2. Search 搜索与规划

1. 搜索问题可以由五个部分定义：

- initial state
- possible actions(and state associated actions)
- transition model
- goal test
- path cost

2. 问题假定

- observable states 完全可观测 精确知道自己当前的状态，如所处的位置等等
- discrete states
- deterministic transition 确定性转移，执行一个action会得到确定的state

3. 图上的搜索

1. the search doesn't change the world, only actions change the world
2. essence of search: following up one option now and putting the others aside
3. 伪代码

```
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
        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
```

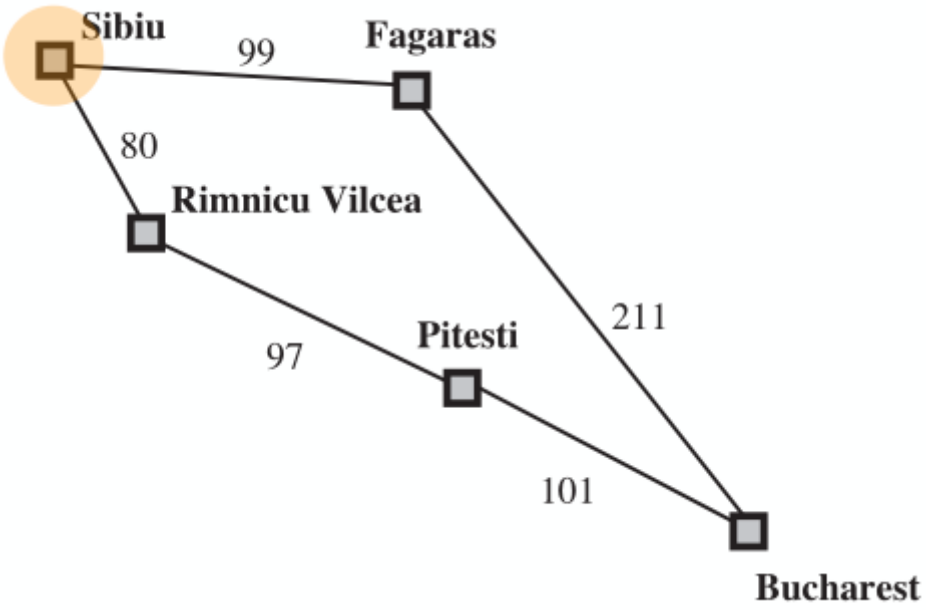
4. all search algorithms share this tree search structure they vary primarily according to how they choose which state to expand --- the so-called search strategy
5. 注意上面的搜索代码中，是在取出状态后才判断是否达到目标，而不是在扩展时判断。（具体参考书中一致代价搜索的例子）
6. 搜索策略的评估
    - completeness 是否一定会找到解
    - 时间和空间复杂性
    - 最优性 找到的是否一定是最优解

4. 无信息搜索策略

1. 无信息搜索仅使用问题定义的信息

- BFS
- DFS
- 一致代价搜索Uniform-cost search
- 深度受限的搜索
- 迭代加深搜索

2. BFS
   - 扩展深度最小的未扩展的节点
   - 评估：
     - 一定会结束
     - 时间空间复杂度均为$O(b^{d+1})$，b是每个节点的最大分支数，d是最优解的深度
     - 在边权重为1时，最优
   - 使用先进先出队列实现
3. DFS
   - 扩展最深的未扩展的节点
   - 评估
     - 可能在无限深度时，不会停止，并且要注意标记已经visit过的点
     - 时间复杂度$O(b^m)$，m为最大深度
     - 空间$O(bm)$
     - 不是最优（这里说的是第一次搜索到的结果不一定最优，而不是说将所有状态搜索完成后得不到最优解）
   - 使用后进先出队列实现
4. 一致代价搜索
   - 使用优先队列实现
   - 例子（注意跑的时候要在取出状态后再判断是否达到终点，如果在加入状态时判断，会把上面的路判断为最优）



   - 评估

        - 一定可终止，如果每条边权值为正
        - 时间和空间复杂度与最优解的权重有关
        - 一定最优
5. 深度受限的搜索
    - 伪代码

```
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 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
```

6. 迭代加深搜索
    - 循环加深深度受限搜索的最大深度
    - 虽然看起来浅层的很多节点被多次展开，但在渐进意义下，其时间复杂度与BFS相同而空间复杂度与DFS相同
    - 在单位权重下，会得到最优解
7. 总结

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

5. 启发式（有信息）的搜索策略

1. greedy
    - 评估
        - 不一定会终止，如果状态空间有限，并且检查环的话，可以终止
        - 时间空间复杂度均为$O(b^m)$
        - 不是最优
2. A^*
    - 思想：避免扩展代价已经很大的边
    - 估值函数$f(n) = g(n) + h(n)$

    - $g(n)$是当前代价
    - $h(n)$是对从当前位置到目标位置的代价的评估
    - $f(n)$是经过n从起点到终点的总代价
  - A^*使用可接受的启发(admissible)函数，即$h(n) \leq h^*(n)$，其中$h^*(n)$是真实的代价
  - A^*最优：可接受和一致性(admissible and consistency)：这里是重点，要理解为什么A^*最优
    - key：the goal state on the optimal path has a smaller value than that on any sub-optimal paths
    - admissble，保证$h(n) \leq h^*(n),\ f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G(1)) \leq g(G(2))$，所以最优路径一定会在其他路径前面优先展开
    - consistent，$h(n) \leq c(n, a, n') + h(n'),\ n'$为中间节点，保证沿着每一条路径，$f(n)$都是非递减的
  - 估值函数$h(n)$越接近真实的值越精确，如果我们有两个满足条件的估值函数$h_a(n), h_b(n)$，可以令$h(n) = max(h_a(n), h_b(n))$获得效果更好的估值函数
  - 估值函数可以从一个简化的问题中的得到，一个简化版本问题的最优解不会比复杂问题的最优解更好

6. Adversarial search

1. Minimax搜索
   - 伪代码

```
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
```

   - 评估
     - 一定结束
     - 如果对手每步都是最优，则会得到最优结果，但如果对手每步不一定最优，至少可以保证得到的结果不比minimax得到的结果差
2. Alpha Beta pruning
   - α是当前MAX得到的最优解，β是当前MIN得到的最优解，
   - 伪代码

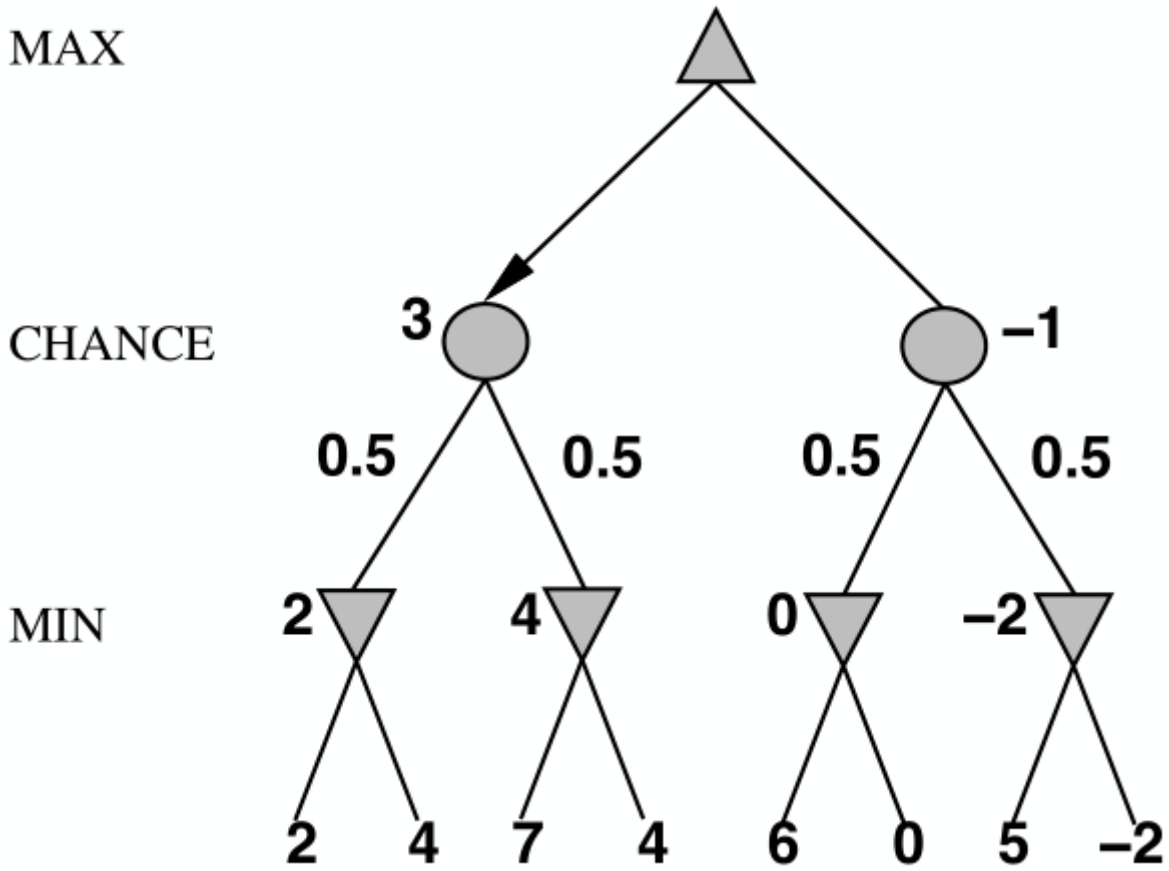
```
function ALPHA-BETA-SEARCH (state) returns an action
    v ←MAX-VALUE (state,−∞,+∞)
    return the action in ACTIONS (state) with value v

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
```

- 性质
  - 剪枝不会影响最终结果
  - 好的搜索顺序可以改进剪枝的效率
  - 完美的剪枝顺序下，时间复杂度为$O\left(b^{\frac{m}{2}}\right)$
  - 在资源受限的条件下，使用剪枝和评估函数减少需要搜索的状态空间

3. 线性评估函数
   - 用当前各个状态的加权和表示当前状态的估值
4. 不确定性动作的搜索时，在每个action之后也会分支，用期望代表分支的值
   -



- $\textsc{Expectiminimax}(s) =$

$$\begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

5. 非完全信息搜索

7. Bandit Search

1. 问题：Multiple arms. Each arm has an expected reward, but unknown, with an unknown distribution, Maximize your award in fixed trials.
2. $\epsilon - greedy$
   - with ϵ probability, try a random arm, with $1 - \epsilon$ probability, try the best arm
   - 选取最优行动时可以使用Upper-confidence bound，即average reward + upper confidence bound, $Q(k) + \sqrt{\frac{2lnn}{n_k}}$，这样会给一些尝试次数少的action更多的奖励
   - 如果要搜索的树比较深，目前还没有估值Q，可以通过多次的rollout获得当前节点的估值

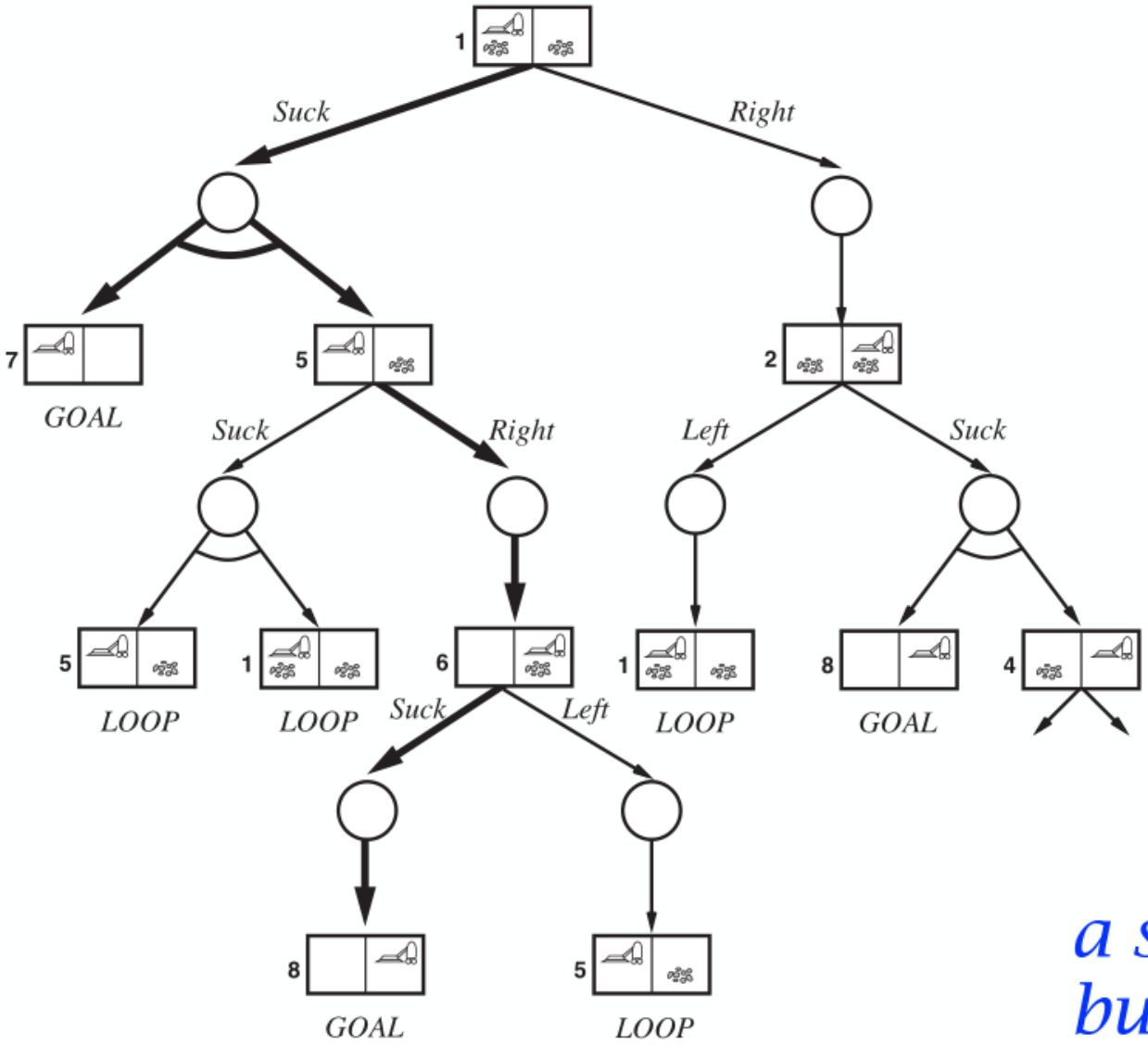
8. Monte-Carlo Tree Search（MCTS）

（对这个算法的理解似乎不太正确）
MCTS中，每个节点包含三个信息，状态，被访问次数，估值

1. 过程
    - 选择 从根节点开始选择一个最右的叶子节点，使用bandit的策略，
    - 扩展 选择了节点后，如果这个节点还没有到达游戏结束状态，则随机选择一个action
    - 模拟 执行action之后得到新的状态，之后rollout， rollout的策略要足够快速，能够短时间重复执行，比如可以随机下，得到当前节点的估值
    - 更新 向上更新所有父节点的估值
2. 在有限次尝试之后，可以达到最优
3. 相比于alpha-beta剪枝，不需要估值函数，估值函数的设计往往是一个难点

9. 不同环境的性质

1. 非确定性行动
    - action会按照概率导致不同的结果
    - AND_OR Search
        - OR node，表示不同的action
        - AND node，表示不同的转移
        - 得到的解不是一条路径而是一棵树
        -



- **function** AND-OR-GRAPH-SEARCH($problem$) **returns** *a conditional plan, or failure*
  OR-SEARCH($problem$.INITIAL-STATE, $problem$, [])

  ---

  function OR-SEARCH($state, problem, path$) **returns** *a conditional plan, or failure*
  if $problem$.GOAL-TEST($state$) **then return** the empty plan
  if $state$ is on $path$ **then return** *failure*
  for each $action$ **in** $problem$.ACTIONS($state$) **do**
  $plan \leftarrow$ AND-SEARCH(RESULTS($state, action$), $problem$, $[state \mid path]$)
  if $plan \neq failure$ **then return** $[action \mid plan]$
  return *failure*

  ---

  function AND-SEARCH($states, problem, path$) **returns** *a conditional plan, or failure*
  for each s_i **in** $states$ **do**
  $plan_i \leftarrow$ OR-SEARCH($s_i, problem, path$)
  if $plan_i$ = *failure* **then return** *failure*
  return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** ... **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

10. 其他 General Solu2on Space Search and CSP

1. 连续空间
   - 连续空间中的搜索可以先离散化，然后进行梯度下降
2. 元启发式搜索算法 Meta-heuristic
   1. 模拟退火算法
   - 伪代码

```
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
```

   - 温度较高时，有随机选择action，温度降低后，开始向最优方向逼近，
   2. Local beam search
   - 这个课件里讲的很不清楚，之后补充
   3. 遗传算法
   -

Encode a solution as a vector,

1: $Pop \leftarrow n$ randomly drawn solutions from $\mathcal{X}$
2: **for** t=1,2,... **do**
3: $Pop^m \leftarrow \{mutate(s) \mid \forall s \in Pop\}$, the mutated solutions
4: $Pop^c \leftarrow \{crossover(s_1, s_2) \mid \exists s_1, s_2 \in Pop^m\}$, the recombined solutions
5: evaluate every solution in Pop^c by $f(s)(\forall s \in Pop^c)$
6: $Pop^s \leftarrow$ selected solutions from Pop and Pop^c
7: $Pop \leftarrow Pop^s$
8: **terminate** if meets a stopping criterion
9: **end for**

mutation: some kind of random changes
crossover: some kind of random exchanges
selection: some kind of quality related selection

4. 元启发式算法性质
    - 不需要函数可导
    - 收敛性

3. CSP Constraint satisfaction problems
    1. 例子
        - 地图染色问题
        - 作业调度问题
    2. 约束传播 CSP中的推理

        在CSP中，算法可以做一种称为约束传播的特殊推理，通过约束来减小一个变量的合法取值范围。从而影响与这个变量关联的变量的取值，如此进行。其核心思想是局部相容性。
    3. 相容性
        - 结点相容，单个变量的所有取值满足约束，则这个变量是结点相容的
        - 弧相容，对变量X_i, X_j，若对D_i中的每个数值在D_j中都存在一些数值满足弧(X_i, X_j)的二元约束，则称X_i相对X_j是弧相容的。
        - 弧相容算法AC-3，维护了一个弧相容集合，首先弹出一个弧(X_i, X_j)，先使X_i相对X_j是弧相容的，如果D_i没有变化，则继续处理，若变化，则需要把每个指向X_i的弧(X_k, X_i)都重新插入队列准备检验，因为D_i的缩小可能导致D_k的缩小，
        -

```
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
```

- 如果弧相容，则得到的CSP与原问题等价，且由于变量取值更少，所以求解更快
- 其实感觉有点像很多图中的问题按照拓扑序或者逆拓扑序求解会更快

4. 回溯法，但无信息的回溯往往是低效的，如何改进
   - minimum remaining values，选择有最少合法取值最少的变量，因为这个变量最可能导致失败进而剪枝
   - Degree heuristic，上面的策略对第一个着色区域是没有帮助的，选择有最多限制的点（例如与最多块相邻的地图块），这个方法对打破僵局很有用
   - Least constraining value，优先选择值给邻居变量留下更多选择，前两个是帮助在搜索中决定下一步选择哪个变量，而这个是帮助变量选择合适的取值
5. 问题的结构
   - 树结构的CSP问题，可以按照拓扑序进行赋值
   - 可以先对部分变量赋值，使剩下的变量能够形成一棵树