The SDMLib solution to the Class Responsibility Assignment Case for TTC2016

Christoph Eickhoff, Lennert Raesch, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

raesch|christoph|zuendorf@uni-kassel.de

This paper describes the SDMLib solution to the Class Responsibility Assignment Case for TTC2016. SDMLib provides reachability graph computation ala Groove. Thus, the simple idea was to provide rules for possible clustering operations and then use the reachability graph computation to generate all possible clusterings. Then, we apply the CRAIndex computation to each generated clustering and identify the best clustering. Of course, this runs into scalability problems, very soon. Thus, we extended our reachability graph computation to do an A* based search space exploration. Therefore, we passed the CRAIndex computation as a metric to our reachability graph computation and in each step, we consider the set of not yet expanded graphs and choose the one, that has the best metric value for expansion. The paper reports about the results we achieved with this approach.

1 Introduction

This paper describes the SDMLib solution to the Class Responsibility Assignment Case for TTC2016 [1]. SDMLib provides reachability graph computation ala Groove [2]. For a given start graph and a given set of rules, the reachability graph computation generates all graphs that may be derived from the start graph by applying all rules at all possible matches as often as possible in all possible orders. Each time a new graph is computed, we search through the set of already computed graphs for an already known isomorphic graph. As proposed by [2], SDMLib computes node and graph certificates which are then used as hash keys to access potentially isomorphic graphs efficiently. The node certificates then also help to do the actual isomorphism test. If a new graph has been generated, we create a so-called reachable state node and we connect the reachable state node of the predecessor graph with the reachable state node for the new graph via a rule application edge labeled with the name of the rule used. In addition, a root node of the graph is attached to the reachable state node. Altogether, the generated reachability graph has a top layer consisting of reachable state nodes connected via rule application edges and each reachable state node refers to the corresponding application graph via a `graphRoot` link. In SDMLib, this whole structure is again a graph, and graph rules may be applied to it in order to find e.g. reachable states with a maximal metric value for the attached application graph or to find states where all successor states have lower metric values or to find the shortest path leading to the best state. Actually, any graph related algorithm may be deployed.

The Class Responsibility Assignment Case challenges the rule orchestration mechanisms provided by the different model transformation approaches. Thus, our solution uses the SDMLib reachability graph computation for rule orchestration. This is a very simple way to apply all rules in all possible ways and in addition we are able to investigate all intermediate results in order to identify which paths through the search space are the most interesting ones. The drawback of this approach is that it wastes a lot of runtime and memory space for copying the whole class model graph each time a rule is applied and for the search of already known isomorphic copies of the generated graphs. As shown in the case description, the number

Submitted to:
TTC 2016

of possible clusterings grows with the Bell number, i.e. for larger examples a complete enumeration of all possible clustering is not possible in a meaningful time. As only a small fraction of the search space can be explored, it might be helpful to be able to investigate all intermediate states to identify the most promising spots for further expansion. Thus, we hope that the flexibility provided by the SDMLib reachability graphs to investigate different intermediate states pays off, in the end.

As it is usually not possible to generate the whole reachability graph for a given example, our reachability graph computation may be restricted to a maximum number of reachable states to be considered. Next, we have extended our reachability graph computation with an A* like search space exploration that takes a metric as parameter and at each step chooses the state with the best metric value for expansion. We have developed two variants of this A* algorithm which will be discussed below.

The next section introduces the rules we use to solve the Class Responsibility Assignment Case and then Section 3 shows the different search strategies we use in this example. Finally, 4 shows our performance measurements. In the last section we sum up our results.

2 The Model Transformation Rules

Our feature clustering approach uses three SDMLib model transformation rules. In the preparation phase we use the rule shown in Figure 1 to create one class for each feature in our class model.

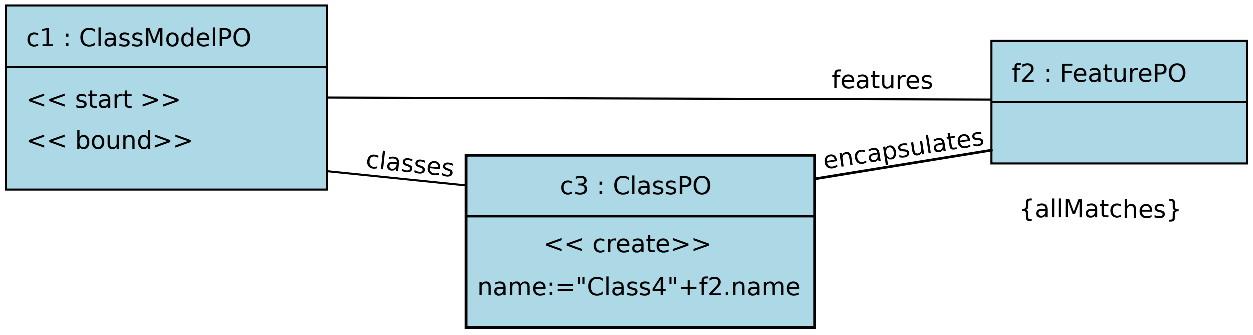

Figure 1: Rule adding initial classes

This rule starts by matching the pattern object `c1` to the `ClassModel` object passed as parameter. Then, `f2` is matched to a Feature object attached to this `ClassModel` object. The {`allMatches`} constraint causes the rule to be applied to all possible matches. Thus, for each `Feature` object in our current `ClassModel`, the <<`create`>> sterotype on pattern object `c3` causes the creation of a new `Class` object. In addition the new `Class` object is attached to the `ClassModel` via a `classes` link and to the `Feature` object via an `encapsulates` link. Finally, the new `Class` object's `name` attribute gets assigned the concatenation of the prefix `"Class4"` and the `name` of the current `Feature`. Thus, after the execution of this rule, each feature has its own class containing just this feature. This class model is then used as starting point for the repetitive application of our clustering rules.

We use two different clustering rules, one for clustering along data dependencies and one for clustering along functional dependencies. Figure 2 shows the data dependency clustering rule `MergeDataDep`:

The matching of this rule starts with the `ClassModel` object which is bound to `c1` at rule invocation. Then we follow a `classes` edge to find a match for `c2`, i.e. a `Class` object in our `ClassModel`. Next, we follow an `encapsulates` edge to match a `Method` object `m4` contained in `c2`. The object matched by `m4` must

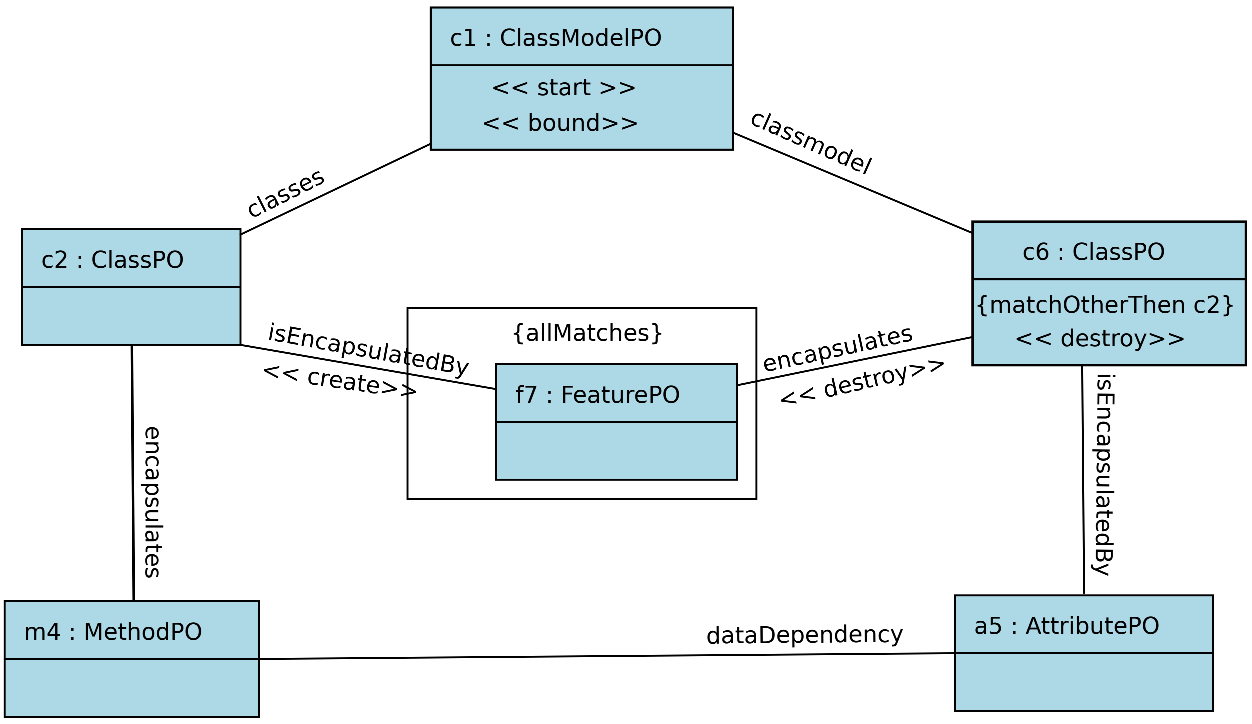

Figure 2: Merging Classes via Attribute Dependencies

have a `dataDependency` edge to an `Attribute` object matched by the pattern object `a5`. This `Attribute` object in turn must be contained in a `Class` matched by `c6`. By default, SDMLib allows homomorphic matches, thus `c2` and `c6` would be allowed to match the same `Class` object. Via the {`matchOtherThen c2`} clause, we enforce isomorphic matching, i.e. `c2` and `c6` must match two different `Class` objects. Finally, the Class matched by `c6` must belong to our `ClassModel c1`. When such a match is found, the subpattern containing the `FeaturePO` pattern object `f7` is executed on all possible matches. Pattern object `f7` matches for all features contained in the `Class` matched by `c6`. (Note, `f7` exploits homomorphic matching and will also match the `Attribute` object already matched by `a5`.) For each `Feature` object, the `encapsulates` edge connecting it to the `Class` matched by `c6` is deleted and a new `isEncapsulatedBy` edge connecting it to the `Class` matched by `c2` is created. After transferring all features to the `Class` matched by `c2`, the `Class` matched by `c6` is destroyed. Thus, the rule shown in Figure 2 merges two classes that are connected via a `dataDependency` into one class.

Figure 3 shows our second clustering rule `MergeFuncDep`. It works similar to rule `MergeDataDep` but it applies to a pair of classes that is connected via a `functionalDependency`.

Our two clustering rules merge classes only if there is a dependency between them. This already utilizes application specific knowledge about our CRAIndex metric. Our search space expansion will start with classes containing only one feature each. Merging classes without a dependency between them is not going to improve the CRAIndex of the resulting class model. Merging classes has the potential to improve the CRAIndex only if the classes contain features that depend on each other. Thus, our clustering rules are already optimized for the optimization of the CRAIndex. Using a different metric would perhabs require a more general clustering strategy. As the metric is evaluated during the search space exploration, it would be

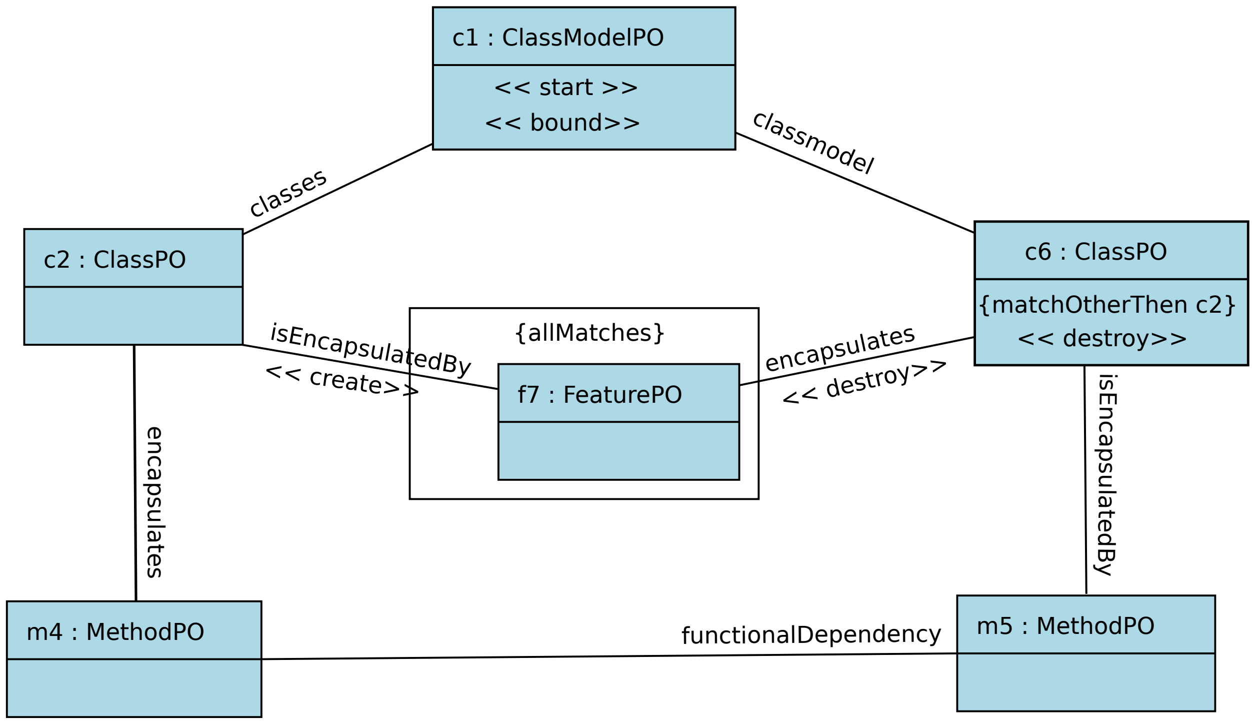

Figure 3: Merging Classes via Method Dependencies

easy to simply merge any two classes, as any graphs resulting from applying a non metric improving rule would immediately be dismissed anyways.

3 The Search Space Expansion Mechanisms

At the beginning, a `ReachbilityGraph` object is initialized with a start graph or `startState` and with a set of `rules` that shall be applied to the different reachable states. Our standard reachability graph computation algorithm is shown in Listing 1. For the standard reachability graph computation, we call explore(depth) where depth is the maximal number of states to be generated. This is used e.g. to terminate the search in case of an unlimited search spaces. First, our expansion algorithm initializes its `todo` list with the `startState` and adds the `startState` to a hash table of reachable `states` where a graph `certificate` is used as key as proposed by [2]. Then, we line 5 loops through the `todo` list until it drains or the maximal `depth` of reachable `states` is reached. Each time, line 6 removes the first element of the `todo` list and chooses it as `current` state. Then line 7 and line 8 iterate through all `rules` and all matches. For each match, we `clone()` the `current` state and apply the rule changes to that clone, resulting in a `newState`. As the `newState` may have been created by other rule applications already, line 10 tries to `find` an `isOldState`, i.e. the `find` operation computes the `certificate` of the `newState` and tries to look it up in the `states` hash table. This may involve a isomorphism check in case of accidentally matching certificates. If no `isOldState` is found, line 12 adds the `newState` to the hash table of reachable `states`, line 13 adds an edge labeled with the applied rule from the `current` state to the `newState`, and line 14 adds the `newState` to our `todo` list. If there is an `isOldState` line 16 just adds an edge from the `current` state to the isoOldState.

```
ReachabilityGraph::explore(depth) {
  todo = new ArrayList();
  todo.add(this.startState);
  states.put(certificate(this.startState), startState);
  while (! todo.isEmpty() && states.size() <= depth) {
    current = todo.get(0); todo.remove(0);
    for(Rule r : this.rules) {
      while (r.findMatch()) {
        newState = current.clone().apply(r);
        isoOldState = find(states, newState);
        if (isoOldState == null){
          states.put(certificate(newState), newState);
          addEdge(current, r, newState);
          todo.add(newState);
        } else {
          addEdge(current, r, isoOldState);
        }
      }
    }
  }
}
```

Listing 1: General Reachability Graph Computation

Our old reachability graph algorithm shown in Listing 1 removes elements from the beginning of its todo list and adds new elements to the end of the todo list. This results in some kind of *breadth first* search strategy.

In many examples we have considered so far, the given rules do not extend the given graph but they only mark certain situations or move elements from one place to another. In such cases the number of different graphs that may be generated is finite and thus the complete reachability graph may be generated. Then you just call our explore method with the maximal Long value. The rules shown in Section 2 reduce the number of classes by one on each rule application. Thus, the number of rule applications is also finite and we might be able to compute the full reachability graph. However, the computation slows down dramatically as soon as the memory used by our reachability graph exceeds the computers physical main memory size (16 giga byte for our test computer). For the larger clustering example E this size is reached by about 25000 reachable states which is only a fraction of the overall possible states. When you cannot search through the whole state space, the breadth first search strategy of our standard algorithm will most likely not reach the interesting parts of the state space but it visits *"early"* states, only. To adjust for a broader use and examples like the current challenge, we have implemented three additional exploration algorithms. It is therefor now possible to choose from the three different modes and still pass the depth of the expansion, offering great flexibility for search space exploration. The three modes are the following:

3.1 Default A$*$ like search space exploration

To enable an A$*$ like search space expansion strategy Listing 2 extends our explore method with a metric parameter. At call time we pass a concrete metric e.g. as Java 8 lambda expression.

```
rg.explore(25000, g -> CRAIndexCalculator.calculateCRAIndex((ClassModel) g));
```

The new explore method just `sorts` the `todo` list in line 6 before choosing a new element. Thereby, each step considers the state with the best CRAIndex for further expansion resulting in a depth first like expansion strategy.

```
1  ReachabilityGraph::explore(depth, metric) {
2     todo = new ArrayList();
3     todo.add(this.startState);
4     states.put(certificate(this.startState), startState);
5     while (! todo.isEmpty() && states.size() <= depth) {
6        sort(todo, metric);
7        current = todo.get(0); todo.remove(0);
8        for(Rule r : this.rules) {
9           while (r.findMatch()) {
10             newState = current.clone().apply(r);
11             isoOldState = find(states, newState);
12             if (isoOldState == null){
13                states.put(certificate(newState), newState);
14                addEdge(current, r, newState);
15                todo.add(newState);
16             } else {
17                addEdge(current, r, isoOldState);
18             }
19          }
20       }
21    }
22 }
```

Listing 2: Default A* based Search Space Expansion

3.2 Metric-based ”Ignore Decline” Mode

Our $A*$ expansion strategy in each step still generates all successors of the `current` state. For example E there are about 400 dependencies, thus, the initial state has about 400 successor states. While this number decreases by one with each rule application, the first 100 rule applications have have 350 successors on average resulting in 35000 states, which already exceeds our memory space. To improve this, Listing 3 improves our $A*$ algorithm by comparing the metric value of the `newState` with the current `bestMetric`. If the metric of the `newState` is lower then the `bestMetric` we ignore the `newState`, cf. line 13, and we do not add it to our reachability graph nor to our `todo` list. Thus, the metric passed to the explore function is evaluated for all graphs. The list of graphs to still be explored is also ordered based on the metric, thus graphs with a better metric are explored earlier on. However we ignore all those graphs that decline our current result.

```
1  ReachabilityGraph::explore(depth, metric) {
2     todo = new ArrayList();
3     todo.add(this.startState);
4     states.put(certificate(this.startState), startState);
```

```
 5     bestMetric = metric(this.startState);
 6     while (!todo.isEmpty() && states.size() <= depth) {
 7        sort(todo, metric);
 8        current = todo.get(0); todo.remove(0);
 9        for(Rule r : this.rules) {
10           while (r.findMatch()) {
11              newState = current.clone().apply(r);
12              if(metric(newState) < bestMetric){
13                 continue;
14              }else{
15                 bestMetric = metric(newState);
16              }
17              isoOldState = find(states, newState);
18              if (isoOldState == null){
19                 states.put(certificate(newState), newState);
20                 addEdge(current, r, newState);
21                 todo.add(newState);
22              } else {
23                 addEdge(current, r, isoOldState);
24              }
25           }
26        }
27     }
28  }
```

Listing 3: Metric based Reachability Graph Computation

3.3 Metric-based ”promote improvement” mode

The second mode promotes any improvements in the applied metric, thus stopping the current expansion step as soon as the metric yields a better result for a newly generated graph. No further rules are applied to the current graph and the discovered graph will now be explored as depicted in Listing 4. When line 16 detects that the `newState` is an improvement compared to the `current` state, line 18 aborts the expansion of the `current` state and we choose the best state of our `todo` list for further expansion. This is again our `newState`, as the old `current` state has been the best state in the previous expansion step and `newState` is even better. In addition, line 17 pushes the old `current` state back into our `todo` list, for later reconsideration.

This algorithm very quickly returns local optimums of the reachability graph. In case it can not further enhance a local optimum it stores the state of expansion on earlier graphs and continues expanding them later on, so it will not only lead to the first local optimum but rather detect a few other, depending on how many steps are necessary to reach them and limited by the depth constraint.

```
 1  ReachabilityGraph::explore(depth, metric) {
 2     todo = new ArrayList();
 3     todo.add(this.startState);
 4     states.put(certificate(this.startState), startState);
 5     improve: while (!todo.isEmpty() && states.size() <= depth) {
```

```
 6        sort(todo, metric);
 7        current = todo.get(0); todo.remove(0);
 8        for(Rule r : this.rules) {
 9           while (r.findMatch()) {
10              newState = current.clone().apply(r);
11              isoOldState = find(states, newState);
12              if (isoOldState == null){
13                 states.put(certificate(newState), newState);
14                 addEdge(current, r, newState);
15                 todo.add(newState);
16                 if(metric(newState) > metric(current){
17                    todo.add(current);
18                    continue improve;
19                 }
20              } else {
21                 addEdge(current, r, isoOldState);
22              }
23           }
24        }
25     }
26  }
```

Listing 4: Metric based depth first Reachability Graph Computation

Note, when the search is exhausted for some state and we go back to earlier states, rule application on those earlier states will first produce the same matches as in earlier runs. These same old matches will be identified by line 11 as `isoOldStates` and thus ignored. However, this requires the computation of a certificate and an isomorphism check. To avoid this effort, our real implementation of the improve expansion stores the number of already created successors for each state and on reconsideration, this number of rule applications is directly ignored.

4 Performance Results

5 Summary

References

[1] M. Fleck, J. Troya, and M. Wimmer. TTC2016 The Class Responsibility Assignment Case. https://github.com/martin-fleck/cra-ttc2016, 2016.

[2] A. Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

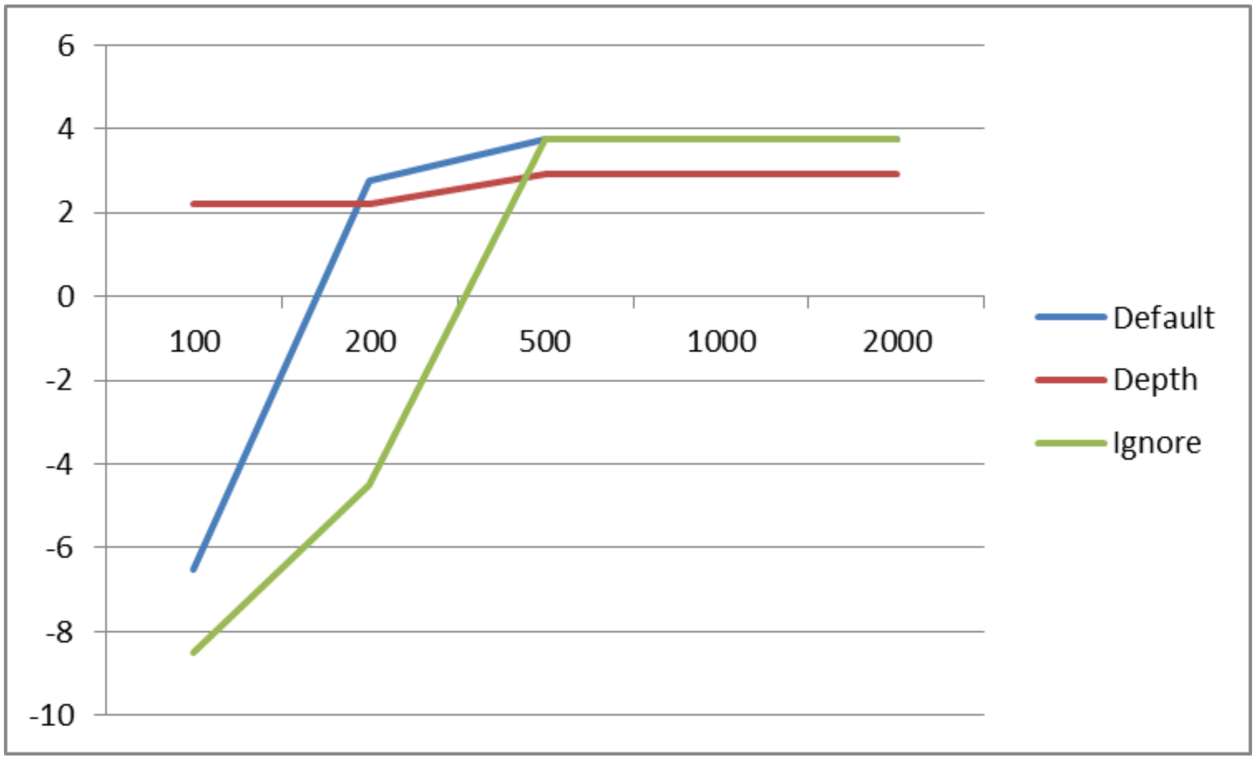

Figure 4: CRAIndex per Search Depth for Example B