PSC - Project report

Daniele Sampietro

September 2025

1 Setup

To build the graph, the starting text is tokenized. Every word becomes a node and an edge is added between adjacent words.

We define a sentence as the concatenation of labels of a path from a node to the final node.

2 Design choices

The underlying structure of a Graph is a map. With this structure Go guarantees the thread-safety of read-only operations; therefore we don't need mutexes or other synchronization primitives.

Since the graph can contain cycles, this could cause non-termination of the sentence generation. To avoid this, there is a parametric hard limit: we generate all sentences long up to max_depth.

Another question was in the type of channel to use in order to implement edges. Clearly they have to be buffered, since we don't want blocking when processing a message, but the buffer size must be large enough to accommodate any possible max_depth.

My solution was to choose an **unbounded channel**. This structure consist of a first In channel (external world $\rightarrow$ buffer), an inner queue as buffer, and an Out channel (buffer $\rightarrow$ external world). We could see an unbounded channel as a buffered channel with an infinite buffer size. This allows a better scalability for bigger texts and for varying depths.

3 Manual

The program exposes the following flags:

- -file_path=FILE: the path of the file to analyze
- -max_depth=NUM: maximum depth of the generated sentence
- -export_graph: enable to export the text network in .dot
- -print_sentences: enable to print all the generated sentences
- -seq: enable to execute the sequential algorithm

To manually run the program, execute in a terminal the following command: go run main.go graph.go file_operation.go strategy.go -file=$FILE -max_depth=$N [-print_sentences] [-seq] [-export_graph], where the flags in brackets are optional, and the variable prefixed with "$" are user-passed parameters.

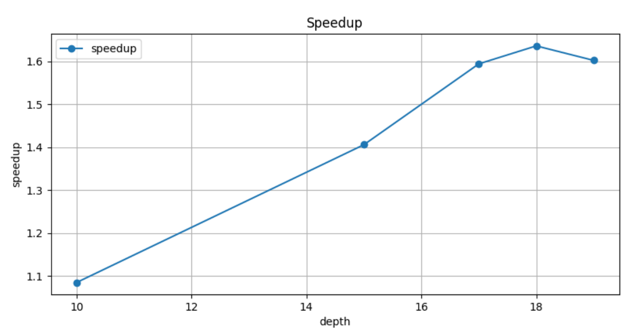
Otherwise the script run.sh is a nice wrapper:
run.sh -file=$FILE -max_depth=$N [-seq] [-print_sentences] [-export_graph]

4 Metrics

The program was tested on a machine with 32GB RAM, and a CPU with 12 logical cores @ 3.7 GHz, measuring execution times of both sequential and parallel version, and the speedup.

The file tested was graphs/long.txt

depth	num_sentences	time_seq[s]	time_par[s]	speedup
10	5,865	0.06	0.06	1.085
15	342,850	6.86	4.88	1.406
17	1,764,478	40.57	25.45	1.594
18	3,955,626	100.53	61.45	1.636
19	8,955,735	267.69	167.08	1.602



We observe a superlinear speedup: this is not surprising since the sequential version is a series of DFS, while the parallel version takes full advantage of the "distributed" nature of the graph structure with message-passing: recursive function calls in the DFS are inherently heavier than message-passing.

The insight is that for this problem, a sequential algorithm is not the best idea to exploit the natural concurrency of the model. Each node is independent and communicates only with its neighbors, so a goroutine-per-node design maps directly to the structure of the text network. This not only enables true parallel exploration of sentences, but also improves locality and reduces overhead compared to a monolithic DFS.