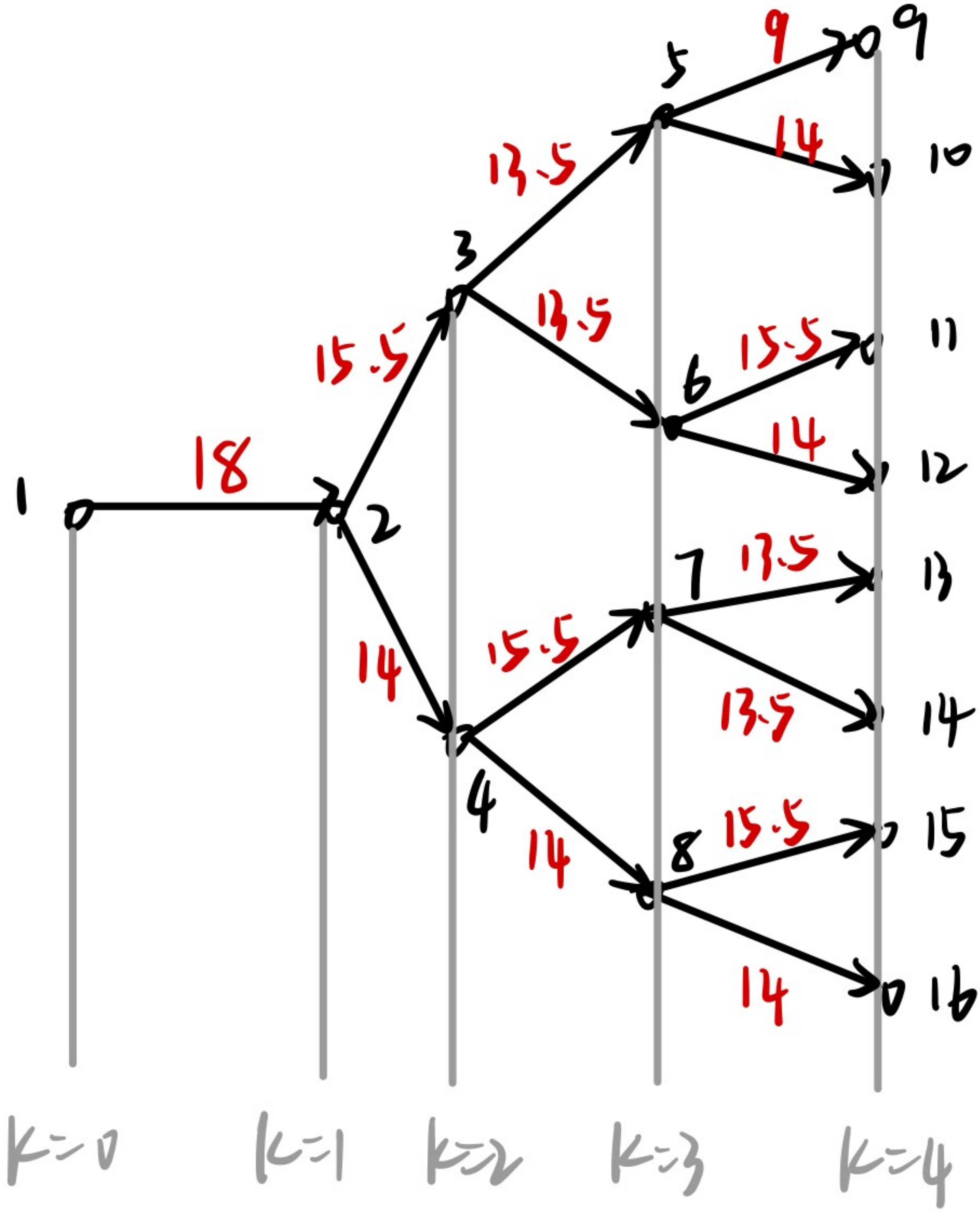
# 设备更新问题的最短路解法

## 问题建模

之前我们使用动态规划的方法解决这一设备更新问题，而现在我们考虑使用最短路解法解决这一问题。首先需要解决的问题是如何将原问题通过图的形式进行表达。通过分析原问题我们可以知道，其通过决策可能达到的状态可以用$(k,t)$来表征，其中$k$为时间节点，$t$为此时设备的役龄。在每个状态都有可以选择两种策略中的一种，保留($Keep$)或者更换（$Change$）设备，以进行状态的转移。

所以实际上我们可以根据时间顺序，建立一个**有向无环图**（实际上是一个完全二叉树），图中的点为可能达到的状态（起点为购入汽车的起始节点，终点为最后一年），边权为某一步决策所带来的收益。我们实际上希望的是找到到终状态$(4,t)$的最短路径（因为树的特性，所以当终点确定的时候到起点的路径是完全确定的），根据题意建图如下：




其中$1-16$为节点编号，红色为边权值，是由之前的分析得到的。

| t | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| r(t)-u(t) | 18 | 15.5 | 13.5 | 9 |
| s(t)-2 | 15 | 14 | 13.5 | 13. |

由于我们希望收益最大，而最短路的目的是使得边权之和最小，所以我们对于所有**边权取负值**。得到了对应的图之后我们使用$Dijkstra$算法解决该单源最短路问题。

## 代码

```
import numpy as np
from queue import PriorityQueue

class Graph:
    def __init__(self, n):
        self.v = n
        self.edges = [[-1 for i in range(n)] for j in range(n)]
        self.visited = []

    # add edge to a unidirected graph
    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight

def dijkstra(graph, s):
    distance = [np.inf for i in range(graph.v)]
    distance[s] = 0
    to_visit = PriorityQueue()
    to_visit.put((0, s))

    while not to_visit.empty():
        (dist, current_v) = to_visit.get()
        graph.visited.append(current_v)
        for neighbor in range(graph.v):
            if graph.edges[current_v][neighbor] != -1 and neighbor not in graph.visited:
                d = graph.edges[current_v][neighbor]
                old_dis = distance[neighbor]
                new_dis = distance[current_v] + d
                if new_dis < old_dis:
                    to_visit.put((new_dis, neighbor))
                    distance[neighbor] = new_dis
    return distance

if __name__ == '__main__':

    # initialize graph
    start_v = 1
    n = 16
    G = Graph(n+1)
    G.add_edge(1, 2, -18)
    G.add_edge(2, 3, -15.5)
    G.add_edge(2, 4, -14)
    G.add_edge(3, 5, -13.5)
    G.add_edge(3, 6, -13.5)
    G.add_edge(4, 7, -15.5)
    G.add_edge(4, 8, -14)
    G.add_edge(5, 9, -9)
    G.add_edge(5, 10, -14)
    G.add_edge(6, 11, -15.5)
    G.add_edge(6, 12, -14)
    G.add_edge(7, 13, -13.5)
```

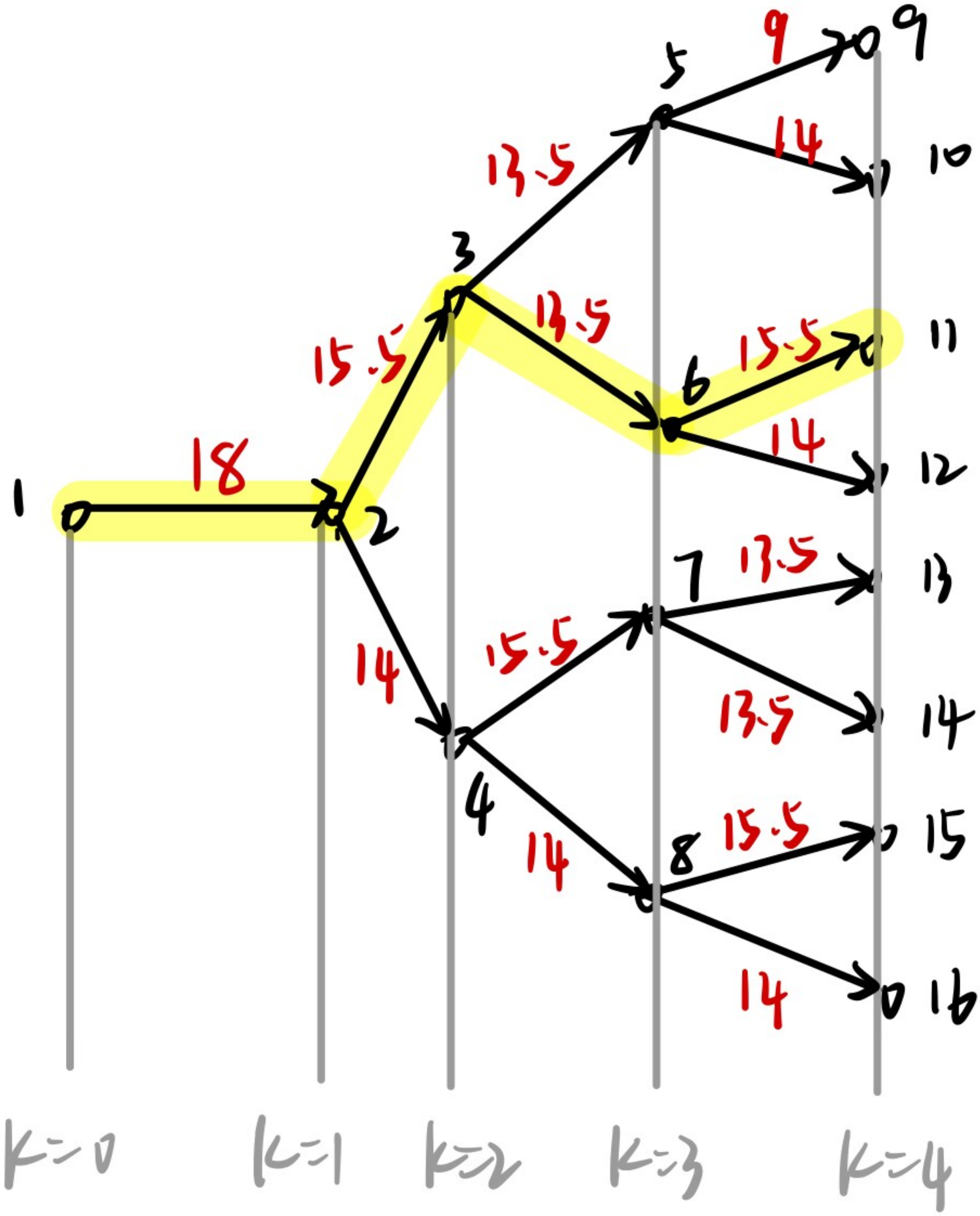
```python
    G.add_edge(7, 14, -13.5)
    G.add_edge(8, 15, -15.5)
    G.add_edge(8, 16, -14)

    # get distance to all other vertices
    distance = dijkstra(G, start_v)[1:]
    print(distance)
    print("min distance ", min(distance))
    print("dist vertice of the path with the minimum distance: ",
          distance.index(min(distance))+1)
```

## 结果

```
[0, -18, -33.5, -32, -47.0, -47.0, -47.5, -46, -56.0, -61.0, -62.5, -61.0, -61.0, -61.0, -61.5, -60]
min distance  -62.5
dist vertice of the path with the minimum distance:  11
```

可以看到结果中，最大收益为$62.5$（万元），对应的是到达节点$11$，返回到图中也就得到路径为：$1->2->3->6->11$




对应的策略为 `保留->保留->更新->保留` 。与我们之前通过动态规划方法手算的结果相同。

## 思考

本问题的动态规划与最短路解法，二者实际上存在某种程度的相似性，都是从某一个确定的最优状态转移到下一个最优状态，将规模比较大的问题转化成为其子问题。实际上最短路解决方案是相对固定的，主要的难点是如何将原问题构造成一个图。