**Research on Efficient Hop-Constrained s-t Simple Path Enumeration**

Author: Shi Wenlan

**Abstract:**

We study the hop constraint s-t path enumeration problem: Given a graph G, enumerate all simple paths p from a source vertex s to a target vertex t with the number of hops not larger than k. The state-of-the-art still leaves some interesting future work. And we want to further develop a better algorithm.

**Acknowledgement:**

This project is supported by the grant “Undergraduate Research Awards” awarded by Undergraduate Research Awards Selection Committee in 2021.

1. **Introduction**

The graph is a data structure used in various areas. It represents information of different entities(vertices) and their relationships(edges). To evaluate the relationship between two entities, we need to enumerate simple paths from one entity (vertex s) to another entity (vertex t). Intuitively, the longer the path, the weaker the relationship between two entities, and enumerating long paths often requires a lot of calculation. So, it is natural to impose a hop constraint k to s-t path enumeration.

To enumerate all simple paths from s to t with the number of hops less than or equal to k in polynomial delay, Peng et al. (2019) proposed Barrier-based constrained DFS (BC-DFS). However, barrier update incurs high overhead, so it is still not fast enough. To meet the rigid time constraint in real-world applications, Sun et al. (2021) proposed the PathEnum algorithm. This algorithm proposed new research ideas such as a lightweight index and join optimizer, and there is still a lot of room for optimization.

If this problem can be solved, this technology can be used in many areas such as detecting money laundering and build knowledge networks, and have good development prospects.

1. **Problem Statement and Formulation**

Hop constraint s-t path enumeration: Given a graph G, enumerate all simple paths p from a source vertex s to a target vertex t with the number of hops not larger than k.

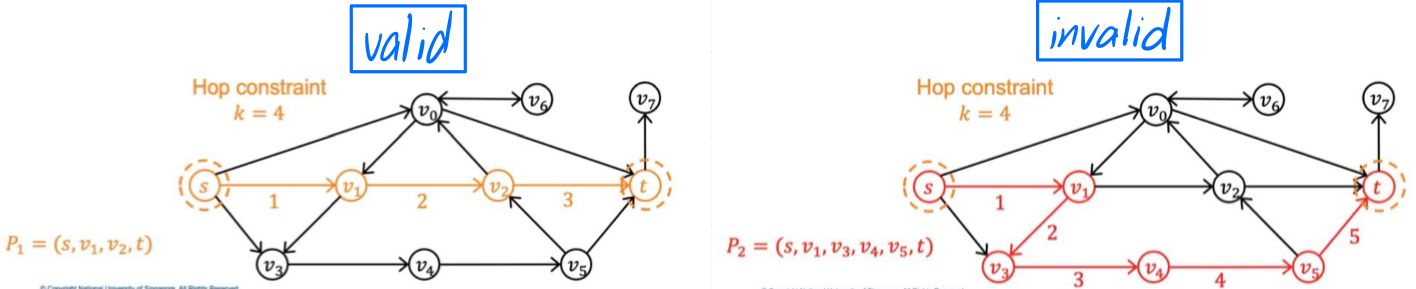


Figure : path sample

Figure 1 shows the sample of path that we need.

In this example, for given k = 4, the path we need to enumerate are: s-v0-t, s-v1-v2-t, s-v1-v2-v0-t, s-v3-v4-v5-t and s-v0-v1-v2-t.

And we want to enumerate these paths as fast as possible.

1. **Progress Made (Intermediate Results)**

In the past cycle, I met with professors every week to discuss research.

I measured the time-consuming of PathEnum when k=7. At this time, the time to search for the path on the subgraph occupied the main time-consuming. So, during this period, my main focus was to use parallel algorithms to accelerate path enumeration part.

My parallelization idea is to split the DFS on the subgraph into a one-step BFS plus multiple DFS parallelisms. In detail, my parallel version is to perform a BFS at s first to obtain the out neighbors of s. Then, perform multiple DFSs with the out neighbors of s as the starting point, t as the target, within the depth limit (k-1) and without going through s (e.g. if s has three out neighbors a, b and c, then we need three DFSs: from a to t, from b to t, and from c to t). The tasks of multiple DFSs should be assigned to different threads for simultaneous execution.

During the period, I studied the paper of Hao, K. et al. (2021) about Distributed hop-constrained st simple path enumeration at billion scale. This paper describes how to split a large graph and store it on different machines and perform hybrid search paradigm in parallel on multiple machines, and how to achieve load balance. I think the idea of load balance for multiple machines in parallel and that for multiple threads is similar. I feel that for small constraint k, the number of out neighbors of the starting point can be used to roughly estimate the workload of a DFS task. If the number of out neighbors of the starting point is large and the depth limit is deep, consider another round of BFS to further subdivide tasks to reduce the granularity of load balance.

The parallel library I am currently using is OpenMP. I've implemented multiple versions of the parallel implementation (including using an array to store the data for each thread and using a reduction statement), but it didn't get the desired effect. I guessed that there might be something wrong with the way of using it, so I wrote small cases to check the problem repeatedly, and asked for the assistance of my tutor and classmates. However, the desired effect has not been achieved so far. I will continue to investigate for a while, and if there is no progress, consider switching to other parallel libraries.

Recent experiments have found that recursive functions are not supported within the scope of openMP's reduction. Using recursive functions will cause data races, resulting in incorrect results. In addition, if the global variable is set to private and the function is called in the parallel area to modify the global variable, the global variable It is only readable in the scope of the called function, and the result of reading global variables in other scopes is wrong. The bug has been fixed by modifying DFS to a non-recursive mode. When the experiment with k=7 is performed on the soc-Epinions1 dataset, the speedup ratio of 8 threads is 6.9 times.

**UPDATE：**

**Code’s core idea:** run parallel dfs (explained in the blue paragraph above) on the subgraph built by path-enume algorithm.

**Core code:**

uint64\_t CycleEnumerator::para\_dfs(uint32\_t ele) {

// printf("start: thread: %d, ele: %d\n", omp\_get\_thread\_num(), ele);

uint64\_t c = 0;

std::vector<bool> vis(digraph\_->num\_vertices\_, false); // only include vertices on the path utill now

vis[src\_] = true;

vis[ele] = true;

std::vector<node> sta; // stack to store the vertices to explore next

sta.push\_back({src\_, ele, 1});

std::vector<uint32\_t> res(length\_constraint\_ + 1, -1); // path from s to t (explored)

res[0] = src\_;

int p = 0; // pointer to parent layer

while (!sta.empty()){

int step = sta.back().step;

uint32\_t v = sta.back().ele;

uint32\_t par = sta.back().parent;

// std::cout << "visit: parent: " << par << ", element: " << v << ", step: " << step << std::endl;

if (step > p+1){

for (;p < step-1;){

vis[par] = true;

res[++p] = par;

}

}

else if (step < p+1){

for (;p > step-1; p--){

vis[res[p]] = false;

res[p] = -1;

}

}

sta.pop\_back();

if (step >= length\_constraint\_) continue;

uint32\_t budget = length\_constraint\_ - step - 1;

uint32\_t neighbor\_offset = single\_bigraph\_[v];

uint32\_t start = single\_bigraph\_offset\_[neighbor\_offset];

uint32\_t end = single\_bigraph\_offset\_[neighbor\_offset + budget + 1];

for (uint32\_t i = start; i < end; ++i) {

// printf("dfs: thread: %d, ele: %d\n", omp\_get\_thread\_num(), ele);

uint32\_t w = single\_bigraph\_adj\_[i];

if (w == dst\_) {

c += 1;

}

else if (step == length\_constraint\_ - 2 && !vis[w]) {

c += 1;

}

else if (!vis[w]) {

sta.push\_back({v ,w, step + 1});

}

}

}

return c;

}

void CycleEnumerator::dfs\_on\_bigraph(uint32\_t u, uint32\_t k) {

// printf("s: %d, t: %d\n", src\_, dst\_);

// k is cost; length\_constraint\_ - k is the remaining budget; minus 1 is the cost of moving to a out neighbor.

uint32\_t budget = length\_constraint\_ - k - 1;

uint32\_t neighbor\_offset = single\_bigraph\_[u];

uint32\_t start = single\_bigraph\_offset\_[neighbor\_offset];

uint32\_t end = single\_bigraph\_offset\_[neighbor\_offset + budget + 1];

std::vector<std::pair<int, uint32\_t>> neigh;

int j = end - start;

for (uint32\_t i = start; i < end; ++i) {

uint32\_t v = single\_bigraph\_adj\_[i];

int degree = single\_bigraph\_offset\_[single\_bigraph\_[v] + budget] - single\_bigraph\_offset\_[single\_bigraph\_[v]];

neigh.push\_back({degree,v});

}

std::sort(neigh.begin(), neigh.end(), cmp);

neighbors\_access\_count\_ += (end - start);

omp\_set\_num\_threads(NumThreads); // firstprivate(Tvisited)

#pragma omp parallel for reduction(+:count\_) schedule(dynamic)

for (int i = 0; i < j; ++i) {

// auto tp3 = std::chrono::high\_resolution\_clock::now();

uint32\_t v = neigh[i].second;

if (v == dst\_) {

count\_ += 1;

}

else if (k == length\_constraint\_ - 2) {

count\_ += 1;

}

else {

count\_ += para\_dfs(v);

}

// auto tp4 = std::chrono::high\_resolution\_clock::now();

// auto dur = std::chrono::duration\_cast<std::chrono::nanoseconds>(tp4 - tp3).count();

// std::cout << "Thread " << omp\_get\_thread\_num() << " finished work, used time: " << dur / (double)1000000000 << " seconds." << std::endl;

}

}

**Experiment data until now:**

datasets are from <http://snap.stanford.edu/data/>

dfs avg is average query time of path-enume algorithm on 1000 queries from hot-point (the top 10% vertices of degree size) to hot-point

parallel is average query time of my parallel version algorithm on the same 100 queries for path-enume algorithm

acceleration rate = dfs avg / parallel dfs avg

#num of thread = 8, k = 7, dataset = soc-Epinions1

dfs avg = 58.660706391999994

parallel dfs avg = 8.390045470000004

acceleration rate = 6.9917030368608915

#num of thread = 20, k = 7, dataset = soc-Epinions1

dfs avg = 58.660706391999994

parallel dfs avg = 5.554151268000004

acceleration rate = 10.561596823977599

#num of thread = 30, k = 7, dataset = soc-Epinions1

dfs avg = 58.660706391999994

parallel dfs avg = 5.3886840379999965

acceleration rate = 10.885905719900371

#num of thread = 40, k = 7, dataset = soc-Epinions1

dfs avg = 58.660706391999994

parallel dfs avg = 5.486659052

acceleration rate = 10.691516610753672

#num of thread = 8, k = 7, dataset = web-google

dfs avg = 0.004450925999999998

parallel dfs avg = 0.001707849999999999

acceleration rate = 2.6061574494247157

#num of thread = 20, k = 7, dataset = web-google

dfs avg = 0.004450199999999998

parallel dfs avg = 0.0018977400000000008

acceleration rate = 2.3449998419172258

#num of thread = 20, k = 10, dataset = web-google

dfs avg = 9.662936491999998

parallel dfs avg = 1.5895904160000005

acceleration rate = 6.078884469066901

#num of thread = 8, k = 7, dataset = wiki-topcats

dfs avg = 6.741701656000002

parallel dfs avg = 1.5827161360000002

acceleration rate = 4.259577256246537

#num of thread = 8, k = 7, dataset = wikiTalk

experiment in process...

1. **Research Plan & Expected Outcome**

Month 1: Run more datasets and start writing paper drafts

Dataset plan to use: US Patents, DBpedia, Web-google, Web-standford, Twitter-social, Baidu-baike, Wiki-trust, Soc-Epinsion1, Web-uk-2005, WikiTalk, Soc-Slashdot0922, LiveJournal, Rec-dating, Bio-grid-yeast and Twitter-mpi.

Month 2: Experimental evaluation and algorithm optimization.

Think about tricks in load balance of threads.

Month 3: Algorithm optimization and paper writing.

Finish the paper as soon as possible.

1. **Appendix (if necessary)**

(Supporting documents can be put here)

1. **References**

Peng, Y., Zhang, Y., Lin, X., Zhang, W., Qin, L., & Zhou, J. (2019). Hop-constrained st Simple Path Enumeration: Towards Bridging Theory and Practice. Proc. VLDB Endow., 13(4), 463-476.

Sun, S., Chen, Y., He, B., & Hooi, B. (2021, June). PathEnum: Towards Real-Time Hop-Constrained st Path Enumeration. In Proceedings of the 2021 International Conference on Management of Data (pp. 1758-1770).

Hao, K., Yuan, L., & Zhang, W. (2021). Distributed hop-constrained st simple path enumeration at billion scale. Proceedings of the VLDB Endowment, 15(2), 169-182.