GpSense: A GPU-friendly method for common-sense subgraph matching in massively parallel architectures

Ha-Nguyen Tran and Erik Cambria

School of Computer Science and Engineering
Nanyang Technological University
{hntran,cambria}@ntu.edu.sg

Abstract. In the context of common-sense reasoning, spreading activation is used to select relevant concepts in a graph of common-sense knowledge. When such a graph starts growing, however, the number of relevant concepts selected during spreading activation tends to diminish. In the literature, such an issue has been addressed in different ways but two other important issues have been rather under-researched, namely: performance and scalability. Both issues are caused by the fact that many new nodes, i.e., natural language concepts, are continuously integrated into the graph. Both issues can be solved by means of GPU accelerated computing, which offers unprecedented performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. To this end, we propose a GPU-friendly method, termed *GpSense*, which is designed for massively parallel architectures to accelerate the tasks of common-sense querying and reasoning via subgraph matching. We show that GpSense outperforms the state-of-the-art algorithms and efficiently answers subgraph queries on a large common-sense graph.

1 Introduction

When communicating with each other, people provide just the useful information and take the rest for granted. This 'taken for granted' information is what is termed 'common-sense' – obvious things people normally know and usually leave unstated. Common-sense is not the kind of knowledge we can find in Wikipedia, but it consists in all the basic relationships among words, concepts, phrases, and thoughts that allow people to communicate with each other and face everyday life problems. Common-sense is an immense society of hard-earned practical ideas, of multitudes of life-learned rules and exceptions, dispositions and tendencies, balances and checks.

Common-sense computing [3] has been applied in many different branches of artificial intelligence, e.g., personality detection [19], handwritten text recognition [26], multimodality [18], and social data analysis [6]. In the context of sentic computing [2], in particular, common-sense is represented as a semantic network of natural language concepts interconnected by semantic relations.

This kind of representation presents two major implementation issues: performance and scalability, both due to the fact that many new concepts learnt through crowd-sourcing [5] are continuously integrated into the graph. These issues are also the crucial problems of querying and reasoning over large-scale commonsense knowledge bases (KBs). The core function of commonsense querying and reasoning is subgraph matching which is defined as finding all matches of a query graph in a database graph. Subgraph matching is usually a bottleneck for the overall performance because it involves subgraph isomorphism which is known as an NP-complete problem [7].

Many proposed methods for subgraph matching problem are backtracking algorithms [24, 8, 9, 12], with novel techniques for filtering candidates sets and re-arranging visit order. However, all of them use only small database graphs, and thus, there still remains the question of scalability on large graphs. Some recent methods utilize indexing techniques to deal with large graphs, through building the index may take long time and large memory space [22, 27, 28]. Another approach is based on distributed computing [1, 23]. By finding results on many machines simultaneously, those algorithms are able to deal with large-scale graphs. However, an issue with these methods is that the communication between a large number of machines is costly.

Graphics Processing Units (GPUs) have recently become popular computing devices because of their massive parallel. Such basic graph operations as breadth-first search [10, 13, 16], shortest path [10, 15], and minimum spanning tree [25] on large graphs can be implemented on GPUs efficiently. The previous backtracking methods for subgraph matching, however, cannot be straightforwardly applied to GPUs due to their inefficient uses of GPU memories and SIMD-optimized GPU multi-processors [14].

In this paper, we propose *GpSense*, an efficient and scalable method for solving the subgraph matching problem on large common-sense KBs. GpSense is based on a *filtering-and-joining* strategy which is designed for massively parallel architecture of GPUs. In order to optimize the performance, we utilize a series of optimization techniques which contribute to increase the GPU occupancy, reduce workload imbalance and especially enhance common-sense reasoning tasks. The rest of the paper is structured as follows: Section 2 introduces the background of the subgraph matching problem and filtering-and-joining approach; Section 3 discusses how to transform common-sense KBs to directed graphs; Section 4 describes the GPU implementation in details; experiment results are shown in Section 5; finally, Section 6 concludes the paper.

2 Subgraph Matching Problem

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 ⊆ G', if there is an injective function (or a* match*) f: V → V', such that ∀ (u, v) ∈ E, (f(u), f(v)) ∈ 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.*

2.2 GPU Approach for Subgraph Matching

In this subsection, we introduce an approach to solve the subgraph matching problem on General-Purpose Graphics Processing Units (GPGPUs). The approach is based on a *filtering-and-joining* strategy which is specially designed for massively parallel computing architecture of modern GPUs. 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. After that, the main procedure will 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).

After 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 candidates 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 to create a good node order for the main searching task. It first picks a query node which potentially contributes to minimize the sizes of candidate sets of query nodes and edges. Since we do not know the number of candidates in the beginning, we estimate it by using a node ranking function $f(u) = \frac{deg(u)}{freq(u.label)}$ [9, 23], 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. After choosing the first node, *generate_query_plan* follows its neighborhood to find the next nodes which has not been selected and is connected to at least one node in the node order. The process terminates when all query nodes are chosen.

The Filtering Phase: The purpose of this phase is to reduce the number of node candidates and thus decrease the amount of edge candidates as well as 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 suffers the low occupancy problem since only threads associated with *true* elements of $c\_set[u]$ works while the other threads are idle. To deal with the problem, *collect_node_candidates* collects *true* elements of $c\_set[u]$ into an array $c\_array[u]$. As a result, each running thread can easily be mapped to a candidate of u. After that the *filter_neighbor_candidates* function will filter the candidates of nodes adjacent to u based on $c\_array[u]$. This device function follows a warp-based execution approach. The details of these parallel functions will be discussed in Section 4.

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 part of writing obtained edge candidates to candidate edge arrays. In order to output the candidates to an array, we employ the *two-step output scheme* [12] to find the 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 fashion to produce the final subgraph matching solutions.

3 Common-sense Knowledge 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 [17], some recent common-sense KBs, e.g., SenticNet [4], represent data in the form of a semantic network and make it available to be used in natural language processing. In particular, the collected pieces of knowledge are integrated in the semantic network as triples of the format $< \textit{concept-relation-concept} >$. By considering triples as directed labeled edges, the KB naturally becomes a directed graph. Figure 1 shows a semantic graph representation of a part of common-sense knowledge graph for the concept *cake*.

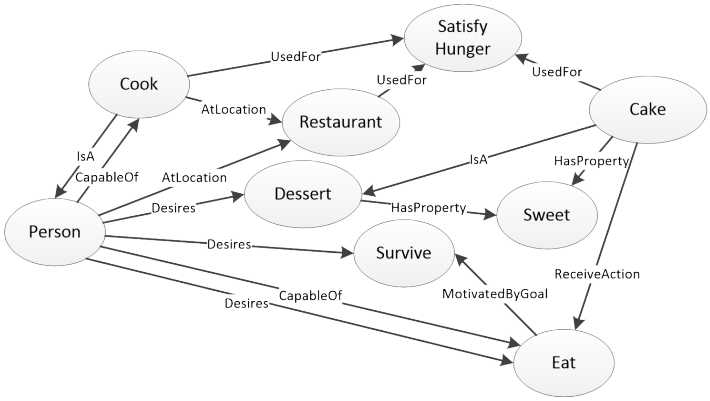


Fig. 1. Common-sense knowledge graph

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 flat graph using direct transformation. This method maps concepts to node IDs and maps relations to labels of edges. Note that the obtained graph contains no node labels because 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 KD is depicted in Figure 2.

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, the query graphs in common-sense querying and reasoning tasks contain two types of nodes: concept nodes and variable nodes.

Concept	Node ID
Person	v_0
Cook	v_1
Restaurant	v_2
Dessert	v_3
Survive	v_4
Eat	v_5
Satisfy Hunger	v_6
Sweet	v_7
Cake	v_8

Table 1. Node Mapping Table

Relation	Edge Label
IsA	r_0
CapableOf	r_1
AtLocation	r_2
Desires	r_3
UsedFor	r_4
HasProperty	r_5
MotivatedBy	r_6
ReceiveAction	r_7

Table 2. Edge Label Mapping Table

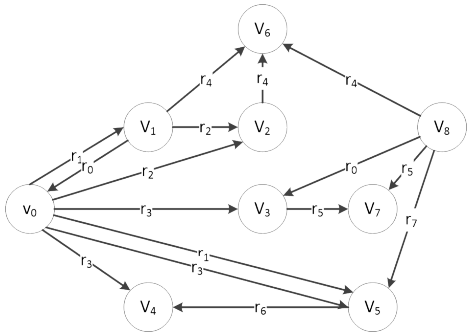
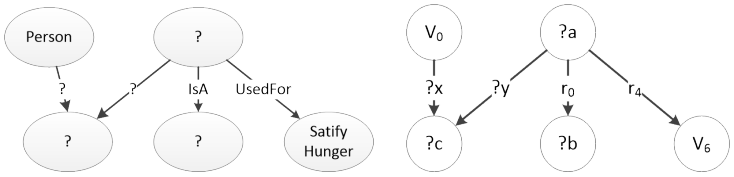


Fig. 2. Direct transform of Common-sense KB

A concept node can only be mapped to one node ID in the data graphs while a variable node may have many node candidates in the data graph. Similarly, query edges are also categorized into variable edges 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 *Satisfy Hunger* correspond to two data nodes with IDs of v_0 and v_6. Two query relations *IsA* and *UsedFor* are mapped to edge labels r_0 and r_4. The query graph also contains 2 variable edges $?x$, $?y$ and 3 variable nodes $?a$, $?b$, $?c$. The direct transformation is a common and simple approach to naturally convert a semantic network to a directed graph.



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

Fig. 3. Direct transformation of Common-sense query

4 Common-sense Subgraph Matching

In this section, we introduce the complete implementation of our common-sense subgraph matching method, namely *GpSense*, on large-scale common-sense KBs using GPUs. In order to support common-sense querying and reasoning, optimization techniques are also applied to GpSense.

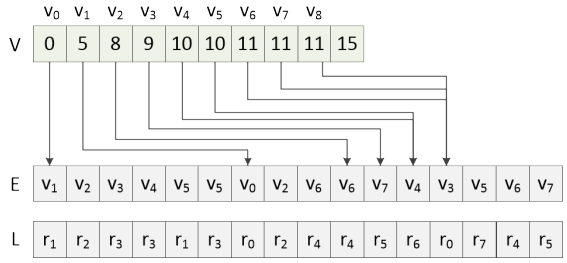


Fig. 4. Graph representation of the data graph in Figure 2

4.1 Graph Representation

In order to maintain and efficiently process the data graph $G(V, E)$ on GPUs, we use three array structures: an array whose size is identical to the size of V plus one, termed *nodes array*, and another array consisting of adjacency lists of all nodes in V, termed *edges array*. The nodes array has pointers to the adjacency lists of the nodes in the edges array. The additional, the last element of the nodes array indicates the length of the edges array. The last array with the size of $|E|$ stores the labels of all edges in the data graph. Figure 4 shows the representation of the graph illustrated in Figure 2 in the GPU memory.

The advantage of the data structure is that nodes in the adjacency list of a node are stored next to each other in the GPU memory. During GPU execution, consecutive threads can access consecutive elements in the memory. Therefore, we can avoid the random access problem and decrease the accessing time for GPU-based methods consequently.

4.2 GPU Implementation

In this subsection, we describe the implementation of parallel functions such as *collect_node_candidates*, and *filter_neighbor_candidates* in detail. These functions are based on two optimization techniques: occupancy maximization to hide memory access latency and warp-based execution to take advantage of the coalesced access and to deal with workload imbalance between threads within a warp.

collect_node_candidates: The purpose of this device function is to collect the candidates of a query node u in the *boolean* array $c\_set[u]$ to a candidate array $c\_array[u]$. The output of this function will maximize the GPU occupancy (i.e., maximize the number of running threads) for the next procedures. GpSense

executes the task by adopting a stream compaction algorithm [11] to gather elements with the *true* values in $c\_set[u]$ to the output array $c\_array[u]$. The algorithm employs prefix scan to calculate the output addresses and to support writing the results in parallel. The example of collecting candidate nodes of $?c$ is depicted in Figure 5. By taking advantage of c_array, candidate nodes v_1, v_2, v_3, v_4, v_5 can easily be mapped to consecutive active threads. As a result, our method achieves a high occupancy.

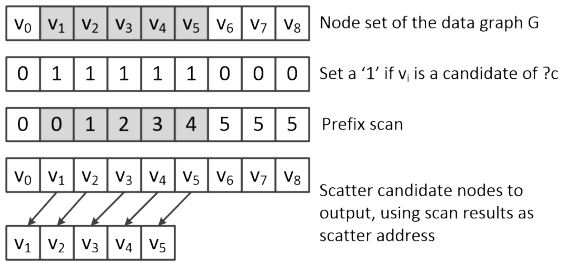


Fig. 5. Collect candidate nodes of $?c$

filter_neighbor_candidates: GpSense follows the adjacent edges of u to filter the candidates of query nodes connected to u. The step might suffer from warp divergence because of the diverse sizes of adjacency lists of u's candidates. To overcome the problem, we employ a *coarse-grained* and *warp-based* method inspired by Hong et al. [13]. In this approach, an entire warp is responsible for the adjacency list of a candidate. Figure 6 shows an example of filtering candidate nodes of $?a$ based on the candidate set of $?c$, $C(?c) = \{v_1, v_2, v_3, v_3, v_5\}$. Each candidate of $?c$ is mapped to a warp to filter the candidates of its adjacency node $?a$.

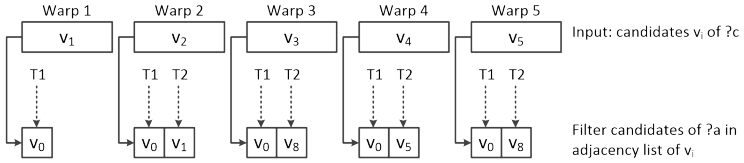


Fig. 6. Filter candidates of $?a$ based on candidate set of $?c$

4.3 Optimization Techniques

In this subsection, we introduce optimizations that we apply to enhance the efficiency of the subgraph matching problem in common-sense querying and reasoning.

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 labeled. They are 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.

Using the above observations, we make a modification for generating 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 maximum number of nodes which is not in the order among the remain 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 Common-sense graph in Figure 4 is an example of outgoing graph representation. Given a query graph in Figure 3, assume that we only use an outgoing graph as the data graph. Based on the above query plan generator, node v_0 is the first node in the order. After that we filter the candidates of $?c$ based on v_0. Since $?c$ 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 some issues in this approach: 1) We need to spend time to scan all the data graph nodes. 2) The number of candidates can be very large because the filtering condition is weak. To overcome the 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 $?c$. The number of candidates of $?a$, therefore, is much smaller than that in the previous approach. Consequently, GpSense can reduce a large amount of intermediate results during execution which is one of the most crucial issues for GPU applications.

5 Experiment Results

We evaluate the performance of GpSense in comparison with state-of-the-art subgraph matching algorithms, including VF2 [8], QuickSI (QSI) [22], GraphQL (GQL) [12] and TurboISO [9]. The experiments are conducted on SenticNet and its extensions [20, 21]. The query graphs are extracted from the data graph by picking a node in SenticNet and following BFS fashion to achieve other nodes. We choose nodes in the dense area of SenticNet to ensure that 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 CUDA Toolkit 6.0 running on the 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.

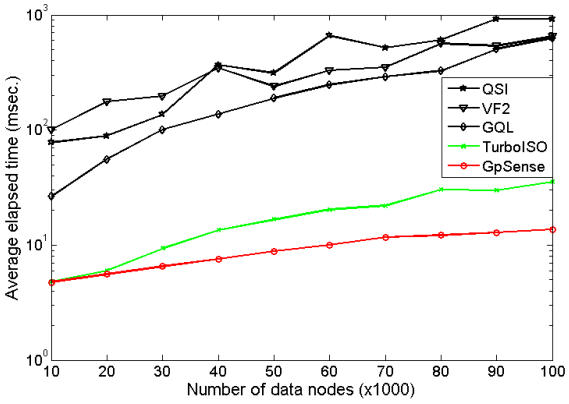
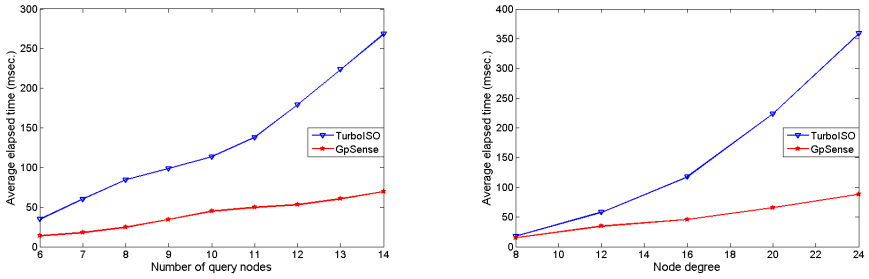


Fig. 7. Comparison with state-of-the-art methods

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 vertices which is primarily used for sentiment analysis. In this experiment, we extract subsets of SenticNet with the size varying from 10,000 nodes to 100,000 nodes. All the data graphs can fit into GPU memory. The query graphs contain 6 nodes.

Figure 7 shows that GpSense clearly outperforms VF2, QuickSI and GraphQL. Compared to TurboISO, our GPU-based algorithm obtains the 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 8a shows the performance results of GpSense and TurboISO on the query graphs whose numbers of nodes vary from 6 to 14. Figure 8b shows their performance results when the node degree increases from 8 to 24, where the number of query nodes is fixed to 10. As shown in the two figures, the performance of TurboISO drops significantly while that of GpSense does not.



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

Fig. 8. Comparison with TurboISO

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

6 Conclusion

In this paper, we introduced an efficient GPU-friendly method for answering subgraph matching queries over large-scale common-sense KBs. Our method, GpSense, is based on a *filtering-and-joining* approach which is suitable to be executed on massively parallel architecture of GPUs. Along with efficient GPU techniques of coalescence, warp-based and shared memory utilization, GpSense provides a series of optimization techniques which contribute to enhance the performance of subgraph matching-based common-sense reasoning tasks. Experiment results show that our method outperforms previous backtracking-based algorithms on CPUs and can efficiently answer subgraph matching queries on large-scale common-sense KBs.

References

1. M. Brocheler, A. Pugliese, and V. S. Subrahmanian. Cosi: Cloud oriented subgraph identification in massive social networks. In *International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 248–255. IEEE, 2010.
2. E. Cambria and A. Hussain. *Sentic computing: a common-sense-based framework for concept-level sentiment analysis*, volume 1. Springer, 2015.
3. 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. LNCS. Springer, 2009.
4. E. Cambria, D. Olsher, and D. Rajagopal. SenticNet 3: a common and commonsense knowledge base for cognition-driven sentiment analysis. In *Twenty-eighth AAAI conference on artificial intelligence*, pages 1515–1521, 2014.
5. E. Cambria, D. Rajagopal, K. Kwok, and J. Sepulveda. GECKA: game engine for commonsense knowledge acquisition. In *The Twenty-Eighth International Flairs Conference*, pages 282–287, 2015.
6. E. Cambria, H. Wang, and B. White. Guest editorial: Big social data analysis. *Knowledge-Based Systems*, 69:1–2, 2014.
7. 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.
8. 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.
9. 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.

10. P. Harish and P. Narayanan. *Accelerating large graph algorithms on the GPU using CUDA*, pages 197–208. Springer, 2007.
11. M. Harris, S. Sengupta, and J. D. Owens. *GPU Gems 3 - Parallel prefix sum (scan) with CUDA*, chapter 39. NVIDIA Corporation, 2007.
12. 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.
13. 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.
14. 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.
15. 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.
16. D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.
17. E. T. Mueller. *Commonsense Reasoning: An Event Calculus Based Approach*. Morgan Kaufmann, 2014.
18. 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.
19. S. Poria, A. Gelbukh, B. Agarwal, E. Cambria, and N. Howard. *Common sense knowledge based personality recognition from text*, pages 484–496. LNCS. Springer, 2013.
20. S. Poria, A. Gelbukh, E. Cambria, D. Das, and S. Bandyopadhyay. Enriching SenticNet polarity scores through semi-supervised fuzzy clustering. In *IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 709–716, 2012.
21. 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 *IEEE International Conference on Signal Processing (ICSP)*, volume 2, pages 1251–1255. IEEE, 2012.
22. 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.
23. 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.
24. J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.
25. 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.
26. 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.
27. 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.
28. P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.