- A sentence \( \alpha \) is true in model \( m \).
  - \( m \) satisfies \( \alpha \)
  - \( m \) is a model of \( \alpha \)
- \( M(\alpha) \): the set of all models of \( \alpha \)
Logical Reasoning

- **Entailment:**
  - $\alpha |= \beta$: “sentence $\alpha$ entails sentence $\beta$”
  - $\alpha |= \beta$ iff in every model where $\alpha$ is true, $\beta$ is also true
  - $\alpha |= \beta$ iff $M(\alpha) \subseteq M(\beta)$
Logical Inference

• Entailment can be applied to derive conclusions
• Given KB with propositional logic sentences, check if sentence $\alpha$ is true
• Naive approach: Model-checking
  – Enumerates all possible models to check that $\alpha$ is true in all models in which KB is true
  – $M(KB) \subseteq M(\alpha)$
• Better approach: Theorem proving
  – Search for a sequence of proof steps (applications of inference rules) leading from $\alpha$ to $\beta$
Propositional Logic - Syntax

- Proposition Symbol: P, Q, W_{1,2}, GoHokies...
- Logical connectives: \( \neg, \land, \lor, \Rightarrow, \Leftrightarrow \)
- Complex sentences (Formula): \( \neg P, P \land Q, P \Rightarrow Q \)

**Sentence Grammar**

\[
\begin{align*}
\text{Sentence} & \rightarrow \text{AtomicSentence} \mid \text{ComplexSentence} \\
\text{AtomicSentence} & \rightarrow \text{True} \mid \text{False} \mid P \mid Q \mid R \mid \ldots \\
\text{ComplexSentence} & \rightarrow (\text{Sentence}) \\
& \mid \neg \text{Sentence} \\
& \mid \text{Sentence} \land \text{Sentence} \\
& \mid \text{Sentence} \lor \text{Sentence} \\
& \mid \text{Sentence} \Rightarrow \text{Sentence} \\
& \mid \text{Sentence} \Leftrightarrow \text{Sentence}
\end{align*}
\]

**Operator Precedence**

\( \neg, \land, \lor, \Rightarrow, \Leftrightarrow \)
Propositional Logic - Semantics

- The semantics defines the rules for determining the truth of a sentence with respect to a particular model.

- $P \land Q$ is true iff $P$ is true and $Q$ is true in $m_1 = \{P = \text{true}, Q = \text{true}\}$.

- $P \land Q$ is NOT true iff $P$ is false and $Q$ is true in $m_2 = \{P = \text{false}, Q = \text{true}\}$.

### Truth Table

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$\neg P$</th>
<th>$P \land Q$</th>
<th>$P \lor Q$</th>
<th>$P \Rightarrow Q$</th>
<th>$P \iff Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
Theorem Proving

• Apply rules of inference directly to the sentences in KB to construct a proof of the desired sentence without consulting models

• Logical equivalences
• Validity
• Satisfiability
Logical equivalences

- $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$

\[
\begin{align*}
(\alpha \land \beta) & \equiv (\beta \land \alpha) \quad \text{commutativity of } \land \\
(\alpha \lor \beta) & \equiv (\beta \lor \alpha) \quad \text{commutativity of } \lor \\
((\alpha \land \beta) \land \gamma) & \equiv (\alpha \land (\beta \land \gamma)) \quad \text{associativity of } \land \\
((\alpha \lor \beta) \lor \gamma) & \equiv (\alpha \lor (\beta \lor \gamma)) \quad \text{associativity of } \lor \\
\neg(\neg\alpha) & \equiv \alpha \quad \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) & \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition} \\
(\alpha \Rightarrow \beta) & \equiv (\neg\alpha \lor \beta) \quad \text{implication elimination} \\
(\alpha \leftrightarrow \beta) & \equiv ((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination} \\
\neg(\alpha \land \beta) & \equiv (\neg\alpha \lor \neg\beta) \quad \text{De Morgan} \\
\neg(\alpha \lor \beta) & \equiv (\neg\alpha \land \neg\beta) \quad \text{De Morgan} \\
(\alpha \land (\beta \lor \gamma)) & \equiv ((\alpha \land \beta) \lor (\alpha \land \gamma)) \quad \text{distributivity of } \land \text{ over } \lor \\
(\alpha \lor (\beta \land \gamma)) & \equiv ((\alpha \lor \beta) \land (\alpha \lor \gamma)) \quad \text{distributivity of } \lor \text{ over } \land
\end{align*}
\]
Validity

• A sentence is valid if it is true in all models. E.g., \( P \lor \neg P \)

• Tautologies: valid sentences, they are necessarily true

• Every valid sentence is logically equivalent to True

• For any sentences \( \alpha \) and \( \beta \), \( \alpha \models \beta \) iff the sentence \( (\alpha \Rightarrow \beta) \) is valid (true in all models)

• If \( (\alpha \Rightarrow \beta) \) is true in all models, then \( \alpha \models \beta \)
Satisfiability (SAT)

- A sentence is satisfiable if it is true in some model
- SAT problem: the problem of determining the satisfiability of sentences in propositional logic (NP-complete problem)
- CSPs ask whether the constraints are satisfiable by some assignment.

<table>
<thead>
<tr>
<th>$B_{1,1}$</th>
<th>$B_{2,1}$</th>
<th>$P_{1,1}$</th>
<th>$P_{1,2}$</th>
<th>$P_{2,1}$</th>
<th>$P_{2,2}$</th>
<th>$P_{3,1}$</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$KB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Validity and Satisfiability

- $\alpha$ is valid if $\neg \alpha$ is unsatisfiable (can’t find a single true in any model)
- $\alpha$ is satisfiable (find true in some model) if $\neg \alpha$ is not valid
- $\alpha \models \beta$ if the sentence $\left( \alpha \land \neg \beta \right)$ is unsatisfiable
Inference Rules

• Modus Ponens: \[
\frac{\alpha \Rightarrow \beta, \; \alpha}{\beta}
\]
  – Whenever any sentences of the form \( \alpha \Rightarrow \beta \) and \( \alpha \) are given, sentence \( \beta \) can be inferred
  – \( \alpha \Rightarrow \beta \) and \( \alpha \) are in KB, new sentence \( \beta \) can be derived
  – \( \text{Rain} \Rightarrow \text{Wet}, \; \text{Rain} \)
  \[
  \frac{\text{Rain} \land \text{Wet}}{\text{Wet}}
  \]
  , Wet can be inferred

• And-Elimination: \[
\frac{\alpha \land \beta}{\alpha}
\]
  – From a conjunction, any of the conjuncts can be inferred
  – \( \text{Rain} \land \text{Wet} \)
  \[
  \frac{\text{Rain} \land \text{Wet}}{\text{Rain}}
  \]
  , Wet can be inferred

• Monotonicity: if \( KB \models \alpha \) then \( KB \land \beta \models \alpha \)
Conjunctive Normal Form (CNF)

• Every logical sentence has a logically equivalent conjunctive normal form
• A sentence expressed as a conjunction of clauses is said to be in CNF
• We can formulate all the information contained in our knowledge base as one large conjunctive normal form
Unification

• Takes two sentences and returns a substitution $\theta$ which is the most general **unifier**, if one exists

• $\text{UNIFY}(p, q) = \theta$ where $\text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)$ or “fail” if no such $\theta$ exists

• $\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) = \{x/\text{Jane}\}$

• $\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Bill})) = \{x/\text{Bill}, y/\text{John}\}$

• $\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Mother}(y))) = \{y/\text{John}, x/\text{Mother}(\text{John})\}$

• $\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Ryan})) = \text{failure}$
Substitution

- A substation $\theta$ is a mapping from variables to terms
- $\text{SUBST}(\theta, \alpha)$ denotes the result of applying the substation $\theta$ to the sentence $\alpha$
- $\text{SUBST}$$\{x/\text{ryan}\}, P(x)) = P(\text{ryan})$
- $\text{SUBST}$$\{x/\text{ryan}, y/z\}, P(x) \land Q(x, y)) = P(\text{ryan}) \land Q(\text{ryan}, z)$
## Summary

<table>
<thead>
<tr>
<th>Propositional Logic</th>
<th>First-order Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Checking</td>
<td>N/A</td>
</tr>
<tr>
<td>Modus pones</td>
<td>Generalized Modus pones*</td>
</tr>
<tr>
<td>Forward chaining</td>
<td>Forward chaining*</td>
</tr>
<tr>
<td>Backward chaining</td>
<td>Backward chaining*</td>
</tr>
<tr>
<td>Resolution</td>
<td>Resolution*</td>
</tr>
</tbody>
</table>

* Unification and substitution
Exam Topics

• Search
• Game
• Constraint Satisfaction Problem
• Logic / First-order Logic
• Markov Decision Process
Definition of MDP

• A set of state $S$
• A set of actions $a$
• A start state
• Possibly one or more terminal states
• A transition model $T(s, a, s')$ from $s$ to $s'$ with an action $a$. (or $P(s' | s, a)$)
• A reward function $R(s, a, s')$ from $s$ to $s'$ with an action $a$. It may be positive or negative
• Possibly a discount factor $0 \leq \gamma \leq 1$
MDP

Nondeterministic search problems

- Agent’s utility depends on a sequence of decisions
- 80% will go up, 10% will go left and 10% will go right
- If hit the wall, the agent stop at the same place
- Two terminal states have reward +1 and -1
- All other transitions have a reward of -0.04
- Agent’s goal is to get maximize sum of rewards
Maximum Expected Utility (MEU)

- $EU(a)$: Expected utility of an action $a$ given the evidence
- $P(\text{Result}(a) = s')$: The probability of reaching $s'$ by doing action $a$ in the current state
- $EU(a) = \sum_{s'} P(\text{Result}(a) = s') U(s')$
- Principle: A rational agent should choose the action that maximizes the agent’s expected utility

$$\text{action} = \arg\max_a EU(a)$$

- Maximize the sum of rewards!
- Refer rewards now to rewards later
Discount factor

- Discount factor $\gamma$ is a number between 0 and 1
  - $\gamma$ is close to 0, rewards in the distant future is insignificant
  - $\gamma$ is close to 1, an agent is more willing to wait for long-term rewards
  - $\gamma$ is 1, discount rewards is purely additive rewards
- Values of rewards decay exponentially
Discounting

• We (human) like have rewards now then at later time
• Monetary:
  – Use money to buy stock and possible earn more money.
  – Put money in the bank and get interest.
• Uncertainty: stuff happens.
• $\gamma^0 \rightarrow \gamma^1 \rightarrow \gamma^2$
• $\gamma = 0.5$, $U([1,2,3]) = 1*1+0.5*2+0.25*3$
• It looks like a gamma chance of ending the process at every step and helps algorithms converge
Example: Discounting

$$U([s_0, a_0, s_1, a_1, s_2, ...]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + ...$$

- **Given:**
  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic
- **Q1:** For $\gamma = 1$, what is the optimal policy?
- **Q2:** For $\gamma = 0.1$, what is the optimal policy?
- **Q3:** For which $\gamma$ are West and East equally good when in state d?
Utilities of States

- An initial state $s$
- A policy $\pi$
- $S_t$: the state the agent reaches at time $t$ when executing policy $\pi$
- Expected utility: $U^\pi(s) = E \left[ \sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \right]$
- Expectation $E$: probability distribution over state sequences determined by $s$ and $\pi$
- Optimal policy: optimal action for every state $s$
- Optimal expected utility: $U^{\pi^*}(s)$
The Bellman Equation

- Expected utility: \( U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})] \)
- \( P(s'|s,a) \): the probability of reaching state \( s' \) if action \( a \) is done in state \( s \)
- \( R(s,a,s') \): the agent receives a reward from \( s \) to \( s' \) via action \( a \)
- Optimal policy \( \pi \) for a state \( s \): 
  \[ \pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')] \]
- The utility of a state \( s \): (Bellman equations) 
  \[ U(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s,a) [R(s,a,s') + \gamma U(s')] \]
- Utilities of a states are solutions of the set of Bellman equations
Bellman equations for the 4*3 World

- \( U(1,1) = \max \{ \)
  
  \[
  0.8(-0.04 + \gamma U(1,2)) + 0.1(-0.04 + \gamma U(2,1)) + 0.1(-0.04 + \gamma U(1,1))
  \]

  Up

  \[
  0.9(-0.04 + \gamma U(1,1)) + 0.1(-0.04 + \gamma U(1,2))
  \]

  Left

  \[
  0.9(-0.04 + \gamma U(1,1)) + 0.1(-0.04 + \gamma U(2,1))
  \]

  Down

  \[
  0.8(-0.04 + \gamma U(2,1)) + 0.1(-0.04 + \gamma U(1,2)) + 0.1(-0.04 + \gamma U(1,1))
  \]

  Right

\}

The utilities of the states in the 4×3 world with \( \gamma = 1 \) and \( r = -0.04 \) for transitions to nonterminal states.
Policy Extraction

- Turn values into a policy
- Do one step expectimax
Computing Actions from Q-Values

• Assume that we have the optimal values $U^*(s)$

• Do one step expectimax and get the optimal Q-values

• $\pi^*(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')]$

• $Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U^*(s')]$

• $\pi^*(s) = \arg \max_{a \in A(s)} Q^*(s,a)$

• Optimal policy is simply picking the largest q-value in each state
Q-function

- Also called action-utility function
- \( Q(s, a) \) is the expected utility if taking a given action in a given state
- Utilities \( U(s) = \max_a Q(s, a) \)
- \( \pi^*(s) = \arg\max_a Q(s, a) \)
- \( Q(s, a) = \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma \max_{a'} Q(s', a')] \)
- \( U(s) = \max_a \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma U(s')] \)
Optimal Quantities

- $U^*(s) = \max_a Q^*(s, a)$

- $Q^*(s, a) = \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U^*(s')]$

- $U^*(s) = \max_a \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U^*(s')]$
Value Iteration

• Start with initial values of zero. U0(s) = 0
• Do Bellman update
• Repeat until convergence
• Complexity of each iteration: O(S^2A)
Bellman Update

- Let $U_i(s)$ be the utility value for state $s$ at the $i$th iteration
- Bellman update:
  $$U_{i+1}(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma U_i(s')]$$
- The update is to be applied simultaneously to all the states at each iteration
Policy Iteration

• Alternative approach for optimal policy:
  – Step 1: **Policy evaluation**: calculate utilities for some fixed policies until convergence
  – Step 2: **Policy improvement**: update policy using one-step look-ahead with resulting converged utilities as future values
  – Repeat steps until policy converges (policy improvement step yields no change in the utilities)

• It is still optimal
• Can converge (much) faster under some conditions
• Efficiency: $O(S^2)$ per iteration
Fixed Policies

Do the optimal action

- Expectimax trees max over all actions to compute the optimal values

Do what \( \pi \) says to do

- Expectimax trees max one action per state to compute the optimal values
Utilities for a Fixed Policy

• Compute the utility of a state $s$ under a fixed initial random policy

• Expected total discounted rewards starting in $s$ and following policy $\pi$

\[
U^\pi(s) = E \left[ \sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \right]
\]

• $U^\pi(s) = \sum_{s'} P(s'|s, \pi(s))[R(s, \pi(s), s') + \gamma U^\pi(s')]$
Policy Evaluation

- Given a policy $\pi_i$, calculate $U_i = U^{\pi_i}$, the utility of each state if $\pi_i$ were to be executed.
- $U_i(s) = \sum_{s'} P(s'|s, \pi_i(s))[R(s, \pi_i(s), s') + \gamma U_i(s')]$
Policy Improvement

• Calculate a new MEU policy $\pi_{i+1}$ using one-step look-ahead based on $U_i$
• Get a better policy using policy extraction
• $\pi^*(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma U_i(s')]$
Demo: Policy Improvement

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.12</td>
<td>-0.12</td>
<td>0.65</td>
<td>+1</td>
<td></td>
</tr>
<tr>
<td>-0.12</td>
<td>-0.21</td>
<td>-0.12</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-0.12</td>
<td>-0.12</td>
<td>-0.12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccc}
-0.12 & -0.12 & 0.65 \\
-0.12 & -0.21 & -0.12 \\
-0.12 & -0.12 & -0.12 \\
\end{array}
\]

\[
\begin{array}{ccc}
0.57 & -0.08 & 0.80 \\
-0.08 & -0.12 & 0.80 \\
-0.12 & -0.12 & -0.12 \\
\end{array}
\]

\[
\begin{array}{ccc}
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\end{array}
\]

\[
\begin{array}{ccc}
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\end{array}
\]

\[
\begin{array}{ccc}
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\end{array}
\]

\[
\begin{array}{ccc}
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\end{array}
\]

\[
\begin{array}{ccc}
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\rightarrow & \rightarrow & \rightarrow \\
\end{array}
\]
Comparison

<table>
<thead>
<tr>
<th>Value Iteration</th>
<th>Policy Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every iteration updates the values (Bellman update)</td>
<td>Every iteration updates the utilities with fixed policy (simplified Bellman update)</td>
</tr>
<tr>
<td>Compute Q-Values</td>
<td>Compute Q-Values</td>
</tr>
<tr>
<td></td>
<td>After the policy is evaluated, a new policy is chosen (one-step lookahead)</td>
</tr>
<tr>
<td>Stop until value converges</td>
<td>Stop until policy converges</td>
</tr>
<tr>
<td>Policy extraction (one-step lookahead)</td>
<td></td>
</tr>
<tr>
<td>Dynamic program</td>
<td>Dynamic program</td>
</tr>
</tbody>
</table>