Maze Routing

Paper Reading

- maze routing 问题实际上是单源最短路（SSSP）问题。本次作业给出的数据中，所有边的权重均在1~5之间。
- 我主要采用了Δ stepping算法，因此列出算法相关的基础算法和一个改进算法。

Dijkstra Algorithm

- Dijkstra算法是求解单源最短路问题最经典的算法之一。
- Dijkstra维护两个集合S与D，在集合S中的点均是已经确定最短路径的点，在D中的点中距离S最近的那一个也可以确定自己的最短路径，于是从D加入S。由此循环，直到D为空集。

```
DIJKSTRA(G, w, v0)
for each vertex u ∈ V(G)
    u.dist = ∞
v0.dist = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each vertex v ∈ V(G) such that (u, v) ∈ E(G)
        if v.dist > u.dist + w(u, v)
            v.dist = u.dist + w(u, v)
            DECREASE-KEY(Q, v, v.dist)
```

Bellman Ford Algorithm

- bellmanford循环地对每条边进行松弛操作，即如果节点v有与之相邻的节点u，使得$d[v] > d[u] + c(u, v)$，那么更新$d[v]$为$d[u] + c(u, v)$

Δ Stepping Algorithm

- 这个算法采取分段处理的模式，将所有端点根据该端点到原点的暂时距离tent对Δ的倍数k划分到桶$B[k]$中。
- 具体来讲，对于桶$B[i]$，其中存放的是到源点暂时距离倍数为$[i-1, i)$的点。与这些点v相邻的端点u，如果$tent[v] + c(v, u) < tent[u]$，那么更新$u$的暂时距离。具体分为两种情况
    - $c(v, u) < \Delta$：这种情况下，更新的u的暂时距离可能还在同一个桶$B[i]$中，因此需要反复判断这个桶是否是非空的
    - $c(v, u) > \Delta$：这种情况下，更新的u的暂时距离一定超过桶$B[i]$可容纳的范围
- 更新后的距离可能与原距离不在一个桶内，因此要将u从原来的桶移动到新桶中

```
foreach v ∈ V do                                       -- Initialize node data structures
    heavy(v) := {(v, w) ∈ E : c(v, w) > Δ}             -- Find heavy edges
    light (v) := {(v, w) ∈ E : c(v, w) ≤ Δ}            -- Find light edges
    tent  (v) := ∞                                     -- Unreached
relax(s, 0); i := 0                                    -- Source node at distance 0
while ¬isEmpty(B) do                                   -- Some queued nodes left
    S := ∅                                             -- No nodes deleted for this bucket yet
    while B[i] ≠ ∅ do                                  -- New phase
        Req := {(w, tent(v) + c(v, w)) : v ∈ B[i] ∧ (v, w) ∈ light(v)}
        S := S ∪ B[i]; B[i] := ∅                       -- Remember deleted nodes
        foreach (v, x) ∈ Req do relax(v, x)            -- This may reinsert nodes
    od
    Req    := {(w, tent(v) + c(v, w)) : v ∈ S   ∧ (v, w) ∈ heavy(v)}
    foreach (v, x) ∈ Req do relax(v, x)                -- Relax previously deferred edges
    i := i + 1                                         -- Next bucket

Procedure relax(v, x)                                  -- Shorter path to v?
    if x < tent(v) then                                -- Yes: decrease-key respectively insert
        B[⌊tent(v)/Δ⌋] := B[⌊tent(v)/Δ⌋] \ {v}         -- Remove if present
        B[⌊x      /Δ⌋] := B[⌊x      /Δ⌋] ∪ {v}         -- Insert into new bucket
        tent(v) := x
```

Radius Stepping

- 与Δ stepping类似，但是将Δ stepping固定的radii更改为灵活可变的。
  - 具体来讲，对于已经获得最小值的点，将他们加入集合S, radius stepping给每个点设置了一个vertex radius $r(\cdot)$，i次循环达到的点中，只有距离不大于$min_{v\in V/S_{i-1}}\{\delta(v)+r(v)\}$的点，他们的邻居节点的暂时距离被更新。之后，这些符合要求的节点被加入集合S
  - $r(\cdot)$的计算
    - $r(\cdot)$取任何值都是正确的，但是小于K半径的取值使得内循环的bellman ford算法执行不超过k+2步

Algorithm 1: The RADIUS-STEPPING Algorithm.

Input: A graph $G=(V,E,w)$, vertex radii $r(\cdot)$, and a source vertex s.
Output: The graph distances $\delta(\cdot)$ from s.

$\delta(\cdot)\leftarrow+\infty,\delta(s)\leftarrow 0$
foreach $v\in N(s)$ **do** $\delta(v)\leftarrow w(s,v)$
$S_0\leftarrow\{s\}, i\leftarrow 1$
while $|S_{i-1}|<|V|$ **do**
  $d_i\leftarrow\min_{v\in V\setminus S_{i-1}}\{\delta(v)+r(v)\}$
  repeat
    foreach $u\in V\setminus S_{i-1}$ $s.t.$ $\delta(u)\leqslant d_i$ **do**
      foreach $v\in N(u)\setminus S_{i-1}$ **do**
        $\delta(v)\leftarrow\min\{\delta(v),\delta(u)+w(u,v)\}$
  until *no* $\delta(v)\leqslant d_i$ *was updated*
  $S_i=\{v\mid\delta(v)\leqslant d_i\}$
  $i=i+1$
return $\delta(\cdot)$

实现细节

- 读入文件：使用fstream的fin重定向。按行读取，在每一行中按字符顺序将字符串转换为数字。将D和S点权重均设置为0保证结果准确
- 结构体：使用node，保存点坐标，以及到S的目前最优的距离。delta-stepping还额外有prev_x, prev_y记录路径上前一个点。
- dijkstra串行算法：为了方便向并行算法转化，我维护了一个优先队列，这个队列中都是还需要再次计算的节点。对每个节点v，节点的邻接节点就是上下左右四个点，因此，在确认这四个点没有越界以及没有障碍后，计算邻居节点经过节点v到源点S的距离与不经过v到原点的距离，如果前者小于后者，说明这个邻居节点可更新与之相关的节点，因此将这个邻居节点再次压入优先队列中。如果一个节点没有再被压入队列，就意味着这个节点不再能更新经过它的节点的路径权重，也就意味着这一点到源点S的距离已经是最优的。因此，当队列为空，意味着所有点到S的距离都已经达到最优。
- delta-stepping并行算法：这个算法结合了dijkstra算法和bellman ford算法，具体算法内容已经在第一部分写过。在实现中，我将B设置为vector数组，将REQ和S设置为与线程数大小一样的vector数组，方便并行。算法主要分为：
  - 初始化
  - 循环查看每一个B[i]
    - 清空REQ，此时有THREAD_NUM个REQ，使用parallel for分发给每个线程
    - 循环light weight边，此时有B[i].size()个循环，使用parallel for并行
    - 主线程清空B[i]
    - relax，每个线程负责清空一个REQ，将线程共用的变量设置在critical session 中

    - heavy weight边无需考虑原B[i]，现在所有原B[i]中的点已经分在THREAD_NUM个S中，每个线程负责将一个S中的heavy weight点插入REQ中
    - relax
    - i++，开始检查新的桶B
- 寻找来路：每次更新权重时，维护数组step[10000][10000][2]记录导致权重更新的邻居的坐标。在找到最优点后，从D到S寻找路径并弹入stack中，打印时从stack中依次弹出
- 输出：使用fstream的fout将cout重定向至result.txt

实验结果

	平均运行时间/s	占用内存
Dijkstra	95	见下
Delta-stepping	280	见下

Dijkstra占用内存：记录maze本身，记录最终距离，记录路径，共10000*10000*(1 + 1 + 2)*sizeof(int)；队列长度未知。

Delta-stepping占用内存：B中最多储存的node数数量级为万（5*sizeof(int)*k*10000），其中每个B中的元素均要进入S，有更新的元素进入REQ

总之，实验结果非常糟糕，在时间上完全输给了串行算法，并且更改delta和线程数都不能改善这一点。我反思了几点不足，列在下面。

可改进的地方

- B可改进：可以改为B[THREAD_NUM][5/delta+1]，因为在程序运行过程中，只会同时出现从队列中弹出的属于B[i]的点，和加上权重后更新的边，这个更新后的边的权重不会超过delta*i + 5，也可以在B中直接实现并行，同时大大节省空间
- relax可改进：由于B现在是多线程共享的变量，导致relax中对B的修改只能在critical session中进行，降低运行效率，使用如上B的修改后，对B的增加和删除可以并行，可以将耗时的查找及修改vector同时交由多个线程执行。