Agents: concept
right function: performance measure for environment
rationality: maximum "expected utility"
rational agents: no omniscient, no clairvoyant
PEAS: performance measure, environment
actuator(执行), sensor(感应器)
environment type 判断方法:
deterministic(决定的). episodic(不连贯的)
reflex agent can be rational
NP-hard solutions can be verified in polynomials time
PSPACE: canbe solved in polynomial amount of space
EXPTIME: can be solved in exponential time $O(2^{p(n)})$

Search:
search problem consists of:
state space action set Actions(s)
transition model Succ(s,a) start state
step cost function c(s,a,s') goal test
state space graph: search proble数学表示
Nodes: world states Arcs: successors
General tree search
problem, strategy ⇒ solution / failure
initialization
loop do
no candidates ⇒ failure
choose a leaf node to expand (strategy)
node contains goal state ⇒ return
expand the node and adding -
-resulting nodes in the tree
DFS: $O(b^m)$ space: $O(bm)$ no optimal
BFS: $O(b^s)$ space: $O(b^s)$ optimal iff cost≡1
Uniform Search: (find the least-cost)
永远探索最近的点,
time $O(b^{\frac{C^*}{\varepsilon}})$ C^*: solution cost, ε: arc cost at least
C^*/ε tiers: space $O(b^{\frac{C^*}{\varepsilon}})$ optimal!
缺点: explore options in every direction
no information about goal location
Unifor Search
每次都像BFS一样同层扫描, explore cost最小的点,

same property for search algorithm:
1. all fringe are priority queues
2. DFS and BFS can avoid log(n)
3. one implementation ⇒ variable queueing object

Heuristic (manhattan, Enclidean)
admissible heuristic: $h^*(n)$: true cost for nearest goal
$0 \le h(n) \le h^*(n)$
Dominance: $h_a \ge h_c$ if $\forall(n): h_a(n) \ge h_c(n)$
max of admissible heuristic is admissible
consistency of heuristic:
estimated cost ≤ actual costs
heuristic design: often use relaxed problems

A* Search
g(n): path cost h(n): goal proximity
⇒ $f(n) = g(n) + h(n)$
A* search is optimal iff h(n) is admissible
Graph Search
像tree search一样,但是不再访问visited nodes
consistent heuristic ⇒ { A* expanded nodes increase; n比suboptimal更先到达 } ⇒ A* graph search is optimal

Search Games:
Initial state: $s_0 \in S$ Players: Players(s) {1,2,...,N}
Actions: $a \in$ Actions(s) Transition: Succ(s,a) $\in S$
Terminal test: Terminal-Tests(s)
Terminal value: Utility(s, p) for player P
Minimax Search: DFS
① minimax-decision(s):
action = argmax(min-value((succ(s,a))))
return action (in Actions(s)
② min-value(s):
if terminal-Test(s) ⇒ return (Utility(s))
v = +∞
for a in Actions(s):
v = min(v, max-value(Succ(s,a)))
return (v) 在所有值中取min (对手)
③ max-value(s):
if terminal-Test(s) ⇒ return (Utility(s))
v = +∞
for a in Actions(s):
v = max(v, min-value(Succ(s,a)))
return (v) (我)
Another way:
① minimax-decision(s)
return $\arg\max_a$ (value(succ(s,a)))

② value(s)
if terminal-test(s) ⇒ Utility(s)
if Player(s) = MAX ⇒ max(Value(Succ(s,a)))
if Player(s) = MIN ⇒ min(. . .)

minimax properties:
adversay assumed to be rational
minimax is optimal against perfect opponent
time $\in O(b^m)$ Space $O(bm)$
处理 limits: 底层 level 用评估函数来赋值
use iteration deepening for anytime algorithm >> deeper is better, evaluation function is confusing
2. Game tree pruning: 发现没用就剪掉
α: MAX's best choice β: MIN's best choice
max-value(state, α, β):
$v = -\infty$
for Successor in Succ(s,a):
$v = \max(v, \text{min-value}(\text{Successor}, \alpha, \beta))$
if $v \geq \beta \Rightarrow$ return v
$\alpha = \max(\alpha, v)$, return v
min 也一样, 把 $v \geq \beta$ 改成 $v \leq \alpha$, max $\Rightarrow$ min, $\alpha \Leftrightarrow \beta$
time $(O(b^{\frac{m}{2}})$ double solvable depth!

Search Chance
1. Independent Decision-making: 每人分别行动
2. Joint State/Action Spaces: 结合所有 agents
3. Coordinated Decision Making: confirm joint plans
Each Agent propoes their actions and computer
4. Alternate Searching One Agent a Time
Generalized minimax: 每人都有 Utility, 不同的维度
Expectimax
① decision(s):
return $\arg\max_a$ Value(Succ(s,a))
value(s):
If Terminal-test(s) $\Rightarrow$ return Utility(s)
If Player(s)=MAX $\Rightarrow$ return max among actions
If Player(s)=CHANCE $\Rightarrow$ sum$_a$ Pr(a) * Value(Succ(s,a))
Expectiminimax
前代码加一行: If = MIN $\Rightarrow$ return min
Property: minimax are invariant with respect to monotonic
expectiminimax are with positive affine transformation
should take real value
Maximum Expected Utility:
Outcome: 结果 lotteries: distribution Peference: $\succsim$
completness: $L \succsim L'$ or $L' \succsim L$ Transitivity: 可传递
Independence: $(L \succsim L') \Rightarrow [p, L; 1-p, L''] \succsim [p, L'; 1-p, L'']$
Continuity: $(L \succsim L' \succsim L'') \Rightarrow [p, L; 1-p, L''] \sim L'$ $\exists p$
$\succsim$ satisfies 4 axioms iff $L \succsim L' \Leftrightarrow EU(L) \geq EU(L')$
但大部分的 EU 不会满足 $EU(L) = \sum_{o \in O} p_L(o) U(o)$ < concave > convex
Decision Theory: pick a strategy to maximize utility given world outcomes
Game Theory: given other player's strategy

Game Theory
dominant: $\forall k' \neq k, u_i(\pi_{i,k}, \vec{\pi}_{-i}) \geq u_i(\pi_{ik'}, \vec{\pi}_{-i})$ 除 π_i 以外策略合集
对于玩家 i 来说, 无论其他人出什么, 选择策略 k 都 max
strictly dominant: $>$
Nash Equilibrium: $\forall i, u_i(\vec{\pi}) \geq u_i(\pi_i', \vec{\pi}_{-i})$
当其它人都不改变策略时, 没有人会改变
strict nash equilibrium: $\forall i, u_i(\vec{\pi}) > u_i(\pi_i', \vec{\pi}_{-i})$
寻找 pure nash 方法: 找出 dominant $\Rightarrow$ 删掉被 dominant 的
下划线法: 划出每一行and每一列最大值, 有重合即为纳什均衡
Theorem: finite number of players+actions $\Rightarrow \exists$ nash
混合 nash 均衡求法: 设 A 为 $p_1, \dots p_n$, B 为 $q_1, \dots q_n$
$\Rightarrow p_1 + \dots + p_n = 1$, $q_1 + \dots + q_n = 1$ 且 A 的配置需要让 B 觉得每件事期望相同
correlated Equilibrium: 即, 站在上帝视角
$\forall \pi_i' \in \Pi_i$, $\sum_{\pi_2} p_0(\pi_1, \pi_2) u_1(\pi_1, \pi_2) \geq \sum_{\pi_2} p_0(\pi_1, \pi_2) u_1(\pi_1', \pi_2)$
Pareto 最优: 每一个人都达到了最大 outcome

Markov Decision Process
$s \in S$: set of states $a \in A$: set of actions
$T(s,a,s')$: transition probability (如果采取 a, 到 s' 的可能性是多少)
$R(s,a,s')$ / $R(s)$: reward
MDP: non-determinstic problems present 和 future 是独立的
π^*: π give an action for each state. π^* 表达最优
discounting: γ. 每走一步, 幂次乘以 γ
如何解决 infinite Utilities?
1. 限制 horizon (finite step or finite s policy for π)
2. Discounting
$V^*(s)$ value of utility starting in s
$Q^*(s,a)$ value of q-state (s,a)
$V_k(s)$ the optimal value of s in k steps
$O(S^2A)$ and will conveige to optimal
公式大全
$V^*(s) = \max_a Q^*(s,a)$
$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
Bellman: $V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
Value: $V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s)]$
Q-iteration $Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$
Policy extraction $\pi_V(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$
Policy evaluation: $V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')]$
Policy improv: $\pi_{new}(s) = \arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')]$
Policy evaluation: $O(S^2)$ per iteration
value Iteration: $O(AS^2)$ slow / policy converges before max

Reinforcement Learning

receive feedback in the form of rewards, max expected difference from MDP: X T or R

Direct evaluation: average all the sampled values under π

waste information about state connections, long-time

Temperal Difference Learning

实时更新 $V(s)$, policy fix

$sample = R(s,\pi(s),s') + \gamma V^{\pi}(s')$

$V^{\pi}(s) = (1-\alpha)V^{\pi}(s) + \alpha\, sample = V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

exponential moving average: $\alpha\downarrow \Rightarrow$ give converging

TD value Learning problems: 也可以改成 Q learning

$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$

$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\, sample$

Q-learning converges to optimal policy even if acting suboptimally

前提: explore enough; learning rate small enough

Explore Scheme

random actions: ε: act randomly 1-ε: follow policies

$\Rightarrow$ lower ε overtime

Exploration Function: u-value estimate n-visit count

$f(u,n) = u + k/n$ modified Q-Update:

$Q(s,a) = \alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$

Constraint Satisfaction Problems

constraint Graphs:

① each constraint relates 2 variables, arcs are constrain

Search Method:

DFS - BFS - Backtracking Search

Backtracking search is the basic uniformed algorithm

每次检查一个node, 如果不符合constraint就后退

Backtracking Search 伪代码

```
backtracking-search(csp):
    return recursive-backtracking({}, csp)
recursive-backtracking(assignment, csp)
    if assignment is complete ⇒ return assignment
    var = select-unassigned-variable(variable[csp], assignment, csp)
    for value in order-domain(var, assignment, csp):
        if value is consistent < assignment ∪ constraints[csp] ⇒:
            add[var=value] to assignment
            result = recursive-backtracking(assignment, csp)
            if result ≠ failure ⇒ return result
            remove {var=value} from assignment
    return failure
```

改进算法方法:

ordering, filter, structure

① filter: Forward checking: enforcing consistency of arcs pointing to each new assignment

遍历每一个点, 把不符合的踢出去

Consistency

Arc: X→Y is consistent iff for every x有有y满足

Arc consistency of an Entire CSP: all arcs are consistent

伪代码:

```
AC-3(csp): inputs: csp, a binary CSP
local variables: queue, a queue of arcs
  while queue is not empty:
      (A,B) = remove-first(queue)
      if remove-inconsistent(A,B) ⇒
          for node in neighbours[A] then
              add(node, A) to queue
remove-first(A,B) returns true iff succeeds
    remove = false
    for x in Domain[A]
        if no value y in Domain[B] allows(x,y) is consistent
            then delete x from Domain[A]
                 removed = true
    return removed
```

$O(n^2d^3)$ → reduce $O(n^2d^2)$

Ordering: minimum reducing value

优先向于去掉可以留下最少的点,

Least constraining value

优先向于去掉constraints最小的点,

K-consistency: for each k-1 nodes, could be extended

strong k- ··· : k-1, k-2, ... 1 consistent

Tree-structured CSPs

if the constraint graph has no loop, CSP $O(nd^2)$

① order: order variables from parents to children

② remove backward:

for i=n:2, RemoveInconsistent(Parent(X_i), X_i)

for i=1:n, assign X_i consistently with Parent(X_i)

Cutset Conditioning

选一个点, 使得去掉这个点, 就可以变成tree

再当此点为所有情况时遍历考虑.

Tree Decomposition

分割成一个个小问题, 用共同边连接

Iterative Algorithm

variable selection: 随机分配

choose heuristic with min constraints

Climb hilling: local and global and flat

simulated annealing: 给一定机率往 next 随机

simulated-annealing(problem, schedule)
inputs: problem, a mapping from time to "temperature"
for t from 1 to ∞:
T = schedule(t)
if T=0 ⇒ return current
next = randomly selected successor of current
ΔE = value[next] - value[current]
Δ>0 ⇒ current = next
else: current = next only if probability $e^{\frac{\Delta E}{T}}$