CS50’s Introduction to Artificial Intelligence with Python
CS50 的人工智能入门与 Python

OpenCourseWare 开放课程

Donate 捐赠 (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan

戴维·J·马兰 (https://cs.harvard.edu/malan/)

malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 3 讲座 3

Optimization 优化

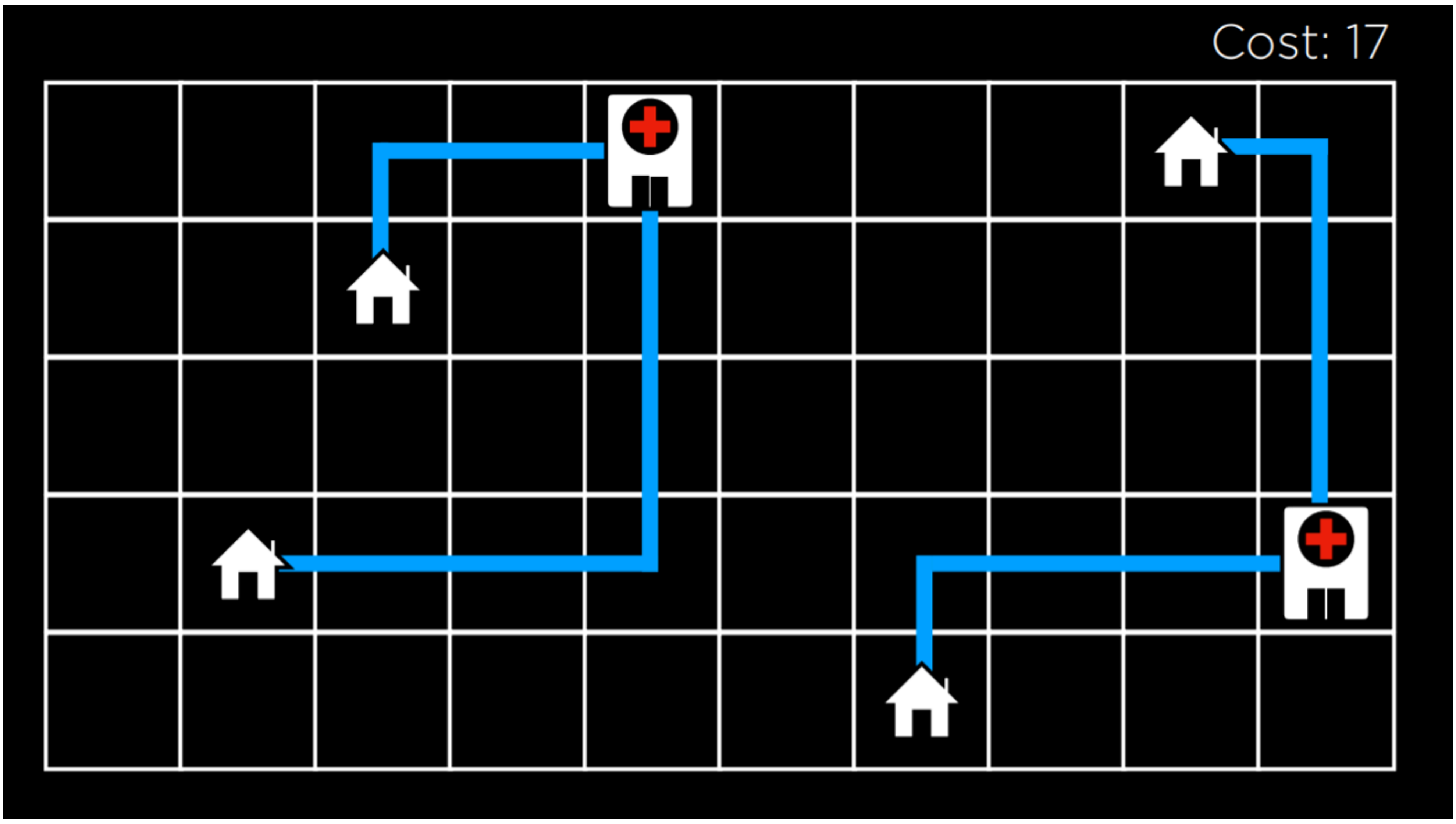
Optimization is choosing the best option from a set of possible options. We have already encountered problems where we tried to find the best possible option, such as in the minimax algorithm, and today we will learn about tools that we can use to solve an even broader range of problems.

优化是从一组可能的选择中选择最佳选项的过程。我们已经遇到了尝试找到最佳选项的问题，例如在 minimax 算法中，今天我们将学习可以用来解决更广泛问题的工具。

Local Search 本地搜索

Local search is a search algorithm that maintains a single node and searches by moving to a neighboring node. This type of algorithm is different from previous types of search that we saw. Whereas in maze solving, for example, we wanted to find the quickest way to the goal, local search is interested in finding the best answer to a question. Often, local search will bring to an answer that is not optimal but “good enough,” conserving computational power. Consider the following example of a local search problem: we have four houses in set locations. We want to build two hospitals, such that we minimize the distance from each house to a hospital. This problem can be visualized as follows:

本地搜索是一种搜索算法，它只维护一个节点，并通过移动到相邻节点来进行搜索。这种类型的算法与我们之前看到的搜索类型不同。例如，在迷宫求解中，我们想要找到最快到达目标的方式，而本地搜索则关注于找到问题的最佳答案。通常，本地搜索会带来一个非最优但“足够好”的答案，以节省计算能力。考虑以下本地搜索问题的例子：我们有四个位于特定位置的房屋。我们想要建造两个医院，使得每个房屋到医院的距离最小。这个问题可以可视化为以下形式：

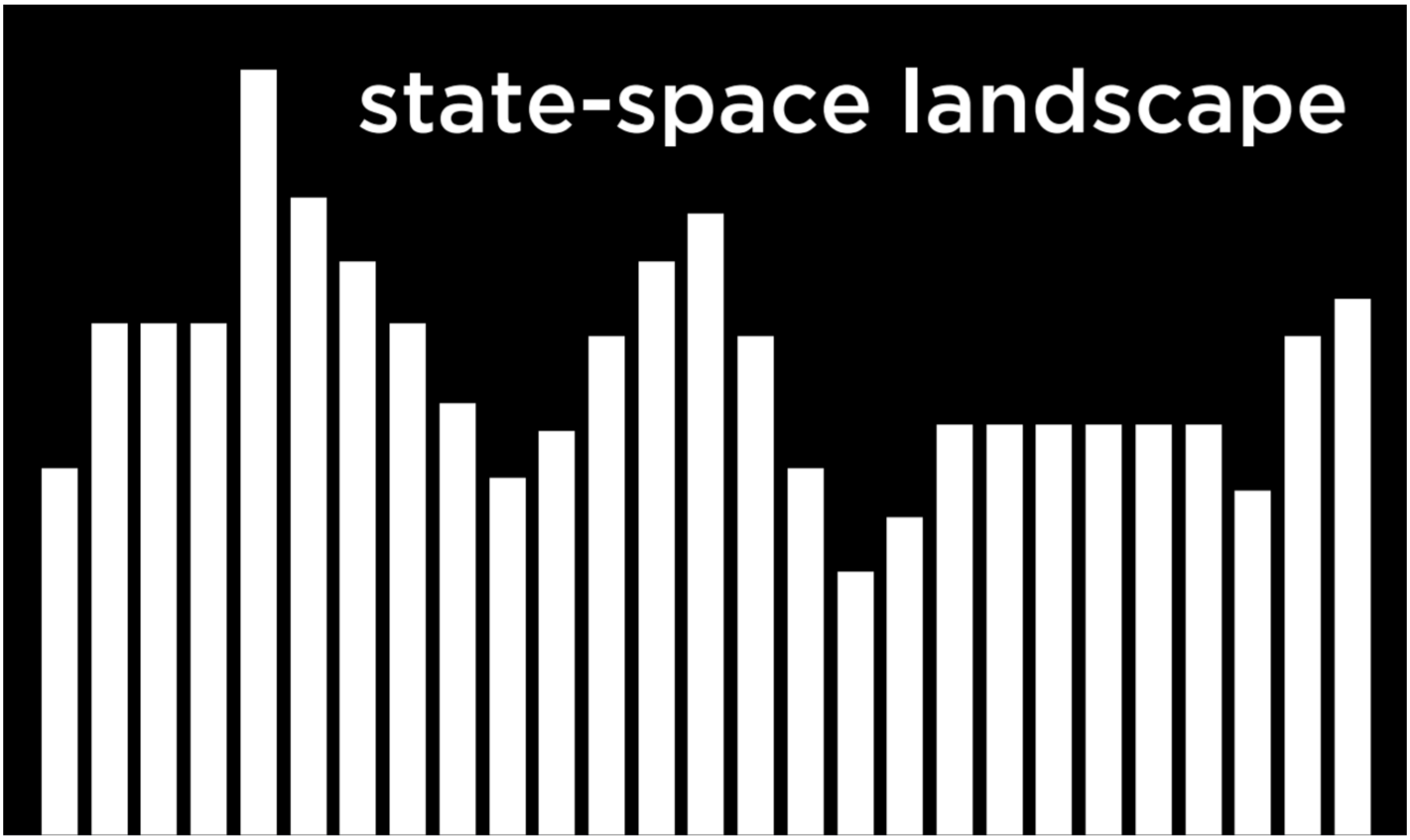


In this illustration, we are seeing a possible configuration of houses and hospitals. The distance between them is measured using Manhattan distance (number of moves up, down, and to the sides; discussed in more detail in lecture 0), and the sum of the distances from each house to the nearest hospital is 17. We call this the **cost**, because we try to minimize this distance. In this case, a state would be any one configuration of houses and hospitals.

在这幅插图中，我们看到了房屋和医院可能的配置。它们之间的距离使用曼哈顿距离进行测量（向上、向下和侧面的移动次数；在第 0 讲中进行了更详细的讨论），每栋房屋到最近医院的距离之和为 17。我们称这个为成本，因为我们试图最小化这个距离。在这种情况下，状态就是任何一种房屋和医院的配置。

Abstracting this concept, we can represent each configuration of houses and hospitals as the state-space landscape below. Each of the bars in the picture represents a value of a state, which in our example would be the cost of a certain configuration of houses and hospitals.

摘要这个概念，我们可以将每种房屋和医院的配置表示为下面的状态空间景观。图片中的每个条形代表状态的一个值，在我们的例子中，这将是特定配置下房屋和医院成本的值。



Going off of this visualization, we can define a few important terms for the rest of our discussion:

基于这个可视化，我们可以定义以下讨论中的一些重要术语： Translated Text:

- An **Objective Function** is a function that we use to maximize the value of the solution.

  目标函数是我们用来最大化解决方案价值的函数。

- A **Cost Function** is a function that we use to minimize the cost of the solution (this is the function that we would use in our example with houses and hospitals. We want to minimize the distance from houses to hospitals).

  成本函数是一种函数，我们使用它来最小化解决方案的成本（这是我们使用房子和医院的例子中的函数。我们希望最小化房子到医院的距离）。

- A **Current State** is the state that is currently being considered by the function.

  当前状态是正在由函数考虑的状态。

- A **Neighbor State** is a state that the current state can transition to. In the one-dimensional state-space landscape above, a neighbor state is the state to either side of the current state. In our example, a neighbor state could be the state resulting from moving one of the hospitals to any direction by one step. Neighbor states are usually similar to the current state, and, therefore, their values are close to the value of the current state.

  邻居状态是指当前状态可以转换到的状态。在上方的一维状态空间景观中，邻居状态是当前状态两侧的状态。在我们的例子中，邻居状态可能是通过将医院向任何方向移动一步所得到的状态。邻居状态通常与当前状态相似，因此它们的值接近当前状态的值。

Note that the way local search algorithms work is by considering one node in a current state, and then moving the node to one of the current state's neighbors. This is unlike the minimax algorithm, for example, where every single state in the state space was considered recursively.

请注意，本地搜索算法的工作方式是考虑当前状态下的一个节点，然后将该节点移动到当前状态的邻居之一。这与最小最大算法不同，例如，在最小最大算法中，状态空间中的每个状态都会递归地考虑。

Hill Climbing 山攀登 ∈ local search.

爬山算法

Hill climbing is one type of a local search algorithm. In this algorithm, the neighbor states are compared to the current state, and if any of them is better, we change the current node from the current state to that neighbor state. What qualifies as better is defined by whether we use an objective function, preferring a higher value, or a decreasing function, preferring a lower value.

山爬行是一种局部搜索算法。在该算法中，会将邻居状态与当前状态进行比较，如果其中任何一个状态更好，就会将当前节点从当前状态更改为该邻居状态。什么被认为是更好的定义取决于我们是否使用目标函数，偏好更高的值，还是使用递减函数，偏好更低的值。

取决于我们的定义

A hill climbing algorithm will look the following way in pseudocode:

爬山算法的伪代码如下所示： ``` 爬山算法(): 1. 初始化当前最优解为随机解 2. 初始化当前最优解的值为当前最优解的初始值 3. 循环直到满足停止条件: a. 计算当前最优解的邻居解 b. 如果邻居解的值大于当前最优解的值: i. 将邻居解设为新的当前最优解 ii. 更新当前最优解的值为邻居解的值 c. 否则: i. 终止循环 4. 返回当前最优解 ```

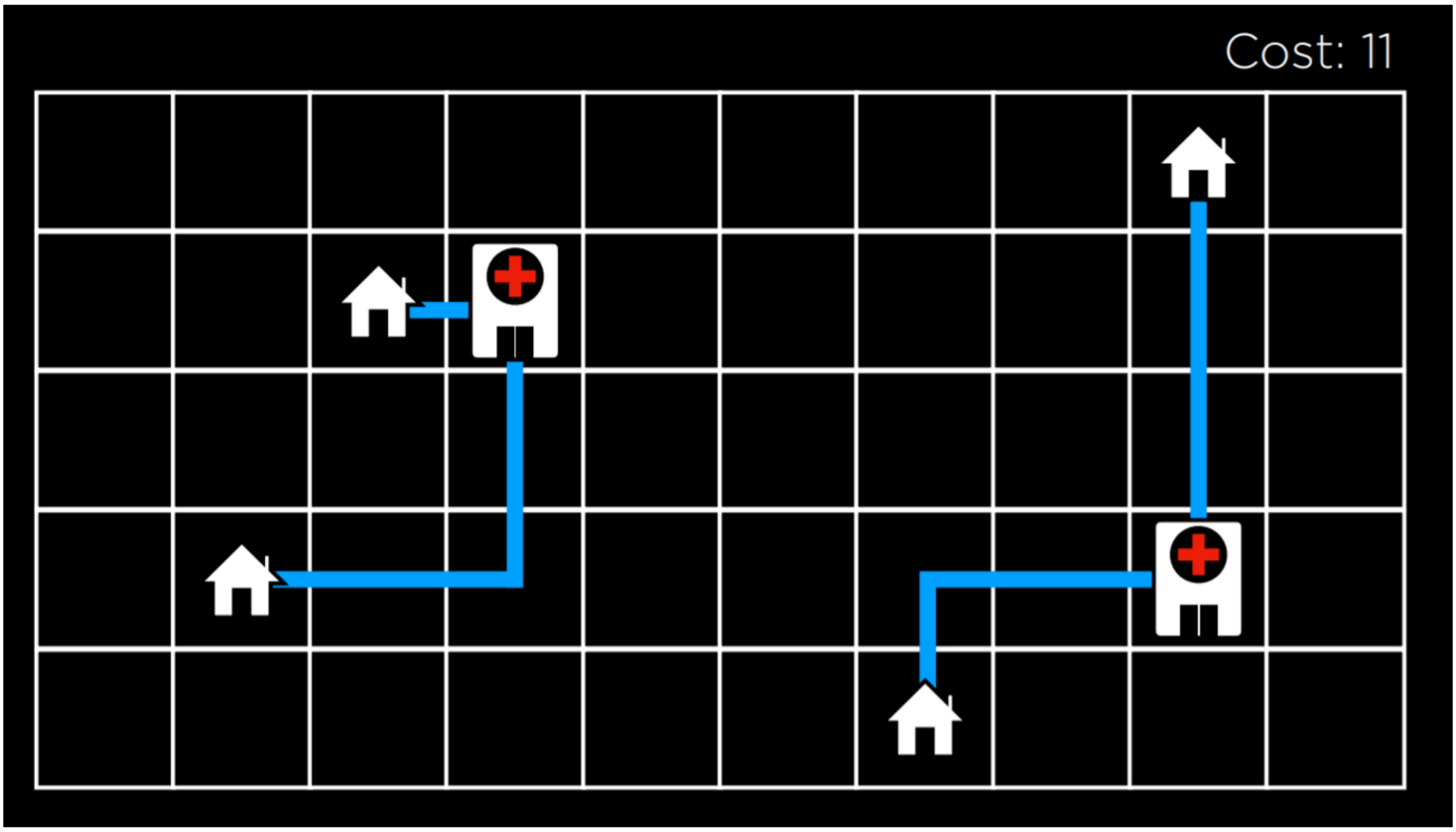
```
function Hill-Climb(problem):
函数 Hill-Climb(问题)：

  ■ current = initial state of problem
    当前 = 问题的初始状态
  ■ repeat: 重复： Translated Text: 重复
      ■ neighbor = best valued neighbor of current
        邻居 = 当前最佳评估邻居
      ■ if neighbor not better than current:
        如果邻居不如当前的：
          ■ return current 返回当前
      ■ current = neighbor 当前 = 邻居
```

In this algorithm, we start with a current state. In some problems, we will know what the current state is, while, in others, we will have to start with selecting one randomly. Then, we repeat the following actions: we evaluate the neighbors, selecting the one with the best value. Then, we compare this neighbor's value to the current state's value. If the neighbor is better, we switch the current state to the neighbor state, and then repeat the process. The process ends when we compare the best neighbor to the current state, and the current state is better. Then, we return the current state.

在这个算法中，我们从当前状态开始。在某些问题中，我们可能知道当前状态是什么，而在其他问题中，我们可能需要从随机选择一个状态开始。然后，我们重复以下操作：评估邻居状态，选择价值最高的那个。然后，我们将这个邻居状态的价值与当前状态的价值进行比较。如果邻居状态更好，我们就将当前状态切换到邻居状态，然后重复这个过程。当我们将最好的邻居状态与当前状态进行比较，且当前状态更好时，这个过程结束。然后，我们返回当前状态。

Using the hill climbing algorithm, we can start to improve the locations that we assigned to the hospitals in our example. After a few transitions, we get to the following state:

使用爬山算法，我们可以开始改进我们在示例中为医院分配的位置。经过几次转换，我们到达了以下状态：



At this state, the cost is 11, which is an improvement over 17, the cost of the initial state. However, this is not the optimal state just yet. For example, moving the hospital on the left to be underneath the top left house would bring to a cost of 9, which is better than 11. However, this version of a hill climbing algorithm can't get there, because all the neighbor states are at least as costly as the current state. In this sense, a hill climbing algorithm is short-sighted, often settling for solutions that are *better* than some others, but not necessarily the *best* of all possible solutions.

当前状态的成本为 11，这比初始状态的成本 17 有所改进。然而，这还不是最优状态。例如，将左侧的医院移动到最左上角的房子下方，成本将降至 9，这比 11 要好。然而，这种版本的爬山算法无法达到这个状态，因为所有相邻状态的成本都至少与当前状态相同。从这个意义上说，爬山算法目光短浅，往往满足于比某些其他解决方案更好的解决方案，但不一定是最优的全部可能解决方案。

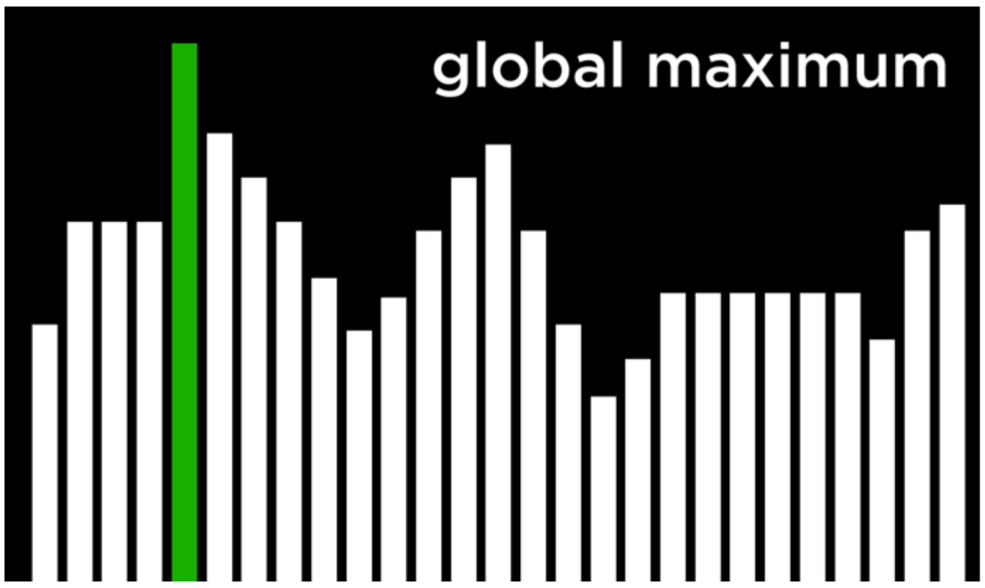
Local and Global Minima and Maxima

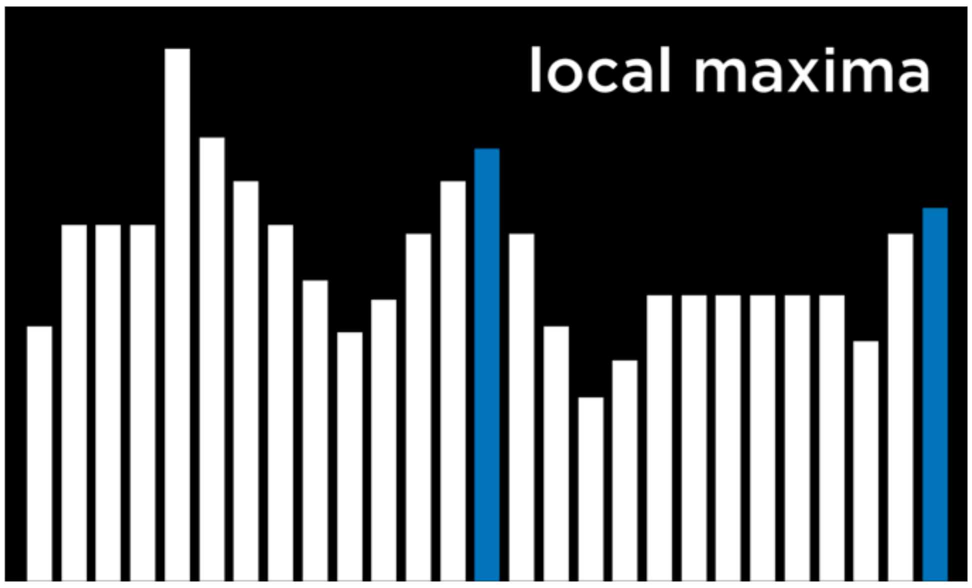
局部和全局最小值与最大值

As mentioned above, a hill climbing algorithm can get stuck in local maxima or minima. A ***local* maximum** (plural: maxima) is a state that has a higher value than its *neighboring states*. As opposed to that, a ***global* maximum** is a state that has the highest value of *all states* in the state-space.

如上所述，爬山算法可能会陷入局部最大值或最小值。局部最大值（复数形式：最大值）是指其周围状态值都比它低的状态。与此相反，全局最大值是指状态空间中所有状态中值最高的状态。

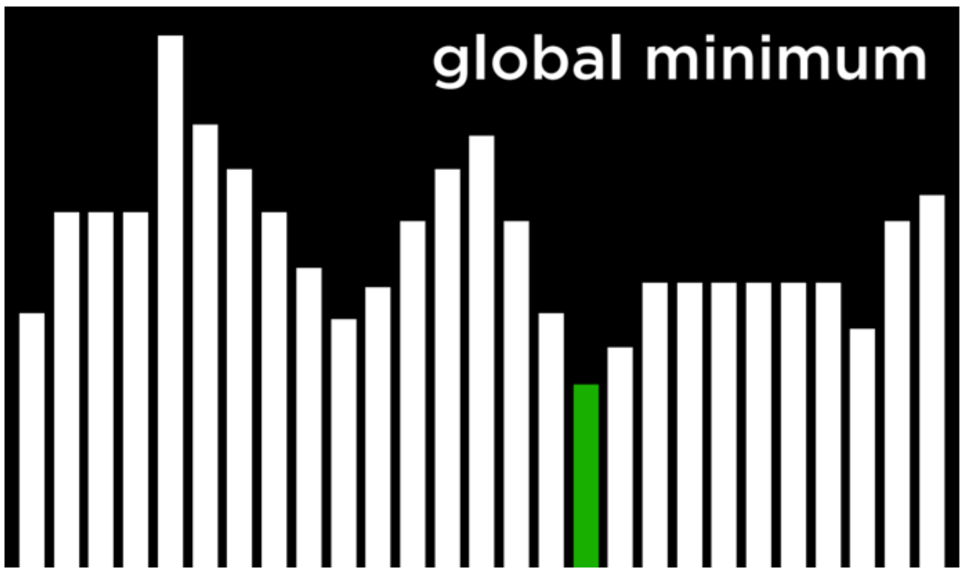
如图：

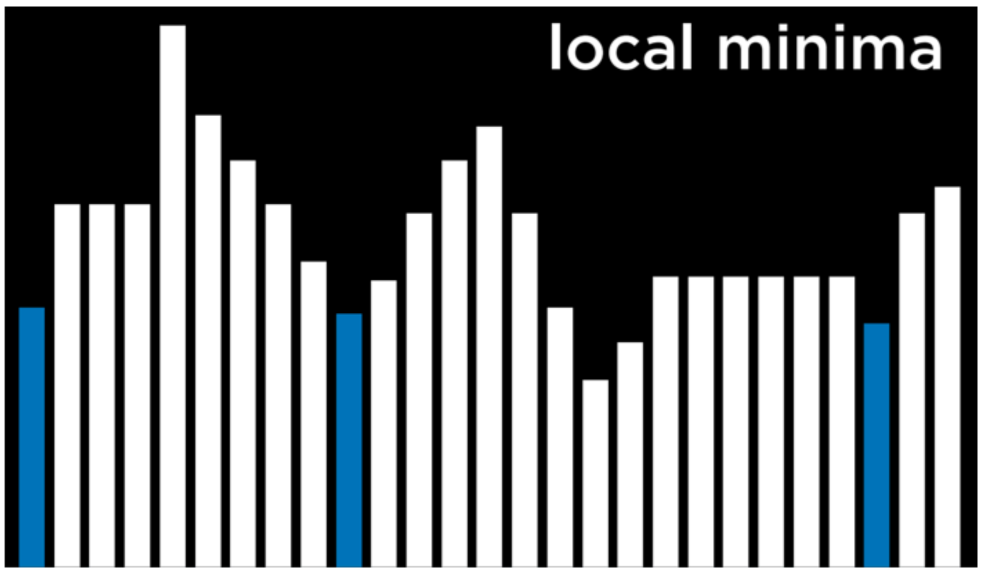




In contrast, a ***local* minimum** (plural: minima) is a state that has a lower value than its *neighboring states*. As opposed to that, a ***global* minimum** is a state that has the lowest value of *all states* in the state-space.

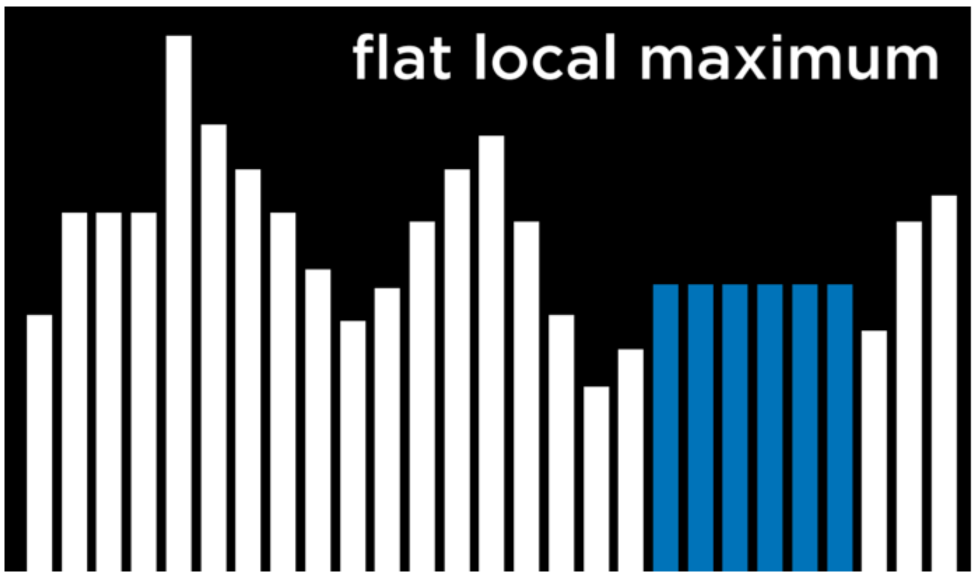
相比之下，局部最小值（复数：最小值）是指其周围状态值较低的状态。与之相反，全局最小值是指状态空间中所有状态中值最低的状态。

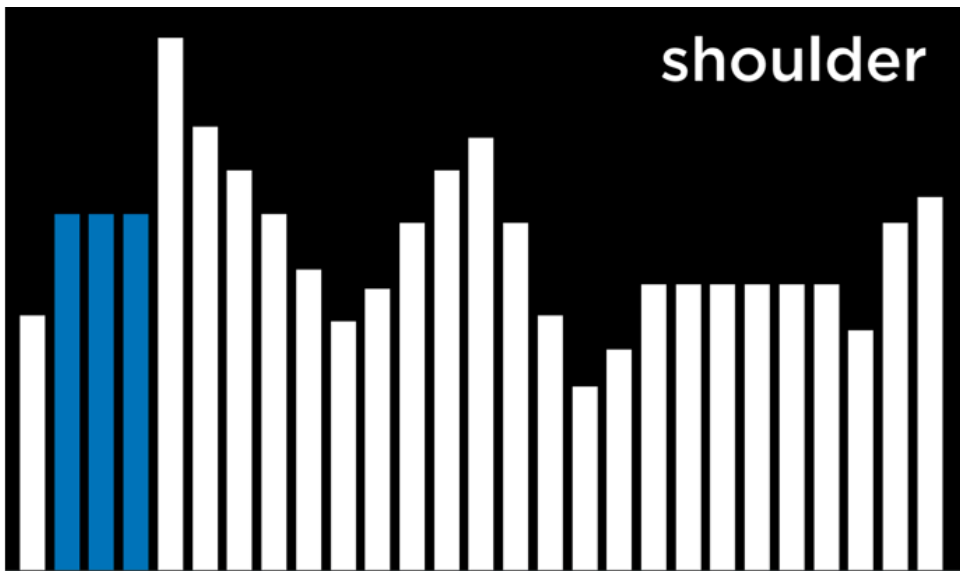




The problem with hill climbing algorithms is that they may end up in local minima and maxima. Once the algorithm reaches a point whose neighbors are worse, for the function's purpose, than the current state, the algorithm stops. Special types of local maxima and minima include the **flat local maximum/minimum**, where multiple states of equal value are adjacent, forming a plateau whose neighbors have a worse value, and the **shoulder**, where multiple states of equal value are adjacent and the neighbors of the plateau can be both better and worse. Starting from the middle of the plateau, the algorithm will not be able to advance in any direction.

爬山算法的问题在于它们可能会停留在局部最小值和最大值处。一旦算法达到一个点，其邻居对于函数的目的而言比当前状态更差，算法就会停止。局部最大值和最小值的特殊类型包括平坦的局部最大值/最小值，其中多个相等价值的状态相邻，形成一个平台，其邻居的价值更差，以及肩膀，其中多个相等价值的状态相邻，平台的邻居可以同时更好和更差。从平台的中间开始，算法将无法在任何方向上前进。





Hill Climbing Variants 山攀登变体 爬山算法变体

Due to the limitations of Hill Climbing, multiple variants have been thought of to overcome the problem of being stuck in local minima and maxima. What all variations of the algorithm have in common is that, no matter the strategy, each one still has the potential of ending up in local minima and maxima and no means to continue optimizing. The algorithms below are phrased such that a higher value is better, but they also apply to cost functions, where the goal is to minimize cost.

由于爬山法的局限性，人们提出了多种变体来克服陷入局部最小值和最大值的问题。所有算法变体的共同点是，无论采用何种策略，每一个都仍然有可能最终陷入局部最小值和最大值，并且没有方法继续优化。下面列出的算法表述为值越高越好，但它们也适用于成本函数，目标是最小化成本。

- **Steepest-ascent**: choose the highest-valued neighbor. This is the standard variation that we discussed above.

  梯度上升：选择最高值的邻居。这是我们之前讨论的标准变体。

- **Stochastic**: choose randomly from higher-valued neighbors. Doing this, we choose to go to any direction that improves over our value. This makes sense if, for example, the highest-valued neighbor leads to a local maximum while another neighbor leads to a global maximum.

  随机选择：从价值较高的邻居中随机选择。这样做，我们选择前往任何能提高我们价值的方向。如果最高价值的邻居导致局部最大值，而另一个邻居导致全局最大值，这就有意义了。

- **First-choice**: choose the first higher-valued neighbor.

  首选：选择第一个更高价值的邻接点。

- **Random-restart**: conduct hill climbing multiple times. Each time, start from a random state. Compare the maxima from every trial, and choose the highest amongst those.

  随机重启：多次执行爬山法。每次，从一个随机状态开始。比较每次试验中的最大值，选择其中最高的。

- **Local Beam Search**: chooses the *k* highest-valued neighbors. This is unlike most local search algorithms in that it uses multiple nodes for the search, and not just one.

  本地束搜索：选择 k 个最高价值的邻居。与大多数局部搜索算法不同，它使用多个节点进行搜索，而不仅仅是单个节点。

Although local search algorithms don't always give the best possible solution, they can often give a good enough solution in situations where considering every possible state is computationally infeasible.

尽管局部搜索算法并不总是能给出最佳解决方案，但在考虑所有可能状态在计算上不可行的情况下，它们通常可以给出足够好的解决方案。

Simulated Annealing 模拟退火

Although we have seen variants that can improve hill climbing, they all share the same fault: once the algorithm reaches a local maximum, it stops running. Simulated annealing allows the algorithm to "dislodge" itself if it gets stuck in a local maximum.

尽管我们看到了可以改进爬山的变体，但它们都存在同样的问题：一旦算法达到局部最大值，它就会停止运行。模拟退火允许算法"摆脱"自己，如果它被困在局部最大值中。

Annealing is the process of heating metal and allowing it to cool slowly, which serves to toughen the metal. This is used as a metaphor for the simulated annealing algorithm, which starts with a high temperature, being more likely to make random decisions, and, as the temperature decreases, it becomes less likely to make random decisions, becoming more "firm." This mechanism allows the algorithm to change its state to a neighbor that's worse than the current state, which is how it can escape from local maxima. The following is pseudocode for simulated annealing:

退火是将金属加热并使其缓慢冷却的过程，这有助于使金属变硬。这被用作模拟退火算法的隐喻，该算法以高温开始，更有可能做出随机决定，随着温度的降低，它变得不太可能做出随机决定，变得更加"坚定"。这种机制允许算法将其状态更改为当前状态比邻居状态更差的状态，这就是它如何能够逃离局部最大值的方式。以下是模拟退火的伪代码：

```
function Simulated-Annealing(problem, max):
函数 Simulated-Annealing(问题, max):

  ■ current = initial state of problem
    当前 = 问题的初始状态
  ■ for t = 1 to max:
      ■ T = Temperature(t) T = 温度(t)
      ■ neighbor = random neighbor of current
        邻居 = 随机当前邻居
      ■ ΔE = how much better neighbor is than current
        ΔE = 邻居比当前好多少
      ■ if ΔE > 0:
        如果 ΔE > 0：
          ■ current = neighbor 当前 = 邻居
      ■ with probability e^(ΔE/T) set current = neighbor
        以概率 e^(ΔE/T) 设置当前值 = 邻居值
  ■ return current 返回当前
```

The algorithm takes as input a problem and *max*, the number of times it should repeat itself. For each iteration, *T* is set using a Temperature function. This function returns a higher value in the early iterations (when *t* is low) and a lower value in later iterations (when *t* is high). Then, a random neighbor is selected, and *ΔE* is computed such that it quantifies how better the neighbor state is than the current state. If the neighbor state is better than the current state (*ΔE* > 0), as before, we set our current state to be the neighbor state. However, when the neighbor state is worse (*ΔE* < 0), we still might set our current state to be that neighbor state, and we do so with probability e^(*ΔE/T*). The idea here is that a more negative *ΔE* will result in lower probability of the neighbor state being chosen, and the higher the temperature *T* the higher the probability that the neighbor state will be chosen. This means that the worse the neighbor state, the less likely it is to be chosen, and the earlier the algorithm is in its process, the more likely it is to set a worse neighbor state as current state. The math behind this is as follows: e is a constant (around 2.72), and *ΔE* is negative (since this neighbor is worse than the current state). The more negative *ΔE*, the closer the resulting value to 0. The higher the temperature *T* is, the closer *ΔE/T* is to 0, making the probability closer to 1.

T，早期时更高，随时间逐渐减小、

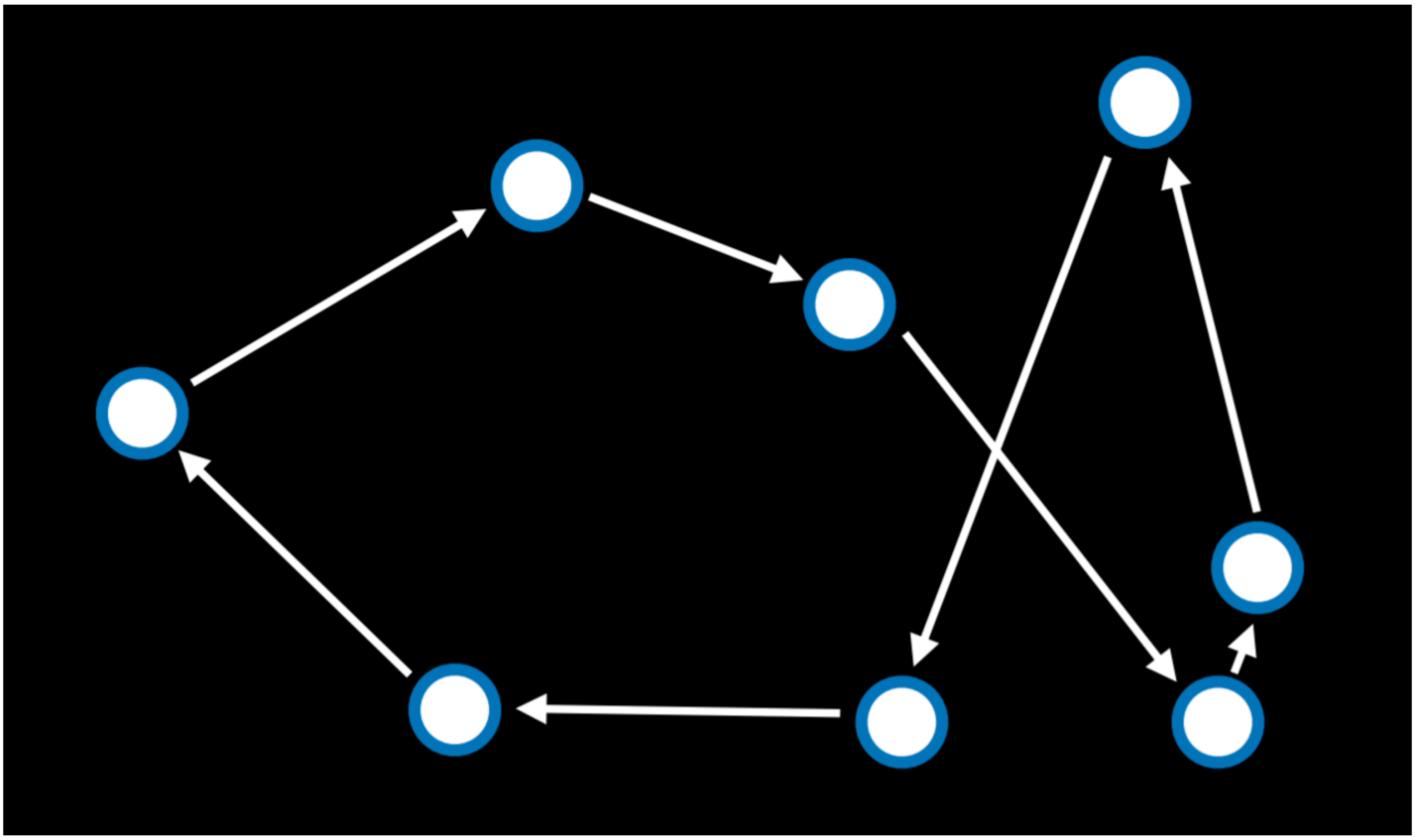
算法接受一个问题和 max 作为输入，表示它应该重复的次数。对于每一轮迭代，使用温度函数设置 T。该函数在早期迭代（当 t 较低时）返回较高的值，在后期迭代（当 t 较高时）返回较低的值。然后，选择一个随机邻居，并计算ΔE，以量化邻居状态比当前状态好多少。如果邻居状态比当前状态好（ΔE > 0），则如之前所述，我们将当前状态设置为邻居状态。然而，当邻居状态更差（ΔE < 0）时，我们仍然可能会将当前状态设置为那个邻居状态，并以概率 e^(ΔE/T)进行设置。这里的思路是，更负的ΔE 将导致邻居状态被选择的概率更低，温度 T 越高，选择邻居状态的概率越高。这意味着邻居状态越差，被选中的可能性越低，算法在其过程中的早期阶段，更有可能将更差的邻居状态设置为当前状态。背后的数学原理如下：e 是一个常数（大约为 2。第 72 行），并且ΔE 为负（因为这个邻居状态比当前状态差）。ΔE 的负值越大，最终值越接近 0。温度 T 越高，ΔE/T 越接近 0，使得概率越接近 1。

Traveling Salesman Problem

旅行推销员问题

In the traveling salesman problem, the task is to connect all points while choosing the shortest possible distance. This is, for example, what delivery companies need to do: find the shortest route from the store to all the customers' houses and back.

旅行推销员问题的目标是连接所有点，同时选择最短的距离。例如，这正是物流公司需要做的事情：找到从商店到所有客户家的最短路线，然后返回。



In this case, a neighbor state might be seen as a state where two arrows swap places. Calculating every possible combination makes this problem computationally demanding (having just 10 points gives us 10!, or 3,628,800 possible routes). By using the simulated annealing algorithm, a good solution can be found for a lower computational cost.

在这种情况下，邻国可能被视为两个箭头交换位置的国家。计算所有可能的组合会使这个问题在计算上变得复杂（只有 10 个点就给我们提供了 10!或 3,628,800 种可能的路线）。通过使用模拟退火算法，可以以较低的计算成本找到一个较好的解决方案。

Linear Programming 线性规划

Linear programming is a family of problems that optimize a linear equation (an equation of the form $y = ax_1 + bx_2 + \ldots$).

线性规划是一类问题，旨在优化线性方程（形式为 $y = ax_1 + bx_2 + \ldots$）。

Linear programming will have the following components:

线性规划将包含以下组成部分： Translation:

- A cost function that we want to minimize: $c_1x_1 + c_2x_2 + \ldots + c_nx_n$. Here, each x_- is a variable and it is associated with some cost c_-.

  我们想要最小化的成本函数：$c_1x_1 + c_2x_2 + \ldots + c_nx_n$。在这里，每个 x_- 都是一个变量，并且与一些成本 c_- 相关联。

- A constraint that's represented as a sum of variables that is either less than or equal to a value ($a_1x_1 + a_2x_2 + \ldots + a_nx_n \leqslant b$) or precisely equal to this value ($a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$). In this case, x_- is a variable, and a_- is some resource associated with it, and b is how much resources we can dedicate to this problem.

  必须是和

  约束表示为变量的和，要么小于或等于某个值（$a_1x_1 + a_2x_2 + \ldots + a_nx_n \leqslant b$），要么精确等于这个值（$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$）。在这种情况下，$x_-$ 是一个变量，a_- 是与它关联的一些资源，b 是我们能为这个问题分配的资源量。

- Individual bounds on variables (for example, that a variable can't be negative) of the form $l_i \leqslant x_i \leqslant u_i$.

  变量（例如，变量不能为负）的个体限制形式为 $l_i \leqslant x_i \leqslant u_i$。

Consider the following example:

考虑以下示例： Translated Text:

- Two machines, X_1 and X_2. X_1 costs \$50/hour to run, X_2 costs \$80/hour to run. The goal is to minimize cost. This can be formalized as a cost function: $50x_1 + 80x_2$.

  两台机器，X_1 和 X_2。运行 X_1 每小时成本为 50 美元，运行 X_2 每小时成本为 80 美元。目标是尽量降低成本。这可以形式化为一个成本函数：$50x_1 + 80x_2$。

- X_1 requires 5 units of labor per hour. X_2 requires 2 units of labor per hour. Total of 20 units of labor to spend. This can be formalized as a constraint: $5x_1 + 2x_2 \leqslant 20$.

  X_1 每小时需要 5 个劳动单位。X_2 每小时需要 2 个劳动单位。总共可以花费 20 个劳动单位。这可以形式化为一个约束条件：$5x_1 + 2x_2 \leqslant 20$。

- X_1 produces 10 units of output per hour. X_2 produces 12 units of output per hour. Company needs 90 units of output. This is another constraint. Literally, it can be rewritten as $10x_1 + 12x_2 \geqslant 90$. However, constraints need to be of the form ($a_1x_1 + a_2x_2 + \ldots + a_nx_n \leqslant b$) or ($a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$). Therefore, we multiply by (-1) to get to an equivalent equation of the desired form: $(-10x_1) + (-12x_2) \leqslant -90$.

  X_1 每小时生产 10 个单位的输出。X_2 每小时生产 12 个单位的输出。公司需要 90 个单位的输出。这是另一个限制条件。实际上，可以重写为 $10x_1 + 12x_2 \geqslant 90$。然而，限制条件需要以 ($a_1x_1 + a_2x_2 + \ldots + a_nx_n \leqslant b$) 或 ($a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$) 的形式存在。因此，我们乘以（-1）以得到所需形式的等价方程：$(-10x_1) + (-12x_2) \leqslant -90$。

An optimizing algorithm for linear programming requires background knowledge in geometry and linear algebra that we don't want to assume. Instead, we can use algorithms that already exist, such as Simplex and Interior-Point.

线性规划优化算法需要几何和线性代数的背景知识，这不是我们想要假设的。相反，我们可以使用已经存在的算法，例如单纯形法和内部点法。

The following is a linear programming example that uses the scipy library in Python:

以下是使用 Python 中的 scipy 库进行线性规划的示例： Translated Text:

```
import scipy.optimize
     目标函数
# Objective Function: 50x_1 + 80x_2
# Constraint 1: 5x_1 + 2x_2 <= 20
# Constraint 2: -10x_1 + -12x_2 <= -90 约條件

result = scipy.optimize.linprog(
    [50, 80],  # Cost function: 50x_1 + 80x_2
    A_ub=[[5, 2], [-10, -12]],  # Coefficients for inequalities
    b_ub=[20, -90],  # Constraints for inequalities: 20 and -90
)

if result.success:
    print(f"X1: {round(result.x[0], 2)} hours")
    print(f"X2: {round(result.x[1], 2)} hours")
else:
    print("No solution")
```

Constraint Satisfaction 约束满足

Constraint Satisfaction problems are a class of problems where variables need to be assigned values while satisfying some conditions.

约束满足问题是一类问题，其中需要为变量分配值，同时满足某些条件。

Constraints satisfaction problems have the following properties:

约束满足问题具有以下特性： 1. 定义明确的变量和域：问题中定义了变量，并为每个变量指定了可能的值集合，即域。 2. 明确的约束条件：存在一组规则或条件，这些规则或条件定义了变量值之间的关系，必须满足这些约束才能构成有效的解决方案。 3. 求解目标：目标是找到一组变量值的组合，使得所有约束条件都得到满足。这组满足所有约束的值组合构成了问题的一个解。 4. 可能的解的数量：约束满足问题可能有零个、一个或多个解。在某些情况下，问题可能没有解，因为没有一组变量值能够满足所有约束。 5. 求解方法：约束满足问题通常使用回溯算法、逻辑推理、启发式搜索等方法来求解。这些方法通过逐步尝试不同的变量值组合，以找到满足所有约束的解

- Set of variables ($x_1, x_2, ..., x_n$)

  变量集合（x_1，x_2，...，x_n）

- Set of domains for each variable $\{D_1, D_2, ..., D_n\}$

  每个变量的域集合 $\{D_1, D_2, ..., D_n\}$

- Set of constraints C

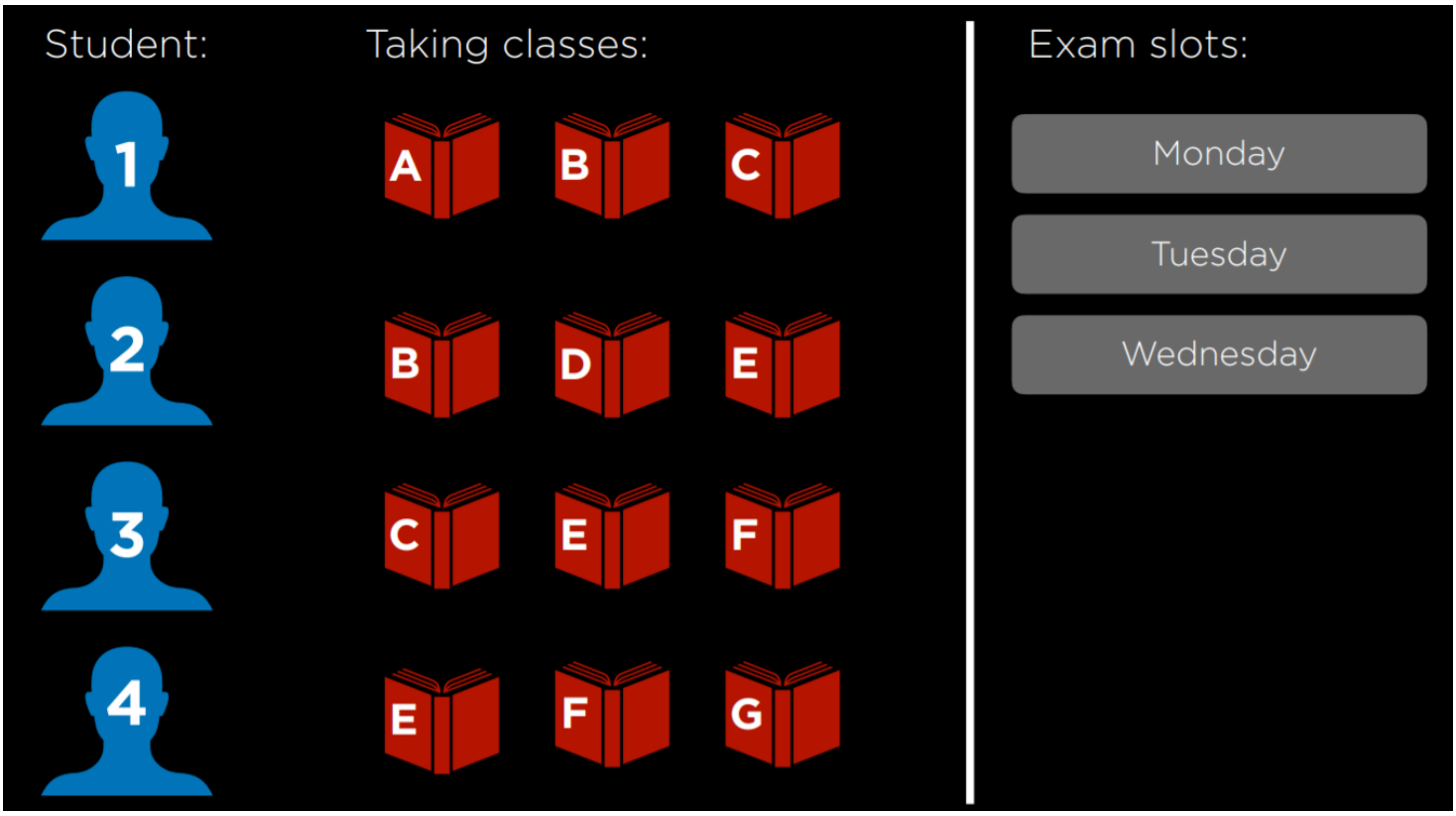
  约束集合 C

Sudoku can be represented as a constraint satisfaction problem, where each empty square is a variable, the domain is the numbers 1-9, and the constraints are the squares that can't be equal to each other.

数独可以表示为一个约束满足问题，其中每个空格是一个变量，域是数字 1-9，约束是不能相等的方格。

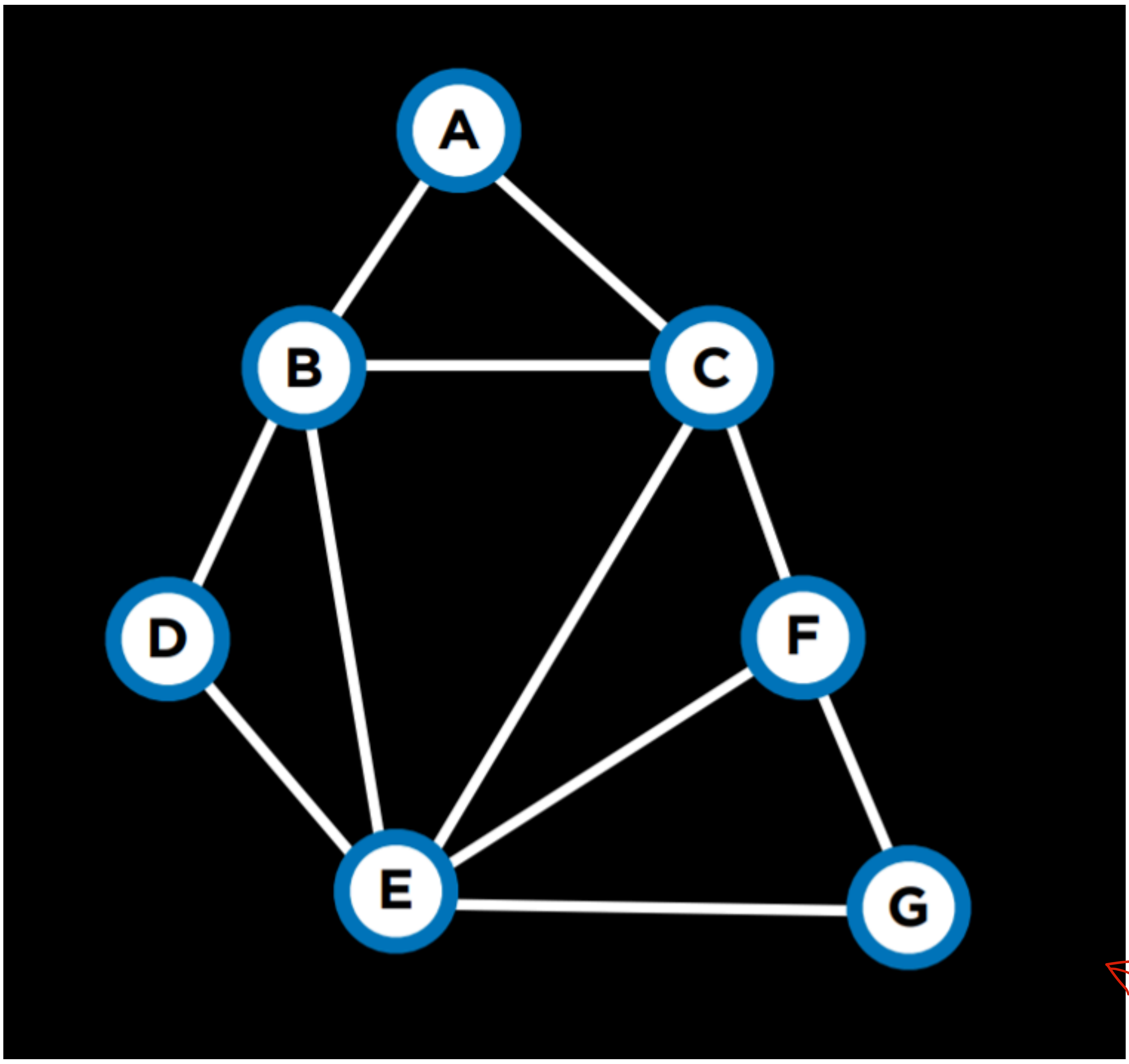
Consider another example. Each of students 1-4 is taking three courses from A, B, ..., G. Each course needs to have an exam, and the possible days for exams are Monday, Tuesday, and Wednesday. However, the same student can't have two exams on the same day. In this case, the variables are the courses, the domain is the days, and the constraints are which courses can't be scheduled to have an exam on the same day because the same student is taking them. This can be visualized as follows:

再考虑一个例子。学生 1-4 中的每一个人都从 A、B、...、G 中选修了三门课程。每门课程都需要进行考试，可能的考试日期是星期一、星期二和星期三。然而，同一名学生不能在同一天有两场考试。在这种情况下，变量是课程，领域是日期，约束是哪些课程不能安排在同一天进行考试，因为同一名学生正在选修这些课程。这可以可视化如下：



This problem can be solved using constraints that are represented as a graph. Each node on the graph is a course, and an edge is drawn between two courses if they can't be scheduled on the same day. In this case, the graph will look this:

这个问题可以通过表示为图的约束来解决。图中的每个节点代表一门课程，如果两门课程不能安排在同一天，则在它们之间绘制一条边。在这种情况下，图将看起来像这样：



A few more terms worth knowing about constraint satisfaction problems:

关于约束满足问题，以下是一些需要了解的术语： 1. 约束：定义问题中变量之间的关系或限制。 2. 变量：问题中需要找到满足所有约束的值的元素。 3. 域：每个变量可能取值的集合。 4. 解：满足所有约束的变量赋值的集合。 5. 满足：一组变量赋值使得所有约束都得到满足。 6. 枚举法：通过尝试所有可能的变量赋值来找到解的方法。 7. 回溯法：一种在枚举法中用于解决约束满足问题的算法，当遇到无法满足约束的路径时，回溯并尝试其他路径。 8. 一致性：约束满足问题中变量赋值的性质，确保在任何时刻，所有约束都得到满足。 9. 约束网络：表示问题中变量和约束之间关系的图形模型。 10. 约束逻辑编程：一种编程范式，其中程序通过定义变量、约束和解来解决问题

- A **Hard Constraint** is a constraint that must be satisfied in a correct solution.

  硬约束是一种必须在正确解决方案中满足的约束。

- A **Soft Constraint** is a constraint that expresses which solution is preferred over others.

  软约束是一种表达哪种解决方案优于其他解决方案的约束。

- A **Unary Constraint** is a constraint that involves only one variable. In our example, a unary constraint would be saying that course A can't have an exam on Monday $\{A \neq Monday\}$.

  一元约束是指只涉及一个变量的约束。在我们的例子中，一元约束可以表述为课程 A 不能在周一进行考试{A ≠ Monday}。

- A **Binary Constraint** is a constraint that involves two variables. This is the type of constraint that we used in the example above, saying that some two courses can't have the same value $\{A \neq B\}$.

  二元约束是一种涉及两个变量的约束。这是我们上面在示例中使用的那类约束，表示某些两门课程不能有相同的值{A ≠ B}。

Node Consistency 节点一致性

Node consistency is when all the values in a variable's domain satisfy the variable's unary constraints.

节点一致性是指变量域中所有值都满足该变量的一元约束。

For example, let's take two courses, A and B. The domain for each course is {*Monday, Tuesday, Wednesday*}, and the constraints are {*A ≠ Mon, B ≠ Tue, B ≠ Mon, A ≠ B*}. Now, neither A nor B is consistent, because the existing constraints prevent them from being able to take every value that's in their domain. However, if we remove Monday from A's domain, then it will have node consistency. To achieve node consistency in B, we will have to remove both Monday and Tuesday from its domain.

例如，我们考虑两门课程，A 和 B。每门课程的领域是{周一，周二，周三}，约束条件是{A ≠ 周一，B ≠ 周二，B ≠ 周一，A ≠ B}。现在，A 和 B 都不一致，因为现有的约束条件阻止它们能够取其领域中的每一个值。但是，如果我们从 A 的领域中移除周一，那么它将具有节点一致性。为了在 B 中实现节点一致性，我们将不得不从其领域中移除周一和周二。

Arc Consistency 弧一致性 (?)

Arc consistency is when all the values in a variable's domain satisfy the variable's binary constraints (note that we are now using "arc" to refer to what we previously referred to as "edge"). In other words, to make X arc-consistent with respect to Y, remove elements from X's domain until every choice for X has a possible choice for Y.

弧一致性是指变量域中的所有值满足变量的二元约束（现在我们将"弧"称为我们之前称为"边"的东西）。换句话说，为了使 X 相对于 Y 达到弧一致性，从 X 的域中移除元素，直到 X 的每一选择都有可能对应 Y 的一种选择。

X相对Y弧一致：不需满足所有Y，只需满足所有X

Consider our previous example with the revised domains: A:{*Tuesday, Wednesday*} and B:{*Wednesday*}. For A to be arc-consistent with B, no matter what day A's exam gets scheduled (from its domain), B will still be able to schedule an exam. Is A arc-consistent with B? If A takes the value Tuesday, then B can take the value Wednesday. However, if A takes the value Wednesday, then there is no value that B can take (remember that one of the constraints is A ≠ B). Therefore, A is not arc-consistent with B. To change this, we can remove Wednesday from A's domain. Then, any value that A takes (Tuesday being the only option) leaves a value for B to take (Wednesday). Now, A is arc-consistent with B. Let's look at an algorithm in pseudocode that makes a variable arc-consistent with respect to some other variable (note that csp stands for "constraint satisfaction problem").

考虑我们之前的例子，修订后的域为：A:{周二, 周三} 和 B:{周三}。为了使 A 与 B 弧一致，无论 A 的考试安排在哪个日子（从其域中），B 仍然能够安排考试。A 与 B 弧一致吗？如果 A 取值为周二，那么 B 可以取值为周三。然而，如果 A 取值为周三，那么 B 无法取任何值（记住其中一个约束是 A ≠ B）。因此，A 与 B 不弧一致。要改变这一点，我们可以从 A 的域中删除周三。然后，A 取任何值（唯一的选择是周二）都会为 B 留下取值（周三）。现在，A 与 B 弧一致。让我们看一个伪代码算法，该算法使一个变量与另一个变量弧一致（注意，csp 表示"约束满足问题"）。

↓

```
function Revise(csp, X, Y):
函数 Revise(csp, X, Y):

  - revised = false 修订 = false
  - for x in X.domain:
    for x in X.域:
      - if no y in Y.domain satisfies constraint for (X,Y):
        如果 Y 的域中没有 y 满足(X,Y)的约束条件：
          - delete x from X.domain 从 X.domain 中删除 x
          - revised = true 修订 = true
  - return revised 返回修订版
```

This algorithm starts with tracking whether any change was made to X's domain, using the variable *revised*. This will be useful in the next algorithm we examine. Then, the code repeats for every value in X's domain and sees if Y has a value that satisfies the constraints. If yes, then do nothing, if not, remove this value from X's domain.

此算法以跟踪 X 的域是否进行了任何更改开始，使用变量 revised。这将在我们接下来检查的算法中很有用。然后，代码对 X 的每个域值重复，查看 Y 是否有一个值满足约束。如果是，则什么都不做，如果不是，则从 X 的域中移除这个值。

(与X相邻接的Y，不是所有点都与X相邻接)

Often we are interested in making the whole problem arc-consistent and not just one variable with respect to another. In this case, we will use an algorithm called AC-3, which uses Revise:

经常我们对使整个问题一致化感兴趣，而不仅仅是两个变量之间的关系。在这种情况下，我们将使用一个名为 AC-3 的算法，该算法使用 Revise：

```
function AC-3(csp): 函数 AC-3(csp):

  - queue = all arcs in csp
    队列 = CSP 中的所有弧
  - while queue non-empty:  当队列非空：
```

  - *(X, Y)* = Dequeue(*queue*) (X, Y) = queue.dequeue()
  - if Revise(*csp, X, Y*):

    如果修订(csp, X, Y):

    - if size of *X*.domain == 0:

      如果 X.domain 的大小为 0：

      - return *false* 返回 false
    - for each *Z* in *X*.neighbors - {*Y*}:

      对于 X 的邻居集合中每个 Z - {Y}：

      - Enqueue(queue, (*Z,X*)) 入队(queue, (Z,X))
- return true 返回 true

This algorithm adds all the arcs in the problem to a queue. Each time it considers an arc, it removes it from the queue. Then, it runs the Revise algorithm to see if this arc is consistent. If changes were made to make it consistent, further actions are needed. If the resulting domain of X is empty, it means that this constraint satisfaction problem is unsolvable (since there are no values that X can take that will allow Y to take any value given the constraints). If the problem is not deemed unsolvable in the previous step, then, since X's domain was changed, we need to see if all the arcs associated with X are still consistent. That is, we take all of X's neighbors except Y, and we add the arcs between them and X to the queue. However, if the Revise algorithm returns false, meaning that the domain wasn't changed, we simply continue considering the other arcs.

此算法将问题中的所有弧线添加到队列中。每次考虑一个弧线时，将其从队列中移除。然后，运行修订算法以查看此弧线是否一致。如果进行了修改使其一致，则需要进一步行动。如果 X 的结果域为空，这意味着此约束满足问题无法解决（因为 X 无法取任何值，使得 Y 可以取任何给定约束的值）。如果在上一步中未判定问题无法解决，则，由于 X 的域发生了变化，我们需要查看与 X 关联的所有弧线是否仍然一致。也就是说，我们取除了 Y 之外的所有 X 的邻居，然后将它们与 X 之间的弧线添加到队列中。但是，如果修订算法返回 false，意味着域没有改变，我们只需继续考虑其他弧线。

即 (Z, X)

While the algorithm for arc consistency can simplify the problem, it will not necessarily solve it, since it considers binary constraints only and not how multiple nodes might be interconnected. Our previous example, where each of 4 students is taking 3 courses, remains unchanged by running AC-3 on it.

虽然弧一致性算法可以简化问题，但它不一定能解决问题，因为它只考虑二元约束，而不考虑多个节点可能的相互连接方式。我们之前的例子，即 4 名学生每人选修 3 门课程，运行 AC-3 算法后不会改变。

We have encountered search problems in our first lecture. A constraint satisfaction problem can be seen as a search problem:

我们在第一堂课中遇到了搜索问题。约束满足问题可以被视为搜索问题：

目的就是求解

- Initial state: empty assignment (all variables don't have any values assigned to them).

  初始状态：空任务（所有变量都没有任何值被分配）。
- Actions: add a {*variable* = *value*} to assignment; that is, give some variable a value.

  操作：向赋值中添加 {变量 = 值}；也就是说，给某个变量赋一个值。
- Transition model: shows how adding the assignment changes the assignment. There is not much depth to this: the transition model returns the state that includes the assignment following the latest action.

  过渡模型：展示添加分配如何改变分配。这没有太多深度：过渡模型返回包含最新操作后分配状态的状态。
- Goal test: check if all variables are assigned a value and all constraints are satisfied.

  目标测试：检查所有变量是否都被分配了值，所有约束是否都得到满足。
- Path cost function: all paths have the same cost. As we mentioned earlier, as opposed to typical search problems, optimization problems care about the solution and not the route to the solution.

  路径成本函数：所有路径的成本相同。正如我们之前提到的，与典型搜索问题不同，优化问题关心的是解决方案，而不是达到解决方案的途径。

However, going about a constraint satisfaction problem naively, as a regular search problem, is massively inefficient. Instead, we can make use of the structure of a constraint satisfaction problem to solve it more efficiently.

然而，以一种直觉的方式处理约束满足问题，就像处理一个常规的搜索问题一样，是非常低效的。相反，我们可以利用约束满足问题的结构来更有效地解决它。

Backtracking Search 回溯搜索

Backtracking search is a type of a search algorithm that takes into account the structure of a constraint satisfaction search problem. In general, it is a recursive function that attempts to continue assigning values as long as they satisfy the constraints. If constraints are violated, it tries a different assignment. Let's look at the pseudocode for it:

回溯搜索是一种考虑约束满足(搜索问题结构)的搜索算法。一般来说，它是一个递归函数，尝试在满足约束的情况下继续分配值。如果违反了约束，它会尝试不同的分配。让我们看看它的伪代码： ```cpp void backtrack_search() { if (所有变量都已分配值) { 找到解决方案; 返回; } 为未分配值的变量选择一个值; 如果该值满足所有约束 { 对下一个未分配值的变量递归调用 backtrack_search(); } 如果递归调用返回无解 { 返回上一个步骤，选择下一个不同的值; } } ```

```
function Backtrack(assignment, csp):
函数 Backtrack(assignment, csp):

- if assignment complete: 如果任务完成：
    - return assignment 归还作业
- var = Select-Unassigned-Var(assignment, csp)    选择未赋值的变量.
  变量 = Select-Unassigned-Var(assignment, csp)
- for value in Domain-Values(var, assignment, csp):
    - if value consistent with assignment:
      如果值与分配一致： Translated Text:
      (即满足约束条件)
        - add {var = value} to assignment
          将 {var = value} 添加到赋值操作
        - result = Backtrack(assignment, csp)    递归，为下一个变量赋值.
          结果 = 回溯(分配, csp)
        - if result ≠ failure:
          如果结果 ≠ 失败：
            - return result 返回结果
        - remove {var = value} from assignment
          从赋值中移除 {var = value}
- return failure 返回失败
```

In words, this algorithm starts with returning the current assignment if it is complete. This means that, if the algorithm is done, it will not perform any of the additional actions. Instead, it will just return the completed assignment. If the assignment is not complete, the algorithm selects any of the variables that do not have an assignment yet. Then, the algorithm tries to assign a value to the variable, and runs the Backtrack algorithm again on the resulting assignment (recursion). Then, it checks the resulting value. If it is not *failure*, it means that the assignment worked out, and it should return this assignment. If the resulting value is *failure*, then the latest assignment is removed, and a new possible value is attempted, repeating the same process. If all possible values in the domain returned *failure*, this means that we need to backtrack. That is, that the problem is with some previous assignment. If this happens with the variable we start with, then it means that no solution satisfies the constraints.

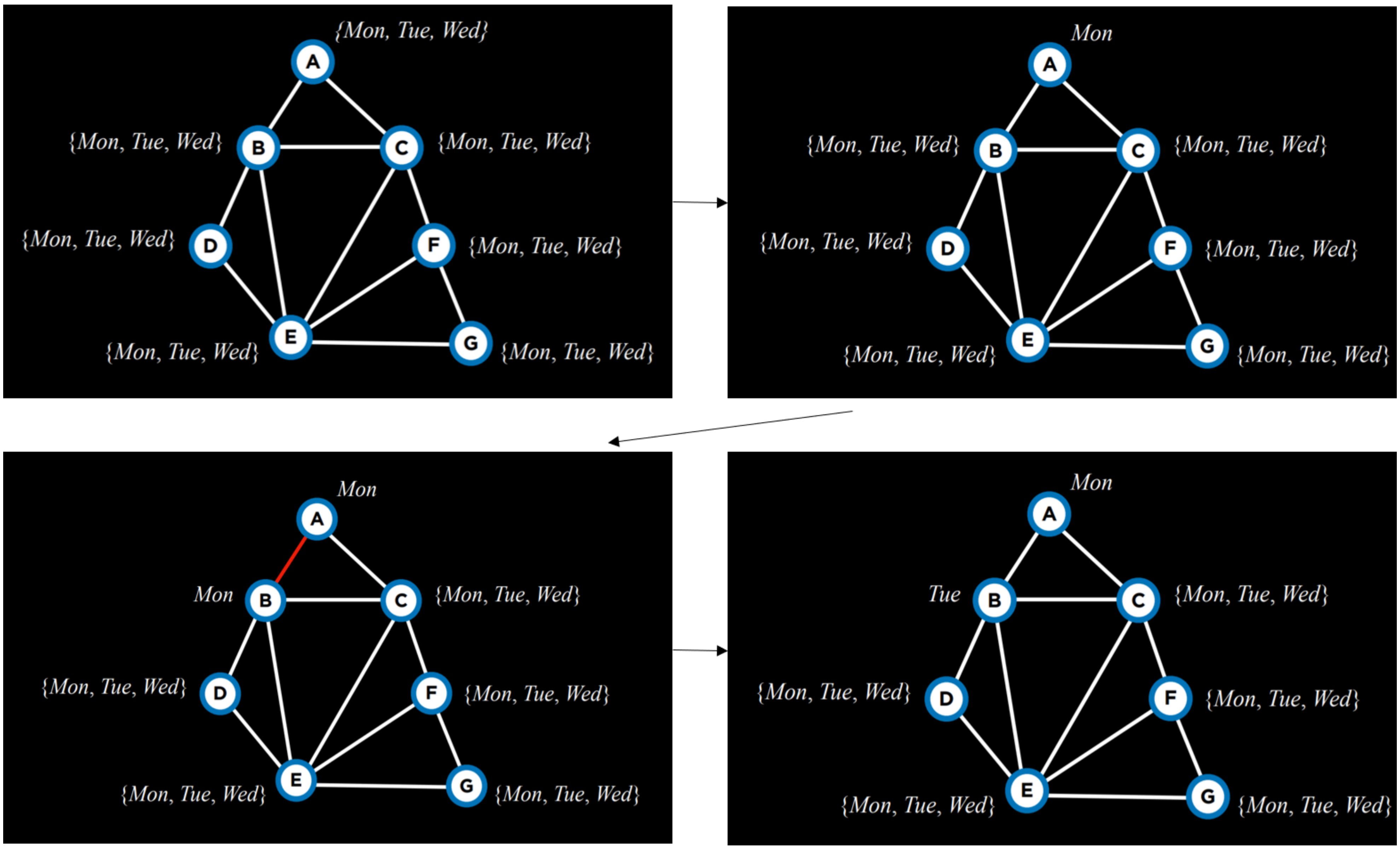
解释

在文字上，该算法从返回当前的分配开始，如果当前的分配已经完成。这意味着，如果算法已经完成，它将不会执行任何额外的操作。相反，它只会返回已完成的分配。如果分配没有完成，算法会选择任何还没有分配值的变量。然后，算法尝试为该变量分配一个值，并对结果的分配再次运行回溯算法（递归）。然后，检查结果的值。如果它不是失败，这意味着分配成功，应该返回这个分配。如果结果的值是失败，这意味着最近的分配被删除，并尝试新的可能值，重复同样的过程。如果域中所有可能的值都返回失败，这意味着我们需要回溯。也就是说，问题可能与之前的某个分配有关。如果这种情况发生在我们开始的变量上，这意味着没有任何解决方案满足约束。

Consider the following course of action:

例子

考虑以下行动方案：



We start with empty assignments (top left). Then, we choose the variable A, and assign to it some value, Monday (top right). Then, using this assignment, we run the algorithm again. Now that A already has an assignment, the algorithm will consider B, and assign Monday to it (bottom left). This assignment returns false, so instead of assigning a value to C given the previous assignment, the algorithm will try to assign a new value to B, Tuesday (bottom right). This new assignment satisfies the constraints, and a new variable will be considered next given this assignment. If, for example, assigning also Tuesday or Wednesday to B would bring to a failure, then the algorithm would backtrack and return to considering A, assigning another value to it, Tuesday. If also Tuesday and Wednesday return *failure*, then it means we have tried every possible assignment and the problem is unsolvable.

我们从空任务开始（左上角）。然后，我们选择变量 A，并将其赋值为某个值，星期一（右上角）。然后，使用这个赋值，我们再次运行算法。既然 A 已经有了赋值，算法将考虑 B，并将星期一赋给它（左下角）。这个赋值返回了假，因此算法不会根据之前的赋值给 C 赋值，而是尝试给 B 赋新的值，星期二（右下角）。这个新的赋值满足了约束，根据这个赋值，将考虑下一个变量。如果，例如，将星期二或星期三赋给 B 会导致失败，那么算法将回溯并回到考虑 A，给它赋另一个值，星期二。如果星期二和星期三也导致失败，那就意味着我们尝试了所有可能的赋值，问题无法解决。

In the source code section, you can find an implementation from scratch of the backtrack algorithm. However, this algorithm is widely used, and, as such, multiple libraries already contain an implementation of it.

在源代码部分，你可以找到回溯算法从头开始的实现。然而，这个算法被广泛使用，因此，已经有多库包含了它的实现。

Inference 推断

Although backtracking search is more efficient than simple search, it still takes a lot of computational power. Enforcing arc consistency, on the other hand, is less resource intensive. By interleaving backtracking search with inference (enforcing arc consistency), we can get at a more efficient algorithm. This algorithm is called the **Maintaining Arc-Consistency** algorithm. This algorithm will enforce arc-consistency after every new assignment of the backtracking search. Specifically, after we make a new assignment to X, we will call the AC-3 algorithm and start it with a queue of all arcs (*Y,X*) where Y is a neighbor of X (and not a queue of all arcs in the problem). Following is a revised Backtrack algorithm that maintains arc-consistency, with the new additions in **bold**.

即 AC-3 算法

尽管回溯搜索比简单搜索更有效率，但它仍然需要大量的计算能力。另一方面，强制弧一致性消耗的资源较少。通过将回溯搜索与推理（强制弧一致性）交织在一起，我们可以得到一个更高效的算法。这个算法被称为保持弧一致性算法。这个算法将在回溯搜索的每次新分配后强制执行弧一致性。具体来说，在我们对 X 进行新的~~分~~配后，我们将调用 AC-3 算法，并从所有与 X 相邻的（Y,X）弧的队列开始（而不是从问题中的所有弧的队列开始）。以下是保持弧一致性算法的修订版回溯算法，其中加粗的部分是新增的内容。

赋值

function Backtrack(*assignment*, *csp*):

函数 Backtrack(assignment, csp):

- if *assignment* 分配 complete: 如果~~任~~务完成：
  - return *assignment* 归还作~~业~~ 分配

- *var* = Select-Unassigned-Var(*assignment, csp*)

  变量 = Select-Unassigned-Var(assignment, csp)

- for *value* in Domain-Values(*var, assignment, csp*):
  - if *value* consistent with *assignment*:

    如果值与分配一致： Translated Text:

    - **add {*var* = *value*} to *assignment***

      将 {var = value} 添加到赋值操作

    - ***inferences* = Inference(*assignment, csp*)**

      推断 = Inference(分配, csp)

    - if *inferences* ≠ *failure*:

      如果推理 ≠ 失败：

      - add *inferences* to *assignment* 赋值/分配

        在任~~务~~中添加推断

    - *result* = Backtrack(*assignment, csp*)

      结果 = 回溯(分配, csp)

    - if *result* ≠ *failure*:

      如果结果 ≠ 失败：

      - return *result* 返回结果

    - *remove* {*var* = *value*} **and *inferences*** from *assignment*

      删除{var = value}以及赋值语句的推断

- return failure 返回失败

The Inference function runs the AC-3 algorithm as described. Its output is all the inferences that can be made through enforcing arc-consistency. Literally, these are the new assignments that can be deduced from the previous assignments and the structure of the constrain satisfaction problem.

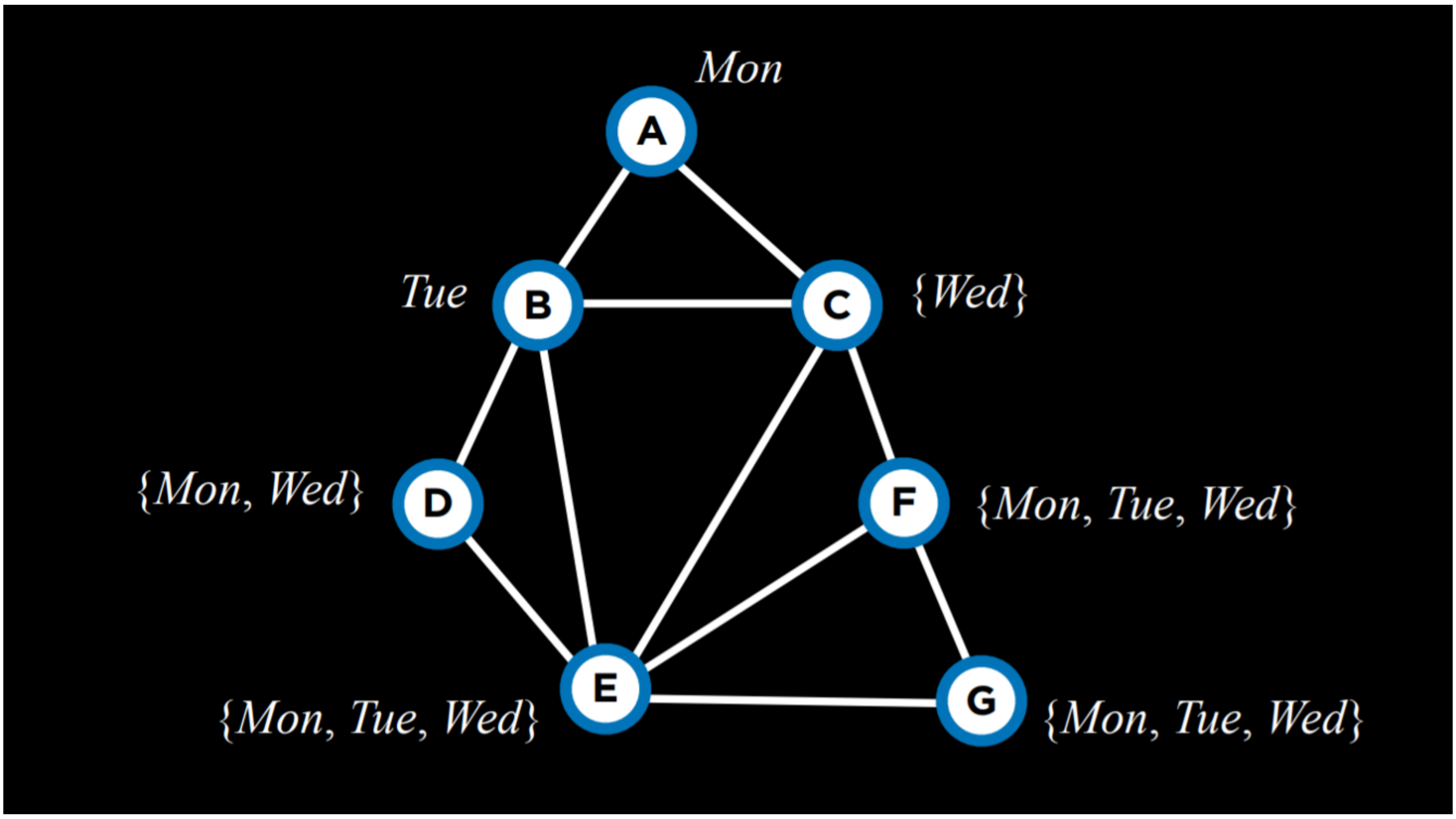
推断函数运行 AC-3 算法，如所描述的那样。其输出是通过强制弧一致性可以做出的所有推断。字面上讲，这些是从之前的分配和约束满足问题的结构中可以推导出的新分配。

There are additional ways to make the algorithm more efficient. So far, we selected an unassigned variable randomly. However, some choices are more likely to bring to a solution faster than others. This requires the use of heuristics. A heuristic is a rule of thumb, meaning that, more often than not, it will bring to a better result than following a naive approach, but it is not guaranteed to do so.

优化算法效率的额外方法有很多。到目前为止，我们随机选择了一个未分配的变量。然而，并非所有选择都能更快地带来解决方案。这需要使用启发式方法。启发式方法是一种经验法则，意味着它通常能带来比盲目方法更好的结果，但并不保证一定能达到最佳结果。

Minimum Remaining Values (MRV) is one such heuristic. The idea here is that if a variable's domain was constricted by inference, and now it has only one value left (or even if it's two values), then by making this assignment we will reduce the number of backtracks we might need to do later. That is, we will have to make this assignment sooner or later, since it's inferred from enforcing arc-consistency. If this assignment brings to failure, it is better to find out about it as soon as possible and not backtrack later.

最小剩余值（MRV）就是这样一种启发式方法。这里的思路是，如果通过推理缩小了变量的域，现在只剩下一个值（甚至如果有两个值），那么通过进行这个分配，我们可以在稍后需要回溯时减少回溯次数。也就是说，我们迟早必须做出这个分配，因为它是由强制弧一致性得出的。如果这个分配导致失败，最好尽早发现并避免稍后进行回溯。

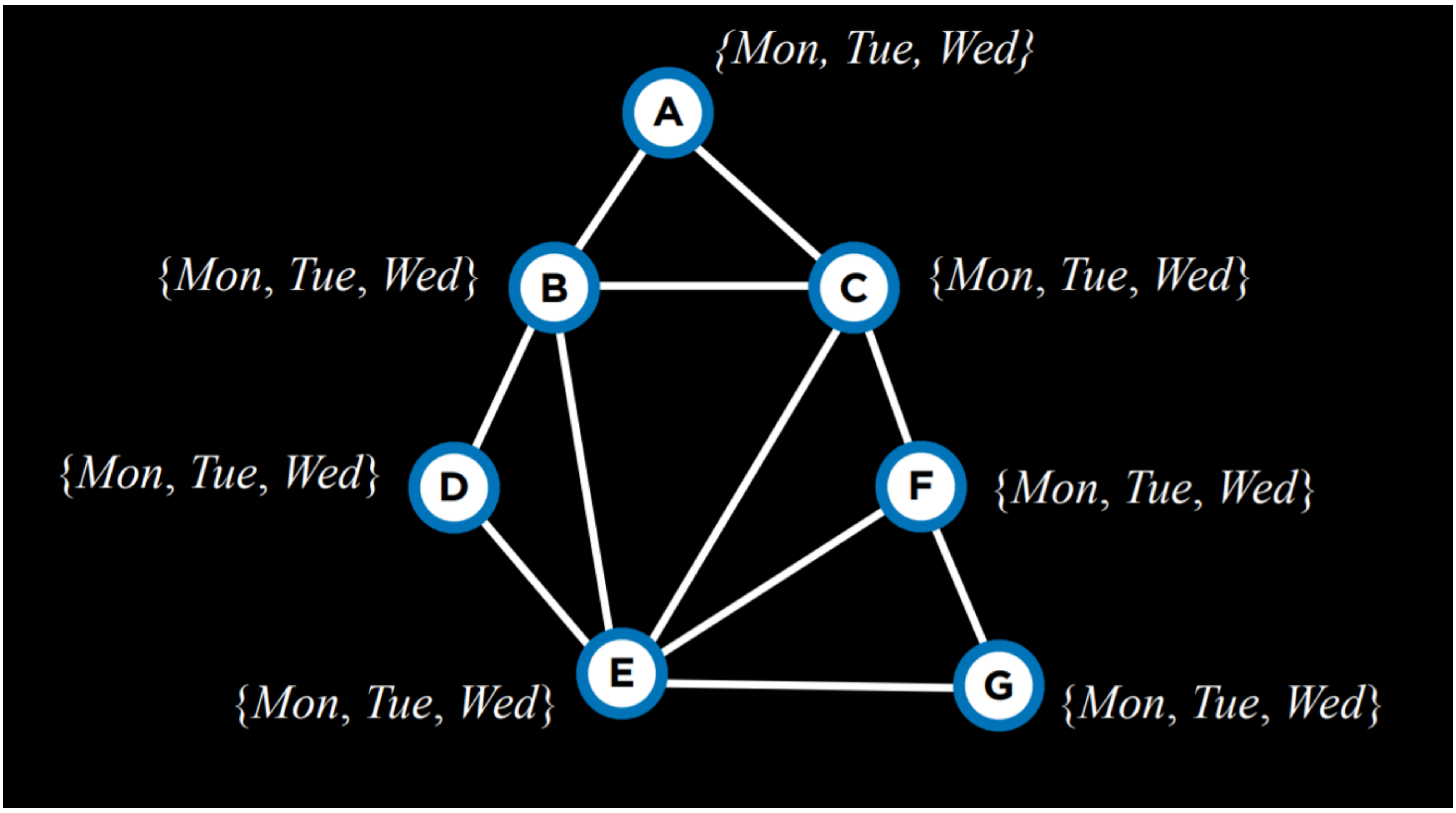


For example, after having narrowed down the domains of variables given the current assignment, using the MRV heuristic, we will choose variable C next and assign the value Wednesday to it.

例如，根据当前的分配缩小了变量的领域后，使用 MRV 启发式，我们将选择变量 C 并为其分配值“星期三”。

The **Degree** heuristic relies on the degrees of variables, where a degree is how many arcs connect a variable to other variables. By choosing the variable with the highest degree, with one assignment, we constrain multiple other variables, speeding the algorithm’s process.

度量启发式方法依赖于变量的度，度是指连接一个变量与其他变量的弧的数量。通过选择度数最高的变量，并进行一次赋值，我们可以约束多个其他变量，从而加速算法的过程。



For example, all the variables above have domains of the same size. Thus, we should pick a domain with the highest degree, which would be variable E.

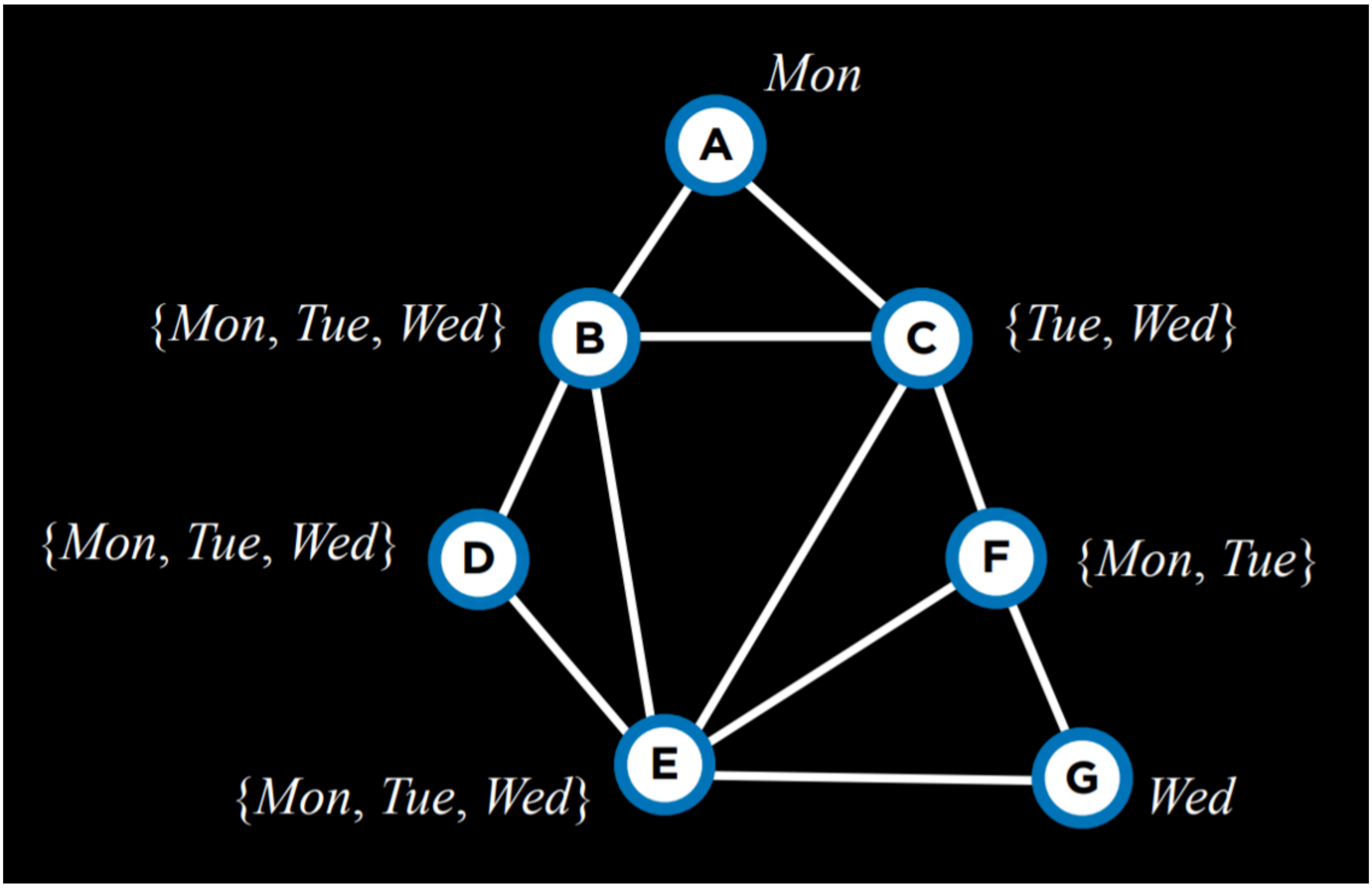
例如，上述所有变量的域大小相同。因此，我们应该选择具有最高阶数的域，这将是变量 E。

Both heuristics are not always applicable. For example, when multiple variables have the same least number of values in their domain, or when multiple variables have the same highest degree.

两种启发式方法并非总是适用。例如，当多个变量在其域中具有相同最少值的数量时，或者当多个变量具有相同的最高阶数时。

Another way to make the algorithm more efficient is employing yet another heuristic when we select a value from the domain of a variable. Here, we would like to use the **Least Constraining Values** heuristic, where we select the value that will constrain the least other variables. The idea here is that, while in the degree heuristic we wanted to use the variable that is more likely to constrain other variables, here we want this variable to place the least constraints on other variables. That is, we want to locate what could be the largest potential source of trouble (the variable with the highest degree), and then render it the least troublesome that we can (assign the least constraining value to it).

提高算法效率的另一种方法是在选择变量域中的值时应用另一种启发式方法。在这里，我们希望使用最少约束值启发式方法，即选择对其他变量约束最少的值。这里的理念是，在度启发式方法中，我们希望使用最有可能约束其他变量的变量，而在这里，我们希望这个变量对其他变量的约束最少。也就是说，我们想要找到可能的最大问题源（具有最高度的变量），然后尽可能地将其变为最小问题（为其分配最少约束的值）。



For example, let's consider variable C. If we assign Tuesday to it, we will put a constraint on all of B, E, and F. However, if we choose Wednesday, we will put a constraint only on B and E. Therefore, it is probably better to go with Wednesday.

例如，让我们考虑变量 C。如果我们将其分配为星期二，我们将对 B、E 和 F 的所有都施加约束。然而，如果我们选择星期三，我们只会对 B 和 E 施加约束。因此，选择星期三可能更好。

To summarize, optimization problems can be formulated in multiple ways. Today we considered local search, linear programming, and constraint satisfaction.

总结起来，优化问题可以以多种方式表述。今天，我们考虑了局部搜索、线性规划和约束满足。