MicroTales: Story Planning Benchmark Problems that Scale along Several Dimensions

Version 1.0

Technical Report

October 8, 2025

Stephen G. Ware and Molly Siler
Narrative Intelligence Lab
Department of Computer Science
University of Kentucky

Abstract

This document defines MicroTales, a collection of elements that can be combined into story planning problems of varying size and difficulty. Problems simulate the challenge of telling a structured story in a game where a player typically controls one of the characters and an algorithm controls the rest. MicroTales fills a gap among existing AI benchmark problems by featuring active non-player characters, a wide variety of actions, and the possibility to soft lock the problem. Its purpose is to provide a clearly defined common environment to compare different storytelling algorithms without defining what makes a good story.

Contents

Motivation

Story planning is the challenge of generating a narrative that meets certain requirements on its structure and content. It is difficult because a story planner needs to look ahead and consider many possible stories before finding one that satisfies all of the solution criteria.

There has now been several decades of academic research on narrative planning [13, 4]. Each new research project tends to define one or a few new story planning problems, typically ones that highlight the storytelling features being studied by that project. In a previous effort [12], we collected several of these problems and put them into a common syntax to make it easier to test an algorithm on a variety of "naturally occurring" problems from several authors.

What we feel is missing, and what we are trying to provide with this project, is a single story domain that can generate a wide variety of storytelling problems that vary in size and difficulty. We are inspired by projects like the International Planning Competition [9], AI Gymnasium [2], and TextWorld [2], but we find many of their problems do not capture the storytelling challenges we are interested in studying. We also hope that this project will help to make story planning problems more accessible to non-academic audiences.

This document describes MicroTales, a single storytelling domain based on very simple fantasy Computer Role Playing Games. Each problem has:

- Objects that exist, such as characters and locations.
- Variables that can be assigned values to define the world's current state.
- An initial state that describes the configuration of all elements at the start.
- Goals for each character and for the story itself.
- Actions, which have preconditions that limit when they can occur and effects that define how they change the world state.

We define each of these parts in detail. MicroTales problems are highly modular, with different elements becoming available only when certain mechanics are enabled. For example, the `Undead` mechanic contains characters, items, locations, and actions related to ghosts. When the `Undead` mechanic is enabled, these elements can be used in a problem.

Our definitions do not assume any particular implementation. We want it to be possible to implement MicroTales in everything from the Planning Domain Definition Language [3], used to define classical planning problems, to an AI Gymnasium environment [10], used to define a Markov Decision Process for reinforcement learning, to even non-digital implementations, like a board game.

Design Goals

Game-Like Storytelling Environment: MicroTales is designed to imitate a simple game environment where a player controls one character and an *experience manager* (which might be a human game master or an algorithm) controls the other characters and the environment [7]. One of the central challenges in this kind of environment is telling a good story while allowing the player significant agency to make meaningful choices. Problems like this are often solved interactively, meaning that new solutions are generated during play to adapt to unexpected player actions. This does not mean all implementations of MicroTales need to be interactive. You may find it helpful to generate problems that are solved only once; our goal is for them to reflect the kinds of problems that an experience manager would need to solve interactively. In other words, if an algorithm can solve a wide variety of MicroTales problems, it would probably make a good experience manager if it could be made to run interactively.

Event-Level Storytelling: Stories can be generated at many levels of granularity. The smallest "atomic unit" of story content in MicroTales is an event, which typically has a subject, verb, and usually one or two objects. An event would correspond to roughly one sentence of a written story. Early storytelling systems like Universe [5] and Facade [6] create stories one scene at a time, and an atomic piece of storytelling content corresponds to roughly one paragraph of a written story. Systems based on text generation, like AI Dungeon [11], generate stories one word at a time. Event-level storytelling exists between these two poles.

Scalable along Several Dimensions: It should be possible to generate a wide variety of problems of varying size and difficulty, from trivially easy to nearly impossible. MicroTales can vary in the size of the map, the number of characters, the number of items, the types of actions that are available, and the number and complexity of goals that need to be achieved. The ability to generate a wide variety of problems has several advantages. By generating problems of different sizes and with different features turned on or off, we can make a controlled study of what makes story planning problems easy or hard to solve. Having many problems to train on also encourages learning-based approaches to generalize rather than overfit to a small number of specific problems.

Following Genre Tropes: While MicroTales is meant to allow a wide variety of problems, it is also attempting to follow the genre tropes of simple Medieval computer Role Playing Games. This serves two purposes. First, it allows players

of an interactive MicroTales game to leverage existing genre knowledge and minimize the amount of onboarding required to understand the game's rules. Second, MicroTales is intentionally attempting to represent "naturally occurring" problems. Some things are unlikely or impossible, and some things are over-represented. MicroTales is not meant to represent every conceivable situation or action. It is not meant to be the only type of benchmark problem that storytelling algorithms should be tested on. We are attempting to add to, not restrict, the variety of storytelling problems available for testing.

Active NPCs: One purpose of MicroTales is to fill a gap we perceive in the kinds of games available to test storytelling algorithms, namely story games with active non-player characters (NPCs). Many story games, like the interactive fiction *Zork* [1] and similar games available in the TextWorld environment [2], rely on the player character to take most of the actions in the story. Actions by others are often short and direct reactions to to the player. An ideal MicroTales problem will feature NPCs taking actions motivated by their own goals, ideally in ways that are predictable or at least believable to the player. Solving a MicroTales problem should involve the player anticipating the actions of NPCs and creating a situation that encourages NPCs to act in a way that accomplishes the player's goal.

Wide Variety of Actions: Many planning and reinforcement learning benchmark problems have a limited variety of action types available. For example, the classic *Blocks World* planning problem [8] has only one type of action—moving a block onto another block or onto the table. While it might have a large number of possible actions when we consider all blocks, it has only one *type* of action. Similarly, the *Cart Pole* reinforcement learning problem [10] has only two actions: push the cart left or right. The action can vary in the force applied, but still there is only one or two types of action. We find that storytelling problems tend to involve a large number of action types. MicroTales has many types of actions with different kinds of preconditions and effects.

Soft Locks are Possible: A soft lock means a solvable problem has been put into a state from which no solutions are possible. The potential to soft lock a puzzle makes it challenging because solvers must plan ahead. Solvers that rely on performing random actions—like Monte Carlo random rollouts or random exploration in Markov Decision Process solvers—will need to account for soft locks. Many benchmark problems used to test classical planning algorithms [9] do not allow soft locks. Stories often feature irreversible actions that require careful planning, which is why soft locks are allowed, and even encouraged, in MicroTales.

Deterministic Actions: Actions in MicroTales are deterministic, meaning that when an action occurs we know with 100% certainty what the resulting state will be afterward. Determinism allows a wide variety of search and planning techniques to be applied. Real world problems are not deterministic, but MicroTales is not designed to simulate a real world problem; it is designed to simulate a game in a virtual environment where the experience manager has full control over the virtual world. While a single MicroTales problem is deterministic, the larger problem of experience management in an interactive MicroTales game is usually viewed as nondeterministic since the player may take unexpected actions.

Complex Multi-Agent Interactions: Most problems will feature more than one character. Many existing AI benchmarks feature only a single agent or assume agents are strictly cooperative or competitive. In MicroTales, each character has their own goals, and they should aid or thwart one another to the extent that their goals align or diverge. One of the central challenges of storytelling is ensuring that each character behaves realistically while maintaining a story's central structure. MicroTales provides an environment to compare emergent storytelling via multi-agent systems to centralized storytelling via planning.

No Model of Storytelling Enforced: MicroTales enables the study of what makes a story in a story game believable, interesting, and fun. It tries to avoid enforcing any particular model of storytelling in its rules. Action preconditions are meant to be minimal requirements on when an action would be "physically possible" in the virtual world. For example, one character can only `give` an item to another if they are in the same location and if the giver has the item. The preconditions do not require that the receiver wants the item; it is up to the algorithm and its model of storytelling to determine whether this action makes sense. To study a model of good storytelling, it must be possible to generate bad stories. Many AI benchmark problems define success as reaching a clear goal via a shortest or lowest cost path, but many MicroTales problems will have "solutions" that are bad stories—that is, a sequence of actions that achieves the story goals but is made of actions that are not believable or fun. The quality of a MicroTales solution should not be judged solely on whether it achieves goals or on the number of actions it contains. A solution must be evaluated as a story, but MicroTales does not define what makes a good story. In other words, MicroTales defines a set of objects, actions, and goals so that two researchers can be said to be working in the same domain, but it does not define success in that domain.

Problem Curation Encouraged: MicroTales defines a problem's elements and when they can be used, but it makes no claim that all generated problems are interesting. Many MicroTales problems will have no way to reach a goal state, and many that have a reachable goal state may not have any interesting or good stories that do so. In other words, generating a uniform sample of MicroTales problems is not likely to result in a set of interesting problems. Problem generators will need to apply further criteria and quality checks to ensure problems are interesting. Generating or curating interesting problem sets by hand is also encouraged.

```
name: "Robin Hood"
version: 1.0
mechanics:
- Crime    # Adds jail; allows lawful; allows arrest and release actions
- Marriage # Adds chapel; allows smitten; allows marry action
- Monarchy # Adds castle and Crown; allows ambitious; enthrone action
- Theft    # Adds bandit, Coin, and Sword; allows steal action
- Violence # Adds knight and Sword; allows vengeful; attack action
locations:
- Sherwood : forest   # Sherwood forest
- Nottingham : castle # Nottingham Castle
- Jail : jail         # The Jail
- Abbey : chapel      # Fountains Abbey
paths: # Sherwood Forest connects all locations
- [Sherwood, Nottingham]
- [Sherwood, Jail]
- [Sherwood, Abbey]
characters:
  Robin : bandit   # Robin Hood is a bandit.
  John : noble     # Prince John is a noble.
  Sheriff : knight # The Sheriff of Nottingham is a knight.
  Marian : noble   # Maid Marian is a noble.
initial: # Only need to list values if they are not defaults
- [location, Robin, Sherwood] # Robin starts in Sherwood Forest.
- [right, John, Crown]        # Prince John starts with a Crown.
goals:
- [Robin, greedy, Robin]           # Robin wants coins.
- [Robin, smitten, Robin, Marian]  # Robin wants to marry Marian.
- [John, ambitious, John]          # John wants to be king.
- [John, smitten, John, Marian]    # John wants to marry Marian.
- [John, vengeful, John]           # John wants his enemies dead.
- [Sheriff, ambitious, John]       # The sheriff wants John to be king.
- [Sheriff, lawful]                # The sheriff wants no criminals.
- [Marian, ambitious, Marian]      # Marian wants to be monarch.
- [smitten, Robin, Marian]         # Story goal: Robin marries Marian.
```

Figure 1: An example MicroTales problem in YAML format.

Example Problem

Figure 1 gives an example MicroTales problem that approximates a simple version of the Robin Hood setting. The story goal, given on the last line, is for Robin Hood and Maid Marian to get married. This section describes some example stories that are possible in this problem and why they might be desirable or undesirable. We assume this problem is presented as an interactive game where the player controls Robin Hood.

```
walk(Robin, Nottingham) # Robin goes to Nottingham Castle.
take(Robin, Coin, John) # Robin is greedy, so he wants coins.
attack(Sheriff, Robin)  # The Sheriff of Nottingham hurts Robin.
attack(Sheriff, Robin)  # The sheriff kills Robin.
```

This is a simple but unsuccessful story. Robin robs Prince John, but the Sheriff of Nottingham kills Robin for his crimes. This example does not achieve the story goal, but if the problem is presented as an interactive puzzle it might be a reasonable but unsuccessful attempt by the player to achieve Robin's goals. Suppose the player tries a different approach.

```
walk(Robin, Castle)          # Robin goes to Nottingham Castle.
take(Robin, Crown, John)     # Robins steals John's crown.
enthrone(Robin)              # Robin is crowned king.
arrest(Sheriff, Robin, Jail) # Lawful sheriff throws Robin in jail.
walk(Marian, Sherwood)       # Marian on her way to release Robin.
release(Marian, Jail)        # Marian unlocks the jail.
walk(Robin, Jail, Sherwood)  # Robin on his way to marry Marian.
walk(Robin, Abbey)           # Robin goes to the chapel.
walk(Marian, Abbey)          # Marian goes to the chapel.
marry(Robin, Marian)         # Marian becomes queen by marriage.
```

This story is more interesting. Robin steals the `Crown` and becomes the `monarch`. The storyteller is faced with two possible responses from the sheriff: `attack` like in the previous story, or `arrest` Robin and move him to the `jail`. Killing Robin would end the story before the goal is achieved, but arresting him leaves it open for a happy ending later. Robin can't do anything from the jail, but Marian comes to free him. Marian's plan is to marry Robin and satisfy her

ambitious goal as well as the story goal. We consider this story more interesting because non-player characters take an active role.

While this makes a plausible story, it may not work when presented as a puzzle, since the player might wonder why Robin was allowed to take several actions in a row without any response from the sheriff. Next, we'll consider a story that enforces strict turn-taking, always allowing one player action after one NPC action.

```
enthrone(John, Nottingham)  # John is crowned king.
# The player chooses to wait, passing their turn.
walk(John, Sherwood)        # John on his way to marry.
take(Robin, Coin, John)     # Robin steals John's money.
walk(Marian, Sherwood)      # Marian on her way to marry.
take(Robin, Crown, John)    # Robin steals John's crown.
walk(John, Abbey)           # John on his way to marry.
walk(Robin, Nottingham)     # Robin on his way to be crowned.
walk(Marian, Abbey)         # Marian on her way to marry.
enthrone(Robin)             # Robin is crowned king.
take(Sheriff, Crown, Robin) # Sheriff steals Robin's crown.
walk(Robin, Sherwood)       # Robin on his way to marry.
walk(Sheriff, Sherwood)     # Sheriff on his way to John.
walk(Robin, Abbey)          # Robin on his way to marry.
walk(John, Sherwood)        # John on his way to the sheriff.
marry(Robin, Marian)        # Marian becomes queen by marriage.
```

John starts by achieving his ambitious goal and becoming the monarch. He and Marian then walk in the direction of Fountains Abbey to marry. On the way, John gets robbed by Robin Hood. The player waiting for John to pass by is a prime example of a player anticipating active NPCs. Robin then goes to Nottingham Castle and crowns himself the new monarch before the wedding can take place. The sheriff takes the crown from Robin and goes to Sherwood Forest hoping to give the crown to John. Robin makes his way to Fountains Abbey and marries Marian, achieving the story goal.

In this example, our model of storytelling assumes characters only witness actions that occur in their location (except enthrone, which gets announced to everyone throughout the kingdom). The sheriff is not aware of Robin's crimes, since he was not in Sherwood Forest during the robbery. This model of limited observability is not required. MicroTales does not model character beliefs; that is left to the storyteller, who is free to use this or any other model. For this problem,

we think it makes the characters seems more realistic, and it adds a new dimension to the puzzle by encouraging Robin to avoid witnesses to his crimes.

Given the sheriff does not know about Robin's crimes, it may seem odd that he would `take` the `Crown`. This theft makes him a `criminal` and goes against his `lawful` goal. The sheriff takes the crown because he hopes to return it to John so that John can again take the throne. After returning the crown, the sheriff plans to `arrest` himself, satisfying all his goals, though the story ends before he can complete his plan. Does it make sense for the sheriff to `arrest` himself? This is up to the storyteller.

Notice also how the sheriff and John meet in Sherwood Forest. According to our model, the sheriff saw John leave for the `forest`, but did not see John continue to the Abbey, so the sheriff wrongly believes John is in the forest. John did not see the sheriff go to the forest, so John wrongly believes the sheriff is in the `castle`. Luckily, these two meet by chance in the forest despite their wrong beliefs thanks to some clever coordination by the storyteller.

Definitions

A MicroTales problem is defined by several explicit elements which imply the inclusion of other elements. The explicit elements of a problem are:

- An optional name or title.
- An optional version number, to check compatibility. This is the version of MicroTales, not of the problem itself.
- A list of mechanics, which define what elements are available to use.
- A list of locations, each defined by a name and type.
- A list of paths that connect locations.
- A list of characters, each defined by a name and type.
- A list of initial value assignments to the problem's variables.
- A list of goals for the characters and the story itself.

Figure 1 shows an example problem in YAML format.

Many elements of a problem are not explicitly listed but are implied by what is explicitly listed. For example, each problem defines variables, and the current state of the problem is an assignment of a value to each of its variables. Variables are not explicitly listed because they are implied by the mechanics, locations, and characters. Similarly, actions are not explicitly listed because which actions are available is implied by the other elements.

A problem's elements cannot be created or destroyed. The only thing about a problem that changes over time is the values assigned to its variables. New locations cannot be added to the map, nor can existing locations be removed. New characters cannot be created or removed (though they can be killed when the relevant mechanics are enabled). The items a character is carrying are not first class elements of a problem, so they do not need to be explicitly defined, and they can be created and destroyed (see the section on Constants for how items work).

The following subsections define MicroTales elements in detail. Rules which must be followed are *required*. When something is *recommended*, it means that a problem generator should try to follow those rules when it can, but it may choose to ignore those rules in service of some other design goal. When a definition says something is a *default*, it means that a problem assumes the rule will be followed. If a default is not followed, the problem must explicitly state some other value. For example, a `noble` character starts with a `Coin` in their `left` hand by default. The problem does not need to list this under its initial facts because it will be assumed. If the noble should not start with a coin, the problem should explicitly list their `left` hand as `Null`.

Style and Capitalization Throughout this document, we capitalize specific, individual objects. This includes universal constants like `True`, `False`, and `Null`,

the names of all mechanics, the names of all constants, and the names given to individual objects defined in example problems, like the character name `Robin`. We use lower case for types, which includes all location types, character types, variable names, goal names, and action names. For example, `Robin` is the name of a specific character, so we capitalize his name, but his character type, `bandit`, is a type and is written in lower case. The variable `health(Robin)` has a lower case name because `health` is a type of variables, but the value `Healthy`, which can be assigned to that variable, is a single individual constant and is capitalized.

Mechanics

A mechanic is a group of related problem elements that become available when a mechanic is included in a problem. Mechanics are designed to allow additional types of actions in a problem, but doing so means enabling the locations, characters, and goals that would make those actions possible and relevant.

Figure 1 shows an example list of mechanics. In YAML, the list of mechanics is simply the key `mechanics` followed by a sequence of strings. The list of mechanics is optional and can be omitted, but problems with no mechanics are likely to be uninteresting.

Some mechanics require that other mechanics are also enabled. For example, the `Undead` mechanic requires some way for characters to die, which is enabled by both the `Sickness` and `Violence` mechanics. One of these two must be enabled in order to enable `Undead`. A problem should explicitly list all mechanics that it uses, even those required by other mechanics. For example, a problem that includes `Undead` should also explicitly list `Sickness` or `Violence` or both.

Several problem elements can be enabled by more than one mechanic. For example, several mechanics enable potions of some sort. All of these mechanics enable the `sorcerer` character type, whose special actions revolve around using potions.

Not all of the elements enabled by a mechanic need to be used in the problem, though at least one should be used. We recommend this rule of thumb: if a mechanic could be removed without modifying the problem at all, then it is not really enabled and should not be listed as part of the problem.

`Alchemy`

- Requires `Enchantment`, `Healing`, `Sickness`, `Stealth`, `Teleportation`, or `Undead`.
- Enables the `laboratory` location type.
- Enables the `Flower` item constant.
- Enables the `brew` action.

`Commerce`

- Enables the `market` location type.
- Enables the `merchant` character type.
- Enables the `Coin` item constant.
- Enables the `greedy` goal.
- Enables the `sell` action.

`Crafting`

- Requires `Commerce`, `Crime`, `Monarchy`, `Theft`, or `Violence`.

- Enables the `workshop` location type.
- Enables the `Ore` item constant.
- Enables the `craft` action.

`Crime`

- Enables the `jail` location type.
- Enables the `knight` character type.
- Enables the `criminal` variable type.
- Enables the `Sword` item constant.
- Enables the `chaotic` and `lawful` goals.
- Enables the `arrest`, `jailbreak`, and `release` actions.

`Enchantment`

- Enables the `noble` and `sorcerer` character types.
- Enables the `CharmPotion` item constant.
- Enables the `charm` action.

`Forgiveness`

- Requires `Crime`.
- Enables the `chapel` location type.
- Enables the `cleric` character type.
- Enables the `forgive` and `repent` actions.

`Healing`

- Requires `Sickness` or `Violence`.
- Enables the `cleric` and `sorcerer` character types.
- Enables the `HealPotion` item constant.
- Enables the `heal` action.

`Marriage`

- Enables the `chapel` location type.
- Enables the `spouse` variable type.
- Enables the `Flower` item constant.
- Enables the `smitten` goal type.
- Enables the `marry` action.

`Monarchy`

- Enables the `castle` location type.
- Enables the `noble` character type.
- Enables the `monarch` variable type.
- Enables the `Crown` item constant.

- Enables the `ambitious` goal type.
- Enables the `enthrone` action.

`Sickness`

- Enables the `sorcerer` character type.
- Enables the `health` variable type.
- Enables the `CursePotion` item constant.
- Enables the `hateful` and `loving` goal types.
- Enables the `curse`, `die`, `loot`, `recover`, and `sicken` actions.

`Stealth`

- Enables the `camp` location type.
- Enables the `bandit` and `sorcerer` character types.
- Enables the `visible` variable type.
- Enables the `HidePotion` item constant.
- Enables the `hide` action.

`Teleportation`

- Enables the `sorcerer` character type.
- Enables the `TeleportPotion` item constant.
- Enables the `teleport` action.

`Theft`

- Enables the `bandit` character type.
- Enables the `Coin` and `Sword` item constants.
- Enables the `greedy` goal type.
- Enables the `take` action.

`Undead`

- Requires `Sickness` or `Violence`.
- Enables the `graveyard` location type.
- Enables the `cleric` and `sorcerer` character types.
- Enables the `BanishPotion` and `RaisePotion` item constants.
- Enables the `banish`, `curse`, `raise`, and `rise` actions.

`Violence`

- Enables the `knight` character type.
- Enables the `health` variable type.
- Enables the `Sword` item constant.
- Enables the `hateful` and `loving` goal types.
- Enables the `attack`, `die`, `loot`, and `recover` actions.

Locations and Paths

MicroTales problems take place on a map made of one or more connected locations. A location has a unique name and a type. The type determines what special actions are available in that location. There can be more than one location of the same type on a map as long as they have different names. For example, `Sherwood` and `Mirkwood` might both be locations of type `forest`. We recommend avoiding duplicate locations when possible simply for the sake of variety.

Figure 1 shows an example list of locations. In YAML, the list of locations is the key `locations` followed by a sequence of one-element maps that each associate one unique string name to a string type, where the type is one of those defined below. A sequence of maps is necessary because the order locations are defined in matters. The list of locations is not optional, because a problem must define at least one location.

Four location types are always available: the `cave`, `crossroads`, `forest`, and `village`. These locations are similar in function—characters can `drop` or `pickup` an `item`. The `cave` and `forest` have default `items` waiting to be picked up when some mechanics are enabled.

The order that locations are defined in affects the default starting locations of characters. For example, the `bandit` character type's default starting location is the first `camp` or `crossroads` location that is defined in the problem. If a problem contains both a `camp` and a `crossroads`, the `bandit` will start in whichever one is defined first in the list of locations. Similarly, if the problem has more than one location of type `camp`, the first one will be used. If there are no locations that a character prefers to start in, they default to starting in the first location in the list. These defaults do not apply if a problem explicitly states a character's starting location.

After defining the list of locations, a problem should define how they are connected via paths. A path is a pair of locations. A path goes in both directions. For example, if `Sherwood` and `Nottingham` are connected, then a character can `walk` from one to the other and back again. When defining a path, the two locations can be given in either order. Each location type defines a minimum and maximum number of paths it should have connecting it to other locations.

As shown in Figure 1, in YAML the list of paths is the key `paths` followed by a sequence of pairs of strings that match the names of locations. The list of paths is optional if no locations are connected, though only very small problems will have no paths.

We recommend that locations be placed on a 2 dimensional grid where each location is of equal size and paths always lead North, South, East or West. This is not required, but it makes it easier to visualize MicroTales problems for an

audience.

The existence of paths is important for the `walk` action, but paths are not defined as variables because they do not change (though the `jail` location type can be `locked` to restrict movement). We recommend that all locations be accessible to one another. In other words, it should be possible to `walk` a route from any location to any other.

`camp`

- Only available when `Stealth` is enabled.
- Required to have between 1 and 2 connections.
- Any character can `hide` here.

`castle`

- Only available when `Monarchy` is enabled.
- Required to have between 1 and 4 connections.
- A character can `enthrone` here when holding a `Crown` to become the `monarch`.

`cave`

- Required to have between 1 and 2 connections.
- When `Crafting` is enabled, the `item` at this location is `Ore` by default.
- A character can `drop` or `pickup` an item here.

`chapel`

- Only available when `Forgiveness` or `Marriage` is enabled.
- Required to have between 1 and 3 connections.
- A character can `repent` for their crimes here.
- Two characters can `marry` here.

`crossroads`

- Required to have 2 to 4 connections.
- A character can `drop` or `pickup` an item here.

`forest`

- Required to have 0 to 4 connections.
- A character can `drop` or `pickup` an item here.
- When `Alchemy` is enabled, the `item` at this location is `Flower` by default.
- A `Flower` can `grow` here.

`graveyard`

- Only available when `Undead` is enabled.

- Required to have 0 to 2 connections.
- A `Ghost` can be `raised` or can `rise` here.

`jail`

- Only available when `Crime` is enabled.
- Required to have exactly 1 connection.
- Characters can only `walk` into or out of this location when it is not `locked`.
- The `arrest` action sends a character here and causes it to be `locked`.
- The `jailbreak` and `release` actions unlock this location.

`laboratory`

- Only available when `Alchemy` is enabled.
- Required to have 1 to 2 connections.
- Any character can `brew` a potion from a `Flower` here.

`market`

- Only available when `Commerce` is enabled.
- Required to have 1 to 4 connections.
- Any character except the `merchant` can `sell` an item here to get a `Coin`.

`village`

- Required to have 0 to 4 connections.
- A character can `drop` or `pickup` an item here.

`workshop`

- Only available when `Crafting` is enabled.
- Required to have 1 to 2 connections.
- Any character can `craft` one metal item into another here.

Characters

Characters are agents who act in the story world. Like locations, each character has a unique name and a type. There can be more than one character of the same type in a problem as long as they have different names. We recommend avoiding duplicate character types when possible simply for the sake of variety.

Figure 1 shows an example list of characters. In YAML, the list of characters is the key `characters` followed by a mapping of string names to string types, where the type is one of those defined below. The list of characters is not optional, because a problem must define at least one character.

Character types define what special actions a character can take. They prohibit some character goals and recommend others. Many character types also define a default starting location and starting items. These defaults do not apply if a problem explicitly states some other starting location or starting items.

The only character type that is always available is the `peasant`, which has no special qualities. All other character types are enabled by mechanics.

`bandit`

- Only available when `Stealth` or `Theft` is enabled.
- Starts at the first `camp` or `crossroads` defined in the problem by default.
- Required not to be `lawful`.
- Recommended to be `greedy`.
- Can `hide` at any location, not just a `camp`.
- Can `take` from another character even when unarmed.

`cleric`

- Only available when `Forgiveness`, `Healing`, or `Undead` is enabled.
- Starts at the first `chapel` or `graveyard` defined in the problem by default.
- Required not to be `vengeful` (unless they become a `Ghost`; see Goals).
- Recommended to be `friendly`.
- Can `banish` a `Ghost` even if they are not carrying a `BanishPotion`.
- Can `forgive` a character their crimes.
- Can `heal` a character even if they are not carrying a `HealPotion`.

`knight`

- Only available when `Crime` or `Violence` is enabled.
- Starts at the first `castle` or `market` defined in the problem by default.
- Starts with a `Sword` in their `left` hand by default.
- Recommended to be `ambitious` for a `noble`.
- Recommended to be `lawful`.
- Can `attack` even when not carrying a `Sword`.

`merchant`

- Only available when `Commerce` is enabled.
- Starts at the first `market` or `crossroads` defined in the problem by default.
- Recommended to start with at least one item of any type except a `Coin`.
- Required not to be `kind`.
- Recommended to be `greedy`.
- Cannot `sell` items at the `market`.

`noble`

- Only available when `Enchantment` or `Monarchy` is enabled.
- Starts at the first `castle` or `chapel` defined in the problem by default.
- Starts with a `Coin` in their `left` hand by default.
- Recommended to be `ambitious`.
- Can `charm` another character even when not carrying a `CharmPotion`.

`peasant`

- Starts at the first `village` or `workshop` defined in the problem by default.

`sorcerer`

- Only available when `Enchantment`, `Healing`, `Sickness`, `Stealth`, `Teleportation`, or `Undead` is enabled.
- Starts at the first `laboratory` or `graveyard` defined in the problem by default.
- Recommended to start with at least one potion of any type.
- Recommended to be `vengeful`.
- Can `brew` a potion even when not in a `laboratory`.

Variables and Initial Values

Variables are features of a problem that are assigned one of several possible values. All of a problem's variables with their assigned values define a problem state. Different actions are possible depending on the current state, and actions change the values assigned to variables. Actions are the only way to change variable values.

A problem does not explicitly define its variables; they are implied by the other elements in the problem. For example, when the `Violence` mechanic is enabled the problem must have a `health` variable for every character.

A variable is identified by a name followed by zero to many arguments. In the example above, `health` is the name of the variable, and it has one argument, the name of the character. If a problem defines two characters named `Robin` and `Marian`, then it will have two `health` variables, one for each of them. In text, we would write those variables as `health(Robin)` and `health(Marian)`, though in YAML we use a different style, as discussed below.

Some variables have more than one argument, and the order of arguments matters. For example, every problem tracks the `relationship` between each pair of characters, so our example problem will need the variables `relationship(Robin, Marian)` and `relationship(Marian, Robin)`. The first variable tracks how `Robin` regards `Marian`, and the second tracks how `Marian` regards `Robin`. Relationships are not symmetric, so these variables can have different values.

The variable definitions in this section begin with the variable name, followed by one or more parameters in parentheses, followed by a colon and a description of the variable's possible values. Sometimes a value is simply Boolean, meaning `True` or `False`. Sometimes a value is a type, like a location. For example, every character has a `location` variable, and its possible values are the names of every location defined in the problem, plus `Null`to represent a situation where the character is nowhere on the map. Sometimes a variable's value is a set of pre-defined constants. For example, a `relationship` variable can have three possible values: `Friend`, `Null`, and `Enemy`, which mean that the first character considers the second a friend, has no relationship, or considers the second an enemy respectively. Many variables have one of a pre-defined list of constants as their possible values. All constants are defined in the next section.

All variables have a default initial value they will be assigned, but these defaults are ignored if the problem explicitly defines a variable's initial value. Figure 1 shows an example list of initial values. In YAML, this list is the key `initial` followed by a sequence of sequences. Each initial value sequence includes the variable name (defined below), its arguments, and finally the initial value to assign

to that variable. For example, if we want `Robin` to start with a `Sword`, we would write:

```
[left, Robin, Sword]
```

This sequence has three parts: the variable name, the variable's arguments (in this case, one argument), and finally the value to assign to the variable—i.e. the item to put in Robin's `left` hand. In text, we would write this value assignment as:

```
left(Robin) = Sword
```

Defining the initial value of a variable with multiple arguments looks like this:

```
[relationship, John, Robin, Enemy]
```

Some variables, like the `relationship` between each character, are counter to MicroTales's design principle of modeling only necessary physical elements of the world and making no commitments to any particular model of storytelling. These variables exist because they are needed to define some of the goals that characters and stories have. Whenever possible, we try to avoid using them to limit when actions are possible.

`criminal (` *character* **`):`** Boolean

- Only available when `Crime` is enabled.
- Tracks whether the *character* has committed any crimes.
- Default value is `False`.
- The `chaotic` goal is satisfied when characters are criminals and not in a `jail`.
- The `lawful` goal is satisfied when no characters are criminals or when all criminals are in a `jail`.
- It is a crime to `arrest` a character who is not a `criminal`.
- It is a crime to `release` prisoners from the `jail`.
- When `Sickness` or `Undead` is enabled, it is a crime to `curse` a character who is not a criminal.
- When `Theft` is enabled, it is a crime to `take` an item from a living character who is not a criminal.
- When `Violence` is enabled, it is a crime to `attack` a character who is not a criminal.

`health (` *character* **`):`** {`Healthy`, `Hurt`, `Dead`, `Ghost`}

- Only available when `Sickness` or `Violence` is enabled.
- Tracks the capabilities of the *character* to do certain actions.
- Default value is `Healthy`, meaning a character can act normally.
- A `Hurt` character can do most actions but cannot `walk`.
- A `Dead` character cannot do most actions.
- When `Undead` is enabled, a `Ghost` character cannot do most actions, but they can `curse` and `walk`.

- The `loving` goal is satisfied by a character being `Healthy`.
- The `hateful` goal is satisfied by a character being `Dead`.
- The `attack` and `curse` actions degrades a target character's health (`Healthy` to `Hurt` or `Hurt` to `Dead`).
- The `sicken` action causes a `Healthy` character to become `Hurt`.
- The `die` action causes a `Hurt` character to become `Dead`.
- The `heal` and `recover` actions causes a `Hurt` character to become `Healthy`.
- The `raise` and `rise` actions cause a `Dead` character to become a `Ghost` and sets their `location` to a `graveyard`.
- The `banish` action causes a `Ghost` to become `Dead`.

`item (` *location* **`) :`** the name of any item or `Null`

- The *location* must be a `cave`, `crossroads`, `forest`, or `village`.
- Tracks what item is lying on the ground at the *location*.
- Default value is `Null`, which means there is no item at this location.
- Characters can `drop` items they are holding.
- Characters can `pickup` a location's item.
- When `Crafting` is enabled, `Ore` is in every `cave` by default.
- When `Alchemy` is enabled, a `Flower` is in every `forest` by default.
- When `Alchemy` is enabled, a `Flower` can `grow` in a `forest`.

`left (` *character* **`) :`** any item or `Null`

- The item a character is holding in their left hand.
- Default value is usually `Null`, which means this hand is empty, but some character types start with items in their hands by default (see Characters).
- A character's `left` and `right` hands behave the same way.

`location (` *character* **`) :`** the name of any location or `Null`

- The *character*'s current position on the map.
- The value `Null`means the character is nowhere on the map.
- Character types specify default locations. When the specified locations are not available, the default value is the first location defined in the problem.
- The `homesick` goal is satisfied by a character being at a certain location.
- Many actions require that a character be at a certain location or in the same location as another character they are interacting with.
- Character locations can change via the `arrest`, `teleport`, and `walk` actions.

`locked (` *jail* **`) :`** Boolean

- Only available when `Crime` is enabled.
- The argument must be a location of type `jail`.
- Tracks whether the `jail` is locked or unlocked.

- Default value is `False`.
- When a *jail* is locked, characters cannot `walk` to or from it.
- The `arrest` action moves a character to a jail and locks the jail.
- The `jailbreak` and `release` actions unlock a jail.

`monarch ( ) :` the name of any character or `Null`

- Only available when `Monarchy` is enabled.
- Tracks which character is the ruler.
- Default value is `Null`, meaning no character is currently the monarch.
- The `ambitious` goal is satisfied by a certain character becoming the monarch.
- When holding a `Crown` in a `castle`, a character can `enthrone` themselves to become the monarch.

`relationship (` *character*, *other* **`) :`** {`Friend`, `Null`, `Enemy`}

- How the first *character* views the *other* character. The arguments must be different characters.
- Default value is `Null`, which means the *character* has no relationship or a neutral relationship to the *other*.
- The value `Friend` means the *character* views the *other* positively.
- The value `Enemy` means the *character* views the *other* negatively.
- Relationships are not symmetric. The *other*'s relationship to the *character* can be different.
- The `friendly` goal is satisfied by a character regarding other `Healthy` characters as `Friend`.
- The `kind` goal is satisfied by other characters regarding a character as a `Friend`.
- The `spiteful` goal is satisfied by other characters regarding a character as an `Enemy`.
- The `vengeful` goal is satisfied by a character not regarding other living characters as `Enemy`.
- The `charm`, `give`, `heal`, and `release` actions can cause a relationship to improve (`Enemy` to `Null`or `Null`to `Friend`).
- The `arrest`, `attack`, `banish`, `curse`, and `take` actions can cause a relationship to degrade (`Friend` to `Null`or `Null`to `Enemy`).

`right (` *character* **`) :`** any item or `Null`

- The item a character is holding in their right hand.
- Default value is usually `Null`, which means this hand is empty, but some character types start with items in their hands by default (see Characters).
- A character's `left` and `right` hands behave the same way.

`spouse (` *character* **`) :`** the name of any character or `Null`

- Only available when `Marriage` is enabled.
- Tracks who is married to the *character*.
- Default value is `Null`, meaning the *character* is not married.
- Marriage is symmetric. When character `A` has spouse `B`, then character `B` must also have spouse `A`.
- The `smitten` goal is satisfied when a pair of characters are one another's spouses.
- The `marry` action causes two characters to become one another's spouses.
- When characters die via the `attack`, `curse`, or `die` actions, both members of a marriage have their spouse set to `Null`.

`visible (` *character* **`) :`** Boolean

- Only available when `Stealth` is enabled.
- Tracks whether a character can be seen by others.
- Default value is `True`.
- Invisible characters can usually act normally, but actions usually cannot be done to invisible characters.
- Characters can become invisible via the `hide` action.
- Characters become visible when they `walk` to a new location.

Constants

Constants are special values that can be assigned to some variables. Constants are not explicitly defined; they are implied by which mechanics are enabled. For example, the `Ghost` constant is one possible value for a `health` variable, and it represents a character who has been raised from the dead. However, if the `Undead` mechanic is not enabled, this constant is not available, even if the other constants that can be assigned to `health` variables are.

One use of constants is the items that characters carry and use in the story. MicroTales elements cannot be created or removed, but items are not first class elements. Each character has a `left` and `right` hand variable, and the possible values that can be assigned are the item constants or `Null`to represent holding nothing. This means each character can hold up to two items, one in each hand. Consider the `sell` action, where a character trades an item for a `Coin`. The action's precondition requires that the character have the item they are selling in either their `left` or `right` hand. The effects of the action replace that hand's value with a `Coin` constant, effectively destroying the original item and creating a new `Coin` item.

`BanishPotion`

- Only available when `Undead` is enabled.
- This is a potion item that can be held in a character's `left` or `right` hand.
- Allows any character to `banish` a `Ghost`, setting its `health` to `Dead`.
- When `Alchemy` is enabled, it can be created from a `Flower` via the `brew` action.

`CharmPotion`

- Only available when `Enchantment` is enabled.
- This is a potion item that can be held in a character's `left` or `right` hand.
- Allows any character to `charm` another, causing the target's `relationship` toward the charmer to improve.
- When `Alchemy` is enabled, it can be `brew`ed from a `Flower`.

`Coin`

- Only available when `Commerce` or `Theft` is enabled.
- This is a metal item that can be held in a character's `left` or `right` hand.
- The `greedy` goal is satisfied by having coins.
- Any item can be converted into a coin via the `sell` action at the `market`.
- When `Crafting` is enabled, it can be be `crafted` from any other metal item or into any other metal item.

`Crown`

- Only available when `Monarchy` is enabled.
- This is a metal item that can be held in a character's `left` or `right` hand.
- Needed for a character to `enthrone` themselves in a `castle` to become the `monarch`.
- When `Crafting` is enabled, it can be be `crafted` from any other metal item or into any other metal item.

`CursePotion`

- Only available when `Sickness` or `Undead` is enabled.
- This is a potion item that can be held in a character's `left` or `right` hand.
- Allows any character to `curse` another, lowing the target's `health`.
- When `Alchemy` is enabled, it can be `brewed` from a `Flower`.

`Dead`

- Only available when `Sickness` or `Violence` is enabled.
- Assigned to a character's `health` to indicate they are not alive and cannot do most actions.
- The `attack`, `curse`, and `die` actions can kill a character.
- The `raise` and `rise` actions can make a dead character a `Ghost`.

`Enemy`

- Assigned to a `relationship` to indicate the first character regards the second negatively.
- The `spiteful` goal is satisfied by other characters regarding a character this way.
- The `vengeful` goal is satisfied by a character not regarding others this way, or by a character's enemies being dead.
- The `arrest`, `attack`, `banish`, `curse`, and `take` actions can cause a `relationship` set to `Null`to change to this value.

`Flower`

- Only available when `Alchemy` or `Marriage` is enabled.
- This is an item that can be held in a character's `left` or `right` hand.
- Can `grow` in a `forest`.
- When `Alchemy` is enabled, this `item` is in every `forest` by default.
- When `Alchemy` is enabled, it can be `brewed` into any potion.

`Friend`

- Assigned to a `relationship` to indicate the first character regards the second positively.

- The `friendly` goal is satisfied by a character regarding others this way, or by a character's friends becoming healthy.
- The `kind` goal is satisfied by other characters regarding a character this way.
- The `charm`, `give`, `heal`, and `release` actions can cause a `relationship` set to `Null`to change to this value.

`Ghost`

- Only available when `Undead` is enabled.
- Assigned to a character's `health` to indicate they are a ghost which can only take the `curse` and `walk` actions.
- The `rise` and `raise` actions can make a `Dead` character into a ghost.
- The `banish` action can make a ghost `Dead`.

`HealPotion`

- Only available when `Healing` is enabled.
- This is a potion item that can be held in a character's `left` or `right` hand.
- Allows any character to `heal` another, changing their `health` to `Healthy`.
- When `Alchemy` is enabled, it can be `brewed` from a `Flower`.

`Healthy`

- Only available when `Sickness` or `Violence` is enabled.
- Assigned to a character's `health` to indicate they are able to take most actions.
- The `heal` and `recover` actions can make a `Hurt` character healthy.
- The `attack`, `curse`, and `sicken` actions can make a healthy character `Hurt`.

`HidePotion`

- Only available when `Stealth` is enabled.
- This is a potion item that can be held in a character's `left` or `right` hand.
- Allows any character to `hide`, changing `visible` to `False`.
- When `Alchemy` is enabled, it can be `brewed` from a `Flower`.

`Hurt`

- Only available when `Sickness` or `Violence` is enabled.
- Assigned to a character's `health` to indicate they are able to take most actions but cannot `walk`.
- The `attack`, `curse`, and `sicken` actions can make a `Healthy` character hurt.
- The `attack`, `curse`, and `die` actions can make a hurt character `Dead`.

`Ore`

- Only available when `Crafting` is enabled.

- This is a metal item that can be held in a character's `left` or `right` hand.
- When `Crafting` is enabled, this `item` is in every `cave` by default.
- When `Crafting` is enabled, it can be be `crafted` from any other metal item or into any other metal item.

`RaisePotion`

- Only available when `Undead` is enabled.
- This is a potion item that can be held in a character's `left` or `right` hand.
- Allows any character to `raise` a `Dead` character into a `Ghost`.
- When `Alchemy` is enabled, it can be `brewed` from a `Flower`.

`Sword`

- Only available when `Crime`, `Theft`, or `Violence` is enabled.
- This is a metal item that can be held in a character's `left` or `right` hand.
- Most characters need this to `arrest` and `attack` others.
- Most characters need this to `take` an item when the victim is `Healthy`.
- When `Crafting` is enabled, it can be be `crafted` from any other metal item or into any other metal item.

`TeleportPotion`

- Only available when `Teleportation` is enabled.
- This is a potion item that can be held in a character's `left` or `right` hand.
- Allows any character to `teleport` to any location.
- When `Alchemy` is enabled, it can be `brewed` from a `Flower`.

Goals

Goals are objectives. There are two kinds of goals in MicroTales, *character goals* and *story goals*. Character goals are personal objectives that individual characters want to achieve. Story goals are objectives that the storyteller should try to achieve with the story as a whole.

Similar to variables, a goal is one of the names define below followed by zero to many arguments. Figure 1 shows an example list of goals. In YAML, it is the key `goals` followed by a sequence of sequences. If the sequence is a character goal, the first element is the name of one of the problem's characters. If the sequence is a story goal, this name is omitted. The next elements in both types are the goal name (defined below), followed by the goal's arguments.

For example, this character goal in YAML format says that `John` wants `John` to be the `monarch`:

```
[John, ambitious, John]
```

The first part, `John`, specifies which character has this goal. The rest specifies the goal `ambitious(John)`, which is the objective that `John` be the `monarch`. Characters can also have goals for other characters:

```
[Sheriff, ambitious, John]
```

The goal of `ambitious(John)` is the same, but it is held by the `Sheriff`. The `Sheriff` also wants `John` to be `monarch`.

To make this a story goal instead of a character goal, we simply omit the character from the start of the sequence:

```
[ambitious, John]
```

This means that `ambitious(John)` is a goal for the story as a whole rather than any particular character. The storyteller should try to tell a story where `John` becomes the `monarch`.

Two character goals always apply and do not need to be explicitly listed. When `Sickness` or `Violence` is enabled, all characters prefer being `Healthy` over `Hurt` and prefer `Hurt` over `Dead`. Second, if `Undead` is enabled, all `Ghosts` are `vengeful`. This means they want to kill any character who is an `Enemy`.

Goals are persistent, meaning they do not go away once they are achieved. For example, when a character is `greedy`, they want `Coins`. Once both of their hands are holding `Coins`, they cannot have more, but they do not stop being `greedy`. If they should ever lose one of their coins, they will again want more.

MicroTales does not define the relative importance of one goal to another, and when a goal can be achieved in multiple ways, it only defines ordinal preferences. For example, if a character is both `greedy` and `friendly`, MicroTales does not specify whether it is more important to that character to have `Coins` or `Friends`; this is up to the storyteller. Similarly, a `friendly` character would prefer to have

2 `Friends` over 1 `Friend`, but it is not necessarily twice as good to have 2 friends than it is to have 1. Again, this is left up to the storyteller.

MicroTales tries to avoid imposing any particular model of what makes a good story. Some good stories may not achieve goals, and stories that achieve goals are not necessarily good stories. If goals can be ignored, why define them at all? First, MicroTales is meant to study the challenges of story planning. Goals provide constraints that make planning a story challenging because the storyteller needs to look ahead to reason about what is and is not possible when deciding what actions to take. Second, in an interactive setting, goals are what will be communicated to players about what the characters want and what the objectives are for the story. Stories are meant to involve multiple active characters, so anticipating how non-player characters will act requires information about what they want.

`ambitious (` *character* **`)`**

- Only available when `Monarchy` is enabled.
- Achieved when *character* is the `monarch`.
- When `Marriage` is enabled, this goal is also achieved when *character*'s `spouse` is the `monarch`.

`chaotic ( )`

- Only available when `Crime` is enabled.
- Achieved by maximizing the number of characters who are `criminals`, are not `Dead`, and free (not in a `jail`).
- Opposed to `lawful`.
- This goal improves each time an additional character meets the above requirements. Typically this happens when a living free character commits a crime, but it can also happen when a criminal leaves a `jail` or is `raised` from the dead. Potential crimes include the `arrest`, `attack`, `curse`, `release`, and `take` actions. Characters can leave a `jail` via the `teleport` action or the `walk` action if the jail is not `locked`, perhaps because of the `jailbreak` or `release` actions. Characters can return from the dead via the `raise` and `rise` actions.
- This goal worsens each time a character stops meeting the above requirements. This can happen when a character's crimes are erased, when a criminal moves to a `jail`, or when a criminal dies. Crimes are removed via the `forgive` and `repent` actions. Characters can move to a `jail` via the `arrest`, `teleport`, and `walk` actions. Characters can die via the `attack`, `curse`, and `die` actions.

`friendly (` *character* **`)`**

- Achieved by maximizing the number of other characters who *character* regards as a `Friend` and whose `health` is `Healthy`. In other words, this goal is achieved by maximizing the number of *other* characters for whom `relationship(character, other) = Friend` and `health(other) = Healthy`.
- Opposed to `vengeful`.
- This goal improves each time *character* regards a new `Healthy` character as a `Friend` or someone *character* regards as a `Friend` becomes `Healthy`. Relationships can improve via the `charm`, `give`, `heal`, or `release` actions. Characters can become `Healthy` via the `heal` and `recover` actions.
- This goal worsens each time *character* stops regarding a `Healthy` character as a `Friend` or someone *character* regards as a `Friend` stops being `Healthy`. Relationships can degrade via the `arrest`, `attack`, `banish`, `curse`, and `take` actions. Characters can stop being `Healthy` via the `attack`, `curse`, and `sicken` actions.

`greedy (` *character* `)`

- Only available when `Commerce` or `Theft` is enabled.
- Achieved by maximizing the number of `Coin` items *character* is holding.

`hateful (` *character* `)`

- Only available when `Sickness` or `Violence` is enabled.
- Achieved when *character*'s `health` is not `Healthy`.
- Opposed to `loving`.

`homesick (` *character*, *location* `)`

- Achieved when *character*'s `location` is *location*.

`kind (` *character* `)`

- Achieved by maximizing the number of other characters who regard *character* as a `Friend`. In other words, this goal is achieved by maximizing the number of *other* characters for whom `relationship(other, character) = Friend`.
- Opposed to `spiteful`.
- This goal improves each time an additional other character regards *character* a `Friend`. Relationships can improve via the `charm`, `give`, `heal`, or `release` actions.
- This goal worsens each time another character stops regarding *character* as a `Friend`. Relationships can degrade via the `arrest`, `attack`, `banish`, `curse`, and `take` actions.

`lawful ( )`

- Only available when `Crime` is enabled.
- Achieved by maximizing the number of characters who are not `criminals` or who are `criminals` but are `Dead` or in a `jail`.
- Opposed to `chaotic`.
- This goal improves each time an additional character meets the above requirements. This can happen when the `forgive` or `repent` actions make the character no longer a criminal, when a criminal moves to a `jail`, or when a criminal dies.
- This goal worsens each time a character stops meeting the above requirements. Typically this happens when a living free character commits a crime (potential crimes include the `arrest`, `attack`, `curse`, `release`, and `take` actions), but it can also happen when a criminal leaves a `jail` or returns from the dead via the `raise` or `rise` actions.

`loving (` *character* `)`

- Only available when `Sickness` or `Violence` is enabled.
- Achieved when *character*'s `health` is `Healthy`.
- Opposed to `hateful`.

`smitten (` *character*, *spouse* `)`

- Only available when `Marriage` is enabled.
- Achieved when *character*'s `spouse` is *spouse*.

`spiteful (` *character* `)`

- Achieved by maximizing the number of other characters who regard *character* as an `Enemy`. In other words, this goal is achieved by maximizing the number of *other* characters for whom `relationship(other, character) = Enemy`.
- Opposed to `kind`.
- This goal improves each time an additional other character regards *character* an `Enemy`. Relationships can degrade via the `arrest`, `attack`, `banish`, `curse`, and `take` actions.
- This goal worsens each time another character stops regarding *character* as an `Enemy`. Relationships can improve via the `charm`, `give`, `heal`, or `release` actions.

`vengeful (` *character* `)`

- Achieved by maximizing the number of other characters who *character* does not regard as an `Enemy` or whose `health` is `Dead`. In other words,

this goal is achieved by maximizing the number of *other* characters for whom `relationship(character, other)` ≠ `Enemy` or `health(other)` = `Dead`.

- Opposed to `friendly`.
- This goal improves each time *character* stops regarding another non-`Dead` character as an `Enemy` or someone that *character* regards as an `Enemy` becomes `Dead`. Relationships can improve via the `charm`, `give`, `heal`, or `release` actions. Characters can die via the `attack`, `curse`, and `die` actions.
- This goal worsens each time *character* regards another non-`Dead` character as an `Enemy` or someone that *character* regards as an `Enemy` stop being `Dead`. Relationships can degrade via the `arrest`, `attack`, `banish`, `curse`, and `take` actions. Characters can stop being `Dead` via the `raise` and `rise` actions.

Actions

Actions are events that happen in a story. An action has preconditions that limit when it can happen and effects that modify the world state.

A problem does not explicitly define actions; they are implied by the other elements in the problem. Any time an action is enabled by a mechanic it is available in a story. For example, the `enthrone` action is always available in every story where `Monarchy` is enabled, though it can only happen when a character is in a `castle` and holding a `Crown`.

Like variables and goals, actions are parameterized. The list of parameters are given in parentheses after the name.

An action's preconditions are meant as minimal restrictions for an action to seem believable based on the world state. Action preconditions do not attempt to restrict when an action makes narrative sense. For example, it probably does not make narrative sense for a `greedy` character to `give` away a `Coin`, but MicroTales does not prevent this because one purpose of MicroTales is to study different models of when actions make sense. Ideally, a character's actions are determined by their character goals and the story goals, though this is not required, and it is not enforced by preconditions. To study the generation of good stories, it must be possible to generate bad stories.

Some actions have an active version that is done by a character and a passive version that simply happens. For example, when a character is `Hurt`, they can be made `Healthy` again via by the `heal` action, which is actively done by a character, but they can also simply get better via the `recover` action. Actions not taken by characters provide flexibility in solving problems, but may be perceived as a *deus ex machina*.

For brevity, we list preconditions and effects on all variables, even if they are not used in a problem. For example, many actions require that a character not be `Dead`, but this only applies if `health` variables exist. If the `Sickness` or `Violence` mechanics are not enabled, `health` variables do not exist, so preconditions and effects on those variables are ignored.

We use a few common shorthand phrases in action preconditions:

- "A character is alive," means the character's `health` variable is `Healthy` or `Hurt`.
- "A character is holding an *item*," means that character's `left` or `right` hand variable is set to that item constant.
- "A character has an empty hand," means that character's `left` or `right` hand variable is set to `Null`.
- "A character is now holding one fewer *item*" means that one of their hands that was previously *item* is now `Null`. In other words, if `left` was *item*, then

`left` is now `Null`. If `left` was not *item* and `right` was *item*, then `right` is now `Null`. (Note it is possible for a character to be holding two copies of the same item; only one is removed.)
- "A character is now holding one additional *item*" means that one of their hands that was previously `Null`is now *item*.

The only actions that always available in every problem are basic movement, the `walk` action, and basic item interactions, the `pickup`, `drop`, `give`, and `trade` actions.

`arrest (` *character*, *target*, *jail* `)`

Preconditions:
- Only available when `Crime` is enabled.
- The *jail* must be any location of type `jail`.
- The *character* and *target* character may be the same.
- The *character* is alive.
- The *character* is holding a `Sword`.
- The *target* character is alive.
- The *target* is `visible`.
- The *target* is not in a `locked jail`.
- The *character* and *target* are at the same `location`.

Effects:
- The *target*'s `location` is now the *jail*.
- The *jail* is now `locked`.
- If the *target* is not a `criminal`, the *character* is now a `criminal`.
- If the `relationship` between *target* and *character* was `Friend`, it is now `Null`.
- If the `relationship` between *target* and *character* was `Null`, it is now `Enemy`.

`attack (` *character*, *target* `)`

Preconditions:
- Only available when `Violence` is enabled.
- The *character* and *target* character may be the same.
- The *character* is alive.
- The *character* is holding a `Sword` or is a `knight`.
- The *target* character is alive.
- The *target* is `visible`.
- The *character* and *target* are at the same `location`.

Effects:
- If the *target* was `Healthy`, they are now `Hurt`.
- If the *target* was `Hurt`, they are now `Dead`.
- If the *target* is not a `criminal`, the *character* is now a `criminal`.

- If the `relationship` between *target* and *character* was `Friend`, it is now `Null`.
- If the `relationship` between *target* and *character* was `Null`, it is now `Enemy`.
- If the *character*'s `spouse` was *other*, then *character*'s spouse is now `Null`and *other*'s `spouse` is now `Null`.

`banish (` *character*, *target* **`)`**

Preconditions:

- Only available when `Undead` is enabled.
- The *character* is alive.
- The *character* is holding a `BanishPotion` or is a `cleric`.
- The *target* character is a `Ghost`.
- The *character* and *target* are at the same `location`.

Effects:

- The *target* is now `Dead`.
- The *target*'s `location` is now `Null`.
- If *character* is not a `cleric`, they are now holding one fewer `BanishPotion`.
- If the `relationship` between *target* and *character* was `Friend`, it is now `Null`.
- If the `relationship` between *target* and *character* was `Null`, it is now `Enemy`.

`brew (` *character*, *potion* **`)`**

Preconditions:

- Only available when `Alchemy` is enabled.
- The *potion* is a potion item constant.
- The *character* is alive.
- The *character* is holding a `Flower`.
- The *character* is in a `laboratory` or is a `sorcerer`.

Effects:

- The *character* is now holding one fewer `Flower`.
- The *character* is now holding one additional *potion*.

`charm (` *character*, *target* **`)`**

Preconditions:

- Only available when `Enchantment` is enabled.
- The *character* and *target* character are different.
- The *character* is alive.
- The *character* is holding a `CharmPotion` or is a `noble`.
- The *target* character is alive.
- The *target* character is `visible`.
- The *character* and *target* are at the same `location`.

Effects:

- If the `relationship` between *target* and *character* was `Enemy`, it is now `Null`.
- If the `relationship` between *target* and *character* was `Null`, it is now `Friend`.
- If *character* is not a `noble`, they are now holding one fewer `CharmPotion`.

`craft (` *character*, *material*, *item* `)`

Preconditions:

- Only available when `Crafting` is enabled.
- The *material* is a metal item constant.
- The *item* is a metal item constant.
- The *material* and *item* are different.
- The *character* is alive.
- The *character* is holding *material*.
- The *character* is at a `workshop`.

Effects:

- The *character* is no longer holding the *material*.
- The *character* is now holding the *item*.

`curse (` *character*, *target* `)`

Preconditions:

- Only available when `Sickness` or `Undead` is enabled.
- The *character* and *target* character may be the same.
- The *character* is alive and holding a `CursePotion`, or the *character* is a `Ghost`.
- The *target* is alive.
- The *target* is `visible`.
- The *character* and *target* are at the same `location`.

Effects:

- If the *target* was `Healthy`, they are now `Hurt`.
- If the *target* was `Hurt`, they are now `Dead`.
- If the *target* is not a `Ghost`, they are now holding one fewer `CursePotion`.
- If the *target* is not a `criminal`, the *character* is now a `criminal`.
- If the `relationship` between *target* and *character* was `Friend`, it is now `Null`.
- If the `relationship` between *target* and *character* was `Null`, it is now `Enemy`.
- If the *target*'s `spouse` was *other*, then *target*'s spouse is now `Null`and *other*'s `spouse` is now `Null`.

`die (` *character* `)`

Preconditions:

- Only available when `Sickness` or `Violence` is enabled.
- The *character* is `Hurt`.

Effects:

- The *character* is now `Dead`.
- The *character* is now `visible`.
- If the *character*'s `spouse` was *other*, then *character*'s spouse is now `Null`and *other*'s `spouse` is now `Null`.

`drop (` *character*, *item* `)`

Preconditions:

- The *item* is an item constant.
- The *character* is alive.
- The *character* is holding the *item*.
- The *character*'s `location` is a `cave`, `crossroads`, `forest`, or `village`.

Effects:

- The *character* is no longer holding the *item*.
- The `item` at the *character*'s `location` is now *item*. (Note: This may replace and permanently "destroy" the previous `item` at this `location`.)

`enthrone (` *character* `)`

Preconditions:

- Only available when `Monarchy` is enabled.
- The *character* is alive.
- The *character* is holding a `Crown`.
- The *character*'s `location` is a `castle`.

Effects:

- The `monarch` is now *character*.

`forgive (` *character*, *target* `)`

Preconditions:

- Only available when `Forgiveness` is enabled.
- The *character* and *target* character may be the same.
- The *character* is alive.
- The *character* is a `cleric`.
- The *target* is alive.
- The *target* is `visible`.
- The *target* is a `criminal`.
- The *character* and *target* are at the same `location`.

Effects:

- The *target* is no longer a `criminal`.

`give (` *character*, *item*, *target* `)`

Preconditions:

- The *character* and *target* character are different.

- The *item* is an item constant.
- The *character* is alive.
- The *character* is `visible`.
- The *character* is holding the *item*.
- The *target* is alive.
- The *target* is `visible`.
- The *target* has an empty hand.
- The *character* and *target* are at the same `location`.

Effects:

- The *target* now has one additional *item*.
- The *character* now has one fewer *item*.
- If the `relationship` between *target* and *character* was `Enemy`, it is now `Null`.
- If the `relationship` between *target* and *character* was `Null`, it is now `Friend`.

`grow (` *forest* `)`

Preconditions:

- Only available when `Alchemy` or `Marriage` is enabled.
- The *forest* is a `forest`.
- The *forest*'s `item` is `Null`.

Effects:

- The *forest*'s `item` is now `Flower`.

`heal (` *character*, *target* `)`

Preconditions:

- Only available when `Healing` is enabled.
- The *character* and *target* character may be the same.
- The *character* is alive.
- The *character* is holding a `HealPotion` or is a `cleric`.
- The *target* is `Hurt`.
- The *target* is `visible`.
- The *character* and *target* are at the same `location`.

Effects:

- The *target* is now `Healthy`.
- If the *character* is not a `cleric`, the *character* now has one fewer `HealPotion`.
- If the `relationship` between *target* and *character* was `Enemy`, it is now `Null`.
- If the `relationship` between *target* and *character* was `Null`, it is now `Friend`.

`hide (` *character* `)`

Preconditions:

- Only available when `Stealth` is enabled.
- The *character* is alive.
- The *character* is `visible`.
- The *character* is at a `camp` or is a `bandit`.

Effects:

- The *character* is no longer `visible`.

`jailbreak (` *jail* `)`

Preconditions:

- Only available when `Crime` is enabled.
- The *jail* is a `jail`.
- The *jail* is `locked`.

Effects:

- The *jail* is no longer `locked`.

`loot (` *character*, *item*, *target* `)`

Preconditions:

- Only available when `Sickness` or `Violence` is enabled.
- The *item* is an item constant.
- The *character* is alive.
- The *target* is `Dead`.
- The *character* and *target* are at the same `location`.

Effects:

- The *character* is now holding one additional *item*.
- The *target* is now holding one fewer *item*.

`marry (` *character*, *spouse* `)`

Preconditions:

- Only available when `Marriage` is enabled.
- The *character* and *spouse* character are different.
- The *character* is alive.
- The *character* is `visible`.
- The `spouse` of *character* is `Null`.
- The *spouse* is alive.
- The *spouse* is `visible`.
- The `spouse` of *spouse* is `Null`.
- The *character* and *spouse* are at the same `location`.
- The *character*'s and *spouse*'s `location` is a `chapel`.

Effects:

- The `spouse` of *character* is now *spouse*.
- The `spouse` of *spouse* is now *character*.

`pickup (` *character* `)`

Preconditions:

- The *character* is alive.
- The *character*'s `location` is a `cave`, `crossroads`, `forest`, or `village`.
- The `item` at the *character*'s `location` is an item constant.
- The *character* has an empty hand.

Effects:

- The *character* is now holding one more of the `item` that was at their `location`.
- The `item` at the *character*'s `location` is now `Null`.

`raise (` *character*, *target*, *graveyard* `)`

Preconditions:

- Only available when `Undead` is enabled.
- The *character* is alive.
- The *character* is holding a `RaisePotion`.
- The *target* is `Dead`.
- The *character* and *target* are at the same `location` or the *character* is at the *graveyard*.

Effects:

- The *character* is now holding one fewer `RaisePotion`.
- The *target* is now a `Ghost`.
- The *target*'s `location` is now the *graveyard*.
- The *target*'s `left` and `right` hands are now `Null`.

`recover (` *character* `)`

Preconditions:

- Only available when `Sickness` or `Violence` is enabled.
- The *character* is `Hurt`.

Effects:

- The *character* is now `Healthy`.

`release (` *character*, *jail* `)`

Preconditions:

- Only available when `Crime` is enabled.
- The *jail* is a `jail`.
- The *character* is alive.
- The *jail* is `locked`.
- The *character*'s `location` has a path to the *jail*.

Effects:

- The *jail* is no longer `locked`.

- If there exists a character whose is a `criminal` and whose `location` is the *jail*, the *character* is now a `criminal`.
- For every *other* character whose `location` is the *jail*: If the `relationship` between *other* and *character* was `Enemy`, it is now `Null`. If the `relationship` was `Null`, it is now `Friend`.

`repent (` *character* `)`

Preconditions:

- Only available when `Forgiveness` is enabled.
- The *character* is alive.
- The *character* is a `criminal`.
- The *character*'s `location` is a `chapel`.

Effects:

- The *character* is no longer a `criminal`.

`rise (` *character*, *graveyard* `)`

Preconditions:

- Only available when `Undead` is enabled.
- The *character* is `Dead`.

Effects:

- The *character* is now a `Ghost`.
- The *character*'s `location` is now the *graveyard*.
- The *character*'s `left` and `right` hands are now `Null`.

`sell (` *character*, *item* `)`

Preconditions:

- Only available when `Commerce` is enabled.
- The *character* is not a `merchant`.
- The *item* is an item constant.
- The *item* is not a `Coin`.
- The *character* is alive.
- The *character* is holding the *item*.
- The *character*'s `location` is a `market`.

Effects:

- The *character* is now holding one fewer of the *item*.
- The *character* is now holding one additional `Coin`.

`sicken (` *character* `)`

Preconditions:

- Only available when `Sickness` is enabled.
- The *character* is `Healthy`.

Effects:

- The *character* is now `Hurt`.

`take (` *character*, *item*, *target* `)`

Preconditions:

- Only available when `Theft` is enabled.
- The *character* and *target* character are different.
- The *item* is an item constant.
- The *character* is alive.
- The *character* is holding a `Sword`, or the *character* is a `bandit`, or the *target* is `Hurt`.
- The *character* has an empty hand.
- The *target* is `visible`.
- The *character* and *target* are at the same `location`.

Effects:

- The *character* is now holding one additional *item*.
- The *target* is now holding one fewer *item*.
- If the *target* is not a `criminal`, the *character* is now a `criminal`.
- If the `relationship` between the *target* and *character* was `Friend`, it is now `Null`.
- If the `relationship` between the *target* and *character* was `Null`, it is now `Enemy`.

`teleport (` *character*, *location* `)`

Preconditions:

- Only available when `Teleportation` is enabled.
- The *location* exists on the map.
- The *character* is alive.
- The *character* is holding a `TeleportPotion`.
- The *character*'s `location` is not `Null`.

Effects:

- The *character*'s `location` is now *location*.
- The *character* is now holding one fewer `TeleportPotion`.

`trade (` *character*, *price*, *target*, *item* `)`

Preconditions:

- The *character* and *target* character are different.
- The *price* and *item* are different item constants.
- The *character* is alive.
- The *character* is `visible`.
- The *character* is holding the *price*.
- The *target* is alive.
- The *target* is `visible`.

- The *target* is holding the *item*.
- The *character* and *target* are at the same `location`.

Effects:

- The *character* is now holding one fewer *price*.
- The *character* is now holding one additional *item*.
- The *target* is now holding one fewer *item*.
- The *target* is now holding one additional *price*.

`walk (` *character*, *location* `)`

Preconditions:

- The *character* is `Healthy` or a `Ghost`.
- There is a path from the *character*'s `location` to *location*.

Effects:

- The *character*'s `location` is now *location*.
- The *character* is now `visible`.

Version History

Version 1.0

- First public release of this document.

License

Acknowledgments

Thank you to Mira Fisher, Lasantha Senanayake, Gage Birchmeier, and other members of the University of Kentucky Narrative Intelligence Lab for their assistance in the design of MicroTales.

This work was supported in part by the U.S. National Science Foundation under Grant No. 2145153 and the U.S. Army Research Office under Grant No. W911NF-24-1-0195. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Army Research Office.

Grant #2145153

Grant #W911NF-24-1-0195

Bibliography

[1] Tim Anderson, Marl Blank, Bruce Daniels, and Dave Lebling. Zork I: the great underground empire. Infocom, 1980.

[2] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. TextWorld: a learning environment for text-based games. In *Computer Games*, pages 41–75. Springer, 2019.

[3] Malik Ghallab, Adele Howe, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL—the Planning Domain Definition Language, 1998.

[4] Ben Kybartas and Rafael Bidarra. A survey on story generation techniques for authoring computational narratives. *IEEE Transactions on Computational Intelligence and Artificial Intelligence in Games*, 9(3):239–253, 2016.

[5] Michael Lebowitz. Story-telling as planning and learning. *Poetics*, 14(6):483–502, 1985.

[6] Michael Mateas and Andrew Stern. Structuring content in the Façade interactive drama architecture. In *Proceedings of the 1st AAAI conference on Artificial Intelligence and Interactive Digital Entertainment*, 2005.

[7] Mark O. Riedl and Vadim Bulitko. Interactive narrative: an intelligent systems approach. *AI Magazine*, 34(1):67–77, 2013.

[8] John Slaney and Sylvie Thiébaux. Blocks World revisited. *Artificial Intelligence*, 125(1):119–153, 2001.

[9] Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 International Planning Competition. *AI Magazine*, 45(2):280–296, 2024. doi: https://doi.org/10.1002/aaai.12169. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/aaai.12169`.

[10] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: a standard interface for reinforcement learning environments, 2024. URL `https://arxiv.org/abs/2407.17032`.

[11] Nick Walton. AI Dungeon. `https://aidungeon.com/`, 2019.

[12] Stephen G. Ware and Rachelyn Farrell. A collection of benchmark problems for the Sabre narrative planner. Technical report, Narrative Intelligence Lab, University of Kentucky, November 2023.

[13] R. Michael Young, Stephen G. Ware, Bradly A. Cassell, and Justus Robertson. Plans and planning in narrative generation: a review of plan-based approaches to the generation of story, discourse and interactivity in narratives. *Sprache und Datenverarbeitung, Special Issue on Formal and Computational Models of Narrative*, 37(1-2):41–64, 2013.