Mid-term Review
Chapters 2-6

• Review Agents (2.1-2.3)
• Review Constraint Satisfaction (6.1-6.4)
• 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)

• Please review your quizzes and old CS-171 tests
  • At least one question from a prior quiz or old CS-171 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
• Properties of Task Environments
  – Fully vs. partially observable; single vs. multi agent;
    deterministic vs. stochastic; episodic vs. sequential; static
    vs. dynamic; discrete vs. continuous; known vs. unknown
• Basic Definitions
  – Percept, percept sequence, agent function, agent program
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
Agents and environments

- **Percept**: agent’s perceptual inputs at an instant

- The **agent function** maps from percept sequences to actions:
  \[ f : P^* \rightarrow A \]

- The **agent program** runs on the physical **architecture** to produce \( f \)

- **agent = architecture + program**
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.
Task Environment

• Before we design an intelligent agent, we must specify its “task environment”:

PEAS:

Performance measure
Environment
Actuators
Sensors
Environment types

• Fully observable (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.

• Deterministic (vs. stochastic): The next state of the environment is completely determined by the current state and the action executed by the agent. (If the environment is deterministic except for the actions of other agents, then the environment is strategic)

• Episodic (vs. sequential): An agent’s action is divided into atomic episodes. Decisions do not depend on previous decisions/actions.

• Known (vs. unknown): An environment is considered to be "known" if the agent understands the laws that govern the environment's behavior.
Environment types

- **Static** (vs. **dynamic**): The environment is unchanged while an agent is deliberating. (The environment is **semidynamic** if the environment itself does not change with the passage of time but the agent's performance score does)

- **Discrete** (vs. **continuous**): A limited number of distinct, clearly defined percepts and actions.

How do we **represent** or **abstract** or **model** the world?

- **Single agent** (vs. **multi-agent**): An agent operating by itself in an environment. Does the other agent interfere with my performance measure?
Review Constraint Satisfaction
Chapter 6.1-6.4

• What is a Constraint Satisfaction Problem (6.1)
• Backtracking Search for CSP (6.3)
  • Variable Ordering
    • Minimum Remaining Values, Degree Heuristic
  • Value Ordering --- Least Constraining Value
• Constraint Propagation (6.2)
  • Forward Checking, Arc Consistency
• Local search for CSPs (6.4)
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.

• An assignment is **consistent** when no constraints are violated.

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

• **A solution is:**
  – A complete and consistent assignment.
  – All variables assigned, all constraints satisfied.

• 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
Backtracking search

• **Similar to Depth-first search**
  – At each level, picks a single variable to explore
  – Iterates over the domain values of that variable

• **Generates kids one at a time, one per value**

• **Backtracks when a variable has no legal values left**

• **Uninformed algorithm**
  – No good general performance
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RRECURSIVE-BACTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
Minimum remaining values (MRV)

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

- 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?*
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RRECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
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 application of the heuristic with states and possible 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.

- When a variable is assigned a value, update all neighbors in the constraint graph.
- *Forward checking stops after one step and does not go beyond immediate neighbors.*

- Terminate search when any variable has no legal values.
Forward checking
Check only neighbors; delete any inconsistent values

• Assign \( \{WA=\text{red}\} \)

• Effects on other variables connected by constraints to WA
  – \( NT \) can no longer be red
  – \( SA \) can no longer be red
Forward checking
Check only neighbors; delete any inconsistent values

- Assign \( \{Q=\text{green}\} \)

- 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

We already have failure, but Forward Checking is too simple to detect it now.
Forward checking
Check only neighbors; delete any inconsistent values

- If \( V \) is assigned \textit{blue}

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

- FC has detected that partial assignment is \textit{inconsistent} with the constraints and backtracking can occur.
Arc consistency algorithm (AC-3)

function AC-3(csp) returns false if inconsistency found, else true, may reduce csp domains
inputs: csp, a binary CSP with variables \{X_1, X_2, ..., X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp
/* initial queue must contain both (X_i, X_j) and (X_j, X_i) */
while queue is not empty do
    \((X_i, X_j) \leftarrow \text{REMOVE-FIRST(queue)}\)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
        if size of \(D_i = 0\) then return false
        for each \(X_k\) in \text{NEIGHBORS}[X_i] \setminus \{X_j\} do
            add \((X_k, X_i)\) to queue if not already there
    return true

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff we delete a value from the domain of \(X_i\)
removed \(\leftarrow false\)
for each \(x\) in \text{DOMAIN}[X_i] do
    if no value \(y\) in \text{DOMAIN}[X_j] allows \((x,y)\) to satisfy the constraints between \(X_i\) and \(X_j\) then delete \(x\) from \text{DOMAIN}[X_i]; removed \(\leftarrow true\)
return removed

(from Mackworth, 1977)
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

- An Arc $X \rightarrow Y$ is consistent if for every value $x$ of $X$ there is some value $y$ of $Y$ consistent with $x$ (note that this is a directed property)

- Consider state of search after WA and Q are assigned & FC is done:

  $SA \rightarrow NSW$ is consistent if $SA=blue$ and $NSW=red$
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

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

• $NSW \rightarrow SA$ is consistent if
  $NSW=red$ and $SA=blue$
  $NSW=blue$ and $SA=???

• $NSW=blue$ can be pruned: No current domain value of $SA$ is consistent
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

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

- Continue to propagate constraints….
  - Check $V ightarrow NSW$
  - Not consistent for $V = red$
  - Remove red from $V$
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

Continue to propagate constraints….

• **SA → NT** is not consistent
  – and cannot be made consistent (Failure!)

• Arc consistency detects failure earlier than FC
  – Requires more computation: *Is it worth the effort??*
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
Local search for CSP

function MIN-CONFLICTS(csp, max_steps) return solution or 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 VARIABLES[csp]
    value ← the value v for var that minimizes
             CONFLICTS(var,v,current,csp)
    set var = value in current
return failure
Min-conflicts example 1

Use of min-conflicts heuristic in hill-climbing.
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
A problem is defined by five items:

- **Initial state**: e.g., "at Arad"
- **Actions**: \( \text{Actions}(X) = \text{set of actions available in State } X \)
- **Transition model**: \( \text{Result}(S,A) = \text{state resulting from doing action } A \text{ in state } S \)
- **Goal test**: e.g., \( x = \text{"at Bucharest"}, \text{Checkmate}(x) \)
- **Path cost**: (additive, i.e., the sum of the step costs)
  - \( c(x,a,y) = \text{step cost of action } a \text{ in state } x \text{ to reach state } y \)
  - assumed to be \( \geq 0 \)

A solution is a sequence of actions leading from the initial state to a goal state.
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.
Solutions to Repeated States

- **Graph search**
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have $9! = 362,880$ states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.
  - “visited?” test usually implemented as a hash table

---

36
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 \( \text{Actions}(S) \) and \( \text{Result}(S,A) \) functions associated with the problem.
General tree search

function `TREE-SEARCH(problem, fringe)` returns a solution, or failure

`fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)`

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

function `EXPAND(node, problem)` returns a set of nodes

`successors ← the empty set`

for each `action, result in SUCCESSOR-FN(problem)(STATE[node])` do
  `s ← a new NODE`
  `PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result`
  `PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)`
  `DEPTH[s] ← DEPTH[node] + 1`
  add `s` to `successors`

return `successors`
function GRAPH-SEARCH( problem, fringe) returns 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)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end loop
Breadth-first graph search

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0 if
problem.GOAL-TEST(node.STATE) then return SOLUTION(node) frontier ←
a FIFO queue with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
            frontier ← INSERT(child, frontier)

Figure 3.11 Breadth-first search on a graph.
Uniform cost search: sort by $g$

A* is identical but uses $f=g+h$
Greedy best-first is identical but uses $h$

**Function** `UNIFORM-COST-SEARCH(problem)` **returns** a solution, or failure

1. node ← a node with `STATE = problem.INITIAL-STATE`, `PATH-COST = 0`
2. frontier ← a priority queue ordered by `PATH-COST`, with `node` as the only element
3. explored ← an empty set
4. loop do
   - if `EMPTY?(frontier)` then return `failure`
   - node ← `POP(frontier)` /* chooses the lowest-cost node in `frontier` */
   - if `problem.GOAL-TEST(node.STATE)` then return `SOLUTION(node)`
   - add `node.STATE` to `explored`
   - for each action in `problem.ACTIONS(node.STATE)` do
     - child ← `CHILD-NODE(problem, node, action)`
     - if `child.STATE` is not in `explored` or `frontier` then
       - frontier ← `INSERT(child, frontier)`
     - else if `child.STATE` is in `frontier` with higher `PATH-COST` then
       - replace that `frontier` node with `child`

**Figure 3.14** Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for `frontier` needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.
Depth-limited search & IDS

function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if Goal-Test[problem](State[node]) then return Solution(node)
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure

function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
When to do Goal-Test? Summary

• For DFS, BFS, DLS, and IDS, the goal test is done when the child node is generated.
  – These are not optimal searches in the general case.
  – BFS and IDS are optimal if cost is a function of depth only; then, optimal goals are also shallowest goals and so will be found first

• For GBFS the behavior is the same whether the goal test is done when the node is generated or when it is removed
  – $h(\text{goal})=0$ so any goal will be at the front of the queue anyway.

• For UCS and A* the goal test is done when the node is removed from the queue.
  – This precaution avoids finding a short expensive path before a long cheap path.
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.

**Review example Uniform-cost search**

— Slides 25-34, Lecture on “Uninformed Search”
Search strategy evaluation

• 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 $\infty$)
  – (for **UCS**: $C^*$: true cost to optimal goal; $\varepsilon > 0$: minimum step cost)
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>$O(b^{1+C*/\epsilon})$</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>$O(b^{1+C*/\epsilon})$</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 $\geq \epsilon > 0$
[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 $\geq \epsilon > 0$)
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

- \( h(n) = \) estimate of cost from \( n \) to \( goal \)
  - e.g., \( h(n) = \) straight-line distance from \( n \) 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 $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach $n$
- $h(n)$ = estimated cost from $n$ to goal
- $f(n)$ = estimated total cost of path through $n$ 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

\[
\begin{align*}
f(n') &= g(n') + h(n') \quad \text{(by def.)} \\
&= g(n) + c(n,a,n') + h(n') \quad \text{(} g(n')=g(n)+c(n.a.n')\text{)} \\
&\geq g(n) + h(n) = f(n) \quad \text{(consistency)} \\
f(n') &\geq f(n)
\end{align*}
\]

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

• Theorem:
  If \( h(n) \) is consistent, A* using \texttt{GRAPH-SEARCH} is optimal
  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)
Random Restart Wrapper

• These are stochastic local search methods
  – Different solution for each trial and initial state

• Almost every trial hits difficulties (see below)
  – Most trials will not yield a good result (sadly)

• Many random restarts improve your chances
  – Many “shots at goal” may, finally, get a good one

• Restart a random initial state; many times
  – Report the best result found; across many trials
Random Restart Wrapper

BestResultFoundSoFar <- infinitely bad;
UNTIL ( you are tired of doing it ) DO {
    Result <- ( Local search from random initial state );
    IF ( Result is better than BestResultFoundSoFar ) THEN ( Set BestResultFoundSoFar to Result );
}
RETURN BestResultFoundSoFar;

Typically, “you are tired of doing it” means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that Result improvements are small and infrequent, e.g., less than 0.1% Result improvement in the last week of run time.
Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the dimensionality of the search space increases to high dimensions.

- **Problems:** depending on state, can get stuck in local maxima
  - Many other problems also endanger your success!!
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

```plaintext
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 $e^{ΔE/T}$
```

Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.
P(accepting a worse successor)

Decreases as Temperature $T$ decreases
Increases as $|\Delta E|$ decreases

(Sometimes step size also decreases with $T$)

| $|\Delta E|$ | High | Medium | Low | High | Medium | Low |
|-------------|------|--------|-----|------|--------|-----|

$e^{\Delta E / T}$

next ← a randomly selected successor of current
\[ \Delta E \leftarrow \text{VALUE}[\text{next}] - \text{VALUE}[\text{current}] \]
if $\Delta E > 0$ then current ← next
else current ← next only with probability $e^{\Delta E / T}$
Genetic algorithms (Darwin!!)

- A state = a string over a finite alphabet (an individual)
- Start with $k$ randomly generated states (a population)
- Fitness function (= our heuristic objective function).
  - Higher fitness values for better states.
- Select individuals for next generation based on fitness
  - $P(\text{individual in next gen.}) = \frac{\text{individual fitness}}{\Sigma \text{population fitness}}$
- Crossover fit parents to yield next generation (off-spring)
- Mutate the offspring randomly with some low probability
fitness = #non-attacking queens

probability of being in next generation = fitness/(Σ_i fitness_i)

- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28
- Σ_i fitness_i = 24+23+20+11 = 78
- P(child_1 in next gen.) = fitness_1/(Σ_i fitness_i) = 24/78 = 31%
- P(child_2 in next gen.) = fitness_2/(Σ_i fitness_i) = 23/78 = 29%; etc
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.
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.
  - **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 "best" next move.
An optimal procedure: The Min-Max method

Will find the **optimal strategy and best next move** for Max:

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 move leading to the child of highest value.**
Two-Ply Game Tree
Two-Ply Game Tree

Minimax maximizes the utility of the worst-case outcome for Max

The minimax decision
Pseudocode for Minimax Algorithm

```plaintext
function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game
return arg max_{a \in ACTIONS(state)} MIN-VALUE(Result(state,a))

function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← +∞
for a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(Result(state,a)))
return v

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

Properties of minimax

• **Complete?**
  – Yes (if tree is finite).

• **Optimal?**
  – Yes (against an optimal opponent).
  – Can it be beaten by an opponent playing sub-optimally?
    • No. (Why not?)

• **Time complexity?**
  – $O(b^m)$

• **Space complexity?**
  – $O(bm)$ (depth-first search, generate all actions at once)
  – $O(m)$ (backtracking search, generate actions one at a time)
Cutting off search

MinimaxCutoff is identical to MinimaxValue except
1. Terminal? is replaced by Cutoff?
2. Utility is replaced by 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 $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov
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

Black to move
White slightly better

White to move
Black winning

For chess, typically *linear* weighted sum of features

\[ \text{Eval}(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s) \]

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

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

- 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 \( \alpha \geq \beta \)
Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game
$v ← MAX-VALUE(state, -∞, +∞)$
return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
$v ← -∞$
for $a$ in ACTIONS(state) do
$v ← MAX(v, MIN-VALUE(Result(s,a), $\alpha$, $\beta$))$
if $v ≥ $ then return $v$
$\alpha ← MAX(\alpha, v)$
return $v$

(MIN-VALUE is defined analogously)
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.
Alpha-Beta Example Revisited

Do DF-search until first leaf

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

\[ \alpha, \beta, \text{initial values} \]

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

\[ \alpha, \beta, \text{passed to kids} \]

Review Detailed Example of Alpha-Beta Pruning in lecture slides.
MIN updates $\beta$, based on kids
MIN updates $\beta$, based on kids.
No change.
Alpha-Beta Example (continued)

MAX updates $\alpha$, based on kids.

$\alpha = 3$

$\beta = +\infty$

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

MAX

MIN

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

\( \alpha, \beta, \text{passed to kids} \)
\[ \alpha = 3 \]
\[ \beta = +\infty \]
Alpha-Beta Example (continued)

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

MIN updates \( \beta \), 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.

Review Detailed Example of Alpha-Beta Pruning in lecture slides.
Mid-term Review
Chapters 2-6

• Review Agents (2.1-2.3)
• Review Constraint Satisfaction (6.1-6.4)
• 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)

• Please review your quizzes and old CS-171 tests
  • At least one question from a prior quiz or old CS-171 test will appear on the mid-term (and all other tests)