Shortest Path Algorithm with Heaps

March 31, 2024

Contents

1 Introdution **3**

2 Data Structure/Algorithm **4**
2.1 Dijkstra Algorithm 4
2.2 Fiboacci Heap 4
2.3 Binominal Heap 8

3 About Benchmark **10**
3.1 What is Benchmark 10
3.2 Why use it 11

4 Test Result **11**

5 Analysis **11**
5.1 Time Complexity 11
5.2 Space Complexity 12

1 Introdution

Shortest path problems are ones of the most fundamental combinatorial optimization problems with many applications. The efficient algorithm Dijkstra's algorithm is created be Edsger W. Dijkstra in 1956.

In this project we are going to compute the shortest paths using Dijkstra's algorithm. We'll implement the algorithm with two different heap structures, Binomial heap and Fibonacci heap. The goal of this project is to find the best data structure for the Dijkstra algorithm.

For testing, the USA road networks and the benchmark for the 9th DIMACS Implementation Challenge will be used. And more than 1000 query will be used for evaluating the run times.

2 Data Structure/Algorithm

2.1 Dijkstra Algorithm

Algorithm 1 basic structure of dijsktra code

Input: $G(V, E)$
Output: $V.dist = shortestpath$

```
build the heap
source.distance ← 0
insert(source)
temp ← extractMin()
while temp do
   known[temp] ← 1
   distance[temp] ← temp.distance
   if temp = target then
      return distance[temp]
   end if
   for every nodes in the adjacent list of temp do
      node.distance = arc-length + distance[temp]
      insert(node)
   end for
   while known[temp] do
      temp = extractMin()
   end while
end while
```

2.2 Fiboacci Heap

A Fibonacci heap is composed of a collection of connected trees that result in a forest-like structure. Each tree within the Fibonacci heap follows a "heap-ordered" structure. The insertion and merge operation of this data structure just cost $O(1)$, and the extractMin operation costs $O(logN)$

Fibonacci Heap maintains a pointer to the minimum values. All tree roots are connected using a circular doubly linked list, so all trees can be accessed using a single pointer.

Algorithm 2 Fibonacci heap–insert

```
addNode( node , root )
node->left = root->left
root->left->right = node
node->right = root
root->left = node
insert( node )
if keyNum=0 then
  min = node
else
  addNode(node,min)
  if node->key = min->key then
    min = node
  end if
end if
key++
```

The merge of two heaps:join root lists of Fibonacci heaps H1 and H2 and make a single Fibonacci heap H.Then compare the minimum elements of the two heaps. The pseudocodes are as follows:

Algorithm 3 Fibonacci heap–merge

```
catList(a,b)
tmp = a->right
a->right = b->right
b->right->left = a
b->right = tmp
tmp->left = b
combine( node )
if node==null then
  return
end if
if node->maxDegree > this->maxDegree then
  swap( node,this )
end if
if this->min==null then
  this->min = node->min
  this->keyNum = node->keyNum
  free(node->cons)
  delete node
else if  node->min=null  then
  free(node->cons)
  delete other
else
  catList( this->min,node->min )
  if this->key->node->key  then
    this->key = node->key
  end if
  this->keyNum += node->keyNum
  free( node->cons)
  delete node
end if
```

Algorithm 4 Fibonacci heap–extractMin

```
removing min node means removing the tree it belongs to.
extractMin()
p = min
if p==p->right then
    min = null
else
    removeNOde(p)
    min = p->right
end if
p->left = p->right = p
return p
removeMin()
if min==null then
    return
end if
child = null
m = min
while m->child do
    child = m->child
    removeNode(child)
    if child->right = child then
        m->child = null
    else
        m->child = child->right
    end if
    addNode(child,min)
    child->parent = null
end while
removeNode(m)
if m->right= m then
    min = null
else
    min = m->right
    consolidate()
end if
keyNum–
delete m
```

2.3 Binomial Heap

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows the Min Heap property. And there can be at most one Binomial Tree of any degree.

Algorithm 5 Binomial heap–merge

$treeMerge(p, c)$
$parent[c] \leftarrow p$
$sibling[c] \leftarrow child[p]$
$child[p] \leftarrow c$
return p
$heapMerge(H1, H2)$
$node1 \leftarrow head[H1]$
$node2 \leftarrow head[H2]$
while node1 != NULL and node2 != NULL **do**
  if degree[node1] ¡= degree[node2] **then**
    $H3 \leftarrow node1$
    $node1 \leftarrow sibling[node1]$
  else
    $H3 \leftarrow node2$
    $node2 \leftarrow sibling[node2]$
  end if
  if pre = NULL **then**
    $pre \leftarrow H3$
    $head[heap] \leftarrow H3$
  else
    $sibling[pre] \leftarrow H3$
    $pre \leftarrow H3$
  end if
end while
if node1 != NULL **then**
  $sibling[H3] \leftarrow node1$
else
  $sibling[H3] \leftarrow node2$
end if
return heap

Algorithm 6 Binomial heap–union

```
union(H)
H ← heapMerge(H1, H2)
x ← head[H]
pre ← NULL
while sibling[x] != NULL do
  if degree[x] != degree[sibling[x]] then
    pre ← x
    x ← sibling[x]
  else if sibling[sibling[x]] != NULL then
    if  degree[x] = degree[sibling[sibling[x]]] then
      pre ← x
      x ← sibling[x]
    else
      x ← treeMerge(x, sibling[x])
    end if
    if pre then
      sibling[pre] ← x
    else
      heap ← x
    end if
  else
    x ← treeMerge(x, sibling[x])
    if pre != NULL then
      sibling[pre] ← x
    else
      heap ← x
    end if
  end if
end while
return heap
```

Insert k to a Binomial Heap. This operation first creates a Binomial Heap with a single key k, then calls union on H and the new Binomial heap.

Extract the minimum element of the heap. This also uses a union. We first call getMin to find the minimum key Binomial Tree, then remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. This requires O(logN) time.

Algorithm 7 Binomial heap–insert and removeMin

```
insert(H, n)
value[newHeap] ← n
heap ← head[H]
if heap = NULL then
  heap ← newHeap
else
  heap ← union(heap, newHeap)
end if
return heap
removeMin(H)
minnode ← heapMin(H)
node ← head[H]
while sibling[node] and sibling[node]  do
  node ← sibling[node]
end while
sibling[node] ← sibling[sibling[node]]
newode ← child[minnode]
nextnode ← null
while newnode do
  head[newH] ← newnode
  parent[newnode] ← null
  tmpnode ← sibling[newnode]
  sibling[newnode] ← nextnode
  nextnode ← newnode
  newnode ← tmpnode
end while
return union(H, newH)
```

3 About Benchmark

3.1 What is Benchmark

The benchmark test platform is a research paradigm used to measure the performance of algorithms and compare the performance of different algorithms. The DIMACS Challenge website contains the shortest benchmark platform.

Specifically, we only use the short-circuit problem (no-nega-ARC) in which there is No Negative side Single-Source Shortest Path Problem (NSSP) test benchmark.

3.2 Why use it

1

Reduce the impact of machine hardware with reference algorithms.

Since the program runtime is affected by many underlying architectures, in order to exclude machine influences on the performance of the comparison algorithms, DIMACS recommends using the time "relative to the standard algorithm" and comes with a standard NSSP solver.

2

A more precise timer.

The timer used by benchmark is ¡sys/time.h¿ from Linux, which provides a precision of 0.01ms.

3

Generate test data in batches.

The platform's '.ss' test files are not written to death, but are randomly generated by code, generating more than 400 individual test cases per graph.

Finally, the total running time is averaged to obtain the solving performance of the algorithm on this graph. Doing so avoids testpoint bias for the best or worst case.

4 Test Result

5 Analysis

5.1 Time Complexity

The construct of the adjacent list will cost $O(V+E)$.With per insertion costing $O(1)$,the total cost of insertion will be $O(E)$.Extracting minimum elements will cost $O(VlogE)$.

The time complexity will be $O(VlogE)$ for most sparse matrix.Which is more quickly than the $O(V^2)$ complexity when using a table.

5.2 Space Complexity

With the adjacent list costing $O(V + E)$ and the heap costing $O(E)$,the space complexity is $O(V + E)$.

Delariation

We hereby declare that all the work done in this project titled "Roll Your Own Mini Search Engine" is of our independent effort as a group.