Fantasy Map Story Builder: An LLM Powered Tool for Automatically Generating Fictional World Maps

Teni Asade, Computer Science, James Glenn, Computer Science, Yale University

INTRODUCTION

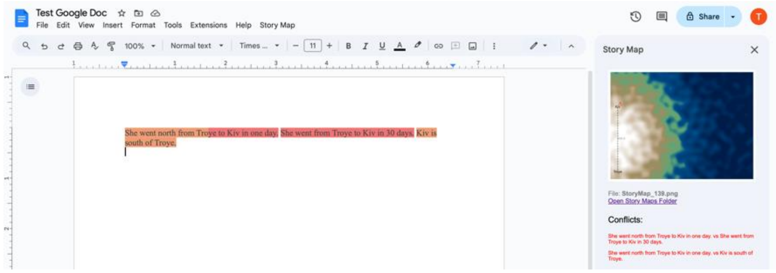
My project is *Story Mapper*, a tool that generates spatial maps from narrative text using Natural Language Processing and constraint satisfaction. The project investigates how fictional narrative descriptions can be translated into formal spatial constraints.

The resulting Google Docs plug-in ingests an author's text, extracts locations and movement, and creates a visual map reflecting the described world.

The final system not only visualizes fictional worlds but also detects and flags spatial inconsistencies (e.g. "*Lok is north of Ayo. Ayo is north of Lok*"), offering both a practical authoring aid and a novel intersection between narrative generation and spatial reasoning.

MOTIVATION

Fantasy writers must construct vast, complex worlds. All the existing digital tools to aid worldbuilding exist outside the writing process. Their use requires an interruption that fragments creative flow and increases cognitive load. This paper asks a simple but significant question: **Can we leverage Natural Language Processing (NLP) and large language models (LLMs) to automatically generate maps from narrative text as authors write?**


Test Google Doc
File Edit View Insert Format Tools Extensions Help Story Map
Share
100% Normal text Times ... 11
Story Map

She went north from Troye to Kiv in one day. She went from Troye to Kiv in 30 days. Kiv is south of Troye.

File: StoryMap_139.png
Open Story Maps Folder

Conflicts:

She went north from Troye to Kiv in one day. vs She went from Troye to Kiv in 30 days.

She went north from Troye to Kiv in one day. vs Kiv is south of Troye.

A screenshot of a Google document where Story Mapper has both generated a map from the text and highlighted conflicts within the text.

METHODOLOGY

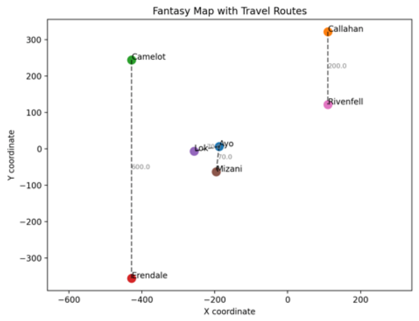
The system consists of two primary components: a back-end program for map generation written in Python and a Google Docs integration for text ingestion, the displaying of the map, and the highlighting of conflicts written in Google Apps Script.

The backend is comprised of three modules:

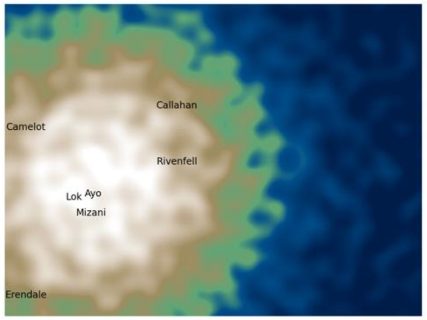
1. The **extractor module** parses narrative sentences to identify key elements such as locations, directions, and distances. It uses spaCy for dependency parsing and named entity recognition, regular expressions for converting descriptions of time into number of days, as well as the word2number Python library to convert written numerals into numeric form. When temporal data are encountered (e.g. "*30 days on foot*"), the system uses a calibrated average travel speed to estimate corresponding spatial distances (e.g. 600 km). Directional and distance conflicts present in the text are also found and stored by this module.

2. The **solver module** formulates a constraint satisfaction problem (CSP) in which each location is a variable with unknown coordinates. The extracted relationships are modeled as constraints. Using NumPy and SciPy, the solver computes a spatial configuration that satisfies the maximum number of constraints and updates the map accordingly.

3. The **visualizer module** transforms the solver's coordinates into a visual terrain map. It first normalizes all points to fit within a fixed image size. The renderer then constructs a heightmap-like terrain by placing Gaussian bumps centered at each normalized point. The module then overlays the location labels, pairwise routes with annotated distances, and visual warnings for flagged conflicts.



An intermediate representation of the different locations described in a text mapped onto a grid as coordinates. This is the result of the extracted locations and relationships being passed into the solver.



The corresponding map output by the visualizer module.

PIPELINE

Let us walk through the processing of some example text: "*The group travelled from Callahan to Rivenfell on foot in 30 days. Callahan is north of Rivenfell.*"

First, relevant information is extracted from the input. Extracted information: ((Rivenfell, Callahan), 30 days), (Rivenfell, Callahan, north). This information is stored in the form of distance and directional constraints such as {('Callahan', 'Rivenfell'): [(600.0, 'real')]} and {('Callahan', 'Rivenfell'): [((0, -1))]}.

These constraints are passed to the solver module. If a solution for the constraint satisfaction problem is found, the resulting coordinates are returned e.g. {'Callahan': (np.float64(11040.299910716261), np.float64(1959.0870424244456))}.

In the event of a conflicting description (e.g. "*Rivenfell is north of Callahan. Rivenfell is south of Callahan*"), the first description is used as the constraint and any other descriptions are stored and marked as conflicts both visually on the map as well as in the document.