Exploiting Graph Structure for Accelerating the Calculation of Shortest Paths in Wordnets

Holger Wunsch
Collaborative Research Center 441 "Linguistic Data Structures"
University of Tübingen, Germany
wunsch@sfs.uni-tuebingen.de

Abstract

This paper presents an approach for substantially reducing the time needed to calculate the shortest paths between all concepts in a wordnet. The algorithm exploits the unique "star-like" topology of wordnets to cut down on time-expensive calculations performed by algorithms to solve the all-pairs shortest path problem in general graphs. The algorithm was applied to two wordnets of two different languages: Princeton WordNet (Fellbaum, 1998) for English, and GermaNet (Kunze and Lemnitzer, 2002), the German language wordnet. For both wordnets, the time needed for finding all shortest paths was brought down from several days to a matter of minutes.

1 Introduction

Significant effort has been devoted in linguistic research to the problem of determining the semantic distance[1] between two concepts. Many of the approaches that were developed to provide a solution for the task use wordnets as their basic knowledge resource. Budanitsky and Hirst (2006) present an extensive number of approaches for determining lexical semantic relatedness based on the Princeton WordNet (Fellbaum, 1998). A large number of these solutions have in common that at some point in the calculation, the length of the shortest path that connects the two concepts in question has to be taken into account. The length of the shortest path between two concepts is thus a vital piece of information in virtually any approach for determining their semantic distance.

From a computer science perspective, a wordnet is a directed graph. The nodes in the graph correspond to the concepts that are stored in the wordnet. Two nodes are connected by an edge if there exists a semantic relation between the two concepts that correspond to the nodes. The edges are directed, reflecting the directedness of semantic relations in wordnets, such as the relation of hypernymy. The problem of finding the shortest path between two nodes in a graph has been well studied in computer science. Sedgewick (1990) presents two algorithms:

- *Dijkstra's algorithm* finds the shortest path between two concepts in quadratic time, and is extensible to find the shortest paths between all concepts in cubic time.
- The *Floyd-Warshall algorithm*, which is very easy to implement, solves the all-pairs shortest path problem in cubic time as well. The Floyd-Warshall algorithm served as the baseline algorithm for this paper.

Both algorithms operate on general directed graphs. The structure of a directed graph is sketched in figure 1. Given two nodes, there exist multiple different paths that connect the two nodes. In figure 1, in order to get from L to D, one could take the path L–G–E–D, but also L–G–C–A–F–E–D, and several other paths. An algorithm that looks for the shortest path must deal with this situation and basically be able to consider all alternatives.

Computing of the length of all shortest paths with the Floyd-Warshall algorithm takes about 120

[1] see Budanitsky and Hirst (2006) for a critical discussion of the term "semantic distance".

Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008), pages 1001–1008
Manchester, August 2008

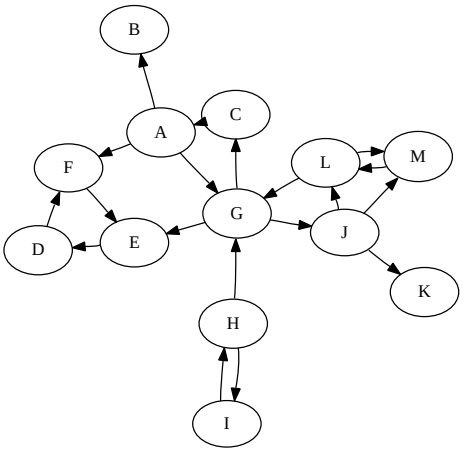

Figure 1: Schematic structure of a general graph. Example taken from Sedgewick (1990).

hours for GermaNet, using a compiler-optimized C implementation on a machine equipped with two AMD Opteron processors. GermaNet, which contains 53,312 synsets[2] in its current release 5, is of moderate size, compared to the Princeton WordNet, which contains more than twice as many synsets. The execution time of 120 hours is acceptable if the wordnet does not change, and the shortest paths are calculated once and then kept for later use. However, when the task involves the modification of the wordnet itself and the repeated recalculation of all shortest paths, this is unacceptable. In such cases the execution time will most likely prohibit research involving such techniques.

It has been shown that there is no faster way for solving the all-pairs shortest paths problem in general graphs. However, the structure of wordnets is somewhat different to that of a general directed graph. It can best be described to be similar to that of a star, as depicted in figure 2. In the middle, there is a unique top node that dominates all other nodes[3]. In the center area around the top node, the synsets represent general concepts. Closer to the fringe, the synsets become more and more specific. A typical configuration in this region would be a biological taxonomy of animals, or specific kinds of vehicles. The number of such specialized synsets in the outer regions of the wordnet outweighs the number of core concepts by several orders of magnitude. In other words, the majority of synsets in a wordnet is part of tree structures that are arranged around a relatively small core graph.

In the remainder of this paper, we will present an approach to solving the all-pairs shortest paths problem in wordnets that is superior in execution time by consistently exploiting this specific structure of wordnets.

2 Finding Paths In a Wordnet

We implemented the Floyd-Warshall algorithm as our baseline. The Floyd-Warshall algorithm uses a dynamic programming approach. The basic idea is to find all shortest paths by first computing and storing the paths lengths between nodes that are close to each other, and then moving on to longer paths by combining the results of the shorter paths. The Floyd-Warshall algorithm uses a matrix that contains for each pair of nodes the length of the shortest path that connects the two nodes. The matrix is initialized by setting the length of the shortest path between any two adjacent nodes to 1[4]. All other fields are set to ∞ (which indicates that no path has been found (yet)). Then the algorithm checks for any pair of nodes x and y whether there exists a node z such that the path from x to y that runs along z is shorter than the path between x and y that has been found so far.

Floyd-Warshall Algorithm

N: set of nodes
p: matrix that contains the lengths of the shortest path between any node x and $y \in N$.

Initialization Step
for all $x, y \in N$ **do**
  if $neighbors(x, y)$ **then**
    $p_{xy} := 1$
  else
    $p_{xy} := \infty$
  end if
end for

[2] A *synset* is a set of words that are synonymous. Throughout this paper, we will use the term *synset* synonymously to *nodes in a graph*, since the relevant relations (which are represented by edges in the graph) hold between synsets.

[3] In most wordnets such as Princeton WordNet or GermaNet, there is usually no unique top node. Instead, there is a set of most general concepts, called *unique beginners*. In the example in figure 2, the unique beginners would be the synsets s1, s3, s11, s15, and s23. The algorithm presented here requires the wordnet to be a connected graph, therefore we stipulate an explicit artificial top node.

[4] The Floyd-Warshall algorithm can also be used to calculate shortest paths in weighted graphs in which case the matrix would be initialized with the weights of the edges between adjacent nodes. We will focus on unweighted graphs in this paper.

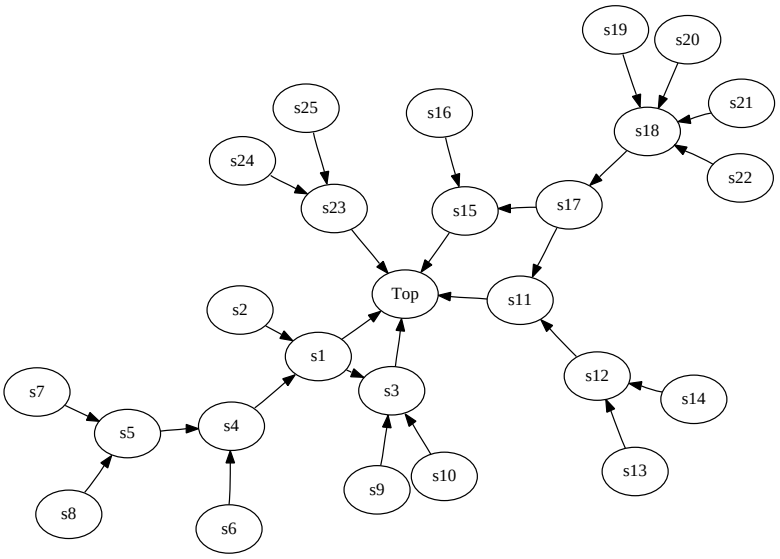

Figure 2: Schematic star-like structure of a wordnet.

Shortest Path Calculation
for all $z \in N$ **do**
  for all $x, y \in N$ **do**
    $p_{xy} := min(p_{xy}, p_{xz} + p_{zy})$
  end for
end for

Since the *shortest* of all possible paths must be found, the Floyd-Warshall algorithm has to check for *any* node z if the path x–z–y is the shortest. Returning to the example in figure 2, the shortest path between x = s1 and y = s16 is s1–Top–s15–s16. The Floyd-Warshall algorithm also considers the other possible paths, such as s1–**s3**–Top–s15–s16 or s1–Top–**s11**–**s17**–s15–s16, which are eventually discarded because they turn out to be not the shortest connection.

Now consider a path such as the one illustrated in figure 3, which is the shortest undirected path between the synsets s8 and s20[5]. In order to find this shortest path, the Floyd-Warshall algorithm would check *all possible combinations* of nodes x–z–y in order to find a potential shorter alternative. To the human reader it is obvious that this is an unnecessary amount of work: there is only one path that leads from s8 to s1, because the part of the wordnet rooted in s4 is in fact a *tree*, and paths connecting two nodes in trees are always unique. The same is true for the structure rooted in s18. Moreover, there is a unique link from s4 to s1, and from s18 to s17. The synsets s1 and s17 are part of the core structure of the wordnet with higher density. Here, between s1 and s17 there do exist more than one paths which must all be considered by the path finding algorithm.

The complexity of the Floyd-Warshall algorithm results from the necessity of considering all possible paths through a general graph in order to find the shortest one. Wordnets are graphs – but, as explained above, they have a very special structure: There is a core graph consisting of a limited number of synsets, but the majority of all synsets is arranged in tree structures that are attached to the core graph, which gives wordnets the star-like structure described earlier.

This structure, which is specific to wordnets, can be exploited such that expensive calculations are only performed where necessary – in the core graph of the wordnet – while specialized cheaper calculations are used in the outer parts.

3 Structure-Adapted Shortest Path Search

The observations about the structure of wordnets and the nature of general algorithms to find all shortest paths lead to the conclusion that an algorithm that is adapted to the specific structure of wordnets should be superior over general algorithms with respect to execution time. In this section, we will present such a structure-adapted approach. It operates in two stages: In the first stage,

[5] Since the edges in the graph correspond to the hypernymy relations in the wordnet, they all point towards the top node. In order to get from s8 to s20, one must first follow the path up to the top node along the hypernymy axis, and then down to s20 in the opposite direction. Therefore, paths are undirected.

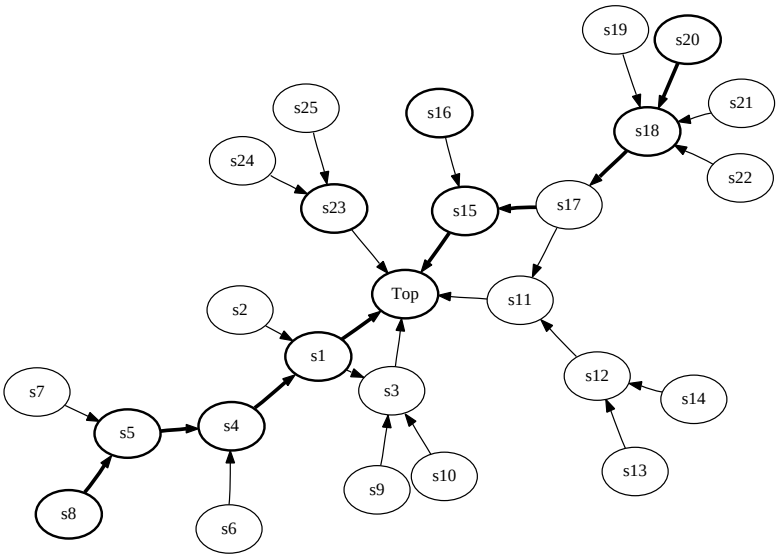

Figure 3: A path through the wordnet.

all nodes in the wordnet are classified whether they are part of the core graph or a peripheral tree structure. The second stage is the shortest path search proper, which uses the information about the structure of the wordnet that was acquired in the first stage.

3.1 Stage 1: Node classification

In the first stage, the algorithm determines whether a node belongs to the graph proper which constitutes the core network, or to a tree structure in the periphery. The algorithm classifies nodes in four classes, which are illustrated in figure 4.

1. **Inner nodes**: Inner nodes belong to the graph in the center of the network. A node is an inner node if it has more than one parent node, or if one of its children is an inner node. In figure 4, inner nodes have a white background.

2. **Root nodes**: A root node, as suggested by its name, is the root node of a tree. Root nodes have a unique parent node, which must be an inner node. In figure 4, root nodes have a dark gray background, and thick borders.

3. **Tree nodes**: Tree nodes are part of a tree, i.e. they have one unique parent node. This parent node must either be a root node, or a tree node as well. In figure 4, tree nodes have a dark gray background (and thin borders).

4. **Leaf nodes**: Leaf nodes have a unique parent node, which must be an inner node. They do not have any child nodes. As such, leaf nodes are actually a special case of a root node. But for performance reasons, they will be handled separately from root nodes. In figure 4, leaf nodes have a light gray background.

Two remarks are in order. The tree structures referred to in this classification rely on a well defined parent-child relation. The hypernymy relation, which is the only relation between synsets we consider in this paper, is a directed relation that satisfies this property: if synset x is a hypernym of synset y, then x is a parent node of y.

The difference between the terms *tree node* and *leaf node* may seem a little arbitrary – considering for example s8, which is a tree node, and s9, which is a leaf node. Both nodes do not have child nodes. However, from a performance perspective, it is advantageous to treat leaf nodes and tree nodes differently in the algorithm, which is why two different nodes types are assumed.

3.2 Stage 2: Shortest Path Search

The second stage is the actual pathfinding step. The underlying basic idea is to *split* the calculation: Consider the sample path in figure 3 between synsets s8 and s20. This path runs through three regions of the wordnet. The synsets in the first part of the path, s8-s5-s4 all belong to the tree that is rooted in synset s4. Then the path enters the core

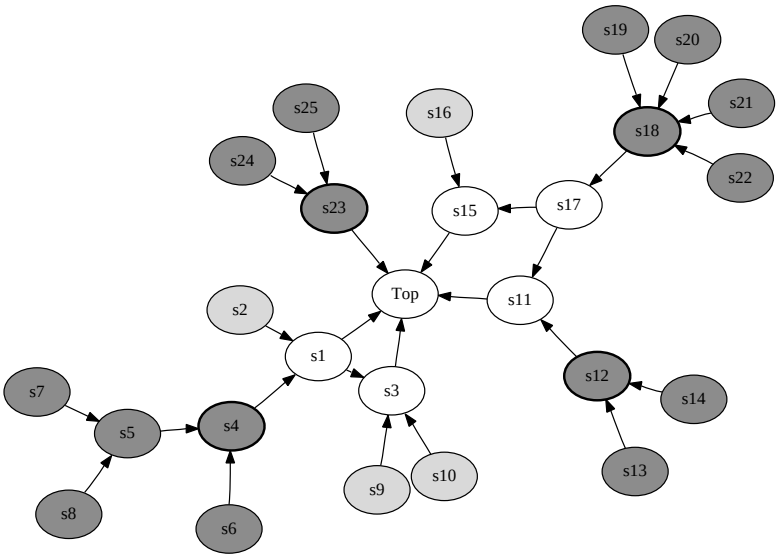

Figure 4: Node classes. White: inner nodes; dark gray: tree nodes; dark gray with thick border: root nodes; light gray: leaf nodes.

graph and runs along s1-Top-s15-s17[6]. The third part of the path is s18-s20. Again, the corresponding synsets are members of a tree whose root is s18.

The important point is that for any part of a path that runs though a tree, this is the *only possible* path through the tree. A general algorithm for finding the shortest path out of a set of multiple possible paths is not necessary here. This way, it is possible to restrict the application of general but time-expensive algorithms to the area of the wordnet where this is needed – the core of the network, while elsewhere it is sufficient to just determine the length of a path through a tree, a task which can be solved very efficiently.

Returning to our example, the algorithm splits the calculation of the shortest path between s8 and s20 as follows: Both s8 and s20 are part of a tree, as determined in step 1. The root of the tree that s8 is a member of is s4. The length of the path between s2 and s4 is 2. s4's parent node is s1. Root nodes have a unique parent node by definition. Therefore we know that only s1 can be the next node on the path, and its distance is 1, since s4 and s1 are neighbors. So, the length of the path up to s1 is 3. In the same fashion, the length of the other part of the path that is located in a tree, the path between s20 and s17, can be computed. The length of this part is 2. Now there remains the part between node s1 and s17. This part of the path runs through the core graph. Here, a general algorithm for finding shortest paths, such as the Floyd-Warshall algorithm, must be used to compute the path's length. The length of the shortest possible path within this region turns out to be 3. Now the lengths of all parts of the path have been determined – and the total length of the shortest path is just the sum of the three parts, which is 8.

[6]there are actually two possible paths of the same length (8) – the other possible path would be the one that runs along synset s11.

4 Implementation

This section will present pseudo-code for the two stages of structure-adapted shortest path search.

4.1 Stage 1: Node classification

Stage 1 explores a wordnet and classifies every node whether it is an inner node, or a root, tree, or leaf node. The procedure starts out by classifying as leaf nodes all nodes that are childless and that have one unique parent node. Note however that the definition of a leaf node requires that its parent node be an inner node. This constraint can not be checked at this point, since information about inner nodes is not yet available. Therefore the check is postponed until later.

Next, any node that has more than one parent node is classified as an inner node. The transitive closure of all of this node's parent nodes is classified as inner nodes as well.

The remaining nodes that have not been classi-

fied so far are either root or tree nodes. A node is a root node if its parent node is an inner node, otherwise it is a tree node.

The last step is to check the constraint on leaf nodes which has been postponed. As stated in the definition, leaf nodes are childless and have an inner node as parent. Nodes with no children whose parent node is either a root node or a tree node are *not* leaf nodes, but rather tree nodes. Thus, each potential leaf node is visited to ensure that its parent node is in fact an inner node in which case the classification as a leaf node remains unchanged. Otherwise, the node is reclassified as tree node.

The pseudocode of the node classification algorithm is listed below.

Node Classification Algorithm[7]

N: set of nodes
Node classes: $inner$, $leaf$, $root$, $tree$, $undefined$

Classify leaf nodes
for all $n \in N$ **do**
  if $|children(n)| = 0 \wedge |parents(n)| = 1$ **then**
    $assign\_class(n, leaf)$
  end if
end for

Classify inner nodes
for all $n \in N$ **do**
  if $|parents(n)| > 1$ **then**
    if $class(n) \neq inner$ **then**
      $assign\_class(n, inner)$
      {All parent nodes of inner nodes are inner nodes as well}
      for all $m \in parents^*(n)$ **do**
        $assign\_class(m, inner)$
      end for
    end if
  end if
end for

Classify root and tree nodes
for all $n \in N$ **do**
  if $class(n) = undefined$ **then**
    if $class(parents(n)[0]) = inner$ **then**
      $assign\_class(n, root)$
    else
      $assign\_class(n, tree)$
    end if
  end if
end for

Reclassify leaf nodes as tree nodes if they are children of tree nodes
for all $n \in N$ **do**
  if $class(n) = leaf$ **then**
    if $class(parents(n)[0]) = root \vee class(parents(n)[0]) = leaf$ **then**
      $assign\_class(n, tree)$
    end if
  end if
end for

[7] **Notes on the notation:** $parents(n)$ and $children(n)$ are functions that return a list of all parent or child nodes of the node n. $parents(n)[0]$ returns the first node in the list of parent nodes of n. $parents^*(n)$ is the transitive closure of all parent nodes of n. $|parents(n)|$ is the number of parent nodes of n.

4.2 Stage 2: Finding Shortest Paths

The actual calculation of shortest paths takes place in stage 2. In order to calculate the shortest paths between two nodes x and y, two main cases are considered.

4.2.1 x and y do not belong to the same tree

x and y do not belong to the same tree if the root nodes of the trees that x and y are members of are different: $root(x) \neq root(y)$.

The path from x to y is then split into three parts:

- l_{xi_x}: the length of the subpath from x to the first inner node i_x on the path.
- l_{yi_y}: the length of the subpath from y to the first inner node i_y on the path.
- $l_{i_x i_y}$: the length of the subpath from i_x to i_y, which runs through the core.

The following cases are considered by the algorithm:

1. x **is an inner node**: $i_x = x$, and $l_{xi_x} = 0$.
2. x **is a tree node**: l_{xi_x} is the length of the path from x to the root node of the tree r_x, plus 1 to get from r_x to i_x: $l_{xi_x} := l_{xr_x} + 1$.
3. x **is a leaf node or a root node**: l_{xi_x} is 1.
4. y **is an inner node**: $i_y = y$, and $l_{yi_y} = 0$.
5. y **is a tree node**: l_{yi_y} is the length of the path from y to the root node of the tree r_y, plus 1 to get from r_y to i_y: $l_{yi_y} := l_{yr_y} + 1$.
6. y **is a leaf node or a root node**: l_{yi_y} is 1.

The length of the path $l_{i_x i_y}$ (the path running through the core graph) is calculated using the Floyd-Warshall algorithm, that is $l_{i_x i_y} = p_{i_x i_y}$ (see the description of the Floyd-Warshall algorithm above).

The total length of the shortest path is then $l_{xy} := l_{xi_x} + l_{i_x i_y} + l_{yi_y}$.

4.2.2 x and y belong to the same tree

This is a special case that is treated differently from all other cases. Let z be the lowest node in the tree that dominates both x and y (where z may be equal to x or y). Then $l_{xy} = l_{xz} + l_{yz}$.

Shortest Paths Algorithm

Input:
Two nodes $x, y \in N$.
Path matrix p for nodes in the core graph as calculated by Floyd-Warshall

Output:
The length of the shortest path between x and y, l_{xy}.
if $((class(x) = root \vee class(x) = tree) \wedge (class(y) = root \vee class(y) = tree)) \wedge root(x) = root(y)$ **then**
  First case: x and y belong to the same tree
  Let $z \in S$ be the lowest common subsumer of x and y.
  $l_{xy} := l_{xz} + l_{yz}$
  return l_{xy}
else
  Second case: x and y do not belong to the same tree
  $i_x := x$;
  $i_y := y$;
  if $class(x) = tree$ **then**
    $r_x := root(x)$
    $l_{xi_x} := l_{xr_x} + 1$
  else if $class(x) = root \vee class(x) = leaf$ **then**
    $l_{xi_x} := 1$
  else
    $l_{xi_x} := 0$
  end if
  if $class(y) = tree$ **then**
    $r_y := root(y)$
    $l_{yi_y} := l_{yr_y} + 1$
  else if $class(y) = root \vee class(y) = leaf$ **then**
    $l_{yi_y} := 1$
  else
    $l_{yi_y} := 0$
  end if
  $l_{xy} := l_{xi_x} + l_{i_x i_y} + l_{yi_y}$
end if

5 Experiments and Results

5.1 The Data

In order to assess the performance of structure adapted path search, we experimented with two different wordnets. For English, we looked at Princeton WordNet (Fellbaum, 1998) in its current release 3, which contains 117,659 synsets. Only 4,250 of these synsets belong to the core network. The remaining 113,410 nodes are members of peripheral tree structures, which amounts to 96 % of all nodes.

Furthermore, we applied the approach to GermaNet (Kunze and Lemnitzer, 2002) for German. The architecture of GermaNet is modelled after Princeton WordNet and is largely compatible. GermaNet, in its current release 5.0, contains 53,312 synsets. Out of these, 8,728 synsets are members of the core of the network (i.e. the part of the network that is a graph proper). The remaining 44,273 synsets are part of peripheral tree structures (or leaf nodes). Hence, 83% of the synsets in GermaNet are part of substructures that do not require a general algorithm for calculating shortest paths.

The topology of GermaNet is thus slightly different than that of WordNet. The fact that more nodes belong to the core graph indicates that GermaNet's density with respect to the hypernymy relation is higher than the density of WordNet on the level of the more abstract concepts.

5.2 Application of the Algorithm

We conducted our experiments on a machine equipped with two AMD Opteron 250 processors running at 2.4 GHz and 8 GB of main memory.

For the path-finding stage, we implemented two C programs. Both operate on the same input. Both programs were compiled using gcc's `-O3` option for maximum optimization.

The first program only used the Floyd-Warshall algorithm – the node classes were effectively ignored. For WordNet, the estimated processing time was at least 35 days. This value was computed by interpolating the time that had passed to process 15,000 synsets. At this point, the tests were aborted. Since Floyd-Warshall becomes

Princeton WordNet	
Synsets	117,659
Inner nodes	4,250
Root nodes	7,174
Tree nodes	56,532
Leaf nodes	49,704
Node classification time	ca. 1 second
Floyd-Warshall path search	$>$ 35 days
Structure-adapted path search	9 minutes
GermaNet	
Synsets	53,312
Inner nodes	8,728
Root nodes	4,641
Tree nodes	18,949
Leaf nodes	20,683
Node classification time	1.2 seconds
Floyd-Warshall path search	120 hours
Structure-adapted path search	40 minutes

Table 1: Structure-adapted path search – Summary of results

slower the more nodes have been processed, this value is likely to be even higher. GermaNet contains only half of the synsets, and the Floyd-Warshall algorithm completed in 120 hours[8]. The result, a matrix of 53,312×53,312 elements, containing the lengths of the shortest paths between all nodes, was written to a binary file whose size was roughly 3 GB.

In the second program, we implemented the structure-adapted shortest path search approach. For calculating shortest paths in the core graph area of the network, we used the same implementation of the Floyd-Warshall algorithm as in the first program. With the same input data and the same machine, we were able to bring the execution time down to 40 minutes for GermaNet, and only 9 minutes for WordNet. This includes creating the output file, which had the same binary format as the one generated by the first program. The difference in processing time between WordNet and GermaNet stems from the smaller core graph in WordNet, which allows for even more nodes to be excluded from the time consuming Floyd-Warshall calculation. The results of the experiments are summarized in table 1.

[8]The complexity of the Floyd-Warshall algorithm is cubic, therefore twice as many synsets result in a processing effort that is eight times higher.

6 Discussion

The experiments show that with a thorough preanalysis of the structure of a wordnet and consistent usage of this additional information, the time it takes to calculate all shortest paths can be reduced dramatically. This is because most nodes in a wordnet are part of substructures that are proper trees and not general graphs. In trees, it is possible to calculate the length of a path very much more efficiently than in an arbitrarily-structured graph.

We successfully applied structure-adapted path search to two wordnets, the Princeton WordNet and GermaNet. Since the algorithm does not rely on concrete properties of a specific wordnet, it can easily be applied to wordnets for other languages.

The benefits of structure adaptation diminish with increasing density of the network as more and more nodes become part of the core graph. In this case, the execution time will approach that of the Floyd-Warshall algorithm. It is also obvious that the approach does not generalize to arbitrary graphs. As long as the structure of graphs is similar to the star-like structure of wordnets, we expect the approach to be beneficial in applications involving such graphs as well.

References

Budanitsky, A. and Hirst, G. (2006). Evaluating WordNet-based Measures of Lexical Semantic Relatedness. In *Computational Linguistics*, volume 32. Association for Computational Linguistics.

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA.

Kunze, C. and Lemnitzer, L. (2002). GermaNet – Representation, Visualization, Application. In *Proceedings of LREC*, pages 1485–1491.

Sedgewick, R. (1990). *Algorithms in C*. Addison Wesley.