Simple Conceptual Graphs Revisited: Hypergraphs and Conjunctive Types for Efficient Projection Algorithms

Jean-François Baget

INRIA Rhône-Alpes
baget@inrialpes.fr

Abstract. Simple Conceptual Graphs (SGs) form the cornerstone for the "Conceptual Graphs" family of languages. In this model, the subsumption operation is called projection; it is a labelled graphs homomorphism (a NP-hard problem). Designing efficient algorithms to compute projections between two SGs is thus of uttermost importance for the community building languages on top of this basic model.
This paper presents some such algorithms, inspired by those developped for Constraint Satisfaction Problems. In order to benefit from the optimization work done in this community, we have chosen to present an alternate version of SGs, differences being the definition of these graphs as hypergraphs and the use of conjunctive types.

1 Introduction

Introduced in [22], Simple Conceptual Graphs (or SGs) have evolved into a family of languages known as Conceptual Graphs (CGs). In the basic SG model, the main inference operator,*projection*, is basically a labelled graph homomorphism [10]. Intuitively, a projection from a SG H into a SG G means that all information encoded in H is already present in G. This operation is logically founded, since projection is sound and complete w.r.t. the first-order logics semantics Φ.

As a labelled graph homorphism, deciding whether a SG projects into another one is a NP-complete problem [10]; moreover, considering this *global* operation (instead of a sequence of local operations, as in [22]) allows to write more efficient algorithms. Indeed, this formulation is very similar to the *homomorphism theorem*, considered vital for database query optimization [1]. Designing more efficient projection algorithms is of uttermost importance to the GG community, not only for SGs reasonings, but for usual extensions of this basic model: reasonings in nested CGs can be expressed as projection of SGs [3], CG rules rely on enumerating SGs projections [20], a tableaux-like method for reasonings on full CGs uses SGs projection to cut branches of the exploration tree [16]...

The `BackTrack` algorithm (or BT) [13] has naturally been used in numerous CG applications to compute projections (e.g. in the platform CoGITaNT [12]). However, the CG community has not collectively tried to improve it. On the other hand, the Constraint Satisfaction Problem (CSP) community has been

A. de Moor, W. Lex, and B. Ganter (Eds.): ICCS 2003, LNAI 2746, pp. 229–242, 2003.

working for the last 20 years on various BT improvements. Although [18] have pointed out the strong connections between SG-Projection and CSP, this work has mainly led to exhibit new polynomial subclasses for SG-Projection. Our goal in this paper is to propose a translation of some generic algorithms developped in the CSP community to the SG-Projection problem.

Though numerous algorithms have been proposed to solve CSPs, we considered the following criteria to decide which one(s) to adopt: 1) the algorithm must be *sound* and *complete* (it must find only, and all projections); 2) the algorithm must be *generic*, *i.e.* not developped for a particular subclass of the problem; 3) the algorithm must add as little *overhead cost* as possible. Let us now precise the third point. Many CSP algorithms rely on powerful filtering techniques to cope with exceptionally difficult instances of the problem (*i.e.* when the graphs involved are dense random graphs, corresponding to the *phase transition* [21]). However, when the graphs involved are sparse and well-structured graphs (and it seems to be the case for SGs written by human beings), these algorithms, inducing a high overhead cost, are not as efficient as those presented here.

The algorithms presented here are `BackMark` [11] and `Forward Checking` [14]. We present them in an unified way, using an original data structure that allow to write them with small code modification, and to execute them at little overhead cost. It was a surprise to see, in [8], how much more efficient was the `Forward Checking` algorithm when we considered SGs as *hypergraphs*, instead as their associated *bipartite graph* (as is usually done for SGs). We have then decided to present SGs as hypergraphs (following the proposal of [7], whose reasons were essentially to simplify definitions). Integrating a *type conjunction* mechanism (as done in [2, 4, 9]) also allows us to reduce the number of projections from a graph into another one, while keeping the meaningful ones.

This paper is organized in two distinct parts. In Section 2 (Syntax and Semantics), we recall main definitions and results about SGs, in our hypergraph formalism. Section 3 (Algorithms) is devoted to projection algorithms.

2 Syntax and Semantics

This section is devoted to a definition of SGs as hypergraphs. Syntax is given for the support (encoding ontological knowledge), and for SGs themselves (representing assertions). The notion of consequence is defined by model-theoretic semantics, and we show that projection is sound and complete with respect to these semantics. For space requirements, definitions are given without much examples. However, we discuss at the end of this section how our definitions relate to usual ones (as in [17], for example).

2.1 Syntax

The support encodes the vocabulary available for SGs labels: individual markers will be used to name the entities represented by nodes, and relation types, used to label hyperarcs, are ordered into a type hierarchy. SGs encode entities (the nodes) and relations between these entities.

Definition 1 (Support). *A* support *is a tuple* $\mathcal{S} = \langle \mathcal{M}, T_1, ..., T_k \rangle$ *of pairwise disjoint partially ordered sets such that:*

- $\mathcal{M}$*, the set of* markers*, contains a distinguished element, the* generic marker $*$*, greater than the other pairwise non comparable* individual markers*;*
- $T_1, ..., T_k$ *are the sets of* types*, and a type* $t \in T_i$ *is said to be of* arity i*.*

We note $\leq$ *the order relation on elements of the support, be it types or markers.*

Definition 2 (Simple Conceptual Graphs). *A* simple conceptual graph*, defined over a support* $\mathcal{S}$*, is a tuple* $G = (V, U, mark, type)$ *where* V *and* U *are two finite disjoint sets, and mark and type are two mappings.*

- V *is the set of* nodes*, and* $mark : V \rightarrow \mathcal{M}$ *labels each node by a* marker *of* $\mathcal{M}$*. Nodes labelled by* $*$ *are called* generic*, other are called* individual*.*
- $U \subseteq V^+$ *is a set of hyperarcs called* relations*, defined as non empty tuples of nodes, and* $type : U \rightarrow (2^{T_1} \cup \ldots \cup 2^{T_k}) \setminus \{\emptyset\}$ *labels each relation of size* i *by a non empty set of types of arity* i*.*

Though similar definitions of SGs as hypergraphs have already been proposed (e.g. in [7]), the lack of any type for nodes and the typing of relations by sets of types must be explained. Indeed, the type of an unary relation incident to a node can be seen as the type for this node (sect 2.5), and the set of types labelling a node can be understood as the conjunction of these types (sect. 2.2).

2.2 Model Theoretic Semantics

Motivations for expressing semantics in model theory are stated in [15]:
"A model-theoretic semantics for a language assumes that the language refers to a world*, and describes the minimal conditions that a world must satisfy in order to assign an appropriate meaning for every expression in the language. A particular world is called an* interpretation *[...] The idea is to provide an abstract, mathematical account of the properties that any such interpretation must have, making as few assumptions as possible about its actual nature or intrinsic structure. [...] It is typically couched in the language of set theory simply because that is the normal language of mathematics."*

Definition 3 (Interpretations, models). *An* interpretation *of a support* $\mathcal{S}$ *is a pair* $(\mathcal{D}, \mathcal{I})$*, where* $\mathcal{D}$ *is a set called the* domain*, and the* interpretation function $\mathcal{I}$ *associates to each individual marker in* $\mathcal{M}$ *a distinct element of* $\mathcal{D}$*, and to each type* $t \in T_i$ *a subset of* $\mathcal{D}^i$ *such that* $t \leq t' \Rightarrow \mathcal{I}(t) \subseteq \mathcal{I}(t')$*.*

Let G *be a SG defined over* $\mathcal{S}$*. An interpretation* $(\mathcal{D}, \mathcal{I})$ *of* $\mathcal{S}$ *is a* model *of* G *if there exists a mapping* $\sigma : V \rightarrow \mathcal{D}$ *(called a proof that* $(\mathcal{D}, \mathcal{I})$ *is a model of* G*) such that:*

- *for each individual node* x*,* $\sigma(x) = \mathcal{I}(mark(x))$*;*
- $\forall\ r = (x_1, \ldots, x_i) \in U(G)$*,* $\forall\ t \in type(r), (\sigma(x_1), \ldots, \sigma(x_i)) \in \mathcal{I}(t)$*.*

Note that a set of types is interpreted as the conjunction of these types. As usual in logics, the notion of consequence follows the notion of models:

Definition 4 (Consequence). *Let* G *and* H *be two SGs defined a support* $\mathcal{S}$*. We say that* H *is a* consequence *of* G *if all models of* G *are also models of* H*.*

2.3 Projection

The inference mechanism (answering the question "is all information encoded in graph H already present in graph G?") was initially presented as a sequence of elementary, local operations trying to transform a SG H into a SG G [22]. These operations have been completed [17] to handle the case of disconnected graphs.

The reformulation of this mechanism as a labelled graph homomorphism called *projection* [10] allowed to write backtrack-based, more efficient algorithms.

Definition 5 (Projection). *Let G and H be two SGs defined over a support $\mathcal{S}$. A* projection *from H into G is a mapping $\pi : V(H) \to V(G)$ such that:*

- *for each node $x \in V(H)$, $mark(\pi(x)) \leq mark(x)$;*
- *for each relation $r = (x_1, \ldots, x_i) \in U(H)$, $\forall t \in type(r)$, there exists a relation $r' = (\pi(x_1), \ldots, \pi(x_i)) \in (G)$ such that $\exists t' \in type(r'), t' \leq t$.*

2.4 Main Equivalence Result

It remains now to prove that projection is sound and complete with respect to the model-theoretic semantics defined above. The following theorems have been proven respectively in [23] and [18], where the semantics were given by translating the graphs in first-order logic formulas.

Theorem 1 (Soundness). *Let G and H be two SGs defined over a support $\mathcal{S}$. If there exists a projection from H into G, then H is a consequence of G.*

Both [24] and [18] exhibited independently a counterexample to projection completeness w.r.t. the logical semantics. They proposed different solutions to this problem: [24] proposed to modify the SG H, putting it in *anti-normal form*, and [18] proposed to modify G, putting it in *normal form*. We present here the second solution, since a consequence of the first one is to lose structural information on the graph H, leading to less efficient projection algorithms.

Definition 6 (Normal Form). *A SG is said in* normal *form if no two distinct nodes share the same individual marker.*

To put a SG H into its normal form $\text{nf}(H)$, we fusion into a single node all nodes sharing the same individual marker. Hyperarcs incident to the original nodes are made incident to the node resulting from this fusion. Should we obtain multiple relations (*i.e.* defined by the same tuple of nodes), we keep only one of those (our hypergraphs are not multigraphs, as usually defined for SGs), and label it by a type obtained by the union of the original types (*i.e.* their conjunction). Note that since H and $\text{nf}(H)$ have the same semantics, Th. 1 remains true if we consider $\text{nf}(H)$ instead of H.

Theorem 2 (Completeness). *Let G and H be two SGs defined over a support $\mathcal{S}$. If H is a consequence of G, then there exists a projection from H into $nf(G)$.*

Proof. Soundness ($\Rightarrow$) and completeness ($\Leftarrow$) rely on seeing an interpretation $(\mathcal{D}, \mathcal{I})$ of a support $\mathcal{S}$ as a SG $\mathcal{G}(\mathcal{D}, \mathcal{I})$ over $\mathcal{S}$, and a proof that $(\mathcal{D}, \mathcal{I})$ is a model as a projection.

- consider each element d of $\mathcal{D}$ as a node, whose marker is the individual marker m such that $\mathcal{I}(m) = d$ if defined, the generic marker $*$ otherwise;
- for every tuple $(d_1, \ldots, d_i)$, consider the set of types $\{t_1, \ldots, t_p)$ such that, for $1 \leq j \leq p$, $(d_1, \ldots, d_i) \in \mathcal{I}(t_j)$. If this set $\{t_1, \ldots, t_p)$ is non empty, then the graph contains a relation $(d_1, \ldots, d_i)$ whose type is $\{t_1, \ldots, t_p)$.

Lemma 1. *Let G be a SG defined over $\mathcal{S}$, and $(\mathcal{D}, \mathcal{I})$ be an interpretation of $\mathcal{S}$. Then σ proves that $(\mathcal{D}, \mathcal{I})$ is a model of G iff σ is a projection from G into $\mathcal{G}(\mathcal{D}, \mathcal{I})$.*

($\Rightarrow$) Let us suppose there exists a projection from H into G. We have to prove that every model of G is a model of H. Let us take an arbitrary model $(\mathcal{D}, \mathcal{I})$ of G (this is proven by a mapping σ). Then (Lemma 1, $\Rightarrow$ part), σ is a projection from G into $\mathcal{G}(\mathcal{D}, \mathcal{I})$. As the composition of two projections is a projection, then $\pi \circ \sigma$ is a projection from H into $\mathcal{G}(\mathcal{D}, \mathcal{I})$, thus (Lemma 1, $\Leftarrow$ part) $(\mathcal{D}, \mathcal{I})$ is a model of H.

($\Leftarrow$) Let us now suppose that every model of G is a model of H. In particular, we consider the model $(\mathcal{D}, \mathcal{I})$ of G and H that is *isomorphic* to G, *i.e.* $\mathcal{I}$ establishes a bijection from $V(G)$ to $\mathcal{D}$, and $(d_1, \ldots, d_i) \in \mathcal{I}(t), t \in T_i$ if and only if there is a type $t' \in \text{type}(\mathcal{I}^{-1}(d_1), \ldots, \mathcal{I}^{-1}(d_i))$ such that $t' \leq t$. Note that such a bijection $\mathcal{I}$ is possible only since G is a SG in normal form. Then $\mathcal{I}$ is a proof that $(\mathcal{D}, \mathcal{I})$ is a model of G. Let us call σ the proof that $(\mathcal{D}, \mathcal{I})$ is a model of H. Check that $\sigma \circ \mathcal{I}^{-1}$ is a projection from H into G, verifying that $\mathcal{I}^{-1}$ is a projection, and conclude with Lemma 1. □

2.5 Relationships with "Usual SGs"

Above definitions differ obviously from usual ones. Before pointing out the differences, we first rely on graphical representation to highlight their similarities.

Nodes of a SG are represented by rectangles. If $n \geq 2$ and $(x_1, \ldots, x_n)$ is a relation, we draw it by an oval, write its type inside the oval, then for $1 \leq i \leq n$, we draw a line between this oval and the rectangle representing x_i and write the number i next to it. Inside the rectangle representing a node, we write the string "$T : M''$ where T is the union of the types of unary relations incident to the node, and M is the individual marker labelling the node (we write nothing if the graph is generic). Moreover, to highlight that a set of types is interpreted as a conjunction of these types, we write $t_1 \sqcap \ldots \sqcap t_p$ for the set $\{t_1, \ldots, t_p\}$.

Up to our introduction of conjunctive types, this is exactly the drawing of a SG in a traditional sense, and these objects have exactly the same semantics. Indeed, SGs are usually defined as bipartite labelled multigraphs, where the two classes of nodes (concept nodes and relation nodes) correspond to the objects obtained when considering the *incidence bipartite* graphs of our hypergraphs.

Though conjunctive types may seem to add expressiveness to SGs, [2, 4, 9] have pointed out it is not the case. Indeed, SGs usually allow multiple relations (in our sense) being defined by the same tuple of nodes, and such a set of relations has the same interpretation as a single relation whose type is the conjunction of their types. And since a concept node type has the same interpretation as a

unary relation type, is it possible to encode, *at a semantic level*, type conjunction in "usual SGs". However, since type conjunction is not integrated *at a syntactic level*, it is a problem when trying to fusion concept nodes having different types.

3 Algorithms

Before presenting the algorithms themselves, we recall that SG-Projection is an NP-complete problem ([10], using a reduction to Clique).

Theorem 3 (Complexity). *The problem* SG-Projection *(given a support $\mathcal{S}$, deciding whether or not a SG H projects into a SG G) is NP-complete.*

3.1 Number of Projections

Enumeration of projections from a SG H into a SG G benefits from our definitions. With usual SGs definitions, relations are nodes, that have to be projected.

Consider the following example: the support $\mathcal{S}$ contains only binary relation types $\{r, r_1, \ldots, r_k\}$ where $r_1, \ldots, r_k$ are pairwise non comparable and r is greater than all the other ones, and a unique unary relation type (or concept type), t. We build the graph G as a chain composed of n generic (concept) nodes, each one linked to the next by k relation nodes, respectively typed $r_1, \ldots, r_k$. Then consider the graph H, composed of a chain of nodes linked by relations typed r. Following the usual SGs definitions (where both concept nodes and relation nodes have to be projected), we have exacly $(n-1)^k$ projections from H into G. With our definitions, the graph G is a chain of n nodes, each one linked to the next by *one* relation node typed $r_1 \sqcap \ldots \sqcap r_k$, and there is exacly one projection from H into G, essentially capturing the meaning of the $(n-1)^k$ previous ones.

This potentially exponential gain in the number of projections can be important in CG models that rely on enumerating projections, such as rules [20].

3.2 BackTrack

The naïve algorithm searching one or all projections from a SG H into a SG G builds all mappings from $V(H)$ into $V(G)$, and tests whether one of them is a projection. This process can be seen as the building of a *seach tree*. Its nodes represent partial mappings of $V(H)$ into $V(G)$. The root represents the empty mapping. If a node N of the tree represents the partial mapping π, and x is the next unmapped node of $V(H)$, then the sons of N represent the mappings $\pi \cup \{(x, y_1)\}, \ldots, \pi \cup \{(x, y_p)\}$ where $y_1, \ldots, y_p$ are the nodes of G. Leaves of this search tree represent all mappings from $V(H)$ into $V(G)$. `BackTrack` [13] (or BT) explores this search tree with a depth-first traversal, but it does not wait to reach leaves to check whether a mapping is a projection. If a mapping from a subset of nodes of H is not a projection, then it cannot be extended to a projection (the node of the search tree corresponding to this partial mapping is called a *failure node*). Checking if we have a partial projection at each node

The version of BT we present is an iterative one. As suggested in [19], it allows a better control of the stack of recursive calls, and is thus more convenient for enhancements presented later on. It will be used as the body of all BT improvements presented here: only the functions it calls should be rewritten. Let us briefly explain this algorithm. Following the writing of `Solution-Found`, it can compute one projection (when called, this function stops the execution, and returns the current mapping), or all projections (when called, it saves the current mapping, and all mappings will be read at the end of the program).

The variable `level` corresponds to the current level in the search tree, and thus identifies the node we have to examine (`current-Node`). The variable `up` will be set to `true` when the current mapping does not correspond to a projection, and a backtrack is required. Every node has a field `image` used to store its image by the curent partial projection. Let us now examine the different functions called by this algorithm. `Order` totally orders (consider an arbitrary order) nodes of the graph. We denote by $x_i = V(H)[i]$ the ith node, and by $\text{pre}_V(x_i) = \{x_1, \ldots, x_{i-1}\}$ the nodes placed before him in this table. In the same way, $\text{pre}_U(x_i) = \{(y_1, \ldots, y_k) \in U(H) | \exists 1 \leq j \leq k y_j = x_i \text{ and } \forall 1 \leq j \leq k, y_j \in \text{pre}_V(x_i) \cup \{x_i\}\}$. `Next-Level`$(x_i)$ (where x_i is at position i in the table) returns $i + 1$, and `Previous-Level`(x_i) returns $i - 1$. `Find-Candidates`(x_i) stores in the field `image` of x_i the first node $y \in V(G)$ that is a *candidate* for x_i: it means that $\text{mark}(y) \leq \text{mark}(x_i)$, and that $\forall r = (y_1, \ldots, y_k) \in \text{pre}_U(x_i), r' = (y_1.\text{image}, \ldots, y_k.\text{image}) \in U(G)$. We say that r is *supported* by r'. `Other-Candidates`(x_i) stores in the field `image` of x_i the next candidate unexplored since the last call to `Find-Candidates`(x_i). Both return `true` if they found a result, and `false` otherwise.

In the worst case, BackTrack explores the whole search tree, and has thus the same complexity as the naïve algorithm. However, in practical cases, it often cuts branches of this tree without needing to explore them to the end.

3.3 Types Comparisons

An immediate problem is the cost of type comparisons. Each time we look for a possible candidate for a node x, we compute type comparisons for all relations of $\text{pre}_U(x)$ incident to x. The number of operations performed can thus be expressed (in a simplified way) by $N \times k \times S$, where N is the number of candidates considered in the whole backtracking process, k the maximum size of pre_U sets, and S is the maximum cost of a type comparison. Since both N and S can be exponential (imagine types expressed in a description logics language), it is unwise to multiply these factors.

The solution is easy. We have at most $m_H \times m_G$ different types comparisons (where m_H and m_G respectively denote the number of relations in H and G). So each time one is computed, we can store its result in a table. BT will now run in $(N \times k) + (m_H \times m_G \times S)$, and the two exponential factors are no more multipliers for each other. In usual SG formalisms, S is polynomial (types hierarchies are given by their cover rellation): we can then consider $N \times k$ as the only

```
Data: A support S, two SGs H and G defined on S (H ≠ ∅).
Result: All projections from H into nf(G).

H ← Order(H);
level ← 1;
up ← FALSE;
while (level ≠ 0) do
    if (level = |V(H)| + 1) then
        Solution-Found(H);
        up ← TRUE;
    else current-Node ← V_H[level] ;
    if (up) then
        if (Other-Candidates(current-Node, G)) then
            up ← FALSE;
            level ← Next-Level(current-Node);
        else level ← Previous-Level(current-Node) ;
    else
        if (Find-Candidates(current-Node, G)) then
            level ← Next-Level(current-Node);
        else
            up ← TRUE;
            level ← Previous-Level(current-Node);
```

significant factor, and forget the cost of type comparisons. Another advantage is that conjunctive types are compiled as a whole, and so it avoids to backtrack along multiple relations: conjunctive types are an algorithmic optimization.

3.4 BackMarking

We first extend to hypergraphs a datastructure proposed for BT in [6], not to reduce the size of the search tree, but to check candidates more efficiently.

If $\text{pre}_U(x) = r_0, \ldots, r_k$, and we suppose that r_0, if it exists, is a unary relation, then we provide (when calling `Order`(H)) the node x with k fields noted $\Delta_1(x), \ldots, \Delta_k(x)$. This data structure (the Δs) will be used to incrementally compute sets of candidates. When calling `Find-Candidates`(x), $\Delta_1(x)$ is built by taking all nodes y of G such that, if $\text{image}(x) = y$, r_1 is supported in G, then by removing from this set all nodes such that r_0 is not supported (if it exists). Then the following Δs are computed in the following way:

$$\Delta_{i+1}(x) = \{y \in \Delta_i(x) \mid y = \text{image}(x) \Rightarrow r_{i+1} \text{is supported in } G\}$$

Now `Find-Candidates`(x) computes this set, and Δ_k contains exactly all candidates for x. So the only work for `Other-Candidates` will be to iterate through this set. Let us consider the example in Fig. 1. It represents a step of the BT algorithm where images have been found for x_1 and x_2, and we are computing possible candidates for x. Assuming that all types are pairwise non comparable, check that $\Delta_1(x) = \{y_1, y_3\}$ (r_1 removed nothing, but r_0 removed y_2), and that $\Delta_2 = \{y_3\}$. The only possible candidate for x is y_3.

The complexity of the "natural" candidates research algorithm (Sect. 3.2) is in $\mathcal{O}(n_G^2 \times k)$, where n_G is the number of nodes of G and k the maximum arity of relations. But the worst-case complexity of the Δ based algorithm is

$\mathcal{O}(n_G^3 \times k)$. There are two reasons to claim that this algorithm is still better: 1) we can implement `BackMark`, `Forward-Checking`, and `BackJump` on top of this algorithm at no additional overhead cost, and 2) an average case analysis shows that the natural algorithm still runs in $\mathcal{O}(n_G^2 \times k)$, while the Δ algorithm runs in $\mathcal{O}(n_G \times k \times R)$, where R is a factor decreasing with the density of G (see [6]).

Should we order the relations of $\mathrm{pre}_U(x)$ in a particular way, our algorithm naturally evolves into `BackMark` [11]. Let x be a node of H. We order relations of $\mathrm{pre}_U(x)$ in the following way (it can be done in linear time in the initialization phase, by the `Order` function): for each $r_j \in \mathrm{pre}_U(x)$, we note $\mathrm{last}(x, r_j)$ the greater node (as defined by `Order`) incident to r_j. We order relations of $\mathrm{pre}_U(x)$ in such a way that if $\mathrm{last}(x, r_i)$ is before $\mathrm{last}(x, r_j)$, then r_i is before r_j.

Suppose now that, sometimes during the backtrack, we have built the different $\Delta_i(x)$, and that a failure forced to go up the search tree. What's happening when x becomes again the current node ? Let $(r_1, \ldots, r_k)$ be the non unary relations of $\mathrm{pre}_U(x)$, ordered as indicated above. Then if the other arguments of $r_1, \ldots, r_p$ in $\mathrm{pre}_V(x)$ did not change their images since the last candidates search, then $\Delta_1(x), \ldots, \Delta_p(x)$ do not need to be updated. When calling `Find-Candidate`, we look for the first $x_q \in \mathrm{pre}_V(x)$ that changed since its last call, and build the Δ_i by beginning at Δ_q.

Note that this algorithm has been written at no overhead cost. To evaluate its interest, let us point out that the more "little backtracks" we encounter, the more we gain. And this is precisely the case on difficult instances of the problem.

3.5 Forward Checking(s)

The idea of `Forward Checking` (FC) [14] is to draw earlier the consequences when chosing a candidate. We present a first version,when all relations are binary, that benefits greatly from our Δ data structure. Then, following [8], we point out that naive adaptations to hypergraphs are not efficient enough.

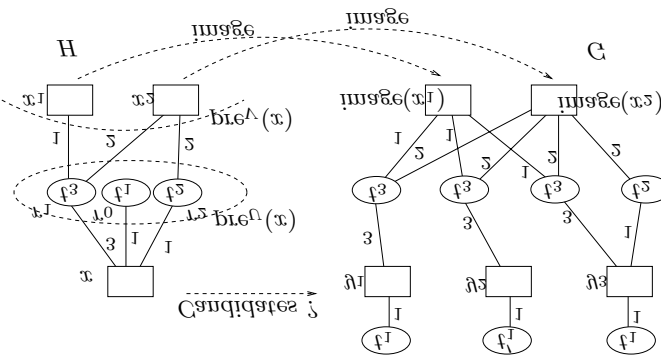

Fig. 1. Searching for possible candidates.

The Binary Case. Consider the example in Fig. 2, where all relations are binary. We have projected the subgraph H containing $\text{pre}_V(x)$, and have extended this partial projection to x (into y). Now it can take an exponentially long time to project the nodes $x_1, \ldots, x_k$. When at least we succeed, we remark that x_{k+1} had no possible candidate, and so this whole work was useless. To avoid this problem, we want the following property to be verified when considering candidates:
(FC): "If a candidate for x is y, then for each neighbour z of $pre_V(x) \cup \{x\}$, the current partial projection can be extended to a projection of z.

Rewriting the functions `Find-Candidates` and `Other-Candidates`, we obtain the algorithm FC, in its binary case, that respects the above property and is sound and complete for projection (note that our implementation, using Δs, naturally includes the `BackMark` algorithm, still at no overhead cost). `Find-Candidates` only puts an iterator at the beginning of the last Δ of x (that already contains all found candidates of x), then calls `Other-Candidates`. `Other-Candidates` advances the iterator, answer FALSE if the whole Δ has been explored, then calls `FC-Propagates`: if the result is TRUE (the property *FC* is verified), `Other-Candidates` also returns TRUE. Otherwise, it calls itself again to examine following candidates. So all checks are done in the function `FC-Popagates`, that looks ahead of the current node. This function considers all relations in $\text{post}_U(x)$, relations incident to x that are not in $\text{pre}_U(x)$. For each of these relations r, linking x to a node z, x contains a pointer to the associated $\Delta_i(z)$ (noted $\Delta'(x, r)$). This structure can be initialized in linear time during the call to `Order`. Then, when called, for each $r \in \text{post}_U(x)$, `FC-Propagates` builds the list $\Delta'(x, r)$, as indicated for the `BackMark` algorithm. If one of these Δ' becomes empty, then `FC-Propagates` stops and returns FALSE, otherwise it returns TRUE (after having correctly built all Δs for the neighbours of x).

We obtain this algorithm just by changing the order in which `BackMark` performs some operations. It has been experimentally proven very efficient.

FC for Hypergraphs However, we are interested in n-ary relations, and the above algorithm only works in the binary case. A natural way is to consider the incidence bipartite of the hypergraph: nodes remain nodes, relations become nodes, and x is the ith node incident to r is translated into an edge labelled i between x and r. This is exactly the drawing of the hypergraph, viewed as a bipartite multigraph (it corresponds to the usual definition of SGs). Then it is sufficient to run the algorithm presented above on this associated binary graph. However, it cannot be considered as a true generalization of FC to hypergraphs: it propagates from (concept) node to (relation) node, then from (relation) node to (concept) node; it looks only half a step ahead. We will call **FC** this first "generalization" of the binary FC. In Fig. 3, when trying to project node A of H into node a of G, this *binary propagation* sees that there will be no problem to project the relation node R. However, looking a bit further should have been better...

To cope with this problem, it has been proposed to propagate at distance 2 in the incidence bipartite. This algorithm has been called **FC+**, but experiments showed it not as efficient as expected. The CSP community began then to be interested in hypergraphs themselves, and not in their incidence bipartite.

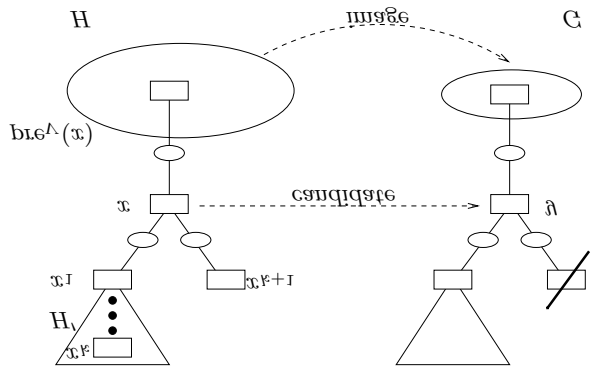

Fig. 2. Looking ahead of the current node.

The first question to answer is *when to propagate* ? As illustrated in Fig. 3, we can either *delay* the propagation (*i.e.* check if a relation is supported as soon as all its arguments save one have been projected), or rely on an *immediate propagation* (*i.e.* check if a relation is supported as soon as one of its aruments has been projected). With *delayed propagation*, we only check if R is supported when A and B have already been projected. Suppose that A has been projected in a, B in c, and we are checking whether C can be projected in e. We deduce that R will not be supported, whatever the image of D. But we should have remarked it sooner. With *immediate propagation*, as soon as we try to project A into a, we see that R will not be supported.

Immediate propagation always leads to smaller research trees. However, cost of propagation must be taken into account, and only experiments (done for CSPs) can show that cuts done in the research tree are worth that cost.

The next questions are *"where to propagate ?"*, and *"how many times ?"*. In the binary case, it is sufficient to propagate once along relations incident to the current node, but it is not the case in hypergraphs, as shown by Fig. 4. Suppose that A has been projected in a and B in g. When we immediately propagated from A, we deduced that (D, E) should only be projected in (b, d), (c, e) or (b, f). Candidates for D are $\{b, c\}$, and those for E are $\{d, e, f\}$. When we immediately propagated from B, we did not restrict these sets. The current node is now C, and we try to project it into h: propagating this choice, we remove f from the candidates of E. Let us check again the supports for the relation incident to A and the one incident to B (note they are not incident to C). The one incident to A does not remove any candidates, but the one incident to B proves that the only images for (D, E) are now (b, e) and (b, d): candidates for D and E are now respectively $\{b\}$ and $\{d, e\}$. Let us check again supports for the relation incident to A: candidates for D and E are now respectively $\{b\}$ and $\{d\}$.

So by propagating along relations non incident to the current node, we have removed more candidates, and more so by checking many times the supports for the same relation. It gives us many choices to implement an hypergraph gener-

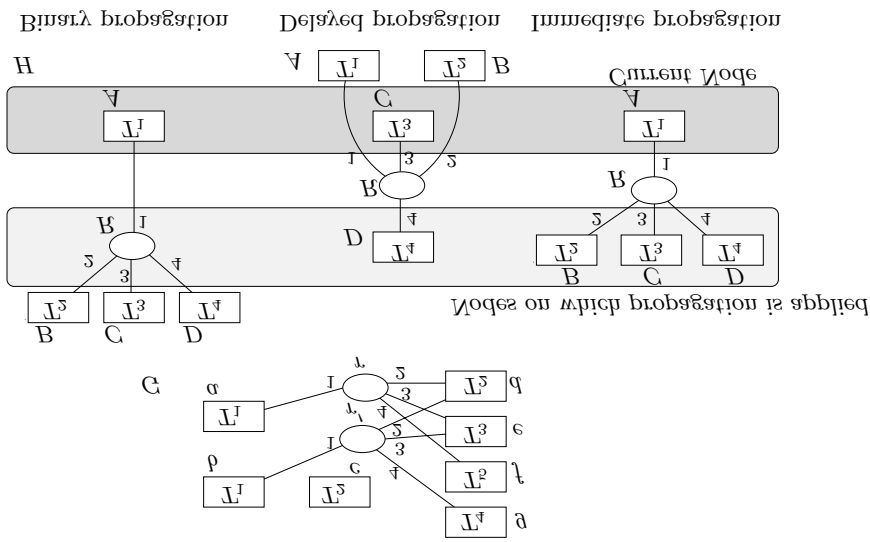

Fig. 3. When will propagation be called?

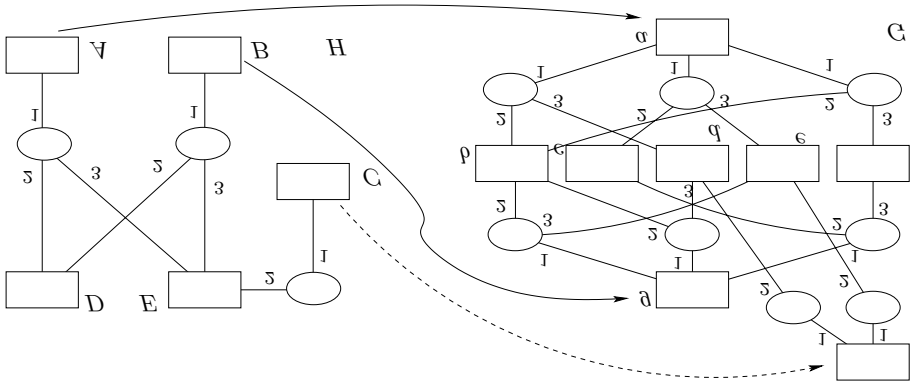

Fig. 4. Propagating along relations non incident to the current node.

alization of FC (though all obtained algorithms are sound and complete, these choices define how much the research tree will be cut). 1) Do we only propagate along relations of $\text{post}_U(x)$, relations incident to x that have an argument in $\text{post}_V(x)$ (we call it *local propagation*), or do we propagate also along relations that have an argument in $\text{pre}_V(x)$ and another in $\text{post}_V(x)$ (we call it a *global propagation*). 2) Do we only propagate once (we call it *in one step*), or repeat propagations as long as one removes some candidates (we call it *in many step*). The following algorithms, implemented and experimented [8] in the CSP community, can be defined with the above criteria:

– **FC** uses the binary FC on the incidence bipartite graph

- **FC+** uses a binary FC, modified to propagate at distance 2, on the incidence bipartite graph (it is equivalent to **nFC1**)
- **nFC0** uses delayed local propagation
- **nFC2** uses immediate local propagation, in one step
- **nFC3** uses immediate local propagation, in many step
- **nFC4** uses immediate global propagation, in one step
- **nFC5** uses immediate lobal propagation, in many step

We say that some propagation algorithm X is *stronger* than a propagation algorithm Y if, considering nodes of H and G in the same order, the research tree generated by X is a subtree of the one generated by Y. Note that, though it means that X generates fewer backtracks than Y, it does not mean that X is more efficient: the cost of propagation can be prohibitive. Fig. **??** compares the theoretical strengths of these different algorithms. Experiments in [8] show that, though there is no clear "best algorithm", there is really worst ones: **FC** and **FC+**. This is a justification for our hypergraphs point of view. Though **nFC5** is the best one when relations strongly constraint our choices (a sparse order on a big set of types, or sparse SGs), **nFC0** is the best when SGs are dense and few relation types are available. A good trade off is **nFC2**, whose efficiency is always satisfying, and can be implemented easily using our Δ data structure [5].

4 Conclusion

We have presented in this paper algorithms developped in the CSP community, that really improve algorithms used today for comparing SGs (by example, [12] uses an algorithm equivalent to **FC+**). This work is not a mere translation: the data structure we propose (the Δs) improves candidate search (in average case) and can be used to implement at the same time `BackMark` and **nFC2** without overhead cost. Moreover, our criteria describing the different FC generalizations to hypergraphs unify the different, unintuitive definitions given in [8].

This work has shown that considering SGs as hypergraphs greatly improved efficiency of projection. Experimental results rely on those done for CSPs [8]. Adding conjunctive types was needed for two reasons: to get rid of the problems encountered when fusioning nodes having different types, and to optimize backtracking process. Finally, though algorithms optimization is not a primary goal of our community, it should be interesting to follow the evolution of CSP algorithms.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. F. Baader, R. Molitor, and S. Tobies. Tractable and Decidable Fragments of Conceptual Graphs. In *Proc. of ICCS'99*, pages 480–493. Springer, 1999.
3. J.-F. Baget. A Simulation of Co-Identity with Rules in Simple and Nested Graphs. In *Proc. of ICCS'99*, pages 442–455. Springer, 1999.

4. J.-F. Baget. Extending the CG Model by Simulations. Number 1867 in Lecture Notes in Artificial Intelligence, pages 277–291. Springer, 2000.
5. J.-F. Baget. *Repr senter des connaissances et raisonner avec des hypergraphes: de la projection la d rivation sous contraintes*. PhD thesis, Un. Montpellier II, 2001.
6. J.-F. Baget and Y. Tognetti. Backtracking through Biconnected Components of a Constraint Graph. In *Proceedings of IJCAI'01*, pages 291–296, Vol. 1, 2001.
7. Peter Becker and Joachim Hereth. A Simplified Data Model for CGs. `http://tockit.sourceforge.net/papers/cgDataModel.pdf`, 2001.
8. C. Bessière, P. Meseguer, E. C. Freuder, and X. Larrosa. On Forward Checking for Non-Binary Constraint Satisfaction. In *Proc. of CP'99*, pages 88–102. Springer, 1999.
9. T. H. Cao. *Foundations of Order-Sorted Fuzzy Set Logic Programming in Predicate Logic and Conceptual Graphs*. PhD thesis, Un. of Queensland, 1999.
10. M. Chein and M.-L. Mugnier. Conceptual Graphs: fundamental notions. *Revue d'Intelligence Artificielle*, 6(4):365–406, 1992.
11. J. Gaschnig. Performance measurement and analysis of certain search algorithms. Research report CMU–CS–79–124, Carnegie-Mellon University, 1979.
12. D. Genest and E. Salvat. A Platform Allowing Typed Nested Graphs: How CoGITo Became CoGITaNT. In *Proc. of ICCS'98*, pages 154–161. Springer, 1998.
13. S. W. Golomb and L. D. Baumert. Backtrack Programming. *Journal of the ACM*, 12:516–524, 1965.
14. R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–314, 1980.
15. P. Hayes. RDF Model Theory. W3c working draft, 2001.
16. G. Kerdiles. Projection: A Unification Procedure for Tableaux in Conceptual Graphs. In *Proc. of TABLEAUX'97*, pages 138–152. Springer, 1997.
17. M.-L. Mugnier. Knowledge Representation and Reasonings Based on Graph Homomorphism. In *Proc. of ICCS'00*, pages 172–192. Springer, 2000.
18. M.-L. Mugnier and M. Chein. Repr senter des connaissances et raisonner avec des graphes. *Revue d'Intelligence Artificielle*, 10(1):7–56, 1996.
19. P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.
20. E. Salvat and M.-L. Mugnier. Sound and Complete Forward and Backward Chainings of Graphs Rules. In *Proc. of ICCS'96*. Springer, 1996.
21. B. Smith. Locating the Phase Transition in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, 81:155–181, 1996.
22. J. F. Sowa. Conceptual Graphs for a Database Interface. *IBM Journal of Research and Development*, 20(4):6–57, 1976.
23. J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading, MA, 1984.
24. M. Wermelinger. Conceptual Graphs anf First-Order Logic. In *Proc. of ICCS'95*. Springer, 1995.