Artificial Intelligence (CS303)

Lecture 3: Problem-Specific Search

Hints for this lecture

- The more we know about the problem characteristic/structure, the better we can solve it.

Outline of this lecture

- **Make Search Algorithms Less General**
- **Gradient-based Methods for Numerical Optimization**
- **Quadratic Programming Problems**
- **Constraint Satisfaction Problems**
- **Adversarial Search**

Make Search Algorithms Less General

- The search methods talked previously are rather general, i.e., applicable to any problem.

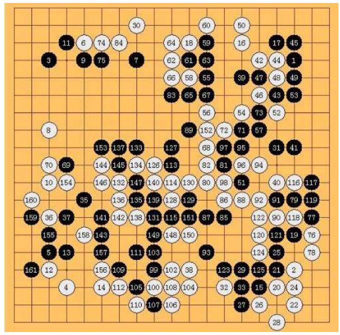
- Generality is nice and worthy of pursuit, while it usually conflicts with the other desired features of algorithms, e.g., efficiency.

- A search method is general because the characteristic/structure (no matter we know or not) of the problem **is not taken into account** when designing the search method.

Make Search Algorithms Less General

- When designing an algorithm for a problem (class), taking the problem characteristics into account usually helps us get the desired solution by **searching only a part of the search/state space**, making the search more efficient.

- In some cases, we do know something about problem.



We know all previous steps

- As long as a problem (class) is **of sufficient significance**, it is worthy of designing problem-specific algorithm for it.

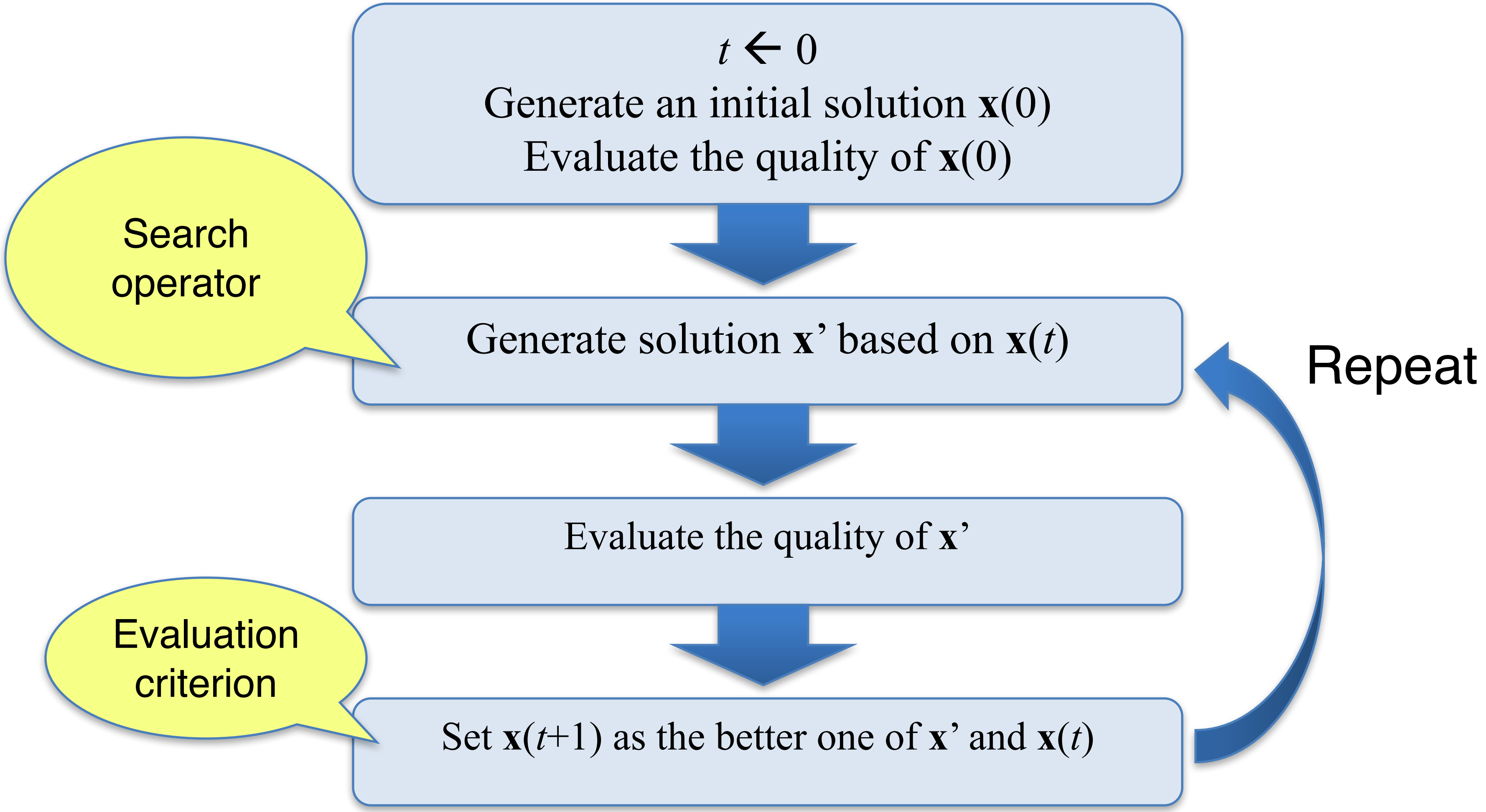
Make Search Algorithms Less General

- Again, consider the ubiquitous optimization problems.

$$
\begin{aligned}
&\text{maximize} \quad f(x) \\
&\text{subject to: } g_i(x) \leq 0, \quad i = 1 \ldots m \\
&\qquad\qquad\quad\; h_j(x) = 0, \quad j = 1 \ldots p
\end{aligned}
$$

- What do you mean by “problem characteristic”? Most basically:
  - What is *x* ?
  - What is *f* ?
  - Does *f* fulfill some properties that would lead to a more efficient search?

Recall The General Framework for Search



Gradient-based Methods for Numerical Optimization

- Suppose the objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is **continuous and differentiable** (thus the gradient could be calculated)

Gradient methods compute

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

to increase/reduce f, e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

Quadratic Programming Problems

- The objective function is a **quadratic** function of x
  - stronger condition than just differentiable.
- The constraints are linear functions of x

$$
\begin{aligned}
&\text{maximize} \quad f(x) \\
&\text{subject to: } g_i(x) \le 0, \quad i = 1 \ldots m \\
&\qquad\qquad\quad\;\; h_j(x) = 0, \quad j = 1 \ldots p
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
&\min f(x) = q^T x + \frac{1}{2} x^T Q x \\
&s.t.\ Ax = a \\
&Bx \le b \\
&x \ge 0
\end{aligned}
$$

Quadratic Programming Problems

- We take an even stronger condition as example
  - no constraints.
  - The objective function is not only quadratic, but also **convex**.
    - f(x) is a convex function of x

$$\min f(x) = q^T x + \frac{1}{2} x^T Q x$$

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y).$$

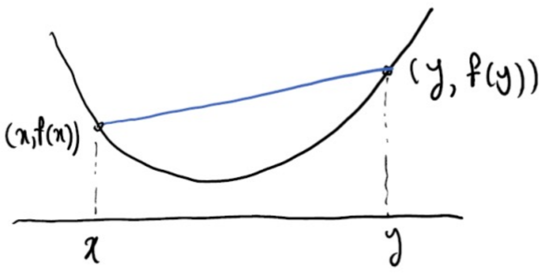
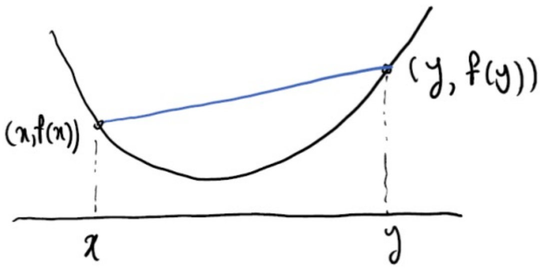


Figure 1: An illustration of the definition of a convex function

Quadratic Programming Problems

- How to solve such a problem by search?
  - Simply set the derivative of *f* to 0, and solve a linear system
  - No need to *search* at all!

$$\min f(x) = q^T x + \frac{1}{2} x^T Q x$$



- More practical cases still needs search (e.g., conjugate gradient method for QP with (e.g., with constraints), recall the Lagrange multiplier technique.

Quadratic Programming Problems

- How do I know the objective function is convex?
  - A sufficient condition: Q is positive definite.

$$q^T[\lambda x+(1-\lambda)y]+\frac{1}{2}[\lambda x+(1-\lambda)y]^T Q[\lambda x+(1-\lambda)y]$$

$$\lambda q^T x+(1-\lambda)q^T y+\frac{1}{2}\lambda^2 x^T Q x+\frac{1}{2}(1-\lambda)^2 y^T Q y$$

$$+\frac{1}{2}\lambda(1-\lambda)x^T Q y+\frac{1}{2}\lambda(1-\lambda)y^T Q x$$

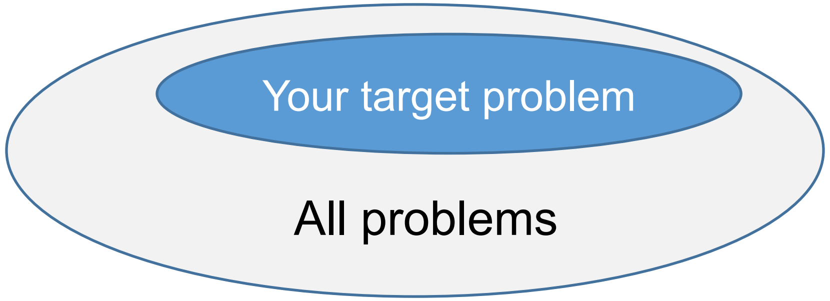
$$\leq$$

$$\lambda q^T x+\frac{1}{2}\lambda x^T Q x+(1-\lambda)q^T y+\frac{1}{2}(1-\lambda)y^T Q y$$

Quadratic Programming Problems

Lesson learned from the simple example

- if the problem have very good property, we can even reduce the search process to a single step (solve analytically).
- Needs to carefully check whether the “good property” holds.
- Intuitively, better property corresponds to stronger conditions
  - more unlikely to hold
  - application-domain of search algorithm developed based on such properties is more restrictive.



Constraint Satisfaction Problems

Standard search problem:

state is a "black box"—any old data structure
that supports goal test, eval, successor

CSP:

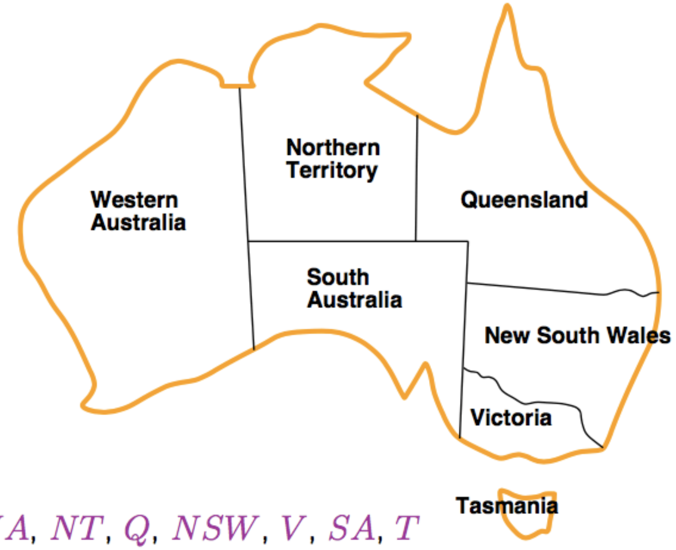
state is defined by variables X_i with values from domain D_i

goal test is a set of constraints specifying
allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

Allows useful **general-purpose** algorithms with more power than standard search algorithms

Example: Map Coloring



Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Variants of CSPs

- Unary constraints involve a single variable.
- Binary constraints involve pairs of variables.
- Higher-order constraints involve 3 or more variables.
- Preferences (Soft constraints), e.g., red is better than green, is often represented by a cost for each variable assignment (i.e., the target is to minimize the cost).

Real-world CSPs

- **Assignment problems**
- **Timetabling problems**
- **Hardware configuration**
- **Floorplanning**
- **Factory scheduling**
- **…**

What is the search tree of a CSP?

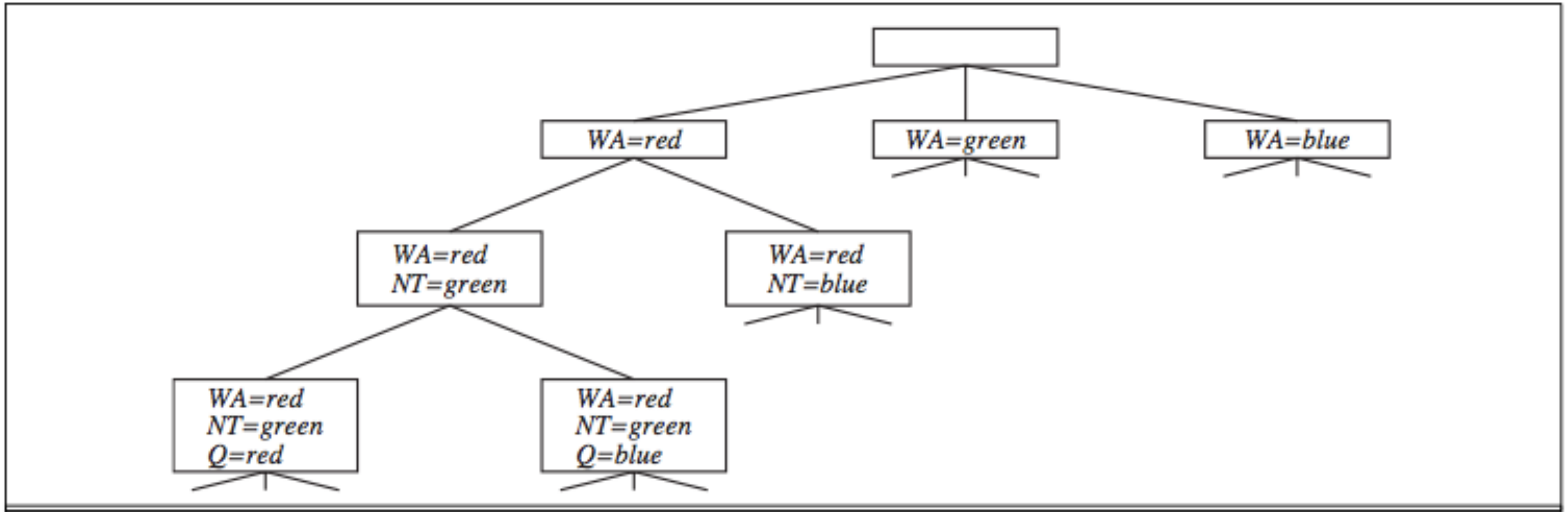
Characteristics of CSPs

- **Commutativity: the order of assigning values to variables does not affect the final outcome.**

- **The constrains provide additional information that could be represented by a constraint graph.**

Commutativity

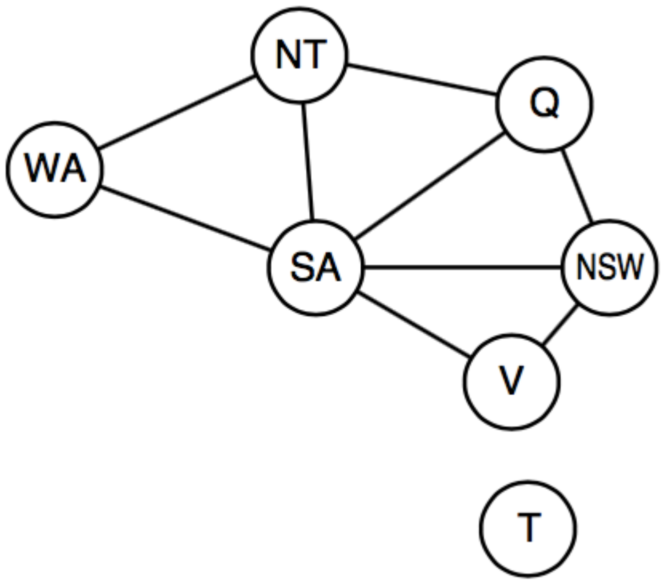
- Commutativity help us formulate the search tree (only 1 variable needs to be considered at each node in the search tree).



Constraint Graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure
to speed up search. E.g., Tasmania is an independent subproblem!

Inference

- A constraint graph allows the agent to do inference in addition to search.
- Inference basically means checking local consistency (or detecting inconsistency)
  - Node consistency
  - Arc Consistency
  - Path Consistency
  - *K*-consistency
  - Global consistency

- Inference helps prune the search tree, either before or during the search.

Backtracking Search for CSP

- **Depth-first search, assign a value to unassigned variables recursively.**
- **If inconsistency occurred, move 1 step back to try another value.**

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Improving Backtracking Search (1)

Applying inference

Applying inference

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    n RECURSIVE-BACKTRACKING({ }, csp)

    on RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

Improving Backtracking Search (2)

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Choosing variables with minimum numbers of remaining value

Improving Backtracking Search (3)

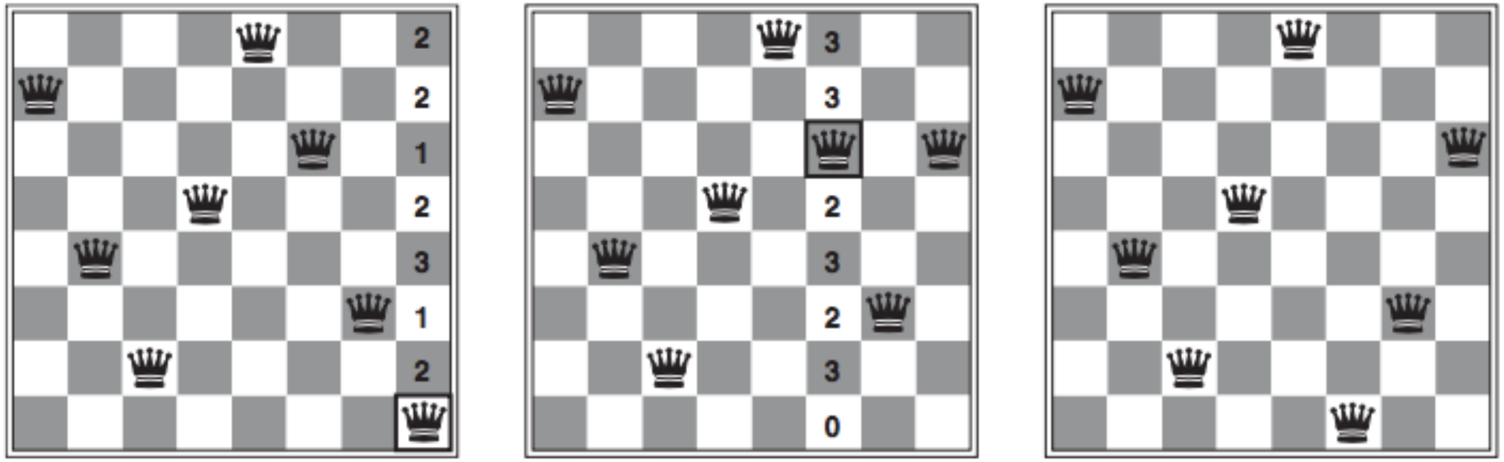
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Maintain a conflict set and do backjumping

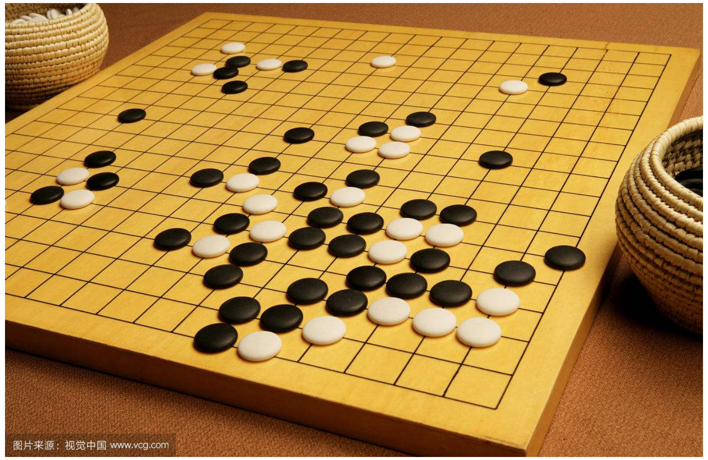
Local Search for CSP

- CSP can be actually reformulated as a constraint optimization problem, for which the objective function is to minimize the constraint violation.
- Working in the solution space (complete solution formulation)
- Iteratively select a conflicted variable and assign a different value to it.
- Choose the value that leads to the minimum cost.



Games as A Search Problem

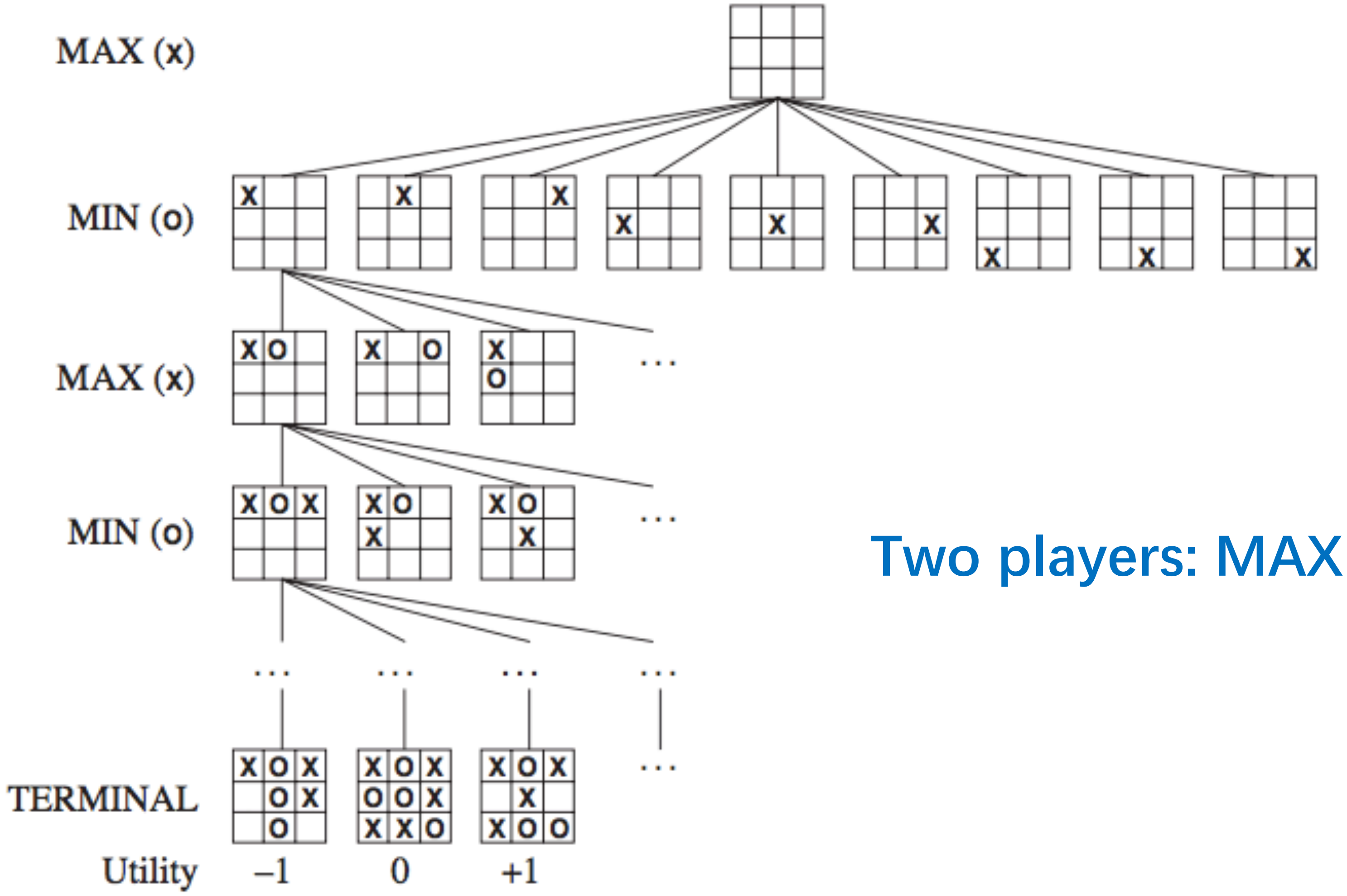
- More than 1 agent/player, with conflicting goals.
- We consider two players, zero-sum games.



Definition of a game

- S_0: The **initial state**, which specifies how the game is set up at the start.
- $\text{PLAYER}(s)$: Defines which player has the move in a state.
- $\text{ACTIONS}(s)$: Returns the set of legal moves in a state.
- $\text{RESULT}(s, a)$: The **transition model**, which defines the result of a move.
- $\text{TERMINAL-TEST}(s)$: A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- $\text{UTILITY}(s, p)$: A **utility function** (also called an objective function or payoff function),

Tic-Tac-Toe Search Tree

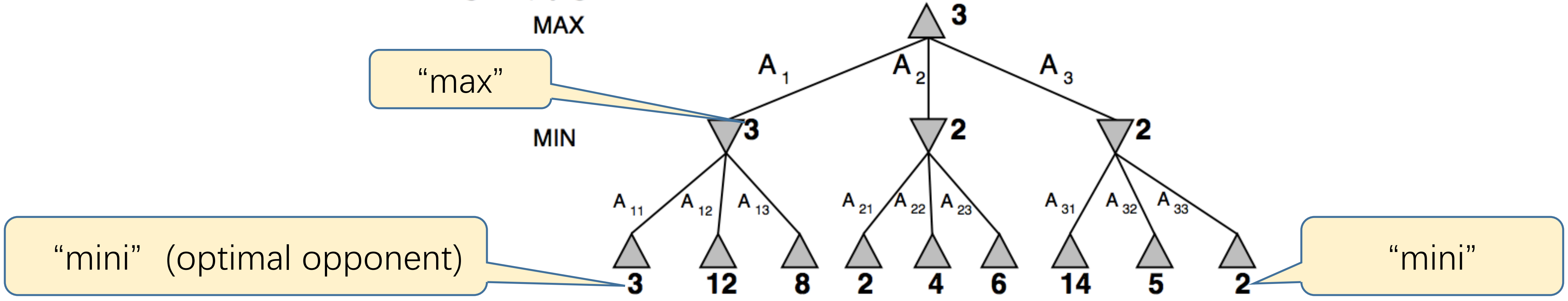


Two players: MAX and MIN

Optimal decision in games

- Assume the game is deterministic and perfect information is available
- Idea (for MAX): choose the move to position the highest minimax value

E.g., 2-ply game:



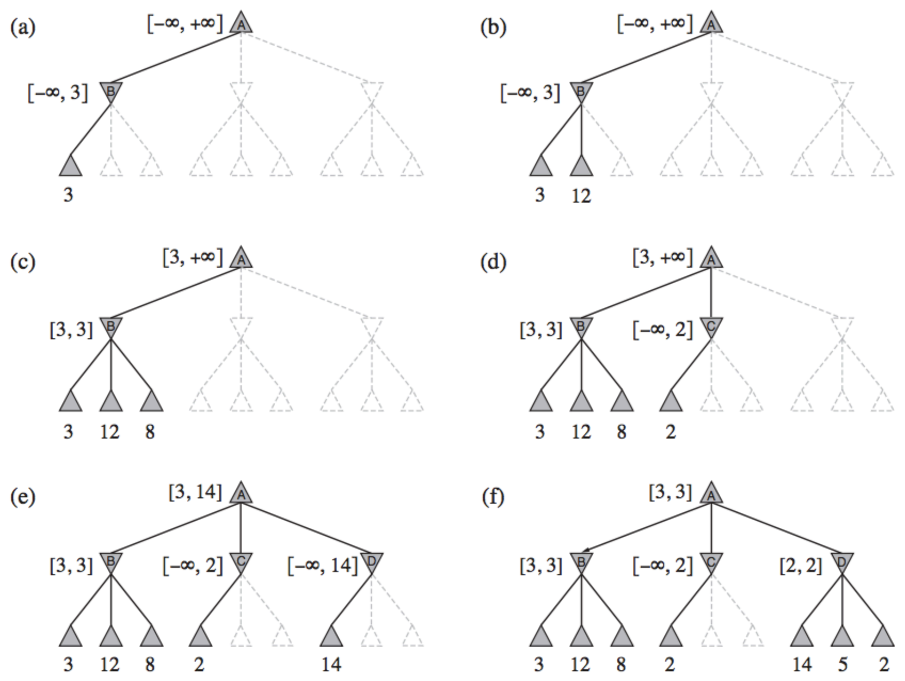
$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Minimax Algorithm

- Perform a complete depth-first search of the game tree.
- Recursively compute the minimax values of each successor state.
- Maximize the worst-case outcome for MAX.

Alpha-Beta Pruning

- A simplification of minimax algorithm.
- Remove (unneeded) part of the minimax tree from consideration.



Alpha: the value of the best (i.e., **highest**-value) choice we have found so far at any choice point along the path for **MAX**.

Beta: the value of the best (i.e., **lowest**-value) choice we have found so far at any choice point along the path for **MIN**.

Alpha-Beta Pruning

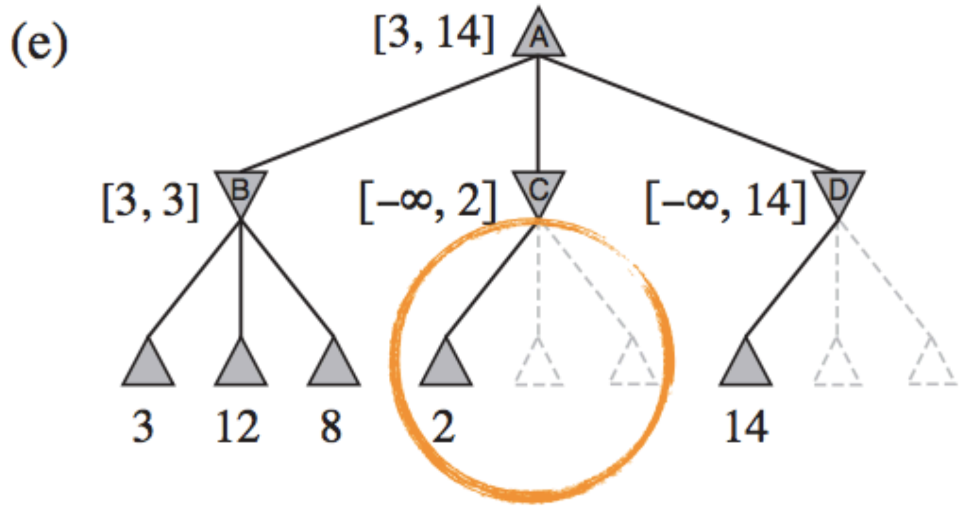
```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

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

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

The search order is important

- It might be worthwhile to examine first the successors that are likely to be the best.



Further acceleration?

- Alpha-beta pruning does **not** affect the final result. (good!)
- With perfect ordering, the time complexity is still high, i.e., O($b^{m/2}$). (bad…)

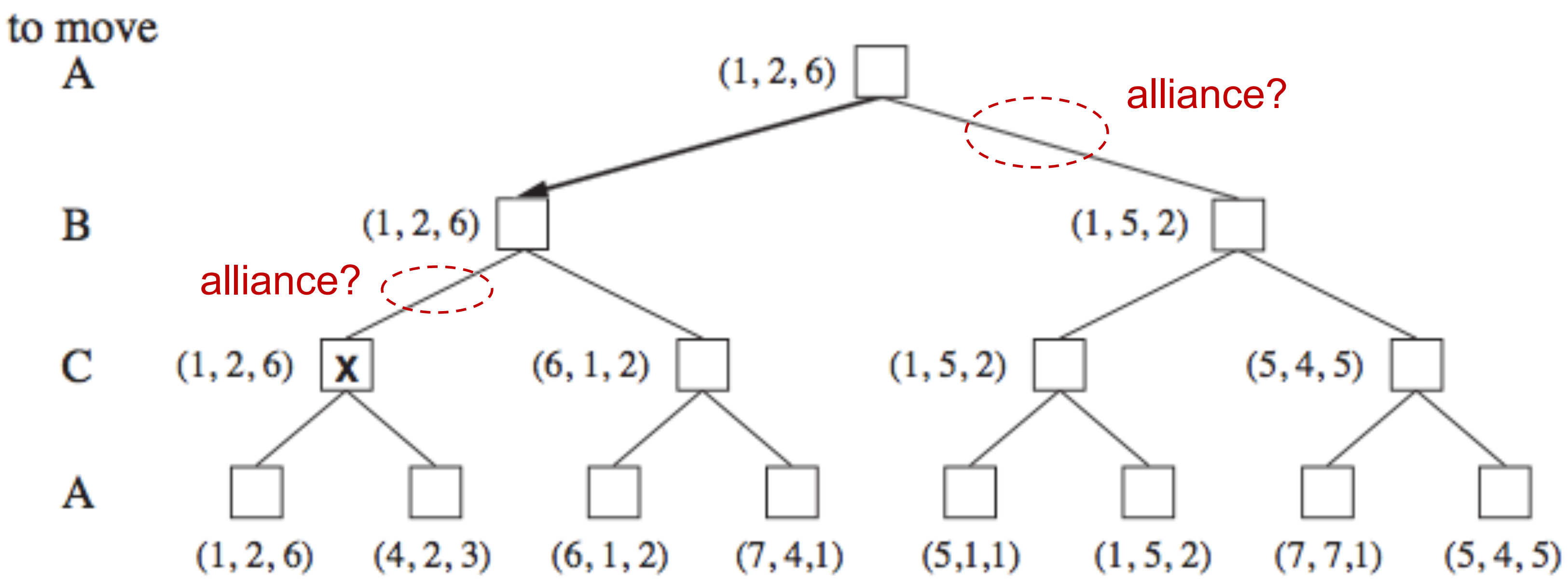
In practice:

- Replace the **UTILITY** function with a heuristic evaluation function **EVAL**
- Use **CUTTOFF-Test** instead of **TERMINAL-Test**

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

More Complex Situations

- games with more than 2 players
- 2-players game that is not zero-sum
- Minimax or Alpha-Beta Pruning don't apply



Question to ask when tackling a search problem

- How to represent the search space?
  - Search Tree (state space)
  - Solution space
- What is the objective function and constraint, and algorithm in textbook already good enough?
- Which algorithmic framework to choose?
  - Tree search, e.g., Un-informed Search, Heuristic Search (A*...)
  - Direct search in the solution space, e.g., Hill Climbing, Simulated Annealing, Genetic Algorithm...
- How to define concrete components of the algorithm framework?
  - General-purpose operators in literature
  - Problem-specific operators, designed based on domain knowledge

Always trade-off among solution quality, efficiency, and your domain knowledge

The End of The **Search** Section.