Implementation of Cache-Oblivious Priority Queue to Solve Single Source Shortest Path Problem

Sushil Shrestha[a]

[a] *sushilsh@hawaii.edu*

Keyword: Bucket Heap, Cache-Oblivious Data Structures, Single Source Shortest Path

Abstract: This research implements the cache-oblivious data structure and single source shortest path algorithm using the data structure. The paper also provides run-time performance comparison between single source shortest path algorithm using binary heap and cache-oblivious data structure.

1. Introduction

Most of classic algorithms does not take the I/O operations into consideration. In theory, they might have an optimal solution, but in practice they suffer from the I/O operations. Taking I/O operations into consideration helps algorithms to perform better because different memory have different speeds of data retrieval and ignoring the speed of data retrieval might adversely affect the algorithms in runtime. To solve that problem, algorithms that take an account of caching(or memory hierarchy) are preferred. The cache-oblivious data structure make optimal use of the cache hierarchy present in modern computer architecture and try to reduce the I/O operations to disk. The cache-oblivious data structures are not aware of the number of blocks or memory available, hence are more general. In this research, we will take a look at implementing a cache-oblivious priority queue and use it to solve the single source shortest path algorithm.

In this section, we will provide an overview of general types of algorithms used in practice. We will look into RAM model algorithms, external memory algorithms and cache-oblivious algorithms. In the next section, we will discuss about the cache-oblivious priority queue: bucket heap. Third section will deal with the implementation details and results of the experiments we conducted for this paper. Fourth section contains discussion and also includes future enhancements for the project.

1.1. RAM model algorithms

First type of algorithms that addresses the issue are the models that assumes cost of memory access is not significant. They assume all memory access are equal and only focus on time bound of the algorithms. They are also known as the RAM model. RAM model assumes all data can be loaded upfront to memory and cost to access the data is constant. Where as in practice, there are different memory hierarchy in computers and access to different levels of the memory hierarchy is significantly different. Also while working on large amount of data that doesn't fit in RAM, RAM model suffers from I/O operations while fetching and accessing the data. So we need algorithms that take these memory hierarchy into consideration.

1.2. External Memory / Cache-aware Algorithms

The memory located closer to CPU are significantly faster than the memory located further away from the CPU. In that sense, RAM model algorithms suffers significant degradation in performance during run-time since they don't consider the memory hierarchy. External Memory or Cache-aware algorithms are designed considering the memory hierarchy present in current computers. The external memory algorithms (I/O algorithms) are designed to optimize the fetch and access of data from the external memory. The external memory algorithms assumes standard two level memory hierarchy with block transfers. The two levels namely 1) limited cache memory that is near to CPU and cheap to access and 2) unlimited disk memory that is far from CPU and expensive to access. The cache has limited memory (M). The disk is divided into blocks of size B. Access to a single memory location in a block results in transfer of whole block B from disk to cache. The number of blocks that can fit in the cache is M/B. Generally it is assumed that $M \geq B^2$ which is also referred to as tall-cache assumption.

The main objective of the cache-aware memory algorithms is to reduce the block transfers from disk to cache. The paper by Vitter[1] presents a survey of external memory algorithms and data structures with analysis of their I/O costs. The paper also mentions significant performance increase of the external memory algorithms compared to other methods used in practice. However the cache-aware algorithms are depended upon M and B. And it also fails to capture multiple layers of memory hierarchy present in current machines.

1.3. Cache-Oblivious Algorithms

Cache-oblivious model is upgraded versions of external-memory algorithms that doesn't depend on values of B and M. This makes cache-oblivious algorithms perform well between any level of memory hierarchy with different values of B and M. One of the advantages of the cache oblivious method as mentioned in [2] paper is self-tuning of parameters. Cache-efficient algorithms requires to know about cache parameters which are not readily available making it difficult to extract automatically. Code portability is hard under parameter tuning. Cache-oblivious algorithm, since they are not depended on cache parameters have natural advantage and are designed to work well on all machines without modification. However cache-oblivious model are dependent on page replacement strategy. To deal with the issue, the cache-oblivious algorithms assumes an ideal cache with optimal page replacement strategy.

Demaine[2] provides an overview of cache-oblivious algorithms and the I/O analysis for each of the algorithms. The paper also presents static and dynamic data structures for cache-oblivious algorithms.

1.4. Single source shortest path

Single source shortest path is a well known problem in graph theory and is applicable for wide range of applications in different fields. Single source shortest path problem can be defined as: a graph is defined by $G(V, E)$ where V are the vertices and E are the edges, every edge E has a non-negative weight w assigned to it. The cost of using a path to reach from initial vertex u to destination vertex v is the sum of weights of edges used in the path. Our objective for the problem is to find the shortest path between vertices u and v that has minimum weight assigned to the path.

For this paper, we will consider single source shortest path for weighted connected simple graph with positive weights on the edges. General solution for the single source shortest path problem is presented in Algorithm 1.

Algorithm 1 SSSP(G, u, v)

```
1:  for i = 1 to n do
2:      dist[i] = ∞                          ▷ initialize distance of path to all vertex as ∞
3:  end for
4:  dist[u] = 0                              ▷ initialize distance of path to all initial vertex as 0
5:  S = φ
6:  while v ∉ S do                           ▷ loop until we reach the final vertex v
7:      u' = a vertex with minimum dist[u'] and ∉ S
8:      S = S ∪ {u'}
9:      for all vertices v' ∉ S do
10:         if dist[u'] + w(u', v') < dist[v'] then   ▷ update the distance if new distance is better than past
11:             dist[v'] = dist[v'] + w(u', v')
12:         end if
13:     end for
14: end while
15: return dist[v]
```

First of all we initialize distance of vertices($dist$) to infinity except for the starting vertex which is initialized to 0. Next we iterate over at most V steps, find lowest distance from $dist$ array, get its neighbours from adjacency list of the vertex and update the $dist$ array if we have better path with smaller weight. We can implement the algorithm using binary heap tree and linked list data structure. We can use distance as a key to store the vertices in a binary heap and use DELETEMIN and DECREASEKEY operations on the binary heap to maintain the minimum distance in the heap. And we can use a linked list data structure to store adjacency list of the vertices.

Kumar and Schwabe[3] purpose an external memory algorithm to address the single source shortest path problem using tournament tree as the data structure replacing the binary heap for searching the minimum

distance. The paper also shows the improved algorithm has I/O bound of $O(V + \frac{E}{B} \log_2 \frac{E}{B})$: $O(V + \frac{E}{B})$ to read the adjacency lists and $O(\frac{E}{B} \log_2 \frac{E}{B})$ for all the tournament tree operations.

The tournament tree purposed by Kumar and Schwabe[3] is not cache-oblivious and to make the problem cache-oblivious, Brodal and et al.[4] presents a cache-oblivious priority queue, bucket heap.

2. Bucket Heap

Bucket heap is a cache-oblivious priority queue that efficiently supports a weak DecreaseKey operation[4]. Bucket heap performs Insert and DecreaseKey operations using a single Update operation. Bucket heap supports Update, Delete, DeleteMin operations with $O((1/B) \log_2 N/B)$ amortized cost of operations. Single source shortest path algorithm using the bucket heap data structure achieves the performance similar to the cache-aware algorithm[3] with $O(V + (E/B) \log_2 E/B)$ I/O operations.

A bucket heap consists of a series of buckets $B_1, B_2, ..., B_q$ and signal buffers $S_1, S_2, ..., S_{q+1}$ where q is number of buckets. q varies over time but is always bounded by $\lceil \log_4 N \rceil$. The buckets and signals are stored consecutively in memory as: $S_1, B_1, S_2, B_2, ..., B_q, S_{q+1}$.

Each bucket (B_i) has capacity of 2^{2i+1} and each signal buffer has capacity of 2^{2i}. Each bucket store number of elements that has a priority and value. The items stored in bucket maintains heap property that for any two bucket B_i and B_j for $i < j$, every elements stored in B_i has smaller priority than any element stored in B_j.

Signal Buffers store three types of signals; Update(x, p), Delete(x) and Push(x, p). Update(x, p) signal inserts element x with priority p into the priority queue if x is not in the priority queue otherwise replace the current element x with minimum priority. Delete(x) signal removes the element x from the heap. Push(x, p) signal pushes elements from B_i bucket to B_{i+1} bucket when bucket B_i overflows.

The DeleteMin, Update, and Delete operations are implemented as follows:

Algorithm 2 DeleteMin

1: Fill(1)
2: remove item with lowest priority from B_1
3: **return** item with lowest priority

Algorithm 3 Update(x, p)

1: Insert Update signal to S_1 bucket with item x and priority p
2: Empty(S_1)

Algorithm 4 Delete(x)

1: Insert Delete signal to S_1 bucket with item x
2: Empty(S_1)

The DeleteMin operation uses the Fill operation that makes sure that current bucket is not empty by moving items from higher level buckets to current bucket. Fill(1) makes sure the lowest level bucket has items with minimum priority.

The Update and Delete operations uses the Empty operation which makes sure that the signals in current signal buffer are handled and empties the buffer. The function empties the current signal buffer by executing the operations as per to the signal type and transferring all the unhandled signals to next signal buffer in the sequence.

The algorithm 5 and 7 are used for implementation of Empty and Fill operations.

3. Implementation and Results

3.1. RAM model implementation

Single source shortest path algorithm as presented in Algorithm 1 was implemented using a linked list to store adjacency list and binary heap to store distance. Binary heap was also used to find the node with minimum distance.

Algorithm 5 EMPTY(S_i)

if $i = q + 1$ **then**
  $q = q + 1$
  create new B_q and S_{q+1}
end if
p_{max} = FINDMAXPRIORITY(B_i)
for *signal* in S_i **do** ▷ execute the signals
  if *signal.type* = UPDATE **then**
    if *signal.x* $\in B_i$ with priority p'' **then**
      Replace element with $(x, \text{MIN}(signal.p, p''))$
      Mark *signal* as handled
    else if *signal.x* $\notin B_i$ and *signal.priority* $\leq p_{max}$ **then** ▷ Delete Signal for proceeding buckets
      Insert element $(x, signal.p)$ into B_i
      signal.type = DELETE
    end if
  else if *signal.type* = PUSH **then**
    if *signal.x* $\in B_i$ with priority p'' **then**
      Replace (x, p'') with $(x, signal.p)$
    else
      Insert $(x, signal.p)$ into B_i
    end if
    Mark *signal* as handled
  else if *signal.type* = DELETE **then**
    if *signal.x* $\in B_i$ **then**
      Delete x from B_i
    end if
  end if
end for
if $i < q$ or $S_{i+1} \neq \phi$ **then** ▷ if this is not last bucket or S_{i+1} has signals
  for *signal* in S_i **do** ▷ if this is the last bucket no transfer required
    if *signal* is not handled **then**
      Insert *signal* into S_{i+1}
    end if
  end for
end if
$S_i = \phi$
if $|B_i| > 2^{2i}$ **then** ▷ if this bucket experienced overflow, transfer items to next
  Find 2^{2i}th smallest priority p in B_i
  for $item in B_i$ **do**
    if $item.p > p$ **then**
      Delete *item* from B_i
      Insert PUSH($item.x, item.p$) signal into S_{i+1}
    end if
  end for
  $numItemsToRemove = |B_i| - 2^{2i}$
  for *item* in B_i **do**
    if $numItemsToRemove = 0$ **then**
      break
    end if
    if $item.p = p$ **then**
      Delete *item* from B_i
      Insert PUSH($item.x, item.p$) signal into S_{i+1}
      $numItemsToRemove = numItemsToRemove - 1$
    end if
  end for
end if
if $|S_{i+1} > 2^{2i+1}$ **then** ▷ if next signal buffer experienced overflow, empty that signal buffer
  EMPTY(S_{i+1})
end if

Algorithm 6 FindMaxPriority(B_i)

if $i = q$ and $S_{q+1} = \phi$ **then** ▷ if B_i is last bucket and last signal bucket is empty
    return ∞
end if
if $B_i = \phi$ **then** ▷ if current Bucket is empty
    return $-\infty$
else
    Maximum priority in B_i
end if

Algorithm 7 Fill(B_i)

Empty(S_i)
if $S_{i+1} \neq \phi$ **then** ▷ ensures that next bucket is up to date
    Empty(S_{i+1})
end if
if $|B_{i+1}| < 2^{2i}$ and $i < q$ **then** ▷ if this is not last bucket
    Fill(B_{i+1})
end if
Find $(2^{2i} - |B_i|)$th smallest priority p in B_{i+1}
for *item* in B_{i+1} **do**
    if $item.p < p$ **then**
        Delete *item* from B_{i+1}
        Insert *item* into B_i
    end if
end for
for *item* in B_{i+1} **do**
    if $|B_i| = 2^{2i}$ **then**
        break
    end if
    if $item.p = p$ **then**
        Delete *item* from B_{i+1}
        Insert *item* into B_i
    end if
end for
q =Maximum index j for which $B_j \neq \phi$ or $S_{j+1} \neq \phi$

S.N.	Small Graph (n=8, e=24)		Network Science graph filtered (n=111, e=588)		Network Science graph (n=379, e=1828)	
	Binary heap	Bucket heap	Binary heap	Bucket heap	Binary heap	Bucket heap
1	40	63	541	2752	520	7968
2	40	76	318	2872	609	8272
3	44	63	301	2851	503	6679
4	35	61	366	2051	542	8723
5	36	81	224	1809	411	8021
6	35	91	437	2029	632	8132
7	44	79	239	1812	416	9269
8	88	61	248	1547	438	9019
9	45	65	294	2455	440	8701
10	42	64	277	1921	439	7223
Average Time (μs)	44.9	70.4	324.5	2209.9	500.4	8200.7

Table 1: Comparison of execution time of the algorithms

3.2. Cache-oblivious algorithm implementation

For the cache-oblivious algorithm for the single source shortest path problem, the binary heap used to keep track of the minimum distance in RAM model was replaced by the bucket heap data structure from [4]. And different graphs were used to measure the run-time for the algorithm. Table 1 provides a summary of execution time of the algorithm under different graphs.

The implementations were tested using 3 graphs. First graph was borrowed from the book by Rosen[5]. It contained 8 nodes and 24 edges. Second and third graph was extracted from the co authorship in network science graph[6]. The original graph had 1589 nodes and 2742 edges. For our purpose the single largest connected subgraph was extracted from the graph. The largest connected subgraph had 379 nodes and 1828 edges. Further lowest degree nodes were removed from the largest connected graph to obtain 3rd graph with 111 nodes and 588 edges. The graph algorithm were tested using all three graphs. Table 1 contains the execution time required to run the algorithms with 3 different graph as input.

4. Discussion

Table 1 presents the comparison between two different models with different input. The results show that binary heap implementation is better compared to the bucket heap implementation of the single source shortest path algorithm. The result is the opposite of what we had expected. First reason for unexpected result might be due to the implementation details that might have been missed during the implementation of the bucket heap. For this paper, bucket heap was implemented using higher level structures such as classes and structs. The choice of the data structures used and organization of code might have been a major factor in the degraded performance of the implementation of algorithm using bucket heap. Second in addition to implementation details, for this paper only small graph were used to test the algorithms. The cache-oblivious algorithms are designed to get performance advantage for problems that doesn't fit in primary memory. So the possibility is that the low performance might also be due to smaller problem sets.

As future enhancement, first enhancement would be to implement the bucket heap that supports floating points numbers. For this project, we have assumed key and priority are non-negative integer that is less than INT_MAX as provided by C++ environment. Another thing I would like to keep as future enhancement is to perform profiling of the code and analyze the performance of different sections of the code and improve the code to perform better. Third enhancement for the project might be to run the program in bigger graphs and compare the performance of the algorithms. Further more parallel version of the algorithm can be implemented and comparisons of the performance can be done.

5. Conclusion

Implementation details affects the algorithm performance. Even better algorithms if not implemented efficiently may result in bad performance of the algorithm, as we have seen in our case. The execution time for the bucket heap in our implementation performed poor compared to normal binary heap implementation. One of the primary reasons might be due to the implementation of the algorithm. So even if algorithms are proven theoretically to work better, the implementation details matter when algorithms are used in real life.

References

[1] J. S. Vitter, et al., Algorithms and data structures for external memory, Foundations and Trends® in Theoretical Computer Science 2 (4) (2008) 305–474.

[2] E. D. Demaine, Cache-oblivious algorithms and data structures, Lecture Notes from the EEF Summer School on Massive Data Sets 8 (4) (2002) 1–249.

[3] V. Kumar, E. J. Schwabe, Improved algorithms and data structures for solving graph problems in external memory, in: Proceedings of SPDP'96: 8th IEEE Symposium on Parallel and Distributed Processing, IEEE, 1996, pp. 169–176.

[4] G. S. Brodal, R. Fagerberg, U. Meyer, N. Zeh, Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths, in: Scandinavian Workshop on Algorithm Theory, Springer, 2004, pp. 480–492.

[5] K. H. Rosen, Discrete Mathematics & Applications, Mcgraw-Hill, 1999.

[6] Network data.
URL `http://www-personal.umich.edu/ mejn/netdata/`