# Dijkstra算法

李朋飞

2019-12-04

# 维基百科

- Dijkstra's algorithm to find the shortest path between a and b. It picks the unvisited vertex with the low distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

# 算法思想

- 估计的最短路径值渐渐地被更加准确的值替代，直至得到最优解。

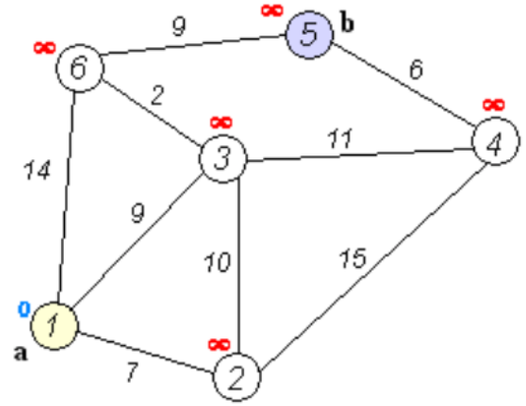
- 以贪心法选取未被处理的具有最小权值的节点，然后对其的出边进行松弛操作。

# 算法的4个步骤

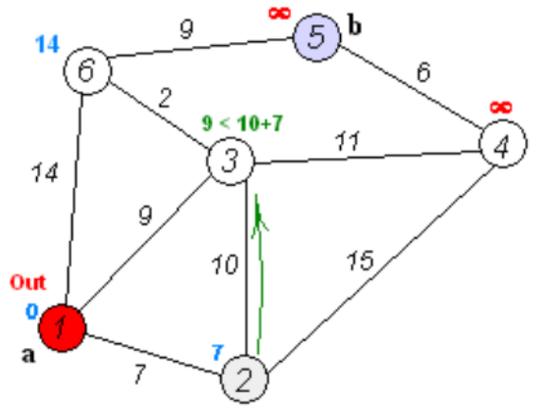
- 步骤一：找出距离源点最近的未被访问节点
- 步骤二：计算与其相邻的未被访问过节点的距离，比较是否更短，如果是，则更新相邻节点的距离
- 步骤三：重复步骤一和步骤二，直到所有节点都被访问过一遍
- 步骤四：得到源点到目标点的最短路径

# 预先设定

- 源点的距离为0
- 其他节点的距离为∞（无穷大）



# 松弛




- 定义未被访问过的节点的集合为U

- 先计算与源点1相邻的点的距离，与其相邻的点有2、3、6；点2当前的距离为∞，从点1到点2的距离为7(0+7)，7<∞，更新点2的距离为7，并把点2加入到集合U中 ；按同样的方式处理点3的距离更新为9，点6的距离更新为14，点3和点6也加入到集合U中；

- 计算完与源点相邻点的距离后，从集合U中选出距离源点最近的点，是点2

- 再计算与点2相邻的未被访问到的点的距离，即点3和点4的距离；从点2到点3的距离为17(7+10)，大于点3当前的距离9，不更新，把点3加入到未被访问到的节点集合中（点3已在集合U中，要避免重复）；从点2到点4的距离为22（7+15），小于∞，更新点4的距离为22，把点4加入到未被访问到的节点集合中

- 这个过程就叫做“松弛”

- 接下来再从未被访问过的点3、4、5中选出距离源点最近的点3，重复上述过程，直到结束

# 节点路径信息

### 初始化

| 节点 | 距离 |
| --- | --- |
| 点1 | 0 |
| 点2 | ∞ |
| 点3 | ∞ |
| 点4 | ∞ |
| 点5 | ∞ |
| 点6 | ∞ |

### 计算源点相邻点距离

| 父节点 | 节点 | 距离 |
| --- | --- | --- |
| 点1 | 点2 | 7 |
| 点1 | 点3 | 9 |
| 点1 | 点6 | 14 |
| | 点4 | ∞ |
| | 点5 | ∞ |

### 计算点2相邻点距离

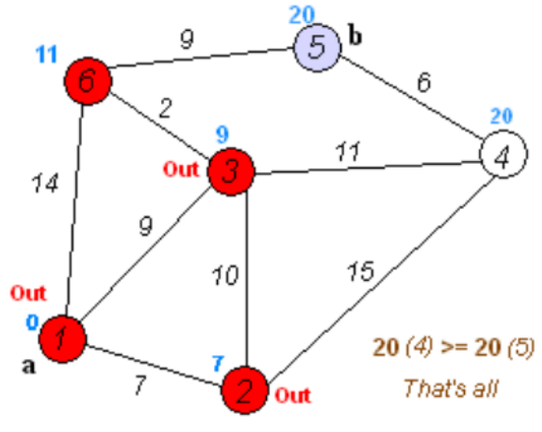
| 父节点 | 节点 | 距离 |
| --- | --- | --- |
| 点1 | 点2 | 7 |
| 点1 | 点3 | 9 |
| 点1 | 点6 | 14 |
| 点2 | 点4 | 22 |
| | 点5 | ∞ |

### 计算点3相邻点距离

| 父节点 | 节点 | 距离 |
| --- | --- | --- |
| 点1 | 点2 | 7 |
| 点1 | 点3 | 9 |
| 点3 | 点6 | 11 |
| 点3 | 点4 | 20 |
| | 点5 | ∞ |

### 计算点6相邻点距离

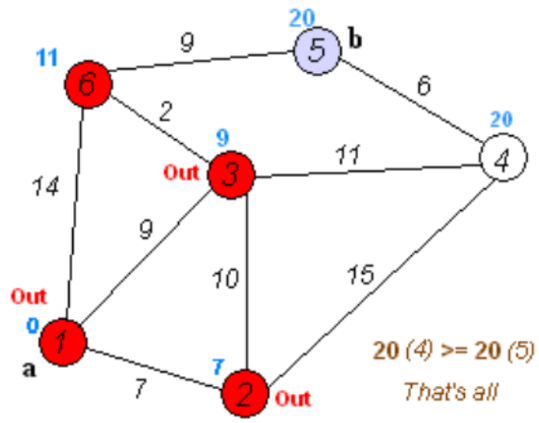
| 父节点 | 节点 | 距离 |
| --- | --- | --- |
| 点1 | 点2 | 7 |
| 点1 | 点3 | 9 |
| 点3 | 点6 | 11 |
| 点3 | 点4 | 20 |
| 点6 | 点5 | 20 |



# 构造路径

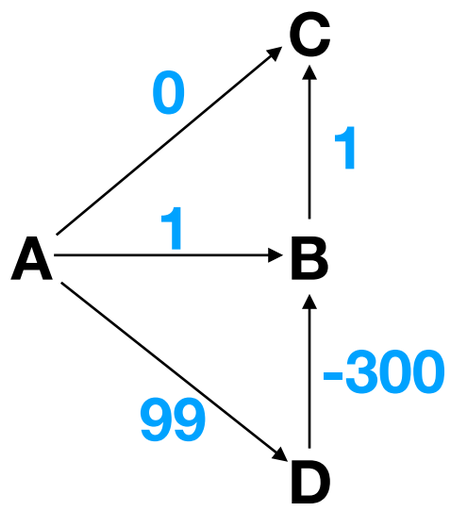
- 计算完路径后从目标节点开始根据其父节点，不断的往前寻找，直到找到源点未知，即为源点到目标节点的最短路径

- 从a到b的路径为：a->3->6->b



# 算法要注意的地方

- 权重不能是负的
- 可能存在多条最长/最短路径
- 避免无限循环检测(Infinite loop detected)
- 可能找不到路径（No path found）



# 应用场景

- 网络路由协议：IS-IS、OSPF

- 地图：路线

# 相关算法

- Dijkstra最短路算法（单源最短路）
- Bellman–Ford算法（解决负权边问题）
- SPFA算法（Bellman-Ford算法改进版本）
- Floyd最短路算法（全局/多源最短路）

# 实现

- Go语言：https://github.com/RyanCarrier/dijkstra（号称Go语言中最快）

# RyanCarrier/dijkstra代码之Vertex结构体

```go
//Vertex is a single node in the network, contains it's ID, best distance (to
// itself from the src) and the weight to go to each other connected node (Vertex)
type Vertex struct {
    //ID of the Vertex
    ID int // 点的ID
    //Best distance to the Vertex
    distance      int64 // 起始点src到本点的距离（一开始被初始化为无穷大）
    bestVerticies []int // 父点
    //A set of all weights to the nodes in the map
    arcs map[int]int64 // 本点到其他点对应的权重
}
```

# RyanCarrier/dijkstra代码之Graph结构体

```
//Graph contains all the graph details
type Graph struct {
    best         int64 // 最优距离
    visitedDest bool  // 是否访问的目标点
    //slice of all verticies available
    Verticies        []Vertex        // 所有点集合
    visiting         dijkstraList    // 未访问到的点集合
    mapping          map[string]int // 当点不是用整数表示时，使用mapping映射点对应的整数id
    usingMap         bool            // 是否使用了map
    highestMapIndex int             // map中最大的id
}
```

# RyanCarrier/dijkstra代码之预设置

```
func (g *Graph) setup(shortest bool, src int, list int) {
    //-1 auto list
    //Get a new list regardless
    if list >= 0 {
        g.forceList(list)
    } else if shortest {
        g.forceList(-1)
    } else {
        g.forceList(-2)
    }
    //Reset state
    g.visitedDest = false
    //Reset the best current value (worst so it gets overwritten)
    // and set the defaults *almost* as bad
    // set all best verticies to -1 (unused)
    if shortest {
        g.setDefaults(int64(math.MaxInt64)-2, -1) // 把每个点的distance设置为无穷大
        g.best = int64(math.MaxInt64)
    } else {
        g.setDefaults(int64(math.MinInt64)+2, -1) // 把每个点的distance设置为无穷小
        g.best = int64(math.MinInt64)
    }
    //Set the distance of initial vertex 0
    g.Verticies[src].distance = 0 // 把原始点distance设置为0
    //Add the source vertex to the list
    g.visiting.PushOrdered(&g.Verticies[src]) // 把原始点添加到visiting中
}
```

# RyanCarrier/dijkstra代码之选路过程

```go
func (g *Graph) postSetupEvaluate(src, dest int, shortest bool) (BestPath, error) {
    var current *Vertex
    oldCurrent := -1
    for g.visiting.Len() > 0 {
        //Visit the current lowest distanced Vertex
        //TODO WTF
        current = g.visiting.PopOrdered() // 从visiting中取出一个点
        if oldCurrent == current.ID {
            continue
        }
        oldCurrent = current.ID
        //If the current distance is already worse than the best try another Vertex
        if shortest && current.distance >= g.best { // 如果点前点的“距离”大于最优距离，则没必要在进行
            continue
        }
        for v, dist := range current.arcs {
            //If the arc has better access, than the current best, update the Vertex being touched
            // 求最短路径时，如果当前点的累积距离加到相邻点的距离小于相邻点的距离，则继续
            if (shortest && current.distance+dist < g.Verticies[v].distance) ||
                // 求最长路径时，如果当前点的累积距离加到相邻点的距离大于相邻点的距离，则继续
                (!shortest && current.distance+dist > g.Verticies[v].distance) {
                // 如果当前点的父节点是相邻点，且相邻点不是目标点，则出现了循环，返回错误
                if current.bestVerticies[0] == v && g.Verticies[v].ID != dest {
                    //also only do this if we aren't checkout out the best distance again
                    //This seems familiar 8^)
                    return BestPath{}, newErrLoop(current.ID, v)
                }
                g.Verticies[v].distance = current.distance + dist // 更新相邻点的累积距离
                g.Verticies[v].bestVerticies[0] = current.ID      // 设置相邻的父节点
                if v == dest {                                    // 如果相邻点是目标点，则记录下来
                    //If this is the destination update best, so we can stop looking at
                    // useless Verticies
                    g.best = current.distance + dist
                    g.visitedDest = true
                    continue // Do not push if dest
                }
                //Push this updated Vertex into the list to be evaluated, pushes in
                // sorted form
                g.visiting.PushOrdered(&g.Verticies[v])
            }
        }
    }
    return g.finally(src, dest)
}
```

# RyanCarrier/dijkstra代码之构造路径

```go
func (g *Graph) bestPath(src, dest int) BestPath {
    var path []int
    for c := g.Verticies[dest]; c.ID != src; c = g.Verticies[c.bestVerticies[0]] {
        path = append(path, c.ID)
    }
    path = append(path, src)
    for i, j := 0, len(path)-1; i < j; i, j = i+1, j-1 {
        path[i], path[j] = path[j], path[i]
    }
    return BestPath{g.Verticies[dest].distance, path}
}
```