SE3352: Algorithm Design December 19, 2021

Notes 1 – Reachability on Large-scale Graphs

Instructor: *Guoqiang Li* Scribes: *Zihong Lin*

1 Introduction

The notes will focus on the reachability on large-scale graphs(hereinafter to be referred as 'reachability problem'). Many physical systems and items can be abstracted into a graph. Naturally, we can abstract the relationship network into a graph(The vertices are the people, while the edges are the relationship between people. Commonly, we can abstract 'a knows b' into $a \to b$). In modern application scenarios, the graph is usually large-scale. So research on the reachability problems is very meaningful.

2 Problem description

Definition 1 (Reachable)**.** *Let $G = (V, E)$ be a directed graph. For any two vertices $u, v \in V$, if there exists a path from u to v, we say v is **reachable** from u, denoted as $u \to v$.*

Definition 2 (Reachability problem)**.** *Given a directed graph $G = (V, E)$ and a pair of vertices $u, v \in V$, **the reachability problem** is to decide whether $u \to v$, and return TRUE or FALSE accordingly.*

3 Algorithms/Models

The algorithms of reachability problem can be further subdivided into index-based and index-free algorithms. Table 1 [1] shows some of the mainstream algorithms. Unless otherwise specified, we refer n as the number of nodes(vertices) and m as the number of edges.

Table 1: The mainstream algorithms on graph reachability.

Method	Query Time	Construction	Index Size
DFS/BFS	$O(n+m)$	$O(n+m)$	$O(n+m)$
Transitive Closure	$O(1)$	$O(nm) = O(n^3)$	$O(n^2)$
Optimal Tree Cover (Agrawal et al., SIGmOD'89)	$O(n)$	$O(nm) = O(n^3)$	$O(n^2)$
GRIPP (Triβl et al., SIGMOD'07)	$O(m-n)$	$O(n+m)$	$O(n+m)$
Dual-Labeling (Wang et al., ICDE'06)	$O(1)$	$O(n+m+t^3)$	$O(n+t^2)$
Optimal Chain Cover (Jagadish, TODS'90)	$O(k)$	$O(nm) = O(n^3)$	$O(nk)$
2-HOP (SODA 2002)	$O(nm^{1/2})$	$O(n^3\|T_C\|) = O(n^5)$	$O(m^{1/2})$
3-HOP (Yang Xiang et al., SIGMOD '09)	$O(logn+k)$	$O(kn^2)$	$O(nk)$

3.1 Index-based algorithm

The reachability index is a summary of all or part of the reachability information on the graph obtained by pre-computing the graph before querying (all reachability information is the information about whether any pair of vertex is reachable). Unless otherwise specified, the accessibility index so far is calculated based on the directed acyclic graph extracted from the original graph, rather than directly calculated from the original graph.

Specifically, this kind of directed acyclic graph compresses each strongly connected component in the original directed graph into a super node(vertex). If there is a node in one strongly connected component pointing to another strongly connected component, then connect the edges between the corresponding two super nodes. This method of compressing the original digraph can obviously save its reachability information losslessly: the two nodes are reachable if and only if they are reachable between the super nodes to which they belong. At the same time, this method can reduce the space overhead of the index, and the structural characteristics of the directed acyclic graph also make the establishment of some indexes easier, so it is widely used.

Definition 3 (Strongly connected component(SCC))**.** *Given a directed graph $G = (V, E)$, a strongly connected component is* ***a maximal set of vertices*** *$C \subseteq V$ such that for every pair of vertices $u, v \in C$, $u \to v$ and $v \to u$.*

Claim 4. *Compressing all the strongly connected components of a given graph $G = (V, E)$ will derive a* ***directed acyclic graph****.*

Proof. Denoting each strongly connected component as C_i, and its corresponding super node as v_i', the compressed graph will be $G' = (V', E')$, where V' is the set of all the super node v_i', and E' is the set of edges between super nodes.

Assume there exists a cycle in the compressed graph G', then for all the super node v_i' in this cycle, the original vertices it stands for will form a bigger strongly connected components, which makes a contradictory. □

Claim 5. *The compressed graph G' does not lose reachability information of the original graph.*

Proof. For any given two strongly connected components C_i and C_j, denote their corresponding super nodes as u' and v'. For any given two orginal vertices u and $v(u \in C_i$ and $v \in C_j)$, obviously, $u \to v \Leftrightarrow u' \to v'$. □

Usually, we can compute the strongly connected components of a given graph through **Tarjan algorithm**, whose time complexity is $O(n + m)$.

The method of compression is the basic and important pretreatment for index-based algorithms. Directed acyclic graph is easier to handle with and will save the space needed by index.

Index-based algorithms can be further divided into three categories: **Tree Cover**, **Chain Cover** and **Hop Cover**.

3.1.1 Optimal Tree Cover(Interval Labeling)

Definition 6 (Postorder traverse)**.** *Process all nodes of a tree by recursively processing all subtrees, then finally processing the root.*

The steps of the algorithm:

- First, establish the spanning tree of the directed acyclic graph(if the directed acyclic graph is divided into multiple connected components, attach all the spanning trees to the virtual root node, establishing a spanning tree forest);
- Then, give each node a two-dimensinal coordinate or interval $[i, j]$ through post-order traversal, where i is the smallest post-order traversal number in the subtree rooted at the current node and j is the post-order traversal number of the current node.
- Given two nodes u, v, if $I_v \subseteq I_u$, we think that $u \to v$.

Claim 7. *For any given tree, the post-order traversal number of the root node is bigger than any descendant nodes.*

Proof. Since the root node will be the last node to visit, the claim is obvious. □

Claim 8. *Given two nodes u and v, and their intervals $I_u = [i_u, j_u]$ and $I_v = [i_v, j_v]$, $I_v \subseteq I_u \Rightarrow u \to v$.*

Proof. If $I_v \subseteq I_u$, then $i_u \leq i_v$ and $j_u > j_v$. According to claim 6, we can infer that u is the ancestor of v in the spanning tree. Obviously, $u \to v$. □

Claim 9. *Tree-Cover may produce false negatives.*

Proof. It is possible that some edges are not included in the spanning tree, which may cause $I_v \subsetneq I_u$ given $u \to v$. □

To avoid false negatives, we need to store more information. That is what **Optimal Tree Cover**[2] does: store more interval labels.

1. Index Construction Steps:
   - Connect all nodes with no predecessors to a dummy node-root.(Same as above, to create a forest.)
   - Find a spanning tree for the DAG(requires a topological sorting first).
   - Label nodes according to tree edges (postorder).
   - For each non-tree edge (u,v)(in reverse topological order of the nodes): $label(v) = label(v) \cup label(u)$
   - If $label(a) \subseteq label(b)$ for labels inherited from nodes a, b, then keep only $label(b)$.

   Claim 10. *The index construction takes $O(nm) = O(n^3)$ time in the worst cases.*

   Proof. Analyze step by step.

   - Topological sorting will traverse all the edges and nodes, yielding $O(n + m)$
   - Postorder traverse will traverse the spanning tree, yielding $O(n)$

- In the worst cases, the graph may be a **bipartite graph**. Assume one side is A and another is B, and all edges are only from side A to side B and each node in side A points to all the nodes in side B. Then each node in side A will have $O(n)$ labels. Notice that the algorithm will traverse all the non-tree edges(the 4th step), and check if $label(a) \subseteq label(b)$(the 5th step). The former is $O(m)$ and the latter is $O(n)$ (for the number of labels is $O(n)$), then the time complexity would be $O(mn) = O(n^3)$.

□

2. Query processing

   For any given node u, assume its labels are $label(u) = [u_{start}, u_{end}], [u_{start_1}, u_{end_1}]$.... Given another arbitrary node v, we can infer that $u \to v \Leftrightarrow \exists i : (u_{start_i} \leq v_{end} < v_{end_i})$

   Claim 11. *The query is $O(n)$.*

   Proof. In the worst cases, one node may inherit labels from all the other $n-1$ nodes, yielding n nodes. One query may traverse all the label to get the answer, which would take $O(n)$ time. □

3. Space

   Claim 12. *The Optimal Tree Cover will take $O(n^2)$ space in the worst cases.*

   Proof. In the worst cases, the graph may be a **bipartite graph**. We assume the number of nodes on one side of the bipartite graph is x, then the other would be $n-x$. If each one of these x nodes inherit $n-x$ labels from those $n-x$ nodes, then we will need $(n-x)(x+1)$ labels, yielding $O(n^2)$. □

3.1.2 GRIPP

GRIPP[3] is another indexed-based algorithm using tree-cover. This algorithm is based on both pre- and postorder labeling.

For the pre- and postorder labeling of a given tree, each node will be given three values, namely v_{pre}, v_{post} and v_{depth}. The preorder value v_{pre} is assigned the first time node v is encountered during the traversal. The postorder value v_{post} is assigned after all successor nodes of v have been traversed. The depth of v, v_{depth} is also assigned during the depth-first traversal. The depth of the root node of the tree is 0. The depth of any node v in the tree is the distance to the root node. Only one counter is used to calculate pre- and postorder values.

Example 13. *Figure 1 is an of A pre- and postorder labeled tree with depth information.*

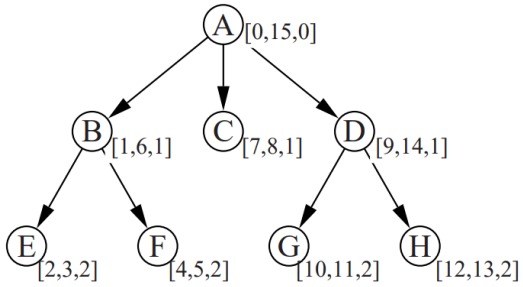


Figure 1: A pre- and postorder labeled tree with depth information.

Claim 14. *Given two nodes u and v, $u \to v \Leftrightarrow u_{pre} < v_{pre} < u_{post}$.*

Proof. If $u \to v$, then u must be the ancestor of v. So obviously, $u_{pre} < v_{pre}$. Notice that only one counter is used to calculate pre- and postorder values, then obviously, $v_{pre} < u_{post}$. We can observe this property in the example shown in figure 1. □

The above method is based on a tree, but in most cases, a DAG is not a tree. GRIPP is created to cope with more general cases. GRIPP uses special index table named **IND(G)**.

1. **Index Construction**

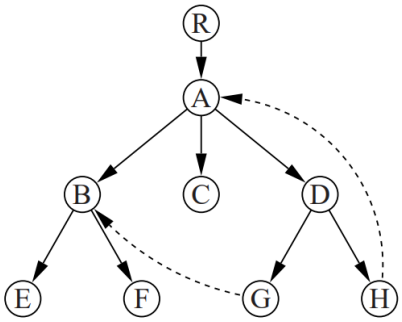
   Steps:

   - Choosing the root node as the starting node, traverse the tree.
   - If an arbitrary node v is visited at the first time, push the triple $(v_{pre}, v_{post}, v_{depth})$ to $IND(G)$ as a **tree instance** and traverse in subtrees.
   - If an arbitrary node v is visited again, push the triple to $IND(G)$ as a **non-tree instance** but not traverse.

   Definition 15. *(Tree and non-tree instances) Let $IND(G)$ be the index table of graph $G = (V, E)$. Let $v \in V$ be a node of G and v' be an instance of v in $IND(G)$. v' is a tree instance of v, iff it was the first instance created for v in $IND(G)$. Otherwise v' is a non-tree instance of v.*

   In the following, we refer to any instance in $IND(G)$ of a node v as v', to a tree instance as v^T, and to a non-tree instance as v^N. The set of tree instances in $IND(G)$ is I^T and the set of non-tree instances is I^N. In analogy, the set of tree edges is E^T and the set of non-tree edges E^N.

   Example 16. *Figure 2(a) shows a graph and Figure 2(b) shows its index table, resulting from a traversal in lexicographic order of node labels. Nodes A and B have two instances in $IND(G)$ because they have two incoming edges.*



(a) A graph G

node	pre	post	depth	type
R	0	21	0	tree
A	1	20	1	tree
B	2	7	2	tree
E	3	4	3	tree
F	5	6	3	tree
C	8	9	2	tree
D	10	19	2	tree
G	11	14	3	tree
B	12	13	4	non-tree
H	15	18	3	tree
A	16	17	4	non-tree

(b) The index table IND(G)

Figure 2: Graph G and its GRIPP index table IND(G). Solid lines in the graph represent tree edges, dashed lines are non-tree edges.

Claim 17. *The index construction takes $O(n+m)$ time.*

Proof. Done through **depth first traversal**, taking $O(n+m)$ time. □

2. **Query processing**

Definition 18. *(Order Tree) Let $G = (V, E)$ and let $IND(G)$ be its index table. The order tree, $O(G)$, is a tree that contains all instances of $IND(G)$ as nodes connected by all edges of G.*

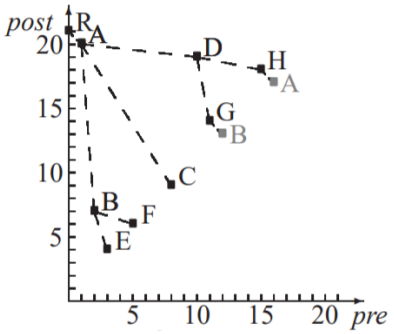


Figure 3: Pre-/ postorder plane for GRIPP index table from Figure 2(b). Dotted lines indicate O(G). Non-tree instances are displayed in gray.

Definition 19. *(Reachable Instance Set) Let $v \in V$ be a node of graph G and $v^T \in IND(G)$ its tree instance. The reachable instance set of v, written $RIS(v)$, is the set of all instances that are reachable from v^T in $O(G)$, i.e., have a preorder value in $[v^T_{pre}, v^T_{post}]$.*

Definition 20. *(Hop Node) Let v, $w \in V$ and w^N be a non-tree instance of w. If $w^N \in RIS(v)$ then w is called a hop node for v.*

Algorithm 1: Function to answer $reach(v, w)$ using the GRIPP index.

```
1  used_hops ← ∅;
2  used_stops ← ∅;
3  if w ∈ RIS(v) then
4  |  return true;
5  else
6  |  used_hops ← used_hops ∪ (v);
7  |  if v ∈ STOP_NODES then
8  |  |  used_stops ← used_stops ∪ (v);
9  |  |  return false;
10 |  else
11 |  |  while non_tree_inst ← nextStop(RIS(v)) do
12 |  |  |  tree_inst ← getTreeInst(non_tree_inst);
13 |  |  |  if reachability(tree_inst, w) then
14 |  |  |  |  return true;
15 |  |  |  end
16 |  |  end
17 |  |  if isInRIS(v, used_stops) then
18 |  |  |  return false;
19 |  |  end
20 |  |  H_1 ... H_n ← getUsedHopsInRIS(v);
21 |  |  non_tree_instances ← getNonTreeInst(RIS(v)\RIS(H_1)\...\RIS(H_n));
22 |  |  foreach non_tree_inst ∈ non_tree_instances do
23 |  |  |  tree_inst ← getTreeInst(non_tree_inst);
24 |  |  |  if !hasChildren(tree_inst) then
25 |  |  |  |  continue;
26 |  |  |  end
27 |  |  |  if tree_inst ∈ used_hops then
28 |  |  |  |  continue;
29 |  |  |  end
30 |  |  |  if isInRIS(tree_inst, used_hops) then
31 |  |  |  |  continue;
32 |  |  |  end
33 |  |  |  if reachability(tree_inst, w) then
34 |  |  |  |  return true;
35 |  |  |  end
36 |  |  |  if isInRIS(tree_inst, used_hops) then
37 |  |  |  |  return false;
38 |  |  |  end
39 |  |  end
40 |  |  return false;
41 |  end
42 end
```

Given two nodes u and v, if we want to know whether $u \to v$.

Steps of query:

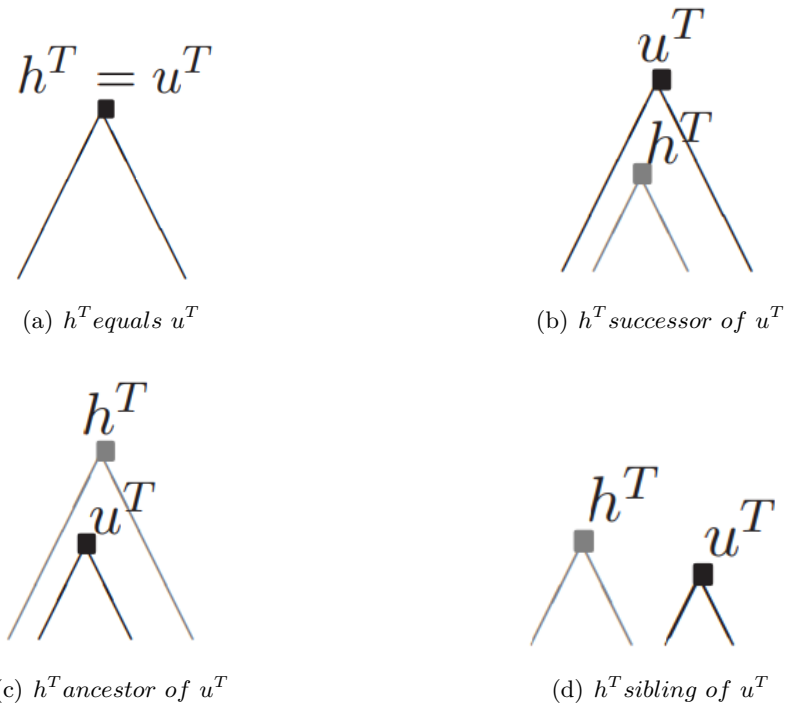
- Check whether $v \in RIS(u)$.
- If not, check all hop nodes and recursively check their reachable instance sets.

Claim 21. *The query would take $O(m-n)$ time in the worst cases.*

Proof. In the worst cases, the query would traverse all the hop nodes. Each hop node is derived from a non-tree edge, the number of which is $O(m-n)$. Therefore, the query would take $O(m-n)$ time. □

Claim 22. *The efficiency of query would be improved by pruning.*

GRIPP keeps a list U of all nodes that were used to retrieve a reachable instance set, i.e., the start node and the hop nodes. Assume we have found a new hop node h. Whether $RIS(h)$ should be considered depends on the location of the tree instance h^T of h relative to the tree instances of nodes in U. Here are the four possibility position of h^T and $u^T, u \in U$.



(a) h^T *equals* u^T

(b) h^T *successor of* u^T

(c) h^T *ancestor of* u^T

(d) h^T *sibling of* u^T

Figure 4: Possible locations of h^T of hop node h relative to u^T, $u \in U$

Explanation:

(a) h^T is equal to the tree instance of any node in U;

(b) h^T is the successor if the tree instance of at least one node in U;

(c) h^T is ancestor of the tree instance of at least one node in U and neither (a) and (b) is true;

(d) h^T is sibling to the tree instances of all nodes in U.

Skip Strategy

- For case (a) and (b), obviously, $RIS(h) \subseteq RIS(u)$, their is no need to traverse h, just skip it;
- For case (d), no pruning is possible;
- For case (c), we only have to consider instances from $RIS(h)$ whose preorder values lie outside the pre- and postorder range of u^T.

Stop Strategy

Definition 23. *(Stop Node) Let $s \in V$ be a node of graph G and let $RIS(s)$ be its reachable instance set in $O(G)$. s is called a stop node iff all non-tree instances in $RIS(s)$ also have their corresponding tree instances in $RIS(s)$ or are a non-tree instance of s.*

When we reach the tree instance of a stop node s, we only have to check if the target node v is in $RIS(s)$.

3. **Space**

   Store the information of nodes and the index table would take $O(n + m)$.

3.1.3 Optimal Chain Cover

Definition 24. *(Chain Cover) Let $C = \{c_1, c_2, , c_k\}$ the set containing some chains of a given graph $G = (V, E)$. If $\forall u \in V, \exists i : u \in c_i$ and $\forall i \neq j, c_i \cap c_j = \varnothing$, then C is of chain cover of G.*

Definition 25. *(Optimal Chain Cover) A chain cover C with least number of chains is an **optimal chain cover**.*

1. **Index Construction**

   Claim 26. *Optimal Chain Cover $\leq_p$ Min-Flow.*

   Given a graph $G = (V, E)$, we can construct another graph $H = (V', E')$. Any node v_i is presented as a directed edge (x_i, y_i) in H. A source node is added into H that links to every node with in-degree 0 in H, and a sink node is added that is linked by every node with out-degree 0 in H. Each flow in H corresponds to a chain in G. We will get an optimal chain cover by finding the min-flow from the source node to the sink node(such that every edge $(x_i, y_i) \in E'$ has a positive flow).

   Such method can be solved through **Ford Fulkerson's algorithm** in $O(n^3)$ time.

2. **Space**

   For each node $v_i \in G$, it is denoted a **chaincode**, which is a list of pairs: $(1, p_{i,1}), (2, p_{i,2}), ..., (k, p_{i,k})$. Each pair $(j, p_{i,j})$ means that the node v_i can reach any node s from the position $p_{i,j}$ in the j-th

chain. If v_i cannot reach any node in the j-th chain, then $p_{i,j} = +\infty$. The chain cover index contains chaincode(v_i) for every node v_i in G. The chaincode will take $O(nk)$ space in total(where k is the number of chains).

3. **Query processing**

   If the number of chain k is small enough to store the index into a 2D table, then the query yields $O(1)$. Otherwise storing it into lists and indexing the lists yields $O(logn + k)$ query time.

3.1.4 2 Hop-Cover

In a **2-hop cover**, a node u in G is assigned to a 2-hop code, $2hopcode(u) = (L_{in}(v), L_{out}(v))$, where $L_{in}(v)$ and $L_{out}(v)$ are subsets of the nodes in G[5].

Query A reachability query $u \to v$ is to be answered true iff $P_{2hop}(2hopcode(u), 2hopcode(v)) = L_{out}(v) \cap L_{in}(v) \neq \varnothing$.

Main idea If we take one node u as a center node, for all nodes w in the ancestors of u(denoted as $ancs(u)$) and all nodes v in the descendants of u(denoted as $desc(u)$), $w \to v$.

Claim 27. *Finding a minimum 2-hop cover in a graph is **NP hard**.*

Approximate Solution Greedily pick the node with the highest compression as a center node. The approximate ratio is $\log n$.

1. Index Construction: Complicated and costly $O(n^3|T_C|) = O(n^5)$.
2. Query processing: $O(m^{1/2})$
3. Space: $O(nm^{1/2})$

3.2 Index-free algorithms

Index-free algorithms are various. **Traversal**, **Heuristic Search**, and **Random Walks** are two common algorithms.

3.2.1 Naïve Traversal

DFS/BFS can solve reachability problem without index, saving lots of space. But such methods are naïve for large-scale graph, for the methods will traverse the graph which in the worst cases yields $O(n + m)$. Though DFS can be optimized by pruning, but it's still not practical for large-scale graph.

3.2.2 Bidirectional Search(Two-end BFS)

Bidirectional search is a graph search algorithm which find smallest path form source to goal vertex. It runs two simultaneous search:

- Forward search form source/initial vertex toward goal vertex

- Backward search form goal/target vertex toward source vertex

Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. The search terminates when **two graphs intersect**.

In many cases it is faster, it dramatically reduce the amount of required exploration. Suppose if branching factor of tree is b and distance of goal vertex from source is d, then the normal BFS/DFS searching complexity would be $O(b^d)$. On the other hand, if we execute two search operation then the complexity would be $O(b^{\frac{d}{2}})$ for each search and total complexity would be $O(b^{\frac{d}{2}} + b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$ which is far less than $O(b^d)$.

3.2.3 Heuristic search

Heuristic algorithms, such as A^*, are used to improve the efficiency.

A^* selects the path that minimizes $f(n) = g(n) + h(n)$, where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. Typical implementations of A^* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty).

If the heuristic function h is admissible, meaning that it never overestimates the actual minimal cost of reaching the goal, then A^* is itself admissible (or optimal) if we do not use a closed set. If a closed set is used, then h must also be monotonic (or consistent) for A^* to be optimal. This means that for any pair of adjacent nodes x and y, where $d(x, y)$ denotes the length of the edge between them, we must have:

$$h(x) \leq d(x, y) + h(y)$$

This ensures that for any path X from the initial node to x:

$$L(X) + h(x) \leq L(X) + d(x, y) + h(y) = L(Y) + h(y)$$

where L is a function that denotes the length of a path, and Y is the path X extended to include y. In other words, it is impossible to decrease (total distance so far + estimated remaining distance) by extending a path to include a neighboring node.

Monotonicity implies admissibility when the heuristic estimate at any goal node itself is zero, since (letting $P = (f, v_1, v_2, \ldots, v_n, g)$ be a shortest path from any node f to the nearest goal g):

$$h(f) \leq d(f, v_1) + h(v_1) \leq d(f, v_1) + d(v_1, v_2) + h(v_2) \leq \ldots \leq L(P) + h(g) = L(P)$$

The time complexity of A^* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: $O(bd)$, where b is the branching factor (the average number of successors per state).

3.2.4 ARROW

The index-based algorithms work well when the graph is not updated or updated infrequently, while in practical applications the graph is usually large-scale and dynamic(such as social networks). **ARROW**(Approximating Reachability using Random-walks Over Web-scale graphs)[6] has been created to better cope with dynamic networks.

Definition 28. *(Graph Diameter) The diameter of graph is the maximum distance between the pair of vertices.*

Main Idea Given a reachability query (u, v) on a graph snapshot(since the grpah is **dynamic**) $G_t(V_t, E_t)$ at time t, with $|V_n| = n$, ARROR constructs two sets of 'stops', $F(u) = s : u \to s$ and $B(v) = t : t \to v$. ARROW answers TRUE iff $F(u) \cap B(v) \neq \varnothing$. It's kind of like 2-Hop labeling, but notice that ARROW stores no index. The sets $F(u)$ and $B(v)$ are constructed through **random walks**.

Random Walks ARROR constructs $r = c_{\text{numWalks}} \times \sqrt[3]{n^2 \ln n}$ random walks, where c_{numWalks} is a parameter to be set. Each walk is of length $l = c_{\text{walkLength}} \times$ diam from u on G_t, where *diam* is the diameter of the graph and $c_{\text{walkLength}}$ is the diameter to be set.

Algorithm 2: ARROW

```
1  Algorithm ARROW(G_t, u, v)
2  wL ← c_walkLength × diam;
3  F(u) ← B(v) ← ∅;
4  for w in 1...c_numWalks × ∛(n² ln(n)) do
5  |   currentNode ← u;
6  |   for s in 1...wL do
7  |   |   neighbour ← random neighbour of currentNode;
8  |   |   F(u) = F(u)∪ {neighbour};
9  |   |   currentNode ← neighbour;
10 |   end
11 end
12 return F(u) ∩ B(v) ≠ ∅;
```

Theoretical bound on number of walks

Claim 29. $r = c_{numWalks} \times \sqrt[3]{n^2 \ln n}$ *is a pessimistic upper bound on the number of walks required to obtain the correct anwser with high probability.*

Proof. Given a strongly connected graph G with vertex set V, such that $|V| = n$, let the random walk conducted on the directed edges in the forward direction have transition matrix $\overrightarrow{P}$ and stationary distribution $\overrightarrow{\pi}$, and the random walk on the directed edges traversed in the reverse direction have transition matrix $\overleftarrow{P}$ and stationary distribution $\overleftarrow{\pi}$. Given two integers $k, t^* > 0$, consider two collections of random walks of length t^*, $\{(X_t^i)_{t\leq 0}, X_0^i = x : 1 \leq i \leq k\}$ started from $x \in V$ conducted according to $\overrightarrow{P}$ and $\{(Y_t^i)_{t\leq 0}, Y_0^i = y : 1 \leq i \leq k\}$ started from $y \in V$ conducted according to $\overleftarrow{P}$.

Futher define $S_x = \left\{X_{t^*}^i : 1 \leq i \leq k\right\}, S_y = \left\{Y_{t^*}^i : 1 \leq i \leq k\right\}$

If

$$t^* \geq \max\left\{t_{\text{mix}}^{\overrightarrow{P}}\left(\frac{1}{\sqrt{2}n}\right), t_{\text{mix}}^{\overleftarrow{P}}\left(\frac{1}{\sqrt{2}n}\right)\right\}$$

then $P\{S_x \cap S_y \neq \varnothing\} \geq 1 - 1/n$ if

$$k \geq k^* = \left\{\frac{16n^2 \ln n}{(\alpha(\overrightarrow{\pi}, \overleftarrow{\pi}))^2}\right\}^{\frac{1}{3}}$$

whenever

$$k^* \cdot \left(\max\{\max\{\mu(i) : 1 \leq i \leq n\}, \max\{\nu(i) : 1 \leq i \leq n\}\} + \frac{1}{2n}\right) < 1$$

where

$$\alpha(\overrightarrow{\pi}, \overleftarrow{\pi}) = n\left(\sum_{i=1}^{n} \max\left\{0, \overrightarrow{\pi}(i) - \frac{1}{2n}\right\} \cdot \max\left\{0, \overleftarrow{\pi}(i) - \frac{1}{2n}\right\}\right)$$

In particular, if $\alpha(\overrightarrow{\pi}, \overleftarrow{\pi})$ is a constant then $k^* = \theta\left(\sqrt[3]{n^2 \ln n}\right)$ □

Diameter estimation Determine the exact diameter of a given graph is usually very costly in practical application, so ARROW uses an approximate method instead: Selecting the longest distance recorded across 10 rounds of BFS from randomly selected nodes on the given graph.

The problem of escape from SCCs Assume that a walk begun from a vertex v and run for t steps stays within its SCC with probability $p(t, v)$. If we conduct $c\sqrt[3]{n^2 \ln n}$ independent walks for some appropriately chosen c, then we can argue by Chernoff bounds that $c'\sqrt[3]{n^2 \ln n}$ of them stay within the SCC with probability $1 - 1/n$ if

$$p(t, v) \geq \frac{\ln n}{\sqrt[3]{n^2 \ln n}}$$

This bound decreases with n faster than $n^{-2/3}$ and, so, is a relatively modest requirement, especially from those vertices that are deep within the SCC.

Complexity The sizes of sets $F(u)$ and $B(v)$, and the time taken to construct and intersect them are all $r \cdot l$, where r is the number of random walks and l is the length of walks, as defined in Section II-A. Therefore, query answering time is $\mathcal{O}\left(\sqrt[3]{n^2 \ln n} \cdot \text{ diam }\right)$

3.2.5 Other methods

In practical application, we may be interested in not only the information of reachability, but also the distance of two nodes. So usually, reachability is determined at the same time when solving the shortest path problem. This is very common in the application of map, such as Google Map.

Table 2 shows some mainstream shortest-path algorithms.

Table 2: Three mainstream shortest-path algorithm.

Method	Time Complexity	Space Complexity
Dijkstra	$O(V + E \log V)$	$O(V^2)$
Floyd	$O(V^3)$	$O(V^2)$
Bellman Ford	$O(VE)$(average) and $O(V^3)$(worst)	$O(V)$

4 Practical Algorithm

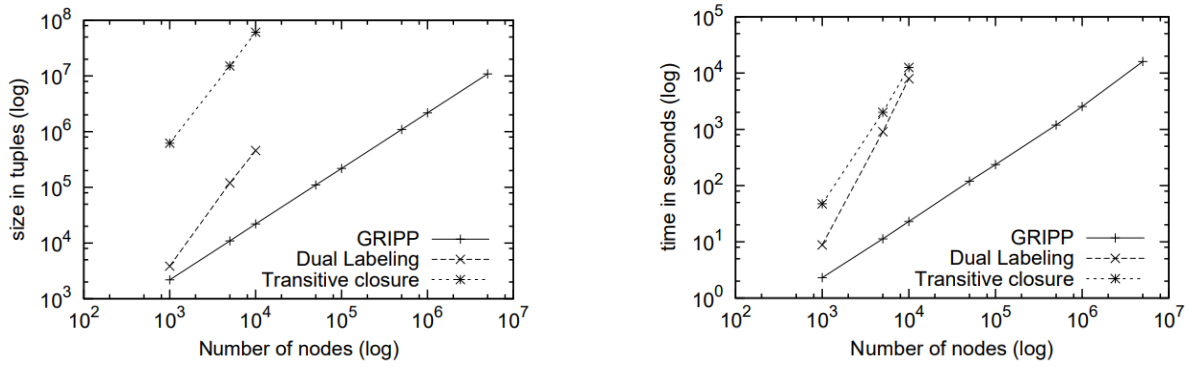
A graph can be categoried into dynamic(frequently updated) graph and static(infrequently updated). For the static one, the typically best method is to construct an index(which will take a relatively long time in construction but save huge amount of time in query), while for the dynamic one, traversal or random walk is considered practical.

4.1 Static(infrequently updated) Graph

We have introduced lots of index-based algorithms above, like 'Optimal tree cover', '2-Hop labeling', etc, in which 'GRIPP' excels.

As is displayed in Table 1, for GRIPP, its index construction yields $O(n + m)$, its query yields $O(m - n)$, and it needs $O(n + m)$ space for index. If given a dense graph($m \approx n^2$), the construction and query will both yield $O(n^2)$. If given a sparse graph($m \approx kn$), the query will yield $O(n)$. So GRIPP performs well when given a sparse graph, and is still acceptable given a very dense graph. But its construction time is still too long for a dynamic graph. So it's appropriate for static(infrequently updated) graph(useful in biological analysis), but not for dynamic one like social network.

Fortunately, although graphs are often updated in real time, in many application scenarios we do not need to update graphs frequently to deal with immediate changes. Facing this kind of application scenario, by pre-establishing indexes (using distributed parallel computing frameworks such as Spark), good query efficiency can be provided.



(a) Average size(tuples)

(b) Average time (sec)

Figure 5: Average time and size for the GRIPP index table, Dual Labeling on the component graph, and the transitive closure for synthetic scale-free graphs with 100 % more edges than nodes.[4]

4.2 Dynamic(frequently updated) Graph

Faced with the problem of accessibility of dynamic graphs, we can often use traversal. But for a large-scale graph, traversal may traverse all edges and nodes in the worst case, which is unacceptable. Moreover, in many application scenarios, we often do not need to give a very accurate answer (such as judging whether two people know each other in a social network), so using some approximation or randomization algorithms will greatly improve query efficiency.

ARROW's query time complexity is $O\left(\sqrt[3]{n^2 \ln n} \cdot \text{diam}\right)$. When given a dense graph($m \approx n^2$), the common BFS/DFS yields $O(m+n) = O(n^2)$. In this case, ARROW's query performs better than BFS/DFS. That is to say, ARROW sacrifices a little accuracy to get better performance.

The table 3 shows the experimental results of some datasets.[6]

Table 3: Dynamic Graph Results (Timeout = 2 hours, $c_{\text{numWalls}} = 0.01, c_{\text{walkLength}} = 1$)

Dataset	Method	Insert(ms)	Delete(ms)	Query(ms)	Init(ms)	Mem(MB)	Total(ms)	Acurracy
Facebook	DAGGER	0.22	90.405	0.1655	446.091	4007.7	8244220	
	ARROW	0.00025	0.00049	0.7868	3.437	12.69	**674.99**	0.956
	BBFS	0.000192	0.000321	6.8775	0.00016	13.172	4324.954	
	BFS	0.000143	0.00031	18.8432	0.000138	6.809	10286.772	
Enron	DAGGER	864.27	1.08188	0.0048	604.135	44.04	TIMEOUT	
	ARROW	0.000229	0.000514	0.0468	6.45	18.53	**2818.158**	1
	BBFS	0.000263	0.000627	1.0011	0.00014	18.537	3091.943	
	BFS	0.00021	0.00051	2.1536	0.00027	8.631	3470.022	
Epinions	DAGGER	55.7711	71.0257	2.577	1786.1	1453.75	TIMEOUT	
	ARROW	0.000135	0.000191	6.599943	95.166	17.42	**808.029**	0.998
	BBFS	0.00017	0.00024	14.345296	0.00024	18.086	8198.45	
	BFS	0.00011	0.00013	14.966	0.000086	8.86	8561.26	
Flickr	DAGGER	NA	NA	NA	TIMEOUT	NA	TIMEOUT	
	ARROW	0.000223	0.000532	285.872	4354.84	415.011	**45334.908**	0.986
	BBFS	0.000307	0.000703	463.7533	0.00022	470.299	290706.677	
	BFS	0.0002491	0.001004	665.575	0.000149	208.584	405832.625	

4.3 Improved Heuristic Searching

Above we introduced the heuristic algorithm A^*. While the admissibility criterion of A^* guarantees an optimal solution path, it also means that A^* must examine all equally meritorious paths to find the optimal path. To compute approximate shortest paths, it is possible to speed up the search at the expense of optimality by relaxing the admissibility criterion. Oftentimes we want to bound this relaxation, so that we can guarantee that the solution path is no worse than $(1+\varepsilon)$ times the optimal solution path. This new guarantee is referred to as ε-admissible. There are a number of ε-admissible algorithms:

- **Weighted $A*$/Static Weighting's**[7] If $h_a(n)$ is an admissible heuristic function, in the weighted version of the A^* search one uses $h_w(n) = \varepsilon h_a(n), \varepsilon > 1$ as the heuristic function, and perform the A^* search as usual (which eventually happens faster than using ha since fewer nodes are expanded).

The path hence found by the search algorithm can have a cost of at most ε times that of the least cost path in the graph.[8]

- **Dynamic Weighting**[9] uses the cost function $f(n) = g(n) + (1 + \varepsilon w(n))h(n)$, where

$$w(n) = \begin{cases} 1 - \frac{d(n)}{N} & d(n) \leq N \\ 0 & \text{otherwise} \end{cases}$$

and where $d(n)$ is the depth of the search and N is the anticipated length of the solution path.

- **Sampled Dynamic Weighting**[10] uses sampling of nodes to better estimate and debias the heuristic error.

- A^*_ε[11] uses two heuristic functions. The first is the FOCAL list, which is used to select candidate nodes, and the second hF is used to select the most promising node from the FOCAL list.

- A_ε[12] selects nodes with the function $Af(n) + Bh_F(n)$, where A and B are constants. If no nodes can be selected, the algorithm will backtrack with the function $Cf(n) + Dh_F(n)$, where C and D are constants.

- **Alpha**A^*[13] attempts to promote depth-first exploitation by preferring recently expanded nodes. AlphaA^* uses the cost function $f_\alpha(n) = (1 + w_\alpha(n))\, f(n)$, where

$$w_\alpha(n) = \begin{cases} \lambda & g(\pi(n)) \leq g(\tilde{n}) \\ \Lambda & \text{otherwise} \end{cases}$$

where λ and Λ are constants with $\lambda \leq \Lambda$, $\pi(n)$ is the parent of n, and $\tilde{n}$ is the most recently expanded node.

5 Concluding Remarks

In practical applications, there are different algorithms dealing with different graphs. We cannot find a universal algorithm that works in all situations. Index-based algorithms are more suitable for static or infrequently updated graphs, and are suitable for application scenarios that require high precision and high query efficiency, such as scientific research and analysis.

There are many index-based algorithms, tree-cover, chain-cover, two-hop cover... We can't say that a certain algorithm is suitable for all static graphs. Some algorithms are more efficient for dense graphs while some are more efficient for sparse graphs. Some are suitable for large-scale graphs while some are suitable for small-scale graphs. Some are slow to construct but very fast to query while some are very fast to query but very slow to construct. Rather than choosing a fixed algorithm, it is better to heuristically select the most suitable algorithm according to the shape of the graph, the number of nodes, and the density of the graph. To form a unified framework that can deal with different graphs is one of the directions worth studying in the future.

Whether using traditional traversal, heuristic search or random walk can all cope with the real-time dynamic changes of the graph. Although traditional traversal guarentees to determine the reachability

between two points, it may traverse all vertices and edges in the worst case. Unfortunately, most of the graphs in reality are very large, and such traversal is unacceptable.

In order to improve efficiency, we usually use heuristic algorithms or random algorithms (random walks) or some approximate algorithms in applications. These algorithms greatly improve the efficiency of query and meet the needs of most applications. (Because in many daily applications, we often don't need 100% accurate answers.)

Although these algorithms have significantly improved efficiency in most cases, in some cases, the algorithms may degenerate into ordinary traversal algorithms. Moreover, algorithms such as random walk sacrifice a part of accuracy in exchange for higher query efficiency, which may not be applicable in some scenarios with high accuracy requirements. Of course, you can classify queries, by using random walk algorithms for requests that do not require high accuracy, and using DFS/BFS for requests that require high accuracy.

In reality, graphs are generally very large, and even $O(n)$ algorithms are sometimes unacceptable. However, reasonable application of distributed and parallel computing frameworks, such as Spark, Storm, etc., can improve efficiency by taking the adventage of parallel computation.

Graph is a very interesting data structure. It perfectly abstracts many real systems. People's various researches on graphs have been going on for a long time, and I think these researches will continue to develop endlessly. With the continuous improvement of computing power, the continuous development of technology and the continuous maturity of theories, many things that were considered impossible in the past have become reality. Let me think about it here: judging whether two points are reachable (connected) in a circuit is very simple and very fast. Just use an electric meter to check the resistance at both ends. If the resistance is infinite, it means that the two points are not reachable to each other. This physical judgment is often much faster than a logical judgment. I was wondering whether it is possible to transfer a huge graph from the virtual logical world to the physical world, and judge the reachability in a physical way? Of course this is too whimsical at present, but people 200 years ago would not have thought that I would type these words in front of a high-tech machine.

References

[1] Yu J X, Cheng J. Graph reachability queries: A survey[M]//Managing and Mining Graph Data. Springer, Boston, MA, 2010: 181-215.

[2] Agrawal R, Borgida A, Jagadish H V. Efficient management of transitive relationships in large data and knowledge bases[J]. ACM SIGMOD Record, 1989, 18(2): 253-262.

[3] Trißl S, Leser U. GRIPP Indexing and Querying Graphs Based on Pre-and Postorder Numbering[M]. Humboldt-Univ. zu Berlin, 2006.

[4] Trißl S, Leser U. Fast and practical indexing and querying of very large graphs[C]//Proceedings of the 2007 ACM SIGMOD international conference on Management of data. 2007: 845-856.

[5] D'angelo G, D'emidio M, Frigioni D. Fully dynamic 2-hop cover labeling[J]. Journal of Experimental Algorithmics (JEA), 2019, 24: 1-36.

[6] Sengupta N, Bagchi A, Ramanath M, et al. Arrow: Approximating reachability using random walks over web-scale graphs[C]//2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE, 2019: 470-481.

[7] Pohl, Ira (1970). "First results on the effect of error in heuristic search". Machine Intelligence. 5: 219–236.

[8] Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley. ISBN 978-0-201-05594-8.

[9] Pohl, Ira (August 1973). "The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving" (PDF). Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI-73). 3. California, USA. pp. 11–17.

[10] Köll, Andreas; Hermann Kaindl (August 1992). "A new approach to dynamic weighting". Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92). Vienna, Austria. pp. 16–17.

[11] Pearl, Judea; Jin H. Kim (1982). "Studies in semi-admissible heuristics". IEEE Transactions on Pattern Analysis and Machine Intelligence. 4 (4): 392–399. doi:10.1109/TPAMI.1982.4767270. PMID 21869053. S2CID 3176931.

[12] Ghallab, Malik; Dennis Allard (August 1983). "A – an efficient near admissible heuristic search algorithm" (PDF). Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83). 2. Karlsruhe, Germany. pp. 789–791. Archived from the original (PDF) on 2014-08-06.

[13] Reese, Bjørn (1999). "AlphA*: An -admissible heuristic search algorithm". Archived from the original on 2016-01-31. Retrieved 2014-11-05.