SPLZ: An Efficient Algorithm for Single Source Shortest Path Problem Using Compression Method

Jingwei Sun · Guangzhong Sun

Abstract Efficient solution of the single source shortest path (SSSP) problem on road networks is an important requirement for numerous real-world applications. This paper introduces an algorithm for the SSSP problem using compression method. Owning to precomputing and storing all-pairs shortest path (APSP), the process of solving SSSP problem is a simple lookup of a little data from precomputed APSP and decompression. APSP without compression needs at least 1TB memory for a road network with one million vertices. Our algorithm can compress such an APSP into several GB, and ensure a good performance of decompression. In our experiment on road network of Northwest USA (about 1.2 millions vertices), our method can achieve about three orders of magnitude faster than Dijkstra algorithm based on binary heap.

Keywords Shortest Path · Compression · Road network

Jingwei Sun
Department of Computer Science and Technology, University of Science and Technology of China
E-mail: sunjw@mail.ustc.edu.cn

Guangzhong Sun
Department of Computer Science and Technology, University of Science and Technology of China
E-mail: gzsun@ustc.edu.cn
1 Introduction

The single source shortest path (SSSP) problem is a classic algorithm problem, and is also a model for numerous real-world applications, such as navigation, facilities location, logistics planning. Generally, given a graph $G = (V,E)$, and a source vertex $s$, the goal of SSSP problem is to find the shortest paths from $s$ to all other vertices in the graph.

Effective precomputation plays an important role of many efficient algorithms of SSSP problem. This kind of algorithms contains two phases: precomputing some supporting information offline, and computing the final results online. A straightforward precomputation method is precomputing all-pair shortest path (APSP) and storing it in memory. Then the time complexity of SSSP problem online is $O(\|V\|)$ for a simple lookup and output. However, space consumption for raw data of APSP needs at least $O(\|V\|^2)$. It’s impractical to run this algorithm on an ordinary machine for a large-scale graph. For example, a graph with one million vertices needs about 1 TB memory.

In this paper, we develop a compression method to reduce the space cost of precomputation effectively, and ensure a linear time complexity for decompressing at the same time. We call it shortest path with Lempel Ziv (SPLZ), which means it is a modification of LZ77 algorithm.

In our experiment on road network of Northwest USA (about 1.2 million vertices), SPLZ can compress the APSP of this graph (about 1.4 TB) into several GB. It is affordable for a high-end PC or an ordinary workstation. If memory is insufficient, we can store the compressed APSP into extended memory. Our experiments show that using extended memory can still achieve a good performance of decompressing.

There are three main contributions of our paper:

- We design an effective compression scheme for storing APSP data. With this method, we can take full advantages of information generated by precomputation. When memory isn’t enough, we can store the compressed APSP into extended memory and still keep a good performance of decompression.

- We develop a fast algorithm named SPLZ to solve SSSP problem. Our algorithm on single core can achieve about three orders of magnitude faster than Dijkstra algorithm based on binary heap. This performance is equivalent to PHAST on GPU, which is the state-of-the-art algorithm for single source shortest path problem.

- SPLZ is simple to be implemented. SPLZ don’t use complex data structures and elaborate skills. In the offline phase, SPLZ uses Dijkstra and LZ77 with a little modification. In the online phase, all the operations of SPLZ for solving SSSP is copying an array of length $O(\|V\|)$.

The remainder of this paper is organized as follows: Section 2 describes related works. Section 3 introduces the basic idea of SPLZ. Section 4 details the implement of SPLZ. Section 5 reports the experimental results. Conclusion is made in Section 6.
2 Related works

SSSP problem has been widely researched. Dijkstra\cite{16} algorithm is the most classic method for SSSP problem. To improve the performance of Dijkstra, researchers have adopted numerous type of priority queue. It has a series of modification like DIKB\cite{15}, DIKBD\cite{10}, DIKH\cite{12}, DIKR\cite{2}. They are also called label setting algorithm. Another classic algorithm for SSSP problem is Bellman-Ford\cite{6}. It is classified to another type of algorithm called label correcting algorithm. Besides Bellman-Ford, label correcting algorithm includes many others like PAPE\cite{25}, TWO-Q\cite{24}, THRESH\cite{19}, SLF\cite{7}. They are based on different label correcting strategies. Both label setting algorithm and label correcting algorithm can be described by a unified framework\cite{17}.

Some algorithms accelerate solving the SSSP problem with parallelism. The classic parallel algorithms for SSSP problem contain parallel asynchronous label correcting method\cite{8} and \(\Delta\)-stepping\cite{23}. Goldberg et al\cite{11} pointed out that traditional parallel algorithm for SSSP problem usually do not take full advantages of modern CPU architecture, like multi-core, SSE. \(\Delta\)-stepping has less acceleration on large-scale road network\cite{22}. In 2011, Delling and Goldberg et al developed PHAST\cite{13}, which is the fastest algorithm at present. PHAST makes full use of SSE, multi-core, and is elaborately designed to obtain a low cache missing. Its performance of GPU modification on large scale road network is up to three orders of magnitude faster than Dijkstra on a high-end CPU.

Precomputing methods replace the time consumption online with time and space consumption offline. It’s a efficient approach for solving shortest path problem on large-scale graph. Many high performance algorithm for point-to-point shortest path problem have a precomputing process, including Highway Hierarchies\cite{26}, Transit Node Routing\cite{4}, ALT\cite{20} et al. In 2008, Geisberger proposed Contraction Hierarchies\cite{18} algorithm. It’s not only a good algorithm for solving the answer of shortest path, but also a efficient method for precomputing. Many outstanding algorithm like Transit Node Routing\cite{3}, PHAST and Hub-based labeling\cite{1} adopted it as precomputing method.

Some algorithms have good performance online, but space consumption of precomputing is too huge to be loaded in memory. Compression methods are practical way to reduce the space consumption. SILC\cite{27}, PCPD\cite{28}, and CPD\cite{9} use compression method to reduce space complexity of storing APSP. These methods are to solve point to point shortest path problem on spatial networks (graphs where each node is labeled with coordinates). They store APSP in the form of first move table. First move table can obtain high compression power by taking the advantage of path coherence\cite{27}. 
3 Basic SPLZ

3.1 Main idea

The main idea of SPLZ is to precompute APSP and then compress it for online lookup. In SPLZ, APSP is stored in the form of shortest path tree (SPT). Here SPT is an array of length $|V|$, which records the last move of the shortest path from a source vertex to all other vertices. For example, if we want to calculate the shortest path from vertex $v$ to other vertices, we will present the result by $SPT(v)$, which is a tree with root $v$. If $SPT(v)[u] = w$, it means that the predecessor of vertex $u$ along the shortest path from $v$ to $u$ is vertex $w$. Edge $(w, u)$ is the last move from $v$ to $u$, so SPT is also called last move table. Traditional algorithm for SSSP problem like Dijkstra usually present the result in the form of SPT too. If we store APSP straightforwardly, the space complexity is $O(|V|^2)$, for we need to store $|V|$ SPTs.

Path coherence described in [27] reveals that vertices contained in a coherent region share the first segment of their shortest path from a fixed vertex. For last move, path coherence still makes sense with a bit change on description: vertices contained in a coherent region share the last segment of their shortest path to a fixed vertex. Path coherence implies that the data among multiple SPTs contain a large number of reduplicative sequence. This feature will lead to a high compression ratio with LZ-family algorithm.

SILC [27], PCPD [28], and CPD [9] adopt first move table. First move table has the similar feature like last move table, but it is suitable to solving point-to-point shortest path problem. When solving SSSP problem, first move table will result in frequently looking up compressed data. So SPLZ adopts last move table as the form of storage of result.

Generally, SPLZ contains 3 parts: (1) calculating the APSP, (2) compressing the APSP offline, (3) decompressing the APSP online. We adopt Dijkstra algorithm to calculating the APSP. Other existing methods, like Bellman-Ford-Moore, Floyd-Warshall, are also feasible. Now we focus on the compressing and decompressing method of SPLZ, which is a variant of LZ77.

3.2 LZ77 algorithm

Let $data$ be a string to be compressed, and $i$ bytes has been compressed. The compression procedure of LZ77 is as follows:

**Algorithm 1** The compression procedure of LZ77

1: while ($i \leq \text{len}(data)$) do
2: dict=data[i-DICT_SIZE..i];
3: (location, length) = longest_match(substring of dict, prefix of data[i..end]);
4: output (location, length);
5: i=i+length;
6: end while
Dict is a subsequence of data which slides with i increases. Longest_match looks for the longest common subsequence between dict and the prefix of uncompressed data, and then returns location and length of the common subsequence. Finally the compressed data is an array of (location, length) pairs.

Let packed_data be an array of two-tuples to be decompressed, and assume i bytes has been decompressed. The decompression procedure of LZ77 is as follows:

Algorithm 2 The decompression procedure of LZ77
1: for all (location, length) in packed_data do
2:   dict=unpacked_data[i-DICT_SIZE..i];
3:   output dict[location..location+length];
4:   i=i+length;
5: end for

Decompression speed of LZ family algorithm is fast, but decompressing some particular data from compressed data is dependent on previous data because of the slide of dictionary. Assuming that there are compressed data contains n SPTs, from which we hope to decompress a particular SPT, we need decompress $\frac{n}{2}$ SPTs on average before the specified SPT. This process results in a large amount of redundant operation.

3.3 Fixed-dictionary compression

To avoid redundant operations, we fix the dictionary. This means the dictionary is a fixed-size and fixed-location sequence in the front of the raw data. Data in the dictionary will not be compressed, to achieve a faster decompressing speed. These modification result in a lower compression ratio. We should point out that, SPLZ is not a compression algorithm for general situation, but an algorithm for solving SSSP problem.

Let data be a set of SPT, and data[0], the first SPT in data, be the dictionary. Index[i] is the starting position of i-th SPT in compressed data. That is to say, index[i + 1] − index[i] is equal to the compressed length of i-th SPT. Let len(packed_data) be the total size of compressed data. Algorithm 3 and Algorithm 4 separately describe the process of compressing data and decompressing the m-th SPT. There are many methods for finding the longest match [5], we use a simple implement with a modification of KMP [21].

The first line of Algorithm 3 is just an assignment to a pointer, without copying any real data. Array index is used for locating a specified SPT in compressed data. So all the operation of decompressing a specified SPT is copying data which is of length n, and time complexity O(n).
Algorithm 3 The compression procedure of SPLZ

1: dict=data[0];
2: output dict;
3: for all spt in data-data[0] do
4:    index[number of spt in data]=len(packed_data);
5:    while (spt!=NULL) do
6:       (location, length)=longest_match(substring of dict, prefix of spt);
7:       output (location, length);
8:       spt=spt-matched prefix of spt;
9:    end while
10: end for

Algorithm 4 The decompression procedure of SPLZ

1: dict=packed_data[0..sizeof(SPT)];
2: for i=index[m] to index[m+1] do
3:    (location, length)=packed_data[i];
4:    output dict[location..location+length];
5: end for

Fig. 1: Graph G

![Graph G](image)

### Table 1: The adjacency list of G

| Vertex | Adjacent vertices of $V_i$ |
|--------|-----------------------------|
| $V_0$  | $V_2$                       |
| $V_1$  | $V_2$                       |
| $V_2$  | $V_0$, $V_1$, $V_4$, $V_5$  |
| $V_3$  | $V_2$, $V_5$                |
| $V_4$  | $V_2$, $V_5$                |
| $V_5$  | $V_3$, $V_4$                |

3.4 An example of SPLZ compression and decompression

Assume that there is a graph G as Figure 1 shows. After running Dijkstra algorithm for each vertex, we get six SPTs shown in Table 2. If $SPT(V_i)[j] = k$, it means the precursory vertex of $V_j$ on the shortest path from $V_i$ to $V_j$ is $V_k$. 
Table 2: SPT of each vertex

| V_i   | SPT(V_0) | SPT(V_1) | SPT(V_2) | SPT(V_3) | SPT(V_4) | SPT(V_5) |
|-------|----------|----------|----------|----------|----------|----------|
|       | 0 0 0 0 0 0 | 0 0 1 0 0 0 | 0 0 0 0 0 0 | 0 0 2 0 0 0 | 0 0 3 0 0 0 | 0 0 2 1 1 0 |

Table 3: SPT of each vertex after converting

To obtain an effective compression, we convert Table 2 to another form. In Table 3, $SPT(V_i)[j] = k$ means the precursory vertex of $V_j$ on the shortest path from $V_i$ to $V_j$ is the $k$-th adjacent vertex of $V_j$. We define $SPT(V_i)[i] = 0$.

Then we select $SPT(V_2)$ as the dictionary. Every SPT is compressed into an array of two-tuples $(location, length)$. Note that sometimes a number in a SPT might not exist in the dictionary. For example, in Table 3 $SPT(V_1)[2]=1$, but "1" don’t exist in the dictionary. At this situation, we set $length=0$ and $location$=the number excluded in dictionary.

To make a simple illustration, here we assume that each two-tuples needs two bytes, therefore after compressing, the index array is: (0, 2, 8, 10, 16, 24, 34).

The total length of output is 40 bytes, for the length of dictionary is 6 bytes and the length of compressed data is 34 bytes. The effectiveness of compression seems poor, because the scale of graph in our example is too small.

When solving a SSSP problem online, for example, calculating the $SPT(V_1)$, the steps are:

1. Find out that $index[1] = 2$ and $index[2] - index[1] = 6$. In other words, the length of compressed $SPT(V_1)$ is 6 and its start location in whole compressed data is 2.
2. Convert every two-tuples to original data by looking up the dictionary. For example, when handling the two-tuples (0, 2), we intercept the subsequence of dictionary, which begins at 0 and is of length 2. All the conversion is: (0, 2) $\rightarrow$ 0 0; (1, 0) $\rightarrow$ 1; (0, 3) $\rightarrow$ 0 0 0.
3. Concatenate these subsequences to one array: (0 0 1 0 0 0). This array is $SPT(V_1)$.

4 Details of implement

4.1 The key factor affecting the compression ratio

When we use fixed-dictionary compression, the compression ratio is mainly decided by the similarity between the dictionary and data to be compressed. Similarity between two SPT has a negative correlation with path-len between the source vertex of the two SPT. Let $\text{path-len}(u,v)$ be the number of edges along the shortest path between vertex $u$ and vertex $v$. We use the proportion of common edges among two SPT to measure the similarity between two SPT. Figure 2 present an experiment result.

Experiment of Figure 2 is based on graph data USA-road-t.NW. The less path-len of two vertex $u$ and $v$, the higher similarity between $SPT(u)$ and $SPT(v)$. This result is reasonable in real-world. For example, assume there are three location A, B, and C. Both the distance of (A, C) and (B, C) is 10 km. If the distance between A and B is one meter, we could guess that the shortest path (A, C) and (B, C) are almost the same. When we choose $SPT(u)$ as dictionary and compress $SPT(v)$, the impact of $\text{path-len}(u,v)$ on the compression ratio is as Figure 3 shows. The result points out that the compression ratio decreases fast with increasing path-len. To reduce the space consumption, it is necessary to limit the path-len between the dictionary and the SPT to be compressed.

4.2 Regions partition

If we choose only one SPT, which is of source vertex $u$, as the dictionary in a large scale graph, there always are some vertices far away from $u$. By partitioning the graph into a series of regions with smaller size, we can choose...
We choose the SPT of a vertex which is closest to the geometrical center as the dictionary. Partition ensures that in a region, the path-len between vertex of dictionary and other vertices don’t exceed the diameter of the region.

When partitioning the graph into numerous disjoint regions, we should keep the distance of vertices among a region as close as possible. It can be handled as a clustering problem. We use k-means, a simple but effective clustering method, to partition the graph. The simplest attribute for clustering vertices in a road network is coordinate, and we adopt it. Actually, there are numerous methods to partition the graph without coordinate. Coordinate is an unessential condition for SPLZ.

Due to that data in the dictionary will not be compressed and output in raw form, if the number of regions is excessive, uncompressed data would occupy a large proportion in final output. If the number of regions is small, we cannot ensure that the diameter of a region is significantly less than the diameter of the total graph.

It is difficult to analyze the exact relation between number of regions and the final compression ratio. Intuitively, to reach both small number of regions and small size of every region, we assume that optimal number of regions is in form of $C \times \sqrt{|V|}$, and choose a proper value of $C$ by experiment.

4.3 Multi-step compression

Assume that vertices of SPT($u$) among a particular region is as Figure 4 shows. In region showed in Figure 4, all SPT except $SPT(u)$ is compressed with dictionary $SPT(u)$. We call this process one-step compression. Path-len between most SPT in this region and $SPT(u)$ is 3 or 4.

We can reduce the path-len by multi-step compression. For example, let the grandparent of every SPT be its dictionary. As for vertex $u_4$, $SPT(u_4)$ will be compressed with dictionary $SPT(u_2)$ and $SPT(u_2)$ will be compressed.
Fig. 4: Part of SPT(u) within a region

with dictionary $SPT(u)$. When we decompress $SPT(u_4)$, we must decompress $SPT(u_2)$ at first. It is so-called two-step compression for vertex $u_4$. By applying similar operation to all vertex, their path-len is decreased to 1 or 2.

We call the path-len between a SPT and its dictionary len-to-dic. Figure 3 tell us that shorter path-length leads to a higher compression ratio. But the reduction of space costs brings higher time cost. If $len - to - dic$ is $d$, the time to decompress $SPT(u_4)$ is $\lceil \text{path-len}(u, u_4)/d \rceil$ times of one-step compression. By controlling $len - to - dic$, we can adjust the point of balance between space costs of compression and time costs of decompression online.

We define $len - to - dic = \infty$ when we use one-step compression.

4.4 Code of compressed data

The compressed data are array of $(location, length)$. If location and length are fixed-length integer, data compressed by method in Section 3.3 can be compressed one more time by entropy coding, but entropy coding has poor performance on decompression. We adopt a variable length coding to encode location and length. Though our coding method cannot obtain the compression ratio as high as entropy coding, it has almost no negative effect on decompression speed. The code also is prefix coding, so there is no ambiguity when we decode it.

In the data stream of compressed data, location is presented by differential coding. We just record the difference between every location and its predecessor except the first one, because difference between two adjacent location usually smaller than their real value. Differential coding might result in a shorter code. Value of length doesn’t have such a feature, so we record its real value.

Encoding method for length and location in detail is separately in Table 4 and Table 5. In the first line of Table 4, "uncompressed" means that some
bytes don’t appear in the dictionary, so these bytes cannot be compressed. In our method, the value of such a byte must be no more than 15. It is reasonable for a real-world road network, for number of branch of real-world road usually smaller than 15. In the case of that degree of a vertex \( u \) is more than 15, we can add a virtual vertex \( u' \) to the graph. Let the distance between \( u \) and \( u' \) be zero and assign excess edges of \( u \) to \( u' \).

5 Experiment

5.1 Environment

The code is written in C++, and compiled by VC++ 2010. The program includes two parts: precomputing offline and calculating the SSSP online. The experiments run on a PC, with 3.4 GHz Intel i7-4770(4 cores), 24GB RAM and 2TB hard disk. For parallelly precomputing, we use OpenMP. Data of graph are downloaded from [http://www.dis.uniroma1.it/~challenge9](http://www.dis.uniroma1.it/~challenge9), which are benchmarks for the 9th DIMACS Implementation Challenge[14]. The data set we used is "Northwest USA", with 1207945 vertices and 2840208 edges, and the type of graph is "Distance graph". The total size of the APSP of "Northwest USA" is about 1359 GB. In all of our experiment, the unit of data size is GB.
The source code of our experiments is released\(^1\). Everyone can download and share it.

5.2 Precomputing

Operation of precomputing consist of computing the APSP and compressing it. The target of compressing is to reduce the space consumption of APSP. So the compression ratio is a important feature for measuring the effectiveness of precomputing. The number of regions has impact on compression ratio. We choose different setting of parameter \(C\) for \(C \times \sqrt{|V|}\) as the number of regions separately. For every parameter setting, the running time of precomputing is about 19 hours. Table 6 shows the effect of number of regions on the compression ratio.

The number of regions determine the average size of each region, and the size of a region has effect on the path-len between vertices in a region. The less the path-len between vertices, the higher the compression ratio. So it seems that more number of regions may lead to higher compression ratio. However, Table 6 demonstrates that when the number of regions is more than a certain value, the compression ratio falls down. It is owing to that, with the number of regions increases, the proportion of dictionary increases. We select a vertex as the representative vertex for each region. The SPT of the representative vertex is the dictionary of that region. To ensure that the dictionary is available at the immediate time of decompressing, dictionary will not be compressed. Although more number of regions leads to a higher compression ratio of single SPT, the total size of final data will increase because the increased size of dictionaries.

If the dictionaries occupy a high proportion of the final output, we can consider to compress the dictionaries. But it will result in two problems. One is the extra time cost for decompressing dictionary when decompressing data. Another one is, actually, it is difficult to find a proper "dictionary" for compressing dictionaries, which intrinsically have less data-redundancy. In other words, the compression ratio of compressing the dictionaries is much lower than compressing SPTs.

We set \(1 \times \sqrt{|V|}\) as the number of regions for following experiment.

5.3 Multi-step compression

Table 6 shows the results of one-step compressing with different number of regions. Multi-step compressing can obtain a higher compression ratio as shown in Figure 5. SPLZ can adjust the compression ratio by controlling the parameter \(len - to - dic\), which was described in Section 4.3. This capability of SPLZ makes it adaptable to different capacity of memory. With the \(len - to - dic\) decreases, the compression ratio increases. It is due to the similarity between the SPT to be compressed and its dictionary is higher when the parameter

\(^1\) https://github.com/asds25810/SPLZ
Table 6: Effect of number of regions on compression ratio

| C   | Raw size | Compressed size | Compression ratio | Dictionary size | Proportion of dictionary |
|-----|----------|-----------------|-------------------|----------------|-------------------------|
| 0.5 | 1359     | 15.5            | 87                | 0.62           | 4.0%                    |
| 1   | 1359     | 12.3            | 110               | 1.24           | 10.1%                   |
| 2   | 1359     | 10.9            | 125               | 2.47           | 22.7%                   |
| 4   | 1359     | 10.5            | 129               | 4.94           | 47.1%                   |
| 8   | 1359     | 14.3            | 95                | 9.89           | 69.2%                   |

Fig. 5: Effect of Len-to-dic on compression ratio

len \( \text{to} \) dic decreasing. SPLZ achieves the highest compression ratio when \( \text{len} \text{to} \text{dic} = 1 \). The APSP of size about 1359 GB can be compressed to 2.67 GB. It is affordable for an ordinary PC.

5.4 Online performance

After precomputing, we test the time costs of solving the SSSP problem for a particular vertex. We load the compressed APSP in memory, and randomly generate a series of queries. Each query input a vertex \( v \), and request \( SPT(v) \) as output. Table 7 shows the average costs of handling a query. The parameter len-to-dic evidently affects the performance. Higher len-to-dic makes a lower space consumption. However, the reduced space consumption is repayed by increasing time costs. Len-to-dic should be decided by the bottleneck of different applications.

If the memory capacity is not enough to store the compressed APSP, we can consider place it to hard disk. When SPLZ handles a query, it looks up the compressed SPT from hard disk, and then decompresses it and returns the result. Hard disk is cheap and has high capacity. Although the latency of access
Table 7: Time and space cost online

| Methods  | Len-to-dic | Time (us) | Space(RAM) | Space(disk) |
|----------|------------|-----------|-------------|-------------|
| SPLZ on memory | ∞      | 172       | 12.38       | -           |
|          | 16        | 271       | 8.72        | -           |
|          | 8         | 414       | 5.89        | -           |
|          | 4         | 715       | 4.13        | -           |
|          | 2         | 1199      | 3.17        | -           |
|          | 1         | 2096      | 2.67        | -           |
| SPLZ on disk | ∞      | 176       | 1.24        | 11.14       |
|          | 16        | 277       | 1.24        | 7.48        |
|          | 8         | 424       | 1.24        | 4.65        |
|          | 4         | 726       | 1.24        | 2.89        |
|          | 2         | 1226      | 1.24        | 1.93        |
|          | 1         | 2141      | 1.24        | 1.43        |
| Dijkstra | -         | 217291    | -           | -           |

is significantly higher than that of memory, our experiment shows that hard disk has a satisfactory performance after handling enough number of queries. The time costs on disk in Table 7 is the stable performance after handling a large number of random queries. It has little difference to the performance on memory.

Figure 6 reveals more detail of the process how the performance of SPLZ on hard disk approximates the performance of SPLZ on memory. We tested 200,000 random queries using SPLZ on hard disk, and record the minimal, average and maximal time costs of different range. The overall trend in Figure 6 is that, with the number of queries handled increases, the average time cost for handling one query drop off. In each statistical interval, the maximal cost is significantly higher than the minimal cost, while the minimal value stay a small constant value. In fact, the increasing performance of SPLZ on hard disk is due to cache and prefetch of system and hardware. If SPLZ access data which have been cached or prefetched, the time cost is close to the minimal costs. Otherwise SPLZ needs to read data from hard disk, and the time costs is as high as the maximal costs. The average cost reveals the hit rate.

In real-world application, there may not be enough number of queries. We can artificially generate some queries for "warming up", to keep a good performance of SPLZ on hard disk.

The average time costs of Dijkstra algorithm based on binary heap is about 217 ms on our experiment graph. An experiment in [13] shows that, without parallel, PHAST achieves about 30 times faster than Dijkstra based on binary heap. In our experiment, the performance of SPLZ is almost three orders of magnitude faster than Dijkstra based on binary heap, if let $\text{len-to-dic}=\infty$. 
Fig. 6: Relation between time costs of decompression from disk and times of queries

| Methods                  | Time costs(us) |
|--------------------------|----------------|
| SPLZ                     | 169            |
| memcpy                   | 81             |
| for-loop assignment      | 428            |

Table 8: Time costs of solving a SSSP problem by SPLZ and copying an array of length $\|V\|$.

Although SPLZ need about 12 GB space costs if $len – to – dic=\infty$, storing the compressed APSP into disk would solve this problem.

5.5 Lower bound of SSSP problem

In fact, the time costs of solving SSSP problem has a natural lower bound. Whatever methods we use to calculate the shortest path from a vertex to all vertices in a graph, finally we must fill the result to an array of length $\|V\|$ as output. So the natural lower bound is the time costs of copying an array of length $\|V\|$. We compares the time costs of SPLZ and array copying in Table 8.

We test two type of method to copying array. Memcpy() is a standard function in C library. Considering many algorithm successively output their result in a loop, we also test copying an array by for-loop(assigning the elements one by one). The results in 8 shows that the performance of SPLZ is close to the lower bound.
6 Conclusion

In this paper, we presented SPLZ, an algorithm for solving single source shortest path problem on road network. SPLZ is about three orders of magnitude faster than Dijkstra based on binary heap. Such performance is equivalent to the performance of PHAST\[13\] on GPU. Compared with the time costs of array copying, which is a natural lower bound of SSSP problem, SPLZ shows a significantly high performance online. Even though SPLZ consumes more memory yet, this problem can be solved by storing compressed data into hard disk or by adjusting the parameter len-to-dic.

Future research will focus on developing a more efficient preprocess method. In our experiments, SPLZ can solve SSSP problem on a road network with about 1.2 millions vertices, and it is enough for many application. But we should admit that, SPLZ still can’t deal with a more large-scale road network (for example, the USA road network, with about 24 millions vertices and 58 millions edges), because of the huge time costs for precomputing. We can make efforts to two aspects. One is to adopt an algorithm better than Dijkstra to calculate APSP, the other is to use a more efficient methods to find the longest match while compressing the APSP.

Acknowledgement

This work is supported by Natural Science Foundation of China (No. 61033009 and No. 61303047) and Anhui Provincial Natural Science Foundation (No. 1208085QF106).

References

1. Abraham I, Delling D, Goldberg AV, Werneck RF (2011) A hub-based labeling algorithm for shortest paths in road networks. In: Experimental Algorithms, Springer, pp 230–241
2. Ahuja RK, Mehlhorn K, Orlin J, Tarjan RE (1990) Faster algorithms for the shortest path problem. Journal of the ACM (JACM) 37(2):213–223
3. Arz J, Luxen D, Sanders P (2013) Transit node routing reconsidered. In: Experimental Algorithms, Springer, pp 55–66
4. Bast H, Funke S, Sanders P, Schultes D (2007) Fast routing in road networks with transit nodes. Science 316(5824):566–566
5. Bell T, Kulp D (1993) Longest-match string searching for ziv-lemel compression. Software: Practice and Experience 23(7):757–771
6. Bellman R (1956) On a routing problem. Tech. rep., DTIC Document
7. Bertsekas DP (1993) A simple and fast label correcting algorithm for shortest paths. Networks 23(8):703–709
8. Bertsekas DP, Guerriero F, Musmanno R (1996) Parallel asynchronous label-correcting methods for shortest paths. Journal of Optimization Theory and Applications 88(2):297–320
9. Botea A, Baier JA, Harabor D, Hernández C (2013) Moving target search with compressed path databases. Proceedings of ICAPS-13
10. Cherkassky BV, Goldberg AV, Radzik T (1996) Shortest paths algorithms: Theory and experimental evaluation. Mathematical programming 73(2):129–174
11. Cherkassky BV, Georgiadis L, Goldberg AV, Tarjan RE, Werneck RF (2009) Shortest-path feasibility algorithms: An experimental evaluation. Journal of Experimental Algorithmics (JEA) 14:7
12. Cormen TH, Leiserson CE, Rivest RL, Stein C, et al (2001) Introduction to algorithms, vol 2. MIT press Cambridge
13. Delling D, Goldberg AV, Nowatzyk A, Werneck RF (2013) Phast: Hardware-accelerated shortest path trees. Journal of Parallel and Distributed Computing 73(7):940–952
14. Demetrescu C, Goldberg AV, Johnson DS (2009) The Shortest Path Problem: Ninth DIMACS Implementation Challenge, vol 74. American Mathematical Soc.
15. Dial RB (1969) Algorithm 360: Shortest-path forest with topological ordering [h]. Communications of the ACM 12(11):632–633
16. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numerische mathematik 1(1):269–271
17. Gallo G, Pallottino S (1986) Shortest path methods: A unifying approach. Netflow at Pisa pp 38–64
18. Geisberger R, Sanders P, Schultes D, Delling D (2008) Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: Experimental Algorithms, Springer, pp 319–333
19. Glover F, Klingman D, Phillips N (1985) A new polynomially bounded shortest path algorithm. Operations Research 33(1):65–73
20. Goldberg AV, Harrelson C (2005) Computing the shortest path: A search meets graph theory. In: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, pp 156–165
21. Knuth DE, Morris JH Jr, Pratt VR (1977) Fast pattern matching in strings. SIAM journal on computing 6(2):323–350
22. Madduri K, Bader DA, Berry JW, Crobak JR (2006) Parallel shortest path algorithms for solving large-scale instances
23. Meyer U, Sanders P (2003) δ-stepping: a parallelizable shortest path algorithm. Journal of Algorithms 49(1):114–152
24. Pallottino S (1984) Shortest-path methods: Complexity, interrelations and new propositions. Networks 14(2):257–267
25. Pape U (1974) Implementation and efficiency of moore-algorithms for the shortest route problem. Mathematical Programming 7(1):212–222
26. Sanders P, Schultes D (2005) Highway hierarchies hasten exact shortest path queries. In: Algorithms–Esa 2005, Springer, pp 568–579
27. Sankaranarayanan J, Alborzi H, Samet H (2005) Efficient query processing on spatial networks. In: Proceedings of the 13th annual ACM international workshop on Geographic information systems, ACM, pp 200–209
28. Sankaranarayanan J, Samet H, Alborzi H (2009) Path oracles for spatial networks. Proceedings of the VLDB Endowment 2(1):1210–1221
29. Ziv J, Lempel A (1977) A universal algorithm for sequential data compression. IEEE Transactions on information theory 23(3):337–343