搜索

Uniformed Search

大题：给一个search问题，能把搜索过程画出来

- 广度优先搜索
- 一致代价搜索
- 深度优先搜索
- 深度受限搜索
- 迭代加深的深度优先搜索
- 双向搜索（可能不会考）

Informed Search

A*

必考

- tree-search（没有memory）和graph-search（有memory，有额外存储开销）要掌握，工程的A*不考
- 启发式函数的条件，如admissible、consistent的条件，多个admissible，如何生成个更强大的A*
- 大题：
  - 给定算法，搜索过程画图
  - 自主设计启发式函数
  - A* 最优性证明

local search

可能出逻辑判断题

- 模拟退火（偶尔考）
- local beam search 和 遗传算法可以过一下
- 爬山法（大概率会出现）

CSP会考

- 整个求解过程要会写
- 要了解基本的优化方法（比如约束传播，前向检查，向前看）
- 考试难度不会超过作业难度

Game Playing

- Minmax Alpha-Belta剪枝必考！！！！！！！！！！！（难点！高分需要认真看！要多做点题）
- 后面基本不会考，顶多概念

逻辑

逻辑Agent

了解基本概念就行，最多判断题

命题逻辑

- 命题逻辑化CNF
- 可能CNF上用归结做推导

一阶逻辑

- Skorn化变成clause
- 找出最一般合一（必考），不可合一的说不可合一，可以合一的要算出MGU
- 一阶逻辑下用归结反驳的公式推出一些结论

学习

贝叶斯网

必考，大题

- 给你个贝叶斯网，基本的对错判断，算概率（就用变量消元法算就行）
- 判断两个变量是否独立

监督学习

- Decision Tree
- KNN
- 线性预测
- 逻辑回归
- SVM

SVM要花很大精力吃透的东西，可能考大题（证明，或计算，压轴题！！！！！！！！！！！！！！！！！！！！！！！！！！！）

其它主要考小题

非监督学习

- 聚类
- PCA

Tree-Search & Graph Search

Tree-Search

Basic idea:

offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```
function Tree-Search (problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    (根据不同策略选择扩展节点)
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

```
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
```

Graph Search

General Tree Search vs. Graph Search

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

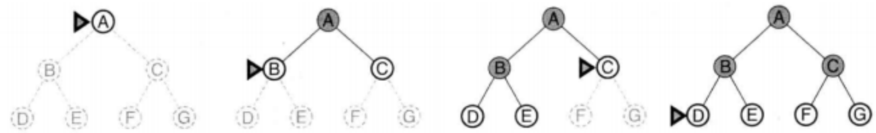
Main variations:

- Which leaf node to expand next
- Whether to check for repeated states
- Data structures for frontier, expanded nodes

Uninformed Search Strategies

广度优先搜索

Breadth-first Search 广度优先搜索



- Frontier 选用 FIFO (first-in, first-out) 队列
- 完备性：完备 (if d is finite)
- 最优性：若每条边 cost 一致 (if cost=1 per step)，则一定返回最优解；否则不一定
- 时间复杂度:
  - 访问节点数 $\leq 1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
  - If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $\leq 1 + b + b^2 + b^3 + \cdots + b^d + b(b^d - 1) = O(b^{d+1})$
- 空间复杂度：$O(b^d)$ or $O(b^{d+1})$ (扩展节点做 goal test)，所有节点均被存储

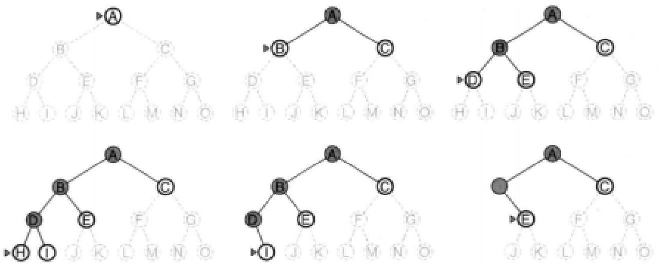
一致代价搜索

Uniform-cost Search 一致代价搜索

- Expand least-cost unexpanded node
- Implementation: Frontier = queue ordered by path cost
- Frontier 选用优先级队列，先扩展 Path-Cost 小的节点；若所有路径 cost 一致，则一致代价搜索等于广度优先搜索
- 完备性：完备，若单步 cost 有下界 ϵ (if step cost $\geq \epsilon$)
- 最优性：最优，nodes expanded in increasing order of $g(n)$
- 时间复杂度：# of nodes with g $\leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$，其中 C^* 代表最优解所需的代价
- 空间复杂度：# of nodes with g $\leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

深度优先搜索

Depth-first Search 深度优先搜索



- Frontier 选用 LIFO (last-in, first-out) 队列
- 完备性：No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path ⇒ complete in finite spaces
- 最优性：不保证最优
- 时间复杂度：访问节点数 $\leq 1 + b + b^2 + b^3 + \cdots + b^m = O(b^m)$
  - terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- 空间复杂度：$O(bm)$, *i.e.*, linear space! 只存储一条根到叶节点的路径，以及该路径上节点未被扩展的兄弟节点

深度受限搜索

Depth-limited Search 深度受限搜索

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

- 目的为了避免深度优先搜索一条路走到黑的尴尬
- 设置一个深度界限 l，若节点深度大于 l 时，不则再扩展
- 返回的结果：
  - 有解
  - 无解
  - 在 l 范围内无解
- 完备性：不是
- 最优性：不保证
- 时间复杂度：$O(b^l)$
- 空间复杂度：$O(bl)$
- Solves infinte-depth path problem
- if $l < d$, possibly incomplete
- If $l > d$, not optimal

迭代加深的深度优先搜索

Iterative Deepening Search 迭代加深的深度优先搜索

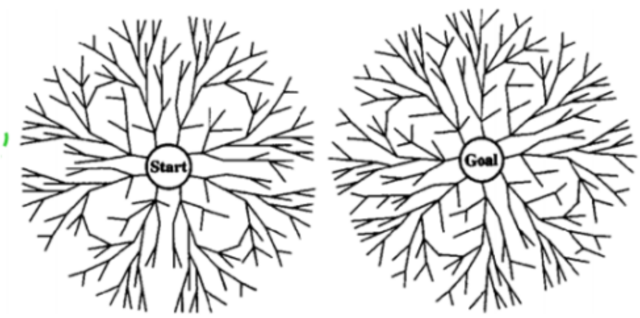
由 Depth-limited search 演化而成，每轮增加深度限制

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem
    for depth 0 to ∞ do
        result DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

- 使用不同的深度界限 $k = 0, 1, 2, \ldots$ 不断重复执行“深度受限搜索”
- 结合了广度优先搜索和深度优先搜索的优点
- 完备性：完备
- 最优性：若每条边 cost 一致 (if step cost $= 1$)，则一定返回最优解；否则不一定
- 时间复杂度：$db + (d-1)b^2 + (d-2)b^3 \cdots + b^d = O(b^d)$
- 空间复杂度：$O(bd)$，深度界限达到 d

双向搜索

Bidirectional Search 双向搜索



- 从初态和终态同时进行广度优先搜索，但其中的难点是：终态可能不好描述（比如 n 皇后问题的终态）；由终态返回的函数可能不好描述
- 完备性：完备
- 最优性：若每条边 cost 一致，则一定返回最优解；否则不一定
- 时间复杂度：$O(b^{d/2})$
- 空间复杂度：$O(b^{d/2})$

最佳优先搜索

tree-search 和 graph-search的一种，根据评价函数来选择扩展节点

Evaluation function h(n) (heuristic function 启发函数)= estimate of cost from n to the closest goal(节点 n 到目标节点的最低耗散路径的耗散估计值)

Greedy Search

视图扩散离目标节点最近的点

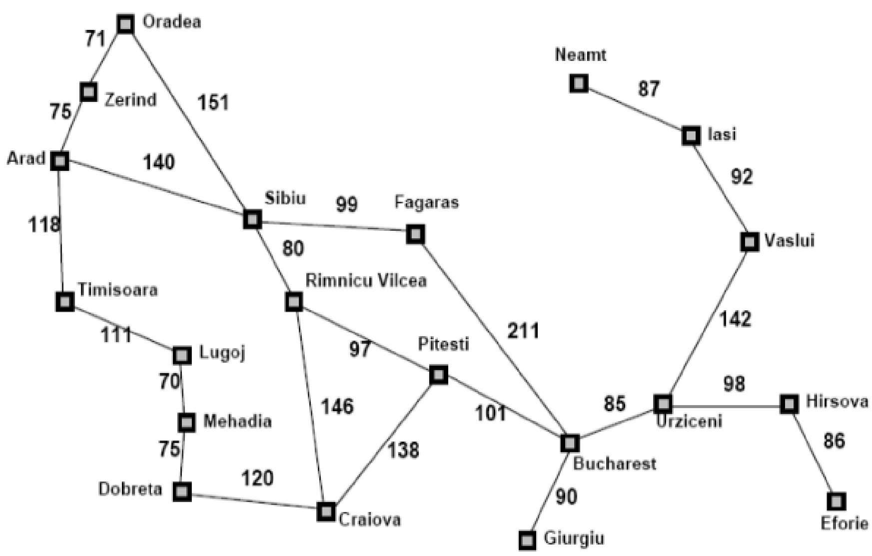
complete? No — can get stuck in loops, e.g. from Iasi to Fagaras,
Iasi→ Neamt→Iasi→ Neamt →
Complete in finite space with repeated-state checking

Time? $O(b^m)$, but a good heuristic can give dramatic improvement

Space? $O(b^m)$ — keeps all nodes in memory
b: Branch factor, d: Solution depth, m: Maximum depth

Optimal? No

- Uniform-cost orders by path cost, or backward cost $g(n)$
- Greedy orders by goal proximity, or forward cost $h(n)$
- A* Search orders by the sum: $f(n) = g(n) + h(n)$



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A*

Evaluation function: $f(n) = g(n) + h(n)$

- $g(n)$ = cost so far to reach n
  到达节点 n 的耗散
- $h(n)$ = estimated cost to goal from n
  启发函数：从节点 n 到目标节点的最低耗散路径的耗散估计值
- $f(n)$ = estimated total cost of path through n to goal
  经过节点 n 的最低耗散的估计函数

- admissible heuristic $h(n) \le h^*(n)$ true cost
  - If h(n) is admissible, A* using TREE-SEARCH is optimal
- consistent euristic: if for every node n, every successor n' of n generated by any action a, $h(n) \le c(n, a, n') + h(n')$
  - Consistency可推得admissibility
  - f(n)沿任何一条路径都是非递减的
  - If h(n) is consistent, A* using GRAPH-SEARCH is optimal
- 如果有一个可采纳启发式的集合 $\{h_1, \ldots, h_m\}$
  $h(n) = max(h_1(n), \ldots, h_m(n))$ 可采纳并比成员启发式更有优势

Local Search

解是目标状态，而搜索路径无关紧要。

保存当前状态并尝试优化它。

爬山法

```
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
```

Random-restart hill climbing overcomes local maxima

模拟退火法

```
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^(Δ E/T)
```

局部剪枝搜索

1. Keep track of *k* states rather than just one
2. Start with *k* randomly generated states
3. At each iteration, all the successors of all *k* states are generated
4. If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

遗传算法

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population **种群**)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function **适应度函数**). Higher values for better states
- Produce the next generation of states by selection, crossover, and mutation (**选择, 杂交, 变异**)

CSP

带单变量赋值的对CSP问题的深度优先搜索叫做backtracking search

```
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
```

以何顺序选择变量?

- 最少剩余值MRV：合法值最少的变量
- 度启发式：给剩余变量增加更多约束

如何给变量赋值?

- 最少约束值：从剩余变量中排除最少值

如何提前检测到不可避免的失败?

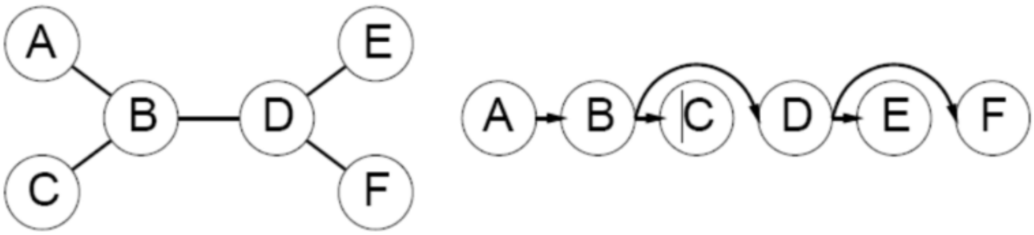
- 前向检查：如果任何未赋值变量值域为空时停止搜索
- 约束传播：用约束条件来减少变量值域
- 弧相容：

```
function AC-3( csp) returns the CSP, possibly with reduced 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

    while queue is not empty do
        (X_i, X_j) ← REMOVE-FIRST(queue)
        if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
            then delete x from DOMAIN[X_i]; removed ← true
    return removed
```

$O(n^2 d^3)$ (but detecting all inconsistencies is NP-hard)

- tree-structured CSP

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



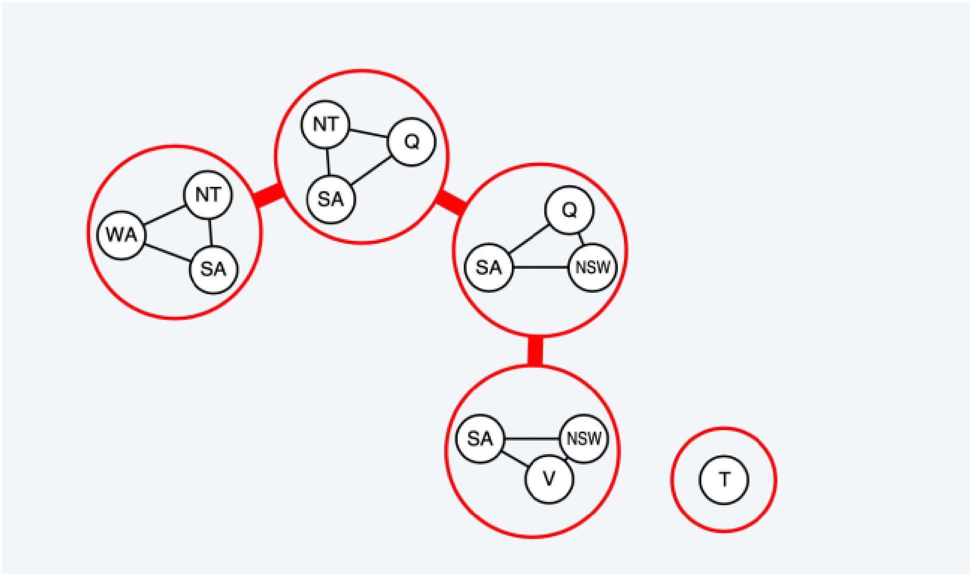
2. Apply arc-consistency to (X_k, X_i), when X_k is the parent of X_i
For i from n down to 2, apply $REMOVEINCONSISTENT(Parent(X_i), X_i)$

3. Now one can start at X_1 assigning values from the remaining domains without creating any conflict in one sweep through the tree!
For i from 1 to n, assign X_i consistently with $Parent(X_i)$

Complexity: $O(n \cdot d^2)$

- 树分解:

- Decompose problem into a set of connected sub-problems, where two sub-problems are connected when they share a constraint
- Solve sub-problems independently and combine solutions



Game Playing

Minimax 算法

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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

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

$\alpha - \beta$剪枝

α是到目前为止在路径上的任意选择点发现的 MAX 的最佳（即最大值）选择

β是到目前为止在路径上的任意选择点发现的 MIN 的最佳（即最小值）选择

Whenever β < α, the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play

```
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
```

逻辑Agent