1. **爬山搜索与贪心最佳优先搜索算法类似，是一个标准的迭代改进算法。爬山搜索的主要问题是什么?**
2. 描述模拟退火搜索。描述模拟退火搜索算法与局部搜索的区别和联系。

爬山算法带来了几个缺点。可能经过大量的搜索后仍未找到答案，但没有更多的方法可以改善状况。如果出现这种情况，可能已经达到了局部最大值、高原区，或者山脊。

局部最大值：

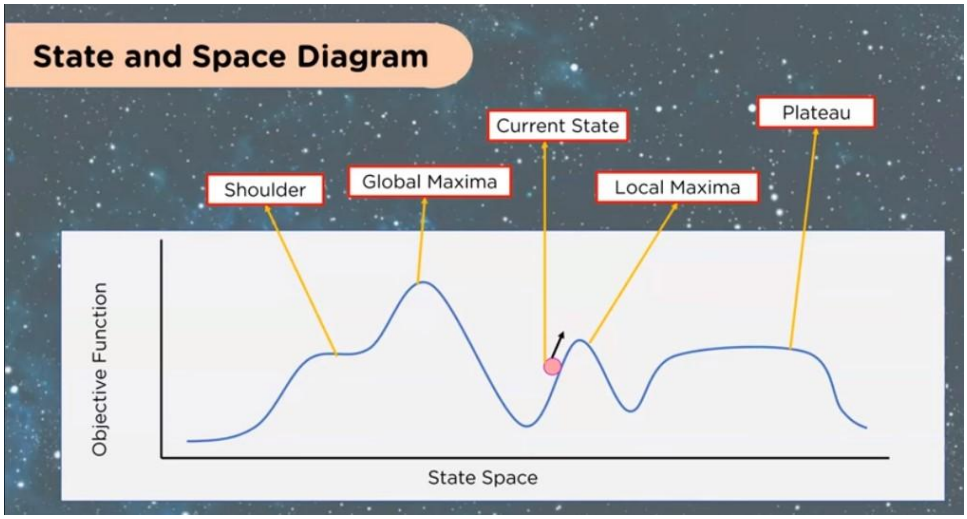
这是一种比所有相邻状态都要好，但不如某些较远状态的状态（前面可能有更好的解决方案，这种解决方案被称为全局最大值）。从这个位置出发，所有的动作似乎都变得更糟。如果这种情况发生，应该回到之前的状态，尝试从不同角度解决问题。

高原区：

搜索空间中的一片水平区域，邻近的所有状态都具有相同的值。无法提前选择最佳路径。在这种情况下，应执行急转弯，并尝试到达搜索区域的另一个区域。它也被称为平顶最大值。

山脊：

这是搜索空间中的一个区域，比周围地形更高，但无法通过单一方向的单一移动到达。这个特殊的局部最大值是独一无二的。在这里，你应该在考试前使用两个或更多的指导原则，这涉及到同时向多个方向前进。



3. **描述模拟退火搜索。描述模拟退火搜索算法与局部搜索的区别和联系。**

模拟退火（SA）是一种受冶金学中退火过程启发的随机局部搜索算法。它是一种用于近似全局优化各种优化问题的有效且广泛使用的技术。与局部搜索算法类似，它从问题的一个初始解开始，然后通过对当前解进行小的修改来迭代地探索邻近的解决方案。然而，与爬山算法等局部搜索算法不同，模拟退火允许偶尔进行劣化转换。这种进行劣化转换的能力有助于克服陷入局部最优的问题。

模拟退火引入了温度参数的概念，它控制着接受劣解的概率。在迭代过程中，算法会生成一个邻近解并根据成本或目标函数评估其质量。

如果新解比当前解更好，它就会被接受为新的当前解。然而，即使新解更糟，它仍可能以一定概率被接受。接受劣解的概率由一个公式决定，该公式依赖于温度和新旧解之间的成本差异。这个公式的设计使得在温度较高时接受劣解的概率更高。随着算法的进行，温度逐渐降低。随着迭代的进行，转移到劣解的可能性变小，过程趋于稳定。

模拟退火继续迭代并探索搜索空间，直到满足停止准则（如达到最大迭代次数或达到最终温度）或观察到没有进一步的改进为止。在迭代过程中获得的最佳解是问题最优解的一个近似值。

在这个回答中，我们将深入算法的步骤并检查其伪代码。继续之后，我们将应用此算法。

这里是问题的正式描述：

让 S 代表一个给定优化问题所有可能解决方案的集合。让特定的解决方案 $S \in S$。成本函数 $f : S \to \mathbb{R}$ 为每个解决方案赋予一个实数作为成本值，$f(S)$ 代表解决方案 S 的成本。我们的目标是在所有可能的解决方案 S 上最小化成本函数 $f(S)$。这可以被公式化为寻找最小值：$\min f(S)$。

如果我们想要最大化 $f(S)$，我们可以通过取负 $f(S)$ 函数来将问题看作是一个最小化问题：$\max f(S) = \min (-f(S))$。

以下是算法的步骤：

1. 初始化：从问题的一个初始解开始。这可以是一个随机解或通过某种方法得到的解决方案。
2. 定义参数：设置初始温度和冷却时间表。通常，初始温度 $T_0 = 100$ 并且通过公式

T_new = αT_current 进行冷却，其中 α ∈ [0.7, 0.99] 是一个冷却因子。设置每个温度的迭代次数：通常，这个数字在 100 到 1000 之间。

3. 迭代过程：执行迭代直到满足停止标准。这可以是最大迭代次数或达到最终温度。

4. 生成一个邻近解：修改当前解以生成一个邻近解。修改可以是交换变量或改变值。

5. 评估邻近解：计算邻近解的成本或目标函数值，表示其质量。邻近解 S_neighbor 的成本值 f(S_neighbor) 应被评估。

6. 接受标准：确定是否接受邻近解作为新的当前解。如果邻近解比当前解更好，接受它：S_new = S_neighbor 如果 f(S_current) > f(S_neighbor)。如果邻近解更糟，也可能根据接受概率公式接受它：S_new = S_neighbor 如果 exp((-f(S_neighbor) - f(S_current)) > r，其中 r ∈ [0, 1] 是某个随机数。

7. 更新温度：根据冷却时间表调整温度：T_new = αT_current。

8. 重复：返回步骤四并继续迭代过程。

9. 终止：当满足停止标准时停止算法。

10. 输出：返回在迭代过程中获得的最佳解（具有最低成本值）。这个解是对问题全局最优解的近似。

```
{
    //Get initial random solution
    Solution currentSolution = InitSolution(problemInstance);
    //Initialize parameters
    double T = InitTemperature();
    const double coolingFactor = InitCoolingFactor();
    const int stepsPerT = InitStepsPerTemperature();
    Solution bestSolution = solution;
    while(!StopCriteria())
    {
        for(int i = 0; i < stepsPerT; ++i)
        {
            //Generate neighbor solution
            Solution nextSolution = GetNeighbor(currentSolution);
            // Calculate difference between cost values of new and current solutions
            double delta = nextSolution.GetCostValue() - currentSolution.GetCostValue();
            // Acceptance criterion
            if(delta < 0 || exp(-delta / T) > GetRandom(0, 1))
            {
                currentSolution = nextSolution;
                if(currentSolution.GetCostValue() < bestSolution.GetCostValue())
                {
                    bestSolution = currentSolution;
                }
            }

        }
        T *= coolingFactor;
    }
    return solution;
}
```