

**AGENT'S SPATIAL BEHAVIOR IN PEDESTRIAN  
MOVEMENT: AN EMPIRICAL STUDY USING  
GEOMASON IN UNIVERSITY OF THE  
PHILIPPINES BAGUIO**

BY

ROSSEAU NILO G. MAAMOR

A SPECIAL PROBLEM SUBMITTED TO THE  
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE  
COLLEGE OF SCIENCE  
THE UNIVERSITY OF THE PHILIPPINES  
BAGUIO, BAGUIO CITY

AS PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF  
BACHELOR OF SCIENCE IN COMPUTER SCIENCE

JUNE 2024

This is to certify that this Special Problem entitled "**Agent's Spatial Behavior in Pedestrian Movement: An Empirical Study using GeoMASON in University of the Philippines Baguio**", prepared and submitted by **Rosseau Nilo G. Maamor** to fulfill part of the requirements for the degree of **Bachelor of Science in Computer Science**, was successfully defended and approved on June 7, 2024.

JOEL M. ADDAWE, PH.D.  
Special Problem Adviser

The Department of Mathematics and Computer Science endorses the acceptance of this Special Problem as partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science .

GILBERT R. PERALTA, DR.RER.NAT.  
Chair  
Department of Mathematics and  
Computer Science

# Table of Contents

Acknowledgments . . . . .	vii
Abstract . . . . .	viii
List of Tables . . . . .	xi
List of Figures . . . . .	xiii
Chapter 1. Introduction . . . . .	1
1.1 Background of the Study . . . . .	1
1.2 Statement of the Problem . . . . .	2
1.3 Objective of the Study . . . . .	3
1.3.1 General Objective of the Study . . . . .	3
1.3.2 Specific Objective of the Study . . . . .	3
1.4 Significance of the Study . . . . .	3
1.5 Scope and Limitation . . . . .	4
Chapter 2. Review of Related Literature . . . . .	5
Chapter 3. Preliminaries . . . . .	8
3.1 Urban Environment and Cognitive Map . . . . .	10
3.1.1 A. Urban Environment (UE) . . . . .	10
3.1.2 B. Graph Theory (GT) . . . . .	11
3.1.3 Centrality in Graph Theory . . . . .	14
3.1.4 C. Cognitive Map (CM) . . . . .	15
3.1.5 D. Urban Elements and Path Attributes (UE & PA) . . . . .	17
3.1.6 E. Dual Graph (DG) . . . . .	18
3.1.7 F. Behavior Mechanisms (BM) . . . . .	19
3.2 Questionnaire and Data . . . . .	22
3.2.1 G. Empirical Research (ER) . . . . .	22
3.3 Cluster Analysis and Agent Typologies . . . . .	24
3.3.1 H. Z-Score (ZS) . . . . .	24
3.3.2 I. K-Means Clustering and Algorithm (KMCA) . . . . .	25
3.3.3 J. Silhouette Score (SS) . . . . .	26
3.3.4 K. Variance Ratio Criterion (VRC) . . . . .	26
3.3.5 L. Agent Typologies (AT) . . . . .	27
3.4 ABM and Pedestrian Flow . . . . .	28
3.4.1 M. Individual-Pedestrian-Agent Terminologies (IPAT) . . . . .	28

3.4.2	N. Agent-Based Modeling (ABM) . . . . .	31
3.4.3	O. Monte-Carlo Simulation (MCS) . . . . .	34
3.5	Configurations Comparison . . . . .	35
3.5.1	P. Configurations . . . . .	35
3.5.2	Q. Wilcoxon Signed-Rank Test (WSRT) . . . . .	36
3.5.3	R. Comparison . . . . .	36
3.6	Small Scale Replication (SSR) . . . . .	37
3.6.1	Step 1. Cognitive Map from Urban Environment . . . . .	37
3.6.2	Step 2. Summarized Empirical Data . . . . .	37
3.6.3	Step 3. Cluster Analysis and Agent Typologies . . . . .	43
3.6.4	Step 4. Agent-Based Model (ABM) Configurations and Simulations	45
3.6.5	Step 5. Configurations Comparison . . . . .	50
3.6.6	SSR Conclusion . . . . .	56
	 Chapter 4. Methodology . . . . .	58
4.1	Overview . . . . .	58
4.2	Step 1: Urban Environment and Cognitive Map . . . . .	61
4.3	Step 2: Questionnaire and Data . . . . .	70
4.3.1	Section I: Participant Demographic Information . . . . .	72
4.3.2	Section II: Study Route 0 Navigational Task 1 (UPB Main Entrance to Kolehiyo ng Agham (KA) Building) . . . . .	72
4.3.3	Section III: Study Route 1 Navigational Task 2 (UPB Main Entrance to College of Social Science (CSS) Building) . . . . .	74
4.3.4	Section IV: Study Route 2 Navigational Task 3 (UPB Main Entrance to College of Arts and Communication (CAC) Building) .	75
4.3.5	Section V: Preferences for Certain Route Characteristics . . . . .	77
4.3.6	Summarized Empirical Data . . . . .	79
4.4	Step 3: Cluster Analysis and Agent Typologies . . . . .	82
4.4.1	Silhouette score . . . . .	89
4.4.2	Variance Ratio Criterion . . . . .	90
4.4.3	Empirical Data Cluster Analysis and Agent Typologies . . . . .	91
4.5	Step 4. ABM and Pedestrian Flow . . . . .	95
4.5.1	ABM Calibration . . . . .	99
4.5.2	Empirical Data ABM Simulation per Configuration . . . . .	102
4.6	Step 5. Configurations Comparison . . . . .	106
	 Chapter 5. Results and Discussion . . . . .	108
5.1	Configuration Comparison . . . . .	109
5.1.1	Volume of Configuration 3 Clusters . . . . .	109
5.1.2	Configuration 1, 2, and 3 Comparison Where Configuration 1 Acts as the Baseline . . . . .	111

Chapter 6. Conclusion and Recommendation . . . . .	115
6.1 Conclusion . . . . .	115
6.2 Recommendations . . . . .	117
List of References . . . . .	119
Appendix A. Questionnaire for Pedestrian Movement Analysis . . . . .	130
Appendix B. Questionnaire Data for Pedestrian Movement Analysis . . . . .	136
Appendix C. Configuration 3 Clusters, Configuration 1, and Configuration 2 Statistics . . . . .	149
Appendix D. Agent Route Choice Statistics . . . . .	151
Appendix E. Agent Volume Per Street Segment Tables . . . . .	152
Appendix F. Source Code . . . . .	173

# Acknowledgments

I would like to extend my deepest gratitude to all those who have made this research possible and provided their invaluable support.

First and foremost, I express my sincerest appreciation and gratitude to Professor Joel M. Addawe, my thesis adviser, for his unwavering guidance, insightful feedback, and constant encouragement throughout the course of this research. I thank his dedication to cook me during consultation defenses, which really helped me to cook this research. He deserves all my highest praises, gratitude, and respect, for all the guidance he gave me. I also applaud his patience and dedication in helping me navigate through the complexities of this study. I am truly grateful for his mentorship, expertise, and most especially, his time, which I consumed in abundance (seven face-to-face consultations so far). He will remain as the most memorable and most impactful mentor and teacher throughout my academic life.

Special thanks are due to Dr. Gabriele Filomena, whose expertise was instrumental in shaping the research design and methodology. Their responses throughout the two semesters to fix the bugs and errors found in the model has been a cornerstone of this project. Without their expertise, I might have been stuck debugging and troubleshooting the model.

I thank Dr. Dymphna N. Javier, my Professor in Geology 11, for helping me understand the usage of ArcGIS. Their guidance and expertise in the field of Geology has been a great help in understanding the spatial data used in this research. I also thank Sir Lee Javellana, Ma'am Ashlyn Balangcod, and Ma'am Criselda Libatique, for their assistance regarding Python, Java, and Wilcoxon Test, respectively. I also thank ate Aliah Zabala, for her tips on how to do CMSC190 Part 2 with time and efficiency in mind.

I am grateful to my peers and colleagues especially to Khan, for their camaraderie and intellectual exchange that enriched my research experience. I am grateful to the Python, Java, and ArcGIS communities in Discord, for their prompt responses to my queries and for providing valuable insights that helped me overcome technical challenges on my codes. I am also grateful to ading Lulina for their motivation and emotional support.

I would like to express my heartfelt gratitude to the University of the Philippines Alumni Association in America, for providing me financial support throughout college.

My heartfelt gratitude goes to the University of the Philippines Baguio, for providing the resources and facilities that enabled me to conduct this research. I am thankful for the opportunity to pursue my academic interests and contribute to the body of knowledge in the field of Computer Science.

Lastly, I would like to thank my siblings and friends. Special mention to my mother, Solima Maamor, for her patience, understanding, and support, which bolstered my spirits and sustained me through this journey. I am eternally grateful for giving me the opportunity to pursue my passion, and the freedom to do what I need to do. I dedicate this research to them, my family, as a testament to their unwavering love and support.

This research would not have been possible without the collective effort and support of all those mentioned above, as well as many others who have contributed in various ways. I am deeply thankful to each one of them.

# Abstract

## Agent's Spatial Behavior in Pedestrian Movement: An Empirical Study using GeoMASON in University of the Philippines Baguio

Rosseau Nilo G. Maamor  
University of the Philippines, 2024

Adviser:  
Joel M. Addawe, Ph.D.

In this study, we investigate the pedestrian route choice behavior of an agent using University of the Philippines Baguio (UPB) campus. We used 68 survey responses and the integration of agent-based modeling (ABM) for the simulation. In our experiment, we consider the demographic distribution, walking preferences, and the influence of variables Distance, Landmarks, and Barriers respectively on pedestrian route choices. For the questions, the diversity in route choice strategies and the impact of cluster-based empirical path selection (heterogeneity) versus average-based empirical path selection (homogeneity) are considered for movement patterns. The ABM simulations tested three configurations: Random Non-Empirical Path Selection, Average-Based Empirical Path Selection, and Cluster-Based Empirical Path Selection. Both empirical configurations produced more realistic (average of 5.83% difference per path segment) pedestrian movement patterns than the random configuration, with the cluster-based approach showing an all-positive difference (average of 5.69% difference per path segment). The results demonstrate the value of incorporating empirical data and considering individual preferences in ABM frameworks. In conclusion, using UPB campus, the findings underscore the importance of heterogeneity and empirical data in ABM to accurately represent real-world phenomena and inform decision-making processes.

# List of Tables

3.1	Example Questionnaire Responses . . . . .	38
3.2	Example Path Probabilities . . . . .	39
3.3	Path Probabilities for Different Cases . . . . .	41
3.4	SSR Socio-Demographic Profile of the Participants (N = 5). . . . .	41
3.5	Number of clusters and Silhouette Score using K-means Algorithm, in comparison to VRC values. . . . .	44
3.6	Example Configuration 1 Results . . . . .	46
3.7	Example Mean of Individual Route Choice Strategies . . . . .	47
3.8	Example Configuration 2 Results . . . . .	47
3.9	Example Configuration 3 Results . . . . .	48
3.10	Example Configuration 1 and 2 Wilcoxon Signed-Rank Test . . . . .	50
3.11	Critical Values of W in Wilcoxon Signed-Rank Test . . . . .	52
3.12	Example Wilcoxon Signed-Rank Test for Configuration 1 and 3 . . . . .	53
3.13	Example Wilcoxon Signed-Rank Test . . . . .	54
4.1	The Socio-Demographic Profile of the Participants (N = 68). . . . .	79
4.2	The behavioral components of the ABM and the corresponding parameters that regulate the definition of the agent's route choice behavior. They fall within PP: Prospective Planning and SP: Situated Planning. The ABM parameters represent probabilities, and they can assume values between 0.0 and 1.0. In the empirically-based configurations of the ABM, the sample attribute values are used to obtain the ABM parameter values. Sample attributes may be cluster attributes when agent typologies are built.	86
4.3	Number of clusters and Silhouette Score using K-means Algorithm, in comparison to VRC values. . . . .	91
4.4	Average Utilization of Variables for Clusters 1,2, and 3 Tabulated Data. Agent Typologies of the Clusters. . . . .	93
4.5	Agent Typologies Demographics per Cluster in Probabilities. . . . .	94

4.6 Configuration 1 Volume per Path Segment Tabulated Data . . . . .	102
4.7 Configuration 2 Volume per Path Segment Tabulated Data . . . . .	104
4.8 Configuration 3 Volume per Path Segment Tabulated Data . . . . .	105
5.1 Configuration 3 Clusters Volume per Path Segment Tabulated Probability Data . . . . .	110
5.2 Statistically Significant Volume Difference of Configuration 2 and Configuration 3 from Configuration 1. . . . .	112
C.1 Configuration 3 Clusters, Configuration 1, and Configuration 2 Statistics	150
D.1 Variable Statistics per Agents . . . . .	151
E.1 Run 0: Agent Volumes Per Street Segment . . . . .	153
E.2 Run 1: Agent Volume Per Street Segment . . . . .	154
E.3 Run 2: Agent Volume Per Street Segment . . . . .	155
E.4 Run 3: Agent Volume Per Street Segment . . . . .	156
E.5 Run 4: Agent Volume Per Street Segment . . . . .	157
E.6 Run 5: Agent Volume Per Street Segment . . . . .	158
E.7 Run 6: Agent Volume Per Street Segment . . . . .	159
E.8 Run 7: Agent Volume Per Street Segment . . . . .	160
E.9 Run 8: Agent Volume Per Street Segment . . . . .	161
E.10 Run 9: Agent Volume Per Street Segment . . . . .	162
E.11 Run 10: Agent Volume Per Street Segment . . . . .	163
E.12 Run 11: Agent Volume Per Street Segment . . . . .	164
E.13 Run 12: Agent Volume Per Street Segment . . . . .	165
E.14 Run 13: Agent Volume Per Street Segment . . . . .	166
E.15 Run 14: Agent Volume Per Street Segment . . . . .	167
E.16 Run 15: Agent Volume Per Street Segment . . . . .	168
E.17 Run 16: Agent Volume Per Street Segment . . . . .	169
E.18 Run 17: Agent Volume Per Street Segment . . . . .	170
E.19 Run 18: Agent Volume Per Street Segment . . . . .	171
E.20 Run 19: Agent Volume Per Street Segment . . . . .	172

# List of Figures

3.1	Preliminaries Overview Illustration . . . . .	9
3.2	University ( $U$ ) as an Urban Environment. . . . .	11
3.3	University ( $U$ ) illustrated using Graph Theory. . . . .	13
3.4	Cognitive Map Example . . . . .	16
3.5	Illustration of the Urban Elements . . . . .	17
3.6	Illustration of the Path Attribute . . . . .	18
3.7	Dual Graph Example . . . . .	19
3.8	Illustration of the Agent's Behavior Mechanism . . . . .	20
3.9	Individual-Pedestrian-Agent Illustration . . . . .	29
3.10	Mathematical Individual-Pedestrian-Agent Illustration . . . . .	30
3.11	Individual-Pedestrian-Agent Relation . . . . .	31
3.12	SSR Cognitive Map . . . . .	37
3.13	Route Choice Importance in Pedestrian Route Choice . . . . .	42
3.14	Example Configuration 1 Volume per Path Segment. . . . .	46
3.15	Example Configuration 2 Volume per Path Segment. . . . .	48
3.16	Example Configuration 3 Volume per Path Segment. . . . .	49
3.17	SSR Cluster 1 . . . . .	49
3.18	SSR Cluster 2 . . . . .	49
3.19	Example Volume Difference of Configuration 2 and Configuration 3 . . . . .	55
4.1	Methodology Overview Illustration . . . . .	58
4.2	Methodology Step 1 Overview Illustration . . . . .	61
4.3	University of the Philippines Baguio ( $U$ ) as Urban Environment shown in Enhanced Contrast Map. . . . .	62
4.4	<b>UEPD PPD</b> (edges) shown in blue, layered on top of the University ( $U$ ) Map. . . . .	63
4.5	<b>UEPD PID</b> (nodes) shown in red, layered on top of the University ( $U$ ) Map. . . . .	63

4.6	Combined Retrieved Map <b>PPD</b> shown in blue, <b>PID</b> shown in red, layered on top of the base map. . . . .	64
4.7	Retrieved PPD (edges) shown in blue. . . . .	64
4.8	Retrieved PID (nodes) shown in red. . . . .	64
4.9	Combined Retrieved UEPD where PPD is shown in blue, PID is shown in red. . . . .	65
4.10	Identified edges shown in blue. . . . .	66
4.11	Identified nodes shown in red. . . . .	66
4.12	Landmarks shown in orange in the UPB path network. . . . .	67
4.13	Barriers shown green in the UPB path network. . . . .	68
4.14	Cognitive Map of the University of the Philippines Baguio (UPB). . . . .	69
4.15	Methodology Step 2 Overview Illustration . . . . .	70
4.16	Variable Importance in Pedestrian Route Choice. Box plot of the variables extracted from the responses to the questionnaire for the entire study sample. The boxes colored in red represent the distribution of the variables describing the probability of manifesting a certain behavior. Variables colored in red were included in the cluster analysis. . . . .	80
4.17	Methodology Step 3 Overview Illustration . . . . .	82
4.18	Average Utilization of Variables for Clusters 1,2, and 3. Agent Typologies of the Clusters. . . . .	93
4.19	Methodology Step 4 Overview Illustration . . . . .	95
4.20	Configuration 1 Volume per Path Segment . . . . .	102
4.21	Configuration 2 Volume per Path Segment . . . . .	103
4.22	Configuration 3 Volume per Path Segment . . . . .	105
4.23	Methodology Step 5 Overview Illustration . . . . .	106
5.1	Results and Discussion Step Overview Illustration . . . . .	108
5.2	Comparison of the Volumes per Path Segment in Clusters 1, 2, and 3, shown in probabilities. . . . .	109



# Chapter 1

## Introduction

Understanding pedestrian movement is crucial for designing safe and efficient urban spaces for pedestrian use, particularly in densely populated areas such as university campuses. This study, entitled “Agent’s Spatial Behavior in Pedestrian Movement: An Empirical Study Using GeoMASON in University of the Philippines Baguio,” aims to explore how individuals navigate through the three main buildings within the built environment of the University of the Philippines Baguio (UPB) campus. These three main buildings are: Kolehiyo ng Agham (KA) Building, College of Social Science (CSS) Building, and College of Arts and Communication (CAC) Building. By employing GeoMASON, an agent-based modeling framework, this research will provide insights into the spatial behavior of pedestrians, which can inform better campus planning and management strategies.

The University of the Philippines Baguio (UPB) campus provides a simple case study for investigating pedestrian behavior and agent-based modeling. The campus is characterized by a pedestrian activity such as commuting to/from university. The campus environment features a variety of pathways, buildings, and open spaces, which contribute to a simple spatial configuration that influences pedestrian movement. By studying pedestrian behavior within the UPB campus, we can gain insights into the factors shaping route choices, walking preferences, and interactions of urban elements and path attributes among the pedestrians in UPB.

### 1.1 Background of the Study

Pedestrian movement research has gained significant attention due to its implications for urban planning, public safety, and transportation efficiency [25]. Traditional methods of studying pedestrian behavior, such as direct observation and surveys, have limitations in

capturing the dynamic and complex nature of human movement. Agent-based modeling (ABM) [4] has emerged as a powerful tool to simulate and analyze the interactions of individuals within a given space. GeoMASON [95], an extension of the MASON simulation library [67] for geospatial data, offers advanced capabilities for modeling spatial environments and the behaviors of agents within them.

The University of the Philippines Baguio, with its unique topography and dense population, presents an ideal case study for examining pedestrian dynamics. By integrating real-world data into GeoMASON, this study aims to create a detailed and accurate model of pedestrian movement on campus. The findings will contribute to the broader field of urban studies and provide actionable insights for improving campus infrastructure and safety.

This study aims to develop an empirically-based ABM for simulating pedestrian movement in the University of the Philippines Baguio (UPB) campus. By integrating survey data with computational modeling techniques, the study seeks to understand the complexities of pedestrian behavior and its implications for agent-based modeling.

The study provides a foundation for future research on pedestrian behavior and agent-based modeling in the context of UPB. By considering heterogeneity and empirical data in ABM frameworks, researchers can accurately represent real-world phenomena and inform decision-making processes in the context of the UPB case study area.

## 1.2 Statement of the Problem

Despite the importance of understanding pedestrian movement for urban planning and safety, there is a lack of comprehensive studies focused on university campuses, particularly in the context of the University of the Philippines Baguio. Existing research on pedestrian behavior often relies on simplified models that do not capture the complexities of human movement in real-world environments (i.e. heterogeneity of human behavior). This study aims to address this gap by developing an empirically-based ABM for simulating pedestrian movement in the UPB campus.

## 1.3 Objective of the Study

### 1.3.1 General Objective of the Study

The general objective of this study is to present an empirically-based ABM for the simulation of pedestrian movement in the University of the Philippines Baguio (UPB) that accounts for behavioral heterogeneity in pedestrian route choice strategies.

### 1.3.2 Specific Objective of the Study

The specific objectives of this study is to answer the following research questions:

1. What is the diversity in route choice strategies as concerns the usage of Distance, Landmarks, and Barriers?
2. To what extent does the variation in the agents' cluster-based empirical path selection — i.e. a model including agent typologies vs a model with average-based empirical path selection — generate different movement patterns across the path network?

## 1.4 Significance of the Study

We consider this research significant for several reasons. First, this research will provide a detailed analysis of pedestrian behavior in a university setting, which contributes to the academic literature on urban studies, agent-based modeling, and studies where University of the Philippines Baguio is the case study area. Second, the insights gained from this study can inform the design and management of campus infrastructure, which can enhance the safety and accessibility for all path networks inside the UPB. Third, the methodology developed can be applied to other campuses and similar urban environments, which offers a valuable tool for planners and researchers to study pedestrian behavior and spatial dynamics in various contexts.

## 1.5 Scope and Limitation

The scope of this study is limited to the University of the Philippines Baguio campus. The research will focus on analyzing pedestrian movement within this specific spatial environment using the GeoMASON framework. Data collection will include pedestrian counts, movement patterns, and spatial characteristics of the campus. The study will not cover pedestrian behavior outside the campus or in other university settings.

Delimitations of the study include:

1. The study focuses on three main buildings within the UPB campus: Kolehiyo ng Agham (KA) Building, College of Social Science (CSS) Building, and College of Arts and Communication (CAC) Building. The buildings KA, CSS, and CAC serve as the destination points for pedestrian movement analysis. Meanwhile, the UPB Main Entrance serves as the origin point for pedestrian movement analysis.
2. The study only considers the following pedestrian activities: commuting to/from university, and attending the university for study or work.
3. The study only considers the three main route choice strategies: Distance, Landmarks, and Barriers.
4. The study only considers the Barriers in UPB as severing or natural barriers.
5. The study uses GeoMASON to model pedestrian movement and analyze route choice strategies based on Distance, Landmarks, and Barriers.
6. The study integrates survey data with computational modeling techniques to develop an empirically-based ABM for simulating pedestrian behavior.

# Chapter 2

## Review of Related Literature

Walking is a fundamental mode of transportation [78] and a key component of urban life [1]. It is a sustainable, healthy, and equitable form of mobility that contributes to the vibrancy and livability of cities. Walking is not only a means of getting from one place to another but also a way of experiencing and interacting with the urban environment [54]. Walking in urban spaces is a complex and multifaceted activity, extensively explored in the domains of transport geography [54, 81], urban geography [25, 71], and psychological research [1, 18, 42]. It is more than a mere mode of travel; walking is recognized as a political act [70], a form of urban emancipation [19], and a significant social practice [72]. Moreover, it is a predictor of physical and mental well-being [27, 82, 85], and a key element in discussions on sustainable, low-carbon mobility that contributes to creating liveable cities [96, 93]. Understanding pedestrian distribution in urban spaces is essential for observing pedestrian interactions with the urban environment [101], advancing theories of pedestrian behavior [104], and informing urban design [98].

Agent-Based Modelling (ABM) has emerged as a powerful tool to study and simulate pedestrian movement in urban environments. An ABM is composed of autonomous entities, or agents, that follow defined behavioral rules to make decisions and interact with their surroundings [6]. These models allow researchers to explore how individual behaviors aggregate into macroscopic Patterns [22]. ABMs have proven invaluable in understanding geographical and social phenomena [17, 53] and have been used to examine human interactions within social and spatial contexts [56, 57]. In the context of pedestrian movement, ABMs have been used to study route choice behavior [81, 32], pedestrian crossing behavior [81], and realistic locomotion [97]. These models have provided insights into the complex and dynamic nature of pedestrian movement in urban areas [91].

The intricate and dynamic nature of way finding in urban areas [91] makes ABM

a fitting method for analyzing pedestrian behavior. Previous studies using simulation models have focused on various aspects of pedestrian movement, such as route choice behavior [81, 32], pedestrian crossing behavior [81], and realistic locomotion [97]. However, traditional approaches in transport geography and Geo Informatics Science often treat walking as a homogeneous means of transportation[71], based on rational and utilitarian assumptions about human behavior [66]. This perspective limits the development of realistic simulation models, as it overlooks the complexities of urban form and its impact on pedestrian behavior [51]. Existing ABMs typically assume homogeneous agents optimizing specific road attributes, such as distance or angular change [60, 77], thereby simplifying the nuanced relationship between urban form and pedestrian behavior.

Contrary to this, recent research has emphasized the significance of urban elements like landmarks, regions, and barriers in pedestrian movement simulation to enhance the agents' cognitive representations of the environment [33, 32]. These models, while more authentic, still rely on homogeneous groups of agents, which is inconsistent with evidence of diverse route choice strategies among pedestrians [43, 45, 61, 89].

The concept of empirically based ABMs, introduced by Janssen and Ostrom in 2006 [59], advocates for models grounded in empirical data to validate patterns observed in simulations. This approach calls for incorporating behavioral heterogeneity to reflect realistic interactions and outcomes [86]. Empirical data, such as questionnaires and observations, should inform the behavioral rules in ABMs, a method widely adopted in research on residential choice [46, 9, 28]. Similarly, crowd motion studies have utilized data-driven approaches to model phenomena like crowd egress and congestion [21, 102, 47, 48]. These studies have shown that empirical data can enhance the accuracy and validity of simulation models by capturing the complexities of human behavior in urban environments.

Despite these advancements, existing pedestrian movement models often overlook behavioral heterogeneity. This oversight may stem from the challenges of formalizing diverse behaviors into algorithms [57, 59]. Research on route choice strategies typically focuses on vehicular traffic or general way finding behavior [43, 58, 105], or investigates deviations from the shortest path [35, 45, 64, 103], making it difficult to generalize these findings into ABM parameters due to specific environmental factors involved.

The most recent known pedestrian movement model that incorporates behavioral heterogeneity is the GeoMASON model advanced by Filomena [31]. This model uses a Geographic Information System (GIS) to represent the urban environment and integrates the MASON library to simulate pedestrian movement. The model incorporates urban elements and path attributes to enhance the agents' cognitive maps, allowing for more realistic route choice behavior. However, the model focuses on the pedestrian movement on a city scale, so the findings may not be generalizable to other urban environments, especially in the university scale where the environment is more controlled and less complex.

This study aims to address the gap in the literature by developing an empirically grounded ABM that captures the behavioral heterogeneity of pedestrians in a university environment. The model will incorporate urban elements and path attributes specific to the university environment to enhance the agents' cognitive maps, allowing for more realistic route choice behavior. The study will use empirical data collected through questionnaires to inform the agents' behavioral rules and validate the model's outcomes. By focusing on a university environment, the study aims to provide insights into pedestrian movement in a controlled and less complex urban setting, which contributes to the broader understanding of pedestrian behavior in university environments.

# Chapter 3

## Preliminaries

**Definition 3.1 Case Study** - Refers to the research study entitled “Agent’s Spatial Behavior in Pedestrian Movement: An Empirical Study using GeoMASON in University of the Philippines Baguio”

**Definition 3.2 Concept** - An idea or thought that represents a general understanding or category of things.

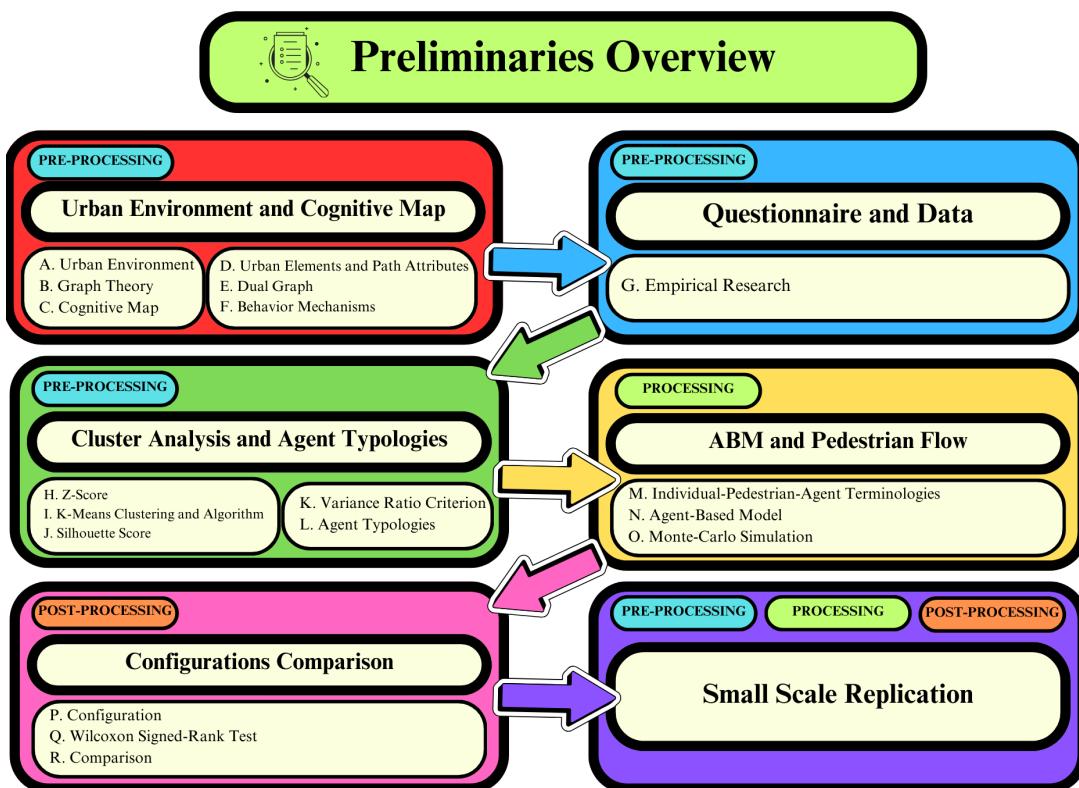
**Definition 3.3 Abstraction** - The process of generalization by reducing the information content of a **concept** [3.2], typically in order to retain only information which is relevant for the **case study** [3.1].

**Definition 3.4 Preliminary** - Describe the concepts done in preparation for a main part of the case study.

**Definition 3.5 Manuscript** - Refers to a written document written by a student, which presents the **abstraction** [3.1] of the preliminaries, methodology, analysis, and conclusions of the **case study** [3.1].

For us to understand the **case study**[3.1], **abstraction**[3.3] is needed. Our abstraction covers the eighteen key concepts, which are considered the **preliminaries** [3.4]: Urban Environment (UE) (3.1.1), Graph Theory (GT) (3.1.2), Cognitive Map (CM) (3.1.4), Urban Elements and Path Attributes (UE & PA) (3.1.5), Dual Graph (DG) (3.1.6), Behavior Mechanisms (BM) (3.1.7), Empirical Research (ER) (3.2.1), Z-Score (ZS) (3.3.1), K-Means Algorithm (KMA) (3.3.2), Silhouette Score (SS) 3.3.3, Variance Ratio Criterion (VRC) (3.3.4), Agent Typologies (AT) (3.3.5), Individual-Pedestrian-Agent Terminologies (IPAT) (3.4.1), Agent-Based Model (ABM) (3.4.2), Monte-Carlo Simulations (MCS)

(3.4.3), Configurations (3.5.1), Wilcoxon Signed-Rank Test (WSRT) (3.5.2), and Comparison (3.5.3). Other important **concepts** in the case study are discussed over the course of this abstraction.



**Figure 3.1:** Preliminaries Overview Illustration

## 3.1 Urban Environment and Cognitive Map

### 3.1.1 A. Urban Environment (UE)

**Definition 3.6** *Urban Environment* - refers to the physical and functional characteristics of a space or area used by the case study as setting.

**Definition 3.7** *Building* - refers to a structure with roof and walls.

**Definition 3.8** *Road* - refers to a wide way leading from one place to another, especially one with a prepared surface that vehicles can use.

**Definition 3.9** *Path* - refers to a way or track laid down for walking.

**Definition 3.10** *Intersection* - refers to a place where two or more roads or paths meet.

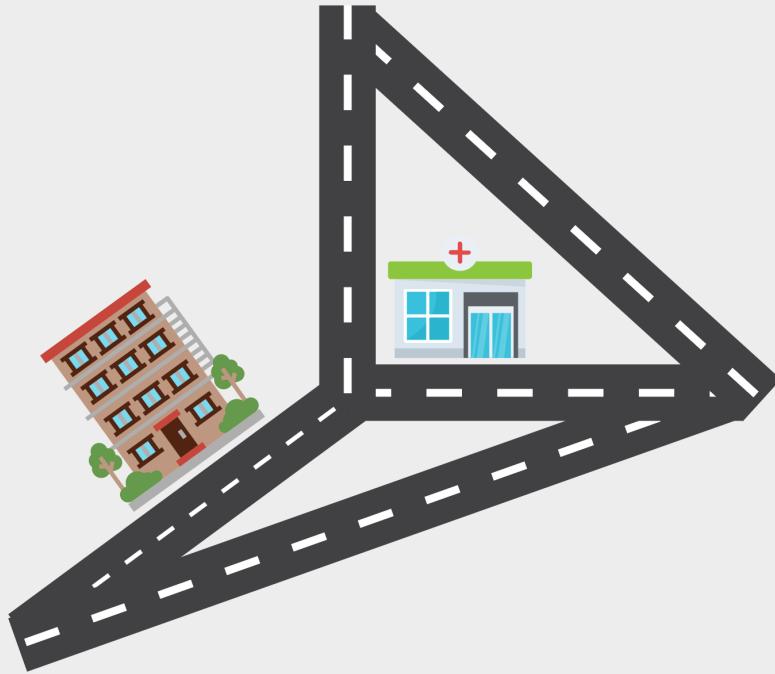
**Definition 3.11** *Streets* - refers to two or more roads or paths in an urban environment, typically with buildings on one or both sides.

**Definition 3.12** *Natural elements* - refers to the physical characteristics of the environment that are not created or modified by humans.

**Definition 3.13** *Artificial elements* - refers to the physical characteristics of the environment that are created or modified by humans.

An urban environment considered in this case study contains **buildings** [3.7], **streets** [3.11], **natural elements** [3.12], and **artificial elements** [3.13]. Streets in an urban environment is composed of **roads** [3.8], **paths** [3.9], and **intersections** [3.10].

Consider the example of an urban environment University (*U*).

Example 1: Urban Environment: University ( $U$ )

**Figure 3.2:** University ( $U$ ) as an Urban Environment.

In Figure 3.2, the University ( $U$ ) is an urban environment with buildings, streets, and natural elements. The streets are composed of roads and intersections. The buildings are artificial elements, while the trees are natural elements. The urban environment University ( $U$ ) is the setting of the case study.

### 3.1.2 B. Graph Theory (GT)

**Definition 3.14** *Graph* - refers to a collection of vertices and edges.

**Definition 3.15** *Graph Theory* - refers to the study of graphs, which are mathematical structures used to model pairwise relations between objects.

**Definition 3.16** *Nodes* - refers to a point where two or more curves, lines, or edges meet.

**Definition 3.17** *Edges* - refers to a line segment connecting two nodes in a graph.

**Definition 3.18** *Degree* - refers to the number of edges connected to a node in a graph.

**Definition 3.19** *Path* - refers to a sequence of edges that connect a sequence of distinct nodes.

**Definition 3.20** *Cycle* - refers to a path that starts and ends at the same node.

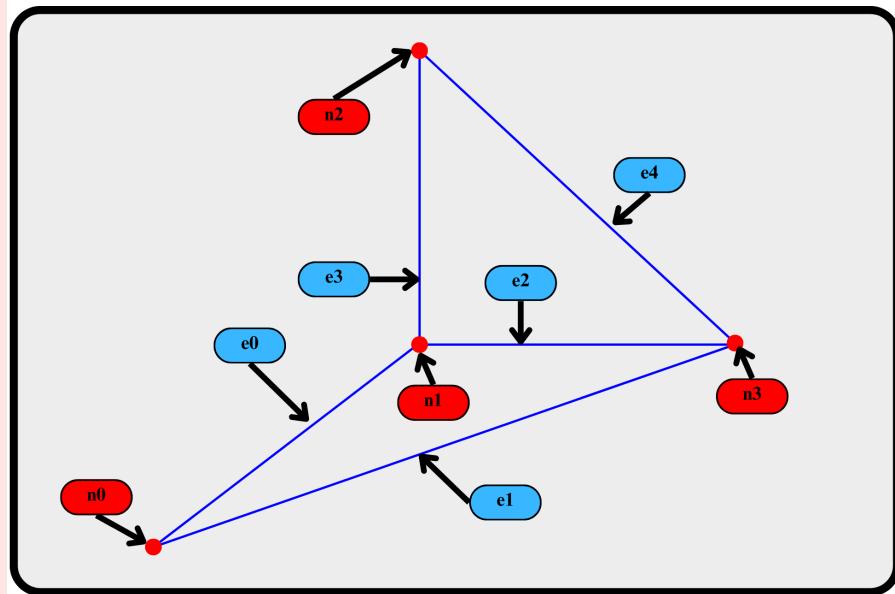
**Definition 3.21** *Connected Graph* - refers to a graph in which there is a path between every pair of nodes.

**Definition 3.22** *Directed Graph* - refers to a graph in which edges have a direction.

**Definition 3.23** *Weighted Graph* - refers to a graph in which edges have weights.

#### Example 2: Example of Graph Theory

From Example 1, we can represent the urban environment University ( $U$ ) as a graph. In this graph, the intersections are nodes ( $n$ ), and the roads are edges ( $e$ ). The nodes represent the intersections, and the edges represent the roads connecting them. The graph can be used to model the connectivity of the urban environment and analyze the relationships between different elements.



**Figure 3.3:** University ( $U$ ) illustrated using Graph Theory.

## Nodes and Edges in Graph Theory

In graph theory, we create a visual representation of relationships. Nodes and edges provide a way to depict complex relationships with just these two basic elements.

- **Nodes (Vertices):** Nodes are the fundamental units in a graph. They represent intersections. Each node has a unique identity and can hold additional information or attributes.
- **Edges:** Edges are the lines or arcs that connect nodes and represent relationships or interactions between them. They are the bridges that establish connections. Edges illustrate the roads or pathways in University ( $U$ ).

There are two main types of edges:

1. **Unweighted Edges:** These edges simply represent a connection between nodes without any additional information attached. For example, in a transportation system, an unweighted edge means the edge will have no additional information like attribute-increase caused by other elements near the edge. This means that

the probability of traversing the edge is based on its existence and not on any other factor.

2. **Weighted Edges:** Edges that carry additional information called weights. These weights can represent various things such as distance between cities in a map, or the cost of a movement between two destinations. In Example 2, a weighted edge can have an additional information like attribute-increase caused by other elements near the edge, which can affect the probability of traversing the edge.

Understanding nodes and edges helps in grasping the essence of graph theory. It allows us to model real-world scenarios by identifying and leveraging the connections between various entities.

Lastly, we use nodes and edges in cases like analyzing connectivity, pathfinding algorithms. In essence, nodes and edges in graph theory provide a simple yet powerful way to represent relationships between entities, enabling us to unravel intricate connections and patterns.

### 3.1.3 Centrality in Graph Theory

**Definition 3.24** *Centrality* - deals with distinguishing important nodes in a graph.

In graph theory, centrality measures quantify the relative importance of nodes within a network. They provide a way to identify key nodes that play significant roles in the structure and dynamics of the graph. Centrality can be defined in various ways depending on the specific context and requirements of the analysis. Some common centrality measures include:

1. **Degree Centrality:** The simplest centrality measure, which is based on the number of edges incident to a node. Nodes with a high degree centrality are those that are directly connected to many other nodes in the network.
2. **Betweenness Centrality:** Measures the extent to which a node lies on the shortest paths between other nodes in the network. Nodes with high betweenness centrality are crucial for maintaining connectivity and facilitating communication within the network.

3. **Closeness Centrality:** Measures how close a node is to all other nodes in the network in terms of geodesic distance (the shortest path length). Nodes with high closeness centrality can efficiently interact with other nodes in the network.
4. **Eigenvector Centrality:** Takes into account not only the number of connections a node has but also the importance of its neighbors. Nodes with high eigenvector centrality are connected to other nodes that are themselves well-connected.
5. **Information Centrality:** Measures the influence of a node in terms of controlling the flow of information through the network.

Different centrality measures capture different aspects of node importance within a network, and the choice of which measure to use depends on the specific characteristics of the network and the goals of the analysis.

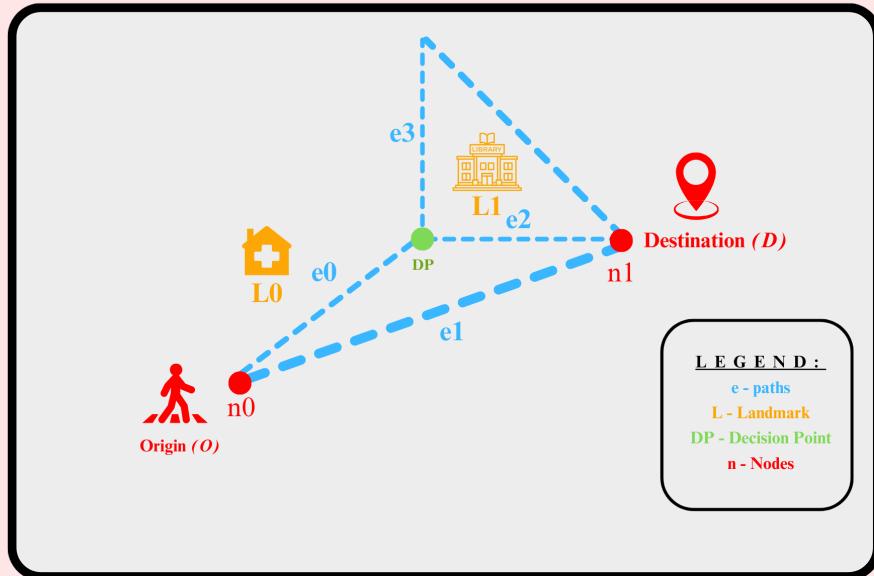
### 3.1.4 C. Cognitive Map (CM)

**Definition 3.25** *Cognitive Map* - a mental representation of the spatial layout and features of the real world urban environment, translated into a virtual environment.

Cognitive Map [3.25] is based on the perception and memory of meaningful urban elements, such as landmarks that help individuals orient themselves and navigate in an urban environment. In a cognitive map, there are certain nodes and edges that represent the origin and destination of a route. In addition, all nodes represent the decision points, while the edges represent the paths an individual can take to reach their destination.

Now, we translate University ( $U$ ) into a cognitive map.

## Example 3: Cognitive Map Example

**Figure 3.4:** Cognitive Map Example

In this example, an individual has a cognitive map of the University ( $U$ ). Now, the individual has their origin, which represent their starting point, and their destination, which represent their end point. The individual only needs to decide on decision point that connect the three different paths to reach the destination. The cognitive map helps the individual navigate the urban environment and make decisions on which path to take.

As we proceed, we see that Cognitive maps are important for modelling pedestrian route choice behavior, as they reflect how individuals use different types of information to formulate their routes. Agents in the ABM were provided cognitive maps that incorporate urban elements and path attributes.

### 3.1.5 D. Urban Elements and Path Attributes (UE & PA)

**Definition 3.26** *Element* - A part or aspect of something abstract, especially one that is essential or characteristic.

**Definition 3.27** *Feature* - A typical quality or an important part of something.

**Definition 3.28** *Perception* - State of being or process of becoming aware of something through the human senses.

**Definition 3.29** *Urban Elements* - The **feature** of the **urban environment** that affect the perception and behavior of pedestrians.

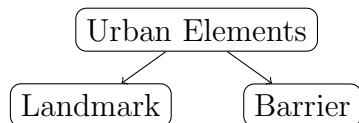
**Definition 3.30** *Property* - An attribute, quality, or characteristic of something.

**Definition 3.31** *Landmark* - Specific, distinctive, and easily identifiable objects or structures that serve as reference points for orientation and navigation in an urban environment.

**Definition 3.32** *Barrier* - Elements that hinder or prevent pedestrian movement across the University.

**Definition 3.33** *Path Attributes* - The **property** of the path segments that affect the pedestrian perception of the environment and influence their route choice behavior.

The landmarks, regions, barriers, and path attributes are described as follows:



**Figure 3.5:** Illustration of the Urban Elements

#### Example 4: Landmark and Barrier Example

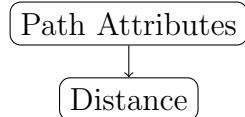
- Landmarks: They can be **local**, meaning on-route landmarks; or **global**, meaning distant landmarks.

Specific example of Local Landmark based on University (U) in Figure 3.4 is  $L_0$ . If the individual takes the path  $e_1$ , both  $L_0$  and  $L_1$  are considered Global landmarks.

General examples are buildings, monuments, statues, fountains, etc.

- Barriers: They can be natural (such as water or green areas) or artificial (such as highways or railways). Individuals can perceive barriers as attractive or repulsive, depending on their preference or aversion for certain street segments.

**Path Attributes:** These are attributes of street segments that influence the individual's perception of their desirability or convenience for walking. We consider one type of path attributes, namely the Distance.



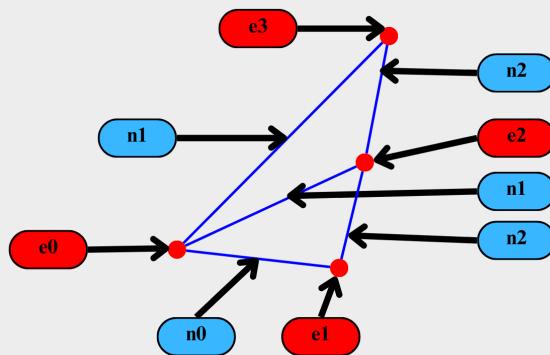
**Figure 3.6:** Illustration of the Path Attribute

### 3.1.6 E. Dual Graph (DG)

A method in space syntax that considers edges as nodes and nodes as edges. In urban street networks, large avenues made of several segments become single nodes, while intersections with other avenues or streets become links (edges). This method is particularly useful for revealing hierarchical structures in a planar network. Based on a planar network such as urban streets [3.2], space syntax proposes to consider line segments differently from traditional graph theory, where links (edges) are streets and intersections are nodes (vertices). The axial map [3.3] first defines the line segments based on their continuity.

Now, the dual graph represents those segments as nodes and their intersections as links. It allows for discovering hidden structural properties of planar networks, such as the hierarchy and the true connectivity of the network.

## Example 5: Dual Graph Example

**Figure 3.7:** Dual Graph Example

In this dual graph representation of the urban environment University ( $U$ ), the line segments are represented as nodes, while the intersections are represented as links. This representation allows for a different perspective on the connectivity and structure of the urban environment, revealing hidden patterns and relationships that may not be immediately apparent in the original graph.

We can see that all the nodes are connected to each other, forming a fully connected graph.

### 3.1.7 F. Behavior Mechanisms (BM)

**Definition 3.34** *Behavior* - Refers to the actions or activities that individuals perform within the environment.

**Definition 3.35** *Behavior Mechanism* - Rules and parameters that regulate how the

individuals use **urban element** and road attributes to formulate their **routes**.

**Behavior Mechanism (BM)** is from a pedestrian's route choice behavior, which is then computer inputted to be an agent's route choice behavior.

**Definition 3.36** *heuristics* - Practical problem-solving techniques or methods used to find quick and effective solutions when dealing with complex issues.

**Definition 3.37** *homogeneous* - of the same kind or alike.

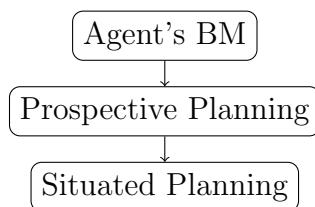
**Definition 3.38** *heterogeneous* - diverse in character or content.

**Definition 3.39** *Stochastic Discrete Parameter* - A type of parameter in the agent-based model that regulates the agent's behavior by assigning a probability value to a certain choice or action.

**Definition 3.40** *Stochastic Preference Parameter* - A type of parameter that regulate an agent's reliance on urban elements and path attributes in their Cognitive Map. These parameters influence the agent's decisions and interactions with the environment during navigation, such as preference for segments along natural barriers or aversion to crossing severing barriers.

A base BM, termed as probability ( $p$ ), is a general term referring to an agent using a certain BM, **Stochastic Discrete Parameter** or **Stochastic Preference Parameter** in the ABM. When subscript letters are included with  $p$ , the resulting term is to indicate the specific BM or referred preference parameter of  $p$ .

Here is a simplified illustration of the Agent's Behavior Mechanism:



**Figure 3.8:** Illustration of the Agent's Behavior Mechanism

Here is the overview of the Agent's Prospective and Situated Planning:

**Definition 3.41** *Prospective Planning* - A phase of route choice behavior in which an agent formulates a rough idea of the route between an origin and a destination, based on the information available in the agent's Cognitive Map.

**Definition 3.42** *Situated Planning* - A phase of route choice behavior in which an agent makes real-time decisions based on the immediate environment and the agent's current location.

## 3.2 Questionnaire and Data

### 3.2.1 G. Empirical Research (ER)

**Definition 3.43** *Empirical* - Originating in or based on real world observation or experience.

**Definition 3.44** *Set* -A collection of distinct objects forming a group.

**Definition 3.45** *Data* -A collection of facts, statistics, or information that are represented in various formats. These can include numbers, words, measurements, observations or even just descriptions of things.

**Definition 3.46** *Questionnaire* - A **set** of questions used to collect information or **data** from the humans in relation to the given problem.

**Definition 3.47** *Question* - A written statement designed to elicit specific information, opinions, or responses from an **individual**.

Empirical Research is a way of gaining knowledge by using the data, called empirical evidence, obtained from direct or indirect observation or experience.

**Definition 3.48** *Route* - A way or course taken in getting from a starting point to a destination.

**Definition 3.49** *Preference* -A greater liking for one alternative over another or others.

#### Example 6: Sample Questionnaire

To obtain data from individuals, **questions** are needed. Some questions inside a **questionnaire** may include:

##### Section 1: Basic Demographic information

1. What is your sex at birth?  Male  Female  Prefer not to say.
2. How old (in years) are you?
3. What are all categories that apply to you when in University (*U*)?

- I work or used to work
- I study or used to study

## Section 2: Walking Behavior in University

In this section, you will be asked about your walking behavior in the University ( $U$ ). Think about walking trips at least 200 meters long (approximately) and that are not aimed at reaching your private vehicle.

1. If you are to go from Origin ( $O$ ) to Destination ( $D$ ), Do you consider DISTANCE?
2. If you are to go from Origin ( $O$ ) to Destination ( $D$ ), Do you consider LANDMARK?

The individual's ( $IP$ ) answers on the questionnaire are called **empirical evidence or pedestrian data** ( $P_1$ ).

The aim of the **questionnaire** is to understand the factors that influence a person's choice of route when navigating through a University ( $U$ ). The questions are designed to gather information on the following:

1. The respondent's demographic and relation to the University ( $U$ ) ( $C$ ).
2. The respondent's preferred route choice strategy when traveling from Origin ( $O_1$ ) to Destination ( $D_1$ ). Either considering or not considering Distance.
3. The respondent's preferred route choice strategy when traveling from Origin ( $O_1$ ) to Destination ( $D_1$ ). Either considering or not considering Landmark.

This information could be used for urban planning purposes, such as improving road networks, traffic management, and University ( $U$ ) infrastructure based on residents' preferences and behaviors.

**Definition 3.50** *Operation* -An action that is carried out to accomplish a given task.

**Definition 3.51** *Parameter* -A numerical or other measurable factor forming one of a

set that defines a system or sets the conditions of its **operation**.

### 3.3 Cluster Analysis and Agent Typologies

#### 3.3.1 H. Z-Score (ZS)

Before we can understand the Z-Score, we need to understand the concept of mean and standard deviation.

**Definition 3.52** *Mean* - The average of a set of numbers.

**Definition 3.53** *Variance* - A measure of the spread between numbers in a dataset.

**Definition 3.54** *Standard Deviation* - A measure of the amount of variation or dispersion of a set of values.

Mean is calculated by adding up all the numbers in a set and then dividing by the number of values in the set. Standard deviation is calculated by finding the square root of the variance. Variance is calculated by finding the average of the squared differences between each value and the mean.

**Definition 3.55** *Z-Score* - A statistical measure that quantifies the number of standard deviations a data point is from the mean of a dataset.

To compute for the Z-score, we use the formula:

$$Z = \frac{X - \mu}{\sigma} \quad (3.1)$$

Where  $X$  is the data point,  $\mu$  is the mean of the dataset, and  $\sigma$  is the standard deviation of the dataset.

Now consider the following example:

### Example 7: Z-Score Sample Computation

Consider the answers of five individuals to Section 2 Question 2 of the sample questionnaire [6]. The answers are as follows: [1, 0, 1, 0, 1]. The mean of this set is 0.6, and the standard deviation is 0.5. The Z-Scores for each individual's answer are calculated as follows:

$$Z_1 = \frac{1 - 0.6}{0.5} = 0.8$$

$$Z_2 = \frac{0 - 0.6}{0.5} = -1.2$$

$$Z_3 = \frac{1 - 0.6}{0.5} = 0.8$$

$$Z_4 = \frac{0 - 0.6}{0.5} = -1.2$$

$$Z_5 = \frac{1 - 0.6}{0.5} = 0.8$$

The Z-Score helps in understanding how far a data point is from the mean of the dataset. It is a useful measure for identifying outliers and understanding the distribution of data points in a dataset.

### 3.3.2 I. K-Means Clustering and Algorithm (KMCA)

**Definition 3.56** *Clustering* - The process of grouping similar data points together.

**Definition 3.57** *Algorithm* - A set of rules or steps used to solve a problem or accomplish a task.

**Definition 3.58** *K-Means Clustering* - A type of clustering algorithm that aims to partition  $n$  observations into  $k$  clusters in which each observation belongs to the cluster with the nearest mean.

The K-Means Clustering Algorithm is an iterative algorithm that divides a group of  $n$  data points into  $k$  clusters based on the mean of the data points. The algorithm works as follows:

1. Choose the number of clusters  $k$ .
2. Randomly select  $k$  data points as the initial cluster centroids.
3. Assign each data point to the nearest cluster centroid.
4. Calculate the mean of each cluster and update the cluster centroids.
5. Repeat steps 3 and 4 until the cluster centroids no longer change significantly.
6. The algorithm converges when the cluster centroids no longer change significantly.
7. The final clusters are formed based on the mean of the data points in each cluster.
8. The algorithm assigns each data point to the cluster with the nearest mean.

### 3.3.3 J. Silhouette Score (SS)

**Definition 3.59** *Silhouette Score* - A measure of how similar an object is to its own cluster compared to other clusters.

The Silhouette Score is a measure of how similar an object is to its own cluster compared to other clusters. It ranges from -1 to 1, where a high value indicates that the object is well-matched to its own cluster and poorly matched to neighboring clusters. The Silhouette Score is calculated as follows:

$$S = \frac{b - a}{\max(a, b)} \quad (3.2)$$

Where  $a$  is the mean distance between a data point and all other points in the same cluster, and  $b$  is the mean distance between a data point and all other points in the nearest cluster. The Silhouette Score is used to evaluate the quality of clustering algorithms and helps in determining the optimal number of clusters.

### 3.3.4 K. Variance Ratio Criterion (VRC)

**Definition 3.60** *Criterion* - A principle or standard by which something may be judged or decided.

**Definition 3.61** *Variance Ratio Criterion* - A criterion used to determine the optimal number of clusters in a clustering algorithm.

The Variance Ratio Criterion (VRC) is a criterion used to determine the optimal number of clusters in a clustering algorithm. It is based on the ratio of the between-cluster variance to the within-cluster variance. The VRC is calculated as follows:

$$VRC = \frac{\text{Between-Cluster Variance}}{\text{Within-Cluster Variance}} \quad (3.3)$$

The VRC helps in identifying the number of clusters that best represent the underlying structure of the data. A higher VRC value indicates a better clustering solution.

### 3.3.5 L. Agent Typologies (AT)

**Definition 3.62** *Typology* - A classification system based on common characteristics or traits.

**Definition 3.63** *Agent Typologies* - A classification system that categorizes agents based on their behavior, preferences, or other characteristics.

Agent Typologies are used to classify agents based on their behavior, preferences, or other characteristics. By categorizing agents into different typologies, we can better understand their interactions with the environment and how they make decisions. Agent Typologies can help in developing more realistic agent-based models and predicting the behavior of different types of agents in various scenarios.

## 3.4 ABM and Pedestrian Flow

### 3.4.1 M. Individual-Pedestrian-Agent Terminologies (IPAT)

Here are the different terms of the humans involved and observed in the case study according to the use cases. There are observed implicit use cases for the terms: **agent**, **pedestrian**, and **individual**.

**Definition 3.64** *Real World* - Refers to the physical world around humans, where things exist, events happen, and humans live their lives. It includes everything that can seen, touched, heard, and experienced directly. This also encompasses the everyday surroundings, nature, buildings, humans, objects, and all the things that make up the physical reality.

**Definition 3.65** *Virtual World* - Refers to the simulated world in a computer, where events are based on the interaction of the objects inside the environment.

**Definition 3.66** *Individual* - General term for the virtual world or real-world human beings that are studied. These are human beings that answered the research study **questionnaire**.

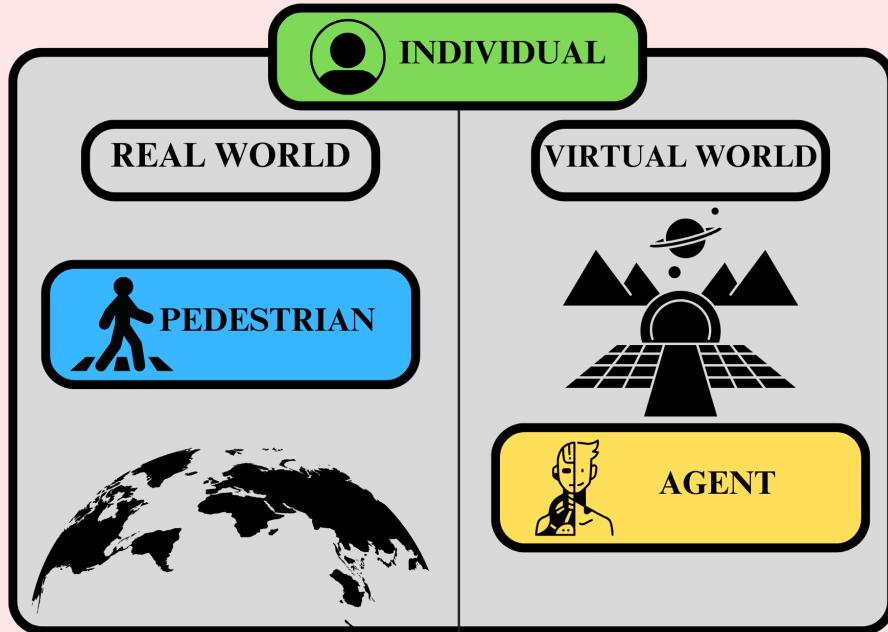
**Definition 3.67** *Pedestrian* - An individual in the real-world who is walking on the street or sidewalk.

**Definition 3.68** *Agent* - An autonomous simulation that represents an individual within a system. These individuals in a virtual world are programmed to follow a set of rules and interact with each other and their environment. They can have attributes, behaviors, and the ability to make decisions based on their surroundings and the information they possess.

To start with, An **individual** is a general term that refers to *any human being*, regardless of their mode of transportation or walking activity. Next, a **pedestrian** is a person who walks in an **urban environment** for different purposes, such as commuting, working, studying, or leisure. Lastly, an **agent** is a **computer program** that simulates the behavior of an individual in a complex system.

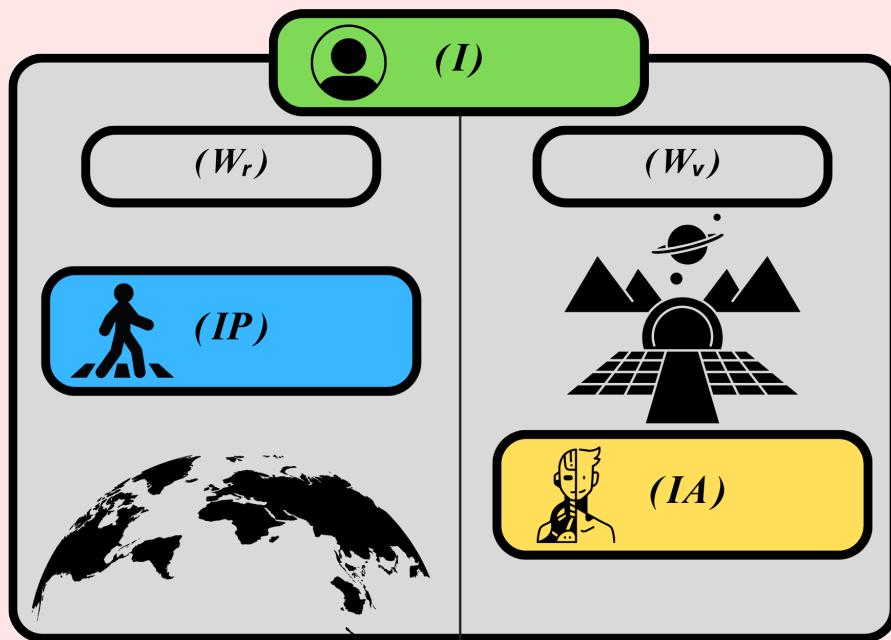
## Example 8: Illustration of Individual Collection

To show the collection of IPA, here is an illustration:



**Figure 3.9:** Individual-Pedestrian-Agent Illustration

From here on, this manuscript shall use the following mathematical naming convention: Let  $(I)$  be the individual. Individuals are the set of humans involved in the research study. Now, there are two types of individuals based on their worlds ( $W$ ), the Real World ( $W_r$ ) and Virtual World ( $W_v$ ).  $W_r$  individuals are pedestrian ( $IP$ ), and  $W_v$  individuals are agents ( $IA$ ).

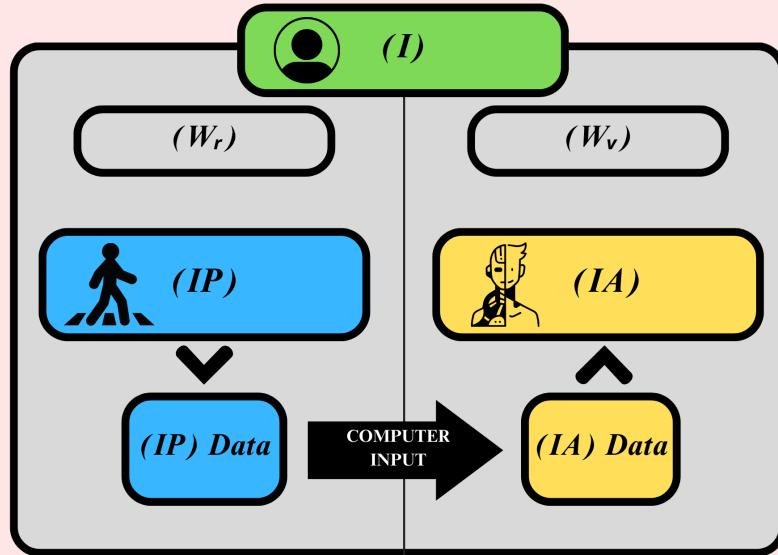


**Figure 3.10:** Mathematical Individual-Pedestrian-Agent Illustration

Now, Example 8 only show the human collection. To illustrate the human relations within the set we use Example 9.

**Example 9:** Illustration of Individual Relation

To show the relation of IPA, here is an illustration:



**Figure 3.11:** Individual-Pedestrian-Agent Relation

Based on figure 3.11, Data are collected from pedestrian (*IP*). The (*IP*) Data came from questionnaire as shown in Example 6. Now, these (*IP*) Data is inputted on the computer to convert (*IP*) Data into (*IA*) Data. These (*IA*) Data then become the basis of (*IA*) behavior.

*IP* data is influenced by urban elements and path attributes, which is explored in the next subsection.

### 3.4.2 N. Agent-Based Modeling (ABM)

**Definition 3.69** *Model* - Simplified representation or description of a real-world individual, situation, system, or phenomenon to better understand or study it.

**Definition 3.70** *Simulation* - Refers to the process of creating a **model** or imitation of a real-world system, process, or phenomenon on a computer or in a controlled environment. This imitation is designed to mimic the behavior and characteristics of the real-world

system to study, analyze, or predict the individuals' behavior without directly interacting with the real-world system.

**Definition 3.71** *Paradigm* - Refers to a widely accepted and commonly practiced way of doing things within the field or discipline of Computer Science. It encompasses the fundamental beliefs, methods, and assumptions that the case study researchers share and use as a structured outline for conducting the research study.

**Definition 3.72** *Agent-Based Modeling* - This is the computer **simulation paradigm** used by the research to study the interactions between the humans called **agents**, things, places, and time.

**Definition 3.73** *Variance* - A statistical measurement of the spread between numbers in a data set.

**Definition 3.74** *Stochastic Model* - A method for predicting statistical properties of possible outcomes by accounting for random **variance** in one or more **parameter** over time.

**Definition 3.75** *Bottom-up Approach* - Starting with specific details, observations, or individual elements and then gradually building up to form a larger understanding or conclusion.

Second, the Agent-Based Modeling (ABM) a **stochastic model** with **bottom-up approach** meaning ABM is a method to predict possible outcomes' statistical properties through accounting random variance (More on random variance in Methodology) in one or more parameters over time [4].

In a **bottom-up approach**, **models** are used to construct an experimental plan through simulations. In addition, bottom-up approach starts with empirical evidence or observations, focuses on pattern identification, inductive (specific to general observations), exploratory in areas with limited-existing knowledge, and encourages open-ended exploration and innovation [52]. The bottom up approach in the case study will be more apparent over the course of the proceeding abstraction.

**Definition 3.76** *Macro-level Pattern* - Refers to a large-scale trend or observation that can be observed when looking at data or phenomena from a broader perspective. It's like seeing the "big picture" or identifying common characteristics or behaviors that occur on a grand scale.

**Definition 3.77** *Micro-level Pattern* - Refers to a specific, detailed, or small-scale trend, behavior, or observation within a larger system or data set. It involves examining individual components, actions, or details to identify specific patterns or characteristics.

**Definition 3.78** *Complex System* - System made up of many components or agents that interact with each other, and these interactions can lead to intricate and sometimes unpredictable behavior or patterns.

*Macro-level patterns* emerge from the ABM. These **Macro-level patterns** are the collective and emergent outcomes of individual behavior and interactions in a **complex system**.

#### Example 10: Pedestrian Route Choice Strategies and Patterns

The different route choice strategies ( $RCS$ ) of pedestrian ( $P$ ) can result in different patterns of pedestrian movement ( $P_m$ ) flows in urban spaces ( $US$ ). From this example, the patterns of pedestrian movement flows (collective and emergent outcomes) in urban spaces (complex system) are the macro-level patterns, while the different route choice strategies of pedestrian agents (individual behavior) are the micro-level patterns. Let:

$RCS$  – the set of different route choice strategies of pedestrian agents.

$P$  – the set of pedestrian.

$P_b$  – the behavior of pedestrian.

$P_m$  – the patterns of pedestrian movement flows in urban spaces.

$US$  – the urban spaces.

We can then express the relationship as follows:

$$P_m(US) = f(RCS, P_b)$$

This equation signifies that the patterns of pedestrian movement flows ( $P_m$ ) in urban spaces ( $US$ ) are a function ( $f$ ) of the different route choice strategies ( $RCS$ ) of pedestrian agents and the set of pedestrian agents ( $P$ ).

This mathematical representation reflects the idea that the patterns of pedestrian movement flows are influenced by both the RCS of pedestrians and the collective behavior of all pedestrians in urban spaces.

### 3.4.3 O. Monte-Carlo Simulation (MCS)

**Definition 3.79** *Monte-Carlo Simulation* - A computerized mathematical technique that uses random sampling to model the behavior of complex systems or processes.

Monte-Carlo Simulation is a computerized mathematical technique that uses random sampling to model the behavior of complex systems or processes. It is used to understand the impact of risk and uncertainty in prediction and forecasting models. Monte-Carlo Simulation involves generating random numbers to simulate the behavior of a system over time. The simulation is repeated multiple times to estimate the range of possible outcomes and the likelihood of each outcome occurring.

**Definition 3.80** *Random Sampling* - A method of selecting a subset of individuals or data points from a larger population in a way that each individual or data point has an equal chance of being selected.

**Definition 3.81** *Prediction* - A statement about what will happen in the future based on evidence or reasoning.

**Definition 3.82** *Forecasting* - The process of making predictions about the future based on past and present data and analysis.

**Definition 3.83** *Outcome* - The result or consequence of an action, event, or process.

**Definition 3.84** *Likelihood* - The probability of a particular outcome or event occurring.

**Definition 3.85** *Range* - The area or extent covered by something.

**Definition 3.86** *Uncertainty* - The state of being uncertain or not knowing what will happen in the future.

**Definition 3.87** *Risk* - The possibility of loss or other adverse or unwelcome outcomes.

## 3.5 Configurations Comparison

### 3.5.1 P. Configurations

**Definition 3.88** *Configuration* - The arrangement or set-up of elements or components in a system.

**Definition 3.89** *Parameter* - A numerical or other measurable factor that defines a system or sets the conditions of its operation.

**Definition 3.90** *Configuration Comparison* - The process of evaluating and contrasting different arrangements or set-ups of elements or components in a system to determine the most effective or optimal configuration.

**Definition 3.91** *Optimal* - The best or most effective solution or arrangement in a given situation.

**Definition 3.92** *Effective* - Producing the intended or desired result.

**Definition 3.93** *Contrast* - To compare in order to show differences.

**Definition 3.94** *Evaluation* - The process of assessing or judging the effectiveness, value, or quality of something.

**Definition 3.95** *Arrangement* - The way in which things are organized or set out.

**Definition 3.96** *Component* - A part or element of a larger system or structure.

**Definition 3.97** *Random* - Made, done, or happening without method or conscious decision.

### 3.5.2 Q. Wilcoxon Signed-Rank Test (WSRT)

**Definition 3.98** *Test* - A procedure or method used to evaluate or measure the performance, quality, or reliability of something.

**Definition 3.99** *Wilcoxon Signed-Rank Test* - A non-parametric statistical test used to compare two related samples to assess whether their population means differ significantly.

**Definition 3.100** *Non-parametric* - A statistical method that does not assume a specific distribution for the data.

**Definition 3.101** *Statistical* - Relating to the use of data and numbers to analyze and interpret information.

**Definition 3.102** *Significance* - The quality of being worthy of attention or importance.

**Definition 3.103** *Population* - The entire group of individuals or items that are the subject of a study.

**Definition 3.104** *Volume* - The amount of space that a substance or object occupies.

### 3.5.3 R. Comparison

**Definition 3.105** *Comparison* - The process of examining the similarities and differences between two or more things.

**Definition 3.106** *Similarity* - The quality or state of being alike or similar.

**Definition 3.107** *Difference* - A point or way in which people or things are not the same.

**Definition 3.108** *Examination* - The act of looking at or considering something carefully in order to discover something.

**Definition 3.109** *Diverse* - Showing a great deal of variety or difference.

## 3.6 Small Scale Replication (SSR)

We consider two Route Choice Strategies:

- Use of Distance. Considering the distance between the origin to destination.
- Use of Landmark. Considering landmarks to orient pedestrians during the route going from the origin to destination.

### 3.6.1 Step 1. Cognitive Map from Urban Environment

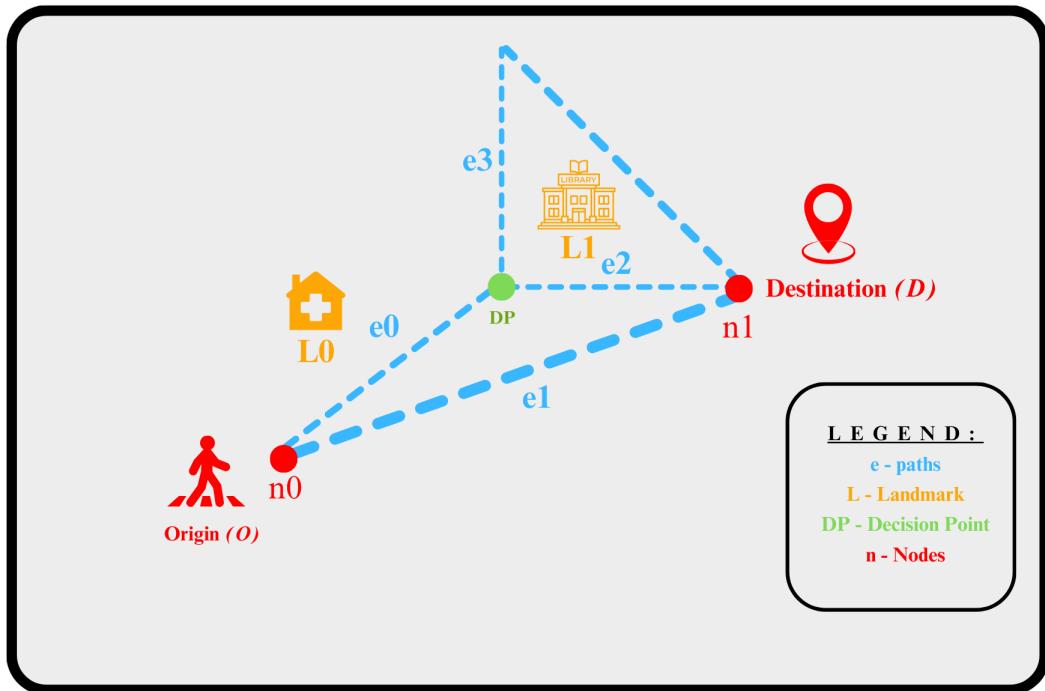


Figure 3.12: SSR Cognitive Map

### 3.6.2 Step 2. Summarized Empirical Data

Consider the questionnaire based on the route choice strategies:

### 1. Distance Use

- **Q1:** Do you consider **distance** on your route from Origin ( $O$ ) to Destination ( $D$ )? Yes or No.

### 2. Landmark Use

- **Q2:** Do you consider **landmark** on your route from Origin ( $O$ ) to Destination ( $D$ )? Yes or No.

For this questionnaire, Yes = 1 and No = 0. We consider five responses ( $S_n$  where  $n = 1, 2, 3, 4, 5$ ) to the questionnaire:

$S_n$	Distance	Landmark
1	1	0
2	0	0
3	1	1
4	0	1
5	1	1

**Table 3.1:** Example Questionnaire Responses

Based on the responses, we have:

- Distance:  $\frac{3}{5} = 0.6$ . 60% of the respondents consider distance on their route.
- Landmark:  $\frac{3}{5} = 0.6$ . 60% of the respondents consider landmarks on their route.

Now, the survey data indicates that 60% of the respondents consider distance and 60% consider landmarks during their route. Meanwhile, 40% of the respondents do not consider distance, and 40% do not consider landmarks. The results suggest that the respondents use a combination of distance and landmarks to navigate the urban environment.

We now get the probability of crossing the paths based on the empirical data:

Suppose that each landmark entails the same weight. That is, for each landmark  $L_i$  (where  $i \in \{1, 2, \dots, n-1\}$ ), the weight  $w_i$  is equal to 1. In other words,  $w_i = 1$  for all  $i$ .

$$w_i = 1 \quad \text{for } i = 1, 2, \dots, n - 1$$

Suppose that without the empirical data, the probability of choosing each path is equal. That is,  $p_{e_n} = \frac{1}{e_n}$ , where  $p$  is the probability and  $n \in \{1, 2, \dots, n - 1\}$ .

For the case of the Cognitive Map in Figure 3.12, we have the following probabilities when the empirical data is not considered:

$$p_{e_n} = \frac{1}{e_n} = \frac{1}{4} = 0.25$$

Now, we consider the survey data. Each path will have different weights based on the formula:

$$T = \frac{Q1}{Q1 + Q2} + \frac{Q1}{Q1 + Q2} \quad (3.4)$$

Where Total (T) must always be 1.

For the Cognitive Map in Figure 3.12, consider all the possible combination of choices. For this, we use the formula

$$C = 2^n \quad (3.5)$$

Where C is the number of combinations and n is the number of route choices.

Now,  $C = 2^n = 2^2 = 4$ . There are 4 possible combinations of choices for the Cognitive Map in Figure 3.12, considering the route choices: Distance (D) and Landmark (L).

	D	L
Case 1	0	0
Case 2	0	1
Case 3	1	0
Case 4	1	1

**Table 3.2:** Example Path Probabilities

We compute the probabilities for each path:

**Case 1:** If distance and landmark are not considered, we use the base probability for all paths, that is each path has equal probability of being chosen.

$$pe_n = \frac{1}{n}$$

where  $n = \{1, 2, \dots, n - 1\}$

$$pe_1 = \frac{1}{4} = 0.25, pe_2 = \frac{1}{4} = 0.25, pe_3 = \frac{1}{4} = 0.25, pe_4 = \frac{1}{4} = 0.25$$

**Case 2:** If distance is considered, but landmark is not considered, we use the probability based on the survey data.

$pe_1 = \frac{3}{5} = 0.60$ , since  $e_1$  has the shortest path from origin to destination and does not have landmark.

$$pe_2 = \frac{\frac{2}{5}}{4} = \frac{0.40}{3} = 0.1333$$

$$pe_3 = (\frac{2}{5} - pe_2)(0.60) = (0.40 - 0.1333)(0.60) = 0.2667(0.60) = 0.16$$

$$pe_4 = (\frac{2}{5} - pe_2)(0.40) = (0.40 - 0.1333)(0.40) = 0.2667(0.40) = 0.1067$$

**Case 3:** If distance is not considered, but landmark is considered, we use the probability based on the survey data.

$$pe_1 = \frac{2}{5} - 0.25 = 0.40 - 0.25 = 0.15, \text{ since } e_1 \text{ does not have landmark.}$$

$$pe_2 = \frac{\frac{3}{5}}{3} + 0.25 = \frac{0.60}{3} + 0.25 = 0.20 + 0.25 = 0.45$$

$$pe_3 = \frac{0.60}{3} = 0.20$$

$$pe_4 = \frac{0.60}{3} = 0.20$$

**Case 4:** If distance and landmark are considered, we use the probability based on the survey data.

For this case, we consider the probability of the respondents who consider both distance and landmark.

$pe_1 = \frac{1}{4} = 0.25(0.40) = 0.10$ , since  $e_1$  has the shortest path from origin to destination but does not have landmark.

$$pe_2 = \frac{1}{4} = 0.25 + (0.25(0.60)) = 0.40$$

$$pe_3 = \frac{1}{4} = 0.25 + (0.25(0.60)) = 0.40$$

$$pe_4 = \frac{1}{4} = 0.25(0.40) = 0.10$$

We tabulate all the resulting probabilities in Table 3.2.

Path	Case 1	Case 2	Case 3	Case 4
1	0.25	0.60	0.15	0.10
2	0.25	0.1333	0.45	0.40
3	0.25	0.16	0.20	0.40
4	0.25	0.1067	0.20	0.10

**Table 3.3:** Path Probabilities for Different Cases

Now, we evaluated the results from the questionnaire.

We present the summarized data below. The Demographic Information: General Information, Age Categories, Relationship to the University (U), and statistical data.

### Socio-Demographic

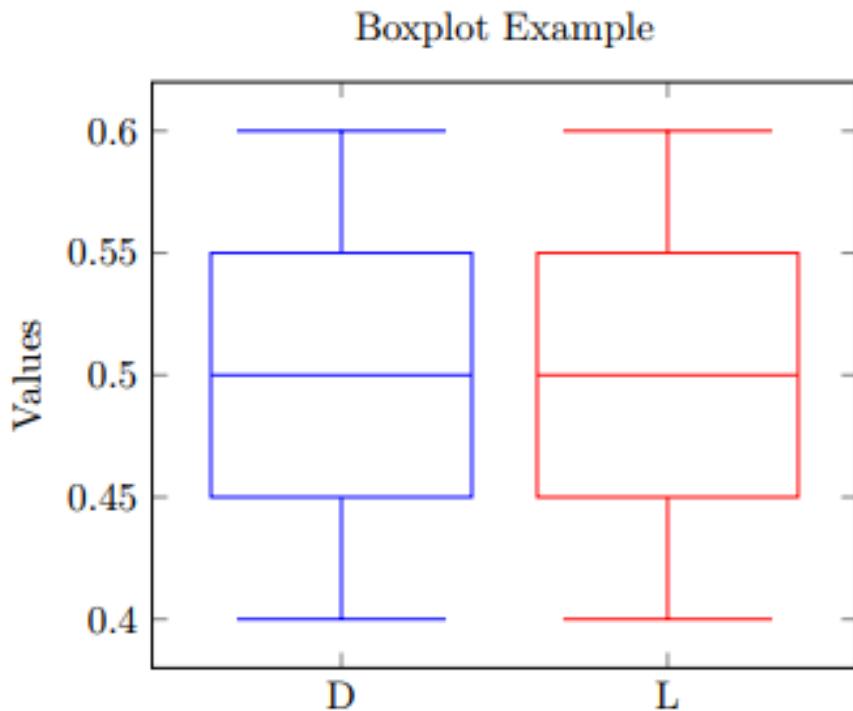
	Demographics	Frequency	Percentage
Gender	Female	2	40.0%
	Male	2	40.0%
	Prefer not to declare	1	20.0%
Age	< 18	0	0.0%
	18–25	5	100.0%
	26–33	0	0.0%
	34–41	0	0.0%
	42–49	0	0.0%
	50–57	0	0.0%
	58–65	0	0.0%
	66–73	0	0.0%
	> 74	0	0.0%
Statistical Data	Mean Age	21	
	Standard Deviation Age	0	
	Mean Response Duration	1 minute	
Relationship with UPB	Study/Studied here	5	100.0%

**Table 3.4:** SSR Socio-Demographic Profile of the Participants (N = 5).

The demographic distribution reveals that the majority of participants identified as female (40.0%) and male (40.0%), followed by Non-binary (20.0%). The mean age of participants was 21, with a standard deviation of 0. The average response duration of participation in the study was 1 minute.

All participants had a relationship with the University ( $U$ ), with 100.0% of participants having studied or currently studying at the institution.

### Visualizing Overall Importance of Each Variable (Probabilities)



**Figure 3.13:** Route Choice Importance in Pedestrian Route Choice

The figure provides a comprehensive visualization of the overall importance of various Route Choice in influencing pedestrian route. The data is presented in probabilities to offer quantitative insights. Now, both the box plot for Distance (D) and Landmark (L) shows that individuals hold quite different opinions on the importance of distance in their route choice. The box plot shows that the study participants consider distance and landmark as equal factors on their route choice.

### 3.6.3 Step 3. Cluster Analysis and Agent Typologies

**Intuitively assigning clusters:**

We use the previously computed Z-scores in Example 7 to assign the participants to clusters. We consider the Z-scores for the Distance (D) and Landmark (L) variables. The Z-scores are as follows:

$$\begin{bmatrix} 0.8 & 0.8 \\ 0.8 & 0.8 \\ 0.8 & 0.8 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1.2 & -1.2 \\ -1.2 & -1.2 \end{bmatrix}$$

Centroids:

$$\text{Centroid 1} = (0.8, 0.8)$$

$$\text{Centroid 2} = (-1.2, -1.2)$$

#### Assign Data Points to Clusters

Points  $(0.8, 0.8)$  go to Centroid 1, and points  $(-1.2, -1.2)$  go to Centroid 2.

#### Silhouette Score Calculation

The silhouette score  $s(i)$  for a point  $i$  is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

Where:  $-a(i)$  is the average distance of  $i$  to all other points in the same cluster.  $-b(i)$  is the minimum average distance of  $i$  to points in a different cluster.

For points at  $(0.8, 0.8)$ :

$$a(i) = 0 \quad (\text{since they are identical})$$

$$b(i) = \text{distance to the other cluster centroid} = \sqrt{(0.8 - (-1.2))^2 + (0.8 - (-1.2))^2} = \sqrt{8} \approx 2.83$$

For points at  $(-1.2, -1.2)$ :

$$a(i) = 0 \quad (\text{since they are identical})$$

$$b(i) = \text{distance to the other cluster centroid} = \sqrt{8} \approx 2.83$$

Silhouette score for each point:

$$s(i) = \frac{2.83 - 0}{\max(0, 2.83)} = 1$$

Average silhouette score for  $n = 2$ :

$$\text{Silhouette Score} = 1$$

### Variance Ratio Criterion (VRC)

VRC is defined as:

$$\text{VRC} = \frac{\text{between-group dispersion}}{\text{within-group dispersion}} \times \frac{N - k}{k - 1}$$

For  $n = 2$ : -Between-group dispersion (B):

$$B = 3 \times \| (0.8, 0.8) - (-1.2, -1.2) \|^2 + 2 \times \| (-1.2, -1.2) - (0.8, 0.8) \|^2 = 3 \times 8 + 2 \times 8 = 40$$

-Within-group dispersion (W):

$$W = 0 \quad (\text{all points in each cluster are identical})$$

Therefore, VRC for  $n = 2$  is infinite since W is 0.

Since we arrived at a Silhouette Score of 1, and VRC of 0 in  $n=2$  clusters, we can conclude that the clusters are well-defined and optimized. However, if we want to show  $n= 3$  and  $n=4$  clusters, it will yield to the same Silhouette Score and VRC of 0, since the sample empirical data have identical points.

### Cluster Analysis

Number of Clusters	Silhouette Score	Variance Ratio Coefficient
2	1.00	0.00

**Table 3.5:** Number of clusters and Silhouette Score using K-means Algorithm, in comparison to VRC values.

Table 4.3 presents the results of applying the K-means algorithm with two numbers of clusters. The parameters considered include the algorithm used, the number of clusters ( $n$  clusters), the Silhouette Score, and the Variance Ration Criterion. The Silhouette Score, a measure of how well-defined the clusters are, is used to assess the quality of each clustering configuration. A higher Silhouette Score indicates better-defined clusters. Since the silhouette score is 1, the clusters are well-defined and optimized. The VRC is used to evaluate the dispersion of data points within and between clusters. A VRC of 0 indicates that the clusters are well-separated and distinct.

### Agent Typologies

The cluster analysis results in two distinct groups of agents. The agents in each cluster exhibit similar characteristics and behaviors, which can be used to define agent typologies. The agent typologies are as follows:

- **Cluster 1:** Agents in this cluster consider both distance and landmarks in their route choice strategies. They are more likely to choose paths that consider distance and use landmarks to navigate the urban environment.
- **Cluster 2:** Agents in this cluster do not consider distance or landmarks in their route choice strategies. They are less likely to choose paths based on distance or landmarks and may rely on other factors to navigate the urban environment.

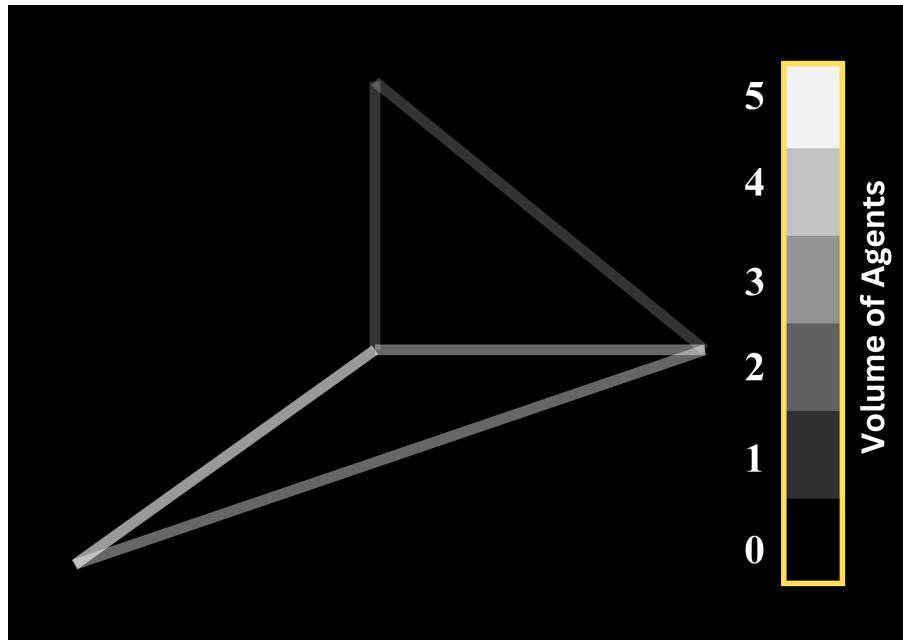
We recall the computations in Table 3.3 and the visualization in Figure 3.13 to understand the agent typologies better. In Table 3.3, we use Case 1 and Case 4 since they represent the most optimal clusters for the empirical data.

#### 3.6.4 Step 4. Agent-Based Model (ABM) Configurations and Simulations

##### Configuration 1: Random Non-Empirical Path Selection

Running the ABM in Configuration 1: All paths have the same probability of being chosen by agents. Consider the Configuration 1 result of 1 run per agent.

Agents	Path
$I_0$	$e_1$
$I_1$	$e_1$
$I_2$	$e_0, e_2$
$I_3$	$e_0, e_3$
$I_4$	$e_0, e_2$

**Table 3.6:** Example Configuration 1 Results**Figure 3.14:** Example Configuration 1 Volume per Path Segment.

The Figure 3.14 is the path network map with outlined paths against a black background. The white lines connecting various points on the map represent pathways within the map. The intensity of these connections is represented by varying shades of white, with brighter lines indicating more prominent or active pathways. In this map,  $e_0$  has the highest volume, while  $e_3$  has the least volume.

This **Configuration 1** serves as a baseline for comparing other configurations. After

running the ABM in Configuration 1, we now run the ABM in Configuration 2 and Configuration 3.

In **Configuration 2**, we consider the average route choice probabilities based on the survey data.

In **Configuration 3**, we consider the all the probabilities based on the computed clusters of the survey data.

### Configuration 2: Average-Based Empirical Path Selection

Individual	Distance	Landmark
1	1	0
2	0	0
3	1	1
4	0	1
5	1	1
Mean	0.6	0.6

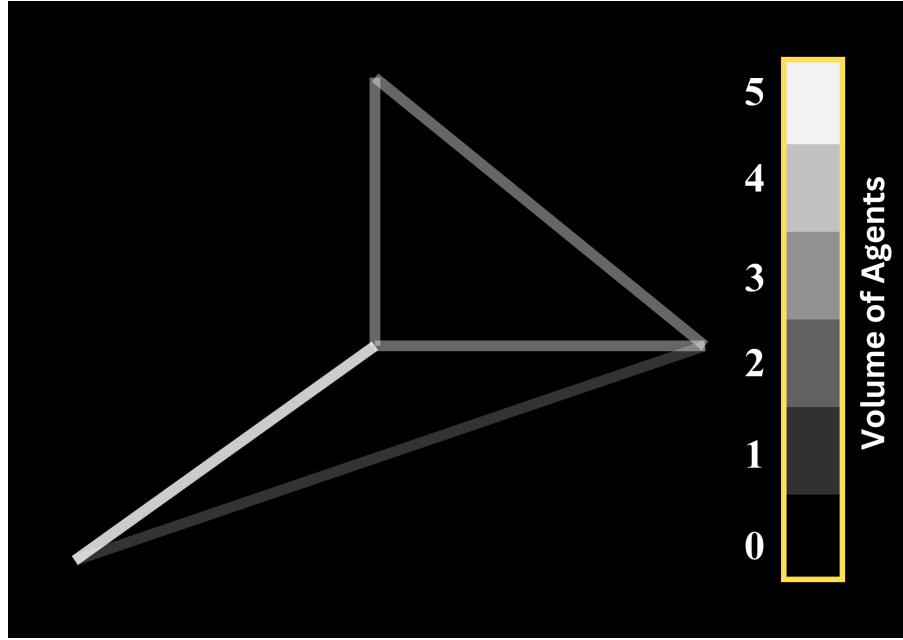
**Table 3.7:** Example Mean of Individual Route Choice Strategies

Now, we can simulate in configuration 2 so that all agents will consider Distance and Landmark 60% of the time.

Consider the Configuration 2 results below with run per agent:

Agents	Path
$I_0$	$e_1, e_2$
$I_1$	$e_1, e_3$
$I_2$	$e_1, e_3$
$I_3$	$e_1, e_2$
$I_4$	$e_0$

**Table 3.8:** Example Configuration 2 Results



**Figure 3.15:** Example Configuration 2 Volume per Path Segment.

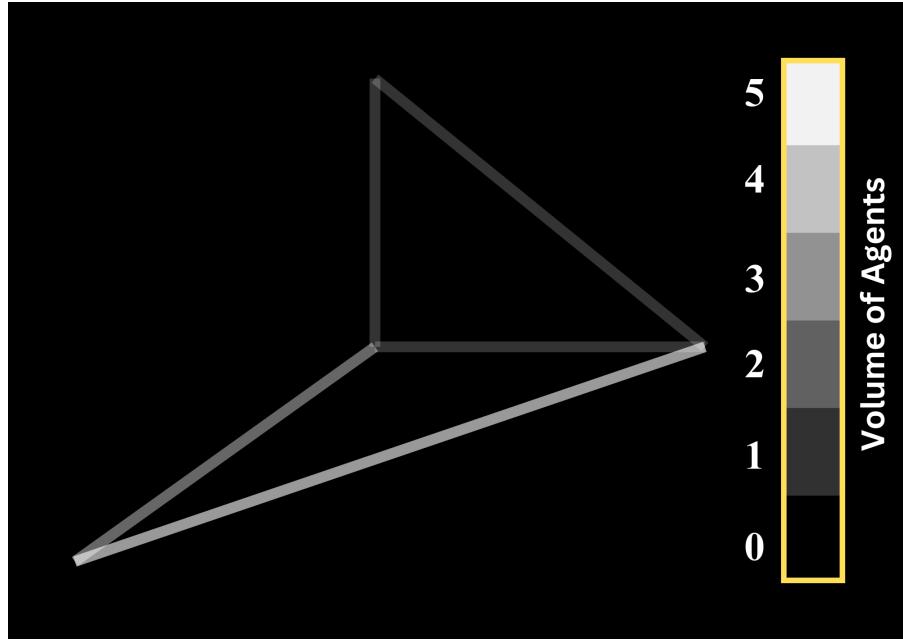
The Figure 3.15 is the path network map with outlined paths against a black background. The white lines connecting various points on the map represent pathways within the map. The intensity of these connections is represented by varying shades of white, with brighter lines indicating more prominent or active pathways. In this map,  $e_1$  has the highest volume, while  $e_0$  has the least volume.

### Configuration 3: Cluster-based Empirical Path Selection

Lastly, we consider the Configuration 3 results below with run per agent:

Agents	Path
$I_0$	$e_1, e_2$
$I_1$	$e_1, e_3$
$I_2$	$e_0$
$I_3$	$e_0$
$I_4$	$e_0$

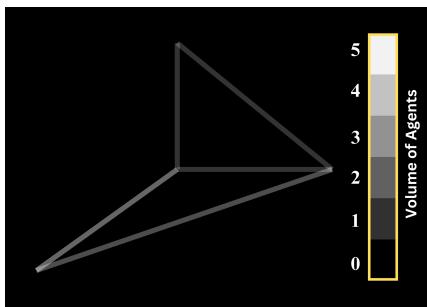
**Table 3.9:** Example Configuration 3 Results



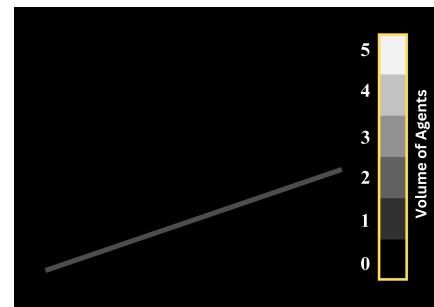
**Figure 3.16:** Example Configuration 3 Volume per Path Segment.

The Figure 3.16 is the path network map with outlined paths against a black background. The white lines connecting various points on the map represent pathways within the map. The intensity of these connections is represented by varying shades of white, with brighter lines indicating more prominent or active pathways. In this map,  $e_0$  has the highest volume, while both  $e_2$  and  $e_3$  has the least volume.

#### Volume of Heterogeneous Configuration Clusters



**Figure 3.17:** SSR Cluster 1



**Figure 3.18:** SSR Cluster 2

In Figures 3.17 and 3.18, there are visual differences with the paths taken by the agent, based on their priority when choosing a path. Cluster 1 shows preference to consider both Distant Barriers. Meanwhile, Cluster 2 shows the preference not consider both Distance and Landmark.

### 3.6.5 Step 5. Configurations Comparison

#### Differences Between Configuration 1 and Configuration 2

Now, we use the Wilcoxon Signed-Rank Test since the sample size is small and the data are not normally distributed.

Agent	Configuration 1	Configuration 2	Difference	Absolute Difference	Rank	Signed Rank
$e_1$	2	1	1	1	3	3
$e_2$	3	4	-1	1	3	-3
$e_3$	2	2	0	0	1	1
$e_4$	1	2	-1	1	3	-3

**Table 3.10:** Example Configuration 1 and 2 Wilcoxon Signed-Rank Test

For the Wilcoxon Signed-Rank Test, we have the following hypotheses:

1.  $H_0$ : Median difference is zero
2.  $H_1$ : Median difference is not zero where  $\alpha = 0.05$

Next, we get the test statistic ( $W$ ), defined as the sum of positive ranks ( $W+$ ), and the sum of negative ranks ( $W-$ ). If the null hypothesis ( $H_0$ ) is true, we expect to see a similar number of lower and higher ranks that are both positive and negative. If the research hypothesis ( $H_1$ ) is true, we expect to see higher and positive ranks.

For this SSR,  $W+ = 3 + 1 = 4$  and  $W- = 3 + 3 = 6$ . The sum of the positive and negative ranks will always equal to  $\frac{n(n+1)}{2} = \frac{4(4+1)}{2} = 10$  this is also equal to  $|W+| + |W-| = 4 + 6 = 10$ . The test statistic is  $W = 6$ .

Now, we determine if  $W$  supports the null or research hypothesis by determining the critical value of  $W$  such that if the observed value of  $W$  is less than or equal to the critical value, we reject  $H_0$  in favor of  $H_1$ . Meanwhile, if the observed value of  $W$  exceeds the critical value, we do not reject  $H_0$ .

Now, using the sample size ( $n = 4$ ) and the one-sided level of significance ( $\alpha = 0.05$ ), we determine the appropriate one-sided critical value. We use the following table of Critical Values of  $W$ :

Two-Sided Test $\alpha$	0.1	0.05	0.02	0.01
One-Sided Test $\alpha$	0.05	0.025	0.01	0.005
n				
4				
5	1			
6	2	1		
7	4	2	0	
8	6	4	2	0
9	8	6	3	2
10	11	8	5	3
11	14	11	7	5
12	17	14	10	7
13	21	17	13	10
14	26	21	16	13
15	30	25	20	16
16	36	30	24	19
17	41	35	28	23
18	47	40	33	28
19	54	46	38	32
20	60	52	43	37
21	68	59	49	43
22	75	66	56	49
23	83	73	62	55
24	92	81	69	61
25	101	90	77	68

**Table 3.11:** Critical Values of W in Wilcoxon Signed-Rank Test

In Table 3.11, the critical value  $W_c$  for a one-sided test with  $\alpha = 0.05$  and  $n = 4$  is undefined. Since the observed value of  $W = 6$  and the decision rule is to reject  $H_0$

if  $W \leq W_c$ . We do not reject the  $H_0$  no matter what since  $W$  will always be greater than  $W_c$  when  $n = 1, 2, 3, 4$ . This implies that Configuration 1 and Configuration 2 are not significantly different.

### Differences Between Configuration 1 and Configuration 3

Path	Configuration 1	Configuration 3	Difference	Absolute Difference	Rank	Signed Rank
$e_1$	2	3	-1	1	3	-3
$e_2$	3	2	1	1	3	3
$e_3$	2	1	1	1	3	3
$e_4$	1	1	0	0	1	1

**Table 3.12:** Example Wilcoxon Signed-Rank Test for Configuration 1 and 3

For the Wilcoxon Signed-Rank Test, we have the following hypotheses:

1.  $H_0$ : Median difference is zero
2.  $H_1$ : Median difference is not zero where  $\alpha = 0.05$

Next, we get the test statistic ( $W$ ), defined as the sum of positive ranks ( $W+$ ), and the sum of negative ranks ( $W-$ ). If the null hypothesis ( $H_0$ ) is true, we expect to see a similar number of lower and higher ranks that are both positive and negative. If the research hypothesis ( $H_1$ ) is true, we expect to see higher and positive ranks.

For this SSR,  $W+ = 3 + 3 + 1 = 7$  and  $W- = 3$ . The sum of the sum of the ranks will always equal to  $\frac{n(n+1)}{2} = \frac{4(4+1)}{2} = 10$  this is also equal to  $|(W+)| + |(W-)| = 7 + 3 = 10$ . The test statistic is  $W = 3$ .

Now, we determine if  $W$  supports the null or research hypothesis by determining the critical value of  $W$  such that if the observed value of  $W$  is less than or equal to the critical value, we reject  $H_0$  in favor of  $H_1$ . Meanwhile, if the observed value of  $W$  exceeds the critical value, we do not reject  $H_0$ .

Now, using the sample size ( $n = 4$ ) and the one-sided level of significance ( $\alpha = 0.05$ ), we determine the appropriate one-sided critical value. We use the table of Critical Values of  $W$  in Table 3.11.

In Table 3.11, the critical value  $W_c$  for a one-sided test with  $\alpha = 0.05$  and  $n = 4$  is undefined. Since the observed value of  $W = 3$  and the decision rule is to reject  $H_0$  if  $W \leq W_c$ . Thus, we do not reject the  $H_0$  no matter what since  $W$  will always be greater than  $W_c$  when  $n = 1,2,3,4$ . This implies that Configuration 1 and Configuration 3 are not significantly different.

#### Differences Between Configuration 2 and Configuration 3

Now, we use the Wilcoxon Signed-Rank Test since the sample size is small and the data are not normally distributed.

Path	Configuration 2	Configuration 3	Difference	Absolute Difference	Rank	Signed Rank
$e_1$	1	3	-2	2	3	-3
$e_2$	4	2	2	2	3	3
$e_3$	2	1	1	1	2	2
$e_4$	2	1	1	1	2	2

**Table 3.13:** Example Wilcoxon Signed-Rank Test

For the Wilcoxon Signed-Rank Test, we have the following hypotheses:

1.  $H_0$ : Median difference is zero
2.  $H_1$ : Median difference is not zero where  $\alpha = 0.05$

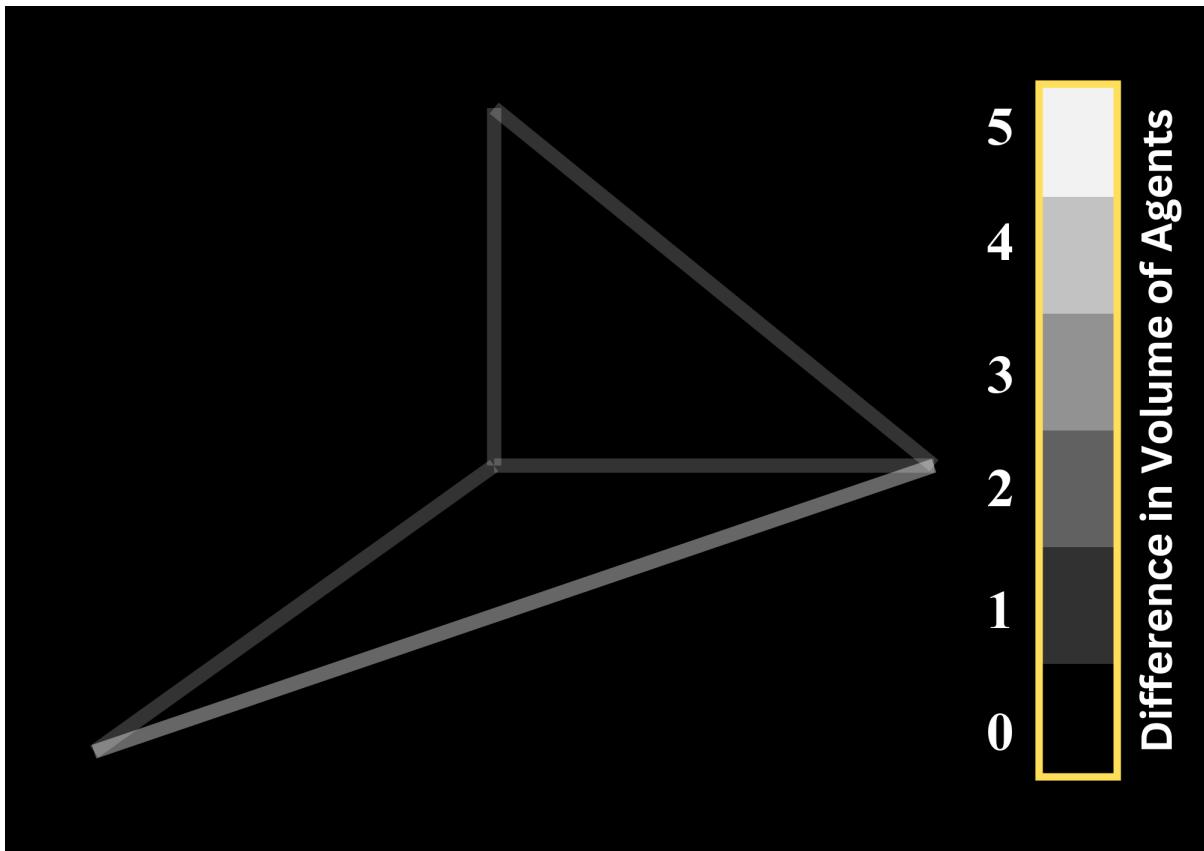
Next, we get the test statistic ( $W$ ), defined as the sum of positive ranks ( $W+$ ), and the sum of negative ranks ( $W-$ ). If the null hypothesis ( $H_0$ ) is true, we expect to see a similar number of lower and higher ranks that are both positive and negative. If the research hypothesis ( $H_1$ ) is true, we expect to see higher and positive ranks.

For this SSR,  $W+ = 3 + 2 + 2 = 7$  and  $W- = -3$ . The sum of the sum of the ranks will always equal to  $\frac{n(n+1)}{2} = \frac{4(4+1)}{2} = 10$  this is also equal to  $|(W+)| + |(W-)| = 7 + 3 = 10$ . The test statistic is  $W = 3$ .

Now, we determine if  $W$  supports the null or research hypothesis by determining the critical value of  $W$  such that if the observed value of  $W$  is less than or equal to the critical value, we reject  $H_0$  in favor of  $H_1$ . Meanwhile, if the observed value of  $W$  exceeds the critical value, we do not reject  $H_0$ .

Now, using the sample size ( $n = 4$ ) and the one-sided level of significance ( $\alpha = 0.05$ ), we determine the appropriate one-sided critical value. We use the following table of Critical Values of  $W$  based on Table 3.11.

The critical value  $W_c$  for a one-sided test with  $\alpha = 0.05$  and  $n = 4$  is undefined. Since the observed value of  $W = 3$  and the decision rule is to reject  $H_0$  if  $W \leq W_c$ . Thus, we do not reject the  $H_0$  no matter what since  $W$  will always be greater than  $W_c$  when  $n = 1, 2, 3, 4$ . This implies that the Configuration 2 and Configuration 3 are not significantly different.



**Figure 3.19:** Example Volume Difference of Configuration 2 and Configuration 3

In Figure 3.19 and the Wilcoxon Test, the brighter the color, the higher the difference in volume. The white shades represent the volume of difference. The paths taken by the agents in configuration 2 does not differ from the paths taken by agents in configuration 3 agents. This difference means that the use of empirical data cannot capture the complexity of pedestrian route choice in the University ( $U$ ) when using a population size of 5 and two variables namely Distance and Landmark.

### 3.6.6 SSR Conclusion

In conclusion, the Small Scale Replication (SSR) study successfully gathered valuable insights into pedestrian route choice behavior using a questionnaire with five responses and the evaluation of two Route Choice Strategies. The demographic information revealed a 2:2:1 ratio (female, male, prefer not to say) in the representation of sex at birth, with participants predominantly falling in the 18–27 age group. The majority of participants were found to be affiliated in the case-study area, with a significant percentage either working or studying there.

The visualization of the overall importance of each urban element using probabilities presented a comprehensive understanding of pedestrian route choice factors. The application of the K-means algorithm revealed a well-defined clustering structure, and the subsequent examination of the chosen partitions provided unique insights into different behavioral characteristics.

The implementation of the Agent-Based Model (ABM) in Configuration 1 (Random Non-Empirical Path Selection), Configuration 2 (Average-Based Empirical Path Selection), and Configuration 3 (Cluster-Based Empirical Path Selection) showcased the baseline, uniform, and diverse scenarios. The comparison of volumes between these configurations illustrated no distinct patterns, demonstrating the non-efficacy of the ABM in capturing variations in pedestrian behavior in University ( $U$ ), considering that there are only five gathered survey participants and only two variables considered as variables and route choice strategy.

The volume analysis of Heterogeneous Configuration clusters highlighted preferences for specific Route Choice Strategies within each cluster. Notably, Cluster 1 shown a

preference to consider both Distance and Landmark, while Cluster 2 shown a preference to not consider both Distance and Landmark.

Finally, the comparison between Configuration 1, Configuration 2 and Configuration 3 revealed no significant differences in the paths taken by agents, emphasizing the inability of the Small Scale Replication Agent-Based Model to capture the complexity of pedestrian route choice in a more realistic manner.

In summary, the Small Scale Replication, through a systematic analysis of questionnaire data, demographic information, clustering, and ABM configurations, provides a nuanced understanding of pedestrian route choice behavior. However, the SSR study does not contribute valuable insights in using empirical data for the ABM when considering University (U) with five individuals and two variables for pedestrian route choice behavior.

# Chapter 4

## Methodology

### 4.1 Overview

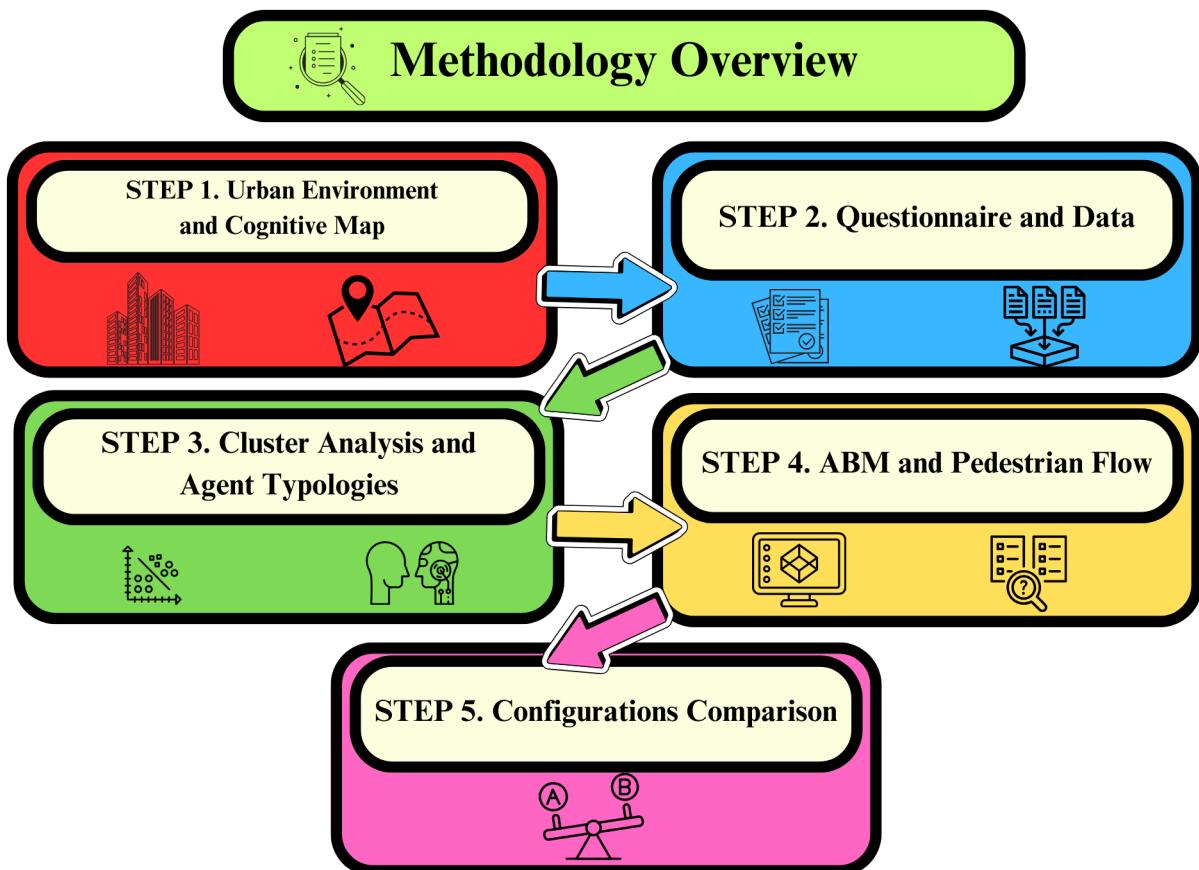


Figure 4.1: Methodology Overview Illustration

Now that we established the important **concepts**, we now use these concepts to perform the **empirical ABM** for **pedestrian behavior**. For this methodology, we will do five main steps:

Step 1. Define the urban environment and construct the cognitive map.

Step 2. Design the questionnaire and collect the data.

Step 3. Perform cluster analysis and derive the agent typologies.

Step 4. Run the ABM

Step 5. Compare the pedestrian flows using the different configurations.

For step 1, the Urban Environment (UE) consists of the path network, buildings, and natural and artificial elements of the University ( $U$ ). For a recall, a **Cognitive Map** is a representation of the urban environment that includes meaningful Urban Elements (UE) such as **Landmarks**, and **Barriers**, as well as Path Attributes (PA) such as **Distance**. The UE and PA are assigned scores or values based on their salience or importance for pedestrian navigation. We will detail the urban environment and cognitive map as we go through the methodology.

For step 2, the questionnaire aims to investigate how pedestrians use UE and PA to formulate their routes in the University ( $U$ ). The questionnaire consists of five sections, each containing different types of questions such as navigational tasks 1 to 3, barrier preferences, and participant demographic information. Next, we gather the participants through the local population of University ( $U$ ).

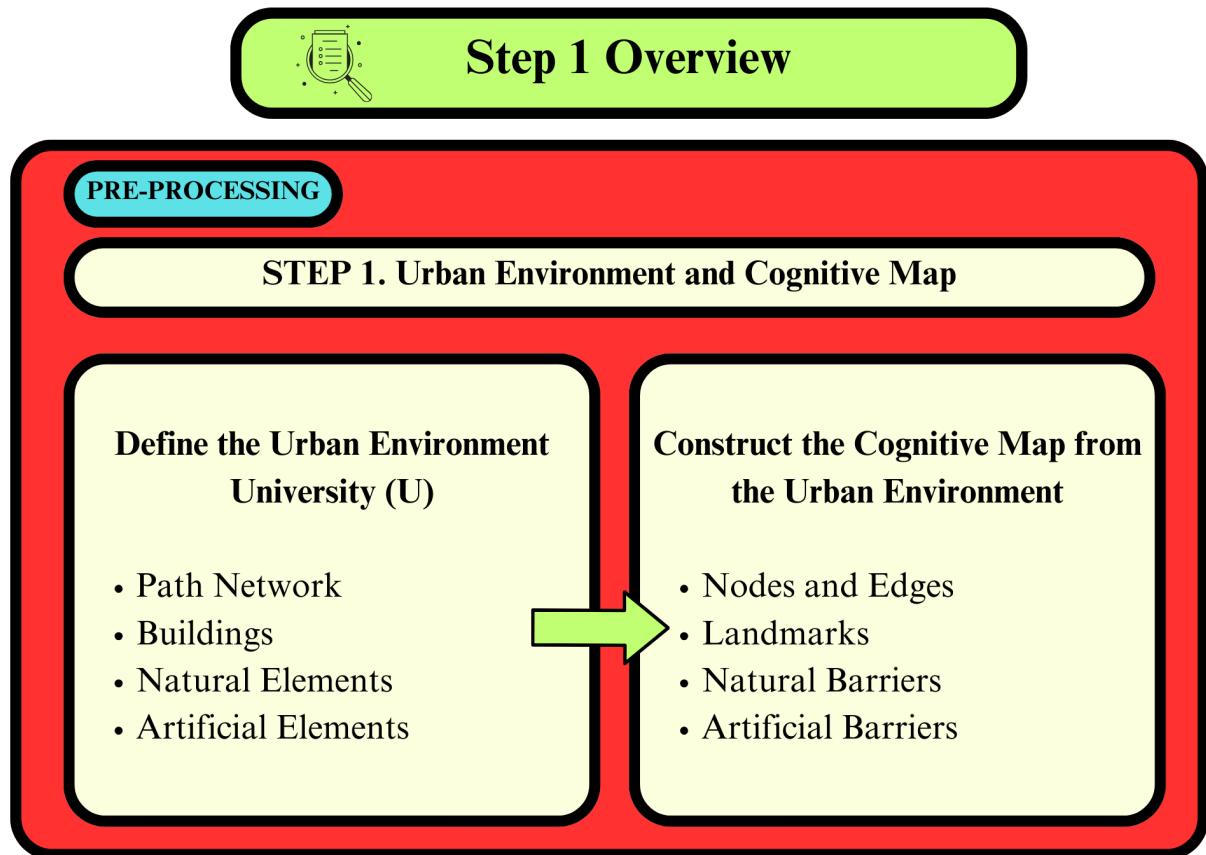
For step 3, after gathering the data from an online tool, we now perform cluster analysis. As a recall, cluster analysis is a statistical technique that identifies groups of individuals who share similar route choice strategies based on their responses on the questionnaire. In cluster analysis, we use k-means algorithm, which requires specifying the desired number of clusters. Now, The optimal number of clusters is determined by comparing the silhouette coefficient and the variance ratio criterion of different partitions. The optimized cluster analysis results in clusters characterized by different probabilities of using UE and PA. These clusters are used to define the agent typologies in the ABM, i.e. the behavioral parameters that regulate the agent's reliance on UE and PA.

For step 4, we show the theory behind the implementation of the ABM. The ABM

simulates the movement of pedestrian agents across the urban environment. The main idea of the ABM is to generate trips between different pairs of locations (OD pairs) and store the number of times a path segment is crossed by the agent, thus computing pedestrian volumes per segment. For this ABM we configure it into three different configurations, which are the Configuration 1: Random Non-Empirical Path Selection, Configuration 2: Average-Based Empirical Path Selection, and Configuration 3: Cluster-Based Empirical Path Selection. In Configuration 1, agent behavior is random and not informed by empirical data. In Configuration 2, agents behavior is based on average values of the variables obtained from the questionnaire. Meanwhile, in Configuration 3, agent behavior is based on the cluster analysis and the agent typologies.

After that, we compare the pedestrian flows resulting from the different configurations through Python codes, which is much easier to understand using Jupiter Notebook.

## 4.2 Step 1: Urban Environment and Cognitive Map



**Figure 4.2:** Methodology Step 1 Overview Illustration

For this methodology, we define the Urban Environment University ( $U$ ) as the University of the Philippines Baguio. To retrieve this map, we use a physical or online map.



**Figure 4.3:** University of the Philippines Baguio (*U*) as Urban Environment shown in Enhanced Contrast Map.

Next, we create a **Cognitive Map** from the Urban Environment by retrieving the **Urban Environment Pedestrian Data (UEPD)** and **Urban Element (UE)** directly from the base map. When we say UEPD, it consists the following:

- **Pedestrian Intersection Data (PID)**, which are the nodes. PID is the intersection of pedestrian paths to another pedestrian path.
- **Pedestrian Paths Data (PPD)**, which are the edges. This is the line or path from one intersection to another.

For better understanding of the nodes and edges involved in UEPD, recall **Section 3.1.2: Nodes and Edges in Graph Theory**.

For the **UEPD** retrieval, there are two methods: Manual Tracing or automatically through the online website map data, such as OpenStreetMap website.

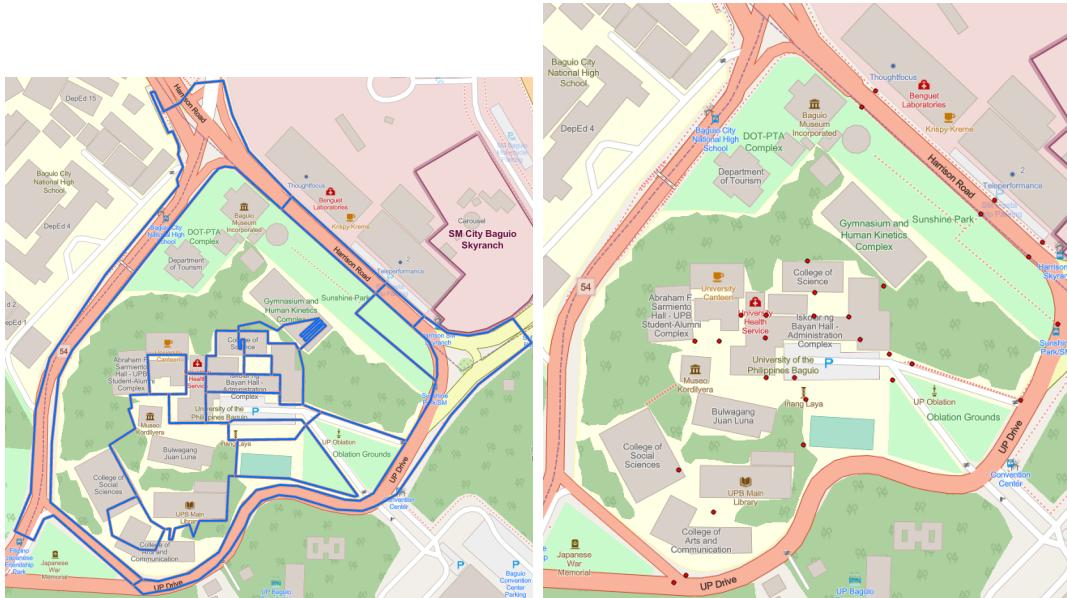
### UEPD Manual Tracing

Manually, we trace the OpenStreetMap's pedestrian paths shown in the **University** (*U*) map. We only trace the 500-meter distance from the map center. This traced paths shall consist of nodes and edges.

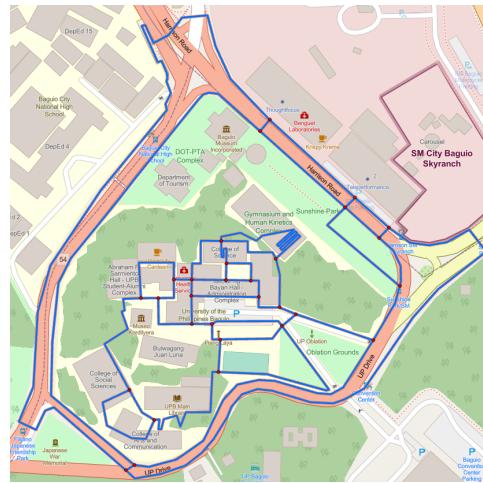
### UEPD Automatic Extraction

Automatically, we use the OpenStreetMap's feature to extract nodes (**PID**) and edges (**PPD**) of the map data. After that, we delete the extracted nodes and edges beyond 500 m distance from the center through a GIS manipulation software.

Next, we show the **UEPD** layered on top of the **University** (*U*) map through Figures 4.5 and 4.4.

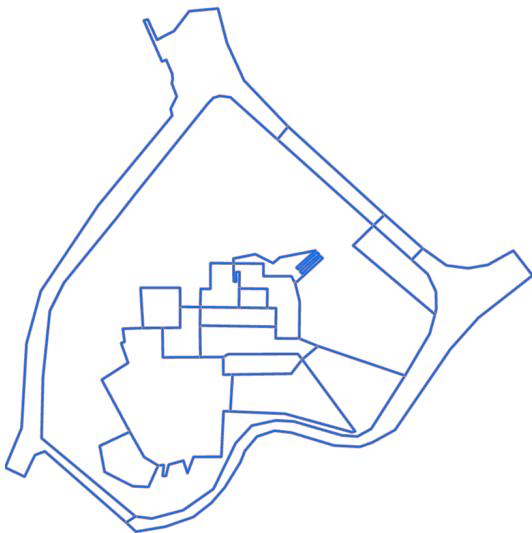


**Figure 4.4:** UEPD PPD (edges) shown in blue, layered on top of the University (*U*) Map. **Figure 4.5:** UEPD PID (nodes) shown in red, layered on top of the University (*U*) Map.

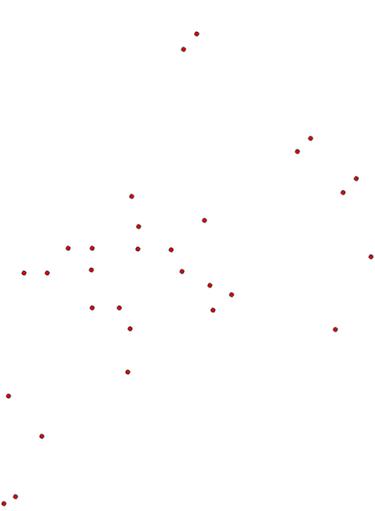


**Figure 4.6:** Combined Retrieved Map **PPD** shown in blue, **PID** shown in red, layered on top of the base map.

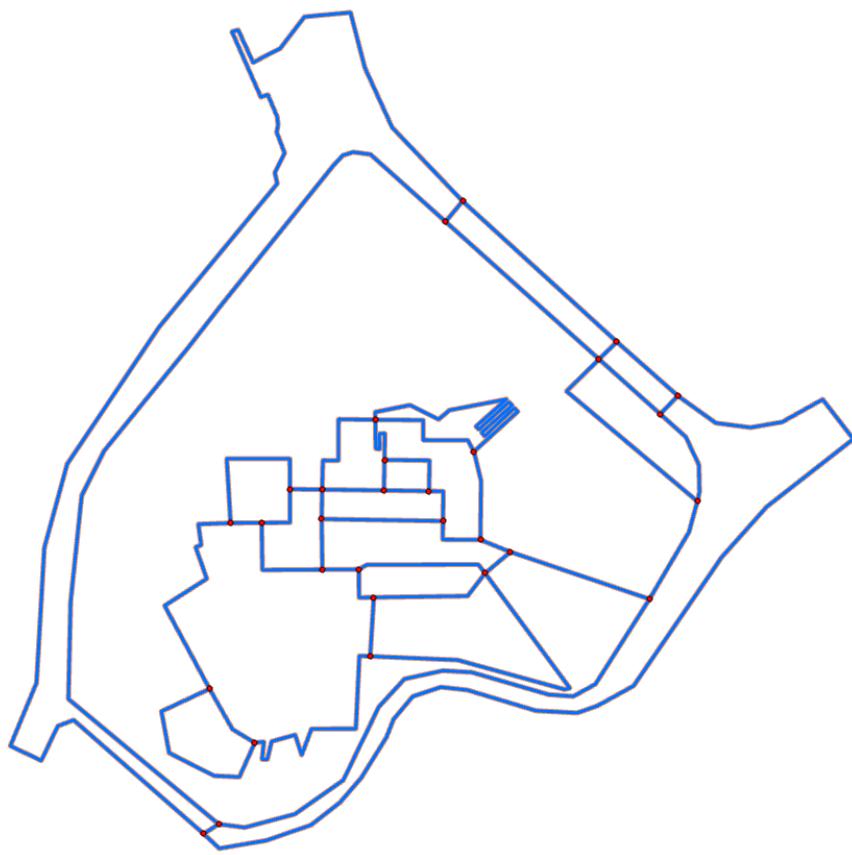
After that, we can now remove the base map of **University** ( $U$ ) from the **UEPD**. Now, we show the retrieved **UEPD** through Figures 4.8, 4.7, and 4.9.



**Figure 4.7:** Retrieved PPD (edges) shown in blue.

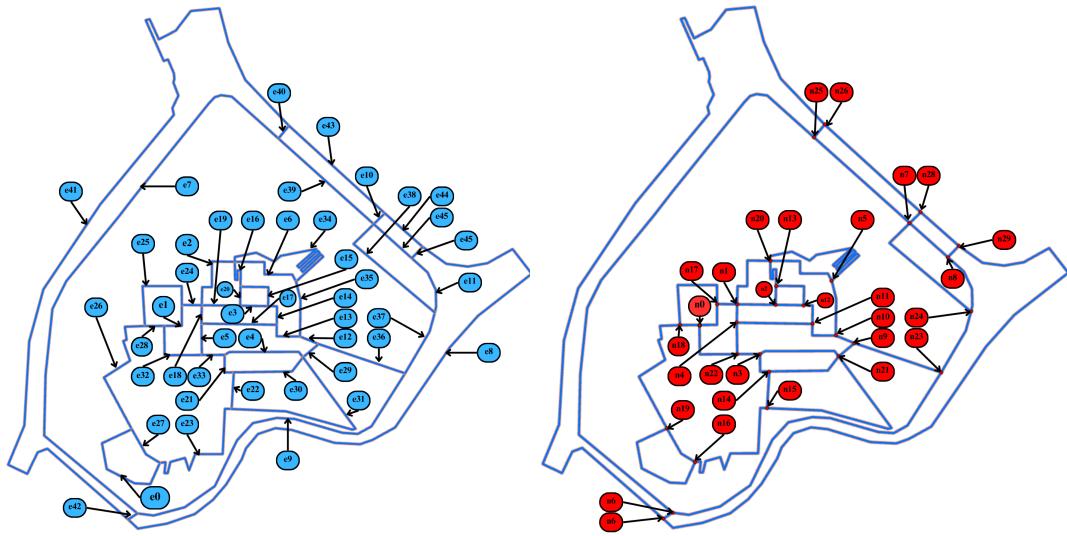


**Figure 4.8:** Retrieved PID (nodes) shown in red.



**Figure 4.9:** Combined Retrieved UEPD where PPD is shown in blue, PID is shown in red.

Now, we individually identify the nodes and edges.



**Figure 4.10:** Identified edges shown in blue.  
**Figure 4.11:** Identified nodes shown in red.

The path network shall either have urban elements and path attributes. These urban elements and path attributes affect the Route Choice Behavior of the pedestrians. In addition, the selection of the route is based on which of the routes will answer the aim and objectives of the study. To do this, we consider the routes that exhibit the Urban Elements and Path Attributes through different Route Choices.

#### Example 11: Different Route Choices

##### Route Choice Through Urban Elements (RCTUE)

1. **L: Landmarks**, which means using distant orienting landmarks to guide the direction of the route.
2. **B: Barriers**, which means using barriers as sub-goals along the route.

##### Route Choice Through Path Attribute (RCTRA)

1. **D: Distance**, which means considering distance when traversing the route.

Now, we show the inclusion of both the Landmarks and Barriers in the UPB path network.

First, we identify each landmark.



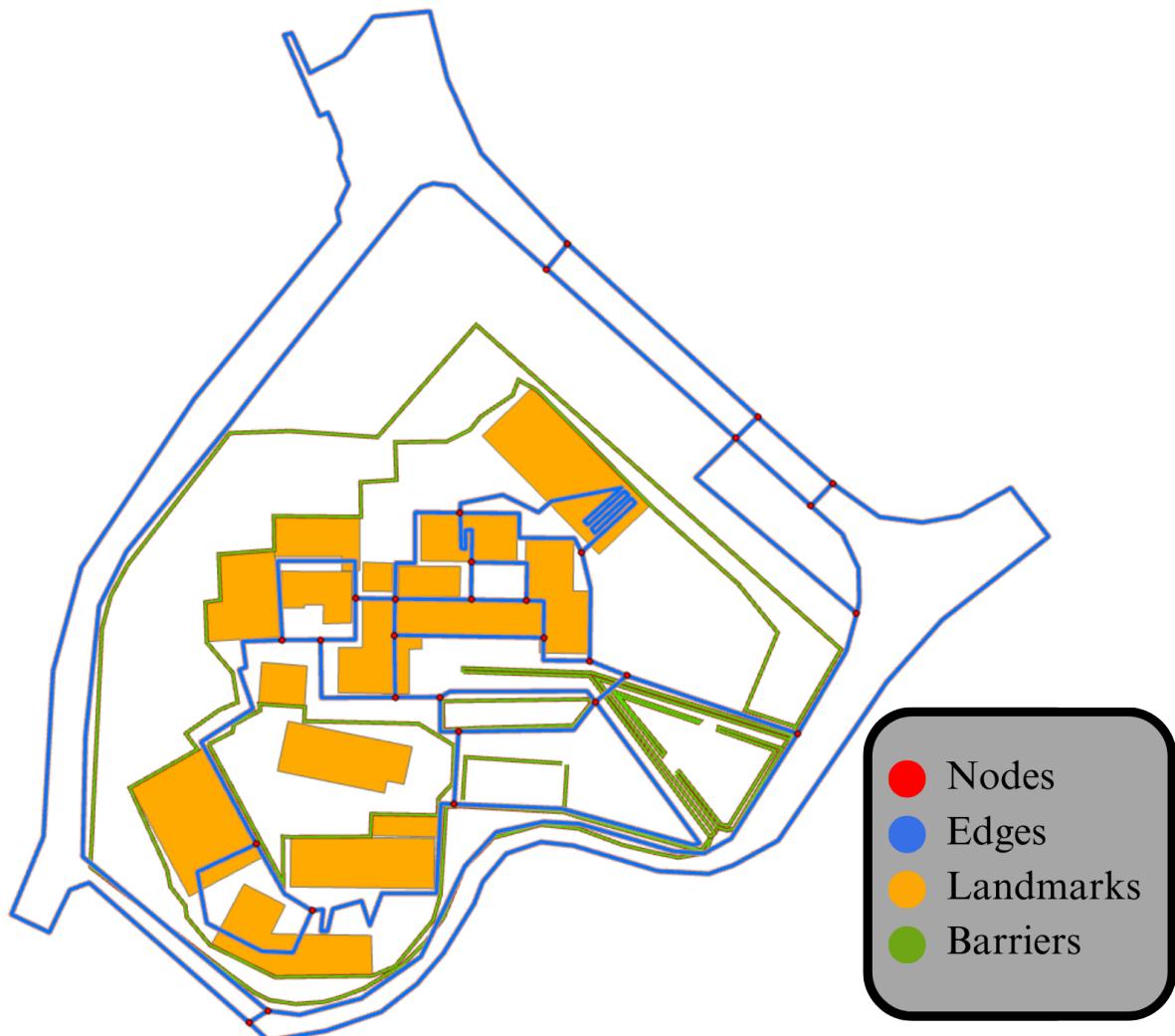
**Figure 4.12:** Landmarks shown in orange in the UPB path network.

Next, we show all the barriers inside UPB path network:



**Figure 4.13:** Barriers shown green in the UPB path network.

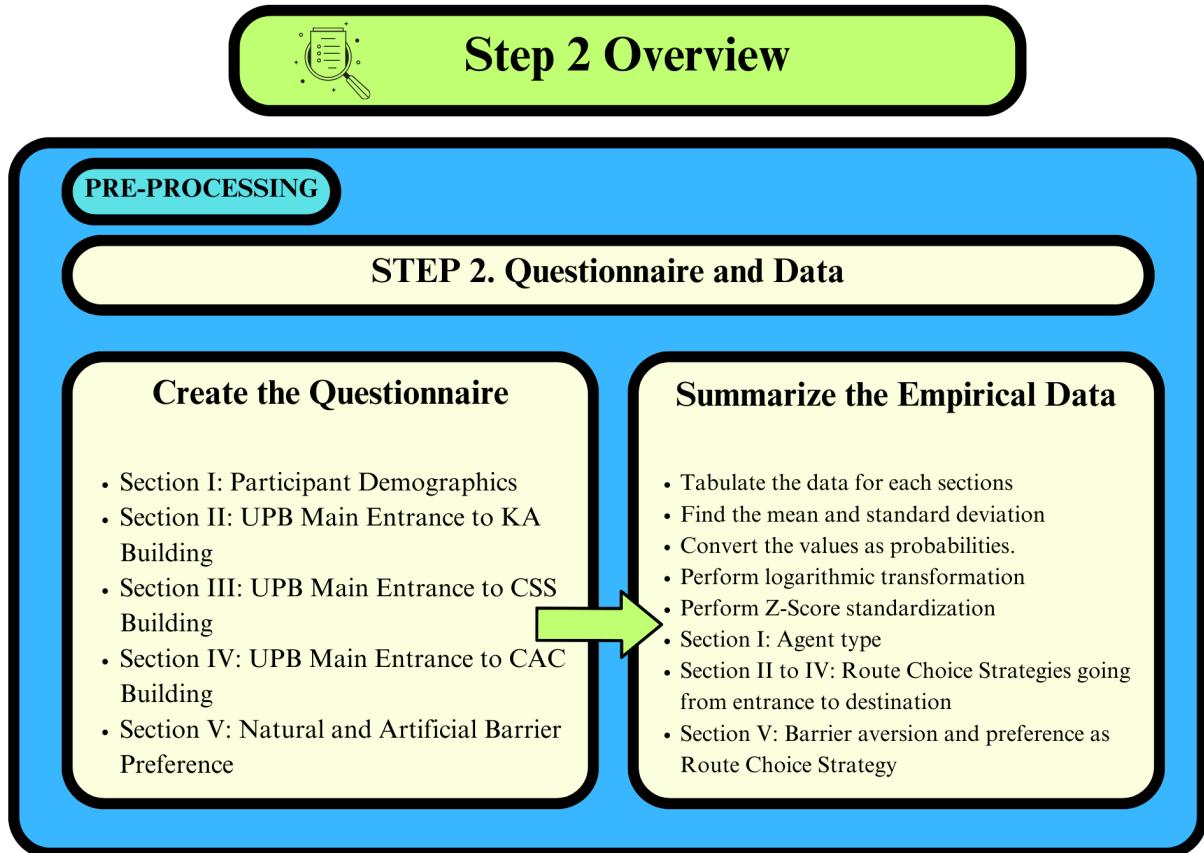
Now that we have the path network and Urban Elements, the Cognitive Map of UPB is now completed. The complete Cognitive Map is shown in Figure 4.14 below:



**Figure 4.14:** Cognitive Map of the University of the Philippines Baguio (UPB).

Now that we already defined the Urban Environment and the Cognitive Map, we can now proceed to the next step of the methodology.

### 4.3 Step 2: Questionnaire and Data



**Figure 4.15:** Methodology Step 2 Overview Illustration

To design the questionnaire, now we need to make it align with the research questions and objectives. For this methodology, the aim is to present an empirically-based ABM for the simulation of pedestrian movement in urban spaces that accounts for behavioral heterogeneity in pedestrian route choice strategies. Meanwhile, the research questions are as follows:

- a What is the diversity in route choice strategies as concerns the usage of Distance, Landmarks, and Barriers?

- b To what extent does the variation in the agents' cluster-based empirical path selection — i.e. a model including agent typologies vs a model with average-based empirical path selection — generate different movement patterns across the path network?

Next, we assign the case study area and Urban Elements to include. For the case study area, we already identified it in **Step 1: Urban Environment and Cognitive Map**, which is University of the Philippines Baguio (*U*). Following the case study area, we identified the two types of Urban Elements that shape the Cognitive Map of pedestrians: Landmarks (*L*) and Barriers (*B*). Meanwhile, the Path Attribute is Distance (*D*).

We can now advance with the design of the questionnaire. For this, we have the questionnaire structure and content. The general idea of the questionnaire is to collect information that will satisfy the research questions and objectives. So, we create five sections for the questionnaire, which will collect five different information from the participants:

#### Example 12: Study Questionnaire Sections

- I. Participant Demographic Information
- II. Study Route 0: Navigational Task 1 (UPB Main Entrance to Kolehiyo ng Agham (KA) Building)
- III. Study Route 1: Navigational Task 2 (UPB Main Entrance to College of Social Sciences (CSS) Building)
- IV. Study Route 2: Navigational Task 3 (UPB Main Entrance to College of Arts and Communication (CAC) Building)
- V. Barrier Preferences

In Section I, demographic information such as age, gender, belonging to a certain category (i. e. student), time spent in the case study area was collected.

Sections II, III, and IV, the core of the questionnaire, consists of three navigational questions. For these three navigational questions, we question the participants if they

consider Distance, Landmarks, and Barriers when navigating from the UPB Main Entrance to the KA Building, CSS Building, and CAC Building, respectively. For this methodology, we refer to the Cognitive Map in Figure 4.14 to use the OD pairs (refer to Figure 4.11 for node ID) (n23, n13), (n23, n16), and (n23, n19) used in Section II, III, and IV, respectively.

Finally, Section V, aims to collect preferences for certain route characteristic, specifically when dealing with Barriers. The online tool automatically records the responses of the individuals in a generated table.

#### 4.3.1 Section I: Participant Demographic Information

For Section I, We ask three questions, which are:

Example 13: Section I: Participant Demographic Information Questions

1. What is your gender?
2. How old are you?
3. Which of the following categories apply to you?
  - a. I work or used to work in the UPB ( $U$ ).
  - b. I study or used to study in UPB ( $U$ ).

#### 4.3.2 Section II: Study Route 0 Navigational Task 1 (UPB Main Entrance to Kolehiyo ng Agham (KA) Building)

For Section II, the participants are asked to traverse a route between UPB Main Entrance and the KA Building by considering or not considering Distance, Landmarks, and/or Barriers.

Section II has four main questions, which are:

Example 14: Section II: Study Route 0 Navigational Task 1 (UPB Main Entrance to KA Building)

1. Are you familiar with the following two locations, the UPB Main Entrance and KA Building?
2. Do you consider Distance when going from UPB Main Entrance and KA Building?
3. Do you consider Landmarks when going from UPB Main Entrance and KA Building?
4. Do you consider Barriers when going from UPB Main Entrance and KA Building?

For Section II Question 1 to 4, participants can answer the question by yes or no.

Example 15: Section II Q1: Are you familiar with the following two locations, the UPB Main Entrance and KA Building

- Yes
- No

Example 16: Section II Q4: Do you consider **Distance** when going from UPB Main Entrance and KA Building?

- Yes
- No

Example 17: Section II Q4: Do you consider **Landmarks** when going from UPB Main Entrance and KA Building?

- Yes
- No

Example 18: Section II Q4: Do you consider **Barriers** when going from UPB Main Entrance and KA Building?

- Yes
- No

#### **4.3.3 Section III: Study Route 1 Navigational Task 2 (UPB Main Entrance to College of Social Science (CSS) Building)**

For Section III, the participants are asked to traverse a route between UPB Main Entrance and the CSS Building by considering or not considering Distance, Landmarks, and/or Barriers. The procedure is the same as Section II. Section III has four main questions, which are:

Example 19: Section III: Study Route 1 Navigational Task 2 (UPB Main Entrance to CSS Building)

1. Are you familiar with the following two locations, the UPB Main Entrance and CSS Building?
2. Do you consider Distance when going from UPB Main Entrance and CSS Building?
3. Do you consider Landmarks when going from UPB Main Entrance and CSS Building?
4. Do you consider Barriers when going from UPB Main Entrance and CSS Building?

For Section III Question 1–4, participants can answer the question by yes or no.

Example 20: Section III Q1: Are you familiar with the following two locations, UPB Main Entrance and CSS Building?

- Yes
- No

Example 21: Section III Q2: Do you consider **Distance** when going from UPB Main Entrance and CSS Building?

- Yes
- No

Example 22: Section III Q3: Do you consider **Landmarks** when going from UPB Main Entrance and CSS Building?

- Yes
- No

Example 23: Section III Q4: Do you consider **Barriers** when going from UPB Main Entrance and CSS Building?

- Yes
- No

#### 4.3.4 Section IV: Study Route 2 Navigational Task 3 (UPB Main Entrance to College of Arts and Communication (CAC) Building)

For Section IV, the participants are asked to traverse a route between UPB Main Entrance and the CAC Building by considering or not considering Distance, Landmarks, and/or Barriers. The procedure is the same as Section II and III. Section IV has four main questions, which are:

Example 24: Section IV: Study Route 2 Navigational Task 3 (UPB Main Entrance to CAC Building)

1. Are you familiar with the following two locations, the UPB Main Entrance and CAC Building?
2. Do you consider Distance when going from UPB Main Entrance and CAC Building?
3. Do you consider Landmarks when going from UPB Main Entrance and CAC Building?
4. Do you consider Barriers when going from UPB Main Entrance and CAC Building?

For Section IV Question 1–4, participants can answer the question by yes or no.

Example 25: Section IV Q1: Are you familiar with the following two locations, UPB Main Entrance and CAC Building?

- Yes
- No

Example 26: Section IV Q2: Do you consider **Distance** when going from UPB Main Entrance and CAC Building?

- Yes
- No

For Section IV Question 3, participants should refer to the video and choose their appropriate answer from there.

Example 27: Section IV Q3: Do you consider **Landmarks** when going from UPB Main Entrance and CAC Building?

- Yes
- No

Example 28: Section IV Q4: Do you consider **Barriers** when going from UPB Main Entrance and CAC Building?

- Yes
- No

#### 4.3.5 Section V: Preferences for Certain Route Characteristics

For Section V, we ask four main questions to determine the preferences of pedestrians when it involves the consideration of Barriers on their route choice strategy. The Section V main questions are:

Example 29: Section V: Preferences for Certain Route Characteristics

1. How important are the following properties of a route if you have to reach a certain place by foot?

For Section V Question 4 there are seven sub questions, where they choose an appropriate response from: **Not Important**, **Slightly Important**, **Important**, **Fairly Important**, and **Very Important**. The main question is “How important are the following properties of a route if you have to reach a certain place by foot? ” The sub questions are:

Example 30: Section V Q4

How important are the following properties of a route if you have to reach a certain place by foot?

1. The route traverses or extends along green areas (e.g. parks, gardens, forests)

2. The route crosses safe areas (e.g. good lighting, presence of other people, the area is known)
3. The route crosses interesting or useful landmarks

We determined the sample size by adopting the approach advanced by Formann [37], according to which, for cluster analysis, the sample should include at least  $2^u$  subjects or observations, where  $u$  represents the number of variables. Given that  $u$  is equal to 3 variables in this case study (Distance, Landmark, and Barrier), the minimum sample size was 8.

### 4.3.6 Summarized Empirical Data

#### Socio-Demographic Information

	<b>Demographics</b>	<b>Frequency</b>	<b>Percentage</b>
Gender	Female	28	41.2%
	Male	35	51.5%
	Non-binary	1	1.5%
	Prefer not to declare	4	5.9%
Age	< 18	0	0.0%
	18–25	67	98.5%
	26–33	1	1.5%
	34–41	0	0.0%
	42–49	0	0.0%
	50–57	0	0.0%
	58–65	0	0.0%
	66–73	0	0.0%
	> 74	0	0.0%
Statistical Data	Mean Age	21.235	
	Standard Deviation Age	1.594	
	Mean Response Duration	8.12 minutes	
Relationship with UPB	Study/Studied here	68	100.0%

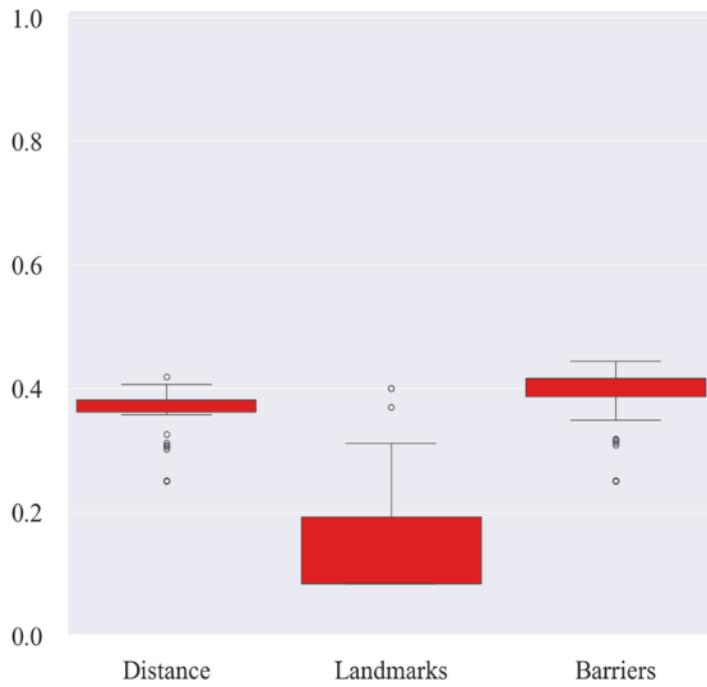
**Table 4.1:** The Socio-Demographic Profile of the Participants (N = 68).

The table 4.1 presents the demographic distribution of participants in the case study. The sample consisted of 68 participants, with a slightly higher proportion of males (51.5%) compared to females (41.2%). A minimal fraction of the participants identified as non-binary (1.5%). Additionally, 5.9% of the participants preferred not to declare their gender. The participants' age distribution is uniform, with the largest proportion falling within the 18–25 age group (98.5%). Minimal representation is observed in the 26–33 age group (1.5%), while other age groups have no representation in this case study.

The mean age of the participants was 21.235 years old, with a standard deviation of 1.594, showing a homogeneous age bracket where participants are in their twenties.

We further investigated the links or relationships participants had with the case-study area. All participants study or studied in the UPB (100.0%). This indicates a strong association between the participants and the case-study area, with all participants having a direct connection to the area.

### Visualizing Overall Importance of Each Variable (Probabilities)

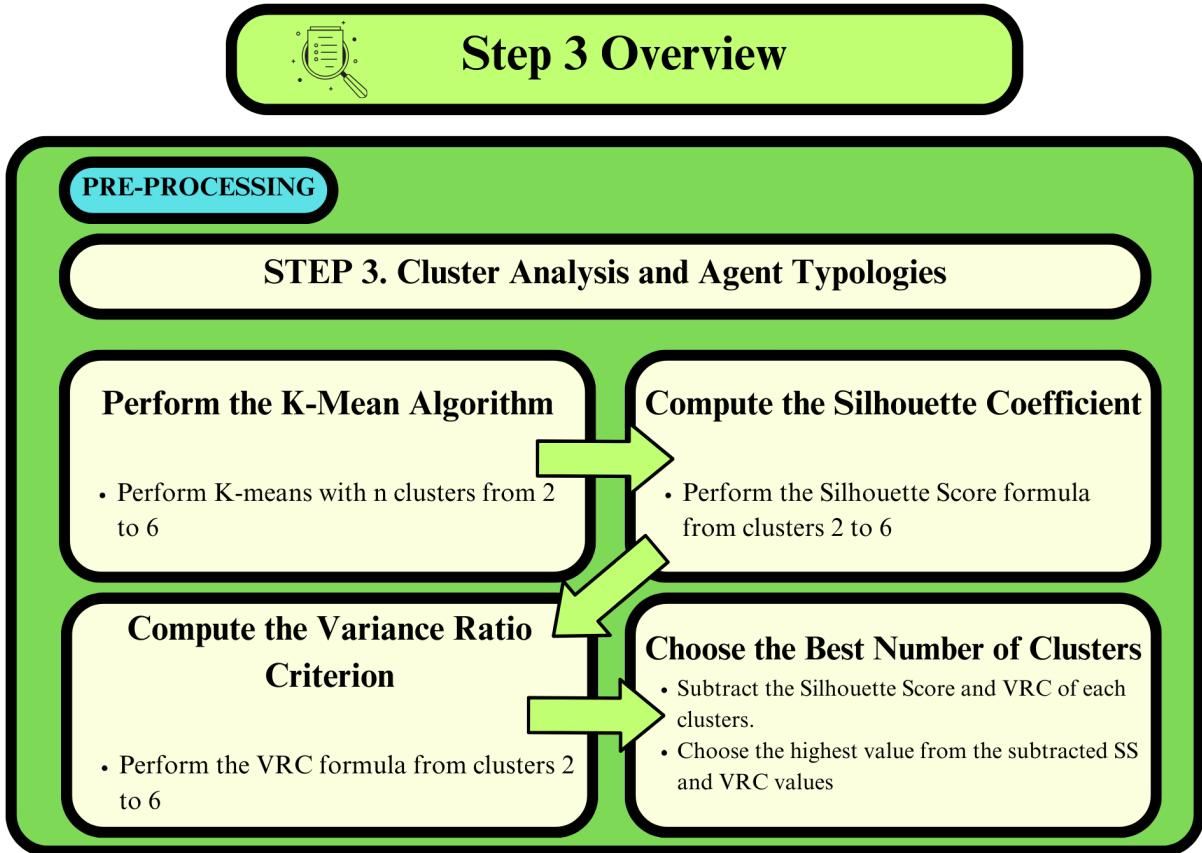


**Figure 4.16:** Variable Importance in Pedestrian Route Choice. Box plot of the variables extracted from the responses to the questionnaire for the entire study sample. The boxes colored in red represent the distribution of the variables describing the probability of manifesting a certain behavior. Variables colored in red were included in the cluster analysis.

The Figure 4.16 provides a comprehensive visualization of the overall importance of variables in influencing pedestrian route choice. The data is presented in probabilities

to offer a quantitative insight into the significance each variable. The figure presents three distinct variables: namely Distance, Landmarks, and Barriers. Each category is represented with a bar indicating its probability score along with whiskers to depict the range of variance. The figure shows variable Barriers has the highest probability with a score between 0.4 to 0.5. This indicates that pedestrians are highly influenced by barriers when selecting their routes. Following the Barriers, the variable Distance has a probability score between 0.2 to 0.4, suggesting that distance is a moderate influence on pedestrian route choice. Lastly, the variable Landmarks have the lowest probability score between 0.1 to 0.2, indicating that landmarks have the least influence on pedestrian route choice. We observe that there is lesser variability in probabilities for distance, and barriers. On the other hand, landmarks exhibit higher variability in probabilities. This suggests that the influence of landmarks on pedestrian route choice may vary significantly among individuals. We also observe that there are many outliers in distance, and barriers. Meanwhile, landmarks has little outliers.

## 4.4 Step 3: Cluster Analysis and Agent Typologies



**Figure 4.17:** Methodology Step 3 Overview Illustration

Now that we have the data, we now perform a cluster analysis. As a recall, cluster analysis is a method to group similar individuals based on their Route Choice Strategies (RCS). The general steps of Cluster Analysis are as follows:

**Example 31: General Procedures of Cluster Analysis**

Procedure 1: Use the responses from the questionnaire where the individuals indicated their responses when walking between different locations in the University ( $U$ ), and what factors they would consider when choosing a route. Calculate for the probabilities.

Procedure 2: Extract three variables from the questionnaire that represented the probability of using different urban elements (landmarks, and barriers) and path attributes (distance) when planning a route.

Procedure 3: Apply logarithmic transformation and a standardization to the variables to reduce the skewness and make them comparable.

Procedure 4: Run a k-means algorithm with different numbers of desired clusters to find the optimal partition of the data. The k-means algorithm assigns each individual to the closest cluster centroid, based on a measure of distance, and updates the centroids until they stabilize.

Procedure 5: Evaluate the quality of the partitions by using the silhouette coefficient and the variance ratio criterion, which measure the intra-cluster similarity and the inter-cluster difference, respectively. We also examine the cluster attributes and sizes subjectively.

Procedure 6: Choose the partition with two clusters as the best one, because it had a reasonable silhouette score, a low variance ratio coefficient, and meaningful differences between the clusters. Each cluster represented a group of individuals with similar route choice strategies.

For Procedure 1, we observe the data. For this methodology, the responses pointed to eight factors pedestrians consider when choosing a route.

We need to calculate the probabilities for Section I to V.

To calculate the probabilities:

1. Assign a value to the questions, which we already did in the Sample Data Set for each Section.
2. Calculate the frequencies. Count the number of times each response was chosen by the participants. This gives the frequency of each response.
3. Calculate the probabilities. The probability of each response is then calculated by

dividing the frequency of each response by the total number of responses.

$$\text{probability} = \frac{\text{frequency}}{\text{total number of responses}}$$

For Procedure 2, we identified that these six factors from Step 1 can be variables to represent the probability of using different Urban Elements and Path Attributes. The six variables can now be considered as the ABM Parameters. The ABM Parameters are as follows:

**Example 32: ABM Parameters representing the probability of using different urban elements and path attributes**

- Landmarks ( $p_L$ )
- Barriers ( $p_B$ )
- Distance ( $p_D$ )

Recall that these are the variables indicated in Section 3.1.7. However, we eliminate some variables for simplicity of University ( $U$ ) for this example.

From the pedestrian data, we can also obtain the sample attributes. The sample attributes are the mean and standard deviation values of the variables obtained from the responses to the questionnaire. Sample attributes are used to calibrate the parameters of the agent-based model (ABM) that simulates pedestrian movement in cities.

The mean value ( $\mu$ ) represents the average probability of a certain behavior or preference over the entire study sample, while the standard deviation value ( $\sigma$ ) measures the variation or dispersion of the probabilities around the mean.

**Example 33: Mean and Standard Deviation General Example**

For example, the sample attribute  $\mu_D$  indicates the mean probability of completing a route exclusively due to D, and  $\sigma_D$  indicates how much this probability varies across the study sample. Different sample attributes are used for different behavioral components of the ABM.

We expound more on  $\mu$  and  $\sigma_c$ .

We use the  $\mu$  variable to represent the mean value of a sample attribute or a cluster attribute. The mean value indicates the average tendency of using a certain urban element or path attribute within the sample or the cluster.

#### Example 34: Mean Implication Example

For example,  $\mu_D$  is the mean value of the probability of formulating a Distance-Based path. A high  $\mu_D$  means that the participants in the sample or the cluster are more likely to use Distance as a way to segment the **University** ( $U$ ), and plan their routes in  $SR$ .

Meanwhile, we use the variable  $\sigma$  in sample attributes to represent the standard deviation of a certain variable over the entire study sample or within a cluster. It is used to model the variation or diversity of the agent's behavior in the ABM.

#### Example 35: Standard Deviation Implication Example

For example, a high value of  $\sigma_L$  for the probability of using Landmarks means that the agents in that sample or cluster have different preferences for using  $L$  in their route choice. A low value of  $\sigma$  means that the agents are more consistent or homogeneous in their behavior, meaning they are all using Landmarks to plan their routes in  $SR$ .

The variable  $\sigma$  is also used to compute the value of the agent's stochastic discrete parameter ( $p_j$ ) as a normal distribution with a mean of  $\mu$  and a standard deviation of  $\sigma$ . This allows the agent to have a probabilistic behavior that reflects the empirical data.

Now that we explained how the ABM parameter and Sample Attributes was obtained, we tabulate the information for easier viewing:

Phase	Behavioral component	ABM parameter	Sample attributes
PP	Completing a route exclusively considering Distance	$p_D$	$\mu_D, \sigma_D$
SP	Relying on Landmarks	$p_L$	$\mu_L, \sigma_L$
	Preference for path segments along or within Barriers	$p_B$	$\mu_B, \sigma_B$

**Table 4.2:** The behavioral components of the ABM and the corresponding parameters that regulate the definition of the agent’s route choice behavior. They fall within PP: Prospective Planning and SP: Situated Planning. The ABM parameters represent probabilities, and they can assume values between 0.0 and 1.0. In the empirically-based configurations of the ABM, the sample attribute values are used to obtain the ABM parameter values. Sample attributes may be cluster attributes when agent typologies are built.

For Procedure 3, we now apply logarithmic transformation and z-scores standardization of the variables in Step 1. We do logarithmic transformation to change the scale of the data so that very large or very small values are reduced. This transformation makes the data more symmetrical and less skewed. The formula for the logarithmic transformation is given as follows:

$$y = \log(x + 1) \quad (4.1)$$

Where  $y$  is the transformed value, and  $x$  is the original value. The logarithmic transformation is applied to each variable in the dataset.

We now do z-scores standardization since the variables had different scales and distributions, which could affect the cluster analysis results. For example, some variables had higher values and more variation than others, which could make them more influential in the clustering process. Z-score standardization is a method of transforming a variable into a standardized form that has a mean of zero and a standard deviation of one. This allows comparing variables that have different units or scales. To perform z-score standardization, the following formula is used:

$$z = \frac{x - \mu}{\sigma} \quad (4.2)$$

Where  $z$  is the standardized value,  $x$  is the original value,  $\mu$  is the mean of the variable, and  $\sigma$  is the standard deviation of the variable. First we compute the mean and standard deviation of the logarithmic transformed data.

For Procedure 4, we use k-means algorithm with different numbers of desired clusters, ranging from 2 to 9. The k-means algorithm works by:

1. **Initialization:** Choose  $k$  initial cluster centers. These can be randomly selected from the data points, or they can be randomly generated.
2. **Assignment:** Assign each data point to the nearest cluster center. This is typically done using Euclidean distance, but other distance measures can also be used.
3. **Update:** Calculate the new cluster centers as the mean (average) of all data points assigned to each cluster.
4. **Repeat:** Repeat the assignment and update steps until the cluster assignments no longer change, or until a maximum number of iterations is reached.

The pseudocode for the k-means algorithm is given as follows:

```

RecursiveKMeans{data, k, centroids}{

    \If{convergence_criteria_met or max_iterations_reached}{

        return centroids;

    }

    clusters <- AssignToClusters{data, centroids};

    new_centroids <- UpdateCentroids{clusters};

    return RecursiveKMeans{data, k, new_centroids};

}

AssignToClusters{data, centroids}{

    clusters <- initialize_empty_clusters(k);

    For{each data_point in data}{

        min_distance <- infinity;

        nearest_centroid <- null;
    }
}

```

```

    For{each centroid in centroids}{

        distance <- calculate_distance(data_point, centroid)\;

        If{distance < min_distance}{

            min_distance <- distance;

            nearest_centroid <- centroid;

        }

    }

    add data_point to clusters[nearest_centroid];

}

return clusters;
}

UpdateCentroids{clusters}{

    new_centroids <- []\;

    \For{each cluster in clusters}{

        If{cluster is not empty}{

            new_centroid <- calculate_mean(cluster);

            add new_centroid to new_centroids;

        }

        Else{

            If a cluster is empty,
            keep the previous centroid unchanged
            add centroid_of_previous_iteration to new_centroids;

        }

    }

    return new_centroids;
}

```

### Recursive K-Means Algorithm

For Procedure 5, we evaluate the quality of each partition using two criteria: the silhouette score and the variance ratio criterion.

The silhouette score and the variance ratio criterion are two methods to evaluate the quality of a clustering structure, i.e., how well the clusters are separated and how similar their values are internally. We use these methods to choose the optimal number of clusters for the k-means algorithm, which partitions the data into groups based on the Route Choice Strategies.

#### 4.4.1 Silhouette score

This method computes the average silhouette coefficient of all observations, which measures how similar an observation is to its own cluster compared to other clusters. The coefficient ranges from -1 to 1, where higher values indicate better clustering. For this methodology, we calculate the silhouette score for 2 clusters. We choose the number of clusters that maximized the silhouette score, which is 2 in our case.

##### Example 36: Silhouette Score

The silhouette score considers both intra-cluster and inter-cluster distances, and it is computed for a certain partition as the average silhouette coefficient of all observations:

$$S = \frac{\sum_{x=1}^N \frac{b_x - a_x}{\max(a_x, b_x)}}{N} \quad (4.3)$$

Where

- N is the number of observations,
- $a_x$  indicates the mean distance from x to all other entities in x's cluster, and
- $b_x$  the mean distance from x to all the other points not belonging to x's cluster.

The coefficient can range between -1 (objects are closer to other clusters' objects than to their own cluster's members) and 1 (compact clusters, well separated from others)

#### 4.4.2 Variance Ratio Criterion

This method compares the between-cluster sum-of-squares (SSB) and the within-cluster sum-of-squares (SSW), which indicate how much variation is explained by the clustering and how much is left unexplained. The method computes the ratio of SSB to SSW, adjusted by the number of clusters (K) and the number of observations (N). Higher values of this ratio indicate better clustering.

We calculate the variance ratio criterion for 3 clusters. We chose the number of clusters that minimized the coefficient  $\omega$ , which measures the relative change in the variance ratio criterion between consecutive iterations.

Now, to compute for the SSB and SSW we use the following formula:

$$SSB = \sum (n_i - \bar{X})^2 \quad (4.4)$$

and

$$SSW = \sum_{i=1}^k (n_i - \bar{X}_i)^2 \quad (4.5)$$

Where

- $k$  is the number of clusters,
- $N_i$  is the number of items in the  $i$ -th cluster,
- $\bar{X}_i$  is the mean of the  $i$ -th cluster,
- $N$  is the number of items in all clusters, and
- $\bar{X}$  is the grand mean, the mean of the clusters.

##### Example 37: Variance Ration Criterion

The Variance Ratio Criterion coefficient  $\omega$ , an index designed to determine the optimal number of clusters. The VRC score of a partition is computed as:

$$VRC = \frac{\frac{SS_b}{K-1}}{\frac{SS_W}{N-K}} \quad (4.6)$$

where

- $SS_b$  is the between-cluster sum-of-squares,
- $SS_W$  indicates the within-cluster sum-of-squares,
- K is the number of clusters, and
- N is the number of observations.

The coefficient can range between -1 (objects are closer to other clusters' objects than to their own cluster's members) and 1 (compact clusters, well separated from others)

For Procedure 6, we choose the partition with two clusters as the best one, because it had a reasonable silhouette score, a low variance ratio coefficient, and meaningful differences between the clusters. Each cluster represented a group of individuals with similar route choice strategies.

#### 4.4.3 Empirical Data Cluster Analysis and Agent Typologies

##### Cluster Analysis

Number of Clusters	Silhouette Score	Variance Ratio Coefficient
3	0.800	549.06
4	0.841	896.749
5	0.828	1025.605
6	0.834	1411.464

**Table 4.3:** Number of clusters and Silhouette Score using K-means Algorithm, in comparison to VRC values.

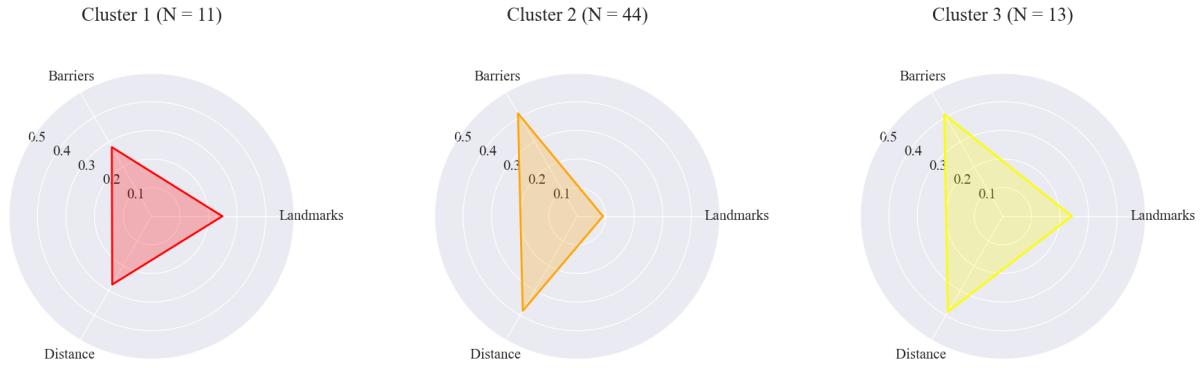
Table 4.3 presents the results of applying the K-means algorithm with varying numbers of clusters. The Silhouette Score, computed using Equation 4.3, a measure of how

well-defined the clusters are, is used to assess the quality of each number of clustering groups. The silhouette score considers both intra-cluster and inter-cluster distances, and it is computed for a certain partition as the average silhouette coefficient of all observations. A higher Silhouette Score indicates better-defined clusters. Based on the results, the highest Silhouette Score of 0.841 is achieved with six clusters, followed by four clusters with a score of 0.834. The other clusters, although exhibit slightly lower Silhouette Scores, still indicate a high degree of clustering differentiation. The three cluster shows the lowest Silhouette Score of 0.800.

Table 4.3 also delves into the specifics of the selected clusters by providing values for the VRC metrics across different cluster structures. Omega and VRC was computed using the Equation 4.6. The study aimed to identify the optimal number of clusters that maximize the effectiveness of the clustering algorithm in capturing the underlying patterns in the data.

Among the considered options, the cluster structure with three clusters exhibits the lowest VRC value (549.06). This indicates a substantial distinction between the clusters and a high level of variance explained by the clustering. The subsequent clusters, with four to six clusters, demonstrate varying degrees of Omega and VRC, offering insights into how the data is partitioned. Based on these values, we choose three clusters due to the optimal values of Silhouette Score, and VRC.

### Examining the Chosen Number of Clusters (3 Clusters)



**Figure 4.18:** Average Utilization of Variables for Clusters 1,2, and 3. Agent Typologies of the Clusters.

Cluster	Number of Individuals	Distance	Landmark	Barrier
1	11	0.385	0.241	0.411
2	44	0.276	0.248	0.279
3	13	0.382	0.090	0.415

**Table 4.4:** Average Utilization of Variables for Clusters 1,2, and 3 Tabulated Data. Agent Typologies of the Clusters.

In Figure 4.18 and Table 4.4 presents the results of a clustering analysis, where data is divided into three distinct clusters based on their route choice strategies.

Based on the Figure 4.18 and Table 4.4, Cluster 1 with 11 individuals shows high consideration for Barriers, moderate for Distance, and low for Landmarks. In Cluster 2 with 44 individuals shows relatively equal consideration of Barriers, Distance, and Landmarks within their route. However, there is a slightly higher consideration for Barriers, followed by Distance and Landmarks. In Cluster 3 with 13 individuals shows high consideration for Barriers, moderate consideration for Distance, and least consideration in Landmarks.

Each cluster represents a unique combination of the behavior values of the individuals when considering the three variables. These variations in probability values provide

insights into the different characteristics that are imported into the ABM. The clusters are used to define the agent typologies in the ABM, which will be used to simulate pedestrian movement in the UPB ( $U$ ).

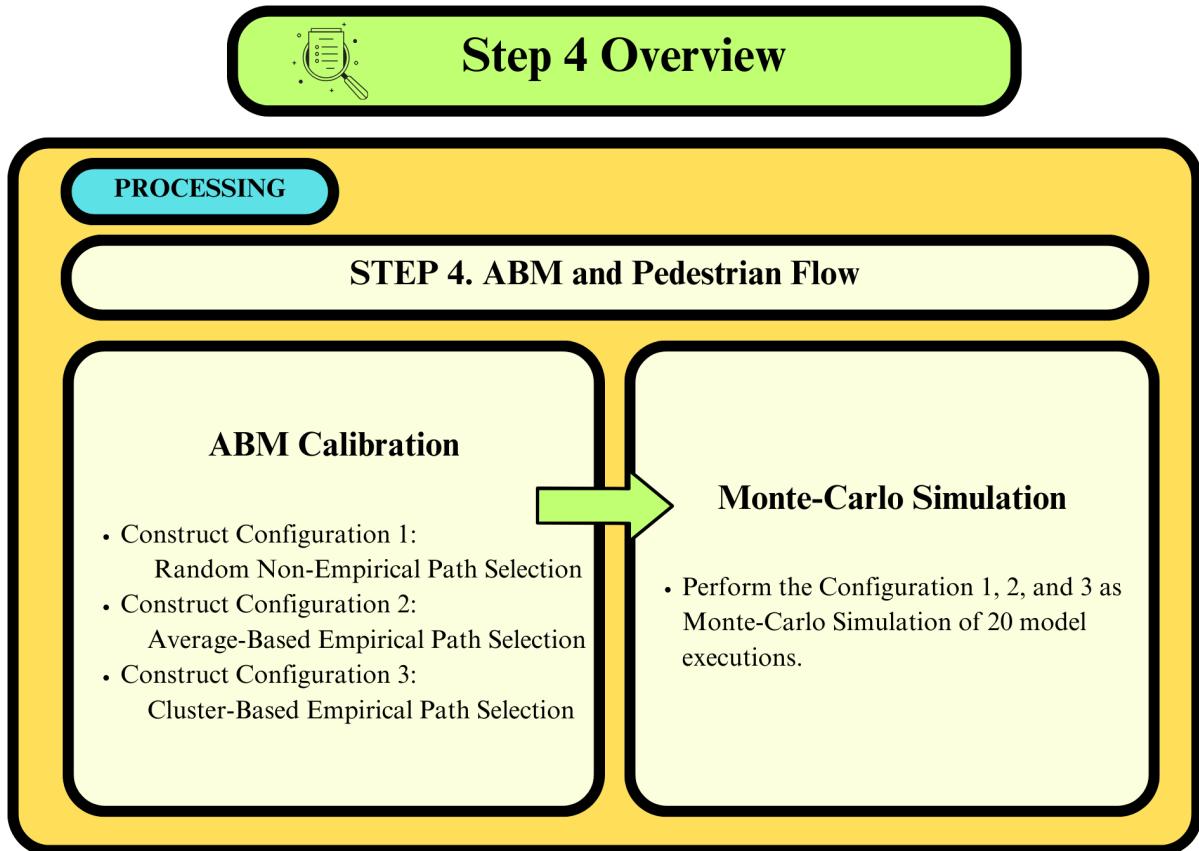
Cluster	Number of Individuals	Mean Age	Gender			
			Male	Female	Non-Binary	Prefer Not To Say
1	11	20.818	27.300	63.600	0.000	9.100
2	44	21.386	54.500	36.400	2.300	6.800
3	13	21.077	61.500	38.500	0.000	0.000

**Table 4.5:** Agent Typologies Demographics per Cluster in Probabilities.

Table 4.5 presents the demographics of the agent typologies per cluster. The table provides insights into the distribution of individuals across the clusters when the simulating pedestrian movement in the UPB ( $U$ ). The table shows the number of individuals, the mean age of the individuals, and the probability of individuals being male, female, non-binary, or preferring not to say in each cluster. The gender of the individuals in each cluster are represented in probabilities so that when the ABM increases the number of agents, it can use the probabilities to assign the gender of each agent.

The Cluster 1 has the highest proportion of female individuals, Cluster 2 has the highest mean age among the clusters and highest proportion of non-binary individuals, while Cluster 3 has the highest proportion of male individual.

## 4.5 Step 4. ABM and Pedestrian Flow



**Figure 4.19:** Methodology Step 4 Overview Illustration

After we verified the clusters through Silhouette Score and VRC, we can now proceed to the creation of the Java codes. We use Eclipse IDE. In addition, we use MASON, a fast agent-based simulation library core in Java, designed to be the foundation for large custom-purpose Java simulations. We implement the ABM using GeoMASON, a MASON extension that adds support for vector and raster geospatial data. In theory, we aim to simulate the behaviors of the three clusters of **Route Choice**. So, we create groups of algorithms for **Agents**, **Cognitive Map**, **Route Choice**, and **Utilities**. These algorithms were already explained in the Methodology on how we get these algorithms theoretically. Refer to the Appendix for the codes of these algorithms.

## A Agents Algorithm Group

In the Agents Algorithm Group, we create the algorithms for the **Agent**, **Agent Properties**, **Empirical Agent Properties**, and **Empirical Agents Group**.

The **Agent class** represents an agent in the pedestrian simulation along paths between origin and destination nodes.

The **Agent Properties class** represents the properties and preferences of an agent in the simulation. These properties influence the agent's navigation behavior and route choices.

The **Empirical Agent Properties class** is a subclass of **Agent Properties** that represents the properties of an agent in a pedestrian simulation with empirical-based parameters. It extends the base **Agent Properties class** to incorporate additional parameters.

The **Empirical Agents Group class** represents a group of empirical agents in the simulation. It encapsulates group-specific parameters and attributes that influence the behavior of agents within the group. These parameters include probabilities for various route choice properties, such as using elements, heuristics, region-based navigation, sub-goals, distant landmarks, and other route properties. Additionally, it stores information about natural and severing barriers and the group's share in the population.

## B Cognitive Map Algorithm Group

In the Cognitive Map Algorithm Group, we create the algorithms for the **Agent Cognitive Map**, **Agent Properties**, **Barrier**, **Barrier Integration**, **Cognitive Map**, **Gateway**, **Landmark Integration**, and **Region**.

The **Agent Cognitive Map class** represents an agent's cognitive map, which provides access to various map attributes.

The **Barrier class** represents a barrier in the cognitive map, defined by a unique identifier, geometry, edges it affects, and its type.

The **Barrier Integration class** returns a set of barriers in the direction of the destination node from a given location.

The **Cognitive Map class** represent a community share cognitive map (or Image of the University ( $U$ )) used for storing meaningful information about the environment and, in turn, navigating.

The **Gateway class** represents a gateway connecting two nodes.

The **Landmark Integration class** manages the integration of landmarks into a graph.

## C Engine Algorithm Group

In the Engine Algorithm Group, we create the algorithms for the **Environment**, **Export**, **Flow Handler**, **Import**, **Parameters**, **PedSimUniversity** ( $U$ ), and **Populate**.

The **Environment class** is responsible for preparing the simulation environment, including junctions, buildings, barriers, and regions.

The **Export class** is responsible for saving the simulation results to specified output directories.

The **Flow Handler class** provides methods for updating various data related to agent movement and route storing in the simulation.

The **Import class** is responsible for importing various data files required for the simulation based on the selected simulation parameters. It includes methods for importing distances, barriers, landmarks and sight lines, path network graphs, and empirical agent groups data.

The **Parameters class** contains global parameters and settings for the PedSimUniversity ( $U$ ) simulation. These parameters are used to configure various aspects of the simulation, including simulation mode, agent behavior, and data import options.

The **PedSimUniversity** ( $U$ ) **class** represents the main simulation environment.

The **Populate class** is responsible for generating test agents, building the OD matrix, and populating empirical groups for pedestrian simulation.

## D Route Choice Algorithm Group

In the Route Choice Algorithm Group, we create the algorithms for the **Barrier Based Navigation**, **Complexity**, **Global Landmarks Path Finder**, **Landmarks Navigation**, **Path Finder**, **Path Distance Path Finder**, **Route**, and **Route Planner**.

The **Barrier Based Navigation class** is a series of functions for computing a sequence of barrier sub-goals in the space between an origin and a destination.

The **Complexity class** computes way finding easiness and legibility complexity for navigation. It includes methods for calculating way finding easiness within a space or region based on distance and landmarks.

The **Global Landmarks Path Finder class** is a series of functions that support the generation of routes calling different methods.

The **Landmarks Navigation class** is a series of functions that support landmark-based navigation. It also supports landmarkness computation, identification of landmarks and way finding easiness of a certain space.

The **Path Finder class** provides common functionality for computing navigation paths using various algorithms and graph representations.

The **Region Based Navigation class** is a series of functions for computing a sequence of region-gateways between the space generated by an origin and a destination. This class generates therefore the so-called navigational coarse plan.

The **Path Distance Path Finder class** is a pathfinder for path-distance based route calculations.

This class extends the functionality of the base class **PathFinder**.

The **Route class** is a class for storing the sequence of GeomPlanarGraphDirectedEdge in a path and the sequence of NodeWrappers. It supports shortest-path algorithms and provides some utilities.

The **Route Planner class** is responsible for calculating a route for an agent within a pedestrian simulation. It considers the agent's route choice properties and strategies to determine the optimal path from an origin node to a destination node.

## E Utilities Algorithm Group

In the Cognitive Map Algorithm Group, we create the algorithms for the **Route Data**, and **String Enumeration**.

The **Route Data** store information about the walked routes.

The **String Enumeration** enumerates the strings for Route Choice, Route Choice Property, Groups, Landmark Types, and Barrier Types.

### 4.5.1 ABM Calibration

ABM calibration is the process of adjusting the parameters of the agent-based model (ABM) to match the empirical data collected from the questionnaire. Here, we compare three different ABM configurations: Configuration 1: Random Non-empirical Path Selection, Configuration 2: Average-Based Empirical Path Selection, and Configuration 3: Cluster-Based Empirical Path Selection.

In the Configuration 1, the parameter values are randomly drawn from uniform distributions, without using any empirical data. This configuration serves as a baseline to evaluate the other configurations where the agents do not use any urban elements or path attributes to formulate their routes, but randomly choose a direction at each decision point.

In the Configuration 2, the parameter values are derived from the mean of the sample attributes, which are the variables obtained from the questionnaire responses over the entire study sample. This configuration assumes that all agents have the same average behavior.

In the Configuration 3, the parameter values are inherited from the cluster attributes, which are the variables obtained from the cluster analysis of the questionnaire responses. This configuration accounts for the behavioral heterogeneity of the agents, who belong to different clusters or typologies.

The parameter values are used to regulate the agent's reliance on urban elements and path attributes for formulating their routes, as well as their preference or aversion for segments near natural or severing barriers.

We evaluate the ABM configurations by comparing the distribution of pedestrian agents across the path network, resulting from the different parameter values. We use descriptive statistics and visualization tools to analyze the movement patterns and volumes of the agents.

The ABM evaluation is the process of comparing the movement patterns of pedestrian agents across the path network resulting from three different ABM configurations: Configuration 1, 2, and 3.

The ABM evaluation aims to answer the research question:

- To what extent does the variation in the agents' cluster-based empirical path selection — i.e. a model including agent typologies vs a model with average-based empirical path selection — generate different movement patterns across the path network?

The ABM evaluation consists of running the ABM with each configuration, each with a different random seed, and computing the median volumes of pedestrian agents per path segment across the iterations. We execute the three configurations 5 times each as Monte Carlo simulations to balance the randomness entailed by the selection of the OD pairs and the stochastic functions of the ABM.

Here, when we mean Monte Carlo Simulation, it is a mathematical technique that predicts possible outcomes of an uncertain event.

#### Example 38: Monte Carlo Simulation Procedure

To perform a Monte Carlo simulation, we follow these four steps:

1. Define the problem and the uncertain variables. Identify the input variables that have uncertainty or randomness, which for this methodology is the Route Choice. Assign probability distributions to these variables based on the available data or expert judgment.
2. Generate random samples of the input variables. Use a random number generator to draw values from the probability distributions of the input variables. Repeat this process for n times (e.g. 10,000) to create a set of scenarios or trials.

3. Run the simulation model for each scenario. Plug in the sampled values of the input variables into the ABM. Calculate the output variables of increase, or decrease.
4. Analyze the results and draw conclusions. Summarize the distribution of the output variables using descriptive statistics, such as mean, median, standard deviation, or percentiles. Visualize the results using histograms, box plots, or scatter plots. Identify the most likely outcomes, the best and worst cases, and the sensitivity of the output variables to the input variables.

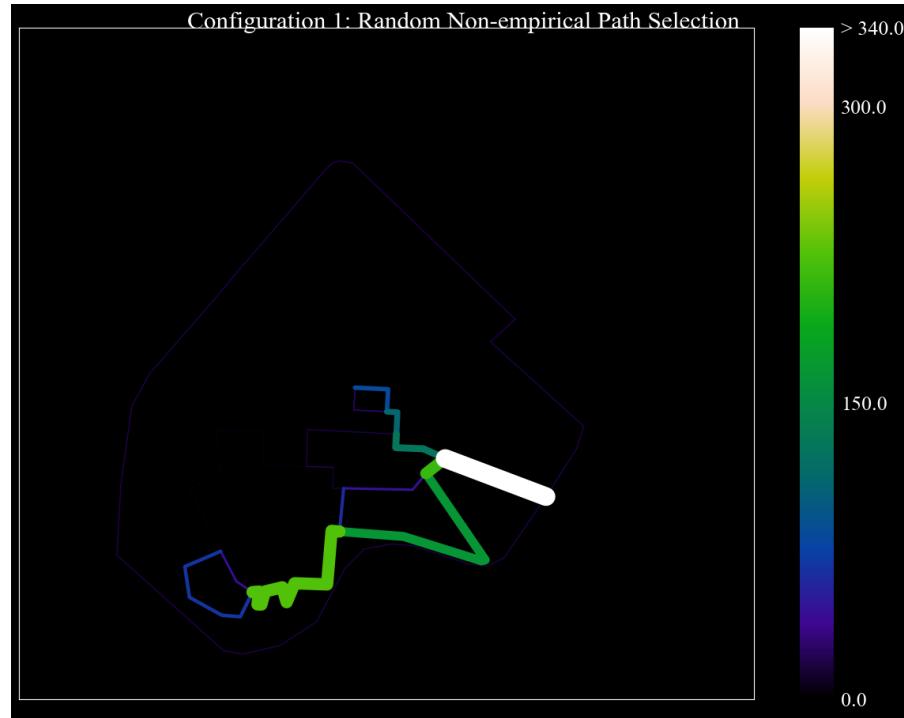
The volumes are then visualized on a map and compared qualitatively and quantitatively. At the end of a single model execution, the ABM stores the number of times a path segment was crossed by the agents, thus computing pedestrian volumes per segment.

For each configuration, the pedestrian volumes were determined from the pedestrian counts of the path segments (median value over the  $T = 20$  model executions). To visualize the distribution of the pedestrian agents, we generated a figure representing the pedestrian volumes for the Configuration 1 across the entire path network. Furthermore, for each segment, we verified whether the frequency distribution of the pedestrian volumes over the  $T = 20$  executions in the Configuration 2 and Configuration 3 differed significantly from the Configuration 1.

Next, we use the Wilcoxon test, a non-parametric version of the t-test. The Wilcoxon test compares the differences between two paired samples or two related groups. It is also known as the Wilcoxon signed-rank test or the Wilcoxon matched-pairs test. It is used when the data are not normally distributed or when the sample size is small.

### 4.5.2 Empirical Data ABM Simulation per Configuration

#### Configuration 1: Random Non-Empirical Path Selection



**Figure 4.20:** Configuration 1 Volume per Path Segment

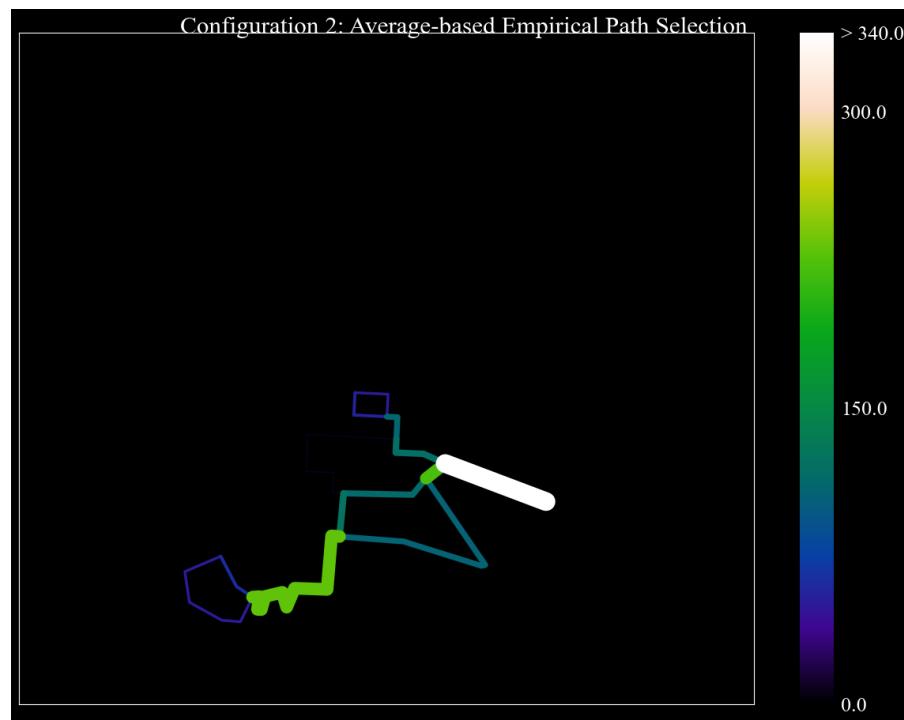
edgeID	Volume								
0	66	11	0	21	0	31	26	41	0
1	0	12	127	22	13	32	0	42	0
2	0	13	127	23	60	33	4	43	0
3	26	14	109	24	225	34	0	44	0
4	0	15	83	25	0	35	0	45	0
5	18	16	0	26	4	36	340	46	0
6	0	17	18	27	43	37	19		
7	19	18	0	28	4	38	19		
8	0	19	0	29	212	39	19		
9	19	20	26	30	46	40	0		

**Table 4.6:** Configuration 1 Volume per Path Segment Tabulated Data

The Figure 4.20 is the path network subjected into Configuration 1 where agents choose path randomly. The colors represent the volume of agents passing through each path segment, with brighter colors indicating higher volumes. The map is divided into various segments, each representing a different path. The volume of agents passing through each segment is quantified by the color intensity, with brighter colors indicating higher volumes of agents passing the path segment.

This Configuration 1 served as a baseline for comparing Configuration 2 and Configuration 3. After running the ABM in Configuration 1, we now run the ABM in Configuration 2 and Configuration 3.

### Configuration 2: Average-based Empirical Path Selection



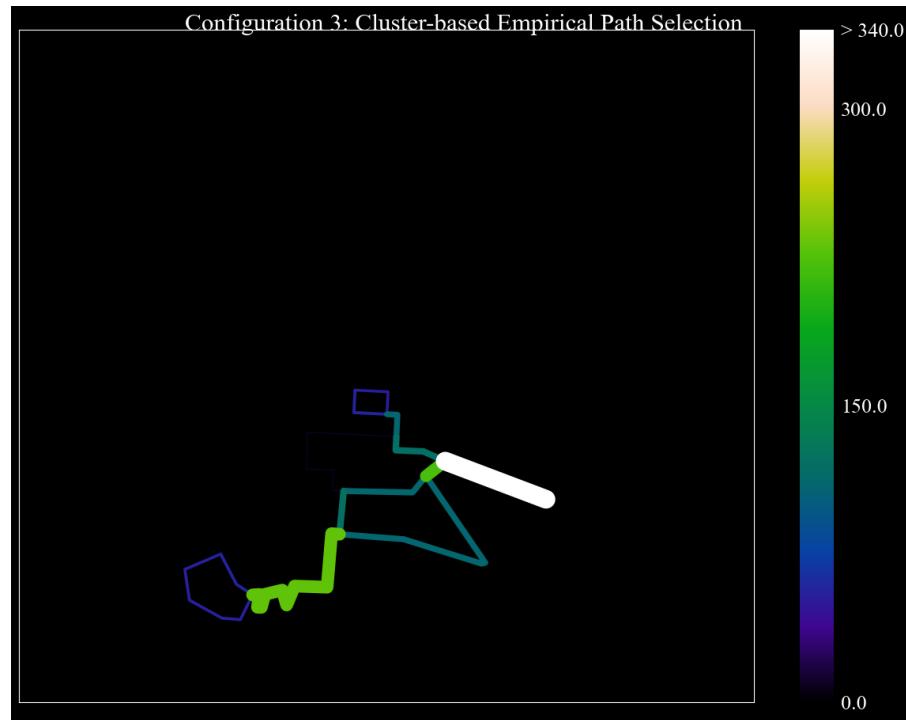
**Figure 4.21:** Configuration 2 Volume per Path Segment

edgeID	Volume	edgeID	Volume	edgeID	Value	edgeID	Volume	edgeID	Volume
0	51	11	0	21	9	31	106	41	0
1	0	12	118	22	124	32	0	42	0
2	0	13	118	23	231	33	0	43	0
3	57	14	109	24	0	34	0	44	0
4	0	15	51	25	0	35	0	45	0
5	9	16	0	26	0	36	340	46	0
6	0	17	9	27	60	37	0		
7	0	18	0	28	0	38	0		
8	0	19	0	29	221	39	0		
9	0	20	57	30	113	40	0		

**Table 4.7:** Configuration 2 Volume per Path Segment Tabulated Data

The Figure 4.21 is the path network subjected into Configuration 2 where agents choose path based on the average path weight and probabilities for each path segment from the empirical data. The colors represent the volume of agents passing through each path segment, where black is the absolute 0 agents passed, and 340 as the maximum number of agents passed. As the color moves from dark to brighter colors, the volumes of agents who passed the path segment increases.

### Configuration 3: Cluster-based Empirical Path Selection



**Figure 4.22:** Configuration 3 Volume per Path Segment

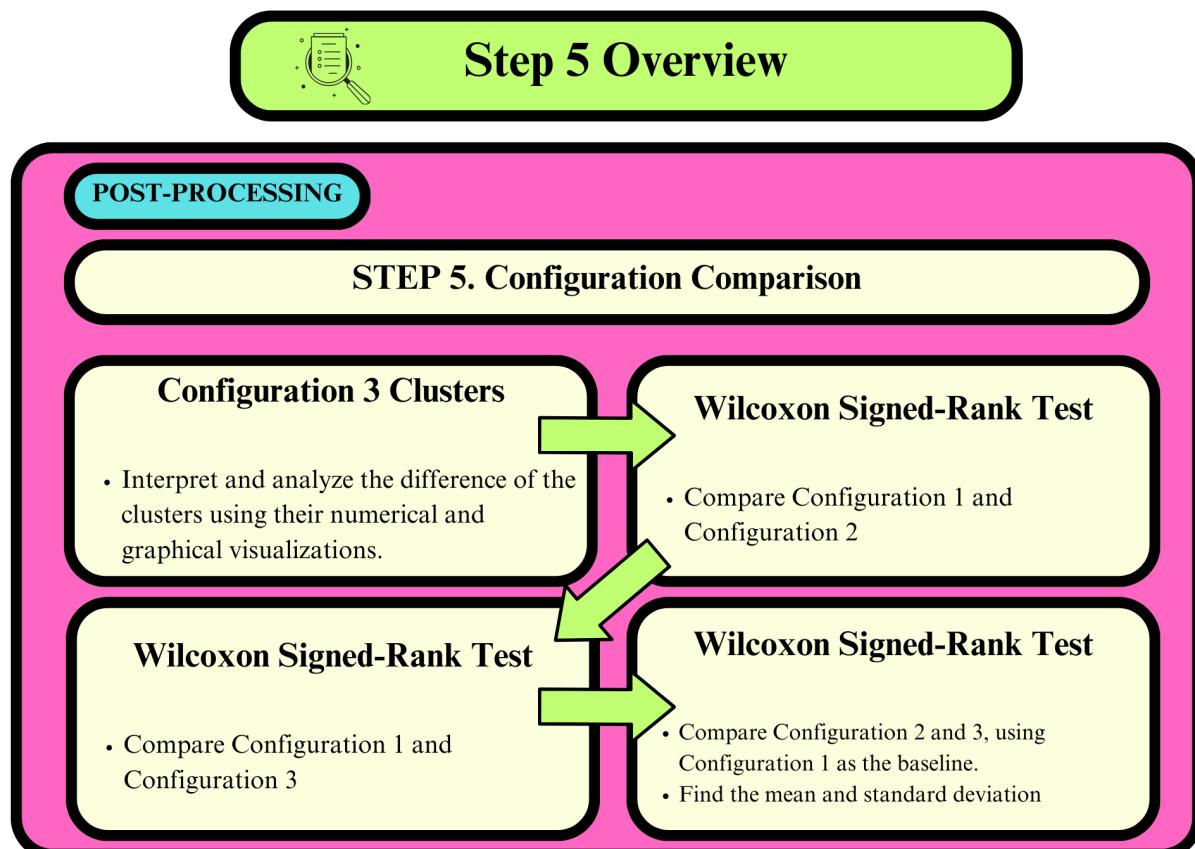
edgeID	Volume	edgeID	Volume	edgeID	Value	edgeID	Volume	edgeID	Volume
0	55	11	0	21	10	31	110	41	0
1	0	12	119	22	121	32	0	42	0
2	0	13	119	23	231	33	10	43	0
3	56	14	109	24	0	34	0	44	0
4	0	15	53	25	0	35	0	45	0
5	10	16	0	26	0	36	340	46	0
6	0	17	10	27	57	37	0		
7	0	18	0	28	0	38	0		
8	0	19	0	29	220	39	0		
9	0	20	56	30	111	40	0		

**Table 4.8:** Configuration 3 Volume per Path Segment Tabulated Data

The Figure 4.22 is the path network subjected into Configuration 3 where agents choose path based on the cluster path weight and probabilities for each path segment from the empirical data. The colors represent the volume of agents passing through each path segment, where black is the absolute 0 agents passed, and 340 as the maximum number of agents passed. As the color moves from dark to brighter colors, the volumes of agents whom passed the path segment increases.

## 4.6 Step 5. Configurations Comparison

### ABM Evaluation



**Figure 4.23:** Methodology Step 5 Overview Illustration

**Example 39: Wilcoxon Test Procedure**

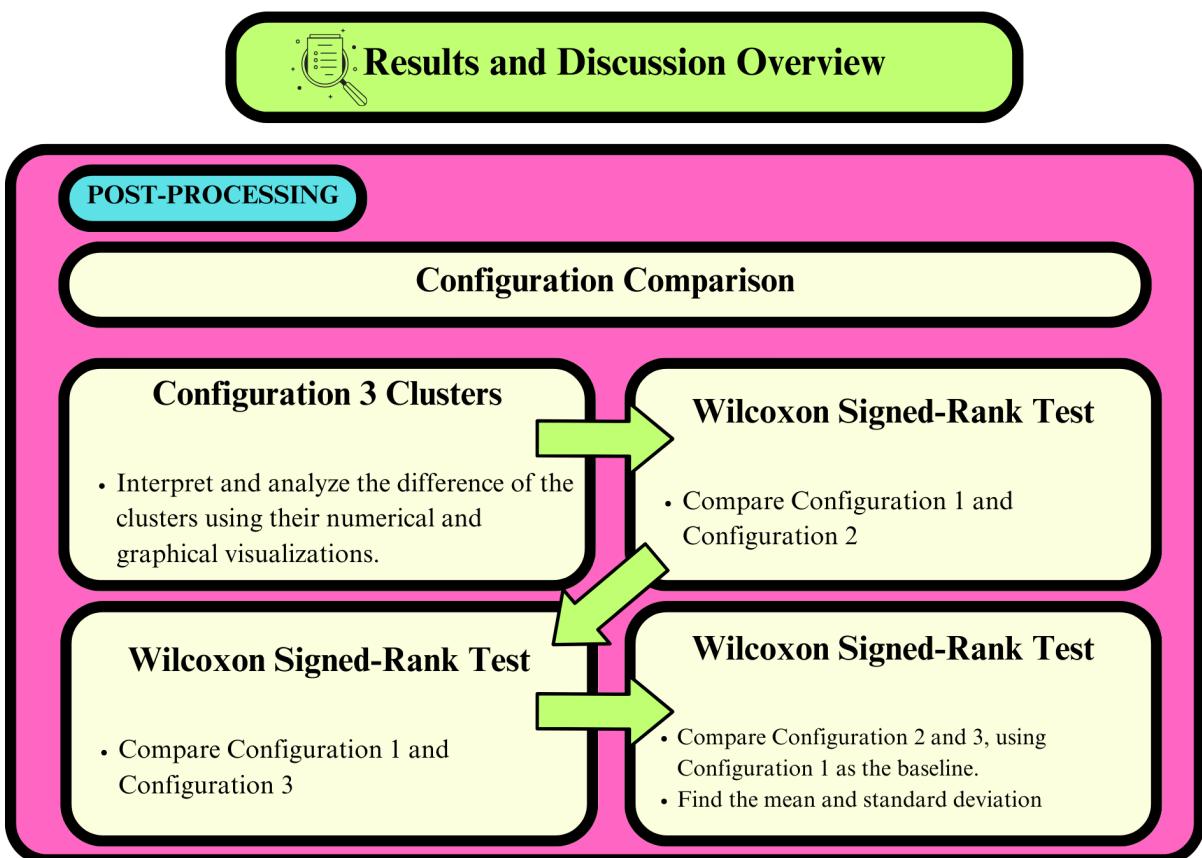
To perform the Wilcoxon test, the following steps are required:

1. Calculate the difference between each pair of observations and rank the absolute values of the differences from smallest to largest.
2. Assign a positive or negative sign to each rank, depending on the sign of the difference.
3. Sum the positive and negative ranks separately and obtain the test statistic  $W$ , which is the smaller of the two sums.
4. Compare  $W$  to a critical value from a Wilcoxon table or calculate a p-value.
5. Reject the null hypothesis of no difference between the paired samples or groups if  $W$  is smaller than the critical value or the p-value is less than the significance level.

We use Wilcoxon test for this purpose with a  $0.05 \alpha$  value. Thereby, we obtain a second figure indicating for which segments the empirically based configurations generated statistically significant different pedestrian volumes from the Configuration 1.

# Chapter 5

## Results and Discussion



**Figure 5.1:** Results and Discussion Step Overview Illustration

In these results and discussion, we fulfilled the following:

Report the findings of the study and the cluster analysis, as well as the movement patterns generated from the different ABM configurations. We used tables and graphical visualization to illustrate the data and the outcomes of the simulation. We compared and contrasted the results of the three configurations used, highlighted the differences

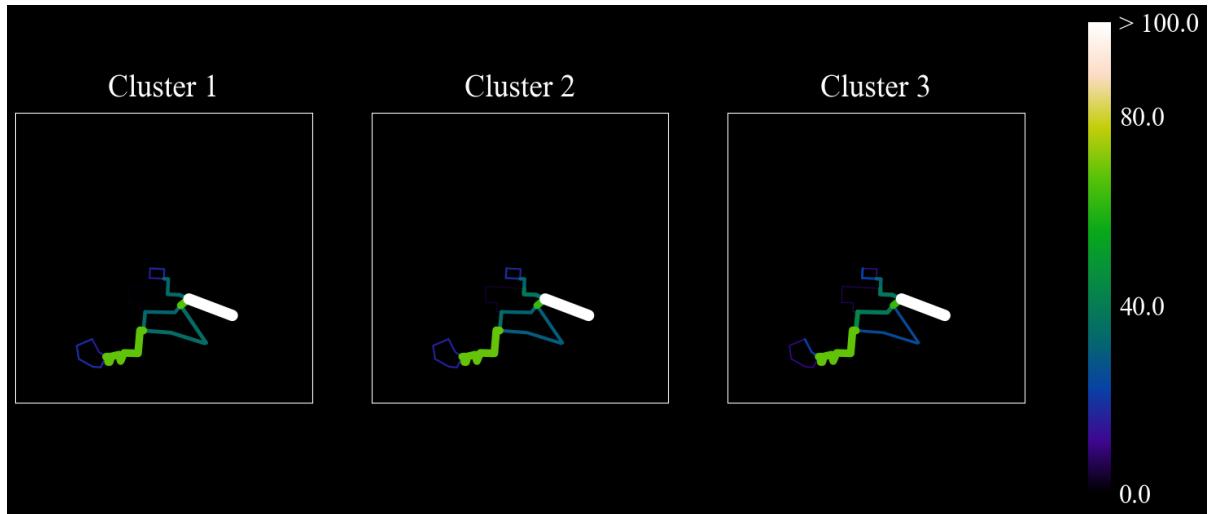
in terms of pedestrian volumes, and usage of urban elements. Included here are the descriptive statistics and measures of optimal clustering to support the analysis.

The study questionnaire gathered 71 responses from individuals. Only 68 responses were accepted after filtering those who completed the survey in less than a minute. Now, we evaluated the results from Section I, II, III, IV, and V.

Meanwhile, we presented the summarized data below. The Demographic Information: General Information, Age Categories, Links or relationship to the case-study area, Reasons for walking data are used to find the percentages.

## 5.1 Configuration Comparison

### 5.1.1 Volume of Configuration 3 Clusters



**Figure 5.2:** Comparison of the Volumes per Path Segment in Clusters 1, 2, and 3, shown in probabilities.

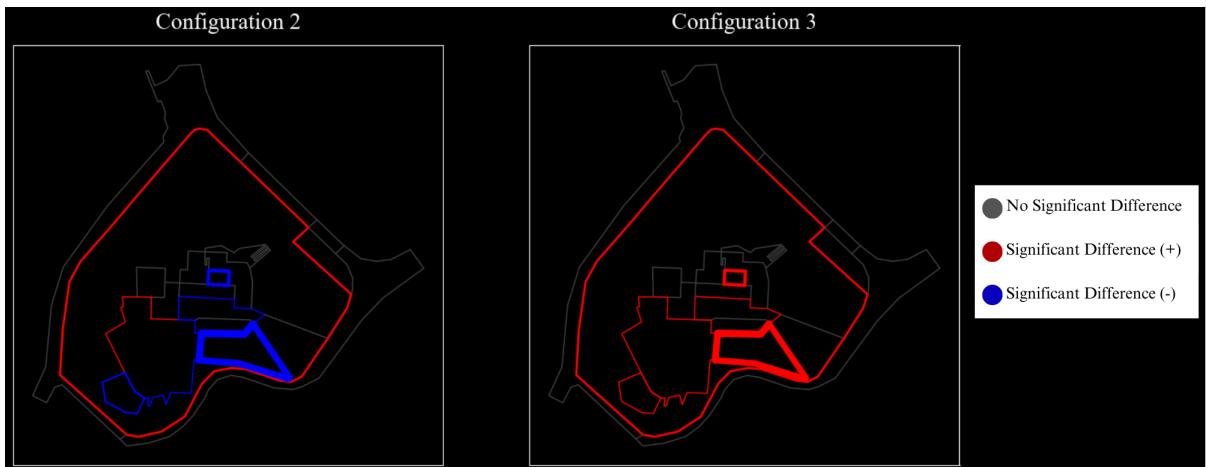
edgeID	Cluster (%)		
	1	2	3
0	17.73	16.36	11.54
1	0.00	0.00	0.00
2	0.00	0.00	0.00
3	15.45	18.18	21.54
4	0.00	0.00	0.00
5	1.36	3.64	6.15
6	0.00	0.00	0.00
7	0.00	0.00	0.00
8	0.00	0.00	0.00
9	0.00	0.00	0.00
10	0.00	0.00	0.00
11	0.00	0.00	0.00
12	33.41	37.27	37.69
13	33.41	37.27	37.69
14	32.27	31.82	32.31
15	17.05	16.36	11.54
16	0.00	0.00	0.00
17	1.36	3.64	6.15
18	0.00	0.00	0.00
19	0.00	0.00	0.00
20	15.45	18.18	21.54
21	1.36	3.64	6.15
22	33.64	34.55	43.08
23	67.73	68.18	67.69
edgeID	Cluster (%)		
	1	2	3
24	0.00	0.00	0.00
25	0.00	0.00	0.00
26	0.00	0.00	0.00
27	15.91	16.36	20.77
28	0.00	0.00	0.00
29	66.59	62.73	62.31
30	32.05	31.82	38.46
31	34.55	30.00	24.62
32	0.00	0.00	0.00
33	1.36	3.64	6.15
34	0.00	0.00	0.00
35	0.00	0.00	0.00
36	100.00	100.00	100.00
37	0.00	0.00	0.00
38	0.00	0.00	0.00
39	0.00	0.00	0.00
40	0.00	0.00	0.00
41	0.00	0.00	0.00
42	0.00	0.00	0.00
43	0.00	0.00	0.00
44	0.00	0.00	0.00
45	0.00	0.00	0.00
46	0.00	0.00	0.00

**Table 5.1:** Configuration 3 Clusters Volume per Path Segment Tabulated Probability Data

In Figure 5.2, there are visual differences with the paths taken by the agent, based on their priority when choosing a path. Cluster 1 shows the route choice strategy which

focuses on barriers and distance in UPB. Cluster 2 shows the route choice strategy that aims to balance the use of barriers, landmarks, and distance. In Cluster 3 shows the route choice strategy focused on high use of barriers and distance, and moderate use of landmarks.

### 5.1.2 Configuration 1, 2, and 3 Comparison Where Configuration 1 Acts as the Baseline



**Figure 5.3:** Statistically significant volume difference of Configuration 2 and Configuration 3 in movement flows of pedestrian agents across the path network. Configuration 1 is the baseline of the comparison. Red represent positive difference. Blue represent negative difference. Gray shades represent no significant difference. “+” indicates segments for which the volumes generated by the Configuration 2 or Configuration 3 are higher than the Configuration 1; “-” indicates segments with lower volumes.

edgeID	Configurations							p Value Classification	
	Averaged Volume of Configuration per Path			Absolute Difference (p Value)					
	1	2	3	2–1	(2–1)%	3–1	(2–1)%	2–1	3–1
0	66	51	55	15	4.41	11	3.23	Significant diff. (+)	Significant diff. (+)
1	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
2	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
3	26	57	56	30	8.82	30	8.82	Significant diff. (+)	Significant diff. (+)
4	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
5	18	9	10	9	2.65	8	2.35	Significant diff. (+)	Significant diff. (+)
6	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
7	19	0	0	19	5.58	19	5.58	Significant diff. (-)	Significant diff. (+)
8	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
9	19	0	0	19	5.58	19	5.58	Significant diff. (-)	Significant diff. (+)
10	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
11	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
12	127	118	119	9	2.65	8	2.35	Significant diff. (+)	Significant diff. (+)
13	127	118	119	9	2.65	8	2.35	Significant diff. (+)	Significant diff. (+)
14	109	109	109	0	0.00	0	0.00	No significant diff.	No significant diff.
15	83	51	53	32	9.41	30	8.82	Significant diff. (+)	Significant diff. (+)
16	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
17	18	9	10	9	2.65	8	2.35	Significant diff. (+)	Significant diff. (+)
18	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
19	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
20	26	57	56	30	8.82	30	8.82	Significant diff. (+)	Significant diff. (+)
21	13	9	10	4	1.18	3	0.88	Significant diff. (+)	Significant diff. (+)
22	60	124	121	64	18.82	61	17.94	Significant diff. (+)	Significant diff. (+)
23	225	231	231	6	1.76	6	1.76	Significant diff. (+)	Significant diff. (+)
24	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
25	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
26	4	0	0	4	1.18	4	1.18	Significant diff. (-)	Significant diff. (+)
27	43	60	57	17	5.00	14	4.12	Significant diff. (+)	Significant diff. (+)
28	4	0	0	4	1.18	4	1.18	Significant diff. (-)	Significant diff. (+)
29	212	221	220	9	2.65	8	2.35	Significant diff. (+)	Significant diff. (+)
30	46	113	111	67	19.71	65	19.12	Significant diff. (+)	Significant diff. (+)
31	165	106	110	58	17.06	55	16.18	Significant diff. (+)	Significant diff. (+)
32	4	0	0	4	1.18	4	1.18	Significant diff. (-)	Significant diff. (+)
33	22	9	10	13	3.82	12	3.53	Significant diff. (+)	Significant diff. (+)
34	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
35	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
36	340	340	340	0	0.00	0	0.00	No significant diff.	No significant diff.
37	19	0	0	19	5.59	19	5.59	Significant diff. (-)	Significant diff. (+)
38	19	0	0	19	5.59	19	5.59	Significant diff. (-)	Significant diff. (+)
39	19	0	0	19	5.59	19	5.59	Significant diff. (-)	Significant diff. (+)
40	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
41	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
42	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
43	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
44	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
45	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
46	0	0	0	0	0.00	0	0.00	No significant diff.	No significant diff.
Mean					5.98		5.69		

**Table 5.2:** Statistically Significant Volume Difference of Configuration 2 and Configuration 3 from Configuration 1.

In Figure 5.3 and Table 5.2, the three Configurations are subjected to Wilcoxon Signed Rank Test. The paths taken by the Configuration 2 agents significantly differ from the paths taken by the Configuration 3 agents. Naturally, there are 340 agents passing through the path segment e36, which is the UPB Main Entrance. Next, the path segments e22, e23, and e29 (refer to Figure 4.10) are the next most used paths by the agents in the three configurations.

In Configuration 2, the p value classification shows that there are significant differences in the paths taken by the agents when compared with Configuration 1. There are observed positive and negative differences the p value classification for Configuration 2.

In Configuration 3, the p value classification shows that there are only positive differences in the paths taken by the agents when compared with Configuration 1. This result indicates that the path taken in Configuration 3 always have higher agents compared to Configuration 1.

As shown in this figure, the use of UPB Entrance is the most crucial path when navigating to the three buildings of UPB: the KA building, CSS building, and CAC building. In addition, we see that Configuration 3 agents going from UPB Entrance to CSS or CAC Building prefers the path the passes through the Court A and UPB Library.

Lastly, we compare the mean of the differences in the volumes of the three configurations. The mean difference of Configuration 2 from Configuration 1 is 5.98%, while the mean difference of Configuration 3 from Configuration 1 is 5.69%. This result indicates that the Configuration 2 and 3 agents take different paths compared to Configuration 1 agents.

These results imply that the use of empirical data can provide a more realistic approach to the pedestrian route choice in a small urban environment such as the University of the Philippines Baguio. The results corroborate the findings of the study by Filomena in 2022 [31], which established that accounting empirical date in pedestrian route choice in a city environment is a more realistic approach in modelling the behavior of pedestrians. The same as in the city, the use of empirical data in a small urban environment can provide a more realistic approach to the pedestrian route choice.

In terms of behavioral heterogeneity, the results show that the use of empirical data can provide a more realistic approach to the pedestrian route choice in a small urban

environment such as the University of the Philippines Baguio. Considering that the study area is a small urban environment and the variables only considered are Distance, Landmarks, and Barriers, the average difference of 5.98% and 5.69% volume per path segment in Configuration 2 and 3, respectively, is a significant improvement in the model, meaning that the difference will scale as the number of agents increases.

# Chapter 6

## Conclusion and Recommendation

### 6.1 Conclusion

The comprehensive analysis of 68 survey responses conducted on the pedestrian route choice behavior within the University of the Philippines Baguio (UPB) campus provided valuable insights into the demographic distribution, walking preferences, and the influence of variables: Distance, Landmarks, and Barriers on pedestrian route choices. By integrating survey data with agent-based modeling (ABM) simulations, this study aimed to understand the complexities of pedestrian movement within the UPB case study area. The following research questions guided the study: (a) What is the diversity in route choice strategies as concerns the usage of Distance, Landmarks, and Barriers? (b) To what extent does the variation in the agents' cluster-based empirical path selection — i.e. a model including agent typologies vs a model with average-based empirical path selection — generate different movement patterns across the path network?

The demographic analysis revealed a predominantly young participant group, with a strong representation of individuals aged 18 to 25, reflecting the student population of UPB. The study also highlighted the gender distribution, with a slightly higher proportion of males participating. Additionally, all participants had a direct link to the case-study area, either as current or former students of UPB, reinforcing the relevance of the study to the UPB spatial behavior.

The survey results indicated that commuting to/from university was the most prevalent walking activity among participants, emphasizing the importance of understanding pedestrian behavior within an academic setting. By transforming average ratings into probabilities, the study provided a standardized measure for comparing the perceived importance of various walking purposes, facilitating further analysis.

Cluster analysis revealed distinct route choice strategies among participants, with

three identified clusters demonstrating different preferences for barriers, distance, and landmarks. Cluster 1, 2, and 3 were characterized by high importance of Barriers, moderate importance of Distance, and low importance of Landmarks, respectively. This observed pattern indicates the consistent influence of physical obstacles on pedestrian route choices of individuals in University of the Philippines Baguio. The analysis also identified variations in the importance of Distance, Landmarks, and Barriers across clusters, which highlight the variation of pedestrian preferences within the UPB campus.

These findings underscore the importance of cluster-based empirical path selection in pedestrian behavior and the importance of considering individual preferences in agent-based simulations.

The ABM simulations compared three configurations: Configuration 1: Random Non-Empirical Path Selection, Configuration 2: Average-Based Empirical Path Selection, and Configuration 3: Cluster-Based Empirical Path Selection.

Both the average-based and cluster-based empirical path selection configurations demonstrated more realistic pedestrian movement patterns compared to the random non-empirical path selection configuration. However, the cluster-based empirical path selection configuration exhibited an all-positive differences in pedestrian movement patterns, reflecting that the influence of individual preferences — i.e. the use of behavioral heterogeneity — as represented using cluster-based empirical path selection approach, shows no similarity to the random non-empirical path selection. The results indicated that incorporating empirical data, most especially clustered empirical data, from the survey into the ABM led to significant differences in pedestrian movement patterns, which indicate the value of incorporating clustered real-world behavior into modeling approaches relating to the study of spatial behavior in the University of the Philippines Baguio.

In conclusion, the study successfully developed and applied an empirically-based ABM to investigate pedestrian route choice strategies in UPB. Through the integration of survey data with computational modeling techniques, the study provided insights into the complex dynamics of pedestrian behavior and its implications for agent-based modeling. Through the University of the Philippines Baguio as the case study area, we have shown the importance of considering heterogeneity (i.e. cluster-based empirical approach) and

empirical data in ABM frameworks to accurately represent real-world phenomena and inform decision-making processes in small-scale areas.

## 6.2 Recommendations

The study provides a foundation for future research on pedestrian behavior and agent-based modeling in the context of UPB. The following recommendations aim to build on the study's findings and address potential areas for further investigation:

1. **Longitudinal Studies:** Conducting longitudinal studies to track pedestrian behavior over time could provide valuable insights into the stability and variability of route choice strategies. By analyzing changes in walking patterns and preferences, researchers can gain a deeper understanding of the factors influencing pedestrian movement within the UPB campus.
2. **3-Dimensional Cognitive Map:** Developing a 3-dimensional cognitive map of the UPB campus could enhance the realism of agent-based simulations and capture the spatial complexity of pedestrian movement. By incorporating detailed spatial information, researchers can model the campus environment more accurately and explore how 3D spatial configurations influence route choices.
3. **Dynamic Environments:** Integrating dynamic elements, such as weather conditions, time of day, and campus events, into agent-based models could enhance the realism of simulations and capture the temporal dynamics of pedestrian behavior. By incorporating these variables, researchers can explore how external factors influence route choices and pedestrian movement patterns.
4. **Real-Time Data Collection:** Implementing real-time data collection methods, such as GPS tracking and mobile applications, could provide researchers with more accurate and detailed information on pedestrian behavior. By collecting data in real-time, researchers can capture spontaneous route choices and interactions, enhancing the validity of agent-based simulations.

5. **Scenario Analysis:** Conducting scenario analysis to explore the impact of different campus configurations, infrastructure changes, and policy interventions on pedestrian behavior could inform campus planning and design decisions. By simulating various scenarios, researchers can assess the effectiveness of different interventions and identify strategies to optimize pedestrian movement within the UPB campus.
6. **Multi-Agent Systems:** Extending the study to incorporate multi-agent systems and social network analysis could provide insights into the social interactions and collective behavior of pedestrians within the UPB campus. By modeling pedestrian interactions and group dynamics, researchers can explore how social networks influence route choices and pedestrian movement patterns.
7. **Collaborative Research:** Collaborating with urban planners, architects, and policymakers could facilitate the integration of pedestrian behavior research into campus planning and design processes. By working with stakeholders, researchers can translate empirical findings into actionable recommendations and promote sustainable and pedestrian-friendly environments within the UPB campus.

## List of References

- [1] M. A. ALFONZO, *To walk or not to walk? the hierarchy of walking needs*, Environment and Behavior, 37 (2005), pp. 808–836.
- [2] G. L. ALLEN, *Spatial abilities, cognitive maps, and wayfinding - bases for individual differences in spatial cognition and behavior*, in Wayfinding behavior: Cognitive mapping and other spatial processes, R. G. Golledge, ed., Johns Hopkins University Press, Baltimore, MD, 1999, pp. 46–80.
- [3] G. L. ALLEN AND R. G. GOLLEDGE, *Wayfinding in urban environments*, in Threats from car traffic to the quality of urban life, T. Garling and L. Steg, eds., Elsevier, Amsterdam, 2007, pp. 79–101.
- [4] C. U. AUTHORS, *Agent-based modeling*, Oct 2022.
- [5] S. P. AUTHORS, *Stochastic model*.
- [6] E. BONABEAU, *Agent-based modeling: Methods and techniques for simulating human systems*, Proceedings of the National Academy of Sciences, 99 (2002), pp. 7280–7287.
- [7] A. BORNIOLI, G. PARKHURST, AND P. L. MORGAN, *Psychological wellbeing benefits of simulated exposure to five urban settings: An experimental study from the pedestrian's perspective*, Journal of Transport & Health, 9 (2018), pp. 105–116.
- [8] D. G. BROWN, S. PAGE, R. RIOLO, M. ZELLNER, AND W. RAND, *Path dependence and the validation of agent-based spatial models of land use*, International Journal of Geographical Information Science, 19 (2005), pp. 153–174.
- [9] D. G. BROWN AND D. T. ROBINSON, *Effects of heterogeneity in residential preferences on an agent-based model of urban sprawl*, Ecology and Society, 11 (2006).
- [10] C. M. BUCHMANN, K. GROSSMANN, AND N. SCHWARZ, *How agent heterogeneity, model structure and input data determine the performance of an empirical abm - a*

- real-world case study on residential mobility*, Environmental Modelling & Software, 75 (2016), pp. 77–93.
- [11] M. CADWALLADER, *Cognitive distance in intraurban space*, in Environmental knowing, G. T. Moore and R. G. Golledge, eds., Dowden, Hutchinson & Ross, Stroudsburg, PA, 1976, pp. 316–324.
  - [12] T. CALINSKI AND J. HARABASZ, *A dendrite method for cluster analysis*, Communications in Statistics, 3 (1974), pp. 1–27.
  - [13] C. CASTELFRANCHI, *Through the agents' minds: Cognitive mediators of social action*, Mind & Society, 1 (2000), pp. 109–140.
  - [14] R. CONTE AND M. PAOLUCCI, *On agent-based modeling and computational social science*, Frontiers in Psychology, 5 (2014), pp. 1–9.
  - [15] M. W. D. CONTRIBUTORS, *Merriam-Webster's Collegiate Dictionary*, Merriam-Webster, 11th ed., 2003.
  - [16] O. CONTRIBUTORS, *Planet image, university of the philippines baguio*, retrieved from <https://planet.osm.org>. Accessed 10 May 2024.
  - [17] A. T. CROOKS, C. CASTLE, AND M. BATTY, *Key challenges in agent-based modelling for geo-spatial simulation*, Computers, Environment and Urban Systems, 32 (2008), pp. 417–430.
  - [18] C. D. DARKER, M. LARKIN, AND D. P. FRENCH, *An exploration of walking behaviour - an interpretative phenomenological approach*, Social Science & Medicine, 65 (2007), pp. 2172–2183.
  - [19] M. DE CERTEAU, *The practice of everyday life*, University of California Press, Berkeley, CA, 1984.
  - [20] K. DOVEY AND E. PAFKA, *The urban density assemblage: Modelling multiple measures*, Urban Design International, 19 (2014), pp. 66–76.

- [21] D. C. DUVIVES, W. DAAMEN, AND S. P. HOOGENDOORN, *State-of-the-art crowd motion simulation models*, Transportation Research Part C: Emerging Technologies, 37 (2013), pp. 193–209.
- [22] J. M. EPSTEIN, *Agent-based computational models and generative social science*, Complexity, 4 (1999), pp. 41–60.
- [23] R. A. EPSTEIN AND L. K. VASS, *Neural systems for landmark-based wayfinding in humans*, Philosophical Transactions of the Royal Society B: Biological Sciences, 369 (2014).
- [24] D. ESPOSITO, S. SANTORO, AND D. CAMARDA, *Agent-based analysis of urban spaces using space syntax and spatial cognition approaches: A case study in bari, italy*, Sustainability, 12 (2020), p. 4625.
- [25] J. EVANS AND P. JONES, *The walking interview: Methodology, mobility and place*, Applied Geography, 31 (2011), pp. 849–858.
- [26] B. S. EVERITT, S. LANDAU, M. LEESE, AND D. STAHL, *Cluster Analysis*, John Wiley & Sons, Chichester, UK, 5th ed., 2011.
- [27] A. FERDMAN, *Walking and its contribution to objective well-being*, Journal of Planning Education and Research, (2019).
- [28] L. E. FERNANDEZ, D. G. BROWN, R. W. MARANS, AND J. I. NASSAUER, *Characterizing location preferences in an exurban population: Implications for agent-based modeling*, Environment and Planning B: Planning and Design, 32 (2005), pp. 799–820.
- [29] G. FILOMENA, *g-filomena/pedsimcity-evaluation: Pedsimcity-empirical based agent-based model: Preparation and evaluation (v1.05)*, Zenodo, (2022).
- [30] ———, *g-filomena/pedsimcity: Pedsimcity-empirical based agent-based model (v1.11)*, Zenodo, (2022).

- [31] G. FILOMENA, L. KIRSCH, A. SCHWERING, AND J. A. VERSTEGEN, *Empirical characterisation of agents' spatial behaviour in pedestrian movement simulation*, Journal of Environmental Psychology, 82 (2022), p. 101807.
- [32] G. FILOMENA, E. MANLEY, AND J. A. VERSTEGEN, *Perception of urban subdivisions in pedestrian movement simulation*, PLoS One, 15 (2020), p. e0244099.
- [33] G. FILOMENA AND J. A. VERSTEGEN, *Modelling the effect of landmarks on pedestrian dynamics in urban environments*, Computers, Environment and Urban Systems, 86 (2021), p. 101573.
- [34] G. FILOMENA, J. A. VERSTEGEN, AND E. MANLEY, *A computational approach to 'the image of the city'*, Cities, 89 (2019), pp. 14–25.
- [35] J. C. FOLTÊTE AND A. PIOMBINI, *Deviations in pedestrian itineraries in urban areas: A method to assess the role of environmental factors*, Environment and Planning B: Planning and Design, 37 (2010), pp. 723–739.
- [36] A. K. FORMANN, *Die Latent-Class-Analyse: Einführung in die Theorie und Anwendung*, Beltz, Weinheim, 1984.
- [37] ——, *Die Latent-Class-Analyse: Einführung in die Theorie und Anwendung*, Beltz, Weinheim, 1984.
- [38] A. FORSYTH, M. HEARST, J. M. OAKES, AND K. H. SCHMITZ, *Design and destinations: Factors influencing walking and total physical activity*, Urban Studies, 45 (2008), pp. 1973–1996.
- [39] S. FRANKLIN AND A. GRAESSER, *Is it an agent, or just a program? a taxonomy of autonomous agents*, in Lecture Notes in Computer Science, vol. 1193, Springer, 1996, pp. 21–35.
- [40] T. GARLING, J. SAISA, A. BOOK, AND E. LINDBERG, *The spatiotemporal sequencing of everyday activities in the large-scale environment*, Journal of Environmental Psychology, 6 (1986), pp. 261–280.

- [41] B. GATERSLEBEN AND M. ANDREWS, *When walking in nature is not restorative - the role of prospect and refuge*, Health & Place, 20 (2013), pp. 91–101.
- [42] C. J. GIDLOW, M. V. JONES, G. HURST, D. MASTERSON, D. CLARK-CARTER, M. P. TARVAINEN, G. SMITH, AND M. NIEUWENHUIJSEN, *Where to put your best foot forward: Psycho-physiological responses to walking in natural and urban environments*, Journal of Environmental Psychology, 45 (2016), pp. 22–29.
- [43] R. G. GOLLEDGE, *Path selection and route preference in human navigation: A progress report*, in Spatial information theory A theoretical basis for GIS. COSIT 1995, A. U. Frank and W. Kuhn, eds., Springer, 1995, pp. 207–222.
- [44] ——, *Human wayfinding and cognitive maps*, in Wayfinding behavior: Cognitive mapping and other spatial processes, R. G. Golledge, ed., Johns Hopkins University Press, 1999, pp. 5–45.
- [45] Z. GUO AND B. P. Y. LOO, *Pedestrian environment and route choice: Evidence from new york city and hong kong*, Journal of Transport Geography, 28 (2013), pp. 124–136.
- [46] H. C. HAACKE, F. ENSSLE, D. HAASE, AND T. LAKES, *How to derive spatial agents: A mixed-method approach to model an elderly population with scarce data*, Population, Space and Place, (2022), p. e2551.
- [47] M. HAGHANI, *Empirical methods in pedestrian, crowd and evacuation dynamics: Part i. experimental methods and emerging topics*, Safety Science, 129 (2020), p. 104743.
- [48] ——, *Empirical methods in pedestrian, crowd and evacuation dynamics: Part ii. field methods and controversial topics*, Safety Science, 129 (2020), p. 104760.
- [49] M. HAGHANI AND M. SARVI, *How perception of peer behaviour influences escape decision making: The role of individual differences*, Journal of Environmental Psychology, 51 (2017), pp. 141–157.

- [50] J. HAN, M. KAMBER, AND J. PEI, *Cluster analysis: Basic concepts and methods*, in Data mining concepts and techniques, Morgan Kaufmann Publishers, Waltham, MA, 3rd ed., 2012, pp. 443–495.
- [51] S. L. HANDY, *Urban form and pedestrian choices: Study of austin neighborhoods*, Transportation Research Record: Journal of the Transportation Research Board, 1552 (1996), pp. 135–144.
- [52] M. K. B. E. HENDERSON, *Top-down vs. bottom-up research*, Nov 2022.
- [53] A. HEPPENSTALL, N. MALLESON, AND A. T. CROOKS, "space, the final frontier": How good are agent-based models at simulating individuals and space in cities?, *Systems*, 4 (2016), p. 9.
- [54] M. R. HILL, *Walking, crossing streets and choosing pedestrian routes: A survey of recent insights from the social/Behavioral sciences*, University of Nebraska, Lincoln, NE, 1984.
- [55] C. HOLSCHER, T. TENBRINK, AND J. M. WIENER, Would you follow your own route description? cognitive strategies in urban route planning, *Cognition*, 121 (2011), pp. 228–247.
- [56] W. JAGER, Enhancing the realism of simulation (eros): On implementing and developing psychological theory in social simulation, *The Journal of Artificial Societies and Social Simulation*, 20 (2017), p. 14.
- [57] W. JAGER AND A. ERNST, Introduction of the special issue “social simulation in environmental psychology”, *Journal of Environmental Psychology*, 52 (2017), pp. 114–118.
- [58] O. JAN, A. J. HOROWITZ, AND Z.-R. PENG, Using global positioning system data to understand variations in path choice, *Transportation Research Record*, (2000), pp. 37–44.
- [59] M. A. JANSSEN AND E. OSTROM, Empirically based, agent-based models, *Ecology and Society*, 11 (2006), p. 37.

- [60] B. JIANG AND T. JIA, *Agent-based simulation of human movement shaped by the underlying street structure*, International Journal of Geographical Information Science, 25 (2011), pp. 51–64.
- [61] Y. KATO AND Y. TAKEUCHI, *Individual differences in wayfinding strategies*, Journal of Environmental Psychology, 23 (2003), pp. 171–188.
- [62] J. KERRIDGE, J. HINE, AND M. WIGAN, *Agent-based modelling of pedestrian movements: The questions that need to be asked and answered*, Environment and Planning B: Planning and Design, 28 (2001), pp. 327–341.
- [63] H. KIM, *Walking distance, route choice, and activities while walking: A record of following pedestrians from transit stations in the san francisco bay area*, Urban Design International, 20 (2015), pp. 144–157.
- [64] P.-L. KOH AND Y.-D. WONG, *Influence of infrastructural compatibility factors on walking and cycling route choices*, Journal of Environmental Psychology, 36 (2013), pp. 202–213.
- [65] S. P. LLOYD, *Least squares quantization in pcm*, IEEE Transactions on Information Theory, 28 (1982), pp. 129–137.
- [66] H. LORIMER, *Walking: New forms and spaces for studies of pedestrianism*, in Geographies of Mobilities: Practices, Spaces, Subjects, T. Cresswell and P. Merriman, eds., Ashgate, 2010, pp. 19–33.
- [67] S. LUKE, *Multiagent simulation and the mason version 21 library*, tech. rep., Department of Computer Science, George Mason University, Fairfax, VA, 2022.
- [68] K. LYNCH, *The image of the city*, MIT Press, Cambridge, MA, 1960.
- [69] H. A. MALLOT AND K. BASTEN, *Embodied spatial cognition: Biological and artificial systems*, Image and Vision Computing, 27 (2009), pp. 1658–1670.
- [70] O. MASON, J. SARMA, J. D. SIDAWAY, A. BONNETT, P. HUBBARD, G. JAMIL, J. MIDDLETON, M. O’NEILL, J. RIDING, AND M. ROSE, *Interventions in walking methods in political geography*, Political Geography, 106 (2023), p. 102937.

- [71] J. MIDDLETON, *Walking in the city: The geographies of everyday pedestrian practices*, Geography Compass, 5 (2011), pp. 90–105.
- [72] ——, *The socialities of everyday urban walking and the 'right to the city'*, Urban Studies, 55 (2018), pp. 296–315.
- [73] G. W. MILLIGAN AND M. C. COOPER, *An examination of procedures for determining the number of clusters in a data set*, Psychometrika, 50 (1985), pp. 159–179.
- [74] S. MÜNZER, B. C. FEHRINGER, AND T. KÜHL, *Validation of a 3-factor structure of spatial strategies and relations to possession and usage of navigational aids*, Journal of Environmental Psychology, 47 (2016), pp. 66–78.
- [75] T. MURALEETHARAN AND T. HAGIWARA, *Overall level of service of urban walking environment and its influence on pedestrian route choice behavior*, Transportation Research Record: Journal of the Transportation Research Board, 2002 (2007), pp. 7–17.
- [76] C. G. OF BAGUIO WEBSITE CONTRIBUTORS, *City government of baguio*. <https://new.baguio.gov.ph/home>. Accessed: 2024-05-19.
- [77] I. OMER AND N. KAPLAN, *Using space syntax and agent-based approaches for modeling pedestrian volume at the urban scale*, Computers, Environment and Urban Systems, 64 (2017), pp. 57–67.
- [78] L. D. ORTIZ, A. POLHEMUS, A. KEOGH, N. SUTTON, W. REMMELE, C. HANSEN, F. KLUGE, B. SHARRACK, C. BECKER, T. TROOSTERS, W. MAETZLER, L. ROCHESTER, A. FREI, M. PUHAN, AND J. GARCIA-AYMERICH, *What is walking? a systematic review and meta-ethnography*, European Respiratory Journal, 60 (2022).
- [79] N. OWEN, N. HUMPEL, E. LESLIE, A. BAUMAN, AND J. F. SALLIS, *Understanding environmental influences on walking*, American Journal of Preventive Medicine, 27 (2004), pp. 67–76.

- [80] F. G. C. PAAS, *Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach*, Journal of Educational Psychology, 84 (1992), pp. 429–434.
- [81] E. PAPADIMITRIOU, G. YANNIS, AND J. GOLIAS, *A critical assessment of pedestrian behaviour models*, Transportation Research Part F: Traffic Psychology and Behaviour, 12 (2009), pp. 242–255.
- [82] J. PUCHER AND R. BUEHLER, *Walking and cycling for healthy cities*, Built Environment, 36 (2010), pp. 319–414.
- [83] S. F. RAILSBACK AND V. GRIMM, *Agent-based and Individual-based Modelling: A Practical Introduction*, Princeton University Press, Princeton, NJ, second ed., 2019.
- [84] D. A. RODRÍGUEZ, L. MERLIN, C. G. PRATO, T. L. CONWAY, D. COHEN, J. P. ELDER, K. R. EVENSON, T. L. MCKENZIE, J. L. PICKREL, AND S. VEBLEN-MORTENSON, *Influence of the built environment on pedestrian route choices of adolescent girls*, Environment and Behavior, 47 (2015), pp. 359–394.
- [85] J. ROE AND P. ASPINALL, *The restorative benefits of walking in urban and rural settings in adults with good and poor mental health*, Health & Place, 17 (2011), pp. 103–113.
- [86] M. D. ROUNSEVELL, D. T. ROBINSON, AND D. MURRAY-RUST, *From actors to agents in socio-ecological systems models*, Philosophical Transactions of the Royal Society B: Biological Sciences, 367 (2012), pp. 259–269.
- [87] P. J. ROUSSEEUW, *Silhouettes: A graphical aid to the interpretation and validation of cluster analysis*, Journal of Computational and Applied Mathematics, 20 (1987), pp. 53–65.
- [88] C. SARKAR, C. WEBSTER, M. PRYOR, D. TANG, S. MELBOURNE, X. ZHANG, AND L. JIANZHENG, *Exploring associations between urban green, street design and walking: Results from the greater london boroughs*, Landscape and Urban Planning, 143 (2015), pp. 112–125.

- [89] F. SHATU, T. YIGITCANLAR, AND J. BUNKER, *Shortest path distance vs. least directional change: Empirical testing of space syntax and geographic theories concerning pedestrian route choice behaviour*, Journal of Transport Geography, 74 (2019), pp. 37–52.
- [90] A. W. SIEGEL AND S. H. WHITE, *The development of spatial representations of large-scale environments*, Advances in Child Development and Behavior, 10 (1975), pp. 9–55.
- [91] H. J. SPIERS AND E. A. MAGUIRE, *The dynamic nature of cognition during wayfinding*, Journal of Environmental Psychology, 28 (2008), pp. 232–249.
- [92] D. STEINLEY, *Standardizing variables in k-means clustering*, in Classification, Clustering, and Data Mining Applications, Springer, 2004, pp. 53–60.
- [93] E. STRATFORD, G. WAITT, AND T. HARADA, *Walking city streets: Spatial qualities, spatial justice, and democratising impulses*, Transactions of the Institute of British Geographers, 45 (2020), pp. 123–138.
- [94] A. STRUYF, M. HUBERT, AND P. ROUSSEEUW, *Clustering in an object-oriented environment*, Journal of Statistical Software, 1 (1996), pp. 1–30.
- [95] K. SULLIVAN, M. COLETTI, AND S. LUKE, *Geomason: Geospatial support for mason*, tech. rep., Department of Computer Science, George Mason University, Fairfax, VA, 2010.
- [96] M. TIGHT, P. TIMMS, D. BANISTER, J. BOWMAKER, J. COPAS, A. DAY, D. DRINKWATER, M. GIVONI, A. GÜHNEMANN, M. LAWLER, J. MACMILLEN, A. MILES, N. MOORE, R. NEWTON, D. NGODUY, M. ORMEROD, M. O’SULLIVAN, AND D. WATLING, *Visions for a walking and cycling focussed urban transport system*, Journal of Transport Geography, 19 (2011), pp. 1580–1589.
- [97] P. M. TORRENS, *Moving agent pedestrians through space and time*, Annals of the Association of American Geographers, 102 (2012), pp. 35–66.
- [98] ——, *Computational streetscapes*, Computation, 4 (2016), pp. 1–38.

- [99] J. M. WIENER AND H. A. MALLOT, '*fine-to-coarse*' route planning and navigation in regionalized environments, *Spatial Cognition and Computation*, 3 (2003), pp. 331–358.
- [100] F. WILCOXON, *Individual comparisons by ranking methods*, Biometric Bulletin, 1 (1945), p. 80.
- [101] A. WILLIS, N. GJERSOE, C. HAVARD, J. KERRIDGE, AND R. KUKLA, *Human movement behaviour in urban spaces: Implications for the design and modelling of effective pedestrian environments*, *Environment and Planning B: Planning and Design*, 31 (2004), pp. 805–828.
- [102] S. YANG, T. LI, X. GONG, B. PENG, AND J. HU, *A review on crowd simulation and modeling*, *Graphical Models*, 111 (2020), p. Article 101081.
- [103] Y. YANG AND A. V. DIEZ-ROUX, *Walking distance by trip purpose and population subgroups*, *American Journal of Preventive Medicine*, 43 (2012), pp. 11–19.
- [104] J. ZACHARIAS, *Pedestrian behavior and perception in urban walking environments*, *Journal of Planning Literature*, 16 (2001), pp. 3–18.
- [105] S. ZHU AND D. LEVINSON, *Do people use the shortest path? an empirical test of wardrop's first principle*, *PLoS One*, 10 (2015), p. Article e0134322.

# **Appendix A**

## **Questionnaire for Pedestrian Movement Analysis**

Walking Behavior in University of the Philippines Baguio

Using QUALTRICS Survey Tool

### **Survey Flow**

- Standard: Section 0 (2 Questions)
- Standard: Section I (4 Questions)
- Standard: Section II (5 Questions)
- Standard: Section III (5 Questions)
- Standard: Section IV (5 Questions)
- Standard: Section V (2 Questions)
- Standard: Section VII (2 Questions)

Start of Block: Section 0

Q0 Thank you for helping this study. The study is entitled “Agent’s Spatial Behavior in Pedestrian Movement: An Empirical Study Using GeoMASON in University of the Philippines Baguio” conducted by Rousseau Nilo Maamor as part of his BS Computer Science Special Problem. This study is aimed at understanding and simulating peoples’ specific route choices strategies (distance, landmark, barrier) when navigating through urban environment by foot. The University of the Philippines Baguio (UPB) is used as the urban environment and case study.

Please remember that there are no right or wrong answers. The questionnaire is structured in different sections, but you are free to leave the questionnaire at any time.

You can also start it and complete it at a later time. In addition, only Section I, II, III, and IV are required. There are 15 required questions and 5 optional questions in this survey. All in all, this survey will only take two to five minutes of your valuable time while also helping the researcher graduate.

If you have questions or doubts about your participation, you can email rgmaamor@up.edu.ph or message him on his Facebook Messenger.

Please consider sharing the survey.

Thank you for your participation!

Page Break

### **S0H0 Timing**

First Click (1)

Last Click (2)

Page Submit (3)

Click Count (4)

End of Block: Section 0

Start of Block: Section I

### **S1Q1 REQUIRED**

How would you describe your gender?

- Male (1)
- Female (2)
- Non-binary / third gender (3)
- Prefer not to say (4)

S2Q2 REQUIRED How old are you (in years)? \_\_\_

S3Q3 REQUIRED Which of the following applies to you?

- I study or I used to study in UPB (1)
- I work or I used to work in UPB (2)

### **S1H0 Timing**

First Click (1)

Last Click (2)

Page Submit (3)

Click Count (4)

End of Block: Section I

Start of Block: Section II

S2Q1 REQUIRED Are you familiar with the following two locations, the UPB Main Entrance and Kolehiyo ng Agham (KA) Building?

- Yes (1)
- No (2)

S2Q2 REQUIRED (Distance — extent or total amount of movement between an origin and a destination.) Do you consider distance when going from UPB Main Entrance to KA Building?

- Yes (1)
- No (2)

S2Q3 REQUIRED (Landmark — a building in UPB that is easily seen and recognized from a distance, especially one that enables you to establish your location. Example: Main Entrance Guard House, Iskolar ng Bayan building)

Do you consider landmark when going from UPB Main Entrance to KA Building?  Yes (1)

- No (2)

S2Q4 REQUIRED (Barrier — thing that prevents movement or access. Example: KA Quad plant area, construction area near KA Quad.)

Do you consider barrier when going from UPB Main Entrance to KA Building?  Yes (1)

- No (2)

S2H0 Timing First Click (1)

Last Click (2)

Page Submit (3)

Click Count (4)

End of Block: Section II

Start of Block: Section III

S3Q1 REQUIRED Are you familiar with the following two locations, the UPB Main Entrance and College of Social Science (CSS) Building?

- Yes (1)

- No (2)

S3Q2 REQUIRED (Distance — extent or total amount of movement between an origin and a destination.) Do you consider distance when going from UPB Main Entrance to CSS Building?

- Yes (1)

- No (2)

S3Q3 REQUIRED (Landmark — a building in UPB that is easily seen and recognized from a distance, especially one that enables someone to establish their location. Example: Cafeteria, Alumni building) Do you consider landmark when going from UPB Main Entrance to CSS Building?

- Yes (1)

- No (2)

S3Q4 REQUIRED (Barrier — thing that prevents movement or access. Example: Freedom Park plant area, “Limbo Rack” stairs to CSS building) Do you consider barrier when going from UPB Main Entrance to CSS Building?

- Yes (1)

- No (2)

S3H0 Timing

First Click (1)

Last Click (2)

Page Submit (3)

Click Count (4)

End of Block: Section III

Start of Block: Section IV

S4Q1 REQUIRED Are you familiar with the following two locations, the UPB Main Entrance and College of Arts and Communication (CAC) Building?

- Yes (1)

- No (2)

S4Q2 REQUIRED (Distance — extent or total amount of movement between an origin and a destination.) Do you consider distance when going from UPB Main Entrance to CAC Building?

- Yes (1)
- No (2)

S4Q3 REQUIRED (Landmark — a building in UPB that is easily seen and recognized from a distance, especially one that enables someone to establish their location. Example: Teatro Amianan, UPB Library.) Do you consider landmark when going from UPB Main Entrance to CAC Building?

- Yes (1)
- No (2)

S4Q4 REQUIRED (Barrier — thing that prevents movement or access. Example: Construction area near CAC Dap-ay, plant area near library.) Do you consider barrier when going from UPB Main Entrance to CAC Building?

- Yes (1)
- No (2)

S4H0 Timing

First Click (1)

Last Click (2)

Page Submit (3)

Click Count (4)

End of Block: Section IV

Start of Block: Section V

S5Q1 OPTIONAL How important are the following properties of a route if you have to reach a certain place by foot? Not at all important (1) Slightly important (2) Moderately important (3) Very important (4) Extremely important (5) The route traverses or extends along green areas (1) o o o o o The route crosses safe areas (e.g. good lighting, presence of other people, the area is known) (2) o o o o o The route crosses interesting or useful landmarks (3) o o o o o

S5H0 Timing

First Click (1)

Last Click (2)

Page Submit (3)

Click Count (4)

End of Block: Section V

Start of Block: Section VII

S7 Captcha Verification

S7H0 Timing

First Click (1)

Last Click (2)

Page Submit (3)

Click Count (4)

End of Block: Section VII

# **Appendix B**

## **Questionnaire Data for Pedestrian Movement Analysis**

id	startdate	datestamp	duration	startlanguage	lastpage	PD1	PD2	PD3[SQ001]	PD3[SQ002]
1	2024-04-15 10:20:29	2024-04-15 10:23:31	180	EN	20	A1	22	Y	Y
2	2024-04-15 10:32:35	2024-04-15 10:33:49	73	EN	20	A4	21	Y	
3	2024-04-15 10:36:02	2024-04-15 10:37:00	57	EN	20	A1	22	Y	
4	2024-04-15 10:39:48	2024-04-15 10:42:29	158	EN	20	A1	22	Y	
5	2024-04-15 10:48:32	2024-04-15 10:49:40	66	EN	20	A1	22	Y	
6	2024-04-15 11:59:20	2024-04-15 12:03:04	222	EN	20	A1	22	Y	
7	2024-04-15 12:21:41	2024-04-15 12:23:07	84	EN	20	A1	22	Y	
8	2024-04-15 12:32:00	2024-04-15 12:34:28	146	EN	20	A2	20	Y	
9	2024-04-15 12:47:53	2024-04-15 12:51:29	215	EN	20	A2	22	Y	
10	2024-04-15 12:54:37	2024-04-15 12:57:32	174	EN	20	A2	26	Y	
11	2024-04-15 12:52:25	2024-04-15 12:58:31	365	EN	20	A2	20	Y	
12	2024-04-15 12:58:02	2024-04-15 13:03:58	355	EN	20	A1	20	Y	
13	2024-04-15 13:01:58	2024-04-15 13:04:10	132	EN	20	A1	19	Y	
14	2024-04-15 13:01:31	2024-04-15 13:05:03	212	EN	20	A2	20	Y	
15	2024-04-15 13:01:38	2024-04-15 13:06:53	314	EN	20	A1	18	Y	
16	2024-04-15 13:03:02	2024-04-15 13:15:07	724	EN	20	A1	22	Y	
17	2024-04-15 13:24:05	2024-04-15 13:28:44	278	EN	20	A1	22	Y	
18	2024-04-15 13:31:46	2024-04-15 13:36:44	297	EN	20	A2	21	Y	
19	2024-04-15 13:38:25	2024-04-15 13:41:52	207	EN	20	A2	22	Y	
20	2024-04-15 13:48:29	2024-04-15 13:53:34	304	EN	20	A2	23	Y	
21	2024-04-15 13:58:15	2024-04-15 14:00:46	150	EN	20	A1	21	Y	
22	2024-04-15 13:59:02	2024-04-15 14:03:15	252	EN	20	A4	24	Y	
23	2024-04-15 13:59:47	2024-04-15 14:05:18	330	EN	20	A1	24	Y	
24	2024-04-15 13:59:54	2024-04-15 14:05:44	349	EN	20	A1	22	Y	
25	2024-04-15 13:09:26	2024-04-15 14:06:51	3443	EN	20	A2	20	Y	





PD3 SQ003]	VD000a	VD000	VD001	VD002	VD100a	VD100	VD101	VD102	VD200a
Y	A1		A1	Y	A1	A1	A1	A1	Y
Y	A1	A1	A1	Y	A1	A1	A1	A1	Y
Y	A1	A1	A1	Y	A1	A1	A1	A1	Y
Y	A1	A2	A1	N	A1	A2	A1	A1	Y
Y	A2	A1	A2	Y	A2	A1	A1	A1	Y
Y	A2	A2	A2	Y	A2	A2	A2	A2	N
N	A1	A2	A1	N	A2	A2	A2	A2	N
Y	A2	A1	A2	Y	A1	A1	A2	A2	Y
Y	A1	A2	A1	Y	A1	A1	A2	A2	Y
Y	A1	A1	A2	A1	Y	A1	A1	A1	Y
Y	A1	A1	A1	A2	Y	A1	A1	A1	Y
Y	A2	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A1	A1	A1	Y	A1	A1	A1	Y
Y	A2	A2	A2	A2	Y	A1	A1	A1	Y
Y	A1	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A1	A2	A1	Y	A1	A2	A1	Y
Y	A2	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A1	A2	A1	Y	A1	A2	A1	Y
Y	A1	A1	A1	A2	Y	A1	A1	A1	Y
Y	A2	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A2	A2	A2	Y	A1	A2	A2	Y
Y	A1	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A1	A2	A1	Y	A1	A2	A1	Y
Y	A1	A1	A1	A1	Y	A1	A1	A1	Y
Y	A1	A1	A2	A1	Y	A1	A2	A1	Y
Y	A1	A1	A1	A1	Y	A1	A1	A1	Y

	Y	A2	A2	A1	Y	A1	A2	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	A2	Y	A1	A1	Y
	Y	A1	A1	A2	A1	Y	A1	A1	Y
	Y	A2	A2	A2	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A2	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A2	A1	Y	A1	A2	A1	Y
	Y	A2	A1	A1	Y	A1	A1	A2	Y
	Y	A1	A1	A2	A1	Y	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A2	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A2	Y
	Y	A1	A2	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A2	A1	Y	A1	A1	Y
	Y	A1	A1	A1	A2	Y	A1	A2	Y
	Y	A2	A1	A1	Y	A1	A1	A1	Y
	Y	A2	A2	A1	Y	A1	A2	A1	Y
	Y	A1	A2	A1	Y	A1	A2	A1	Y
	Y	A1	A1	A2	A1	N	A1	A2	A1
	Y	A1	A1	A1	A2	Y	A1	A1	A1
	Y	A2	A2	A2	A2	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A1	A2	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A2	A2	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A2	A2	A1	A1	Y	A1	A1	A1
	Y	A1	A2	A1	A1	Y	A1	A1	A1
	Y	A1	A1	A2	A1	Y	A1	A1	A1
	Y	A1	A1	A1	A2	Y	A1	A1	A1
	Y	A1	A1	A1	A1	Y	A1	A1	A1
	Y	A2	A2	A1	A1	Y	A1	A1	A1
	Y	A1	A2	A1	A1	Y	A1	A1	A1
	Y	A2	A2	A1	A1	Y	A1	A1	A1

	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A2	A1	Y	A1	A2	A2	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A2	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A2	Y
	Y	A2	A1	A1	Y	A1	A1	A1	Y
	Y	A2	A2	A1	Y	A1	A2	A1	Y
	Y	A2	A1	A1	Y	A1	A1	A1	Y
	Y	A2	A1	A1	Y	A1	A1	A1	Y
	Y	A2	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A1	Y
	Y	A1	A1	A1	Y	A1	A1	A2	Y
	Y	A2	A1	A2	Y	A1	A1	A1	Y

VD200	VD201	VD202	SP4[SQ003]	SP4[SQ004]	SP4[SQ007]	WB1	WB2[SQ001]	WB2[SQ002]	WB3[SQ001]
A1	A1	A1							
A1	A1	A1							
A1	A1	A1							
A2	A1	A2	2	4	5	1	4	3	4
A1	A1	A2	5	4	5	7	5	4	5
A2	A2	A2	4	5	2	7	1	5	1
A2	A2	A2							
A2	A1	A1	3	2	4	3	1	5	1
A1	A1	A1	4	5	4	4	4	5	2
A1	A1	A1	4	5	3	4	1	5	2
A1	A1	A1	4	5	2	5	5	5	1
A1	A1	A1	3	3	5	5	1	5	4
A1	A1	A1	3	3	2	5	2	3	3
A1	A1	A1	4	5	4	4	3	5	2
A1	A1	A1	4	5	5	4	1	5	
A1	A1	A1	4	4	4	4	1	4	2
A1	A1	A1	5	5	3	5	1	5	3
A2	A2	A1	5	4	3	5	1	5	2
A1	A1	A1	2	2	3	5	5	5	3
A2	A1	A2	4	5	2	5		5	1
A2	A1	A1	5	5	3				1
A1	A1	A1	4	4	3	3	3	5	3
A1	A2	A2	2	4	2	4	1	5	2
A1	A1	A1	4	5	3	5	2	5	1
A1	A2	A1	4	3	4				

A1	A2	A1	2	4	3	5	5	5	3
A1	A1	A1	4	4	0	1	1	1	1
A2	A1	A1	4	5	2	1	4	1	1
A1	A2	A1	4	5	4	4	5	5	1
A1	A1	A2	3	3	2	4	1	5	1
A1	A1	A1	4	4	1	3	5	5	2
A1	A1	A1	3	5	2	7	5	5	1
A1	A2	A1	5	4	3	5	4	4	4
A1	A1	A2	3	4	2	4	4	4	3
A1	A1	A1	4	4	4	4	1	4	1
A1	A1	A1	5	5	5	1	5	5	5
A1	A2	A1	3	4	2	4	1	5	1
A1	A1	A1	4	4	3	4	1	5	3
A2	A1	A2	5	4	4	5	5	5	1
A1	A1	A1	4	4	5	3	5	5	3
A1	A2	A1	2	4	4	5	1	5	3
A1	A2	A2							
A1	A1	A1	4	3	2	4	1	5	4
A2	A2	A2	3	4	1	0	1	3	1
A1	A1	A1	3	4	1	2	2	2	1
A1	A1	A1	5	4	5		5	5	1
A1	A1	A1	4	3	4	1	5	5	3
A1	A1	A1	4	4	3	6	1	4	3
A1	A1	A1	3	4	3	6	1	5	3
A1	A1	A1	5	5	4	5	5	5	3
A1	A1	A1	4	3	2	4	1	5	1

A1	A1	A1	5	5	4	4	5	5	2
A1	A2	A1	5	5	2	5	1	4	2
A1	A1	A1	5	1	5	5	5	5	1
A1	A1	A1	5	5	4		5	5	1
A2	A1	A2	2	3	2	4	1	4	1
A1	A1	A1	3	5	3	4	1	5	1
A1	A1	A1	5	5	4	4	2	5	1
A2	A1	A1	4	1	1	4	4	4	3
A1	A1	A1	5	5	4	4	1	5	1
A1	A1	A1	5	5	1	4	1	5	1
A1	A1	A1	5	5	2	2	2	3	1
A1	A1	A1	4	5	4	5	1	3	4
A1	A1	A1	4	4	3	5		4	3
A1	A1	A1	5	5	4	5	4	5	4
A1	A1	A1	4	5	4	5	4	4	1
A1	A1	A1							
A1	A1	A1	4	5	3	5	5		1
A1	A1	A2	5	5	4	4		5	1
A1	A1	A1	3	4	2	4		4	2
A1	A1	A1	5	5	5	5	1	5	1

WB3[SQ002]	WB4[SQ001]	WB4[SQ002]	WB4[SQ003]	WB4[SQ004]
3	4	5	4	5
4	4	4	5	4
1	5	5	4	5
1	5	3	4	4
2	2	4	4	4
2	4	4	4	4
1	3	2	4	4
1	2	4	5	5
1	3	4	4	4
2	4	2	4	3
	5	5	5	5
1	4	4	4	4
1	4	3	4	2
2	4	4	3	4
2	4	4	3	3
1	4	4	3	2
	4			
1	5	5	2	2
2	5	5	4	5
1	4	2	4	4

2	3	3	3	4	4
1	2	2	2	5	4
1	2	2	2	4	5
1	3	2	4	4	2
1	5	5	4	4	5
1	5	5	5	5	5
1	4	3	4	4	5
4	5	5	4	4	4
2	4	4	3	3	4
1	4	2	4	4	5
1	4	3	5	5	5
1	5	4	2	2	2
3	5	5	5	5	5
1	4		5	5	2
3	4	5	5	4	4
3	4	4	4	4	4
1	4	4	5	3	3
1	2	1	2	1	1
1	4	4	4	2	2
1	4	4	5	5	5
1	4	4	3	3	3
1	5	4	4	4	4
1	5	5	4	4	4
2	2	1	3	3	3
1	5	4	3	3	4

1	3		5	4
2	3	3	4	4
1	5	5	5	5
1	4	4	5	4
1	5	5	5	5
2	5	5	4	4
1	4	3	4	3
2	4	4	4	3
1	5	5	5	5
1	5	5	5	3
1	3	3	3	3
3	2	4	4	4
2	4	4	4	4
3	2	3	4	4
1	5	5	4	4
1	4	4	4	4
1	4	2	3	3
1	2	1	4	4
1	5	4	5	4

# **Appendix C**

## **Configuration 3 Clusters, Configuration 1, and Configuration 2 Statistics**

*Appendix C. Configuration 3 Clusters, Configuration 1, and Configuration 2 Statistics 150*

	Mean Distance	StDev Distance	Mean Landmarks	StDev Landmarks	Mean Barriers	StDev Barriers	Portion
CLUSTER 1	0.381886045	0.010112237	0.090023477	0.012539046	0.415362886	0.012136054	0.647058824
CLUSTER 2	0.275572818	0.029468854	0.248162091	0.03998972	0.278837909	0.033221477	0.161764706
CLUSTER 3	0.385320385	0.026355518	0.241281	0.091232821	0.411103615	0.031394518	0.191176471
CONFIGURATION 2	0.365344882	0.043587608	0.144521603	0.085795794	0.392463691	0.054472434	1
CONFIGURATION 1	0.365344882	0.043587608	0.144521603	0.085795794	0.392463691	0.054472434	1

**Table C.1:** Configuration 3 Clusters, Configuration 1, and Configuration 2 Statistics

# Appendix D

## Agent Route Choice Statistics

	Distance	Landmarks	Barriers	Natural Barriers	Severing Barriers		Distance	Landmarks	Barriers	Natural Barriers	Severing Barriers
1	0.762	0.168	0.832	0	0	35	0.722	0.233	0.767	4	4
2	0.762	0.168	0.832	0	0	36	0.762	0.168	0.832	5	5
3	0.762	0.168	0.832	0	0	37	0.5	0.5	0.5	3	3
4	0.539	0.437	0.563	2	4.5	38	0.762	0.168	0.832	4	3.5
5	0.673	0.265	0.735	5	4.5	39	0.605	0.329	0.671	5	4
6	0.5	0.5	0.5	4	3.5	40	0.762	0.168	0.832	4	4
7	0.5	0.5	0.5	0	0	41	0.5	0.5	0.5	2	4
8	0.676	0.259	0.741	3	3	42	0.5	0.5	0.5	0	0
9	0.722	0.233	0.767	4	4.5	43	0.744	0.195	0.805	4	2.5
10	0.744	0.195	0.805	4	4	44	0.612	0.366	0.634	3	2.5
11	0.762	0.168	0.832	4	3.5	45	0.762	0.168	0.832	3	2.5
12	0.762	0.168	0.832	3	4	46	0.65	0.303	0.697	5	4.5
13	0.762	0.168	0.832	3	2.5	47	0.762	0.168	0.832	4	3.5
14	0.762	0.168	0.832	4	4.5	48	0.722	0.233	0.767	4	3.5
15	0.762	0.168	0.832	4	5	49	0.744	0.195	0.805	3	3.5
16	0.722	0.233	0.767	4	4	50	0.762	0.168	0.832	5	5
17	0.762	0.168	0.832	5	4	51	0.722	0.233	0.767	4	2.5
18	0.5	0.5	0.5	5	3.5	52	0.762	0.168	0.832	5	4.5
19	0.762	0.168	0.832	2	2.5	53	0.5	0.5	0.5	5	3.5
20	0.584	0.366	0.634	4	3.5	54	0.762	0.168	0.832	5	3
21	0.762	0.168	0.832	5	4	55	0.762	0.168	0.832	5	5
22	0.762	0.168	0.832	4	3.5	56	0.584	0.366	0.634	2	2.5
23	0.5	0.5	0.5	2	3	57	0.762	0.168	0.832	3	4
24	0.762	0.168	0.832	4	4	58	0.762	0.168	0.832	5	4.5
25	0.5	0.5	0.5	4	3.5	59	0.61	0.367	0.633	4	1
26	0.5	0.5	0.5	2	3.5	60	0.762	0.168	0.832	5	4.5
27	0.762	0.168	0.832	4	4	61	0.762	0.168	0.832	5	3
28	0.744	0.195	0.805	4	5	62	0.762	0.168	0.832	5	5
29	0.612	0.366	0.634	4	4.5	63	0.762	0.168	0.832	4	4.5
30	0.584	0.366	0.634	3	2.5	64	0.762	0.168	0.832	4	3.5
31	0.762	0.168	0.832	4	2.5	65	0.762	0.168	0.832	5	4.5
32	0.762	0.168	0.832	3	3.5	66	0.762	0.168	0.832	4	4.5
33	0.5	0.5	0.5	5	3.5	67	0.762	0.168	0.832	0	0
34	0.624	0.302	0.698	3	3	68	0.673	0.265	0.735	5	4.5

**Table D.1:** Variable Statistics per Agents

## Appendix E

## Agent Volume Per Street Segment Tables

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	39	11	10	50	64
1	0	0	0	0	0
2	0	0	0	0	0
3	34	11	16	62	24
4	0	0	0	0	0
5	2	1	2	6	12
6	0	0	0	0	0
7	0	0	0	0	29
8	0	0	0	0	0
9	0	0	0	0	29
10	0	0	0	0	0
11	0	0	0	0	0
12	67	21	24	113	119
13	67	21	24	113	119
14	65	20	22	107	107
15	31	9	6	45	83
16	0	0	0	0	0
17	2	1	2	6	12
18	0	0	0	0	0
19	0	0	0	0	0
20	34	11	16	62	24
21	2	1	2	6	10
22	71	20	23	127	54
23	155	35	43	233	231
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	2
27	36	8	4	58	42
28	0	0	0	0	2
29	153	34	41	227	221
30	69	19	21	121	44
31	84	15	20	106	177
32	0	0	0	0	2
33	2	1	2	6	14
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	29
38	0	0	0	0	29
39	0	0	0	0	29
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.1:** Run 0: Agent Volumes Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	48	12	4	54	75
1	0	0	0	0	0
2	0	0	0	0	0
3	38	10	14	65	31
4	0	0	0	0	0
5	4	2	5	11	18
6	0	0	0	0	0
7	0	0	0	0	22
8	0	0	0	0	0
9	0	0	0	0	22
10	0	0	0	0	0
11	0	0	0	0	0
12	82	21	27	130	137
13	82	21	27	130	137
14	78	19	22	119	119
15	40	9	8	54	88
16	0	0	0	0	0
17	4	2	5	11	18
18	0	0	0	0	0
19	0	0	0	0	0
20	38	10	14	65	31
21	4	2	5	11	17
22	67	16	36	120	61
23	142	36	43	221	220
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	1
27	32	9	19	70	48
28	0	0	0	0	1
29	138	34	38	210	203
30	63	14	31	109	44
31	75	20	7	101	159
32	0	0	0	0	1
33	4	2	5	11	19
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	22
38	0	0	0	0	22
39	0	0	0	0	22
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.2: Run 1: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	42	9	10	39	78
1	0	0	0	0	0
2	0	0	0	0	0
3	37	7	12	56	32
4	0	0	0	0	0
5	5	4	6	13	13
6	0	0	0	0	0
7	0	0	0	0	17
8	0	0	0	0	0
9	0	0	0	0	17
10	0	0	0	0	0
11	0	0	0	0	0
12	80	19	28	125	125
13	80	19	28	125	125
14	75	15	22	112	112
15	38	8	10	56	80
16	0	0	0	0	0
17	5	4	6	13	13
18	0	0	0	0	0
19	0	0	0	0	0
20	37	7	12	56	32
21	5	4	6	13	10
22	72	15	27	136	58
23	145	40	43	228	225
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	3
27	30	11	13	76	34
28	0	0	0	0	3
29	140	36	37	215	215
30	67	11	21	123	48
31	73	25	16	92	167
32	0	0	0	0	3
33	5	4	6	13	16
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	17
38	0	0	0	0	17
39	0	0	0	0	17
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.3: Run 2: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	33	9	5	44	54
1	0	0	0	0	0
2	0	0	0	0	0
3	36	13	7	62	31
4	0	0	0	0	0
5	2	1	8	9	21
6	0	0	0	0	0
7	0	0	0	0	19
8	0	0	0	0	0
9	0	0	0	0	19
10	0	0	0	0	0
11	0	0	0	0	0
12	76	18	23	115	127
13	76	18	23	115	127
14	74	17	15	106	106
15	38	4	8	44	75
16	0	0	0	0	0
17	2	1	8	9	21
18	0	0	0	0	0
19	0	0	0	0	0
20	36	13	7	62	31
21	2	1	8	9	15
22	70	21	33	128	65
23	146	38	50	234	228
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	6
27	30	9	18	60	44
28	0	0	0	0	6
29	144	37	42	225	213
30	68	20	25	119	50
31	76	17	17	106	163
32	0	0	0	0	6
33	2	1	8	9	27
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	19
38	0	0	0	0	19
39	0	0	0	0	19
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.4: Run 3: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	30	9	16	48	70
1	0	0	0	0	0
2	0	0	0	0	0
3	47	10	12	54	31
4	0	0	0	0	0
5	1	3	3	12	17
6	0	0	0	0	0
7	0	0	0	0	16
8	0	0	0	0	0
9	0	0	0	0	16
10	0	0	0	0	0
11	0	0	0	0	0
12	72	19	21	117	122
13	72	19	21	117	122
14	71	16	18	105	105
15	24	6	6	51	74
16	0	0	0	0	0
17	1	3	3	12	17
18	0	0	0	0	0
19	0	0	0	0	0
20	47	10	12	54	31
21	1	3	3	12	9
22	80	19	25	133	54
23	149	39	47	235	227
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	8
27	35	10	11	63	33
28	0	0	0	0	8
29	148	36	44	223	218
30	79	16	22	121	45
31	69	20	22	102	173
32	0	0	0	0	8
33	1	3	3	12	25
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	16
38	0	0	0	0	16
39	0	0	0	0	16
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.5: Run 4: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	37	7	5	53	60
1	0	0	0	0	0
2	0	0	0	0	0
3	33	10	15	64	34
4	0	0	0	0	0
5	1	0	6	9	18
6	0	0	0	0	0
7	0	0	0	0	17
8	0	0	0	0	0
9	0	0	0	0	17
10	0	0	0	0	0
11	0	0	0	0	0
12	73	22	27	124	133
13	73	22	27	124	133
14	72	22	21	115	115
15	39	12	6	51	81
16	0	0	0	0	0
17	1	0	6	9	18
18	0	0	0	0	0
19	0	0	0	0	0
20	33	10	15	64	34
21	1	0	6	9	16
22	66	15	33	115	62
23	148	33	44	225	223
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	2
27	32	4	17	49	40
28	0	0	0	0	2
29	147	33	38	216	207
30	65	15	27	106	46
31	82	18	11	110	161
32	0	0	0	0	2
33	1	0	6	9	20
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	17
38	0	0	0	0	17
39	0	0	0	0	17
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.6: Run 5: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	40	6	5	45	69
1	0	0	0	0	0
2	0	0	0	0	0
3	40	8	20	56	27
4	0	0	0	0	0
5	2	1	2	15	15
6	0	0	0	0	0
7	0	0	0	0	24
8	0	0	0	0	0
9	0	0	0	0	24
10	0	0	0	0	0
11	0	0	0	0	0
12	66	18	28	122	122
13	66	18	28	122	122
14	64	17	26	107	107
15	24	9	6	51	80
16	0	0	0	0	0
17	2	1	2	15	15
18	0	0	0	0	0
19	0	0	0	0	0
20	40	8	20	56	27
21	2	1	2	15	11
22	75	22	27	124	52
23	156	38	39	233	229
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	4
27	32	8	16	62	34
28	0	0	0	0	4
29	154	37	37	218	218
30	73	21	25	109	41
31	81	16	12	109	177
32	0	0	0	0	4
33	2	1	2	15	19
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	24
38	0	0	0	0	24
39	0	0	0	0	24
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.7: Run 6: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	43	8	13	51	68
1	0	0	0	0	0
2	0	0	0	0	0
3	32	12	13	51	30
4	0	0	0	0	0
5	5	3	3	7	19
6	0	0	0	0	0
7	0	0	0	0	13
8	0	0	0	0	0
9	0	0	0	0	13
10	0	0	0	0	0
11	0	0	0	0	0
12	74	23	24	117	129
13	74	23	24	117	129
14	69	20	21	110	110
15	37	8	8	59	80
16	0	0	0	0	0
17	5	3	3	7	19
18	0	0	0	0	0
19	0	0	0	0	0
20	32	12	13	51	30
21	5	3	3	7	12
22	75	19	25	121	68
23	151	35	44	230	223
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	7
27	32	9	12	66	42
28	0	0	0	0	7
29	146	32	41	223	211
30	70	16	22	114	56
31	76	16	19	109	155
32	0	0	0	0	7
33	5	3	3	7	26
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	13
38	0	0	0	0	13
39	0	0	0	0	13
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.8: Run 7: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	34	11	11	57	67
1	0	0	0	0	0
2	0	0	0	0	0
3	35	6	13	65	27
4	0	0	0	0	0
5	7	0	8	9	18
6	0	0	0	0	0
7	0	0	0	0	24
8	0	0	0	0	0
9	0	0	0	0	24
10	0	0	0	0	0
11	0	0	0	0	0
12	77	14	28	113	122
13	77	14	28	113	122
14	70	14	20	104	104
15	35	8	7	39	77
16	0	0	0	0	0
17	7	0	8	9	18
18	0	0	0	0	0
19	0	0	0	0	0
20	35	6	13	65	27
21	7	0	8	9	15
22	79	19	26	125	63
23	150	41	45	236	233
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	3
27	39	10	12	60	47
28	0	0	0	0	3
29	143	41	37	227	218
30	72	19	18	116	48
31	71	22	19	111	170
32	0	0	0	0	3
33	7	0	8	9	21
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	24
38	0	0	0	0	24
39	0	0	0	0	24
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

Table E.9: Run 8: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	39	10	2	48	60
1	0	0	0	0	0
2	0	0	0	0	0
3	35	9	18	67	29
4	0	0	0	0	0
5	3	3	4	9	16
6	0	0	0	0	0
7	0	0	0	0	19
8	0	0	0	0	0
9	0	0	0	0	19
10	0	0	0	0	0
11	0	0	0	0	0
12	74	21	33	127	134
13	74	21	33	127	134
14	71	18	29	118	118
15	36	9	11	51	89
16	0	0	0	0	0
17	3	3	4	9	16
18	0	0	0	0	0
19	0	0	0	0	0
20	35	9	18	67	29
21	3	3	4	9	14
22	76	21	27	127	63
23	149	37	36	222	220
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	2
27	37	6	10	56	42
28	0	0	0	0	2
29	146	34	32	213	206
30	73	18	23	118	49
31	73	16	9	95	157
32	0	0	0	0	2
33	3	3	4	9	18
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	19
38	0	0	0	0	19
39	0	0	0	0	19
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.10:** Run 9: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	34	4	7	47	68
1	0	0	0	0	0
2	0	0	0	0	0
3	38	9	8	50	24
4	0	0	0	0	0
5	2	4	7	10	22
6	0	0	0	0	0
7	0	0	0	0	20
8	0	0	0	0	0
9	0	0	0	0	20
10	0	0	0	0	0
11	0	0	0	0	0
12	70	23	20	110	122
13	70	23	20	110	122
14	68	19	13	100	100
15	30	10	5	50	76
16	0	0	0	0	0
17	2	4	7	10	22
18	0	0	0	0	0
19	0	0	0	0	0
20	38	9	8	50	24
21	2	4	7	10	15
22	73	27	39	137	56
23	152	36	52	240	233
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	7
27	38	14	20	70	42
28	0	0	0	0	7
29	150	32	45	230	218
30	71	23	32	127	41
31	79	9	13	103	177
32	0	0	0	0	7
33	2	4	7	10	29
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	20
38	0	0	0	0	20
39	0	0	0	0	20
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.11:** Run 10: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	36	7	3	47	65
1	0	0	0	0	0
2	0	0	0	0	0
3	35	10	18	68	29
4	0	0	0	0	0
5	3	0	3	10	13
6	0	0	0	0	0
7	0	0	0	0	10
8	0	0	0	0	0
9	0	0	0	0	10
10	0	0	0	0	0
11	0	0	0	0	0
12	80	20	31	135	138
13	80	20	31	135	138
14	77	20	28	125	125
15	42	10	10	57	96
16	0	0	0	0	0
17	3	0	3	10	13
18	0	0	0	0	0
19	0	0	0	0	0
20	35	10	18	68	29
21	3	0	3	10	8
22	68	18	28	115	50
23	143	35	37	215	210
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	5
27	28	7	14	48	25
28	0	0	0	0	5
29	140	35	34	205	202
30	65	18	25	105	42
31	75	17	9	100	160
32	0	0	0	0	5
33	3	0	3	10	18
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	10
38	0	0	0	0	10
39	0	0	0	0	10
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.12:** Run 11: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	42	9	5	57	65
1	0	0	0	0	0
2	0	0	0	0	0
3	31	9	18	55	19
4	0	0	0	0	0
5	3	1	7	9	18
6	0	0	0	0	0
7	0	0	0	0	30
8	0	0	0	0	0
9	0	0	0	0	30
10	0	0	0	0	0
11	0	0	0	0	0
12	71	16	28	113	122
13	71	16	28	113	122
14	68	15	21	104	104
15	37	6	3	49	85
16	0	0	0	0	0
17	3	1	7	9	18
18	0	0	0	0	0
19	0	0	0	0	0
20	31	9	18	55	19
21	3	1	7	9	13
22	71	19	28	130	54
23	152	40	44	236	231
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	5
27	35	8	14	56	43
28	0	0	0	0	5
29	149	39	37	227	218
30	68	18	21	121	41
31	81	21	16	106	177
32	0	0	0	0	5
33	3	1	7	9	23
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	30
38	0	0	0	0	30
39	0	0	0	0	30
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.13:** Run 12: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	41	6	8	60	66
1	0	0	0	0	0
2	0	0	0	0	0
3	32	9	8	58	24
4	0	0	0	0	0
5	4	3	3	11	18
6	0	0	0	0	0
7	0	0	0	0	12
8	0	0	0	0	0
9	0	0	0	0	12
10	0	0	0	0	0
11	0	0	0	0	0
12	76	25	23	125	132
13	76	25	23	125	132
14	72	22	20	114	114
15	40	13	12	56	90
16	0	0	0	0	0
17	4	3	3	11	18
18	0	0	0	0	0
19	0	0	0	0	0
20	32	9	8	58	24
21	4	3	3	11	14
22	69	20	31	119	57
23	148	33	45	226	222
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	4
27	34	13	12	54	44
28	0	0	0	0	4
29	144	30	42	215	208
30	65	17	28	108	43
31	79	13	14	107	165
32	0	0	0	0	4
33	4	3	3	11	22
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	12
38	0	0	0	0	12
39	0	0	0	0	12
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.14:** Run 13: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	42	9	9	66	64
1	0	0	0	0	0
2	0	0	0	0	0
3	31	8	10	54	26
4	0	0	0	0	0
5	5	3	4	8	18
6	0	0	0	0	0
7	0	0	0	0	23
8	0	0	0	0	0
9	0	0	0	0	23
10	0	0	0	0	0
11	0	0	0	0	0
12	80	20	24	120	130
13	80	20	24	120	130
14	75	17	20	112	112
15	44	9	10	58	86
16	0	0	0	0	0
17	5	3	4	8	18
18	0	0	0	0	0
19	0	0	0	0	0
20	31	8	10	54	26
21	5	3	4	8	11
22	83	21	27	118	61
23	145	38	45	228	221
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	7
27	40	10	13	57	52
28	0	0	0	0	7
29	140	35	41	220	210
30	78	18	23	110	50
31	62	17	18	110	160
32	0	0	0	0	7
33	5	3	4	8	25
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	23
38	0	0	0	0	23
39	0	0	0	0	23
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.15:** Run 14: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	40	13	11	68	81
1	0	0	0	0	0
2	0	0	0	0	0
3	21	11	15	50	18
4	0	0	0	0	0
5	4	2	1	4	17
6	0	0	0	0	0
7	0	0	0	0	19
8	0	0	0	0	0
9	0	0	0	0	19
10	0	0	0	0	0
11	0	0	0	0	0
12	66	19	21	103	116
13	66	19	21	103	116
14	62	17	20	99	99
15	41	6	5	49	81
16	0	0	0	0	0
17	4	2	1	4	17
18	0	0	0	0	0
19	0	0	0	0	0
20	21	11	15	50	18
21	4	2	1	4	13
22	76	12	26	121	58
23	158	38	45	241	237
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	4
27	48	6	11	61	44
28	0	0	0	0	4
29	154	36	44	237	224
30	72	10	25	117	45
31	82	26	19	120	179
32	0	0	0	0	4
33	4	2	1	4	21
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	19
38	0	0	0	0	19
39	0	0	0	0	19
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.16:** Run 15: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	39	9	8	55	69
1	0	0	0	0	0
2	0	0	0	0	0
3	34	7	11	58	24
4	0	0	0	0	0
5	2	5	3	18	18
6	0	0	0	0	0
7	0	0	0	0	25
8	0	0	0	0	0
9	0	0	0	0	25
10	0	0	0	0	0
11	0	0	0	0	0
12	73	22	23	126	126
13	73	22	23	126	126
14	71	17	20	108	108
15	37	10	9	50	84
16	0	0	0	0	0
17	2	5	3	18	18
18	0	0	0	0	0
19	0	0	0	0	0
20	34	7	11	58	24
21	2	5	3	18	11
22	72	22	28	131	60
23	149	38	45	232	225
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	7
27	35	14	14	64	43
28	0	0	0	0	7
29	147	33	42	214	214
30	70	17	25	113	49
31	77	16	17	101	165
32	0	0	0	0	7
33	2	5	3	18	25
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	25
38	0	0	0	0	25
39	0	0	0	0	25
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.17:** Run 16: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	35	10	6	55	70
1	0	0	0	0	0
2	0	0	0	0	0
3	30	10	16	52	23
4	0	0	0	0	0
5	5	1	4	7	19
6	0	0	0	0	0
7	0	0	0	0	16
8	0	0	0	0	0
9	0	0	0	0	16
10	0	0	0	0	0
11	0	0	0	0	0
12	73	22	25	117	129
13	73	22	25	117	129
14	68	21	21	110	110
15	38	11	5	58	87
16	0	0	0	0	0
17	5	1	4	7	19
18	0	0	0	0	0
19	0	0	0	0	0
20	30	10	16	52	23
21	5	1	4	7	14
22	84	18	29	120	60
23	152	34	44	230	225
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	5
27	41	10	16	63	43
28	0	0	0	0	5
29	147	33	40	223	211
30	79	17	25	113	46
31	68	16	15	110	165
32	0	0	0	0	5
33	5	1	4	7	24
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	16
38	0	0	0	0	16
39	0	0	0	0	16
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.18:** Run 17: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	33	8	1	51	63
1	0	0	0	0	0
2	0	0	0	0	0
3	34	12	17	62	21
4	0	0	0	0	0
5	2	3	6	9	21
6	0	0	0	0	0
7	0	0	0	0	24
8	0	0	0	0	0
9	0	0	0	0	24
10	0	0	0	0	0
11	0	0	0	0	0
12	75	19	33	125	137
13	75	19	33	125	137
14	73	16	27	116	116
15	39	4	10	54	95
16	0	0	0	0	0
17	2	3	6	9	21
18	0	0	0	0	0
19	0	0	0	0	0
20	34	12	17	62	21
21	2	3	6	9	16
22	79	26	31	109	62
23	147	39	38	224	219
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	5
27	42	14	15	62	45
28	0	0	0	0	5
29	145	36	32	215	203
30	77	23	25	100	46
31	68	13	7	115	157
32	0	0	0	0	5
33	2	3	6	9	26
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	24
38	0	0	0	0	24
39	0	0	0	0	24
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.19:** Run 18: Agent Volume Per Street Segment

edgeID	GROUP1	GROUP2	GROUP3	POPULATION	NULLGROUP
0	42	6	9	52	61
1	0	0	0	0	0
2	0	0	0	0	0
3	28	13	14	55	19
4	0	0	0	0	0
5	3	1	1	11	22
6	0	0	0	0	0
7	0	0	0	0	25
8	0	0	0	0	0
9	0	0	0	0	25
10	0	0	0	0	0
11	0	0	0	0	0
12	65	25	21	117	128
13	65	25	21	117	128
14	62	24	20	106	106
15	34	11	6	51	87
16	0	0	0	0	0
17	3	1	1	11	22
18	0	0	0	0	0
19	0	0	0	0	0
20	28	13	14	55	19
21	3	1	1	11	18
22	75	18	28	124	65
23	158	31	45	234	230
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	4
27	33	9	12	59	46
28	0	0	0	0	4
29	155	30	44	223	212
30	72	17	27	113	47
31	83	13	17	110	165
32	0	0	0	0	4
33	3	1	1	11	26
34	0	0	0	0	0
35	0	0	0	0	0
36	220	55	65	340	340
37	0	0	0	0	25
38	0	0	0	0	25
39	0	0	0	0	25
40	0	0	0	0	0
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	0	0	0	0	0

**Table E.20:** Run 19: Agent Volume Per Street Segment

# Appendix F

## Source Code

All the code and instructions are provided in github repository:  
[https://github.com/Rssn1GM/ASBPM\\_UPB](https://github.com/Rssn1GM/ASBPM_UPB)

```
-----REQUIREMENTS-----
geopandas==0.4.1
importlib-metadata==0.0.0
matplotlib==3.0.3
networkx==2.2
numpy==1.16.2
osmnx==0.10
pandas==0.24.2
Shapely==1.6.4
python-louvain==0.13
python=3.7

-----01-Extracting_Nodes_Paths_Local_Files - UPB.ipynb-----

# # Street Network Analysis: Nodes and Paths
# ## Working with Shapefiles
import osmnx as ox, networkx as nx, matplotlib.cm as cm, pandas as pd, numpy as np
import geopandas as gpd
%matplotlib inline
import warnings
warnings.simplefilter(action="ignore")
pd.options.display.float_format = '{:20.2f}'.format
pd.options.mode.chained_assignment = None
import cityImage as ci

# ### Important: find EPSG of your case-study area
# ### Specifying the name of the city for convenience

#initialise path, names, etc.
city_name = 'UPB'
saving_path ='Outputs/'+city_name+'/'
epsg = 4326
#26986
crs = 'EPSG:' +str(epsg)

# ## 1. Loading Shapefile
```

```

# Provide files' directories and loading the data
# The polygon is not strictly necessary. It is needed when clipping the Street Network

#specify your loading paths here
loading_path_polygon = None
input_path = 'Input/' + city_name + '/UPB_500.shp'

# ### A note:
# To avoid the edge effect when computing betweennes centrality values,
# consider using an area larger than your actual case-study area.

# Leave "None" for attributes not contained in the dataframe or for attributes of no interest.
dict_columns = {"roadType_field": None, "direction_field": None, "speed_field": None, "name_field": None}
# if all items are None, don't pass the dictionary in the function

nodes_graph, edges_graph = ci.get_network_fromFile(input_path, epsg)

# ## 2 Cleaning and simplyfing the Street Network - gdf
#
# At the end of the previous section two files are obtained: nodes and edges (vertexes and links).

Below, before creating the actual graph, the two datasets are cleaned, simplified and corrected.
#
# Cleaning functions handle (through boolean parameters):
# * Duplicate geometries (nodes, edges).
# * Pseudo-nodes.
# * 'remove_islands': Disconnected islands.
# * 'dead_ends' Dead-end street segments.
# * 'self_loops' Self-Loops.
# * 'same_vertexes_edges' Edges with same from-to nodes, but different geometries.
# * 'fix_topology' This creates nodes and breaks street segments at intersections.
It is primarily useful for poorly formed datasets (usually OSM derived networks are topologically correct).
It accounts for segments classified as bridges or tunnels in OSM.
#
# 'same_vertexes_edges' handles edges with same pair of u-v nodes but different geometries.
When 'True', it derives a center line between the two segments, unless one of the two
segments is longer than the other (>10%)..
In this case, the shorter segment is deleted.

nodes_graph, edges_graph = ci.clean_network(nodes_graph, edges_graph, dead_ends = True, remove_islands = True,
                                             same_vertexes_edges = True, self_loops = True, fix_topology = False)

plot = ci.plot_gdf(edges_graph, black_background =
                    False, figsize = (10,10), title = city_name + ': Street Network',
                           color = 'black')

```

```

# Obtaining the graph from the case-study area and computing the centrality measures
graph = ci.graph_fromGDF(nodes_graph, edges_graph)

# ## 3. - Nodes centrality
#
# On the primal graph representation of the street network,
# the following centrality measures are computed on nodes,
# on the basis of road distance:
#
# * Road Distance Shortest Path Betweenness centrality ('Bc_Rd').
# * Information Centrality ('Bc_Rw')
# * Straightness centrality ('Sc')
(see [Crucitti et al. 2006] (https://journals.aps.org/pre/abstract/10.1103/PhysRevE.73.036125)).  

# * Reach centrality (e.g. 'Rc_400', 'Rc_600')
(readataped from [Sevtsuk & Mekonnen 2012]
(https://www.sutd.edu.sg/cmsresource/idc/papers/2012\_Urban\_Network\_Analysis\_A\%20New\%20Toolbox\_for\_ArcGIS.pdf))
- it measures the importance of a node based on the number of services
(e.g. commercial activities) reachable by that node (for instance within a buffer).
#
# 'measure' accepts 'betweenness_centrality', 'information_centrality', 'straightness_centrality',
# 'closeness_centrality'. The function returns dictionaries, which are going to be merged in the GDF below.
#
# The first measure ('Bc_Rd') is here used to identify "Lynchian" nodes.

# %%
"""
"measure" accepts "betweenness_centrality", "information_centrality", "straightness_centrality",
"closeness_centrality".
The function returns dictionaries, which are going to be merged in the GDF below.
"""

# betweenness centrality
Bc_Rd = ci.centrality(graph, nodes_graph, measure = "betweenness_centrality", weight = 'length',
normalized = False)
# information centrality
Bc_Rw = ci.centrality(graph, nodes_graph, measure = "information_centrality", weight = 'length',
normalized = False)
# straightness centrality
Sc = ci.centrality(graph, nodes_graph, measure = "straightness_centrality", weight = 'length',
normalized = False)

# ##### If using POI from OSM:
Choose which of the following amenities you consider relevant as "services" for computing reach centrality.
Check [OSM amenities] (https://wiki.openstreetmap.org/wiki/Key:amenity) for details

# %%

```

```

amenities = ['arts_centre', 'atm', 'bank', 'bar', 'bbq', 'bicycle_rental',
'bicycle_repair_station', 'biergarten',
'boat_rental', 'boat_sharing', 'brothel', 'bureau_de_change', 'cafe', 'car_rental', 'car_sharing',
'car_wash', 'casino', 'childcare',
'cinema', 'clinic', 'college', 'community_centre', 'courthouse', 'crematorium', 'dentist',
'dive_centre', 'doctors',
'driving_school', 'embassy', 'fast_food', 'ferry_terminal', 'fire_station', 'food_court', 'fuel',
'gambling', 'gym',
'hospital', 'ice_cream', 'internet_cafe', 'kindergarten', 'kitchen', 'language_school',
'library', 'marketplace',
'monastery', 'motorcycle_parking', 'music_school', 'nightclub', 'nursing_home',
'pharmacy', 'place_of_worship',
'planetarium', 'police', 'post_office', 'prison', 'pub', 'public_building', 'ranger_station',
'restaurant', 'sauna',
'school', 'shelter', 'shower', 'social_centre', 'social_facility', 'stripclub', 'studio',
'swingerclub', 'theatre',
'toilets', 'townhall', 'university', 'veterinary']

# %%
# reach centrality pre-computation, in relation to Point of Interests or any other point-geodataframes
convex_hull_wgs = ci.convex_hull_wgs(edges_graph)
services = nodes_graph.assign(services=0).astype({'services': int})
#services = ox.geometries.geometries_from_polygon(convex_hull_wgs, tags = {'amenity':True}).to_crs(epsg=epsg)
#services = services[services.amenity.isin(amenities)]
#services = services[services['geometry'].geom_type == 'Point']

# using a 50 mt buffer
#graph = ci.weight_nodes(nodes_graph, services, graph, field_name = 'services', radius = 1)

# Reach Centrality
Rc400 = nodes_graph.assign(Rc400=0).astype({'Rc400': int})
Rc600 = nodes_graph.assign(Rc600=0).astype({'Rc600': int})
#Rc400 = ci.reach_centrality(graph, weight = 'length', radius = 0.1, attribute = 'services')
#Rc600 = ci.reach_centrality(graph, weight = 'length', radius = 0.5, attribute = 'services')

# %%
## Appending the attributes to the geodataframe
dicts = [Bc_Rd, Bc_Rw, Sc]
columns = ['Bc_Rd', 'Bc_Rw', 'Sc']
for n,c in enumerate(dicts):
    nodes_graph[columns[n]] = nodes_graph.nodeID.map(c)

# ### 3.1 Nodes' centrality - Visualisation

# %%
col = ['Bc_Rd', 'Bc_Rw', 'Sc']

```

```

titles = ['Betweennes Centrality - Shortest Path']

# ##### Single figure visualisation with Lynch-like Breaks

# %%
for n, column in enumerate(columns):
    #rescaling values for 0-1 visualisation
    ci.plot_gdf(nodes_graph, column = column, cmap = 'magma', classes = 5, scheme = 'natural_breaks',
                geometry_size = 5, figsize = (5,5), base_map_gdf = edges_graph, base_map_zorder = 0,
                legend = True)

# ## 4. Paths
#
# On the primal graph representation of the street network,
# the following centrality measures are computed on edges:
# * Road Distance Betweenness centrality.
# * Angular Betweenness centrality (On the dual graph representation of the street network)

# %%
# Road Distance betweenness centrality
Eb = nx.edge_betweenness_centrality(graph, weight = 'length', normalized = False)

# %%
# appending to the geodataframe
if 'Eb' in edges_graph.columns:
    edges_graph.drop(['Eb'], axis = 1, inplace = True)
edges_graph = ci.append_edges_metrics(edges_graph, graph, [Eb], ['Eb'])

# ## Dual graph analysis
#
# Here street-segments are transformed into nodes (geographically represented by their centroids).
# Fictional links represent instead intersections.
# Thus if two segments are connected in the actual street network,
# a link in the dual graph representation will be created by connecting the corresponding nodes.
# This process allows to take advantage of angular relationships in centrality measures
# computation and other network operations.

# %%
# Creating the dual geodataframes and the dual graph.
nodesDual_graph, edgesDual_graph = ci.dual_gdf(nodes_graph, edges_graph, epsg)
dual_graph = ci.dual_graph_fromGDF(nodesDual_graph, edgesDual_graph)

# %%
plot = ci.plot_gdf(edgesDual_graph, black_background = False,
                    figsize = (10,10), title = city_name+' : Dual Graph representation',
                    color = 'black', geometry_size = 0.7)

```

```

# ### Angular Betweenness - Paths

# %%
# Angular-change betweenness centrality
Ab = nx.betweenness_centrality(dual_graph, weight = 'rad', normalized = False)
edges_graph['Ab'] = edges_graph.edgeID.map(Ab)

# ### 4.2 Paths (Edges' centrality) visualisation

# %%
columns = ['Eb', 'Ab']
titles = ['Betweenness - Distance SP', 'Betweenness - Cum. Angular Change SP']
cbar_dict = {'cbar' : True, 'cbar_min_max' : True, 'cbar_shrink' : 0.50}

for n, column in enumerate(columns):
    edges_graph[column+'_sc'] = ci.scaling_columnDF(edges_graph[column])

columns_rescaled = [column+'_sc' for column in columns]
plot = ci.plot_grid_gdf_columns(edges_graph, classes = 8, columns = columns_rescaled, titles = titles,
                                geometry_size = 1.9,
                                nrows = 1, ncols = 2, scheme = 'Natural_Breaks', cmap = 'magma', figsize = (15 ,10),
                                **cbar_dict)

# ## Exporting

# %%
print(nodes_graph.dtypes)

# %%
print(edges_graph.dtypes)

# %%
#Debugging purposes
#numpy_int64_columns = edges_graph.select_dtypes(include=['int64']).columns
#print(numpy_int64_columns)
#edges_graph[['u', 'v']] = edges_graph[['u', 'v']].astype(int)

# %%
# provide path
saving_path = 'Outputs/' + city_name + '/'

# primal graph
nodes_graph.to_file(saving_path + 'nodes.shp', driver='ESRI Shapefile')

```

```

edges_graph.to_file(saving_path+'edges.shp', driver = 'ESRI Shapefile')

# %%
# dual graph
nodesDual_graph.drop('intersecting', axis=1).to_file(saving_path+'nodesDual.shp', driver = 'ESRI Shapefile')
edgesDual_graph.to_file(saving_path+'edgesDual.shp', driver = 'ESRI Shapefile')

-----02-District identification - UPB.ipynb-----

# %%
import osmnx as ox, networkx as nx, matplotlib.cm as cm, pandas as pd, numpy as np
import geopandas as gpd
from shapely.geometry import Point, mapping
%matplotlib inline

import warnings
warnings.simplefilter(action="ignore")
pd.options.display.float_format = '{:20.2f}'.format
pd.options.mode.chained_assignment = None

import cityImage as ci

# %%
city_name = 'UPB'
epsg = 4326
crs = 'EPSG:' + str(epsg)

# %% [markdown]
# ### Download from OSM the drive network

# %% [markdown]
# Choose between the following methods:
# * 'OSMplace', provide an OSM place name (e.g. City).
# * 'polygon', provide an WGS polygon of the case-study area.
# * 'distance_from_address', provide a precise address and define parameter 'distance'

# %%
#place = 'Muenster, Germany'
#download_method = 'OSMplace'
#distance = None

#nodes_graph, edges_graph = ci.get_network_fromOSM(place, download_method, 'drive', epsg, distance = distance)
#nodes_graph, edges_graph = ci.clean_network(nodes_graph, edges_graph, dead_ends = True, remove_islands = True,
#                                             self_loops = True, same_vertexes_edges = True)

# Creating and saving the dual geodataframes

```

```

#nodesDual_graph, edgesDual_graph = ci.dual_gdf(nodes_graph, edges_graph, epsg)

# %% [markdown]
# ### Alternatively, Load from local path (when already processed the network)

# %%
input_path = 'Outputs/' + city_name + '/'

nodes_graph = gpd.read_file(input_path + 'nodes.shp')
edges_graph = gpd.read_file(input_path + 'edges.shp')
nodesDual_graph = gpd.read_file(input_path + 'nodesDual.shp')
edgesDual_graph = gpd.read_file(input_path + 'edgesDual.shp')

try:
    nodes_graph, edges_graph = nodes_graph.to_crs(epsg=epsg), edges_graph.to_crs(epsg=epsg)
    nodesDual_graph, edgesDual_graph = nodesDual_graph.to_crs(epsg=epsg), edgesDual_graph.to_crs(epsg=epsg)
except:
    nodes_graph.crs, edges_graph.crs = crs, crs
    nodesDual_graph.crs, edgesDual_graph.crs = crs, crs

nodes_graph.index, edges_graph.index, nodesDual_graph.index =
nodes_graph.nodeID, edges_graph.edgeID, nodesDual_graph.edgeID
nodes_graph.index.name, edges_graph.index.name, nodesDual_graph.index.name = None, None, None

# %% [markdown]
# ### Visualise

# %%
fig = ci.plot_gdf(edges_graph, black_background = False, figsize = (10,10),
title = city_name + ': Street Network - drive')

# %% [markdown]
# ## District identification

# %%
# creating dual_graph
dual_graph = ci.dual_graph_fromGDF(nodesDual_graph, edgesDual_graph)

# %% [markdown]
# Different weights can be used to extract the partitions. 'None' indicates that no weights will be used
# (only topological relationships will matter).
The function returns a GeoDataFrame with partitions assigned to edges,
# with column named as 'p_name_weight' (e.g. 'p_length').

# %%
weights = ['length', 'rad', None]
districts = edges_graph.copy()

```

```

for weight in weights:
    districts = ci.identify_regions(dual_graph, districts, weight = weight)

# %%
# visualising
columns = ['p_length', 'p_rad', 'p_topo']
titles = ['Partition - Distance', 'Partition - Angles', 'Partition - Topological']

nlabels = max([len(districts[column].unique()) for column in columns])
cmap = ci.rand_cmap(nlabels = nlabels, type_color='bright')
fig = ci.plot_grid_gdf_columns(districts, columns = columns, titles = titles, geometry_size = 1.5,
                                cmap = cmap, black_background = True,
                                legend = False, figsize = (15, 10), ncols = 3, nrows = 1)

# %%
for n, column in enumerate(columns):
    cmap = ci.rand_cmap(nlabels = len(districts[column].unique()), type_color='bright')
    partitions = ci.polygonise_partitions(districts, column, convex_hull = False)
    ci.plot_gdf(partitions, column = column, cmap = cmap, title = titles[n], black_background = True,
                figsize = (7, 5),
                base_map_gdf = districts, base_map_color = 'grey', base_map_zorder = 1)

# %%
# saving
saving_path = 'Outputs/' + city_name + '/entireNetwork/'
districts.to_file(saving_path + city_name + "_edges.shp", driver='ESRI Shapefile')

# %% [markdown]
# ## Assigning regions to the Pedestrian Street Network
# ### Case-study area (pedestrian walkable network). It should be obtained via notebook 01-Nodes_Paths
# This set of functions assigns regions to a walkable network for further modelling in Pedestrian Simulation, for example.
# The simply identifiaction of regions (districts) region from the urban configuration ends above.

# %%
input_path = 'Outputs/' + city_name + '/'
nodes_graph_ped = gpd.read_file(input_path + 'nodes.shp')
edges_graph_ped = gpd.read_file(input_path + 'edges.shp')

try:
    nodes_graph_ped, edges_graph_ped = nodes_graph_ped.to_crs(epsg=epsg), edges_graph_ped.to_crs(epsg=epsg)
except:
    nodes_graph_ped.crs, edges_graph_ped.crs = crs, crs

nodes_graph_ped.index, edges_graph_ped.index = nodes_graph_ped.nodeID, edges_graph_ped.edgeID
nodes_graph_ped.index.name, edges_graph_ped.index.name = None, None

```

```

# %% [markdown]
# #### Otherwise, get it from OSM:
#
# Choose between the following methods:
# * 'OSMplace', provide an OSM place name (e.g. City).
# * 'polygon', provide an WGS polygon of the case-study area.
# * 'distance_from_address', provide a precise address and define parameter 'distance'
# (which is otherwise not necessary)

# %%
#place = 'Domplatz, Muenster, Germany' ## must be different from the area used for the drive network
#download_method = 'distance_from_address'
#distance = 2500

#nodes_graph_ped, edges_graph_ped = ci.get_network_fromOSM(place, download_method, 'walk', epsg,
distance = distance)
#nodes_graph_ped, edges_graph_ped = ci.clean_network(nodes_graph_ped, edges_graph_ped, dead_ends = True,
#
#                                         remove_islands = True, self_loops = True, same_vertexes_edges = True)

# %% [markdown]
# #### Option 2 - Load it from local path

# %% [markdown]
# #### Visualising

# %%
fig = ci.plot_gdf(edges_graph_ped, black_background = False, figsize = (10,10), title = city_name+':
    Street Network - walk',
    color = 'red', geometry_size = 0.3)

# %%
fig = ci.plot_gdf(edges_graph_ped, scheme = None, black_background = False, figsize = (10,10), title =
    city_name+': Entire Street Network vs Case-study area', color = 'red', geometry_size = 0.2,
    base_map_gdf = edges_graph, base_map_color = 'black', base_map_alpha = 0.3)

# %% [markdown]
# ## Assigning nodes and edges in the pedestrian network to Partitions

# %%
# choosing the type of partition to be used
column = 'p_length'
min_size_district = 10

# %%
dc = dict(districts[column].value_counts())

## ignore small portions

```

```

to_ignore = {k: v for k, v in dc.items() if v <= min_size_district}
tmp = districts[~((districts[column].isin(to_ignore)) | (districts[column] == 999999))].copy()

partitions = ci.polygonise_partitions(tmp, column)
nodes_graph_ped = ci.district_to_nodes_from_polygons(nodes_graph_ped, partitions, column)
nodes_graph_ped[column] = nodes_graph_ped[column].astype(int)

# %%
# visualisation
#cmap = ci.rand_cmap(nlabels = len(nodes_graph_ped[column].unique()), type_color='bright')
#fig = ci.plot_gdf(nodes_graph_ped, column = column, title = 'Districts assigned to pedestrian SN', cmap = cmap,
#                   geometry_size = 3.5, base_map_gdf = edges_graph_ped,
#                   base_map_color = 'white', base_map_alpha = 0.35,
#                   black_background = True, legend = False, figsize = (10,10))

# %% [markdown]
# ### Fixing disconnected districts and assigning nodes to existing connected districts

# %%
nodes_graph_ped = ci.amend_nodes_membership(nodes_graph_ped, edges_graph_ped, column, min_size_district)

# assigning gateways
nodes_graph_ped = ci.find_gateways(nodes_graph_ped, edges_graph_ped, column)

# %% [markdown]
# ## Visualising the final division

# %%
#cmap = ci.rand_cmap(nlabels = len(nodes_graph_ped[column].unique()), type_color='bright')
#fig = ci.plot_gdf(nodes_graph_ped, column = column,
#                  title = 'Districts assigned to pedestrian SN - after correction',
#                  cmap = cmap, geometry_size = 3.5, base_map_gdf = edges_graph_ped, base_map_color = 'white',
#                  base_map_alpha = 0.35, black_background = True, legend = False, figsize = (10,10))

# %%
nodes_graph_ped['district'] = nodes_graph_ped[column].astype(int)

# %%
# saving
saving_path = 'Outputs/' + city_name + '/'
nodes_graph_ped.to_file(saving_path + "nodes.shp", driver='ESRI Shapefile')
edges_graph_ped.to_file(saving_path + "edges.shp", driver='ESRI Shapefile')

-----03-Landmarks_Local_Files - UPB.ipynb-----
# %% [markdown]
# Four scores are here computed to extract computation landmarks: structural, visual, cultural, pragmatic.

```

```

# The user has to provide at least an attribute shapefile containing the height attribute
(the only mandatory field).
# See below for further instructions on the data sources.
#
# The user can also provide, seaprately, information about cultural/historical landmarks
(e.g. position of listed important buildings); see *3 - Cultural component*

# %%
import pandas as pd, numpy as np, geopandas as gpd, osmnx as ox
from shapely.geometry import LineString
%matplotlib inline

import warnings
warnings.simplefilter(action="ignore")
pd.options.display.float_format = '{:20.2f}'.format
pd.options.mode.chained_assignment = None

import cityImage as ci

# %% [markdown]
# ## Loading the layers
#
# Provide the input path where the building layer of the city/area of interest is stored.
#
# Landmarks are extracted from buildings that are within the chosen case-study area ('buildings' geodataframe). Other buildings around the case study area have to be included in the analysis in order to correctly compute the landmark measures, namely, to avoid that buildings located along the boundaries of the case-study area benefit from the fact that in the 'buildings' geodataframe outer buildings are not included.
#
# In other words, 'buildings' is a subset of 'obstructions'.
#
# One can set 'option' to:
# * '1' - Load one's 'buildings' and 'obstructions' gdfs from a local path. <span style="color:blue">Choose when: Files are ready and contains the 'height', and a 'land_use' fields.</span>
#
# * '2' - Get a large set of building from OSM ('obstructions'), extract the 'buildings', and attach information regarding 'height' and 'land_use' use from an existing file that it is in one's possession (if this file is for example issued by an official data source and therefore more complete than OSM). This step makes sense only if one's not comfortable with the shapes of the building footprints in their possession. Often these files contain footprints too precise and not fitting the purposes of this work. Instead, OSM provides already simplified footprints. <span style="color:blue"> Choose when: Having a shapefile with the 'height' and 'land_use' attributes but it is preferred to use the more simplified OSM geometries and attach the attributes to the OSM footprints.</span>
#
# * '3' - Load a buildings dataframe that would correspond to 'obstructions' and obtain the 'buildings' geodataframe by using a certain distance or a case-study area polygon (see below for details).

```

```

<span style="color:blue"> Choose when: Having a shapefile containing buildings in a large area, including and
outside one's case study area; one is comfortable with the geometries and the 'height' and 'land_use' attributes
are contained in such a file.</span>
#
# Buildings smaller than 200 square meters are kept out of the analysis, as well as buildings whose height is
lower than imt (when the 'height' is filled).
#
# ### Setting the coordinate systems of the case-study area

# %%
city_name = 'UPB'
epsg = 4326
crs = 'EPSG:' + str(epsg)
input_path = 'Input/' + city_name + '/'
option = 1

# %%
# from local
case_study_area = gpd.read_file(input_path + city_name + '_city_centre.shp').to_crs(epsg=epsg).iloc[0].geometry

# %%
#specify the land use field's name in the file
height_field = 'height'
base_field = 'base'
land_use_field = 'land_use'

# %% [markdown]
# ### Option 1
# Using already prepared 'buildings' and 'obstructions' datasets. One should be aware that 'buildings' should be
a perfect subset of 'obstructions'. This means that all the buildings in 'buildings' should be in
'obstructions', sharing the same attributes.
#
# * The datasets should contain fields named 'base' and 'height', representing the base height of the building
(could be set to 0) and the height of the building.
# * The 'buildingID' should be already set correctly, that is, matching in the 'obstructions'and 'buildings'
GeoDataFrames.

# %%
if option == 1:
    buildings = gpd.read_file(input_path + city_name + '_buildings.shp').to_crs(epsg=epsg)
    obstructions = gpd.read_file(input_path + city_name + '_obstructions.shp').to_crs(epsg=epsg)
    buildings.index, obstructions.index = buildings.buildingID, obstructions.buildingID
    buildings.index.name, obstructions.index.name = None, None
    obstructions['land_use_raw'] = obstructions[land_use_field]
    obstructions.drop(land_use_field, axis = 1, inplace = True)

# %% [markdown]

```

```

# ### Option 2

# Here one would download the obstructions from OpenStreetMap using a large metropolotin area of one's case-study
area.

# One has also the chance to attach attributes from a file in their possession and simplify the footprints.
# The 'buildings' GeoDataFrame is than obtained by using an OSM polygon representing the actual case-study area
(and, in case, a buffer around it).

# %%
#simplify = True
#if option == 2:
#    place = ''
#    obstructions_tmp = ci.get_buildings_fromOSM(place, 'OSMplace', epsg = epsg, distance = None)
#    obstructions_tmp = obstructions_tmp[obstructions_tmp.area > 1]

## simplify geometries?
#    if simplify:
#        obstructions_tmp = ci.simplify_footprints(obstructions_tmp, crs)
#        obstructions_tmp['base'] = 0.0

# attach attributes from..
#    attribute_file = gpd.read_file(input_path+city_name+'_buildings_obstructions.shp').to_crs(epsg=epsg)

# provide the name of the height, base_height, and land use attributes in your file
#    obstructions = ci.attach_attributes(obstructions_na, attribute_file, height_field = height_field,
#                                         base_field = base_field,
#                                         lan_use_field = lan_use_field)

# getting the polygon
#    case_study_area = ox.geocode_to_gdf('Muenster Mitte').to_crs(epsg=epsg).iloc[0].geometry

#    buildings = obstructions[obstructions.geometry.within(case_study_area)]
#    buildings.index = buildings.buildingID
#    buildings.index.name = None

# %% [markdown]
# ### Option 3
#
# If one doesn't have a precise case-study area polygon to pass: Use the parameter 'distance_from_center' to
indicate till how far away from the center of the case-study area one wants to include buildings. When the user
doesn't provide the case-study area polygon nor the distance parameter, 'buildings' and 'obstructions' overlap;
they have the same extent as the original input file.

#
# 'height_field' and 'base_field' should indicate the maximum and the base elevations attributes of the
buildings, in the original .shp file. When 'base_field' is not provided, 'base' is automatically set to 0.
'height_field' is necessary to perform the landmark extraction. If such a feature is not at disposal, one should
set 'height_field' to 'None'.

```

```

# %%
#if option == 3:
#    buildings, obstructions = ci.get_buildings_fromFile(input_path+city_name+'_obstructions.shp', epsg,
#                                                       case_study_area = case_study_area, distance_from_center = 0,
#                                                       height_field = None, base_field = base_field,
#                                                       land_use_field = land_use_field)

# %% [markdown]
# ### Optional readjustments

# %%
obstructions['land_use_raw'].unique()

# %%
land_use_to_disregard = ['stockroom', 'warehouse', 'garage', 'stable', 'workshop', 'converter', 'pumping station',
                          'barracks', 'barn',
                          'greenhouse', 'shed', 'greenhouse (botany)', 'bunker', 'shelter', 'heating plant', 'parking', 'garage',
                          'electricity substation']
obstructions = obstructions[~((obstructions.land_use_raw.isin(land_use_to_disregard)) & (obstructions.area
< 200))]

# %%
obstructions = obstructions[~((obstructions.land_use_raw.isnull()) & (obstructions.height.isnull()))]
height = obstructions[~obstructions.height.isnull()].height.quantile(0.25)
obstructions['height'] = obstructions['height'].where(obstructions['height'] > 0.0, height)
buildings = buildings[buildings.buildingID.isin(obstructions.buildingID)]

# %% [markdown]
# ## Visualisation

# %%
fig = ci.plot_gdf(buildings, scheme = None, black_background = False, figsize = (10,10),
                    title = city_name+': Buildings vs Obstructions', base_map_gdf = obstructions,
                    base_map_color = 'black',
                    base_map_alpha = 0.3)

# %% [markdown]
# ## 1 - Structural component

# %% [markdown]
# It considers:
# * The area of the building.
# * Its distance from the road.
# * The number of adjacent buildings, within the given 'neighbours_radius', around it.
# * A measure of 2d advance visibility, expanded till a maximum distance from the center of the building on the
basis of the 'advance_vis_expansion_distance' parameter.

```

```

# %%
# reading the street network and passing it in the function to compute the structural score
edges = gpd.read_file('Outputs/'+city_name+'/edges.shp')

# The 'buffer' parameter indicates distance within the algorithms look for neighbouring buildings
buildings = ci.structural_score(buildings, obstructions, edges, advance_vis_expansion_distance = 300,
neighbours_radius = 150)

# %% [markdown]
# ## 2 - Visual component

# %% [markdown]
# #### Sight Lines Construction
# This is a computationally expensive step that may takes hours (in particular the intervisibility check that
verifies whether the sight lines are actually visible or not).
# One way to reduce it is to cluster nodes and therefore generate a lower number of sight lines.
# The function takes two parameters, besides the observer and the targets GeoDataFrames:
# * 'distance_along': It regulates the interval along the exterior of each building for which a point is
identified and used as a target point. If the exterior of the building is shorter than the parameter, at least
one point is identified.
# * 'distance_min_observer_target': The function only computes sight lines for nodes and buildings whose distance
is higher than this parameter.
#
#
# ##### One can choose between two methods to compute the 3dvis score:
#
# 'combined' --> It computes a combined score that takes into account, per each building:
# * The number of sight lines towards it.
# * Their average length.
# * The maximum length amongst them.
#
# 'longest' --> It just uses the longest sight line's length as a score
#
#
# %%

nodes = gpd.read_file('Outputs/'+city_name+'/nodes.shp')
#sight_lines = ci.compute_3d_sight_lines(nodes, buildings, distance_along = 300, distance_min_observer_target =
300)
#buildings, sight_lines = ci.visibility_score(buildings, sight_lines, method = 'combined')
buildings.head()

# %%
saving_path = 'Outputs/'+city_name+ '/'
#sight_lines.to_file(saving_path+'landmarks.shp', driver='ESRI Shapefile')

# %%

```

```

# 'None' stands for sight_lines. Sight_lines are not considered for now.
#fig = ci.plot_gdf(None, alpha = 0.06, color = 'white', base_map_gdf = obstructions, base_map_color = 'yellow',
#                   geometry_size = 0.09, base_map_alpha = 0.4)

# %% [markdown]
# ## 3 - Cultural component

# %% [markdown]
# It measures the symbolic/cultural value of each building.
# *This part is case-study specific, see other examples for generalizable methods*

# %%
cultural_elements = gpd.read_file(input_path+'otherSources/'+city_name+'_historic_landmarks.shp').to_crs
(epsg=epsg)
cultural_elements = gpd.GeoDataFrame(columns=['name', 'geometry'], crs=buildings.crs)
buildings = ci.cultural_score(buildings, cultural_elements)

# %% [markdown]
# ## 4 - Pragmatic component

# %% [markdown]
# Before computing the pragmatic score, the 'land_use_raw' raw field is categorised into more granular
categories. One can fill in the following categories to accomodate descriptors not accounted for. Run the 1st and
the 2nd cells below so to check which descriptors in 'land_use_raw' are not inclued at the moment
in the macro categories.

# %%
# introducing classifications and possible entries

adult_entertainment = ['brothel', 'casino', 'swingerclub', 'stripclub', 'nightclub', 'gambling']

agriculture = ['shed', 'silo', 'greenhouse', 'stable', 'agricultural and forestry', 'greenhouse (botany)',
               'building in the botanical garden']

attractions = ['attractions', 'attraction', 'aquarium', 'monument', 'gatehouse', 'terrace', 'tower',
               'attraction and leisure',
               'information', 'viewpoint', 'tourist information center', 'recreation and amusement park', 'zoo',
               'exhibition hall, trade hall', 'boathouse', 'bath house, thermal baths', 'entertainment hall',
               'sauna']

business_services = ['bank', 'service', 'offices', 'foundation', 'office', 'atm', 'bureau_de_change',
                      'post_office',
                      'post_office;atm', 'coworking_space', 'conference_centre', 'trade and services',
                      'trade and services building',
                      'customs office', 'insurance', 'tax_office', 'post', 'administrative building',
                      'facility building',
                      'residential building with trade and services', 'data_center', 'tax office']

```

```

commercial = [ 'commercial', 'retail', 'pharmacy', 'commercial;educa', 'shop', 'supermarket', 'books',
              'commercial services',
              'commercial land', 'car_wash', 'internet_cafe', 'driving_school', 'marketplace', 'fuel',
              'car_sharing',
              'commercial and industry buidling', 'crematorium', 'commercial building', 'commercial and industry
              building',
              'commercial building to traffic facilities (general)', 'funeral parlor', 'gas station', 'car wash',
              'pumping station', 'boat_rental', 'boat_sharing', 'bicycle_rental', 'car_rental', 'dive_centre']

culture = ['club_house', 'gallery', 'arts_centre', 'cultural facility', 'cultural_centre', 'theatre', 'cinema',
           'studio',
           'exhibition_centre', 'music_school', 'theater', 'castle', 'museum', 'culture']

eating_drinking = ['bbq', 'restaurant', 'fast_food', 'cafe', 'bar', 'pub', 'accommodation, eating and drinking',
                   'ice_cream', 'kitchen', 'food_court', 'cafe;restaurant', 'biergarten']

education_research = ['university', 'research', 'university building', 'education and research',
                      'research_institute',
                      'research_institu']

emergency_service = [ 'fire brigade', 'fire_station', 'police', 'emergency_service', 'resque_station',
                      'ranger_station',
                      'security']

general_education = ['school', 'college', 'kindergarten', 'education', 'education and health', 'childcare',
                     'language_school', 'children home', 'nursery', 'general education school']

hospitality = [ 'hotel', 'hostel', 'guest_house', 'building for accommodation', 'hotel, motel, pension',
                'refuge']

industrial = ['industrial', 'factory', 'construction', 'manufacturing and production', 'gasometer', 'workshop',
               'production building', 'manufacture']

medical_care = ['hospital', 'doctors', 'dentist', 'clinic', 'veterinary', 'medical care', 'nursing_home',
                 'sanatorium, nursing home', 'retirement home', 'healthcare', 'mortuary']

military_detainment = ['general aviation', 'barracks', 'military', 'penitentiary', 'prison']

other = ['toilets', 'picnic_site', 'hut', 'storage_tank', 'canopy', 'toilet', 'bunker, shelter', 'shelter',
          'warehouse',
          'converter', 'garage', 'garages', 'parking', 'block', 'roof', 'no', 'exempt', 'exempt 121a',
          'bicycle_parking']

public = ['townhall', 'public_building', 'library', 'civic', 'courthouse', 'public', 'embassy',
          'public infrastructure', 'community_centre', 'court', 'district government', 'government',
          'residential building with public facilities']

```

```

religious = ['church', 'place_of_worship', 'convent', 'rectory', 'chapel', 'religious building',
            'monastery', 'nuns home',
            'vocational school', 'cathedral', 'religious buildings']
residential = ['apartments', None, 'NaN', 'residential', 'flats', 'houses', 'building', 'residential land',
               'residential building', 'student dorm', 'building usage mixed with living', 'house'
               'exempt', 'apartments 4-6 units', 'apartments 7 units above', 'residential condo unit',
               'mixed use res/comm',
               'residential three family', 'residential two family', 'house', 'commercial condo unit',
               'condominium master',
               'condominium parking', 'residential single family']

social = ['social_facility', 'community_centre', 'community buidling', 'dormitory', 'social_centre',
          'social serives building', 'social services', 'community hall', 'commercial social facility',
          'recreational']

sport = ['stadium', 'sport and entertainment', 'sports or exercise facility', 'gym', 'sports building',
         'sports hall',
         'horse riding school', 'swimming pool', 'sport hall', 'bowling hall', 'indoor swimming pool',
         'grandstand']

transport = ['transport', 'road transport', 'station', 'subway_entrance', 'bus_station',
             'shipping facility building',
             'train_station', 'railway building', 'railway station and london underground station',
             'railway station',
             'london underground station', 'light rapid transit station',
             'light rapid transit station and railway station']

utilities = ['gas supply', 'electricity supply', 'electricity substation', 'waste treatment building',
            'water supply', 'waste water treatment plant', 'smokestack', 'supply systems', 'waste management',
            'water works',
            'heating plant', 'boiler house', 'telecommunication']

# %%
obstructions['land_use_raw'] = obstructions['land_use_raw'].str.lower()
categories = [adult_entertainment, agriculture, attractions, business_services, commercial, culture,
              eating_drinking,
              education_research, emergency_service, general_education, hospitality, industrial, medical_care,
              military_detainment,
              other, public, religious, residential, social, sport, transport, utilities]
strings = ['adult_entertainment', 'agriculture', 'attractions', 'business_services', 'commercial', 'culture',
           'eating_drinking',
           'education_research', 'emergency_service', 'general_education', 'hospitality', 'industrial',
           'medical_care', 'military_detainment', 'other', 'public', 'religious', 'residential', 'social',
           'sport', 'transport', 'utilities']

all_uses = [item for sublist in categories for item in sublist]

```

```

land_uses = list(obstructions.land_use_raw.unique())
to_categorise = [item for item in land_uses if item not in all_uses]

# shows existing raw categories to classify
to_categorise

# %%
# and finally classify
obstructions = ci.classify_land_use(obstructions, 'land_use', 'land_use_raw', categories, strings)

# %% [markdown]
# ### Pragmatic meaning computation

# %% [markdown]
# ##### Computing the pragmatic score and assigning it to 'buildings_gdf'.
# The pragmatic component is computed on the 'obstructions' GeoDataFrame as the measure is based on the adjacent buildings' land use categories. It is a measure of entropy, or information, for which an unusual land-use category within an area receives higher score. The 'buffer' parameter indicates the extension of the area that is used to compute the score of a building, on the basis of its land-use's frequency.

# %%
obstructions = ci.pragmatic_score(obstructions, research_radius = 200)
buildings = pd.merge(buildings, obstructions[['prag', 'land_use', 'buildingID']], how = 'left', on = "buildingID")

# %% [markdown]
# ### URBAN DMA Categorisation
# The land use can also be cagtegorised into 3 categories see
https://journals.sagepub.com/doi/full/10.1177/0042098018819727

# %%
visit = ['adult_entertainment', 'attractions', 'culture', 'eating_drinking', 'sport', 'religious',
         'social', 'sport']
work = ['agriculture', 'business_services', 'commercial', 'education_research',
        'emergency_service', 'general_education',
        'industrial', 'medical_care', 'military_detaiment', 'public', 'transport', 'utilities']
live = ['hospitality', 'residential']

categories = [visit, work, live]
strings = ['visit', 'work', 'live']
buildings['land_use'] = ''
buildings = ci.classify_land_use(buildings, 'DMA', 'land_use', categories, strings)

# %%
# visualising
from matplotlib.colors import LinearSegmentedColormap
colors = ['red', 'gray', 'blue', 'gold']
cmap = LinearSegmentedColormap.from_list('cmap', colors, N=len(colors))

```

```

fig = ci.plot_gdf(buildings, column = 'DMA', title = 'Land use', black_background = True, figsize = (10,10),
cmap = cmap,
legend = True)

# %% [markdown]
# # 5 - Final scores

# %% [markdown]
# ### Defining Global indexes and components weights and assigning the score

# %%
print(buildings.columns)

# %%
global_indexes_weights = {'F3dvis': 0.50, 'fac': 0.30, 'height': 0.20, 'area': 0.30, 'F2dvis':0.30,
'neigh': 0.20, 'road': 0.20}
global_components_weights = {'vScore': 0.50, 'sScore' : 0.30, 'cScore': 0.20, 'pScore': 0.10}
buildings = ci.compute_global_scores(buildings, global_indexes_weights, global_components_weights)

# %% [markdown]
# ### Define Local indexes and components weights

# %%
local_indexes_weights = {'F3dvis': 0.50, 'fac': 0.30, 'height': 0.20, 'area': 0.40, 'F2dvis': 0.00,
'neigh': 0.30, 'road': 0.30}
local_components_weights = {'vScore': 0.25, 'sScore' : 0.35, 'cScore':0.10, 'pScore': 0.30}
rescaling_radius = 1000 # to define the local dimension, choose based on the extent of the case study area

buildings = ci.compute_local_scores(buildings, local_indexes_weights, local_components_weights,
rescaling_radius = rescaling_radius)

# %% [markdown]
# ### Visualisation of the Components

# %%
columns = ['sScore_sc', 'vScore_sc', 'pScore', 'cScore']
titles = ['Structural Score', 'Visual Score', 'Pragmatic Score', 'Cultural Score']
fig = ci.plot_grid_gdf_columns(gdf = buildings, columns = columns, titles = titles, black_background = True,
figsize = (15,10), scheme = 'fisher_jenks', cmap = 'magma', cbar = True, nrows = 2, ncols = 2)

# %% [markdown]
# ### Visualisation of the Global and Local Landmark Scores

# %%
columns = ['gScore_sc', 'lScore_sc']
titles = ['Global Score', 'Local Score']
fig = ci.plot_grid_gdf_columns(gdf = buildings, columns = columns, titles = titles, black_background = True,

```

```

figsize = (20,10), scheme = 'fisher_jenks', cmap = 'magma', cbar = True, nrows = 1, ncols = 2)

# %%
buildings.to_file(saving_path+'landmarks.shp', driver='ESRI Shapefile')

-----04-Barriers_UPB.ipynb-----
# %%

import pandas as pd
import geopandas as gpd
%matplotlib inline

import warnings
warnings.simplefilter(action="ignore")

import cityImage as ci

# %%
# initialise path, names, etc.
city_name = 'UPB'
epsg = 4326
crs = 'EPSG:' + str(epsg)

# %% [markdown]
# ### Download the network
#
# Choose between the following methods:
# * 'OSMplace', provide an OSM place name (e.g. City).
# * 'polygon', provide an WGS polygon of the case-study area.
# * 'distance_from_address', provide a precise address and define parameter 'distance' (which is otherwise not necessary)

# %%
# download graph and clean (see network notebook for details on the cleaning process)

#place = 'University of the Philippines Baguio, Baguio City, Philippines'
#download_method = 'distance_from_address'
#distance = 500
#nodes_graph, edges_graph = ci.get_network_fromOSM(place, download_method, 'walk', epsg, distance = distance)
#nodes_graph, edges_graph = ci.clean_network(nodes_graph, edges_graph, dead_ends = True,
remove_disconnected_islands = True,
#                                     self_loops = True, same_uv_edges = True)

# %% [markdown]
# ### Load from local path

# %%
loading_path = 'Outputs/' + city_name + '/'

```

```

nodes_graph = gpd.read_file(loading_path+'nodes.shp')
edges_graph = gpd.read_file(loading_path+'edges.shp')
barriers = gpd.read_file(loading_path+city_name+'_barriers.shp')
try:
    nodes_graph, edges_graph = nodes_graph.to_crs(epsg=epsg), edges_graph.to_crs(epsg=epsg)
except:
    nodes_graph.crs, edges_graph.crs = crs, crs

nodes_graph.index, edges_graph.index = nodes_graph.nodeID, edges_graph.edgeID
nodes_graph.index.name, edges_graph.index.name = None, None

# %%
#fig = ci.plot_gdf(edges_graph, scheme = None, black_background = False, figsize = 15, alpha = 1.0,
#color = 'black',
#                   title = city_name+": Street Network")

# %% [markdown]
# ## Barriers identification

# %% [markdown]
# Choose between the following methods:
# * *OSMplace*, provide an OSM place name (e.g. City).
# * *distance_from_address*, provide a precise address and define parameter "distance"
# (which is otherwise not necessary)
# * *polygon*, provide a Polygon (coordinates must be in units of latitude-longitude degrees).

# %%
# define method and create envelope with wgs coordinate system
download_method = 'polygon'
place = ci.envelope_wgs(edges_graph)
distance = None

# %%
road_barriers = gpd.GeoDataFrame()
secondary_road = gpd.GeoDataFrame() # optional - only in relation to walkability
water_barriers = gpd.GeoDataFrame()
railway_barriers = gpd.GeoDataFrame()
park_barriers = gpd.GeoDataFrame()

# %%
#barriers = road_barriers.append(secondary_road)
#barriers = water_barriers.append(barriers)
#barriers = railway_barriers.append(barriers)
#barriers = park_barriers.append(barriers)
barriers.reset_index(inplace = True, drop = True)
barriers['barrierID'] = barriers.index.astype(int)

```

```

envelope = edges_graph.unary_union.envelope
barriers_within = barriers[barriers.intersects(envelope)]

# %% [markdown]
# ### Visualisation

# %%
#fig = ci.plot_barriers(barriers_within, black_background = False, fig_size = 15,
title = city_name+' Barriers', legend = True)

# %%
# save barriers_gdf
saving_path = 'Outputs/'+city_name '/'
barriers.to_file(saving_path+"barriers.shp", driver='ESRI Shapefile')

# %% [markdown]
# ### Assigning barriers to street segments

# %% [markdown]
# ### Type of barriers
#
# Choose between the following methods:
# * *Positive barriers* - pedestrian perspective: Waterbodies, Parks.
# * *Negative barriers* - pedestrian perspective: Major Roads, Railway Structures.
# * *Structuring barriers* - barriers which structure and shape the image of the city: Waterbodies,
Major roads, Railways.

# %%
sindex = edges_graph.sindex

# along and within POSITIVE BARRIERS
# rivers
edges_graph = ci.along_water(edges_graph, barriers_within)
# parks
#edges_graph = ci.along_within_parks(edges_graph, barriers_within)

edges_graph = edges_graph.assign(a_rivers='[]').astype({'a_rivers': str})
edges_graph = edges_graph.assign(w_parks='[]').astype({'w_parks': str})

# altogheter
edges_graph['p_barr'] = edges_graph['a_rivers']+edges_graph['w_parks']
#edges_graph['p_barr'] = edges_graph.apply(lambda row: list(set(row['p_barr'])), axis = 1)

#edges_graph = gpd.GeoDataFrame(columns=['p_barr', 'n_barr', 'a_rivers', 'w_parks'])

# along NEGATIVE BARRIERS
tmp = barriers_within[barriers_within['type'].isin(['railway', 'road', 'secondary_road'])]
edges_graph['n_barr'] = edges_graph.apply(lambda row: ci.barriers_along(row['edgeID'], edges_graph,

```

```

        tmp, sindex,
        offset = 25), axis = 1)

# crossing any kind of barrier but parks - STRUCTURING BARRIERS
edges_graph = ci.assign_structuring_barriers(edges_graph, barriers_within)

# %% [markdown]
# ## Visualisation

# %%
# positive barriers
edges_graph = edges_graph.assign(p_bool=0).astype({'p_bool': int})
edges_graph['p_bool'] = edges_graph.apply(lambda row: True if len(row['p_barr']) > 0 else False, axis = 1)
tmp = edges_graph[edges_graph.p_bool == True].copy()
#fig = ci.plot_gdf(tmp, black_background = False, figsize = 15, color = 'red', title = city_name+':
Streets along parks and rivers',
#                  legend = False, base_map_gdf = edges_graph, base_map_alpha = 0.3, base_map_color = 'black')

# %%
# negative barriers
edges_graph = edges_graph.assign(n_bool=0).astype({'n_bool': int})
edges_graph['n_bool'] = edges_graph.apply(lambda row: True if len(row['n_barr']) > 0 else False, axis = 1)
tmp = edges_graph[edges_graph.n_bool == True].copy()
#fig = ci.plot_gdf(tmp, black_background = False, figsize = 15, color = 'red',
title = city_name+': Streets along negative barriers',
#                  legend = False, base_map_gdf = edges_graph, base_map_alpha = 0.3, base_map_color = 'black')

# %% [markdown]
# ### Save

# %%
# convertin list fields to string
#to_convert = ['a_rivers', 'w_parks', 'n_barr', 'p_barr']
edges_graph_string = edges_graph.copy()
#for column in to_convert:
#    edges_graph_string[column] = edges_graph_string[column].astype(str)

saving_path = 'Outputs/' + city_name + '/'
edges_graph_string.to_file(saving_path + "edges.shp", driver='ESRI Shapefile')
nodes_graph.to_file(saving_path + "nodes.shp", driver='ESRI Shapefile')

-----06-EmpiricalABM_Evaluation.ipynb-----
# %%
import pandas as pd, numpy as np, seaborn as sns, geopandas as gpd, matplotlib.pyplot as plt
from scipy.stats import f_oneway
import pingouin as pin
import matplotlib

```

```
from shapely.geometry import Point, mapping
%matplotlib inline

import warnings
warnings.simplefilter(action="ignore")

pd.set_option("display.precision", 2)
pd.options.display.float_format = '{:20.2f}'.format
pd.set_option('display.float_format', lambda x: '%.3f' % x)
pd.options.mode.chained_assignment = None

import cityImage as ci, ABManalysis as af

#libraries for clustering
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.metrics import silhouette_samples, silhouette_score

import os

import fastkde

# %% [markdown]
# ### Preparing and cleaning the routes

# %% [markdown]
# ### Coordinate System of the case study area for cartographic visualisations

# %%
print(os.getcwd())
city_name = 'UPB'
epsg = 4326
crs = 'EPSG:' + str(epsg)

# %% [markdown]
# # Questionnaire
# ## 1. Pre-processing

# %%
raw = pd.read_csv(r"C:\Users\RNGM\PedSimCity-EvaluationP2\Input\empiricalABM\UPB_responses_matrix_71.csv")
# raw responses
raw['startdate'] = raw.apply(lambda row: row['startdate'][11:], axis = 1)
raw['datestamp'] = raw.apply(lambda row: row['datestamp'][11:], axis = 1)
raw['duration'] = raw.apply(lambda row: af.compute_duration(row['startdate'], row['datestamp']), axis = 1)
raw['PD2'] = raw['PD2'].fillna(raw['PD2'].mean())
raw['age'] = raw['PD2'].astype(int)
print('Number of total participants:', str(len(raw)))
```

```

# %% [markdown]
# ### 1.1 Disregard subjects who took less than n minutes to complete the study (Not yet needed.)

# %%
# cleaning
raw = raw[raw['duration'] >= 1].copy()
print('Number of participants whose record was not disregarded ', str(len(raw)))
limit = {'A1':18, 'A2':25, 'A3':33, 'A4':41, 'A5':49, 'A6':57, 'A7':65, 'A8':73, 'A9':150}
# print("age issue", len(raw[~raw.FQ1.isnull()]['PD2', 'FQ1', 'limit'][raw.age > raw.limit]) +
#       len(raw[~raw.FQ1.isnull()]['PD2', 'FQ1', 'limit'][raw.age < raw.limit-7]))

# %%
raw.rename(columns={"PD1": "sex"}, inplace = True)
values = ['A1', 'A2', 'A3', 'A4']
new_values = ["male", "female", "non-binary", "prefer not to"]
for n, value in enumerate(values):
    raw['sex'].replace(value, new_values[n], inplace = True)

# %% [markdown]
# ### 1.1 Disregarding subjects who always chose the same response when asked how to proceed along the route

# %%
def check(index):
    response = list(raw[to_check].loc[index][raw[to_check].loc[index].notna()])
    percentageA1 = response.count("A1")/len(response)
    percentageA2 = response.count("A2")/len(response)
    if ((percentageA1 > 0.65) | (percentageA2 > 0.65)) & (raw.loc[index].duration < 30):
        return True
    return False

# checking videos
video_columns = [col for col in raw if col.startswith('VD')]
to_remove = ['VD000a', 'VD000', 'VD001[SQ001]', 'VD002a', 'VD002', 'VD003[SQ001]', 'VD004a', 'VD004', 'VD005[SQ001]']
#to_check = [item for item in video_columns if item not in to_remove]
to_check = [item for item in video_columns]
raw['allSame'] = raw.apply(lambda row: check(row.name), axis = 1)

# %% [markdown]
# ## 2. Demographic information

# %% [markdown]
# ### 2.1 General information

# %%
nr = 'Number'
pr = 'participants'

```

```

N = len(raw)
print(nr+' of female '+pr +': '+str(len(raw[raw.sex == 'female']))+',',
      '+str(round(len(raw[raw.sex == 'female'])/N*100,1))+%'')
print(nr+' of male '+pr +': '+str( len(raw[raw.sex == 'male']))+',',
      '+str(round(len(raw[raw.sex == 'male'])/N*100,1))+%'')
print(nr+' of non-binary '+pr +': '+str(len(raw[raw.sex == 'non-binary']))+',',
      '+str(round(len(raw[raw.sex == 'non-binary'])/N*100, 1))+%'')
print(nr+' of '+pr+' who preferred not to declare their gender:
      '+str(len(raw[raw.sex == 'prefer not to']))+', '+
      str(round(len(raw[raw.sex == 'prefer not to'])/N*100, 1))+%'')
print()
print("Participants' mean age:", round(raw['age'].mean(),3))
print("Participants' std age:", round(raw['age'].std(), 3))
print()
print('Mean duration:', str(round(raw.duration.mean(), 2))+" minutes")

# %% [markdown]
# ### 2.2 Age categories

# %%
Ga = len(raw[raw.age < 18])
Gb = len(raw[(raw.age >= 18) & (raw.age <= 25)])
Gc = len(raw[(raw.age >= 26) & (raw.age <= 33)])
Gd = len(raw[(raw.age >= 34) & (raw.age <= 41)])
Ge = len(raw[(raw.age >= 42) & (raw.age <= 49)])
Gf = len(raw[(raw.age >= 50) & (raw.age <= 57)])
Gg = len(raw[(raw.age >= 58) & (raw.age <= 65)])
Gh = len(raw[(raw.age >= 66) & (raw.age <= 73)])
Gi = len(raw[(raw.age >= 74)])

print("Number of participants per age group")
print()
print('< 18: '+str(Ga)+', '+str(round(Ga/N*100, 1))+ '%')
print('18 - 25: '+str(Gb)+', '+str(round(Gb/N*100, 1))+ '%')
print('26 - 33: '+str(Gc)+', '+str(round(Gc/N*100, 1))+ '%')
print('34 - 41: '+str(Gd)+', '+str(round(Gd/N*100, 1))+ '%')
print('42 - 49: '+str(Ge)+', '+str(round(Ge/N*100, 1))+ '%')
print('50 - 57: '+str(Gf)+', '+str(round(Gf/N*100, 1))+ '%')
print('58 - 65: '+str(Gg)+', '+str(round(Gg/N*100, 1))+ '%')
print('66 - 73: '+str(Gh)+', '+str(round(Gh/N*100, 1))+ '%')
print('> 74:' +str(Gi)+', '+str(round(Gi/N*100, 1))+ '%')

# %% [markdown]
# ### 2.3 Links or relationship to the case-study area (UPB)

# %%

```

```

Ga = len(raw[~raw['PD3[SQ001]'].isnull()])
Gb = len(raw[~raw['PD3[SQ002]'].isnull()])
Gc = len(raw[~raw['PD3[SQ003]'].isnull()])

nr = "Number of participants who "

print(nr+'Study in UPB: '+str(Ga)+', '+str(round(Ga/N*100,1))+ '%')
print(nr+'Work in UPB: '+str(Gb)+', '+str(round(Gb/N*100,1))+ '%')
print(nr+'Occasionally visit UPB: '+str(Gc)+', '+str(round(Gc/N*100,1))+ '%')

# %% [markdown]
# ### 2.4 Reasons for walking

# %%
columns = ['WB2[SQ001]', 'WB2[SQ002]']
labels = ['Commuting to/from work', 'Commuting to/from university']

#'WB2[SQ003]', 'WB2[SQ004]', 'WB2[SQ005]',
#'For social activities', 'For exercise or free-time activities', 'For other daily errands and commitments'

for column in columns:
    raw[column].replace('A1', 1, inplace = True)
    raw[column].replace('A2', 2, inplace = True)
    raw[column].replace('A3', 3, inplace = True)
    raw[column].replace('A4', 4, inplace = True)
    raw[column].replace('A5', 5, inplace = True)

total_prob = 0.0
for n, column in enumerate(columns):
    prob = round(raw[column].mean(),3)
    total_prob += prob
    print(labels[n]+':', str(round(raw[column].mean(),3))) # value on the Likert scale
    print("percentage: ", str(round(raw[column].mean()*100/5,3))+'%')
    if n < len(columns)-1:
        print()

# %% [markdown]
# ##### *Reasons for Walking: Transformation into probabilities*

# %%
for n, column in enumerate(columns):
    actual_prob = round(raw[column].mean()/total_prob,2)
    print(str(labels[n]) + ': '+str(actual_prob))

# %% [markdown]
# ## 3. Familiariaty, spatial knowledge and preferences
# ### 3.1 Time spent living in the case-study area (*familiarity*) (SKIP)

```

```

# %%
# raw[['PD4[SQ001]', 'PD4[SQ002]']] = raw[['PD4[SQ001]', 'PD4[SQ002]']].fillna(0.0)
# raw['PD4[SQ002]'].replace(1984.0, 2021-1984, inplace = True)
# raw['familiarity'] = raw['PD4[SQ001]']/12 + raw['PD4[SQ002]']
# raw['familiarity'].plot.kde()

# %% [markdown]
# ### 3.2 Self-reported spatial knowledge

# %%
columns = ['WB4[SQ001]', 'WB4[SQ002]', 'WB4[SQ003]', 'WB4[SQ004]']
for column in columns:
    raw[column].replace('A1', 1, inplace = True)
    raw[column].replace('A2', 2, inplace = True)
    raw[column].replace('A3', 3, inplace = True)
    raw[column].replace('A4', 4, inplace = True)
    raw[column].replace('A5', 5, inplace = True)
raw['knowledge'] = (raw['WB4[SQ001]'] + raw['WB4[SQ002]'] + raw['WB4[SQ003]'] + raw['WB4[SQ004]'])/4
raw['knowledge'] = raw['knowledge'].fillna(0.00)
print(raw['knowledge'])
raw['knowledge'].plot.kde()

# %%
columns = ['WB4[SQ001]', 'WB4[SQ002]', 'WB4[SQ003]', 'WB4[SQ004]', 'knowledge']
raw[columns].corr(method='pearson')

# %% [markdown]
# ### 3.3 Preference for and aversion to barriers (SKIP)

# %%
columns = ['SP4[SQ004]', 'SP4[SQ005]', 'SP4[SQ007]']
for column in columns:
    raw[column].replace('A1', 0.00, inplace = True)
    raw[column].replace('A2', 0.25, inplace = True)
    raw[column].replace('A3', 0.50, inplace = True)
    raw[column].replace('A4', 0.75, inplace = True)
    raw[column].replace('A5', 1.00, inplace = True)

raw['preferenceNatural'] = raw['SP4[SQ004]'] ## preference for Natural Barriers
raw['preferenceNatural'] = raw['preferenceNatural'].fillna(0.00)
raw['aversionSevering'] = (raw['SP4[SQ005]'] + raw['SP4[SQ007]'])/2 ## Aversion to Severing Barriers
raw['aversionSevering'] = raw['aversionSevering'].fillna(0.00)

# %% [markdown]
# ## 4. Video Tasks: route choice behaviour variables
# ##### *Preliminary columns cleaning*

```

```

# %%
video_columns = [col for col in raw if col.startswith('VD')]
to_remove = ['VD000a', 'VD100a', 'VD200a']

columns = ['id'] + video_columns
columns = [item for item in columns if item not in to_remove] # remove not necessary video columns
responses = raw[columns].copy() # only response to the video tasks

# %%
# cleaning and rename columns and prepare a legible dataframe
for column in responses.columns:
    if column == 'id':
        continue
    responses.rename({column: column[2:]}, axis=1, inplace = True)

for column in responses.columns:
    if column == 'id':
        continue
    if ('a' in column) | ('b' in column):
        continue
    new_column = '-'.join(column[i:i+3] for i in range(0, len(column), 3))
    responses.rename({column: new_column}, axis=1, inplace = True)

# %% [markdown]
# ##### *Loading the routes used in the survey*

# %%
survey_routes = gpd.read_file("Outputs/Routes_sections.shp").to_crs(crs)
summary = pd.DataFrame(columns = ['id']+list(survey_routes.video.unique()))

summary = summary.reindex(sorted(summary.columns), axis=1)

for row in responses.itertuples():
    sectors = responses.loc[row.Index].notna().dot(responses.columns+',').rstrip(',')
    sectors = sectors.replace('-', ',')
    sectors_list = sectors.strip('').split(',')
    for sector in sectors_list:
        summary.at[row.Index, sector] = 1
        summary.at[row.Index, 'id'] = responses.loc[row.Index].id

summary.fillna(0, inplace = True)
for column in summary.columns:
    summary.rename(columns={column: str(column)}, inplace = True)

# %%

```

```

summary.head(69)

# %% [markdown]
# ### 4.1 Obtaining the general statistics

# %%
survey_routes['routeChoice'] = survey_routes.apply(lambda row: row['routeChoic'].replace(" ",""), axis = 1)
survey_routes['routeChoice'] = survey_routes.apply(lambda row: row['routeChoice'].strip('][').split(','),
                                                 axis = 1)
survey_routes.drop('routeChoic', inplace = True, axis = 1)

# %% [markdown]
# ##### *Computing for each subject how much they resorted to certain urban elements or road costs*

# %%
route_variables = ['usingElements','noElements', 'distanceHeuristic', 'routeMarks', 'barriers',
'preferenceNatural', 'aversionSevering']
other_variables = ["length", "minimisation_length", "combined_length"]

#'usingElements','noElements', 'onlyAngular', 'distanceHeuristic', 'angularHeuristic', 'regions', 'barriers',
'distantLandmarks', 'preferenceNatural', 'aversionSevering'
# "minimisation_length","combined_length"
video0 = [col for col in summary if col.startswith('0')]
video1 = [col for col in summary if col.startswith('1')]
video2 = [col for col in summary if col.startswith('2')]
videos = [video0, video1, video2]
routes_stats = af.set_routes_stats(summary, survey_routes, videos)

# %%
## appending variables regarding the preference for natural/severing barriers and spatial knowledge

raw.index = raw['id']
routes_stats.index.name = None
routes_stats['preferenceNatural'] = raw['preferenceNatural']
routes_stats['aversionSevering'] = raw['aversionSevering']
routes_stats['knowledge'] = raw['knowledge']
routes_stats.head(70)

# %% [markdown]
# ##### *Obtaining the input matrix for cluster analysis*

# %%
input_matrix = routes_stats[route_variables].copy()
input_matrix.replace(0.0, 0.01, inplace = True)
cluster_variables = ['distanceHeuristic', 'routeMarks', 'barriers']
#'onlyDistance', 'onlyAngular', 'distanceHeuristic', 'angularHeuristic', 'regions', 'routeMarks',
'barriers','distantLandmarks'

```

```

for column in cluster_variables:
    if column in ['onlyDistance', 'onlyAngular']:
        input_matrix[column] = input_matrix[column] * input_matrix['noElements']
    else:
        input_matrix[column] = input_matrix[column] * input_matrix['usingElements']
input_matrix = input_matrix.astype(float)

print(input_matrix)

# %% [markdown]
# ### 4.2 Visualising overall importance of each urban element (probabilities)

# %%
sns.set()
sns.set_color_codes()
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize=(30, 12))

## obtaining dataframe for boxplot visualisation
tab = pd.DataFrame(columns = ['variable', 'value'])
labels = ['Using\nUrban Elements', 'Only Minimization\nHeuristic', 'Distance', 'Landmarks', 'Barriers',
          'Natural\nBarriers', 'Severing\nBarriers']

index = 0
for subject in input_matrix.index:
    for n, variable in enumerate(route_variables):
        tab.at[index, 'variable'] = labels[n]
        tab.at[index, 'value'] = input_matrix.loc[subject][variable]
        index += 1

palette = ['blue']*2 + ['red']*8 + ['lime']*2
ax = sns.boxplot(x="variable", y="value", data=tab, palette = palette)
ax.set_ylim(0.0, 1.01)
ax.set_ylabel(' ', fontsize = 27, labelpad = 30, fontfamily = 'Times New Roman')
ax.set_xlabel(' ', labelpad = 30, fontfamily = 'Times New Roman')

for tick in ax.get_yticklabels():
    tick.set_fontname('Times New Roman')
for tick in ax.get_xticklabels():
    tick.set_fontname('Times New Roman')

ax.tick_params(axis='both', labelsize= 27, pad = 20)

# %%
fig.savefig("Outputs/Figures/empiricalABM/f4test_69responses.pdf", bbox_inches='tight')

# %%

```

```
print(cluster_variables)
input_matrix[cluster_variables].mean()
input_matrix[cluster_variables]

# %%
input_matrix[cluster_variables].corr(method='pearson')

# %% [markdown]
# ## 5. Cluster Analysis
# ### 5.1 Variables transformation

# %%
#X = input_matrix[cluster_variables].copy()
#fig = plt.figure(figsize = (20, 10))
#for n, column in enumerate(list(X.columns)):
#    ax = fig.add_subplot(2,4,n+1)
#    ax = X[column].plot.kde()
#    ax.set_title(column)

# Assuming input_matrix is a DataFrame and cluster_variables is a list of columns to plot
X = input_matrix[cluster_variables].copy()

if len(X) > 0 and X.shape[1] > 1:
    fig = plt.figure(figsize=(20, 10))
    for n, column in enumerate(list(X.columns)):
        if len(X[column]) > 1 and len(X[column].unique()) > 1:
            ax = fig.add_subplot(2, 4, n + 1)
            ax = X[column].plot(kind='kde', ax=ax)
            ax.set_title(column)

    plt.tight_layout()
    plt.show()

# %% [markdown]
# ##### *Logarithmic transformation and standardisation*

# %%
X_log = af.log_transf(input_matrix[cluster_variables])
X_log_stand = X_log.copy()

for column in X_log.columns:
    try:
        X_log_stand[column] = af.standardise_column(X_log, column)
    except ValueError as e:
        print(f"Error standardizing column {column}: {e}. Skipping...")
```

```

# %% [markdown]
# ### 5.2 Obtaining and comparing different clustered structures obtained with the k-means algorithm

# %%
list_scores = []

def pipe_kmeans(X_toFit):

    X_tmp = X_toFit.copy()
    for n_clusters in range(2, 8):
        clusterer = KMeans(n_clusters = n_clusters, n_init = 2000).fit(X_toFit)
        labels = clusterer.labels_

        score = round(silhouette_score(X_tmp, labels, metric= 'sqeuclidean'), 3)
        param_score = {'algorithm': 'k-means', 'n_clusters' : n_clusters, 'score' : score,
                      'clusterer' : clusterer}
        list_scores.append(param_score)

    pipe_kmeans(X_log_stand)
    clustering = pd.DataFrame(list_scores)
    clustering.sort_values(by = 'score', ascending = False)

# %%
#def pipe_kmeans_VCR(X):

#    VRCs = []
#    X_tmp = X.copy()
#    X_toFit = X.copy()

#    for n_clusters in range(2, 11):

#        clusterer = KMeans(n_clusters = n_clusters, n_init = 2000).fit(X_toFit)
#        X_tmp['cluster'] = clusterer.labels_

#        f = 0.0
#        for var in cluster_variables:
#            arrays = []
#            for cluster in list(X_tmp['cluster'].unique()):
#                arrays.append(X_tmp[var+'_log'][X_tmp.cluster == cluster])

#            if n_clusters == 2:
#                statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1])).statistic
#            if n_clusters == 3:
#                statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]),
#                                     np.array(arrays[2])).statistic
#            if n_clusters == 4:
#                statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]), np.array(arrays[2]),
#                                     np.array(arrays[3])).statistic
#            VRCs.append(statistic)
#    return VRCs

```

```

#                                     np.array(arrays[3])).statistic
#
#         if n_clusters == 5:
#             statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]), np.array(arrays[2]),
#                                   np.array(arrays[3]),
#                                   np.array(arrays[4])).statistic
#
#         if n_clusters == 6:
#             statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]), np.array(arrays[2]),
#                                   np.array(arrays[3]),
#                                   np.array(arrays[4]),np.array(arrays[5])).statistic
#
#         if n_clusters == 7:
#             statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]), np.array(arrays[2]),
#                                   np.array(arrays[3]),
#                                   np.array(arrays[4]),np.array(arrays[5]),np.array(arrays[6])).statistic
#
#         if n_clusters == 8:
#             statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]), np.array(arrays[2]),
#                                   np.array(arrays[3]),
#                                   np.array(arrays[4]), np.array(arrays[5]), np.array(arrays[6]),
#                                   np.array(arrays[7])).statistic
#
#         if n_clusters == 9:
#             statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]), np.array(arrays[2]),
#                                   np.array(arrays[3]),
#                                   np.array(arrays[4]),np.array(arrays[5]), np.array(arrays[6]), np.array(arrays[7]),
#                                   np.array(arrays[8])
#                               ).statistic
#
#         if n_clusters == 10:
#             statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]), np.array(arrays[2]),
#                                   np.array(arrays[3]),
#                                   np.array(arrays[4]), np.array(arrays[5]), np.array(arrays[6]), np.array(arrays[7]),
#                                   np.array(arrays[8]), np.array(arrays[9])).statistic

#
#         f += statistic
#
#         VRCs.append(f)
#
#     return VRCs

#VRCs = pipe_kmeans_VCR(X_log_stand)

def pipe_kmeans_VCR(X):

    VRCs = []
    X_tmp = X.copy()
    X_toFit = X.copy()

    for n_clusters in range(2, 9):

        clusterer = KMeans(n_clusters = n_clusters, n_init = 2000).fit(X_toFit)
        X_tmp['cluster'] = clusterer.labels_

```

```

f = 0.0
for var in cluster_variables:
    arrays = []
    for cluster in list(X_tmp['cluster'].unique()):
        arrays.append(X_tmp[var+'_log'][X_tmp.cluster == cluster])

if n_clusters == 2:
    statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1])).statistic
if n_clusters == 3:
    statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]),
                         np.array(arrays[2])).statistic
if n_clusters == 4:
    statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]),
                         np.array(arrays[2]),
                         np.array(arrays[3])).statistic
if n_clusters == 5:
    statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]),
                         np.array(arrays[2]),
                         np.array(arrays[3]),
                         np.array(arrays[4])).statistic
if n_clusters == 6:
    statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]),
                         np.array(arrays[2]),
                         np.array(arrays[3]),
                         np.array(arrays[4]),
                         np.array(arrays[5])).statistic
if n_clusters == 7:
    statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]),
                         np.array(arrays[2]),
                         np.array(arrays[3]),
                         np.array(arrays[4]),
                         np.array(arrays[5]),
                         np.array(arrays[6])).statistic
if n_clusters == 8:
    statistic = f_oneway(np.array(arrays[0]), np.array(arrays[1]),
                         np.array(arrays[2]),
                         np.array(arrays[3]),
                         np.array(arrays[4]),
                         np.array(arrays[5]),
                         np.array(arrays[6]),
                         np.array(arrays[7])).statistic
f += statistic
VRCs.append(f)
return VRCs

VRCs = pipe_kmeans_VCR(X_log_stand)

# %% [markdown]
# ### 5.3 Choosing the best structure

# %%
print("Clusters\tOmega\t\tVRC")
for n, VRC in enumerate(VRCs):
    if (n == 0) | (n == len(VRCs)-1):
        continue
    index = (VRCs[n+1]-VRC)-(VRC-VRCs[n-1])
    print(f"\t{n+2}\t\t{round(index, 3)}\t\t{round(VRC, 3)}")

```

```

# %%
chosen = clustering.loc[1]
labels_cluster = np.array(chosen.clusterer.labels_.copy())
labels_cluster += 1

# %%
input_with_cluster = input_matrix.copy()
input_with_cluster['cluster'] = labels_cluster

# %% [markdown]
# ### 5.4 Examining the chosen partition

# %%
input_with_cluster['knowledge'] = raw['knowledge']
cluster_stats = input_with_cluster.groupby("cluster").mean()
cluster_stats

# %%
sns.set()
sns.set_color_codes()

colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet', 'pink',
          'c', 'black']

clusters_stats = input_with_cluster.groupby(by = 'cluster').mean()
figsize = (22.5, (15/2*3))
fig = plt.figure(figsize = figsize)
plot_variables = ['routeMarks', 'barriers', 'distanceHeuristic']
cluster_v = np.array(plot_variables)
labels = ['Landmarks', 'Barriers', 'Distance']
plt.rcParams['font.family'] = 'Times New Roman'
to_plot = list(clusters_stats.index)

for cluster in to_plot:

    if cluster != 'population':
        tmp = clusters_stats.loc[cluster, cluster_v].values

    angles = np.linspace(0, 2*np.pi, len(labels), endpoint=False)

    # close the plot
    tmp = np.concatenate((tmp,[tmp[0]]))
    angles = np.concatenate((angles,[angles[0]]))

    if cluster != 'population':
        ax = fig.add_subplot(3, 3, cluster, polar = True)

```

```

color = colors[cluster-1]

ax.plot(angles, tmp, '- ', color = color, linewidth=2)
ax.fill(angles, tmp, color = color, alpha=0.25)
ax.set_thetagrids((angles * 180/np.pi)[0:len(plot_variables)], labels, fontsize = 15)
ax.yaxis.set_ticks([0.10, 0.20, 0.30, 0.40, 0.50])
ax.tick_params(axis='y', labelsize= 15)
ax.tick_params(axis='x', labelsize= 15, pad = 5)
ax.set_rlabel_position(150)

if cluster != 'population':
    nr = len(input_with_cluster[input_with_cluster.cluster == cluster])
    title = "Cluster "+str(cluster) + " (N = "+str(nr)+")"
    ax.set_title(title, va = 'bottom', fontsize = 20, pad = 50, fontfamily = 'Times New Roman')

fig.subplots_adjust(wspace=0.20, hspace = 0.60)

print(clusters_stats)

# %%
fig.savefig("Outputs/Figures/empiricalABM/f5_69.pdf", bbox_inches='tight')

# %% [markdown]
# ##### *Spatial skills - Do they explain the variation in the route choice behaviour variables?*

# %%
clusters = list(set(labels_cluster))
for cluster in clusters:
    knowledge = input_with_cluster[input_with_cluster.cluster == cluster]['knowledge']
    print("cluster", cluster, "spatial skills/knowledge mean:", round(knowledge.mean(), 3),
          "std:", round(knowledge.std(), 3))

# %%
tab = pd.DataFrame(columns=['cluster', 'variable', 'value'])
labels = ['Distance', 'Landmarks', 'Barriers', 'Knowledge']

index = 0
for subject in input_with_cluster.index:
    for n, variable in enumerate(cluster_variables + ['knowledge']):
        if n < len(labels):
            tab.at[index, 'variable'] = labels[n]
        else:
            tab.at[index, 'variable'] = 'Unknown'
    tab.at[index, 'cluster'] = "Cluster " + str(int(input_with_cluster.loc[subject]['cluster']))
    tab.at[index, 'value'] = input_with_cluster.loc[subject][variable]
    index += 1

```

```

tab['value'] = tab.value.astype(float)
tab.sort_values('cluster', inplace=True)

# %%
# Check if the value column has at least two unique values for the knowledge variable
if len(tab[tab.variable == 'knowledge']['value'].unique()) < 2:
    print("The value column does not have at least two unique values for the knowledge variable.")
else:
    # Perform the t-test
    pin.pairwise_ttests(data=tab[tab.variable == 'knowledge'], dv='value', between='cluster')

# %% [markdown]
# ##### *Demographic characteristics*

# %%
raw['cluster'] = labels_cluster
print('Age')
cluster_deom_stats = raw.groupby("cluster")['age'].mean()
print(cluster_deom_stats)

# %%
print('Gender')
cluster_deom_stats = raw.groupby(["cluster", 'sex'])['sex'].count()
for cluster in range(1, max(labels_cluster)+1):
    print("cluster", cluster)
    for gender in list(cluster_deom_stats.loc[cluster].index):
        print(gender, round(cluster_deom_stats.loc[cluster][gender]/cluster_deom_stats.loc[cluster].sum() *100, 1))
    print()

# %% [markdown]
# ## 6. Final input data for building agent typologies in the ABM

# %%
columns = []
for column in input_with_cluster.columns:
    if column in ['knowledge', 'cluster']:
        continue
    columns.append(column+"_mean")
    columns.append(column+"_std")

groups = ['GROUP'+str(cluster) for cluster in input_with_cluster['cluster'].unique()]
indexes = groups + ['POPULATION', 'NULLGROUP']
clusters_gdf = pd.DataFrame(index = indexes, columns = columns)
variables = route_variables

clusters = input_with_cluster['cluster'].unique()

```

```

for index in indexes:
    for variable in variables:
        if index not in ['POPULATION', 'NULLGROUP']:
            cluster = int(index[5:])
            clusters_gdf.at[index, variable+'_mean'] =
                input_with_cluster[input_with_cluster.cluster == cluster][variable].mean()
            clusters_gdf.at[index, variable+'_std'] =
                input_with_cluster[input_with_cluster.cluster == cluster][variable].std()
        else:
            clusters_gdf.at[index, variable+'_mean'] = input_with_cluster[variable].mean()
            clusters_gdf.at[index, variable+'_std'] = input_with_cluster[variable].std()

if index not in ['POPULATION', 'NULLGROUP']:
    cluster = int(index[5:])
    clusters_gdf.at[index, 'portion'] =
        len(input_with_cluster[input_with_cluster.cluster == cluster])/len(input_with_cluster)
else:
    clusters_gdf.at[index, 'portion'] = 1.00

# %%
clusters_gdf

# %%
## exporting the files
clusters_gdf.to_csv('Outputs/empiricalABM/clusters_71.csv') ## this is imported into the ABM
routes_stats.to_csv('Outputs/empiricalABM/routes_stats_71.csv')

-----06-EmpiricalABM_AB.ipynb-----
# %%
import matplotlib as mpl, pandas as pd, numpy as np, geopandas as gpd, matplotlib.pyplot as plt,
pingouin as pin
import networkx as nx

from math import sqrt
import pylab
import ast

import warnings
warnings.filterwarnings('ignore')

pd.set_option("display.precision", 2)
pd.options.display.float_format = '{:20.2f}'.format

import ABManalysis as af
import cityImage as ci
import gisVis as gv
import importlib

```

```

# %% [markdown]
# ## 1. Loading Data
# ##### *Coordinate System of the case study area for cartographic visualisations*

# %%
# initialise path, names, etc.
city_name = 'UPB'
epsg = 4326
crs = 'EPSG:' + str(epsg)

# %%
input_path = 'Input/empiricalABM/'
output_path = 'Outputs/empiricalABM/'

edges = gpd.read_file(input_path+'edges.shp', driver='ESRI Shapefile')
nodes = gpd.read_file(input_path+'nodes.shp', driver='ESRI Shapefile')

nodes.index, edges.index = nodes.nodeID, edges.edgeID
nodes.index.name, edges.index.name = None, None

# %% [markdown]
# #### 1.1 Loading the simulation's volumes and assigning them to the segments of the street network

# %%
summary_clusters = pd.read_csv(input_path+'clusters_71.csv', index_col = 0)
clusters = ['GROUP1', 'GROUP2', 'GROUP3']
configurations = ['NULLGROUP', 'POPULATION', 'HETERO']
columns = ['edgeID'] + summary_clusters.index.to_list() + ['HETERO']
runs = 20

# creating a list of dataframes, one dataframe per each run containing the street segments' volumes
df_list = []
for run in range(runs):
    # change the file name to the current date
    df_tmp = pd.read_csv(input_path+'streetVolumes/20240507_'+str(run)+'.csv')
    df_tmp['HETERO'] = df_tmp[clusters].sum(axis=1)
    for n, c in enumerate(columns):
        df_tmp.rename(columns={c: columns[n] + "_" + str(run)}, inplace=True)
    df_list.append(df_tmp)

df = pd.concat(df_list, axis = 1)
df['edgeID'] = df.edgeID_0
for run in range(runs):
    df.drop(['edgeID' + "_" + str(run)], axis = 1, inplace = True)

```

```

# creating a list of only 3 dataframes, one per configuration, from the previous list of dataframes
cf_df_list = []
for configuration in configurations:
    cf_df_list.append(df[[col for col in df if col.startswith(configuration)]+['edgeID']])

# %%
print('Configuration 1 DF')
cf_df_list[0].head(68)

# %%
print('Configuration 2 DF')
cf_df_list[1].head(68)

# %%
print('Configuration 3 DF')
cf_df_list[2].head(68)

# %% [markdown]
# ### 1.2 Computing the volumes per group/agent typology (Heterogeneous configuration)

# %%
# repeating the procedure but only considering the heterogeneous configuration, and differentiate the
# volumes by cluster
columns = ['edgeID'] + clusters
runs = 20

# creating a list of dataframes, one dataframe per each run containing the street segments' volumes
df_list_clusters = []
for run in range(runs):
    # change the file name to the current date
    df_tmp = pd.read_csv(input_path+'streetVolumes/20240507_'+str(run)+'.csv')
    for n, c in enumerate(columns):
        df_tmp.rename(columns={c: columns[n]+'_'+str(run)}, inplace=True)
    df_list_clusters.append(df_tmp)

df = pd.concat(df_list, axis = 1)
df['edgeID'] = df.edgeID_0
for run in range(runs):
    df.drop(['edgeID'+"_"+str(run)], axis = 1, inplace = True)

# creating a list of only 3 dataframes, one per configuration, from the previous list of dataframes
cf_df_list_clusters = []
for cluster in clusters:
    cf_df_list_clusters.append(df[[col for col in df if col.startswith(cluster)]+['edgeID']])

# %%

```

```

# aggregate the runs' values
edges = af.aggregate_runs(cf_df_list, configurations, edges, ddof = 0)
edges = af.aggregate_runs(cf_df_list_clusters, clusters, edges, ddof = 0)

# %% [markdown]
# ### 1.3 Computing *p_values* to assess to what extent the homogeneous and heterogeneous
configurations produce different results from the null configuration

# %%
# getting again the volumes from each run
df = pd.concat(df_list, axis = 1)
df['edgeID'] = df.edgeID_0
for run in range(runs):
    df.drop(['edgeID'+"_"+str(run)], axis = 1, inplace = True)

# %%
labels = ['null', 'homogeneous', 'heterogeneous']
edges_list = list(edges['edgeID'])

# creating a list of dataframes. Each dataframe represents a street segment.
# It contains the volumes of each run,
# per each configuration
df_edges = []
for edge in edges_list:
    df_tmp = pd.DataFrame(columns=['value','configuration'])

    for n, df in enumerate(df_list):
        df = df.copy()
        df.index = df['edgeID_'+str(n)]
        row = df.loc[edge]
        values = [row['NULLGROUP_'+str(n)], row['POPULATION_'+str(n)], row['HETERO_'+str(n)]]
        for nn, value in enumerate(values):
            df_tmp.loc[-1] = [value, labels[nn]]
        df_tmp.index = df_tmp.index + 1

    df_tmp['value'] = df_tmp['value'].astype(int)
    df_edges.append(df_tmp)

# %%
print("Example of a street segment's df (3 configurations x 20 runs = 60 values)")
df_edges[0].head(69)

# %%
edges['null_pvalue'] = 0.000
edges['hetero_pvalue'] = 0.000
edges['homo_pvalue'] = 0.000
edges['homo_pvalue_cl'] = "No significant diff."

```

```

edges['hetero_pvalue_cl'] = "No significant diff."

# performing wilcoxon test per each edge and
for n, df_edge in enumerate(df_edges):
    x = df_edge[df_edge['configuration'] == 'null']['value'].to_list()
    y = df_edge[df_edge['configuration'] == 'heterogeneous']['value'].to_list()
    z = df_edge[df_edge['configuration'] == 'homogeneous']['value'].to_list()

    # null vs heterogeneous
    if x != y:
        hetero = pin.wilcoxon(x = x, y = y, alternative='two-sided')
        hetero_pvalue = hetero.iloc[0]['p-val']
    else:
        hetero_pvalue = 1.0

    # null vs homogeneous
    if x != z:
        homo = pin.wilcoxon(x = x, y = z, alternative='two-sided')
        homo_pvalue = homo.iloc[0]['p-val']
    else:
        homo_pvalue = 1.0

    # assigning the result to the edges gdf
    edge = edges_list[n]
    edges.at[edge, 'hetero_pvalue'] = hetero_pvalue
    edges.at[edge, 'homo_pvalue'] = homo_pvalue

    if edges.loc[edge]['hetero_pvalue'] <= 0.05:
        if edges.loc[edge]['hetero_pvalue'] > edges.loc[edge]['null_pvalue']:
            color_hetero = "Significant diff. (+)"
        else:
            color_hetero = "Significant diff. (-)"
        edges.at[edge, 'hetero_pvalue_cl'] = color_hetero

    if edges.loc[edge]['homo_pvalue'] <= 0.05:
        if edges.loc[edge]['POPULATION'] > edges.loc[edge]['null_pvalue']:
            color_homo = "Significant diff. (+)"
        else:
            color_homo = "Significant diff. (-)"
        edges.at[edge, 'homo_pvalue_cl'] = color_homo

# %%
## Saving volumes
edges.to_file(output_path+'pedSim_empirical_71.shp', driver='ESRI Shapefile')

# %% [markdown]
# ### 1.2 Loading the simulation's individual routes

```

```

# %%
## reloading routes if already processed
#routes_gdfs = []
#n = 1
#for n, configuration in enumerate(configurations):
#    routes_gdf = gpd.read_file(output_path+'_'+configuration+'_routes.shp', driver='ESRI Shapefile')
#    routes_gdf = gpd.read_file(output_path+'routes/20240507_'+str(n)+'.shp', driver='ESRI Shapefile')
#    routes_gdf['edgeIDs'] = routes_gdf.apply(lambda row: af.get_edgesID(row, routes_gdf.columns), axis = 1)
#    routes_gdfs.append(routes_gdf)

# %%
# or processing them from the simulation output
input_path_routes = 'Input/empiricalABM/routes/'
routes_gdfs = []

for run in range(runs):
    # change the file name to the current date
    run_gdf = gpd.read_file(input_path_routes+'20240507_'+str(run)+'.shp')
    run_gdf.set_crs(crs)

    for n, configuration in enumerate(configurations):
        if configuration == 'HETERO':
            configuration_gdf = run_gdf[run_gdf.groupby().isin(clusters)].copy()
        else:
            configuration_gdf = run_gdf[run_gdf.groupby() == configuration].copy()
        if run == 0:
            routes_gdf = configuration_gdf.copy()
            routes_gdfs.append(configuration_gdf)
            continue

        routes_gdf = routes_gdfs[n].copy()
        routes_gdf = routes_gdf.append(configuration_gdf)
        routes_gdf.reset_index(inplace = True, drop=True)
        routes_gdf = routes_gdf.where(pd.notnull(routes_gdf), None)
        routes_gdf.replace({'None':None}, inplace = True)
        routes_gdfs[n] = routes_gdf

# %%
# preparing the routes gdf
for n, configuration in enumerate(configurations):
    #routes_gdf = routes_gdfs[n].copy()
    routes_gdf['edgeIDs'] = routes_gdf.apply(lambda row: af.get_edgesID(row, routes_gdf.columns), axis=1)
    routes_gdf['O'] = routes_gdf['O'].astype(int)
    routes_gdf['D'] = routes_gdf['D'].astype(int)
    routes_gdfs[n] = routes_gdf

```

```

# %% [markdown]
# ## 2. Route statistics
# ### 2.1 Deviation from the road-distance shortest path

# %%
## computing the deviation from the road-distance shortest path

#graph = ci.graph_fromGDF(nodes, edges, nodeID = "nodeID")
#distances = {}
#routes_gdf = routes_gdfs[0].copy()
#for routes_gdf in routes_gdfs:
#    for row in routes_gdf.itertuples():
#        source = routes_gdf.loc[row[0]]['O']
#        target = routes_gdf.loc[row[0]]['D']
#        if str(source)+"-"+str(target) in distances:
#            continue
#
#        distance = nx.shortest_path_length(graph, source=source, target=target, weight='length',
#                                            method='dijkstra')
#        distances[str(source)+"-"+str(target)] = distance

#####
graph = ci.graph_fromGDF(nodes, edges, nodeID="nodeID")
distances = {}

for sources_gdf in routes_gdfs:
    for row in sources_gdf.itertuples():
        source = sources_gdf.loc[row[0]]['O']
        target = sources_gdf.loc[row[0]]['D']
        if str(source) + "-" + str(target) in distances:
            continue
#
#        if not nx.has_path(graph, source, target):
#            distance = float('0')
#            # Or set a default value of your choice
#            # distance = 1000000 # Example
#        else:
#            try:
#                distance = nx.shortest_path_length(graph, source=source, target=target, weight='length',
#                                                    method='dijkstra')
#            except KeyError:
#                distance = float('0')
#
#        distances[str(source) + "-" + str(target)] = distance

# %% [markdown]
# #### Basic route statistics per configuration

```

```

# %%
labels = ['null', 'homogeneous', 'heterogeneous']
for n, configuration in enumerate(configurations):

    routes_gdf = routes_gdfs[n].copy()
    print()
    print("Within the", labels[n], "configuration:")
    print("median routes' length:", round(routes_gdf.geometry.length.median(),3))
    print("mean routes' length:", round(routes_gdf.geometry.length.mean(),3))
    routes_gdf['sp_dist'] = routes_gdf.apply(lambda row: distances[str(row['O'])+"-"+str(row['D'])]), axis = 1
    routes_gdf['sp_dev'] = routes_gdf.geometry.length/routes_gdf['sp_dist']
    print("median routes' deviation from SP", round(routes_gdf['sp_dev'].median(),3))
    routes_gdfs[n] = routes_gdf

# %%
labels = ['null', 'homogeneous', 'heterogeneous']
for n, configuration in enumerate(configurations):

    routes_gdf = routes_gdfs[n].copy()
    print()
    print("Within the", labels[n], "configuration:")
    print("median routes' length:", round(routes_gdf.geometry.length.median(),3))
    print("mean routes' length:", round(routes_gdf.geometry.length.mean(),3))
    routes_gdf['sp_dist'] = routes_gdf.apply(lambda row: distances[str(row['O'])+"-"+str(row['D'])]), axis = 1
    routes_gdf['sp_dev'] = routes_gdf.geometry.length/routes_gdf['sp_dist']
    print("median routes' deviation from SP", round(routes_gdf['sp_dev'].median(),3))
    routes_gdfs[n] = routes_gdf

# %% [markdown]
# ##### Basic route statistics per group

# %%
for cluster in list(clusters):
    tmp = routes_gdf[routes_gdf.group == cluster].copy()
    print()
    print("Within", cluster+":")
    print("median routes' length:", round(tmp.geometry.length.median(),3))
    print("mean routes' length:", round(tmp.geometry.length.mean(),3))
    tmp['sp_dist'] = tmp.apply(lambda row: distances[str(row['O'])+"-"+str(row['D'])]), axis = 1
    tmp['sp_dev'] = tmp.geometry.length/tmp['sp_dist']
    print("median routes' deviation from SP", round(tmp['sp_dev'].median(),3))

# %% [markdown]
# ### 2.2 Elements usage

# %% [markdown]
```

```

# ##### Elements usage per configuration

# %%
variables = ['barrier', 'routeMark', 'regions', 'distant']

for n, configuration in enumerate(configurations):
    print()
    print("Within the", labels[n], "configuration:")
    for variable in variables:
        l = len(routes_gdfs[n])
        print(variable, str(round(len(routes_gdfs[n][(routes_gdfs[n][variable] == 1)])/l*100,2))+"%")

    routes_gdfs[n]["using"] = routes_gdfs[n].barrier +routes_gdfs[n].routeMark + routes_gdfs[n].regions +
    routes_gdfs[n].distant
    print("using elements", str(round(len(routes_gdfs[n][(routes_gdfs[n]["using"] > 0)])/l*100,2))+"%")
    print("severing", round(routes_gdfs[n].severing.mean(),3))
    print("natural", round(routes_gdfs[n].natural.mean(),3))

# %% [markdown]
# ##### Elements usage per group

# %%
for cluster in clusters:
    print()
    print("Within", cluster+":")
    tmp = routes_gdfs[2][routes_gdfs[2].group == cluster].copy()
    l = len(tmp)
    for variable in variables:
        print(variable, str(round(len(tmp[tmp[variable] == 1])/l*100,2)) +"%")

    tmp["using"] = tmp.barrier +tmp.routeMark + tmp.regions + tmp.distant
    print("using elements", str(round(len(tmp[tmp["using"] > 0])/l*100,2))+"%")
    print("severing", round(tmp.severing.mean(),3))
    print("natural", round(tmp.natural.mean(),3))

# %%
# export the routes
for n, configuration in enumerate(configurations):
    routes_gdfs[n].drop(['edgeIDs'], axis = 1, errors = 'ignore').to_file(
        output_path+configuration+'_routes.shp', driver='ESRI Shapefile')

# %% [markdown]
# ## 3. Volumes

# %%
subtitles = ['Configuration 1: Random Non-empirical Path Selection']
norm = mpl.colors.Normalize(vmin = 0, vmax = 340)

```

```

fig = gv.plot_gdf(edges, column = configurations[0], cmap = gv.kindlmann(), norm = norm, title = subtitles[0],
figsize = (15,10),
    legend = False, cbar = True, cbar_ticks = 3, cbar_max_symbol = True, cbar_shrinkage = 1.0,
    geometry_size_factor = 0.05, axes_frame = True)

# Create a tabular representation of the graph
column = configurations[0]
edges_config = edges[[column]]
df = pd.DataFrame(edges_config)
print(df.head(47))

# %%
subtitles = ['Configuration 2: Average-based Empirical Path Selection']
norm = mpl.colors.Normalize(vmin = 0, vmax = 340)
fig = gv.plot_gdf(edges, column = configurations[1], cmap = gv.kindlmann(), norm = norm, title = subtitles[0],
figsize = (15,10),
    legend = False, cbar = True, cbar_ticks = 3, cbar_max_symbol = True, cbar_shrinkage = 1.0,
    geometry_size_factor = 0.05, axes_frame = True)

# Create a tabular representation of the graph
column = configurations[1]
edges_config = edges[[column]]
df = pd.DataFrame(edges_config)
print(df.head(47))

# %%
subtitles = ['Configuration 3: Cluster-based Empirical Path Selection']
norm = mpl.colors.Normalize(vmin = 0, vmax = 340)
fig = gv.plot_gdf(edges, column = configurations[2], cmap = gv.kindlmann(), norm = norm, title = subtitles[0],
figsize = (15,10),
    legend = False, cbar = True, cbar_ticks = 3, cbar_max_symbol = True, cbar_shrinkage = 1.0,
    geometry_size_factor = 0.05, axes_frame = True)

# Create a tabular representation of the graph
column = configurations[2]
edges_config = edges[[column]]
df = pd.DataFrame(edges_config)
print(df.head(47))

# %%
fig.savefig("Outputs/Figures/empiricalABM/f6_71.pdf", bbox_inches='tight')

# %%
subtitles = [ 'Configuration 1', 'Configuration 2', 'Configuration 3',]
cols = ['null', 'homo', 'hetero']
norm = mpl.colors.Normalize(vmin = 0, vmax = 340)

```

```

fig = gv.plot_gdf_grid(edges, ncols = 3, figsize = (20, 7), columns = configurations, cmap = gv.kindlmann(),
                       titles = subtitles, norm = norm, legend = False, cbar = True, cbar_ticks = 4,
                       cbar_max_symbol = True,
                       cbar_shrinkage = 0.70, geometry_size_factor = 0.04, axes_frame = True,
                       black_background = True)

# %%
for cluster in clusters:
    edges[cluster+"_%"] = edges[cluster]/(68*summary_clusters.loc[cluster].portion*5)*100

columns = [cluster+"%" for cluster in clusters]
titles = ['Cluster 1','Cluster 2','Cluster 3']

# Create a tabular representation of the graph
df = pd.DataFrame(edges, columns=columns)
print(df.head(47))

norm = mpl.colors.Normalize(vmin = 0, vmax = 100)
fig = gv.plot_gdf_grid(edges, ncols = 3, figsize = (20,10), columns = columns, titles = titles,
                       cmap = gv.kindlmann(),
                       norm = norm, legend = False, cbar = True, cbar_ticks = 3, cbar_max_symbol = True,
                       geometry_size_factor = 0.10, axes_frame = True, black_background = True)

# %%
fig.savefig("Outputs/Figures/empiricalABM/f8_71.pdf", bbox_inches='tight')

# %% [markdown]
# ## 3.1 Differences between configurations

# %%
importlib.reload(gv)

# %%
# building the colormap

from matplotlib.colors import LinearSegmentedColormap

black_l = gv.lighten_color('black', 0.75)
blue = gv.lighten_color('blue', 1.00)
red = gv.lighten_color('red', 1.00)
list_colors = [black_l, blue, red]

subtitles = ['Configuration 2', 'Configuration 3']
edges['homo_diff'] = abs(edges['POPULATION'] - edges['NULLGROUP'])
edges['hetero_diff'] = abs(edges['HETERO'] - edges['NULLGROUP'])

lw_columns = ['homo_diff', 'hetero_diff'] ## regulating line width

```

```

columns = ['homo_pvalue_cl', 'hetero_pvalue_cl']

cmap = LinearSegmentedColormap.from_list('custom_cmap', list_colors)
fig = gv.plot_gdf_grid(edges, columns = columns, classes = 5, cmap = cmap, titles = subtitles, legend = True,
                       axes_frame = True, scheme = None, geometry_size_columns = lw_columns,
                       geometry_size_factor = 0.10, black_background = True)

# Create a tabular representation of the graph
df = pd.DataFrame(edges, columns=['POPULATION', 'NULLGROUP', 'HETERO', 'homo_diff', 'hetero_diff'] + columns)
print(df.head(47))

# %%
fig.savefig("Outputs/Figures/empiricalABM/f7_71.pdf", bbox_inches='tight')

# %% [markdown]
# ## 3.2 Configuration volumes' statistics

# %%
import pysal.explore as ps
subtitles = [ 'Configuration 1', 'Configuration 2', 'Configuration 3',]

for n, configuration in enumerate(configurations):
    print()
    print(subtitles[n])
    print('mean is', round(edges[configuration].mean(),2))
    print('STD is', round(edges[configuration].std(),2))
    print('max is', round(edges[configuration].max(),2))
    sg = ps.inequality.gini.Gini(list(edges[configuration]))
    print('Gini coefficient is', round(sg.g,2))

-----ABManalysis.py-----

import pandas as pd, numpy as np, geopandas as gpd
import cityImage as ci
import math
from shapely.geometry import*
from shapely.ops import*
import ast
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt
import copy

def aggregate_runs(df_list, route_choice_models, edges_gdf, ddof = 0):
    """
    This function aggregates a series of dataframes generated by the ABM's runs.
    Per each passed route choice model, it extracts the volumes per street segment in each run,
    computes the median, mean and SD.
    """

```

Finally, it attaches such information, to the edges\_gdf file so to associate directly the volumes with the street segment.

```

Parameters
-----
df_list: List of Pandas DataFrame
    A list containing a dataframe per each run with the corresponding volumes
route_choice_models: List of String
    The list of the abbreviation of the route choice models
edges_gdf: LineString GeoDataFrame
    The GeoDataFrame of the edges of a street network
ddof: int
    the Delta Degrees of Freedom for computing the standard deviation

Returns
-----
LineString GeoDataFrame
"""

stat_runs = pd.DataFrame(index= df_list[0].index, columns=['edgeID'] + route_choice_models)
stat_runs['edgeID'] = df_list[0].edgeID
stat_runs.index = stat_runs['edgeID']

for n, df in enumerate(df_list):
    df.index = df['edgeID']
    tmp = df[[col for col in df if col.startswith(route_choice_models[n])]] # only run values
    df['median'] = tmp.median(axis = 1)
    df['mean'] = tmp.mean(axis = 1)
    df['std'] = tmp.std(axis = 1, ddof = ddof)

    stat_runs[route_choice_models[n]] = df["median"]
    stat_runs[route_choice_models[n]+'_std'] = df['std']
    stat_runs[route_choice_models[n]+'_mean'] = df['mean']

stat_runs.index.name = None
edges_gdf = pd.merge(edges_gdf, stat_runs, left_on = "edgeID", right_on = "edgeID", how = 'left')
edges_gdf.index = edges_gdf.edgeID
edges_gdf.index.name = None

return edges_gdf

def get_edgesID(row, columns):
"""
The ABM stores the sequence of edgeIDs traversed in each route in different columns,
given the 254 characters limit per field.
This function merges such lists in one List.

```

```

Parameters
-----
row: Pandas Series
    A row of the GeoDataFrame containing the routes
columns: List of String
    A routes GeoDataFrame's columns

Returns
-----
List of Integer
"""

edgeID_string = row['edgeIDs_0']
counter = 1
while True:
    if 'edgeIDs_'+str(counter) not in columns:
        break
    if row['edgeIDs_'+str(counter)] is None:
        break
    edgeID_string = edgeID_string + row['edgeIDs_'+str(counter)]
    counter += 1

return ast.literal_eval(edgeID_string)

def compute_deviation_from(edges_gdf, route_choice_models, comparison, max_err):
    """
    It computes the deviation of the volumes emerging from a series of scenarios (route_choice_models)
    from a term of comparison (e.g. observational counts, another scenario volumes).
    The term of comparison has to be a column of values in the edges_gdf GeoDataFrame.

    Parameters
    -----
    edges_gdf: LineString GeoDataFrame
        The GeoDataFrame of the edges of a street network
    route_choice_models: List of String
        The list of the abbreviations of the route choice models
    comparison: String
        The name of the column used for the comparison with the ABM volumes
    max_err: int
        The max standard deviation error to use as a bound

    Returns
    -----
    LineString GeoDataFrame
    """

```

```

for n, column in enumerate(route_choice_models):
    edges_gdf[column+'_std_err'] = 0
    edges_gdf[column+'_diff'] = 0

    for row in edges_gdf.iterrows():
        median = edges_gdf.loc[row[0]][column] #column = median of the criterion considered
        stdv = edges_gdf.loc[row[0]][column+'_std']

        diff = median-edges_gdf.loc[row[0]][comparison]

        if (median == 0) & (stdv == 0) & (edges_gdf.loc[row[0]][comparison] == 0):
            std_err = 0
        elif (stdv == 0) & (diff > 0):
            std_err = max_err
        elif (stdv == 0) & (diff < 0):
            std_err = -max_err
        elif (stdv == 0) & (diff == 0):
            std_err = 0
        else:
            std_err = diff/stdv

        # set columns

        edges_gdf.at[row[0], column+'_std_err'] = std_err
        edges_gdf.at[row[0], column+'_diff'] = abs(diff)

        edges_gdf[column+'_std_err'].replace(np.inf, max_err, inplace = True)
        edges_gdf[column+'_std_err'].replace(-np.inf, -max_err, inplace = True)

    return edges_gdf

def find_tracks_OD(index, track_geometry, unionStreet, nodes_gdf):
    """
    It determines the origin and destination of a GPS track.
    This should be cleaned and snapped to the street network.
    This function should be executed per row by means of the df.apply(lambda row : ...) function.

    Parameters
    -----
    index: int
        the track's index
    track_geometry: LineString
        The track's geometry
    unionStreet: MultiLineString
        The unary union of the street network GeoDataFrame
    nodes_gdf: Point GeoDataFrame
        The GeoDataFrame of the nodes of a street network
    """

```

```

>Returns
-----
tuple
"""

p_origin = Point(track_geometry.coords[0])
counter = 1
while (not p_origin.within(unionStreet)):
    p_origin = Point(track_geometry.coords[counter])
    counter += 1

x_origin, y_origin = track_geometry.coords[counter][0], track_geometry.coords[counter][1]
try: origin = nodes_gdf[(nodes_gdf.x == x_origin) & (nodes_gdf.y == y_origin)][['nodeID']].iloc[0].nodeID
except: origin = ci.distance_geometry_gdf(Point(x_origin, y_origin), nodes_gdf)[1]

p_destination = Point(track_geometry.coords[-1][0], track_geometry.coords[-1][1])
counter = -2
while (not p_origin.within(unionStreet)):
    p_destination = Point(track_geometry.coords[counter])
    counter += -1

x_destination, y_destination = track_geometry.coords[counter][0], track_geometry.coords[counter][1]
try: destination =
nodes[(nodes_gdf.x == x_destination) & (nodes_gdf.y == y_destination)][['nodeID']].iloc[0].nodeID
except: destination = ci.distance_geometry_gdf(Point(x_destination, y_destination), nodes_gdf)[1]

return (origin, destination)

def line_at_centroid(line_geometry, offset):
"""
Given a LineString, it creates a LineString that intersects the given geometry at its centroid.
The offset determines the distance from the original line.
This fictional line can be used to count precisely the number of trajectories intersecting a segment.
This function should be executed per row by means of the df.apply(lambda row : ...) function.

Parameters
-----
line_geometry: LineString
    A street segment geometry
offset: float
    The offset from the geometry

>Returns
-----
LineString
"""

```

```

left = line_geometry.parallel_offset(offset, 'left')
right = line_geometry.parallel_offset(offset, 'right')

if left.geom_type == 'MultiLineString': left = _merge_disconnected_lines(left)
if right.geom_type == 'MultiLineString': right = merge_disconnected_lines(right)

if (left.is_empty == True) & (right.is_empty == False): left = line_geometry
if (right.is_empty == True) & (left.is_empty == False): right = line_geometry
left_centroid = left.interpolate(0.5, normalized = True)
right_centroid = right.interpolate(0.5, normalized = True)

fict = LineString([left_centroid, right_centroid])
return(fict)

def count_at_centroid(line_geometry, trajectories_gdf):
    """
    Given a LineString geometry, it counts all the geometries in a LineString GeoDataFrame
    (the GeoDataFrame containing GPS trajectories).
    This function should be executed per row by means of the df.apply(lambda row : ...) function.

    Parameters
    -----
    line_geometry: LineString
        A street segment geometry
    tracks_gdf: LineString GeoDataFrame
        A set of GPS tracks

    Returns
    -----
    int
    """

    intersecting_tracks = trajectories_gdf[trajectories_gdf.geometry.intersects(line_geometry)]
    return len(intersecting_tracks)

def traversed_nodes(track_geometry, lines_at_centroids_gdf):
    """
    Given a GPS trajectory geometry and a GeoDataFrame of all the lines built at the street segments' centroid
    (see above and jupyter notebook), the function generates the list of nodes of a street network traversed by
    the given route. This function should be executed per row by means of the
    df.apply(lambda row : ...) function.

    Parameters
    -----
    track_geometry: LineString
        The track's geometry
    
```

```

lines_at_centroids_gdf: LineString GeoDataFrame
    A GeoDataFrame of LineString built perpendicular to the centroid of each street network's segment

Returns
-----
List
"""

tmp = lines_at_centroids_gdf[lines_at_centroids_gdf.geometry.intersects(track_geometry)]
edgeIDs = tmp['edgeID']
u = list(lines_at_centroids_gdf[lines_at_centroids_gdf.edgeID.isin(edgeIDs)].u)
v = list(lines_at_centroids_gdf[lines_at_centroids_gdf.edgeID.isin(edgeIDs)].v)
traversed_nodes = list(set(u+v))
try: traversed_nodes.remove(origin)
except: pass
try: traversed_nodes.remove(destination)
except: pass

return traversed_nodes


def _merge_disconnected_lines(list_lines):
"""
Given a list of LineStrings, even disconnected, it merges them in one LineString.

Parameters
-----
list_lines: List of LineString
    A list of LineString to connect

Returns
-----
LineString
"""

new_line = []
for n, i in enumerate(list_lines):
    coords = list(i.coords)
    if n < len(list_lines)-1: coords.append(list_lines[n+1].coords[-1])
    new_line = new_line + coords

line_geometry = LineString([coor for coor in new_line])
return(line_geometry)

def generate_routes_stats(routes_gdf_list, route_choice_models, labels, titles = None):
"""
This function generate a translated table that can be used for statistical tests

```

(e.g. Anova, Games-Howell test), for each passed route choice model.

Parameters

-----

`routes_gdf_list: List of GeoDataFrames`

A list containing a GeoDataFrame of routes per each route choice model

`route_choice_models: List of String`

The list of the abbreviation of the route choice models for which the statistics are desired

`labels: List of String`

The labels of the variables on which the statistics should be computed.

`titles: List of String`

The titles of the variables investigated (for visualisation purposes).

Returns

-----

`Pandas DataFrame`

"""

```
if titles == None: titles = labels
```

```
length = len(routes_gdf_list[0])
```

```
route_choice_models_col = []
```

```
values = []
```

```
type_stats = []
```

```
for n, label in enumerate(labels):
```

```
    for nn, model in enumerate(route_choice_models):
```

```
        route_choice_models_col = route_choice_models_col + [model] * length
```

```
        type_stats = type_stats+ [titles[n]] * length
```

```
        col = list(routes_gdf_list[nn][label])
```

```
        values = values + col
```

```
route_stats = pd.DataFrame({'routeChoice': route_choice_models_col, 'values': values, 'type': type_stats})
```

```
return route_stats
```

```
def local_landmarkness_track(traversed_nodes, nodes_gdf, buildings_gdf):
```

```
    """
```

It computes the accumulated local landmarkness of a route (a sequence of traversed nodes).

This is designed for computing the local landmarkness of a GPS trajectory.

This function should be executed per row by means of the `df.apply(lambda row : ...)` function.

Parameters

-----

`traversed_nodes: List of Node`

The list of traversed nodes by a track

`nodes_gdf: Point GeoDataFrame`

The GeoDataFrame of the nodes of a street network

```

buildings_gdf: Polygon GeoDataFrame
    The GeoDataFrame of the buildings of a city, with landmark scores

Returns
-----
float
"""

lL = 0.0
for n in traversed_nodes:
    if len(nodes_gdf.loc[n].loc_land) == 0: continue
    node_ll = buildings_gdf[buildings_gdf.buildingID.isin(nodes_gdf.loc[n].loc_land)]['lScore_sc'].max()
    lL += node_ll

return lL


def global_landmarkness_track(destination, traversed_nodes, nodes_gdf, buildings_gdf):
    """
    It computes the accumulated global landmarkness of a route (a sequence of traversed nodes).
    This is designed for computing the global landmarkness of a GPS trajectory.
    This function should be executed per row by means of the df.apply(lambda row : ...) function.

Parameters
-----
destination: int
    The nodeID of the destination node of a GPS track
traversed_nodes: List of Node
    The list of traversed nodes by a track
nodes_gdf: Point GeoDataFrame
    The GeoDataFrame of the nodes of a street network
buildings_gdf: Polygon GeoDataFrame
    The GeoDataFrame of the buildings of a city, with landmark scores

Returns
-----
float
"""

destination_node = nodes_gdf.loc[destination]
destination_geometry = destination_node.geometry
anchors = destination_node.anchors
if len(anchors) == 0: return 0.0

else:
    gL = 0.0

```

```

        for n in traversed_nodes:
            node = nodes_gdf.loc[n]
            if node.geometry.distance(destination_geometry) <= 300:
                continue
            visible_anchors = [item for item in node.dist_land if item in anchors]
            if len(visible_anchors) == 0:
                continue

        else:
            node_geometry = node.geometry
            distance_destination = destination_geometry.distance(node_geometry)
            node_gL = 0.0

            for building in visible_anchors:
                landmark = buildings_gdf.loc[building]
                score = landmark.gScore_sc;
                distance_landmark = destination_geometry.distance(landmark.geometry)
                if (distance_landmark == 0.0):
                    distance_landmark = 0.1
                distance_weight = distance_destination/distance_landmark
                if (distance_weight > 1.0):
                    distance_weight = 1.0
                score = score*distance_weight;
                if score > node_gL:
                    node_gL = score

            gL += node_gL

    return gL

## Landmarkness on routes

def local_landmarkness_route(row, nodes_gdf, edges_gdf, buildings_gdf):
    """
    It computes the accumulated local landmarkness of a route generated by an ABM.
    This function should be executed per row by means of the df.apply(lambda row : ...) function.

    Parameters
    -----
    row: Pandas Series
        A row of the GeoDataFrame containing the routes
    nodes_gdf: Point GeoDataFrame
        The GeoDataFrame of the nodes of a street network
    edges_gdf: LineString GeoDataFrame
        The GeoDataFrame of the edges of a street network
    buildings_gdf: Polygon GeoDataFrame
        The GeoDataFrame of the buildings of a city, with landmark scores
    """

```

```

Returns
-----
float
"""

origin = row.O
destination = row.D

u = list(edges_gdf[edges_gdf.edgeID.isin(row.edgeIDs)].u)
v = list(edges_gdf[edges_gdf.edgeID.isin(row.edgeIDs)].v)
traversed_nodes = list(set(u+v))
if origin in traversed_nodes: traversed_nodes.remove(origin)
if destination in traversed_nodes: traversed_nodes.remove(destination)

lL = 0.0
for n in traversed_nodes:
    if len(nodes_gdf.loc[n].loc_land) == 0: continue
    node_ll = buildings_gdf[buildings_gdf.buildingID.isin(nodes_gdf.loc[n].loc_land)]['lScore_sc'].max()
    lL += node_ll

return lL

def global_landmarkness_route(row, nodes_gdf, edges_gdf, buildings_gdf):
"""
It computes the accumulated global landmarkness of a route generated by an ABM.
This function should be executed per row by means of the df.apply(lambda row : ...) function.

Parameters
-----
row: Pandas Series
        A row of the GeoDataFrame containing the routes
nodes_gdf: Point GeoDataFrame
        The GeoDataFrame of the nodes of a street network
edges_gdf: LineString GeoDataFrame
        The GeoDataFrame of the edges of a street network
buildings_gdf: Polygon GeoDataFrame
        The GeoDataFrame of the buildings of a city, with landmark scores

Returns
-----
float
"""

origin = row.O
destination = row.D

```

```

destination_node = nodes_gdf.loc[destination]
destination_geometry = destination_node.geometry
anchors = destination_node.anchors
if len(anchors) == 0: return 0.0
else:
    gL = 0.0

    u = list(edges_gdf[edges_gdf.edgeID.isin(row.edgeIDs)].u)
    v = list(edges_gdf[edges_gdf.edgeID.isin(row.edgeIDs)].v)
    traversed_nodes = list(set(u+v))
    if origin in traversed_nodes: traversed_nodes.remove(origin)
    if destination in traversed_nodes: traversed_nodes.remove(destination)

    for n in traversed_nodes:
        node = nodes_gdf.loc[n]
        if node.geometry.distance(destination_geometry) <= 300:
            continue
        visible_anchors = [item for item in node.dist_land if item in anchors]
        if len(visible_anchors) == 0:
            continue
        else:
            node_geometry = node.geometry
            distance_destination = destination_geometry.distance(node_geometry)
            node_gL = 0.0

            for building in visible_anchors:
                landmark = buildings_gdf.loc[building]
                score = landmark.gScore_sc;
                distance_landmark = destination_geometry.distance(landmark.geometry)
                if (distance_landmark == 0.0):
                    distance_landmark = 0.1
                distance_weight = distance_destination/distance_landmark
                if (distance_weight > 1.0):
                    distance_weight = 1.0;
                score = score*distance_weight;
                if score > node_gL:
                    node_gL = score

            gL += node_gL

    return gL

## Landmark integration

def assign_anchors_to_nodes(nodes_gdf, buildings_gdf, radius = 2000, threshold = 0.3, nr_anchors = 5):
    """
    The function assigns a set of anchoring or orienting landmark (within a certain radius) to each node in a

```

```

nodes GeoDataFrame.
Amongst the buildings GeoDataFrame, only the one with a global landmark score higher than a certain threshold
are considered.

The global landmarks around a node, within the radius, may work as orienting landmarks towards the node,
if the landmark is visible in other locations across the city.

Parameters
-----
nodes_gdf: Point GeoDataFrame
    The GeoDataFrame of the nodes of a street network
buildings_gdf: Polygon GeoDataFrame
    The GeoDataFrame of the buildings of a city, with landmark scores
radius: float
    It determintes the area within which the point of references for a node are searched
threshold: float
    It regulates the selection of global_landmarks buildings (lanmdarks),
    by filtering out buildings whose global landmark score is lower than the argument.
nr_anchors: int
    The maximum number of anchors per node. This speeds up further computations and also takes into
    account the fact that people may associate only a certain
    ammount of anchors with a destination (the anchors selected are the always the best in terms of score;
    e.g. if nr_anchors == 5, the best 5 anchors in terms of gScore_sc are kept).

Returns
-----
GeoDataFrame
"""

global_landmarks = buildings_gdf[buildings_gdf.gScore_sc >= threshold]
global_landmarks = global_landmarks.round({"gScore_sc":3})
sindex = global_landmarks.sindex

nodes_gdf["anchors"] = nodes_gdf.apply(lambda row: _find_anchors(row["geometry"], global_landmarks, sindex,
radius, nr_anchors), axis = 1)

return nodes_gdf

def _find_anchors(node_geometry, global_landmarks, global_landmarks_sindex, radius, nr_anchors):
"""
The function finds the set of anchoring or orienting landmark (within a certan radius) to a given node.

Parameters
-----
node_geometry: Point
    The geometry of the node considered
global_landmarks: Polygon GeoDataFrame
    The GeoDataFrame of the global_landmarks buildings of a city

```

```

global_landmarks_sindex: Rtree spatial index
    The spatial index on the GeoDataFrame of the global landmarks of a city
radius: float
    It determinates the area within which the point of references for a node are searched
nr_anchors: int
    The maximum number of anchors per node. This speeds up further computations and also takes into account
    the fact that people may associate only a certain
    ammount of anchors with a destination (the anchors selected are the always the best in terms of score;
    e.g. if nr_anchors == 5, the best 5 anchors in terms of gScore_sc are kept).

>Returns
-----
List
"""

list_anchors = []
b = node_geometry.buffer(radius)
possible_matches_index = list(global_landmarks_sindex.intersection(b.bounds))
possible_matches = global_landmarks.iloc[possible_matches_index]
precise_matches = possible_matches[possible_matches.intersects(b)]

if len(precise_matches) == 0:
    pass
else:
    precise_matches.sort_values(by = "gScore_sc", ascending = False, inplace = True)
    anchors = precise_matches.iloc[0:nr_anchors]
    list_anchors = anchors["buildingID"].tolist()

return list_anchors

def assign_3d_visible_landmarks_to_nodes(nodes_gdf, buildings_gdf, sight_lines, threshold = 0.3):
"""
The function assigns to each node in a nodes' GeoDataFrame the set of visibile buildings,
on the basis of pre-computed 3d sight_lines.
Only global landmarks, namely buildings with global landmarkness higher than a certain threshold,
are considered as buildings.

Parameters
-----
nodes_gdf: Point GeoDataFrame
    The GeoDataFrame of the nodes of a street network
buildings_gdf: Polygon GeoDataFrame
    The GeoDataFrame of the buildings of a city, with landmark scores
sight_lines: LineString GeoDataFrame
    The GeoDataFrame of 3d sight lines from nodes to buildings.
    The nodeID and buildingID fields are expected to be in this GeoDataFrame,

```

```

        referring respectively to obeserver and target of the line
threshold: float
    It regulates the selection of global_landmarks buildings (lanmdarks),
    by filtering out buildings whose global landmark score is lower than the argument

Returns
-----
GeoDataFrame
"""

global_landmarks = buildings_gdf[buildings_gdf.gScore_sc >= threshold]
global_landmarks = global_landmarks.round({"gScore_sc":3})
index_global_landmarks = buildings_gdf["buildingID"].values.astype(int)
sight_lines_to_global_landmarks = sight_lines[sight_lines["buildingID"].isin(index_global_landmarks)]

nodes_gdf["dist_land"] = nodes_gdf.apply(lambda row: _find_visible_landmarks(row["nodeID"],
global_landmarks, sight_lines_to_global_landmarks), axis = 1)
return nodes_gdf

def _find_visible_landmarks(nodeID, global_landmarks, sight_lines_to_global_landmarks):
"""
The function finds the set of visibile buildings from a certain node,
on the basis of pre-computed 3d sight_lines.
Only global landmarks, namely buildings with global landmarkness higher than threshold,
are considered as buildings.

Parameters
-----
nodesID: int
    The nodeID of the node considered
global_landmarks: Polygon GeoDataFrame
    The GeoDataFrame of buildings considered global landmarks
sight_lines_to_global_landmarks: LineString GeoDataFrame
    The GeoDataFrame of 3d sight lines from nodes to buildings (only landmarks).
    The nodeID and buildingID fields are expected to be in this GeoDataFrame,
    referring respectively to obeserver and target of the line.

Returns
-----
List
"""

# per each node, the sight lines to the global_landmarks landmarks are extracted.
global_landmarks_list, scores_list = [], []
sight_node = sight_lines_to_global_landmarks[sight_lines_to_global_landmarks["nodeID"] == nodeID]
ix_global_landmarks_node = list(sight_node["buildingID"].values.astype(int))
global_landmarks_from_node = global_landmarks[global_landmarks["buildingID"].isin(ix_global_landmarks_node)]

```

```

global_landmarks_from_node.sort_values(by = "gScore_sc", ascending = False, inplace = True)
if len(global_landmarks_from_node) == 0:
    pass
else:
    global_landmarks_list = global_landmarks_from_node["buildingID"].tolist()

return global_landmarks_list

def assign_local_landmarks_to_nodes(nodes_gdf, buildings_gdf, radius = 50):
    """
    The function assigns a set of adjacent buildings (within a certain radius)
    to each node in a nodes GeoDataFrame.

    Parameters
    -----
    nodes_gdf: Point GeoDataFrame
        The GeoDataFrame of the nodes of a street network
    buildings_gdf: Polygon GeoDataFrame
        The GeoDataFrame of the buildings of a city, with landmark scores
    radius: float
        The radius which regulates the search of adjacent buildings

    Returns
    -----
    GeoDataFrame
    """

buildings_gdf = buildings_gdf.copy()
buildings_gdf = buildings_gdf.round({"gScore_sc":3})
sindex = buildings_gdf.sindex
nodes_gdf["loc_land"] = nodes_gdf.apply(lambda row: _find_local_landmarks(row["geometry"],
    buildings_gdf, sindex, radius), axis = 1)
return nodes_gdf

def _find_local_landmarks(node_geometry, buildings_gdf, buildings_gdf_sindex, radius):
    """
    The function finds the set of adjacent buildings (within a certain radius) for a given node.

    Parameters
    -----
    node_geometry: Point
        The geometry of the node considered
    buildings_gdf: Polygon GeoDataFrame
        The GeoDataFrame of the buildings of a city, with landmark scores
    buildings_gdf_sindex: Rtree spatial index
        The spatial index on the GeoDataFrame of the buildings of a city
    radius: float

```

```

The radius which regulates the search of adjacent buildings

Returns
-----
List
"""

list_local = []
b = node_geometry.buffer(radius)
possible_matches_index = list(buildings_gdf_sindex.intersection(b.bounds))
possible_matches = buildings_gdf.iloc[possible_matches_index]
precise_matches = possible_matches[possible_matches.intersects(b)]

if len(precise_matches) == 0:
    pass
else:
    precise_matches = precise_matches.sort_values(by = "lScore_sc", ascending = False).reset_index()
    list_local = precise_matches["buildingID"].tolist()
    list_scores = precise_matches["lScore_sc"].tolist()

return list_local


def regionBased_variables(edgeIDs, route_geometry, nodes_gdf, edges_gdf):
    """
    This function computes a set of variables for a certain ABM route,
    given the sequence of semgnets traversed and its geometry.
    It returns the portion walked along pedestrian streets, major roads,
    natural barriers, nr of regions traversed.

    Parameters
    -----
    edgeIDs: List of Integer
        The sequence of edgeIDs (segments) traversed by the route
    route_geometry: LineString or MultiLineString geometry
        The route's geometry
    nodes_gdf: Point GeoDataFrame
        The GeoDataFrame of the nodes of a street network
    edges_gdf: LineString GeoDataFrame
        The GeoDataFrame of the edges of a street network

    Returns
    -----
    Pandas DataFrame
    """

districts = {}
for edgeID in edgeIDs:

```

```

edgeID = int(edgeID)
u = edges_gdf.loc[edgeID].u
v = edges_gdf.loc[edgeID].v
length = edges_gdf.loc[edgeID].geometry.length
if nodes_gdf.loc[u].district == nodes_gdf.loc[v].district:
    d = nodes_gdf.loc[u].district
    districts[d] = round(districts.get(d, 0) + length, 2)

pedestrian_portion = edges_gdf[(edges_gdf.edgeID.isin(edgeIDs)) &
(edges_gdf['pedestrian'] == 1)]['length'].sum()/route_geometry.length
major_roads_portion = edges_gdf[(edges_gdf.edgeID.isin(edgeIDs)) &
(edges_gdf['highway'] == 'primary')]['length'].sum()/route_geometry.length
naturale_barriers_portion = edges_gdf[(edges_gdf.edgeID.isin(edgeIDs)) &
(edges_gdf['p_bool'] == 1)]['length'].sum()/route_geometry.length
return pedestrian_portion, major_roads_portion, naturale_barriers_portion, districts

def count_regions(row, nodes_gdf):
    """
    This function counts the number of regions crossed along a route.

    Parameters
    -----
    row: GeoDataFrames Series
        The row of a GeoDataFrames containing routes generated in the ABM
    nodes_gdf: Point GeoDataFrame
        The GeoDataFrame of the nodes of a street network

    Returns
    -----
    Pandas DataFrame
    """
    count = 0
    if not nodes_gdf.loc[row.0].district in row['districts']:
        count += 1
    if not nodes_gdf.loc[row.D].district in row['districts']:
        count +=1
    return len(row['districts']) + count

def generate_ax_hcolorbar(cmap, fig, ax, nrows, ncols, text_color, font_size, norm = None, ticks = 5,
symbol = False):
    """
    This function generates horizontal colorbars for a grid of subplots.

    Parameters
    -----

```

```

cmap: String, Colormap instance
    The used color map
fig: Figure
    The Figure object
ax: AxesSubplot
    A subplot axes of the grid
nrows: Integer
    The number of rows
ncols: Integer
    The number of columns
text_color: String, Array, etc.
    The color of the text
font_size: Integer
    The font size
norm: matplotlib.colors.Normalize instance
    Array of normalised data, if applies
ticks: Integer
    The number of ticks along the bar
symbol: boolean
    If True shows the ">" symbol instead of the max bound (e.g. when normalising)

>Returns
-----
Pandas DataFrame
"""

if font_size is None:
    font_size = 20

sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
sm._A = []
vr_p = 1/30.30
hr_p = 0.5/30.30

width = ax.get_position().width
x = ax.get_position().x0
y = ax.get_position().y0 - 0.070
height = 0.025
pos = [x, y, width, height]
cax = fig.add_axes(pos, frameon = False)
cax.tick_params(size=0)
cb = plt.colorbar(sm, cax=cax, orientation='horizontal')
tick_locator = ticker.MaxNLocator(nbins=ticks)
cb.locator = tick_locator
cb.update_ticks()
cb.outline.set_visible(False)

```

```

if symbol:
    cax.set_xticklabels([round(t,1) if t < norm.vmax else "> "+str(round(t,1)) for t in cax.get_xticks()])

else:
    cax.set_xticklabels([round(t,1) for t in cax.get_xticks()])

plt.setp(plt.getp(cax.axes, "xticklabels"), size = 0, color = text_color,
fontfamily = 'Times New Roman',
        fontsize=(font_size-font_size*0.33))

def set_routes_stats(summary_responses, survey_routes, videos):

    route_variables = ['onlyDistance', 'onlyAngular', 'distanceHeuristic', 'angularHeuristic', 'regions',
    'routeMarks',
           'barriers', 'distantLandmarks', 'usingElements', 'noElements']
    admitted = ['DS', 'DL', 'DG', 'DB', 'DR', 'AC', 'AL', 'AG', 'AB', 'AR']

    routes_stats = pd.DataFrame(index = summary_responses['id'], columns = route_variables )

    def _route_composition(index):

        routes = []
        complementary_routes = []

        total_length = 0.0
        total_minimisation_length = 0.0
        total_elements_length = 0.0

        for video in videos:
            route_composition = dict((rC, 0.0) for rC in route_variables)

            sections = summary_responses.loc[index][video][summary_responses.loc[index][video] != 0].index.values
            traversed_sections = survey_routes[survey_routes.video.isin(sections)]
            route_length = traversed_sections.length.sum()
            total_length += route_length
            minimisation_length = 0.0
            elements_length = 0.0
            overlap_sub_goals = 0.0

            route_models_all = traversed_sections.routeChoice
            route_models_tmp = list(set([item for sublist in route_models_all for item in sublist]))

            if 'not' in route_models_tmp:
                route_models_tmp.remove('not')
            route_models = list(set([item for item in route_models_tmp if item in admitted]))
```

```

for row in traversed_sections.itertuples():

    length_section = traversed_sections.loc[row.Index].length
    rC_section = traversed_sections.loc[row.Index].routeChoice
    minimisation = False

    if any('D' in rc_model for rc_model in rC_section):
        if any('DS' in rc_model for rc_model in rC_section):
            minimisation = True
            route_composition['onlyDistance'] += length_section
        if any('D' in rc_model for rc_model in rC_section if rc_model != 'DS'):
            route_composition['distanceHeuristic'] += length_section

    if any('A' in rc_model for rc_model in rC_section):
        if any('AC' in rc_model for rc_model in rC_section):
            minimisation = True
            route_composition['onlyAngular'] += length_section
        if any('A' in rc_model for rc_model in rC_section if rc_model != 'AC'):
            route_composition['angularHeuristic'] += length_section

    if minimisation:
        minimisation_length += length_section

    elements = False
    if any('R' in rc_model for rc_model in rC_section):
        route_composition['regions'] += length_section
        elements = True
    if any('G' in rc_model for rc_model in rC_section):
        route_composition['distantLandmarks'] += length_section
        elements = True
    if any('B' in rc_model for rc_model in rC_section):
        route_composition['barriers'] += length_section
        elements = True
    if any('L' in rc_model for rc_model in rC_section):
        route_composition['routeMarks'] += length_section
        elements = True
    if (any('B' in rc_model for rc_model in rC_section)) & (any('L' in rc_model for rc_model in rC_section)):
        overlap_sub_goals += length_section

    if elements:
        elements_length += length_section

D = route_composition['onlyDistance']
A = route_composition['onlyAngular']
M = minimisation_length # possible overlaps included as this is only used to derive 'noElements'

```

```

vs 'usingElements'

if M == 0.0:
    route_composition['onlyDistance'] = 0.5
    route_composition['onlyAngular'] = 0.5
else:
    S = D+A # here excluding overlaps
    route_composition['onlyDistance'] = float("{0:.3f}".format(D/S))
    route_composition['onlyAngular'] = float("{0:.3f}".format(A/S))

R = route_composition['regions']
O = route_composition['routeMarks'] - (overlap_sub_goals/2)
B = route_composition['barriers'] - (overlap_sub_goals/2)
DL = route_composition['distantLandmarks']
LR = route_composition['distanceHeuristic']
LA = route_composition['angularHeuristic']
E = elements_length

#route_composition['regions'] = float("{0:.3f}".format(R/E))
route_composition['routeMarks'] = float("{0:.3f}".format(O/E))
route_composition['barriers'] = float("{0:.3f}".format(B/E))
route_composition['distantLandmarks'] = float("{0:.3f}".format(DL/E))
route_composition['distanceHeuristic'] = float("{0:.3f}".format(LR/(LR+LA)))
route_composition['angularHeuristic'] = float("{0:.3f}".format(LA/(LR+LA)))

route_composition['noElements'] = M/(E+M)
route_composition['usingElements'] = E/(E+M)

total_minimisation_length += minimisation_length
total_elements_length += elements_length
routes.append(route_composition)

final_composition = dict((rC, 0.0) for rC in route_variables)
for key,_ in final_composition.items():
    final_composition[key] = float("{0:.3f}".format((routes[0][key]+routes[1][key]+routes[2][key])/3))
subject = summary_responses.loc[index]['id']
routes_stats.at[subject, key] = final_composition[key]

routes_stats.at[subject, "length"] = total_length
routes_stats.at[subject, "combined_length"] = total_elements_length
routes_stats.at[subject, "minimisation_length"] = total_minimisation_length

for row in summary_responses.itertuples():
    _route_composition(row.Index)

return routes_stats

```

```

def compute_duration(startime, endtime):

    start = int(startime[:2])*60 + int(startime[3:5])
    end = int(endtime[:2])*60+ int(endtime[3:5])
    if end < start:
        end += (24*60)

    return end-start

# def with variable object of interest
def standardisation(df):
    df = df.copy()
    for column in df.columns:
        df[column+"_std"] = (df[column]-df[column].mean())/df[column].std()

    for column in df.columns:
        if 'std' in column:
            continue
        df.drop(column, axis = 1, inplace = True)
    return df

def log_transf(df):

    df = df.copy()
    for column in df.columns:
        df[column+"_log"] = np.log(df[column])
        n = 1
        while (df[column+"_log"].nsmallest(n).iloc[-1] == -np.inf):
            n+=1
        df[column+"_log"].replace(-np.inf, df[column+"_log"].nsmallest(n).iloc[-1], inplace = True)

        n = 1
        while (df[column+"_log"].nlargest(n).iloc[-1] == np.inf):
            n+=1
        df[column+"_log"].replace(np.inf, df[column+"_log"].nlargest(n).iloc[-1], inplace = True)

    for column in df.columns:
        if 'log' in column:
            continue
        df.drop(column, axis = 1, inplace = True)
    return df

def standardise_column(df, column):

    series = (df[column]-df[column].mean())/df[column].std()

```

```
    return series

-----gisVis.py-----
import pandas as pd, numpy as np, matplotlib as mpl

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as cols
import matplotlib.patches as mpatches
import matplotlib.ticker as ticker
import colorsys

from mpl_toolkits.axes_grid1 import make_axes_locatable, ImageGrid
from mpl_toolkits.mplot3d.art3d import Line3DCollection
from matplotlib.colors import LinearSegmentedColormap, TwoSlopeNorm
import mapclassify, pylab, colorsys
pd.set_option("display.precision", 3)

class Plot():

    def __init__(self, figsize, black_background, title):

        fig, ax = plt.subplots(1, figsize=figsize)

        # background black or white - basic settings
        rect = fig.patch
        if black_background:
            text_color = "white"
            rect.set_facecolor("black")
        else:
            text_color = "black"
            rect.set_facecolor("white")

        font_size_primary = figsize[0]*1.50
        font_size_secondary = figsize[0]*1.25

        fig.suptitle(title, color = text_color, fontsize=font_size_primary, fontfamily = 'Times New Roman')
        fig.subplots_adjust(top=0.96)

        plt.axis("equal")
        self.fig, self.grid = fig, ax
        self.font_size_primary, self.font_size_secondary = font_size_primary, font_size_secondary
        self.text_color = text_color

class MultiPlot():
```

```

def __init__(self, figsize, nrows, ncols, black_background, title = None):

    fig, grid = plt.subplots(nrows = nrows, ncols = ncols, figsize = figsize)

    rect = fig.patch
    if black_background:
        text_color = "white"
        rect.set_facecolor("black")
    else:
        text_color = "black"
        rect.set_facecolor("white")

    font_size_primary = figsize[0]*1.35
    font_size_secondary = figsize[0]*1.15

    if title is not None:
        fig.suptitle(title, color = text_color, fontsize = font_size_secondary,
                     fontfamily = 'Times New Roman',
                     ha = 'center', va = 'center')
        fig.subplots_adjust(top=0.92)

    self.fig, self.grid = fig, grid
    self.font_size_primary, self.font_size_secondary = font_size_primary, font_size_secondary
    self.text_color = text_color

def _single_plot(ax, gdf, column = None, scheme = None, bins = None, classes = 7, norm = None, cmap = None,
                color = 'red', alpha = 1.0,
                legend = False, geometry_size = 1.0, geometry_size_column = None, geometry_size_factor = None,
                zorder = 0):
    """
    It plots the geometries of a GeoDataFrame, coloring on the bases of the values contained in column,
    using a given scheme, on the provided Axes.
    If only "column" is provided, a categorical map is depicted.
    If no column is provided, a plain map is shown.

    Parameters
    -----
    ax: matplotlib.axes object
        the Axes on which plotting
    gdf: GeoDataFrame
        GeoDataFrame to be plotted
    column: string
        Column on which the plot is based
    scheme: string
        classification method, choose amongst: https://pysal.org/mapclassify/api.html
    bins: list
        bins defined by the user
    """

```

```

classes: int
    classes when scheme is not "None"
norm: array
    a class that specifies a desired data normalisation into a [min, max] interval
cmap: string, matplotlib.colors.LinearSegmentedColormap
    see matplotlib colormaps for a list of possible values or pass a colormap
color: string
    categorical color applied to all geometries when not using a column to color them
alpha: float
    alpha value of the plotted layer
legend: boolean
    if True, it shows the legend
geometry_size: float
    markersize, when plotting a Point GeoDataFrame or linewidth when plotting a LineString GeoDataFrame
geometry_size_column: string
    name of the column, if any, of the GeoDataFrame whose values are to regulate the size of the
    geometries
geometry_size_factor: float
    to control to what extent the values of the geometry_size_column impact the geometry_size
    For a Point GeoDataFrame, it rescales the geometry_size_column provided from 0 to 1 and applies the
    factor (e.g. rescaled variable's value [0-1] * factor).
zorder: int
    zorder of this layer; e.g. if 0, plots first, thus main GeoDataFrame on top; if 1, plots last,
    thus on top.
"""

gdf = gdf.copy()
categorical = True
if (column is not None):
    if (gdf[column].dtype != 'O'):
        gdf = gdf.reindex(gdf[column].abs().sort_values(ascending = True).index)

# categorical map
if (column is not None) & (scheme is None) & (norm is None) & (cmap is None):
    cmap = rand_cmap(len(gdf[column].unique()))

if (norm is not None) | (scheme is not None):
    categorical = False
    color = None
    if cmap is None:
        cmap = kindlmann()

if (column is not None) & (not categorical):
    if(gdf[column].dtype == 'O'):
        gdf[column] = gdf[column].astype(float)

if bins is None:

```

```

c_k = {None}
if classes is not None:
    c_k = {"k" : classes}
else:
    c_k = {'bins':bins, "k" : len(bins)}
scheme = 'User_Defined'

if gdf.iloc[0].geometry.geom_type == 'Point':
    if (geometry_size_factor is not None):
        scaling_columnDF(gdf, geometry_size_column)
        gdf['geometry_size'] = np.where(gdf[geometry_size_column+'_sc'] >= 0.20,
                                         gdf[geometry_size_column+'_sc']*geometry_size_factor, 0.40) # marker size
        geometry_size = gdf['geometry_size']

    gdf.plot(ax = ax, column = column, markersize = geometry_size, categorical = categorical, color = color,
              scheme = scheme, cmap = cmap,
              norm = norm, alpha = alpha, legend = legend, classification_kwds = c_k, zorder = zorder)

elif gdf.iloc[0].geometry.geom_type == 'LineString':
    if geometry_size_factor is not None:
        geometry_size = [(abs(value)*geometry_size_factor) if (abs(value)*geometry_size_factor) > 1.1 else
                         1.1 for value in gdf[geometry_size_column]]

    gdf.plot(ax = ax, column = column, categorical = categorical, color = color, linewidth = geometry_size,
              scheme = scheme, alpha = alpha,
              cmap = cmap, norm = norm, legend = legend, classification_kwds = c_k, capstyle = 'round',
              joinstyle = 'round', zorder = zorder)

else:
    gdf.plot(ax = ax, column = column, categorical = categorical, color = color, scheme = scheme,
              edgecolor = 'none', alpha = alpha, cmap = cmap,
              norm = norm, legend = legend, classification_kwds = c_k, zorder = zorder)

def plot_gdf(gdf, column = None, title = None, black_background = True, figsize = (15,15), scheme = None,
             bins = None, classes = None, norm = None,
             cmap = None, color = None, alpha = None, legend = False, geometry_size = 1.0,
             geometry_size_column = None,
             geometry_size_factor = None, cbar = False, cbar_ticks = 5, cbar_max_symbol = False,
             cbar_min_max = False, cbar_shrinkage = 0.75,
             axes_frame = False, base_map_gdf = pd.DataFrame({"a" : []}), base_map_color = None,
             base_map_alpha = 0.4, base_map_geometry_size = 1.1,
             base_map_zorder = 0):

    """
    It plots the geometries of a GeoDataFrame, coloring on the bases of the values contained in column,
    using a given scheme.
    """

```

```

If only "column" is provided, a categorical map is depicted.
If no column is provided, a plain map is shown.

Parameters
-----
gdf: GeoDataFrame
    GeoDataFrame to be plotted
column: string
    Column on which the plot is based
title: string
    title of the plot
black_background: boolean
    black background or white
fig_size: float
    size of the figure's side extent
scheme: string
    classification method, choose amongst: https://pysal.org/mapclassify/api.html
bins: list
    bins defined by the user
classes: int
    number of classes for categorising the data when scheme is not "None"
norm: array
    a class that specifies a desired data normalisation into a [min, max] interval
cmap: string, matplotlib.colors.LinearSegmentedColormap
    see matplotlib colormaps for a list of possible values or pass a colormap
color: string
    categorical color applied to all geometries when not using a column to color them
alpha: float
    alpha value of the plotted layer
legend: boolean
    if True, show legend, otherwise don't
cbar: boolean
    if True, show colorbar, otherwise don't; when True it doesn't show legend
cbar_ticks: int
    number of ticks along the colorbar
cbar_max_symbol: boolean
    if True, it shows the ">" next to the highest tick's label in the colorbar (useful when normalising)
cbar_min_max: boolean
    if True, it only shows the labels of the lowest and highest ticks of the colorbar
cbar_shrink: float
    fraction by which to multiply the size of the colorbar
axes_frame: boolean
    if True, it shows the axes' frame
geometry_size: float
    markersize, when plotting a Point GeoDataFrame or linewidth when plotting a LineString GeoDataFrame
geometry_size_column: string
    name of the column, if any, of the GeoDataFrame whose values are to regulate the size of the geometries

```

```

geometry_size_factor: float
    to control to what extent the values of the geometry_size_column impact the geometry_size
    For a Point GeoDataFrame, it rescales the geometry_size_column provided from 0 to 1 and
    applies the factor (e.g. rescaled variable's value [0-1] * factor).
base_map_gdf: GeoDataFrame
    a desired additional layer to use as a base map
base_map_color: string
    color applied to all geometries of the base map
base_map_alpha: float
    base map's alpha value
base_map_geometry_size: float
    base map's marker size when the base map is a Point GeoDataFrame

base_map_zorder: int
    zorder of the layer; e.g. if 0, plots first, thus main GeoDataFrame on top; if 1, plots last, thus on top.

>Returns
-----
fig: matplotlib.figure.Figure object
    the resulting figure
"""

# fig,ax set up
plot = Plot(figsize = figsize, black_background = black_background, title = title)
fig, ax = plot.fig, plot.grid

_set_axes_frame(axes_frame, ax, black_background, plot.text_color)
ax.set_aspect("equal")

zorder = 0
if (not base_map_gdf.empty):
    _plot_baseMap(gdf = base_map_gdf, ax = ax, color = base_map_color,
    geometry_size = base_map_geometry_size, alpha = base_map_alpha,
    zorder = base_map_zorder )
if base_map_zorder == 0:
    zorder = 1

if geometry_size_column is None:
    geometry_size_column = column

_single_plot(ax, gdf, column = column, scheme = scheme, bins = bins, classes = classes, norm = norm,
cmap = cmap, color = color, alpha = alpha,
    geometry_size = geometry_size, geometry_size_column = geometry_size_column,
    geometry_size_factor = geometry_size_factor,
    zorder = zorder, legend = legend)

```

```

if legend:
    _generate_legend(plot, ax, black_background)
elif cbar:
    if norm is None:
        min_value = gdf[column].min()
        max_value = gdf[column].max()
        norm = plt.Normalize(vmin = min_value, vmax = max_value)

    _generate_colorbar(plot, cmap, norm = norm, ticks = cbar_ticks, symbol = cbar_max_symbol,
    min_max = cbar_min_max, shrinkage = cbar_shrinkage)

return fig

def plot_gdfs(list_gdfs = [], column = None, ncols = 2, main_title = None, titles = [], black_background = True,
    figsize = (15,30), scheme = None,
    bins = None, classes = None, norm = None, cmap = None, color = None, alpha = None,
    legend = False, cbar = False,
    cbar_ticks = 5, cbar_max_symbol = False, cbar_min_max = False, cbar_shrinkage = 0.75,
    axes_frame = False,
    geometry_size = None, geometry_size_column = None, geometry_size_factor = None):

"""
It plots the geometries of a list of GeoDataFrame, containing the same type of geometry.
Coloring is based on a provided column (that needs to
be a column in each passed GeoDataFrame), using a given scheme.
If only "column" is provided, a categorical map is depicted.
If no column is provided, a plain map is shown.

Parameters
-----
list_gdfs: list of GeoDataFrames
    GeoDataFrames to be plotted
column: string
    Column on which the plot is based
main_title: string
    main title of the plot
titles: list of string
    list of titles to be assigned to each quadrant (axes) of the grid
black_background: boolean
    black background or white
fig_size: float
    size figure extent
scheme: string
    classification method, choose amongst: https://pysal.org/mapclassify/api.html
bins: list
    bins defined by the user
classes: int
"""

It plots the geometries of a list of GeoDataFrame, containing the same type of geometry.
Coloring is based on a provided column (that needs to
be a column in each passed GeoDataFrame), using a given scheme.
If only "column" is provided, a categorical map is depicted.
If no column is provided, a plain map is shown.

Parameters
-----
list_gdfs: list of GeoDataFrames
    GeoDataFrames to be plotted
column: string
    Column on which the plot is based
main_title: string
    main title of the plot
titles: list of string
    list of titles to be assigned to each quadrant (axes) of the grid
black_background: boolean
    black background or white
fig_size: float
    size figure extent
scheme: string
    classification method, choose amongst: https://pysal.org/mapclassify/api.html
bins: list
    bins defined by the user
classes: int

```

```

    number of classes for categorising the data when scheme is not "None"
norm: array
    a class that specifies a desired data normalisation into a [min, max] interval
cmap: string, matplotlib.colors.LinearSegmentedColormap
    see matplotlib colormaps for a list of possible values or pass a colormap
color: string
    categorical color applied to all geometries when not using a column to color them
alpha: float
    alpha value of the plotted layer
legend: boolean
    if True, show legend, otherwise don't
cbar: boolean
    if True, show colorbar, otherwise don't; when True it doesn't show legend
cbar_ticks: int
    number of ticks along the colorbar
cbar_max_symbol: boolean
    if True, it shows the ">" next to the highest tick's label in the colorbar (useful when normalising)
cbar_min_max: boolean
    if True, it only shows the labels of the lowest and highest ticks of the colorbar
cbar_shrink: float
    fraction by which to multiply the size of the colorbar
axes_frame: boolean
    if True, it shows the axes' frame
geometry_size: float
    markersize, when plotting a Point GeoDataFrame or linewidth when plotting a LineString GeoDataFrame
geometry_size_column: string
    name of the columnn, if any, of the GeoDataFrame whose values are to regulate the size of the geometries
geometry_size_factor: float
    to control to what extent the values of the geometry_size_column impact the geometry_size
    For a Point GeoDataFrame, it rescales the geometry_size_column provided from 0 to 1 and applies
    the factor (e.g. rescaled variable's value [0-1] * factor).

```

**Returns****-----**

```

fig: matplotlib.figure.Figure object
    the resulting figure
"""

if ncols == 2:
    nrows, ncols = int(len(list_gdfs)/2), 2
    if (len(list_gdfs)%2 != 0):
        nrows = nrows+1
else:
    nrows, ncols = int(len(list_gdfs)/3), 3
    if (len(list_gdfs)%3 != 0):
        nrows = nrows+1

```

```

multiPlot = MultiPlot(figsize = figsize, nrows = nrows, ncols = ncols, black_background = black_background,
                      title = main_title)

fig, grid = multiPlot.fig, multiPlot.grid

if nrows > 1:
    grid = [item for sublist in grid for item in sublist]
if cbar:
    legend = False
    if (norm is None):
        min_value = min([gdf[column].min() for gdf in list_gdfs])
        max_value = max([gdf[column].max() for gdf in list_gdfs])
        norm = plt.Normalize(vmin = min_value, vmax = max_value)

if nrows > 1:
    axes = [item for sublist in grid for item in sublist]
else:
    axes = grid

legend_ax = False
for n, ax in enumerate(axes):
    _set_axes_frame(axes_frame, ax, black_background, multiPlot.text_color)
    ax.set_aspect("equal")
    if n > len(list_gdfs)-1:
        continue # when odd nr of gdfs

    gdf = list_gdfs[n]
    if len(titles) > 0:
        ax.set_title(titles[n], loc='center', fontfamily = 'Times New Roman',
                     fontsize = multiPlot.font_size_primary, color = multiPlot.text_color, pad = 15)

    geometry_size_column = column
    if geometry_size_columns != []:
        geometry_size_column = geometry_size_columns[n]

    if (legend) & (ax == axes[-1]):
        legend_ax = True

    _single_plot(ax, gdf, column = column, scheme = scheme, bins = bins, classes = classes,
                norm = norm, cmap = cmap, color = color,
                alpha = alpha, legend = legend_ax, geometry_size = geometry_size,
                geometry_size_column = geometry_size_column,
                geometry_size_factor = geometry_size_factor)

if legend:
    _generate_legend(multiPlot, ax, black_background)

```

```

    elif cbar:
        _set_colorbar(multiPlot, cmap, norm = norm, ticks = cbar_ticks, symbol = cbar_max_symbol,
                      min_max = cbar_min_max,
                      shrinkage = cbar_shrinkage)

    return fig

def plot_gdf_grid(gdf = None, columns = [], ncols = 2, titles = [], black_background = True,
                  figsize = (15,15), scheme = None, bins = None,
                  classes = None, norm = None, cmap = None, color = None, alpha = None, legend = False,
                  cbar = False,
                  cbar_ticks = 5, cbar_max_symbol = False, cbar_min_max = False, cbar_shrinkage = 0.75,
                  axes_frame = False, geometry_size = None,
                  geometry_size_columns = [], geometry_size_factor = None):
    """
    It plots the geometries of a GeoDataFrame, coloring on the bases of the values contained in the
    provided columns, using a given scheme.
    If only "column" is provided, a categorical map is depicted.
    If no column is provided, a plain map is shown.

    Parameters
    -----
    gdf: GeoDataFrame
        GeoDataFrame to be plotted
    column: string
        Column on which the plot is based
    title: string
        title of the plot
    black_background: boolean
        black background or white
    fig_size: float
        size figure extent
    scheme: string
        classification method, choose amongst: https://pysal.org/mapclassify/api.html
    bins: list
        bins defined by the user
    classes: int
        number of classes for categorising the data when scheme is not "None"
    norm: array
        a class that specifies a desired data normalisation into a [min, max] interval
    cmap: string, matplotlib.colors.LinearSegmentedColormap
        see matplotlib colormaps for a list of possible values or pass a colormap
    color: string
        categorical color applied to all geometries when not using a column to color them
    alpha: float
        alpha value of the plotted layer
    legend: boolean

```

```

        if True, show legend, otherwise don't
cbar: boolean
        if True, show colorbar, otherwise don't; when True it doesn't show legend
cbar_ticks: int
        number of ticks along the colorbar
cbar_max_symbol: boolean
        if True, it shows the ">" next to the highest tick's label in the colorbar (useful when normalising)
cbar_min_max: boolean
        if True, it only shows the labels of the lowest and highest ticks of the colorbar
cbar_shrink: float
        fraction by which to multiply the size of the colorbar
axes_frame: boolean
        if True, it shows the axes' frame
geometry_size: float
        markersize, when plotting a Point GeoDataFrame or linewidth when plotting a LineString GeoDataFrame
geometry_size_columns: List
        List of the name of the columnn, if any,
        of the passed GeoDataFrames whose values are to regulate the size of the geometries
geometry_size_factor: float
        to control to what extent the values of the geometry_size_column impact the geometry_size
        For a Point GeoDataFrame, it rescales the geometry_size_column provided from 0 to 1
        and applies the factor (e.g. rescaled variable's value [0-1] * factor).
geometry_size_factor: float
        when provided, it rescales the column provided, if any,
        from 0 to 1 and it uses the geometry_size_factor to rescale the line
        width accordingly
        (e.g. rescaled variable's value [0-1] * factor), when plotting a LineString GeoDataFrame
"""

if ncols == 2:
    nrows, ncols = int(len(columns)/2), 2
    if (len(columns)%2 != 0):
        nrows = nrows+1
else:
    nrows, ncols = int(len(columns)/3), 3
    if (len(columns)%3 != 0):
        nrows = nrows+1

multiPlot = MultiPlot(figsize = figsize, nrows = nrows, ncols = ncols, black_background = black_background)
fig, grid = multiPlot.fig, multiPlot.grid

legend_ax = False
if cbar:
    legend = False
    if norm is None:
        min_value = min([gdf[column].min() for column in columns])
        max_value = max([gdf[column].max() for column in columns])

```

```

norm = plt.Normalize(vmin = min_value, vmax = max_value)

if nrows > 1:
    axes = [item for sublist in grid for item in sublist]
else:
    axes = grid

for n, ax in enumerate(axes):
    ax.set_aspect("equal")
    _set_axes_frame(axes_frame, ax, black_background, multiPlot.text_color)

    if n > len(columns)-1:
        continue # when odd nr of columns

    column = columns[n]
    if len(titles) > 0:
        ax.set_title(titles[n], loc='center', fontfamily = 'Times New Roman',
                     fontsize = multiPlot.font_size_primary, color = multiPlot.text_color, pad = 15)

    geometry_size_column = column
    if geometry_size_columns != []:
        geometry_size_column = geometry_size_columns[n]
    if (legend) & (ax == axes[-1]):
        legend_ax = True

    _single_plot(ax, gdf, column = column, scheme = scheme, bins = bins, classes = classes, norm = norm,
                cmap = cmap, color = color,
                alpha = alpha, legend = legend_ax, geometry_size = geometry_size,
                geometry_size_column = geometry_size_column,
                geometry_size_factor = geometry_size_factor)

    if legend:
        _generate_legend(multiPlot, ax, black_background)
    elif cbar:
        _generate_colorbar(plot = multiPlot, cmap = cmap, norm = norm, ticks = cbar_ticks,
                           symbol = cbar_max_symbol, min_max = cbar_min_max,
                           shrinkage = cbar_shrinkage)

return fig

def _plot_baseMap(gdf = None, ax = None, color = None, geometry_size = None, alpha = 0.5, zorder = 0):

    if gdf.iloc[0].geometry.geom_type == 'LineString':
        gdf.plot(ax = ax, color = color, linewidth = geometry_size, alpha = alpha, zorder = zorder)
    if gdf.iloc[0].geometry.geom_type == 'Point':
        gdf.plot(ax = ax, color = color, markersize = geometry_size, alpha = alpha, zorder = zorder)
    if gdf.iloc[0].geometry.geom_type == 'Polygon':

```

```

gdf.plot(ax = ax, color = color, alpha = alpha, zorder = zorder)

def plot_multiplex(M, multiplex_edges):

    node_Xs = [float(node["x"]) for node in M.nodes.values()]
    node_Ys = [float(node["y"]) for node in M.nodes.values()]
    node_Zs = np.array([float(node["z"])*2000 for node in M.nodes.values()])
    node_size = []
    size = 1
    node_color = []

    for i, d in M.nodes(data=True):
        if d["station"]:
            node_size.append(9)
            node_color.append("#ec1a30")
        elif d["z"] == 1:
            node_size.append(0.0)
            node_color.append("#ffffcc")
        elif d["z"] == 0:
            node_size.append(8)
            node_color.append("#ff8566")

    lines = []
    line_width = []
    geometry_sizeidth = 0.4

    # edges
    for u, v, data in M.edges(data=True):
        xs, ys = data["geometry"].xy
        zs = [M.node[u]["z"]*2000 for i in range(len(xs))]
        if data["layer"] == "intra_layer":
            zs = [0, 2000]

        lines.append([list(a) for a in zip(xs, ys, zs)])
        if data["layer"] == "intra_layer":
            line_width.append(0.2)
        elif data["pedestrian"] == 1:
            line_width.append(0.1)
        else:
            line_width.append(geometry_sizeidth)

    fig_height = 40
    lc = Line3DCollection(lines, linewidths=line_width, alpha=1, color="#ffffff", zorder=1)

    west, south, east, north = multiplex_edges.total_bounds
    bbox_aspect_ratio = (north - south) / (east - west)*1.5

```

```

fig_width = fig_height +90 / bbox_aspect_ratio/1.5
fig = plt.figure(figsize=(15, 15))
ax = fig.gca(projection="3d")
ax.add_collection3d(lc)
ax.scatter(node_Xs, node_Ys, node_Zs, s=node_size, c=node_color, zorder=2)
ax.set_ylim(south, north)
ax.set_xlim(west, east)
ax.set_zlim(0, 2500)
ax.axis("off")
ax.margins(0)
ax.tick_params(size=which="both", direction="in")
fig.canvas.draw()
ax.set_facecolor("black")
ax.set_aspect("equal")

return(fig)

def _generate_legend(plot, ax, black_background):
    """
    It generate the legend for a figure.

    Parameters
    -----
    ax: matplotlib.axes object
        the Axes on which plotting
    text_color: string
        the text color
    font_size: int
        the legend's labels text size
    """
    leg = ax.get_legend()
    for handle in leg.legendHandles:
        if not isinstance(handle, mpl.lines.Line2D):
            handle._legmarker.set_markersize(plot.fig.get_size_inches()[0]*0.90)

    final_legend = plot.fig.legend(handles = leg.legendHandles, labels = [t.get_text() for t in leg.texts],
                                   loc = 'center right',
                                   bbox_to_anchor = (1.10, 0.5))

    if black_background:
        text_color = 'black'
    else:
        text_color = 'white'

    plt.setp(final_legend.texts, family= 'Times New Roman', fontsize = plot.font_size_secondary,
            color = text_color, va = 'center')
    final_legend.get_frame().set_linewidth(0.0) # remove legend border

```

```

final_legend.set_zorder(102)

if not black_background:
    final_legend.get_frame().set_facecolor('black')
    final_legend.get_frame().set_alpha(0.90)
else:
    final_legend.get_frame().set_facecolor('white')
    final_legend.get_frame().set_alpha(0.75)

leg.remove()
plot.fig.add_artist(final_legend)

def _generate_colorbar(plot = None, cmap = None, norm = None, ticks = 5, symbol = False,
                      min_max = False, shrinkage = 0.95):
    """
    It plots a colorbar, given some settings.

    Parameters
    -----
    fig: matplotlib.figure.Figure
        The figure container for the current plot
    pos: list of float
        the axes positions
    sm: matplotlib.cm.ScalarMappable
        a mixin class to map scalar data to RGBA
    norm: array
        a class that specifies a desired data normalisation into a [min, max] interval
    text_color: string
        the text color
    font_size: int
        the colorbar's labels text size
    ticks: int
        the number of ticks along the colorbar
    symbol: boolean
        if True, it shows the ">" next to the highest tick's label in the colorbar (useful when normalising)
    cbar_min_max: boolean
        if True, it only shows the ">" and "<" as labels of the lowest and highest ticks' the colorbar
    """

    cb = plot.fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax=plot.grid, shrink = shrinkage)
    tick_locator = ticker.MaxNLocator(nbins=ticks)
    cb.locator = tick_locator
    cb.update_ticks()
    cb.outline.set_visible(False)

    ticks = list(cb.get_ticks())
    for t in ticks:

```

```

        if (t == ticks[-1]) & (t != norm.vmax) :
            ticks[-1] = norm.vmax

    if min_max:
        ticks = [norm.vmin, norm.vmax]

    cb.set_ticks(ticks)
    cb.ax.set_yticklabels([round(t,1) for t in ticks])
    if symbol:
        cb.ax.set_yticklabels([round(t,1) if t < norm.vmax else ">" +str(round(t,1)) for t in cb.ax.get_yticks()])

    plt.setp(plt.getp(cb.ax, "yticklabels"), color = plot.text_color, fontfamily = 'Times New Roman',
             fontsize= plot.font_size_secondary)

def _set_axes_frame(axes_frame = False, ax = None, black_background = False, text_color = 'black'):
    """
    It draws the axis frame.

    Parameters
    -----
    ax: matplotlib.axes
        the Axes on which plotting
    black_background: boolean
        it indicates if the background color is black
    text_color: string
        the text color
    """
    if not axes_frame:
        ax.set_axis_off()
        return

    ax.xaxis.set_ticklabels([])
    ax.yaxis.set_ticklabels([])
    ax.tick_params(axis= 'both', which= 'both', length=0)

    for spine in ax.spines:
        ax.spines[spine].set_color(text_color)
    if black_background:
        ax.set_facecolor('black')

def normalize(n, range1, range2):

    delta1 = range1[1] - range1[0]
    delta2 = range2[1] - range2[0]
    return (delta2 * (n - range1[0]) / delta1) + range2[0]

# Generate random colormap

```

```

def rand_cmap(nlabels, type_color ='soft'):
    """
    It generates a categorical random color map, given the number of classes

    Parameters
    -----
    nlabels: int
        the number of categories to be coloured
    type_color: string {"soft", "bright"}
        it defines whether using bright or soft pastel colors, by limiting the RGB spectrum

    Returns
    -----
    cmap: matplotlib.colors.LinearSegmentedColormap
        the color map
    """
    if type_color not in ('bright', 'soft'):
        type_color = 'bright'

    # Generate color map for bright colors, based on hsv
    if type_color == 'bright':
        randHSVcolors = [(np.random.uniform(low=0.0, high=0.8),
                          np.random.uniform(low=0.2, high=0.8),
                          np.random.uniform(low=0.9, high=1.0)) for i in range(nlabels)]

        # Convert HSV list to RGB
        randRGBcolors = []
        for HSVcolor in randHSVcolors:
            randRGBcolors.append(colorsys.hsv_to_rgb(HSVcolor[0], HSVcolor[1], HSVcolor[2]))

    random_colormap = LinearSegmentedColormap.from_list('new_map', randRGBcolors, N=nlabels)

    # Generate soft pastel colors, by limiting the RGB spectrum
    if type_color == 'soft':
        low = 0.6
        high = 0.95
        randRGBcolors = [(np.random.uniform(low=low, high=high),
                          np.random.uniform(low=low, high=high),
                          np.random.uniform(low=low, high=high)) for i in range(nlabels)]

    random_colormap = LinearSegmentedColormap.from_list('new_map', randRGBcolors, N=nlabels)

    return random_colormap

def kindlmann():
    """

```

```

It returns a Kindlmann color map. See https://ieeexplore.ieee.org/document/1183788

>Returns
-----
cmap: matplotlib.colors.LinearSegmentedColormap
    the color map
"""

kindlmann_list = [(0.00, 0.00, 0.00, 1), (0.248, 0.0271, 0.569, 1), (0.0311, 0.258, 0.646, 1),
                  (0.019, 0.415, 0.415, 1), (0.025, 0.538, 0.269, 1), (0.0315, 0.658, 0.103, 1),
                  (0.331, 0.761, 0.036, 1), (0.768, 0.809, 0.039, 1), (0.989, 0.862, 0.772, 1),
                  (1.0, 1.0, 1.0)]
cmap = LinearSegmentedColormap.from_list('kindlmann', kindlmann_list)
return cmap

def lighten_color(color, amount=0.5):
    """
    This function can be found here https://gist.github.com/ihincks/6a420b599f43fc7dbd79d56798c4e5a,
    author: Ian Hincks.

    Lightens the given color by multiplying (1-luminosity) by the given amount.
    Input can be matplotlib color string, hex string, or RGB tuple.

    Examples:
    >> lighten_color('g', 0.3)
    >> lighten_color('#F034A3', 0.6)
    >> lighten_color((.3,.55,.1), 0.5)
    """

try:
    c = cols.cnames[color]
except:
    c = color

c = colorsys.rgb_to_hls(*cols.to_rgb(c))
return colorsys.hls_to_rgb(c[0], 1 - amount * (1 - c[1]), c[2])

def scaling_columnDF(df, column, inverse = False):
    """
    It rescales the values in a dataframe's columns from 0 to 1

    Parameters
    -----
    df: pandas DataFrame
        a DataFrame
    column: string
        the column name, representing the column to rescale

```

```

inverse: boolean
    if true, rescales from 1 to 0 instead of 0 to 1
-----
"""

df[column+"_sc"] = (df[column]-df[column].min())/(df[column].max()-df[column].min())
if inverse:
    df[column+"_sc"] = 1-(df[column]-df[column].min())/(df[column].max()-df[column].min())

-----Agent.java-----
package pedSim.agents;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

import org.javatuples.Pair;
import org.locationtech.jts.geom.Coordinate;
import org.locationtech.jts.geom.GeometryFactory;
import org.locationtech.jts.geom.LineString;
import org.locationtech.jts.linearref.LengthIndexedLine;
import org.locationtech.jts.planargraph.DirectedEdge;

import pedSim.cognitiveMap.AgentCognitiveMap;
import pedSim.engine.Parameters;
import pedSim.engine.PedSimCity;
import pedSim.routeChoice.Route;
import pedSim.routeChoice.RoutePlanner;
import sim.engine.SimState;
import sim.engine.Steppable;
import sim.engine.Stoppable;
import sim.graph.EdgeGraph;
import sim.graph.NodeGraph;
import sim.util.geo.MasonGeometry;
import sim.util.geo.PointMoveTo;

/**
 * This class represents an agent in the pedestrian simulation. Agents move
 * along paths between origin and destination nodes.
 */
public final class Agent implements Steppable {

    private static final long serialVersionUID = 1L;
    PedSimCity state;
    public Integer agentID;

```

```
// Initial Attributes
public List<Pair<NodeGraph, NodeGraph>> OD = new LinkedList<>();
public NodeGraph originNode = null;
public NodeGraph destinationNode = null;
public List<DirectedEdge> directedEdgesSequence = new ArrayList<>();
private List<Integer> edgeIDsSequence;
boolean reachedDestination = false;
public Integer tripsDone = 0;

// point that denotes agent's position
// private Point location;
private AgentProperties agentProperties;
public AgentCognitiveMap cognitiveMap;

Stoppable killAgent;
protected MasonGeometry agentLocation;
// How much to move the agent by in each step()

double speed = 0.0;

// start, current, end position along current line
EdgeGraph currentEdge = null;
double startIndex = 0.0;
double currentIndex = 0.0;
double endIndex = 0.0;

// used by agent to walk along line segment
int linkDirection = 1;
int indexOnEdgesSequence = 0;
int pathDirection = 1;
protected LengthIndexedLine indexedSegment = null;
public Route route = new Route();

/**
 * Constructor Function. Creates a new agent with the specified agent
 * properties.
 *
 * @param state the PedSimCity simulation state.
 */
public Agent(PedSimCity state) {

    this.state = state;
    final GeometryFactory fact = new GeometryFactory();
    agentLocation = new MasonGeometry(fact.createPoint(new Coordinate(10, 10)));
    getGeometry().isMovable = true;

    if (!OD.isEmpty()) {
```

```
        originNode = (NodeGraph) OD.get(tripsDone).getValue(0);
        Coordinate startCoord = null;
        startCoord = originNode.getCoordinate();
        updateAgentPosition(startCoord);
    }

}

/***
 * Initialises the agent properties.
 */
public void initialiseAgentProperties() {
    cognitiveMap = new AgentCognitiveMap();
    agentProperties = new AgentProperties();
}

/***
 * Initialises the agent properties with an empirical group.
 *
 * @param empiricalGroup the empirical group for agent properties.
 */
public void initialiseAgentProperties(EmpiricalAgentsGroup empiricalGroup) {
    cognitiveMap = new AgentCognitiveMap();
    agentProperties = new EmpiricalAgentProperties(this, empiricalGroup);
}

/***
 * Moves the agent to the given coordinates.
 *
 * @param c the coordinates.
 */
public void updateAgentPosition(Coordinate c) {
    PointMoveTo pointMoveTo = new PointMoveTo();
    pointMoveTo.setCoordinate(c);
    state.agents.setGeometryLocation(agentLocation, pointMoveTo);
}

/***
 * Performs agent's stepping action in the simulation.
 *
 * @param state The simulation state.
 */
@Override
public void step(SimState state) {

    if (reachedDestination || destinationNode == null)
        try {
            handleReachedDestination();
        }
        catch (Exception e) {
            logger.error("Error handling reached destination: " + e.getMessage());
        }
}
```

```

        } catch (Exception e) {
            e.printStackTrace();
        }
    else
        keepWalking();
}



```

```

        * Initialises the directedEdgesSequence (the path) for the agent.
    */
    public void initialisePath() {
        this.directedEdgesSequence = route.directedEdgesSequence;
        // set up how to traverse this first link
        indexOnEdgesSequence = 0;
        EdgeGraph firstEdge = (EdgeGraph) directedEdgesSequence.get(0).getEdge();
        // Sets the Agent up to proceed along an Edge
        setupEdge(firstEdge);
        // update the current position for this link
        updateAgentPosition(indexedSegment.extractPoint(currentIndex));
        updateData();
        tripsDone += 1;
    }

    /**
     * Updates data related to the volumes on the segments traversed.
     */
    public void updateData() {
        getSequenceEdges();
        state.flowHandler.updateEdgeData(this, directedEdgesSequence);
        state.flowHandler.storeRouteData(this, edgeIDsSequence);
    }

    /**
     * Plans the route for the agent.
     *
     * @throws Exception
     */
    protected void planRoute() throws Exception {
        if (Parameters.empirical)
            ((EmpiricalAgentProperties) agentProperties).randomizeRouteChoiceParameters();
        if (Parameters.verboseMode) {
            if (agentProperties.routeChoice != null)
                System.out.println("Agent " + agentProperties.routeChoice);
            else
                System.out.println(((EmpiricalAgentProperties) agentProperties).groupName);
            System.out.println(" - origin " + originNode.getID() +
                " destination " + destinationNode.getID());
        }
        final RoutePlanner planner = new RoutePlanner(originNode, destinationNode, this);
        route = planner.definePath();
    }

    /**
     * Gets the sequence of edge IDs for the agent's path.
     */

```

```

protected void getSequenceEdges() {

    edgeIDsSequence = new ArrayList<>();
    for (DirectedEdge directedEdge : directedEdgesSequence) {

        int edgeID = ((EdgeGraph) directedEdge.getEdge()).getID();
        edgeIDsSequence.add(edgeID);
    }
}

/**
 * Moves the agent along the computed route.
 */
protected void keepWalking() {
    // move along the current segment
    speed = updateSpeed();
    currentIndex += speed;

    // check to see if the progress has taken the current index beyond its goal
    // given the direction of movement. If so, proceed to the next edge
    if (linkDirection == 1 && currentIndex > endIndex) {
        final Coordinate currentPos = indexedSegment.extractPoint(endIndex);
        updateAgentPosition(currentPos);
        transitionToNextEdge(currentIndex - endIndex);
    } else if (linkDirection == -1 && currentIndex < startIndex) {
        final Coordinate currentPos = indexedSegment.extractPoint(startIndex);
        updateAgentPosition(currentPos);
        transitionToNextEdge(startIndex - currentIndex);
    } else {
        // just update the position!
        final Coordinate currentPos = indexedSegment.extractPoint(currentIndex);
        updateAgentPosition(currentPos);
    }
}

/**
 * Updates the agent's speed based on the move rate and link direction.
 *
 * @return The updated speed of the agent.
 */
private double updateSpeed() {
    speed = progress(Parameters.moveRate);
    return speed;
}

/**
 * Transitions to the next edge in the {@code directedEdgesSequence}.

```

```

*
* @param residualMove The amount of distance the agent can still travel this
*                      step.
*/
void transitionToNextEdge(double residualMove) {

    // update the counter for where the index on the directedEdgesSequence is
    indexOnEdgesSequence += pathDirection;

    // check to make sure the Agent has not reached the end of the
    // directedEdgesSequence already
    // depends on where you're going!
    if (pathDirection > 0 && indexOnEdgesSequence >= directedEdgesSequence.size()
        || pathDirection < 0 && indexOnEdgesSequence < 0) {
        reachedDestination = true;
        indexOnEdgesSequence -= pathDirection; // make sure index is correct
        return;
    }

    // move to the next edge in the directedEdgesSequence
    final EdgeGraph edge = (EdgeGraph) directedEdgesSequence.get(indexOnEdgesSequence).getEdge();
    setupEdge(edge);
    speed = updateSpeed();
    currentIndex += speed;

    // check to see if the progress has taken the current index beyond its goal
    // given the direction of movement. If so, proceed to the next edge
    if (linkDirection == 1 && currentIndex > endIndex)
        transitionToNextEdge(currentIndex - endIndex);
    else if (linkDirection == -1 && currentIndex < startIndex)
        transitionToNextEdge(startIndex - currentIndex);
}

/**
 * Sets the agent up to proceed along an edge.
 *
 * @param edge The EdgeGraph to traverse next.
 */
void setupEdge(EdgeGraph edge) {

    currentEdge = edge;
    // transform GeomPlanarGraphEdge in LineString
    final LineString line = edge.getLine();
    // index the LineString
    indexedSegment = new LengthIndexedLine(line);
    startIndex = indexedSegment.getStartIndex();
    endIndex = indexedSegment.getEndIndex();
}

```

```
linkDirection = 1;

// check to ensure that Agent is moving in the right direction (direction)
final double distanceToStart = line.get startPoint().distance(agentLocation.geometry);
final double distanceToEnd = line.get endPoint().distance(agentLocation.geometry);

if (distanceToStart <= distanceToEnd) {
    // closer to start
    currentIndex = startIndex;
    linkDirection = 1;
} else if (distanceToEnd < distanceToStart) {
    // closer to end
    currentIndex = endIndex;
    linkDirection = -1;
}

/**
 * Computes the agent's speed.
 *
 * @param val The value used for computing speed.
 * @return The computed speed.
 */
private double progress(double val) {
    return val * linkDirection;
}

/**
 * Sets the stoppable reference for the agent.
 *
 * @param a The stoppable reference.
 */
public void setStoppable(Stoppable a) {
    this.killAgent = a;
}

/**
 * Gets the geometry representing the agent's location.
 *
 * @return The geometry representing the agent's location.
 */
public MasonGeometry getGeometry() {
    return agentLocation;
}

public AgentProperties getProperties() {
    return agentProperties;
```

```
}

/**
 * Gets the cognitive map.
 *
 * @return The cognitive map.
 */
synchronized public AgentCognitiveMap getCognitiveMap() {
    return cognitiveMap;
}

}

-----AgentProperties.java-----
package pedSim.agents;

import pedSim.utilities.StringEnum;
import pedSim.utilities.StringEnum.BarrierType;
import pedSim.utilities.StringEnum.LandmarkType;
import pedSim.utilities.StringEnum.RouteChoice;

/**
 * The 'AgentProperties' class represents the properties and preferences of an
 * agent in the simulation. These properties influence the agent's navigation
 * behaviour and route choices.
 */
public class AgentProperties {

    // for general routing
    public RouteChoice routeChoice;
    public boolean onlyMinimising = false;
    public boolean minimisingDistance = false;
    public boolean minimisingAngular = false;

    public boolean localHeuristicDistance = false;
    public boolean localHeuristicAngular = false;

    // landmarkNavigation related parameters
    public boolean usingLocalLandmarks = false;
    public boolean usingDistantLandmarks = false;

    // region- and barrier-based parameters
    public boolean regionBasedNavigation = false;
    public boolean barrierBasedNavigation = false;
    public boolean preferenceNaturalBarriers = false;
    public boolean aversionSeveringBarriers = false;
```

```
public double naturalBarriers = 0.0;
public double naturalBarriersSD = 0.10;
public double severingBarriers = 0.0;
public double severingBarriersSD = 0.10;

// the ones possibly used as sub-goals
public BarrierType barrierType;
public LandmarkType landmarkType;

/**
 * Constructs an AgentProperties instance associated with the given agent.
 *
 */
public AgentProperties() {

}

/**
 * Sets the route choice for the agent and updates related properties
 * accordingly.
 *
 * @param routeChoice The selected route choice for the agent.
 */
public void setRouteChoice(RouteChoice routeChoice) {
    this.routeChoice = routeChoice;
    onlyMinimising = false;
    minimisingDistance = routeChoice == RouteChoice.ROAD_DISTANCE;
    minimisingAngular = routeChoice == RouteChoice.ANGULAR_CHANGE;

    onlyMinimising = minimisingDistance || minimisingAngular;
    if (onlyMinimising)
        return;

    // localHeuristics
    localHeuristicDistance = containsDistance();
    localHeuristicAngular = containsAngular();

    // landmarks
    activateLandmarks();
    regionBasedNavigation = containsRegion();
    barrierBasedNavigation = containsBarrier();

    if (barrierBasedNavigation) {
        preferenceNaturalBarriers = true;
        aversionSeveringBarriers = true;
        naturalBarriers = 0.70;
    }
}
```

```
        severingBarriers = 1.30;
        barrierType = BarrierType.ALL;
    }
}

/***
 * Checks if the given route choice contains a "DISTANCE" component.
 *
 * @return True if "DISTANCE" is found in the route choice descriptor, false
 *         otherwise.
 */
private boolean containsDistance() {
    return routeChoice.toString().contains("DISTANCE");
}

/***
 * Checks if the given route choice contains an "ANGLE" component.
 *
 * @return True if "ANGLE" is found in the route choice descriptor, false
 *         otherwise.
 */
private boolean containsAngular() {
    return routeChoice.toString().contains("ANGLE");
}

/***
 * Checks if the given route choice contains a "REGION" component.
 *
 * @return True if "REGION" is found in the route choice descriptor, false
 *         otherwise.
 */
private boolean containsRegion() {
    return routeChoice.toString().contains("REGION");
}

/***
 * Checks if the given route choice contains a "BARRIER" component.
 *
 * @return True if "BARRIER" is found in the route choice descriptor, false
 *         otherwise.
 */
public boolean containsBarrier() {
    return routeChoice.toString().contains("BARRIER");
}

/***
 * Activates the use of local and/or global landmarks based on the specified
 *
```

```
* conditions. Sets the values for usingLocalLandmarks and usingDistantLandmarks
* based on the presence of local and global landmarks.
*/
private void activateLandmarks() {

    usingLocalLandmarks = containsLocal();
    usingDistantLandmarks = containsGlobal();

    if (containsAllLandmarks()) {
        usingDistantLandmarks = true;
        if (!routeChoice.toString().contains("DISTANT"))
            usingLocalLandmarks = true;
    }

    if (usingLocalLandmarks)
        landmarkType = StringEnum.LandmarkType.LOCAL;
}

/**
 * Checks if the given route choice contains a "LOCAL" component.
 *
 * @return True if "LOCAL" is found in the route choice descriptor, false
 *         otherwise.
 */
private boolean containsLocal() {
    return routeChoice.toString().contains("LOCAL");
}

/**
 * Checks if the given route choice contains a "DISTANT" component.
 *
 * @return True if "DISTANT" is found in the route choice descriptor, false
 *         otherwise.
 */
private boolean containsGlobal() {
    return routeChoice.toString().contains("DISTANT");
}

/**
 * Checks if the given route choice contains a "DISTANT" component.
 *
 * @return True if "DISTANT" is found in the route choice descriptor, false
 *         otherwise.
 */
private boolean containsAllLandmarks() {
    return routeChoice.toString().contains("LANDMARKS");
}
```

```
}

-----EmpiricalAgentProperties.java-----
package pedSim.agents;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;

import org.javatuples.Pair;

import pedSim.utilities.StringEnum.BarrierType;
import pedSim.utilities.StringEnum.Groups;
import pedSim.utilities.StringEnum.LandmarkType;
import pedSim.utilities.StringEnum.RouteChoiceProperty;
import sim.util.geo.Utilities;

/**
 * ‘EmpiricalAgentProperties‘ is a subclass of ‘AgentProperties‘ that represents
 * the properties of an agent in a pedestrian simulation with empirical-based
 * parameters. It extends the base ‘AgentProperties‘ class to incorporate
 * additional parameters.
 */
public class EmpiricalAgentProperties extends AgentProperties {

    public Groups groupName;
    EmpiricalAgentsGroup group;
    boolean usingElements = false;
    boolean elementsActivated = false;

    List<Double> elementsProbability = new ArrayList<>(Arrays.asList(0.0, 0.0));
    List<Double> minimisationProbability = new ArrayList<>(Arrays.asList(0.0, 0.0));
    List<Double> localHeuristicsProbability = new ArrayList<>(Arrays.asList(0.0, 0.0));
    List<Double> regionBasedProbability = new ArrayList<>(Arrays.asList(0.0));
    List<Double> subGoalsProbability = new ArrayList<>(Arrays.asList(0.0, 0.0));
    List<Double> distantLandmarksProbability = new ArrayList<>(Arrays.asList(0.0));

    Map<RouteChoiceProperty, Double> elementsMap = new HashMap<>();
    Map<RouteChoiceProperty, Double> minimisationMap = new HashMap<>();
    Map<RouteChoiceProperty, Double> localHeuristicsMap = new HashMap<>();
    Map<RouteChoiceProperty, Double> regionBasedMap = new HashMap<>();
    Map<RouteChoiceProperty, Double> subGoalsMap = new HashMap<>();
    Map<RouteChoiceProperty, Double> distantLandmarksMap = new HashMap<>();
```

```

Map<RouteChoiceProperty, Double> randomElementsMap = new HashMap<>();

List<RouteChoiceProperty> elements = new ArrayList<>(
    Arrays.asList(RouteChoiceProperty.USING_ELEMENTS, RouteChoiceProperty.NOT_USING_ELEMENTS));
List<RouteChoiceProperty> minimisation = new ArrayList<>(
    Arrays.asList(RouteChoiceProperty.ROAD_DISTANCE, RouteChoiceProperty.ANGULAR_CHANGE));
List<RouteChoiceProperty> localHeuristics = new ArrayList<>(
    Arrays.asList(RouteChoiceProperty.ROAD_DISTANCE_LOCAL, RouteChoiceProperty.ANGULAR_CHANGE_LOCAL));
List<RouteChoiceProperty> subGoals = new ArrayList<>(Arrays.asList(RouteChoiceProperty.LOCAL_LANDMARKS,
    RouteChoiceProperty.BARRIER_SUBGOALS, RouteChoiceProperty.NO_SUBGOALS));
List<RouteChoiceProperty> regionBased = new ArrayList<>(
    Arrays.asList(RouteChoiceProperty.REGION_BASED, RouteChoiceProperty.NOT_REGION_BASED));
List<RouteChoiceProperty> distantLandmarks = new ArrayList<>(
    Arrays.asList(RouteChoiceProperty.USING_DISTANT, RouteChoiceProperty.NOT_USING_DISTANT));
List<RouteChoiceProperty> randomElements = new ArrayList<>(
    Arrays.asList(RouteChoiceProperty.REGION_BASED, RouteChoiceProperty.LOCAL_LANDMARKS,
        RouteChoiceProperty.BARRIER_SUBGOALS, RouteChoiceProperty.USING_DISTANT));

// Constants for magic numbers and strings
private static final double MIN_NATURAL_BARRIERS = 1.00;
private static final double MAX_NATURAL_BARRIERS = 0.00;
private static final double MIN_SEVERING_BARRIERS = 1.00;
private static final double MAX_SEVERING_BARRIERS = 2.00;

/**
 * Constructs an instance of EmpiricalAgentProperties for an agent, initializing
 * it with properties from the specified EmpiricalAgentsGroup.
 *
 * @param agent The agent for which properties are being set.
 * @param group The EmpiricalAgentsGroup containing the properties to initialize
 *             this agent's properties.
 */
public EmpiricalAgentProperties(Agent agent, EmpiricalAgentsGroup group) {
    super();
    this.group = group;
    this.groupName = this.group.groupName;
}

/**
 * Sets route choice parameters for the agent based on a group's properties.
 * This method updates the agent's route choice probabilities and properties
 * according to the properties defined for the agent's group.
 */
public void setParametersFromGroup() {
    if (groupName.equals(Groups.NULLGROUP))
        return;
}

```

```

// minimisation
Pair<Double, Double> probabilityUsingElements = new Pair<>(group.probabilityUsingElements,
    group.probabilityUsingElementsSD);
Pair<Double, Double> probabilityNotUsingElements = new Pair<>(group.probabilityNotUsingElements,
    group.probabilityNotUsingElementsSD);
ArrayList<Pair<Double, Double>> elementsDis = new ArrayList<>(
    Arrays.asList(probabilityUsingElements, probabilityNotUsingElements));
updateProbabilities(elementsProbability, elementsDis);
mapProbabilities(elements, elementsProbability, elementsMap);

// only minimisation
Pair<Double, Double> probabilityOnlyRoadDistance = new Pair<>(group.probabilityRoadDistance,
    group.probabilityRoadDistanceSD);
Pair<Double, Double> probabilityOnlyAngularChange = new Pair<>(group.probabilityAngularChange,
    group.probabilityAngularChangeSD);
ArrayList<Pair<Double, Double>> onlyMinimisationDis = new ArrayList<>(
    Arrays.asList(probabilityOnlyRoadDistance, probabilityOnlyAngularChange));
updateProbabilities(minimisationProbability, onlyMinimisationDis);
mapProbabilities(minimisation, minimisationProbability, minimisationMap);

// heuristics
Pair<Double, Double> probabilityRoadDistance = new Pair<>(group.probabilityLocalRoadDistance,
    group.probabilityLocalRoadDistanceSD);
Pair<Double, Double> probabilityAngularChange = new Pair<>(group.probabilityLocalAngularChange,
    group.probabilityLocalAngularChangeSD);
ArrayList<Pair<Double, Double>> localHeuristicsDis = new ArrayList<>(
    Arrays.asList(probabilityRoadDistance, probabilityAngularChange));
updateProbabilities(localHeuristicsProbability, localHeuristicsDis);
mapProbabilities(localHeuristics, localHeuristicsProbability, localHeuristicsMap);

// Coarse plan
Pair<Double, Double> probabilityRegionBased = new Pair<>(group.probabilityRegionBasedNavigation,
    group.probabilityRegionBasedNavigationSD);
final ArrayList<Pair<Double, Double>> pRegionsDis =
    new ArrayList<>(Arrays.asList(probabilityRegionBased));
updateProbabilities(regionBasedProbability, pRegionsDis);
for (RouteChoiceProperty region : regionBased) {
    double p = region.equals(RouteChoiceProperty.NOT_REGION_BASED) ?
        1.0 - regionBasedProbability.get(0)
        : regionBasedProbability.get(0);
    regionBasedMap.put(region, p);
}

// subgoals
Pair<Double, Double> probabilityLocalLandmarks = new Pair<>(group.probabilityLocalLandmarks,
    group.probabilityLocalLandmarksSD);
Pair<Double, Double> probabilityBarrierSubGoals = new Pair<>(group.probabilityBarrierSubGoals,

```

```

        group.probabilityBarrierSubGoalsSD);

ArrayList<Pair<Double, Double>> pSubGoalsDis = new ArrayList<>(
    Arrays.asList(probabilityLocalLandmarks, probabilityBarrierSubGoals));

updateProbabilities(subGoalsProbability, pSubGoalsDis);
for (RouteChoiceProperty subGoal : subGoals) {
    double p = subGoal.equals(RouteChoiceProperty.LOCAL_LANDMARKS)
        ? subGoalsProbability.get(subGoals.indexOf(subGoal))
        : 1.00 - subGoalsProbability.stream().mapToDouble(d -> d).sum();
    subGoalsMap.put(subGoal, p);
}

// distant landmarks
Pair<Double, Double> pDistantLandmarksTmp = new Pair<>(group.probabilityDistantLandmarks,
    group.probabilityDistantLandmarksSD);
ArrayList<Pair<Double, Double>> distantLandmarksDis =
    new ArrayList<>(Arrays.asList(pDistantLandmarksTmp));
updateProbabilities(distantLandmarksProbability, distantLandmarksDis);

for (RouteChoiceProperty landmark : distantLandmarks) {
    double p = landmark.equals(RouteChoiceProperty.NOT_USING_DISTANT)
        ? 1.00 - distantLandmarksProbability.get(0)
        : distantLandmarksProbability.get(0);
    distantLandmarksMap.put(landmark, p);
}

// other elements
randomElementsMap.put(randomElements.get(0), regionBasedProbability.get(0));
randomElementsMap.put(randomElements.get(1), subGoalsProbability.get(0));
randomElementsMap.put(randomElements.get(2), subGoalsProbability.get(1));
randomElementsMap.put(randomElements.get(3), distantLandmarksProbability.get(0));

// other route properties
naturalBarriers = rescale(0.0, 1.0, MAX_NATURAL_BARRIERS, MIN_NATURAL_BARRIERS,
    group.naturalBarriers);
this.naturalBarriersSD = group.naturalBarriersSD;
severingBarriers = rescale(0.0, 1.0, MIN_SEVERING_BARRIERS, MAX_SEVERING_BARRIERS,
    group.severingBarriers);
this.severingBarriersSD = group.severingBarriersSD;
}

/**
 * Updates the probabilities of a list of values based on a probability
 * distribution.
 *
 * @param probabilities The list of probabilities to update.
 * @param pDistribution The probability distribution as a list of pairs

```

```

*
*           representing mean and standard deviation.
*/
public void updateProbabilities(List<Double> probabilities, List<Pair<Double, Double>> pDistribution) {
    for (final Double d : probabilities) {
        final int index = probabilities.indexOf(d);
        final double p = Utilities.fromDistribution(pDistribution.get(index).getValue0(),
            pDistribution.get(index).getValue1(), null);
        probabilities.set(index, p);
    }
}

/**
 * Maps a list of properties to their corresponding probabilities and stores
 * them in a map.
 *
 * @param properties      The list of properties to map.
 * @param propertiesProbability The list of probabilities associated with each
 *                             property.
 * @param propertiesMap      The map to store the properties and their
 *                           probabilities.
 */
private void mapProbabilities(List<RouteChoiceProperty> properties, List<Double> propertiesProbability,
    Map<RouteChoiceProperty, Double> propertiesMap) {

    for (final RouteChoiceProperty property : properties) {
        final int index = properties.indexOf(property);
        final double probability = propertiesProbability.get(index);
        propertiesMap.put(property, probability);
    }
}

/**
 * Randomly assigns route choice parameters to the agent based on its route
 * choice group and other settings. This method initialises various route choice
 * properties such as minimisation approaches, use of elements, and local
 * minimisation heuristics based on probability distributions specified in the
 * group and settings.
 */
public void randomizeRouteChoiceParameters() {
    reset();

    if (this.groupName.equals(Groups.NULLGROUP))
        fromUniform();
    else
        setParametersFromGroup();

    final Random random = new Random();
}

```

```

// using elements or not
List<RouteChoiceProperty> keys = new ArrayList<>(elementsMap.keySet());
double pRandom = random.nextDouble() * elementsMap.values().stream().mapToDouble(d -> d).sum();
double limit = 0.0;
for (final RouteChoiceProperty key : keys) {
    double p = elementsMap.get(key);
    if (pRandom <= p + limit) {
        usingElements = key.equals(RouteChoiceProperty.USING_ELEMENTS);
        break;
    }
    limit += p;
}

// minimisation approaches
if (!usingElements) {
    keys.clear();
    keys = new ArrayList<>(minimisationMap.keySet());
    pRandom = random.nextDouble() * minimisationMap.values().stream().mapToDouble(d -> d).sum();
    limit = 0.0;
    for (final RouteChoiceProperty key : keys) {
        final double p = minimisationMap.get(key);
        if (pRandom <= p + limit) {
            minimisingDistance = key.equals(RouteChoiceProperty.ROAD_DISTANCE);
            minimisingAngular = !minimisingDistance;
            break;
        }
        limit += p;
    }
    return;
}

if (naturalBarriers < 0.95)
    preferenceNaturalBarriers = true;
if (severingBarriers > 1.05)
    aversionSeveringBarriers = true;

// local minimisation heuristic
keys.clear();
keys = new ArrayList<>(localHeuristicsMap.keySet());
pRandom = random.nextDouble() * localHeuristicsMap.values().stream().mapToDouble(d -> d).sum();
limit = 0.0;
for (final RouteChoiceProperty key : keys) {
    final double p = localHeuristicsMap.get(key);
    if (pRandom <= p + limit) {
        localHeuristicDistance = key.equals(RouteChoiceProperty.ROAD_DISTANCE_LOCAL);
        localHeuristicAngular = !localHeuristicDistance;
    }
}

```

```

        break;
    }
    limit += p;
}

while (usingElements && !elementsActivated)
    activateElements();
}

/***
 * Resets all route choice properties and related flags to their default states.
 * This method is used to clear any previously assigned route choice parameters.
 */
public void reset() {
    usingElements = false;
    elementsActivated = false;
    onlyMinimising = false;
    minimisingDistance = false;
    minimisingAngular = false;
    localHeuristicDistance = false;
    localHeuristicAngular = false;
    barrierType = null;
    usingLocalLandmarks = false;
    barrierBasedNavigation = false;
    regionBasedNavigation = false;
    usingDistantLandmarks = false;
    preferenceNaturalBarriers = false;
    aversionSeveringBarriers = false;
}

/***
 * Initialises route choice probabilities uniformly for various route choice
 * properties. This method assigns equal probabilities to all available choices
 * for each route choice property. It is used when the agent belongs to the
 * "nullGroup" or when route choice settings are uniform.
 */
private void fromUniform() {
    initializeUniformProbabilities(elements, elementsMap);
    initializeUniformProbabilities(minimisation, minimisationMap);
    initializeUniformProbabilities(localHeuristics, localHeuristicsMap);
    initializeUniformProbabilities(regionBased, regionBasedMap);
    initializeUniformProbabilities(subGoals, subGoalsMap);
    initializeUniformProbabilities(distantLandmarks, distantLandmarksMap);

    naturalBarriers = 0.00 + Math.random() * (1.00 - 0.00);
    severingBarriers = 1.00 + Math.random() * (2.00 - 1.00);
}

```

```

/**
 * Rescales a given value from an old range to a new range.
 *
 * @param oldMin The minimum value of the old range.
 * @param oldMax The maximum value of the old range.
 * @param newMin The minimum value of the new range.
 * @param newMax The maximum value of the new range.
 * @param value The value to be rescaled.
 * @return The rescaled value within the new range.
 */
private double rescale(double oldMin, double oldMax, double newMin, double newMax, double value) {
    double oldRange = oldMax - oldMin;
    double newRange = newMax - newMin;
    return (value - oldMin) * newRange / oldRange + newMin;
}

/**
 * Initialises route choice probabilities uniformly for a list of route choice
 * properties. This method assigns equal probabilities to all available choices
 * for each route choice property.
 *
 * @param properties The list of route choice properties to initialise
 *                   probabilities for.
 * @param propertyMap The map to store the initialised probabilities for each
 *                   property.
 */
private void initializeUniformProbabilities(List<RouteChoiceProperty> properties,
    Map<RouteChoiceProperty, Double> propertyMap) {
    final double probability = 1.0 / Double.valueOf(properties.size());
    for (final RouteChoiceProperty property : properties)
        propertyMap.put(property, probability);
}

/**
 * Activates route choice elements based on randomised probabilities. This
 * method randomly activates route choice elements (e.g., region-based,
 * subgoals, global landmarks) based on the specified probabilities, thereby
 * affecting the agent's route choice behaviour.
 */
private void activateElements() {
    final Random random = new Random();
    List<RouteChoiceProperty> keys = new ArrayList<>(regionBasedMap.keySet());

    double pRandom = random.nextDouble() * regionBasedMap.values().stream().mapToDouble(d -> d).sum();
    double limit = 0.0;
    for (final RouteChoiceProperty key : keys) {

```

```

        final double p = regionBasedMap.get(key);
        if (pRandom <= p + limit) {
            regionBasedNavigation = key.equals(RouteChoiceProperty.REGION_BASED);
            elementsActivated = true;
            break;
        }
        limit += p;
    }

    // subgoals
    keys.clear();
    keys = new ArrayList<>(subGoalsMap.keySet());
    pRandom = random.nextDouble() * subGoalsMap.values().stream().mapToDouble(d -> d).sum();
    limit = 0.0;
    for (final RouteChoiceProperty key : keys) {
        final double p = subGoalsMap.get(key);
        if (pRandom <= p + limit) {
            usingLocalLandmarks = key.equals(RouteChoiceProperty.LOCAL_LANDMARKS);
            barrierBasedNavigation = key.equals(RouteChoiceProperty.BARRIER_SUBGOALS);
            elementsActivated = true;
            barrierType = BarrierType.SEPARATING;
            landmarkType = LandmarkType.LOCAL;
            break;
        }
        limit += p;
    }

    // global landmarks
    keys.clear();
    keys = new ArrayList<>(distantLandmarksMap.keySet());
    pRandom = random.nextDouble() * distantLandmarksMap.values().stream().mapToDouble(d -> d).sum();
    limit = 0.0;
    for (final RouteChoiceProperty key : keys) {
        final double p = distantLandmarksMap.get(key);
        if (pRandom <= p + limit) {
            elementsActivated = true;
            usingDistantLandmarks = key.equals(RouteChoiceProperty.USING_DISTANT);
            break;
        }
        limit += p;
    }
}

-----EmpiricalAgentGroup.java-----
package pedSim.agents;

```

```
import pedSim.utilities.StringEnum.Groups;

/**
 * The 'EmpiricalAgentsGroup' class represents a group of empirical agents in
 * the simulation. It encapsulates group-specific parameters and attributes that
 * influence the behaviour of agents within the group. These parameters include
 * probabilities for various route choice properties, such as using elements,
 * minimisation approaches, heuristics, region-based navigation, subgoals,
 * distant landmarks, and other route properties. Additionally, it stores
 * information about natural and severing barriers and the group's share in the
 * population.
 */
public class EmpiricalAgentsGroup {

    public Groups groupName;
    public double share = 0.0;

    // only minimisation
    double probabilityRoadDistance = 0.0;
    double probabilityRoadDistanceSD = 0.0;
    double probabilityAngularChange = 0.0;
    double probabilityAngularChangeSD = 0.0;

    double probabilityNotUsingElements = 0.0;
    double probabilityNotUsingElementsSD = 0.0;
    double probabilityUsingElements = 0.0;
    double probabilityUsingElementsSD = 0.0;

    // local heuristics
    double probabilityLocalRoadDistance = 0.0;
    double probabilityLocalRoadDistanceSD = 0.0;
    double probabilityLocalAngularChange = 0.0;
    double probabilityLocalAngularChangeSD = 0.0;

    // segmentation
    double probabilityRegionBasedNavigation = 0.0;
    double probabilityRegionBasedNavigationSD = 0.0;
    double probabilityLocalLandmarks = 0.0;
    double probabilityLocalLandmarksSD = 0.0;
    double probabilityBarrierSubGoals = 0.0;
    double probabilityBarrierSubGoalsSD = 0.0;

    // other route properties
    double probabilityDistantLandmarks = 0.0;
    double probabilityDistantLandmarksSD = 0.0;
    double naturalBarriers = 0.0;
    double naturalBarriersSD = 0.0;
```

```
double severingBarriers = 0.0;
double severingBarriersSD = 0.0;

/**
 * Sets the properties of the agent's group based on the provided group name and
 * attributes. This method is responsible for configuring various route choice
 * properties and other parameters for agents belonging to a specific group.
 *
 * @param groupName The name of the agent's group.
 * @param attributes An array of attributes containing group-specific
 *                   parameters.
 */
public void setGroup(Groups groupName, String[] attributes) {

    this.groupName = groupName;
    if (groupName.equals(Groups.NULLGROUP))
        return;

    probabilityUsingElements = Float.parseFloat(attributes[1]);
    probabilityUsingElementsSD = Float.parseFloat(attributes[2]);
    probabilityNotUsingElements = Float.parseFloat(attributes[3]);
    probabilityNotUsingElementsSD = Float.parseFloat(attributes[4]);

    // onlyMinimisation
    probabilityRoadDistance = Float.parseFloat(attributes[5]);
    probabilityRoadDistanceSD = Float.parseFloat(attributes[6]);
    probabilityAngularChange = Float.parseFloat(attributes[7]);
    probabilityAngularChangeSD = Float.parseFloat(attributes[8]);

    // heuristics
    probabilityLocalRoadDistance = Float.parseFloat(attributes[9]);
    probabilityLocalRoadDistanceSD = Float.parseFloat(attributes[10]);
    probabilityLocalAngularChange = Float.parseFloat(attributes[11]);
    probabilityLocalAngularChangeSD = Float.parseFloat(attributes[12]);

    // Regions
    probabilityRegionBasedNavigation = Float.parseFloat(attributes[13]);
    probabilityRegionBasedNavigationSD = Float.parseFloat(attributes[14]);

    // subGoals
    probabilityLocalLandmarks = Float.parseFloat(attributes[15]);
    probabilityLocalLandmarksSD = Float.parseFloat(attributes[16]);
    probabilityBarrierSubGoals = Float.parseFloat(attributes[17]);
    probabilityBarrierSubGoalsSD = Float.parseFloat(attributes[18]);

    // other route properties
    probabilityDistantLandmarks = Float.parseFloat(attributes[19]);
```

```

probabilityDistantLandmarksSD = Float.parseFloat(attributes[20]);

naturalBarriers = Float.parseFloat(attributes[21]);
naturalBarriersSD = Float.parseFloat(attributes[22]);
severingBarriers = Float.parseFloat(attributes[23]);
severingBarriersSD = Float.parseFloat(attributes[24]);

if (this.groupName.equals(Groups.POPULATION))
    return;
share = Float.parseFloat(attributes[25]);
}

}

-----ParametersPanel.java-----
package pedSim.applet;

import java.awt.Button;
import java.awt.Font;
import java.awt.Frame;
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.util.ArrayList;
import java.util.Arrays;

import pedSim.engine.Parameters;

/**
 * A graphical user interface panel for configuring various simulation
 * parameters.
 */
public class ParametersPanel extends Frame {
    private static final long serialVersionUID = 1L;
    private static final int X = 10;
    private int y = 50;
    private static final int Y_SPACE_BETWEEN = 30;
    TextField localPathField = new TextField(null);
    ArrayList<TextField> doubleTextFields = new ArrayList<>();
    ArrayList<TextField> booleanTextFields = new ArrayList<>();

    String[] doubleStrings = { "Max distance of a candidate Local Landmark from a Node (m)",
        "Max distance of a Global Landmark (as anchor) from a Node",

```

```

    "Max Number of Anchors to be identified for each Destination",
    "Min distance Between a node and the agent destination for considering 3D Visibility",
    "Min score for a building to be considered a Global Landmark (0.0 - 1.0)",
    "Min score for a building to be considered a Local Landmark (0.0 - 1.0)",
    "Percentile for choosing Salient Nodes (on centrality values) (0.0 - 1.0)",
    "Wayfinding Easiness Threshold above which local landmarks are not identified",
    "Weight Global Landmarkness in combination with Distance Edge Cost (0.0 - 1.0)",
    "Weight Global Landmarkness in combination with Angular Edge Cost (0.0 - 1.0)",
    "Minimum distance necessary to activate Region-based navigation (m)",
    "Wayfinding Easiness Threshold above which local landmarks are not identified within regions" };

Double defaultValues[] = { Parameters.distanceNodeLandmark, Parameters.distanceAnchors,
    Double.valueOf(Parameters.nrAnchors), Parameters.threshold3dVisibility,
    Parameters.globalLandmarkThreshold,
    Parameters.localLandmarkThreshold, Parameters.salientNodesPercentile,
    Parameters.wayfindingEasinessThreshold, Parameters.globalLandmarknessWeightDistance,
    Parameters.globalLandmarknessWeightAngular, Parameters.regionBasedNavigationThreshold,
    Parameters.wayfindingEasinessThresholdRegions };

/**
 * Constructs the Parameters Panel.
 */
public ParametersPanel() {
    super("Parameters Panel");
    setLayout(null);

    for (String string : doubleStrings) {
        Double defaultValue = defaultValues[Arrays.asList(doubleStrings).indexOf(string)];
        addDoubleField(string, defaultValue, X, y);
        y += Y_SPACE_BETWEEN;
    }

    addBooleanField("Identify Destinations based on DMA Approach", Parameters.usingDMA, X, y);

    Label localPathLabel = new Label(
        "Fill this field with the local path containing the data, only if running as Java Project,
        not from JAR");
    localPathLabel.setBounds(10, 480, 450, 20);
    localPathLabel.setFont(new Font("Arial", Font.ITALIC, 12));
    add(localPathLabel);
    localPathLabel = new Label("e.g.: C:/Users/YourUser/Scripts/pedsimcity/src/main/resources/");
    localPathLabel.setBounds(10, 500, 350, 20);
    localPathLabel.setFont(new Font("Arial", Font.ITALIC, 12));
    add(localPathLabel);

    localPathField.setBounds(360, 500, 350, 20);
    add(localPathField);
}

```

```
Button applyButton = new Button("Apply");
applyButton.setBounds(10, 550, 80, 30);
applyButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        adjustParameters();
        closePanel();
    }
});
add(applyButton);

setSize(800, 600);
setVisible(true);

addWindowListener((WindowListener) new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        closePanel();
    }
});
}

/**
 * Adds double-interpreter field to the panel for adjusting simulation
 * parameters.
 *
 * @param fieldName      The name of the parameter.
 * @param defaultValue  The default value for the parameter.
 * @param x              The x-coordinate for the field.
 * @param y              The y-coordinate for the field.
 */
private void addDoubleField(String fieldName, double defaultValue, int x, int y) {
    Label label = new Label(fieldName + ":");
    TextField textField = new TextField(Double.toString(defaultValue));
    label.setBounds(x, y, 600, 20);
    textField.setBounds(x + 600, y, 100, 20);
    add(label);
    add(textField);
    doubleTextFields.add(textField);
}

/**
 * Adds a boolean-interpreter field to the panel for adjusting simulation
 * parameters.
 *
 * @param fieldName      The name of the parameter.

```

```
* @param defaultValue The default value for the parameter.
* @param x           The x-coordinate for the field.
* @param y           The y-coordinate for the field.
*/
/***
 * Adds a boolean-interpreter field to the panel for adjusting simulation
 * parameters.
 *
 * @param fieldName   The name of the parameter.
 * @param defaultValue The default value for the parameter.
 * @param x           The x-coordinate for the field.
 * @param y           The y-coordinate for the field.
 */
private void addBooleanField(String fieldName, boolean defaultValue, int x, int y) {
    Label label = new Label(fieldName + ":");
    TextField textField = new TextField(Boolean.toString(defaultValue));
    label.setBounds(x, y, 600, 20);
    textField.setBounds(x + 600, y, 100, 20);
    add(label);
    add(textField);
    booleanTextFields.add(textField);
}

public static void main(String[] args) {
    ParametersPanel frame = new ParametersPanel();
    frame.addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
}

/**
 * Adjusts the simulation parameters based on the values entered in text fields.
 * Parses the values and updates the corresponding parameters in the Parameters
 * class.
 */
private void adjustParameters() {

    Parameters.distanceNodeLandmark = Double.parseDouble(doubleTextFields.get(0).getText());
    Parameters.distanceAnchors = Double.parseDouble(doubleTextFields.get(1).getText());
    Parameters.nrAnchors = (int) Double.parseDouble(doubleTextFields.get(2).getText());
    Parameters.threshold3dVisibility = Double.parseDouble(doubleTextFields.get(3).getText());
    Parameters.globalLandmarkThreshold =
        Double.min(Double.parseDouble(doubleTextFields.get(4).getText()), 0.95);
}
```

```
Parameters.localLandmarkThreshold =
    Double.min(Double.parseDouble(doubleTextFields.get(5).getText()), 0.95);
Parameters.salientNodesPercentile =
    Double.min(Double.parseDouble(doubleTextFields.get(6).getText()), 1.0);
Parameters.wayfindingEasinessThreshold =
    Double.min(Double.parseDouble(doubleTextFields.get(7).getText()), 1.0);
Parameters.globalLandmarknessWeightDistance =
    Double.min(Double.parseDouble(doubleTextFields.get(8).getText()), 1.0);
Parameters.globalLandmarknessWeightAngular =
    Double.min(Double.parseDouble(doubleTextFields.get(9).getText()), 1.0);
Parameters.regionBasedNavigationThreshold = Double.parseDouble(doubleTextFields.get(10).getText());
Parameters.wayfindingEasinessThresholdRegions = Double
    .max(Double.parseDouble(doubleTextFields.get(11).getText()), 1.0);
Parameters.usingDMA = Boolean.parseBoolean(booleanTextFields.get(0).getText());
if (localPathField.getText() != null) {
    Parameters.localPath = localPathField.getText();
    Parameters.javaProject = true;
}
}

/**
 * Closes the Panel.
 */
private void closePanel() {
    setVisible(false);
    dispose();
}
}

-----PedSimCityApplet.java-----
package pedSim.applet;

import java.awt.Button;
import java.awt.Checkbox;
import java.awt.Choice;
import java.awt.Color;
import java.awt.Frame;
import java.awt.Label;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;
```

```
import java.util.stream.IntStream;

import pedSim.agents.Agent;
import pedSim.engine.Environment;
import pedSim.engine.FlowHandler;
import pedSim.engine.Import;
import pedSim.engine.Parameters;
import pedSim.engine.PedSimCity;
import sim.engine.SimState;


```

```
modeLabel.setBounds(10, 40, 100, 20);
add(modeLabel);

modeChoice = new Choice();
modeChoice.setBounds(140, 40, 230, 20);
//modeChoice.add("Testing Landmarks");
//modeChoice.add("Testing Urban Subdivisions");
//modeChoice.add("Testing Specific Route Choice Models");
modeChoice.add("Empirical ABM");
modeChoice.addItemListener(this); // Add ItemListener to handle changes in modeChoice
add(modeChoice);

Label cityNameLabel = new Label("City Name:");
cityNameLabel.setBounds(10, 70, 80, 20);
add(cityNameLabel);

cityName = new Choice();
cityName.setBounds(140, 70, 150, 20);
updateCityNameOptions(); // Set initial options based on the default selection of modeChoice
add(cityName);

specificODcheckBox = new Checkbox("Testing Specific ODs");
specificODcheckBox.setEnabled(true);
specificODcheckBox.setBounds(10, 100, 300, 40);
specificODcheckBox.addItemListener(this);
add(specificODcheckBox);

Button otherOptionsButton = new Button("Other Options");
otherOptionsButton.setBounds(10, 150, 150, 30);
otherOptionsButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        openParametersPanel();
    }
});
add(otherOptionsButton);

Button routeChoiceButton = new Button("Choose Route-Choices");
routeChoiceButton.setEnabled(false);
routeChoiceButton.setBounds(10, 200, 150, 30);
routeChoiceButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        openTestPanel();
    }
});
add(routeChoiceButton);

modeChoice.addItemListener(new ItemListener() {
```

```
public void itemStateChanged(ItemEvent e) {
    String selectedMode = modeChoice.getSelectedItem();
    // Enable the button only if the selected mode is "Testing Specific Route
    // Choice// Models"
    routeChoiceButton.setEnabled(selectedMode.equals("Testing Specific Route Choice Models"));
    specificODcheckBox.setEnabled(!selectedMode.equals("Empirical ABM"));
}
});

verboseCheckBox = new Checkbox("Verbose y/n");
verboseCheckBox.setEnabled(true);
verboseCheckBox.setBounds(10, 230, 300, 40);
verboseCheckBox.addItemListener(this);
add(verboseCheckBox);

Color color = new Color(0, 220, 0);
startButton = new Button("Run Simulation");
startButton.setBounds(10, 280, 120, 50);
startButton.setBackground(color);
add(startButton);

endButton = new Button("End Simulation");
endButton.setBackground(Color.PINK);

startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        endButton.setBounds(10, 280, 120, 50);
        startButtonParallel.setVisible(false);
        add(endButton);
        startButton.setVisible(false);
        simulationThread = new Thread(() -> startSimulation(false));
        simulationThread.start();
    }
});
};

endButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (simulationThread != null && simulationThread.isAlive()) {
            System.exit(0);
        }
    }
});
};

jobLabel = new Label("Executing Job Nr:");
jobLabel.setBounds(140, 280, 120, 20);
jobLabel.setVisible(false);
```

```
        add(jobLabel);

        remainingTripsLabel = new Label("Trips left (jobs avg):");
        remainingTripsLabel.setBounds(140, 310, 170, 20);
        remainingTripsLabel.setVisible(false);
        add(remainingTripsLabel);

        // Parallel
        startButtonParallel = new Button("Run in Parallel");
        startButtonParallel.setBounds(10, 350, 120, 50);
        startButtonParallel.setBackground(color);
        add(startButtonParallel);

        startButtonParallel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                endButton.setBounds(10, 350, 120, 50);
                startButton.setVisible(false);
                add(endButton);
                startButtonParallel.setVisible(false);
                simulationThread = new Thread(() -> startSimulation(true));
                simulationThread.start();
            }
        });
    });

    jobsLabel = new Label("Parallelising ? Jobs");
    jobsLabel.setBounds(140, 350, 120, 20);
    jobsLabel.setVisible(false);
    add(jobsLabel);

    remainingTripsLabelParallel = new Label("Trips left (jobs avg):");
    remainingTripsLabelParallel.setBounds(140, 380, 170, 20);
    remainingTripsLabelParallel.setVisible(false);
    add(remainingTripsLabelParallel);

    setSize(450, 450);
    setVisible(true);

}

/**
 * Handles changes in the state of the SpecificODcheckbox by either opening or
 * closing the SpecificODpanel.
 *
 * @param e The ItemEvent that triggered the change in the state of the
 *          SpecificODcheckbox.
 */
public void ODpanelStateChanged(ItemEvent e) {
```

```
SpecificODpanel specificODpanel = new SpecificODpanel();
if (specificODcheckbox.getState())
    specificODpanel.handleSpecificODCheckbox(specificODcheckbox);
else
    specificODpanel.closeSpecificODCheckbox();
}

/**
 * Opens the 'OtherOptionsPanel', allowing the user to configure additional
 * simulation options.
 */
private void openParametersPanel() {
    ParametersPanel parametersFrame = new ParametersPanel();
    parametersFrame.setVisible(true); // Display the frame
}

/**
 * Opens the 'TestPanel', which enables the user to select testing options and
 * parameters.
 */
private void openTestPanel() {
    TestPanel testPanel = new TestPanel();
    testPanel.setVisible(true); // Display the frame
}

/**
 * Initiates the simulation with the selected parameters and starts the
 * simulation process. This method sets the city name, simulation mode, and
 * other parameters before running the simulation.
 */
private void startSimulation(boolean runInParallel) {

    Parameters.cityName = cityName.getSelectedItem();
    Parameters.stringMode = modeChoice.getSelectedItem();
    Parameters.defineMode();

    // Run the simulation with the updated parameters
    runSimulation(runInParallel);
}

/**
 * Initiates the simulation with the selected parameters and starts the
 * simulation process. This method sets the city name, simulation mode, and
 * other parameters before running the simulation.
 */
private void runSimulation(boolean runInParallel) {
```

```

try {
    Import importer = new Import();
    importer.importFiles();
} catch (Exception e) {
    // Handle the exception or log an error message
    e.printStackTrace();
    // Additional handling logic...
}

Environment.prepare();
LOGGER.info("Environment Prepared. About to Start Simulation");
if (runInParallel) {
    remainingTripsCount = Parameters.empirical
        ? Parameters.numAgents * Parameters.numberTripsPerAgent * Parameters.jobs * 3
        // nr configurations
        : Parameters.numAgents * Parameters.numberTripsPerAgent * Parameters.jobs;
    updateRemainingTripsLabel(true);
    jobsLabel.setText("Parallelising " + Parameters.jobs + " Jobs");
    jobsLabel.setVisible(true);
    remainingTripsLabelParallel.setVisible(true);

} else {
    remainingTripsCount =
    Parameters.empirical ? Parameters.numAgents * Parameters.numberTripsPerAgent * 3 // nr
    // configurations
        : Parameters.numAgents * Parameters.numberTripsPerAgent;
    updateRemainingTripsLabel(false);
    jobLabel.setVisible(true);
    remainingTripsLabel.setVisible(true);
}

ArrayList<FlowHandler> flowHandlers = new ArrayList<>();

if (runInParallel)
    IntStream.range(0, Parameters.jobs).parallel().forEach(job -> {
        final SimState state = new PedSimCity(System.currentTimeMillis(), job);
        state.start();
        List<Agent> agentList = ((PedSimCity) state).getAgentsList();
        while (state.schedule.step(state)) {
            remainingTripsCount = agentList.parallelStream()
                .mapToInt(agent -> agent.Od.size() - agent.tripsDone).sum() * Parameters.jobs;
            updateRemainingTripsLabel(true);
        }
        flowHandlers.add(((PedSimCity) state).flowHandler);
    });

else {
}

```

```
        for (int job = 0; job < Parameters.jobs; job++) {
            jobLabel.setText("Executing Job Nr: " + job);
            final SimState state = new PedSimCity(System.currentTimeMillis(), job);
            state.start();
            List<Agent> agentList = ((PedSimCity) state).getAgentsList();
            while (state.schedule.step(state)) {
                remainingTripsCount = agentList.parallelStream()
                    .mapToInt(agent -> agent.OD.size() - agent.tripsDone).sum();
                updateRemainingTripsLabel(false);
            }
            flowHandlers.add(((PedSimCity) state).flowHandler);
        }
    }

    handleEndSimulation();
}

private void handleEndSimulation() {
    Label endLabel = new Label("Simulation has ended. Close the window to exit.");
    endLabel.setBounds(10, 410, 300, 30);
    add(endLabel);
}

/**
 * The main entry point for the PedSimCityApplet application.
 *
 * @param args an array of command-line arguments (not used in this
 *             application).
 */
public static void main(String[] args) {
    PedSimCityApplet applet = new PedSimCityApplet();
    applet.addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosing(WindowEvent e) {
            applet.dispose();
        }
    });
}

// This method updates the available options for cityName based on the selected
// modeChoice
private void updateCityNameOptions() {
    cityName.removeAll(); // Clear existing options

    // Get the selected modeChoice
    String selectedMode = modeChoice.getSelectedItem();
```

```

// Add cityName options based on the selected mode
if (selectedMode.equals("Testing Landmarks")) {
    cityName.add("London");
    cityName.add("Muenster");
} else if (selectedMode.equals("Testing Urban Subdivisions")) {
    cityName.add("London");
    cityName.add("Paris");
    cityName.add("Muenster");
} else if (selectedMode.equals("Empirical ABM")) {
    cityName.add("UPB");
    cityName.add("Muenster");
} else if (selectedMode.equals("Testing Specific Route Choice Models")) {
    cityName.add("Muenster");
}
cityName.validate(); // Validate the layout to reflect changes in options
}

/**
 * Invoked when an item's state changes. This method handles changes in the
 * state of specificODcheckbox and modeChoice components.
 *
 * @param e an ItemEvent object that provides information about the event (e.g.,
 *          which item's state changed).
 */
@Override
public void itemStateChanged(ItemEvent e) {
    if (e.getSource() == specificODcheckbox)
        ODpanelStateChanged(e);
    // Call the ODpanelStateChanged method when specificODcheckbox state changes
    else if (e.getSource() == modeChoice)
        updateCityNameOptions();
    else
        Parameters.verboseMode = true;
}

/**
 * Updates the text of the remainingTripsLabel to display the count of missing
 * trips.
 */
private void updateRemainingTripsLabel(boolean runInParallel) {
    if (runInParallel)
        remainingTripsLabelParallel.setText("Trips left (jobs avg): " + remainingTripsCount);
    else
        remainingTripsLabel.setText("Trips left: " + remainingTripsCount);
}
}

```

```
-----SpecificODpanel.java-----
package pedSim.applet;

import java.awt.Button;
import java.awt.Checkbox;
import java.awt.Frame;
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import pedSim.engine.Parameters;

/**
 * A GUI panel for handling specific origin-destination routes.
 */
public class SpecificODpanel extends Frame {

    private static final long serialVersionUID = 1L;
    private static TextField originsField;
    private static TextField destinationsField;
    private static Frame specificODFrame;

    /**
     * Handles the specific origin-destination checkbox action.
     *
     * @param specificODcheckbox The checkbox that triggers this action.
     */
    public void handleSpecificODCheckbox(Checkbox specificODcheckbox) {

        // Open a new window for entering origins and destinations
        specificODFrame = new Frame("Testing Specific Routes");
        specificODFrame.setLayout(null);

        Label originsLabel = new Label("Origins:");
        originsLabel.setBounds(10, 40, 80, 20);
        specificODFrame.add(originsLabel);
        originsField = new TextField();
        originsField.setBounds(140, 40, 400, 20);
        specificODFrame.add(originsField);

        Label destinationsLabel = new Label("Destinations:");
        destinationsLabel.setBounds(10, 70, 80, 20);
```

```
specificODFrame.add(destinationsLabel);
destinationsField = new TextField();
destinationsField.setBounds(140, 70, 400, 20);
specificODFrame.add(destinationsField);

Button saveButton = new Button("Save");
saveButton.setBounds(10, 100, 80, 30);
saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        inputODs();
    }
});
specificODFrame.add(saveButton);
specificODFrame.setSize(600, 150);
specificODFrame.setVisible(true);

specificODFrame.addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        // Update the state of the checkbox when the window is closed
        specificODcheckbox.setState(false);
        specificODFrame.dispose();
    }
});
}

/**
 * Closes the specific origin-destination input window.
 */
public void closeSpecificODCheckbox() {
    if (specificODFrame != null) {
        specificODFrame.dispose();
        specificODFrame = null;
    }
}

/**
 * Parses user-input origins and destinations, validates them, and updates
 * relevant parameters.
 */
private static void inputODs() {
    // Parse the values and update UserParameters
    String[] originsArray = originsField.getText().split(",");
    String[] destinationsArray = destinationsField.getText().split(",");

    // Validate the lengths of origins and destinations
    if (originsArray.length != destinationsArray.length) {
```

```
        showErrorMessage("The number of origins must match the number of destinations.");
        return;
    }

    Integer[] origins;
    Integer[] destinations;

    try {
        origins = new Integer[originsArray.length];
        destinations = new Integer[destinationsArray.length];

        for (int i = 0; i < originsArray.length; i++) {
            origins[i] = Integer.parseInt(originsArray[i].trim());
        }

        for (int i = 0; i < destinationsArray.length; i++) {
            destinations[i] = Integer.parseInt(destinationsArray[i].trim());
        }
    } catch (NumberFormatException ex) {
        showErrorMessage("Invalid input. Please enter valid integers for origins and destinations.");
        return;
    }

    Parameters.originsTmp = origins;
    Parameters.destinationsTmp = destinations;
    Parameters.testingSpecificOD = true;
    specificODFrame.dispose(); // Close the window
}

/**
 * Displays an error message or alert with the given message.
 *
 * @param message The error message to display.
 */
private static void showErrorMessage(String message) {
    // Display an error message or show an alert
    System.err.println("Error: " + message);
}
```

-----TestPanel.java-----

```
package pedSim.applet;

import java.awt.Button;
import java.awt.Checkbox;
import java.awt.Frame;
```

```
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;

import pedSim.engine.Parameters;
import pedSim.utilities.StringEnum.RouteChoice;

/**
 * A graphical user interface (GUI) panel for setting and saving test
 * parameters.
 */
public class TestPanel extends Frame {

    private static final long serialVersionUID = 1L;
    private List<RouteChoice> selectedChoices;
    private TextField jobsTextField;
    private TextField numTripsPerAgentField;

    /**
     * Initialises the TestPanel GUI with checkboxes and input fields.
     */
    public TestPanel() {
        setTitle("Testing panel");
        setLayout(null);

        selectedChoices = new ArrayList<>();

        // Add an action listener to the Save button to get the selected choices
        Button saveButton = new Button("Save");
        saveButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                getTestParameters(e); // Call the method 'getTestParameters()' when the button is clicked
            }
        });

        int value = 30;
        // Create checkboxes for each RouteChoice
        for (RouteChoice choice : RouteChoice.values()) {
            Checkbox checkbox = new Checkbox(choice.toString());
            checkbox.setName(choice.name());
            checkbox.setBounds(10, value, 400, 20);
            value += 30;
            add(checkbox);
        }
    }
}
```

```
// Additional components for setting user-defined parameters
Label jobsLabel = new Label("Jobs:");
add(jobsLabel);
jobsLabel.setBounds(10, value, 80, 20);
jobsTextField = new TextField();
jobsTextField.setBounds(170, value, 100, 20);
add(jobsTextField);
value += 30;

Label numTripsPerAgentLabel = new Label("Number of Trips per Agent:");
add(numTripsPerAgentLabel);
numTripsPerAgentLabel.setBounds(10, value, 155, 20);
numTripsPerAgentField = new TextField();
numTripsPerAgentField.setBounds(170, value, 100, 20);
add(numTripsPerAgentField);
value += 30;

saveButton.setBounds(10, value, 80, 30);
add(saveButton);
setSize(400, 700);
setVisible(true);
}

/**
 * Retrieves the selected choices from checkboxes and user-defined parameters
 * from input fields.
 *
 * @param e The ActionEvent triggered by the Save button.
 */
public void getTestParameters(ActionEvent e) {
    // Update the selectedChoices list based on the checkboxes
    selectedChoices.clear();

    for (int i = 0; i < getComponentCount(); i++) {
        if (getComponent(i) instanceof Checkbox) {
            Checkbox checkbox = (Checkbox) getComponent(i);
            if (checkbox.getState()) {
                selectedChoices.add(RouteChoice.valueOf(checkbox.getName()));
            }
        }
    }
    // Print the selected choices
    System.out.println("Selected Choices: " + selectedChoices);

    // Update user-defined parameters
    try {
```

```

        Parameters.routeChoiceUser = new RouteChoice[selectedChoices.size()];
        for (int counter = 0; counter < selectedChoices.size(); counter++) {
            Parameters.routeChoiceUser[counter] = selectedChoices.get(counter);
        }
        Parameters.jobs = Integer.parseInt(jobsTextField.getText());
        Parameters.numberTripsPerAgent = Integer.parseInt(numTripsPerAgentField.getText());
    } catch (NumberFormatException ex) {
        System.err.println("Invalid input for jobs or numAgents.");
    }
    Parameters.testingModels = true;
    this.dispose(); // Close the window
}

/**
 * Gets the list of selected RouteChoice values.
 *
 * @return A list of selected RouteChoice values.
 */
public List<RouteChoice> getSelectedChoices() {
    return selectedChoices;
}
}

```

-----AgentCognitiveMap.java-----

```

package pedSim.cognitiveMap;

import sim.field.geo.VectorLayer;

/**
 * Represents an agent's cognitive map, which provides access to various map
 * attributes. In this version of PedSimCity, this is a simple structure
 * designed for further developments.
 */
public class AgentCognitiveMap extends CommunityCognitiveMap {

    /**
     * Constructs an AgentCognitiveMap.
     */
    public AgentCognitiveMap() {

    }

    /**
     * Gets the local landmarks from the cognitive map.
     *
     * @return The local landmarks.
     */

```

```
public VectorLayer getLocalLandmarks() {
    return CommunityCognitiveMap.localLandmarks;
}

/**
 * Gets the global landmarks from the cognitive map.
 *
 * @return The global landmarks.
 */
public VectorLayer getGlobalLandmarks() {
    return CommunityCognitiveMap.globalLandmarks;
}

/**
 * Gets the barriers from the cognitive map.
 *
 * @return The barriers.
 */
public VectorLayer getBarriers() {
    return CommunityCognitiveMap.barriers;
}

/**
 * Verifies if any barriers are represented (contained) in the cognitive map.
 *
 * @return The barriers.
 */
public boolean containsBarriers() {
    return !getBarriers().getGeometries().isEmpty();
}
}

-----Barrier.java-----
package pedSim.cognitiveMap;

import java.util.ArrayList;

import sim.graph.EdgeGraph;
import sim.util.geo.MasonGeometry;

/**
 * Represents a barrier in the cognitive map, defined by a unique identifier,
 * geometry, edges it affects, and its type.
 */
public class Barrier {
```

```
/**  
 * The unique identifier of the barrier.  
 */  
public int barrierID;  
  
/**  
 * The MasonGeometry representing the barrier's shape.  
 */  
public MasonGeometry masonGeometry;  
  
/**  
 * The list of edges along the barrier.  
 */  
public ArrayList<EdgeGraph> edgesAlong = new ArrayList<EdgeGraph>();  
  
/**  
 * The type of the barrier, such as "water," "park," etc.  
 */  
public String type;  
}  
  
-----BarrierIntegration.java-----  
package pedSim.cognitiveMap;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.HashMap;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Map;  
import java.util.Set;  
  
import org.locationtech.jts.geom.Geometry;  
  
import pedSim.agents.Agent;  
import pedSim.utilities.StringEnum.BarrierType;  
import sim.field.geo.VectorLayer;  
import sim.graph.EdgeGraph;  
import sim.graph.NodeGraph;  
import sim.graph.SubGraph;  
import sim.util.geo.Angles;  
import sim.util.geo.AttributeValue;  
import sim.util.geo.MasonGeometry;  
  
public class BarrierIntegration {
```

```

/**
 * Returns a set of barriers in the direction of the destination node from a
 * given location.
 *
 * @param currentLocation The current node.
 * @param destinationNode The destination node.
 * @param agent           The agent whose barrier type should be considered.
 * @return A mapping of the viewField and the barrierIDs intersecting it.
 */
public Map<Geometry, Set<Integer>>
intersectingBarriers(NodeGraph currentLocation, NodeGraph destinationNode,
                     Agent agent) {

    Map<Geometry, Set<Integer>> viewFieldIntersectingBarriers = new HashMap<Geometry, Set<Integer>>();
    Set<Integer> intersectingBarrierIDs = new HashSet<>();
    BarrierType agentBarrierType = agent.getProperties().barrierType;
    Geometry viewField = Angles.viewField(currentLocation, destinationNode, 70.0);

    VectorLayer barriers = agent.cognitiveMap.getBarriers();
    if (barriers.getGeometries().isEmpty())
        return viewFieldIntersectingBarriers;

    List<MasonGeometry> intersectingGeometries = barriers.intersectingFeatures(viewField);
    if (intersectingGeometries.isEmpty())
        return viewFieldIntersectingBarriers;

    intersectingGeometries.stream().filter(barrierGeometry -> {
        String barrierType = barrierGeometry.getStringAttribute("type");
        return agentBarrierType.equals(BarrierType.ALL)
            || (agentBarrierType.equals(BarrierType.POSITIVE)
                && (barrierType.equals("park") || barrierType.equals("water")))
            || (agentBarrierType.equals(BarrierType.NEGATIVE) && (barrierType.equals("railway")
                || barrierType.equals("road") || barrierType.equals("secondary_road")))
            || (agentBarrierType.equals(BarrierType.SEPARATING) && !barrierType.equals("park")));
    }).forEach(barrierGeometry -> intersectingBarrierIDs.add(barrierGeometry.
        getIntegerAttribute("barrierID")));

    viewFieldIntersectingBarriers.put(viewField, intersectingBarrierIDs);
    return viewFieldIntersectingBarriers;
}

/**
 * Sets the barrier information for an EdgeGraph based on attribute values. This
 * method parses attribute strings representing different types of barriers,
 * such as positive barriers, negative barriers, rivers, and parks, and
 * populates the corresponding lists in the EdgeGraph. The method retrieves
 * attribute values for positive barriers ("p_barr"), negative barriers

```

```

* ("n_barr"), rivers ("a_rivers"), and parks ("w_parks") from the EdgeGraph's
* attributes. It parses these strings to extract barrier IDs and adds them to
* the appropriate lists: positiveBarriers, negativeBarriers, waterBodies, and
* parks. Additionally, it combines positive and negative barriers into the
* 'barriers' list for convenient access.
*
* @param edge The EdgeGraph for which barrier information is being set.
*/
public static void setEdgeGraphBarriers(EdgeGraph edge) {

    List<Integer> positiveBarriers = new ArrayList<>();
    List<Integer> negativeBarriers = new ArrayList<>();
    List<Integer> barriers = new ArrayList<>(); // all the barriers
    List<Integer> waterBodies = new ArrayList<>();
    List<Integer> parks = new ArrayList<>();

    final String pBarriersString = edge.attributes.get("p_barr").getString();
    final String nBarriersString = edge.attributes.get("n_barr").getString();
    final String riversString = edge.attributes.get("a_rivers").getString();
    final String parksString = edge.attributes.get("w_parks").getString();

    if (!pBarriersString.equals("[]")) {
        final String p = pBarriersString.replaceAll("[^-?0-9]+", " ");
        for (final String t : Arrays.asList(p.trim().split(" ")))
            positiveBarriers.add(Integer.valueOf(t));
    }
    edge.attributes.put("positiveBarriers", new AttributeValue(positiveBarriers));

    if (!nBarriersString.equals("[]")) {
        final String n = nBarriersString.replaceAll("[^-?0-9]+", " ");
        for (final String t : Arrays.asList(n.trim().split(" ")))
            negativeBarriers.add(Integer.valueOf(t));
    }
    edge.attributes.put("negativeBarriers", new AttributeValue(negativeBarriers));

    if (!riversString.equals("[]")) {
        final String r = riversString.replaceAll("[^-?0-9]+", " ");
        for (final String t : Arrays.asList(r.trim().split(" ")))
            waterBodies.add(Integer.valueOf(t));
    }
    edge.attributes.put("waterBodies", new AttributeValue(waterBodies));

    if (!parksString.equals("[]")) {
        final String p = parksString.replaceAll("[^-?0-9]+", " ");
        for (final String t : Arrays.asList(p.trim().split(" ")))
            parks.add(Integer.valueOf(t));
    }
    edge.attributes.put("parks", new AttributeValue(parks));
}

```

```

        barriers.addAll(positiveBarriers);
        barriers.addAll(negativeBarriers);
        edge.attributes.put("barriers", new AttributeValue(barriers));
    }

    /**
     * It stores information about the barriers within a given SubGraph.
     *
     * @param subGraph The SubGraph for which the barrier information is being set.
     */
    public static void setSubGraphBarriers(SubGraph subGraph) {

        List<Integer> graphBarriers = new ArrayList<>();
        for (EdgeGraph childEdge : subGraph.getEdges()) {
            graphBarriers.addAll(subGraph.getParentEdge(childEdge).attributes.get("barriers").getArray());
            // remove duplicates
            graphBarriers = new ArrayList<>(new HashSet<>(graphBarriers));
            subGraph.attributes.put("graphBarriers", new AttributeValue(graphBarriers));
        }
    }

    /**
     * Returns the list of barrier IDs associated with the given subgraph. These
     * barriers represent physical obstacles or features within the subgraph that
     * shape movement and agent's cognitive maps.
     *
     * @param subGraph The SubGraph for which the barrier information is being
     *                 requested.
     *
     * @return The list of barrier IDs within the subgraph.
     */
    public static List<Integer> getSubGraphBarriers(SubGraph subGraph) {
        return subGraph.attributes.get("graphBarriers").getArray();
    }
}

-----CommunityCognitiveMap.java-----
package pedSim.cognitiveMap;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

import org.javatuples.Pair;

```

```
import org.locationtech.jts.geom.Geometry;

import pedSim.engine.Parameters;
import pedSim.engine.PedSimCity;
import sim.field.geo.VectorLayer;
import sim.graph.Building;
import sim.graph.Graph;
import sim.graph.GraphUtils;
import sim.graph.NodeGraph;
import sim.util.geo.MasonGeometry;

/**
 * This class represent a community share cognitive map (or Image of the City)
 * used for storing meaningful information about the environment and, in turn,
 * navigating.
 */
public class CommunityCognitiveMap {

    /**
     * Stores local landmarks as a VectorLayer.
     */
    public static VectorLayer localLandmarks = new VectorLayer();

    /**
     * Stores global landmarks as a VectorLayer.
     */
    public static VectorLayer globalLandmarks = new VectorLayer();

    /**
     * Maps pairs of nodes to gateways.
     */
    public Map<Pair<NodeGraph, NodeGraph>, Gateway> gatewaysMap = new HashMap<>();

    /**
     * Stores barriers as a VectorLayer.
     */
    protected static VectorLayer barriers;

    /**
     * Graphs for navigation.
     */
    static Graph communityNetwork;
    static Graph communityDualNetwork;

    /**
     * Singleton instance of the CognitiveMap.
     */
}
```

```
private static final CommunityCognitiveMap instance = new CommunityCognitiveMap();

Building buildingsHandler = new Building();

/**
 * Sets up the community cognitive map.
 */
public void setCommunityCognitiveMap() {

    if (!PedSimCity.buildings.getGeometries().isEmpty()) {
        identifyLandmarks();
        integrateLandmarks();
    }
    identifyRegionElements();

    barriers = PedSimCity.barriers;
    setCommunityNetwork(PedSimCity.network, PedSimCity.dualNetwork);
}

private void setCommunityNetwork(Graph network, Graph dualNetwork) {
    communityNetwork = network;
    communityDualNetwork = dualNetwork;
}

/**
 * Gets the singleton instance of CognitiveMap.
 *
 * @return The CognitiveMap instance.
 */
public static CommunityCognitiveMap getInstance() {
    return instance;
}

/**
 * Identifies and sets both local and global landmarks in the buildings dataset.
 */
private static void identifyLandmarks() {
    setLandmarks(PedSimCity.buildings);
}

/**
 * Integrates landmarks into the street network, sets local landmarkness, and
 * computes global landmarkness values for nodes.
 */
private static void integrateLandmarks() {
    // Integrate landmarks into the street network
```

```

List<Integer> globalLandmarksID = globalLandmarks.getIntColumn("buildingID");
VectorLayer sightLinesLight = PedSimCity.sightLines.selectFeatures("buildingID",
    globalLandmarksID, true);
// free up memory
PedSimCity.sightLines = null;
LandmarkIntegration landmarkIntegration = new LandmarkIntegration(PedSimCity.network);
landmarkIntegration.setLocalLandmarkness(localLandmarks, PedSimCity.buildingsMap,
    Parameters.distanceNodeLandmark);
landmarkIntegration.setGlobalLandmarkness(globalLandmarks, PedSimCity.buildingsMap,
    Parameters.distanceAnchors,
    sightLinesLight, Parameters.nrAnchors);
}

/**
 * Integrates landmarks into the street network, sets local landmarkness, and
 * computes global landmarkness values for nodes.
 */
private void identifyRegionElements() {

    boolean integrateLandmarks = false;
    if (!PedSimCity.buildings.getGeometries().isEmpty())
        integrateLandmarks = true;

    for (final Entry<Integer, Region> entry : PedSimCity.regionsMap.entrySet()) {
        Region region = entry.getValue();
        if (integrateLandmarks) {
            region.buildings = getBuildingsWithinRegion(region);
            setRegionLandmarks(region);
        }
        BarrierIntegration.setSubGraphBarriers(region.primalGraph);
    }
    barriers = PedSimCity.barriers;
}

/**
 * Sets landmarks (local or global) from a set of buildings (VectorLayer) based
 * on a threshold set initially by the user.
 *
 * @param buildings The set of buildings.
 */
private static void setLandmarks(VectorLayer buildings) {

    List<MasonGeometry> buildingsGeometries = buildings.getGeometries();
    for (final MasonGeometry building : buildingsGeometries) {
        if (building.getDoubleAttribute("lScore_sc") >= Parameters.localLandmarkThreshold)
            localLandmarks.addGeometry(building);
        if (building.getDoubleAttribute("gScore_sc") >= Parameters.globalLandmarkThreshold)

```

```
        globalLandmarks.addGeometry(building);
    }
}

/**
 * Sets region landmarks (local or global) from a set of buildings (ArrayList)
 * based on a threshold set initially by the user.
 *
 * @param region The region for which to set landmarks.
 */
private static void setRegionLandmarks(Region region) {

    for (MasonGeometry building : region.buildings) {
        if (building.getDoubleAttribute("lScore_sc") >= Parameters.localLandmarkThreshold)
            region.localLandmarks.add(building);
        if (building.getDoubleAttribute("gScore_sc") >= Parameters.globalLandmarkThreshold)
            region.globalLandmarks.add(building);
    }
}

/**
 * Gets local landmarks for a specific region.
 *
 * @param region The region for which to get local landmarks.
 * @return A list of local landmarks.
 */
public List<MasonGeometry> getRegionLocalLandmarks(Region region) {
    return region.localLandmarks;
}

/**
 * Gets global landmarks for a specific region.
 *
 * @param region The region for which to get global landmarks.
 * @return A list of global landmarks.
 */
public List<MasonGeometry> getRegionGlobalLandmarks(Region region) {
    return region.globalLandmarks;
}

/**
 * Returns all the buildings enclosed between two nodes.
 *
 * @param originNode      The first node.
 * @param destinationNode The second node.
 * @return A list of buildings.
 */

```

```

public List<MasonGeometry> getBuildings(NodeGraph originNode, NodeGraph destinationNode) {
    Geometry smallestCircle = GraphUtils.enclosingCircleBetweenNodes(originNode, destinationNode);
    return PedSimCity.buildings.containedFeatures(smallestCircle);
}

/**
 * Get buildings within a specified region.
 *
 * @param region The region for which buildings are to be retrieved.
 * @return An ArrayList of MasonGeometry objects representing buildings within
 *         the region.
 */
public List<MasonGeometry> getBuildingsWithinRegion(Region region) {
    VectorLayer regionNetwork = region.regionNetwork;
    Geometry convexHull = regionNetwork.getConvexHull();
    return PedSimCity.buildings.containedFeatures(convexHull);
}

public Graph getKnownNetwork() {
    return communityNetwork;
}

public Graph getKnownDualNetwork() {
    return communityDualNetwork;
}
}

-----Gateway.java-----
package pedSim.cognitiveMap;

import org.javatuples.Pair;

import sim.graph.NodeGraph;

/**
 * Represents a gateway connecting two nodes.
 */
public class Gateway {

    /**
     * The exit node of the gateway.
     */
    public NodeGraph exit;

    /**
     * The entry node of the gateway.
     */
}

```

```
*/  
public NodeGraph entry;  
  
/**  
 * The unique identifier for the gateway.  
 */  
public Pair<NodeGraph, NodeGraph> gatewayID;  
  
/**  
 * The identifier of the node associated with this gateway.  
 */  
public Integer nodeID;  
  
/**  
 * The region identifier for the region the gateway leads to.  
 */  
public Integer regionTo;  
  
/**  
 * The identifier of the edge associated with this gateway.  
 */  
public Integer edgeID;  
  
/**  
 * The distance of the gateway.  
 */  
public Double distance;  
  
/**  
 * Indicates whether this gateway is part of a cognitive map.  
 */  
public boolean cognitiveMap;  
  
/**  
 * The entry angle of the gateway.  
 */  
public Double entryAngle;  
}  
  
-----LandmarkIntegration.java-----
```

```
package pedSim.cognitiveMap;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
import java.util.Map;
```

```

import pedSim.engine.PedSimCity;
import sim.field.geo.VectorLayer;
import sim.graph.Building;
import sim.graph.Graph;
import sim.graph.NodeGraph;
import sim.util.geo.AttributeValue;
import sim.util.geo.MasonGeometry;

/**
 * Manages the integration of landmarks into a graph.
 */
public class LandmarkIntegration {

    // TODO check which this graph thing
    private Graph graph;

    public LandmarkIntegration(Graph graph) {
        this.graph = graph;
    }

    /**
     * It assigns to each node in the graph a list of local landmarks.
     *
     * @param localLandmarks the layer containing all the buildings possibly
     *                      considered as local landmarks;
     * @param buildingsMap   the map of buildings (buildingID, Building);
     * @param radius          the maximum distance from a node to a building for the
     *                      building to be considered a local landmark at the
     *                      junction;
     */
    public void setLocalLandmarkness(VectorLayer localLandmarks, Map<Integer, Building> buildingsMap,
                                    double radius) {

        List<NodeGraph> nodes = graph.nodesGraph;
        nodes.forEach((node) -> {
            List<MasonGeometry> containedLandmarks = localLandmarks
                .featuresWithinDistance(node.getMasonGeometry().geometry, radius);
            for (MasonGeometry masonGeometry : containedLandmarks)
                node.adjacentBuildings.add(buildingsMap.get((int) masonGeometry.getUserData()));
        });
    }

    /**
     * It assigns to each node in the graph a list of distant landmarks and their
     * corresponding global landmarkness values.
     *

```

```

* @param globalLandmarks the layer containing all the buildings possibly
*                         considered as global landmarks;
* @param buildingsMap    the map of buildings (buildingID, Building);
* @param radiusAnchors   the distance radius within which a global landmark is
*                         considered to be an anchor of a node (when intended as
*                         destination node);
* @param sightLines       the layer containing the sight lines;
* @param nrAnchors        the max number of anchors per node, sorted by global
*                         landmarkness;
*/
public void setGlobalLandmarkness(VectorLayer globalLandmarks, Map<Integer, Building> buildingsMap,
        double radiusAnchors, VectorLayer sightLines, int nrAnchors) {

    List<NodeGraph> nodes = graph.nodesGraph;

    nodes.forEach((node) -> {
        ArrayList<Building> anchors = new ArrayList<>(); //
        ArrayList<Double> distances = new ArrayList<>();

        List<MasonGeometry> containedLandmarks = globalLandmarks
            .featuresWithinDistance(node.getMasonGeometry().geometry, radiusAnchors);
        List<Double> gScores = new ArrayList<>();

        if (nrAnchors != -1) {
            for (MasonGeometry masonGeometry : containedLandmarks) {
                gScores.add(masonGeometry.getDoubleAttribute("gScore_sc"));

            }
            Collections.sort(gScores);
            Collections.reverse(gScores);
        }

        for (Object landmark : containedLandmarks) {
            MasonGeometry building = (MasonGeometry) landmark;
            if (nrAnchors != 999999 & building.getDoubleAttribute("gScore_sc")
                < gScores.get(nrAnchors - 1))
                continue;
            int buildingID = (int) building.getUserData();
            anchors.add(buildingsMap.get(buildingID));
            distances.add(building.geometry.distance(node.getMasonGeometry().geometry));
        }

        node.attributes.put("anchors", new AttributeValue(anchors));
        node.attributes.put("distances", new AttributeValue(distances));
    });

    List<MasonGeometry> sightLinesGeometries = sightLines.getGeometries();
}

```

```

        for (MasonGeometry sightLine : sightLinesGeometries) {
            Building building = buildingsMap.getIntegerAttribute("buildingID"));
            NodeGraph node = PedSimCity.nodesMap.getIntegerAttribute("nodeID"));
            if (node != null)
                node.visibleBuildings3d.add(building);
        }
    }

    /**
     * Retrieves the attribute value of the "anchors" for the given NodeGraph.
     *
     * @param node The NodeGraph from which to retrieve the attribute value.
     * @return The attribute value associated with "anchors".
     */
    public static AttributeValue getAnchors(NodeGraph node) {
        return node.attributes.get("anchors");
    }

    /**
     * Retrieves the attribute value of the "distances" for the given NodeGraph.
     *
     * @param node The NodeGraph from which to retrieve the attribute value.
     * @return The attribute value associated with "distances".
     */
    public static AttributeValue getDistances(NodeGraph node) {
        return node.attributes.get("distances");
    }
}

-----NodeWrappper.java-----
package pedSim.dijkstra;

import org.locationtech.jts.planargraph.DirectedEdge;

import sim.graph.NodeGraph;

/**
 * This class stores nodes' metainformation and supports Dijkstra-based
 * algorithms.
 */
public class NodeWrapper {

    public NodeGraph node;
    public NodeGraph nodeFrom;
    public DirectedEdge directedEdgeFrom;
    public NodeGraph commonPrimalJunction;
}

```

```
public double gx, hx, fx, landmarkness;
public int nodesSoFar;
public double pathCost, nodeLandmarkness, pathLandmarkness;
public NodeWrapper previousWrapper;

/** 
 * Constructs a new NodeWrapper for the corresponding NodeGraph.
 *
 * @param node The corresponding NodeGraph.
 */
public NodeWrapper(NodeGraph node) {
    this.node = node;
    gx = 0;
    hx = 0;
    fx = 0;
    nodeFrom = null;
    directedEdgeFrom = null;
    commonPrimalJunction = null;
    pathCost = 0.0;
    nodeLandmarkness = 0.0;
    pathLandmarkness = 0.0;
}
}
```

-----Environment.java-----

```
package pedSim.engine;

import java.util.ArrayList;
import java.util.List;
import java.util.Map.Entry;

import org.javatuples.Pair;
import org.locationtech.jts.geom.Geometry;
import org.locationtech.jts.planagraph.DirectedEdge;
import org.locationtech.jts.planagraph.DirectedEdgeStar;

import pedSim.cognitiveMap.Barrier;
import pedSim.cognitiveMap.BarrierIntegration;
import pedSim.cognitiveMap.CommunityCognitiveMap;
import pedSim.cognitiveMap.Gateway;
import pedSim.cognitiveMap.Region;
import sim.field.geo.VectorLayer;
import sim.graph.Building;
import sim.graph.EdgeGraph;
import sim.graph.GraphUtils;
import sim.graph.NodeGraph;
```

```
import sim.graph.SubGraph;
import sim.util.geo.Angles;
import sim.util.geo.MasonGeometry;

/**
 * The Environment class is responsible for preparing the simulation
 * environment, including junctions, buildings, barriers, and regions.
 */
public class Environment {

    /**
     * Prepares the simulation environment by initializing junctions, buildings,
     * barriers, attributes, dual graph, and regions (if barriers are present).
     */
    public static void prepare() {

        prepareGraph();
        if (!PedSimCity.buildings.getGeometries().isEmpty())
            prepareBuildings();
        if (!PedSimCity.barriers.getGeometries().isEmpty())
            identifyGateways();
        prepareDualGraph();

        if (!PedSimCity.barriers.getGeometries().isEmpty()) {
            integrateBarriers();
            prepareRegions();
        }
    }

    CommunityCognitiveMap cognitiveMap = new CommunityCognitiveMap();
    cognitiveMap.setCommunityCognitiveMap();
}

/**
 * Nodes: Assigns scores and attributes to nodes.
 */
static private void prepareGraph() {

    List<MasonGeometry> geometries = PedSimCity.junctions.getGeometries();

    for (final MasonGeometry nodeGeometry : geometries) {
        // street junctions and betweenness centrality
        final NodeGraph node = PedSimCity.network.findNode(nodeGeometry.geometry.getCoordinate());
        node.setID(nodeGeometry.getIntegerAttribute("nodeID"));
        node.setMasonGeometry(nodeGeometry);
        setCentralityNode(nodeGeometry, node);
        PedSimCity.nodesMap.put(node.getID(), node);
    }
}
```

```

// generate the centrality map of the graph
PedSimCity.network.generateCentralityMap();
createEdgesMap();
}

static private void setCentralityNode(MasonGeometry nodeGeometry, NodeGraph node) {

    double centrality = Double.MAX_VALUE;
    try {
        centrality = nodeGeometry.getDoubleAttribute("Bc_multi");
    } catch (final java.lang.NullPointerException e) {
        centrality = nodeGeometry.getDoubleAttribute("Bc_Rd");
    }
    node.setCentrality(centrality);
}

/**
 * Creates the edgesMap (edgeID, edgeGraph mapping) .
 */
static private void createEdgesMap() {

    List<EdgeGraph> edges = PedSimCity.network.getEdges();
    for (EdgeGraph edge : edges) {
        int edgeID = edge.attributes.getInteger("edgeID");
        edge.setID(edgeID);
        PedSimCity.edgesMap.put(edgeID, edge);
    }
}

/**
 * Landmarks: Assign landmark scores to buildings.
 */
static private void prepareBuildings() {

    List<MasonGeometry> geometries = PedSimCity.buildings.getGeometries();
    for (final MasonGeometry buildingGeometry : geometries) {
        final Building building = new Building();
        building.buildingID = buildingGeometry.getIntegerAttribute("buildID");
        building.landUse = buildingGeometry.getStringAttribute("land_use");
        building.DMA = buildingGeometry.getStringAttribute("DMA");
        building.geometry = buildingGeometry;
        building.attributes.put("globalLandmarkness", buildingGeometry.getAttributes().get("gScore_sc"));
        building.attributes.put("localLandmarkness", buildingGeometry.getAttributes().get("lScore_sc"));

        final List<MasonGeometry> nearestNodes = PedSimCity.junctions
            .featuresWithinDistance(buildingGeometry.getGeometry(), 500.0);
        MasonGeometry closest = null;
}

```

```

        double lowestDistance = 501.0;

        for (final MasonGeometry node : nearestNodes) {
            final double distance = node.getGeometry().distance(buildingGeometry.getGeometry());

            if (distance < lowestDistance) {
                closest = node;
                lowestDistance = distance;
            }
        }

        building.node = closest != null ?
            PedSimCity.network.findNode(closest.getGeometry().getCoordinate()) : null;
        PedSimCity.buildingsMap.put(building.buildingID, building);
    }

    PedSimCity.network.getNodes().forEach((node) -> {
        List<MasonGeometry> nearestBuildings = PedSimCity.buildings
            .featuresWithinDistance(node.getMasonGeometry().geometry, 100);
        for (MasonGeometry building : nearestBuildings) {
            final int buildingID = building.getIntegerAttribute("buildID");
            final String DMA = PedSimCity.buildingsMap.get(buildingID).DMA;
            node.DMA = DMA;
        }
    });
}

/***
 * Gateways: Configures gateways between nodes.
 */
static private void identifyGateways() {

    // creating regions
    for (final NodeGraph node : PedSimCity.network.nodesGraph) {
        node.regionID = node.getMasonGeometry().getIntegerAttribute("district");
        PedSimCity.regionsMap.computeIfAbsent(node.regionID, region -> new Region());
    }

    for (final NodeGraph node : PedSimCity.nodesMap.values()) {
        final Integer gatewayID = node.getMasonGeometry().getIntegerAttribute("gateway");

        if (gatewayID == 0) {
            PedSimCity.startingNodes.add(node.getMasonGeometry());
            continue;
        }

        final Integer regionID = node.regionID;
        for (final EdgeGraph bridge : node.getEdges()) {

```

```

        final NodeGraph oppositeNode = (NodeGraph) bridge.getOppositeNode(node);
        final int possibleRegionID = oppositeNode.regionID;
        if (possibleRegionID == regionID)
            continue;

        final Gateway gateway = new Gateway();
        gateway.exit = node;
        gateway.edgeID = bridge.getID();
        gateway.gatewayID = new Pair<>(node, oppositeNode);
        gateway.regionTo = possibleRegionID;
        gateway.entry = oppositeNode;
        gateway.distance = bridge.getLength();
        gateway.entryAngle = Angles.angle(node, oppositeNode);
        PedSimCity.regionsMap.get(regionID).gateways.add(gateway);
        PedSimCity.gatewaysMap.put(new Pair<>(node, oppositeNode), gateway);
        node.gateway = true;
    }
    node.adjacentRegions = node.getAdjacentRegion();
}
}

/**
 * Centroids (Dual Graph): Assigns edgeID to centroids in the dual graph.
 */
static private void prepareDualGraph() {

    List<MasonGeometry> centroids = PedSimCity.centroids.getGeometries();
    for (final MasonGeometry centroidGeometry : centroids) {
        int edgeID = centroidGeometry.getIntegerAttribute("edgeID");
        NodeGraph centroid = PedSimCity.dualNetwork.findNode(centroidGeometry.geometry.getCoordinate());
        centroid.setID(edgeID);
        centroid.setPrimalEdge(PedSimCity.edgesMap.get(edgeID));
        PedSimCity.edgesMap.get(edgeID).setDualNode(centroid);
        PedSimCity.centroidsMap.put(edgeID, centroid);
    }

    List<EdgeGraph> dualEdges = PedSimCity.dualNetwork.getEdges();
    for (EdgeGraph edge : dualEdges)
        edge.setDeflectionAngle(edge.attributes.get("deg").getDouble());
}

/**
 * Regions: Creates regions' subgraphs and store information.
 */
static private void integrateBarriers() {

    List<EdgeGraph> edges = PedSimCity.network.getEdges();

```

```

        for (EdgeGraph edge : edges)
            BarrierIntegration.setEdgeGraphBarriers(edge);
        generateBarriersMap();
    }

    /**
     * Generates a map of barriers based on provided data.
     */
    private static void generateBarriersMap() {

        // Element 5 - Barriers: create barriers map
        List<MasonGeometry> geometries = PedSimCity.barriers.getGeometries();
        for (final MasonGeometry barrierGeometry : geometries) {
            final int barrierID = barrierGeometry.getIntegerAttribute("barrierID");
            final Barrier barrier = new Barrier();
            barrier.masonGeometry = barrierGeometry;
            barrier.type = barrierGeometry.getStringAttribute("type");
            final ArrayList<EdgeGraph> edgesAlong = new ArrayList<>();

            for (final EdgeGraph edge : PedSimCity.network.getEdges()) {
                ArrayList<Integer> edgeBarriers = edge.attributes.get("barriers").getArray();
                if (!edgeBarriers.isEmpty() && edgeBarriers.contains(barrierID))
                    edgesAlong.add(edge);
            }

            barrier.edgesAlong = edgesAlong;
            barrier.type = barrierGeometry.getStringAttribute("type");
            PedSimCity.barriersMap.put(barrierID, barrier);
        }
    }

    private static void prepareRegions() {

        List<EdgeGraph> edges = PedSimCity.network.getEdges();
        for (EdgeGraph edge : edges) {

            if (edge.getFromNode().regionID == edge.getToNode().regionID) {
                int regionID = edge.getFromNode().regionID;
                edge.regionID = regionID;
                PedSimCity.regionsMap.get(regionID).edges.add(edge);
            } else
                // gateway edge
                edge.regionID = -1;
        }

        for (final Entry<Integer, Region> entry : PedSimCity.regionsMap.entrySet()) {
    
```

```

        int regionID = entry.getKey();
        Region region = entry.getValue();

        final List<EdgeGraph> edgesRegion = region.edges;
        final SubGraph primalGraph = new SubGraph(edgesRegion);
        final VectorLayer regionNetwork = new VectorLayer();
        final List<EdgeGraph> dualEdgesRegion = new ArrayList<>();

        for (final EdgeGraph edge : edgesRegion) {
            regionNetwork.addGeometry(edge.getMasonGeometry());
            NodeGraph centroid = edge.getDualNode();
            centroid.regionID = regionID;
            DirectedEdgeStar directedEdges = centroid.getOutEdges();
            for (final DirectedEdge directedEdge : directedEdges.getEdges())
                dualEdgesRegion.add((EdgeGraph) directedEdge.getEdge());
        }
        final SubGraph dualGraph = new SubGraph(dualEdgesRegion);
        primalGraph.generateSubGraphCentralityMap();
        region.regionID = regionID;
        region.primalGraph = primalGraph;
        region.dualGraph = dualGraph;
        region.regionNetwork = regionNetwork;
    }
}

/**
 * Returns all the buildings enclosed between two nodes.
 *
 * @param originNode      The first node.
 * @param destinationNode The second node.
 * @return A list of buildings.
 */
public List<MasonGeometry> getBuildings(NodeGraph originNode, NodeGraph destinationNode) {
    Geometry smallestCircle = GraphUtils.enclosingCircleBetweenNodes(originNode, destinationNode);
    return PedSimCity.buildings.containedFeatures(smallestCircle);
}

/**
 * Get buildings within a specified region.
 *
 * @param region The region for which buildings are to be retrieved.
 * @return An ArrayList of MasonGeometry objects representing buildings within
 *         the region.
 */
public List<MasonGeometry> getBuildingsWithinRegion(Region region) {
    VectorLayer regionNetwork = region.regionNetwork;
    Geometry convexHull = regionNetwork.getConvexHull();
}

```

```
        return PedSimCity.buildings.containedFeatures(convexHull);
    }
}

-----Export.java-----
package pedSim.engine;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.logging.Logger;

import org.apache.commons.lang3.ArrayUtils;

import pedSim.agents.EmpiricalAgentsGroup;
import pedSim.utilities.RouteData;
import pedSim.utilities.StringEnum;
import pedSim.utilities.StringEnum.RouteChoice;
import sim.field.geo.VectorLayer;
import sim.io.geo.ShapeFileExporter;
import sim.util.geo.CSVUtils;
import sim.util.geo.MasonGeometry;

/**
 * The Export class is responsible for saving the simulation results to
 * specified output directories.
 */
public class Export {

    private String userName = System.getProperty("user.name");
    // Constants for file paths and directories
    public String outputDirectory;
    public String outputRoutesDirectory;
    private static final Logger LOGGER = Logger.getLogger(Export.class.getName());
    private static int nrColumns;
    private static final int FIELD_LIMIT = 254;
    int job;
    FlowHandler flowHandler;

    /**

```

```
* Saves simulation results to specified output directories.  
*  
* @throws Exception If there is an error while saving the results.  
*/  
  
public void saveResults(FlowHandler flowHandler) throws Exception {  
    outputDirectory =  
        "C:" + File.separator + "Users" + File.separator + userName + File.separator + "PedSimCity"  
        + File.separator + "Output";  
    outputRoutesDirectory = outputDirectory;  
    this.flowHandler = flowHandler;  
    this.job = flowHandler.job;  
    setOutputPath();  
    savePedestrianVolumes();  
    saveRoutes();  
    LOGGER.info("Job nr " + job + ": Files successfully exported.");  
}  
  
/**  
 * Sets the output path based on the program's configuration.  
 */  
  
private void setOutputPath() {  
    if (Parameters.empirical)  
        outputDirectory += File.separator + "empirical";  
    else if (Parameters.testingSubdivisions)  
        outputDirectory += File.separator + "subdivisions";  
    else if (Parameters.testingLandmarks)  
        outputDirectory += File.separator + "landmarkNavigation";  
    else if (Parameters.testingModels)  
        outputDirectory += File.separator + "testing";  
  
    outputRoutesDirectory = outputDirectory + File.separator + "routes";  
    outputDirectory += File.separator + "streetVolumes";  
    verifyOutputPath();  
}  
  
/**  
 * Verifies and creates the specified output directory.  
 */  
  
private void verifyOutputPath() {  
    String username = System.getProperty("user.name");  
    outputDirectory = String.format(outputDirectory, username);  
    outputRoutesDirectory = String.format(outputRoutesDirectory, username);  
    createDirectory(outputDirectory);  
    createDirectory(outputRoutesDirectory);  
}  
  
/**
```

```

    * Creates a directory if it does not exist.
    *
    * @param directory The directory path to be created.
    */
private void createDirectory(String directory) {

    File outputCheck = new File(directory);
    if (!outputCheck.exists()) {
        try {
            // Create the output path directory and its parent directories recursively
            Files.createDirectories(Paths.get(directory));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

/**
 * Saves pedestrian volumes data to a CSV file.
 *
 * @param job The job identifier.
 * @throws Exception If there is an error while saving the data.
 */
private void savePedestrianVolumes() throws Exception {

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyyMMdd");
    String currentDate = LocalDate.now().format(formatter);
    final String csvSegments = outputDirectory + File.separator + currentDate + "_" + job + ".csv";
    final FileWriter writerVolumesData = new FileWriter(csvSegments);

    Map<Integer, Map<String, Integer>> volumesMap = flowHandler.volumesMap;

    ArrayList<String> headers;
    if (Parameters.empirical) {
        headers = new ArrayList<>();
        for (final EmpiricalAgentsGroup group : PedSimCity.empiricalGroups)
            headers.add(group.groupName.toString());
    } else {
        List<String> abbreviations = new ArrayList<>();
        for (RouteChoice routeChoice : Parameters.routeChoiceModels) {
            abbreviations.add(StringEnum.getAbbreviation(routeChoice));
        }
        headers = new ArrayList<>(abbreviations);
    }
    headers.add(0, "edgeID");
    CSVUtils.writeLine(writerVolumesData, headers);
}

```

```

        for (int edgeID : volumesMap.keySet()) {
            Map<String, Integer> edgeVolumes = volumesMap.get(edgeID);
            final List<String> row = new ArrayList<>();

            for (final String columnHeader : headers) {
                if (columnHeader.equals("edgeID")) {
                    row.add(0, Integer.toString(edgeID));
                    continue;
                }
                int index = headers.indexOf(columnHeader) - 1;
                String value = "";
                if (Parameters.empirical)
                    value = headers.get(index + 1);
                else
                    value = Parameters.routeChoiceModels[index].toString();
                row.add(Integer.toString(edgeVolumes.get(value)));
            }
            CSVUtils.writeLine(writerVolumesData, row);
        }
        writerVolumesData.flush();
        writerVolumesData.close();
    }

    /**
     * Saves pedestrian volumes data to a CSV file.
     *
     * @param job The job identifier.
     * @throws Exception If there is an error while saving the data.
     */
    private void saveRoutes() throws Exception {

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyyMMdd");
        String currentDate = LocalDate.now().format(formatter);
        final VectorLayer routes = new VectorLayer();
        final String directory =
            outputRoutesDirectory + File.separator + currentDate + "_" + flowHandler.job;
        nrColumns = 0;

        for (final RouteData routeData : flowHandler.routesData) {
            MasonGeometry masonGeometry = new MasonGeometry(routeData.lineGeometry);
            masonGeometry.addIntegerAttribute("O", routeData.origin);
            masonGeometry.addIntegerAttribute("D", routeData.destination);
            if (Parameters.empirical)
                updateEmpiricalRouteData(routeData, masonGeometry);
            else
                masonGeometry.addStringAttribute("routeChoice", routeData.routeChoice);
        }
    }
}

```

```

        formRouteAttributes(masonGeometry, routeData);
        routes.addGeometry(masonGeometry);
    }

    // This is to avoid having geometries without the needed columns' values filled
    // in.
    if (nrColumns > 0) {

        for (int counter = 1; counter <= nrColumns - 1; counter++) {
            List<MasonGeometry> routeGeometries = routes.getGeometries();
            for (final MasonGeometry route : routeGeometries)
                if (route.hasAttribute("edgeIDs_" + counter))
                    continue;
                else
                    route.addAttribute("edgeIDs_" + counter, "None");
            }
        }
        ShapeFileExporter.write(directory, routes);
    }

    /**
     * Forms route attributes and handles splitting long edgeIDs strings.
     *
     * @param masonGeometry The MasonGeometry object representing a route.
     * @param routeData      The route data associated with the route.
     */
    private static void formRouteAttributes(MasonGeometry masonGeometry, RouteData routeData) {
        String edgeIDs = ArrayUtils.toString(routeData.edgeIDsSequence);

        if (edgeIDs.length() <= FIELD_LIMIT) {
            masonGeometry.addAttribute("edgeIDs_0", edgeIDs);
        } else {
            String remainingEdges = edgeIDs;
            for (int counter = 0; remainingEdges.length() > 0; counter++) {
                if (counter >= nrColumns) {
                    nrColumns += 1;
                }

                String currentPart;
                if (remainingEdges.length() > FIELD_LIMIT) {
                    currentPart = remainingEdges.substring(0, FIELD_LIMIT);
                    remainingEdges = remainingEdges.substring(FIELD_LIMIT);
                } else {
                    currentPart = remainingEdges;
                    masonGeometry.addAttribute("edgeIDs_" + counter, currentPart);
                    break;
                }
            }
        }
    }
}

```

```

        masonGeometry.addAttribute("edgeIDs_" + counter, currentPart);
    }
}
}

/**
 * Forms route attributes and handles splitting long edgeIDs strings.
 *
 * @param masonGeometry The MasonGeometry object representing a route.
 * @param routeData      The route data associated with the route.
 */
private static void updateEmpiricalRouteData(RouteData routeData, MasonGeometry masonGeometry) {
    masonGeometry.addStringAttribute("group", routeData.group);
    masonGeometry.addStringAttribute("min", getMinimisation(routeData));
    masonGeometry.addStringAttribute("heur", getHeuristic(routeData));
    masonGeometry.addIntegerAttribute("regions", routeData.regionBased ? 1 : 0);
    masonGeometry.addIntegerAttribute("routeMark", routeData.onRouteMarks ? 1 : 0);
    masonGeometry.addIntegerAttribute("barrier", routeData.barrierSubGoals ? 1 : 0);
    masonGeometry.addIntegerAttribute("distant", routeData.distantLandmarks ? 1 : 0);
    masonGeometry.addDoubleAttribute("natural", routeData.naturalBarriers);
    masonGeometry.addDoubleAttribute("severing", routeData.severingBarriers);
}

/**
 * Gets the minimisation type as a string.
 *
 * @param routeData The route data.
 * @return The minimisation type as a string or null if not applicable.
 */
private static String getMinimisation(RouteData routeData) {
    if (routeData.minimisingDistance)
        return "distance";
    else if (routeData.minimisingAngular)
        return "angular";
    else
        return "none";
}

/**
 * Transforms the local minimisation heuristic as a string.
 *
 * @param routeData The route data.
 * @return The minimisation heuristic as a string or null if not applicable.
 */
private static String getHeuristic(RouteData routeData) {
    if (routeData.localHeuristicDistance)
        return "distance";
}

```

```
        else if (routeData.localHeuristicAngular)
            return "angular";
        else
            return "none";
    }
}

-----FlowHandler.java-----
package pedSim.engine;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.locationtech.jts.geom.Coordinate;
import org.locationtech.jts.geom.GeometryFactory;
import org.locationtech.jts.geom.LineString;
import org.locationtech.jts.planagraph.DirectedEdge;

import pedSim.agents.Agent;
import pedSim.agents.AgentProperties;
import pedSim.agents.EmpiricalAgentProperties;
import pedSim.agents.EmpiricalAgentsGroup;
import pedSim.utilities.RouteData;
import pedSim.utilities.StringEnum.RouteChoice;
import sim.graph.EdgeGraph;
import sim.graph.NodeGraph;

/**
 * The Flow class provides methods for updating various data related to agent
 * movement and route storing in the simulation.
 */
public class FlowHandler {

    public Map<Integer, Map<String, Integer>> volumesMap = new HashMap<Integer, Map<String, Integer>>();
    public List<RouteData> routesData = new ArrayList<>();
    public int job;

    public FlowHandler(int job) {
        initializeEdgeVolumes();
        this.job = job;
    }
}
```

```

/**
 * Updates the edge data on the basis of the passed agent's route and its edges
 * sequence.
 *
 * @param agent           The agent for which edge data is updated.
 * @param directedEdgesSequence The sequence of directed edges travelled by the
 *                             agent.
 */
public synchronized void updateEdgeData(Agent agent, List<DirectedEdge> directedEdgesSequence) {

    AgentProperties agentProperties = agent.getProperties();
    String attributeName = Parameters.empirical ?
        ((EmpiricalAgentProperties) agentProperties).groupName.toString()
        : agentProperties.routeChoice.toString();

    for (DirectedEdge directedEdge : directedEdgesSequence) {
        EdgeGraph edge = (EdgeGraph) directedEdge.getEdge();
        Map<String, Integer> edgeVolume = volumesMap.get(edge.getID());
        edgeVolume.replace(attributeName, edgeVolume.get(attributeName) + 1);
        volumesMap.replace(edge.getID(), edgeVolume);
    }
}

/**
 * Stores route data for an agent based on the specified edge IDs sequence.
 *
 * @param agent           The agent for which route data is stored.
 * @param edgeIDsSequence The sequence of edge IDs traveled by the agent.
 */
public void storeRouteData(Agent agent, List<Integer> edgeIDsSequence) {
    RouteData route = createRouteData(agent);
    List<Coordinate> allCoords = computeAllCoordinates(edgeIDsSequence, agent);
    route.edgeIDsSequence = edgeIDsSequence;
    route.lineGeometry = createLineGeometry(allCoords);
    routesData.add(route);
}

/**
 * Initialises the edge volumes for the simulation. This method assigns initial
 * volume values to edges based on the selected route choice models or empirical
 * agent groups. If the simulation is not empirical, it initialises volumes
 * based on the route choice models. If the simulation is empirical-based, it
 * initialises volumes based on empirical agent groups.
 */
private void initializeEdgeVolumes() {

    for (Object o : PedSimCity.network.getEdges()) {

```

```

        Map<String, Integer> edgeVolumes = new HashMap<String, Integer>();
        if (!Parameters.empirical) {
            for (RouteChoice routeChoice : Parameters.routeChoiceModels) {
                EdgeGraph edge = (EdgeGraph) o;
                edgeVolumes.put(routeChoice.toString(), 0);
                volumesMap.put(edge.getID(), edgeVolumes);
            }
        } else {
            for (EmpiricalAgentsGroup empiricalGroup : PedSimCity.empiricalGroups) {
                EdgeGraph edge = (EdgeGraph) o;
                edgeVolumes.put(empiricalGroup.groupName.toString(), 0);
                volumesMap.put(edge.getID(), edgeVolumes);
            }
        }
    }

    /**
     * Creates and initialises a new RouteData object for the given agent.
     *
     * @param agent The agent for which route data is created.
     * @return A RouteData object containing route information.
     */
    private RouteData createRouteData(Agent agent) {
        RouteData route = new RouteData();
        route.origin = agent.originNode.getID();
        route.destination = agent.destinationNode.getID();

        if (Parameters.empirical)
            route = getDataFromEmpiricalAgent(agent, route);
        else
            route.routeChoice = agent.getProperties().routeChoice.toString();

        return route;
    }

    /**
     * Extracts data from an empirical agent's properties and populates the route
     * data.
     *
     * @param agent The empirical agent for which route data is extracted.
     * @param route The RouteData object to be populated with data.
     * @return A RouteData object containing route information.
     */
    private static RouteData getDataFromEmpiricalAgent(Agent agent, RouteData route) {
        EmpiricalAgentProperties agentProperties = (EmpiricalAgentProperties) agent.getProperties();
        route.group = agentProperties.groupName.toString();

```

```

        route.routeID = agent.originNode.getID() + "-" + agent.destinationNode.getID();
        route.minimisingDistance = agentProperties.minimisingDistance ? true : false;
        route.minimisingAngular = agentProperties.minimisingAngular ? true : false;
        route.localHeuristicDistance = agentProperties.localHeuristicDistance ? true : false;
        route.localHeuristicAngular = agentProperties.localHeuristicAngular ? true : false;
        route.barrierSubGoals = agentProperties.barrierBasedNavigation ? true : false;
        route.distantLandmarks = agentProperties.usingDistantLandmarks ? true : false;
        route.regionBased = agentProperties.regionBasedNavigation ? true : false;
        route.onRouteMarks = agentProperties.usingLocalLandmarks ? true : false;
        route.naturalBarriers = agentProperties.naturalBarriers;
        route.severingBarriers = agentProperties.severingBarriers;
        return route;
    }

    /**
     * Computes all coordinates along the sequence of edges for an agent's route.
     *
     * @param sequenceEdges The sequence of edge IDs traveled by the agent.
     * @param agent          The agent for which coordinates are computed.
     * @return A list of coordinates representing the agent's route.
     */
    private List<Coordinate> computeAllCoordinates(List<Integer> sequenceEdges, Agent agent) {
        List<Coordinate> allCoords = new ArrayList<>();
        NodeGraph lastNode = agent.originNode;

        for (int i : sequenceEdges) {
            EdgeGraph edge = PedSimCity.edgesMap.get(i);
            LineString geometry = (LineString) edge.getMasonGeometry().geometry;
            Coordinate[] coords = geometry.getCoordinates();
            List<Coordinate> coordsCollection = new ArrayList<>(Arrays.asList(coords));

            if (coords[0].distance(lastNode.getCoordinate()) > coords[coords.length - 1]
                .distance(lastNode.getCoordinate())) {
                Collections.reverse(coordsCollection);
            }

            coordsCollection.set(0, lastNode.getCoordinate());

            if (lastNode.equals(edge.getFromNode()))
                lastNode = edge.getToNode();
            else if (lastNode.equals(edge.getToNode()))
                lastNode = edge.getFromNode();
            else
                System.out.println("Something is wrong with the sequence in this agent");

            coordsCollection.set(coordsCollection.size() - 1, lastNode.getCoordinate());
            allCoords.addAll(coordsCollection);
        }
    }
}

```

```

        }
        return allCoords;
    }

    /**
     * Creates a LineString geometry from a list of coordinates.
     *
     * @param coordinates The list of coordinates to create a LineString from.
     * @return A LineString geometry representing the route.
     */
    private static LineString createLineGeometry(List<Coordinate> coordinates) {
        GeometryFactory factory = new GeometryFactory();
        Coordinate[] coordsArray = coordinates.toArray(new Coordinate[0]);
        return factory.createLineString(coordsArray);
    }
}

-----Import.java-----
package pedSim.engine;

import java.io.File;
import java.io.InputStreamReader;
import java.io.Reader;
import java.net.URL;
import java.util.Arrays;
import java.util.logging.Logger;

import com.opencsv.CSVReader;

import pedSim.agents.EmpiricalAgentsGroup;
import pedSim.utilities.StringEnum.Groups;
import sim.field.geo.VectorLayer;
import sim.util.geo.MasonGeometry;

/**
 * This class is responsible for importing various data files required for the
 * simulation based on the selected simulation parameters. It includes methods
 * for importing distances, barriers, landmarks and sight lines, road network
 * graphs, and empirical agent groups data.
 */
public class Import {

    /**
     * The base data directory path for the simulation data files.
     */
    String resourcePath;
}

```

```
private static final Logger LOGGER = Logger.getLogger(Import.class.getName());  
  
/**  
 * Imports various data files required for the simulation based on the selected  
 * simulation parameters.  
 *  
 * @throws Exception If an error occurs during the import process.  
 */  
public void importFiles() throws Exception {  
    resourcePath = Parameters.cityName;  
    if (Parameters.javaProject)  
        resourcePath = Parameters.localPath + resourcePath;  
    if (Parameters.cityName.equals("London")) {  
        if (Parameters.testingLandmarks)  
            resourcePath += "/landmarks";  
        else if (Parameters.testingSubdivisions)  
            resourcePath += "/subdivisions";  
    }  
    if (Parameters.testingLandmarks) {  
        importDistances();  
        readLandmarksAndSightLines();  
    } else if (Parameters.testingSubdivisions)  
        readBarriers();  
    else if (Parameters.empirical) {  
        readLandmarksAndSightLines();  
        readBarriers();  
        importEmpiricalGroups();  
    } else if (Parameters.testingModels) {  
        readLandmarksAndSightLines();  
        readBarriers();  
    }  
    // Read the street network shapefiles and create the primal and the dual graph  
    readGraphs();  
}  
  
/**  
 * Imports GPS trajectory-derived distances required for the simulation.  
 *  
 * @throws Exception If an error occurs during the import process.  
 */  
private void importDistances() throws Exception {  
  
    // Read GPS trajectories distances  
    ClassLoader classLoader = getClass().getClassLoader();  
    URL resourceURL = null;  
    if (Parameters.javaProject)  
        resourceURL = new File(resourcePath + "/tracks_distances.csv").toURI().toURL();
```

```

        else
            resourceURL = classLoader.getResource(resourcePath + "/tracks_distances.csv");
        Reader reader = new InputStreamReader(resourceURL.openStream());
        CSVReader readerDistances = new CSVReader(reader);
        String[] nextLineDistances;

        int row = 0;
        while ((nextLineDistances = readerDistances.readNext()) != null) {
            row += 1;
            if (row == 1)
                continue; // Skip header
            PedSimCity.distances.add(Float.parseFloat(nextLineDistances[2]));
        }
        readerDistances.close();
    }

    /**
     * Reads and imports road network graphs required for the simulation.
     *
     * @throws Exception If an error occurs during the import process.
     */
    private void readGraphs() throws Exception {

        ClassLoader classLoader = getClass().getClassLoader();
        URL urlShp = null;
        URL urlDbf = null;

        try {
            String[] strings = { "/edges", "/nodes", "/edgesDual", "/nodesDual" };
            VectorLayer[] vectorLayers = { PedSimCity.roads, PedSimCity.junctions,
                PedSimCity.intersectionsDual,
                PedSimCity.centroids };

            for (String string : strings) {
                String tmpPath = resourcePath + string;
                if (Parameters.javaProject) {
                    urlShp = new File(tmpPath + ".shp").toURI().toURL();
                    urlDbf = new File(tmpPath + ".dbf").toURI().toURL();
                } else {
                    urlShp = classLoader.getResource(tmpPath + ".shp");
                    urlDbf = classLoader.getResource(tmpPath + ".dbf");
                }
                VectorLayer.readShapefile(urlShp, urlDbf,
                    vectorLayers[Arrays.asList(strings).indexOf(string)]);
                LOGGER.info("SUCCESSFULLY imported " + string);
            }
            // Add log statements to print the content of the buildings VectorLayer
        }
    }
}

```

```

        for (Object obj : PedSimCity.roads.getGeometries()) {
            if (obj instanceof MasonGeometry) {
                //LOGGER.info("Edge " + ((MasonGeometry) obj).getIntegerAttribute("edgeID") +
                " imported.");
            }
        }

        for (Object obj : PedSimCity.junctions.getGeometries()) {
            if (obj instanceof MasonGeometry) {
                //LOGGER.info("Node " + ((MasonGeometry) obj).getIntegerAttribute("nodeID") +
                " imported.");
            }
        }

        for (Object obj : PedSimCity.intersectionsDual.getGeometries()) {
            if (obj instanceof MasonGeometry) {
                //LOGGER.info("Dual Graph Edge " + ((MasonGeometry) obj) + " imported.");
            }
        }

        for (Object obj : PedSimCity.centroids.getGeometries()) {
            if (obj instanceof MasonGeometry) {
                //LOGGER.info("Dual Graph Node " + ((MasonGeometry) obj).getIntegerAttribute("edgeID") +
                " imported.");
            }
        }

        PedSimCity.network.fromStreetJunctionsSegments(PedSimCity.junctions, PedSimCity.roads);
        PedSimCity.dualNetwork.fromStreetJunctionsSegments(PedSimCity.centroids,
        PedSimCity.intersectionsDual);
        LOGGER.info("SUCCESSFULLY imported Graphs.");
    } catch (Exception e) {
        handleImportError("Importing Graphs failed", e);
    }
}

/**
 * Reads and imports landmarks and sight lines data for the simulation.
 *
 * @throws Exception If an error occurs during the import process.
 */
private void readLandmarksAndSightLines() throws Exception {
    ClassLoader classLoader = getClass().getClassLoader();
    URL urlShp = null;
    URL urlDbf = null;
    try {
        String[] strings = { "/landmarks" };
        VectorLayer[] vectorLayers = { PedSimCity.buildings };

```

```

        for (String string : strings) {
            String tmpPath = resourcePath + string;
            if (Parameters.javaProject) {
                urlShp = new File(tmpPath + ".shp").toURI().toURL();
                urlDbf = new File(tmpPath + ".dbf").toURI().toURL();
            } else {
                urlShp = classLoader.getResource(tmpPath + ".shp");
                urlDbf = classLoader.getResource(tmpPath + ".dbf");
            }
            VectorLayer.readShapefile(urlShp, urlDbf,
                                      vectorLayers[Arrays.asList(strings).indexOf(string)]);
        }

        // Add log statements to print the content of the buildings VectorLayer
        for (Object obj : PedSimCity.buildings.getGeometries()) {
            if (obj instanceof MasonGeometry) {
                //LOGGER.info("Building " + ((MasonGeometry) obj).getIntegerAttribute("buildID") +
                " imported.");
            }
        }
        PedSimCity.buildings.setID("buildID");
        LOGGER.info("SUCCESSFULLY imported Landmarks.");
    } catch (Exception e) {
        handleImportError("Importing Landmarks FAILED.", e);
    }

    // Add separate try-catch block for sight_lines2D
    try {
        String[] strings = { "/sight_lines2D" };
        VectorLayer[] vectorLayers = { PedSimCity.sightLines };
        for (String string : strings) {
            String tmpPath = resourcePath + string;
            if (Parameters.javaProject) {
                urlShp = new File(tmpPath + ".shp").toURI().toURL();
                urlDbf = new File(tmpPath + ".dbf").toURI().toURL();
            } else {
                urlShp = classLoader.getResource(tmpPath + ".shp");
                urlDbf = classLoader.getResource(tmpPath + ".dbf");
            }
            VectorLayer.readShapefile(urlShp, urlDbf,
                                      vectorLayers[Arrays.asList(strings).indexOf(string)]);
        }

        PedSimCity.sightLines.setID("sightLineID");
        LOGGER.info("SUCCESSFULLY imported sight_lines2D.");
    } catch (Exception e) {
        handleImportError("Importing sight_lines2D FAILED.", e);
    }
}

```

```
        }

    }

    /**
     * Reads and imports barriers data for the simulation.
     *
     * @throws Exception If an error occurs during the import process.
     */
    private void readBarriers() throws Exception {

        URL urlShp = null;
        URL urlDbf = null;
        try {
            String tmpPath = resourcePath + "/barriers";
            if (Parameters.javaProject) {
                urlShp = new File(tmpPath + ".shp").toURI().toURL();
                urlDbf = new File(tmpPath + ".dbf").toURI().toURL();
            } else {
                ClassLoader classLoader = getClass().getClassLoader();
                urlShp = classLoader.getResource(tmpPath + ".shp");
                urlDbf = classLoader.getResource(tmpPath + ".dbf");
            }
            VectorLayer.readShapefile(urlShp, urlDbf, PedSimCity.barriers);
            LOGGER.info("SUCCESSFULLY imported Barriers.");
        } catch (Exception e) {
            handleImportError("Importing Barriers Failed", e);
        }
    }

    /**
     * Imports empirical agent groups data for the simulation.
     *
     * @throws Exception If an error occurs during the import process.
     */
    private static void handleImportError(String layerName, Exception e) {
        LOGGER.info(layerName);
    }

    /**
     * Imports empirical agent groups data for the simulation.
     *
     * @throws Exception If an error occurs during the import process.
     */
    private void importEmpiricalGroups() throws Exception {

        ClassLoader classLoader = getClass().getClassLoader();
        URL resourceURL = null;
```

```

        if (Parameters.javaProject)
            resourceURL = new File(resourcePath + "/clusters_71.csv").toURI().toURL();
        else
            resourceURL = classLoader.getResource(resourcePath + "/clusters_71.csv");
        LOGGER.info("SUCCESSFULLY imported clusters.");
        Reader reader = new InputStreamReader(resourceURL.openStream());
        CSVReader readerEmpiricalGroups = new CSVReader(reader);
        String[] nextLine;

        int row = 0;
        while ((nextLine = readerEmpiricalGroups.readNext()) != null) {
            row += 1;
            if (row == 1)
                continue;
            final EmpiricalAgentsGroup empiricalGroup = new EmpiricalAgentsGroup();
            final String groupName = nextLine[0];
            empiricalGroup.setGroup(Groups.valueOf(groupName), nextLine);
            PedSimCity.empiricalGroups.add(empiricalGroup);
        }
        readerEmpiricalGroups.close();
    }
}

-----Parameters.java-----
package pedSim.engine;

import pedSim.utilities.StringEnum.RouteChoice;

/**
 * The Parameters class contains global parameters and settings for the
 * PedSimCity simulation. These parameters are used to configure various aspects
 * of the simulation, including simulation mode, agent behavior, and data import
 * options.
 */
public class Parameters {

    // General parameters
    public static String cityName = "UPB";
    public static String stringMode = "";
    public static int jobs = 10;
    public static int numAgents = 68;
    public static int numberTripsPerAgent = 5;

    // in seconds. One step == 10 minutes,
    public static double stepTimeUnit = 600;
    // One step == 10 minutes, average speed 1 meter/sec., --> moveRate = 60*10
}

```

```
// meters per second
private static double pedestrianSpeed = 1.42;
// meters per step;
public static double moveRate;

public static boolean testingLandmarks = false;
public static boolean testingSubdivisions = false;
public static boolean testingModels = false;
public static boolean testingSpecificOD = false;

public static boolean testing = false;
public static boolean empirical = false;
public static boolean userDefined = false;

// Other parameters
public static boolean usingDMA = false;
public static double thresholdTurn;

// Distance between Origin and Destination
public static double minDistance = 1000;
public static double maxDistance = 2500;

// Use the enums for route choice models
public static RouteChoice[] routeChoiceLandmarks =
    { RouteChoice.ROAD_DISTANCE, RouteChoice.LANDMARKS_DISTANCE,
        RouteChoice.ANGULAR_CHANGE, RouteChoice.LANDMARKS_ANGULAR,
        RouteChoice.LOCAL_LANDMARKS_DISTANCE,
        RouteChoice.DISTANT_LANDMARKS };

public static RouteChoice[] routeChoiceSubdivisions =
    { RouteChoice.ANGULAR_CHANGE, RouteChoice.REGION_ANGULAR,
        RouteChoice.BARRIER_ANGULAR, RouteChoice.REGION_BARRIER_ANGULAR };

public static RouteChoice[] routeChoiceUser;
public static RouteChoice[] routeChoiceModels;

public static Integer[] originsTmp = {};
public static Integer[] destinationsTmp = {};

// Landmark Integration
public static double distanceNodeLandmark = 50.0;
public static double distanceAnchors = 2000;
public static double threshold3dVisibility = 300;
public static double globalLandmarkThreshold = 0.30; //
public static double localLandmarkThreshold = 0.30; //
public static double salientNodesPercentile = 0.75; // Threshold Percentile to identify salient nodes
public static int nrAnchors = 25; // to speed-up, it can be higher; it can be lower for more prototypical
```

```
// weight Global Landmarkness in combination with edge costs (road distance)
public static double globalLandmarknessWeightDistance = 0.85;
// weight Global Landmarkness in combination with edge costs (angular change)
public static double globalLandmarknessWeightAngular = 0.95;
public static double regionBasedNavigationThreshold = 500; // Region-based navigation Threshold - meters

// Wayfinding Easiness threshold
public static double wayfindingEasinessThreshold = 0.95; //
    global navigation for local landmark identification
public static double wayfindingEasinessThresholdRegions = 0.85; //
    within regions for local landmark identification

// for development/testing purposes only
public static boolean javaProject = false;
public static boolean verboseMode = false;
public static String localPath = null;

/**
 * Defines the simulation mode and sets simulation parameters based on the
 * selected mode. Called at the beginning of the simulation to configure
 * simulation settings.
 */
public static void defineMode() {

    if (stringMode.equals("Testing Landmarks")) {
        resetParameters();
        testingLandmarks = true;
        routeChoiceModels = routeChoiceLandmarks;
        numAgents = routeChoiceModels.length;
        numberTripsPerAgent = 255;
        jobs = 50;
    } else if (stringMode.equals("Testing Urban Subdivisions")) {
        resetParameters();
        testingSubdivisions = true;
        routeChoiceModels = routeChoiceSubdivisions;
        numAgents = routeChoiceModels.length;
        numberTripsPerAgent = 2000;
        jobs = 10;
    } else if (stringMode.equals("Empirical ABM")) {
        resetParameters();
        empirical = true;
        numAgents = 68;
        numberTripsPerAgent = 5;
        jobs = 20;
    } else if (stringMode.equals("Testing Specific Route Choice Models")) {
        resetParameters();
        testingModels = true;
    }
}
```

```

        routeChoiceModels = routeChoiceUser;
        numAgents = routeChoiceModels.length;
    }
    if (testingSpecificOD)
        numberTripsPerAgent = originsTmp.length;

    isTestingTrue();
    moveRate = stepTimeUnit * pedestrianSpeed;
}

/***
 * Defines the simulation mode and sets simulation parameters based on the
 * selected mode. Called at the beginning of the simulation to configure
 * simulation settings.
 */
private static void resetParameters() {
    testingLandmarks = false;
    testingSubdivisions = false;
    testingModels = false;
    empirical = false;
}

/***
 * Checks if any testing mode (Landmarks, Subdivisions, or Specific Route Choice
 * Models) is active. Updates the 'testing' flag accordingly.
 */
public static void isTestingTrue() {
    testing = testingLandmarks || testingSubdivisions || testingModels;
}
}

-----PedSimCity.java-----
package pedSim.engine;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.javatuples.Pair;
import org.locationtech.jts.geom.Envelope;

import pedSim.agents.Agent;
import pedSim.agents.EmpiricalAgentsGroup;
import pedSim.cognitiveMap.Barrier;
import pedSim.cognitiveMap.Gateway;

```

```
import pedSim.cognitiveMap.Region;
import sim.engine.SimState;
import sim.engine.Stoppable;
import sim.field.geo.VectorLayer;
import sim.graph.Building;
import sim.graph.EdgeGraph;
import sim.graph.Graph;
import sim.graph.NodeGraph;
import sim.util.geo.MasonGeometry;

/**
 * The PedSimCity class represents the main simulation environment.
 */
public class PedSimCity extends SimState {
    private static final long serialVersionUID = 1L;

    // Urban elements: graphs, buildings, etc.
    public static VectorLayer roads = new VectorLayer();
    public static VectorLayer buildings = new VectorLayer();
    public static VectorLayer barriers = new VectorLayer();
    public static VectorLayer junctions = new VectorLayer();
    public static VectorLayer sightLines = new VectorLayer();

    public static Graph network = new Graph();
    public static Graph dualNetwork = new Graph();

    // dual graph
    public static VectorLayer intersectionsDual = new VectorLayer();
    public static VectorLayer centroids = new VectorLayer();

    // supporting HashMaps, bags and Lists
    public static Map<Integer, Building> buildingsMap = new HashMap<>();
    public static Map<Integer, Region> regionsMap = new HashMap<>();
    public static Map<Integer, Barrier> barriersMap = new HashMap<>();
    public static Map<Pair<NodeGraph, NodeGraph>, Gateway> gatewaysMap = new HashMap<>();
    public static Map<Integer, NodeGraph> nodesMap = new HashMap<>();
    public static Map<Integer, EdgeGraph> edgesMap = new HashMap<>();

    public static HashMap<Integer, NodeGraph> centroidsMap = new HashMap<>();

    // OD related variables
    public static List<Float> distances = new ArrayList<>();
    public static List<MasonGeometry> startingNodes = new ArrayList<>();
    // used only when loading OD sets

    public int currentJob;
```

```
public FlowHandler flowHandler;
public static List<EmpiricalAgentsGroup> empiricalGroups = new ArrayList<>();
public static Envelope MBR = null;

public VectorLayer agents;
public ArrayList<Agent> agentsList;

/**
 * Constructs a new instance of the PedSimCity simulation environment.
 *
 * @param seed The random seed for the simulation.
 * @param job The current job number for multi-run simulations.
 */
public PedSimCity(long seed, int job) {
    super(seed);
    this.currentJob = job;
    this.flowHandler = new FlowHandler(job);
    this.agentsList = new ArrayList<>();
    this.agents = new VectorLayer(); // create a new vector layer for each job
}

public List<Agent> getAgentsList() {
    return agentsList;
}

/**
 * Initialises the simulation by defining the simulation mode, initialising edge
 * volumes, and preparing the simulation environment. It then proceeds to
 * populate the environment with agents and starts the agent movement.
 */
@Override
public void start() {
    super.start();
    prepareEnvironment();
    populateEnvironment();
    startMovingAgents();
}

/**
 * Prepares the environment for the simulation. This method sets up the minimum
 * bounding rectangle (MBR) to encompass both the road and building layers and
 * updates the MBR of the road layer accordingly.
 */
private void prepareEnvironment() {
    MBR = roads.getMBR();
    if (!buildings.getGeometries().isEmpty())
        MBR.expandToInclude(buildings.getMBR());
```

```

        if (!barriers.getGeometries().isEmpty())
            MBR.expandToInclude(barriers.getMBR());
        roads.setMBR(MBR);
    }

    /**
     * Populates the simulation environment with agents and other entities based on
     * the selected simulation parameters. This method uses the Populate class to
     * generate the agent population.
     */
    private void populateEnvironment() {
        Populate populate = new Populate();
        if (Parameters.testing)
            populate.populateTests(this);
        if (Parameters.empirical)
            populate.populateEmpiricalGroups(this);
        else {
        }
        // Populate.populate();
    }

    /**
     * Starts moving agents in the simulation. This method schedules agents for
     * repeated movement updates and sets up the spatial index for agents.
     */
    private void startMovingAgents() {
        for (Agent agent : this.agentsList) {
            Stoppable stop = schedule.scheduleRepeating(agent);
            agent.setStoppable(stop);
            schedule.scheduleRepeating(agents.scheduleSpatialIndexUpdater(), Integer.MAX_VALUE, 1.0);
        }
        agents.setMBR(MBR);
    }

    /**
     * Completes the simulation by saving results and performing cleanup operations.
     */
    @Override
    public void finish() {

        try {
            Export export = new Export();
            export.saveResults(flowHandler);
        } catch (final Exception e) {
            e.printStackTrace();
        }
        super.finish();
    }
}

```

```

    }

    /**
     * The main function that allows the simulation to be run in stand-alone,
     * non-GUI mode.
     *
     * @param args Command-line arguments.
     * @throws Exception If an error occurs during simulation execution.
     */
    public static void main(String[] args) throws Exception {

        Parameters.defineMode();
        Import importer = new Import();
        importer.importFiles();
        Environment.prepare();

        for (int job = 0; job < Parameters.jobs; job++) {
            System.out.println("Run number... " + job);
            final SimState state = new PedSimCity(System.currentTimeMillis(), job);
            state.start();
            while (state.schedule.step(state)) {
            }
        }
        System.exit(0);
    }
}

-----Populate.java-----
package pedSim.engine;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import org.javatuples.Pair;

import pedSim.agents.Agent;
import pedSim.agents.EmpiricalAgentsGroup;
import pedSim.utilities.StringEnum.Groups;
import pedSim.utilities.StringEnum.RouteChoice;
import sim.graph.Graph;
import sim.graph.NodeGraph;
import sim.graph.NodesLookup;
import sim.util.geo.MasonGeometry;

/**
 * The Populate class is responsible for generating test agents, building the OD

```

```

 * matrix, and populating empirical groups for pedestrian simulation.
 */
public class Populate {

    private PedSimCity state;
    private Graph network = new Graph();
    private final ArrayList<Pair<NodeGraph, NodeGraph>> OD = new ArrayList<>();

    public static boolean usingDMA = true;
    public static HashMap<String, Double> destinationsDMA = new HashMap<>();
    List<Integer> testOrigins = new ArrayList<>();
    List<Integer> testDestinations = new ArrayList<>();
    final double WORK_SHARE = 0.30;
    final double VISIT_SHARE = 0.46;
    final double RANDOM_SHARE = 0.24;
    final String TYPE_LIVE = "live";

    /**
     * Populates test agents, OD matrix, and empirical groups for pedestrian
     * simulation.
     *
     * @param state The PedSimCity simulation state.
     */
    public void populateTests(PedSimCity state) {

        this.state = state;
        this.network = PedSimCity.network;

        if (Parameters.testingSpecificOD)
            prepareManualODmatrix();
        generateTestODmatrix();
        generateTestAgents();
    }

    /**
     * Prepares a manual OD matrix based on specified test origins and destinations.
     */
    public void prepareManualODmatrix() {

        testOrigins.clear();
        testDestinations.clear();
        for (Integer nodeID : Parameters.originsTmp)
            testOrigins.add(nodeID);
        for (Integer nodeID : Parameters.destinationsTmp)
            testDestinations.add(nodeID);
    }
}

```

```

/**
 * Generates the OD matrix for the test-based simulations.
 */
private void generateTestODmatrix() {

    // a trip per agent
    for (int i = 0; i < Parameters.numberTripsPerAgent; i++) {
        NodeGraph originNode = null;
        NodeGraph destinationNode = null;

        if (Parameters.testingSpecificOD) {
            originNode = PedSimCity.nodesMap.get(testOrigins.get(i));
            destinationNode = PedSimCity.nodesMap.get(testDestinations.get(i));
        } else if (Parameters.testingLandmarks) {
            originNode = NodesLookup.randomNode(network);
            destinationNode = NodesLookup.randomNodeFromDistancesSet(network,
                PedSimCity.junctions, originNode,
                PedSimCity.distances);
        } else if (Parameters.testingSubdivisions) {
            originNode = NodesLookup.randomNodeFromList(network, PedSimCity.startingNodes);
            destinationNode = NodesLookup.randomNodeBetweenDistanceInterval(network, originNode, 1000,
                3000);
        } else if (Parameters.testingModels) {
            originNode = NodesLookup.randomNodeFromList(network, PedSimCity.startingNodes);
            destinationNode = NodesLookup.randomNodeBetweenDistanceInterval(network, originNode,
                Parameters.minDistance, Parameters.maxDistance);
        }

        Pair<NodeGraph, NodeGraph> pair = new Pair<>(originNode, destinationNode);
        this.OD.add(pair);
    }
}

/**
 * Generates test agents for the simulation.
 */
private void generateTestAgents() {

    // One Model, One Agent
    Parameters.numAgents = Parameters.routeChoiceModels.length;
    final RouteChoice[] routeChoiceModels = Parameters.routeChoiceModels;
    for (int agentID = 0; agentID < Parameters.numAgents; agentID++) {
        Agent agent = new Agent(this.state);
        agent.initialiseAgentProperties();
        agent.getProperties().setRouteChoice(routeChoiceModels[agentID]);
        addAgent(agent, agentID, OD);
    }
}

```

```

    }

    /**
     * Adds an agent to the simulation.
     *
     * @param agent      The agent to be added.
     * @param agentID    The identifier of the agent.
     * @param thisAgentODs The OD matrix for this agent.
     */
    private void addAgent(Agent agent, int agentID, ArrayList<Pair<NodeGraph, NodeGraph>> thisAgentODs) {

        MasonGeometry agentGeometry = agent.getGeometry();
        agent.OD = new LinkedList<>(thisAgentODs);
        agentGeometry.isMovable = true;
        agent.agentID = agentID;
        state.agents.addGeometry(agentGeometry);
        state.agentsList.add(agent);
    }

    /**
     * Populates empirical groups for pedestrian simulation.
     *
     * @param state The PedSimCity simulation state.
     */
    public void populateEmpiricalGroups(PedSimCity state) {

        this.state = state;
        this.network = PedSimCity.network;
        final int numODs = Parameters.numAgents * Parameters.numberTripsPerAgent;

        if (Parameters.usingDMA)
            usingDMA(numODs);
        else
            for (int i = 0; i < numODs; i++) {
                //NodeGraph originNode = NodesLookup.randomNode(network);
                //NodeGraph destinationNode =
                NodesLookup.randomNodeBetweenDistanceInterval(network, junctions,
                //      originNode, Parameters.minDistance, Parameters.maxDistance);

                NodeGraph originNode = null;
                // Define the possible destination nodes
                NodeGraph destinationNode = null;

                // Set origin node to nodeID 23
                for (NodeGraph node : network.getNodes()) {
                    if (node.getID() == 23) {

```

```

        originNode = node;
        break;
    }
}

int[] selectableNodes = {13, 16, 19};
int randomIndex = (int) (Math.random() * selectableNodes.length);
int selectedNodeId = selectableNodes[randomIndex];
for (NodeGraph node : network.getNodes()) {
    if (node.getID() == selectedNodeId) {
        destinationNode = node;
        break;
    }
}

while (originNode == destinationNode) {
    int randomIndexForEqualOD = (int) (Math.random() * selectableNodes.length);
    selectedNodeId = selectableNodes[randomIndexForEqualOD];
    for (NodeGraph node : network.getNodes()) {
        if (node.getID() == selectedNodeId) {
            destinationNode = node;
            break;
        }
    }
}

// Make sure the originNode and destinationNode are not the same
//while (originNode == destinationNode) {
//    destinationNode = NodesLookup.randomNode(network);
//}

//while (destinationNode.gateway)
// destinationNode =
    NodesLookup.randomNodeBetweenDistanceInterval(network, junctions, originNode,
    // Parameters.minDistance, Parameters.maxDistance);
Pair<NodeGraph, NodeGraph> pair = new Pair<>(originNode, destinationNode);
OD.add(pair);
originNode = destinationNode = null;
}

assignODmatrixToEmpiricalGroups();
}

/**
 * Generates an OD matrix on the basis of the Urban DMA categorisation.
 *
 * @param numODs The number of OD pairs to generate.
 */

```

```

private void usingDMA(int numODs) {

    setDMAmap();
    final HashMap<String, Integer> nrDestinationsDMA = new HashMap<>();
    int agentsToAllocate = numODs;
    String DMA = "";
    NodeGraph originNode = null;
    NodeGraph destinationNode = null;

    for (String typeDMA : destinationsDMA.keySet()) {
        int nr = (int) (destinationsDMA.get(typeDMA) * numODs);
        if (nr > agentsToAllocate)
            nr = agentsToAllocate;
        nrDestinationsDMA.put(typeDMA, nr);
    }

    for (int i = 0; i < numODs; i++) {
        // Set origin node to nodeID 23
        for (NodeGraph node : network.getNodes()) {
            if (node.getID() == 23) {
                originNode = node;
                break;
            }
        }

        for (String typeDMA : nrDestinationsDMA.keySet()) {
            int nr = nrDestinationsDMA.get(typeDMA);
            if (nr < 1)
                continue;
            else {
                DMA = typeDMA;
                nrDestinationsDMA.put(typeDMA, nr - 1);
                break;
            }
        }
    }

    int[] selectableNodes = {13, 16, 19};
    int randomIndex = (int) (Math.random() * selectableNodes.length);
    int selectedNodeId = selectableNodes[randomIndex];
    for (NodeGraph node : network.getNodes()) {
        if (node.getID() == selectedNodeId) {
            destinationNode = node;
            break;
        }
    }

    while (originNode == destinationNode) {

```

```

        int randomIndexForEqualOD = (int) (Math.random() * selectableNodes.length);
        selectedNodeId = selectableNodes[randomIndexForEqualOD];
        for (NodeGraph node : network.getNodes()) {
            if (node.getID() == selectedNodeId) {
                destinationNode = node;
                break;
            }
        }
    }

    Pair<NodeGraph, NodeGraph> pair = new Pair<>(originNode, destinationNode);
    OD.add(pair);
    originNode = destinationNode = null;
}
}

/**
 * Sets up the DMA map with destination types and shares.
 */
private void setDMAmap() {

    // Specify the initial capacity based on the number of elements to be added
    destinationsDMA = new HashMap<>(3);
    destinationsDMA.put("work", WORK_SHARE);
    destinationsDMA.put("visit", VISIT_SHARE);
    destinationsDMA.put("random", RANDOM_SHARE);
}

/**
 * Assigns the OD matrix to empirical groups for pedestrian simulation.
 */
// private void assignODmatrixToEmpiricalGroups() {
//
//     int agentID = 0;
//     final ArrayList<EmpiricalAgentsGroup> actualEmpiricalGroups = new ArrayList<>();
//     ArrayList<Pair<NodeGraph, NodeGraph>> configurationOD = new ArrayList<>(OD);
//     int numAgentsEmpiricalGroup;
//     int agentsToAllocate = Parameters.numAgents;
//
//     for (EmpiricalAgentsGroup empiricalGroup : PedSimCity.empiricalGroups) {
//         // if testing empiricalGroup
//         if (!empiricalGroup.groupName.equals("POPULATION") &&
//             !empiricalGroup.groupName.equals("NULLGROUP"))
//             actualEmpiricalGroups.add(empiricalGroup);
//         //
//         if (empiricalGroup.groupName.equals("POPULATION") ||
//             empiricalGroup.groupName.equals("NULLGROUP")) {

```

```

////      numAgentsEmpiricalGroup = Parameters.numAgents;
////      configurationOD = new ArrayList<>(OD);
////  }
///
//    // if testing population and nullgroup
//    if (!empiricalGroup.groupName.equals("POPULATION") &&
//        !empiricalGroup.groupName.equals("NULLGROUP"))
//        actualEmpiricalGroups.add(empiricalGroup);
///
//    if (empiricalGroup.groupName.equals("POPULATION") ||
//        !empiricalGroup.groupName.equals("NULLGROUP")) {
//        numAgentsEmpiricalGroup = Parameters.numAgents;
//        configurationOD = new ArrayList<>(this.OD);
//    }
//    if (!empiricalGroup.groupName.equals("POPULATION") ||
//        empiricalGroup.groupName.equals("NULLGROUP")) {
//        numAgentsEmpiricalGroup = Parameters.numAgents;
//        configurationOD = new ArrayList<>(this.OD);
//    }
///
//    // last group
//    else if (actualEmpiricalGroups.size() == PedSimCity.empiricalGroups.size() - 2)
//        numAgentsEmpiricalGroup = agentsToAllocate;
//    // any other group
//    else {
//        numAgentsEmpiricalGroup = (int) (Parameters.numAgents * empiricalGroup.share);
//        agentsToAllocate -= numAgentsEmpiricalGroup;
//    }
///
//    int groupTripsToComplete = numAgentsEmpiricalGroup * Parameters.numberTripsPerAgent;
//    ArrayList<Pair<NodeGraph, NodeGraph>> groupOD = new ArrayList<>(
//        configurationOD.subList(0, groupTripsToComplete));
///
//    int lowLimit = 0;
//    int upLimit = Parameters.numberTripsPerAgent;
///
//    for (int i = 0; i < numAgentsEmpiricalGroup; i++) {
//        ArrayList<Pair<NodeGraph, NodeGraph>> thisAgentODs = new ArrayList<>();
///
//        if (Parameters.numberTripsPerAgent == 1)
//            thisAgentODs.add(groupOD.get(i));
//        else
//            thisAgentODs = new ArrayList<>(groupOD.subList(lowLimit, upLimit));
///
//        Agent agent = new Agent(this.state);
//        agent.initialiseAgentProperties(empiricalGroup);
//        addAgent(agent, agentID, thisAgentODs);

```

```

// agentID++;
// lowLimit = upLimit;
// upLimit = lowLimit + Parameters.numberTripsPerAgent;
// }
// configurationOD = new ArrayList<>(configurationOD.subList(groupTripsToComplete,
// configurationOD.size()));
// System.out.println("GroupName: " + empiricalGroup.groupName +
// "\t No. of agents: " + numAgentsEmpiricalGroup +
// "\t No. of Origin-Destination pairs: " + groupOD.size());
// }
// }

private void assignODmatrixToEmpiricalGroups() {

    int agentID = 0;
    final ArrayList<EmpiricalAgentsGroup> actualEmpiricalGroups = new ArrayList<>();
    ArrayList<Pair<NodeGraph, NodeGraph>> configurationOD = new ArrayList<>(OD);
    int numAgentsEmpiricalGroup;
    int agentsToAllocate = Parameters.numAgents;

    for (EmpiricalAgentsGroup empiricalGroup : PedSimCity.empiricalGroups) {
        if (!empiricalGroup.groupName.equals(Groups.POPULATION)
            && !empiricalGroup.groupName.equals(Groups.NULLGROUP))
            actualEmpiricalGroups.add(empiricalGroup);

        if (empiricalGroup.groupName.equals(Groups.POPULATION)
            || empiricalGroup.groupName.equals(Groups.NULLGROUP)) {
            numAgentsEmpiricalGroup = Parameters.numAgents;
            configurationOD = new ArrayList<>(OD);
        }
        // last group
        else if (actualEmpiricalGroups.size() == PedSimCity.empiricalGroups.size() - 2)
            numAgentsEmpiricalGroup = agentsToAllocate;
        // any other group
        else {
            numAgentsEmpiricalGroup = (int) (Parameters.numAgents * empiricalGroup.share);
            agentsToAllocate -= numAgentsEmpiricalGroup;
        }

        int groupTripsToComplete = numAgentsEmpiricalGroup * Parameters.numberTripsPerAgent;
        ArrayList<Pair<NodeGraph, NodeGraph>> groupOD = new ArrayList<>(
            configurationOD.subList(0, groupTripsToComplete));

        int lowLimit = 0;
        int upLimit = Parameters.numberTripsPerAgent;
    }
}

```

```

        for (int i = 0; i < numAgentsEmpiricalGroup; i++) {
            ArrayList<Pair<NodeGraph, NodeGraph>> thisAgentODs = new ArrayList<>();

            if (Parameters.numberTripsPerAgent == 1)
                thisAgentODs.add(groupOD.get(i));
            else
                thisAgentODs = new ArrayList<>(groupOD.subList(lowLimit, upLimit));

            Agent agent = new Agent(this.state);
            agent.initialiseAgentProperties(empiricalGroup);
            addAgent(agent, agentID, thisAgentODs);

            agentID++;
            lowLimit = upLimit;
            upLimit = lowLimit + Parameters.numberTripsPerAgent;
        }
        configurationOD = new ArrayList<>(configurationOD.subList(groupTripsToComplete,
            configurationOD.size()));
        System.out.println("GroupName: " + empiricalGroup.groupName +
            "\t No. of agents: " + numAgentsEmpiricalGroup +
            "\t No. of Origin-Destination pairs: " + groupOD.size());
    }
}

-----BarrierBasedNavigation.java-----
package pedSim.routeChoice;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.javatuples.Pair;
import org.locationtech.jts.geom.Coordinate;
import org.locationtech.jts.geom.Geometry;

import pedSim.agents.Agent;
import pedSim.cognitiveMap.Barrier;
import pedSim.cognitiveMap.BarrierIntegration;
import pedSim.cognitiveMap.Region;
import pedSim.engine.PedSimCity;

```

```

import sim.graph.EdgeGraph;
import sim.graph.GraphUtils;
import sim.graph.NodeGraph;
import sim.util.geo.MasonGeometry;
import sim.util.geo.Utilities;

/**
 * Series of functions for computing a sequence of barrier sub-goals in the
 * space between an origin and a destination.
 */
public class BarrierBasedNavigation {

    Map<Integer, EdgeGraph> edgesMap = new HashMap<Integer, EdgeGraph>();
    List<NodeGraph> sequence = new ArrayList<>();
    private NodeGraph originNode;
    private NodeGraph currentLocation;
    private NodeGraph destinationNode;
    private Agent agent;
    boolean regionBasedNavigation = false;
    // it stores the barriers that the agent has already been exposed to
    Set<Integer> visitedBarriers = new HashSet<>();

    /**
     * Initialises a new instance of the BarrierBasedNavigation class.
     *
     * @param originNode      the origin node;
     * @param destinationNode the destination node;
     * @param agent            the agent for navigation.
     */
    public BarrierBasedNavigation(NodeGraph originNode, NodeGraph destinationNode, Agent agent,
        boolean regionBasedNavigation) {
        this.originNode = originNode;
        this.destinationNode = destinationNode;
        this.agent = agent;
        this.regionBasedNavigation = regionBasedNavigation;
    }

    /**
     * Computes a sequence of nodes, including origin, destination, and barrier
     * sub-goals, and identifies traversed regions.
     *
     * @return an ArrayList of NodeGraph representing the sequence of sub-goals.
     * @throws Exception
     */
    public List<NodeGraph> sequenceBarriers() throws Exception {
        edgesMap = new HashMap<Integer, EdgeGraph>(PedSimCity.edgesMap);

```



```

* @return A HashMap containing valid barrier IDs and their respective distances
*         from the currentLocation.
*/
protected Map<Integer, Double> findValidBarriers(NodeGraph currentLocation, Region region) {

    this.currentLocation = currentLocation;
    Map<Integer, Double> validBarriers = new HashMap<>();
    BarrierIntegration barrierIntegration = new BarrierIntegration();
    // check if there are good barriers in line of movement towards the destination
    Map<Geometry, Set<Integer>> viewFieldIntersectingBarriers = barrierIntegration
        .intersectingBarriers(currentLocation, destinationNode, agent);

    // no barriers
    if (viewFieldIntersectingBarriers.isEmpty())
        return validBarriers;

    Geometry viewField = (new ArrayList<Geometry>(viewFieldIntersectingBarriers.keySet())).get(0);
    Set<Integer> intersectingBarriers = viewFieldIntersectingBarriers.get(viewField);

    // when region-based, only keep barriers that are actually within the region
    // boundaries
    if (regionBasedNavigation)
        intersectingBarriers.retainAll(new HashSet<>(BarrierIntegration.getSubGraphBarriers
            (region.primGraph)));

    // assuming that these have been "visited" already
    identifyAdjacentBarriers();
    if (intersectingBarriers.isEmpty())
        return validBarriers;

    Coordinate currentCoordinate = currentLocation.getCoordinate();
    Coordinate destinationCoordinate = destinationNode.getCoordinate();

    // for each candidate barrier, check whether it complies with the criteria
    for (int barrierID : intersectingBarriers) {
        if (visitedBarriers.contains(barrierID))
            continue;
        MasonGeometry barrierGeometry = PedSimCity.barriersMap.get(barrierID).masonGeometry;
        Coordinate[] intersections = viewField.intersection(barrierGeometry.geometry).getCoordinates();

        double minDistance = Arrays.stream(intersections).parallel()
            .mapToDouble(intersection ->
                GraphUtils.euclideanDistance(currentCoordinate, intersection)).min()
            .orElse(Double.MAX_VALUE);

        // barriers that are more distant than the destinationNode are disregarded
        if (minDistance > GraphUtils.euclideanDistance(currentCoordinate, destinationCoordinate))

```

```

        continue;

        validBarriers.put(barrierID, minDistance);
    }

    return validBarriers;
}

/**
 * Identifies the barriers surrounding the current location by examining the
 * incoming edges of the node. Adds the identified barriers to the list of
 * adjacent barriers.
 */
private void identifyAdjacentBarriers() {
    // identify barriers around this currentLocation
    List<EdgeGraph> incomingEdges = currentLocation.getEdges();
    for (EdgeGraph edge : incomingEdges) {
        List<Integer> edgeBarriers = edge.attributes.get("barriers").getArray();
        if (!edgeBarriers.isEmpty())
            visitedBarriers.addAll(edgeBarriers);
    }
}

/**
 * Identifies an edge-subGoals associated with the most attractive barrier,
 * complying with certain criteria, and finds the closest edge to them.
 *
 * @param validBarriers a set of valid, intersecting barriers towards the
 *                      destination.
 * @param region       the region of the currentLocation, only for region-based
 *                      navigation.
 * @return a Pair of EdgeGraph and Integer representing the closest edge to the
 *         barrier and the barrierID.
 */
protected Pair<EdgeGraph, Integer> identifyBarrierSubGoal(Map<Integer, Double> validBarriers,
Region region) {

    List<EdgeGraph> regionEdges = new ArrayList<>();
    if (regionBasedNavigation)
        regionEdges = region.edges;

    EdgeGraph edgeGoal = null;

    // sorted by distance (further away first, as it leads your further away)
    Map<Integer, Double> validSorted =
        (LinkedHashMap<Integer, Double>) Utilities.sortByValue(validBarriers, true);
}

```

```

List<Integer> barrierIDs = new ArrayList<>();
List<EdgeGraph> possibleEdgeGoals = new ArrayList<>();

int waterCounter = 0;
int parkCounter = 0;

// When more than one barrier is identified, the farthest water body barrier is
// chosen; if no water bodies are identified, the agent picks the farthest park
// barrier, if any, or, otherwise, the farthest viable severing barrier.
for (int barrierID : validSorted.keySet()) {
    Barrier barrier = PedSimCity.barriersMap.get(barrierID);
    String type = barrier.type;
    List<EdgeGraph> edgesAlong = new ArrayList<>(barrier.edgesAlong);

    // for region-based, only consider edges in the region
    if (regionBasedNavigation) {
        edgesAlong retainAll regionEdges;
        if (edgesAlong.isEmpty())
            continue;
    }

    Map<EdgeGraph, Double> thisBarrierEdgeGoals = keepValidSubGoals(edgesAlong);
    if (thisBarrierEdgeGoals.isEmpty())
        continue;

    // this is considered a good Edge, sort by distance and takes the closest to the
    // current location.
    Map<EdgeGraph, Double> thisBarrierSubGoalSorted = (LinkedHashMap<EdgeGraph, Double>)
        Utilities
        .sortByValue(thisBarrierEdgeGoals, false);
    EdgeGraph possibleEdgeGoal = thisBarrierSubGoalSorted.keySet().iterator().next();

    switch (type) {
        case "water" -> {
            barrierIDs.add(waterCounter, barrierID);
            possibleEdgeGoals.add(waterCounter, possibleEdgeGoal);
            waterCounter++;
            parkCounter++;
        }
        case "park" -> {
            barrierIDs.add(parkCounter);
            possibleEdgeGoals.add(parkCounter, possibleEdgeGoal);
            parkCounter++;
        }
        default -> {
            barrierIDs.add(barrierID);
            possibleEdgeGoals.add(possibleEdgeGoal);
        }
    }
}

```

```

        }

    }

}

if (possibleEdgeGoals.isEmpty() || possibleEdgeGoals.get(0) == null)
    return null;

edgeGoal = possibleEdgeGoals.get(0);
int barrierID = barrierIDs.get(0);
Pair<EdgeGraph, Integer> pair = new Pair<>(edgeGoal, barrierID);
return pair;
}

/** 
 * Checks requirements for selecting barrier sub-goals from a list of edges
 * along a barrier.
 *
 * @param edgesAlong a list of edges along a barrier;
 * @return a HashMap of EdgeGraph and Double representing eligible barrier
 *         sub-goals and their distances.
 */
private Map<EdgeGraph, Double> keepValidSubGoals(List<EdgeGraph> edgesAlong) {

final Map<EdgeGraph, Double> thisBarrierEdgeGoals = new HashMap<>();

for (EdgeGraph edge : edgesAlong) {
    double distanceToEdge = GraphUtils.euclideanDistance(currentLocation.getCoordinate(),
        edge.getCoordsCentroid());
    double distanceToDestination =
        GraphUtils.getCachedNodesDistance(currentLocation, destinationNode);

    if (distanceToEdge > distanceToDestination) {
        continue;
    }

    boolean containsInSequence = sequence.stream().anyMatch(node -> node.getEdges().contains(edge));
    boolean containsInCurrentLocation = currentLocation.getEdges().contains(edge);

    if (containsInSequence || containsInCurrentLocation)
        continue;

    thisBarrierEdgeGoals.put(edge, distanceToEdge);
}
return thisBarrierEdgeGoals;
}
}

```

```

-----Complexity.java-----
package pedSim.routeChoice;

import java.util.ArrayList;
import java.util.List;

import pedSim.agents.Agent;
import pedSim.cognitiveMap.Region;
import pedSim.engine.PedSimCity;
import pedSim.utilities.StringEnum.LandmarkType;
import sim.graph.GraphUtils;
import sim.graph.NodeGraph;
import sim.util.geo.MasonGeometry;

/**
 * The Complexity class computes wayfinding easiness and legibility complexity
 * for navigation. It includes methods for calculating wayfinding easiness
 * within a space or region based on distance and landmarks.
 */
public class Complexity {

    /**
     * Computes the wayfinding easiness within a space between two nodes on the
     * basis of: a) the ratio between the distance between the passed nodes and a
     * certain maximum distance; b) the legibility of the space, based on the
     * presence of landmarks.
     *
     * @param node           The current node.
     * @param destinationNode The destination node.
     * @param agent          The agent navigating the space.
     * @return The wayfinding easiness value.
     */
    public double wayfindingEasiness(NodeGraph node, NodeGraph destinationNode, Agent agent) {

        final double distanceComplexity = GraphUtils.getCachedNodesDistance(node, destinationNode)
            / Math.max(PedSimCity.roads.MBR.getHeight(), PedSimCity.roads.MBR.getWidth());

        double buildingsComplexity = 1.0;
        List<MasonGeometry> buildings =
            new ArrayList<>(agent.cognitiveMap.getBuildings(node, destinationNode));
        List<MasonGeometry> landmarks = new ArrayList<>(getAgentLandmarks(agent, null));
        if (!buildings.isEmpty()) {
            buildings.retainAll(buildings);
            buildingsComplexity = buildingsComplexity(buildings, landmarks);
        }
        double wayfindingComplexity = (distanceComplexity + buildingsComplexity) / 2.0;
        double easiness = 1.0 - wayfindingComplexity;
    }
}

```

```

        return easiness;
    }

    /**
     * Computes the wayfinding easiness within a region on the basis of: a) the
     * ratio between the distance that would be walked within the region considered
     * and the distance between the origin and the destination; b) the legibility of
     * the region based on the presence of landmarks.
     *
     * @param currentNode      The current node within the region.
     * @param exitGateway      The gateway node at the region exit.
     * @param originNode       The origin node of the whole trip.
     * @param destinationNode The destination node of the whole trip.
     * @param agentProperties The properties of the agent.
     * @return The wayfinding easiness value within the region.
     */
    public double wayfindingEasinessRegion(NodeGraph currentNode, NodeGraph exitGateway,
                                           NodeGraph originNode,
                                           NodeGraph destinationNode, Agent agentProperties) {

        double intraRegionDistance = GraphUtils.getCachedNodesDistance(currentNode, exitGateway);
        double distance = GraphUtils.getCachedNodesDistance(originNode, destinationNode);
        double distanceComplexity = intraRegionDistance / distance;
        if (distanceComplexity < 0.25)
            return 1.0;

        Region region = PedSimCity.regionsMap.get(currentNode.regionID);
        double buildingsComplexity = buildingsRegionComplexity(region, agentProperties);
        double wayfindingComplexity = (distanceComplexity + buildingsComplexity) / 2.0;
        return 1.0 - wayfindingComplexity;
    }

    /**
     * Computes the complexity of a certain area based on the presence of landmarks.
     *
     * @param buildings The set of buildings in the area.
     * @param landmarks The set of landmarks in the area.
     * @return The complexity value of the area.
     */
    public double buildingsComplexity(List<MasonGeometry> buildings, List<MasonGeometry> landmarks) {
        return ((double) buildings.size() - landmarks.size()) / buildings.size();
    }

    /**
     * Computes the building-based complexity of a region based on the type of
     * landmarkness (local or global).
     *

```

```

* @param region The region for which complexity is calculated.
* @param agent The agent navigating the region.
* @return The building-based complexity of the region.
*/
public double buildingsRegionComplexity(Region region, Agent agent) {

    List<MasonGeometry> landmarks;
    List<MasonGeometry> buildings = new ArrayList<>(agent.cognitiveMap.getBuildingsWithinRegion(region));
    if (buildings.isEmpty())
        return 1.0;

    LandmarkType agentLandmarkType = agent.getProperties().landmarkType;
    if (agentLandmarkType.equals(LandmarkType.LOCAL))
        landmarks = new ArrayList<>(agent.cognitiveMap.getRegionLocalLandmarks(region));
    else
        landmarks = new ArrayList<>(agent.cognitiveMap.getRegionGlobalLandmarks(region));

    return buildingsComplexity(buildings, landmarks);
}

/**
 * Retrieves landmarks for the agent based at the city or region level (if
 * provided).
 *
 * @param agent The agent for which landmarks are retrieved.
 * @param region The region for which landmarks are retrieved (can be null).
 * @return A list of landmarks for the agent.
 */
public List<MasonGeometry> getAgentLandmarks(Agent agent, Region region) {

    LandmarkType agentLandmarkType = agent.getProperties().landmarkType;
    if (region != null) {
        if (agentLandmarkType.equals(LandmarkType.LOCAL))
            return agent.cognitiveMap.getRegionLocalLandmarks(region);
        else
            return agent.cognitiveMap.getRegionGlobalLandmarks(region);
    } else {
        if (agentLandmarkType.equals(LandmarkType.LOCAL))
            return agent.cognitiveMap.getLocalLandmarks().getGeometries();
        else
            return agent.cognitiveMap.getGlobalLandmarks().getGeometries();
    }
}

-----
GlobalLandmarksPathFinder.java-----
/**
```

```
* Series of functions that support the generation of routes calling different methods.  
*  
*/  
  
package pedSim.routeChoice;  
  
import java.util.ArrayList;  
import java.util.List;  
  
import pedSim.agents.Agent;  
import pedSim.dijkstra.DijkstraGlobalLandmarks;  
import sim.graph.NodeGraph;  
  
public class GlobalLandmarksPathFinder extends PathFinder {  
  
    /**  
     * Formulates a route based on global landmarkness maximisation between the  
     * origin and destination nodes only.  
     *  
     * @return The computed route.  
     */  
    public Route globalLandmarksPath(NodeGraph originNode, NodeGraph destinationNode, Agent agent) {  
  
        this.agent = agent;  
        this.originNode = originNode;  
        this.destinationNode = destinationNode;  
        DijkstraGlobalLandmarks pathfinder = new DijkstraGlobalLandmarks();  
        partialSequence = pathfinder.dijkstraAlgorithm(originNode, destinationNode, destinationNode,  
            directedEdgesToAvoid, agent);  
        route.directedEdgesSequence = partialSequence;  
        route.routeSequences();  
        return route;  
    }  
  
    /**  
     * Formulates a route based on global landmarkness maximisation through a  
     * sequence of intermediate nodes [originNode, ..., destinationNode]. It allows  
     * combining global landmarkness maximisation with a sequence of nodes resulting  
     * for example from the region-based navigation.  
     *  
     * @param sequenceNodes A list of nodes representing the sequence to follow.  
     * @return The computed route.  
     */  
    public Route globalLandmarksPathSequence(List<NodeGraph> sequenceNodes, Agent agent) {  
  
        this.agent = agent;  
        this.sequenceNodes = new ArrayList<>(sequenceNodes);  
    }  
}
```

```

    // originNode
    originNode = sequenceNodes.get(0);
    tmpOrigin = originNode;
    destinationNode = sequenceNodes.get(sequenceNodes.size() - 1);
    this.sequenceNodes.remove(0);

    for (NodeGraph tmpDestination : this.sequenceNodes) {
        moveOn = false;
        // check if this tmpDestination has been traversed already
        if (route.nodesFromEdgesSequence(completeSequence).contains(tmpDestination)) {
            controlPath(tmpDestination);
            tmpOrigin = tmpDestination;
            continue;
        }

        // check if edge in between
        if (haveEdgesBetween())
            continue;

        final DijkstraGlobalLandmarks pathfinder = new DijkstraGlobalLandmarks();
        partialSequence = pathfinder.dijkstraAlgorithm(tmpOrigin, tmpDestination, destinationNode,
            null, agent);

        while (partialSequence.isEmpty() && !moveOn)
            backtracking(tmpDestination);
        tmpOrigin = tmpDestination;
        if (moveOn)
            continue;

        checkEdgesSequence(tmpOrigin);
        completeSequence.addAll(partialSequence);
    }

    route.directedEdgesSequence = completeSequence;
    route.routeSequences();
    return route;
}

-----LandmarkNavigation.java-----
package pedSim.routeChoice;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;

```

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

import org.locationtech.jts.planargraph.DirectedEdge;

import pedSim.agents.Agent;
import pedSim.cognitiveMap.LandmarkIntegration;
import pedSim.cognitiveMap.Region;
import pedSim.engine.Parameters;
import pedSim.engine.PedSimCity;
import sim.field.geo.VectorLayer;
import sim.graph.Building;
import sim.graph.Graph;
import sim.graph.GraphUtils;
import sim.graph.NodeGraph;

/**
 * Series of functions that support landmark-based navigation, landmarkness
 * computation, identification of on-route marks and wayfinding easiness of a
 * certain space.
 */
public class LandmarkNavigation {

    NodeGraph originNode;
    NodeGraph destinationNode;
    private List<NodeGraph> sequence = new ArrayList<>();
    private List<NodeGraph> inRegionSequence = new ArrayList<>();
    private Complexity complexity = new Complexity();
    private Map<NodeGraph, Double> salientNodes = new HashMap<NodeGraph, Double>();
    private Agent agent;
    private NodeGraph currentNode;
    private Graph agentNetwork;

    /**
     * Initialises a new instance of the LandmarkNavigation class with the specified
     * origin node, destination node, and agent.
     *
     * @param originNode      The starting node for navigation.
     * @param destinationNode The target destination node.
     * @param agent           The agent associated with the navigation.
     */
    public LandmarkNavigation(NodeGraph originNode, NodeGraph destinationNode, Agent agent) {
        this.originNode = originNode;
        this.destinationNode = destinationNode;
        this.agent = agent;
        this.agentNetwork = agent.getCognitiveMap().getKnownNetwork();
```

```
}

/**
 * Generates a sequence of intermediate nodes (on-route marks) between the
 * origin and destination nodes on the basis of local landmarkness.
 *
 * @return An ArrayList of on-route marks, including the origin and destination
 *         nodes.
 */
public List<NodeGraph> onRouteMarks() {

    sequence = new ArrayList<>();
    findSalientJunctions(originNode);

    if (salientNodes.isEmpty())
        return sequence;

    // compute wayfinding easiness and the resulting research space
    double wayfindingEasiness = complexity.wayfindingEasiness(originNode, destinationNode, agent);
    double searchDistance = GraphUtils.nodesDistance(originNode, destinationNode) * wayfindingEasiness;
    currentNode = originNode;

    // while the wayfindingEasiness is lower than the threshold the agent looks for
    // intermediate-points.
    while (wayfindingEasiness < Parameters.wayfindingEasinessThreshold) {
        NodeGraph bestNode = findOnRouteMark(salientNodes, searchDistance);
        if (bestNode == null || bestNode.equals(currentNode))
            break;
        sequence.add(bestNode);
        findSalientJunctions(bestNode);
        if (salientNodes.isEmpty())
            return sequence;
        wayfindingEasiness = complexity.wayfindingEasiness(bestNode, destinationNode, agent);
        searchDistance = GraphUtils.nodesDistance(bestNode, destinationNode) * wayfindingEasiness;
        currentNode = bestNode;
        bestNode = null;
    }
    sequence.add(0, originNode);
    sequence.add(destinationNode);
    return sequence;
}

/**
 * Identifies salient junctions within the network space between a given node
 * and the destination node. This method finds salient nodes based on their
 * salience within the specified network space.
 *
```

```

* @param node The node from which to identify salient junctions, in the space
*             with the destination node.
*/
public void findSalientJunctions(NodeGraph node) {

    double percentile = Parameters.salientNodesPercentile;
    salientNodes = new HashMap<NodeGraph, Double>(
        agentNetwork.getSalientNodesWithinSpace(node, destinationNode, percentile));

    // If no salient junctions are found, the tolerance increases till the 0.50
    // percentile;
    // if still no salient junctions are found, the agent continues without
    // landmarks
    while (salientNodes.isEmpty()) {
        percentile -= 0.05;
        if (percentile < 0.50) {
            sequence.add(0, originNode);
            sequence.add(destinationNode);
            break;
        }
        salientNodes = new HashMap<NodeGraph, Double>(
            agentNetwork.getSalientNodesWithinSpace(node, destinationNode, percentile));
    }
}

/**
 * Finds the most salient on-route mark between the current node and the
 * destination node, amongst the salient nodes (junctions).
 *
 * @param currentNode The current node in the navigation.
 * @param salientNodes A map of salient junctions and their centrality scores
 *                     along the route.
 * @param searchDistance The search distance limit from the currentNode for
 *                       evaluating potential nodes.
 * @return The selected node that serves as an on-route mark, or null if none is
 *         found.
 */
private NodeGraph findOnRouteMark(Map<NodeGraph, Double> salientNodes, Double searchDistance) {

    List<NodeGraph> junctions = new ArrayList<>(salientNodes.keySet());
    List<Double> centralities = new ArrayList<>(salientNodes.values());
    double maxCentrality = Collections.max(centralities);
    double minCentrality = Collections.min(centralities);

    List<NodeGraph> sortedJunctions = junctions.stream()
        .filter(candidateNode -> checkCriteria(candidateNode, searchDistance))
        .sorted(Comparator

```

```

        .comparingDouble(candidateNode ->
            calculateScore(candidateNode, minCentrality, maxCentrality)))
        .collect(Collectors.toList());

    return sortedJunctions.isEmpty() ? null : sortedJunctions.get(sortedJunctions.size() - 1);
}

/**
 * Checks the criteria for selecting a candidate node based on various
 * conditions.
 *
 * @param candidateNode The node being evaluated.
 * @param searchDistance The search distance limit for evaluating potential
 *                       nodes.
 * @return {@code true} if the candidate node meets all criteria, {@code false}
 *         otherwise.
 */
private boolean checkCriteria(NodeGraph candidateNode, double searchDistance) {

    return !sequence.contains(candidateNode) && !candidateNode.equals(originNode)
        && agentNetwork.getEdgeBetween(candidateNode, currentNode) == null
        && agentNetwork.getEdgeBetween(candidateNode, originNode) == null
        && GraphUtils.getCachedNodesDistance(currentNode, candidateNode) <= searchDistance;
}

/**
 * Calculates the score for a candidate node based on centrality and distance
 * gain metrics.
 *
 * @param candidateNode The node being evaluated.
 * @param minCentrality The minimum centrality value in the network.
 * @param maxCentrality The maximum centrality value in the network.
 * @return The calculated score for the candidate node, considering centrality
 *         and distance gain.
 */
private double calculateScore(NodeGraph candidateNode, double minCentrality, double maxCentrality) {

    double score = agent.getProperties().usingLocalLandmarks ? localLandmarkness(candidateNode)
        : (candidateNode.getCentrality() - minCentrality) / (maxCentrality - minCentrality);
    double currentDistance = GraphUtils.getCachedNodesDistance(currentNode, destinationNode);
    double distanceGain =
        (currentDistance - GraphUtils.getCachedNodesDistance(candidateNode, destinationNode))
        / currentDistance;
    return score * 0.60 + distanceGain * 0.40;
}

```

```

/**
 * Generates a sequence of intermediate nodes (on-route marks) between the
 * origin and destination nodes on the basis of local landmarkness while passing
 * through region gateways (sequenceGateways).
 *
 * @param sequenceNodes An ArrayList of gateways (nodes at the boundary between
 *                      regions) that need to be traversed on the route.
 * @return An ArrayList of region-based on-route marks, including the origin and
 *         destination nodes.
 */
public List<NodeGraph> regionOnRouteMarks(List<NodeGraph> sequenceNodes) {

    sequence = new ArrayList<>();
    currentNode = originNode;

    for (NodeGraph exitGateway : sequenceNodes) {
        if (exitGateway.equals(originNode) || currentNode.equals(destinationNode))
            continue;
        sequence.add(currentNode);
        if (currentNode.regionID != exitGateway.regionID) {
            currentNode = exitGateway;
            continue;
        }
        inRegionSequence = new ArrayList<>();
        // works also for nodeBasedNavigation only:
        inRegionSequence = onRouteMarksInRegion(exitGateway);
        sequence.addAll(inRegionSequence);
        currentNode = exitGateway;
    }
    sequence.add(destinationNode);
    return sequence;
}

/**
 * Finds within-region on-route marks from the current node to the passed exit
 * gateway node, within the current node's region.
 *
 * @param exitGateway The exit gateway node that marks the boundary of the
 *                    region.
 * @return An ArrayList of in-region on-route marks within the same region,
 *         including the current node and exit gateway.
 */
public List<NodeGraph> onRouteMarksInRegion(NodeGraph exitGateway) {

    Region region = PedSimCity.regionsMap.get(currentNode.regionID);
    findRegionSalientJunctions(region);
    if (salientNodes.isEmpty())

```

```

        return inRegionSequence;
    // compute wayfinding complexity and the resulting easiness
    double wayfindingEasiness = complexity.wayfindingEasinessRegion(currentNode, exitGateway,
        originNode,
        destinationNode, agent);
    double searchDistance =
        GraphUtils.getCachedNodesDistance(currentNode, exitGateway) * wayfindingEasiness;

    // while the wayfindingEasiness is lower than the threshold the agent looks for
    // intermediate-points.
    while (wayfindingEasiness < Parameters.wayfindingEasinessThresholdRegions) {
        NodeGraph bestNode = findOnRouteMarkRegion(exitGateway, salientNodes, searchDistance);

        if (bestNode == null || bestNode.equals(exitGateway) || bestNode.equals(destinationNode))
            break;
        inRegionSequence.add(bestNode);
        findRegionSalientJunctions(region);
        if (salientNodes.isEmpty())
            return inRegionSequence;

        wayfindingEasiness =
            complexity.wayfindingEasinessRegion(bestNode, originNode, destinationNode, exitGateway,
                agent);
        searchDistance = GraphUtils.getCachedNodesDistance(bestNode, exitGateway) * wayfindingEasiness;
        currentNode = bestNode;
        bestNode = null;
    }
    return inRegionSequence;
}

/**
 * Identifies salient junctions within a specific region's graph based on their
 * centrality in the graph.
 *
 * @param region The region for which to identify salient junctions.
 */
private void findRegionSalientJunctions(Region region) {

    double percentile = Parameters.salientNodesPercentile;
    salientNodes = new HashMap<NodeGraph, Double>(region.primalGraph.getSubGraphSalientNodes(percentile));

    // If no salient junctions are found, the tolerance increases till the 0.50
    // percentile;
    // still no salient junctions are found, the agent continues without landmarks
    while (salientNodes.isEmpty()) {
        percentile -= 0.05;
        if (percentile < 0.50)

```

```

        break;
    salientNodes =
        new HashMap<NodeGraph, Double>(region.primalGraph.getSubGraphSalientNodes(percentile));
}
}

/***
 * Finds the most salient on-route mark between the current node and the exit
 * gateway, amongst the salientNodes within a specific region.
 *
 * @param currentNode      The current node in the region.
 * @param exitGateway      The exit gateway node from the region.
 * @param salientNodes     A map of salient junctions and their centrality scores
 *                        within the region.
 * @param searchDistance   The search distance limit for evaluating potential
 *                        nodes.
 * @return The selected node that serves as an on-route mark, or null if none is
 *         found.
 */
private NodeGraph findOnRouteMarkRegion(NodeGraph exitGateway, Map<NodeGraph, Double> salientNodes,
                                         Double searchDistance) {

    List<NodeGraph> junctions = new ArrayList<>(salientNodes.keySet());
    List<Double> centralities = new ArrayList<>(salientNodes.values());
    double currentDistance = GraphUtils.getCachedNodesDistance(currentNode, exitGateway);
    double maxCentrality = Collections.max(centralities);
    double minCentrality = Collections.min(centralities);

    // Optimised sorting using Comparator and Collections.sort
    List<NodeGraph> sortedJunctions =
        junctions.stream().filter(candidateNode -> checkCriteria(candidateNode, exitGateway,
                                                               searchDistance, currentDistance))
        .sorted(Comparator.comparingDouble(candidateNode -> calculateScore(candidateNode, exitGateway,
                                                                           currentDistance, minCentrality, maxCentrality)))
        .collect(Collectors.toList());

    return sortedJunctions.isEmpty() ? null : sortedJunctions.get(sortedJunctions.size() - 1);
}

/***
 * Checks the criteria for selecting a candidate node based on various
 * conditions.
 *
 * @param candidateNode    The node being evaluated.
 * @param exitGateway      The exit gateway node within the region.
 * @param searchDistance   The search distance limit for evaluating potential
 *                        nodes.
 */

```

```

* @param currentDistance The current distance to the exit gateway.
* @return {@code true} if the candidate node meets all criteria, {@code false}
*         otherwise.
*/
private boolean checkCriteria(NodeGraph candidateNode, NodeGraph exitGateway, double searchDistance,
    double currentDistance) {

    return !inRegionSequence.contains(candidateNode) && !candidateNode.equals(currentNode)
        && agentNetwork.getEdgeBetween(candidateNode, currentNode) == null
        && GraphUtils.getCachedNodesDistance(currentNode, candidateNode) <= searchDistance
        && GraphUtils.getCachedNodesDistance(candidateNode, exitGateway) <= currentDistance
        && !sequence.contains(candidateNode);
}

/**
 * Calculates the score for a candidate node based on centrality and gain
 * metrics within a region.
 *
 * @param candidateNode The candidate node for which the score is calculated.
 * @param exitGateway The exit gateway node within the region.
 * @param currentDistance The current distance to the exit gateway.
 * @param minCentrality The minimum centrality value in the network.
 * @param maxCentrality The maximum centrality value in the network.
 * @return The calculated score for the candidate node, considering centrality
 *         and gain metrics.
 */
private double calculateScore(NodeGraph candidateNode, NodeGraph exitGateway, double currentDistance,
    double minCentrality, double maxCentrality) {

    double score;
    if (agent.getProperties().usingLocalLandmarks)
        score = localLandmarkness(candidateNode);
    else
        score = (candidateNode.getCentrality() - minCentrality) / (maxCentrality - minCentrality);
    double gain =
        (currentDistance - GraphUtils.nodesDistance(candidateNode, exitGateway)) / currentDistance;
    return score * 0.50 + gain * 0.50;
}

/**
 * Computes the local landmarkness score for a given node based on local
 * landmarks in its proximity and the landmarks known by the agent.
 *
 * @param node The node for which to calculate local landmarkness.
 * @return The computed local landmarkness score for the node.
 */
private double localLandmarkness(NodeGraph candidateNode) {

```

```

List<Building> nodeLocalLandmarks = new ArrayList<>(candidateNode.adjacentBuildings);
VectorLayer agentLocalLandmarks = agent.getCognitiveMap().getLocalLandmarks();
List<Integer> agentLandmarksIDs = agentLocalLandmarks.getIDs();

for (Building landmark : candidateNode.adjacentBuildings) {
    if (!agentLandmarksIDs.contains(landmark.buildingID))
        nodeLocalLandmarks.remove(landmark);
}

if (nodeLocalLandmarks.isEmpty())
    return 0.0;
List<Double> localScores = new ArrayList<>();
for (Building landmark : nodeLocalLandmarks)
    localScores.add(landmark.attributes.get("localLandmarkness").getDouble());
return Collections.max(localScores);
}

/**
 * Computes the global landmarkness score for a target node based on the global
 * landmarks in its proximity and their relationship with the destination node.
 *
 * @param targetNode      The target node being examined.
 * @param destinationNode The final destination node.
 * @return The computed global landmarkness score for the target node.
 */
public static double globalLandmarknessNode(NodeGraph targetNode, NodeGraph destinationNode) {

    // get the distant landmarks
    List<Building> distantLandmarks = new ArrayList<>(targetNode.visibleBuildings3d);

    if (distantLandmarks.isEmpty())
        return 0.0;

    // get the anchors of the destination
    List<Building> anchors = new ArrayList<>(LandmarkIntegration.getAnchors(destinationNode).getArray());
    double nodeGlobalScore = 0.0;
    double targetDistance = GraphUtils.getCachedNodesDistance(targetNode, destinationNode);
    for (Building landmark : distantLandmarks) {
        if (!anchors.isEmpty() && !anchors.contains(landmark))
            continue;

        double score = landmark.attributes.get("globalLandmarkness").getDouble();
        ArrayList<Double> distances =
            new ArrayList<>(LandmarkIntegration.getDistances(destinationNode).getArray());
        double distanceLandmark = distances.get(anchors.indexOf(landmark));
        double distanceWeight = Math.min(targetDistance / distanceLandmark, 1.0);
        score *= distanceWeight;
    }
}

```

```

        if (anchors.isEmpty())
            score *= 0.90;
        nodeGlobalScore = Math.max(nodeGlobalScore, score);
    }
    return nodeGlobalScore;
}

/**
 * Computes the global landmarkness for a target dual node based on the global
 * landmarks in its proximity and their relationship with the destination node.
 *
 * @param centroid      The current centroid node.
 * @param targetCentroid The target centroid node.
 * @param destinationNode The destination node.
 * @return The computed global landmarkness score for the dual node.
 */
public static double globalLandmarknessDualNode(NodeGraph centroid, NodeGraph targetCentroid,
                                                NodeGraph destinationNode) {

    // current real segment: identifying the node
    DirectedEdge streetSegment = targetCentroid.getPrimalEdge().getDirEdge(0);
    NodeGraph targetNode = (NodeGraph) streetSegment.getToNode(); // targetNode
    if (GraphUtils.getPrimalJunction(centroid, targetCentroid).equals(targetNode))
        targetNode = (NodeGraph) streetSegment.getFromNode();

    return globalLandmarknessNode(targetNode, destinationNode);
}
}

-----PathFinder.java-----
package pedSim.routeChoice;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.locationtech.jts.planargraph.DirectedEdge;

import pedSim.agents.Agent;
import pedSim.dijkstra.DijkstraAngularChange;
import pedSim.dijkstra.DijkstraRoadDistance;
import sim.graph.EdgeGraph;
import sim.graph.Graph;
import sim.graph.GraphUtils;

```

```

import sim.graph.NodeGraph;

/**
 * The 'PathFinder' class provides common functionality for computing navigation
 * paths using various algorithms and graph representations.
 */
public class PathFinder {

    Agent agent;
    Route route = new Route();
    protected Graph agentNetwork;
    NodeGraph originNode, destinationNode;
    NodeGraph tmpOrigin, tmpDestination;
    NodeGraph previousJunction = null;

    List<NodeGraph> sequenceNodes = new ArrayList<>();
    // need order here, that's why it's not hashset
    List<NodeGraph> centroidsToAvoid = new ArrayList<>();
    Set<DirectedEdge> directedEdgesToAvoid = new HashSet<>();

    List<DirectedEdge> completeSequence = new ArrayList<>();
    List<DirectedEdge> partialSequence = new ArrayList<>();

    protected boolean regionBased = false;
    boolean moveOn = false;

    /**
     * Performs backtracking to compute a path in a primal graph from the current
     * temporary origin node to the given temporary destination node. If the
     * temporary origin node is the same as the original origin node, it attempts to
     * skip the temporary destination. Otherwise, it updates the temporary origin
     * node, checks for the existence of a direct segment between the new origin and
     * the destination, and if not found, computes a path from the new origin to the
     * destination while avoiding specified segments.
     *
     * @param tmpDestination The temporary destination node.
     */
    protected void backtracking(NodeGraph tmpDestination) {

        if (tmpOrigin.equals(originNode)) {
            // try skipping this tmpDestination
            moveOn = true;
            return;
        }
        // determine new tmpOrigin
        updateTmpOrigin();
    }
}

```

```

// check if there's a segment between the new tmpOrigin and the destination
final DirectedEdge edge = agentNetwork.getDirectedEdgeBetween(tmpOrigin, tmpDestination);
if (edge != null) {
    if (!completeSequence.contains(edge))
        completeSequence.add(edge);
    moveOn = true; // No need to backtrack anymore
    return;
}

// If not, try to compute the path from the new tmpOrigin
final DijkstraRoadDistance pathFinder = new DijkstraRoadDistance();
directedEdgesToAvoid = new HashSet<>(completeSequence);
partialSequence =
    pathFinder.dijkstraAlgorithm(tmpOrigin, tmpDestination, destinationNode, directedEdgesToAvoid,
        agent);
}

/**
 * Updates the temporary origin node based on the current state of the path. If
 * there are fewer than two segments in the complete sequence, it clears the
 * sequence and sets the temporary origin to the original origin node.
 * Otherwise, it removes the last problematic segment from the complete sequence
 * and updates the temporary origin accordingly.
 */
private void updateTmpOrigin() {
    if (completeSequence.size() < 2) {
        completeSequence.clear();
        tmpOrigin = originNode;
    } else {
        // remove the last problematic segment
        completeSequence.remove(completeSequence.size() - 1);
        tmpOrigin = (NodeGraph) completeSequence.get(completeSequence.size() - 1).getToNode();
    }
}

/**
 * Performs backtracking in the context of dual graph-based pathfinding (angular
 * change). When the agent gets stuck due to the "centroidsToAvoid" set, this
 * method iterates back across nodes and retries to compute the path towards the
 * given tmpDestinationNode.
 */
protected void dualBacktracking() {
    // new tmpOrigin
    try {
        tmpOrigin = (NodeGraph) completeSequence.get(completeSequence.size() - 1).getFromNode();
    } catch (final java.lang.ArrayIndexOutOfBoundsException e) {
        partialSequence.clear();
    }
}

```

```

        return;
    }

    // remove last one which did not work!
    completeSequence.remove(completeSequence.size() - 1);
    centroidsToAvoid.remove(centroidsToAvoid.size() - 1);
    // take new previous junction
    previousJunction = route.previousJunction(completeSequence);
    // check if there's a segment between the new tmpOrigin and the destination
    final DirectedEdge edge = agentNetwork.getDirectedEdgeBetween(tmpOrigin, tmpDestination);

    if (edge != null) {
        if (!completeSequence.contains(edge))
            completeSequence.add(edge);
        moveOn = true; // no need to backtracking anymore
        return;
    }

    List<NodeGraph> dualNodesOrigin = getDualNodes(tmpOrigin, previousJunction);
    List<NodeGraph> dualNodesDestination = getDualNodes(tmpDestination, previousJunction);
    for (final NodeGraph tmpDualOrigin : dualNodesOrigin) {
        for (final NodeGraph tmpDualDestination : dualNodesDestination) {
            final DijkstraAngularChange pathfinder = new DijkstraAngularChange();
            Set<NodeGraph> centroidsToAvoidSet = new HashSet<>(centroidsToAvoid);
            partialSequence = pathfinder.dijkstraAlgorithm(tmpDualOrigin,
                tmpDualDestination, destinationNode, centroidsToAvoidSet, tmpOrigin, agent);
            if (!partialSequence.isEmpty())
                break;
        }
        if (!partialSequence.isEmpty())
            break;
    }
}

/**
 * Retrieves a list of dual nodes connected to the given node.
 *
 * @param node The examined primal node.
 * @param previousJunction The previous junction node used for deriving the
 *                         direction.
 * @return A list of dual nodes connected to the given primal node.
 */
protected ArrayList<NodeGraph> getDualNodes(NodeGraph node, NodeGraph previousJunction) {
    Set<NodeGraph> dualNodesSet = node.getDualNodes(tmpOrigin, tmpDestination, regionBased,
        previousJunction)
        .keySet();
    return new ArrayList<NodeGraph>(dualNodesSet);
}

```

```

    }

    /**
     * Controls and adjusts the sequence of DirectedEdges to ensure that the
     * destinationNode has not been traversed already. If the destinationNode has
     * been traversed, it removes any unnecessary edges and ensures that the path is
     * correctly ordered.
     *
     * @param destinationNode The examined primal destination node.
     */
    protected void controlPath(NodeGraph destinationNode) {
        for (final DirectedEdge directedEdge : completeSequence)
            if (directedEdge.getToNode().equals(destinationNode)) {
                int lastIndex = completeSequence.indexOf(directedEdge);
                completeSequence = new ArrayList<>(completeSequence.subList(0, lastIndex + 1));
                if (route.previousJunction(completeSequence).equals(destinationNode))
                    completeSequence.remove(completeSequence.size() - 1);
            }
        return;
    }

    /**
     * Cleans and adjusts the sequence of DirectedEdges in the dual graph-based
     * path. It checks if the path is one edge ahead and removes the last edge if
     * necessary. It also checks for the presence of an unnecessary edge at the
     * beginning of the path and removes it. This method ensures that the dual
     * graph-based path is correctly ordered and free of unnecessary edges.
     *
     * @param tmpOrigin      The examined primal origin node.
     * @param tmpDestination The examined primal destination node.
     */
    protected void cleanDualPath(NodeGraph tmpOrigin, NodeGraph tmpDestination) {
        // check if the path is one edge ahead
        final NodeGraph firstDualNode = ((EdgeGraph) partialSequence.get(0).getEdge()).getDualNode();
        final NodeGraph secondDualNode = ((EdgeGraph) partialSequence.get(1).getEdge()).getDualNode();

        if (route.previousJunction(partialSequence).equals(tmpDestination))
            partialSequence.remove(partialSequence.size() - 1);
        // check presence of a unnecessary edge at the beginning of the path
        if (GraphUtils.getPrimalJunction(firstDualNode, secondDualNode).equals(tmpOrigin))
            partialSequence.remove(0);
        checkEdgesSequence(tmpOrigin);
    }

    /**
     * Reorders the sequence of DirectedEdges based on their fromNode and toNode,
     * ensuring that they follow a consistent order. This method is used to correct

```

```

* the sequence of DirectedEdges when performing region-based navigation to
* ensure a smooth path.
*
* @param tmpOrigin The examined primal origin node.
*/
protected void checkEdgesSequence(NodeGraph tmpOrigin) {
    NodeGraph previousNode = tmpOrigin;

    final ArrayList<DirectedEdge> copyPartial = new ArrayList<>(partialSequence);
    for (final DirectedEdge edge : copyPartial) {
        NodeGraph nextNode = (NodeGraph) edge.getToNode();
        // need to swap
        if (nextNode.equals(previousNode)) {
            nextNode = (NodeGraph) edge.getFromNode();
            DirectedEdge correctEdge = agentNetwork.getDirectedEdgeBetween(previousNode, nextNode);
            partialSequence.set(partialSequence.indexOf(edge), correctEdge);
        }
        previousNode = nextNode;
    }
}

/**
 * Checks if there are directed edges between two nodes.
 *
 * @return true if edges exist, false otherwise.
 */
protected boolean haveEdgesBetween() {
    // check if edge in between
    DirectedEdge edge = agentNetwork.getDirectedEdgeBetween(tmpOrigin, tmpDestination);

    if (edge == null)
        return false;
    else {
        if (!completeSequence.contains(edge))
            completeSequence.add(edge);
        tmpOrigin = tmpDestination;
        return true;
    }
}
}

-----RoadDistancePathFinder.java-----
package pedSim.routeChoice;

import java.util.ArrayList;
import java.util.HashSet;

```

```

import java.util.List;

import pedSim.agents.Agent;
import pedSim.dijkstra.DijkstraRoadDistance;
import sim.graph.NodeGraph;

/**
 * A pathfinder for road-distance based route calculations. This class extends
 * the functionality of the base class Pathfinder.
 */
public class RoadDistancePathFinder extends Pathfinder {

    /**
     * Formulates a route based on road distance between the given origin and
     * destination nodes using the provided agent properties.
     *
     * @param originNode      the origin node;
     * @param destinationNode the destination node;
     * @param agent            The agent for which the route is computed.
     * @return a {@code Route} object representing the road-distance shortest path.
     */
    public Route roadDistance(NodeGraph originNode, NodeGraph destinationNode, Agent agent) {

        this.agent = agent;
        agentNetwork = agent.getCognitiveMap().getKnownNetwork();
        final DijkstraRoadDistance pathfinder = new DijkstraRoadDistance();
        partialSequence = pathfinder.dijkstraAlgorithm(originNode, destinationNode, destinationNode,
                directedEdgesToAvoid, agent);
        route.directedEdgesSequence = partialSequence;
        route.routeSequences();
        return route;
    }

    /**
     * Formulates a route based on road distance minimisation through a sequence of
     * intermediate nodes [originNode, ..., destinationNode] using the provided
     * agent properties. It allows combining the road-distance local minimisation
     * heuristic with navigational strategies based on the usage of urban elements.
     *
     * @param sequenceNodes sequence of intermediate nodes (e.g. on-route marks,
     *                      gateways) including the origin and the destination
     *                      nodes;
     * @param agent          The agent for which the route is computed.
     * @return a 'Route' object representing the road-distance shortest path for the
     *         sequence of nodes.
     */
    public Route roadDistanceSequence(List<NodeGraph> sequenceNodes, Agent agent) {

```

```

this.agent = agent;
agentNetwork = agent.getCognitiveMap().getKnownNetwork();
this.sequenceNodes = new ArrayList<>(sequenceNodes);

// originNode
originNode = this.sequenceNodes.get(0);
tmpOrigin = originNode;
destinationNode = sequenceNodes.get(sequenceNodes.size() - 1);
this.sequenceNodes.remove(0);

for (final NodeGraph currentNode : this.sequenceNodes) {
    moveOn = false;
    tmpDestination = currentNode;

    // check if this tmpDestination has been traversed already
    if (route.nodesFromEdgesSequence(completeSequence).contains(tmpDestination)) {
        controlPath(tmpDestination);
        tmpOrigin = tmpDestination;
        continue;
    }

    if (haveEdgesBetween())
        continue;

    directedEdgesToAvoid = new HashSet<>(completeSequence);
    DijkstraRoadDistance pathfinder = new DijkstraRoadDistance();
    partialSequence = pathfinder.dijkstraAlgorithm(tmpOrigin, tmpDestination, destinationNode,
                                                   directedEdgesToAvoid, agent);
    while (partialSequence.isEmpty() && !moveOn)
        backtracking(tmpDestination);

    if (moveOn) {
        if (tmpOrigin == originNode)
            continue;
        tmpOrigin = tmpDestination;
        continue;
    }
    checkEdgesSequence(tmpOrigin);
    completeSequence.addAll(partialSequence);
    tmpOrigin = tmpDestination;
}

route.directedEdgesSequence = completeSequence;
route.routeSequences();
return route;
}
}

```

```

-----Route.java-----
package pedSim.routeChoice;

import java.util.ArrayList;
import java.util.List;

import org.locationtech.jts.planargraph.DirectedEdge;

import sim.graph.EdgeGraph;
import sim.graph.GraphUtils;
import sim.graph.NodeGraph;

/**
 * A class for storing the sequence of GeomPlanarGraphDirectedEdge in a path and
 * the sequence of NodeWrappers. It supports shortest-path algorithms and
 * provides some utilities.
 */
public class Route {

    // always primal
    public NodeGraph originNode;
    public NodeGraph destinationNode;
    public List<DirectedEdge> directedEdgesSequence = new ArrayList<>();
    private List<EdgeGraph> edgesSequence = new ArrayList<>();
    private List<NodeGraph> nodesSequence = new ArrayList<>();

    /**
     * Identifies the previous junction traversed in a dual graph path to avoid
     * traversing an unnecessary segment in the primal graph.
     *
     * @param sequenceDirectedEdges A sequence of GeomPlanarGraphDirectedEdge
     *                             representing the path.
     * @return The previous junction node.
     */
    public NodeGraph previousJunction(List<DirectedEdge> sequenceDirectedEdges) {
        // from global graph
        if (sequenceDirectedEdges.size() == 1)
            return (NodeGraph) sequenceDirectedEdges.get(0).getFromNode();

        NodeGraph lastCentroid =
            ((EdgeGraph) sequenceDirectedEdges.get(sequenceDirectedEdges.size() - 1).getEdge())
                .getDualNode();
        NodeGraph otherCentroid =
            ((EdgeGraph) sequenceDirectedEdges.get(sequenceDirectedEdges.size() - 2).getEdge())
                .getDualNode();
}

```

```

        return GraphUtils.getPrimalJunction(lastCentroid, otherCentroid);
    }

    /**
     * Returns all the primal nodes traversed in a path.
     *
     * @param directedEdgesSequence A sequence of GeomPlanarGraphDirectedEdge
     *                             representing the path.
     * @return A list of primal nodes.
     */
    public List<NodeGraph> nodesFromEdgesSequence(List<DirectedEdge> directedEdgesSequence) {

        List<NodeGraph> nodes = new ArrayList<>();
        if (directedEdgesSequence.isEmpty())
            return nodes;

        for (DirectedEdge directedEdge : directedEdgesSequence)
            nodes.add(((EdgeGraph) directedEdge.getEdge()).getFromNode());

        EdgeGraph lastEdge = (EdgeGraph)
            directedEdgesSequence.get(directedEdgesSequence.size() - 1).getEdge();
        nodes.add(lastEdge.getToNode());
        return nodes;
    }

    /**
     * Returns all the centroids (nodes in the dual graph) traversed in a path.
     *
     * @param sequenceDirectedEdges A sequence of GeomPlanarGraphDirectedEdge
     *                             representing the path.
     * @return A list of centroids (dual nodes).
     */
    public List<NodeGraph> centroidsFromEdgesSequence(List<DirectedEdge> sequenceDirectedEdges) {
        List<NodeGraph> centroids = new ArrayList<>();
        for (DirectedEdge planarDirectedEdge : sequenceDirectedEdges)
            centroids.add(((EdgeGraph) planarDirectedEdge.getEdge()).getDualNode());
        return centroids;
    }

    /**
     * Constructs the sequences of nodes and edges from the directed edges sequence.
     * This method populates the {@code nodesSequence} and {@code edgesSequence}
     * lists.
     */
    public void routeSequences() {
        nodesSequence = nodesFromEdgesSequence(directedEdgesSequence);

```

```
        for (DirectedEdge directedEdge : directedEdgesSequence)
            edgesSequence.add((EdgeGraph) directedEdge.getEdge());
        originNode = nodesSequence.get(0);
        destinationNode = nodesSequence.get(nodesSequence.size() - 1);
    }
}

-----RoutePlanner.java-----
package pedSim.routeChoice;

import java.util.ArrayList;
import java.util.List;

import pedSim.agents.Agent;
import pedSim.agents.AgentProperties;
import pedSim.engine.Parameters;
import sim.graph.GraphUtils;
import sim.graph.NodeGraph;

/**
 * The 'RoutePlanner' class is responsible for calculating a route for an agent
 * within a pedestrian simulation. It considers the agent's route choice
 * properties and strategies to determine the optimal path from an origin node
 * to a destination node.
 */
public class RoutePlanner {

    private NodeGraph originNode;
    private NodeGraph destinationNode;
    private AgentProperties agentProperties;
    private List<NodeGraph> sequenceNodes;
    private Agent agent;

    /**
     * Constructs a 'RoutePlanner' instance for calculating a route.
     *
     * @param originNode      The starting node of the route.
     * @param destinationNode The destination node of the route.
     * @param agent           The agent for which the route is being planned.
     */
    public RoutePlanner(NodeGraph originNode, NodeGraph destinationNode, Agent agent) {
        this.originNode = originNode;
        this.destinationNode = destinationNode;
        this.agent = agent;
        this.agentProperties = agent.getProperties();
        this.sequenceNodes = new ArrayList<>();
    }
}
```

```

    }

    /**
     * Defines the path for the agent based on route choice properties and
     * strategies.
     *
     * @return A 'Route' object representing the calculated route.
     * @throws Exception
     */
    public Route definePath() throws Exception {

        if (shouldUseMinimization()) {
            if (agentProperties.minimisingDistance) {
                RoadDistancePathFinder finder = new RoadDistancePathFinder();
                return finder.roadDistance(originNode, destinationNode, agent);
            } else {
                AngularChangePathFinder finder = new AngularChangePathFinder();
                return finder.angularChangeBased(originNode, destinationNode, agent);
            }
        }

        /**
         * Through regions with barrier-subgoals or not.
         */
        if (isRegionBasedNavigation()) {
            RegionBasedNavigation regionsPath =
                new RegionBasedNavigation(originNode, destinationNode, agent);
            sequenceNodes = regionsPath.sequenceRegions();
        }

        /**
         * Sub-goals: only barriers, no regions
         */
        if (agentProperties.barrierBasedNavigation && !isRegionBasedNavigation()) {
            BarrierBasedNavigation barriersPath =
                new BarrierBasedNavigation(originNode, destinationNode, agent, false);
            sequenceNodes = barriersPath.sequenceBarriers();
        }

        /**
         * Sub-goals: localLandmarks, possibly through regions
         */
        else if (agentProperties.usingLocalLandmarks) {
            LandmarkNavigation landmarkNavigation =
                new LandmarkNavigation(originNode, destinationNode, agent);
            if (isRegionBasedNavigation() && !sequenceNodes.isEmpty())
                sequenceNodes = landmarkNavigation.regionOnRouteMarks(sequenceNodes);
        }
    }
}

```

```

        else
            sequenceNodes = landmarkNavigation.onRouteMarks();
    }

    /**
     * Not active in the empirical-based simulation! Distant-landmarks - only when
     * the agent doesn't minimise road costs (in other cases, distant landmarks are
     * already considered when filling the sequence); a) possibly via sub-goals or
     * through regions; b) just based on distant landmarks. else: just
     * global-landmarks maximisation path
    */
    else if (agentProperties.usingDistantLandmarks && !shouldUseLocalHeuristic()) {
        GlobalLandmarksPathFinder finder = new GlobalLandmarksPathFinder();
        if (!sequenceNodes.isEmpty())
            return finder.globalLandmarksPathSequence(sequenceNodes, agent);
        else {
            return finder.globalLandmarksPath(originNode, destinationNode, agent);
        }
    }

    if (sequenceNodes.isEmpty()) {
        if (agentProperties.localHeuristicDistance) {
            RoadDistancePathFinder finder = new RoadDistancePathFinder();
            return finder.roadDistance(originNode, destinationNode, agent);
        } else {
            AngularChangePathFinder finder = new AngularChangePathFinder();
            return finder.angularChangeBased(originNode, destinationNode, agent);
        }
    }

    if (agentProperties.localHeuristicDistance) {
        RoadDistancePathFinder finder = new RoadDistancePathFinder();
        return finder.roadDistanceSequence(sequenceNodes, agent);
    } else {
        AngularChangePathFinder finder = new AngularChangePathFinder();
        return finder.angularChangeBasedSequence(sequenceNodes, agent);
    }
}

/**
 * Checks if the agent should use minimization for route planning.
 *
 * @return True if the agent should use minimization, otherwise false.
 */
private boolean shouldUseMinimization() {
    return agentProperties.onlyMinimising;
}

```

```

    /**
     * Checks if the agent should use local heuristics for route planning.
     *
     * @return True if the agent should use local heuristics, otherwise false.
     */
    private boolean shouldUseLocalHeuristic() {
        return (agentProperties.localHeuristicDistance || agentProperties.localHeuristicAngular);
    }

    /**
     * Verifies if region-based navigation should be enabled for route planning
     * based on distance thresholds. If not, it disables region-based navigation in
     * agent properties.
     */
    private boolean isRegionBasedNavigation() {
        return agentProperties.regionBasedNavigation
            && GraphUtils.getCachedNodesDistance(originNode,
                destinationNode) >= Parameters.regionBasedNavigationThreshold
            && originNode.regionID != destinationNode.regionID;
    }
}

-----RouteData.java-----
package pedSim.utilities;

import java.util.List;

import org.locationtech.jts.geom.LineString;

/**
 * To store information about the walked routes.
 *
 */
public class RouteData {

    public Integer origin;
    public Integer destination;
    public boolean minimisingDistance;
    public boolean minimisingAngular;
    public boolean localHeuristicDistance;
    public boolean localHeuristicAngular;

    public boolean regionBased;
    public boolean onRouteMarks;
}

```

```
public boolean barrierSubGoals;
public boolean distantLandmarks;

public double naturalBarriers;
public double severingBarriers;

public List<Integer> edgeIDsSequence;
public String routeID;
public String group;
public String routeChoice;
public LineString lineGeometry;
}

-----
StringEnum.java-----
package pedSim.utilities;

public class StringEnum {

    public enum RouteChoice {
        ROAD_DISTANCE, ANGULAR_CHANGE, LANDMARKS_DISTANCE, LANDMARKS_ANGULAR, LOCAL_LANDMARKS_DISTANCE,
        LOCAL_LANDMARKS_ANGULAR, DISTANT_LANDMARKS_DISTANCE, DISTANT_LANDMARKS_ANGULAR, DISTANT_LANDMARKS,
        REGION_DISTANCE, REGION_BARRIER_DISTANCE, REGION_ANGULAR, REGION_BARRIER_ANGULAR, BARRIER_DISTANCE,
        BARRIER_ANGULAR,
    }

    public enum RouteChoiceProperty {
        ROAD_DISTANCE, ANGULAR_CHANGE, ROAD_DISTANCE_LOCAL, ANGULAR_CHANGE_LOCAL,
        USING_ELEMENTS, NOT_USING_ELEMENTS,
        LOCAL_LANDMARKS, BARRIER_SUBGOALS, NO_SUBGOALS, REGION_BASED, NOT_REGION_BASED,
        USING_DISTANT, NOT_USING_DISTANT
    }

    public enum Groups {
        NULLGROUP, POPULATION, GROUP1, GROUP2, GROUP3, GROUP4, GROUP5, GROUP6,
    }

    public enum LandmarkType {
        LOCAL, GLOBAL
    }

    public enum BarrierType {
        ALL, POSITIVE, NEGATIVE, SEPARATING,
    }

    public static String getAbbreviation(RouteChoice choice) {

```

```
String[] parts = choice.toString().split("_");
StringBuilder abbreviation = new StringBuilder();
for (String part : parts) {
    abbreviation.append(part.charAt(0));
}
return abbreviation.toString();
}
```