半带宽优化

1. 简介

首先需要注意的是，半带宽优化已经被证明是一个NP完全问题，所以无法在多项式时间得到最优解。现在的算法主要是近似的优化算法，目前主流的算法有CM算法，GPS算法，tabu搜索算法，GRASP算法，模拟冷却算法等。

由于程序的最初设计以速度优先，故在选择的时候排除了耗时巨大的模拟冷却算法，而是采用了偏于经验类的算法。加之使用FORTRAN语言实现图结构困难，所以半带宽部分选择在stap90程序中实现最为原始的CM算法。而对于更加强力一点的算法，则选择用C++予以实现GRASP算法的部分。

1. 算法介绍
   1. CM算法

CM算法全称Cuthill-McKee算法在这两人于1969年名为Reducing the bandwidth of sparse symmetric matrices的论文中提出，是最先被提出的半带宽优化算法。其主要实现较为简单，主要就是通过一个广度优先搜索，每次把入度最小的点标号，并把该点的邻居加入队列中，通过循环得到最终的编号。

* 1. GRASP算法

GRASP算法是在2004年由西班牙Valencia大学的Estefania Piñana教授等人在其文章GRASP and path relinking for matrix bandwidth minimization中提出的。其吸取了前人（CM, GPS, Tabu）的优点，在试验中有着非常优秀的表现。

其主要内容分为三个阶段，构建阶段，提升阶段和路径重连阶段。

构建阶段提出了5中不同的方案，在保证优质初始解的同时也考虑了其随机性，避免其落入局部最优点。在大作业的C++部分实现中，我才用了C1部分，主要原因还是比较易于操作。

提升阶段参考于tabu算法的交换节点编号的部分，不过为了提升性能，GRASP算法仅仅考虑在交换时会改变最大带宽以及最大带宽点的个数的点。

路径重连部分适用于极大规模问题，而且需要有多局部极值，并不适用于本问题，故忽略。

1. 算法实现

在stap90中，如果要使用半带宽优化，为了简便操作，我选择了在输入文件中加入部分信息，也就是在输入ID之后就输入整体的连接矩阵。再次说明一下这一点的必要性，因为半带宽优化关心的是图的整体结构，所以整体的连接矩阵必不可少，而新产生的ID矩阵又必须在组装过程中发挥作用，所以选择了在输入原先的ID之后直接输入连接矩阵。

在具体实现中，因为FORTRAN没有可变数组的支持，我先是构建了一个node模板和一个list模板，具体接口请详见list.f90与node.f90。通过循环将与自由度i有关的所有自由度都放在了链表数组lists(i)之中，然而由于链表的搜索功能是其软肋，所以在实现搜索的时候还是比较麻烦的。

而C++的部分则是比较轻松，同时实现了类似的接口完成了算法实现。

Stap90的实现位于solver\_mode.f90文件中的bdopt函数，C++的实现位于BandwidthOpt文件夹中。

1. 效果

在实验中，我发现由于决定skyline算法时间的主要因素是nwk，也就是skyline存贮中的元素个数，而不是最大带宽，加上桥问题的主要问题在于其部分元素带宽很大，而大多数很小，导致使用了优化虽然可以使带宽从1492降至1356，但是总元素个数增加，时间反而从12s增长至18s，得不偿失。这说明半带宽优化并不适用于所有问题。

稀疏矩阵求解器

1. 简介

首先对稀疏矩阵进行一些简单介绍。

稀疏矩阵主要指非零元素为O(n)的矩阵。其没有必要使用以往的O(n^2)求解，而是可以利用特殊的储存方式，大大缩减存储空间。与此同时，因为实际上的计算大多仅仅与非零元素有关，所以有算法可以将求解线性方程Lx=y的时间降至O(flop)(其中flop为非零元素个数)大大降低其运算时间，是求解有稀疏性质的线性系统的一把利器。

本次采用intel MKL中集成的pardiso求解器，因为其为市面上速度最快的求解器。矩阵的存储方式采用默认的CSC格式。值得一提的是，相比于stap90中原有的skyline求解器，pardiso在空间上有着劣势。因为skyline求解器可以在原地进行L^TDL分解，而CSC存储的矩阵，由于不知道分解后非零元素的位置，需要在过程中被迫开一份O(n^2)的空间。

1. 实现

由于具体的函数调用都是相同的MKL中的pardiso，所以实现中更重要的是一些小细节，而这些小细节就是保证本组成为性能第一的根本。

实现中最主要的部分是如何构建CSC存储所需要的3个数组。当时考虑了是否要利用stap90原有的maxa来改写得到CSC格式，但是仔细思考后，我认为那种方式在初始的时候开辟了过多的内存，所以舍弃，而是采用直接从连接矩阵入手的想法。

而对于这种想法，我采用了与半带宽优化同样的策略，使用链表得到所欲求的数组。但是在后续的优化中，我发现大量的指针不利于编译器优化，在大规模问题中大大拖累了时间，所以毅然决定改为使用可分配数组。但是由于需要整体的大小，只好牺牲额外的空间，不过好在牺牲应该不大。

另外一点，对于稀疏矩阵来说，最重要的性能参数应当是稀疏度，所以我在初步组装之后设计了一种O(n)数量级的crop算法，将组装时加入的0元素去掉，从而将稀疏度降低到0.0036%（对于Job-3）.

最后一个设计上的优势就是我们发现，windows的磁盘设置导致我们无法一次开辟存储超过2G的数组，所以当规模非常大的时候，使用原有的memalloc已经无法满足Job-4的需求，所以我通过使用两个可分配数组来存放CSC中大小为nwk的数组以克服这个问题。

1. 实验效果

在测试中，我们以Job-1 0.25s，Job-2 3s, Job-3 30s的成绩，比第二名在每一项都快至少一倍轻松得到了第一名。

1. 感受

在本组的分配中，我主动选择了速度优化这一个方面，在时间的过程中我学到了很多东西。最重要的两点是：

第一点，是在优化并不完全是设计算法，而是一个对整个问题的把控。

第二点，对于编写软件的人，硬件知识也是必须的。我曾经因为没有把编译器调到x64而导致内存被限制在了4G以内而困扰了很久，包括前文提到的windows不能开2G的数组都是一些常见的硬件知识。

最后，必须要说能够看到本组的程序跑Job-1从最开始的40s到0.2s真的是非常自豪，非常激动。