

Sommario

[Introduction 2](#_heading=h.gjdgxs)

[Design 2](#_heading=h.30j0zll)

[FSA 3](#_heading=h.1fob9te)

[Sentence Similarity Task 4](#_heading=h.3znysh7)

[Simulation manual 4](#_heading=h.2et92p0)

[Technical manual 5](#_heading=h.tyjcwt)

[DisplaySwap 5](#_heading=h.3dy6vkm)

[RandomPrefab 5](#_heading=h.1t3h5sf)

[ConsoleManager 5](#_heading=h.4d34og8)

[OrdisV2 6](#_heading=h.2s8eyo1)

[Usage 6](#_heading=h.17dp8vu)

# Introduction

What follows is the technical documentation regarding the Ordis model.

The idea behind the model is based on a simple and recurring problem: how should a robot operate in a situation – perhaps during an emergency – in order to be as efficient as possible? The solution provided is one of the possible approaches whenever a similar situation occurs.

In particular, the base situation was imagined as follows:

* Ordis (our agent) is in a known environment
* Ordis is already in the environment (the position is not relevant)
* The number ranges from 0 to N (where N is the number of our predefined key points in the environment)
* Ordis can receive communications from an external source and associate them to a known goal

# Design

The main technology used in the design step is the game engine Unity, which was chosen for the vast amount of tools provided for simulating situations like the one we are trying to abstract and the AI APIs.

In particular, the NavMesh class of the AI module was fundamental for the motion plan process.

While Unity is the main block of the project for what regards our robot Ordis, the open-source platform HuggingFace played a central role in a simple yet challenging task, the sentence similarity one.

Follows a brief summary of the main technologies used:

* Unity 2022.3
* HuggingFace Unity API (Sentence Similarity) - https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
* Low Poly Assets from Unity Assets Store

# FSA

Immagine che contiene diagramma, cerchio, linea, testo

Descrizione generata automaticamente

We can abstract our model as a FSA with four different states and nine transitions:

* **Transitions**
  + **Sentence input**: a non empty string has been sent to HuggingFace and will be processed by the task (given that we do not receive an error after the request, timeouts happened quite frequently during the tests);
  + **Task error**: something went wrong during the SS Task (in most cases a timeout error from HuggingFace’s API);
  + **Empty input:** the input is an empty string and is not going to be elaborated;
  + **Task completed:** the result of the SS task has been received, Ordis can now seek all the goals in the environment and plan the path;
  + **Goals fringe is computed:** the goals fringe is ordered by asc. distance from the agent;
  + **Fringe is not empty yet:** a goal has been visited and has been removed from the goals fringe;
  + **Fringe empty:** the full path has been traversed;
  + **Input Override:** a sentence is sent while Ordis is wandering, the new search will begin after the agent reaches the current destination.
* **States:**
  + **Stop:** Ordis is not moving and ready to receive the goal tag;
  + **Sentence Similarity:** the SS Task is being computed;
  + **Plan:** all the goals are listed and ordered by distance from the agent;
  + **Visit Goal:** Ordis is moving toward the closest goal in order to visit it.

# Sentence Similarity Task

The core of the interaction between the user and Ordis lies in the Sentence Similarity Task.

The task accepts in input a list of n candidates and a sentence . The output of the Task will then be a list of n values , each corresponding to the confidence that the sentence matches with the corresponding candidate.

E.g.

# Simulation manual

Once the simulation starts, the user can perform two distinct actions:

* Write an input sentence
  + In order to open the console, press ‘T’ and send the input using ‘Enter’. In case the sentence is elaborated without errors, the console will be cleared;
* Switch camera from Ordis’ POV to a Third-person view (press ‘Tab’).

All the possible goal locations contain a Unity GameObject called ‘SpawnPoint’: on start, on every location a random Unity Prefab is instantiated.

There are three possible prefabs:

* bottle
* book
* rock

When a random prefab is instantiated, the prefab will be marked with a tag previously created (bottle, book or rock): this will help Ordis find them using a Unity function, simulating an already known environment.

Once the SS Task ends, Ordis will seek all the GameObjects with the same tag as the candidate with the higher confidence, calculate the shortest path from each goal if any is present in the map, and start visiting them from nearest to furthest.

The distance from Ordis to a position is calculated using the corners of the path and summing the Vector3 objects which are, in fact, the list of corners.

No spawn is guaranteed, which means that there could only be 2 prefabs present in the map.

Immagine che contiene Rettangolo, schermata, quadrato, design

Descrizione generata automaticamente

View from above of the simulation’s map, sixteen spawn points are present, Ordis is represented as a red capsule in the middle.

# Technical manual

The C# scripts used for the simulation have been collected in the Assets/Scripts directory.

## DisplaySwap

Class for the swap between cameras.

Cam1 is the first person Ordis POV, Cam2 is the third person one.

On the Tab key press, one cam is enabled and the other is disabled.

## RandomPrefab

Class for managing the spawn of random prefabs.

All the possible prefabs are declared as public variables and lie in the LowPolyDungeonsLite prefabs directory.

On simulation start, a random prefab is instantiated and its position set the same as the spawnpoint GameObject. The instantiated prefab is finally assigned the tag that matches his prefab.

In the particular cases of book or bottle, the scale is augmented by one for visibility purposes.

## ConsoleManager

Class for managing the sentences input and the communication with Ordis.

The input field, which is part of the TextMeshPro module, is activated by the key ‘t’.

Unity’s HuggingFace API are used for sending a request and perform the SS Task:

* On error, it will be visible in the debug console;
* On success, Ordis’ Search function will be invoked, and the highest confidence candidate will be passed as goal tag.

## OrdisV2

Second actual iteration of our Ordis model.

Whenever the agent receives an input from the console, the will be invoked, and all the GameObjects with the goal tag will be listed.

A will then be calculated for each one of them, using the function.

The class has an interesting attributes, which is : the corners define the places along a path where it changes direction (ie, the path consists of a number of straight-line moves between corners). Being basically a list of Vector3 positions, the length of a path is calculated by summing the distance between each corner using .

The list of goals positions is then sorted and Ordis can start his navigation by looping this list and visiting the first element: when this is visited, it is removed from the list and the destination is set on the new first element of the list.

Note that the path to a point is determined by the function, which finds the actual shortest path to the point (given it exists and is viable), so it is basically using a greedy approach on every object that meets the goal requirement (the tag).

A similar approach would’ve been one where the agents plans the path on every goal to the nearest non visited object, but the actual implementation was chosen for the possibility to compute the path one time only at the start.

# Usage

Despite being a simple simulation, this model could be very flexible and adapt to many situations, such as:

* Rescuing harmed civilians in a building;
* Retrieving object for a person with impaired mobility;
* Monitoring of patients in a hospital (paired with computer vision, perhaps).