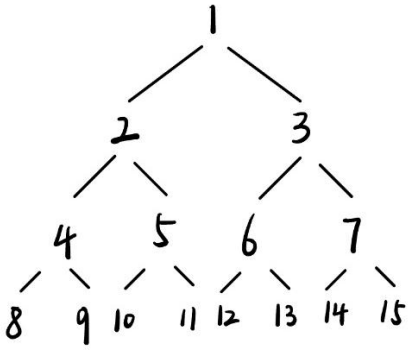
Exercise 17

1. **错。**DFS 可能沿最短路径直接找到目标结点，仅需扩展 **d**（目标结点深度）个结点；而带有可采纳的启发式的 A*算法可能因启发式引导选择探索周边结点以保证找到最优解。所以 DFS 扩展的结点数可能多于 A*算法。
2. **对。**因为代价总是**非负**的，所以结点 n 的实际代价一定大于等于 h(n)=0，所以 h(n)一定是可采纳的。
3. **错。**虽然机器人感知和状态空间通常是连续的，但可通过离散化操作将问题转化为 A*适用的离散状态空间，所以 A*算法在机器人学中仍然有重要作用。
4. **对。**BFS 的完备性与其遍历策略和状态空间的性质相关，与边成本无关。若存在解，即使允许零成本边，BFS 仍能保证在有限分支因子的状态空间中找到解。
5. **错。**此问题中的代价是移动次数，移动一次最多可以消除一个方向上的任意差距，而曼哈顿距离刻画的是两跟方格的距离大小，二者无直接联系，甚至曼哈顿距离往往会过高估计代价。例如，在以 A、B 为对角点的矩形边上无棋子阻挡时，曼哈顿距离一定大于等于 2，而实际代价始终为 2。综上，曼哈顿距离在此问题中是不可采纳的。

Exercise 18

1. 状态空间如下：



2. 各算法访问结点顺序如下：

BFS: 1-2-3-4-5-6-7-8-9-10-11

DLS(limit 3): 1-2-4-8-9-5-10-11

IDS: 1; 1-2-3; 1-2-4-5-3-6-7; 1-2-4-8-9-5-10-11

3. 双向搜索在此问题中效率较高，因为状态空间为严格二叉树，结构简单，反向路径唯一，可以避免冗余搜索。

   正向分支因子为 2，反向分支因子为 1。

4. 是。可以从目标结点开始，由于状态空间为严格二叉树，只能进行单向回溯，所以可以无需搜索，直接沿唯一路径到达结点 1。

5. 将状态进行二进制编码，结点 1 编码为 1，通过向编码右侧添加数来代替分支选择操作。规定添加 0 代表选择向左（选择 2k 分支），添加 1 代表向右（选择 2k+1 分支），如结点 2 的编码为 10，结点 5 的编码为 101。

   那么对任意目标结点 n，其二进制编码的数值即等于 n 的二进制表示，也就可以直接通过二进制转换得到从 1 到达目标结点的路径

Exercise 25

1. **对。**当 UCS 中所有边的代价相等时，到每个结点的成本即等于该结点的深度，此时 UCS 的优先级队列会像 BFS 按层扩展的行为一致。

2. **对。**Best-first tree 搜索按照评估函数 f(n)选择代价最小的结点进行扩展，若令 **f(n)=-depth(n)**，那么 Best-first tree 搜索会优先扩展深度最大的结点，此时即与 DFS 的行为一致。

3. **对。**A*搜索的评估函数为 **f(n)=g(n)+h(n)**。而当 **h(n)=0 时**，**f(n)=g(n)**，此时 A 搜

索完全按照路径成本扩展节点，与 UCS 的行为一致。

Exercise 26

8-puzzle (with Manhattan distance)		
算法名称	**优势**	**缺点**
A*算法	•使用可采纳且一致的启发式保证了最优解。 •优先队列系统性地扩展最低成本路径，扩展节点数较少，效率高。	内存占用高（需存储所有已生成节点的优先队列）
RBFS	内存占用少（仅存储当前路径及兄弟节点的回溯信息）。	•节点扩展数通常多于 A*（因回溯导致重复扩展）。 •递归调用带来的额外开销不可忽略。

TSP (with MST—see)		
算法名称	**优势**	**缺点**
A*算法	使用 MST 启发式可高效估计剩余路径成本，引导搜索方向，从而系统性地找到最优解，减少扩展节点数。	内存占用高（需存储所有已生成节点的优先队列）
RBFS	•内存占用少 •可通过递归剪枝避免存储全部节点。	•启发式计算频繁（每次回溯需重新评估兄弟节点），会增加计算时间。 •对启发式误差敏感，可能导致更多回溯。

综合比较之下，在小规模问题中 A*算法的效率显著优于 RBFS，但在大规模问题或内存受限的情况下，RBFS 可能因为内存占用小的优势而更可行。

在 8-puzzle 启发式中添加小随机数对 RBFS 的影响：

若该随机扰动破坏了启发式的可采纳性，可能导致 RBFS 扩展无关节点，造成扩展的结点数增加，甚至可能导致 RBFS 无法找到最优解而找到次优解。

Exercise 36

h(n) = Manhattan(n)+misplace(n)，其中 misplace(n)表示状态 n 下错位的数字个数，显然这个启发式 h 有时会高估代价。假设 h(n) ≤ C*(n)+c，C*表示实际代价。

证明：设 G2 为一个次优解，其实际代价比最优解 G1 的实际代价多 c，即 C*(G2)>C*(G1)+c。现考虑最佳路径上的任意结点 n，有

f(n)= g(n)+ h(n) ≤ g(n)+ h*(n)+c ≤ C*(G1)+c < C*(G2)

故在 G1 总是先于 G2 被扩展，那么任何实际代价比 C*(G1)多 c 的次优解都不可能被扩展，结论得证。

Exercise 38

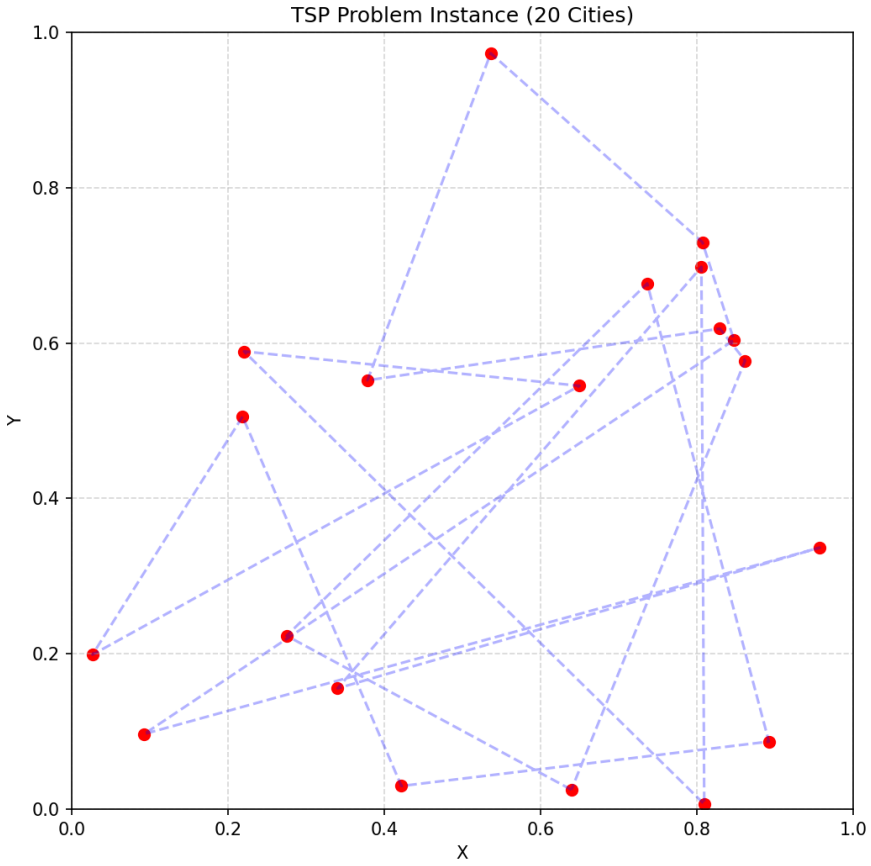
1. TSP 问题是找到一条穿过城市的最小路径，形成一个闭环。MST 在此问题中会生成一个连接所有城市的无环图（即树），它的总长度总是小于或等于闭环。故 MST 是可接受的。
2. MST 给出的从当前结点到目标结点的距离必然是**一条直线距离**或**多条直线距离的和**，在城市之间的距离满足**三角不等式**的情况下，MST 给出的距离必然大于两座城市的直线距离。

3. TSP 问题实例生成器

源码截图:

```python
# TSP问题实例生成器
def generate_cities(n: int, seed: int = None) -> List[Tuple[float, float]]:
    if seed is not None:
        random.seed(seed)
    return [(random.random(), random.random()) for _ in range(n)]
```

可视化运行结果:



4. **MST 的构造使用了 prim 算法，源码如图：**

```python
# Prim算法实现MST
def prim_mst(dist: List[List[float]], nodes: Set[int]) -> float:
    if len(nodes) < 1:
        return 0.0

    visited = set()
    heap = []
    start_node = next(iter(nodes))
    visited.add(start_node)
    mst_cost = 0.0

    # 初始化堆
    for node in nodes - visited:
        heapq.heappush(heap, (dist[start_node][node], node))

    while heap and len(visited) < len(nodes):
        cost, node = heapq.heappop(heap)
        if node not in visited:
            visited.add(node)
            mst_cost += cost
            for neighbor in nodes - visited:
                heapq.heappush(heap, (dist[node][neighbor], neighbor))

    return mst_cost
```

A*搜索的启发式函数及求解器：

```python
def mst_heuristic(self, remaining: Set[int], current_pos: int) -> float:
    if not remaining:
        return self.dist[current_pos][self.start]

    # 计算剩余节点的MST
    mst = prim_mst(self.dist, remaining)

    # 找到连接到当前节点和起点的最小边
    min_to_current = min(self.dist[current_pos][node] for node in remaining) if remaining else 0
    min_to_start = min(self.dist[self.start][node] for node in remaining) if remaining else 0

    return mst + min_to_current + min_to_start
```

```python
def solve(self) -> Tuple[List[int], float]:
    initial_remaining = set(range(self.n)) - {self.start}
    initial_node = Node([self.start], 0.0, initial_remaining)
    initial_node.heuristic = self.mst_heuristic(initial_remaining, self.start)

    heap = [initial_node]
    visited = {}
    best_cost = float('inf')
    best_path = []

    while heap:
        node = heapq.heappop(heap)

        if node.cost >= best_cost:
            continue

        # 终止条件：返回起点
        if not node.remaining and len(node.path) == self.n:
            final_cost = node.cost + self.dist[node.path[-1]][self.start]
            if final_cost < best_cost:
                best_cost = final_cost
                best_path = node.path + [self.start]
            continue

        # 扩展子节点
        current = node.path[-1]
        for next_node in node.remaining:
            new_cost = node.cost + self.dist[current][next_node]
            new_remaining = node.remaining - {next_node}
            new_path = node.path + [next_node]

            # 计算启发值
            heuristic = self.mst_heuristic(new_remaining, next_node)

            # 剪枝
            if new_cost + heuristic >= best_cost:
                continue

            new_node = Node(new_path, new_cost, new_remaining)
            new_node.heuristic = heuristic

            # 检查是否已访问过更优路径
            if new_node.key in visited:
                if visited[new_node.key] <= new_node.cost:
                    continue
            visited[new_node.key] = new_node.cost

            heapq.heappush(heap, new_node)

    return best_path, best_cost
```

运行结果如下：

```
最优路径: [0, 9, 3, 12, 19, 14, 18, 10, 15, 2, 7, 16, 17, 8, 5, 6, 13, 1, 11, 4, 0]
总成本: 3.5232
计算时间: 4.20秒
```