On the Semantics of Queries over Graphs with Uncertainty

A thesis in the

Department of

Computer Science & Software Engineering

Presented in partial fulfilment of the requirements

for the degree of Master of Science

Department of *Computer Science and Software Engineering*

Concordia University

Montreal, Quebec, Canada

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Soyoung Kim

Entitled: On the Semantics of Queries over Graphs with Uncertainty

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Sabine Bergler Chair

Dr. Gosta G. Grahne Examiner

Dr. Hovhannes Harutyunyan Examiner

Dr. Nematollaah Shiri Supervisor

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date _____

Abstract

We study the semantics of queries over uncertain graphs, which are directed graphs in which each edge is associated with a value in $[0, 1]$ representing its certainty. In this work, we consider the certainty values as probabilities and show the challenges involved in evaluating the reachability and transitive closure queries over uncertain/probabilistic graphs. As the evaluation method, we adopted graph reduction from automata theory used for finding regular expressions for input finite state machines. However, we show that different order of eliminating nodes may yield different certainty associated with the results. We then formulate the notion of "correct" results for queries over uncertain graphs, justified based on the notion of common sub-expressions, and identify common paths and avoid their redundant multiple contributions during the reduction. We identify a set of possible patterns to facilitate the reduction process. We have implemented the proposed ideas for answering reachability and transitive closure queries. We evaluated the effectiveness of the proposed solutions using a library of many uncertain graphs with different sizes and structures. We believe the proposed ideas and solution techniques can yield query processing tools for uncertain data management systems.

Acknowledgements

The work on this thesis was challenging, but I have a deeper understanding of mathematics, programming, and engineering. The results of my research were not clear beforehand, and a lot of approaches were pursued to solve problems. I am very grateful to my supervisor, Dr. Nematollaah Shiri. My meetings with him helped me a lot and resulted in good discussions. I also would like to thank my parents Young Ran Yang and Tae Wan Kim, and my brother Kyeong Bum Kim for their never faltering support, love, and trust in me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Real world information is not always definite in the manner that we handle degrees of uncertainty rather than exact information which usually falls under true or false. Intuitively, a piece of data is uncertain if its truth is not established definitely. Such uncertain data exist since we are often uncertain about our observations and understanding in particular in the presence of noise and error [28, 15, 4]. In the age of large data, it occurs in many applications due to data collection processes, data preprocessing methods or privacy-preserving reasons [25, 11, 2]. In these applications, processing queries without considering this uncertainty information can lead to incorrect answers. But what is the correct result and what are the challenges to find one?

1.1 Problems and Motivation

A natural way to capture graph uncertainty is to represent them as probabilistic graphs [9, 12, 21, 20]. Probabilistic graphs are form of uncertain data, in which

the edges are annotated with certainty values or weights indicating their associated probabilities. In this research, we study the semantics of probabilistic graphs and that of queries over such graphs. An example of frequent such queries is the reachability problem [7, 6, 16]. The reachability problem is, given an uncertain graph G and a pair of nodes (i, j), to find the probability of a target node j being reachable from a source node i, which represents the certainty value connecting i to j. We show this probability of weight as $\omega(i, j)$. Intuitively, this is determined by considering all paths from i to j, finding the certainty value of each path, and "merging" or "aggregating" those certainties as one value for the connectivity of i to j. The fact that certainties are interpreted as probabilities, laws and formula of probability are applied when processing the reachability query. For answering the query, disjunction function $\oplus$ is used to "combine" certainties of multiple paths that have the same start and end points and conjunction function $\odot$ is used to aggregate certainties of the sequences of edges that form a path. Essentially we remove all the nodes that are on every path from i to j while combining the associated probabilities of the edges. This process can yield different results when considering different order of merging probabilities and elimination of nodes along the path using particular disjunction function for merge.

To illustrate the issues, consider the uncertain graph in Figure 1.1. Suppose we want to determine $\omega(i, j)$, that is the reachability value from node i to j in this graph.

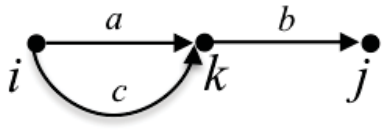


Figure 1.1: An uncertain graph.

As can be seen, there are two paths from i to j, i.e., $\langle a, b\rangle$ and $\langle c, b\rangle$. Suppose we use the probability independence mode *ind* as the disjunction function $\oplus$ and use the arithmetic product operation $\times$ as the conjunction function $\odot$. For the reachability $\omega(i, j)$ problem, we may get the following two values: $v_1 = (a \oplus c) \odot b$ or $v_2 = (a \odot b) \oplus (c \odot b)$, which are different in general, but which one is correct? To be more precise, suppose the weight associated with every edge in this graph is 0.5, that is, $a = b = c = 0.5$. If we use $ind(x, y) = x + y - xy$ as the disjunction function and use the product $\times$ as conjunction, we would obtain the different values $v_1 = 0.375$ and $v_2 = 0.4375$, respectively. It is important to note that while *ind* is a commutative and associative function over, it is not distributive over the conjunction function. Sevo [27] studies the reachability problem but considers *max* as the disjunction function $\oplus$. The function does not pose any challenge for computing $\omega(i, j)$ in the graph as the certainty of the result is independent of the order in which nodes are eliminated. This is because *max* is distributive over the product operation. As mentioned, when the weights are $a = b = c = 0.5$, performing the node elimination process using the *ind* mode as the disjunction $\oplus$, the computed probability value would not be unique, while using *max* we would get 0.25 in either case.

It can be even complicated when cycles are present in the input graphs. Figure 1.2 shows that when the input uncertain graph is acyclic, we face a challenge for answering the reachability problem even if we use *max* as the disjunction function.

For answering the reachability query, the enumeration of paths based on "state-elimination" technique proceeds a replacement of node v_j between start node v_s and end node v_f and its incident edges (v_i, v_j) and (v_j, v_k) by a new edge (v_i, v_k). This process essentially replaces any pair of multiple edges by a single edge and elimination

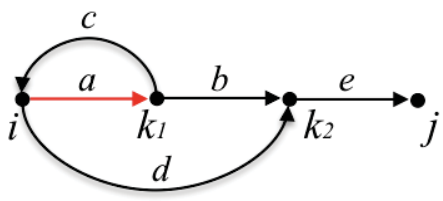


Figure 1.2: An example uncertain graph with cycle.

of any edges incidents to nodes between v_s and v_f until one edge remains which connects v_s to v_f with the considered disjunction and conjunction functions. In this uncertain acyclic graph, we have two cases of nodes elimination i.e., "k_1, k_2" and "k_2, k_1", which yield:

1. $(a \odot c)^* \odot (a \odot b \oplus d) \odot e$

2. $(a \odot c)^* \odot (a \odot b \odot e \oplus d \odot e)$

In both two computations above, edge $a(i, k_1)$ contributes multiple times, which results in overcomputing. The issue is "redundant" repeated computation in the sense that there exist shared paths contributing multiple times. Even though the reduction order does not affect the end results when using *max* as the disjunction function, overcomputation is not desired in the context of our correct semantics in that we avoid generating unnecessary repeated computations.

The above two examples show the challenges when processing reachability queries over uncertain graphs. The main question that may be raised at this point is, what is a "correct" value from a start node to an end node in an uncertain graph and how to compute it.

1.2 Thesis Contributions

The contributions of this thesis are as follows:

1. We study the semantics of uncertain graphs and show that different orders of graph reduction may lead to different results for reachability and transitive closure queries. This is used as a basis to formalize the notion of correct semantics for uncertain graphs and we propose a solution technique for reachability and transitive closure queries over such graphs.

2. We formulate correct semantics for uncertain graphs and propose the corresponding reduction techniques that in general involve several steps, depending on the possible patterns and components that may be present in the input. We propose a set of rules to assist in the reduction process. To validate our techniques, we use a large collection of uncertain graphs of different size and structures, generated and/or collected. We evaluate the performance of the proposed techniques for reachability and transitive closure queries in terms of effectiveness and scalability.

3. Our results indicate correctness of the implementation and effectiveness for the proposed solutions. Potential applications of the proposed solution include development of an extension of the SQL database query language to support transitive closure and reachability queries over uncertain graphs. This capability is needed for next generation systems for management of uncertain data.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 introduces a background and provides a survey of related literature. Chapter 3 studies the semantics of uncertain (probabilistic) graphs and investigates suitable reduction algorithms. Chapter 4 presents a design and implementation of the proposed algorithms followed by a complexity analysis. Chapter 5 illustrates the evaluation results of our proposed solutions using a library of uncertain graphs we created and/or collected, and compiled. Concluding remarks and possible future directions are discussed in Chapter 6.

Chapter 2

Background and Related Work

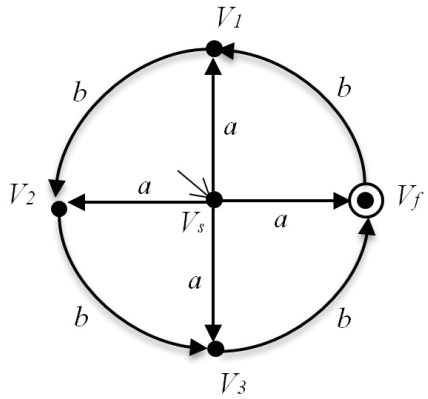
A graph $G = \langle V, E \rangle$ is a data structure where V is a set of objects, called vertices or nodes, and $E \subseteq V \times V$ is a set of edges of the form (x, y), indicating that nodes x and y are connected. A weighted graph is a digraph (V, E) in which every edge (x, y) is associated with a value $\omega(x, y)$, called its *weight* where ω is called a weight assignment function. In a directed graph (also called digraph), presence of an edge (x, y) in E means that node x is connected to node y. An uncertain graph $G = (V, E, \omega)$ is a weighted graph in which $w : E \rightarrow [0, 1]$ is the weight assignment function. Probabilistic graphs are special cases of uncertain graphs in which the weights are probability values. If the weight associated with an edge (x, y) is 1, it means that node x is certainty connected to node y. A weight 0, on the other hand, means that node x is not connected to node y.

In many applications, a frequent query problem [21, 12] in graphs is the reachability problem, which given a weighted graph and a pair of nodes (s, t), it asks whether there is a path from the start node s to the target node t, and if yes, what is the prob-

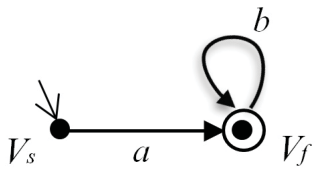
ability of this path, which we denote by $\omega(s,t)$. Answering this is challenging when such paths include cycles. To process reachability queries in probabilistic graphs, we need to identify all the paths between the given nodes and "combine" the probabilities of these paths, using desired conjunction and disjunction functions. Intuitively, a conjunction function combines the weights of the edges that are on the same path while a disjunction function combines the weights of parallel edges/paths that have the same endpoints. Given a graph G and a reachability query (s,t), we say that a node $v \in V$ is *relevant* to this query if v is on any path from s to t. Similarly, we say that an edge $e(v_1, v_2)$ is relevant to a reachability query $\omega(s,t)$ if $(v_1, v_2) \in E$, and v_1 and v_2 are both relevant to (s,t). To answer the reachability query $\omega(s,t)$, we keep s and t but eliminate every other nodes that are relevant to (s,t) one by one in a "disciplined" way, while combining the probabilities of the paths that are eliminated. This process essentially reduces the relevant part of G to a single edge connecting s to t, and computes the weight associated with this edge.

In automata theory, the state-elimination technique is used to minimize the number of states in a given deterministic finite state automaton (DFA). The use of state elimination has also been used to obtain a regular expression for a given finite state automaton. While the minimal equivalent DFA is unique, different order of eliminating states may yield different but equivalent regular expressions. For instance, for the FA shown in Figure 2.1(a), we get two equivalent regular expressions $r_1 = ab^*$ and $r_2 = a(\lambda + b + b^2 + b^3)(b^4)^*$, which represent the same set of strings. This example shows while state elimination technique is useful in automata theory, it is inadequate in our context of probabilistic graphs.

If we view a and b as probabilities, we get different weights obtained from two



(a) A finite automaton M which yields the regular expression $(a + ab + ab^2 + ab^3)(b^4)^*$



(b) Reduced equivalent FA M which yields the regular expression ab^*

Figure 2.1: Digraphs with regular expressions.

regular expressions for the reachability probability of V_s to V_s. Let $a = 0.5$ and $b = 0.4$. For r_1, $(a \oplus ab \oplus ab^2 \oplus ab^3)(b^4)^*$, we get 0.016481 for using *ind* for $\oplus$ and get 0.024 for using *max*. Here we considered only one iteration of the loop from V_f to V_f. This is good enough to show that the results are different even when we consider one iteration for the infinite loop denoted by use of $*$. For the "reduced" regular expression r_2 which the reduced FA in Figure 2.1(b) yields, ab^*, we get $0.5 \times 0.4 = 0.2$ in both *ind* and *max* modes, considering the aggregated weight of a cyclic path (b, b) of length-one is computed as $b^{\lfloor \frac{1}{1-b} \rfloor}$. This is a different value compared to the four different paths between V_s and V_f i.e., $(V_s, V_1, V_2, V_3, V_f), (V_s, V_2, V_3, V_f), (V_s, V_3, V_f)$, and (V_s, V_f). The proposed formula for cycle $\omega^{\lfloor \frac{1}{1-\omega} \rfloor}$, where ω is the probability of the cycle node, is explained in Section 3.

Computing and maintaining the weights in cyclic probabilistic graphs can be complicated since there is no basics to consider a "right" way for computing the weights

in such graphs. Intuitively, nodes in cycles should be removed earlier than others in the graphs. Here the term *fuse* is used to mean "combine all element nodes in a cycle of empty nodes to be a representative solid node" to solve cyclic path problems. We then find a correct way to compute it. To see this, consider the cyclic probabilistic graph in Figure 2.2 and we want to compute $\omega(i, j)$. Conforming to state-elimination

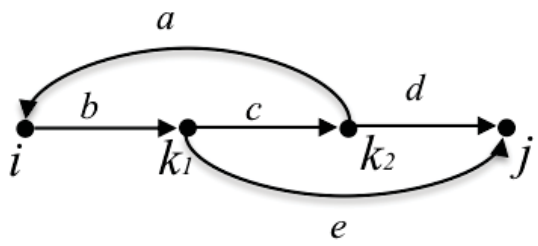
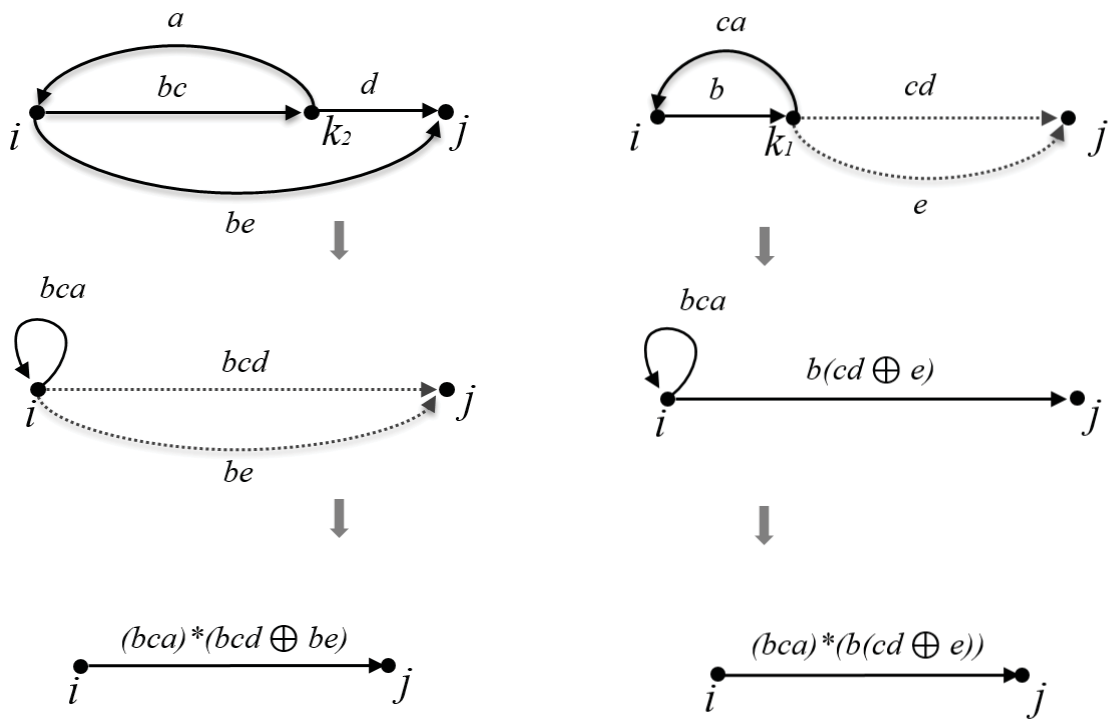


Figure 2.2: Another example probabilistic graph with cycle.

to obtain $\omega(i, j)$, we have two choices: 1) eliminate k_1 first by aggregating and maintaining the weights of incident edges to k_1, and then eliminate k_2 to get the final result by computing and maintaining newly generated weights through reduction, 2) eliminate k_2 first and then k_1 while computing as described in case 1. For the reduction order $'k_1, k_2'$, the combined weights from i to j is $(bca)^*(bcd \oplus be)$, whereas for the reduction $'k_2, k_1'$ we get $(bca)^*b(cd \oplus e)$, described in Figure 2.3.

We are facing two issues to perform the correct computation. The first issue is which order results in least common sub-expressions. This affects the result when using the *ind* mode as the disjunction (aggregation), as the *max* mode returns a unique returned result, that is, independent of any specific order. The other issue we face is *"Has any of the resulting expressions correctly captured the semantics of the underlying uncertain in the reduction process?"*. The fact that some weights such as



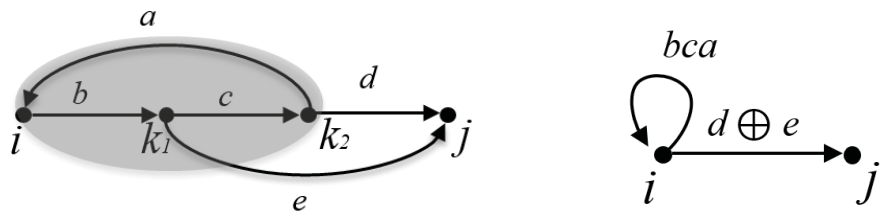
(a) Case 1: (k_1, k_2)

(b) Case 2: (k_2, k_1)

Figure 2.3: Comparison of different order of node reductions.

b or c contribute multiple times to the result, we have to be careful with such over-computing problem when edges are shared more than once along the paths between the endpoints. Intuitively, if nodes v_i and v_j are involved in a same cyclic path, we consider them as "indistinguishable" nodes, and such nodes are combined into a single "solid" node. For example, Figure 2.4(a) includes a cyclic path (i, k_1, k_2, i), shown in grey. Using the proposed technique based on SCCs, this cycle is reduced to a solid node i where the weight of i to itself is multiplication of the associated weights on sequences of edges involved in the cyclic path, which is bca, shown in Figure 2.4(b). Our proposed reduction method considers nodes i, k_1, and k_2 to be equivalent, and

the weights of connectivity from i to j, denoted as $\omega(i, j)$, is $(bca)^*(d \oplus e)$.



(a) Combining (i, k_1, k_2) as a solid node i (b) The reduced graph

Figure 2.4: Proposed reduced graph based on SCCs.

2.1 Shortest Paths in Weighted Automata

In automata theory, finite state automata (FA) is represented as a directed graph where nodes represent states and the edges represent transitions [14]. The set of nodes in a FA includes one start state and any number of final states. Weighted automata are finite automata with numerical weights on transitions in which each transition carries some weight in addition to the input and output labels, which are used in many applications such as text, speech and image processing [18, 3]. In the particular case of a weighted automaton over the probability semiring, nondeterministic choice is replaced by probability distributions on successor states [5]. Such automata over the probability semiring is modeled based on a Bayesian approach where the analysis asymptotically gives probability 1 to the model that is as close as possible to the true model, while we handle independent uncertainty value associated with each edge in a directed graph, which is inadequate in our probabilistic graphs.

To see this, consider a finite automaton M in Figure 2.1. Note that any automaton accepts a set of strings that can be represented by a regular expression and we can use a state-elimination process to convert M into a regular expression r. The components defining an FA M include a set of alphabet $\{a, b\}$, and a finite set of states $\{V_s, V_1, V_2, V_3, V_f\}$, where V_s and V_f are the initial and final states, respectively. Following the state elimination technique, we get a regular expression ab^* from the reduced FA. We can also get $(a + ab + ab^2 + ab^3)(b^4)^*$, which is an equivalent regular expression for M. As mentioned earlier, while the regular expressions obtained from state-elimination process are equivalent, they may not be considered as equivalent when the labels on the edges are probabilities. For example, $(a \oplus ab \oplus ab^2 \oplus ab^3)(b^4)$ and ab^* are equivalent as regular expressions but they yield different probabilities for reaching V_f from V_s. If we apply the proposed cycle reduction to the graph of Figure 2.1(a), then we get $(a \oplus a \oplus a \oplus a)(b^4)^* = 0.024$ and $max\{a, a, a, a\} \times (b^4)^* = 0.0128$ for *ind* and *max*, respectively, by replacing the cyclic path of V_1, V_2, V_3, and V_f by a single solid node, shown in Figure 2.5.

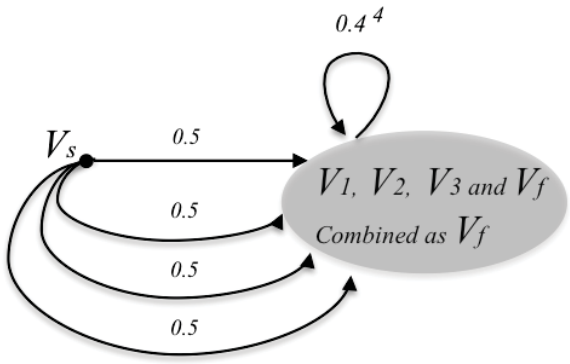


Figure 2.5: The proposed regular expression for FA M.

We can associate finding the least common sub-expressions in computing the reachability query problem with the shortest path problem which finds a path between two nodes in a graph such that the *combined* weights of its constituent edges is minimized [17], that is, it yields the minimal cost of the path. When the weights are probabilities we must avoid overcomputing probabilities. We propose a graph reduction algorithm that considers an order in which the nodes are eliminated and their weights are combined to get the minimal weight.

2.2 Strongly Connected Components

A walk in a digraph is a sequence of nodes, that can be traversed in order to move from one node to another. More formally, a walk of length k from node v_0 to node v_k is a non-empty subgraph $W = (V, E)$, where $V = \{v_0, v_1, .., v_k\}$, $E = \{(v_0, v_1), (v_1, v_2), .., (v_{k-1}, v_k)\}$, where $w(e)$ the weight associated with edge $e = (v_i, v_j)$.

Definition 2.1. Let $E_i = (v_i, v_{i+1})$ be the edges in G, for $i \in [1, k]$. The sequence $P = (v_0, v_1, \ldots, v_k)$ is a walk of length k from v_0 to v_k in that v_i is adjacent to v_{i+1}, for all $i \in [1, k-1]$.

If nodes v_i and v_j are connected, that is, there is a direct path "(v_i, v_j)" and at least one of indirect path "$(v_i, .., v_j)$" between v_i and v_j, weights on the edges in such paths should be combined to compute $\omega(v_i, v_j)$ with considered disjunction functions. The length of a walk P is denoted by $|P|$, in this case $|W| = k$. We say that a path P is cyclic when the path starts and ends at the same node with $|W| \geq 1$. We also say that nodes v_i and v_j are strongly connected being in a same strongly connected

component (SCC) class when v_i and v_j are in a same cyclic path [10], which means there is a path connecting these nodes, regardless of the start node being v_i or v_j. Here we want to formalize a relation of such nodes in a same SCC. To consider the way in which nodes in a same SCC are related to each other, one needs the notion of "indistinguishable" between nodes. Informally, the transitive closure of a digraph G is itself a digraph G^* such that the nodes of G^* are the same as the nodes of G, and G^* has an edge (v_i, v_j), denote by $v_i \to v_j$, whenever G has a directed path from v_j to v_i written $v_j \to v_i$, including the case where (v_i, v_j) is an edge of the original G. v_i is indistinguishable with v_j, if $v_i \to v_j$ and $v_j \to v_i$, written $v_i \leftrightarrow v_j$ such that v_i and v_j are in the same SCC . More formally, the transitive closure of G is a graph $G^* = (V, E^*)$ such that for all v_i, v_j in V there is an edge (v_i, v_j) in E^* if and only if there is any path from v_i to v_j in G. If $(v_i, v_j) \in G^*$ and $(v_j, v_i) \in G^*$, then nodes v_i and v_j become indistinguishable each other, and such nodes are replaced by a single representative solid node in a way v_i and v_j are not distinguishable.

Definition 2.2. For all nodes v_i,v_j in V, v_i and v_j are *indistinguishable* if $(v_i, v_j) \in G^*$ and $(v_j, v_i) \in G^*$. That is, we say that v_j is reachable from v_i and v_i is reachable from v_j as well.

We then write $v_i \to v_k$ if $v_i \to v_j$ and $v_j \to v_k$. Further if $v_i \leftrightarrow v_j$ then $v_j \leftrightarrow v_i$. That is, v_i and v_j are indistinguishable. Tarjan[29] introduced an algorithm that identifies all the maximal SCCs of a digraph. The problem, however, is that for the reduction algorithm, we have to find all the basic cycles in a graph and "combine" them in some order. Tarjan's algorithm itself is not preferred for the fuse step since it only returns the maximal SCCs of G whereas complex cycles involving many nodes

have to be fused in ascending order of the length of elementary cycles individually. We propose an algorithm that proceeds as designed by finding and keeping track of all the cycles by *fusing* them, which returns the correct weight.

As shown in Figure 2.4(b), the cycle path (i, k_1, k_2, i) are combined as a solid node i, and it yields $(bca)^*(d \oplus e)$ which has no common sub-expressions.

Chapter 3

The Proposed Algorithm

We present a set of patterns of paths and its corresponding rules to reduce a probabilistic graph. Our reduction algorithm is carried out through node reduction and edge aggregation in an input graph G until one edge, connecting a source node v_s to a target node v_f, remains. It induces the relevant subgraph of G and finds the reachability weight as $\omega(v_s, v_f)$. When the subgraph has cycles, it reduces cyclic to acyclic and then determines $\omega(v_s, v_f)$. This is done by repeatedly applying reduction rules to the reduced subgraph until no more rules can be applied to obtain the aggregated weight on a single edge (v_s, v_f). The proposed solution techniques underlie the proper order of nodes elimination based on the notion of least common sub-expressions.

3.1 Proposed solution

The key point is to find least common sub-expressions in the final aggregated probabilities, that is, avoiding unnecessary redundant computations. On the one hand, if two or more nodes in a probabilistic graph belong to the same SCC class [19], then

such nodes are considered *indistinguishable* and hence converted into a *solid* node which replaces the corresponding SCC. By treating them indistinguishable, we can avoid overcomputing problems. On the other hand, we strictly regulate the order in which nodes are eliminated. The proposed algorithm is partitioned into three phases. In the first phase, it finds the relevant subgraph of an input G if a source node v_s and a target node v_f in G. The second phases finds the cycles in G', if any, and reduce them into "solid" nodes in a disciplined way. This generate an acyclic graph G''. The last phase requires nodes reduction process in G''. It then computes the reachability value as $\omega(v_s, v_f)$.

3.1.1 Finding the Relevant Subgraph

The first phase of the reduction algorithm is to get the relevant subgraph. We first obtain the set $R(v_s)$ of nodes that are reachable from v_s. To get the set V' of nodes are only relevant to (v_s, v_f), for a node v in $R(v_s)$, if v is contributing to v_f the v is included in V', that is the subgraph of $G = (V, E)$ induced by $V' \subseteq V$, which is $G' = (V', \{(i, j)|(i, j) \in E,\ for\, i, j \in V'\})$.

3.1.2 Reducing Cycles in the Relevant Subgraph

The second phase detects cycles and reduces them in the relevant subgraph G'. The nodes that are on a cyclic path form a strongly connected component, we then replace them by a single solid node. This phase can be omitted if G' is acyclic.

3.1.3 Reducing Nodes in the Reduced Relevant Subgraph

Eliminate all the relevant nodes in G'' but v_s and v_f, using the reduction rules/patterns. This phase eliminates all the nodes but v_s and v_f one by one until one single edge remains connecting v_s to v_f. Finally we get the reachability weight as $\omega(v_s, v_f)$.

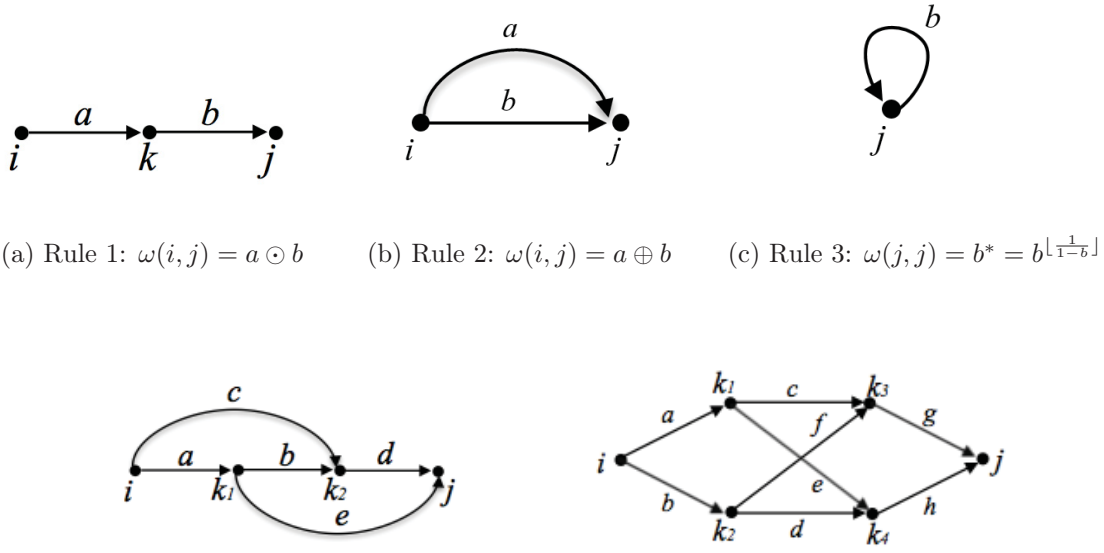
3.2 Types of Nodes

We have two considerations on classifying node types in the presence of cycles in an input graph G. Suppose $G = (V, E)$ has cycles. If a node $v \in V$ is involved in a cyclic path, we say that v is a *cyclic* node. Now consider a graph G that does not include any cycles. Here we denote the number of incoming and outgoing edges of a node v by $deg^-(v)$ and $deg^+(v)$, respectively. Depending on the number of incident edges on a node, we classify the node in an input graph G into five types: *source*, *sink*, *isolated*, *sequence*, and *split*. A source node is a node without any incoming edges, while a sink node is a node without outgoing edges. An isolated node is a node having no incident edges. A node is called a sequence node if it has exactly one incoming and one outgoing edge. Otherwise, a node v is called a split node, that is $deg^-(v) + deg^+(v) > 2$. More specifically, when the summation of incoming edges and outgoing edges of v is greater than 2, we call node v "split" for $deg^-(v) \geq 1$ and $deg^+(v) \geq 1$. Note that phase 1 removes all the "isolated" nodes since isolated node is not connected with any node in G while phase 2 removes all "cyclic" nodes.

We use this categories of nodes to classify the types of paths for defining patterns in that each pattern has its own corresponding rule.

3.3 Types of Patterns and Rules

There are five patterns and its corresponding rules to compute the weight of each pattern of path: *chain*, *choice*, *cycle*, *merge*, and *mesh* as following.



(a) Rule 1: $\omega(i,j) = a \odot b$ (b) Rule 2: $\omega(i,j) = a \oplus b$ (c) Rule 3: $\omega(j,j) = b^* = b^{\lfloor \frac{1}{1-b} \rfloor}$

(d) Rule 4: $(a'b' \oplus c)d' \oplus a'(b'd' \oplus e)$ (e) Rule 5: $(a'c \oplus b'f)g \oplus (a'e \oplus b'd)h$

Figure 3.1: Basic node reduction rules: *chain, choice, cycle, merge*, and *mesh*.

Rule 1: Chain Rule. The chain rule has higher priority over rule 2. If a node has exactly one incoming and outgoing edge, we can remove such node by aggregating incident edges on that node with the considered conjunction function $\odot$, shown in 3.1(a).

Rule 2: Choice Rule. When there are more than one of paths between two nodes in parallel, weights on such paths are aggregated with the considered disjunction function $\oplus$ to be a single edge connecting two endpoints, shown in 3.1(b).

Rule 3: Cycle Rule. The cycle rule defines a weight computation for a cyclic path. Computing the weight of a cyclic path can be complicated when the input graph includes nested cycles. Nodes involved in any cyclic path are reduced to a solid self-loop in a disciplined way by *chain* and *choice* rules. Once a single self-loop node remains we apply the cyclic equation for length-one path. The proposed formula in 3.1(c) is used to compute $\omega(j, j)$ by applying geometric sequence summation equation, explained in Section 3.2.

After applying the reduction rules mentioned above, to get $\omega(v_s, v_f)$ in G', there are no more of *sequence* nodes. From that, v_s is a *source* node, v_f is a *sink* node, and the remaining nodes are *split*. In such case, more than one node should be eliminated in a single node reduction phase. There are two cases: one with a unique topological sort, and the other with multiple topological sort in nodes.

Rule 4: Merge Rule. The merge rule handles the first case where all nodes are removed together at once. It combines parallel paths correctly under different possible scenarios, shown in 3.1(d).

Rule 5: Mesh Rule. The mesh rule is applied in the other case when "there are more than one of topological sort", shown in 3.1(e).

By iteratively applying these reduction rules above, we develop a reduction algorithm that computes the weight associated with every pair of path $\omega(i, j)$ in the graph without any overcomputation.

As for rule 3 corresponding to cycle pattern, to see how to compute the weight associated with a cyclic path, consider a self-loop from node j to itself, and suppose we want to determine $\omega(j, j)$. For this, we use the formula $x^{\lfloor \frac{1}{1-x} \rfloor}$ used (or defined) in [27], where x is the weight of the edge (j, j). Consider again the graph M in Figure

2.1(b). The aggregated weights $\omega(V_s, V_f)$ for the relevant paths from V_s to V_f are obtained as follows:

$$a,\ a \oplus (a \odot b),\ a \oplus (a \odot b) \oplus (a \odot b \odot b),\ \ldots,\ a \cdot (1 \oplus b \oplus bb \oplus bbb \ldots \oplus b^n).$$

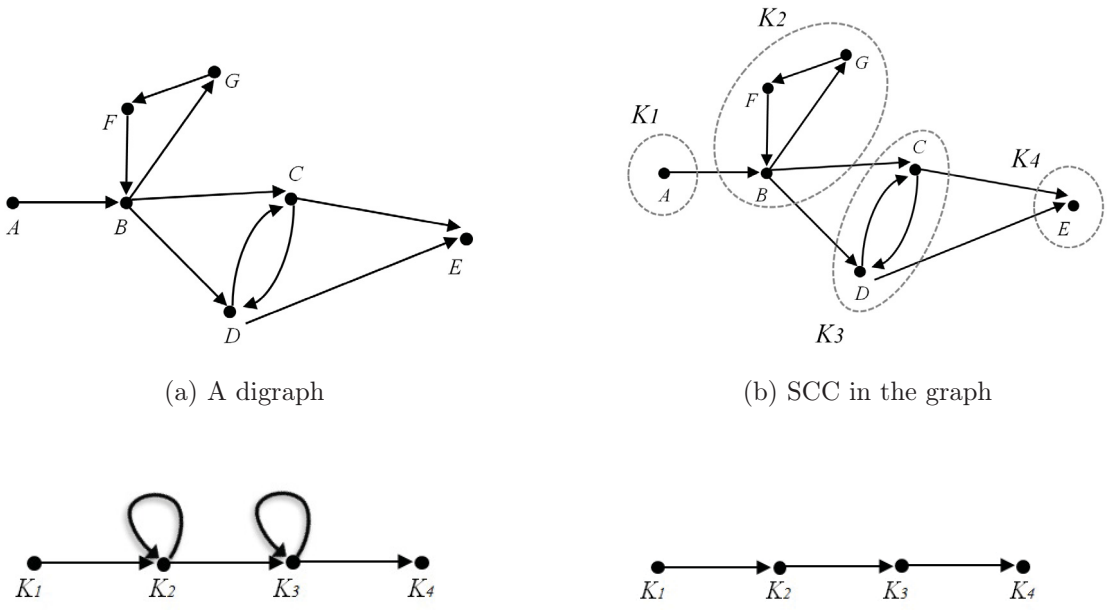
A regular expression for reduced FA M is ab^*, which is $a \odot b^*$. Assume it converges within a finite number n of iterations. In the limit when n approaches infinity, we get the result: $a \odot (1 \oplus b)^n$, which is different in general from the expected, correct result $a \odot b^*$, even if $b^n = b^*$ in regular expression terms. Here, we use the geometric power series to find the most probable number of n. Essentially, we define the number of loops n on edge (v_f, v_f), and the weight of edge $\omega(v_f, v_f) = b$. As for $(a + ab + abb + abbb \ldots + ab^n)$, using geometric sequence with the first term $a_1 = 1$ and the common ratio r, the sum of the first n terms is given by:

$$S_n = \frac{a_1(1 - r^n)}{1 - r}$$

In the special case where $|r| < 1$, as n goes to infinity, S_n converges to $\frac{a}{1-r}$ for $-1 < r < 1$. Here we use this formula where r is associated with probability and its range satisfies the condition. The most promising number of loops for the weight b is $\frac{1}{1-b}$, and the floor of this weight, $\lfloor \frac{1}{1-b} \rfloor$, is the number of loops, where $\lfloor x \rfloor = max\{k \in \mathbb{Z} | k \leq x\}$, the largest integer less than or equal to x.

Example Figure 3.2 shows a digraph containing 7 nodes and its reduced graph, SCCs, and condensed graph based on automata theory. By Definition 2.2, there are four strong components in the graph: $K_1 = \{A\}, K_2 = \{B, G, F\}, K_3 = \{C, D\}$, and

$K_4 = \{G\}$. We get B, G, and F are indistinguishable nodes. In a similar way, C and D are considered indistinguishable. $\omega(A, E)$ is computed from $e_{K_1K_2} \odot {e_{K_2}}^* \odot e_{K_2K_3} \odot {e_{K_3}}^* \odot e_{K_3K_4}$. To put it more clearly, $e_{AB} \odot (e_{BG} \odot e_{GF} \odot e_{FB})^* \odot (e_{BC} \oplus e_{BD}) \odot (e_{CD} \odot e_{CD})^* \odot (e_{BC} \oplus e_{BD})$.



(a) A digraph

(b) SCC in the graph

(c) Condensation based on automata theory

(d) Combining cycles into solid nodes

Figure 3.2: A digraph and its reduction phases.

3.4 Definitions and Notation

We need the following definitions and notation for our graph reduction algorithm.

Definition 3.1. Let $G' = (V', E')$ be the subgraph of a graph $G = (V, E)$ where V' is a set of nodes that are relevant to the endpoints of G and E' is a set of edges

(i, j) for nodes i and j in V'. For node i, we define $\mathcal{R}(i)$ to be the set of nodes in V' which are reachable from i such that $\mathcal{R}(i) = \{j \in V | i \to j\}$. The kleene plus on $\mathcal{R}(i)$ defines the union of $\mathcal{R}(i)$ and $\{i\}$ itself, that is, $\mathcal{R}^+(i) = \mathcal{R}(i) \cup \{i\}$. Further, the set of relevant "intermediate" nodes to (i, j) can be defined as $\mathcal{R}(i) - \mathcal{R}^+(j) = \{k \in V | i \to k \to j\}$ and the relevant set of reachable nodes from i to j is defined as $\mathcal{R}^+(i) - \mathcal{R}(j) = \{k \in V | i \to k \to j\} \cup \{i, j\}$.

Suppose G' is the relevant subgraph of G to nodes (v_s, v_f). Let S be the set of relevant "intermediate" nodes between v_s and v_f, and we want to compute $\omega(v_s, v_f)$. To compute $\omega(v_s, v_f)$, all nodes in S should be removed by nodes reduction rules. Beforehand, we give formal definitions of node types. Note that the degree of a node v is the number of edges incidents on that node, denoted as $deg(v)$. It is important to point out that v_s is a *source* node and v_f is a *sink* node while S does not include any *isolated* nodes since G' includes only relevant nodes to (v_s, v_f). Then node v in S can be classified either *sequence* or *split*.

Definition 3.2. Suppose S is a set of relevant intermediate nodes between a pair of nodes in graph G. Node $v_i \in S$ is a *sequence* node if v_i is on the chain path in that it has only one incoming edge and one outgoing edge, that is $deg(v_i) = 2$. Otherwise, v_i is a *split* node.

By Definition 3.2, a node $v_i \in S$ is a "sequence" node if v_i has only one incoming edge and outgoing edge whereas v_i is a "split" node when $deg(v_i) > 2$ then. v_i in S can be removed by rules 1 and 2 if and only if v_i is a **sequence** node. By reduction of v_i, a **split** node $v_j \in S$ may become **sequence** and then can be removed by rules 1 and 2. To see this, suppose $S = \mathcal{R}(s) - \mathcal{R}^+(t) = \{i, j\}$ in the probabilistic graph

shown in Figure 3.3. We have **sequence** node i and **split** node j by Definition 3.2, since $deg(i) = 2$ and $deg(j) = 3$. After i is eliminated by rules 1 and 2, $deg(j)$ becomes 2. The fact that j is a **sequence** node, j can be removed by rules 1 and 2.

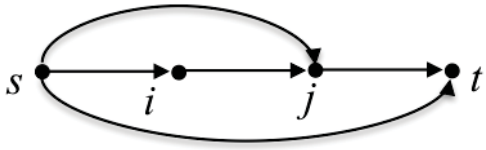


Figure 3.3: An example graph.

3.5 Simultaneous Nodes Reduction

After applying the reduction rules 1, 2, and 3 until no more these three rules are not used immediately, the remaining nodes in S are all split nodes. In this case, we use rules "merge" or "mesh" to eliminate multiple nodes in one step instead of eliminating one by one in some order. It is important to point out that patterns of path in chain, choice, and cycle do not have any topological sort in nodes. However, we need complex cases which may have one or multiple topological ordering of nodes in S. We take this aspect into accounts on the reduction process where we need two additional rules, which we call "merge" and "mesh", respectively.

3.5.1 Topological Sort in Reduction

A topological sort of a partially ordered set of nodes in an acyclic digraph such that if there is a path from node v_i to node v_j in the graph, v_i appears before v_j in the list. Reduction of "multiple" split nodes is done based on the topological ordering.

The output of the topological sort may not be unique, that is, it can have multiple solutions depending on the type of algorithm used for sorting.

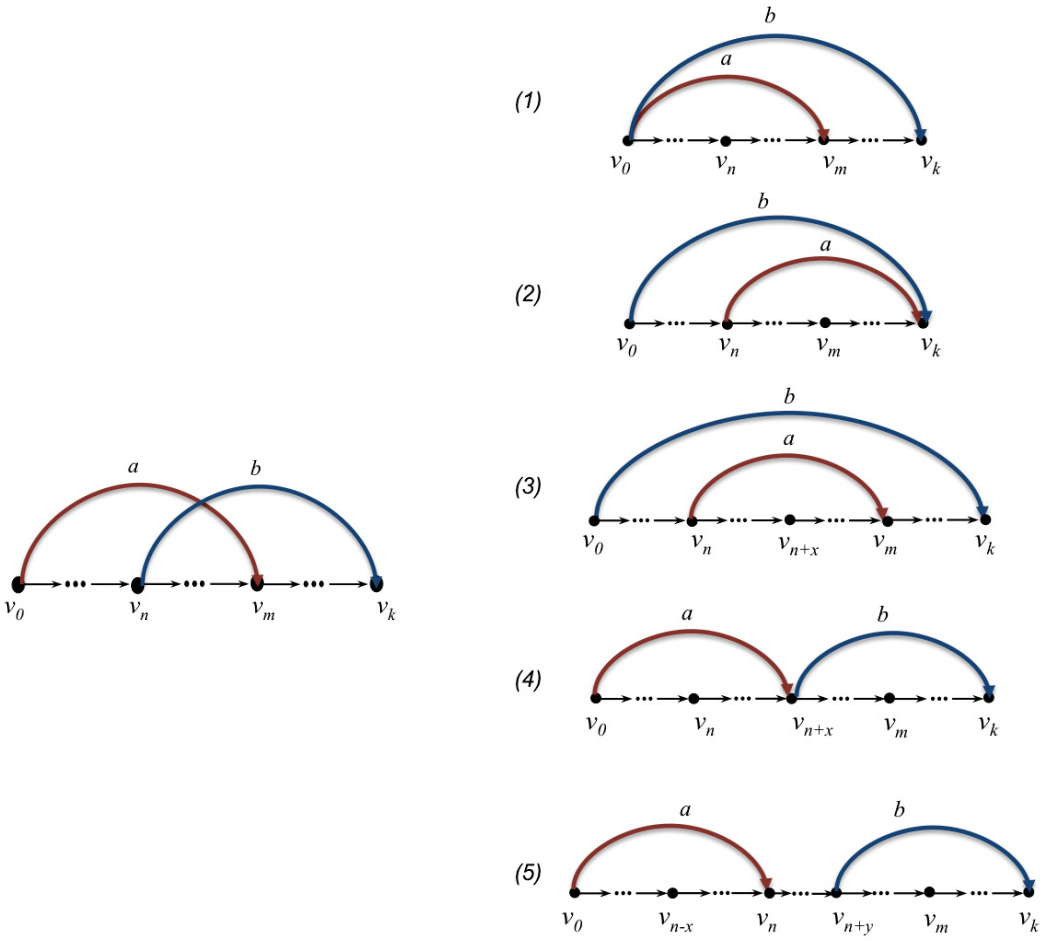
Suppose G' is a graph which is reduced by rules 1, 2, and 3 in previous steps of the reduction process. Then remaining intermediate nodes between the endpoints are only split. In this case, single node reduction results in multiple computations for shared edges. To get a correct value on a basis of least common sub-expressions in paths, we eliminate such nodes in one step. The result of topological sort of G can be unique or multiple outputs. If the output is unique, meaning every node in V has its own unique level, we use rule 4 "merge" for reduction. Otherwise, we use rule 5 "mesh" to reduce nodes in the same level one by one.

3.5.2 Mergeable Edges and Paths

The method we present here relies on the definition of "mergeable" among *bridge* edges. We formalize "bridge" edges and paths based on the notion of "mergeable" by partitioning a set of "all paths" into disjoint mergeable path sets. In the phase of simultaneous nodes reduction, a graph is acyclic since cycles are reduced in the previous phases, denoted by $G_{\mathcal{A}}$ from now on.

3.5.2.1 Properties of mergeable sets

Suppose L is the partially ordered set of nodes in a graph $G_{\mathcal{A}}$ such that $L =< v_0, \ldots, v_n \ldots, v_m, \ldots, v_k >$. The probabilities of the edges (v_0, v_m) and (v_n, v_k) are follows: $\omega(v_0, v_m) : a$ and $\omega(v_n, v_k) : b$. We want to determine edges (v_0, v_m) and (v_n, v_k) in Figure 3.4 are "mergeable" in the context of our notion of mergeable edges. Informally, edges a and b are mergeable when:



(a) Nonmergeable (b) Mergeable

Figure 3.4: A relation between edge a and edge b in the context of "mergeable".

1 a and b share a node: Cases (1),(2), and (4)

2 a and b do not share any node

($f(x)$ *is a set of relevant nodes to* x):

1) $f(a) \cup f(b) = f(a)$ or $f(a) \cup f(b) = f(b)$: Case (3)

2) $f(a) \cap f(b) = \emptyset$: Case (5).

Essentially two edges are "nonmergeable" if those edges are overlapped in the topological ordering-layout, shown in Figure 3.4(a). Hence, edges (v_0, v_m) and (v_n, v_k) are nonmergeable. We explain more formally by giving definitions. The term *mergeable* is used to describe multiple paths that have the same endpoints are capable of being merged into a single path in terms of computing the weights in such parallel paths.

Definition 3.3. Given $G_{\mathcal{A}} = (V, E)$, for an edge (i, j) in E, (i, j) is *bridge* if there are more than one of paths between i and j that is $\mathcal{R}^+(i) - \mathcal{R}(j) \neq \{i, j\}$. Let $\ell_G(E)$ be the set of such bridge edges in $G_{\mathcal{A}}$. (i, j) and another edge (u, v) in $\ell_G(E)$ are *nonmergeable* if the intersection of the relevant nodes set to (i, j) and the relevant node sets to (x, y) is not empty and $u < j$ or $i < v$ is true for $i \rightarrow j, u \rightarrow v$ then $(i, j), (u, v)$ are *nonmergeable*, otherwise (i, j) and (u, v) are *mergeable*.

To sum up, we say that edges (i, j) and (u, v) in the set of bridge edges $\ell_G(E)$ are **mergeable** if $(\mathcal{R}^+(i) - \mathcal{R}(j)) \cap (\mathcal{R}(v) - \mathcal{R}(u)) \neq \emptyset$ or $\mathcal{R}^+(i) - \mathcal{R}(j)$ is a proper subset/superset of $\mathcal{R}^+(v) - \mathcal{R}(u)$, otherwise (i, j) and (u, v) are **nonmergeable**.

A path in $G_{\mathcal{A}}$ is a sequence of nodes $(V_0, V_1, .., V_k)$ such that $(V_i, V_{i+1}) \in E$ for $0 \leq i < k$. Note that all the intermediate nodes between two endpoints in $G_{\mathcal{A}}$ are "split". The distance from V_i to V_j, denoted $d(V_i, V_j)$, is the length of a longest walk from V_i to V_j, instead of shortest path in a general term. Consider a set of all possible paths between a pair of nodes (s, t), denoted by $P = \{P_1, P_2, .., P_m\}$. In our algorithm, we are only interested in path $P_i \in P$ that has "at most one" bridge edge. Suppose a set $\mathcal{P}_{st} = \{P_1, P_2, .., P_n\}$ where a path P_i has only one bridge edge to maximum. We then divide into the two sets $\mathcal{BP}_{st}$ (a set of paths, having "one" bridge edge between s and t) and $\mathcal{LP}_{st}$ (a set of paths including a sequence of length-one

edges between s and t that is "zero" bridge edges):

$$\mathcal{BP}_{st} = \{P_i \in \mathcal{BP}_{st} | \forall e \in \ell_{G_\mathcal{A}}(E),\ n(e) = 1\}$$

$$\mathcal{LP}_{st} = (V_i, V_j) \in E \text{ in } \mathcal{LP}_{st},\ d(V_i, V_j) = 1.$$

In the set $\mathcal{BP}_{st}$, P_i is a sequence of nodes with the form of $(V_{m-d(s,V_m)}, \ldots, V_m, V_{m+\theta}, \ldots, V_{m+\theta+d(V_{m+\theta},t)})$ with length "$d(s, V_m) + \theta + d(V_{m+\theta}, t)$", where $(V_m, V_{m+\theta})$ is a bridge edge with the length θ and m is a node index of topological ordering of $G_\mathcal{A}$. We can say that $\mathcal{LP}_{st}$ is a set of the longest path between s and t. To see this, consider the following graph.

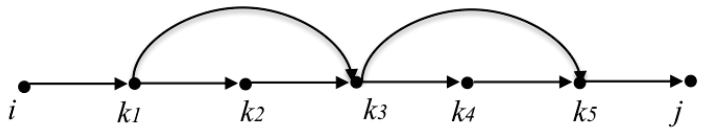


Figure 3.5: An example graph.

We get the set of all paths to (i, j), $P = \{(i, k_1, k_2, k_3, k_4, k_5, j), (i, k_1, k_3, k_4, k_5, j), (i, k_1, k_2, k_3, k_5, j), (i, k_1, k_3, k_5, j)\}$. We get $\mathcal{P}_{ij} = \mathcal{BP}_{ij} \cup \mathcal{LP}_{ij}$:

$$\mathcal{BP}_{ij} = \{(i, k_1, k_3, k_4, k_5, j), (i, k_1, k_2, k_3, k_5, j)\}$$

$$\mathcal{LP}_{ij} = \{(i, k_1, k_2, k_3, k_4, k_5, j)\}$$

As can be seen, path $(i, k_1, k_3, k_5, j) \in P$ is excluded since it has two bridge edges (k_1, k_3) and (k_3, k_5). It is important to note that $|\mathcal{LP}_{ij}|$ can be more than one depending on the number of topological sort of the graph. It is discussed in Section.

The reason of having the two sets separately, we need $\mathcal{BP}$ to determine mergeable paths while $\mathcal{LP}_{ij}$ is index keys to relevant nodes for bridge edges. For example, a

set of relevant nodes to (k_1, k_3) is a sequence of nodes starting from k_1 to k_3, which is $\{k_1, k_2, k_3\}$. Now the question is how paths in $\mathcal{BP}$ can be merged to compute aggregated weights in a single reduction iteration? We divide into $k - cuts$ of paths in $\mathcal{BP}$ in that any path in a same cut is mergeable with each other. Suppose $G_{\mathcal{A}}$ connecting node s with node t and $\ell_{G_{\mathcal{A}}}(E) = (u_i, u_{i+1}), (v_i, v_{i+1})$. We then have $\mathcal{BP}_{st} = \{P_1, P_2\}$. Suppose $P_1 = (s, \ldots, u_i, u_{i+1}, \ldots t)$ and $P_2 = (s, \ldots, v_i, v_{i+1}, \ldots t)$. In order to determine whether P_1 and P_2 are mergeable paths to join in a same group, we need to check whether (u_i, u_{i+1}) and (v_i, v_{i+1}) are mergeable beforehand.

Definition 3.4. Given a probabilistic graph $G_{\mathcal{A}}$ and a pair of nodes (s, t), suppose $\{e_i, e_j\} \in \ell_{G_{\mathcal{A}}}(E)$. We get the set $\mathcal{BP}_{st}$ of paths between s and t where $|\mathcal{BP}_{st}|$ is 2 since $|\ell_{G_{\mathcal{A}}}(E)|$ is 2. Let $\mathcal{BP}_{st}$ be $\{P_1, P_2\}$ where P_1 and P_2 hold e_i and e_j, respectively. P_1 and P_2 are *mergeable paths* if only if e_i and e_j are *mergeable*.

Paths P_i and P_j can be combined as one path and reduced by rules 1 and 2, if P_i and P_j are *mergeable*. To see this, consider again the graph G in Figure 3.3 and we want to compute $\omega(s, t)$, where the probabilities of the edges are follows: $\omega(s, i) : a$, $\omega(i, j) : b$, $\omega(s, j) : c$, $\omega(j, t) : d$, and $\omega(s, t) : e$. The set S of relevant nodes to (s, t) is $\mathcal{R}(s) - \mathcal{R}^+(t) = \{i, j, k\}$. By Definition 3.3, we get $\ell_G(E) = \{c, e\}$, derived from the fact that (s, i) is not bridge in that $\{s, i\} = \mathcal{R}^+(s) - \mathcal{R}(i)$. Similarly, it is easily verified that (i, j) and (j, t) are not bridge as well. Conversely, (s, j) and (s, t) are bridge since $\{s, j\} \neq \mathcal{R}^+(s) - \mathcal{R}(j) = \{s, i, j\}$ and $\{s, t\} \neq \mathcal{R}^+(s) - \mathcal{R}(t) = \{s, i, j, t\}$. We also know that (s, j) and (s, t) are mergeable since $\mathcal{R}^+(s) - \mathcal{R}(j)$ is a strict subset of $\mathcal{R}^+(s) - \mathcal{R}(t)$ such that $\{s, i, j\} \subset \{s, i, j, t\}$. Thus, G can be reduced by rules 1 and 2, in that j is eliminated by chain and choice rules after i is done first. The

next step is how to compute aggregated weights of multiple paths when having such bridge edges. Here we introduce an additional term "shared".

Definition 3.5. An edge (v_i, v_j) is *shared* if $d(v_i, v_j) = 1$ and (v_i, v_j) is shared among paths in multiple times.

Consider the probabilistic graph $G_{\mathcal{A}}$ in Figure 3.6(a) and the probabilities of the edges are follows: $\omega(i, k_1) : a$, $\omega(k_1, k_2) : b$, $\omega(i, k_2) : c$, $\omega(k_2, j) : d$, and $\omega(k_1, j) : e$. The set S of relevant intermediate nodes to (i, j) is $\mathcal{R}(i) - \mathcal{R}^+(j) = \{k_1, k_2\}$. We compute $\omega(i, j)$ by eliminating nodes in S. By Definition 3.3, we get $\ell_{\mathcal{G}}(E) = \{c, e\}$ and $\overline{\ell_G(E)} = \{a, b, d\}$ such that $\ell_G(E) \cup \overline{\ell_G(E)} = G_{\mathcal{A}}(E)$. Also, we deduce that c and e are *nonmergeable*. Since $k_1, k_2 \in S$ are split nodes, these two nodes should be eliminated together. There are three paths to (i, j): (i, k_1, k_2, j), (i, k_1, j), and (i, k_2, j). It is verified that $\mathcal{BP}_{ij} = \{P_1, P_2\}$:

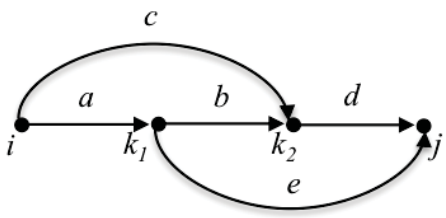
1. $P_1 : (i, k_2, j) = \underline{(i, k_2)}, (k_1, j)$
2. $P_2 : (i, k_1, j) = \underline{(i, k_1)}, (k_2, j)$

We first decide how to merge these two different parallel paths. Consider the following *2-cuts* of the longest path of (i, j) and duplicated paths in Figure 3.6(b) and (c). We define *m-cuts* as disjoint sets of a subset of $\mathcal{BP}$, where m is the number of partitions in a set of mergeable paths, which satisfies the following conditions:

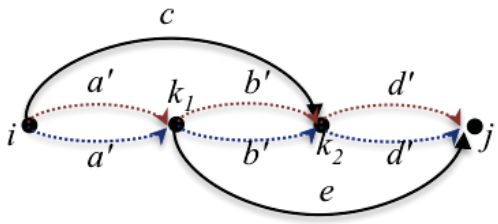
i) $1 < k \leq n$, where n is the number of unique bridge paths between the endpoints.

ii) $A \in P$ and $B \in P$ are disjoint if and only if A and B are "nonmergeable".

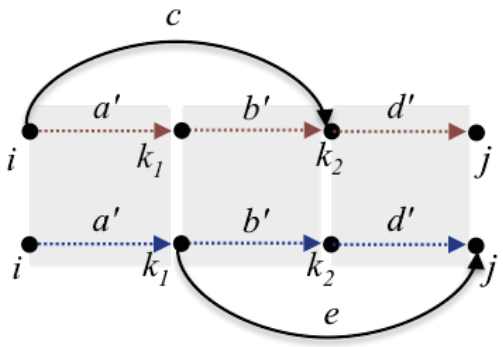
iii) $\forall P_i \in \mathcal{K}$, where $\mathcal{K}$ is a subset of $\mathcal{BP}$, P_i must be mergeable with every $P_j \in \mathcal{K}$.



(a) A graph $G_{\mathcal{A}}$



(b) *2-cuts of* $G_{\mathcal{A}}$



(c) Two parallel paths based on 2-cuts

Figure 3.6: A graph and its cuts.

Going back to the problem of computing $\omega(s,t)$, by Definition 3.5, edges (i,k_1), (k_1,k_2), and (k_2,j) appear more than once in two parallel paths, computing the aggregated weights is $(a'b' \oplus c)d \oplus a'(b'd' \oplus e)$.

At this point, we have two inquiries to complete our proposed approach:

1. when exactly edge e is "shared" and

2. how to compute x' for an edge associated weight with x.

As for question 1, essentially, edge (v_i, v_j) in $\mathcal{LP}$ is "shared" if (v_i, v_j) appears more than once in $\mathcal{BP}$. To see this, consider the following graph G_c in Figure 3.7. The probabilities of the edges are follows: $\omega(i,k_1) : a$, $\omega(k_1,k_2) : b$, $\omega(i,k_3) : d$, $\omega(k_2,k_3) :$

c, $\omega(k_1, k_4)$: e, $\omega(k_2, k_5)$: f, $\omega(k_3, k_4)$: g, $\omega(k_4, k_5)$: h, $\omega(k_4, k_6)$: t, $\omega(k_5, k_6)$: u, $\omega(k_5, j) : w$, and $\omega(k_6, j) : v$.

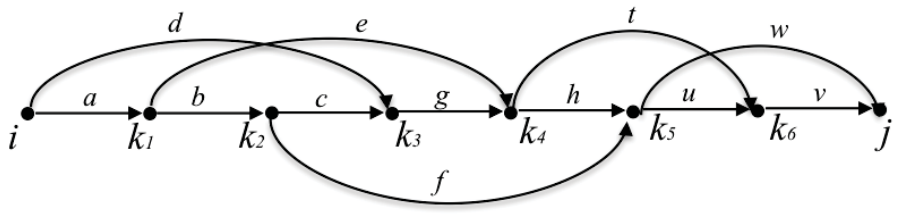


Figure 3.7: A probabilistic graph G_C.

Then the set S of intermediate nodes from (i, j) in G_c is $\mathcal{R}(i) - \mathcal{R}^+(j) = \{k_1, k_2, k_3, k_4, k_5, k_6\}$. We compute $\omega(i, j)$ by eliminating nodes in S. By Definition 3.3, we get $\ell_{G_c}(E_c) = \{d, e, f, t, w\}$ and $\overline{\ell_{G_c}(E_c)} = \{a, b, c, g, h, u, v\}$ such that $\ell_{G_c}(E_c) \cup \overline{\ell_{G_c}(E_c)} = E_c$. By Definition 3.2, the fact that all the nodes in S are split, we eliminate them all together. There are three steps to merge the parallel paths between i and j:

1. Define the sets $\mathcal{BP}_{ij}$ and $\mathcal{LP}_{ij}$ in G_c.

2. Partition $\mathcal{BP}_{ij}$ into disjoints subset of mergeable paths, which generates $\{\mathcal{K}_1\}$, $\{\mathcal{K}_2\}$,.., $\{\mathcal{K}_m\}$, where m is the number of subset of nonmergeable paths:*m-cuts*

3. Compute $\omega(i, j)$ while eliminating nodes in S by rules 1 and 2 for m times.

We get $\mathcal{BP}_{ij} = \{P_1, P_2, P_3, P_4, P_5\}$ from $\ell_{G_c}(E)$, that is;

1. $P_1 : (i, k_3, k_4, k_5, k_6, j) = \underline{(i, k_3)}, (k_3, k_4), (k_4, k_5), (k_5, k_6), (k_6, j)$

2. $P_2 : (i, k_1, k_4, k_5, k_6, j) = (i, k_1), \underline{(k_1, k_4)}, (k_4, k_5), (k_5, k_6), (k_6, j)$

3. $P_3 : (i, k_1, , k_2, k_5, k_6, j) = (i, k_1), (k_1, k_2), \underline{(k_2, k_5)}, (k_5, k_6), (k_6, j)$

4. $P_4 : (i, k_1, k_2, k_3, k_4, k_6, j) = (i, k_1), (k_1, k_2), (k_2, k_3), (k_3, k_4), \underline{(k_4, k_6)}, (k_6, j)$

5. $P_5 : (i, k_1, k_2, k_3, k_4, k_5, j) = (i, k_1), (k_1, k_2), (k_2, k_3), (k_3, k_4), (k_4, k_5), \underline{(k_5, j)}$

along with $\mathcal{LP}_{ij} = (i, k_1, k_2, k_3, k_4, k_5, k_6, j)$, described in Figure 3.8.

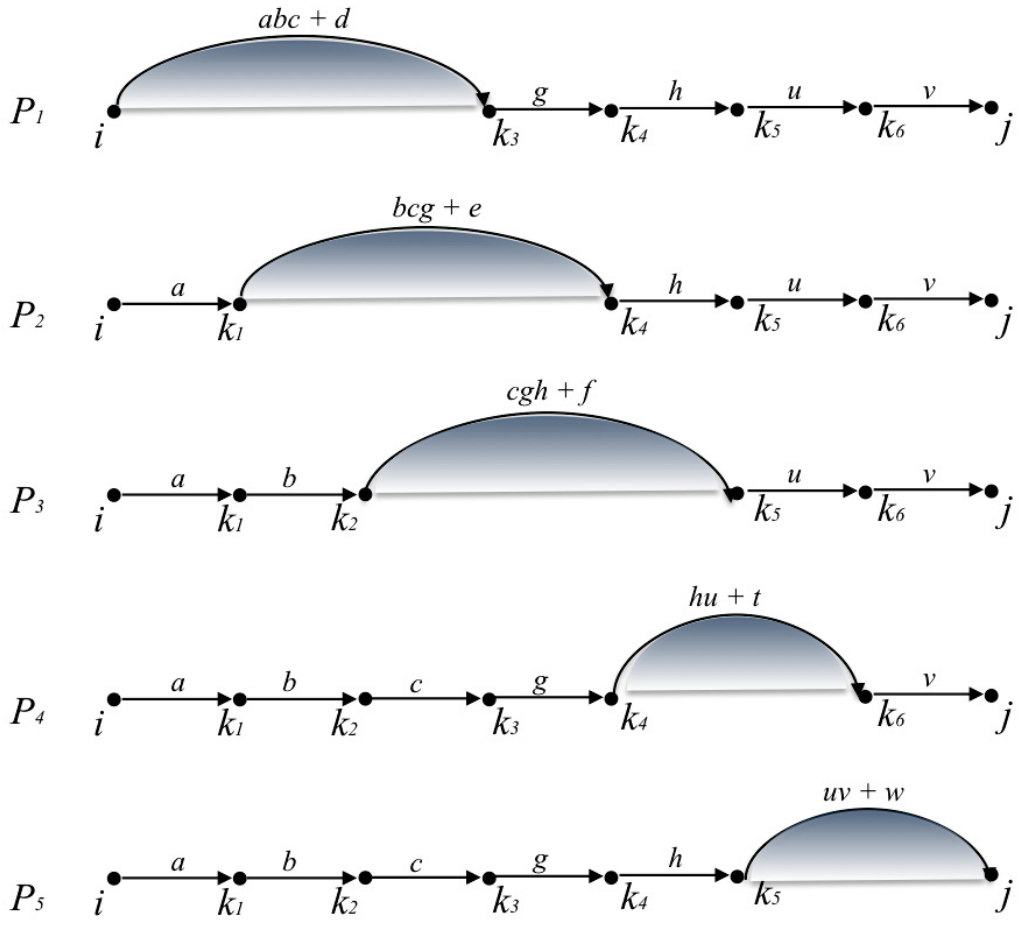


Figure 3.8: $\mathcal{P}$ with computed weights.

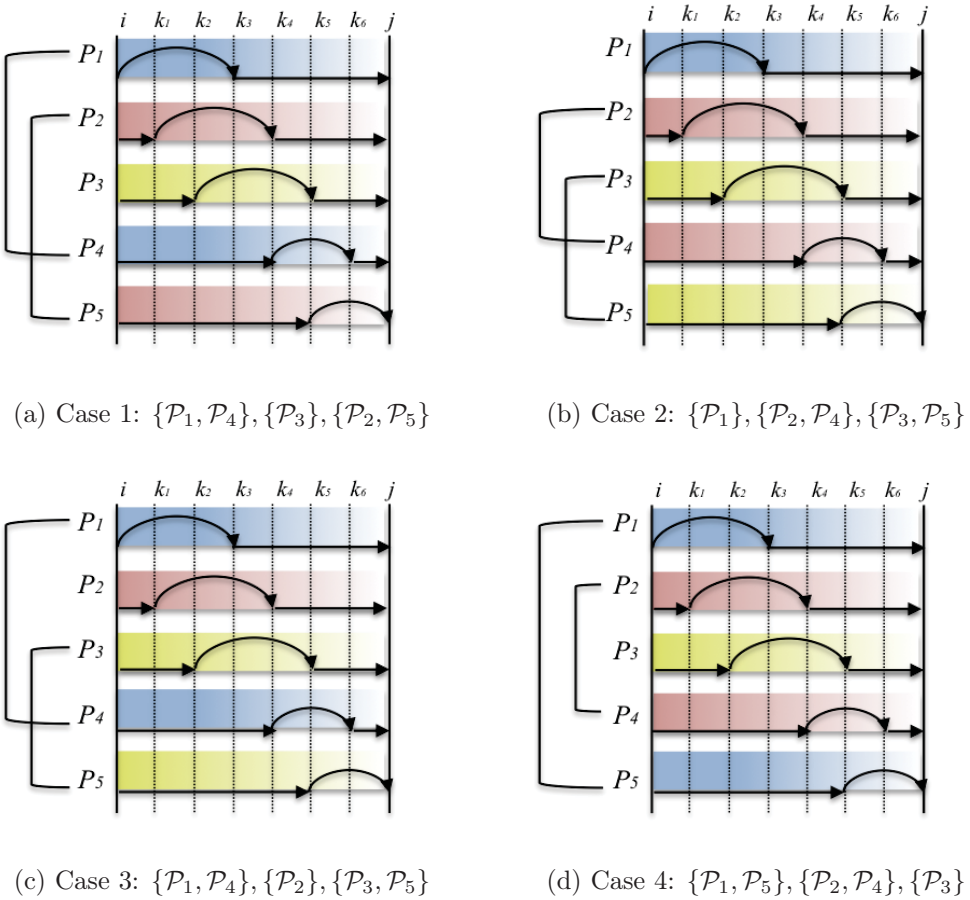
Now, we partition $P_1 \sim P_5$ into *m-cuts* to compute $\omega(i, j)$. Note that if edge $(i, j) \in A$ and edge $(u, v) \in B$ are mergeable then A and B are mergeable. By Definition 3.5, we get four possible cases of partitioning $\mathcal{BP}_{ij}$ into *3-cuts*:

1) $\{P_1, P_4\}, \{P_3\}, \{P_2, P_5\}$

2) $\{P_1\}, \{P_2, P_4\}, \{P_3, P_5\}$

3) $\{P_1, P_4\}, \{P_2\}, \{P_3\}$

4) $\{P_1, P_5\}, \{P_2, P_4\}, \{P_3\}$.



(a) Case 1: $\{\mathcal{P}_1, \mathcal{P}_4\}, \{\mathcal{P}_3\}, \{\mathcal{P}_2, \mathcal{P}_5\}$

(b) Case 2: $\{\mathcal{P}_1\}, \{\mathcal{P}_2, \mathcal{P}_4\}, \{\mathcal{P}_3, \mathcal{P}_5\}$

(c) Case 3: $\{\mathcal{P}_1, \mathcal{P}_4\}, \{\mathcal{P}_2\}, \{\mathcal{P}_3, \mathcal{P}_5\}$

(d) Case 4: $\{\mathcal{P}_1, \mathcal{P}_5\}, \{\mathcal{P}_2, \mathcal{P}_4\}, \{\mathcal{P}_3\}$

Figure 3.9: Four different cases of partition based on the *3-cuts*.

The question now is which case is the correct way of partitioning into mergeable path sets. Beforehand, we need to find the effectiveness of different partitions for computing the aggregated weights. As mentioned in Chapter 1, disjunction is associative;

that is,

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C$$

The operations can be performed in any order. Therefore, in terms of associativity, we associate different paths in any order, and still get the same result in each case. As long as we group the paths conforming to the notion of mergeable paths, the end result will always be unique. We choose case 1 randomly as all different cases yield an equal value.

In graph G_c, $(i, k_1), (k_1, k_2), (k_2, k_3), (k_3, k_4), (k_4, k_5), (k_5, k_6)$, and (k_6, j) in $\mathcal{LP}_{ij}$ are all shared edges. Thus, computing the reachability probability between i and j, $\omega(i, j)$, is $(a'b'c' \oplus d)g'(h'u' \oplus t)v' \oplus a'(b'c'g' \oplus e)h'(h'v' \oplus w) \oplus a'b'(c'g'h' \oplus f)u'v'$.

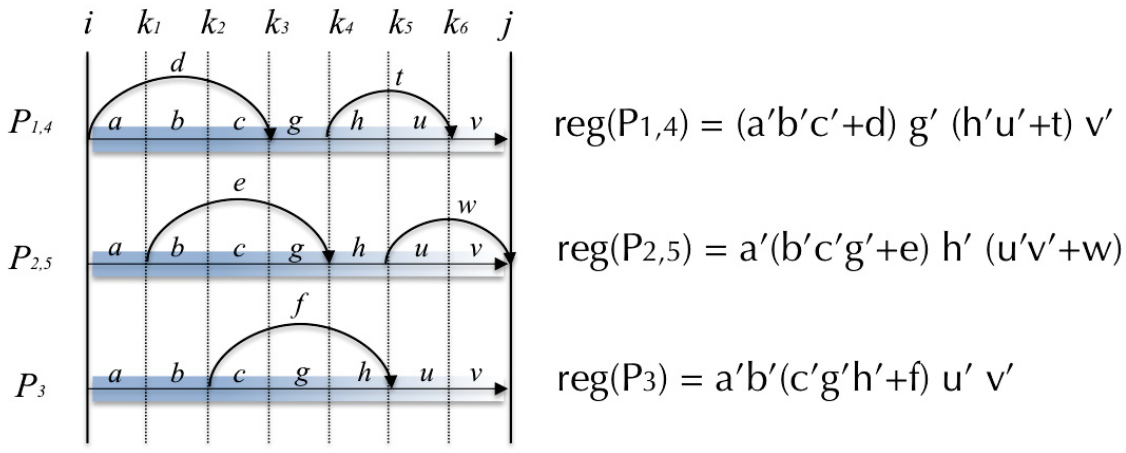


Figure 3.10: The final weights between two endpoints of G_c.

As for simultaneous nodes reduction using the merge rule, we can only handle the case where the topological sort of a given graph has a unique output. The problem arises when the possible list of ordered nodes are not unique. To see this, consider the graph in Figure 3.11. The topological ordering in this graph is

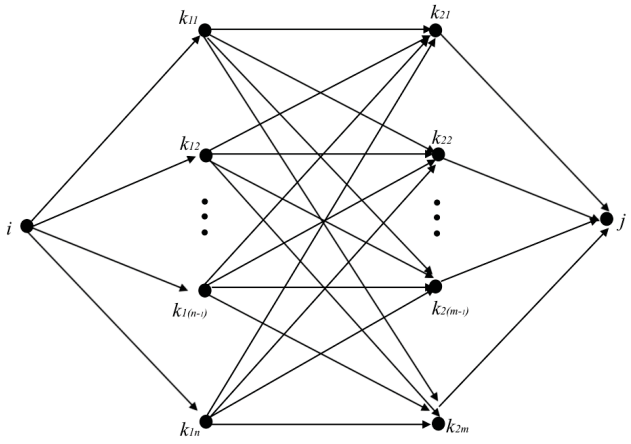


Figure 3.11: A graph which does not yield a unique topological sort.

$\{i\}, \{k_{11}, \ldots, k_{1n}\}, \{k_{21}, \ldots, k_{2m}\}$, and $\{j\}$, which includes non-singleton sets $\{k_{11}, k_{12}$ $\ldots, k_{1n}\}$ and $\{k_{21}, k_{22}, \ldots, k_{2m}\}$. Let $n = 2$ and $m = 2$. The set S of relevant intermediate nodes to the pair (i, j) is $\mathcal{R}(i) - \mathcal{R}^+(j) = \{k_{11}, k_{12}, k_{21}, k_{22}\}$. By Definition 3.2, all the relevant intermediate nodes in S are split nodes. We need an additional rule to manage a special case where all relevant intermediate nodes are "split" and it does not have a unique ordering in such nodes. To handle such cases, we introduce the fifth rule, "mesh" based on the left-hand side of topological ordering of the nodes in a given graph. The proposed solution approach eliminates the nodes in a same set all together at once. We start by eliminating the nodes in the very left of the ordered list. While maintaining the newly updated weights of incident edges to such nodes, we remove multiple nodes in the following set, one by one, until the set of intermediate nodes is empty. The following graph to illustrates this rule. The topological ordering of this graph includes $\{i\}, \{k_{11}, k_{12}\}, \{k_{21}, k_{22}\}, \{k_{31}, k_{32}\}, \{k_{41}, k_{42}\}$, and $\{j\}$. We eliminate the nodes that appear as same order from left. This yields the following

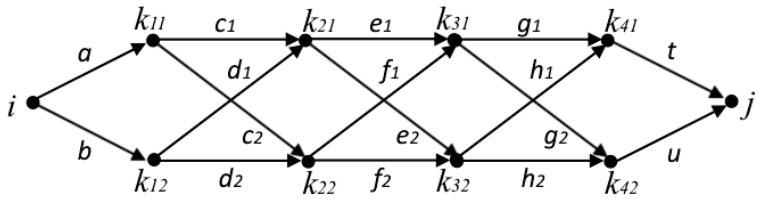


Figure 3.12: An example of "Mesh" graph.

order to eliminate the nodes:

$$k_{11}, k_{12} \Rightarrow k_{21}, k_{22} \Rightarrow k_{31}, k_{32} \Rightarrow k_{41}, k_{42}$$

To compute the weight $\omega(i, j)$, we need to consider all the relevant paths involved. Here, we have $2^4 = 16$ such paths relevant to the pair (i, j). Considering this graph, we note that it can be divided into $2 - cuts$ if we eliminate the nodes from the left. This yields the final weight follows:

$$\{[(a'c_1 \oplus b'd_1)'e_1 \oplus (a'c_2 \oplus b'd_2)'f_1]'g_1 \oplus [(a'c_1 \oplus b'd_1)'e_2 \oplus (a'c_2 \oplus b'd_2)'f_2]'h_1\}t$$
$$\oplus \{[(a'c_1 \oplus b'd_1)'e_1 \oplus a'c_2 \oplus b'd_2)'f_1]'g_2 \oplus [(a'c_1 \oplus b'd_1)'e_2 \oplus (a'c_2 \oplus b'd_2)'f_2]'h_2\}u$$

We can generalize the mesh rule as follows. In the graph of Figure 3.14(a), k_1, k_2, k_3, and k_4 are split nodes, these nodes should be eliminated simultaneously. The topological ordering of nodes in this graph includes $\{i\}, \{k_1, k_2\}, \{k_3, k_4\}$, and $\{j\}$, and the corresponding order for node reduction is:

$$k_1, k_2 \Rightarrow k_3, k_4$$

By Definition 3.3, since edges (i, k_1) and (i, k_2) are shared, we represent them as a' and b', shown in Figure 3.14(b). The proposed rule yields a desired unique semantics for computing reachability queries in case where a graph has multiple topological sort

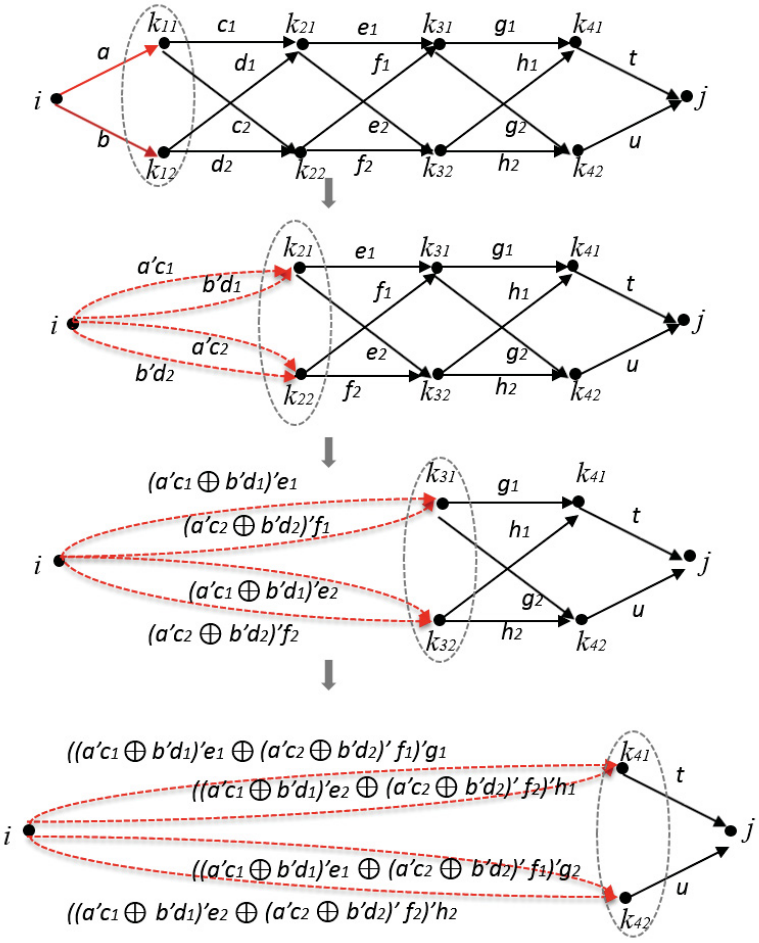
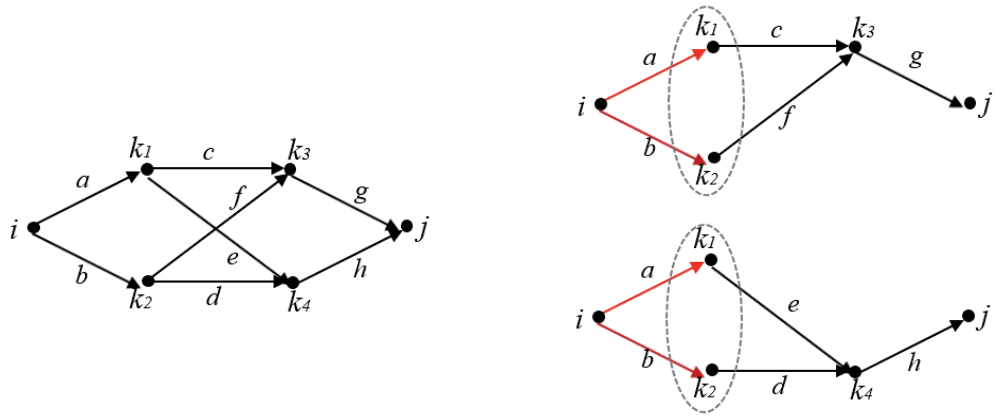


Figure 3.13: Reduction steps of "Mesh" graph.

and simultaneous nodes reduction are needed. We also defined how to compute x' for an edge weight x, when x is to be shared among multiple parallel paths, which means "shared". By Definition 3.5, an edge (v_i, v_j) in graph G is shared if (v_i, v_j) appears at least in two paths. When an edge is shared, its weight is divided "equally" among all the paths between the same endpoints. Hence, an edge e with weight x can be divided into two parallel edges of equal weights x', where $x = x' \oplus x'$. When the number of parallel edges is 2, x' can be easily derived as $1 - \sqrt{1-x}$. How to compute x', if the number of parallel edges is any integer n? To answer this, we use



(a) An input (mesh) graph graph

(b) *2-cuts of the input graph*

Figure 3.14: A graph and its cuts.

the inclusion–exclusion principle that relates the sizes of two sets and their union. Let $A_1, A_2, \ldots, A_n$ be a set of events. For $n = 2$,

$$\mathbb{P}(A_1 \cup A_2) = \mathbb{P}(A_1) + \mathbb{P}(A_2) - \mathbb{P}(A_1 \cap A_2)$$

For $n = 3$, we have that:

$$\begin{aligned}\mathbb{P}(A_1 \cup A_2 \cup A_3) = \mathbb{P}(A_1) + \mathbb{P}(A_2) + \mathbb{P}(A_3) \\ - \mathbb{P}(A_1 \cap A_2) - \mathbb{P}(A_1 \cap A_3) - \mathbb{P}(A_2 \cap A_3) \\ + \mathbb{P}(A_1 \cap A_2 \cap A_3)\end{aligned}$$

and in general, we have the following equation.

$$\begin{aligned}\mathbb{P}\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} \mathbb{P}(A_i) - \sum_{i<j} \mathbb{P}(A_i \cap A_j) \\ + \sum_{i<j<k} \mathbb{P}(A_i \cap A_j \cap A_k) - \cdots + (-1)^{n-1} \mathbb{P}\left(\bigcap_{i=1}^{n} A_i\right),\end{aligned}$$

In our case of dealing with a disjunction function $\oplus$, the inclusion–exclusion principle would yield:

$$\bigoplus_{i=1}^{n} x_i = \sum_{i=1}^{n} x_i - \sum_{i<j} x_i x_j + \sum_{i<j<k} x_i x_j x_k - \ \cdots \ + (-1)^{n-1} \prod_{i=1}^{n} x_i,$$

Going back to the problem of finding x', we consider the last equation above with all x_i is being equal to x'. This yields:

$$\bigoplus_{i=1}^{n} x_i = 1 - (1-x)^n,$$

from which we get $x' = 1 - \sqrt[n]{1-x}$, that is obtained by considering the complement probability of all links being broken. To illustrate this, let $\omega(e) = 0.3$ be the probability for edge e. Then σ_1 is 0.3 without duplicating e. We want to get the value for σ_4 such that $\sigma_4 = \sigma_1 \oplus \sigma_1 \oplus \sigma_1 \oplus \sigma_1$. We get $\sigma_2 = \sigma_1 \oplus \sigma_1 = 0.3 + 0.3 - 0.3 \times 0.3 = 0.51$, $\sigma_3 = \sigma_2 \oplus \sigma_1 = 0.51 + 0.3 - 0.51 \times 0.3 = 0.667$, and lastly $\sigma_4 = \sigma_3 \oplus \sigma_1 = 0.667 + 0.3 - 0.667 \times 0.3 = 0.766$. If we apply the general form $\sigma_n = 1-(1-X)^n$, then we get $\sigma_2 = 1-(1-0.3)^2 = 0.51$, $\sigma_3 = 1-(1-0.3)^3 = 0.667$, and lastly $\sigma_4 = 1 - (1 - 0.3)^4 = 0.766$, which is the same value we obtained directly above. Note that n is the number of cuts when partitioning parallel paths between the same endpoints in a given graph. When an edge e with probability x is shared, we first compute σ_n for $\sigma_1 = x$, and then divide σ_n by the number n of different paths. The result is denoted by x', with $x' = \sigma_n/n$.

3.6 Illustrative examples

To illustrate how the reduction is done by the merge rule where a graph has a unique topological sort, consider the following example.

Example Consider a probabilistic graph $G = (V, E)$ of Figure 3.15(a), where the probabilities of the edges are follows: $\omega(i, k_1) : 0.3$, $\omega(k_1, k_2) : 0.6$, $\omega(i, k_2) : 0.7$, $\omega(k_2, k_3) : 0.8$, $\omega(k_1, k_3) : 0.5$, $\omega(k_2, k_4) : 0.2$, $\omega(k_3, k_4) : 0.1$, $\omega(k_2, j) : 0.8$, and $\omega(k_4, j) : 0.4$, and We want to compute the weight between nodes i and j. Then the set S of relevant nodes of (i, j) is $\mathcal{R}(i) - \mathcal{R}^+(j) = \{k_1, k_2, k_3, k_4\}$. We compute $\omega(i, j)$ by eliminating nodes in S. By Definition 3.3, we get $\ell_G(E) = \{(i, k_2) : 0.7, (k_1, k_3) : 0.5, (k_2, k_4) : 0.2, (k_2, j) : 0.8\}$ and $\overline{\ell_G(E)} = \{(i, k_1) : 0.3, (k_1, k_2) : 0.6, (k_2, k_3) : 0.8, (k_3, k_4) : 0.1, (k_4, j) : 0.4\}$. By Definition 3.2, all the nodes in S are "split" nodes, such nodes should be eliminated together. First, we get the set of all paths to (i, j): $P = \{(i, k_1, k_2, k_3, k_4, j), (i, k_1, k_3, k_4, j), (i, k_1, k_2, j), (i, k_2, k_3, k_4, j), (i, k_2, k_4, j), (i, k_2, j)\}$. We get $\mathcal{P}_{ij} = \mathcal{BP}_{ij} \cup \mathcal{LP}_{ij}$:

$$\mathcal{BP}_{ij} = \{(i, \underline{k_1, k_3}, k_4, j), (i, k_1, \underline{k_2, j}), (i, k_2, k_3, \underline{k_4, j})\}$$

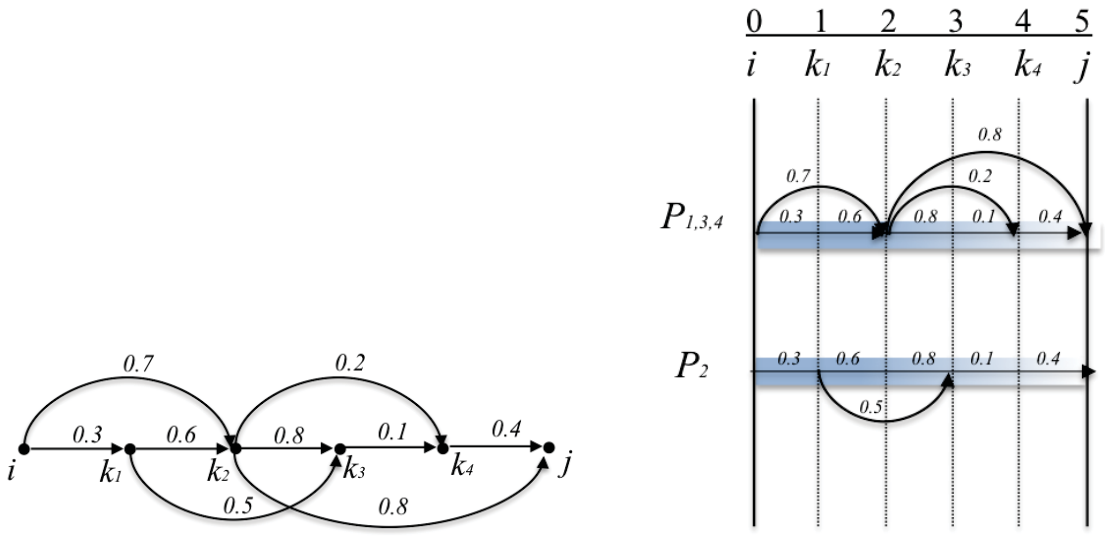
$$\mathcal{LP}_{ij} = \{(i, k_1, k_2, k_3, k_4, j)\}$$

We then partition four different parallel paths in $\mathcal{BP}_{ij}$ into *m-cuts*.

1. $\mathcal{P}_1 : (i, k_2, k_3, k_4, j)$
2. $\mathcal{P}_2 : (i, k_1, k_3, k_4, j)$
3. $\mathcal{P}_3 : (i, k_1, k_2, k_4, j)$
4. $\mathcal{P}_4 : (i, k_1, k_2, j)$

Since P_1, P_3, and P_4 can be merged together but not with P_2, by Definition 3.3, we have *2-cuts* of $\mathcal{BP}_{ij}$, which are $\{P_1, P_3, P_4\}$ and $\{P_2\}$. Note that m=2 is the number of different paths between i and j. By Definition 3.5, the edges (i, k_1), (k_1, k_2), (k_2, k_3), (k_3, k_4), and (k_4, j) are shared since they appear in at least two paths.

Now, we perform the reduction by eliminating all the nodes in S while computing and maintaining the weights by considering the edges we combined, that is: $(0.3' \times 0.6' \oplus 0.7)(0.8 \oplus (0.2 \oplus 0.8' \times 0.1')0.4') \oplus 0.3'(0.6' \times 0.8' \oplus 0.5)0.1' \times 0.4' = 0.59866252$, with $x' = \sigma_2/2$. This manual calculation is used for measuring the correctness of the implementation for our reduction algorithm, which is carried in Chapter 5.



(a) A probabilistic graph G

(b) The 2-cuts of G

Figure 3.15: An example graph.

The reduction algorithm we proposed considers an order in which nodes are eliminated, that are based on the five patterns and its corresponding reduction

rules, which we called, *chain, choice, cycle, merge*, and *mesh* rules. The proposed techniques extend another way of correct semantics in the existing enumeration of paths. As mentioned, the problem of the way of enumeration of paths that the author proposes [27] in terms of node-based reduction has issues with regard to overcomputation. To see this, consider a probabilistic graph G_p of Figure 3.16. Note that G_p is taken from the same paper to compare the difference from our algorithm. Given $G_p = (V, E)$, where the probabilities of the edges are follows: $\omega(A, B) : 0.8$, $\omega(A, C) : 0.2$, $\omega(C, B) : 0.9$, $\omega(B, D) : 0.9$, $\omega(D, C) : 0.9$, $\omega(C, E) : 0.1$, $\omega(D, E) : 0.05$, $\omega(D, F) : 0.05$, $\omega(E, F) : 0.8$, and $\omega(F, G) : 0.7$. Let A and G be the start and end nodes and we want to determine the probability of reaching from A to G, $\omega(A, G)$. Since the result is irrelevant from the reduction order using *max* mode for disjunction function, we choose nodes randomly. First, B and E are eliminated and then C is done after. Elimination of D causes repetition of D three times conforming to rule 3, and the probability added to the edge will be $(0.9 \times 0.9 \times 0.9)^3$. Further reduction reduces G_p to (A, G) with one edge, so that the values added to the edge is the end result, which is $\omega_{max}(A, G) = 0.8 \times 0.9(0.9 \times 0.9 \times 0.9)^3 \times 0.9 \times 0.1 \times 0.8 \times 0.7 = 0.0141$.

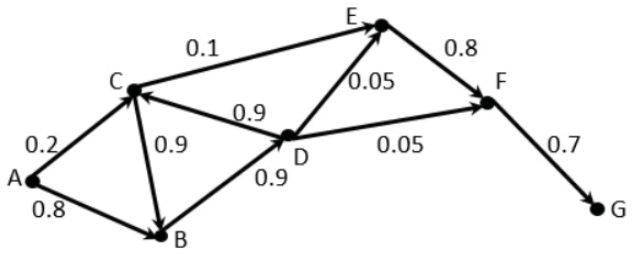


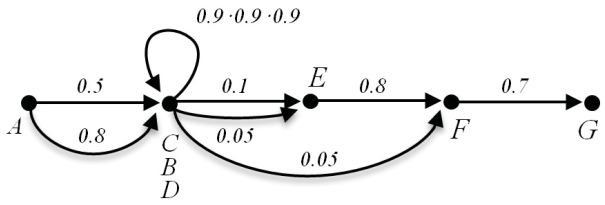
Figure 3.16: A probabilistic graph G_p taken from [27].

We want to correct the way of computing the weights in paths while avoiding unnecessary repeated computations. The state elimination process results in such redundant computations. For example, $\omega(B, D) = 0.9$ is calculated several times during reduction, which is inadequate for reducing nodes in our context of uncertain graphs. Noe that if multiple nodes are on a same cycle, we consider them as indistinguishable nodes, and such nodes are reduced and replaced by a single "solid" node the result. In this case, B, C, D are combined as a C which yields $\omega_{max}(A, G) = 0.8(0.9 \times 0.9 \times 0.9)^3 \times 0.1 \times 0.8 \times 0.7 = 0.01736$. When *ind* mode is used as disjunction function, we get different end results depending on the order in which nodes are reduced. Figure 3.17 describes the reduced graph of G_p by treating nodes in the same SCC indistinguishable as the combined node C. There are $6(= 3!)$ cases of different order in node reduction, shown in Table 1.

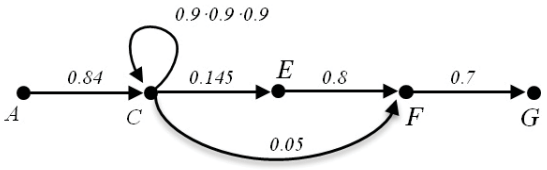
Table 3.1: 6 cases of $\omega_{ind}(AG)$ varying the order of node reduction.

Case	Node order	Algebraic expression	End result
1	C-E-F	$((0.84 \times 0.729^3 \times 0.145 \times 0.8) \oplus (0.84 \times 0.729^3 \times 0.05 \times 0.8))0.7$	0.03739
2	F-E-C	$0.84 \times 0.729^3 \times (0.145 \times 0.8 \times 0.7 \oplus 0.05 \times 0.7)$	0.03689
3	C-F-E	$(0.84 \times 0.729^3 \times 0.145 \times 0.8 \times 0.7) \oplus (0.84 \times 0.729^3 \times 0.0.5 \times 0.7)$	0.03752
4	F-C-E		
5	E-C-F	$0.84 \times 0.729^3 \times (0.145 \times 0.8 \oplus 0.05)0.7$	**0.03649**
6	E-F-C		

As can be seen in Table 1, $< E, C, F >$ and $< E, F, C >$ cases result in the minimal weight, 0.03649, which is the output of the least common algebraic expressions. Hence, E should be removed first, which is our proposed reduction order, that is;



(a) Reduction with SCCs



(b) Reduced graph

Figure 3.17: State elimination process based on ind mode

- The set S of relevant nodes to (A, G) is $\{B, C, D, E, F\}$
- Nodes B,C, and D are combined as C and removed from S; then $S = \{C, E, F\}$
- E is *chain*, C and F are *choice*, therefore E should be eliminated first.
- After elimination of E, both C and F become *chain*.
- The order of reduction between C and F does not affect the result, which is 0.03649 in both cases.

3.7 Methodology

In this section, we describe our methodology for the reduction algorithm in details. The steps of the reduction algorithms are as below.

Steps of reduction algorithms

- Inputs: $< G, (v_s, v_f), m >$, where a probabilistic digraph G, $v_s \in V$ is a source node, $v_f \in V$ is a destination node, and m is a disjunction type from $\{max, ind\}$
- Output: Probability of reaching to v_f from v_s: $\omega(v_i, v_j)$

1. Find the "relevant" subgraph $G' = (V', E)$: all paths (v_s, v_f) in G.
2. Find the cycles in G' and reduce them into solid nodes.
   - Identify the list L of all cycles in G' and sort L in order by their lengths. $L = [C_1|\text{Rest}]$
   - Until L is empty, reduce C_1 into a solid node and remove C_1 from L.

   $\Rightarrow$ It generates G'' which is acyclic.
3. Eliminate all the relevant nodes in G'' but v_s and v_f, using the proposed reduction rules/patterns, which returns $\omega(i, j)$

As for the disjunction function, we have two user-defined modes: *max* and *ind*. Any suitable disjunction functions can be easily updated and utilized based on the users' demand. A single process allowing user to attempt to use different disjunction functions makes our reduction technique important.

3.7.1 Phase 1: Finding the Relevant Subgraph

If the connectivity of (v_s, v_f) is true, Phase 1 finds the relevant subgraph of G, which is an induced subgraph $G' = (V', E')$, where $V' = S \cup \{v_s, v_f\} \subseteq V$ and $S = \mathcal{R}(v_s) - \mathcal{R}^+(v_f)$. Generating the relevant subgraph G' can be simply done by any graph

traversal techniques such as Depth First Search or Breadth First Search when G has no cycles. It, however, is not trivial when G is cyclic since there can be an exponential number of paths connecting the source node with each reachable node. The main goal is relevant paths should be kept with non-duplicated cycles in each path for the reduction algorithm. We use graph reachability and Iterative Deepening Depth First Search (IDDFS) algorithms [13] to handle cyclic path detection.

3.7.2 Phase 2: Reducing Cycles

First, it detects cycles in the relevant subgraph G'. A cycle in a graph is a path of the form $(i_1, i_2, .., i_1)$ with the first and last nodes being identical. For the cycle detection, we identify a set of nodes visited more than once in a path. By traversing the sequence of path starting i_1 and using a data structure to store these nodes, we test whether each subsequent node has already been stored. Hence, if the number of occurrence of node i_1 is more than 2 in the path developed from IDDFS we then determine whether such a path $W = (i_1, .., i_1)$ is cyclic. In that case, we call every node in W as a cyclic node and each cyclic path is reduced to a self-loop edge for a representative "solid" node, using our proposed cycle reduction. Now we reduce cycles in a proper way considering the associated probabilities in every cyclic path as well as overcomputation. Second, the list of cyclic paths are sorted in ascending order of their lengths. We then combine each cycle by a representative "solid" node one by one in left-hand-side of the cycle sets. Let $L =< h[C_1] : C_1, h[C_2] : C_2, .., h[C_n] : C_n >$ be the list of cycles, where $h[C_i]$ is an initial node of C_i which is the primary key in this list and n is the number of all cyclic paths in G'. C_i is the list of cycles with the

form $< C_{i1}, C_{i2}, ... >$ that is for all $C_{ik}, C_{ik+1} \in C_i,\ h[C_{ik}] = h[C_{ik+1}]$ for $C_{ik} < C_{ik+1}$. Reducing cycles are somewhat different from node reduction since it is *edge-based* reduction. To be more specific, reducing a single cyclic path may cause elimination of more than one nodes if the length of the cyclic path is more than 3 where the cycle includes only one node except the initial and end nodes: (i_1, i_2, i_1). One more thing that makes the cycle reduction process tricky is L needs to be updated and sorted by the length at each iteration recursively until L is empty. Taking into account the proposed technique, reducing cycles is illustrated as follows.

1. Identify the list L of all cycles of form $h[C_1] : C_1$ in the subgraph G'.

2. Sort L in ascending order by their lengths.

3. **While** L is not empty **do**:

   Case 1: If C_i is involved in nested cyclic paths

   While C_i is not empty **do**:

   (a) Reduce C_{i1} into a solid node $h[C_1]$ and update. $\omega(C_i, C_i)$

   (b) Remove C_{i1} from C_i.

   (c) Sort L in ascending order of the length.

   Case 2: Otherwise

   (a) Reduce C_i into a solid node $h[C_1]$ and update. $\omega(C_i, C_i)$

   (b) Remove C_1 from L.

4. It returns the reduced acyclic graph G''.

The fuse step is only initiated when G' is cyclic. Then reducing cycles generates the reduced graph G'' which becomes an acyclic graph. During the cycle reduction process, cyclic nodes are eliminated by rules 1, 2, and 3 depending on the path patterns.

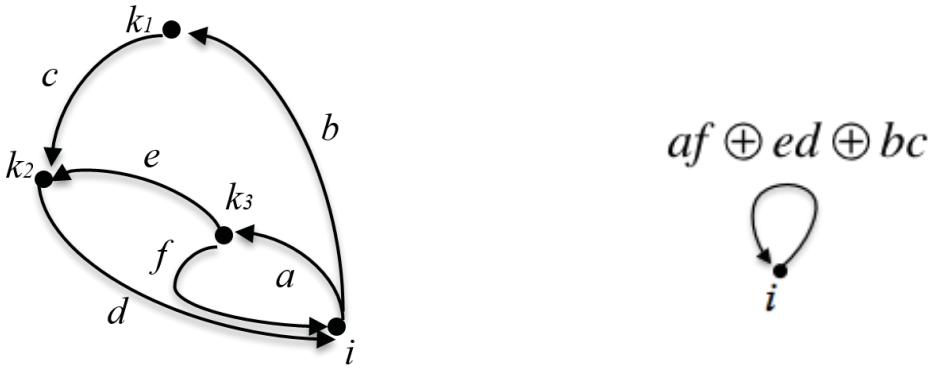
To see this, consider the graph in Figure 3.18(a). The iterations of reducing the cycles in G' are as follows:

1. Fusing (i,k_3,i) as a combined node i generates a self-loop with the weight ***af***
   - L is updated: $L =< i : (i,k_2,i),(i,k_1,k_2,i) >$
2. Fusing (i,k_2,i) with i generates another self-loop with the weight ***ed***
   - L is updated: $L =< i : (i,k_1,i) >$
3. Fusing (i,k_1,i) with i generates the final self-loop with the weight ***bc***
   - L is empty
4. The final weight for the self-loop at node i is $\underline{af \oplus ed \oplus bc}$ and then the fuse step is terminated.

Computing the aggregated weights we get through the fuse step is $af \oplus ed \oplus bc$.

3.7.3 Phase 3: Nodes Reduction

Now, we repeatedly apply to get one edge between v_s and v_f in G''. The fact G'' is acyclic, the **cycle** rule is not applied in this phase. First rule 1 **chain** and rule 2 **choice** are applied to eliminate "sequence" nodes in S. If S is not empty, for the



(a) An original probabilistic graph

(b) Its reduced graph

Figure 3.18: An example probabilistic cyclic graph.

simultaneous nodes reduction, rule 4 **merge** and rule 5 **mesh** are used depending on the uniqueness of topological sort of nodes in S. If there is a unique topological sort, rule 4 is used. Otherwise rule 5 is used to eliminate nodes all together. These rules are applied continuously to get the reachability probability of v_s and v_f while aggregating the weights with considered disjunction function.

Chapter 4

Implementation

Our algorithm returns $\omega(v_s, v_f)$ which is the reachability probability between nodes v_s and v_f in a given probabilistic graph G. The pseudocode of proposed reduction algorithm is described in Algorithm 1.

Our algorithm is based on a generalization of the reduction algorithm proposed by Floyd-Warshall [8], which considers only from the set $\{1, 2, ..., k\}$ as intermediate nodes along the path for finding the shortest path from a start node to an end node in a weighted digraph. The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of nodes. It is able to do this with $O|V|^3$

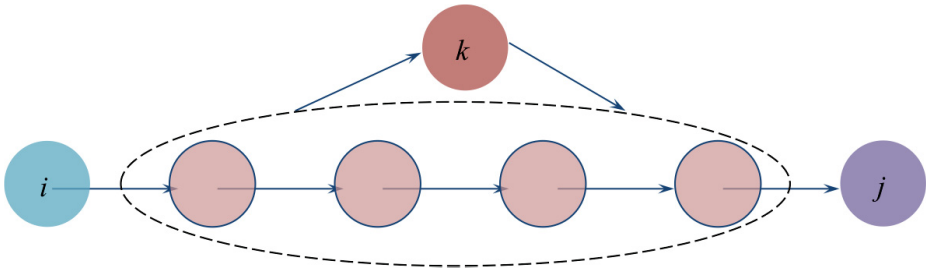


Figure 4.1: Floyd-Warshall algorithm.

Algorithm 1 Reduction algorithm

Inputs: A digraph $G = (V, E, w)$, (v_s, v_f), and disjunction mode $\mathcal{F}_d$

Output: $\omega(v_s, v_f)$

***Phase** 1:* $G' \leftarrow G$

    if $Connect(v_s, v_f)$ is *True* **then**

        Find the subgraph G' of G that is relevant to (v_s, v_f)

        Find all paths from v_s to v_f in G'

    else:

        Return $\omega(v_s, v_f) = 0.0$

***Phase** 2:* $G'' \leftarrow G'$

    Cycle list $L = [\,]$

    for *path* in P

        if cycles exist in *path* **then** reduce cycles

    for c in $sorted(L)$ until $sorted(L)$ is empty

        Compute $\omega(c_1, c_1)$ using the proposed rules

***Phase** 3*

    $S = V'' - \{v_s, v_f\}$

    while $S \neq \emptyset$

        for node k in S

            do reduction $\omega(i, j) \leftarrow path(i, k, j)$

        $S = S$ - $\{k\}$

comparisons in a graph, in worst case where there may be up to $|V|^2$ edges in the graph, and every combination of edges is tested. The shortest possible path from i to j is shown in Figure 4.1.

4.1 Phase 1: Finding the Relevant Subgraph

To check whether v_s and v_f are connected, we find the set $\mathcal{R}(v_s)$ of the relevant nodes from v_s and see if v_f is present in $\mathcal{R}(v_s)$. If not, then the process is terminated: $\omega(v_s, v_f) = 0.0$. This is done through a module called *Reachable(graph, node)*, which takes a graph G and a node v_s as inputs and returns the list of all nodes in G that are reachable from v_s. The input graph is represented as a Dictionary where each node in the graph is a key v_k in the Dictionary, such that (v_k, x) is in an edge in G, and the value associated with v_k is a list of the nodes. The use of Dictionary, namely, hash indexing as a data structure for our reduction algorithm is to speed up the search process. It allows a quick search of edges in the graph: we can search an edge in $O(1)$ and enumeration of the reachable nodes in V from node v_i is $O(|V|)$. The advantage of a Dictionary is that it does not scan through its contents to find values. The key is used to find the desired node rather than having to examine every single node. Hence, this hash table lookup allows us to access a value very quickly. The nodes in the returned list may appear in any order, but should not contain any duplicates. Once we get $\mathcal{R}(v_s)$, we eliminate nodes in $\mathcal{R}(v_s)$ that are not reachable to v_f. The module works by marking nodes that can be reached from start node v_s. Initially, only v_s is marked. Then, the algorithm performs a series of visits through all the nodes in G. If it finds an unmarked node that can be reached from a marked

node, it marks that node. This process stops when an entire pass executes without marking a single unvisited node. At that point, the marked nodes are precisely those that can be reached from v_s. After finding the set $\mathcal{R}(v_s)$ of relevant nodes of v_s, for every node v of $\mathcal{R}(v_s)$, the module checks v is reachable to v_f. The set V' of relevant nodes to (v_s, v_f) includes nodes that are on any path from v_s to v_f that is $V' = \{v \in V' | v \rightarrow v_f,\ for\ v \in \mathcal{R}(v_s)\}$. Then it finds the relevant subgraph $G' = (V', E')$ of G and defines the set S of relevant intermediate nodes which is $S = \mathcal{R}(v_s) - \mathcal{R}^+(v_f)$. We call *reachable(graph, v_f)* again to get $\mathcal{R}(v_f)$. Then finally we get S which induces the subgraph $G' = (V', E')$ where $V' = \{v_s, v_f\} \cup S$ and $E' = V' \times V'$.

In our case for finding the relevant subgraph through phases 1, the fact that we are using Python dictionary, *hashing*, enumerating the reachable nodes requires $O(|V|)$. To get the reachable node from v_s, $\mathcal{R}(v_s)$, and check all nodes in $\mathcal{R}(v_s)$ to v_f, it requires $O(|V| + n|V|)$ where n is the number of nodes in $\mathcal{R}(v_s)$, which is $O(|V|)$.

4.2 Phase 2: Reducing Cycles

A simplistic DFS is not suitable for our reduction algorithm, since DFS, literally going depth-first, is blocked if it has a cycle. It results in such cycle is infinitely a trap. In order to get the desired output of infinite paths to each node in cyclic graphs, a Breadth-First Search (BFS) can be used. This is because being Breadth-first means a cycle does not stop searching non-visited nodes to reach other paths. The problem with BFS is that it consumes much more memory, since it keeps more lists of nodes during the running time. Here we use Iterative Deepening DFS (IDDFS) to solve

this issue. A depth-limited search is run repeatedly by increasing the depth-limit on each iteration until it reaches the defined level of depth. This search is equivalent to BFS, but uses much less memory. On each iteration, it visits the nodes in the graph in the same order as DFS, but the accumulated order in which nodes are first visited is effectively breadth-first.

The module works that, given a graph G' from *Reachable* as an input, we find all possible paths between v_s and v_f using IDDFS. To begin with, we determine whether G' has cycles. Detecting cycles can be done by performing a DFS of the entire graph. That is to say, if a back edge is found during any traversal, the graph contains a cycle. Conversely, if all nodes have been visited and no back edge has been found, the graph is acyclic. The problem, we already mentioned in the previous section, is to find all the cycles in the graph while computing and maintaining the weights on edges through traversal. IDDFS, handling a cyclic graph as the way we want it to be, works by looking for the best search depth d, thus starting with depth limit 0 and make BFS and if the search failed it increases the depth limit by 1 and tries BFS again with depth 1 and so on. Initially, first $d = 0$ and it is increased by 1 until a depth d is reached where the goal is found.

Reducing cycles has two functions: *detectCycles* and *fuseCycles*. Given the set P of all relevant paths from (v_s, v_f) in subgraph G' as inputs, *detectCycles* identifies all elementary cycles if any cyclic paths exist in P, generates the reduced graph G'' by eliminating cyclic nodes while combining them as a single solid node and maintaining the aggregated weights. If G' is acyclic, it returns $G'' = G'$. It works by counting node in path p in P. If node $v_i \in p$ appears more than once then $p[p.\text{index}(v_i, 1):p.\text{index}(v_i, 2)]$ is sliced from p and appended to cycle list L. Once it

loops through for all paths in P then it returns *sortedL* which is sorted a set of cycles L in ascending order of their lengths. *fuseCycles* takes the **sortedL** as input and fuses cycle c in **sortedL** conforming to the node reduction rules. When fusing nodes in each cycle c path from **sortedL**, we define a representative node to fuse the cycle as a *combined* node. A combined node is a head of c if c does not include start node v_s and end node v_f. Otherwise combined node has to be chosen among v_s and v_f. If both nodes are contained in c then end node v_f takes priority over start node v_s. Finally it returns a reduced acyclic graph G''.

4.3 Phase 3: Nodes Reduction

Module *Reduction* takes G'' and S as inputs, and returns the probability of reachability from v_s to v_f, which is $\omega(v_s, v_f)$ while performing the reduction rules.

Reduction can be classified into four major functions:

1) **getBridgeEdges** which defines a set of "bridge" and "shared" edges.

2) **getMergeablePaths** which defines mergeable edges and paths.

3) **findPaths** which finds paths between v_s and v_f that having at most one bridge edge.

4) **recursiveReduction** which applies reduction rules to all the relevant intermediate nodes in S.

From the previous phases, we have the reduced subgraph G'' and the set S of intermediate nodes from (v_s, v_f). The function *getBridgeEdges(edge, graph, dictionary)*

and *defineMergeablePaths(edge, graph, dictionary)* take $G'' = (V'', E'')$ as input and return the number of *cuts* and the list of mergeable paths P between v_s and v_s based on the notion of mergeable using topological sort. *findPaths(paths)* takes P as input and returns two sets $\mathcal{BP}_{st}$ and $\mathcal{LP}_{st}$. The final step *recursiveReduction* is done by taking $\mathcal{BP}_{st}$ and $\mathcal{LP}_{st}$ as inputs. It has an inside-function *getSubstringThree(paths)* which takes path p as input and returns a length-three substring of p, that is, every substring is form of (i, k, j). We reduce k by applying the proposed reduction rules recursively until only one edge remains. Then we get the final reachability probability of reaching v_f from v_s as $\omega(v_s, v_f)$.

4.4 Complexity Analysis

Time complexity is a function that describes time performance an algorithm takes with regard to the input size to the algorithm. Time can be considered as the number of times for executed inner loops, or some other natural unit regarding real time process. The fact that many parameters may affect the time performance such as the use of different programming languages, computer hardware specifications, or other related factors, we differentiate our analysis from wall-clock time, which considers only execution time.

In the first phase of getting the relevant subgraph to (v_s, v_f) in a graph G, first we enumerate the reachable nodes from v_s, which is $O|V|$. We then should eliminate nodes that are not reachable to v_f which is $O(|V|)^2$ in worst case. Hence the complexity of phase 1 is $|V| + |V - 1| \times |V|$ for getting the set of relevant nodes and $|V|^2$ for checking the contributing nodes in $\mathcal{R}(v_s)$ to v_f, which is $O(|V|^2)$.

In the second phase of cycle reduction in G', the reduced graph G'' is generated through IDDFS. The space complexity of IDDFS is $O(bd)$, where b is the branching factor which is the number of children at each node and d is the depth of shortest goal. In the worst case that goal will be in the shortest level in the search tree resulting in generating all tree nodes which are $O(bd)$, therefore the time complexity of IDDFS works out to be: $O(b^d)$. Optimal d can be considered based on the size of strongly connected components and the number of nodes in G'. This can be a depth-limited version of depth-first search which is run repeatedly with increasing depth limits until the goal is found within d. As for cycles reduction, in the worst case scenario, identifying and fusing cycles takes $O(n)$ where n is the number of relevant paths that are found through IDDFS. After fusing cycles, we call recursive function to eliminate intermediate nodes between start and end nodes one by one. This recurrence relation is $t(n) = t(n-1) + C$, which means the complexity is $O(V^2)$ [22].

As for the nodes reduction phase in G'' by nodes reduction rules, the worst-case scenario is $O(|V|^2 \times |V|!)$. This is because finding all possible paths is a hard problem, since there are exponential number of simple paths. Suppose a digraph $G = (V, E)$. Then the simple paths in G would be $V!$, and for each of them our algorithm does at least $|V|^2$ computational steps for each node adjacent to the last one in the path, it does a linear scan over the linked list of previously visited nodes. After counting all the intermediate stages of the search, the worst-case complexity is $O(|V|^2 \times |V|!)$.

Chapter 5

Experimental Studies

In this chapter, we describe the implementation of our algorithm on datasets and evaluate its performance. We then analyze the scalability of our algorithm by conducting numerous experiments using various graphs with different sizes and structures of the original datasets.

5.1 Datasets

The experimental evaluation is twofold: "correctness" of our implementation for the proposed reduction algorithm and its performance for large graphs in terms of "scalability". In the first hand, we generated our own graph examples from literature reviews: 21 graphs, shown in 5.1. This is needed to determine whether our implementation for the reduction algorithm yields correct results. We compared these output results with manually calculated results using Excel. If the program arrives at the same results as the manual calculation, our confidence in it is strengthened. In the example of probabilistic graph G in Figure 3.15, the correct weight of $\omega(i, j)$

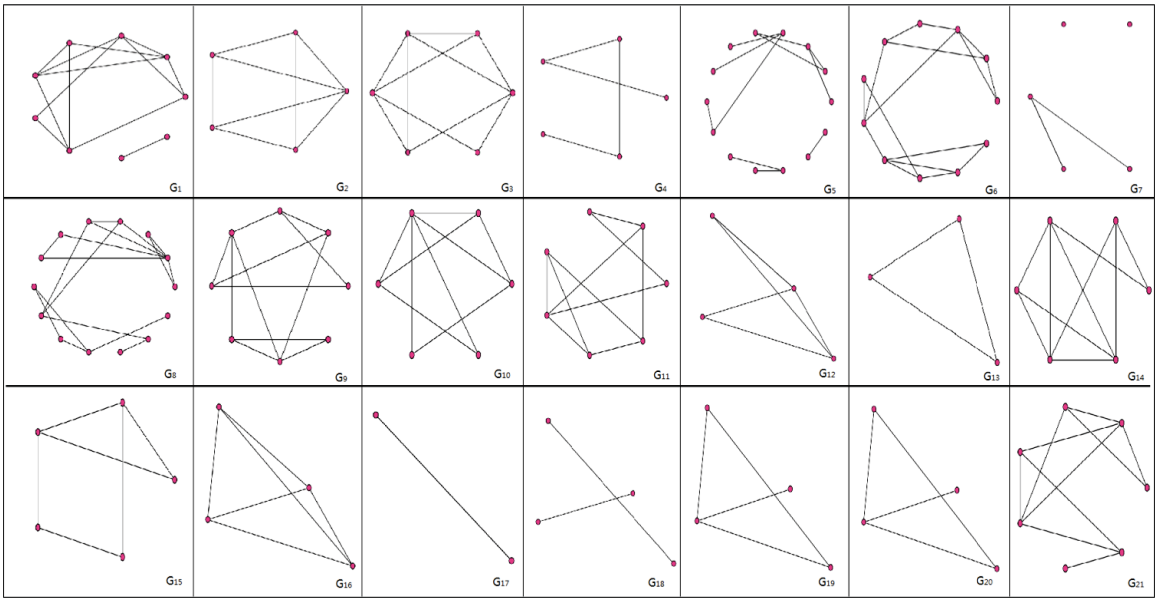


Figure 5.1: Examples of Input Graphs.

is 0.59866252. To see the correctness of our implementation, we ran the program and compared the results with the manual calculation, which is 0.59866252. As can be seen in Figure 5.2, because the results match, we have an increased confidence in the correctness of our program.

In order to evaluate the scalability of the proposed reduction algorithm, the real-world datasets are used: Arxiv HEP-TH[1] (High Energy Physics THeory) citation graph from the *e-print arXiv* which contains all the citations within a dataset of 27,770 papers with 352,807 edges, shown in Table 5.1. This citation graph is a directed

Table 5.1: HEP-PH citation graph data statistics.

Nodes (V)	Edges (E)	V in largest SCC	E in largest SCC	Longest shortest path
27,770	352,807	7,464	116,268	13

[1]https://snap.stanford.edu/data/cit-HepTh.html

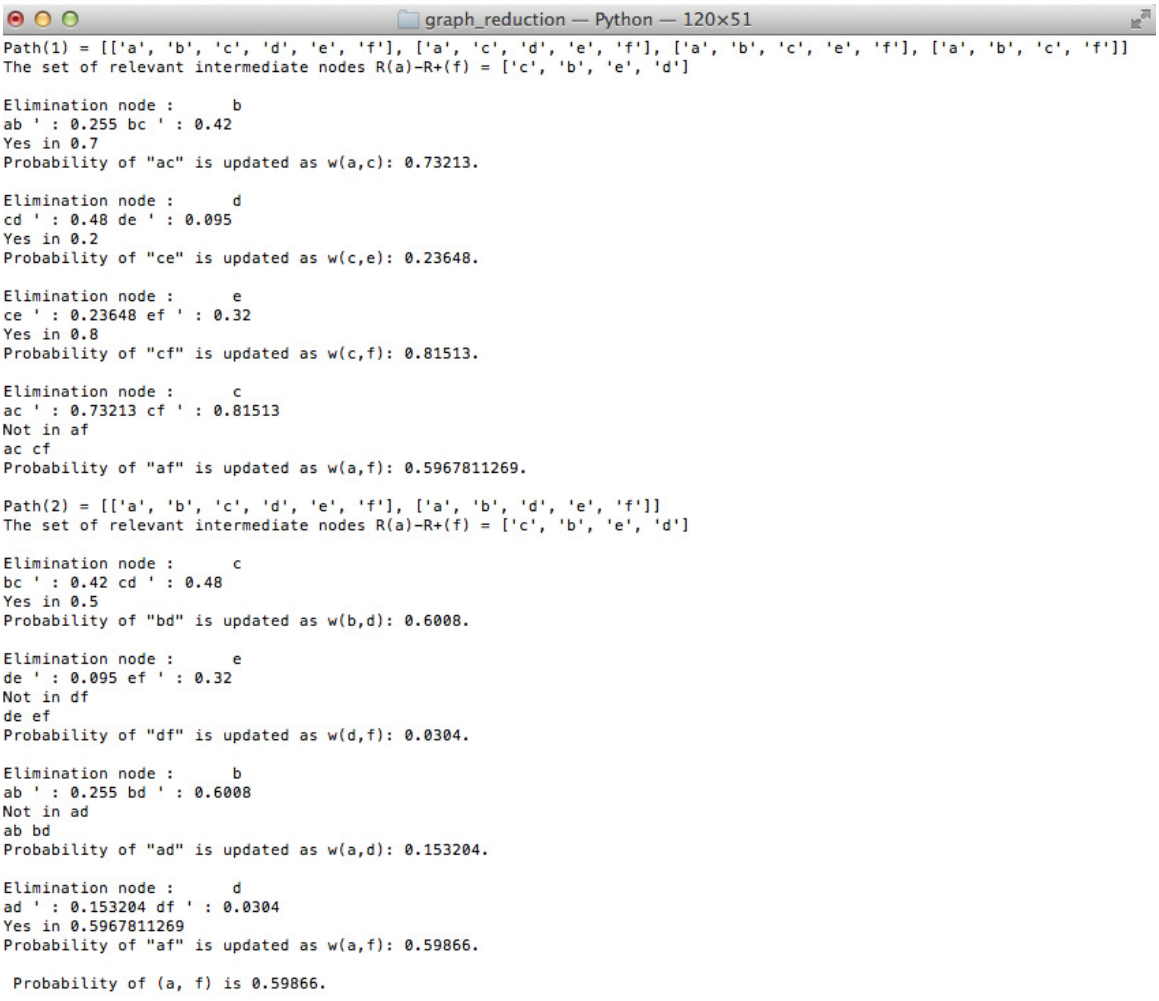


Figure 5.2: Output result of $\omega(i, j)$ in the probabilistic graph of Figure 3.15.

graph, where nodes represent papers, and edges represent citation relationships, that is, if a paper i cites paper j, the graph contains a directed edge from i to j. It covers papers in the period from January 1993 to April 2003, which represents essentially the complete history of its HEP-PH section.

The HEP-PH dataset is composed of two files:

1. cit-HepTh: Paper citation network of Arxiv HEP-TH category

2. cit-HepTh-dates: Time of nodes (paper submission time to Arxiv)

The first dataset contains the citation relationship where each line contains two different user identifiers, namely, *FromNodeId* and *ToNodeId* with SQL standard integer types. The second dataset contains paper submission dates with scientific paper identifier as a primary key. The samples of these datasets are provided in Table 5.2 and Table 5.3, respectively.

Since the provided data is large, the experiments are performed using partial sets of the original data with several times to analyze the scalability of the proposed solution, that is, using sub-graphs of the original graph. It has two databases: cit-HepTh file contains the records for each of the 352,807 citation relations and cit-HepTh-dates file contains 27,700 records of two fields: unique paper identifier and submission dates in the period from January 1993 to April 2003.

With the use of Python library *random.sample(population, k)* which returns a k-length list of unique elements chosen from the population sequence set, we select different sizes of datasets more efficiently, which is a random sampling without replacement: 8 classes of datasets, shown in Figure 5.3. We experimented staring k with size of 100 records/edges and gradually increase the edges by different increments of the number of edges i.e., from 100, then to number of nodes 500, 1000, 2000, 10,000, 50,000, 100,000 and then 150,000. Computing the transitive closure over such sub-graphs is key to find the scalability of our algorithm. We choose such edges randomly from the original dataset.

In this dataset, we are interested in finding the reachability value of a pair of nodes with considered disjunction and conjunction functions. We generate the ex-

Table 5.2: HEP-PH citation graph sample in raw data.

FromNodeId	ToNodeId
9907233	9504304
9704296	9502335
9607354	9504304
9607354	9505235
9502335	9302246
212087	9808221
9808221	9703313

Table 5.3: HEP-PH citation graph with dates sample in raw data.

FromNodeId	ToNodeId
9505235	1995-05-04
9808221	1998-08-04
9808222	1998-08-04
9907233	1999-07-05
9907234	1999-07-05

tended citation graph dataset 3-tuple (*citee*, *citer*, *weight*) and map into a $|E| \times 3$ matrix. Each line contains two different paper identifiers and the associated value as probabilities. If a paper cites, or is cited by, a paper outside the dataset, the graph does not contain any information about this. In order to have edge existence probabilities, we added the field *probability*, associated with a random value in $(0, 1]$ for each record.

```python
import random

if __name__ == '__main__':
    pass

def parse():

    # Initialzation of graph, prob_table, and edges
    graph = {}
    prob_table = {}
    edges = []

    # Open file and read all at once
    with open('HepTH.txt') as cgraph:

        all_edges = cgraph.readlines()
        # Random sampling: 100 of random edges are selected from dataset (all_edges)
        nodes = random.sample(all_edges, 100)
        # A paper is represented as node named 'p' + 'paper ID'
        edges.append(('p'+nodes[0],'p'+nodes[1]))

        # (paper i, paper j) are stroed as (key, value) in graph Dictionary
        if 'p'+nodes[0] in graph:
            graph['p'+nodes[0]].append('p'+nodes[1])
        else:
            graph.update({'p'+nodes[0]:['p'+nodes[1]]})

        if 'p'+nodes[1] not in graph:
            graph.setdefault('p'+nodes[1], [])

        # This generates a random floating point number in the range [0, 1)
        prob_table.update({'p'+nodes[0]+'p'+nodes[1]:round(random.random(),2)})

    cgraph.close()

    return (edges, graph, prob_table)
```

Figure 5.3: Program in Python for Random Sampling.

5.2 Computing the Transitive Closure

It is important to bring out that computing the transitive closure of relations is an uneasy task in terms of the size of input graphs and the presence of cycles. Several approaches were pursued to solve such issues. We first implemented our own function based on the notion of transitivity where the transitive closure of all incoming neighbors are merged to produce the new transitive closure, shown in Figure 5.4. This function, however, takes lots of memory and time since newly generated edges are added to the original database and it runs thorough all the edges in the database repeatedly until no more new edges are found. Our second approach was

```
def transitive_closure(edges):
    closure = set(edges)
    count = 0
    new_path = []
    while True:
        count += 1
        new_path = set((x,w) for x,y in closure for q,w in closure if q == y)
        closure_until_now = closure | new_path
        if closure_until_now == closure:
            break
        closure = closure_until_now

    return closure
```

Figure 5.4: First approach: Function *transitive_closure* snippet.

using Prolog which is the most commonly used logic programming language. It supports non-deterministic programming through backtracking, and pattern matching through unification. We used a interface PySWIP which is a bridge interface in that sending queries to a Prolog database and get responses in Python, shown in Figure 5.5.

The problem with the first and second approaches was execution time. Basically the size of input graphs is quite influential to the transitive closure computations. As can be seen in Figure 5.6, we could not get the transitive closure of G_6, G_7, and G_8. The final conclusion was using well developed built-in library that supports transitive closure computation. NetworkX is a Python language software package for functions of complex networks such as graphs. One of its algorithms handles the transitive closure computation, namely "transitiveclosure" which returns transitive closure of a directed graph[2]. We used this built-in function, which is shown in Figure 5.7. It uses Python library "*dfs_preorder_nodes*" which takes a starting node for depth-first search and return edges in the component reachable from source. A generator of nodes in a depth-first-search pre-ordering. The fact Python provides generators it is

[2]http://orkohunter-networkx.readthedocs.io/en/latest/index.html

```
def transitive_closure_of_edges(graph):
    def generate_edges(graph):
        edges = []
        for node in graph:
            for neighbor in graph[node]:
                print '\n',node, neighbor
                edges.append(node+neighbor)

        return edges

    gen_edges = generate_edges(graph)
    # start a Prolog interpreter instance
    p = Prolog()

    # load the ontology specification
    assertz = Functor('assertz', 1)
    edge = Functor('edge', 2)
    p.assertz('(path(X,Y) :- edge(X,Y))')
    p.assertz('(path(X,Y) :- edge(X,Z), path(Z,Y))')

    for link in gen_edges:
        print link[0], link[1]
        call(assertz(edge(link[0],link[1])))
    # construct look up tables of common facts


    paths = list(p.query('path(X,Y)'))

    tc_edges = []
    for edge in paths:
        if (edge['X'], edge['Y']) not in tc_edges:
            tc_edges.append((edge['X'], edge['Y']))

    return tc_edges
```

Figure 5.5: Second approach: Using Prolog snippet.

more efficient. This is because generators allow for iterative processing of things, one item at a time. This does not cause any issues until how much memory is required when using a normal iterative processing of a list. Basically a large list can take lots of memory. In our case, the algorithm becomes efficient, since we may have a long chain of processes/nodes to compute TC. The generators allow each node to get reachable nodes one at a time. Since function $transitive\_closure\_nx$ takes only acyclic graphs, to handle graphs with or without cycles, we added one more function that takes a set of strongly connected components along with a set of self-loop nodes as inputs. We com-

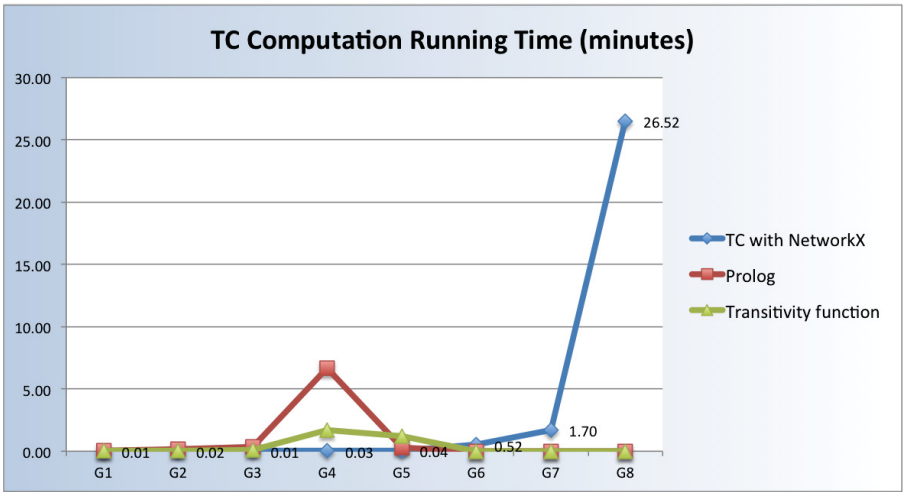


Figure 5.6: Execution time vs. Sub-graph size "without" uncertainties.

```
def transitive_closure_nx(graph):
    def generate_edges(graph):
        edges = []
        for node in graph:
            for neighbor in graph[node]:
                edges.append(node+neighbor)

        return edges

    edges = generate_edges(graph)
    G = nx.DiGraph()
    for v in edges:
        G.add_edge(v[0], v[1])

    edges = list(G.edges())
    print edges

    def transitive_closure(G):
        TC = nx.DiGraph()
        TC.add_nodes_from(G.nodes_iter())
        TC.add_edges_from(G.edges_iter())
        for v in G:
            TC.add_edges_from((v, u) for u in nx.dfs_preorder_nodes(G, source=v)
                              if v != u)
        return TC

    return list(transitive_closure(G).edges())
```

Figure 5.7: Function Transitive closure with NetworkX in Python.

pute the transitive closure of the input graphs. The code is presented in Figure 5.8. To see how it works, consider a graph G with $V = \{a, b, c, d\}$ and $E = \{(a, a), (a, b), (b, a), (b, c), (c, d)\}$. G has four strongly connected components which are {a,b}, {c}

```
ans = input('''\nMenu
1. Edges in the transitive closure of G
2. Find strongly connect components (SCCs) of G
3. Find reachability probability given a pair of target nodes by user
        ''')
  ans = 3
if ans is 1:
    (component_graph, scc, selfloop) = robustTarjan.robust_topological_sort(graph)
    (graph, scc_dict) = robustTarjan.get_reduced_graph(component_graph, scc)
    transitive_closure_edges = robustTarjan.transitive_closure_nx(graph)
    tc = []
    for edge in transitive_closure_edges:
        pre = []
        succ = []
        pre.append(edge[0])
        succ.append(edge[1])
        if tuple(pre) in scc_dict or tuple(succ) in scc_dict:
            if tuple(pre) in scc_dict and tuple(succ) in scc_dict:
                for pre_node in scc_dict[tuple(pre)]:
                    for succ_node in scc_dict[tuple(succ)]:
                        tc.append((pre_node,succ_node))
            elif tuple(pre) in scc_dict and tuple(succ) not in scc_dict:
                for pre_node in scc_dict[tuple(pre)]:
                    tc.append((pre_node, edge[1]))
            elif tuple(pre) not in scc_dict and tuple(succ) in scc_dict:
                for succ_node in scc_dict[tuple(succ)]:
                    tc.append((edge[0], succ_node))

        else:
            tc.append(edge)
    [tc.append((node,node)) for node in selfloop]

    scc = [x for x in scc if len(x)>1]
    for sameclass in scc:
        for component_1 in sameclass:
            for component_2 in sameclass:
                tc.append((component_1, component_2))
```

Figure 5.8: TC computation with SCC components snippet.

and {d}. We are interested in cyclic paths such as (a, a) and (a, b, a). In this case, nodes a and b are combined as a single solid node in our algorithm. Through strong components detection, we generate a lookup table which has a name of combined nodes as key and such components as value where each component is classified based on the existence of self-loop. In this case, the lookup table is $S_1 : [\underline{a}, b]$, where a is a self-loop node. We then have an acyclic graph G_A where it has only two edges (S_1, c) and (c, d). The fact the input graph is acyclic it can be taken as the input for networkX function. The transitive closure of $G = (V, E)$ is a graph $G^* = (V, E^*)$ such that for all x,y in V there is an edge (x, y) in E^* if and only if there is a path

from x to y. Thus, it returns ${E_A}^*$ as (S_1, c),(S_1, d), and (c, d). Additional parts described in Figure 5.4 snippet, for an edge e in ${E_A}^*$ if e includes representative combined component, we replace it with previous components back while generating new edges. For example, (S_1, c) generates $(a, c), (b, c)$ since S_1 has two components. Similarly, (S_1, d) generates $(a, d), (b, d)$ since S_2 has also two components. Once all the edges in ${E_A}^*$ are checked, since nodes in a same SCC are indistinguishable, in other words, for node x and node y in the same SCC there exist paths $(x, y), (y, x), (x, x)$, and y to y itself. We then can add additional edges to ${E_A}^*$ which are $(a, b), (b, a), (a, a)$, and (b, b). The final set of edges in the transitive closure of G is $\{(a, c), (b, c), (a, d), (b, d), (a, b), (b, a), (a, a), (b, b)\}$.

5.3 Results

We performed all the experiments on a MacBook Pro with Linux server, 4 GB memory, and 2.4 GHz Intel Core 2 Duo CPU in Python 2.7. We studied the performance of our proposed algorithm for computing the transitive closure, using the "citation graph" with different sizes created from the HEP-Ph dataset [3], i.e., starting by 100 in the input graph, then to number of nodes 500, 1000, 2000, 10000, 50000, 100000 and then 150000. Clearly, the execution time is exponential in the size of input graph.

$|{E_1}^*|$, the number of edges in the transitive closure (TC) of the first subgraph G_1 where $|V_1| = 73$ and $|E_1| = 100$, is 103. The execution time for both TC and TC with uncertainties is only 1 second. As for G_5 where $|V_3| = 486$, $|E_3| = 1000$, $|{E_3}^*|$ is 3438 and the execution time for TC and TC with uncertainties are 0.01 and 2.53 minutes,

[3] https://snap.stanford.edu/data/cit-HepTh.html

Table 5.4: Execution time on different size of sub-graphs.

				Run-time (minutes)	
Subgraph	|V|	|E|	|E*|	TC	TC with uncertainty
G_1	73	100	103	0.01	0.03
G_2	197	500	1,825	0.02	1.56
G_3	486	1,000	3,438	0.01	2.53
G_4	654	2,000	18,976	0.03	570.63
G_5	5,963	10,000	14,006	0.04	0.04
G_6	19,828	50,000	120,085	0.52	275.73
G_7	18,933	100,000	5,111,122	1.70	-
G_8	18,792	150,000	23,694,798	26.52	-

respectively. Essentially, TC computation time is subject to the graph density in terms of size and structure. Informally, we say that a graph with relatively few edges is sparse, and a graph with many edges is dense. We also can say that a graph G is dense when the number of edges $|E|$ in G is closest to $|V|^2$. It is not surprising that the execution time of computing the transitive closure of graphs depends on the size of graphs and its density. The experimental results on sub-graphs are illustrated in Table 5.4 and Figure 5.8. As can be seen in the execution time table, albeit G_4 has smaller size of nodes and edges than G_5, the fact the density of G_4 is larger than G_5, G_4 takes more time than G_5. It is shown that computing TC of a graph is exponential in the size of graph and newly generated edges through transitivity iterations. To sum up, the execution time is exponential in the size of input graphs.

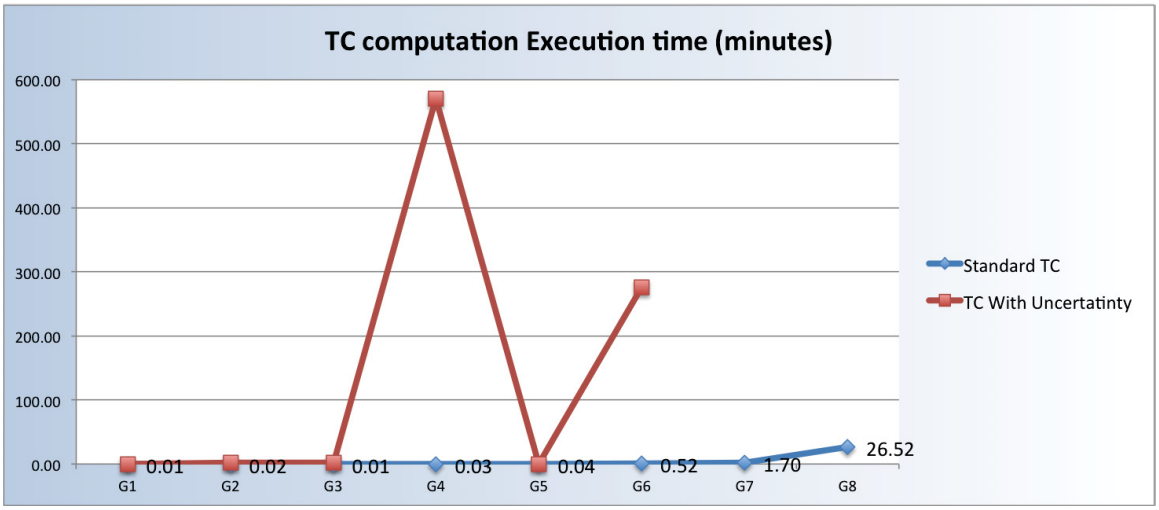


Figure 5.9: Execution time vs. 8 classes of subgraphs.

5.3.1 Scalability Issue

The experiments on the real dataset show that our method is practical. Since the computation time depends on the size of induced subgraph for a given pair of nodes, it is tricky to analyze the exact computation time in the first hand. Though, to show the effectiveness of our algorithm, we define a measure called graph density degree which is the ratio of the number of nodes in an input graph to the number of edges in the transitive closure of the graph: $Density(G) = |V|/|E^*|$. We use this metric in order to demonstrate how efficiently our proposed method can work using different function evaluation as a disjunction mode in very large graphs. Figure 5.10 shows the graph density degree as functions of the number of nodes and edges in sub-graphs. Note that computing an each pair of nodes in sub-graphs utterly depends on the size of relevant sub-graph to such nodes, which explains why computing G_4 takes so much time than G_6.

The main issue to tackle with our algorithm is when the input graph is large,

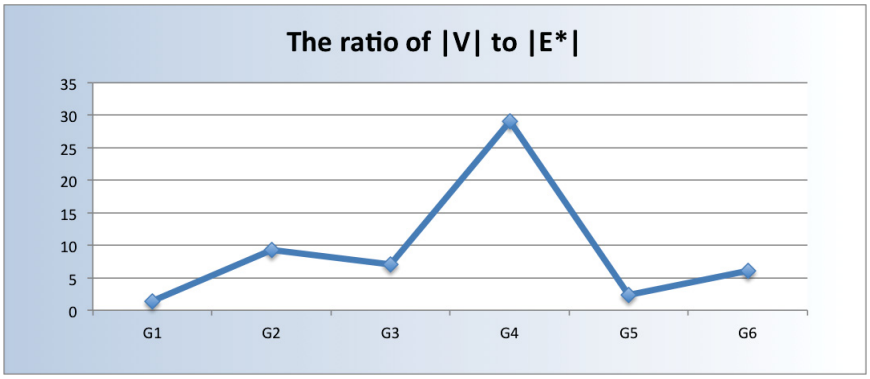


Figure 5.10: The ratio of the number of nodes to the number of transitive closure of edges in 8 classes of Sub-graphs.

there might be exponential number of possible paths between a pair nodes using DFS graph traversal which uses hash indexing with Python data structure "Dictionary" in our case. As for graph G_7, since the number of paths in the graph was 1737030, we had to terminate the transitive closure query (Figure 5.11). This was not a problem for processing the reachability query for a given pair of nodes in the input graph, as our solution would only consider the relevant subgraph. It is important to note that the complexity of either of the transitive closure and reachability problems is exponential, or $O(|V|!)$ to be precise. Basically the fact the order in which nodes are reduced does not affect the end result when using *max*, that is, we do not need simultaneous nodes reduction rules (merge and mesh). In that case, we may use any desired graph traversal algorithm to find the paths to eliminate the nodes one by one in any order, without affecting the result.

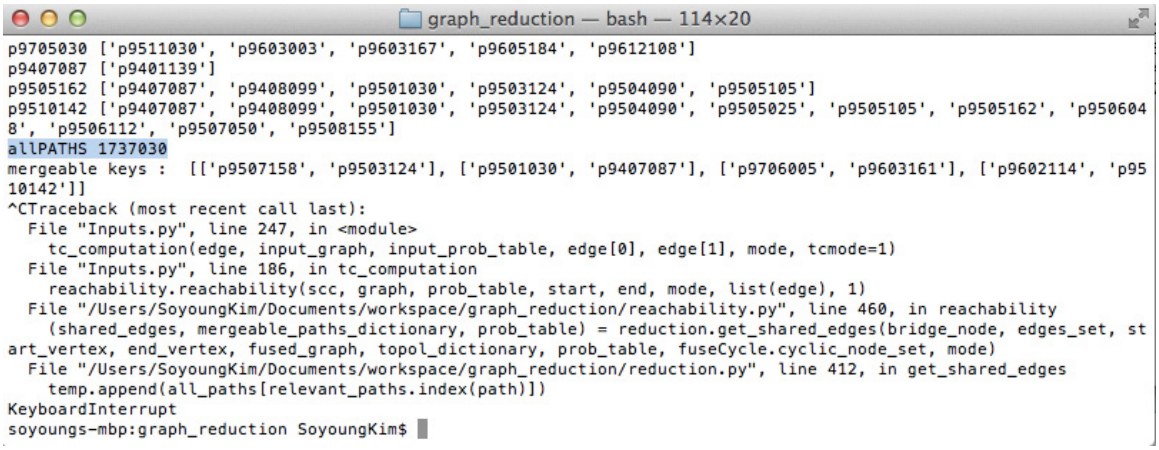


Figure 5.11: The outputs of testing G_7 on terminal.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this paper, we proposed graph reduction algorithms based on the notion of correct semantics for queries over uncertain graphs in terms of the least common subexpressions in more abstract settings and show that different orders of graph reduction may lead to different results for reachability and transitive closure queries. We identified patterns of paths in graphs and introduced its corresponding eduction rules to reduce a graph. We then conducted numerous experiments to see the performance of the proposed solution using various graphs of different sizes in the dataset HEP-TH. The evaluation shows that our algorithm indicates correctness of our implementation as well as effectiveness of the proposed graph reduction method. Potential applications of the proposed solution include development of an extension of the SQL database query language to support transitive closure (TC) and reachability queries over uncertain graphs. We believe that the proposed solution techniques can con-

tribute to existing data management systems for uncertain data.

6.2 Future Research

As future work, the presented algorithms are implementable in their present form, but we need optimizations such as fast graph search for large graphs. It would also naturally bring us into query answering in probabilistic databases. In reachability and TC queries on uncertain and probabilistic data, a tuple is an answer to the query based on tuple-existence uncertainty. It can be extended to data models, which is probabilistic relational data. Then we are only concerned about the certainty values within individual tuples where its value depends on the uncertainty of the tuple itself, which is independent from other tuples. In our research, we have focused on avoiding unnecessary repeated computations in processing queries over probabilistic graphs. In this sense, "read-once functions" can be linked to the way we delved into the query evaluation problem. This approach underlies that, in databases with tuple-independent assumption, the query evaluation is equivalent to computing marginal probabilities of Boolean formulas associated with the tuples in the query result [1]. In that case, the Boolean formulas can be factorized and transformed into a form in which every variable appears at most once. In [26, 24], the authors consider "provenance graphs" in the same read-once based approach, which are directed acyclic graphs represented as event expressions in such a way that most common sub-expressions for the entire relation are not replicated. We plan to explore how our proposed reduction algorithm can be linked with provenance minimization in read-once forms and study how it may contribute to an extension of SQL to support recursion for TC

computations.

6.2.1 Recursive SQL Queries on Graphs with Uncertainty

SQL has been successful in its own impact on the database industry. This comes from strong use of relational algebra which allows set-oriented operations on data formed by rows, columns, and tables. For example, a network model of committee, data can be organized in sets having one set manager (parent) and the several members (child). Such data is called hierarchical model provided all data is captured in a tree/graph structure with records having parent-child-grandchild relationship. Such hierarchical model data is a special case of more general recursive fixed-point queries that compute transitive closures. Recently, hierarchical queries are implemented in Standard SQL:1999 [23]. A recursive query specifies a temporary view set known as a recursive Common Table Expression (CTE). Recursive CTEs can be used to traverse relations as graphs. In other words, we can associate the proposed graph reduction algorithm with computing the transitive closure of a digraph G, denoted by $G^* = (V, E^*)$. It can be regarded as a particular case of transitive closure computation for uncertain graphs in the following sense. Suppose $R(A, B)$ is such a graph and f is a probability function that assigns a probability value to each edge/tuple in R. Then transitive closure of R denoted as R^* includes all pairs of nodes (x, y) in R such that y is reachable from x in R. Now in addition to R and its probability assignment function f, our algorithm also has a pair of nodes (v_s, v_f) as an input and returns the probability of reaching from v_f to v_f. This is "simply" done by computing the transitive closure R^* and then select the tuple (v_s, v_f) in R^* with it

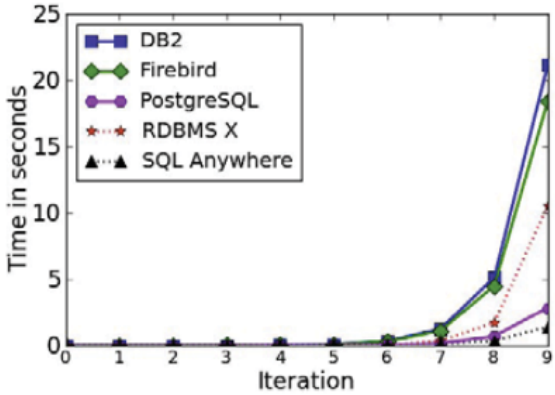
associated probability value, but rather than computing the entire tuples in R^*. Our algorithm does this more "directly" and hence more efficiently since it considers only the relevant nodes/paths in R. So if a database contains the table $R(A, B, v)$, which is an edge from A to B with probability v, the algorithm we implemented computes (v_s, v_f, p), which returns the probability f for the path from v_s to v_f. Moreover if this path has cycles, our algorithm also handles it, more efficiently while in standard transitive closure computation, the computation is slow when the path is acyclic. The whole process is hidden by the user, hence the user can only see the front-end query results.

To see the effectiveness of our algorithm, consider the survey [23] in which the authors investigated the implementation of recursive SQL Common Table Expressions in most of the popular DBMSs such as : IBM DB2 Express-C 9.5, RDBMS X, Sybase SQL Anywhere 11, PostgreSQL 8.4, Firebird 2.1.3 and Microsoft SQL Server 2008. Database schema for the tested data-sets consists of the following relations: CITIES(cid, city) contains 200 records, TRAINS(departure, arrival, railline, tid, price) contains 800 records, and FLIGHTS(departure, arrival, carrier, fid, price) contains 800 records. One of the queries, Q1, in this paper displays all the cities reachable by plane starting from Toronto, the number of connections is limited by the parameter $I(= 1 \cdots 9)$ which limits the recursion depth, shown in Figure 6.1. Note that for cyclic data, using $I = 9$ was enough to fully exhaust the system resources and in many cases caused the database system to crash. As can be seen, the execution time is 2 seconds for depth limit 9. To compare this with our proposed solution, let us consider example $G_3 = (V_3, E_3)$, where E_3 includes 1000 edges, for which the execution time to compute the transitive closure was 0.7892 seconds. This could be

```
WITH destinations (origin, departure, arrival, connections) AS
  (SELECT a.departure, a.departure, a.arrival, 1
   FROM flights a, cities c
   WHERE a.departure = c.cid AND c.city = 'Toronto'
  UNION ALL
   SELECT r.origin, b.departure, b.arrival, r.connections + 1
   FROM destinations r, flights b
   WHERE r.arrival = b.departure AND r.flight_count < I )
SELECT count(*) FROM destinations
```

Listing 1.1. Query Q1

(a) Query Q1



(b) Performance results of Q1 for different DBMS engines

Figure 6.1: (a) Query 1 and (b) its results on a given data.

done faster when searching relevant nodes from a given single node to start with, as opposed to computing the transitive closure of the input relation. This means that the reachability queries over uncertain relations can be evaluated using our proposed technique rather efficiently as expected by restricting to the relevant part of the input graph.

Bibliography

[1] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. *ACM Transactions on Database Systems (TODS)*, 37(4):30, 2012.

[2] K. Arijit and C. Lei. On uncertain graphs modeling and queries. *Proceedings of the VLDB Endowment*, 8(12):2042–2043, 2015.

[3] C. Baier and M. Grosser. Recognizing ω-regular languages with probabilistic automata. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 137–146. IEEE, 2005.

[4] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1316–1325. ACM, 2014.

[5] K. Chatterjee, L. Doyen, and T. A. Henzinger. Probabilistic weighted automata. In *International Conference on Concurrency Theory*, pages 244–258. Springer, 2009.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[7] G. S. Fishman. A comparison of four monte carlo methods for estimating the probability of st connectedness. *IEEE Transactions on reliability*, 35(2):145–155, 1986.

[8] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[9] M. Hua and J. Pei. Probabilistic path queries in road networks: traffic uncertainty aware path selection. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 347–358. ACM, 2010.

[10] J. Jarvis and D. R. Shier. Graph-theoretic analysis of finite markov chains. *Applied mathematical modeling: a multidisciplinary approach*, 1999.

[11] R. Jin, L. Liu, and C. C. Aggarwal. Discovering highly reliable subgraphs in uncertain graphs. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 992–1000. ACM, 2011.

[12] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-constraint reachability computation in uncertain graphs. *Proceedings of the VLDB Endowment*, 4(9):551–562, 2011.

[13] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[14] P. Linz. *An Introduction to Formal Languages and Automata 5th edtion.* John Wiley and Sons Ltd.,Chichester, fifth edition, 2012.

[15] L. Lü and T. Zhou. Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications*, 390(6):1150–1170, 2011.

[16] S. Maniu, R. Cheng, and P. Senellart. Probtree: A query-efficient representation of probabilistic graphs. In *1st International Workshop on Big Uncertain Data, BUDA 2014*, 2014.

[17] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[18] M. Mohri. Weighted finite-state transducer algorithms. an overview. In *Formal Languages and Applications*, pages 551–563. Springer, 2004.

[19] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[20] P. Parchas. Uncertain graph processing through representative instances and sparsification. In *VLDB PhD Workshop, Hawaii, USA*, 2015.

[21] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. K-nearest neighbors in uncertain graphs. *Proceedings of the VLDB Endowment*, 3(1-2):997–1008, 2010.

[22] W. H. Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[23] P. Przymus, A. Boniewicz, M. Burzańska, and K. Stencel. Recursive query facilities in relational databases: a survey. In *Database Theory and Application, Bio-Science and Bio-Technology*, pages 89–99. Springer, 2010.

[24] S. Roy, V. Perduca, and V. Tannen. Faster query answering in probabilistic databases using read-once functions. In *Proceedings of the 14th International Conference on Database Theory*, pages 232–243. ACM, 2011.

[25] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 7–7. IEEE, 2006.

[26] P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. *Proceedings of the VLDB Endowment*, 3(1-2):1068–1079, 2010.

[27] I. Sevo. Probabilitic graphs. *Ser. Math. Inform.*, 28(1):27–42, 2013.

[28] P. Sevon, L. Eronen, P. Hintsanen, K. Kulovesi, and H. Toivonen. Link discovery in graphs derived from biological databases. In *International Workshop on Data Integration in the Life Sciences*, pages 35–49. Springer, 2006.

[29] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.