Towards GPU-based Common-sense Reasoning: Using Fast Subgraph Matching

Ha-Nguyen Tran*, Erik Cambria*, and Amir Hussain†

* School of Computer Science and Engineering, Nanyang Technological University
† School of Natural Sciences, University of Stirling
{hntran,cambria}@ntu.edu.sg,ahu@cs.stir.ac.uk

Abstract. Common-sense reasoning is concerned with simulating cognitive human ability to make presumptions about the type and essence of ordinary situations encountered every day. The most popular way to represent common-sense knowledge is in the form of a semantic graph. Such type of knowledge, however, is known to be rather extensive: the more concepts added in the graph, the harder and slower it becomes to apply standard graph-mining techniques. In this work, we propose a new fast subgraph matching approach to overcome these issues. Subgraph matching is the task of finding all matches of a query graph in a large data graph, which is known to be a non-deterministic polynomial time (NP)-complete problem. Many algorithms have been previously proposed to solve this problem using Central Processing Units (CPUs). In this paper, we present a new Graphics Processing Unit (GPU)-friendly method for common-sense subgraph matching, termed GpSense, which is designed for scalable massively-parallel architectures, to enable next-generation Big Data sentiment analysis and Natural Language Processing (NLP) applications. We show that GpSense outperforms state-of-the-art algorithms and efficiently answers subgraph queries on large common-sense graphs.

1 Introduction

Communication is one of the most important aspects of human cognitive capabilities. Effective communication always has a cost in terms of energy and time, due to information needing to be encoded, transmitted, and decoded, and sometimes such factors can be critical to human life. This is why people normally only provide useful information when communicating, and take the rest for granted. This 'taken for granted' information is termed 'common-sense' knowledge – obvious things people know and usually leave unstated.

Common-sense is not the kind of knowledge we can find in Wikipedia, but comprises all the basic relationships among words, concepts, phrases, and thoughts that allow people to communicate with each other and face everyday life problems. It is a kind of knowledge that sounds obvious and natural to us, but it is actually daedal and multi-faceted. The illusion of simplicity comes from the fact that, as each new group of skills matures, we build more layers on top and tend to forget about previous layers.

Common-sense, in fact, is not a simple thing, rather it should be considered a cognitive repository of practical ideas, with multitudes of life-learned rules and exceptions, dispositions and tendencies, balances and checks [18].

Common-sense computing [4] has been applied to many branches of artificial intelligence, e.g., personality detection [21], handwritten text recognition [29], and social data analysis [7]. In the context of sentic computing [3], in particular, common-sense is represented as a semantic network of natural language concepts interconnected by semantic relations. Besides the methodological problem of relevance (selection of relevant nodes during spreading activation), this kind of representation presents two major implementation issues: performance and scalability, both due to the many new nodes, or natural language concepts learnt through crowdsourcing [6], continuously integrating into the graph. These issues are also crucial problems of querying and reasoning over large-scale common-sense knowledge bases (KBs).

The core function of common-sense reasoning is subgraph matching which is defined as, finding all the matches of a query graph in a database graph. Subgraph matching is usually a bottleneck for the overall performance as it involves subgraph isomorphism which is known as an NP-complete problem [8]. Previous methods for subgraph matching are backtracking algorithms [27, 9, 10, 13], with novel techniques for filtering candidates sets and re-arranging visit order. These algorithms, however, are designed to work only in small-graph settings. The number of candidates grows significantly in medium-to-large-scale graphs, resulting in an exorbitant number of costly verification operations. Several indexing techniques have also been proposed for faster computation [30, 31]; however, the enormous index size makes them impractical for large data graphs [25]. Distributed computing methods [1, 25] have been introduced to deal with large graphs by utilizing parallelism, yet there remains the open problem of high communication costs between the participating machines.

Recently, Graphics Processing Units (GPUs) have become popular computing devices owing to their massive parallel execution power. Fundamental graph algorithms including breadth-first search [11, 14, 17], shortest path [11, 16], and minimum spanning tree [28] on large-scale graphs can be efficiently implemented on GPUs. The previous backtracking methods for subgraph matching, however, cannot be straightforwardly applied to GPUs due to their inefficient use of GPU memories and SIMD-optimized GPU multi-processors [15].

In this paper, we propose *GpSense*, an efficient and scalable method for common-sense reasoning and querying via subgraph matching. GpSense is based on a *filtering-and-joining* strategy which is designed for the massively parallel architecture of GPUs. In order to optimize the performance in depth, we utilize a series of optimization techniques which contribute towards increasing GPU occupancy, reducing workload imbalances and in particular speeding up subgraph matching on common-sense graphs.

Many conmon-sense knowledge graphs, however, contain millions to billions of nodes and edges. These huge graphs cannot be stored on the memory of a single GPU device. We may thus have to use main memory and even hard-

disk, if necessary, as the main storage of the knowledge graphs. To address the issue, we propose a *multiple-level graph compression* technique to reduce graph sizes while preserving all subgraph matching results. The graph compression method converts the data graph to a weighted graph which is small enough to be maintained in GPU memory. We then present a complete GpSense solution which exploits the weighted graph to solve subgraph matching problems.

The rest of this paper is organized as follows: Section 2 introduces background to the subgraph matching problem and related definitions; Section 3 discusses how to transform common-sense KBs to directed graphs; Section 4 gives an overview of the filtering-and-joining approach to solve the subgraph matching problem on GPUs; In section 5, we discuss our graph representation and graph compression method; Section 6 presents the complete GpSense algorithm in detail; Section 7 shows the results of comparative performance evaluations; finally, Section 8 summarizes and concludes the paper.

2 Preliminaries

This section outlines the formal problem statement and introduces the fundamental definitions used in this paper.

2.1 Problem Definition

A graph G is defined as a 4-tuple (V, E, L, l), where V is the set of nodes, E is the set of edges, L is the set of labels and l is a labeling function that maps each node or edge to a label in L. We define the size of a graph G is the number of edges, size(G) =$|E|$.

Definition 1 (Subgraph Isomorphism). *A graph G = (V, E, L, l) is* subgraph isomorphic *to another graph G' = (V', E', L', l'), denoted as $G \subseteq G'$, if there is an injective function (or a* match*) f: $V \rightarrow V'$, such that $\forall$ (u, v) $\in$ E, (f(u), f(v)) $\in$ E', l(u) = l'(f(u)), l(v) = l'(f(v)), and l(u, v) = l(f(u),f(v)).*

A graph G is called a subgraph of another graph G (or G is a supergraph of G), denoted as $G \subseteq G'$ (or $G' \supseteq G$), if there exists a subgraph isomorphism from G to $G'a$.

Definition 2 (Subgraph Matching). *Given a small query graph Q and a large data graph G,* subgraph matching *problem is to find all subgraph isomorphisms of Q in G.*

Next, we explain some basic concepts used in the paper such as matches of a query node, adjacency lists and candidates set of a node.

Definition 3 (Candidate Node). *Given a query graph $Q = (V, E, L, l)$ and a data graph $G = (V', E', L', l')$, a node $v \in V'$ is called a* candidate *or* match *of a node $u \in V$ if $l(u) = l'(v)$, $degree(u) \leq degree(v)$ where $degree(u)$, $degree(v)$ are the number of nodes connected to edges starting node u and v respectively. The set of candidates of u is called* candidates set *of u, denoted as $C(u)$.*

An *adjacency list* of a node u in a graph G is a set of nodes which are the destinations of edges starting from u, denoted as *adj(u)*.

2.2 Subgraph Matching Algorithms

Most state-of-the-art subgraph matching algorithms are based on backtracking strategies, which find matches by either forming partial solutions incrementally, or pruning them if they cannot produce the final results, as discussed in the works of Ullman [27], VF2 [9], QuickSI [24], GADDI [30], GraphQL [13] and SPath [31]. One open issue in these methods is the selection of matching order (or visit order). To address this issue, TurboISO [10] introduces strategies of candidate region exploration and combine-and-permute to compute a good visit order, which makes the matching process efficient and robust.

To deal with large graphs, Sun et al. [25] recently introduced a parallel and distributed algorithm (which we call STW in this paper), in which they decompose the query graphs into 2-level trees, and apply graph exploration and a joint strategy to obtain solutions in a parallel manner over a distributed memory cloud. Unlike STW, our method uses GPUs to preserve the advantages of parallelism during computation, while simultaneously avoiding high communication costs between participating machines.

2.3 Graphics Processing Units (GPUs)

In our study, we adopt GPUs and Nvidia CUDA as our development platform. A GPU is connected to a CPU through a high speed IO bus slot, typically a PCI-Express in current high performance systems. The current PCI-Express v4 is over 30 GB/sec. Each GPU has its own device memory, called *global memory*, up to several gigabytes in current configurations. A GPU consists of a number of *stream multiprocessors* (SMs), each of which executes in parallel with the others. Each SM has multiple *stream processors* (SPs) for parallel execution. The stream processors in a multiprocessor execute in SIMT (Single Instruction, Multiple Thread) fashion in which stream processors in the one SM execute the same instruction at the same time. There is also a small data cache attached to each multiprocessor, called *shared memory*. Shared memory can be accessed by all threads in a SM. This is a low-latency, high-bandwidth, indexable memory which runs at register speeds.

A program running on the GPU is called a kernel, which consists of many thread blocks (groups of threads). Thread blocks are assigned on a stream multiprocessor for parallel execution. Each thread of a thread block is processed on a stream processor in the SM. Since SPs are grouped to share a single instruction unit, threads mapped on these SPs execute the same instruction each cycle, but on different data (i.e., Single Instruction Multiple Data, or SIMD). Once a thread block is assigned to a stream multiprocessor, it is further divided into 32-thread units called *warps*. A warp is the unit of thread scheduling in SMs. When the threads in a warp issue a device memory operation, the instruction is

very slow, usually taking up hundreds of clock cycles, due to the long memory latency. GPUs tolerate memory latency by using a high degree of multi-threading. When one warp stalls on a memory operation, the multiprocessor selects another active warp and switches to one with little overhead.

To best support graphics processors for general purpose computation, several GPGPU (General-Purpose computing on GPUs) languages such as NVIDIA CUDA [1] are available for developers to write programs on GPUs easily. CUDA interface uses standard C code with parallel programming features.

3 Common-sense as a Graph

In this section, we discuss how a common-sense KB can be naturally represented as a graph and how such a KB can be directly transformed to a graph representation.

3.1 Common-sense Knowledge Graph

Instead of formalizing common-sense reasoning using mathematical logic [19], some recent common-sense KBs, e.g., SenticNet [5], represent data in the form of a semantic network and make it available for use in natural language processing (NLP) applications. In particular, the collected pieces of knowledge are integrated in the semantic network as triples, using the format: $< concept\text{-}relation\text{-}concept >$. By considering triples as directed labeled edges, the KB naturally becomes a directed graph. Figure 1 shows a semantic graph representation for part of a common-sense knowledge graph.

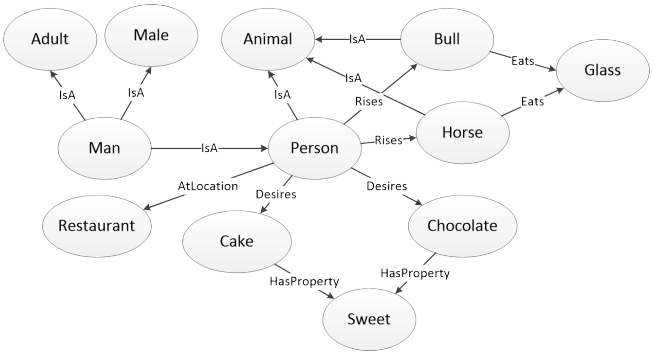


Fig. 1. Common-sense knowledge graph

3.2 Common-sense Graph Transformation

This subsection describes how to directly transform a common-sense KB to a directed graph. The simplest way for transformation is to convert the KB to a

[1] https://developer.nvidia.com/what-cuda

flat graph using direct transformation. This method maps concepts to node IDs, and relations to labels of edges. Note the obtained graph contains no node labels as each node is mapped to a unique ID. Table 1 and 2 show the mapping from concepts and relations of the common-sense KB in Figure 1 to node IDs and edge labels. The transformed graph from the KB is depicted in Figure 2.

Concept	**Node ID**
Adult	v_0
Male	v_1
Man	v_2
Restaurant	v_3
Person	v_4
Animal	v_5
Cake	v_6
Chocolate	v_7
Sweet	v_8
Bull	v_9
House	v_{10}
Glass	v_{11}

Table 1. Node Mapping Table

Relation	**Edge Label**
IsA	r_0
Rises	r_1
AtLocation	r_2
Desires	r_3
Eats	r_4
HasProperty	r_5

Table 2. Edge Label Mapping Table

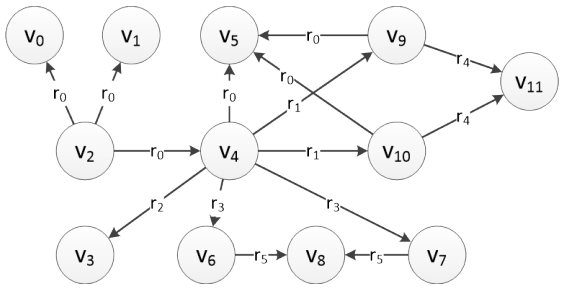
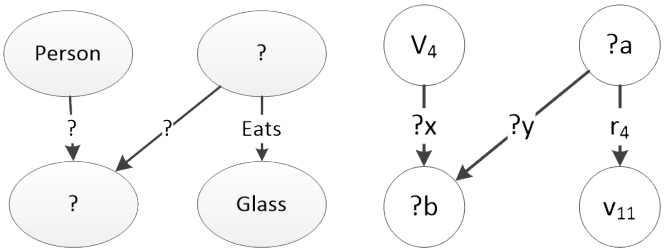


Fig. 2. Direct transform of common-sense KB

In the general subgraph matching problem, all nodes of a query graph q are variables. In order to produce the subgraph isomorphisms of q in a large data graph g, we must find the matches of all query nodes. Unlike the general problem, query graphs in common-sense querying and reasoning tasks contain two types of nodes: concept nodes and variable nodes.

A concept node can only be mapped to one node ID in the data graphs while a variable node may have many node candidates. Similarly, query edges are also categorized into variable and labeled edges. Figure 3 illustrates the conversion of a common-sense query to a directed query graph.

In the sample query transformation, the query concepts *Person* and *Glass* correspond to two data nodes with IDs of v_4 and v_{11}. The relation *Eats* is mapped to the edge label r_4. The query graph also contains 2 variable edges: $?x$, $?y$ and 2 variable nodes: $?a$, $?b$. The direct transformation is a simple and common approach to naturally convert a semantic network to a directed graph.



(a) Common-sense query (b) Transformed query

Fig. 3. Direct transformation of common-sense query

4 GPU-based Subgraph Matching

In this subsection, we introduce a parallel approach to solve the subgraph matching problem on General-Purpose Graphics Processing Units (GPGPUs). Before describing the algorithm in detail, we explain how a data graph is represented in memory. In order to support graph query answering on GPUs, we use two arrays to represent a graph $G = (V, E)$: *nodes array* and *edges array*. The edges array stores the adjacency lists of all nodes in V, from the first node to the last. The nodes array stores the start indexes of the adjacency lists, where the i-th element of the nodes array has the start index of the adjacency list of the i-th node in V. These arrays have been used in previous GPU-based algorithms [11, 14, 17]. Two additional arrays with the lengths of $|V|$ and $|E|$ are used to stored the labels of nodes and edges.

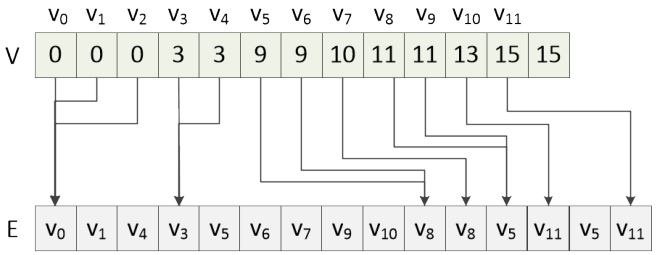


Fig. 4. Graph representation of the data graph in Figure 2

Based on the above graph structure, we propose a simple and efficient subgraph matching algorithm. The approach is based on a *filtering-and-joining* strategy which is specially designed for massively parallel computing architectures of modern GPUs [26]. The main routine of the GPU-based method is depicted in Algorithm 1.

Algorithm 1: GPUSubgraphMatching (q(V, E, L), g(V', E', L'))

Input: query graph q, data graph g
Output: all matches of q in g

```
1  P := generate_query_plan(q, g);
2  forall the node u ∈ P do
3      if u is not filtered then
4          c_set(u) := identify_node_candidates(u, g);
5      c_array(u) := collect_edge_candidates(c_set(u));
6      c_set := filter_neighbor_candidates(c_array(u), q, g);
7  refine_node_candidates(c_set, q, g);
8  forall the edge e (u,v) ∈ E do
9      EC(e) := collect_edge_candidates(e, c_set, q, g);
10 M := combine_edge_candidates(EC, q, g);
11 return M
```

The inputs of the algorithm are a query graph q and a data graph g. The output is a set of subgraph isomorphisms (or matches) of q in g. In the method, we present a match as a list of pairs of a query node and its mapped data node. Our solution is the collection M of such lists. Based on the input graphs, we first generate a query plan for the subgraph matching task (Line 1). The query plan contains the order of query nodes which will be processed in the next steps. The query plan generation is the only step that runs on the CPU. The main procedure will then be executed in two phases: filtering phase (Line 2-7) and joining phase (Line 8-10). In the filtering phase, we filter out node candidates which cannot be matched to any query nodes (Line 2-6).

Upon completion of this task, there still exists a large set of irrelevant node candidates which cannot contribute to subgraph matching solutions. The second task continues pruning this collection by calling the refining function *refine_node_candidates*. In such a function, candidate sets of query nodes are recursively refined until no more can be pruned. The joining phase then finds the candidates of all data edges (Line 8-9) and merges them to produce the final subgraph matching results (Line 10).

Query Plan Generation: *generate_query_plan* procedure is used to create a good node order for the main searching task. It first picks a query node which potentially contributes to minimizing the sizes of candidate sets of query nodes and edges. The number of candidates at the beginning is unknown, so we can estimate it using a node ranking function $f(u) = \frac{deg(u)}{freq(u.label)}$ [10, 25], where

$deg(u)$ is the degree of a query node u and $freq(u.label)$ is the number of data nodes having the same label as u. The score function prefers lower frequencies and higher degrees. Once the first node is chosen, the *generate_query_plan* follows its neighborhood to find the next node which is unselected and connected to at least one node in the node order. The process terminates once all query nodes are chosen.

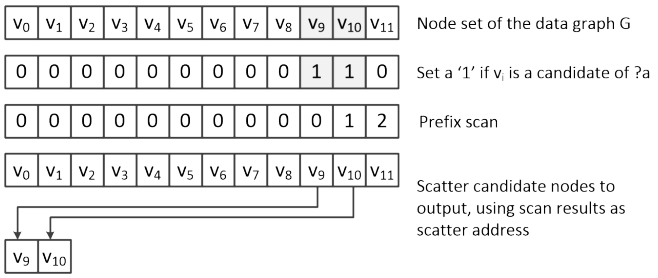


Fig. 5. Collect candidate nodes of $?a$

The Filtering Phase: The purpose of this phase is to reduce the number of node candidates, resulting in a decrease in edge candidates, along with the running time of the joining phase. The filtering phase consists of two tasks: initializing node candidates and refining node candidates. In order to maintain the candidate sets of query nodes, for each query node u we use a *boolean* array, $c\_set[u]$, which has the length of $|V'|$. If $v \in V'$ is a candidate of u, *identify_node_candidates* sets the value of $c\_set[u][v]$ to $true$. The *filter_neighbor_candidates* function, however, will suffer from a low occupancy problem since only threads associated with $true$ elements of $c\_set[u]$ are functional while the other threads are idle. To deal with the problem, *collect_node_candidates* collects $true$ elements of $c\_set[u]$ into an array $c\_array[u]$ by adopting a stream compaction algorithm [12] to gather elements with the $true$ values in $c\_set[u]$ to the output array $c\_array[u]$. The algorithm employs a prefix scan to calculate the output addresses and to support writing of the results in parallel. The example of collecting candidate nodes of $?a$ is depicted in Figure 5. By taking advantage of c_array, candidate nodes v_9, v_{10} can easily be mapped to consecutive active threads. As a result, our method achieves a high occupancy.

After that the *filter_neighbor_candidates* function will filter the candidates of nodes adjacent to u based on $c\_array[u]$. Inspired by the warp-based methods used in BFS algorithms for GPUs [14], we assign to each warp a candidate node $u' \in c\_array[u]$. Within the warp, consecutive threads find the candidates of $v \in adj(u)$ in $adj(u')$. This method takes advantage of coalesced access as the nodes of $adj(u')$ are stored next to each other in memory. It also addresses the warp divergence problem since threads within the warp execute similar operations. Thus, our method efficiently deals with the workload imbalance problem between

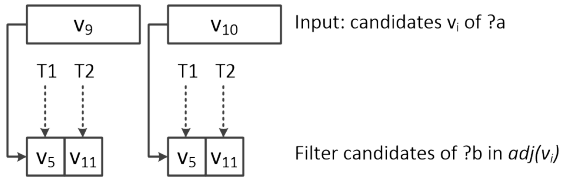


Fig. 6. Filter candidates of ?b based on candidate set of ?a

threads in a warp. Figure 6 shows an example of filtering candidate nodes of ?b based on the candidate set of ?a, $C(?a) = \{v_9, v_{10}\}$.

If a data node has an exceptionally large degree compared to the others, our algorithm deals with it by using an entire block instead of a warp. This solution reduces the workload imbalance between warps within the block.

The Joining Phase: Based on the candidate sets of query nodes, *collect_edge_candidates* function collects the edge candidates individually. The routine of the function is similar to *filter_neighbor_candidates*, but it inserts an additional portion of writing obtained edge candidates. In order to output the candidates to an array, we employ the *two-step output scheme* [13] to find offsets of the outputs in the array and then write them to the corresponding positions. *combine_edge_candidates* merges candidate edges using a warp-centric approach to produce the final subgraph matching solutions. The threads within the warp i should share the partial solution, called $M_i(q)$, and access them frequently. We thus store and maintain $M_i(q)$ in the shared memory instead of the device memory, which efficiently hides the memory stalls.

Issues with large-scale common-sense reasoning: Despite the fact the algorithm can deal with subgraph matching on general graphs efficiently, there still remain a number of issues for applying the approach to common-sense reasoning, specifically: 1) Unlike query graphs in general subgraph matching problems, common-sense query graphs contain concept nodes and variable nodes. We only need to find the matches of nodes in a subset of variable nodes, termed *projection*; 2) Many common-sense knowledge graphs contain millions to billions of nodes and edges. These huge graphs cannot be stored on the memory of a single GPU device. We may thus have to use main memory and even a hard-disk, if necessary, as the main storage of knowledge graphs.

To overcome these issues, the next section introduces a graph compression method to decrease the size of data graphs. Following this, we describe the complete implementation of our method and a series of optimization techniques to enhance the performance of common-sense reasoning.

5 Multi-level Graph Compression

Due to the large size of the data graph, it cannot be maintained within the memory of a single GPU device. The next offline computation aims to reduce the data graph size such that we can fit it into GPU memory while still preserving

the subgraph matching solutions of any query graphs in the original data graph. In a random labeled graph, the distribution of nodes and edges are unpredictable. However, a common-sense knowledge graph contains a lot of similar nodes which share the same group of nodes in their adjacency lists. For example, v_0 and v_1 of the data graphs in Figure 2 are similar nodes they have the same adjacency list. As a result, the two nodes play the same role in the data graphs and can be combined into one hyper-node.

Based on the above observation, we apply a *multi-level compression* technique to compress the data graph. During the graph compressing process, a sequence of smaller graphs $G_i = (V_i, E_i)$ are constructed from the original graph $G = (V, E)$. At each level i, similar nodes are combined to form a weighted node which is defined later. The set of nodes which are combined into the weighted node u after i levels called the *mapping list* of u, denoted as $M(u)$. The compressing task terminates when the size of G_i is small enough to be maintained in GPU memory, as depicted in Figure 7. The final mapping lists are stored in main memory. At each label i, graph G_i is a weighted graph which is defined as follows:

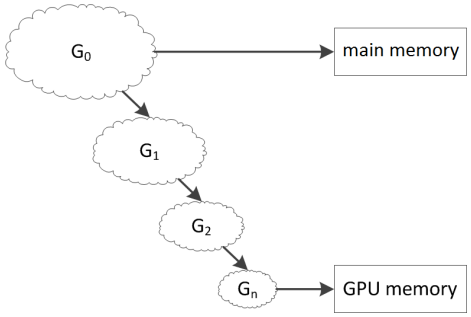


Fig. 7. Multi-level graph compression

Definition 4. *A* weighted graph *at level i is a 5-tuple $G_i = (V_i, E_i, L, l, w)$ where V_i is the set of nodes, E_i is the set of edges, L is the set of labels, l is a labeling function that maps each node to a label in L and w is a weighting function that maps each node or edge to an integer value.*

Each *weighted node* $u \in V_i$ is a combination of $p, q \in V_{i-1}$ and $w(u) = \max(|\{adj(x) \bigcap (M(p) \bigcup M(q)) \mid x \in M(p) \bigcup M(q)\}|)$. Generally, the weight of node u is the maximum degree among nodes in the graph constructed by $M(p) \bigcup M(q)$.

For each *weighted edge* (u, v) starting from u, if $v \in V_i$ is a combination of $n, m \in V_{i-1}$ then $w(u, v) = \max(w(p, n), w(q, n)) + \max(w(p, m), w(q, m))$. Note the initial weight of all edges in the original graph is 1.

An edge to/from v is called a *common edge* two nodes u_1 and u_2 if there exists two edges (u_1, v) and (u_2, v) such that $l(u_1, v) = l(u_2, v) = l_u$, denoted

as $e(l_u, v)$. In the Figure 2, $e(r_0, v_2)$ is a common edge of v_0 and v_1. The list of common edges between u_1 and u_2 is denoted as $common(u_1, u_2)$.

Given a user-defined threshold δ such that $0 < \delta \leq 1$, u and v are called *similar nodes* if $max(|adj(u)|/|common(u,v)|, |adj(v)|/|common(u,v)|) \geq \delta$. These similar nodes, thus, can be combined into a hyper-node in the next graph compression level. By using δ, we can easily adjust the ratio of graph compression at each level.

Weighted Nodes	Mapping List
u'_0	v_0, v_1
u'_1	v_2
u'_2	v_3
u'_3	v_4
u'_4	v_5
u'_5	v_6, v_7
u'_6	v_8
u'_7	v_9, v_{10}
u'_8	v_{11}

Table 3. Mapping list of nodes

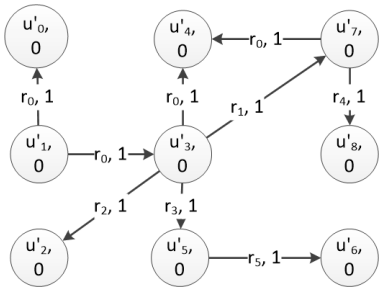


Fig. 8. A sample weighted data graph

Assume that the data graph is the common-sense knowledge graph in Figure 2. After the first level of data graph compression with δ of 1, we obtain a sample weighted data graph G_1 as in Figure 8. Each node is presented as a circle with a label and a weight. At this level, we combine the following pairs of nodes into weighted nodes: (v_0, v_1), (v_6, v_7), (v_9, v_{10}). The mapping lists of nodes in G_1 are illustrated in Table 3. For the real common-sense knowledge graph, i.e. *SenticNet*, the compression ratio is illustrated in Table 4. The ratio is calculated as the total number of nodes and edges of the compressed graph divided by that of the original graph.

Level	Threshole δ	Ratio
1	0.8	61.4%
2	0.7	46.2%
3	0.7	32.2%

Table 4. Compression ratio of SenticNet

The weighted graph G_w, which is obtained after reducing the size of the original data graph, is used for checking subgraph matching solutions of given query graphs. Due to the differences in graph structures of G_w and the original data graph G, we can re-define candidates (or matches) of a query node, as follows:

Definition 5. *Given a query graph $Q = (V, E, L, l)$ and a weighted data graph $G_w = (V_w, E_w, L_w, l_w, w)$, a node $v \in V_w$ is considered as a* candidate *of a node $u \in V$ if $l(u) = l_w(v)$, degree(u) $\leq$ w(v) + $\sum$ w(v,z) where $z \in adj(v)$, denoted as weight(z).*

For example, node u'_7 is a candidate of $?a$ in the query graph in Figure‘3 since $degree(?a) = 2$ which is smaller than $w(u'_7) + w(u'_7, u'_4) + w(u'_7, u'_8) = 2$. Similarly, u'_1 and u'_3 are also candidate nodes of $?a$.

Theorem 1. *Given a query graph $Q = (V_q, E_q)$, a data graph $G = (V, E)$ and a weighted graph $G_w = (V_w, E_w)$ which is the compression result of G. If a node $v \in V$ is a candidate of node $u \in V_q$ then node $x \in V_w$ such that $v \in M(x)$ is also a candidate of u.*

Proof. We need to prove two conditions: 1) u, v, and x have the same label because v is a candidate of u and $v \in M(x)$. 2) Based on the definition of weighted graphs, we can see that degree(v) $\leq$ w(x) + $\sum$ w(x,z) where $z \in adj(x)$ or weight(x). Therefore, degree(u) $\leq$ weight(x). As a result, x is a candidate of u.

Theorem 2. *For each node $u \in V_q$, if node $z \in W_w$ is not a match of u in any subgraph matching solution of Q in G_w then all nodes $v \in M(z)$ are not matches of u in any subgraph matching solution of Q in G.*

Proof. We prove by contradiction. Suppose that there exists a node $v \in Q$ which is in a subgraph matching solution of Q in G, but node z is such that $v \in M(z)$ is not. According to the definition of the above subgraph isomorphism, there is an injective function f: $V_q \rightarrow V$ such that $\forall$ (x, y) $\in E_q$, (f(x), f(y)) $\in E$, l(x) = l(f(x)), l(y) = l(f(y)), and v = f(u). We can see that $\forall$ $(a, b) \in E$, $a \in M(p)$, $b \in M(q)$, $(p, q) \in E_w$. Let a function g: $V_q \rightarrow V_w$ such that $\forall$ $x \in V_q$. $x \in M(g(x))$. Clearly, $f \circ g$ is a subgraph isomorphism from Q to G_w and $z = f \circ g(u)$. This contradicts that z is not in any subgraph matching solution.

6 GpSense

Based on the multi-level graph compression method introduced in the previous section, we propose a complete algorithm for subgraph matching on large-scale common-sense knowledge graphs using GPUs. Figure 9 gives us an overview of the proposed method, termed *GpSense*, for subgraph matching on large common-sense graphs, which cannot fit the global memory of a single GPU device, using both GPUs and CPUs. Rectangles denote tasks while the others represent data structures used in the method.

Our *GpSense* subgraph matching solution comprises two separate tasks: an offline task containing graph compression, and online query answering. Initially, the data graph G is stored in the main memory due to its large size. For offline processes, we start by creating a data structure for the input data graph, as described in Section 4. The data graph can be maintained in a hard-disk or main

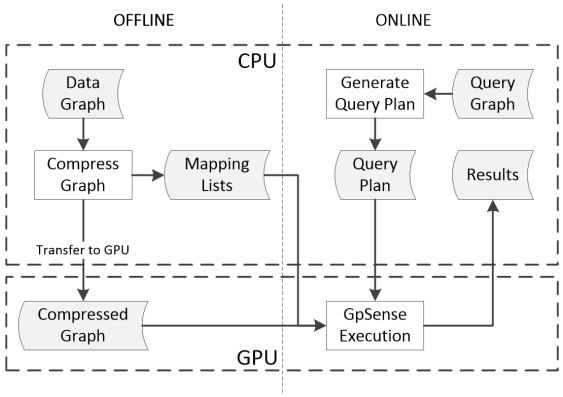


Fig. 9. GpSense overview

memory depending on the size of the data graph and main memory. Assuming we use main memory as the storage of the created index, we then compress the data graph using a multiple-level approach until the obtained graph G' can fit into GPU memory. All mapping lists are also maintained in the main memory. The compressed data graph G' , then, is transferred to GPU memory and stored for GPU execution.

In the online query answering task, after receiving a graph query Q, *GpSense* generates a query plan for the input query graph. The obtained query plan is then transferred to GPU memory. Following this, our method applies the Algorithm 1 on the weighted graph achieved by the graph compression step, to find the subgraph matching results on the GPU. If no solution is found, we can conclude there is no subgraph matching solution from Q to G. Otherwise, based on the achieved solutions and the in-memory mapping lists, we continue searching for the final subgraph matching solutions of Q in G.

Algorithm 1, however, is designed for solving the subgraph matching on a general graph. In order to adapt the algorithm to common-sense reasoning, we introduce some optimization techniques to enhance the performance of *GpSense* on large-scale common-sense knowledge graphs as follows:

Modify the query plan based on the properties of common-sense queries. First, unlike query graphs in general subgraph matching problems, common-sense query graphs contain concept nodes and variable nodes. We only need to find the matches of nodes in a subset of variable nodes, termed *projection*. Second, nodes of a common-sense knowledge graph are not labelled, but mapped to node IDs. Therefore, the frequency of a concept node in a query is 1 and that of a variable node is equal to the number of data nodes. As a result, the ranking function used for choosing the node visiting order cannot work for common-sense subgraph matching.

Based on the above observations, we can make a modification to generate the node order as follows: we prefer picking a concept node u with the maximum

degrees as the first node in the order. By choosing u, we can minimize the candidates of variable nodes connected to u. The next query node v will be selected if v is connected to u and the adjacency list of v consists of the maximum number of nodes which is not in the order among the remaining nodes. We continue the process until edges connected to nodes in the node order can cover the query graph.

Use both incoming and outgoing graph representations: An incoming graph is built based on the incoming edges to the nodes while an outgoing graph is based on the outgoing edges from the nodes. The representation of commonsense graph in Figure 4 is an example of outgoing graph representation. Given a query graph in Figure 3, we assume using only an outgoing graph as the data graph. Based on the above query plan generator, node v_4 is the first node in the order. We then filter the candidates of $?b$ based on v_4. Since $?b$ does not have any outgoing edges, we have to pick $?a$ as the next node and find its candidates by scanning all the data graphs. There are however, some issues with this approach: 1) We need to spend time to scan all the data graph nodes. 2) The number of candidates can be very large as the filtering condition is weak. To overcome this problem, we use an incoming graph along with the given outgoing graph. By using the additional graph, candidates of $?a$ can be easily filtered based on the candidate set of $?b$. The number of candidates of $?a$, therefore, is much smaller than that in the previous approach. Consequently, GpSense can reduce many of the intermediate results during execution, which is a key challenge for GPU applications.

Only use one-time refinement: Ideally, the optimal candidate sets of query nodes are obtained when the refinement is recursively invoked until no candidate is removed from the candidate sets. However, our experiments show most irrelevant candidates are pruned in the first round. The later rounds do not prune out many candidates, but lead to inefficiency and reduce the overall performance. Also, we observe that if the node visiting order is reversed during the refinement, GpSense is more efficient in terms of minimizing the intermediate data, as well as in improving performance.

7 Experiments

We evaluate the performance of GpSense in comparison with state-of-the-art subgraph matching algorithms, including VF2 [9], QuickSI (QSI) [24], GraphQL (GQL) [13] and TurboISO [10]. The experiments are conducted on SenticNet and its extensions [22, 23]. The query graphs are extracted from the data graph by picking a node in SenticNet and following breadth first search (BFS) to select other nodes. We choose nodes in the dense area of SenticNet to ensure the obtained queries are not just trees.

The runtime of the CPU-based algorithms is measured using an Intel Core i7-870 2.93 GHz CPU with 8GB of memory. Our GPU algorithms are tested using the CUDA Toolkit 6.0 running on NVIDIA Tesla C2050 GPU with 3 GB global memory and 48 KB shared memory per Stream Multiprocessor. For each

of those tests, we execute 100 different queries and record the average elapsed time. In all experiments, algorithms terminate only when all subgraph matching solutions are found.

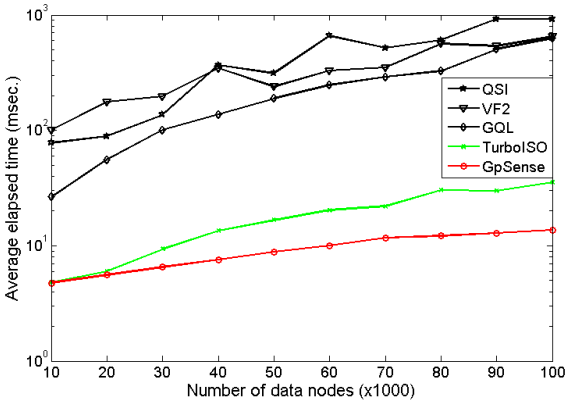


Fig. 10. Comparison with state-of-the-art methods

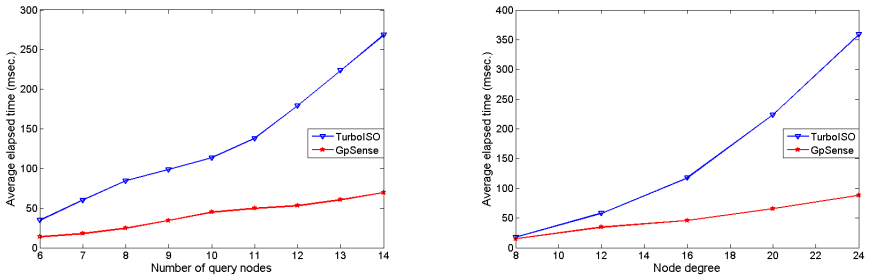
7.1 Comparison with state-of-the-art CPU algorithms

The first set of experiments is to evaluate the performance of GpSense on SenticNet and compare it with state-of-the-art algorithms. SenticNet is a commonsense knowledge graph of about 100,000 nodes, which is primarily used for sentiment analysis [2]. In this experiment, we extract subsets of SenticNet with the size varying from 10,000 to 100,000 nodes. All the data graphs can fit into GPU memory. The query graphs contain 6 nodes.

Figure 10 shows that GpSense clearly outperforms VF2, QuickSI and GraphQL. Compared to TurboISO, our GPU-based algorithm obtains similar performance when the size of the data graphs is relatively small (i.e., 10,000 nodes). However, when the size of data graphs increases, GpSense is more efficient than TurboISO.

Figure 11a shows the performance results of GpSense and TurboISO on the query graphs whose numbers of nodes vary from 6 to 14. Figure 11b shows their performance results when the node degree increases from 8 to 24, where the number of query nodes is fixed to 10. As can be seen in the two figures, the performance of TurboISO drops significantly while that of GpSense does not.

This may be due to the number of recursive calls of TurboISO growing exponentially with respect to the size of query graphs and the degree of the data graph. In contrast, GpSense, with a large number of parallel threads, can handle multiple candidate nodes and edges at the same time, thus its performance remains stable.



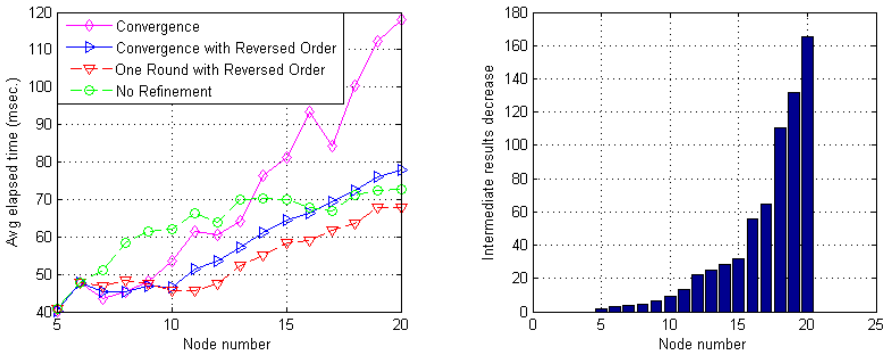
(a) Varying query sizes (b) Varying average degrees

Fig. 11. Comparison with TurboISO

7.2 Effect of Optimization Techniques

Here, we carry out a series of experiments to demonstrate improvements of the proposed refinement function. Figure 12a shows a comparison between *GpSM* with and without the Candidates Refinement function in terms of average elapsed time. We compare four different versions of *GpSense*. The first version implements the refinement function until convergence. The second version is identical to the first apart from reversing the node visit order after the candidates set?s initialization. The third version stops refining after the first round, and also reverses the node visit order. The fourth version does not employ the refinement function. As shown in Figure 12a, the response time is faster when using reversed node visiting order, compared to the original order, and the *GpSense* with a limited number of iterations (i.e. the 3rd version) exhibits the best performance among the four implemented versions.

Figure 12b illustrates the effect of optimization techniques for refinement and two-data graphs utilization. In terms of intermediate results size, when the size of query graph is 20 nodes, the amount of memory that *GpSense* needs to maintain the intermediate results, without use of these techniaues, is up to 150 times more than *GpSense* using refinement and two-data graphs utilization.



(a) Refinement runing time (b) Intermediate results reduction

Fig. 12. Effect of optimization techniques

7.3 Scalability Test

We tested GpSense's scalability against ConceptNet. The number of data nodes varies from 100,000 to 200 million nodes. The data graph is stored as follows: When the data graph is small, i.e., from 100,000 to 20 million nodes, we store it in the GPU global memory. If the node number of the data graph is between 20 million and 200 million, CPU memory is used to maintain the data graph. The number of query nodes is 6.

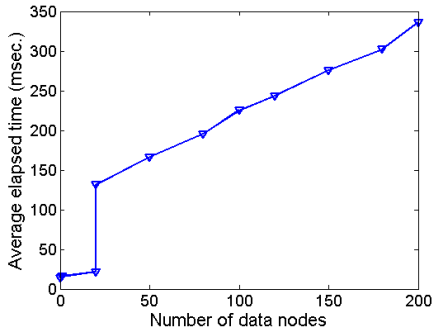


Fig. 13. Scalability tests

When the data graph size is 20 million nodes, we perform two experiments. The first maintains the whole data graph in GPU memory and the second uses CPU memory. As shown in Figure 13, the second experiment answers subgraph matching queries slower than the first experiment, due to the time taken for data transfer from CPU memory to GPU memory.

8 Conclusion

In this paper, we introduced an efficient GPU-friendly method for answering subgraph matching queries over large-scale common-sense KBs. Our proposed method, GpSense, is based on a *filtering-and-joining* approach which is shown to be suitable for execution on massively parallel GPU architectures. Along with efficient GPU techniques of coalescence, warp-based and shared memory utilization, GpSense provides a series of optimization techniques which contribute to enhancing the performance of subgraph matching-based common-sense reasoning tasks. We also present a *multi-level graph compression* method to reduce the size of data graphs which cannot fit into GPU memory, but still preserve query answering correctness. Simulation results show that our method outperforms state-of-the-art backtracking-based algorithms on CPUs, and can efficiently answer subgraph matching queries on large-scale common-sense KBs. For future work, GpSense will be exploited to enable real-time implementation of our newly proposed multi-modal NLP and Big Data sentiment analysis approaches [20, 3].

Conflict of Interest

The authors have received no grants. All the authors declare they have no conflict of interest.

Compliance with Ethical Standards

This study was supported by the National Natural Science Foundation of China (NNSFC) (Grant Numbers 61402386, 61305061 and 61402389). A. Hussain was supported by the Royal Society of Edinburgh (RSE) and NNSFC joint project grant no. 61411130162, and the UK Engineering and Physical Science Research Council (EPSRC) grant no. EP/M026981/1. We also wish to thank the anonymous reviewers who helped improve the quality of the paper.

Ethical Approval

This article does not contain any studies with human participants or animals performed by any of the authors

References

1. M. Brocheler, A. Pugliese, and V. S. Subrahmanian. Cosi: Cloud oriented subgraph identification in massive social networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*, pages 248–255. IEEE, 2010.
2. E. Cambria. Affective computing and sentiment analysis. *IEEE Intelligent Systems*, 31(2):102–107, 2016.
3. E. Cambria and A. Hussain. *Sentic computing: a common-sense-based framework for concept-level sentiment analysis*, volume 1. Springer, 2015.
4. E. Cambria, A. Hussain, C. Havasi, and C. Eckl. *Common sense computing: from the society of mind to digital intuition and beyond*, pages 252–259. Springer, 2009.
5. E. Cambria, D. Olsher, and D. Rajagopal. SenticNet 3: A common and commonsense knowledge base for cognition-driven sentiment analysis. In *AAAI*, pages 1515–1521, Quebec City, 2014.
6. E. Cambria, D. Rajagopal, K. Kwok, and J. Sepulveda. GECKA: Game engine for commonsense knowledge acquisition. In *FLAIRS*, pages 282–287, 2015.
7. E. Cambria, H. Wang, and B. White. Guest editorial: Big social data analysis. *Knowledge-Based Systems*, 69:1–2, 2014.
8. S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.
9. L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

10. W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2013.
11. P. Harish and P. Narayanan. *Accelerating large graph algorithms on the GPU using CUDA*, pages 197–208. Springer, 2007.
12. M. Harris, S. Sengupta, and J. D. Owens. Gpu gems 3, chapter parallel prefix sum (scan) with cuda. 2007.
13. H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.
14. S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, volume 46, pages 267–276. ACM, 2011.
15. J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova. *Lessons learned from exploring the backtracking paradigm on the GPU*, pages 425–437. Springer, 2011.
16. G. J. Katz and J. T. Kider Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55. Eurographics Association, 2008.
17. D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.
18. M. Minsky. *Society of mind*. Simon and Schuster, 1988.
19. E. T. Mueller. *Commonsense Reasoning: An Event Calculus Based Approach*. Morgan Kaufmann, 2014.
20. S. Poria, E. Cambria, N. Howard, G.-B. Huang, and A. Hussain. Fusing audio, visual and textual clues for sentiment analysis from multimodal content. *Neurocomputing*, 174:50–59, 2016.
21. S. Poria, A. Gelbukh, B. Agarwal, E. Cambria, and N. Howard. *Common sense knowledge based personality recognition from text*, pages 484–496. Springer, 2013.
22. S. Poria, A. Gelbukh, E. Cambria, D. Das, and S. Bandyopadhyay. Enriching senticnet polarity scores through semi-supervised fuzzy clustering. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, pages 709–716. IEEE, 2012.
23. S. Poria, A. Gelbukh, E. Cambria, P. Yang, A. Hussain, and T. S. Durrani. Merging senticnet and wordnet-affect emotion lists for sentiment analysis. In *Signal Processing (ICSP), 2012 IEEE 11th International Conference on*, volume 2, pages 1251–1255. IEEE, 2012.
24. H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.
25. Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.
26. H.-N. Tran, J.-j. Kim, and B. He. Fast subgraph matching on large graphs using graphics processors. In *Database Systems for Advanced Applications*, pages 299–315. Springer, 2015.
27. J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.
28. V. Vineet, P. Harish, S. Patidar, and P. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 167–171. ACM, 2009.

29. Q.-F. Wang, E. Cambria, C.-L. Liu, and A. Hussain. Common sense knowledge for handwritten chinese text recognition. *Cognitive Computation*, 5(2):234–242, 2013.
30. S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 192–203. ACM, 2009.
31. P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.