Supplemental Material: The SME algorithm, version 4

This note is a supplement to

Forbus, K., Ferguson, R., Lovett, A., & Gentner, D. Extending SME to Handle Large-Scale Cognitive Modeling. *Cognitive Science*.

It describes the algorithms in Version 4 of the Structure Mapping Engine, including the details behind the theoretical complexity analysis whose results were summarized in Section 5 of that paper. We begin with its inputs, outputs, and operations, then provide a step-by-step description of the algorithm, including the computational complexity of each step. Finally, we combine the complexity results for each step to yield the overall complexity.

All references to figures, tables, and papers refer to the main article. References to sections of the paper are marked as such, otherwise section references refer to this document. As in the paper itself, we use Lisp syntax for representations, and infix mathematical syntax for procedures and algorithms, to make them easy to distinguish.

1 Inputs, outputs, and operations

Comparison. The fundamental operation of SME is Comparison. The Comparison operation takes the following inputs:

- *Base:* A description, consisting of a set of statements in a structured representation language.
- *Target*: A description, also consisting of a set of statements in a structured representation language.
- *Output constraints*: These consist of two numeric parameters:
  - The *Output Limit* is the maximum number of mappings that SME should create. An output limit of three, for example, means that SME will never produce more than three mappings. This is its default setting.
  - The *Score Cutoff* enables SME to consider strong competing matches while ignoring small competitors to the best match. The default score cutoff of 0.8, for example, means that mappings whose structural evaluation is more than 20% below that of the best mapping are pruned.
- *Filters*: An optional set of additional constraints that mappings must satisfy. The set of filters allowed is described in Section **Error! Reference source not found.** of the paper.

As discussed in the paper, the output constraints and filters allow task models to automatically tune the matcher. The output constraints implement capacity limits, while filters enable task models to provide additional criteria to guide the matcher. As also described in the paper, the language for describing filters is tightly constrained, since allowing them to be arbitrary computations violates the spirit of structure-mapping.

The Comparison operation produces as output a set of mappings. A mapping contains

- *Correspondences:* The alignment between base and target represented by this mapping, expressed as a set of pairings of items (statements, entities, and predicates) in the base with items in the target.

- *Structural evaluation*: A numerical estimate of the overall quality of the match.
- *Candidate inferences*: A set of analogical inferences suggested by this alignment, including structural evaluations of the degree to which the candidate inferences are supported by and extrapolate from the mapping (Section 4.4 of the paper). Candidate inferences are computed by default from base to target, but reverse candidate inferences, from target to base, can also be computed on demand. Reverse candidate inferences are used when reasoning about differences, for example.

Mappings and candidate inferences are now first-class entities that can be directly referred to systems using SME. For example, a problem solver might have the explicit goal of extending a mapping or verifying a candidate inference. We believe that this is psychologically plausible, given the human ability to reason about analogies. An *analogy ontology* (Forbus et al. 2002) has been defined to enable analogical operations to smoothly interoperate with other kinds of reasoning when needed. However, SME itself only relies on very basic assumptions about the representation system it is being used with. The representation system must identify relations, attributes, and functions, and provide information about their arity. If minimal ascension is to be used, the representation system must also supply superordinate relationships, which can be used by an optional procedure to evaluate it.

To support incremental mapping, SME now provides two additional operations: *Extend* and *Remap*.

Extend. The Extend operation extends the results of a comparison when new items are added to the base or to the target. (To streamline processing, we do not permit items to be removed from either base or target – if items are removed, a match must be restarted.) Correspondences aligning the new items are created, and then the existing mappings are extended with these new correspondences as appropriate.

Remap. Incremental mapping can require backtracking, since misleading early information can lead to mappings that are suboptimal in light of later additional information. The Remap operation reconstructs the mappings from the kernels, providing the same results that would have been found if the current state of the base and target had been available originally. As described below, both Extend and Remap algorithms are carefully organized so as to preserve previously computed results that are still valid, for maximal efficiency. We believe this is psychologically plausible – essentially, the initial stages remain unguided and parallel, while task-specific influences can operate at the later, serial phase of processing.

2 The SME Algorithm, Step by Step

Extend and Remap are almost entirely defined in terms of the same operations as Comparison. Consequently we focus on Comparison and note along the way how Extend and Remap work.

For analyzing complexity, we decompose the phases from Section 2 of the paper further into the following steps:

Phase One: Constructing the Match Hypothesis Network

1. Finding match hypotheses. Local correspondences (match hypotheses) between items in the base and target are proposed in parallel.

Phase Two: Parallel Evaluation of the Match Hypothesis Network

2. Structural consistency filtering. Match hypotheses that violate structural consistency are removed from further consideration.
3. Structural evaluation propagation. Structural estimates of match quality are constructed for each correspondence based on a trickle-down algorithm that provides a local implementation of systematicity.

Phase Three: Constructing Mappings

4. Kernel creation. A kernel is a potential seed of a mapping. Identifying them and scoring them is the first step towards creating a global construal of a match.
5. (Optional) Filtering. Irrelevant kernels are filtered using automatically imposed constraints from task models, using strictly local criteria.
6. Greedy merge. A small number of global mappings are constructed from the kernels.
7. Candidate inference creation and evaluation. Analogical inferences are generated for each mapping and evaluated in structural terms.

We next describe how each step works in detail.

2.1 Finding Match Hypotheses

Match hypotheses are potential correspondences. A match hypothesis links an item in the base to an item in the target. Conceptually, this stage creates, in parallel, match hypotheses between all pairs of items in the base and target that could correspond. (By "item" we mean expressions, entities and functors.) The result is a network out of which mappings are constructed.

The structure of the match hypothesis network is motivated by the constraints of structure-mapping. The parallel connectivity constraint ties the structural consistency of a match hypothesis to the existence of match hypotheses for the corresponding arguments of the statements that it aligns. For any statement S, we use `arguments(S)` to refer to its arguments. Similarly, given an item A, we use `parents(A)` to refer to the set of statements in which it appears as an argument. For example, in

```
S1: (contains (bloodstream animal12) serotonin)
arguments(S1) = {(bloodstream animal12), serotonin}
S1 ∈ parents(serotonin)
```

Parent and argument relationships are similarly defined for match hypotheses, based on the statements they align. That is, MH1 is in arguments(MH2) if the base and target items aligned by MH1 are corresponding arguments in the statements aligned by MH2, and conversely, MH2 is in parents(MH1). For example,

```
S2: (contains (bloodstream animal6) serotonin)
MH1: (bloodstream animal6) ↔ (bloodstream animal12)
MH2: S2 ↔ S1
MH1 ∈ arguments(MH2)
MH2 ∈ parents(MH1)
```

It is also useful to refer to the predicate, function, or connective involved in a statement. We use the

function `functor` for this purpose. Thus

```
contains = functor(s2)
bloodstream = functor((bloodstream animal6))
```

A match hypothesis that has no parents is called a *root* match hypothesis. Similarly, statements in descriptions that are not themselves arguments of another statement are called root statements. We simply use the word "root" when context makes it clear which type we are talking about.

Here is the match hypothesis construction algorithm:

Match Hypothesis Network construction

Inputs: Base `B`, Target `T`, (optional) procedure `locally-alignable?`

Outputs: Network of match hypotheses `MHS`

1. *Initial network construction*: For each `Bi ∈ Expressions(Base)` and `Ti ∈ Expressions(Target)` such that `Functor(Bi) = Functor(Ti) & not(Ubiquitous(Functor(Bi)))`,
   1.1. Create `MH(Bi, Ti)`
   1.2. `Push(MH(Bi,Ti), MHS)`
   1.3. For each corresponding pair of arguments `Bj`, `Tk` in `MH(Bi,Ti)`, `push(<Bj, Tk>, Queue)`
2. *Network growth:* Until `Queue` is empty, process each `<Bj, Tk>` as follows:
   2.1. If `functor(Bj) = functor(Tk) & not(Ubiquitous(Functor(Bi)))`ignore.
   ;; Step 1 already handled this pair, but if ubiquitous, it didn't – it's part of larger structure, so ;; it's now worth binding
   2.2. If `Bj`, `Tk` are both entities, create `MH(Bj,Tk)`, `push(MH(Bj,Tk),MHS)`
   2.3. If either `Bj` or `Tk` is an entity, ignore.
   2.4. If `functor(Bj)` and `functor(Tk)` are functions and *identical-functions* constraint is false,
      2.4.1. Create `MH(Bj,Tk)`, `push(MH(Bj,Tk),MHS)`
      2.4.2. `Push(<Bj,Tk>, Queue)`
   2.5. If `locally-alignable?` is supplied and `locally-alignable?(Bj,Tk,MH(Bi,Ti))`,
      2.5.1. Create `MH(Bj,Tk)`, `push(MH(Bj,Tk),MHS)`
      2.5.2. For each corresponding pair of arguments B_k, T_l in `MH(Bj,Tk)`,
      Push(<B_k, T_l>, Queue)

An initial set of match hypotheses is constructed based on purely local, structural grounds (Step 1), and then extended based on placing arguments of potentially corresponding statements into alignment (Step 2). The contents of the initial set of match hypotheses and the growth of the network are governed by the tiered identicality constraint[1]. Recall that tiered identicality by default requires relations to match identically. The initial set of match hypotheses is created by finding all pairs of statements Bi ∈ Base and Ti ∈ Target such that

`functor(Bi) = functor(Ti)`

and, if the functor is not a ubiquitous predicate, creating a match hypothesis for it.

[1] See Ferguson (2003) for the special case of creating match hypotheses over pairs of commutative expressions, such as matched group or set expressions. In this case, SME delays creating one-to-one matches until resolved by other non-commutative expression matches, during the merge process. It represents the set of potential matches between commutative expressions in a compact matrix called a *commutatives table*.

This set is grown by propagating outward from the initial set, looking for matches between corresponding arguments of the statements aligned in the initial match hypothesis set. Weaker criteria are used for matching when alignment is suggested by other match hypotheses. Statements whose functors are ubiquitous predicates are matched, since the shared parent provides a reason to do so: not including it would violate parallel connectivity. By default, matches between non-identical functions are allowed when they would support a larger match, since these semantically correspond to cross-dimensional differences. Such cross-dimensional matches are not allowed if the identical-functions filter is in force. The optional procedure `locally-alignable?`, if supplied, implements the non-default cases of tiered identicality. For example, if minimal ascension is used, this procedure must use an appropriate knowledge representation system to ascertain if there is a close-enough common superordinate. All of the examples in the paper and the experiments in Section 3 of the paper use strict identicality or minimal ascension.

2.1.1 Complexity of Finding Match Hypotheses

The default test using tiered identicality is identity of functors, which can be considered a unit-time operation. Other techniques like similarity tables (Holyoak and Thagard, 1989) and minimal ascension (Falkenhainer, 1987) satisfy the unit-time operation assumption. We ignore potentially more complex tests here, since they lie outside the spirit of structure-mapping.

Finding the initial set of match hypotheses requires comparing every statement in the base with every statement in the target to see if their functors are identical, and if so, creating a match hypothesis. On a serial machine, this step is bounded above by $O(n^2)$. Since each comparison is independent, they can be done in parallel in unit time if there are at least n^2 processors available. We have found it useful to pre-sort expressions in the base and target into bins by functor, so that we can simply create match hypotheses between expressions in corresponding bins. In the worst case, where every expression had the same functor, this would still be $O(n^2)$, but in the best case, where every statement within the base and target had a different functor, this would reduce to the complexity of the sort, i.e. $O(nlog(n))$.

The filling out of the match hypothesis forest by generating match hypotheses between corresponding arguments (when possible), is a function of the number of match hypotheses found in the initial step and the depth of each tree of arguments. We ignore the depth-related costs for two reasons. First, argument trees are typically much smaller compared to the total number of statements in the description. Second, all match hypotheses between statements with identical functors have already been found in the initial step, so only entities plus statements involving non-identical functions or ubiquitous predicates will cause new match hypotheses to be created. This means the complexity of the filling in the match hypothesis forest is bounded by $O(n^2)$.

2.2 Structural Consistency Filtering

The collection of match hypotheses as generated provides the threads out of which mappings are woven. However, at this stage in processing it is still inchoate. Local application of structure-mapping constraints prunes all match hypotheses that could never be part of a consistent mapping. Such hypotheses are marked as structurally inconsistent by this stage of processing and subsequently ignored.

Recall that parallel connectivity states that the arguments of a pair of aligned statements must also be aligned. Match hypotheses that violate parallel connectivity are said to be *incomplete*. The first part of detecting incomplete match hypotheses occurs during the construction of the match hypothesis forest, since failure during the attempt to align the arguments of two matching statements indicates that that match hypothesis is incomplete. However, parallel connectivity also implies that all parents of that

match hypothesis are also incomplete. This implication is enforced by propagating incompleteness markers upwards to all parents from incomplete match hypotheses once the forest has been finished.

The 1:1 constraint is enforced by propagating information through the argument relations about structural dependencies of match hypotheses. The `descendants` of a match hypothesis is the set of match hypotheses that it structurally depends upon. For example,

```
MH1: (above triangle32 circle6) ↔ (above triangle18 circle3)
MH2: triangle32 ↔ triangle18
MH3: circle6 ↔ circle3
MH4: above ↔ above
descendants(MH1) = {MH2, MH3, MH4}
```

`descendants` is the transitive closure of the `arguments` relation. We use `descendants` to define the set of `nogoods` for a match hypothesis, i.e., those match hypotheses which, combined with it, would lead to a structurally inconsistent result[2].

We define nogoods recursively as follows:

```
nogoods(MH) = { MHi | MH ≠ MHi
                ∧ ([BaseItem(MH) = BaseItem(MHi)
                    ∨ TargetItem(MH) = TargetItem(MHi)]
                   ∨ [∃ MHj (MHj ∈ descendants(MH))
                       ∧ MHi ∈nogoods(MHj)])}
```

That is, two match hypotheses are together structurally inconsistent if either they directly map the same base item to different target items (or the same target item to different base items) or if corresponding descendants do.

Since descendants and nogoods are heavily used in creating mappings, we compute these sets explicitly and cache them with each match hypothesis. Structurally inconsistent match hypotheses are detected during this process, i.e., when the intersection of a match hypotheses' descendants and nogoods is non-empty.

To support incremental operation, the descendants and nogoods sets are recalculated whenever new match hypotheses are added to the forest. (The same calculation is used when a match is started, with every match hypothesis being new.) Perhaps surprisingly, the structural consistency computations are monotonic with respect to the addition of new items to the base and target. That is, the sets of descendants and nogoods can only grow, not shrink. This is easier to see if one remembers that statements can be added but not modified. This means for any statement its arguments remain constant. (Tweaking a representation to improve the match is carried out by adding redundant items to base or target, but these only give rise to new match hypotheses, rather than replacing or mutating existing ones.) Updates occur by propagating upwards from the lowest-order newly added match hypotheses. The descendants are simply the union of the descendants of the arguments, plus the match hypothesis between the corresponding functors (when the match hypothesis is an expression).

[2] The term nogoods is an analogy with truth maintenance systems, in which nogoods are either sets of inconsistent assumptions or clauses that generate such sets (Forbus & de Kleer, 1993).

The nogoods are simply the union of the nogoods of the arguments with the set of match hypotheses that directly conflict with it (i.e., that satisfy the first disjunct in the definition above).

2.2.1 Complexity of Structural Consistency Filtering

Suppose there are *m* match hypotheses. Organizing the propagation step as an iteration that proceeds from entity matches up through the `parents` relations can be done in such a way that each match hypothesis is processed exactly once, using standard tree traversal algorithms, hence this step is *O(m)*. Again we treat set operations as constant-time, since they can be implemented using bit vectors (or run-length encoded bit vectors) to minimize cost.

2.3 Structural Evaluation Propagation

Structural evaluation implements the systematicity preference. Ultimately, structural evaluation scores will be assigned to each mapping by adding up the scores of the match hypotheses that comprise the mapping. Structural evaluation needs to be done early, since its results are used in guiding the greedy merge process described below. Thus as soon as structural inconsistencies have been removed, a numerical propagation step is used to compute scores for each match hypothesis.

The score of a match hypothesis is computed in two parts. First, there is a local component, a starting score given to every match hypothesis. There are two parameters here: `same-functor` is the score given if the match hypothesis involves statements with identical functors or involves entities, and `different-functor` otherwise. The default values for same-functor and different-functor are $5x10^{-4}$ and $2x10^{-4}$ respectively. Note that their exact values do not matter, only their values in relation to each other. The second part of the score is computed by the *trickle-down rule*: Given `MHa` with parents `MHS`,

```
Score(MHa) ← Score(MHa)+ trickle-down * Sum(Scores(MHS)]
```

where `trickle-down` is a constant indicating the strength of the systematicity constraint. The default value for `trickle-down` is 8. Notice that this algorithm results in high scores for entity match hypotheses that support large structurally consistent matches. Thus the global preference for systematicity is computed by a propagation algorithm operating on local evidence.

In some cases, a match hypothesis can receive trickle-down from parents that are structurally inconsistent with each other. For example,

```
MH1: (above triangle32 circle6) ↔ (above triangle18 circle3)
MH2: (above triangle32 circle6) ↔ (above triangle18 square19)
MH3: triangle32 ↔ triangle18
```

Both `MH1` and `MH2` are parents of `MH3`, but a final mapping could not contain both of them. To avoid inflating the value of a match hypothesis during trickle-down, the algorithm greedily selects a match hypothesis' parents, beginning with the highest-scoring parent, and skipping over any parent that is structural inconsistent with parents already selected. It only applies trickle-down from these selected parents (in this example, either `MH1` or `MH2`).

Earlier versions of SME normalized the score of each node to be a maximum of 1.0. This proved to be problematic for large, deeply nested representations, since many of the lower-level nodes would max out to 1.0. Consequently, we eliminated this limit, and now normalize at the level of the mapping, as discussed below.

2.3.1 Complexity of Structural Evaluation Propagation

Local scores are initialized when match hypotheses are created, so the only cost is that of applying the trickle-down rule. On a serial machine, applying trickle-down can be done by iterating over the match hypotheses, starting at the roots of the match hypothesis forest and working downwards. First, suppose each match hypothesis has only a single parent, so greedily selecting among parents is not a factor. Since this only requires processing each match hypothesis once, the complexity is *O(m)*. On a data-parallel machine, assuming at least *m* processors, the time required will be proportional to the maximum depth of expressions being processed. In the worst case this would be *O(m)*, in the (extremely unnatural) case when the base and target were single expressions with extremely deep nesting, i.e.,

```
(P (P (P … (P e)…)))
```

More typically, there is a small integer d that can be found as an upper bound on the depth of expressions, in which case the data-parallel time required will be constant independent of *m*.

Now, consider the case where match hypotheses have multiple parents. Each match's parents must be sorted by score, so that they can be greedily selected for trickle-down. If a match has p parents, this will require *O(plog(p))* time, however, *p* in our experience is always small, so we ignore the cost of this operation.

2.4 Kernel Creation

Kernels form an important intermediate representation in the creation of a mapping. They represent the place where we believe that the shift from parallel processing to serial processing in analogical matching occurs, and where we suspect that penetrability can begin to occur.

A kernel consists of the union of a structurally consistent root match hypothesis with its descendants. The structural evaluation score of a kernel is the sum of the structural evaluation scores for the match hypotheses that comprise it. The nogoods of a kernel is the union of the nogoods of the match hypotheses that comprise it.

Kernels are found by the following algorithm:

1. For each root `R` in the match hypothesis forest,
   a. If `R` is structurally consistent, then create kernel `K` consisting of `R` ∪ `descendants(R)`.
   b. Otherwise, recurse on `arguments(R)`

It is important to note that match hypotheses can be in multiple kernels. Merging kernels with overlapping base structure can lead to candidate inferences, if the base statements corresponding to the match hypothesis roots are not themselves roots of the base description, as Fig. 6 illustrates.

2.4.1 Complexity of Kernel Creation

An extreme upper bound for the complexity of this step is *O(m)*, where *m* is the number of match hypotheses. This could in theory occur when the match hypothesis forest is extremely shallow, consisting of correspondences between distinct unary predicates with an entity as their argument, leading to *m/2* kernels. The best case would be when the base and target were completely isomorphic structures with a single root, with each subexpression having a unique functor. In that case there would be exactly one kernel. In practice the number is somewhere in between, with good matches having a small number of large kernels and poor matches having a lot of small ones.

2.5 Match Filters

The greedy merge algorithm used to combine kernels into mappings provides a good approximation to the best mapping, in terms of structural consistency. We believe that this is both psychologically accurate and useful computationally. However, in cases where task demands impose additional constraints on mappings, a carefully constrained set of filters, automatically constructed by the larger task model, can be provided as one of the inputs to the match process.

Filters work by eliminating kernels from further consideration, weeding them out before they are used in the merge phase. Here is how they operate:

- `(Excluded bi tj)` $\Rightarrow$ remove kernel if it contains a match hypothesis which maps bi to tj.
- `(Required bi tj)` $\Rightarrow$ remove kernel if it contains a match hypothesis which maps bi to some tk, tk $\neq$ tj or if there is a match hypothesis that maps tj to some bl, bl $\neq$ bi.
- `(Identical-functions)` $\Rightarrow$ remove kernel if it contains a match hypothesis that maps a functor F which is a function to some function G, G $\neq$ F.
- `(require-within-partition-correspondences Att1 Att2)` $\Rightarrow$ remove kernel if it contains a match hypothesis that maps an entity with attribute `Att1` to an entity with attribute `Att2`.

It is important to notice that filtering only determines whether or not kernels are considered in the merge phase, i.e., kernels inconsistent with the filters are not destroyed. This provides the ability to rapidly explore alternate interpretations, since the only work that needs to be re-done when exploring alternate constraints is re-doing the merge process itself.

2.5.1 Complexity of Match Filters

All of the tests for filter constraints are strictly local and rely only on structural properties of the expressions themselves, with the exception of the partition constraints. There, category membership is tested by the existence of attribute statements explicitly within the descriptions, which again is a local operation. Thus applying any of these filters to a single kernel can be considered a constant-time operation, so the complexity of applying them to the set of kernels is simply *O(k)*, where k is the number of kernels.

2.6 Greedy Merge

A mapping is a structurally consistent set of correspondences that is maximal, i.e., adding more match hypotheses would make it structurally inconsistent. Importantly, a comparison can have multiple maximal mappings, due to some interpretations of that comparison being structurally inconsistent with each other.

Mappings are created by merging kernels. As Section **Error! Reference source not found.** of the paper outlines, we now use a greedy merge algorithm instead of an exhaustive algorithm, trading guarantees of optimal outputs for efficiency. We believe that this kind of approximation is psychologically plausible.

Our algorithm proceeds in two phases:

1. For each base root that participates in a kernel, greedily merge the kernels that project to it. This step improves the likelihood of candidate inferences by pre-combining kernels that could lead to them.
2. Greedily merge the solutions found for each base root to form a handful of global mappings.

We give the formal description of the entire algorithm below. We start with the crucial core algorithm combining partial mappings, which we call `GreedyMerge`. Greedy algorithms combine local solutions to form global solutions. They require a notion of solution quality that can be used to impose a preference ordering on local solutions. For constructing interpretations of a match, the local solutions are the kernels and the quality metric is their structural evaluation scores. The core `GreedyMerge` algorithm is:

Algorithm: `GreedyMerge`
Input: A set of partial mappings `PMAPS`, a score cutoff `S`, and a maximum number of desired interpretations `N`
Output: Up to `N` combined mappings, `MAPPINGS`

1. Sort `PMAPS` into a list in descending order, based on their structural evaluation scores.
2. `INTERPS` ← {}; `MAX` ← 0
3. Until `PMAPS` = {}
   3.1. `INTERP` ← pop(`PMAPS`)
   3.2. For each `K` in `PMAPS`,
      3.2.1.If ¬`Nogood(INTERP,K)` then
         3.2.1.1. `INTERP` ← `INTERP` ∪ {K}
         3.2.1.2. `PMAPS`←`PMAPS` – K
   3.3. For each `INTERP-B` in `INTERPS`,
      3.3.1.For each `K` in `INTERP-B`,
         3.3.1.1. If ¬`Nogood(INTERP,K)` then
            3.3.1.1.1. `INTERP` ← `INTERP` ∪ {K}
   3.4. If `score(INTERP)` > `MAX` then `MAX` ←`score(INTERP)`
   3.5. `If score(INTERP)` < `S*MAX` then go to 4.
   3.6. `INTERPS` ←`INTERPS` ∪ {`INTERP`}
   3.7. If |`INTERP`| = `N` then go to 4.
4. `Mappings` ← `map(CreateMapping,INTERPS)`

Note that step 3 allows an interpretation to also include kernels from previously-found interpretations. This step is solely used in the second phase, in which global mappings are discovered.

Algorithm: `CreateMapping`
Input: An interpretation `INTERP`, consisting of a set of partial mappings
Output: A mapping `M`
1. Let `M` be a new mapping
2. `Correspondences(M)` ←apply(∪,map(`correspondences`, `INTERP`))
3. `Score(M)` ← apply(+,map(`score`,`Correspondences(M)`))
4. `CandidateInferences(M)` ← `FindCandidateInferences(M)`

Finding candidate inferences is discussed in the next section.

Intuitively, the `GreedyMerge` algorithm selects the largest partial mapping and merges into it everything that is structurally consistent. Each partial mapping added to the interpretation can rule out others, since it imposes new structural consistency constraints. (The nogoods for a set of partial mappings is simply the union of the nogoods for the partial mappings.) By starting with the largest we improve our chances of getting the best solution. By starting subsequent solutions with the largest remaining partial mapping, we improve our chances of getting a different yet still good solution, since to

be still available it must be structurally inconsistent with earlier solutions. We view the ability to generate multiple interpretations of an analogy as critical. Even with a firm goal in mind, there can still be several ways to interpret an analogy (e.g. the Contras example in Holyoak and Thagard (1989)).

With the `GreedyMerge` algorithm in hand, now we can define `GreedyMap`:

Algorithm: `GreedyMap`
Inputs: A list of kernels `KERNELS`, a set of match constraints `CC`, a score cutoff `S`, and a maximum number of desired interpretations `N`
Outputs: Up to `N` global mappings, `MAPPINGS`
1. Let `BASE-PARTITIONS` = `CalculateBasePartitions(KERNELS)`.
2. Let `CANDIDATES` = apply(∪, map(`GreedyMerge`, `BASE-PARTITIONS`)).
3. `GreedyWeave({},CANDIDATES, S, N)`.

`CalculateBasePartitions` involves sorting the kernels into equivalences classes according to what base root(s) they project onto, and performing `GreedyMerge` within each equivalence class. This step is useful because candidate inferences arise from common base structure, hence pre-merging kernels that project onto the same base increases the likelihood of good candidate inferences.

`GreedyWeave` simply calls `GreedyMerge` on each of the candidates in turn. The process stops when either `N` solutions are generated or a new solution drops below the score cutoff of the previous solution.

Recall that `N`, the maximum number and `S`, the score cutoff, are psychologically motivated. The limited number of mappings that can be produced respects constraints on memory resources, and the score cutoff implements the intuition that an overwhelmingly better mapping swamps consideration of any alternatives. The default value for `N` is 3, and the default value for `S` is 0.8.

2.6.1 Complexity of Greedy Merge

Let us begin with analyzing the `GreedyMerge` algorithm, since it is at the core of the process. The first step, sorting the kernels, is $O(k\log(k))$ in the number of kernels k. Each mapping is found in linear time $O(k)$ because it requires considering first the unused kernels and then the kernels in previous mappings. At most, N mappings are constructed (where $N = 3$ by default). Therefore, the overall complexity is $O(k\log(k))$. This is assuming that the cost of structural consistency tests can be ignored, which is reasonable given fast set intersection/union techniques involving bit-vectors.

Recall that the number of kernels k is worst-case $O(n^2)$ in the size of the base and target. In terms of base and target sizes, then, the complexity of `GreedyMerge` is thus worst case $O(n^2\log(n^2))$, i.e., $O(n^2\log(n))$. In practice the number of kernels is typically much smaller, since the worst-case presumes that every statement is independent. Rich, structured representations with substantial overlap tend to provide fewer and larger kernels, leading to better performance in such situations. This is unlike many match algorithms, where matching larger structures always leads to worse performance.

`CalculateBasePartitions` involves doing a `GreedyMerge` step for each of the base roots. The number of base roots as a function of the size of the base can range anywhere from 1 to $n/2$, the former when the entire base is a coherent focused argument, and the latter where the base consists of a disconnected set of entities, each with a single attribute known about it. (It can never be larger than $n/2$ because entities without any attribute or relational information will not participate in any match hypotheses, and hence will be irrelevant for the algorithm.) Therefore in the worst case, the complexity of `CalculateBasePartitions` is $O(n^2\log(n))$ in the case of isolated entities, with typical case

complexity being much lower than that. Again, with this algorithm, growth in base and target sizes does not always result in growth in processing time or memory – it can actually decrease if the growth makes the relational structures more connected!

Given the tight bounds imposed by N and S, the complexity of GreedyWeave is simply that of GreedyMerge, $O(n^2 log(n))$, since the number of times it is executed depends on them instead of the sizes of the base and target. The worst-case complexity for the GreedyMap step is simply the complexity of its most expensive step, CalculateBasePartitions, and hence is $O(n^2 log(n))$.

2.7 Generating Candidate Inferences

The algorithm for computing candidate inferences is:

Algorithm: FindCandidateInferences
Input: A mapping M
Output: the set of candidate inferences CandidateInferences(M)

1. CandidateInferences(M) ← {}
2. For each R ∈ Roots(Base(M)),
   2.1. When ∃ MH ∈ Correspondences(M) | Root(BaseItem(MH)) = R
   ∧ ¬∃ MH_2 ∈ Correspondences(M) | BaseItem(MH_2)=R
      2.1.1. CandidateInferences(M) ← CandidateInferences(M)
      ∪ ConstructCI(R, M, {})
3. For each CI ∈ CandidateInferences(M), CIStrucuturalEvaluation(CI)

That is, each root expression of the base that intersects the subset of the base that is mapped by M but is not already included in the correspondence gives rise to a candidate inference. Reverse candidate inferences are computed via the same algorithm, using the target as the starting point instead of the base.

There is a subtle issue here concerning how much overlap between the base and the mapping is needed to suggest a candidate inference. The most conservative criterion requires overlapping statements, the most liberal criterion requires only overlapping entities. The conservative criterion limits candidate inferences to filling in causal, inferential, or other higher-order structure involving overlapping statements. The liberal criterion enables candidate inferences to import whole new structures into the target, based on entity overlaps established by other parts of the base. Given the need to evaluate candidate inferences in any case, the default mode of operation is the liberal criterion. However, SME includes a switch for enforcing the conservative criterion, which is implemented by an extra condition in line 2.1 above.

Recall that candidate inferences can introduce new entities into the target, called *skolem entities*. The ConstructCI algorithm must do a tree walk through the base expression, introducing such skolems as necessary. We say that a base item B is mapped in mapping M if there is some match hypothesis in the correspondences of M which has B as its base item. If B is mapped, then its correspondent is the target item for the match hypothesis which mentions B. Similarly, if a target item T is mapped in M, then its correspondent is the base item for the match hypothesis that involves T. (That correspondent, when defined, is a function follows directly from the 1:1 constraint of structure-mapping.) Thus we must introduce skolems for each entity in the base expression that does not have a correspondent. Moreover, we must introduce the same skolem for each occurrence of the entity in the base expression,

since an entity can occur multiple times in the same expression. This is what makes ConstructCI a bit complex, since it must maintain a table of bindings.

Algorithm: ConstructCI
Input: A base item B, a mapping M, and a set of bindings Skolems
Outputs: An expression representing a candidate inference and a set of skolem entities for base items that have no correspondent in the target.

1. If B is an entity,
   1.1. If Mapped(B,M) then return Correspondent(B,M) and Skolems
   1.2. If lookup(B,Skolems) then return value(lookup(B,Skolems)) and Skolems
   1.3. Let Sk(B) be a new skolem constant. Return Sk(b) and Skolems ∪ Bind(B,Sk(B))
2. If B is a functor, if Mapped(B,M) then return Correspondent(B,M) and Skolems, otherwise return B and Skolems
3. Otherwise B is an expression.
   3.1. If Mapped(B,M) then return Correspondent(B,M) and Skolems
   3.2. Otherwise, let cargs = empty list
      3.2.1. Let cfunctor, skolems = ConstructCI(functor(B),M, skolems)
      3.2.2. For each A ∈ arguments(B),
         3.2.2.1. Let carg, newskolems = ConstructCI(A,M,skolems)
         3.2.2.2. Let cargs = cargs ∪ {carg}
         3.2.2.3. Let skolems = newskolems
      3.2.3. Return MakeExpression(cfunctor, cargs)

A subtle issue in ConstructCI is that it assumes that functors lying outside the mapping should be brought over intact. This design choice reflects our intuition that one purpose of analogical matching is to help regularize, and thus extend, one's knowledge. The alternative would be to always create a skolem constant for the functor, and then attempt to replace it with the functor from the base as a separate step. Since candidate inferences always need to be checked for validity in any case, inappropriate carryover of functors will be detected during this process anyway. Our choice to carry them over intact biases the process towards accepting the carryover by default.

The structural evaluation algorithm for computing support and extrapolation scores (CIStructuralEvaluation above) is a variation of the algorithm used for mappings. To compute the support score of a candidate inference, the initial bias plus trickle-down algorithm is executed on just the subset of the correspondences that support it in the mapping and adding up the results.

The extrapolation score of an analogical inference is, roughly, the size of the new information over the total size of the inference. Consider two limiting cases, neither of which can actually occur. If there were no support (i.e., a hallucination), all the information would be new, so the extrapolation score would be 1. If there were nothing new (everything was there already), then the score would be 0. Any real candidate inference will be somewhere in between these two values.

The algorithm for computing extrapolation scores is

1. Apply the trickle-down algorithm to the structure of the inference itself, i.e., as if we were matching the inference to itself

2. The extrapolation score is

$$\frac{score\,(Outside\,)}{score\,(Outside\,) + score\,(Inside\,)}$$

where Inside refers to the items in the candidate inference that are part of the mapping and Outside refers to the items in the candidate inference that are being projected. Using the trickle-down algorithm provides a more conservative score than simply counting items would, since the existence of large structures outside the mapping will lead to higher scores inside the mapping due to trickle-down.

2.7.1 Complexity of Candidate Inference Generation

Since the size of statements is typically small compared to the number of statements in the descriptions, we ignore all variability in cost of recursive traversal of statements, treating it as a constant, and focus instead on the number of candidate inferences there can be, since that can vary as a function of the size of the input descriptions. Because of structural consistency, each base root can participate in at most one candidate inference per mapping. Consequently the growth of candidate inferences is bounded by *O(n)*.

2.8 Extending and Remapping

The intuition behind incremental mapping is that normally people first try to incorporate new information into an ongoing mapping (`Extend`), but that they can reinterpret the analogy if necessary (`Remap`).

Extending a mapping occurs when new information is added to the base or target of a match. Basically, the new information is matched against the other representation, leading to new match hypotheses and new kernels. Notice that since the information is new by assumption, there cannot already be any correspondences pertaining to it in the match. Therefore either some new kernels will be formed, or the new information does not match at all to the other description in its current state. New kernels are then added to existing mapping(s) if they are structurally consistent with them. This can lead to the elimination of candidate inferences, if the new information is about the target and "fills in" the missing structure.

The worst-case complexity of extending a mapping is the same as the Comparison algorithm, since in the worst case one is adding all of the information to an empty base and target. The typical case complexity is of course much lower, since adding one new item to the base (or target) only requires checking it against all of the items in the target (or base), not re-checking any previous base (target) statements.

The `Remap` algorithm simply destroys the existing mappings and re-performs the Greedy Merge algorithm based on the full set of kernels from scratch. Thus the complexity of the Remap operation is simply that of the Greedy Merge algorithm. Psychologically, we believe that the remapping criteria people use is task-specific. Consequently, SME does not automatically remap: The decision to remap must be taken by some external model or system. To help external systems make such decisions SME does provide an estimate of what fraction of the total possible structural evaluation the current mappings represent. When this fraction gets low, it suggests that remapping might lead to a better global interpretation.

2.8.1 Complexity of Extending a Match and Remapping

Extending a match with new items in the base and/or target requires extending the match hypothesis forest and adding the kernels (if any) to the existing mappings. The candidate inferences for the mappings need to be recomputed. This is clearly bounded above by the complexity of computing the

match from scratch, although in practice it is typically far less. Remapping simply re-runs `GreedyMap`, which is $O(n^2 log(n))$, as per the analysis above.

3 Complexity of the SME algorithm

If, as we believe, comparison comprises one of the core processes of cognition, then it is crucial for its computational complexity to be low. SME's computational complexity is in fact quite low. Recall that if the number of items in the base and target is n, the number of kernels k is bounded by n^2. The results of the complexity analysis can be summarized as follows:

Operation	Worst-case time, serial processing
Finding match hypotheses	$O(n^2)$
Structural consistency filtering	$O(n^2)$
Structural evaluation propagation	$O(n^2)$
Kernel creation	$O(n^2)$
Filtering	$O(n^2)$
Greedy merge	$O(n^2 log(n))$
Candidate inference construction	$O(n)$
Extending/Remapping	$O(n^2 log(n))$

Thus the SME algorithm is worst-case $O(n^2 log(n))$ on a serial processor. Most of SME's processing can be done in parallel, as our analyses of individual steps noted. Assuming a data-parallel machine with at least n^2 processing elements to handle match hypothesis networks, the processing for the overall algorithm would be between log and linear, depending on specific assumptions about the parallel architecture.