Shortest Path Algorithm with Heaps

March 31, 2024

Contents

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 $O(logN)$

Algorithm 2 structure of Fibonacci heap

j_1

2.3 Binominal Heap

Algorithm 3 Binomial heap–merge

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

Algorithm 4 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
```

Algorithm 5 Binomial heap–insert 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

pro1

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.

pro2

A more precise timer.

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

pro3

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.