Serdica J. Computing **2** (2008), 369–402

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

CONCEPTUAL INFORMATION COMPRESSION AND EFFICIENT PATTERN SEARCH

Galia Angelova, Stoyan Mihov

Abstract. This paper introduces an encoding of knowledge representation statements as regular languages and proposes a two-phase approach to processing of explicitly declared conceptual information. The idea is presented for the simple conceptual graphs where conceptual pattern search is implemented by the so called projection operation. Projection calculations are organised into off-line preprocessing and run-time computations. This enables fast run-time treatment of NP-complete problems, given that the intermediate results of the off-line phase are kept in suitable data structures. The experiments with randomly-generated, middle-size knowledge bases support the claim that the suggested approach radically improves the run-time conceptual pattern search.

1. Introduction. Knowledge representation arose in Artificial Intelligence (AI) in the late 1970s as a discipline studying formalisms for explicit

ACM Computing Classification System (1998): E.4, F.1.1, G.2.3, G.4, I.2.4.
Key words: data compaction and compression, finite automata, applications, efficiency, semantic networks.

declarative encoding of world semantics. Following the classical AI paradigm, most computations are still performed in run-time, which limits the efficiency of the emerging semantic technologies. On the other hand computational linguistics provides fast text analysis recently. The commercial systems analyse input text with run-time speed of more than 100000 words per second. This efficiency is due to a two-phase approach for (*i*) special off-line preprocessing of the necessary linguistic resources and (*ii*) run-time search within certain carefully prepared data structures. Very large morphological lexicons are encoded off-line as minimal acyclic Finite-State Automata (FSA) [1] and the run-time text analysis is turned to automaton look-up in linear time, depending on the length of the input wordform only. There are algorithms for direct building of the minimal acyclic FSA, by incremental construction of the FSA given a lexicographically-sorted list of all words in the respective finite regular language [2, 3].

In this paper we propose to implement conceptual search using FSA in a two-phase approach: (*i*) off-line computations, which deal with the (relatively) static knowledge base statements and (*ii*) run-time conceptual search, given user requests, by FSA look-ups. Actually the suggestion is to interpret logical formulas as words in certain regular language; thus we extend the computational linguistics' ideas to the positive, existentially-quantified conjunctive formulas that can be considered as knowledge representation statements in a formally-defined closed world. This proposal has been presented informally in [4, 5]. Here we focus on the precise mathematical definitions, the off-line preprocessing algorithm, the complexity issues and the experiments with two test data sets.

The article is structured as follows. Section 2 presents important notions and recalls basic facts. Section 3 introduces an FSA-based encoding of simple conceptual graphs with binary conceptual relations and shows that all their injective generalisations can be encoded off-line as a minimal FSA. Section 4 presents an algorithm for performing injective projection in run-time. Section 5 considers the complexity of the main algorithms. Section 6 discusses proof-of-concept experiments with two randomly generated, middle-size knowledge bases, which test the feasibility of the approach. Section 7 contains some discussion and the conclusion.

2. Basic Notions.

2.1. Conceptual Graphs' Support. Conceptual Graphs (CGs) are a kind of semantic networks which are founded both on logic and graph theory [6]. They consist of *concept types* (denoting the entities, attributes, states and events in the world) and *relation types* (encoding the n-ary conceptual relations which show how the concept types' instances are interconnected). As ordinary graphs,

CGs are bipartite graphs where the concepts are drawn as rectangles and the conceptual relations as ellipses. The CG graphical structure visualises the identity of the variables, constants and predicates in the corresponding logical statements, which encode explicitly world knowledge. CGs do not exist as isolated declarative statements; there is a context fixing the background ontological framework where the particular semantic assertions hold – so called support [7].

Definition 1. *A **support** is a 4-tuple $S = (T_C, T_R, I, \tau)$ where:*

- *T_C is finite, partially ordered set of distinct concept types. The partial order defines the hierarchy of concept types: for x, $y \in T_C$, $x \leq y$ means that x is a subtype of y. Then x is a specialisation of y and y is a generalisation of x; y subsumes x. The universal type $\top$ (top) subsumes all types in T_C. All types in T_C subsume the absurd type $\perp$ (bottom);*
- *T_R is finite, partially ordered set of distinct relation types. The partial order defines the hierarchy of relation types. $T_C \cap T_R = \emptyset$. Each $R \in T_R$ has arity 2 and holds between two different instances of concept types x, $y \in T_C$ or a concept type $x \in T_C$. A pair $(c1_{\max R}, c2_{\max R}) \in T_C \times T_C$ is associated to each relation type $R \in T_R$; it defines the greatest concept types that might be linked by the relation type R. R holds between instances of the concept types x, $y \in T_C$ if $x \leq c1_{\max R}$ and $y \leq c2_{\max R}$. All pairs $(c1_{\max R}, c2_{\max R})$ are called star graphs or basis of the support. If $R_1, R_2 \in T_R$ and $R_1 \leq R_2$, then $c1_{\max R1} \leq c1_{\max R2}$ and $c2_{\max R1} \leq c2_{\max R2}$. The lattice of relation types also has the universal type $\top$ as top node and the absurd type $\perp$ as the bottom node;*
- *I is a set of distinct individual markers which refer to specified concept instances. $T_C \cap I = \emptyset$ and $T_R \cap I = \emptyset$. The generic marker $*$, where $* \notin (T_C \cup T_R \cup I)$, refers to an unspecified individual instance of some specified concept type x. Thus concepts have instances, in contrast to relations. The members of I are not ordered but $i \leq *$ for all $i \in I$;*
- *τ is a mapping from I to T_C and associates individual instances to concept types. If $\tau(i) = x_1, x_2, \ldots, x_n$ for $i \in I$ and x_1, x_2, $\ldots$, $x_n \in T_C$, then there is a lower concept type to which i belongs (e.g. x_1) and $x_1 \leq x_j$ for $2 \leq j \leq n$. In other words, each individual belongs to some type and its supertypes, within one hierarchy branch. Thus τ defines the conformity of individuals to concept types. If $\tau(i) = x$ for $i \in I$ and $x \in T_C$ then i can be generalised by $x : *$ which denotes an unspecified individual instance of x.*

Definition 2. *A* ***simple conceptual graph*** *(SCG) G, defined over a support S, is a connected, finite bipartite graph* $(V = V_C \cup V_R, U, \lambda)$ *where:*

- *The nodes V are defined by* V_C *– the set of concept nodes or c-nodes and* V_R *– the set of relation nodes or r-nodes.* $V_C \neq \emptyset$*, i.e. each SCG contains at least one c-node. For* $x \in V_C$*, type(x) denotes the label of* $x \in T_C$*;*
- *The edges U are defined by a set of ordered pairs* (x, r) *or* (r, y)*, where x,* $y \in V_C$ *and* $r \in V_R$*. Thus the edges are directed either from a c-node to a r-node – like* (x, r)*, or from a r-node to a c-node – like (r,y). The edges* (x, r) *are called* ***incoming arcs*** *to the r-node r while the edges* (r, y) *are called* ***outgoing arcs*** *from the r-node r. For every r-node* $r \in V_R$*, there is one incoming and one outgoing arcs, incident with r;*
- *The mapping* λ *defines correspondences between the elements of S and the nodes of G. It associates labels to the elements of* $V_C \cup V_R$*. Each c-node* $c \in V_C$ *is labeled by a pair* $(C, \text{marker}(C))$*, where* $C \in T_C$ *and* $\text{marker}(C) \in I \cup \{*\}$*. A c-node with generic marker is called a* ***generic node****, it refers to an unspecified individual of the specified concept type. A c-node with individual marker is called an* ***individual node****, it refers to a specified instance of the concept type. Each r-node* $r \in V_R$ *is labeled by a relation type* $R \in T_R$*. The first argument of R is mapped to the c-node linked to the incoming arc of r while its second argument is mapped to the c-node linked to the outgoing arc of r.*

Example 1. Figure 1 introduces a sample support:

- T_C = {STATE, EVENT, ENTITY, ACT, ANIMATE, PHYS-OBJECT, LOVE, EAT, ANIMAL, PIE, PERSON} with partial order shown at Fig. 1;
- T_R = {AGNT, OBJ, EXPR, POSS, PTNT} with partial order and star graphs shown at Figure 1. These labels stand for AGeNT, OBJect, EXPeRiencer, POSSesor and PaTieNT respectively. Since PTNT≤OBJ, the arguments of PTNT in the corresponding star graph are specialisations of the respective arguments of OBJ, i.e. ACT≤ACT and PHYS-OBJECT≤ENTITY;
- I = {John, Sue} which are not ordered;
- τ(John) = PERSON, τ(Sue) = PERSON.

Figure 2 presents a Knowledge Base (KB) of two simple conceptual graphs G_1 and G_2 defined over the support in Figure 1. In natural language they mean: G_1 *'John eats Sue's pie'* and G_2 *'There exists a person who loves himself/herself and loves Sue who eats his/her pie'.* By default the generic markers are omitted, e.g. the concept types (EAT,$*$) in G_1 is labeled by EAT. The individual nodes are

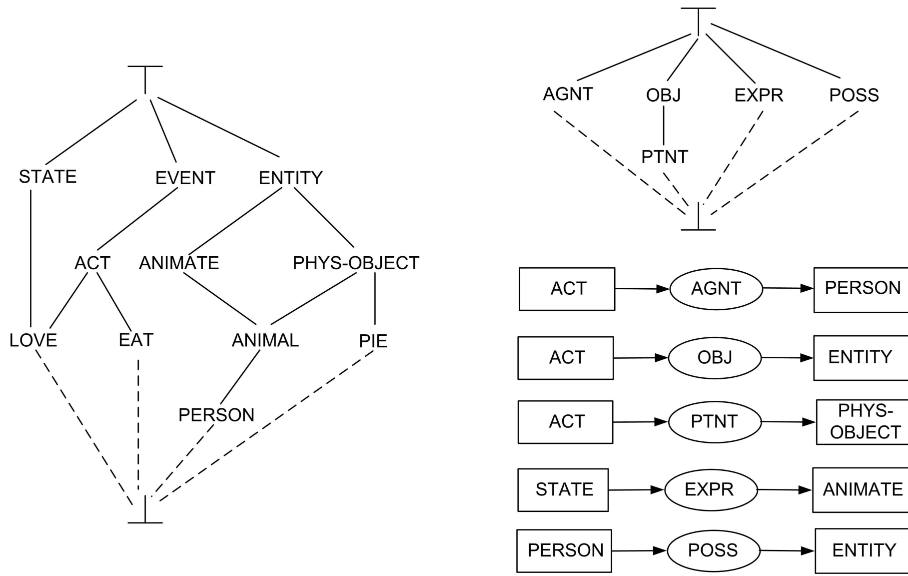


Fig. 1. Partial order of concept and relation types and star graphs for the relation types

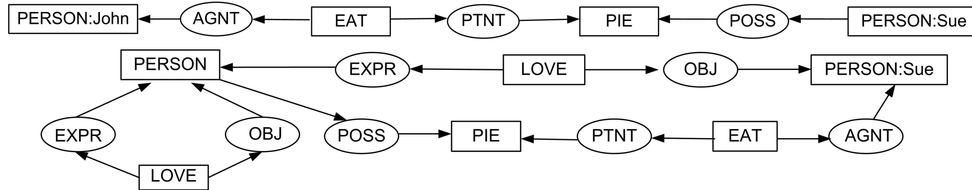


Fig. 2. Graphical representation of the sample SCGs G_1 and G_2

denoted as *TYPE:individual*, e.g. the c-node PERSON:John in G_1. The graph G_2 is called *cyclic* since its underlying ordinary graph is a cyclic one.

According to the definitions in [6, 7], SCGs are multi-graphs and have multi-edges – i.e, several edges between a r-node and one of its c-neighbours. This may happen for the general case of n-ary conceptual relations, when the edges between the r-nodes and their c-neighbours are labeled, so multi-edges may appear as shown at Fig. 3A. But we consider binary conceptual relations only, with one incoming and one outgoing arc; in this case multi-edges are impossible but loops might appear if a conceptual relation holds from/to one instance of a given c-node as shown at Fig. 3B. However, definition 1 introduces supports over relation types $R \in T_R$ which hold either between instances of two distinct concept types, or between two distinct instances of one concept type. Therefore,

we consider SCGs which are equivalent to directed bipartite graphs with at most one edge between each two vertices. Such graphs may contain cycles, like G_2 in Figure 2, but they contain no multi-edges and no loops.

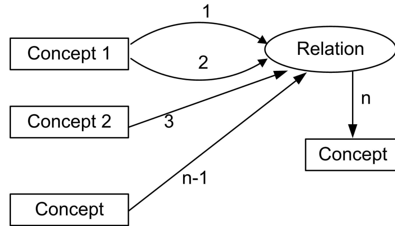


Fig. 3A. Multi-edges numbered 1 and 2 for some n-ary conceptual relation

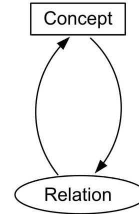


Fig. 3B. Loops for binary conceptual relations

2.2. Logical Interpretation of Simple Conceptual Graphs.

Definition 3 ([6]). *We define the **formula operator** ϕ which translates a SCG into a formula in the first-order predicate calculus. If G is a SCG, let ϕG be a formula constructed as follows:*

- *If G contains k generic c-nodes, assign a distinct variable x_1, x_2, ..., x_k to each one;*
- *For each c-node c of G, let* identifier(c) *be the variable assigned to c if c is a generic node or* marker(c) *if c is an individual node;*
- *Represent each c-node c of G as a monadic predicate whose name is the same as* type(c) *and whose argument is* identifier(c)*;*
- *Represent each r-node r of G as a binary predicate whose name is the same as* type(r)*. For each i, $i = 1, 2$, let the i-th argument of the predicate be the identifier of the c-node linked to the i-th arc of r;*
- *Build ϕG as a concatenation of a quantifier prefix $\exists x_1 \exists x_2 \dots \exists x_k$ to a body consisting of the conjunction of all the predicates for the c-nodes and r-nodes of G.*

It is shown in [8] that SCGs are equivalent to the positive, conjunctive and existential fragment of first order logic without functions. As seen in definition 3, the logical interpretation of a SCG with binary conceptual relations contains binary predicates, one per each binary relation r. We shall call these binary predicates *elementary conjuncts*. In the next sections, we shall record them as triple of labels c_1 r c_2 where the concepts c_1, c_2 are the first and second arguments of the relation r.

Example 2. We present the logical formula corresponding to the SCG G_2 in Example 1:

'*There exists a person who loves himself/herself and loves Sue who eats his/her pie*':

$\exists x \exists y \exists z \exists u \exists v$ PERSON(x) & LOVE(y) & LOVE(z) & PIE(u) & EAT(v) &
& PERSON(Sue) & expr(x,y) & obj(y,x) & expr(x,z) &
& obj(z, PERSON(Sue)) & poss(x,u) & ptnt(v,u) & agnt(v, PERSON(Sue))

There are seven elementary conjuncts in the logical interpretation of G_2, corresponding to the seven binary conceptual relations in the underlying bipartite graph.

Another specific issue about CGs is that V_C and V_R may contain nodes with duplicating labels, because the mapping λ in definition 2 may associate repeating labels to the elements of $V_C \cup V_R$. For instance, Figure 4 contains a SCG which (roughly) means: '*There exist an arc, which has as part a brick and (another) brick*'. When building the logical formula according to definition 3, the generic c-nodes BRICK are juxtaposed distinct variables that are existentially quantified, i.e. these nodes represent different unspecified instances of the concept type BRICK. The r-nodes PART are duplicated too, because the conceptual relations hold between different instances of the duplicating c-nodes.

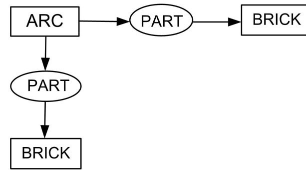


Fig. 4. Duplicating labels of concept and relation nodes (example from [6])

2.3. Projection.

Definition 4 ([6]). *A* ***projection*** π *is a graph morphism defined as follows. Let G and H be two SCG with binary conceptual relations. Then* π: $G \rightarrow H$ *is a graph* πG, *where* $\pi G \subseteq H$ *and:*

- *For each concept c in G,* πc *is a concept in* πG *where* $\mathrm{type}(\pi c) \leq \mathrm{type}(c)$. *If c is individual concept, then* $c = \pi c$.
- *For each conceptual relation* $r(c_1, c_2)$ *in G,* πr *is a conceptual relation in* πG *where* $\mathrm{type}(\pi r) \leq \mathrm{type}(r)$ *and* πr *holds between* πc_1 *and* πc_2, *i.e.* $\pi r(\pi c_1, \pi c_2)$ *is in* πG.

Definition 5 ([9])**.** *Let G and H be two SCGs. An* ***injective projection*** *π is a kind of projection such that $\pi : G \rightarrow H$ is a graph πG, where πG is isomorphic to G.*

Example 3. The projection of a query G onto a KB graph H finds G-compliant conceptual patterns in H. The injective projection supports the semantic extraction when processing world knowledge. Obviously, there can be different projections of G to H. The query G at Figure 5 has empty projection onto G_1 but several non-empty projections onto G_2.

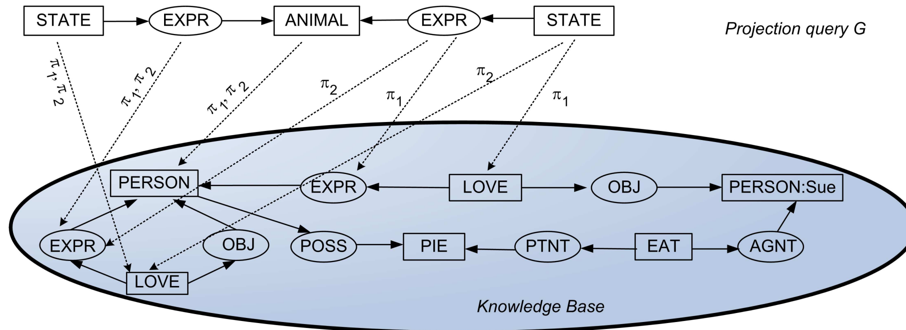
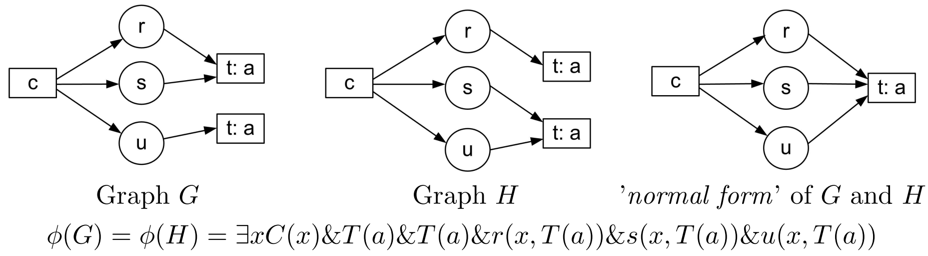


Fig. 5. Injective projection π_1 and projection π_2 from a query graph G onto G_2. For π_1, the two elementary conjuncts of G are mapped onto two distinct elementary conjuncts of G_2. For π_2, both elementary conjuncts of G are mapped onto one elementary conjunct of G_2

The projection mappings depend on the graph nodes but CGs can contain 'redundant' nodes. A specialisation rule *simplify* is defined in [6], to delete duplicating r-nodes between the same concept instances (these r-nodes yield duplicating predicates in the logical interpretation and can be deleted, since $X\&X = X$). In addition, SCGs can contain unnecessary duplication of individual c-nodes, like the SCGs G and H at Figure 6. These graphs have identical logical interpretation, they both project to their '*normal form*' but G does not project onto H and vice versa. Duplicated individual c-nodes can be merged in linear time by a trivial algorithm [8].

Definition 6. *A SCG G is in* ***normal form*** *when it contains no duplicating r-nodes holding between the same concept instances and no duplicating c-nodes with the same individual markers.*

Sowa shows in [6] that given two SCGs H and G where H is specialisation of G (i.e. $H \leq G$), there is a projection from G to H. Chein and Mugnier prove the reciprocal property in [7]. In this way the ground operations over SCGs are



$$\phi(G) = \phi(H) = \exists x C(x) \& T(a) \& T(a) \& r(x, T(a)) \& s(x, T(a)) \& u(x, T(a))$$

Fig. 6. Two SCGs G and H which are incomparable by injective projection [8]

either based on projection, as a particular graph morphism, or on specialisation rules and their reverse generalisation rules.

The calculation of CG projection challenges the researchers for more than twenty years. Given a query graph, its mappings to the KB facts are computed in run-time graph by graph. Counting all projections is an interesting computational problem too. The algorithms for computing projection are either based on first order logic and Prolog inference mechanisms or rely on graph theory. Given two SCGs G and H, it is NP-complete to decide whether $H \leq G$. However there are large classes of SCGs for which polynomial algorithms for projection exist when the underlying ordinary graphs are trees [9, 10]. Projection can be also computed by translating it to the maximum clique problem, to improve the performance for specific SCG types [11]. Thus the most efficient projection calculations are based on algorithms solving equivalent problems in graph theory. These complexity results hold for the case of run-time projection calculation.

2.4. Finite State Automata.

Definition 7. *A* ***finite state automaton*** *(FSA) A is a 5-tuple $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$, where Σ is a finite* alphabet*, Q is a finite set of* states*, $q_0 \in Q$ is the* initial state*, $F \subseteq Q$ is the set of* final states*, and $\Delta \subseteq Q \times \Sigma \times Q$ is the* transition relation*. The transition $\langle q, a, p \rangle \in \Delta$* begins *at state q,* ends *at state p and has the* label *a. A* ***deterministic finite state automaton*** *is a FSA $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$, where Δ is a function $\Delta : Q \times \Sigma \rightarrow Q$.*

Definition 8. *Let $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$ be a FSA. A* ***path*** *c* ***in*** *A is a finite sequence of $k > 0$ transitions: $c = \langle q_1, a_1, q_2 \rangle \langle q_2, a_2, q_3 \rangle \cdots \langle q_k, a_k, q_{k+1} \rangle$, where $\langle q_i, a_i, q_{i+1} \rangle \in \Delta$ for $i = 1, \ldots, k$.*

The integer k is called the length *of c. The state q_1 is called* beginning *of c and q_{k+1} is called the* end *of c. The string $w = a_1 a_2 \cdots a_k$ is called the* label *of c. The* null path *of $q \in Q$ is 0_q, beginning and ending in q with label ε, where ε is the empty symbol. A* successful path *starts at q_0 and ends at a final state.*

Definition 9. *Let $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$ be a FSA. Let Σ^* be the set of all strings over the alphabet Σ, including the empty symbol ε. The* ***generalised transition relation*** *Δ^* is the smallest subset of $Q \times \Sigma^* \times Q$ with the following closure properties:*

- *For all $q \in Q$ we have $\langle q, \varepsilon, q \rangle \in \Delta^*$;*
- *For all $q_1, q_2, q_3 \in Q$ and $w \in \Sigma^*$, $a \in \Sigma$: if $\langle q_1, w, q_2 \rangle \in \Delta^*$ and $\langle q_2, a, q_3 \rangle \in \Delta$, then $\langle q_1, wa, q_3 \rangle \in \Delta^*$.*

Definition 10. *The* ***formal language*** *$L(A)$ accepted by a FSA $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$ is the set of all strings, which are labels of paths leading from the initial to a final state:*

$$L(A) := \{w \in \Sigma^* \mid \exists\ q \in F : \langle q_0, w, q \rangle \in \Delta^*\}.$$

These strings will be called ***words*** *of the language $L(A)$.*

Languages accepted by FSA are **regular** languages. Every finite list of words over a finite alphabet of symbols is a regular language. Given a finite regular language L, there are algorithms which construct a deterministic FSA which accepts this language.

The Myhill-Nerode theorem (see [12]) states that among the deterministic automata that accept a given language L, there is a unique automaton (excluding isomorphisms) that has a minimal number of states. It is called the **minimal** deterministic automaton of the language L. As introduced in [13], the minimization algorithm for a deterministic FSA with m states has complexity $k \times m \times \log(m)$ where k is some constant which depends linearly on the size of the input alphabet. For comparison, the direct construction of the minimal acyclic automaton A in the case of finite regular languages is $O(n \log(m))$, where n is the total number of alphabet symbols in the input list of words and m is the number of the states in A [3].

Definition 11. *Let $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$ be a FSA. Let Σ^+ be the set of all strings w over the alphabet Σ, where $|w| \geq 1$. The automaton A is called* ***acyclic*** *iff for all $q \in Q$ there exist no string $w \in \Sigma^+$ such that $\langle q, w, q \rangle \in \Delta^*$.*

Definition 12. *A* ***deterministic finite state automaton with markers at the final states*** *A is a 7-tuple $A = \langle \Sigma, Q, q_0, F, \Delta, E, \mu \rangle$, where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, $\Delta : Q \times \Sigma \rightarrow Q$ is the transition function, E is a finite set of markers, and μ: $F \rightarrow E$ is a function assigning a marker to each final state.*

We notice that a deterministic FSA with markers at the final states juxtaposes markers to the accepted words. This property will be very helpful for the constructions that are presented in the next sections. Let us recall the following well-known facts:

Proposition 1. *A necessary and sufficient condition for any deterministic automaton (in which every path can be extended to a successful one) to be acyclic is that it recognises a finite set of words. Given a finite regular language L, there are algorithms for construction of an acyclic deterministic FSA, which recognises L.*

Proposition 2. *Let $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$ be a deterministic FSA and $w = a_1 \cdots a_n$ is a word consisting of Σ symbols. The time complexity of a look-up in A with w is $O(n)$.*

3. Off-line encoding of SCGs as finite state automata. We shall deal with the set of all normalised SCGs with binary conceptual relations in certain momentary status of the knowledge base, which is defined according to support S.

3.1. Linear encoding of SCGs as strings of support symbols. Definition 3 shows that a SCG's logical formula contains binary predicates $\mathit{rel}(c_1,c_2)$, where rel is a r-node label and c_1,c_2 are either existentially-quantified variables for the SCG generic c-nodes, or individual markers for the individual c-nodes. We notice that the usage of specific indices at the off-line phase will prevent the run-time recognition of labels, when arbitrary queries will be posed to the system. To avoid any graph-specific indexation in the off-line encoding, we can replace the variables by their respective c-nodes' labels. We can also replace the individual markers by the string *type*:*marker*, where *type* is the label of the corresponding c-node in T_C (this was already done at Fig. 2 where PERSON:Sue denotes an individual c-node of type PERSON). Thus we can encode the monadic predicates of the logical formula within the binary ones. Then the binary predicates

$$\mathit{rel}_1(\mathit{concept}_{11},\mathit{concept}_{12}) \ \& \ \ldots . \ \& \ \mathit{rel}_n(\mathit{concept}_{n1},\mathit{concept}_{n2})$$

can be linearised as a string of triples - support symbols:

$$\mathit{concept}_{11}\ \mathit{rel}_1\ \mathit{concept}_{12}\ \mathit{concept}_{21}\ \mathit{rel}_2\ \mathit{concept}_{22}\ \ldots \mathit{concept}_{n1}\ \mathit{rel}_n\ \mathit{concept}_{n2}$$

where $\mathsf{concept}_{ij}$, $1 \leq i \leq n$, $j = 1, 2$ are either concept type labels or strings *type*:*marker*. This sequence of labels, however, fails to capture the topological structure of the SCGs.

Example 4. To study the graph structures, let us consider Figure 7. It contains distinct configurations of connected elementary conjuncts with duplicating c-nodes:

- *There exists love felt by a person and directed to another person*, Fig. 7A,
- *There exists a person who loves himself/herself*, Fig. 7B,
- *There exists a person who feels love and is object of (another) love*, Fig. 7C.

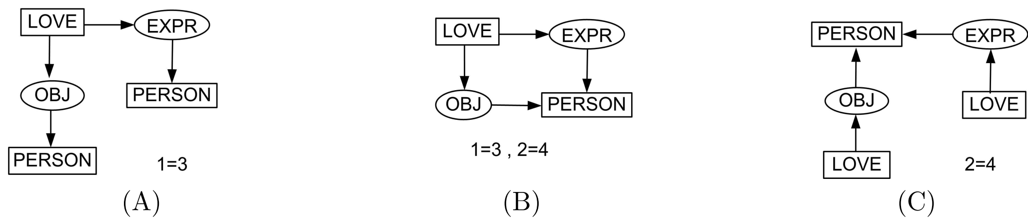


Fig. 7. Two connected conjuncts expr(LOVE, PERSON) & obj(LOVE, PERSON): the same type labels express three different topological structures of concept instances

Obviously, the sequence of conjunct labels triple by triple

LOVE EXPR PERSON LOVE OBJ PERSON

fails to encode the identity of the relation arguments. However, we can invent an enumeration of the positions of the equivalent arguments, thus encoding the information about the classes of arguments' equivalence. Then the indices of the identical arguments can be associated to the linear record as *annotations*. We consider the structural configurations at Figure 7. There are 4 argument positions in the linear record of two connected elementary conjuncts which can be numbered by 1, 2, 3 and 4. Then the three graphs can be encoded as follows:

	Position 1		*Position 2*	*Position 3*		*Position 4*	*Annotation*
Fig.7A:	LOVE	EXPR	PERSON	LOVE	OBJ	PERSON	1=3
Fig.7B:	LOVE	EXPR	PERSON	LOVE	OBJ	PERSON	1=3, 2=4
Fig.7C:	LOVE	EXPR	PERSON	LOVE	OBJ	PERSON	2=4

The associated annotation, which explicates the sets of identical arguments, helps to uniquely distinguish the three graphs at Figure 7A, 7B and 7C.

Definition 13. *Let $S = (T_C, T_R, I, \tau)$ be a KB support according to Definition 1. Let us define an alphabet of support symbols $\Sigma = \{x \mid x \in T_C \text{ or } x \in T_R\} \cup \{x : i \mid x \in T_C, i \in I \text{ and } \tau(i) = x\}$. Let us order the m symbols of Σ using certain (linear) order $\Omega = \langle a_1, a_2, \ldots, a_m \rangle$.*

Let G be a normalised SCG defined according to S, with n conceptual relations which have common arguments since G is connected. Let G have k individual c-nodes and p distinct generic c-nodes, i.e. G has $k+p$ c-nodes which occupy the positions of $2 \times n$ relation arguments. A linearised representation of G is constructed as follows:

- *Let $f(G)$ be a logical formula, juxtaposed to G according to definition 3:*
  $\exists x_1 \exists x_2 \ldots \exists x_p$
  $type_1(x_1)\& \ldots \& type_p(x_p)\&\ type_{p+1}(marker_1)\& \ldots \&\ type_{p+k}(marker_k)\&$
  $rel_1(concept_{11}, concept_{12})\ \&\ \ldots \&\ \ rel_n(concept_{n1}, concept_{n2})$

  where $type_i \in T_C$ for $1 \le i \le p+k$, $rel_i \in T_R$ for $1 \le i \le n$, $marker_i \in I$ for $1 \le i \le k$, and $concept_{ij}$ for $1 \le i \le n$, $j = 1, 2$ is either one of the variables $x_1, \ldots, x_p$ or one of the individuals $type_{p+1}(marker_1), \ldots, type_{p+k}(marker_k)$;

- *For all arguments $concept_{ij}$, $1 \le i \le n$, $j = 1, 2$ which are equal to a variable x_u where $1 \le u \le p$, replace the argument $concept_{ij}$ by the string $type_u{:}x_u$ where $type_u(x_u)$ is a monadic predicate in $f(G)$;*

- *For all arguments $concept_{ij}$, $1 \le i \le n$, $j = 1, 2$ which are equal to $type_{p+u}(marker_u)$ where $1 \le u \le k$, replace the argument $concept_{ij}$ by the string $type_{p+u}{:}marker_u$ where $type_{p+u}(marker_u)$ is a monadic predicate in $f(G)$;*

- *Take the binary predicates in $f(G)$:*
  $$rel_1(concept_{11}, concept_{12})\ \&\ \ldots \&\ rel_n(concept_{n1}, concept_{n2})$$
  where $concept_{ij}$ for $1 \le i \le n$, $j = 1, 2$ is either one of the generic c-nodes $type_1{:}x_1$, $type_2{:}x_2$, ..., $type_p{:}x_p$, or one of the individual c-nodes $type_{p+1}{:}marker_1$, ..., $type_{p+k}{:}marker_k$.
  Represent them as a string seq(G) of n triples with $3 \times n$ positions:
  $$concept_{11}\ rel_1\ concept_{12}\ \ldots\ldots\ldots\ concept_{n1}\ rel_n\ concept_{n2}.$$
  Disregard all substrings '$:x_1$', '$:x_2$', ..., '$:x_p$' in '$type_1{:}x_1$', ..., '$type_p{:}x_p$' respectively, thus pretending that seq(G) consists of Σ symbols only, and sort seq(G) triple by triple according to Ω. When rearranging the labels '$type_1$',

$'type_2'$, …, $'type_p'$ in the sorting process, move together with them the associated substrings $':x_1'$, …, $':x_p'$. If two neighbouring triples consist of identical symbols of Σ, order them in increasing order according to the indices of the variables x_1, x_2,…,x_p which are present as substrings in the arguments's labels $'type_1{:}x_1'$, $'type_2{:}x_2'$, …, $'type_p{:}x_p'$;

- *Let the sorted sequence of n triples be sortedSeq(G). Without loss of generality, it can be denoted by $concept_{11}$ rel_1 $concept_{12}$ … $concept_{n1}$ rel_n $concept_{n2}$. Consider the relation arguments and assign a position index $v = 2*(i-1)+j$ to each argument $concept_{ij}$ for $1 \le i \le n$, $j = 1, 2$. Then sortedSeq(G) can be represented as*

  $$argument_1 \ rel_1 \ argument_2 \ \ldots argument_{2n-1} \ rel_n \ argument_{2n}$$

  Build classes of indices as follows: for every set of q equal arguments $2 \le q \le 2 \times n$ where

  $$argument_{v1} = argument_{v2} = \cdots\cdots = argument_{vq}$$

  construct the string $V = 'v_1 = v_2 = \cdots = v_q'$, where $1 \le v_i \le 2 \times n$ for $i = 1, 2, \ldots, q$ and $v_1 < v_2 < \cdots < v_q$.

  *V will be called a **class of equivalent arguments** for sortedSeq(G).*

- *Let sortedSeq(G) have z distinct sets of equal arguments $\Re_1$, $\Re_2$, …, $\Re_z$. For every set $\Re_i$, let the class of equivalent arguments be V_i which is a string of digits and the symbol $'='$ for $1 \le i \le z$. Sort the list $[V_1, \ldots, V_z]$ in ASCII, rename the sorted list items as $[V_1, V_2, \ldots, V_z]$ and construct the string*

  $$annotation(G) = 'V_1,V_2,\ldots,V_z'$$

- *In sortedSeq(G) delete the substrings $':x_1'$, …, $':x_p'$ from the arguments $concept_{ij}$, $1 \le i \le n$, $j = 1, 2$ which have the format $'type_1{:}x_1'$, …, $'type_p{:}x_p'$. After this deletion, sortedSeq(G) contains only symbols of Σ, i.e. labels in $T_C \cup T_R \cup I$.*

- *The pair $\langle$sortedSeq(G), annotation$(G)\rangle$ will be called **linear record** of G with respect to $f(G)$ and Ω.*

Example 5. The linear records of the three SCGs at Fig. 7 are shown in Example 4. They are unique with respect to the order EXPR<LOVE<OBJ< PERSON since the SCGs' elementary conjuncts contain different relations and the sorting arranges the labels unambiguously. Thus the annotations of these 2-conjunct SCGs are unique. However, there are SCGs with multiple annotations

when the graphs contain multiple identical triples. Let us consider the subgraph of G_2 at Fig. 8 with two triples ′LOVE EXPR PERSON′ where LOVE denotes different generic instances. Obviously, different logical formulas can be built for this graph,

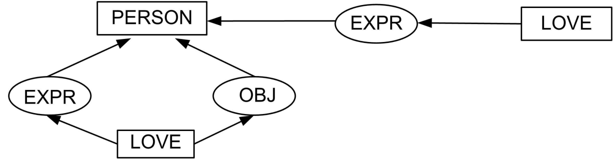


Fig. 8. A subgraph of G_2 with multiple equivalent logical formulas (due to isomorphism)

depending on the assignment of variables to generic concept instances. If x_1 is assigned to PERSON and x_2 and x_3 are assigned to the two generic instances of LOVE, this SCG will have two logical formulas due to the isomorphism between the variables x_2 and x_3:

$\exists x_1 \exists x_2 \exists x_3$ PERSON(x_1)&LOVE(x_2)&LOVE(x_3)&expr(x_2,x_1)&expr(x_3,x_1)&obj(x_2,x_1)

and

$\exists x_1 \exists x_2 \exists x_3$ PERSON(x_1)&LOVE(x_2)&LOVE(x_3)&expr(x_2,x_1)&expr(x_3,x_1)&obj(x_3,x_1).

Two annotation markers will be built using these formulas: ′1=5, 2=4=6′ and ′3=5, 2=4=6′ respectively. But we do not treat the isomorphism of the variables at the moment. Rather, we assume that each SCG G is encoded as a logical formula and the linear record is constructed using the formula's variables. After building the annotation, all variables are deleted.

We note that there are many linearised graph representations for a SCG G, in case that the string *sortedSeq* is not sorted. Their *annotation* enables to reconstruct the respective SCG.

Lemma 1. *Given a logical formula $f(G)$ of a normalised SCG G and an order Ω of its support symbols, there exists an unique linear record of G. Given a linear record of a normalised SCGs H, a logical formula $f(H)$ and a graphical representation of H can be constructed.*

P r o o f. Follows from the construction steps in Definition 13. Actually each SCG has a set of equivalent logical formulas (due to the isomorphisms among the variables). □

The annotations, which encode the topological structure of the SCGs, represent sets of identical relations' arguments. Describing all possible identities of n arguments is connected to the task of partitioning a set with n elements into nonempty, disjoint subsets. Each partition defines an equivalence relation for its members. The number of partitions is given by the so-called Bell numbers

B_1, B_2, ... [14]. In a previous paper we have considered the fifteen classes for partitioning of a set with 4 elements $\{x_1, y_1, x_2, y_2\}$. Only six classes can be interpreted as encodings of the topological links in a SCG with two binary predicates $relation_1(x_1, y_1)$ & $relation_2(x_2, y_2)$ [4]. Our experiments show that the structural variety in the SCGs is much more restricted than the corresponding Bell numbers for the respective number of set elements (B_{20} and B_{24}, see Section 6).

3.2. Off-line construction of a minimal FSA with markers at the final states. We propose to enumerate explicitly all possible injective projection queries with non-empty answers in a given closed world. Further we show how to compress this conceptual resource as a minimal FSA with markers at the final states. The following definition helps us to select and store only the subgraphs which have conceptual interpretation according to the support.

Definition 14. *Let G be a SCG with binary conceptual relations in a KB defined according to some support S. A* ***conceptual subgraph*** *of G is a connected graph* G_{cs} *such that:*

- *as ordinary graph,* $G_{cs} \subseteq G$ *and*
- G_{cs} *is a SCG defined according to the support S.*

Example 6. Figure 9A shows a conceptual subgraph of G_2. Figure 9B contains a 'meaningless' subset of G_2 nodes. Below by subgraphs of SCGs we shall mean conceptual subgraphs.

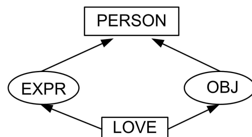


Fig. 9A. A conceptual subgraph of G_2.

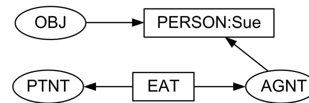


Fig. 9B. Connected nodes of G_2, which do not form a conceptual subgraph

We present an algorithm for construction of a minimal acyclic FSA with markers at the final states, which encodes all KB subgraphs and their injective generalisations. According to Proposition 1, every acyclic FSA can be defined by the finite list of words belonging to the automaton language. Given a KB, there is a finite number of KB subgraphs with a finite number of injective generalisations. Therefore we focus on the construction of the finite regular language which encodes all KB subgraphs and their injective generalisations. This language represents all possible queries that have non-empty injective projections onto the KB at the particular moment. We shall construct a finite list of words in the alphabet of all support symbols. These words will be lexicographically

sorted according to certain linear order. After the construction of the list, we shall apply methods of automata theory to build the minimal FSA.

We shall use the following types:

CHAR-types: *sortedSeq*, *annotation, new_lin_labels*;
LIST-types: *list_alternative_annot* – a list of strings consisting of $\{'1', '2', \ldots, '9', '0', '=', ','\}$;
Arrays of lists: *list_subgraphs, list_gen_graphs*
Arrays: *words_markers*(CHAR,⟨CHAR,CHAR,CHAR,CHAR⟩) and *sorted_words_markers*(CHAR, {⟨CHAR,CHAR,CHAR,CHAR⟩, ..., ⟨CHAR,CHAR,CHAR,CHAR⟩});

Algorithm 1. *Construction of a minimal acyclic FSA with markers at the final states* $A_{KB} = \langle \Sigma, Q, q_0, F, \Delta, E, \mu \rangle$ *which encodes all subgraphs' injective generalisations for a KB of normalised SCGs with binary conceptual relations* $\{G_1, G_2, \ldots, G_n\}$ *over support* S.

Step 1, defining the finite alphabet Σ*:* Let $S = (T_C, T_R, I, \tau)$ be the KB support according to definition 1. Define the alphabet $\Sigma = \{x \mid x \in T_C$ or $x \in T_R\} \cup \{x : i \mid x \in T_C, i \in I$ and $\tau(i) = x\}$. Order the m symbols of Σ using certain linear order $\Omega = \langle a_1, a_2, \ldots, a_m \rangle$. Then Σ is an ordered alphabet.

Step 2, indexing all c-nodes in the KB: Juxtapose distinct integer indices to all KB *c*-nodes, to ensure their default treatment as distinct instances of the generic concept types. Then the linear records of the KB graphs contain no multiple triples built by the same support labels. Define an alphabet for the support labels in the indexed SCGs:

$\Sigma_{KB} = \{a_{ij} \mid a_i \in \Sigma,\ 1 \leq i \leq m$ and j is an index assigned to a KB *c*-node with label $a_i, 1 \leq j \leq p_i$ or $j = 'none'$ when no indices are assigned to $a_i\}$.

Order the symbols of Σ_{KB} according to the linear order

$\Omega_{KB} = \langle a_{1s1}, \ldots, a_{1su}, a_{2p1}, \ldots, a_{2pv}, \ldots, a_{mq1}, \ldots, a_{mqx} \rangle$ where $s_1, s_2, \ldots, s_u$ are the indices assigned to a_1; $p_1, \ldots, p_v$ are the indices assigned to a_2; $q_1, \ldots, q_x$ are the indices assigned to a_m and $s_1 < s_2 < \cdots < s_u$, $p_1 < p_2 < \cdots < p_v, \cdots$ and $q_1 < q_2 < \cdots < q_x$.

Then Σ_{KB} is an ordered alphabet. Define a mapping $\lambda : \Sigma_{KB} \to \Sigma$ where $\lambda(a_{ij}) = a_i$ for each $a_{ij} \in \Sigma_{KB}$, $1 \leq i \leq m$ and j is an index assigned in Σ_{KB} to the symbol $a_i \in \Sigma$.

```
        /* Step 3, computation of all KB (conceptual) subgraphs: */
for i := 1 to n do begin
        list_subgraphs(i) := { G_i^{sub-j} | G_i^{sub-j} is conceptual subgraph of G_i };
end;
        /* Step 4, computation and encoding of all injective generalisations in
                                          the array sorted_words_markers: */
var main_index := 1;
for each i and G_i^{sub-j} in list_subgraphs(i) do begin
        ⟨sortedSeq(G_i^{sub-j}), annotation(G_i^{sub-j})⟩ :=
                          COMPUTE_LINEAR_RECORD (G_i^{sub-j}, Σ_KB);
            /* note that sortedSeq(G_i^{sub-j}) contains no triples with duplicating
            labels of Σ_KB */
        list_gen_graphs(i, j) :=
         COMPUTE_INJ_GEN(sortedSeq(G_i^{sub-j}), annotation(G_i^{sub-j}), Σ_KB, Σ, λ);
            /* all injective generalisations G_1^{gen}, G_2^{gen}, ..., G_q^{gen} of G_i^{sub-j} are
            stored as triple labels in Σ in list_gen_graphs(i, j). The k-th triple
            of G_1^{gen}, G_2^{gen}, ..., G_q^{gen} is computed as a generalisation of the k-th
            triple of sortedSeq(G_i^{sub-j}). The topological structure of G_p^{gen} is
            given by annotation(G_i^{sub-j}) for 1 ≤ p ≤ q */
        for each i, j and G_p^{gen} in list_gen_graphs(i, j) do begin
        ⟨sortedSeq(G_p^{gen}), annotation(G_p^{gen}), new_lin_labels(G_i^{sub-j})⟩ :=
              ENSURE_PROJ_MAPPING(sortedSeq(G_i^{sub-j}), annotation(G_i^{sub-j}),
                                                   Σ_KB, G_p^{gen}, Σ, λ);
        list_alternative_annot :=
              COMPUTE_ISOMORPHISMS(⟨sortedSeq(G_p^{gen}), annotation(G_p^{gen})⟩);
        words_markers(main_index, 1) := sortedSeq(G_p^{gen});
        words_markers(main_index, 2) :=
          ⟨annotation(G_p^{gen}), list_alternative_annot, new_lin_labels(G_i^{sub-j}), G_i⟩;
        main_index := main_index +1; end;
end;
sorted_words_markers := SORT-BY-FIRST-COLUMN(words_markers);
        /* union of the rows with repetitive words in column 1 of the array
                                                   sorted_words_markers */
while sorted_words_markers(*, 1) contains k > 1 repeating words in column 1,
        starting at row p do begin
```

$sorted_words_markers(p, 2) := \{sorted_words_markers(p, 2),$
$sorted_words_markers(p + 1, 2), \ldots, sorted_words_markers(p + k - 1, 2)\};$
for $1 \leq s \leq k - 1$ **do begin** DELETE-ROW($sorted_words_markers(p + s, *)$
end; **end**;
/∗ definition of a finite list of words over Σ ∗/
$L = \{w_1, \ldots, w_z \mid w_i \in sorted_words_markers(*,1),\ 1 \leq i, j \leq z$ and $w_i \leq w_j$ according to Ω, for $i \leq j\}$.

/∗ *Step 5, FSA construction*: ∗/
Consider L as a finite language over Σ, given as a list of z words which are lexicographically sorted according to Ω. For every word $w_i \in L$ there is a marker associated to it, stored in $sorted_words_markers(i, 2)$ for $1 \leq i \leq z$. Apply the algorithm of [2, 3] and build directly the minimal acyclic FSA with markers at the final states $A_{KB} = \langle \Sigma, Q, q_0, F, \Delta, E, \mu \rangle$, which recognises $L = \{w_1, w_2, \ldots, w_z\}$. Then

$F = \{q_{wi} | q_{wi}$ is the end of the path beginning at q_0 with label w_i, for $w_i \in L,\ 1 \leq i \leq z\}$,
$E = \{M_i \mid M_i = sorted_words_markers(i, 2),\ 1 \leq i \leq z\}$ and
$\mu : q_{wi} \rightarrow M_i$ where $q_{wi} \in F$, $sorted_words_markers(i,1) = w_i$ and $sorted_words_markers(i, 2) = M_i$ for $1 \leq i \leq z$.

The following functions are used in Algorithm 1:

function $\langle sortedSeq(G),\ annotation(G) \rangle$ = COMPUTE_LINEAR_RECORD(G, Σ) where G is normalised SCG, represented as a logical formula and Σ is an ordered alphabet. The function builds the linear record of G as shown in definition 13 and returns pair of strings $\langle sortedSeq(G),\ annotation(G) \rangle$.

function $list_gen_graphs(i, j)$=
COMPUTE_INJ_GEN($sortedSeq(G_i^{sub-j}),\ annotation(G_i^{sub-j}), \Sigma_1, \Sigma_2, \lambda$).
Given the linear record of a SCG G_i^{sub-j} in the ordered alphabet Σ_1, this function returns the list of labels of all injective generalisations $G_1^{gen}, G_2^{gen}, \ldots, G_q^{gen}$ of G_i^{sub-j} written in the ordered alphabet Σ_2. The generalisations are calculated using the mapping $\lambda : \Sigma_1 \rightarrow \Sigma_2$, which defines how the symbols of Σ_1 are generalised by symbols of Σ_2. The generalisations in Σ_2 are computed as linear sequence of triple labels where the k-th triple of G_1^{gen}, G_2^{gen}, $\ldots$, G_q^{gen} generalises the k-th triple in $sortedSeq(G_i^{sub-j})$. The linear sequences of generalisations' labels might be unsorted in Σ_2. The topological structure of G_1^{gen}, G_2^{gen}, $\ldots$, G_q^{gen} is encoded by $annotation(G_i^{sub-j})$ since the triples' order in G_i^{sub-j} and its injective

generalisations is the same. Since Σ_1 is an alphabet containing indices for all *c*-nodes of G_i^{sub-j}, the string *sortedSeq*(G_i^{sub-j}) contains no duplicated triples. The injective generalisations in Σ_2 might contain duplicating triples but their order remains the one encoded by *annotation*(G_i^{sub-j}) since the order of the generalised instances is kept when this function computes the injective generalisations.

function $\langle$*sortedSeq*(G^{gen}), *annotation*(G^{gen}), *new_lin_labels*$(G)\rangle$ =
ENSURE_PROJ_MAPPING(*sortedSeq*(G), *annotation*(G), Σ_1, G^{gen}, Σ_2, λ).
The linear record of a SCG G – $\langle$*sortedSeq*(G), *annotation*$(G)\rangle$ – is given in the ordered alphabet Σ_1. The string of labels of the injective generalisation G^{gen} is given in the ordered alphabet Σ_2 and the k-th triple of G^{gen} generalises the k-th triple in *sortedSeq*(G). The generalisation G^{gen} is calculated using the mapping $\lambda : \Sigma_1 \rightarrow \Sigma_2$, which defines how the symbols of Σ_1 are generalised by symbols of Σ_2. This function

(1) Sorts G^{gen} in Σ_2 and produces a linear record of G^{gen}: $\langle$*sortedSeq*(G^{gen}), *annotation*$(G^{gen})\rangle$,

(2) checks whether the order of *c*-nodes in *sortedSeq*(G^{gen}) corresponds to the order of the respective specialised *c*-nodes in *sortedSeq*(G). If yes, *new_lin_labels*(G)=*sortedSeq*(G). If not, the function rearranges *sortedSeq*(G) in such a way that its i-th symbol is specialisation of the i-th symbol of *sortedSeq*(G^{gen}) and stores the rearranged label sequence in *new_lin_labels*(G). In both cases, the topological links of *new_lin_labels*(G) are encoded by *annotation*(G^{gen}). Thus an injective projection $\pi : G^{gen} \rightarrow G$ is encoded.

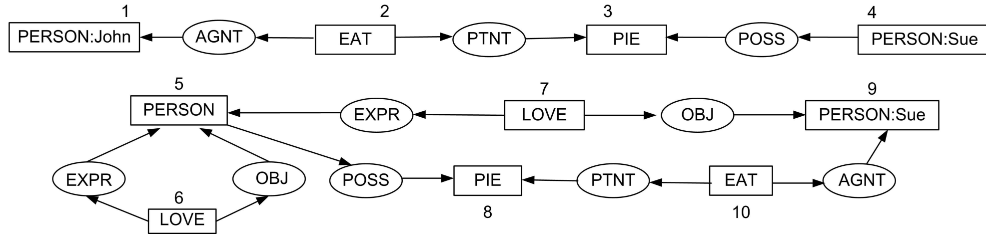
function *list_alternative_annot* =
COMPUTE_ISOMORPHISMS($\langle$*sortedSeq*(G), *annotation*$(G)\rangle$).
Given the linear record of some SCG G, where *sortedSeq*(G) contains multiple triples of identical support symbols, this function constructs a sorted list of all alternative annotations which are due to the isomorphisms among the variables, assigned in the logical formula to the generic concept instances. When *list_alternative_annot* is empty, G has only one annotation which is already computed in the linear record and stored in *annotation*(G).

Example 7. Algorithm 1 is illustrated using the sample KB in example 1. At step 1, an alphabet Σ is defined, which is ordered according to the symbols' order in the Latin alphabet:

Σ = {ACT, AGNT, ANIMAL, ANIMATE, EAT, ENTITY, EVENT, EXPR, LOVE, OBJ, PERSON, PERSON:John, PERSON:Sue, PHYS-OBJECT, PIE, POSS, PTNT, STATE}

Ω : ACT < AGNT< ANIMAL < ANIMATE < EAT < ENTITY < EVENT < EXPR < LOVE < OBJ < PERSON < PERSON : John < PERSON : Sue < PHYS-OBJECT < PIE < POSS < PTNT < STATE

At step 2, unique indices 1-10 are assigned to the ten distinct *c*-nodes in the KB (to ensure the representation of all KB subgraphs as unique linear records):



A new alphabet Σ_{KB} with indices is defined with a respective order Ω_{KB}:

Σ_{KB} = {ACT, AGNT, ANIMAL, ANIMATE, EAT_2, EAT_{10}, ENTITY, EVENT, EXPR, LOVE_6, LOVE_7, OBJ, PERSON_5, PERSON : John_1, PERSON : Sue_4, PERSON : Sue_9, PHYS-OBJECT, PIE_3, PIE_8, POSS, PTNT, STATE } with order

Ω_{KB} : ACT < AGNT < ANIMAL < ANIMATE < EAT_2 < EAT_{10} < ENTITY < EVENT < EXPR < LOVE_6 < LOVE_7 < OBJ < PERSON_5 < PERSON : John_1 < PERSON : Sue_4 < PERSON : Sue_9 < PHYS-OBJECT < PIE_3 < PIE_8 < POSS < PTNT < STATE

In this way we shall maintain two alphabets: Σ_{KB} with indices, to covers the particular KB instances, and Σ with no indices where the future projection queries will be expressed. When computing the generalisations, the mapping λ ensures the treatment of each indexed *c*-node as a non-indexed one. For instance, λ: LOVE_6, LOVE_7 $\rightarrow$ LOVE.

At step 3, all KB subgraphs are computed and stored in *list_subgraphs*. The i-th element of the array contains the list of all the subgraphs of the i-th KB graph G_i. For brevity, we consider only four subgraphs of G_2 with the following linear records:

G_2^{sub-1} : ⟨′LOVE_6 EXPR PERSON_5 LOVE_6 OBJ PERSON_5′, ′1=3, 2=4′⟩,

G_2^{sub-2} : ⟨'LOVE$_6$ EXPR PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$', '2=4'⟩,
G_2^{sub-3} : ⟨'LOVE$_6$ OBJ PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$', '2=4'⟩,
G_2^{sub-4} : ⟨'LOVE$_6$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$', '1=3, 2=4=6'⟩

These four subgraphs belong to *list_subgraphs*(2) together with another 58 subgraphs.

At step 4, all injective generalisations of all KB subgraphs are enumerated as a sorted list of words built over the alphabet Σ. The subgraphs' annotations are turned to respective markers associated to these words. This data is kept in *sorted_words_markers*. Table 1 shows few rows of this array. Column 1 contains sorted generalisation labels in Σ; column 2 lists the subgraphs in Σ_{KB} for which the generalisations are computed. All KB subgraphs are stored together in the corresponding rows of column 2 and thus the compatible subgraphs of G_1 and G_2 are identified. The annotations are parts of the markers in column 2 together with the index of the KB graph from where the subgraphs are taken.

sorted_words_markers(i,1): *sortedSeq(G^{gen})*	sorted_words_markers(i,2): ⟨*annotation(G^{gen}), list_alternative_annot, new_lin_labels(G_2^{sub-j})*, G_2⟩
...	...
LOVE EXPR ANIMAL LOVE OBJ ANIMAL	⟨'1=3, 2=4', [], 'LOVE$_6$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5'$, G_2⟩ ⟨'2=4', [], 'LOVE$_7$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5'$, G_2⟩
....	....
LOVE EXPR ANIMATE LOVE EXPR ANIMATE LOVE OBJ ANIMATE	⟨'1=5, 2=4=6',['3=5, 2=4=6'], 'LOVE$_6$ EXPR PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5'$, G_2⟩
LOVE EXPR ANIMATE LOVE OBJ ANIMATE STATE OBJ ANIMATE	⟨'1=3, 2=4=6', [], 'LOVE$_6$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5$ LOVE$_7$ EXPR PERSON$_5'$, G_2⟩
...	...

Table 1. Sample rows of the array *sorted_words_markers* at the end of step 4

Let us consider the generalisation in column 1

$$\text{LOVE EXPR ANIMATE LOVE EXPR ANIMATE LOVE OBJ ANIMATE} \qquad (g1)$$

computed for G_2^{sub-4}. The linear record of G_2^{sub-4} is constructed by COMPUTE_LINEAR_RECORD in Σ_{KB} and contains no multiple triples with iden-

tical labels. It is sorted according to Ω_{KB} and thus the two triples, starting with LOVE_6, appear as 1st and 2nd triple in $sortedSeq(G_2^{sub-4})$. Then $annotation(G_2^{sub-4})$='1=3, 2=4=6'. However, $(g1)$ is computed in Σ; the function COMPUTE_INJ_GEN produces for $(g1)$ the following string which is not sorted in Σ:

LOVE EXPR ANIMATE LOVE OBJ ANIMATE LOVE EXPR ANIMATE with annotation 1=3, 2=4=6

The function ENSURE_PROJ_MAPPING rearranges $(g1)$ as a sorted sequence of labels $sortedSeq(g1)$ where the triples are ordered in a different manner with $annotation(g1)$='1=5, 2=4=6'. This function also reorders the triples of the subgraph G_2^{sub-4} to correspond to the nodes' order in $sortedSeq(g1)$:

$new_lin_labels(G_2^{sub-4})$ =
'LOVE_6 EXPR PERSON_5 LOVE_7 EXPR PERSON_5 LOVE_6 OBJ PERSON_5'

The string $new_lin_labels(G_2^{sub-4})$ is stored in column 2. In this way for each generalisation G in column 1, its nodes' order corresponds to the order of the specialised nodes in the respective subgraphs in column 2. Moreover, the encoding of the *c*-nodes equivalence $annotation(G)$ is also valid for the subgraph with labels *new_lin_labels* in column 2. Therefore, a (potential) injective projection mapping of a query G (from column 1) to a subgraph (in column 2) is calculated and memorised. In addition $(g1)$ contains multiple triples with identical labels. The function COMPUTE_ISOMORPHISMS is run at step 4 to deal with this issue. It delivers a list of strings *list_alternative_annot* which is also stored in column 2 of Table 1. The annotations of the equivalent isomorphic formulas are memorised off-line to avoid their computation in run time. They are essential because the projection query might be posed to the system as an alternative logical formula which is isomorphic to the one used at the off-line phase.

Some generalisations in column 1 of Table 1 can be projected to more than one KB subgraph listed in column 2. Complex markers are constructed in column 2 at step 4 (as union of single markers) when the repeating strings in column 1 are deleted from the array *sorted_words_markers*. The subgraphs at Figures 7B and 7C are also listed in column 2 as a complex marker with their respective annotations. Thus the conceptual resource is prepared off-line for run-time querying, when a particular projection query will be posed to the system.

Table 2 contains 27 different injective generalisations for the subgraphs $G_2^{sub-1}, \ldots, G_2^{sub-4}$, calculated according to the sample support. Their annotations are grouped into 9 markers:

ACT OBJ ANIMAL LOVE EXPR ANIMAL $M3$
ACT OBJ ANIMAL STATE EXPR ANIMAL $M3$
ACT OBJ ANIMATE LOVE EXPR ANIMATE $M3$
ACT OBJ ANIMATE STATE EXPR ANIMATE $M3$
ACT OBJ PERSON LOVE EXPR PERSON $M3$
ACT OBJ PERSON STATE EXPR PERSON $M3$
LOVE EXPR ANIMAL LOVE EXPR ANIMAL $M2$
LOVE EXPR ANIMAL LOVE EXPR ANIMAL LOVE OBJ ANIMAL $M4$
LOVE EXPR ANIMAL LOVE OBJ ANIMAL $M5$
LOVE EXPR ANIMAL LOVE OBJ ANIMAL STATE EXPR ANIMAL $M4a$
LOVE EXPR ANIMAL STATE EXPR ANIMAL $M6$
LOVE EXPR ANIMATE LOVE EXPR ANIMATE $M2$
LOVE EXPR ANIMATE LOVE EXPR ANIMATE LOVE OBJ ANIMATE $M4$
LOVE EXPR ANIMATE LOVE OBJ ANIMATE $M5$
LOVE EXPR ANIMATE LOVE OBJ ANIMATE STATE EXPR ANIMATE $M4a$
LOVE EXPR ANIMATE STATE EXPR ANIMATE $M6$
LOVE EXPR PERSON LOVE EXPR PERSON $M2$
LOVE EXPR PERSON LOVE EXPR PERSON LOVE OBJ PERSON $M4$
LOVE EXPR PERSON LOVE OBJ PERSON $M5$
LOVE EXPR PERSON LOVE OBJ PERSON STATE EXPR PERSON $M4a$
LOVE EXPR PERSON STATE EXPR PERSON $M6$
LOVE OBJ ANIMAL STATE EXPR ANIMAL $M3$
LOVE OBJ ANIMATE STATE EXPR ANIMATE $M3$
LOVE OBJ PERSON STATE EXPR PERSON $M3$
STATE EXPR ANIMAL STATE EXPR ANIMAL $M6$
STATE EXPR ANIMATE STATE EXPR ANIMATE $M6$
STATE EXPR PERSON STATE EXPR PERSON $M6$

Table 2. A sorted list of all injective generalisations of the subgraphs $G_2^{sub-1}, \ldots, G_2^{sub-4}$

$M1$: $\langle$'1=3, 2=4', [], 'LOVE$_6$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5$', $G_2\rangle$
$M2$: $\langle$'2=4', [], 'LOVE$_6$ EXPR PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$', $G_2\rangle$
$M2a$: $\langle$'2=4', [], 'LOVE$_7$ EXPR PERSON$_5$ LOVE$_6$ EXPR PERSON$_5$', $G_2\rangle$
$M3$: $\langle$'2=4', [], 'LOVE$_6$ OBJ PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$', $G_2\rangle$
$M3a$: $\langle$'2=4', [], 'LOVE$_7$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5$', $G_2\rangle$
$M4$: $\langle$'1=5, 2=4=6', ['3=5, 2=4=6'], 'LOVE$_6$ EXPR PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5$', $G_2\rangle$
$M4a$: $\langle$'1=3, 2=4=6', [], 'LOVE$_6$ EXPR PERSON$_5$ LOVE$_6$ OBJ PERSON$_5$ LOVE$_7$ EXPR PERSON$_5$', $G_2\rangle$
$M5: \quad M1 \cup M3a$
$M6: \quad M2 \cup M2a$

The markers $M2a$, $M3a$ and $M4a$ are created when ENSURE_PROJ_MAPPING rearranges the subgraph' labels. The markers $M5 - M6$ appear when the sorting of *sorted_words_markers*, column 1 groups the repetitive rows together; the repeating values in column 1 are deleted and the markers in column 2 are kept as union of the single markers. The sorted list of generalisations (words over Σ) in column 1, Table 2 is input for the FSA construction.

Fig. 10. Minimal acyclic FSA, encoding all non-empty injective projections onto $G_2^{sub-1} - G_2^{sub-4}$

At step 5, the minimal A_{KB} is constructed directly [2, 3]. Fig. 10 shows the FSA with markers at the final states which accepts words corresponding to

the 27 injective generalisations. The FSA states are presented as circles, the final states as double circles and the transitions – as labeled arcs. An arrow marks the initial state. The FSA has 63 states and 81 transition arcs.

3.3. Discussions Regarding the General Case of Projection. All SCGs with non-empty injective projections to a given KB contain less than k elementary conjuncts, where k is the number of the conceptual relations in the largest KB graph. So given a particular KB status, we can enumerate off-line all these queries and compress them in a minimal FSA. However, such a calculation is impossible for the general projection case, when there is no need for πG to be isomorphic to G. The projection queries might have arbitrary number of elementary conjuncts, so we cannot enumerate in advance all projection queries that have non-empty projections to the KB. If the query length is restricted to some reasonable length, the considerations in section 3.2 can be extended to cover explicit enumeration of all projection mappings. The markers should become more complex, to store all possible mappings in order to memorise in advance all possible projections.

We have to note, however, that the application significance of the non-injective projections is somewhat under question. Since AI operates on types and instances, it remains unclear whether mixing instances in the specialisation process is a good idea. This is done by π_2 in example 3 where the query '*is there an ANIMAL who feels two (different) states*' receives the answer '*there is a PERSON who feels (one type of) love*'. The reasonableness of this answer is disputable. So we believe that the most important, practical pattern search is based on injective projections.

4. Injective Projection as a Run-Time Look-up in the Minimal FSA.

Given a run-time query G, its injective projection to the KB is calculated by a look-up in the minimal acyclic FSA which encodes all KB injective generalisations. Here we shall assume that G is posed to the FSA-archive as a logical formula and that G holds in the same world where the KB is acquired, i.e. we shall use the support S and the ordered alphabet Σ.

Algorithm 2. *Finding all injective projections of a SCG with binary conceptual relations G onto a KB of SCGs with binary conceptual relations, which is encoded by algorithm 1 as a minimal acyclic FSA* $A_{KB} = \langle \Sigma, Q, q_0, F, \Delta, E, \mu \rangle$:

Step 1. Consider the monadic and binary predicates in the logical formula of G. If they contain labels which are not present in the KB support, G has empty

projection onto the KB.

Step 2. $\langle sortedSeq(G),\ annotation(G)\rangle$:= COMPUTE_LINEAR_RECORD(G, Σ).
Build the word w_G=*sortedSeq*(G) and look up in A_{KB} using w_G (i.e. follow the unique A_{KB} path p_G starting at q_0 with label w_G). If p_G does not lead to a final state in A_{KB}, then G has empty injective projection onto the KB. Otherwise take the state q_G of A_{KB}, $q_G \in F$ with marker M_q such that the path p_G ends at q_G.

Step 3. Consider M_q at q_G. M_q is a set of k single 4-tuple markers, where $k \geq 1$:
$\{\langle annotation_1,\ list_alternative_annot_1,\ new_lin_labels_1,\ G_{i1}\rangle, \ldots\ldots\ldots,$
$\langle annotation_k,\ list_alternative_annot_k,\ new_lin_labels_k,\ G_{ik}\rangle\ \}$

for $1 \leq j \leq k$ **do begin**
  if $annotation(G) = annotation_j$ **then** return $\langle new_lin_labels_j,$
    $annotation(G)\rangle$ as an injective projection of G onto G_{ij};
  if $annotation(G) \in list_alternative_annot_j$
    then $annotation(G) := annotation_j$;
    return $\langle new_lin_labels_j,\ annotation(G)\rangle$ as an injective projection of G onto G_{ij};
  end

G has empty projection onto the KB.

Theorem 1. *All injective projections of G, a normalised SCG with binary conceptual relations, onto a KB of normalised SCGs with binary conceptual relations can be calculated by algorithm* 2 *as look-ups in the FSA A_{KB} built by algorithm* 1.

Proof. Algorithm 1 constructs off-line a minimal acyclic FSA with markers at the final states A_{KB}, which encodes all injective generalisations of the KB subgraphs. They are in fact all SCGs that have injective specialisations in the KB at the particular moment and therefore, non-empty injective projections onto the KB. Algorithm 2 interprets the query G as a sequence of symbols w_G. If G has a non-empty projection onto the KB, then $w_G \in L(A_{KB})$ which is ensured by the construction of A_{KB} by algorithm 1. The look-up of a A_{KB} path, starting at q_0 and labeled by w_G, identifies the markers at the final state q_G of A_{KB}, where the actual projection mappings are pre-listed. Algorithm 2 identifies the string *annotation*(G) in case that it is stored in the marker of q_G. Since all isomorphisms are pre-computed by algorithm 1, all possible injective mappings of G onto the KB are stored in the marker. In this way the off-line enumeration of linear sequences of graph labels, together with the associated annotations, ensures

run-time identification of all injective projection mappings of a query G onto a given KB at some particular moment. □

5. Algorithmic complexity. The complexity of the main tasks to be performed off-line and on-line can be estimated separately, as they are separately performed.

The **off-line tasks** have exponential complexity. Let us split them into five main components:

• ***Finding all connected subgraphs of the KB SCGs and recording them as linear records ⟨sortedSeq,annotation⟩.*** Given a SCG with n c-nodes, the search of its subgraphs has complexity $O(2^n)$. Sorting a list of symbols is a $O(k \log k)$ algorithm, where k is the number of the list items. Assigning the *annotation* can be done in linear time, with respect to the number of the subgraph c-nodes. The general complexity of this component is $O(n \times 2^n)$.

• ***Computing all injective generalisations of the KB subgraphs, recording them as linear records, and resorting some of the KB subgraphs to ensure the correspondence of their nodes order to the order of the nodes in the respective linear record of injective generalisations.*** The calculation of all injective generalisations depends on the maximal depth d of the subsumption relation, which is defined in the type hierarchies, on the number of subgraphs m and their maximal length s. Its complexity is $O(m \times d^s)$. As we said above, sorting is a task with complexity $O(k \log k)$ depending on the length k of the symbol strings to be sorted. Calculating the *annotation* of a SCG can be done in linear time. The general complexity of this task is $O(m \times d^s \times (s + \log m))$.

• ***Building all alternative annotations for all the generalisations (by computing the isomorphisms among the duplicating triples of the logical formulas).*** Let *sortedSeq*(G) of the linear record of a generalisation G contains m groups of duplicating triples with lengths correspondingly $k_1, k_2, \ldots, k_m$ elementary conjuncts. The isomorphisms correspond to the permutations of the duplicating triples within the positions occupied by these triples in *sortedSeq*(G). Their calculation has complexity $O(k_1! \times k_2! \times \cdots \times k_m!)$.

• ***Building the FSA with markers at the final states A_{KB}.*** As shown in [3], the complexity of a minimal automaton construction is $O(n \log(m))$, where n is the total number of symbols in the input list of words and m is the number of A_{KB} states.

The FSA A_{KB} can be supported as an alternative KB representation. Adding a new SCG to the KB (i.e. to its FSA) is relatively easy but changes

and updates in general require recalculation of the whole FSA. So we consider the complexity of the FSA updates as another off-line task:

• ***Providing KB updates by assertion and deletion of words in the FSA A_{KB}.*** A single word can be inserted or deleted in linear time, depending on the word length n. However, for a whole SCG, the insertion or deletion of all words-generalisation of subgraphs, can be done for $O(m_1 \times d^{s1})$ where m_1 is the number of generalisations, s_1 is the maximal subgraph length and d is the depth of the subsumption relation.

Run-time tasks: There are two main on-line tasks, given a query G as a logical form:

• ***Presenting G as a sorted sequence of support symbols***, with complexity $O(n \log n)$, where n is the number of G symbols, ***and calculation of its annotation*** for linear time $O(n)$;

• ***Look-up in the FSA A_{KB} by a word w_G.*** Its complexity is clearly $O(n)$, where n is the number of G symbols. No matter how large the KB is, all injective projections of G to the KB are found at once with complexity depending on the input length only.

Comparing these figures to the run-time complexity results, we see the benefits of explicit off-line enumerations. It is also trivial to check whether two SCGs are equivalent.

6. Experimental assessment.

To study the proposed scenario and its practical settings, we have generated automatically two test dataset of supports and SCGs with binary conceptual relations in normal form. In fact we have implemented a workbench for random generation of testing data, given some basic parameters like number of concept and relation types, depth and width of the support hierarchies, number of graphs in the knowledge base, graph length in elementary conjuncts and so on. The parameters of the two experiments are summarised in Table 3. The input text files for FSA construction consist of lines corresponding to all injective generalisations in the experiment. Each line has the format:

$$\langle \textit{sortedSeq}(G_i) \text{ of the linear records of a generalisation } G_i,\ \textit{Marker} \rangle$$

where $1 \leq i \leq 10436190$ for test 1 and $1 \leq i \leq 140031027$ for test 2. The minimal FSAs, which compress all injective generalisations of the test KBs, are built off-line using results of [2, 3].

Number of:	Test 1	Test 2
1. Concept types in the support	600	1025
2. Conceptual relation types in the support	40	10
3. Maximal hierarchy depth	18	24
4. Hierarchy supertypes (average per type)	2,32	2,0009
5. SCGs in the knowledge base	291	329
6. Elementary conjuncts (SCGs length)	3-10	3-12
7. (Conceptual) subgraphs in the KB	6 753	11 146
8. Injective generalisations of all subgraphs	10 436 190	140 031 027
9. Annotation (structural identity) types	13 885	3 618
10. States in the final minimal FSA	2 751 977	23 956 007
11. Transitions in the final minimal FSA	3 972 096	43 347 641
12. Size of the input text file in UNICODE – sorted list of linear records for all generalisations	891,4 MBytes	$\sim$ 13GBytes
13. Size of the minimal FSA without the markers at the final states (only pointers to them are kept)	52,44 MBytes	612,73 MBytes
14. Compression rate input file / minimal FSA	$\sim$ 17times	$\sim 21,2$times
15. Size of the input file (zipped by bzip2)	21,8 MBytes	
16. Ratio zip/FSA	2,4 times	

Table 3. Two tests of Algorithm 1 with randomly-generated supports and knowledge bases

Test 2 was constructed on a bigger and deeper concept hierarchy (Table 3, lines 1 and 3) and with longer SCGs, which contained up to 12 elementary conjuncts (line 6). Therefore there are more subgraphs in test 2 (line 7) and the number of all injective generalisations in test 2 is much higher than in test 1 (line 8). However, the generalisations in test 2 have much more regular topological structure (line 9). This is due to the fact that the SCGs in test 2 are variations of some 4-5 predefined structural patterns. About 30% of the SCGs are connected like stars with one common concept for all elementary conjuncts. In this way all subgraphs are also star-like and the annotations vary less when the labels are sorted. So for test 2, the compression rate of the input file is higher (line 14). In both tests the compression rate is essential and resembles the compression rate for morphological dictionaries, represented as minimal acyclic FSA [2]. For test 1, we have also compared the zipped input file to the FSA size; the FSA is only 2,4 times bigger but ensures run-time look-ups in linear time (line 16).

Since the maximal number of elementary conjuncts in the two experiments

is 10 and 12 respectively, we can compare the corresponding Bell numbers of equivalence classes to the numbers of topological structures in the experimental datasets. We are interested in B_{20} (which corresponds to the equivalence classes of 20 arguments in 10 elementary conjuncts) and B_{24} (24 arguments in 12 elementary conjuncts). Line 9 of Table 3 shows quite limited variety of the topological structures in the experimental data sets while

$$B_{20}= 51\ 724\ 158\ 235\ 372 \text{ and } B_{24}=445\ 958\ 869\ 294\ 805\ 000.$$

7. Conclusion. This paper introduces the idea of storing conceptual information as a regular language. The approach implements off-line as much NP-hard computations as possible and provides exclusive run-time efficiency. The proposal is based on a brute-force enumeration and sorting of all possible injective generalisations which might look unusual from the classical knowledge representation perspective. All SCGs treated by algorithms 1 and 2 have to be normalised and turned to structures with minimal numbers of support symbols, as the FSAs interpret each symbol occurrence as a different one. Some graph mappings are lost in this way, compared to other SCGs representation formats, but we are focused on a pragmatic, practical procedure for fast run-time calculations and our SCG model is insignificantly simplified. The suggested idea is helpful in one more respect: it reveals the similarity among graphs and formulas which consist of the same predicates and arguments but with different links (e.g. the SCGs at Fig. 7A, 7B and 7C). Most AI algorithms would not consider them as *similar* but in some sense they are very *close* since they deal with the same notions.

There are two novel proposals in our approach: (*i*) to enumerate explicitly all generalisations over a unified ordered alphabet, assigning them 'structural' annotations and (*ii*) to encode the whole KB as a single minimal, acyclic automaton, which makes the run-time search dependent on the query length only, no matter how big the KB is. The FSA-based encoding is built on insights and intuitions stemming from both the logical interpretation of SCGs with binary conceptual relations and from their graphical representation. The implementation requires considerable off-line preprocessing and large space. The experiments support our claims regarding the feasibility and the scalability of the approach, since the resulting conceptual resources are turned to compact data structures that are kept in the RAM. The high compression rate is due to the fact that the KB subgraphs and their injective generalisations are relatively uniform structures. Obviously the star graphs in the support impose strong constraints on

the structural patterns while computing injective generalisations; now we see experimental evidences about the 'uniformity'. So the suggested encoding provides ultra-efficient run-time calculation and radically reduces the time for the on-line processing. This approach is reasonable because every search of conceptual patterns is run over a (relatively) static KB, which is not updated at that particular moment. Therefore the KB can be preliminary encoded in a way which provides efficient run-time computations and can be continuously supported off-line in this internal format.

As there is a variety of ways to implement the proposed internal operations, the main ideas are only sketched here. The considerations reveal the important notion of 'conceptual subgraph'; we see that the run-time algorithms for projection computation include calculations and checks of subgraphs that are not conceptual graphs. But these unnecessary computations are inevitable during the run-time projection calculation, as it operates on row data in contrast to our approach, which relies on very precise data preparation to be done off-line.

The philosophy of off-line data preparation is already adopted by many advanced applications. For instance, Google searches constantly the Web and prepares its inverted indices off-line. Everyone uses Google because of its speed, no matter how much space is needed to provide it. The knowledge-based applications will soon have fast access to almost unrestricted memory space, so run-time complexity is the main challenge to face. Therefore, special algorithms should be designed to improve the run-time efficiency. We believe that the efficient internal encodings of conceptual structures are a must as they will enable fast services of the modern semantic systems and their further application success.

REFERENCES

[1] Roche E., Y. Schabes (Eds) Finite State Language Processing. MIT Press, Cambridge, Massachusetts, 1997.

[2] Mihov St. Minimal Acyclic Automata: Constructions, Algorithms, Applications. PhD thesis, Sofia, 2000.

[3] Daciuk J., St. Mihov, B. Watson, R. Watson. Incremental Construction of Minimal Acyclic Finite State Automata. *J. of Computational Linguistics*, **26**, No 1 (2000), 3–16.

[4] Angelova G., S. Mihov. Finite State Automata and Simple Conceptual Graphs with Binary Conceptual Relations. In: Supplementary Proceedings of

the 16th Int. Conf. on Conceptual Structures (ICCS'08), (Eds P. Eklund, O. Haemmerlé), CEUR Workshop Proceedings 2008, ISSN 1613-0073, 139–148.

[5] Angelova G. Efficient Computation with Conceptual Graphs. Book chapter in: (Eds P. Hitzler, H. Scharfe), Conceptual Structures in Practice, Chapman & Hall/Crc Studies in Informatics Series, Vol. **2**, March 2009.

[6] Sowa J. Conceptual Structures – Information Processing in Mind and Machine. Reading, MA Addison Wesley, 1984.

[7] Chein M., M.-L. Mugnier. Conceptual Graphs: fundamental notions. *Revue d'Intelligence Artificielle*, **6**, No 4 (1992), 365–406.

[8] Baget J.-F., M.-L. Mugnier. Extensions of Simple Conceptual Graphs: the Complexity of Rules and Constraints. *Journal of AI Research*, **16** (2002), 425–465.

[9] Mugnier M.-L., M. Chein. Polynomial Algorithms for Projection and Matching, In: Conceptual Structures: Theory and Implementation (Eds H. Pfeiffer, T. Nagle), Springer, Lecture Notes in Artificial Intelligence, Vol. **754**, 1992, 239–251.

[10] Mugnier M.-L.On Generalization/Specialization for Conceptual Graphs. *Jour. of Experimental and Theoretical Computer Science*, **7**, (1995), 325–344.

[11] Croitoru M., E. Compatangelo. A combinatorial approach to conceptual graph projection checking. In: Proceedings of the 24th International Conference of the British Computer Society, Special Group on AI (SGAI'2004), 130–143.

[12] Hopcroft J., R. Motwani, J. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA, 1983.

[13] Hopcroft J. An $n \log n$ algorithm for minimizing states in a finite automaton. In: The Theory of Machines and Computation, (Ed. Z. Kohavi), Academic Press, 1971, 189–196.

[14] Weisstein. E. Bell number. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/BellNumber.html`, 13 January 2009.

Galia Angelova
Institute for Parallel Processing
Bulgarian Academy of Sciences
Acad. G. Bonchev Str., Bl. 25A
1113 Sofia, Bulgaria
e-mail: `galia@lml.bas.bg`

Stoyan Mihov
Institute for Parallel Processing
Bulgarian Academy of Sciences
Acad. G. Bonchev Str., Bl. 25A
1113 Sofia, Bulgaria
e-mail: `stoyan@lml.bas.bg`

Received December 2, 2008
Final Accepted January 8, 2009