



人工智能

Artificial Intelligence

极夜酱

目录

1	搜索	1
1.1	人工智能	1
1.2	状态空间	5
1.3	盲目搜索/无信息搜索	8
1.4	盲目搜索策略	12
1.5	启发式搜索	20
1.6	局部搜索	24
2	约束满足问题	29
2.1	约束满足问题	29
2.2	约束传播	33
2.3	回溯搜索	35
2.4	CSP 局部搜索	38
3	对抗搜索	39
3.1	对抗搜索	39
3.2	Minimax	41
3.3	Alpha-Beta 剪枝	45

Chapter 1 搜索

1.1 人工智能

1.1.1 人工智能 (Artificial Intelligence)

我们称自己为智人，几千年来，我们一直试图了解我们是如何思考和行动的。

AI 企图了解智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器，该领域的研究包括机器人学、语言识别、图像识别、自然语言处理和专家系统等。

人类通过器官（眼、耳、鼻、口、皮肤）从外部环境进行感知，并做出一系列行为（说话、吃饭、移动、决策等）。

智能 Agent 通过传感器（麦克风、摄像头、陀螺仪、声呐等）从外部环境进行感知，从而进行规划、分类、预测等行为。

一个理性的 Agent (Rational Agent) 需要能进行理性的思考和行为，即在现有环境下，做出利益最大化的行为。

1.1.2 AI 的演变

基于逻辑的 AI (Logic-Based AI)

最初的人工智能是基于逻辑学的，数理逻辑的思想和方法一直在知识表示中发挥着重要作用。基于逻辑的 AI 的领域包括常识推理、溯因推理、归纳推理、计算逻辑、概率、规划、决策等。

建立在命题逻辑和谓词逻辑上的 AI 在九十年代前进入了寒冬，直到大量统计学方法引入之后才有了现在大热的机器学习。

专家系统 (Expert System)

专家系统是一个智能计算机程序，其内部含有大量的某个领域专家水平的知识与经验，它根据系统中的知识与经验，进行推理和判断，模拟人类专家的决策过程，以便解决那些需要人类专家处理的复杂问题。

例如医疗诊断系统，它依靠病人的具体病况作为条件来分析，系统可根据病人的病情描述，在已有的知识与经验中，匹配最有可能的病症和治疗方案。

虽然专家系统看起来已经发展了很久，但是它并没有那么可靠。例如在实际的看病过程中，往往医生在除了问询之外，还有自己做推断的部分，目前的专家系统还没有办法做到对病人的病况了解的足够清晰。

其次专家系统的一大问题就是其潜在的风险性，包括无人驾驶汽车也有类似的问题。在发生问题之后，谁来背负责任呢？比如某患者采用了一个医疗专家系统，然后听从专家系统的指示，可结果病情却越来越重。

机器学习 (Machine Learning)

机器学习是一门多领域交叉的学科，涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科，专门研究计算机怎样模拟或实现人类的学习行为，以获取新的知识或技能，重新组织已有的知识结构使之不断改善自身的性能。

例如在自动驾驶中需要通过机器视觉不断看到周围的物体，然后通过机器学习来辨识出这些物体是什么（人、车、交通灯等）。学会辨识图片的过程需要输入大量的车辆的图片给机器学习，最后它就会知道这些图片代表的是车。



图 1.1: 自动驾驶

机器学习最重要的是预测，比如通过大量的图片学会什么是车后，再来一张没有见过的车辆的图片，我们希望机器可以做出正确的判断。因此机器学习包含训练和预测两个部分，并且要保证一定的正确率。

深度学习 (Deep Learning)

深度学习是一种深层次的学习，以教儿童认字为例，按照字从简单到复杂的顺序，让儿童反复看每个字的各种写法，并自己临摹。看得多了，自然就记住了。下次再见到同一个字，就很容易能认出来。认字时，一定是儿童的大脑在接受许多遍相似图像的刺激后，为每个字总结出了某种规律性的东西，下次大脑再看到符合这种规律的图案，就知道是什么字了。

计算机在识别时，也要先把每一个图案反复看很多很多遍，然后在总结出一个规律，以后计算机再看到类似的图案，只要符合之前总结的规律，计算机就能知道这是什么图案。用专业的术语来说，计算机用来学习的、反复看的图片叫训练数据集。

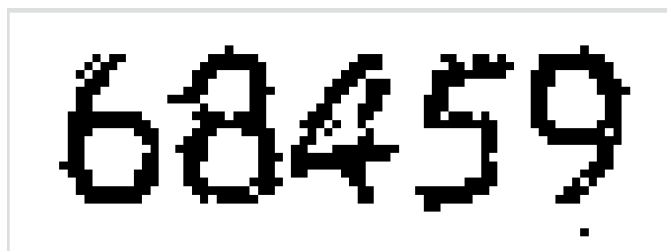


图 1.2: 验证码

深度学习与机器学习的主要区别是在于性能。当数据量很少的时候，深度学习的性能并不好，因为深度学习算法需要大量数据才能很好理解其中蕴含的模式。因此深度学习算法严重依赖高端机，而传统的机器学习算法在低端机上就能运行。深度学习需要 GPU（Graphics Processing Unit）进行大量的矩阵乘法运算。

1.2 状态空间

1.2.1 智能 Agent

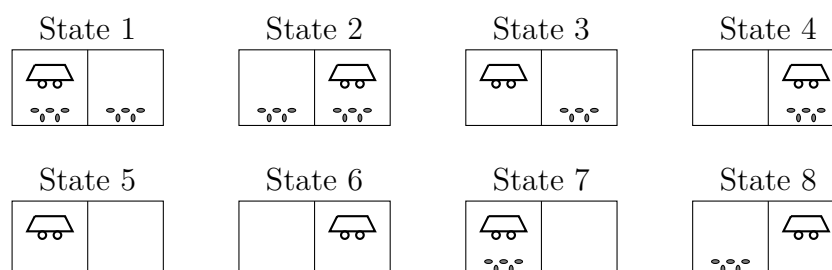
智能 Agent 的设计取决于一系列因素：

- 环境：静态 (static)、动态 (dynamic)
- 表示方案 (representation scheme)：状态 (state)、特征 (feature)、关系 (relation)
- 可观察性 (observability)：完全可观察、部分可观察
- 参数类型：离散 (discrete)、连续 (continuous)
- 不确定性 (uncertainty)：确定性 (deterministic)、随机性 (stochastic)
- 学习：知识是给定的 (已知的)，知识是学来的 (未知的)
- Agent 数量：单 Agent、多 Agent

1.2.2 状态空间 (State Space)

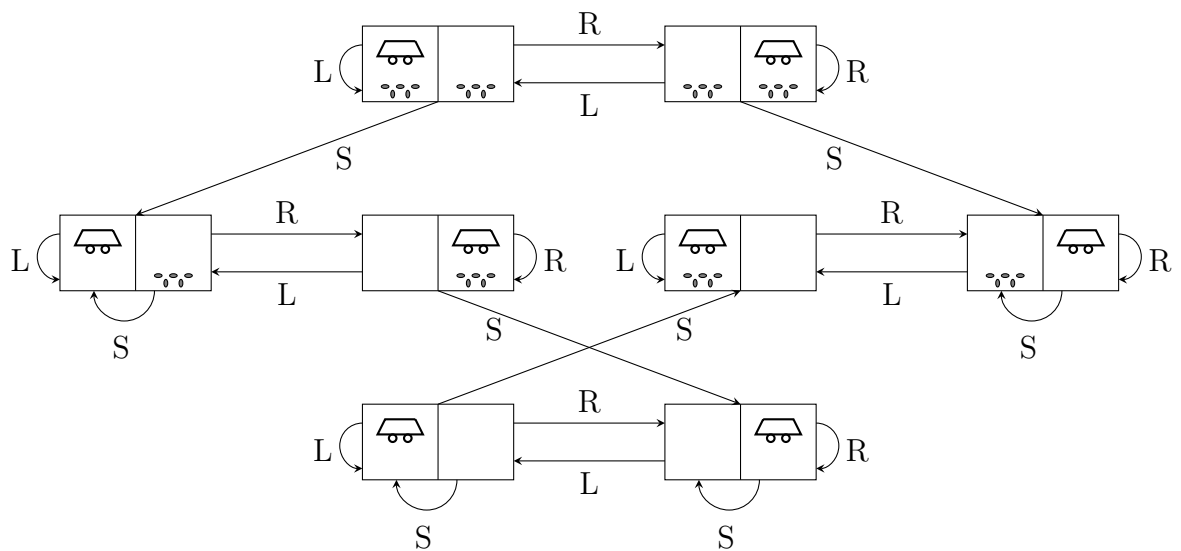
假设有两个有灰尘的房间 A 和 B，吸尘器一开始位于房间 A，吸尘器只能进行左移、右移和吸尘三个操作，最终的目标是将两个房间内的灰尘清扫完毕。

因此根据房间有无灰尘和吸尘器的位置，一共存在 8 种状态。



状态	房间 A	房间 B	吸尘器位置
1	脏	脏	A
2	脏	脏	B
3	干净	脏	A
4	干净	脏	B
5	干净	干净	A
6	干净	干净	B
7	脏	干净	A
8	脏	干净	B

假设以状态 6 作为目标状态 (goal state)，可以得到状态空间图：



提取问题中的特征：

- $status = \{clean, dirty\}$
- $location = \{A, B\}$
- $actions = \{suck, left, right\}$

吸尘器的处理过程可以表示为：

Algorithm 1 VacuumCleaner

```
1: procedure VACUUMCLEANER(status, location)
2:   if status = dirty then
3:     return suck
4:   if location = A and status = clean then
5:     return right
6:   if location = B and status = clean then
7:     return left
```

1.2.3 性能评估 (Performance Measure)

算法的性能评估包含以下几个方面：

- 完备性 (completeness)
 - 完备 (Complete)：算法可以到达目标状态
 - 不完备 (incomplete)：算法无法到达目标状态
- 正确性
- 最优性 (optimality)：算法以最优解（最小代价）找到到达目标状态的路径
- 时间复杂度 (time complexity)：求解所需的时间
- 空间复杂度 (space complexity)：求解所需的存储空间

1.3 盲目搜索/无信息搜索

1.3.1 盲目搜索/无信息搜索 (Uninformed Search)

当需要采取的正确行动不是显而易见时，Agent 就需要提前规划，考虑能够通往目标状态的一系列行动，Agent 所进行的计算过程被称为搜索。

对于一个问题，首先需要确定它的初始状态 (initial state) 和目标状态，其次需要对问题进行建模 (modeling)。一个问题的建模包括 6 个部分：

1. 状态集合
2. 初始状态
3. 行为集合 $Actions(s)$ ：在状态 s 时所有合法的行为集合
4. 状态转移 $Result(s, a)$ ：返回在状态 s 时执行行为 a 后的状态
5. 目标状态：使用 $IsGoal(s)$ 来判断状态 s 是否为目标状态
6. 行为代价函数 (Action Cost Function)：在某个状态执行某个行为所需的代价，使用 $ActionCost(s, a, s')$ 表示在状态 s 执行行为 a 到达状态 s' 所需的代价

在八数码 (8-puzzle) 问题中，在一个 3×3 的方格中，有数字 1-8 和一个空白，最终需要将数字依次排列，使空白位于最后的位置。

八数码问题一共存在 $9! = 362880$ 种状态，其中初始状态为任意一种摆放布局，目标状态为依次排列的布局。

State 1			State 2				State 362880		
2	7	4	2	7	4		1	2	3
5		8		5	8	4	5	6
3	1	6	3	1	6		7	8	

行为包括可以将空白进行上移 (U)、下移 (D)、左移 (L)、右移 (R)，每次移动的代价可以认为都为 1，例如对初始状态中的空白进行左移，其状态转移可表示为 $s_2 = Result(s_1, L)$ 。

在罗马尼亚地图 (Romania map) 中，例如初始位置位于 Arad，最终想要到达 Bucharest。问题的状态为所有的城市，每次行动的代价为城市间的距离。

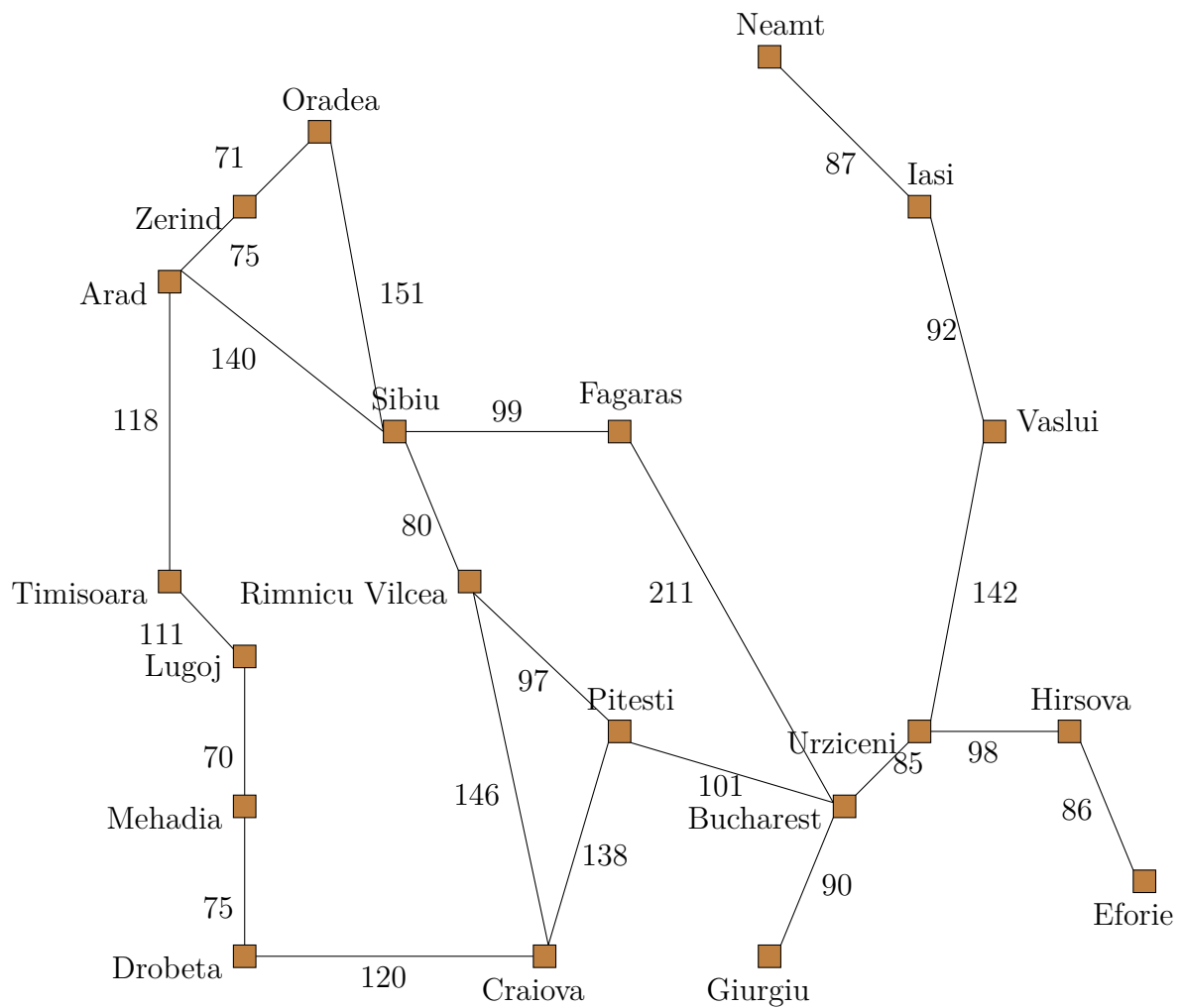


图 1.3: 罗马尼亚地图

1.3.2 搜索树 (Search Tree)

搜索树中的每个结点对应于状态空间中的一个状态，边对应着行为，根结点对应于初始状态。搜索树从根结点开始，每次选择一个结点进行扩展 (expand)，扩

展的过程是递归的，直到搜索到目标结点。

搜索树中的每个结点由 4 部分组成：

1. node.State: 状态信息 (如 Sibiu)
2. node.Parent: 父结点 (如 Arad), 用于回溯
3. node.Action: 从父结点到当前结点的行为 (如 ToSibiu)
4. node.PathCost: 从根结点到当前结点的代价, 用于查找最优解

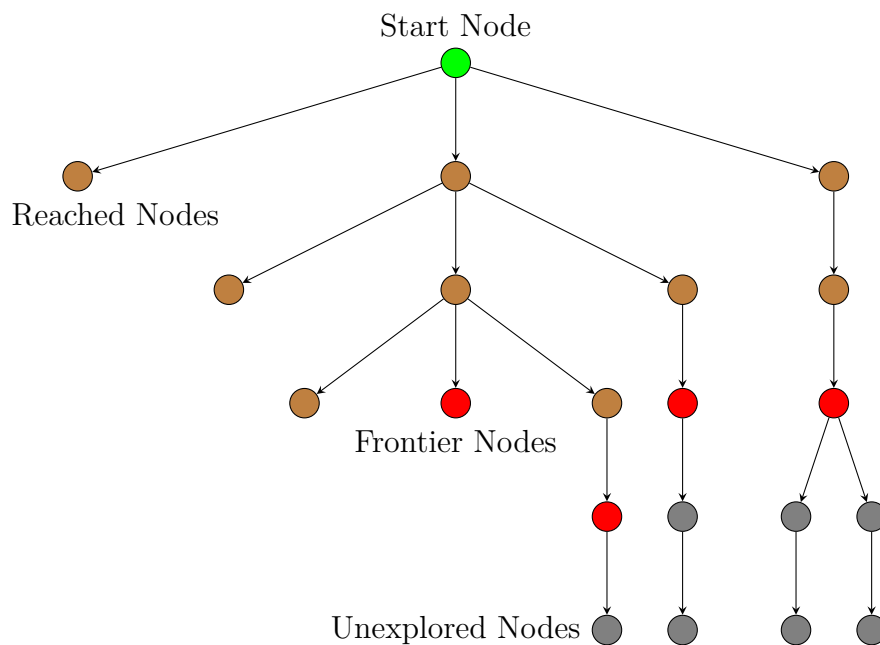
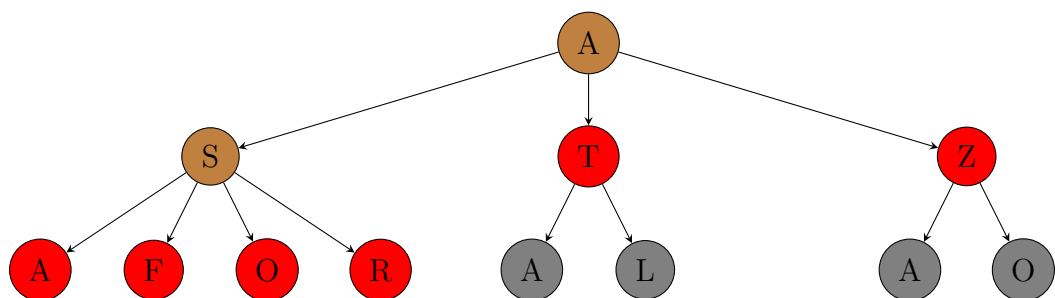
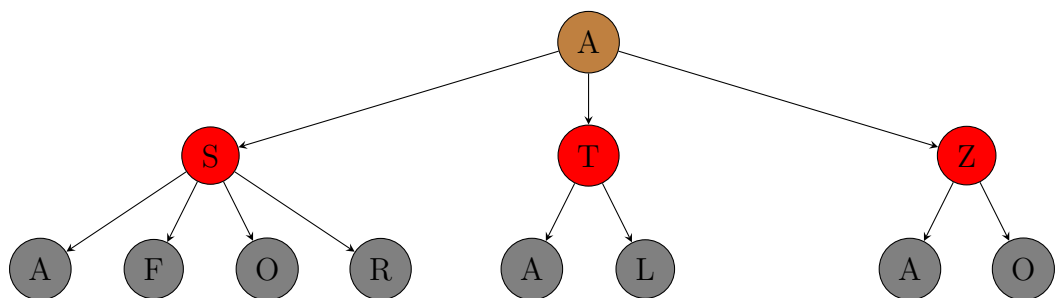
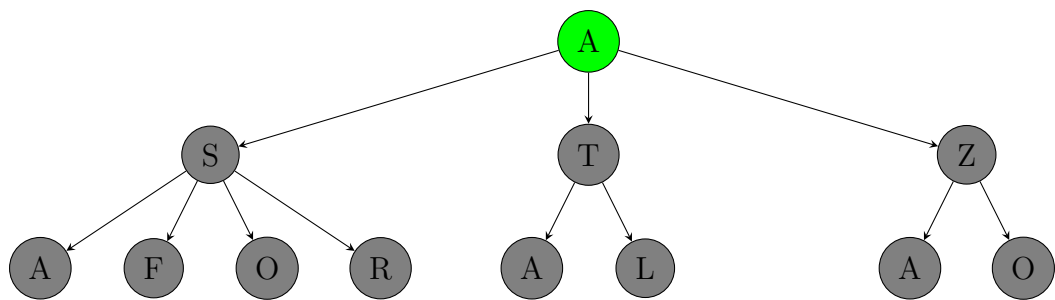


图 1.4: 搜索树

Algorithm 2 Expand a node

```
1: procedure EXPAND(problem, node) yield nodes
2:   s = node.State
3:   for action in problem.Actions(s) do
4:     s' = problem.Result(s, action)
5:     cost = node.PathCost + problem.ActionCost(s, action, s')
6:     yield Node(State=s', Parent=node, Action=action, PathCost=cost)
```



所有已经到达的结点（reached node）都保存在一个哈希表中，以便于避免重复搜索。

而所有前沿结点（frontier node）根据不同的搜索策略，可以使用栈、队列或优先队列保存，从而觉得后续需要扩展的结点。

1.4 盲目搜索策略

1.4.1 广度优先搜索 (BFS, Breadth-First Search)

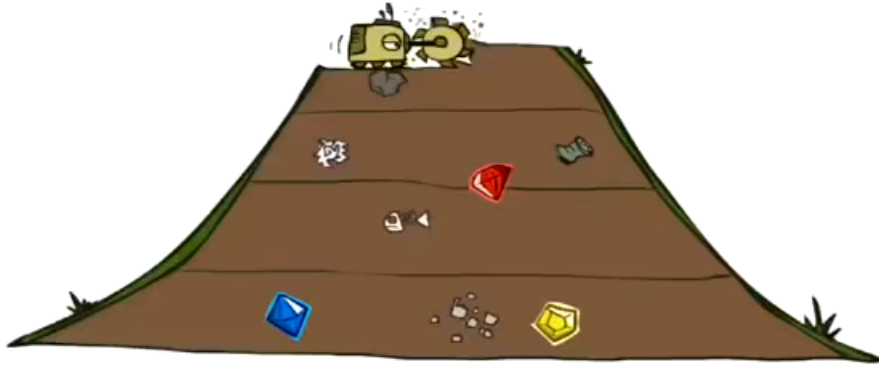
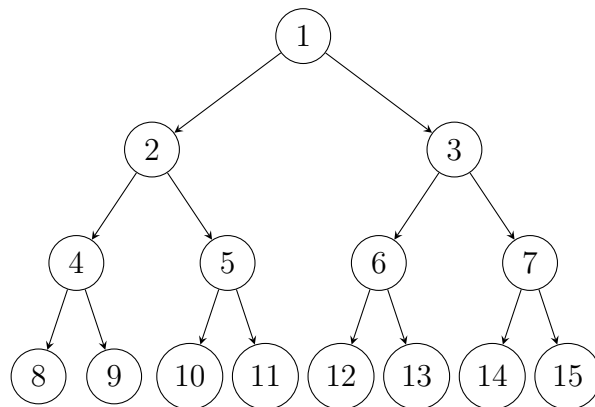


图 1.5: BFS

BFS 首先把从源点相邻的顶点遍历，然后再遍历稍微远一点的顶点，再去遍历更远一点的顶点。



Algorithm 3 BFS

```
1: procedure BFS(problem) returns a solution node or failure
2:   node = Node(problem.Initial)
3:   if problem.IsGoal(node.State) then
4:     return node
5:
6:   frontier = Queue(node)
7:   reached = {problem.Initial}
8:
9:   while not frontier.is_empty() do
10:    node = frontier.dequeue()
11:    for child in Expand(problem, node) do
12:      s = child.State
13:      if problem.IsGoal(s) then
14:        return child
15:      if s is not in reached then
16:        reached.add(s)
17:        frontier.enqueue(child)
18:
19:   return failure
```

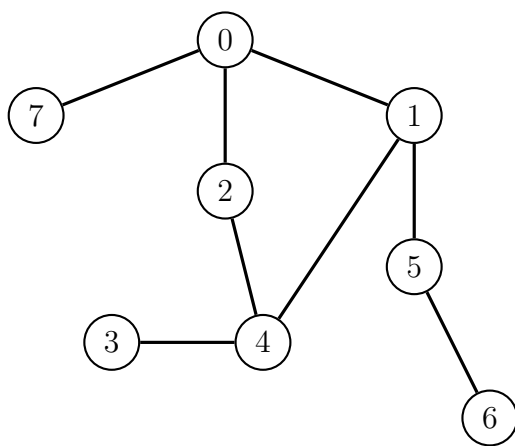
- 时间复杂度: $O(b^d)$
- 空间复杂度: $O(b^d)$ (最多需要保存最后一层结点的数量)
- 完备性: YES (当 d 有穷)
- 最优性: YES (总能找到最浅的目标结点)

1.4.2 深度优先搜索 (DFS, Depth-First Search)



图 1.6: DFS

深度优先搜索是一种一头扎到底的遍历方法，选择一条路，尽可能不断地深入，遇到死路就回退，回退过程中如果遇到没探索的支路，就进入该支路继续深入。



Algorithm 4 DFS

```
1: procedure DFS(problem) returns a solution node or failure
2:   node = Node(problem.Initial)
3:   if problem.IsGoal(node.State) then
4:     return node
5:
6:   frontier = Stack(node)
7:
8:   while not frontier.is_empty() do
9:     node = frontier.pop()
10:    if problem.IsGoal(s) then
11:      return node
12:
13:    if not IsCycle(node) then
14:      for child in Expand(problem, node) do
15:        frontier.push(child)
16:
17:  return failure
```

- 时间复杂度: $O(b^m)$ (当 m 有穷)
- 空间复杂度: $O(bm)$ (只需要保存一层的兄弟结点)
- 完备性: NO (当 m 无穷)
- 最优性: NO (只能找到最左解)

1.4.3 深度限制搜索 (DLS, Depth-Limit Search)

DLS 是 DFS 的变体, 它限制了访问的深度。因为当树的子树过深时, 进行 DFS 会在该子树上消耗太多的时间, 所以在 DLS 中, 若该结点的深度大于限制深度 l , 那就不再继续遍历其子树了。

Algorithm 5 DLS

```
1: procedure DLS(problem, l) returns a solution node or failure or cutoff
2:   frontier = Stack(problem.Initial)
3:   result = failure
4:
5:   while not frontier.is_empty() do
6:     node = frontier.pop()
7:     if Depth(node) > l then
8:       result = cutoff
9:     else if not IsCycle(node) then
10:      for child in Expand(problem, node) do
11:        frontier.push(child)
12:
13:   return result
```

- 时间复杂度: $O(b^l)$
- 空间复杂度: $O(bl)$
- 完备性: NO
- 最优性: NO

1.4.4 迭代加深搜索 (IDS, Iterative Deepening Search)

IDS 是 DLS 的升级版, 即首先允许深度优先搜索 k 层搜索树, 若没有发现可行解, 再将 $k + 1$ 后重复以上步骤搜索。

IDS 将 DFS 的空间优势和 BFS 的时间优势结合起来, 是一种比较好的搜索算法。

Algorithm 6 IDS

```
1: procedure IDS(problem) returns a solution node or failure
2:   for depth = 0 to  $\infty$  do
3:     result = DLS(problem, depth)
4:     if result  $\neq$  cutoff then
5:       return result
```

- 时间复杂度: $O(b^d)$
- 空间复杂度: $O(bd)$
- 完备性: YES
- 最优性: YES

IDS 很明显存在重复的搜索, 在迭代至第 k 层时, 会重复搜索前 $k - 1$ 层的所有结点。虽然这的确浪费了一些时间, 但是它并没有那么不堪。

假设分别计算使用 BFS 和 IDS 所生成的结点数量:

$$N(BFS) = 1 + b^1 + b^2 + \dots + b^d$$

$$N(IDS) = (d)b + (d-1)b^2 + (d-2)b^3 + \dots + (1)b^d$$

假设 $b = 10$, $d = 5$ 时, $N(BFS) = 111110$, $N(IDS) = 123450$ 。可以计算得出 IDS 只比 BFS 多生成了 11% 的结点。

$$\frac{N(IDS) - N(BFS)}{N(BFS)} = 11\%$$

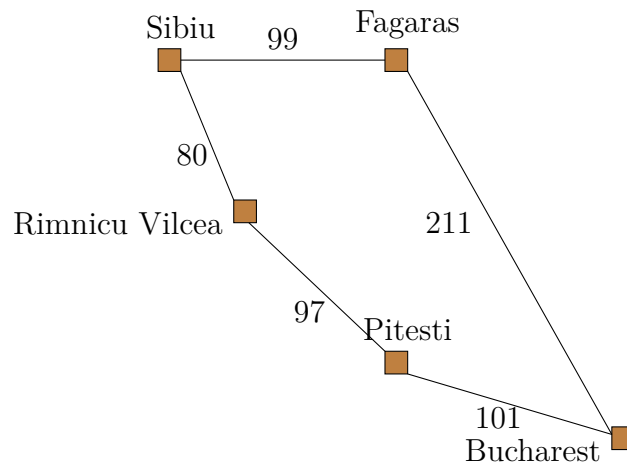
因此, IDS 只多生成了少量的结点, 而节省了大量的空间。

1.4.5 一致代价搜索 (UCS, Uninformed-Cost Search)

由于不同的行为具有不同的代价, 每次选择一个从根结点到当前结点成本最低的路径是一个很好的选择。这被理论计算机科学界称为 Dijkstra 算法, 被 AI 领域称为统一成本搜索 UCS。BFS 可以看作是根据树的深度一层一层扩展的, 而

UCS 则是根据树的代价扩展的。

在罗马尼亚的部分地图中，问题是从 Sibiu 到达 Bucharest。



Sibiu 的后继结点 is Rimnicu Vilcea 和 Fagaras，路径成本分别为 80 和 99。因此扩展代价最低的结点 Rimnicu Vilcea，计算得出到 Pitesti 的路径代价为 $80 + 97 = 177$ 。

其次路径代价最低的是 Fagaras，计算得出到 Bucharest 的路径代价为 $99 + 211 = 310$ 。此时，虽然 Bucharest 是目标结点，但是 UCS 算法并不会在生成结点时判断其是否为目标结点，而是在扩展结点的时候判断。

因此算法会继续选择 Pitesti 进行扩展，计算得出到 Bucharest 的路径代价为 $80 + 97 + 101 = 278$ 。这个具有更小的路径代价，因此它替换之前到达这里的路径，并被添加到前沿结点集合中。最终，它在被扩展时发现是目标结点。

Algorithm 7 UCS

```
1: procedure UCS(problem) returns a solution node or failure
2:   return BestFirstSearch(problem, PathCost)
3:
4: procedure BESTFIRSTSEARCH(problem, PathCost)
5:   node = Node(problem.Initial)
6:   frontier = PriorityQueue(node) # ordered by PathCost
7:   reached = {problem.Initial: node}
8:
9:   while not frontier.is_empty() do
10:     node = frontier.pop()
11:     if problem.IsGoal(node.State) then
12:       return node
13:
14:     for child in Expand(problem, node) do
15:       s = child.State
16:       if s is not in reached or child.PathCost < reached[s].PathCost then
17:         reached[s] = child
18:         frontier.push(child)
19:
20:   return failure
```

- 时间复杂度: $O(b^{1+\lceil C^*/\epsilon \rceil})$
- 空间复杂度: $O(b^{1+\lceil C^*/\epsilon \rceil})$
- 完备性: YES
- 最优性: YES

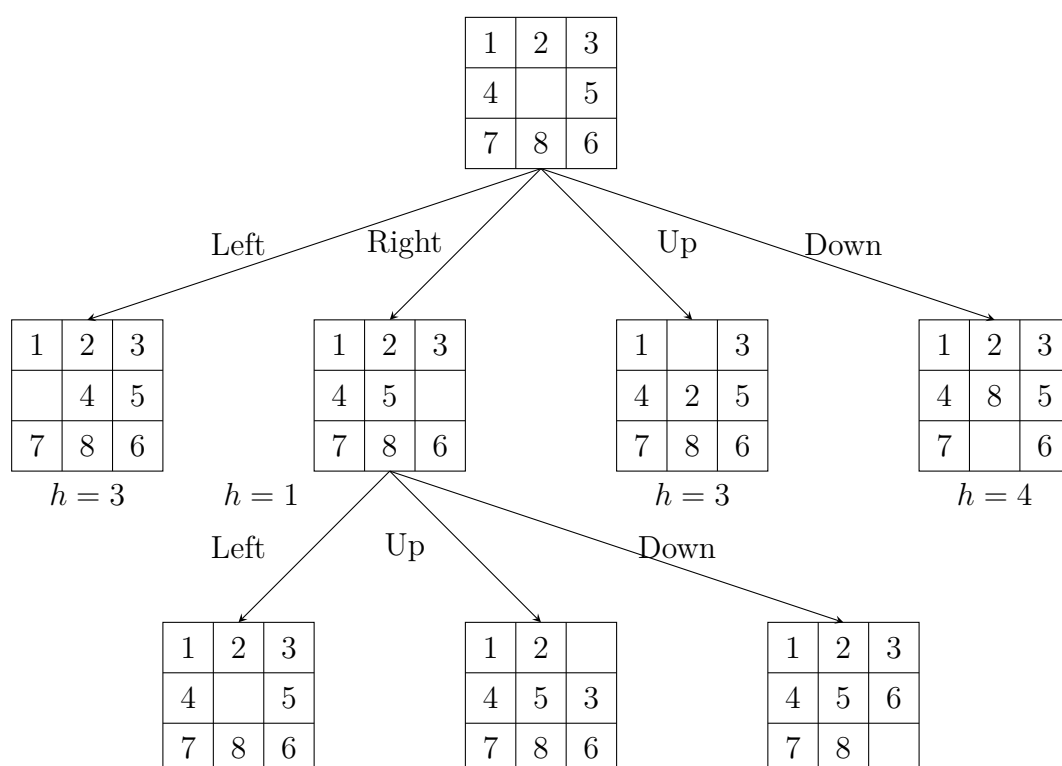
1.5 启发式搜索

1.5.1 启发式搜索 (Informed Search)

盲目搜索只使用问题中提供的信息（如路径代价），而启发式搜索使用额外的信息（如估计值），通过指导向最有希望的方向前进，从而达到减少搜索范围、降低问题复杂度。

然而，启发式策略是极易出错的。在解决问题的过程中启发仅仅是下一步将要采取措施的一个猜想，常常根据经验和直觉来判断。启发式函数 $h(n)$ (heuristic function) 用于表示结点 n 到达目标结点的最小估计代价。

例如在八数码问题中，可以根据处于错误位置的数字的数量来作为引导。

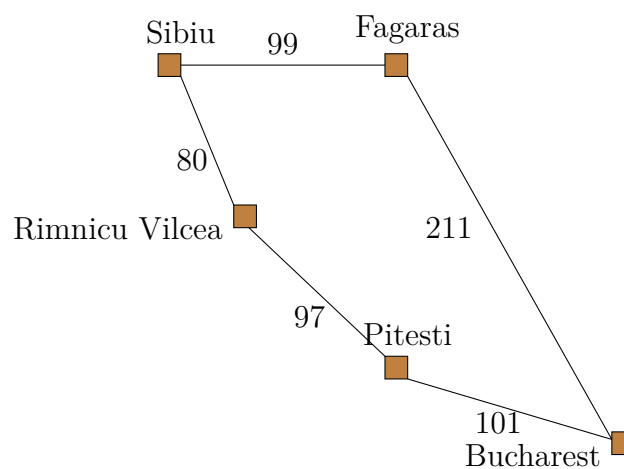


1.5.2 贪婪最佳优先搜索 (Greedy Best-First Search)

所谓贪婪，即只扩展当前代价最小的结点。但是贪心搜索不能保证最优，甚至都不能保证完备，因为它可能陷入死循环。

贪心只根据启发进行搜索，即评估函数 $f(n) = h(n)$ 。

假设 $h(n)$ 是当前城市到 Bucharest 的直线距离，这个直线距离是原问题所不具备的。



Bucharest	0
Fagaras	176
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253

表 1.1: 各个城市到 Bucharest 的直线距离

问题是从 Sibiu 到 Bucharest，GBFS 算法的搜索过程如下：

Node	IsGoal(Node)	s	Frontier	Reached
S	NO		$[S_{253}]$	$\{S_0\}$
S	NO	R_{193}	$[R_{193}]$	$\{S_0, R_{80}\}$
S	NO	F_{176}	$[R_{193}, F_{176}]$	$\{S_0, R_{80}, F_{99}\}$
F	NO	B_0	$[R_{193}, B_0]$	$\{S_0, R_{80}, F_{99}, B_0\}$
F	NO	S_{253}	$[R_{193}, B_0]$	$\{S_0, R_{80}, F_{99}, P_{177}\}$
B	YES			

Algorithm 8 GBFS

```

1: procedure GBFS(problem, f) returns a solution node or failure
2:   node = Node(problem.Initial)
3:   frontier = PriorityQueue(node) # ordered by f
4:   reached = {problem.Initial: node}
5:
6:   while not frontier.is_empty() do
7:     node = frontier.pop()
8:     if problem.IsGoal(node.State) then
9:       return node
10:
11:    for child in Expand(problem, node) do
12:      s = child.State
13:      if s is not in reached or child.PathCost < reached[s].PathCost then
14:        reached[s] = child
15:        frontier.push(child)
16:
17:  return failure

```

1.5.3 A* 搜索 (A* Search)

A* 搜索的评估函数为 $f(n) = g(n) + h(n)$, 其中 $g(n)$ 是从初始结点到结点 n 的实际代价, $h(n)$ 是从结点 n 到达目标结点的最小估计代价。

启发函数会影响 A* 搜索的性能。在极端情况下，当 $h(n) = 0$ ，则 $g(n)$ 将决定结点的优先级，此时算法就退化成了 UCS。

当 $h(n)$ 始终小于等于结点 n 到目标结点的实际代价 $h^*(n)$ ，A* 算法保证一定能够找到最短路径。但是 $h(n)$ 的值越小，算法将遍历越多的结点，导致算法越慢。

当 $h(n)$ 完全等于 $h^*(n)$ 时，A* 算法将很快速地找到最佳路径。可惜并非所有场景下都能做到这一点，因为在到达终点之前，很难确切地算出距离终点的距离。

当 $h(n)$ 远大于 $g(n)$ 时，此时只有 $h(n)$ 产生效果，算法也就变成了 GBFS。

A* 算法对于部分罗马尼亚地图的搜索过程如下：

Node	IsGoal	s	Frontier	Reached
S_{0+253}	NO		$[S_{0+253}]$	$\{S_{0+253}\}$
S_{0+253}	NO	R_{80+193}	$[R_{80+193}]$	$\{S_{0+253}, R_{80+193}\}$
S_{0+253}	NO	F_{99+176}	$[R_{80+193}, F_{99+176}]$	$\{S_{0+253}, R_{80+193}, F_{99+176}\}$
R_{80+193}	NO	$P_{177+100}$	$[F_{99+176}, P_{177+100}]$	$\{S_{0+253}, R_{80+193}, F_{99+176}, P_{177+100}\}$
R_{80+193}	NO	$S_{160+253}$	$[F_{99+176}, P_{177+100}]$	$\{S_{0+253}, R_{80+193}, F_{99+176}, P_{177+100}\}$
F_{99+176}	NO	B_{310+0}	$[P_{177+100}, B_{310+0}]$	$\{S_{0+253}, R_{80+193}, F_{99+176}, P_{177+100}, B_{310+0}\}$
F_{99+176}	NO	$S_{198+253}$	$[P_{177+100}, B_{310+0}]$	$\{S_{0+253}, R_{80+193}, F_{99+176}, P_{177+100}, B_{310+0}\}$
$P_{177+100}$	NO	B_{278+0}	$[B_{310+0}, B_{278+0}]$	$\{S_{0+253}, R_{80+193}, F_{99+176}, P_{177+100}, B_{278+0}\}$
B_{278+0}	YES			

1.6 局部搜索

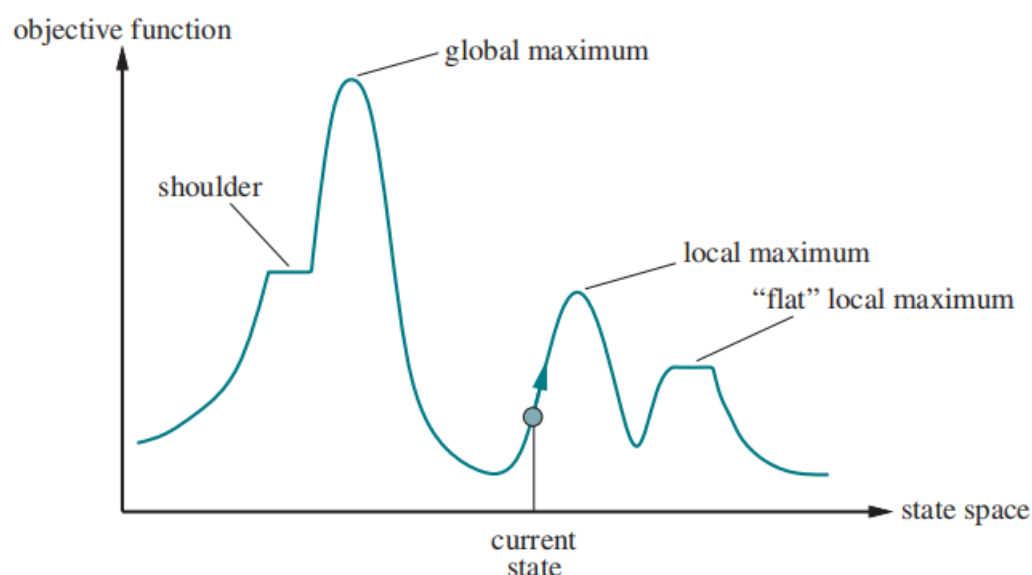
1.6.1 局部搜索 (Local Search)

局部搜索是一种用于解决计算上难以优化的问题（如 NP 问题）的启发式方法。算法每次从当前解的邻域解空间选择一个最好邻居作为下次迭代的当前解，直到达到一个局部最优解。

传统的 N 皇后问题使用的是回溯法（backtrack）解决的，但是一旦问题的规模过大，计算时间复杂度是不能够接受的。

1.6.2 爬山法 (Hill Climbing)

设想去攀登一座山群的最高峰，而此山群有很多的小山峰，且我们对此山群一无所知。那么当我们来到一座小山峰（局部极值点）时，我们会错误地判断这就是最高峰，事实上这有可能是一个很糟糕的解（即与最高峰还差很远）。







Algorithm 9 HillClimbing

```
1: procedure HILLCLIMBING(problem) returns a state that is a local maximum
2:   current = problem.Initial
3:   while true do
4:     neighbor = a highest-valued successor state of current
5:     if Value(neighbor) <= Value(current) then
6:       return current
7:     current = neighbor
```

爬山法解决 N 皇后问题时，利用评估函数（导致皇后冲突的对数）来评估皇后摆放位置的好坏。

例如对于 4 皇后，初始状况皇后都位于第 1 行中，皇后可以被移动到每列的任意位置上。

爬山法对每个皇后能够移动的位置进行评估，例如-4 代表该列的皇后移到到该位置后，会有 4 对皇后产生冲突。

			
-4	-5	-5	-4
-4	-4	-4	-4
-4	-3	-3	-4

依次选择评估值最大的位置，将对应列的皇后移到该位置。

	-6		
-1	-5	-2	-3
-3	-4	-3	-1
-3		-2	-3

		♔	♔
♔			
			0
	♔		

		♔	
♔			
			♔
	♔		

爬山法还有很多变体：

- 随机爬山法 (Stochastic Hill-Climbing)：随机选择比当前结点更优的后继结点。
- 首选爬山法 (First-Choice Hill-Climbing)：选择第一个比当前结点更优的后继结点。
- 随机重新开始爬山法 (Random Restart Hill-Climbing)：通过随机生成的初始状态来进行一系列的爬山法搜索，找到目标时停止搜索。

1.6.3 模拟退火 (Simulated Annealing)

模拟退火是一个由金属退火启发的算法。在物理应用中分子排布可能是紊乱的，如果我们将它升温然后缓慢降温，就可以生成完美的晶形。

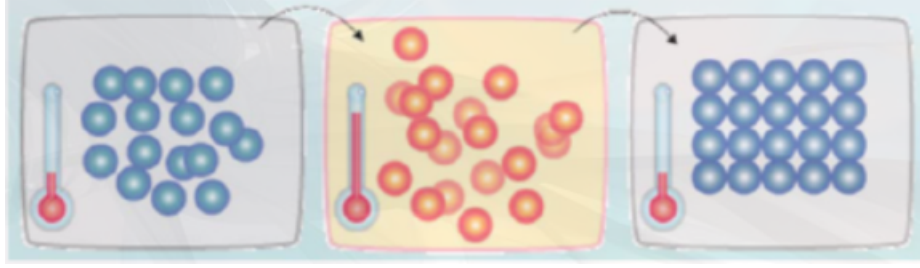


图 1.7: 物理退火

模拟退火算法也是一种局部搜索算法，它的思想是在初始状态下，随机生成一个解，然后在解的邻域中随机选择一个解，如果该解比当前解更好，就接受该解，否则以一定的概率接受该解。

例如在计算最小值的问题中，如果新解比当前解更小，就接受该解，否则以一定的概率接受该解，这个概率随着温度的降低而减小。

$$\Delta E = Value(current) - Value(next)$$

1. 如果 $\Delta E > 0$: 以概率为 1 接受新解
2. 否则: 以概率为 $e^{\frac{\Delta E}{T}}$ 接受新解

其中 T 表示温度，会随着时间（算法的迭代）逐渐降低。接受解的概率会因 T 受到影响：

$$\lim_{T \rightarrow \infty} P(\text{Acceptance of bad move}) = 1$$

$$\lim_{T \rightarrow 0} P(\text{Acceptance of bad move}) = 0$$

Algorithm 10 SimulatedAnnealing

```
1: procedure SIMULATEDANNEALING(problem, schedule) returns a solution
   state
2:   current = problem.Initial
3:   for t = 1 to  $\infty$  do
4:     T = schedule(t)
5:     if T = 0 then
6:       return current
7:
8:     next = problem.RandomSuccessor(current)
9:      $\Delta E$  = Value(current) - Value(next)
10:    if  $\Delta E > 0$  then
11:      current = next
12:    else
13:      current = next with probability  $e^{\frac{\Delta E}{T}}$ 
```

Chapter 2 约束满足问题

2.1 约束满足问题

2.1.1 约束满足问题(CSP, Constraint Satisfaction Problems)

在 CSP 问题中, 存在一组变量, 当每个变量都被赋了一个值, 且能够满足所有的约束条件时, 问题就解决了。

CSP 问题包含:

- $X = \{X_1, X_2, \dots, X_n\}$: 变量集合
- $D = \{D_1, D_2, \dots, D_n\}$: 域 (domain)
- C : 约束集合

约束可分为:

- 一元约束 (unary constraint): 只对一个变量进行约束, 如 $A \neq 3$ 、 $B \neq 4$ 。
- 二元约束 (binary constraint): 对两个变量进行约束, 如 $A < B$ 、 $B < C$ 。
- 多元约束: 对两个以上的变量进行约束, 如 $A + B < C$ 。

例如 $X = \{A, B, C\}$, $dom(A) = dom(B) = dom(C) = \{1, 2, 3, 4\}$, 约束条件为 $A < B$ 和 $B < C$ 。

其中 $A = 2$ 、 $B = 3$ 、 $C = 4$ 是一组满足要求的赋值, 而 $A = 2$ 、 $B = 3$ 、 $C = 1$ 就是一组不满足要求的赋值。

2.1.2 澳大利亚地图着色问题

在澳大利亚地图着色问题中,

- $X = \{WA, NT, SA, Q, NSW, V, T\}$
- $D = \{Red, Green, Blue\}$
- $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq V, SA \neq NSW, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$

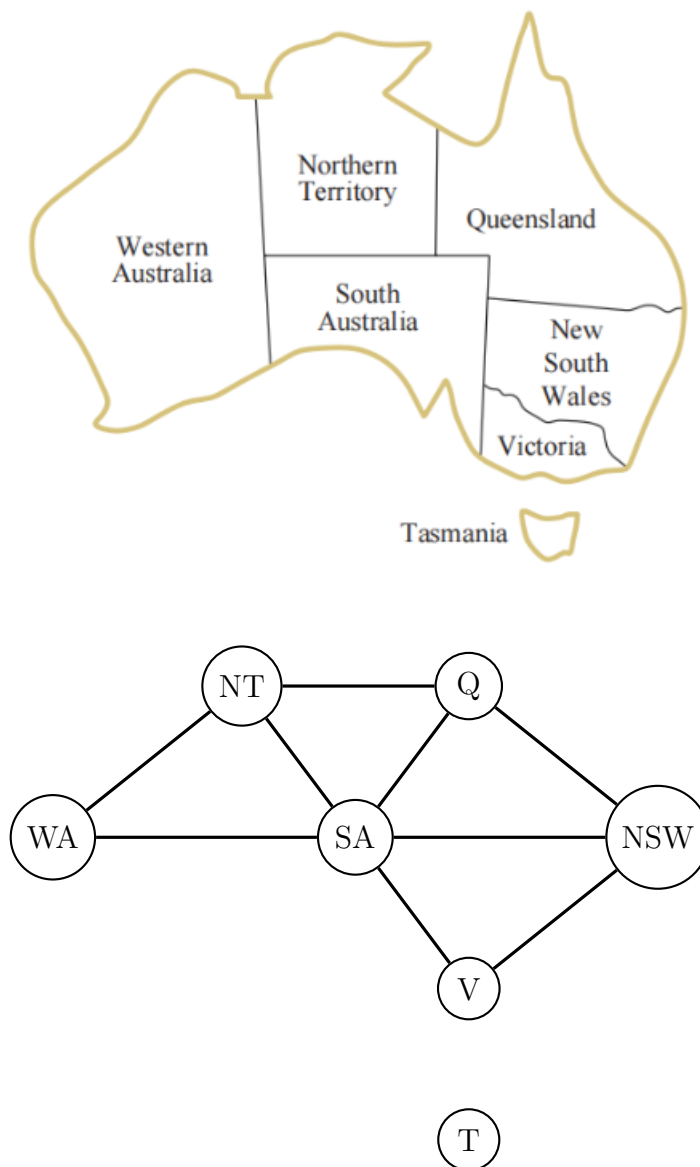


图 2.1: 澳大利亚地图

2.1.3 密码算术谜题

在密码算术谜题中，每个字母代表一个不同的数字。

$$\begin{array}{r} TWO \\ + TWO \\ \hline = FOUR \end{array}$$

- 变量

- $X = \{T, W, O, F, U, R, C_1, C_2, C_3\}$, 其中 C_1 、 C_2 、 C_3 为进位。

- 域

- $dom(T) = dom(W) = dom(O) = dom(F) = dom(U) = dom(R) = \{0, 1, \dots, 9\}$

- $dom(C_1) = dom(C_2) = dom(C_3) = \{0, 1\}$

- 约束

- $AllDiff(T, W, O, F, U, R)$

- $O + O = R = 10C_1$

- $C_1 + W + W = U + 10C_2$

- $C_2 + T + T = O + 10C_3$

- $C_3 = F$

- $T \neq 0$

- $F \neq 0$

2.1.4 数独

数独的目标是在 9×9 的方格中填入数字，使得每一行、每一列和每一个 3×3 的小方格中都包含 $1 \sim 9$ 的数字。

		3		2		6		
9			3		5			1
		1	8		6	4		
		8	1		2	9		
7								8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1		3		

假设数独的行号为 $A \sim I$ ，列号为 $1 \sim 9$ 。

- 变量

- $X = \{A1, \dots, A9, B1, \dots, I1, \dots, I9\}$

- 域

- $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- 约束

- $AllDiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)$
 - $AllDiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)$
 - \dots
 - $AllDiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)$
 - $AllDiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)$
 - \dots
 - $AllDiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)$
 - $AllDiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)$
 - \dots

2.2 约束传播

2.2.1 弧相容 (Arc Consistency)

约束传播 (Constrain Propagation) 的目的是减少变量的合法取值, 有时如果所有变量的域都被减少为 1 时, 它可以直接算出结果。约束传播还可以检测失败的情况, 当某一个变量的域被减少为 0 时, 该问题将无解。

如果变量 X_i 的域 D_i 中的每个值都满足在变量 X_j 的域 D_j 中的值时, 称 X_i 和 X_j 是弧相容的。

例如:

- 变量: X, Y
- 域: $dom(X) = dom(Y) = \{1, \dots, 9\}$
- 约束: $Y = X^2$

当变量 X 取 $\{4, 5, 6, 7, 8, 9\}$ 时, X 与 Y 不满足弧相容。

当变量 Y 取 $\{2, 3, 5, 6, 7, 8\}$ 时, Y 与 X 不满足弧相容。

为了让 X 和 Y 都互相满足弧相容, 需要从 $dom(X)$ 和 $dom(Y)$ 中移除不满足弧相容的值。因此

$$dom(X) = \{0, 1, 2, \dots, 9\} \setminus \{4, 5, 6, 7, 8, 9\} = \{0, 1, 2, 3\}$$

$$dom(Y) = \{0, 1, 2, \dots, 9\} \setminus \{2, 3, 5, 6, 7, 8\} = \{0, 1, 4, 9\}$$

AC-3 (Arc Consistency-3) 算法用于解决 CSP 问题, 早期的 AC 算法效率太低, 而 AC-3 更为简单常用。

Algorithm 11 AC-3

```
1: procedure AC-3(csp) returns false if inconsistency found and true otherwise
2:   queue = [all arcs in csp]
3:
4:   while not queue.is_empty() do
5:      $(X_i, X_j) = \text{queue.dequeue}()$ 
6:     if Revise(csp,  $X_i$ ,  $X_j$ ) then
7:       if size( $D_i$ ) == 0 then
8:         return false
9:       for  $X_k$  in  $X_i$ .Neighbors -  $\{X_j\}$  do
10:        queue.enqueue( $(X_i, X_j)$ )
11:
12:   return true
13:
14: procedure REVISE(csp,  $X_i$ ,  $X_j$ ) returns true iff we revise the domain of  $X_i$ 
15:   revised = false
16:   for  $x$  in  $D_i$  do
17:     if no value  $y$  in  $D_j$  allows  $(x, y)$  to satisfy constraint  $(X_i, X_j)$  then
18:       delete  $x$  from  $D_i$ 
19:       revised = true
20:   return revised
```

2.3 回溯搜索

2.3.1 回溯搜索 (Backtracking Search)

回溯就是简单粗暴的试错方法。例如走迷宫，大多人类一般就是使用回溯法，当走到一条死路，就往回退到前一个岔路，尝试另外一条，直到走出。

回溯搜索与 DFS 类似，回溯搜索每次为变量选择一个赋值，当没有合法的值可以赋给该变量时就回溯返回，尝试别的路径。

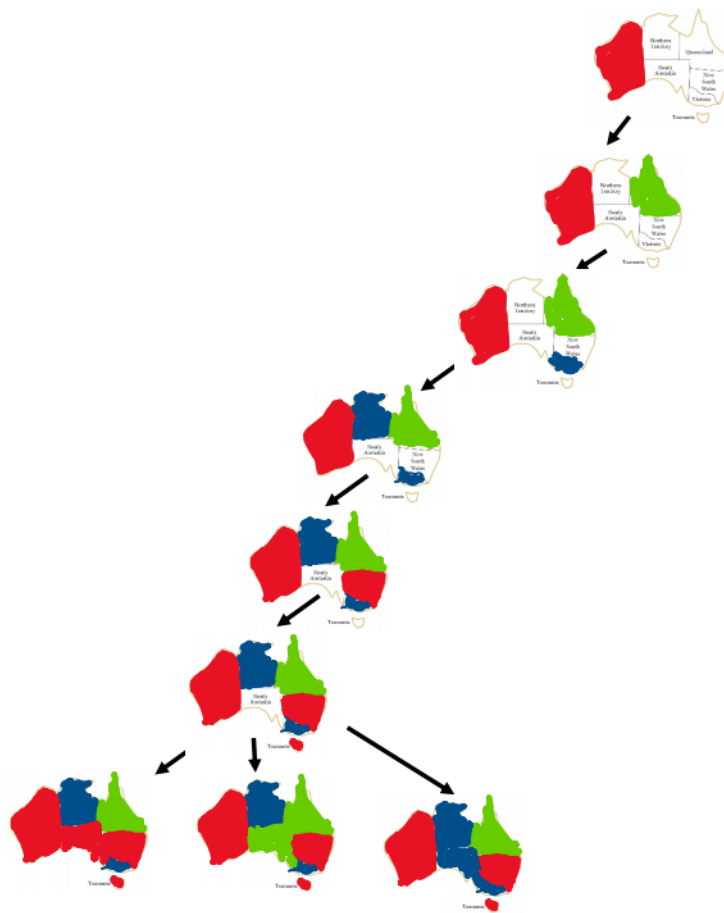


图 2.2: 回溯搜索

Algorithm 12 Backtracking Search

```
1: procedure BACKTRACKINGSEARCH(csp) returns a solution or failure
2:   return Backtrack(csp, {})
3:
4: procedure BACKTRACK(csp, assignment) returns a solution or failure
5:   if assignment is complete then
6:     return assignment
7:
8:   var = SelectUnassignedVariable(csp, assignment)
9:   for value in OrderDomainValues(csp, var, assignment) do
10:    if value is consistent with assignment then
11:      assignment.add(var = value)
12:      inferences = Inference(csp, var, assignment)
13:      if inferences != failure then
14:        csp.add(inferences)
15:        result = Backtrack(csp, assignment)
16:        if result != failure then
17:          return result
18:        csp.remove(inferences)
19:      assignment.remove(var = value)
20:
21:   return failure
```

2.3.2 前向检查 (Forward Checking)

前向检查可用于将还没有进行赋值的候选变量进行适当排除。

在澳大利亚地图着色问题中，一开始，所有省份的颜色都能够选择所有域中的颜色。

当 WA 选择了红色后，相邻的 NT 和 SA 的颜色域就需要删除红色。当 Q 选择

了绿色后，相邻的 NT、NSW 和 SA 的颜色域就需要删除绿色。当 V 选择了蓝色后，相邻的 NSW 和 SA 的颜色域就需要删除蓝色。

	WA	NT	Q	NSW	V	SA	T
init domains							
$WA = red$							
$Q = green$							
$V = blue$							

图 2.3: 前向检查

此时发现，SA 的颜色域为空，说明当前的选择不合法，需要回溯。

2.3.3 变量和取值顺序

变量和取值顺序可以对回溯搜索进行优化，从而减少搜索空间。

一个常用的选择标准是最小剩余值（Minimum Remaining Values），也就是选择域中包含选择最少的变量。在每次给当前变量赋值后，其它变量的域可能会更新，那么包含元素最少的变量，就代表该变量的域可能最早为空，从而产生死结点。

最小剩余值的排序方法认为，既然问题迟早可能暴露，那就选择一种可以让问题尽早暴露的方式。这种方式也被称为 Fail-First。最小剩余值是基于变量进行选择的，而另一种最小约束值（Least Constraining Value）讨论的是当一个变量被选定时，从其域中选择哪个值的方式。

最少约束值优先选择的值应该试图为剩余变量赋值留下最大的空间。因为在 CSP 问题中，无论如何都要对所有的变量进行值的分配，因此必须把困难的变量情况放在前面进行讨论，这样可以提前暴露问题，触发回溯行为，避免不必要的计算。但是对于每个变量的值的选择来说，并不是每个值都会用到，因此只需要选择最有效的值即可。

2.4 CSP 局部搜索

2.4.1 CSP 局部搜索

回溯搜索从一个空的初始状态开始，逐个尝试对变量赋值，当选择不合法时就进行回溯，因此回溯搜索始终保持弧相容。

局部搜索算法对求解许多 CSP 都是很有用的，它从一个完整地随机布局开始，这个初始布局有可能会违反约束，局部搜索的目的就是消除这些矛盾。在为变量选择新值的时候，最明显的启发式是选择与其它变量冲突最少的值。

Algorithm 13 MinConflicts

```
1: procedure MINCONFLICTS(csp, max_steps) returns a solution or failure
2:   current = an initial complete assignment for csp
3:   for i = 1 to max_steps do
4:     if current is a solution for csp then
5:       return current
6:     var = a randomly chosen conflicted variable from csp.Variables
7:     value = the value v for var that minimizes Conflicts(csp, var, v, current)
8:     current.update(var = value)
9:   return failure
```

Chapter 3 对抗搜索

3.1 对抗搜索

3.1.1 对抗搜索 (Adversarial Search)

在一些复杂的博弈论 (Game Theory) 问题中, 每一轮操作都可能有许多决策, 于是就会形成一棵庞大的博弈树。针对这样的问题可以使用对抗搜索, 也被称为博弈搜索 (Game Search), 即在一个竞争的环境中, 多个 agents 之间通过竞争获取利益。

多个 agents 在同一个搜索空间中搜索解决方案的情况, 通常发生在游戏中。每个 agent 是其它 agent 的对手并且彼此竞争, 因此每个 agent 都需要考虑其它 agent 的操作及该操作所产生的影响。对抗搜索的目标就是选择一个能够使收益最大化的策略。

3.1.2 零和游戏 (Zero-Sum Game)

零和游戏意味着游戏双方有着相反的目标, 换句话说, 在游戏的任何终结状态下, 所有玩家获得的总和等于零。例如井字棋 (Tic-Tac-Toe)、象棋 (chess)、围棋 (Go) 等都是零和游戏。

零和游戏是 AI 最常研究的游戏, 这类游戏通常有两个玩家 (称为 MAX 和 MIN) 轮流行动, 游戏是具有完整信息和确定的。

这类游戏可以被形式化为:

- s_0 : 初始状态
- $ToMove(s)$: 在状态 s 下能够行动的玩家
- $Actions(s)$: 在状态 s 下合法行动的集合

- $Result(s, a)$: 在状态 s 下执行行动 a 后的结果
- $IsTerminal(s)$: 判断游戏是否结束
- $Utility(s, p)$: 玩家 p 在终结状态 s 的效用

例如对于井字棋游戏, MAX 下 X, MIN 下 O。初始状态 s_0 为一个 3×3 的空棋盘。

对于某个状态 s_1 :

	1	2	3
1	X	O	X
2		O	O
3		X	

$Actions(s_1) = \{(x, (2, 1)), (x, (3, 1)), (x, (3, 3))\}$ 。

如果 MAX 在 $(2, 1)$ 下 X, 可表示为 $Result(s_1, (x, (2, 1)))$:

	1	2	3
1	X	O	X
2	X	O	O
3		X	

对于某个最终状态 s_2 :

	1	2	3
1	X	O	X
2	X	O	O
3	X	X	O

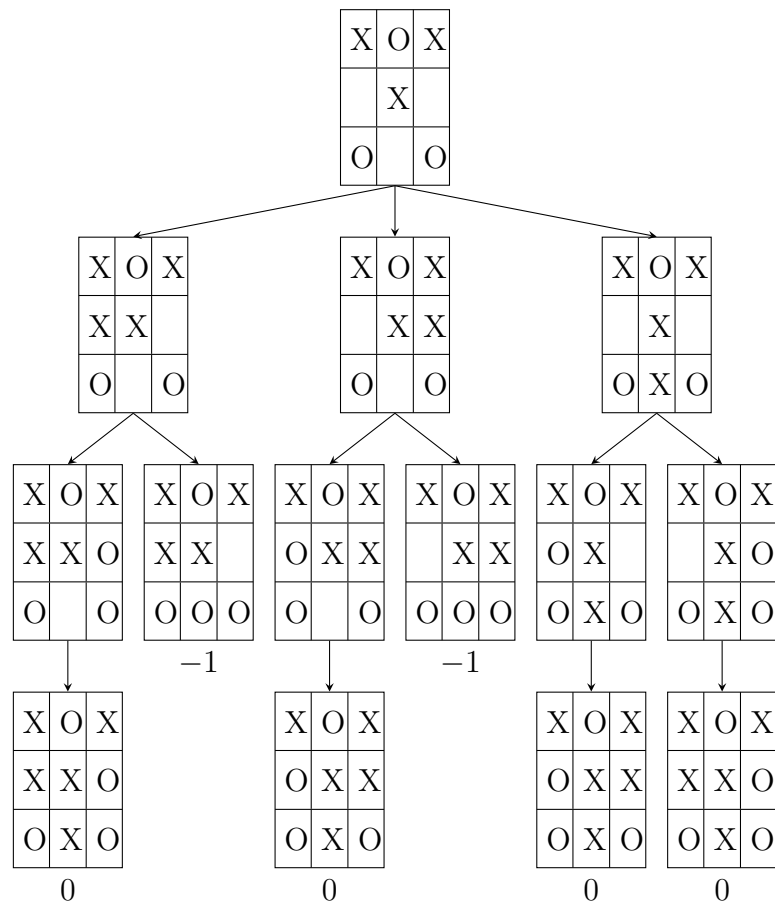
MAX 和 MIN 的效用函数分别表示:

- $Utility(s_2, MAX) = +1$
- $Utility(s_2, MIN) = -1$

3.2 Minimax

3.2.1 博弈树 (Game Tree)

博弈树可以表示两名游戏参与者之间的一场博弈，他们交替行棋，试图获胜。树中的每一个结点都对应于棋盘的一种布局。



对于玩家 MAX 而言，他需要选择能够使效用函数最大化的行动。而对于玩家 MIN 而言，需要选择对 MAX 最不利的行动，即最小化 MAX 的效用 (minimize maximum utility of MAX)。

3.2.2 Minimax

Minimax 是一种递归算法，用于在博弈树中找到最优行动，其中 MAX 将选择最大化值, MIN 将选择最小化值。算法执行深度优先搜索算法以探索完整的游戏树。

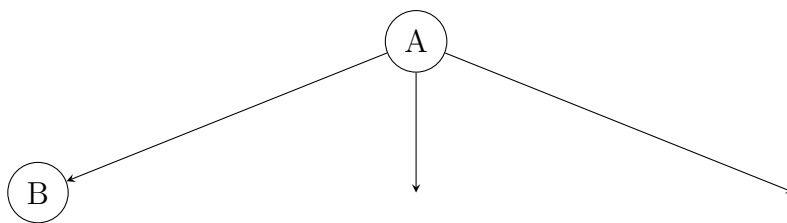
Algorithm 14 Minimax

```
1: procedure MINIMAXSEARCH(game, state) returns an action
2:   player = game.ToMove(state)
3:   value, move = MaxValue(game, state)
4:   return move
5:
6: procedure MAXVALUE(game, state) returns a (utility, move) pair
7:   if game.IsTerminal(state) then
8:     return game.Utility(state, player), null
9:    $v = -\infty$ 
10:  for a in game.Actions(state) do
11:     $v_2, a_2 = \text{MinValue}(\text{game}, \text{game.Result}(\text{state}, a))$ 
12:    if  $v_2 > v$  then
13:       $v, \text{move} = v_2, a$ 
14:  return  $v, \text{move}$ 
15:
16: procedure MINVALUE(game, state) returns a (utility, move) pair
17:  if game.IsTerminal(state) then
18:    return game.Utility(state, player), null
19:   $v = +\infty$ 
20:  for a in game.Actions(state) do
21:     $v_2, a_2 = \text{MaxValue}(\text{game}, \text{game.Result}(\text{state}, a))$ 
22:    if  $v_2 < v$  then
23:       $v, \text{move} = v_2, a$ 
24:  return  $v, \text{move}$ 
```

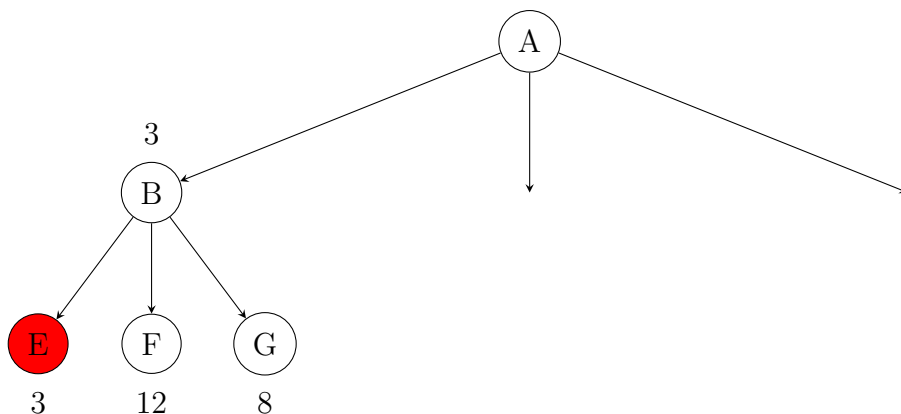
例如有一个只有一个回合的游戏，MAX 先进行行动：



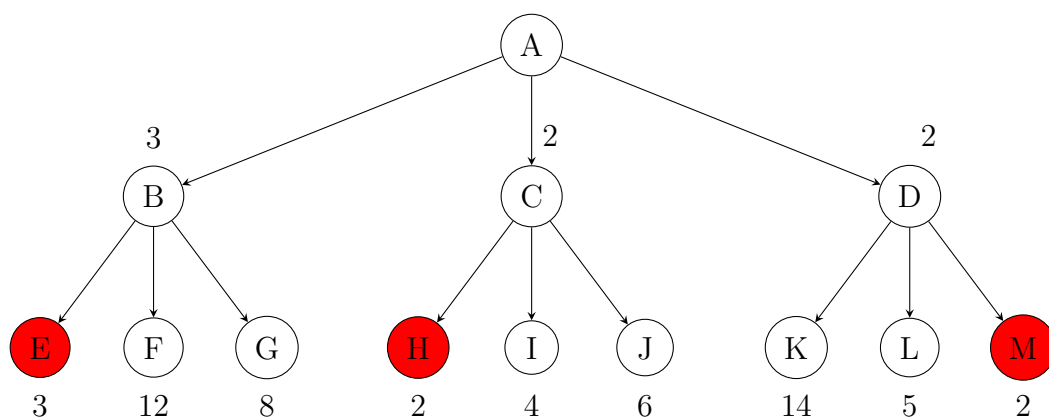
MAX 需要从当前状态 A 中选择一个能使效用最大的行动。根据深度优先搜索的策略，首先选择 A 的第一个子结点 B 。



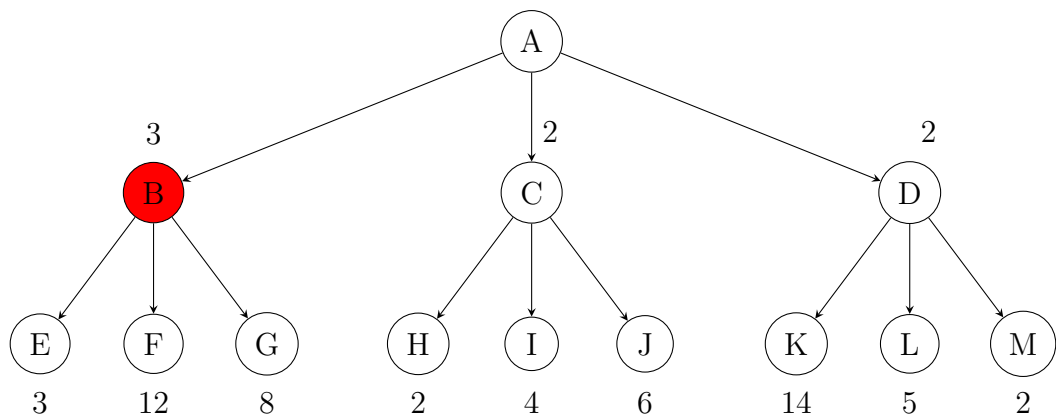
状态 B 为 MIN 的回合，因此需要从 B 中选择一个能使效用最小的行动。



同样，再从 A 的剩余两个子结点 C 和 D 中，计算能使 MIN 效用最小的行动。



最后，MAX 从 A 的三个子结点中选择一个能使效用最大的行动。



- 时间复杂度: $O(b^m)$
- 空间复杂度: $O(bm)$ or $O(m)$
- 完备性: YES (当博弈树有穷时)
- 最优性: YES

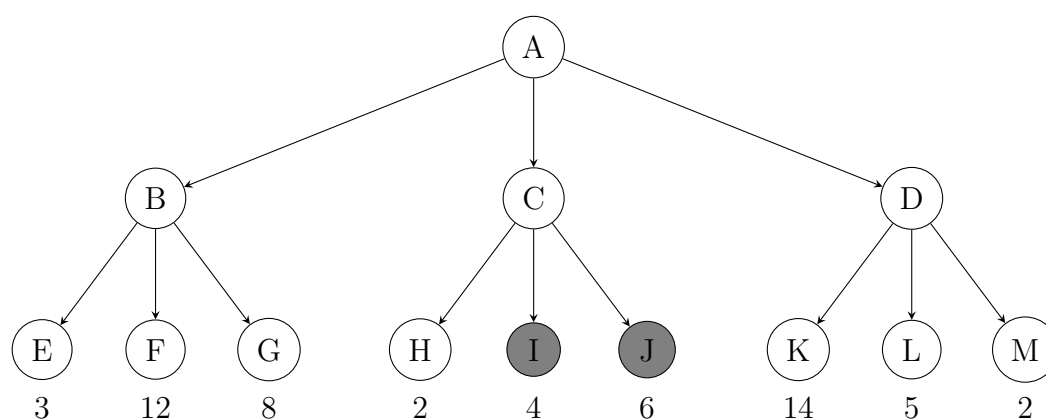
Minimax 的时间复杂度是指数级的, 因此并不适合求解复杂的游戏, 如象棋 (约有 35^{80} 个状态)、围棋 (约有 350^{150} 个状态) 等。

3.3 Alpha-Beta 剪枝

3.3.1 Alpha-Beta 剪枝 (Alpha-Beta Pruning)

为了减少 Minimax 搜索的计算量，Alpha-Beta 剪枝算法可用于裁剪掉博弈树中没有必要的分支。

例如在这棵博弈树中，结点 I 和 J 可以被裁剪掉。



因为在对 B 结点的子树搜索后，将会计算得出 MIN 在 B 结点的最小值为 3。然后在对结点 C 的搜索中， C 的第一个子结点为 2。由于 MIN 只会选择其中的最小值，而 MAX 只会根据这些最小值中的最大值，即：

$$\begin{aligned} \text{Minimax}(\text{root}) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 14) \\ &= 3 \end{aligned}$$

因此 Alpha-Beta 剪枝在搜索完结点 H 后，没有必要继续对 I 和 J 再进行搜索了。

Algorithm 15 AlphaBetaPruning

```
1: procedure ALPHABETASEARCH(game, state) returns an action
2:   player = game.ToMove(state)
3:   value, move = MaxValue(game, state,  $-\infty$ ,  $+\infty$ )
4:   return move
5:
6: procedure MAXVALUE(game, state,  $\alpha$ ,  $\beta$ ) returns a (utility, move) pair
7:   if game.IsTerminal(state) then
8:     return game.Utility(state, player), null
9:    $v = -\infty$ 
10:  for a in game.Actions(state) do
11:     $v2, a2 = \text{MinValue}(\text{game}, \text{game.Result}(\text{state}, a))$ 
12:    if  $v2 > v$  then
13:       $v, \text{move} = v2, a$ 
14:       $\alpha = \max(\alpha, v)$ 
15:    if  $v \geq \beta$  then
16:      return  $v, \text{move}$ 
17:  return  $v, \text{move}$ 
18:
19: procedure MINVALUE(game, state,  $\alpha$ ,  $\beta$ ) returns a (utility, move) pair
20:  if game.IsTerminal(state) then
21:    return game.Utility(state, player), null
22:   $v = +\infty$ 
23:  for a in game.Actions(state) do
24:     $v2, a2 = \text{MaxValue}(\text{game}, \text{game.Result}(\text{state}, a))$ 
25:    if  $v2 < v$  then
26:       $v, \text{move} = v2, a$ 
27:       $\beta = \min(\beta, v)$ 
28:    if  $v \leq \alpha$  then
29:      return  $v, \text{move}$ 
30:  return  $v, \text{move}$ 
```

- 时间复杂度：
 - 随机行动顺序： $O(b^{\frac{3m}{4}})$
 - 最佳行动顺序（启发式算法）： $O(b^{\frac{m}{2}})$
- 空间复杂度： $O(m)$
- 完备性： YES（当博弈树有穷时）
- 最优性： YES