Journal of System Simulation

Volume 29 | Issue 1 Article 13

6-1-2020

Efficient Priority Queue Constructed on Heap-like Data Structure

Binbin Qi
Department of Educational Technology, Nanjing Normal University, Nanjing 210097, China;

Mingyong Pang
Department of Educational Technology, Nanjing Normal University, Nanjing 210097, China;

Efficient Priority Queue Constructed on Heap-like Data Structure

Abstract

Abstract: How to organize data structure is a very important issue for various algorithm implementations. In this paper, *a novel and efficient priority-queue was presented based on a kind of heap-like data structure. The organization structure of the queue is first introduced and then a set of basic operators on the queue were elucidated, the time and space performances of the data structure was also theoretically analyzed. Our data structure has several advantages, including the adaptability of storage space as well as the simplicity and convenience of implementation. Experimental results showed that the heuristic priority queue data structure can effectively improve the performance of various types of simulation systems, relative to its traditional counterparts.* Our data structure can be used to solve a variety of application problems such as combinatorial optimization.

Keywords

data structure, priority queue, heap-like queue, efficiency of algorithm

Recommended Citation

Qi Binbin, *Pang Mingyong*. Efficient Priority Queue Constructed on Heap-like Data Structure[J]. Journal of System Simulation, 2017, 29(1): 91-98.

第 29 卷第 1 期 系统仿真学报© Vol. 29 No. 1
2017 年 1 月 Journal of System Simulation Jan., 2017

一种高效的动态优先队列数据结构

祁彬斌，庞明勇
(南京师范大学教育技术系，江苏 南京 210097)

摘要：数据结构的组织形式在算法实现中占有重要地位。*探讨了优先队列中的数据结构组织问题，给出一种高效的堆式队列数据组织结构，阐明了该数据结构的基本操作,理论上分析了其时间和空间性能，通过实验和应用实例验证了堆式队列数据结构动态存取效率的高效性、存储空间的自适应性以及实现上的简单性。*实验结果表明，堆式优先队列数据结构可有效地提高各类仿真系统的性能，也可用于组合优化等多种应用问题的解决。

关键词：数据结构；优先队列；堆式队列；算法效率
中图分类号：TP391 文献标识码：A 文章编号：1004-731X (2017) 01-0091-08
DOI: 10.16182/j.issn1004731x.joss.201701013

Efficient Priority Queue Constructed on Heap-like Data Structure

Qi Binbin, *Pang Mingyong*
(Department of Educational Technology, Nanjing Normal University, Nanjing 210097, China)

Abstract: How to organize data structure is a very important issue for various algorithm implementations. In this paper, *a novel and efficient priority-queue was presented based on a kind of heap-like data structure. The organization structure of the queue is first introduced and then a set of basic operators on the queue were elucidated, the time and space performances of the data structure was also theoretically analyzed. Our data structure has several advantages, including the adaptability of storage space as well as the simplicity and convenience of implementation. Experimental results showed that the heuristic priority queue data structure can effectively improve the performance of various types of simulation systems, relative to its traditional counterparts.* Our data structure can be used to solve a variety of application problems such as combinatorial optimization.

Keywords: data structure; priority queue; heap-like queue; efficiency of algorithm

引言

在系统仿真领域中，有大量的应用涉及到优先队列结构的数据存取。快捷、高效、便利的优先队列组织方式对相关系统的性能提高发挥着重要作用，如文献[1]结合一种优先队列调度算法，给出了网络仿真软件 NS2(Network Simulator 2)队列调度的扩展方法，该方法在兼顾网络带宽分配均衡性的基础上，有效地保证了系统的效率；文献[2]在事件驱动的大规模碰撞检测仿真中，考虑到算法中存在的大量事件调度情况，根据算法的特点引入了相应的优先队列数据结构，改善了算法的时空效率；文献[3]提出一种基于 Kd-Tree 和优先队列的图像特征匹配算法，该算法利用优先队列搜索给定点的近似最近邻近点，有效地提高了算法的搜索效率，实现了特征的实时匹配；文献[4]给出了图像分析领域中几种常见的优先队列组织结构，对不同结构在具体应用情形下的存取性能进行了评估；为

收稿日期：2015-04-24 修回日期：2015-07-07;
基金项目：国家自然科学基金(41271383, 60873175), 江苏省现代教育技术研究课题(2014-R-33356);
作者简介：祁彬斌(1991-)，男，江苏盐城，博士生，研究方向为数字几何处理；庞明勇(1968-)，男，安徽淮南，博士，教授，博导，研究方向为数字几何处理。

了实现仿真环境下的三维模型 LOD(Levels of Details)效果，文献[5]提出在模型简化过程中以边的折叠代价为优先权构建优先队列，每次从优先队列中选取折叠代价最小的边对模型进行简化操作，在保持高保真度简化效果的同时，确保了 QEM (Quadric Error Metric)算法的高效率；文献[6]提出一种基于“截止期最早最优先”(Earliest Deadline First)策略的动态调度方法，在相关优先队列结构的支持下，提高了分布式数据库 HBase 中文件系统的效率；文献[7]讨论了在采用优先队列的情况下，4 种主要类型的移动网络通信的数据转发性能，并在网络仿真软件 NS2 中对相关性能进行了分析；文献[8]提出一种模拟待泊船只排队进港的优先级调度算法，其中优先权由船舶的不同性质决定，研究结果表明良好的优先队列组织结构能够有效地优化了港口调度算法的运行时间效率，提高了调度算法的实时性。

目前，优先队列的数据组织方式多种多样，不同的数据组织形式对数据存取的性能影响明显。优化的组织结构能够有效发挥优先队列存取效率，有助于提升相关算法的性能，进而提高仿真系统的实时性；而不好的组织结构可能会使算法的时空效率大打折扣[9]。因此，如何有效地组织优先队列的数据结构就显得尤为重要。

本文给出一种高效的优先队列的数据组织结构，该数据结构以堆式数据存取方法为基础，通过引入数据的动态维护和管理机制，能够实现数据存取的高效率。与现有主流优先队列数据结构相比，不但拥有更低的时间复杂度，而且还能确保数据动态存取的高效性、存储空间的自适应性、以及实现上的简单性。

1 优先队列及其组织结构

1.1 优先队列

普通队列(Queue)是一种先进先出(FIFO，First-In，First-Out)的线性表，元素在队列尾部追加，从队列头处删除。而在优先队列(Priority Queue)中，队列中每个元素被赋予不同优先权，出队列的顺序由元素的优先权决定，最高优先权的元素最先被删除，具有最高进先出(Largest-In，First-Out)的行为特性。

传统优先队列通常选用线性表、二叉查找树、堆等多种数据组织方式来表示。其中，线性表作为一种常用的组织结构，按是否进行优先权排序又被分为无序表和有序表两大类。由于实际应用中很难预先估计优先队列的最大长度，故在具体实现上更多地采用链表存储方式。对于有序链表而言，元素进队需按照优先权进行预先排序，排序过程的时间复杂度为 $O(n)$。出队时则可直接移出权值最小的结点，时间复杂度为 $O(1)$。对于无序链表而言，具体情形与之相反，元素进队操作的时间复杂度为 $O(1)$，出队则需遍历队列中的元素，此时的时间复杂度为 $O(n)$。但总体而言，在传统的优先队列数据组织结构中，无论是采用无序链表还是有序链表，均未能有效地降低队列操作的时间复杂度、减少算法的执行时间，致使实际应用时易导致算法的执行效率不高等问题，具有一定的局限性。

1.2 结构组织

本文给出一种高效的堆式优先队列数据结构(以下简称为“堆式队列”)，其具有可动态更新、自动排序等特点，能够动态地调整已排序队列中的元素优先权，显著地改善数据存取的性能。

本文堆式队列的基础是一棵没有限制孩子结点数目的树：树中每一个结点的权值均不大于其孩子结点的权值，权值最小的结点为根结点，同一层次的兄弟之间没有固定的先后次序，如图 1 所示。

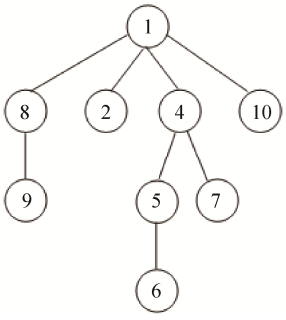


图 1 树表示的堆式队列

Fig.1 An heap-like queue represented by tree

DOI: 10.16182/j.issn1004731x.joss.201701013

树中每个结点的孩子数目可能不同，因此记录孩子信息的结点空间大小也可能不一样。为避免这一情况，本文基于“左孩子-右兄弟”给出了堆式队列的表示方法，如图 2 所示。该表示方法中，每个堆式队列的结点除了包含用于排序的元素外，还涉及前驱结点、左孩子结点、右兄弟结点的信息。若一个结点是其父结点的最左孩子结点，则它的父结点就是前驱结点；否则，其前驱结点为该结点的左兄弟。以图 2 为例，1 为 8 的前驱结点，8 是 2 的前驱结点；8 是 1 的左孩子结点，2 是 8 的右兄弟结点。

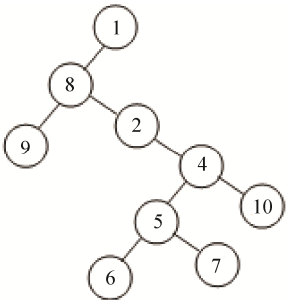


图 2 堆式队列的孩子兄弟表示
Fig.2 Child-sibling representation of heap-like queue in Fig.1

2 基本操作

本文堆式优先队列有“进队”、“出队”、“动态调整优先权”3 种基本操作，通过调用这些基本操作，可以保证优先队列中数据的快速存取和内部次序的自动调整。

2.1 进队操作

进队是为堆式队列添加一个新元素。构造堆式队列时，首先把新元素作为一个新的结点，若队列中尚无任何结点信息，则该结点就是堆式队列的根结点。如果堆式队列中已经存在结点，则比较当前结点和堆式队列根结点之间的大小。若当前结点的权小于根结点的权，则将根结点作为当前结点的左孩子结点(当前结点作为根结点的前驱结点)，此时的当前结点可自动调整为新的根结点；否则，堆式队列根结点保持不变，根结点的左孩子结点(若有)成为当前结点的右兄弟，当前结点替换为根结点的新的左孩子，具体过程参见图 3。

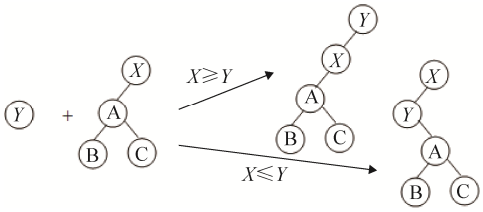
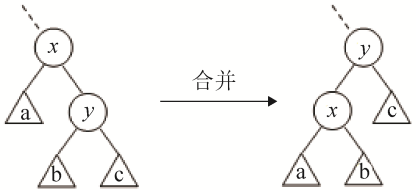


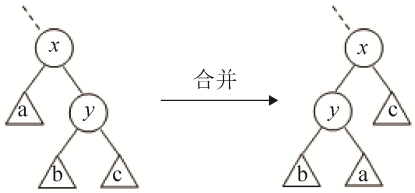
图 3 孩子兄弟表示的堆式队列的进队操作
Fig.3 Enqueue operation of heap-like queue

2.2 出队操作

出队是从堆式队列中移出权值最小的结点，即根结点。当删除并返回根结点后，需从原根结点的所有孩子结点中选取权值最小的成为新的根结点。最简单的做法是：根结点删除后，原堆式队列会变成若干棵子树，依次合并这些子树便能生成一个新的堆式队列。该合并是由若干个合并子操作组成。合并子操作主要是指比较两棵子树根结点权值的大小，将根结点权值较大的子树作为另一棵子树最左边的孩子，原先最左边的孩子变为其第 2 个孩子，第 2 个孩子则变更为第 3 个孩子，依次类推(见图 4)。



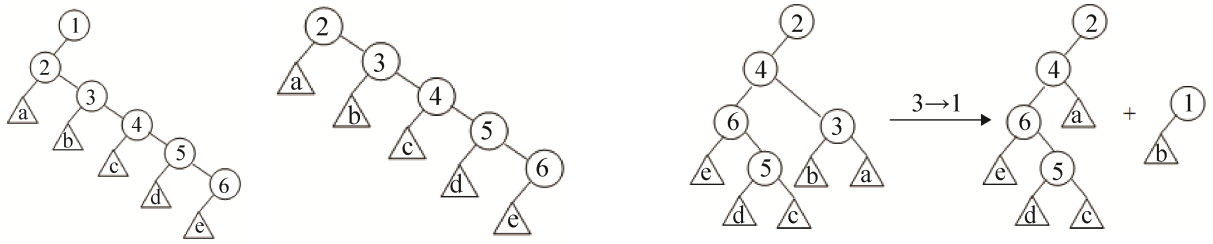
(a) 权值 $y < x$



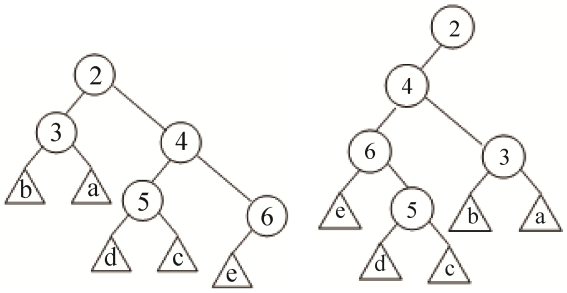
(b) 权值 $y \geq x$

图 4 出队环节中合并子操作
Fig.4 Sub-operation merging in dequeue operation

为了进一步优化堆式队列的结构、提高算法的效率，本文给出两种合并方法，分别称之为“两趟合并法”和“多趟合并法”。在“两趟合并法”中，第一趟是从左到右两两合并兄弟子树，若合并到最后还剩下一颗子树，则该子树保持不变；第二趟则是从右向左，每次合并最右与次右的兄弟子树，直到到达最左为止，如图 5 所示。



(a) 出队前 (b) 删除根结点



(c) 第一次合并 (d) 第二次合并

图 5 “两趟合并法”表示出队操作

Fig.5 Two-passes merging for dequeue operation

“多趟合并法”具体过程为：从左到右两两合并兄弟子树，再重复地合并新得到的子树，直到生成新的堆式队列。例如，根结点拥有 x^1 到 x^8 共 8 棵子树，第一次合并中，把 x^1 和 x^2 合并成 y^1，x^3 和 x^4 合并成 y^2，x^5 和 x^6 合并成 y^3，x^7 和 x^8 合并成 y^4；第二次合并中，从左到右将 y^1 和 y^2 合并成 z^1，y^3 和 y^4 合并成 z^2；重复多次后，所有的兄弟子树将合并为新的堆式队列。理论上，“多趟合并法”相比于“两趟合并法”形成的堆式队列深度更大、结构更优。但实验结果表明，“两趟合并法”合并的次数总体上更少，所以其算法的性能更优。

2.3 动态调整优先权操作

动态调整优先权操作主要包括两部分：降优先权和升优先权。降优先权是降低堆式队列中指定结点的权值。其基本过程是：若指定的结点为根结点，直接修改根结点的权值。若指定结点并非根结点，则将需要降优先权的结点连同其孩子结点从堆式队列中移出，形成一棵新的子树；然后将降优先权后的新子树与原先的堆式队列进行合并，具体过程如图 6 所示。

图 6 堆式队列表示降优先权操作

Fig.6 Key-decreasing operation of heap-like queue

升优先权则是指提升堆式队列中指定结点的权值。提升优先权后有两种情况：一种是指定结点的权值不大于其孩子结点的最小权值，另一种是调整后的结点权值大于其孩子结点的最小权值。就前者而言，尽管升优先权后指定结点的权值发生改变，但是堆式队列的性质并未受到破坏，即堆式队列中每一个结点的权值仍不大于其孩子结点的权值。因此，堆式队列的组织结构不需要进行调整。而对于后一种情况，由于调整后的结点权值大于它的某一个或多个孩子结点的权值，因此堆式队列的数据组织结构需要进行动态调整。其基本过程如下：若指定的结点为根结点，则首先移除并修改根结点；然后对原根结点的所有孩子结点进行动态调整，形成新的子树(可参考出队操作)；最后将新子树与升优先权后的根结点合并，形成新的堆式队列。若指定结点为非根结点，首先将需要升优先权的结点连同其孩子结点从堆式队列中移出；然后将移出的子树动态调整为新的子树(具体操作同指定结点为根结点的情况)；最后将升优先权后的新子树与原先的堆式队列进行合并。

3 性能分析

相比于传统的优先队列组织结构，新的堆式队列的数据组织方式能够有效地提高算法的效率，降低算法在程序实现方面的难度。其中“入队”、“降优先权”均可在常量时间内完成，其算法时间复杂度是 $O(1)$。最坏情况下，$(N-1)$个结点都是根结点的孩子结点(N 是堆式队列中结点的总数)，此时“出队”操作的最坏时间复杂度为 $O(n)$。本文给出的“两趟合并法”和“多趟合并法”在根结点删除后，可对根结点的

孩子结点进行两两合并，该方法能有效减少同一层上孩子结点的数目，使堆式队列结构变得更为平衡，降低运行过程中最坏情况发生的概率。

对特定数据结构中的数据进行的操作，通常由作用于该数据结构的一组基本操作序列来完成。为了更科学地评估这些操作的时间复杂度，文献[10]提出采用分摊复杂度方法来评价操作的性能。分摊复杂度不考虑渐进复杂度(即输入规模无限扩大的情况)，而是考虑每个操作的平均性能，其更符合算法在实际应用中的情况[11]。

势能法[12-13]是常用的分析分摊复杂度的方法，可用来评估每个操作的平均代价。一般而言，势能与整个数据结构而不是特定对象相关联。若某个数据结构需执行 m 个操作，对于每一个 $i(i=1,2,\cdots,m)$，令 t_i 为第 i 个操作的实际代价，Φ_{i-1} 和 Φ_i 分别对应执行操作之前和之后数据结构的势能。因此，第 i 个操作摊还时间 a_i 用势函数 Φ 可定义为：

$$a_i = t_i + \Phi_i - \Phi_{i-1}(i=1,2,\cdots,m) \tag{1}$$

由式(1)， m 个操作总的摊还时间可表示为：

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m}(a_i - \Phi_i + \Phi_{i-1}) = \sum_{i=1}^{m} a_i - \Phi_m + \Phi_0 \tag{2}$$

一般情况下，将 Φ_0 定义为 0，以及 $\Phi_i \geqslant 0$ $(i=1,2,3,\cdots,m)$。因此，可推导出：

$$\sum_{i=1}^{m} t_i \leqslant \sum_{i=1}^{m} a_i \ (i=1,2,\cdots,m) \tag{3}$$

式(3)表明，总的摊还时间是实际运算时间的上界，也就意味着可以根据一组操作的摊还时间估计出其实际运行时间，从而评估操作的性能。接下来，本文将势能法用于出队操作，进而分析出队操作的分摊复杂度，具体过程如下：

定义 $s(x)$ 为以 x 为根的二叉树的结点总数，$r(x)=\log s(x)$ 为结点的势能，n 为堆式队列总的结点数。因此，堆式队列每个结点的势能为[0，$\log n$]，总势能等于各结点势能之和。对于使用“两趟合并法”的出队操作来说，三种情形可能引发势能的变化，分别是：删掉根结点、第一趟合并以及第二趟合并。首先对第一趟合并进行分析，以图 4(a)为例，依据 c 是否为空，分别给出合并后势能变化情况：

$$\begin{cases} \log(s(a)+s(b)+1)-\log(s(b)+1) & c=\text{null} \\ \log(s(a)+s(b)+1)-\log(s(b)+s(c)+1) & c\neq\text{null} \end{cases}$$

通过推导可分析出两种情况势能变化的上限，主要过程如下：若 c 不为空，已知当 x， $y>0$，$x+y\leqslant 1$ 时，此时 $x=y=1/2$， $\log x+\log y$ 取最小值–2。可推出：

$$\begin{aligned} &\log(s(a)+s(b)+1)+\log(s(c))- \\ &2\log(s(a)+s(b)+s(c)+2)= \\ &\log((s(a)+s(b)+1)/(s(a)+s(b)+s(c)+2))+ \\ &\log((s(c))/(s(a)+s(b)+s(c)+2))\leqslant -2 \end{aligned} \tag{4}$$

又由于 $\log(s(c))\leqslant\log(s(b)+s(c)+1)$，可得到：

$$\begin{aligned} &\log(s(a)+s(b)+1)-\log(s(b)+s(c)+1)\leqslant \\ &2\log(s(a)+s(b)+s(c)+2)-2\log(s(c))-2 \end{aligned} \tag{5}$$

因为 $\log(s(x))=\log(s(a)+s(b)+s(c)+2)$，式(5)可化为:

$$\begin{aligned} &\log(s(a)+s(b)+1)-\log(s(b)+s(c)+1)\leqslant \\ &2\log(s(x))-2\log(s(c))-2 \end{aligned} \tag{6}$$

若 c 为空，则

$$\begin{aligned} &\log(s(a)+s(b)+1)-\log(s(b)+1)\leqslant \\ &2\log(s(a)+s(b)+2)=2\log(s(x)) \end{aligned} \tag{7}$$

第一趟合并中， $x_1,x_2,\cdots,x_{2k}$ 两两进行合并操作。此时， $s(x_{2i})\geqslant s(x_{2i+1})$， 总的势能变化为：

$$\begin{aligned} &\sum_{i=1}^{k-1}(2\log s(x_{2i-1})-2\log s(x_{2i})-2)+2\log s(x_{2k-1})\leqslant \\ &2\log s(x_1)-2(k-1)\leqslant 2\log n-2(k-1) \end{aligned} \tag{8}$$

与此同时，出队操作删除根结点时，总的势能下降 $\log n$； 第二趟合并时，势能最多增加 $\log(n-1)$；实际的操作时间为 $2k+1$。由式(1)和(2)可知，出队操作的上限为

$$2\log n-2(k-1)-\log n+\log(n-1)+2k+1$$

最终可得到在“两趟合并法”下，出队操作的分摊复杂度为 $O(\log n)$。

4 实验与应用实例

本文的数据组织结构具有高效性、通用性及可扩展性，可与多种算法进行配合、协同工作，有效

地提高算法的效率。数据结构的实现采用 C++类进行封装，通过调用接口操作，使相关算法在数据存储空间的自适应性、以及实现上的简单性和易用性得到了保证。

4.1 大规模数据排序

排序是指按照某个或某些关键字的大小，把一串记录按照递增或递减次序排列起来的操作[10]。排序算法就是使一串记录按照要求实现排列的方法。在大规模数据处理领域，高效的排序算法越来越受到人们的关注[14]。一般而言，在处理大规模且有序化程度较低的记录时，堆排序的时间复杂度是最优的，其插入、删除、降优先权操作的时间复杂度均为 $O(\log n)$。相比而言，本文堆式队列用于大规模、无序记录排序时，其进队、降优先权的时间复杂度为 $O(1)$，出队时摊还代价为 $O(\log n)$，在特定应用场合，可进一步提高排序算法的效率。

在排序过程中，通过“出队”操作可以获得当前堆式队列的最小权值，“两趟合并法”或“多趟合并法”使得剩余 n–1 个元素的序列重新构建成一个新的堆式队列，从而得到 n 个元素的次小值。如此反复执行，便能得到一个有序序列，最终实现大规模数据的排序。

为了验证上述堆式队列的结论，比较堆排序与堆式队列实现的排序算法之间的效率，作者使用 C++语言实现了这两种算法，并通过随机函数模拟产生不同数量(n)的长整型数据，在 PC 机(CPU Intel Core2×4 2.66 GHz，内存 4 GB)上测试排序时间，其实验结果如表 1 所示。

表 1 两种数据结构实现的优先队列排序时间的对比/ms
Tab.1 Time comparison of two types of sorting algorithms based on different data structures /ms

数据量	堆		堆式队列	
n	插入	删除	进队	出队
200	0.004 7	0.012 5	0.001 5	0.020 3
1 000	0.018 7	0.079 5	0.015 6	0.138 8
5 000	0.092 1	0.457 1	0.064	0.909 5
20 000	0.374	2.106	0.24	4.571
100 000	1.87	12.635	1.25	35.41

表 1 选取了 200，1 000，5 000，20 000，100 000 五种规模的数据量，并在此基础上对堆和堆式队列两种数据结构实现的数据排序进行了时间上的分析。实验结果表明，在初始给定所有数据的情况下，堆结构在排序的时间性能上要稍优于堆式队列。但在仿真实际情况下，初始阶段只会给定部分数据，在系统运行过程中，新的数据会动态地、逐步地添加到优先队列中去。此种情形下，本文给出的堆式队列在数据的动态存取方面优于堆结构。实验结果如表 2 和表 3 所示。

表 2 堆结构下动态存取不同规模数据所用的时间
Tab.2 Time consumed in dynamic access for different scale data under the heap structure /μm

动态存取	优先队列中数据的个数 n			
的规模 m	5 000	20 000	50 000	100 000
200	12.5	14.0	14.5	17.2
1 000	43.7	51.5	57.8	65.6
2 000	82.7	94.5	109.2	124.8
4 000	160.7	181.0	199.7	221.5
8 000	316.7	355.7	382.2	411.8

表 3 堆式队列下动态存取不同规模数据所用的时间/μm
Tab.3 Time consumed in dynamic access for different scale data under the heap-like queue /μm

动态存取	优先队列中数据的个数 n			
的规模 m	5 000	20 000	50 000	100 000
200	4.7	9.4	21.1	38.9
1 000	15.7	26.6	45.3	76.5
2 000	32.4	43.7	68.7	108.7
4 000	60.9	74.9	101.4	151.3
8 000	143.5	159.1	198.1	246.5

表 2 和表 3 选取 5000，20000，50000 以及 100000 四种规模的数据量，分析了堆和堆式队列动态存取数据的性能。由实验结果可知，当优先队列中数据个数确定的情况下，动态存取相同规模的数据，堆式队列在通常情况下较堆结构耗费时间更少，效率更高。

图 7 是动态存取数目确定的情况下，优先队列中的数据量与时间耗费之间的关系。实验结果表明，随着优先队列中数据容量的增加，堆和堆式队列在排序上耗费的时间差距逐渐缩小，但总体而言，堆式队列仍优于堆结构。

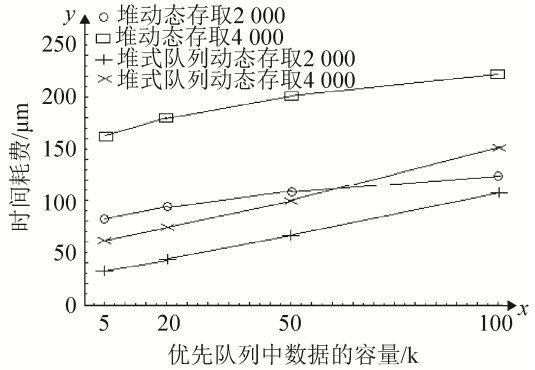


图 7 堆与堆式队列动态存取数据的时间耗费的对比
Fig.7 Comparison of time consumption of dynamic access data under heap and heap-like queue

4.2 计算最短路径

最短路径问题[15]是图论研究中的一个经典算法问题，旨在寻找图(由结点和路径组成)中两结点之间的最短距离。主要特点是以起始点为中心向外层次扩展，直到扩展到终点为止。Dijkstra 算法是当前求解图最短路径的经典算法，但因其遍历过程中计算的结点较多，所以导致算法的效率偏低。我们在本文堆式队列数据结构的基础上实现的 Dijkstra 算法，可进一步提高求解最短路径的效率，并节省系统的内存空间。算法的具体描述如下：

(1) 初始条件：构建空的堆式队列结构的优先队列，指定图中某一结点作为起点，将起点进队；

(2) 对优先队列实现“出队”操作，取出优先级最高的元素(即队列中距离最短的结点)，若该结点就是终点，则记录优先队列的出队序列，作为最短路径，并释放优先队列的内存空间；否则遍历出队结点的所有邻接点，并累加距离；

(3) 若出队结点的邻接点尚未添加到优先队列中，则将该附有相关信息的邻接结点进队；若邻接结点已经存在于优先队列，比较当前新计算的距离与已存在的距离的大小，如果前者更小，则调用“降优先权”操作，调整优先队列中结点的次序；

(4) 反复执行(2)(3)步，直到到达图中的目标结点，即终点；输出最短路径，并释放队列的内存空间。

若图中结点的数目为 V，路径或弧长的数目为 E。传统的 Dijkstra 算法时间复杂度为 $O(V^2)$，基于本文堆式队列的 Dijkstra 算法的时间复杂度为 $O(E+V\log V)$，算法性能得到了较为显著的优化。

4.3 计算最小生成树

日常生活中，经常会遇到在城市之间建设配电网、通信联络网、高速公路网等实际应用问题。在此类问题中，为实现总的耗费最省，一般采用图论中的最小生成树方法。此外，最小生成树在数据通信[16]、模式识别[17]、图像处理[18-19]和生物学[20-22]等领域中也越来越引起研究者的重视和肯定。

在图论中，一个无向连通图 $G=(V,\{E\})$，V 是图中结点的集合， E 是图中边的集合。给每条边 (u,v) $(u,v\in V)$ 添加一个权值 $w(u,v)$，则最小生成树问题本质上就是求解一个无环的边集，该边集连接所有的点并可使总权值最小。经典的最小生成树算法包括两种：Prim 算法和 Kruskal 算法。前者的时间复杂度的计算与图中的边数无关，主要用于求解边稠密图的最小生成树，而后者主要用于求解边稀疏图的最小生成树 T。本文在给定堆式队列数据结构的基础上，给出了 Kruskal 算法的实现，其具体过程如下：

(1) 初始条件：构建空的堆式队列结构的优先队列，输入连通图 G，图中结点的数目 n；

(2) 将图中每个结点作为一个子树，每一条边以权值作为优先权实施进队操作；

(3) 对优先队列实现“出队”操作，取出优先级最高的边，判定该边的两个端点是否来自于同一个子树。若是，则舍去该边。否则将该边添加到 T 中，并将两个端点所处两个子树进行合并；

(4) 反复执行第(3)步，若 T 中已经存在 n–1 个结点，表明最小生成树生成成功，释放优先队列的内存空间。

本文给出了改进的最小生成树算法，相比传统的最小生成树算法，算法时间效率得到了较为明显的提高，为更加高效地解决应用问题提供了支持。

5 结论

本文对优先队列的数据结构进行了探讨，给出了一种高效的堆式队列的数据组织方式，阐明了该数据结构的基本操作，理论上分析了其时间和空间

性能，并通过实验和应用实例的方式探究了堆式队列实现的优先队列实际运行的效率。研究表明，该数据结构具有优异的存取效率和数据的动态更新能力；在实际应用中能够充分地发挥算法的效率，降低算法在程序实现方面的难度。以后的研究主要围绕进一步优化堆式队列结构和提高大规模存取操作性能等方面开展。

参考文献：

[1] 高文宇, 王建新, 陈松乔. 网络仿真软件 NS2 中队列调度算法的扩展 [J]. 系统仿真学报, 2006, 18(2): 521-525. Gao W Y, Wang J X, Chen S Q. Extension of queue scheduling algorithm in NS2 [J]. Journal of System Simulation, 2016, 18(2): 521-525.

[2] Coming D S, Staadt O G. Kinetic sweep and prune for multi-body continuous motion [J]. Computers & Graphics (S0097-8493), 2006, 30(3): 439-449.

[3] 李红波, 赵永耀, 吴渝, 等. 一种基于距离约束的改进 SURF 算法 [J]. 系统仿真学报, 2014, 26(12): 2944-2956. Li H B, Zhao Y Y, Wu Y, et al. Improved SURF algorithm based on distance constraint [J]. Journal of System Simulation, 2014, 26(12): 2944-2956.

[4] Cris L, Luengo H. Revisiting priority queues for image analysis [J]. Pattern Recognition (S0031-3203), 2010, 43(9): 3003-3012.

[5] 李凤霞, 赵邓, 李仲君. 基于外观保持的多分辨率模型生成算法 [J]. 系统仿真学报, 2012, 24(1): 62-66. Li F X, Zhao D, Li Z J. Mesh simplification for 3D models with exterior maintaining [J]. Journal of System Simulation, 2012, 24(1): 62-66.

[6] Zhang C, Wang K, Mu H. A priority queue algorithm for the replication task in hbase [J]. Journal of Software (S1796-217X), 2013, 8(7): 1765-1769.

[7] Liu F, Fu C H. Improving mobile network performance with two queueing buffer allocations of priority-based queueing scheme [J]. Wireless Networks (S1022-0038), 2014, 20(6): 1349-1367.

[8] 杨神化, 施朝健, 关克平, 等. 基于 MAS 和 SHS 智能港口交通流模拟系统的开发与应用 [J]. 系统仿真学报, 2007, 19(2): 289-292. (Yang S H, Shi C J, Guan K P, et al. Intelligent simulation of traffic flow in port area with MAS and SHS [J]. Journal of System Simulation, 2007, 19(2): 289-292.)

[9] 庞明勇, 卢章平. 网格数据处理中的数据结构组织 [J]. 计算机工程与应用, 2004, 40(6): 91-93. (Pang M Y, Lu Z P. Data structure for calculating meshes [J]. Computer Engineering and Applications, 2004, 40(6): 91-93.)

[10] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to algorithms [M]. Cambridge, USA: MIT Press, 2001.

[11] Tarjan R E. Amortized computational complexity [J]. SIAM Journal on Algebraic Discrete Methods (S0186-5212), 1985, 6(2): 306-318.

[12] Fredman M L, Sedgewick R, Sleator D D, et al. The pairing heap: a new form of self-adjusting heap [J]. Algorithmica (S0178-4617), 1986, 3(1): 111-129.

[13] Pettie S. Towards a final analysis of pairing heaps [C]// Proceedings of the 2005 (46th) Annual IEEE Symposium on Foundations of Computer Science. USA: IEEE, 2005: 174-183.

[14] 李建中, 张艳秋. 基于蛇型磁带的海量数据排序算法 [J]. 软件学报, 2003, 14(1): 28-34. (Li J Z, Zhang Y Q. A massive data sort algorithm based on serpentine tape [J]. Journal of Software, 2003, 14(1): 28-34.)

[15] Subramani K, Madduri K. Two-level heaps: a new priority queue structure with applications to the single source shortest path problem [J]. Computing (S0010-485X), 2010, 90(3/4): 113-130.

[16] Guo W, Zhang B, Chen G, et al. A PSO-optimized minimum spanning tree-based topology control scheme for wireless sensor networks [J]. International Journal of Distributed Sensor Networks (S1550-1329), 2013: Article ID 985410.

[17] 汪闽, 周成虎, 裴韬, 等. 一种带控制节点的最小生成树聚类方法 [J]. 中国图象图形学报, 2002, 7(8): 765-770.(Wang M, Zhou C H, Pei T, et al. A MST based clustering method with controlling vertexes [J]. Journal of Image and Graphics, 2002, 7(8): 765-770.)

[18] 黎莹, 戴芳, 郝勇, 等. 基于最小生成树的图像分割 [J]. 计算机工程与应用, 2013, 49(13): 149-151.(Li Y, Dai F, Hao Y, et al. Image segmentation based on minimum spanning tree [J]. Computer Engineering and Applications, 2013, 49(13): 149-151.)

[19] Fabijańska A, Gocławski J. New accelerated graph-based method of image segmentation applying minimum spanning tree [J]. IET Image Processing (S1751-9659), 2014, 8(4): 239-251.

[20] 杨国慧, 周春光, 黄艳新, 等. 最小生成树用于基因表示数据的聚类算法 [J]. 计算机研究与发展, 2003, 40(10): 1431-1435.(Yang G H, Zhou C G, Huang Y X, et al. An algorithm for clustering gene expression data using minimum spanning trees [J]. Journal of Computer Research and Development, 2003, 40(10): 1431-1435.)

[21] 周康, 李刚, 谢振林, 等. 最小生成树 DNA 算法 [J]. 华中科技大学学报(自然科学版), 2012, 40(1): 30-34. (Zhou K, Li G, Xie Z L, et al. DNA algorithm of minimal spanning tree problem [J]. Journal of Huazhong University of Science and Technology (Nature Science), 2012, 40(1): 30-34.)

[22] Neumann F, Witt C. Ant colony optimization and the minimum spanning tree problem [J]. Theoretical Computer Science (S0304-3975), 2010, 411(25): 2406-2413.

https://dc-china-simulation.researchcommons.org/journal/vol29/iss1/13
DOI: 10.16182/j.issn1004731x.joss.201701013