Analysis and Comparison between the Algorithm Time Efficiency of Dijkstra and SPFA

Name Xie Xinfeng

Advisor 林建源

School Nan'an No.1 Middle School of Fujian

Analysis and Comparison between the Algorithm Time Efficiency of Dijkstra and SPFA

Abstract:

Dijkstra and Bellman-ford are the two basic methods to solve Single-source shortest paths problem， and SPFA is the optimized version of Bellman-ford. However, since the algorithm time complexity is 0 (kn), SPFA's time efficiency has some instabilities. This paper will compare the time efficiency between Dijkstra optimized with heap and three kinds of SPFA in different density graphs. Meanwhile, random data is created to test optimized Dijkstra and 3 types of SPFA. Finally, according to the calculation result of two methods' time efficiency, some suggestion is given for their application under different situation.

Key words:

Dijkstra, SPFA, algorithm time efficiency, Application of shortest path algorithm

摘要

Dijkstra 和 Bellman-ford 算法是解决单源最短路问题的两个基本算法，SPFA 算法是 Bellman-ford 的优化版本，但因为算法时间复杂度为 O(kn),所以该算法具有时间效率上的不稳定性。本文就 Dijkstra 在堆优化后与 SPFA 算法在稠密程度不同的图上的时间效率进行讨论对比，并通过制造随机数据对堆优化后 Dijkstra 算法与 3 种 SPFA 算法进行测试。最后得出关于两个算法时间效率的结论，并针对不同层面的运用问题，提出了参考建议。

关键词： Dijkstra　SPFA　算法时间效率　最短路算法应用

Introduction:

Invented by Edsger Wybe Dijkstra, an outstanding computer scientist in the Netherlands, Dijkstra algorithm was used to solve Single-source shortest paths problem with an algorithm time complexity of $O(n^2)$. If optimized by data structure heap, its algorithm time complexity can reach $O(n\log_2 n+m)$. However, Dijkstra algorithm has a great defect that its correctness is false in negative weight graph (counter examples are easily accessible).

Bellman-Ford algorithm was co-founded by Lester Ford and Richard Bellman, who is a famous American mathematician and the founder of dynamic programming. It is supposed to solve Single-source shortest paths problem with negative edge weight. Meanwhile, you can determine whether the FIG contains negative weight ring and its time complexity is O(nm).

Because of the low efficiency of Bellman-Ford algorithm, Duan Fanding of China Southwest Jiaotong University proposed SPFA algorithm in 1994 (full name: Shortest Path Faster Algorithm). As relaxation operation tends to happen only in the shortest path leading node succeeded slack, you can avoid the redundant computation with a rank to record the slack off need. Its algorithm time complexity is O (kn). And k is a number determined by the FIG.

The author is interested in the uncertainty of SPFA. Which algorithm is better, SPFA or Dijkstra? In other words, under what circumstance can Dijkstra be better? And under what circumstance can SPFA be better? What kind of choice should be made when it comes to application? These are the main issues to be discussed here.

This paper contains five sector as followed:

Sector One Basic Theory: a brief introduction to Dijkstra and SPFA.

Sector Two Algorithm Optimization: introduce the methods to optimize Dijkstra with heap, optimize SPFA with sort edge set group and deque. Thus four procedures are set: Dijkstra，SPFA-1，SPFA-2，SPFA-3.

Sector Three Algorithm Efficiency Analysis: analysis of time complexity of Dijkstra and three kinds of SPFA in theory.

Sector Four Experimental Data and Analysis: list out the experimental data of this research, some conclusion reached and their reasons as well.

Sector Five Application of SPFA in Sparse Graph: the efficiency and one application of SPFA in some graph will be proposed here.

The fourth and fifth sectors are the core part of this paper, aiming to solve problems proposed.

In Appendix four procedures used in the experiment are listed, which are written in

C++ language.

In this paper "n" represents the number of points of the FIG; m represents the number of edges of the FIG. Figures here refer to the undirected graph and edge weights are positive. "k" is a random number, determined by the extent of the dense graph. "o" is an asymptotic symbol used to measure time complexity. Refer to "Introduction to Algorithm" in reference bibliography for strict definition.

Sector One - Basic Theory: a brief introduction to Dijkstra and SPFA

1.1 Introduction to Dijkstra algorithm

In a graph with n, single-source shortest paths is supposed to be solved. Suppose we want to figure out the length of shortest paths from each point to node 1. Let the length of shortest paths from node i to node 1 be dis[i]. If there is no node 1 to node i, then let dis[i]= +∞. Vertex are divided into two states: one is with the length of shortest paths uncertain; the other certain. Each time choose one node without certain length of shortest paths with min dis[i], and name ti node s. And then mark node s as node with certain length of shortest paths. And search all its adjacent points, and name them node t. If dis[s]+d[s][t]<dis[t], then dis[t]=dis[s]+d[s][t]. (d[s][t] means the distance from node s to node t). Repeat these procedures until all the points are marked points with certain length of shortest paths.

1.2 Introduction to SPFA

In a graph with n, single-source shortest paths is supposed to be solved. Suppose we want to figure out the length of shortest paths from each point to node 1. Let the length of shortest paths from node i to node 1 be dis[i]. If there is no node 1 to node i, then let dis[i]= +∞. Build a deque with node 1 included. Each time choose a "s" from the top the deque, and search all its adjacent points, and name them node t. If dis[s]+d[s][t]<dis[t], then dis[t]=dis[s]+d[s][t]. (d[s][t] means the distance from node s to node t). Then check whether "t" is included in the deque. If not, put "t" at the end of the deque ; If yes, then no operation is applied. Continue the former procedures till the

deque comes to empty.

Sector Two - Algorithm Optimization

2.1 Optimization of Dijkstra

In the process of finding the min dis, there is a simple and direct way of going through the dis array to find the min value. This approach is very straightforward, but also very time consuming. Given that each time we are supposed to find the min dis of the nodes with uncertain length of shortest paths, we tend to establish a heap. Put all the nodes in the heap initially, and remove the top element of the heap each time. By doing these, we have three advantages: first, each time we can choose the min value of the element; second, elements in the heap are all nodes with uncertain shortest paths; third, the efficiency of heap is higher than linear form. The use of heap is a great optimization of Dijkstra. Its algorithm efficiency will be analyzed later in this paper.

No redundant details about data structure heap will be listed here in this paper. Its time complexity of up down operation conclusion is directed used in this paper. Refer to "Introduction to Algorithms" for detailed introduction and certification.

2.2 Optimization of SPFA

2.2.1 Sorting of edge set group

A storage of graphs by adjacency list is built according to the edge information read-in. In fact, it can be ensure that edge with minimum weight will be in front of group if we insert sorting during the read-in procedure. What can be efit from this optimization process? Suppose vi is ahead of vj in the adjacency list, and we keep dis[s]+d[s][vi]<dis[s]+d[s][vj] during the algorithm process, therefore if both two nodes are relaxation operated and enqueue, then vi is ahead of vj. If it turns out to be dis[vi]+d[vi][vj]<dis[vj], and vj is in the deque, then the relaxation operation is applied but it’s no necessary to put the node in the deque. However without sorting, vj will be ahead of vi if we put both two nodes in the deque when dis[s]+d[s][vi]<dis[s]+d[s][vj]. Then vj will dequeue but enqueue again after relaxation operated by vi， and search all its adjacent points and name them t, all the point t which have been relaxation operated shall meet following formula: dis[vj]+v[vj][t]<dis[t]. This formula is valid even vj enqueue again after relaxation operation, and t will be relaxation operated again which is the repetition of vj’s move and can be avoid. Theoretically, it can be concluded that sorting of edge set group can optimize SPFA algorithm (Attention! here we only consider the situation theoretically, however, the experimental data will surprise all of us, which we will discuss later).

2.2.2 Change deque to double-end deque

Traditional deque structure is FIFO （First Input First Output）.Using double-end deque instead of FIFO in SPFA means that elements can enqueue from both side of deque but dequeue only from the top side. Additional judgment is applied when considering the enqueue of an element vi: if dis[vi]<dis[vj] (here vj is the top element of the deque), then make vi the top element, otherwise put vj in the end of the deque. Double-end deque can help to avoid redundant procedures. After the dequeue of vi, if dis[vi]+d[vi][vj]<dis[vj],then no need to make vj enqueue again after relaxation operating dis[vj]. However if vj enqueue according to traditional method, then it will be relaxation operated by vi after dequeue, and search all its adjacent points, which will result in that all the points which could have had been relaxation operated are still being relaxation operated. And this is the repetition of vj's move when it dequeue first time.

Since the original Dijkstra algorithm time complexity is O(n2), it is meaningless to compare SPFA with original Dijkstra.

Dijkstra stands for Optimize Dijkstra with heap;

SPFA-1 stands for the original SPFA;

SPFA-2 stands for Optimize SPFA with double-end deque;

SPFA-3 stands for optimize SPFA with double-end deque and sorting of edge set group.

Following paragraphs are the efficiency analysis and data test of these four procedures.

Sector Three - Algorithm efficiency analysis

3.1 Dijkstra

Time complexity became to be O(log2n) in choosing node after heap optimization. And it became to be O(log2n) in raising the heap after each relaxation operation (because weight decrease after relaxation operation, only up-regulation is applicable.) Choosing points shall be processed n times while raising the heap. In a word, algorithm time complexity become O(nlog2n+m) after optimization.

3.2 SPFA-1

O(kn)

3.3 SPFA-2

If the Insert sorting process is applied during read-in edge to build adjacency list, then sorting time complexity is O(mlog2m). Time complexity will remain O(kn) when

processing enqueue, dequeue and relaxation operation procedures.(k is underage). In a word, algorithm time complexity is O(kn+mlog2m).

3.4 SPFA-3

Compare to the original SPFA, Double-end deque is a method with slight optimization. The value of K is reduce, but algorithm time complexity will remain O(kn).

Name k as ki in SPFA-i，then it is easy to come to following conclusion； k1>k2, k1>k3。

Sector Four - Experiment data analysis

Four programs (Dijkstra，SPFA-1，SPFA-2，SPFA-3) are wrote, and generated random data with n=1000 and different value of m. Each value of m contains 10 data. Followings are the experiment results which are the arithmetic mean value of these 10 data's running time.

	Dijkstra	SPFA-1	SPFA-2	SPFA-3
m=50000	267ms	282ms	251ms	296ms
m=100000	378ms	414ms	480ms	573ms
m=150000	487ms	544ms	503ms	876ms
m=200000	586ms	678ms	636ms	1179ms
m=250000	679ms	781ms	744ms	1497ms
m=300000	776ms	896ms	849ms	1817ms
m=500000	1159ms	1338ms	1297ms	3105ms
m=600000	1344ms	1577ms	1504ms	3761ms
m=800000	1734ms	2006ms	1954ms	5134ms
m=1000000	2090ms	2384ms	2551ms	6566ms

(Testing with CPU 1.99GHz computer)

With above data, following conclusion are arrived；

Conclusion 1: ki>13 in the graphs of m/n≥5

Reason: For time efficiency, Dijkstra algorithm has a node coefficient of log2n while SPFA has a node coefficient k. During the experiment, it can be easily noticed that

k>log210000= 13.2877 under the situation of m/n≥5.

Conclusion 1 can naturally lead to following deduction:

Deduction 1: SPFA is no better than Dijkstra when dv=d/n≥10.

Reason: m/n≥5 means d≥2*m≥10*n=100000 in the whole graph, therefore dv=d/n ≥10 for each node.

Conclusion 2: Edge changes' affection is bigger in SPFA than in Dijkstra, and it is biggest for SPFA-3 with lowest efficiency,

Reason: It can be seen from conclusion 1 that when m/n≥5, k is bigger. And graph's density indirectly affects algorithm's efficiency through k, which means SPFA can be easier affected by edge than Dijkstra. Why SPFA-3 has lowest efficiency: Sorting of edge set group has a certain effect on algorithm optimization. However it requires preprocess (edge set group sorting), and sorting algorithm has a time complexity of O(mlog2m). Compare to O(m), SPFA-3 can be strongly affected with the increasing of m.

Conclusion 3: There is an uncertainty in double-end deque.

Reason: This conclusion can be seen by comparing two group data of m=100000 and m=1000000 with SPFA-1. Actually, during testing, it can be found that the range and variance of ten data from double-end deque used to calculate arithmetic mean value are rather big .Enqueue judgment testing of double-end deque consumes time. Besides, different graphs require different times of optimization and are impacted by distribution of edge weight. However, time-consuming of testing is rather small compare to sorting, therefore, even though it doesn't produce effects to the optimization, it doesn't' produce much effects to the program either.

Sector Five - Application of SPFA in Sparse Graph

The fact is that SPFA has higher time efficiency in Sparse Graph than in Dijkstra.

Two factors such as m/n，dv=d/n are brought to draw graph's density. And when dv=d/n ≥10 SPFA's efficiency is no higher than Dijkstra's. The author believes that dv=d/n≥ 10 is not qualified to be called as sparse. Since there is still 10 relevant points left for each node, multiple enqueue has a rather high possibility.

Based on above discussion we define sparse as dv=d/n≤5.

The author considers that urban street map is a typical sparse graph, what's its m/n?

In the urban street map, each intersection is a node, and each road is an edge, edge weight is the quantity（time or distance）we focus on. Than all the intersection can be

divided into following types: three fork, four fork and five fork (most of them shall be three of four fork).Suppose that number of three fork, four fork and five fork is n1, n2 and n3, In addition suppose that the map is big enough which means there is road to suburb from the nearest fork. Even though these roads maybe just a few, but we still count them in, Therefore we have below calculation:

$$m = \frac{3n_1 + 4n_2 + 5n_3}{2}$$

$$\frac{m}{n} = \frac{3n_1 + 4n_2 + 5n_3}{2n_1 + 2n_2 + 2n_3} = \frac{3}{2} + \frac{\frac{1}{2}n_2 + n_3}{n_1 + n_2 + n_3}$$

Because n_1, n_2, n_3 are nature numbers (including zero), so

$$\frac{1}{2}n_2 + n_3 \leq n_1 + n_2 + n_3$$

$$\frac{m}{n} = \frac{3}{2} + \frac{\frac{1}{2}n_2 + n_3}{n_1 + n_2 + n_3} \leq 2.5$$

The experimental data shows that, when n=10000, m=50000, SPFA-2 has the highest speed, and all the four algorithms consumes same running time. But what the results will be with a much sparser graph? What if m/n=2.5?

To answer above questions, we add another group of data with n=10000， m=25000, here is the calculation result:

	Dijkstra	SPFA-1	SPFA-2	SPFA-3
m=25000	207ms	203ms	172ms	198ms

(same as above data)

Based on all the results, it can be concluded that, in a sparse such as urban street map, SPFA has a large advantage in running time, even for SPFA-3, which cost time in edge set group sorting(because there is less edge in sparse graph, so less time is used in preprocessing). It can also be concluded that SPFA will perform much better in a large scale graph such as country traffic map.

Taking all the data into account, the author define sparse graph as a graph with m/n≤ 2.5. It has been certified that in the sparse graph a SPFA will be faster under the densest situation. In addition, m/n≤2.5 equals to dv=d/n≤5. All these information lead us to the definition of sparse graph.

With a clear description of sparse, we have following suggestion:

In the application aspect:

It would be better use Dijkstra algorithm to solve sing-source shortest paths problem in density graph.

SPFA-1，SPFA-2，SPFA-3 play almost same role in solving single-source shortest-paths problem in sparse graph. However SPFA-2 would be a better choice when the graph has a large scale of edges which result from a large scale of nodes. In addition, SPFA-1 is much reliable when high efficiency is required.

Conclusion:

According to all the analysis of the comparisons between Dijkstra and SPFA, it can be seen that different algorithm is suitable for different graph, and the reliability of Dijkstra and the efficiency of SPAF are the factors should be taken into account. Between the two most useful methods (Dijkstra and SPFA) in solving sing-source path problem, people tend to select SPFA in computer information competition considering the complexity of writing program. The author was quite curiosity about the k in SPFA with a efficiency of O(kn). In this paper, the author discussing the problem and testing the data, which lead him to a satisfy result. In program writing, Dijkstra will be much complex and much difficult in recording data during optimizing with heap (See four programs in the attachment), but much reliable than SPFA. However, the author would like to know if there is a better mathematics formula to describe the density of a graph. In this paper, it is simply described by m/n and dv=d/n, which also helpful in arriving the research result. Due to the limited of data testing and data analyzing, this paper hasn't completely discuss the issue. The author believes that in the college there will be more and more opportunities to finish this subject. Thanks to this paper, the author learns the importance of the attitude toward the experiments and how to design a control experiment, as well as how to deeply analysis a familiar algorithm and how to write a paper. All of those experiences will be a treasury for the author on his way to a success of science!

Reference:

Introduction to Algorithms wrote by Cormen,T.H.

Attachment:

Dijkstra + optimized with heap

```
#include<fstream>

#include<vector>
```

```
using namespace std;

const int maxn = 100001;
vector<int> vec[maxn],graph[maxn];
int heap[maxn] , dis[maxn], d_to_h[maxn] ,h_to_d[maxn];
int len;

void swap(int& a, int& b){
      int t;
      t=a; a=b; b=t;
    }

void heap_up(int x){
      if (x == 1) return;
      int k;
      do{  k = (x >> 1);
          if (heap[k] > heap[x]){
                    swap(heap[k],heap[x]);
                    swap(d_to_h[h_to_d[k]],d_to_h[h_to_d[x]]);
                    swap(h_to_d[k],h_to_d[x]);
                  }
          else return;
          x = k;
      }while (x > 1);
      return;
    }

void heap_down(int x){
      if ((x << 1)>len) return;
      int k;
```

```
      do{  k = (x << 1);
           if (k<len && heap[k]>heap[k+1]) ++k;
           if (heap[k]<heap[x]){
                 swap(heap[k],heap[x]);
                 swap(d_to_h[h_to_d[k]],d_to_h[h_to_d[x]]);
                  swap(h_to_d[k],h_to_d[x]);
                }
           else return;
           x = k;
      } while ((x << 1)<=len);
      return;
    }

int n,m,dx,dy,val;
void init(){
      ifstream infile;  infile.open("data.in");
      infile >> n >> m ;
      for (int i=1; i<=m; ++i){
         infile >> dx >> dy >> val;
         graph[dx].push_back(dy);
         vec[dx].push_back(val);
         graph[dy].push_back(dx);
         vec[dy].push_back(val);
        }
      infile.close();
    }

const int inf=100000000;
int s,tmp;
bool flag[maxn];
```

```
int re=1, rs=n;


void dijistra(){
      len = 0;
      for (int i=1; i<=n; ++i){
        ++len;
        if (i == re) heap[i]=0;
          else heap[i]=inf;
        h_to_d[i] = i;
        d_to_h[i] = i;
        heap_up(len);
        flag[i] = true;
      }
      for (int i=1; i<=n; ++i){
        s = h_to_d[1];
        dis[s] = heap[1];
        flag[s] = false;


        heap[1] = heap[len];
        d_to_h[h_to_d[len]] = 1;
        h_to_d[1] = h_to_d[len];
        --len;
        heap_down(1);


        for (int j=0; j<graph[s].size(); ++j){
          tmp = graph[s][j]; val = vec[s][j];
          if (flag[tmp])
            if ((dis[s]+val)<heap[d_to_h[tmp]]){
                heap[d_to_h[tmp]]= dis[s]+val;
                heap_up(d_to_h[tmp]);
```

```
                }
            }
          }
        }

void outit(){
      ofstream outfile; outfile.open("data.out");
      outfile << dis[rs];
      outfile.close();
    }
int main(){
  init();
  dijistra();
  outit();
  return 0;
}
```

SPFA-1

```
#include<vector>
#include<queue>
#include<cstring>
#include<string>
#include<fstream>
using namespace std;

const int maxn=100001, inf=1000000000;
vector<int>  map[maxn],val[maxn];
int dis[maxn];
bool flag[maxn];
queue<int> q;
int n,m,x,y,v;
```

```
void init(){
    ifstream infile;  infile.open("data.in");
    infile >> n >> m ;
    for (int i=1; i<=m; ++i){
          infile >> x >> y >> v;
          map[x].push_back(y);
          val[x].push_back(v);
          map[y].push_back(x);
          val[y].push_back(v);
       }
    infile.close();
    return;
    }

int re=1,rs=n;

void spfa(){
    for (int i=1; i<=n; ++i){
         dis[i] = inf; flag[i] = false;
       }
    int s,t,tmp;
    s=re;
    dis[s] = 0; flag[s] = true;
    q.push(s);
    while (!q.empty()){
           s=q.front();
           q.pop();
           flag[s]=false;
           for (int i=0; i<map[s].size(); ++i){
```

```
                t = map[s][i] ; tmp= dis[s]+ val[s][i];
                if (tmp < dis[t]){
                        dis[t] = tmp;
                        if (! flag[t]){
                            q.push(t);
                            flag[t]= true;
                           }
                       }
              }
          }
    return;
    }

void outit(){
    ofstream outfile; outfile.open("data.out");
    outfile << dis[rs];
    outfile.close();
    return;
    }

int main(){
  init();
  spfa();
  outit();
  return 0;
}
```

SPFA-2

```
#include<vector>
#include<deque>
#include<cstring>
```

```
#include<string>
#include<fstream>
using namespace std;

const int maxn=10000, inf=1000000000;
vector<int>  map[maxn],val[maxn];
int dis[maxn];
bool flag[maxn];
deque<int> q;
int n,m,x,y,v;

void init(){
    ifstream infile;  infile.open("data.in");
    infile >> n >> m ;
    for (int i=1; i<=m; ++i){
          infile >> x >> y >> v;
          map[x].push_back(y);
          val[x].push_back(v);
          map[y].push_back(x);
          val[y].push_back(v);
       }
    infile.close();
    return;
    }

int re=1,rs=n;
int s,t,tmp;

void inq(int x){
      flag[x] = true;
```

```
    if (q.empty()){  q.push_back(x);
                  return;
                 }
    tmp = q.front();
    if (dis[x]<dis[tmp])
      q.push_front(x);
    else
      q.push_back(x);
  }

void spfa(){
  for (int i=1; i<=n; ++i){
       dis[i] = inf; flag[i] = false;
     }
  s=re;
  dis[s] = 0; flag[s] = true;
  q.push_back(s);
  while (!q.empty()){
        s=q.front();
        q.pop_front();
        flag[s]=false;
        for (int i=0; i<map[s].size(); ++i){
            t = map[s][i] ; tmp= dis[s]+ val[s][i];
            if (tmp < dis[t]){
                    dis[t] = tmp;
                    if (! flag[t]) inq(t);
                  }
          }
       }
  return;
```

```
    }

void outit(){
    ofstream outfile; outfile.open("data.out");
    outfile << dis[rs];
    outfile.close();
    return;
    }

int main(){
  init();
  spfa();
  outit();
  return 0;
}
```

SPFA-3

```
#include<fstream>
#include<map>
#include<deque>
using namespace std;

const int maxn=100001;
int n,m,dx,dy,val;
int dis[maxn];
multimap<int,int> graph[maxn];
bool flag[maxn];

void init(){
        ifstream infile; infile.open("data.in");
        infile >> n >> m;
```

```
        for (int i=1; i<=m; ++i){
               infile >> dx >> dy >>val;
               graph[dx].insert(make_pair(val,dy));
               graph[dy].insert(make_pair(val,dx));
           }
        infile.close();
        return;
    }

const int inf=100000000;
deque<int> q;
int s,t,tmp;
int re=1, rs=n;

void inq(int x){
      flag[x] = true;
      if (q.empty()){  q.push_back(x);
                    return;
                  }
      tmp = q.front();
      if (dis[x]<dis[tmp])
        q.push_front(x);
      else
        q.push_back(x);
    }

void spfa(){
        for (int i=1; i<=n; ++i) {  dis[i] = inf;
                                  flag[i] = false;
                                }
```

```
        s=re; dis[s]=0; flag[s] = true;
        q.push_back(s);
        while (!q.empty()){
                s = q.front();
                q.pop_front();
                flag[s] = false;
                for (map<int,int>::iterator iter= graph[s].begin();
iter != graph[s].end(); ++iter){
                   t= iter->second;  val= iter->first;
                   if ((dis[s]+val)<dis[t]){  dis[t] = dis[s]+val;
                                          if (!flag[t]) inq(t);
                                        }
                  }
             }
        return;
    }

void outit(){
        ofstream outfile; outfile.open("data.out");
        outfile << dis[rs];
        outfile.close();
        return;
    }
int main(){
  init();
  spfa();
  outit();
  return 0;
}
```