Chapter 1

Graph Pair Similarity

Contributed by Justin DeBenedetto

1.1 Introduction

Abstract Meaning Representations (AMRs) are a way of modeling the semantics of a sentence. This graph-based structure focuses on capturing the concepts involved in a sentence and their relations to each other in order to provide a semantic representation. As such, an AMR graph has node labels corresponding to concepts and directed edge labels corresponding to relations between concepts. Both the set of concept labels and the set of edge labels come from PropBank [4]. The design of AMRs provide single-rooted directed acyclic graphs (DAGs). The single root is considered the "focus" of the sentence and is typically the main verb. The direction of edges flow down from the root, such that the parent is considered more of the focus than the children at each step.

An example AMR can be seen in Figure 1.1. The edge labels seen in this example are 'ARG0' which represents the semantic role of agent and 'ARG1' which represents the semantic role of patient. When defining semantic roles, the agent is the entity doing the action of the verb (typically the subject) and the patient is the object of the verb. In this example, six words in the source sentence become four nodes in the AMR graph. The AMR It is common for the number of nodes to be smaller than the number of words in the source sentence for a variety of reasons including 'to believe' being captured by a single concept node and both 'John' and 'him' referring to the same entity, thus sharing a single concept node. While each AMR graph is produced from a single source sentence, many different source sentences may produce the same AMR graph. This is intentional,

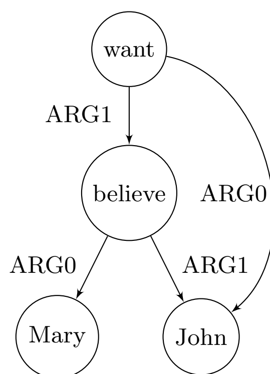

Figure 1.1: Example AMR for the sentence "John wants Mary to believe him."

since the same meaning (captured by the AMR) can be conveyed in various ways (captured by the source sentence(s)).

There are several tasks which are of interest to the community when it comes to working with AMRs. Two of the most central tasks are:

- Generating an AMR from a source sentence
- Generating a sentence from an AMR

Digging further into the task of generating an AMR, since we want to do this automatically, we must have some criteria for what makes a "good" AMR. A common way to approach this problem is to provide humans with source sentences and have them produce the AMRs. Then, train the computer to automatically produce AMRs using the human generated training data. Critical to this process is the ability to score the similarity between two AMR graphs to give the computer a sense of how close its AMR is to the one that we believe to be correct.

When the computer generates candidate AMRs for scoring, it generates multiple candidates at once. Each computer model will have its own method of generating these and assigning some model specific score. The top n candidates according to the computer model will then be re-scored against the correct AMR to determine the models ability to match the correct AMR. This list of candidates is referred to as an **n-best list**. We target scoring these n-best lists throughout this work.

1.2 The Problem as a Graph

AMRs are constructed as graphs as described in Section 1.1 and as shown in Figure 1.1. When processing AMRs, multiple options exist in terms of how to scale or parallelize applications. Since each individual AMR is relatively small (see Section 1.3), it is common to view many AMRs together as a single graph with many distinct connected components. In our task of scoring graph pair similarity, it likely makes more sense to leave each AMR as a separate graph and try to take advantage of the natural parallelism opportunities offered by having distinct graphs.

1.3 Some Realistic Data Sets

The dataset that I am using throughout this work comes from the Linguistic Data Consortium (LDC) and is available for download online[1]. The source sentences are all in English and come from various sources including newswire, discussion forums, and television transcripts. All AMRs are produced by hand by trained linguists and are thus accepted as reliable. Data and statistics presented here come from the general release 1.0 and include $10,312$ AMRs. There is a newer general release which is approximately three times larger which may be used if version 1.0 becomes insufficient.

The average number of nodes for each AMR in our dataset is 17.10 and the average number of edges is 17.07. More than 50% of the AMRs in this dataset are trees. The node and edge count distributions can be seen in Figures 1.2 and 1.3.

Given that our focus in this work is scoring graph pair similarity, we can easily generate artificial but realistic data by adding new nodes and edges drawn from PropBank [4]. These AMRs would not necessarily represent intelligible sentences, and thus are not usable for most AMR purposes, but would be viable AMRs for scoring similarity of larger graphs. To make this artificial data closer

[1] https://amr.isi.edu/download.html

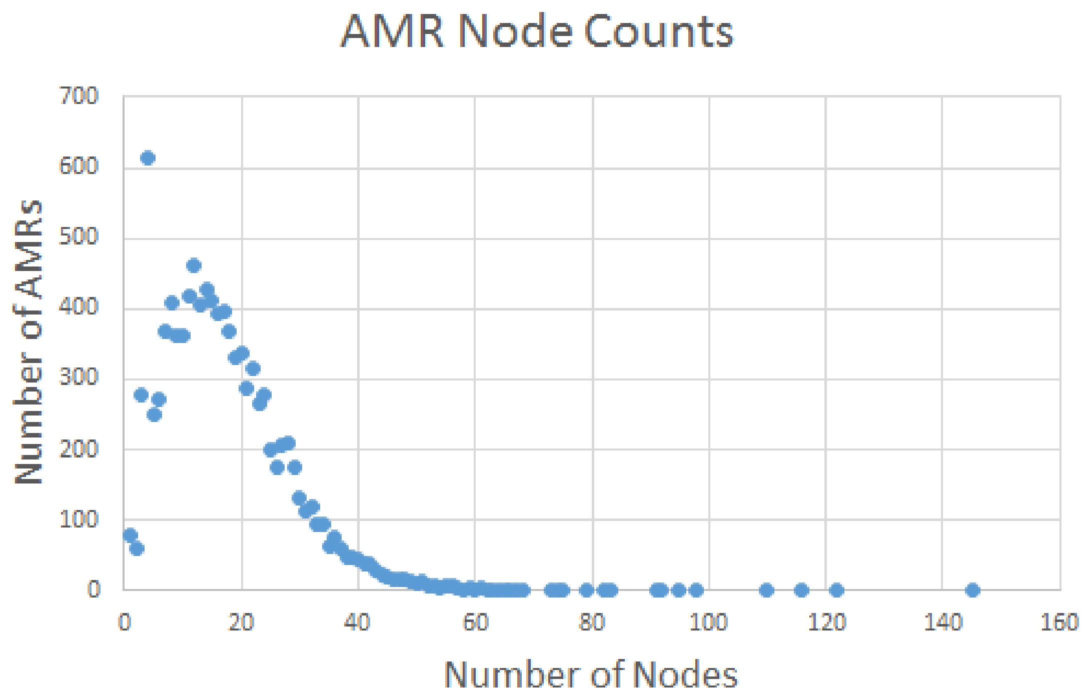

Figure 1.2: Distribution of number of AMRs with given number of nodes.

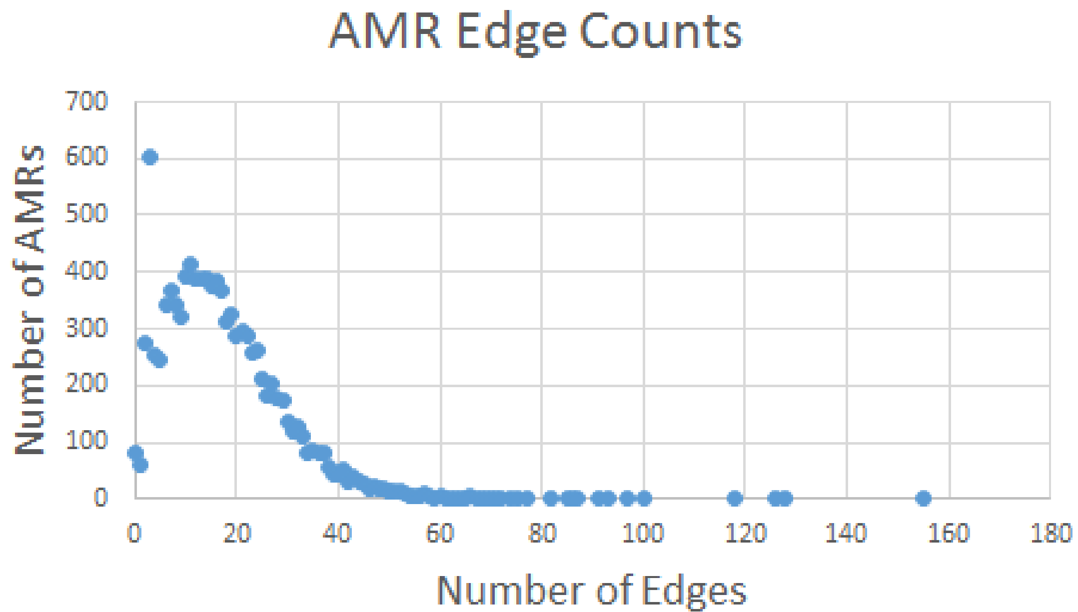

Figure 1.3: Distribution of number of AMRs with given number of edges.

to real English sentences, we can first establish some grammar extracted from the real AMRs which determines a set of possible rules for how each node type and edge type can be combined. Looking at Figure 1.1, this process could be as simple as finding that node label 'believe' can have out edges 'ARG0' and 'ARG1', so if we ever generate 'believe', give it those out edges and find candidate nodes which could use 'ARG0' or 'ARG1' as in edges.

While these artificial AMRs may be useful for testing at scale, they would theoretically correspond to very long sentences. Since sentences of these lengths are unlikely to appear in practice, generating a larger number of smaller AMRs is more closely related to what we expect to happen to real datasets in the future. There are off-the-shelf English sentence to AMR parsers which can also be used to get more realistic data. Once again, how close the produced AMRs are to the original English sentences is not a problem for our task of measuring performance of graph pair similarity scoring.

To obtain n-best lists for scoring, we can either attempt to extract them from existing AMR generators or we can perform some random modifications to the correct AMR. The second approach could be achieved by rewiring existing edges, adding or subtracting edges, and modifying the node and edge labels. This task would be simple and fast and can easily be adapted to produce n-best candidates which are close to or far from the correct AMR.

1.4 GPS-A Key Graph Kernel

For this work we are focused specifically on the task of scoring the similarity of a pair of graphs. There are many ways in which this is done in practice and we briefly discuss two of them here before elaborating on the one specially suited to our AMR application.

Degree distribution measures the number of nodes with a given degree in each graph. These can then be compared to find a measure of similarity. In our directed case, this can be further split into in-degree and out-degree distributions. This is commonly used for measuring similarity of graphs, especially randomly generated graphs. However, given that more than half of our AMRs are trees, this is not a very informative similarity measure for us.

Graph diameter is measured as the maximum distance between any pair of vertices. This can be used as another topological measure of how similar two graphs are to each other. This can once again be thought of in a directed or undirected manner, but it does require that every node is reachable from every other node, so it is more likely to be defined only for the undirected case. Once again, this measure is not highly applicable in our case due to the small nature of our individual AMR graphs.

While there are many other such measures that we could discuss here, one that is standard for use on AMRs given their labeled nodes and edges is called SMATCH [1]. SMATCH is short for semantic match and was developed specifically to handle scoring the similarity between two semantic representations. The first step is to break an AMR graph into triples which capture either the node labeling or the edge labeling and end points. For our example AMR seen in Figure 1.1 we get the following triples:

- instance(a, want)
- instance(b, believe)
- instance(c, Mary)
- instance(d, John)

- ARG1(a, b)
- ARG0(a, d)
- ARG0(b, c)
- ARG1(b, d)

Note that each "instance" is a node which is assigned an alignment (a-d) and its node label. This alignment is marked by a variable (in this case a-d) and is a pairing of a node from one AMR to a node in another AMR. Each edge exists in the form edge-label(node-start, node-end) using the alignment (a-d) for node-start and node-end. The reason for including the alignment of a-d is because aligning two AMRs can be non-trivial. By encoding the nodes as a-d, we can now do the same to another AMR and align a to a, b to b, etc in order to score them.

Once we have the AMRs in this format, we evaluate the precision, recall, and F1 score of each possible alignment of nodes between the two AMRs. Precision is the amount of correct information among all information retrieved and is measured as the number of correct triples divided by the number of triples in the candidate AMR. Recall is the amount of correct information retrieved among all possible correct information and is measured as the number of correct triples divided by the number of triples in the correct AMR. F1 score is then computed as the average of precision and recall.

The SMATCH score is equal to the largest F1 score obtainable by any alignment of nodes. Note that if AMR1 has x nodes and AMR2 has y nodes then the number of aligned nodes is $min(x, y)$. Each node in the smaller AMR is aligned to exactly one node in the larger AMR with no repeats.

A basic implementation of SMATCH can be seen in the following pseudocode in which "nodeMapping" obtains the alignments of nodes from a with nodes from b and "getSMATCH" uses these alignments to obtain F1 scores:

Algorithm 1 Basic SMATCH pseudocode

```
1: procedure getSMATCH(a,b)
2:     maxF1 ← 0
3:     for mapping in nodeMapping(a,b) do
4:         correct ← 0
5:         for alignedPair in mapping do
6:             if labels match then
7:                 correct ← correct + 1
8:         for edges in a do
9:             replace end-points with aligned nodes from b
10:            if new edge exists in b then
11:                correct ← correct + 1
12:        precisionDenominator ← number of triples in b
13:        recallDenominator ← number of triples in a
14:        precision ← correct/precisionDenominator
15:        recall ← correct/recallDenominator
16:        f1 ← (recall + precision)/2
17:        if f1 > maxF1 then
18:            maxF1 ← f1
19:    return maxF1
20: procedure nodeMapping(a,b)
21:    allAlignments ← empty
22:    Select node_a in a
23:    for node_b in b do
24:        newAlignments ← align node_a to node_b
25:        newA ← a − node_a
26:        newB ← b − node_b
27:        newAlignments ← nodeMapping(newA, newB)
28:        append newAlignments to allAlignments
29:    return allAlignments
```

An implementation based on this pseudocode would have complexity of $O(N!|N+E|)$ for each pair of graphs being scored. This is because we have $N!$ ways to align two graphs of N nodes and for each of these alignments we have to process all nodes, N, and edges, E. It is worth noting that if the two AMRs are not the same size, the number of nodes in the alignment matches the smaller AMR. Thus if the smaller AMR has M nodes then we now have to select M nodes from a list of N options where order matters to determine the number of alignments. Therefore the complexity becomes $O(\frac{N!}{(N-M)!}|M+E|)$. In our envisioned application, we score each candidate from an n-best list against the correct AMR. The complexity for scoring an n-best list would then have a constant factor of n in front of it.

1.5 Prior and Related Work

The original paper which introduced SMATCH[1] focused on speeding up the implementation by using heuristics. The main idea was to trade some accuracy in SMATCH score for a much larger speedup in runtime. There were two main heuristics which they implemented and tested. The first

way was to use integer linear programming to solve a constrained version of SMATCH scoring. The second way utilized a hill-climbing scheme to move from random node mappings to ones that would likely increase the F1 score until they could no longer find a similar candidate mapping with a higher score.

There has also been a lot of recent research on AMRs and their applications. Some of this research includes parsing AMRs [2, 6], biomedical applications of AMRs [5, 7], and obtaining AMRs in other languages [3]. Parsing methods include both neural and non-neural approaches, with more of a push toward neural approaches in recent years. Obtaining AMRs in other languages has been an ongoing effort, but currently datasets are only widely available in English and Chinese.

1.6 A Sequential Algorithm

A sequential algorithm could follow the above provided pseudocode almost exactly. The AMRs must be read in first, and for this we use existing code. From there the options for which programming paradigm to use are up to the user. The AMR graphs should ideally be stored in a data structure which allows the user to obtain alignments easily since this is the worst part of the complexity of the above algorithm. Most of the existing code for processing AMRs is written in Python, so it is natural and beneficial for consistency to continue to use this language. NetworkX is a standard graph library to use with Python for sequential implementations. When using NetworkX in Python and using the above provided pseudocode as the body of the program, the complexity matches the complexity provided above.

1.7 A Reference Sequential Implementation

Our implementation uses NetworkX on Python. We first read the AMRs in and convert them into the NetworkX format, specifically using the MultiDiGraph graph type to keep all directed information. It is desirable to use MultiDiGraph also because there can be cases in which two edges have the same endpoints. An example sentence in which this occurs is “I hurt myself”. The AMR for this sentence has only two nodes since “I” and “myself” refer to the same entity, but has two distinct edges from “hurt” to “I” since “I” act as both agent and patient. See Figure 1.4 for the corresponding AMR.

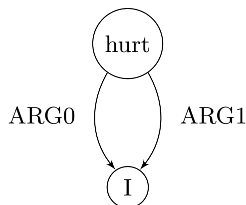

Figure 1.4: Example AMR for the sentence “I hurt myself.”

After the AMRs are stored in NetworkX graphs, the code roughly matches the pseudocode. The NodeMapping function which obtains alignments between the two AMRs is written in the recursive style from the pseudocode. The GetSMATCH function then scores each alignment and stores the highest F1 score obtained.

The output is guaranteed to be the maximum F1 score, since all alignments are exhaustively searched. This is the desired result for our application and verifies that our code is working properly.

1.8 Sequential Scaling Results

The scaling tests were run on a computer running Ubuntu which has 24 GB of RAM and an 8-core Intel I7 3.6 GHz processor, though only one core is used due to the sequential nature of the code. Python 2.7 was used for the implementation since existing code was written for this version. NetworkX version 1.11 was the library used for these tests. The AMRs came from the dataset described in section 1.3 and the first AMR with the desired number of nodes was selected for each test.

The first set of tests were run by scaling the size of both AMRs in the scoring pair at once. The motivation for this type of test is that in typical applications the size of candidate AMRs are close to the size of the correct AMR and thus using AMRs that are of equal size gives us a good idea of what the scaling may look like in practice. The results can be seen in Figure 1.5. These plots show that the runtime grows at a rate comparable to the factorial curve, which matches our complexity analysis above. It is worth noting that these tests were limited to up to 10 nodes because running with 11 nodes used up all of the computer's available physical RAM and prevented the program from making reasonable progress.

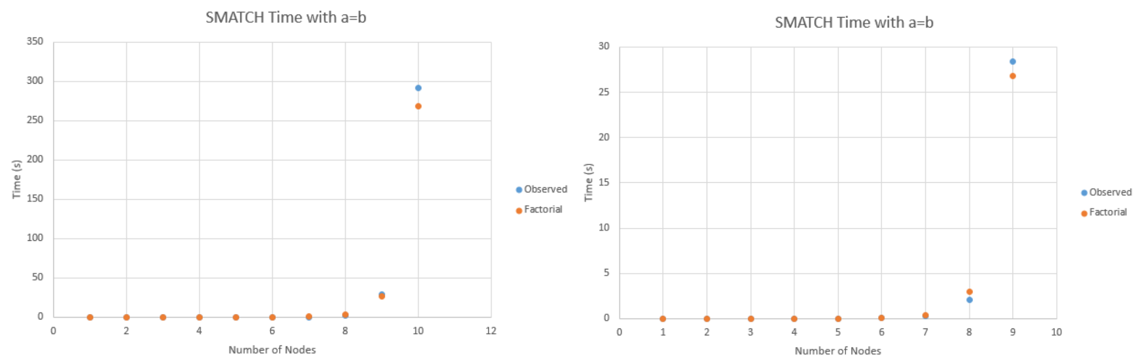

Figure 1.5: Time for sequential implementation to compute SMATCH score when the number of nodes for both AMRs is equal. Ten node datapoint is excluded in chart at right to show trend on smaller data.

The second set of tests fixed the size of one AMR and varied the number of nodes in the other AMR being scored. The motivation for this test is that when scoring all candidates from an n-best list, the correct AMR does not change, but the size of the candidates may vary. To obtain a balance between non-trivial size, relatively low runtime to allow many datapoints, and a size which occurs frequently in practice, I used an AMR with four nodes. In this case, we expect the complexity to be $O(\frac{N!}{(N-M)!}|M+E|)$ (see Section 1.4 for details). Since we fix the size of one AMR at four, $|M+E|$ is fixed for all tests above three nodes. Additionally, the number of terms in $\frac{N!}{(N-M)!}$ is likewise fixed at the number of nodes in the smaller AMR, here four. This gives us the scaling observed which is approximately cubic as shown by the trend lines in Figure 1.6.

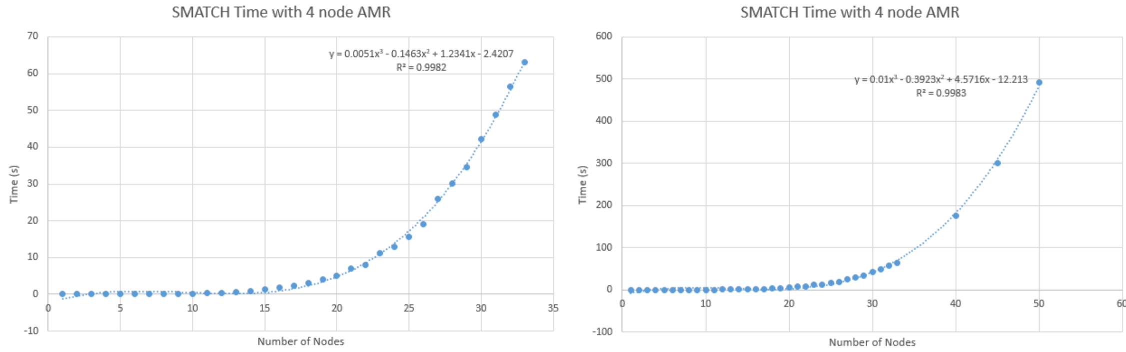

Figure 1.6: Time for sequential implementation to compute SMATCH score when the number of nodes in one AMR is fixed at 4. Three larger datapoints added in chart at right to show trend continues with further scaling.

1.9 An Enhanced Algorithm

When attempting to find the optimal F1 score, the search space can be substantially reduced if we can discard certain subgraph matchings. For example, if we know that an F1 score of 1 can be achieved, we can throw away any alignments which match even a single pair of nodes with mismatched labels. We can reformulate the above pseudocode to combine the alignment and scoring steps in order to avoid wasting computation pursuing unhelpful subgraphs.

The general intuition behind this approach is that we first obtain the score of a "promising" alignment. This establishes a threshold score. Then, we order the nodes in the smaller graph. Next, we match the first node in the smaller graph with a node in the other graph and score this match (do node labels match). Now we continue by scoring the subgraphs which do not contain these two nodes. By scoring as we align the nodes, we can skip an entire subgraph if we already have more incorrect triples (edge or node labels) than our current threshold score. When we encounter a higher F1 score, we update our threshold accordingly.

This works because the size of both graphs is fixed for all alignments, so if there are more incorrect triples in a partial scoring, then the F1 score will be lower for every alignment which contains that partial scoring. There are a couple of factors which affect the benefits of this approach. First, the earlier we find the optimal F1, the faster we have the strictest threshold, the more subgraphs we can exclude. Second, a higher SMATCH score results in a higher threshold and thus also more subgraphs excluded. There are other factors involved, but this gives some sense of how important it is to have closer matches between the two graphs being scored and to establish a high threshold early.

Ideally, the high potential SMATCH score will be taken care of by the n-best list only containing "good" AMR candidates, so we put some effort into establishing a high threshold early. The approach we take is a greedy node label matching. We go through the smaller graph and attempt to match each node with a node in the other graph with the same node label. We match as many nodes as possible in this way, then randomly match the rest. This will clearly be more effective on similar AMRs and not very helpful on AMRs with few shared node labels.

Since this procedure is highly recursive and the subgraph matching and scoring portions do not need to communicate with each other, parallelization is relatively straightforward. First, we match a single node from the first AMR with a single node from the second AMR. Then, we send the subgraphs which do not contain the matched nodes to a worker process to complete the computations. This allows us to handle P subgraphs simultaneously, where P is the number of

concurrent processes.

1.10 A Reference Enhanced Implementation

The enhanced implementation was built from the sequential implementation, and thus used Python 2.7 and NetworkX. For parallelization, the Multiprocessing library for Python was used.

Pseudocode for enhanced implementation of SMATCH:

Algorithm 2 Enhanced SMATCH pseudocode

```
1: procedure getSMATCH(a,b)
2:     maxF1 ← 0
3:     Select node_a in a
4:     alignments ← empty
5:     correct ← 0
6:     incorrect ← 0
7:     for node_b in b do
8:         alignments ← align node_a to node_b
9:         newA ← a − node_a
10:        newB ← b − node_b
11:        if node labels match then
12:            f1 ← subSMATCH(newA, newB, correctCount = 1)
13:        if node labels do not match then
14:            incorrect ← incorrect + 1
15:            if number of incorrect surpasses threshold then
16:                Continue to next node_b
17:            if number of incorrect below threshold then
18:                f1 ← subSMATCH(newA, newB, incorrectCount = 1)
19:        for edges in a do
20:            replace end-points with aligned nodes from b
21:            if new edge exists in b then
22:                correct ← correct + 1
23:        precisionDenominator ← number of triples in b
24:        recallDenominator ← number of triples in a
25:        precision ← correct/precisionDenominator
26:        recall ← correct/recallDenominator
27:        f1 ← (recall + precision)/2
28:        if f1 > maxF1 then
29:            maxF1 ← f1
30:    return maxF1
```

The main differences here are explained in section 1.9. The main idea is that we pair nodes, score them, check to see if we should continue working on them and if so send remaining subgraphs to continue this process. It is worth noting that the complexity of this method is reduced to $O(\frac{N!}{(N-M)!}|E|)$ in the worst case and $O(\frac{N!}{(N-M)!})$ in the case that edges are roughly evenly distributed among nodes (as is fairly typical of AMRs). This reduction comes from sharing scoring among all subgraphs using a particular node matching and thus avoiding large amounts of recomputation.

The threshold cutoff massively reduces the search space when the similarity between the AMRs is high, but does not improve the theoretical worst case complexity.

In the case of the parallel implementation, all the pseudocode remains the same, except that P instances of subSMATCH are run on P processors simultaneously.

1.11 Enhanced Scaling Results

As with the sequential baseline, the scaling tests were run on a computer running Ubuntu which has 24 GB of RAM and an 8-core Intel I7 3.6 GHz processor. Python 2.7 was used for the implementation since existing code was written for this version. NetworkX version 1.11 was the library used for these tests. Parallel runs used the Multiprocessing library. The AMRs came from the dataset described in section 1.3 and the first AMR with the desired number of nodes was selected for each test.

As before, the first set of tests scaled both AMRs simultaneously. In order to see the trend more clearly for the parallel implementation we ran up to 11 nodes. It is worth noting that previously 11 nodes was not feasible because of memory constraints, but by scoring as we match we avoid having to store all possible alignments and thus only use a fraction of the memory used before. For all results shown, “basic” is our starting sequential implementation from above, “enhanced” combines scoring and matching with the threshold cutoff, and “parallel” further uses all 8 cores with the enhanced algorithm. We can see in Figure 1.7 that the enhanced version takes far less time than the basic implementation. The parallel implementation further improves upon the enhanced. The size of the search space grows factorially as the number of nodes increases, and while we are able to prune a substantial amount of this space, we still suffer from a similar scaling issue. This is to be expected from an algorithm that guarantees a correct result to an NP-complete problem as ours does.

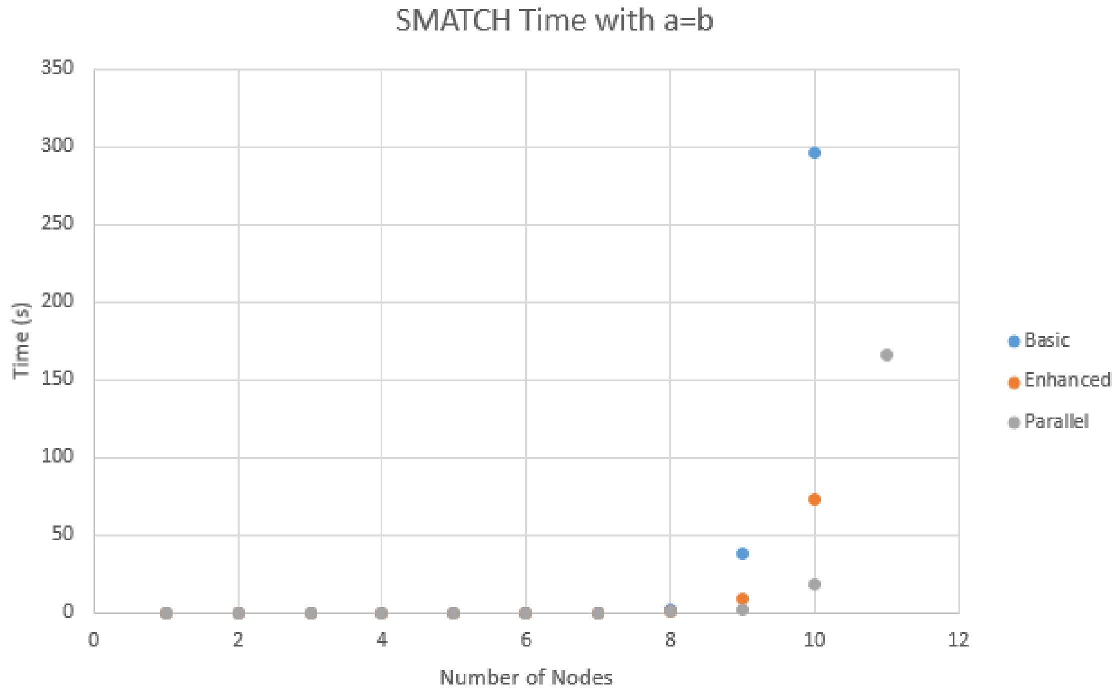

Figure 1.7: Time for implementations to compute SMATCH score when the number of nodes for both AMRs is equal.

The second set of tests fixed the size of one AMR at 4 nodes as was done in the sequential tests. Again, the idea here is that in practice the correct AMR is fixed while the candidates being scored may vary in size. We can see in Figure 1.8 that the cubic trendline still dictates the runtime growth as before, but once again our enhanced and parallel implementations vastly outperform the baseline.

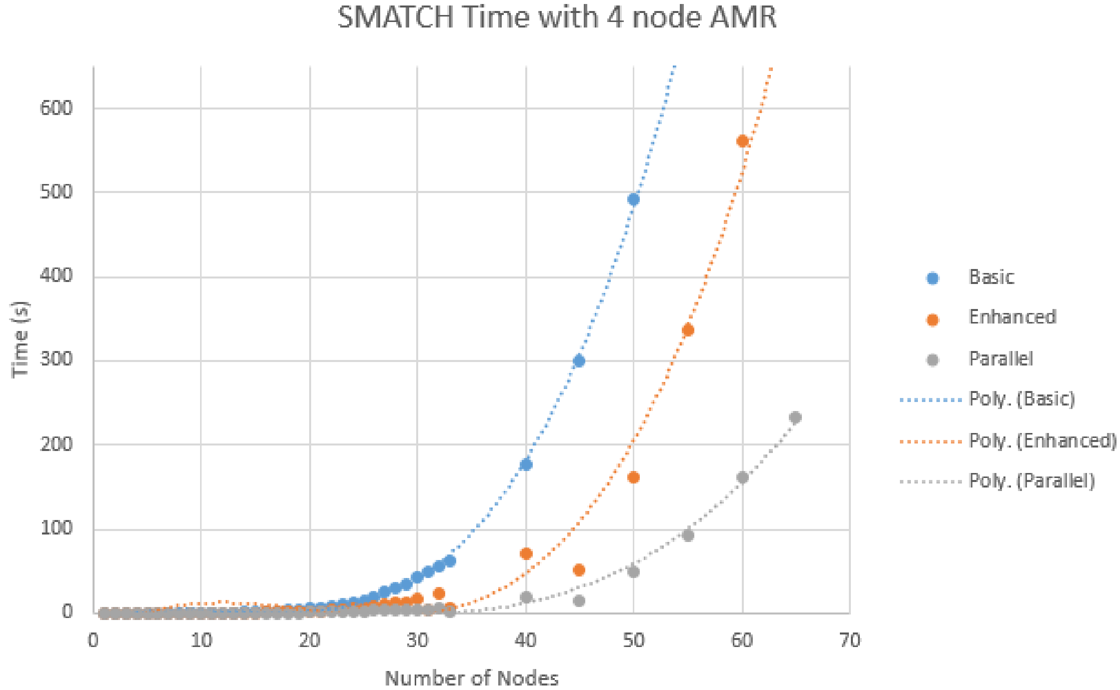

Figure 1.8: Time for implementations to compute SMATCH score when the number of nodes in one AMR is fixed at 4.

Finally, we sought to measure how much of an impact the SMATCH score has on runtime. Our enhanced and parallel methods' pruning strategies rely on high similarity to enable pruning large amounts of the search space. In the previous experiments discussed, there was no reason to believe that the two AMRs being compared in each case had any substantial similarity as they were just distinct sentences taken from the real dataset. To test the impact of high similarity on runtime, a function was written to artificially modify an AMR to generate similar candidate AMRs. The following pseudocode describes this process:

Algorithm 3 Candidate Generation pseudocode

```
1: procedure GETCANDIDATE(AMR,DIFFPERCENT)
2:     for node in amr do
3:         diffPercent chance to change node label
4:     for edge in amr do
5:         diffPercent chance to change edge endpoint
6:         diffPercent chance to change edge label
       return amr
```

The value for "diffPercent" was increased in five point increments from five percent up to one hundred percent. The three types of triples described earlier which determine the SMATCH are all accounted for in this candidate generation as nodes can be relabeled, edges can be relabeled, and edges can be rewired. As we can see in Figure 1.9, as the percent chance of relabelling and rewiring increases (and thus SMATCH score decreases), the amount of time taken by our enhanced and parallel implementations increases. In the best cases, with SMATCH scores above 0.7, our enhanced and parallel implementations run in under a second. In the worst cases, when SMATCH

drops below 0.1, we are forced to search nearly the whole possible search space and get results similar to our previous experiments. This demonstrates that our methods perform very well for AMRs that are very similar, due to our ability to effectively prune a large percentage of the possible search space.

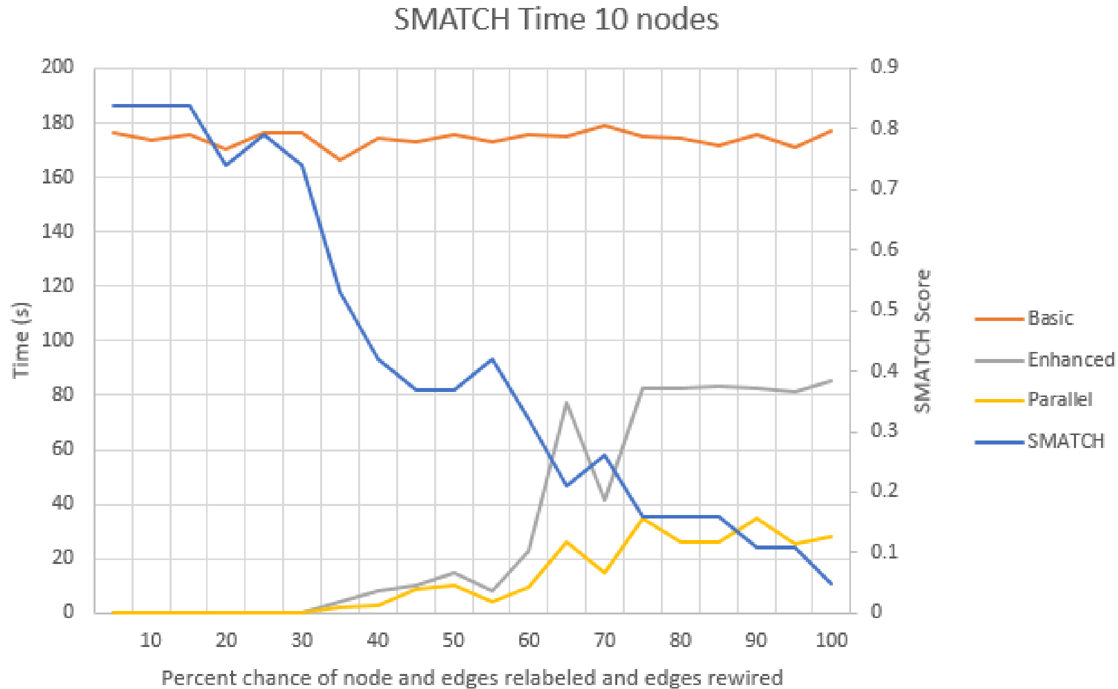

Figure 1.9: Time for implementations to compute SMATCH score when the amount of differences between AMRs increases. Basic, Enhanced, and Parallel runtime uses left y-axis, SMATCH score uses right y-axis.

1.12 Conclusion

This paper has examined SMATCH for graph similarity scoring as applied to abstract meaning representations (AMRs). Finding the SMATCH score for a pair of graphs involves finding the best F1 score for any alignment of nodes between graphs. This results in an NP-complete problem, and thus most of the time a heuristic is applied to trade accuracy for performance. In this work, the optimal result was guaranteed through our algorithms, and we sought to take advantage of the expected use case of candidate AMRs which are similar to the correct AMR they are being compared against. By doing this, we can prune a large amount of the search space in the typical case. We have shown how this improves scalability and lends itself nicely to parallelism.

Overall, the closer the pair of AMRs being compared are to each other, the more reasonable the runtime of our algorithms. If we are training a system using SMATCH, it would be reasonable to utilize an existing heuristic for initial training and move to an exact method like ours once your model is able to create reasonably good candidate AMRs for scoring. For our dataset, the average number of nodes in an AMR is 17.1, so being able to utilize an exact method as ours even just on the smaller AMRs still allows us to cover a large number of AMRs.

1.13 Response to Reviews

Thank you to the reviewers for your helpful feedback. In accordance with your comments, I made the following changes after the first set of reviews:

1. Expanded the description of agent and patient
2. Clarified which direction the edges are directed in the introduction
3. Bolded my definition of n-best list
4. Defined an alignment of nodes between two graphs
5. Added to my existing definitions of precision and recall by giving word descriptions along with the existing formulae
6. Reworded the last paragraph in 1.1
7. Added explanation of how the complexity was derived and how it changes when the two AMRs do not have the same number of nodes

I made the following changes after the second set of reviews:

1. In the sequential scaling section I specified that only one core was used for the basic implementation.
2. In Section 1.7 I fixed a figure reference.
3. Added a brief description of the nodeMapping and getSMATCH functions before the pseudocode for the basic sequential implementation.

Bibliography

[1] Shu Cai and Kevin Knight. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 748–752, 2013.

[2] Chunchuan Lyu and Ivan Titov. Amr parsing as graph prediction with latent alignment. *arXiv preprint arXiv:1805.05286*, 2018.

[3] Noelia Migueles-Abraira. *A Study Towards Spanish Abstract Meaning Representation.* PhD thesis, University of the Basque Country, 2017.

[4] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Computational linguistics*, 31(1):71–106, 2005.

[5] Sudha Rao, Daniel Marcu, Kevin Knight, and Hal Daumé III. Biomedical event extraction using abstract meaning representation. *BioNLP 2017*, pages 126–135, 2017.

[6] Lai Dac Viet, Nguyen Le Minh, and Ken Satoh. Convamr: Abstract meaning representation parsing. *arXiv preprint arXiv:1711.06141*, 2017.

[7] Yanshan Wang, Sijia Liu, Majid Rastegar-Mojarad, Liwei Wang, Feichen Shen, Fei Liu, and Hongfang Liu. Dependency and amr embeddings for drug-drug interaction extraction from biomedical literature. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 36–43. ACM, 2017.