Dijkstra's Algorithm

*** *** ***

2023 年 4 月 4 日

0.1 Introduction

最短路径问题应用非常广泛，而 Dijkstra 算法是解决单元最短路径的经典算法之一，本次项目的主要目的是用不同的堆结构实现该算法，对比不同堆结构对该算法的运行效率的影响，以寻找实现 Dijkstra 算法最合适的数据结构。

0.2 Data Structure / Algorithm Specification

我们使用斐波那契堆和二项队列实现 dijkstra 算法，从起始点开始，每次遍历到始点距离最近且未访问过的顶点的邻接节点，直到扩展到终点为止，并通过斐波那契堆和左偏堆优化该算法。

斐波那契堆通过结合左堆的 decreasekey 和懒惰合并两个操作，来降低 insert、merge 及 decreasekey 操作花费的时间，使得其各自摊还时间为 $\Theta(1)$。

左堆 decreasekey 的基本思路是将值被降低的节点作为堆取出，与原来的堆 merge，虽然左倾的性质可能被破坏，但只需要交换子节点来维护。

懒惰合并，对于一个二项队列而言，即只把二项队列的树连接在一起，当然这可能会破坏二项队列不能有相同的度的树的性质。当然无限制的懒惰合并显然不合理，所以每次 DeleteMin 会重新合并队列中的树，使其重新满足二项队列的性质。

当然斐波那契堆结合这两个操作，需要额外的代价，斐波那契堆在节点中额外维护了一个 mark 变量，将第一次失去一个子节点的非根节点标记，当标记节点再次失去子节点时，将节点从父节点切除并移去标记，称作级联切除。通过级联切除，可以使得树的度控制在 $O(log(n))$，从而使得懒惰合并的 DeleteMin 摊还时间控制在 $O(log(n))$。

对于二项队列实现该算法，我们发现 DecreaseKey 需要上滤的堆交换节点为了避免复杂的 Link 常常通过交换节点的值实现，这就导致外部指向节点的指针在 decreaseKey 后会失效，因此 Decreasekey 需要上滤的堆如果使用模板类将无法应用于 Dijkstra 算法。

证明如下，如果我们在图中引用模板类的堆，因为 DecreaseKey 需要堆节点的指针，我们必须在图中或者堆中存储一个可以通过节点下标寻找的指针。如果我们存在图中，由于 DecreaseKey 在堆中声明，不能获得图中存储的指针，因此将不能更新图中存储的指针出现错误；如果我们存在堆中，因

为堆是根据 comparable X 进行排序，在 Dijkstra 算法中是按路径排序，我们将无法获得结点的下标（如果是 pair<int,int>，则无法获得 second，我们无法保证所有的 comparable 都有 second）。

如果我们采用复杂的 Link 可以解决这个问题，复杂度为 $log(2N)$，但是我们已经没有时间实现了。

0.3 Testing Results

以下是测试程序的输出，测试算法的正确性。测试程序包括向图插入前 8 行的边，以及输出最后由堆优化后的 dijkstra 算法找单源最短路径的结果。(pass)

```
(1,5,4)
(1,2,1)
(2,4,1)
(3,2,1)
(4,3,3)
(5,7,3)
(5,6,5)
(6,7,4)
Vertex Distance from Source
Point 0 : 0
Point 1 : 1
Point 2 : 2
Point 3 : 6
Point 4 : 3
Point 5 : 5
Point 6 : 5
Point 7 : 8
```

以下是对 9th DIMACS Implementation Challenge Core Problem Families 中的不同类型不同大小的图，运行通过斐波那契堆优化 dijstra 算法寻找单源最短路径的平均运行时间表，以及对应的运行时间与节点数的关系图，图像大都满足边数为节点数的 4 倍。(pass)

节点数	边数	平均运行时间
1024	3936	0.0007
2025	7920	0.0016
2048	7904	0.0017
4096	15840	0.0031
8192	31712	0.070
32768	126944	0.0354
65536	253920	0.0848
131072	507872	0.2102
262144	1046528	0.4351
524176	2093808	1.0852
1048576	4063200	1.8155
2096704	8381024	5.1161

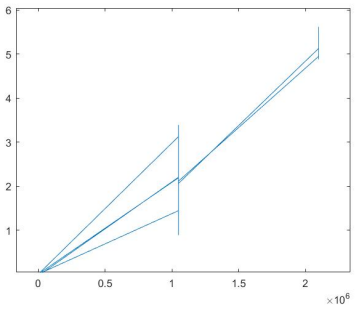

图 1: 原有数据图像

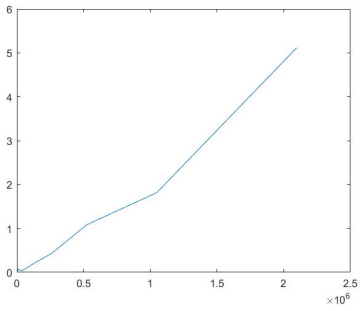

图 2: 整理数据图像

从图像可看出，算法运行时间与节点数的 6 次方基本成线性关系，又有边数大致为节点数的 4 倍，所以符合 dijkstra 算法在斐波那契堆的改进后算法时间界 $O(|V|log|V|+|E|)$。

0.4 Analysis and Comments

	斐波那契堆	二项队列
操作	摊还时间界	最坏时间界
Initial	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$O(1)$
Getmin	$\Theta(1)$	$\Theta(1)$
DeleteMin	$O(log(n))$	$\Theta(log(n))$
Merge	$\Theta(1)$	$O(log(n))$
DecreaseKey	$\Theta(1)$	$\Theta(log(n))$

斐波那契堆分析：对于堆中任意一个节点 X，考虑其第 i 个子节点 C_i，显然当 C_i 与 X 合并的时候，C_i 与 X 有相同的子节点数，又因为 C_i 作为 X 的子节点最多失去一个子节点，所以 C_i 的子节点数至少为 i-2。那么由此可得，对于一棵度为 R 的树 Tr，其拥有度分别至少为 $R-2, R-1, \ldots, 1, 0$ 的 R 棵子树。$|T_R| = |T_0| + \sum_i (R-2)|T_i|$ 容易证明 $|T_R| = F_R + 1$，其中 F_R 是斐波那契数列的第 R 项。又知道斐波那契数以指数增长，因此任意节点的度为 $O(log(n))$。

摊还分析，取势能函数为 Φ= num of tree+2*num of marked node

Merge: $c = O(1)$, $\Delta\Phi = 0$, $c' = O(1)$ 势能不变

Insert: $c = O(1)$, $\Delta\Phi = 1$, $c' = O(1)$ 树的数量增加 1

Deletemin: $c = T + R + log(n) = T + O(log(n))$, $\Delta\Phi = O(log(n)) - T$, $c' = O(log(n))$ R 是包含最小节点树的度，T 是操作前树的数量，树的数量改变最多为 $O(log(n)) - T$

DecreaseKey $c = C + 1$, $\Delta\Phi = 3 - C$, $c' = O(1)$ ，C 为级联切除次数，每次切除标记减一树加一，最后一次切除可能不是标记节点，因此势能变化最多为 $3 - C$

二项队列摊还分析：取势能函数为 Φ = num of tree

Merge: $c = O(logN_1 + logN_2) = O(logN)$, $\Delta\Phi = O(logN)$, $c' = O(logN)$ $N = N_1 + N_2$ 表示节点数量，合并后最多增加 $O(logN)$ 棵树

Insert: $c = C$, $\Delta\Phi = 2 - C$, $c' = O(1)$ 树的数量增加 2-C

DeleteMin: $c = 1 + O(logN1 + logN2) = O(logN)$, $= O(logN1) - 1 = O(logN)$, $c = O(log(N))$，DeleteMin 删除最小值后 merge，而删除最小值只需要常数时间

DecreaseKey: $c = H = O(log(N))$, $\Delta\Phi = 0$, $c' = O(log(N))$,H 为节点

高度，不超过 $log(N)$

在 dijkstra 算法中，运行时间由 $|E|$ 次 decreaseKey 和 $|V|$ 次 insert 及 DeleteMin 决定，对于二项队列，其时间界为 $O((|V|+|E|)log|V|)$，但斐波那契堆的改进使得算法最终时间界可以达到 $O(|V|log|V|+|E|)$。

0.5 References

[1] Thomas H. Cormen,Charles E. Leiserson,Ronald L. Rivest,Clifford Stein, “Introduction to Algorithms”, MIT Press, 2009

[2] Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”

0.6 Declaration

We hereby declare that all the work done in this project titled ”Dijkstra’s Algorithm” is of our independent effort as a group.