Symbolic Artificial Intelligence Techniques to Facilitate Proactive Robot Assistance

Helen Harman

Promotoren: prof. dr. ir. P. Simoens, prof. dr. ir. B. Dhoedt
Proefschrift ingediend tot het behalen van de graad van
Doctor in de industriële wetenschappen: informatica

UNIVERSITEIT
GENT

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. B. Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2019 - 2020

ISBN 978-94-6355-394-0
NUR 984
Wettelijk depot: D/2020/10.500/71

Ghent University
Faculty of Engineering and Architecture
Department of Information Technology

imec
Internet Technology and Data Science Lab

Examination Board:

prof. dr. Pieter Simoens	(supervisor)
prof. dr. Bart Dhoedt	(supervisor)
em. prof. dr. Daniël De Zutter	(chair)
prof. dr. Femke Ongenae	(secretary)
prof. dr. Sofie Van Hoecke	
prof. dr. Tony Belpaeme	
prof. dr. Tom Holvoet	
prof. dr. Mauro Dragone	

Funded by FWO-Vlaanderen

Thesis for acquiring the degree of
Doctor of Information Engineering Technology

Acknowledgements

"*Don't let anyone rob you of your imagination, your creativity, or your curiosity. It's your place in the world; it's your life. Go on and do all you can with it, and make it the life you want to live.*"

Mae C. Jemison

After nearly four years at Ghent University, my PhD has come to its concluding moments. During this journey, I met and was helped by many people. I would therefore like to take the time to thank the people who made this thesis possible.

First, I would like to thank my supervisor prof. Pieter Simoens. Thank you for providing me the opportunity to pursue a PhD and guiding me throughout the process. Having the freedom to explore the Symbolic AI research area has been an interesting and enjoyable experience. I wish you success in all your future research pursuits. I also am grateful to my second supervisor, prof. Bart Dhoedt, for assisting in kicking start my PhD.

The feedback from the members of the examination board has helped me to improve my PhD thesis. I greatly appreciate the time and effort you have taken to read my PhD.

My colleagues have also often provided me with guidance. Christof Mahieu helped hire me; introduced me to the department and its robots, and has kept me company in the office throughout my time here. Kashav Chintamani, thank you for assisting with my FWO proposal and first journal publication. Furthermore, I am thankful to all the department's support staff for making this work possible.

I had the pleasure of sharing the office with and/or being on the same team as Stijn, Piet, Christof, Yara, Ilja, Han, Bo, Wei-Cheng and Inês. There are also many colleagues within the surrounding offices, such as Sam and Jeroen, who I have appreciated gaining advice from. Bedankt en veel succes.

My friends, lectures and supervisors from my previous university and work places encouraged and supported me in my pursuit to move aboard and gain a PhD. This includes my fellow Aberystwyth University ex-MEngers: Connor, Craig, Dan and Sam. I am grateful to you all for your continuing support.

Ghent, July 2020
Helen Harman

Table of Contents

Acknowledgements **i**

Samenvatting **xvii**

Summary **xxi**

1 Introduction **1**
1.1 Symbolic Artificial Intelligence . . . 3
1.2 Intention Recognition . . . 4
1.2.1 Goal and Plan Recognition . . . 4
1.2.2 Goal Recognition Design . . . 5
1.2.3 Action Prediction . . . 5
1.3 Task Planning . . . 5
1.4 Robot Planning . . . 6
1.5 Smart Environments . . . 6
1.6 Research Challenges . . . 7
1.7 Research Contributions . . . 12
1.8 Publications . . . 16
1.8.1 Journal Papers . . . 16
1.8.2 Conference and Workshop Papers . . . 16
References . . . 18

2 Goal Recognition **27**
2.1 Introduction . . . 28
2.2 Background . . . 31
2.2.1 Symbolic Task Planning . . . 32
2.2.2 Goal Recognition . . . 33
2.3 Action Graph Structure and Formal Definitions . . . 33
2.4 Cyclic Action Graph Creation . . . 36
2.4.1 Preprocessing: Multi-Valued Problem Generation . . . 36
2.4.2 Inserting Actions Into an Action Graph . . . 37
2.4.3 Identifying Goal Actions . . . 39
2.4.4 Example . . . 39
2.5 Node Distance Initialisation . . . 40
2.5.1 Node Value Initialisation Algorithm . . . 41
2.5.2 Example . . . 42

2.6 Updating the Goal Probabilities . . . 43
2.6.1 Update Rule 1: Distance From Observed Action . . . 43
2.6.2 Update Rule 2: Change in Distance From the Observed Actions . . . 45
2.6.3 Processes Common to Update Rules 1 and 2 . . . 46
2.7 Experiments . . . 46
2.7.1 Evaluation Metrics . . . 47
2.7.2 Goal Recognition with a Known Initial State . . . 49
2.7.3 Goal Recognition with an Inaccurate Initial State . . . 56
2.8 Related Work . . . 57
2.8.1 Recognition as Parsing . . . 58
2.8.2 Recognition as Planning . . . 59
2.9 Conclusion . . . 59

3 Goal Recognition Design **77**
3.1 Introduction . . . 78
3.2 Related Work . . . 81
3.3 Background . . . 83
3.3.1 Formal Definition . . . 83
3.3.2 Goal Recognition . . . 84
3.4 Distinctiveness Metric . . . 85
3.5 Action Graphs for Modelling GRD Problems . . . 88
3.5.1 Structural Features . . . 88
3.5.2 Action Graph Creation . . . 89
3.6 Find All Non-Distinctive Plan Prefixes . . . 95
3.6.1 Label Which Nodes Belong to Which Goals . . . 95
3.6.2 Extract Non-Distinctive Prefixes . . . 95
3.7 Performing Action Replacement to Reduce ACD . . . 97
3.7.1 Defining Modifications . . . 97
3.7.2 Exhaustive . . . 98
3.7.3 Shrink–Reduce . . . 99
3.8 Performing Action Removal to Reduce ACD . . . 102
3.9 Experiments . . . 106
3.9.1 Action Replacement Scalability Experiments . . . 106
3.9.2 Action Replacement Applied to a Kitchen Domain . . . 110
3.9.3 Action Removal Experiments . . . 113
3.10 Conclusions . . . 116

4 Proactive Robot Assistance **131**
4.1 Introduction . . . 132
4.2 Related Work . . . 134
4.2.1 Intention Recognition . . . 134
4.2.2 Proactive Robot Assistance . . . 136
4.3 Problem Statement . . . 137
4.3.1 Problem Formalisation . . . 137

4.3.2 Running Example: Smart Home Kitchen 138
4.4 Action Graphs 139
4.4.1 Structural Features 139
4.4.2 Creation 139
4.5 Node Value Updates 141
4.5.1 Updating Node Values Based on Observations 141
4.5.2 Node Value Updates with Reverse Actions 144
4.6 Action Prediction with Action Graphs 144
4.7 Proactive Robot Assistance 145
4.8 Experiments: Action Prediction Accuracy 147
4.8.1 Setup 147
4.8.2 Hypothesis Goal Action Prediction 148
4.8.3 Single Subgoal Action Prediction 150
4.8.4 Multiple Interleaving Subgoals 151
4.9 Experiments: Robotic Assistance 152
4.9.1 Setup 152
4.9.2 Results and Discussion 153
4.10 Conclusion and Future Work 156

5 Robot Assistance in Dynamic Smart Environments 167
5.1 Introduction 168
5.2 Related Work 170
5.2.1 Planning in Smart Environments 170
5.2.2 Capability Reasoning 171
5.2.3 Planners with External Reasoners 173
5.2.4 Interleaving Planning and Execution 173
5.3 Background: Hierarchical Planning in the Now 175
5.4 Framework Overview 176
5.5 Incorporating IoT into Continual Planning 178
5.5.1 Generating a Planning Problem 178
5.5.2 Capability Checking 181
5.5.3 Context Monitoring 183
5.5.4 Plan Execution 185
5.6 Designing the Hierarchy for Smart Environments 187
5.6.1 Example Scenario 187
5.6.2 Baseline: All Actions and State Within a Single Level 188
5.6.3 Splitting up Key Concepts 188
5.6.4 Separate Repeated Blocks 188
5.6.5 Reducing Unused Knowledge 192
5.7 Proof of Concept in a Smart Home 192
5.8 Simulated Experiments 195
5.8.1 Exp. 1: Single "remote" Object to Represent all Devices 196
5.8.2 Exp. 2: IoT Device Failure and Plan Quality 197
5.8.3 Exp. 3: Comparison with Planning Everything Upfront 200
5.8.4 Exp. 4: Varying the Number of Levels in the Hierarchy 202

5.9 Conclusions and Future Work 203

6 Learning Symbolic Action Definitions 217
6.1 Introduction 218
6.2 Problem Formalisation 219
6.3 Discover Objects 221
6.3.1 Discover Locations 221
6.3.2 Discover Image Objects 222
6.4 Generate States 223
6.5 Generate Action Definitions 223
6.5.1 Linked Locations 224
6.5.2 Finding Locations' Dependencies 224
6.5.3 Finding Image Objects' Dependencies 225
6.5.4 Remove Unrequired Possible Preconditions 226
6.5.5 Generate Action Definitions 226
6.6 Handling Large State Spaces 226
6.7 Experiments and Discussion 227
6.7.1 Action Definitions 227
6.7.2 Time Complexity 228
6.7.3 Task Planning Results: No Missing Transitions 229
6.7.4 Task Planning Results: Missing Transitions 230
6.8 Related Work 230
6.9 Conclusion 231
References 233

7 Conclusions and Future Research 237
7.1 Research Question Discussion 238
7.2 Future Work 243
References 247

List of Figures

1.1 Example scenario 2

2.1 Overview of our GR approach 30
2.2 Types of order constraints 34
2.3 Example Easy-IPC-Grid problem 40
2.4 Steps taken in insert actions into an Action Graph 41
2.5 Action nodes labelled with their distance from each goal . . 43
2.6 Action nodes' distance from each goal of a grid-based navigation domain. 46
2.7 Performance profiles to compare the recognition times . . . 50
2.8 Average F1-Score when first 10, 30, 50, 70 and 100% of observations have been processed 53
2.9 Average F1-Score when random observations are missing . . 55
2.10 Average F1-Score when the initial state is inaccurate 57

3.1 Example non-distinctive plan prefix 79
3.2 Conceptual overview of our GRD approach 81
3.3 Example: ACD reduction 86
3.4 Example: WCD_{dep} reduction 88
3.5 Example Action Graph 89
3.6 Action Graph creation steps 91
3.7 Action Graph creation steps for a navigation domain 94
3.8 Grid-based navigation problem 94
3.9 Action node and its replacement sub-graphs 98
3.10 Shrink–Reduce example 101
3.11 Action Graph before Shrink–Reduce 102
3.12 GRD problem with alternative actions 103
3.13 GRD problem in which a subset of goals have alternative actions 104
3.14 GRD problem with three goals 105
3.15 Action replacement results graphs: increasing number of variables 108
3.16 Action replacement results graphs: increasing number of values 109

3.17 Action replacement results graphs: increasing number of goals 110
3.18 Action replacement results graphs: performance profiles . . 111
3.19 Action removal results graphs: increasing number of goals . 114
3.20 Action removal results graphs: increasing number of goals . 115
3.21 Action removal results graphs: performance profiles. 115
3.22 Action replacement detail results graphs: increasing number of variables 122
3.23 Action replacement detail results graphs: increasing number of values 122
3.24 Action replacement detail results graphs: increasing number of goals 123
3.25 Action removal detail results graphs: increasing number of goals 123
3.26 Action removal detail results graphs: increasing grid size . . 124

4.1 Conceptual overview 132
4.2 Example DTGs 138
4.3 Action graph creation 140
4.4 Action graph with updated node values 144
4.5 Proactive response overview 146
4.6 Layout of the environment for the simulated experiments . . 147
4.7 Human's and robot's actions for making breakfast 153
4.8 AND-OR tree for making breakfast 158
4.9 AND-OR tree for making dinner 158
4.10 AND-OR tree for making lunch 158
4.11 Updating node values with reverse action 160
4.12 Human's and robot's actions for making dinner 162
4.13 Human's and robot's actions for making lunch 162

5.1 Hierarchical planning in the now 176
5.2 Incorporating IoT devices into continual planning 177
5.3 Continual planning processes 178
5.4 Action, capability, robot and device definition 182
5.5 Composite and primitive action execution 186
5.6 Single domain file 189
5.7 Action definitions split into two domain files 189
5.8 Action definitions split into three domain files 190
5.9 Action definitions split into four domain files 191
5.10 Real world tests 194
5.11 Simulated world 196
5.12 Hierarchical v.s. non-hierarchical planning 202

6.1 Example plan for solving a Puzzle problem 220
6.2 Example plan for solving a ToH problem 221
6.3 Example plan for solving a Lights-Out problem 221

6.4 Transitions and their overlapping changed image area 222
6.5 Discovered locations for a ToH domain 222
6.6 Transition and the discovered image object. 223
6.7 Learning a location's dependencies. 225
6.8 Learning an image object's dependencies. 226

List of Tables

2.1 TP, FP, FN and TN definitions 48
2.2 Run-times per domain 51
2.3 Average size of the Action Graph per domain 52
2.4 Per domain accuracy for when the initial state is known . . . 53
2.5 Per domain accuracy for when observations are missing . . 65
2.6 Per domain accuracy for when the initial state is inaccurate . 66
2.7 Per domain accuracy for when the initial state is inaccurate continued 68

3.1 Shrink–Reduce v.s. Exhaustive 113
3.2 Plans for the goals of the Kitchen domain 121

4.1 Number of TPs and FPs after the first N % of observations . 149
4.2 Action prediction results for subgoals 150
4.3 Action prediction results for 2 subgoals 152
4.4 Detailed results for when 2 subgoals are achieved 161

5.1 State Estimators 179
5.2 Including all devices v.s. single object to represent all devices 197
5.3 Hierarchical v.s. non-hierarchical planning for when a device malfunctions 199
5.4 Effect of varying the number of layers in the hierarchy . . . 203
5.5 Composite actions 206
5.6 Primitive actions 207

6.1 Action definition generation results. 228
6.2 Time spent planning. 229
6.3 Accuracy when differing percentages of transitions are missing 230

List of Acronyms

A	
ACD	Average Case Distinctiveness
AI	Artificial Intelligence
AMA	Action Model Acquisition
ARMS	Action Relation Modelling System
ASP	Answer Set Programming
B	
BDI	Belief Desire Intention
BFS	Breadth First Search
BFT	Breadth First Traversal
BHPN	Belief Hierarchical Planning in the Now
C	
CPU	Central Processing Unit
D	
DTG	Domain Transition Graph
DYAMAND	DYnamic, Adaptive MAnagement of Networks and Devices
E	
ECD	Expected Case Distinctiveness
F	
FD	Fast Downward
FN	False Negatives
FP	False Positives
G	
GR	Goal Recognition
GRD	Goal Recognition Design
H	
HPN	Hierarchical Planning in the Now
HTN	Hierarchical Task Network
HTTP	HyperText Transfer Protocol
I	
IoT	Internet of Things

IoT-O	Internet of Things – Ontology
IoRT	Internet of Robotic Things
J	
JSON	JavaScript Object Notation
M	
MA-PDDL	Mulit-Agent – Planning Domain Definition Language
MAPL	Mulit-Agent Planning Language
MDP	Markov Decision Process
N	
NN	Neural Network
O	
OWL	Web Ontology Language
P	
PDDL	Planning Domain Definition Language
PEIS	Physically Embedded Intelligent Systems
POCL	Partial Order Causal Link planning
PPDDL	Probabilistic Planning Domain Definition Language
R	
RAM	Random Access Memory
RDDL	Relational Dynamic Influence Diagram Language
ROS	Robotic Operating System
S	
SAS	Simplified Action Structure
SLAF	Simultaneous Learning And Filtering
STRIPS	Stanford Research Institute Problem Solver
SRDL	Semantic Robot Description Language
T	
TFD	Temporal Fast Downward
TFD/M	Temporal Fast Downward with Modules
TN	True Negatives
ToH	Towers of Hanoi
ToM	Theory of Mind
TP	True Positives
W	
WCD	Worst Case Distinctiveness

Samenvatting
– Summary in Dutch –

Artificiële Intelligentie (AI) wordt in toenemende mate geïntegreerd in ons dagelijks leven. Het verbeterde vermogen van AI-systemen om de omgeving waar te nemen, te begrijpen, te voorspellen en te manipuleren zijn al binnen vele domeinen zichtbaar, zoals medische diagnostiek, computerspelletjes, voertuigautomatisatie en assistentie door robots. Een artificieel intelligente agent neemt zijn omgeving waar via sensoren die in die slimme omgeving zijn geïntegreerd. Uit de data geproduceerd door deze sensoren kunnen AI-agenten informatie afleiden over de toestand van de omgeving en zo kennis verwerven over welke handelingen een persoon uitvoert in een slimme omgeving. De handelingen van zowel personen als kunstmatige agenten (bv. robots) kunnen symbolisch worden gemodelleerd. Symbolische modellen faciliteren AI agenten om de bedoelingen van een mens te herkennen en zijn eigen acties te plannen.

Methoden voor intentieherkenning beogen om het doel en/of plan van de geobserveerde persoon of agent te herkennen. In een keukenomgeving bijvoorbeeld kan het doel zijn om een ontbijt, middagmaal of avondmaal te maken en het plan zal bestaan uit het nemen van verschillende zaken (bv. brood en een bord) en het gebruik van apparaten (bv. een broodrooster). Kennis van de intenties van een persoon laat een robot toe om assisterende acties te plannen en uit te voeren die helpen om dat doel sneller te bereiken. Tijdens het uitvoeren van zijn plan kan de robot actuele informatie over de toestand van de omgeving verkrijgen van sensoren en actuatoren in de omgeving, het zogenaamde Internet-of-Things (IoT). Door het aansturen van externe actuatoren zoals deurmechanismen en gloeilampen kan een robot taken uitvoeren die hij anders niet zou kunnen uitvoeren.

Dit proefschrift heeft als doel om de intelligentie van robots in dynamische en slimme omgevingen te verbeteren zodat ze proactief hulp kunnen bieden. De eerste 3 hoofdstukken richten zich op drie elementen van intentieherkenning: Doelherkenning (Engels: Goal Recognition (GR)), het ontwerpen van een omgeving om doelherkenning te versnellen (Engels: Goal Recognition Design (GRD)) en actievoorspelling. Voor elk van deze elementen transformeert onze aanpak een symbolisch model in een Actie-

graaf (Engels: Action Graph). Actiegrafen modelleren de afhankelijkheden tussen de acties. De constructiemethode en het gebruik van de graaf zijn specifiek voor elk element.

Onze GR-aanpak is ontwikkeld voor situaties waarin de definitie van de aanvankelijke wereldtoestand onnauwkeurig is, zoals wanneer de status van een object onbekend is of verkeerd wordt gedetecteerd. Een object of persoon kan bijvoorbeeld een onbepaalde locatie hebben omdat het verborgen is achter een ander object. Na het genereren van een Actiegraaf uit het symbolische model worden de knopen van de graaf gelabeld met hun afstand tot het bereiken van elk mogelijk doel. Wanneer een observatie wordt ontvangen, worden de doelprobabiliteiten geüpdatet op basis van de afstand die de geobserveerde actie tot elk doel heeft of de verandering in afstand. De doelen met de hoogste probabiliteit vormen dan de verzameling van voorspelde doelen.

Het opzet van GRD is om het aantal acties te verminderen dat moet worden geobserveerd voordat de intentie van de geobserveerde persoon ondubbelzinning kan worden bepaald. Deze actiesequenties worden niet-onderscheidende planprefixen genoemd. Er worden twee soorten aanpassingen gedaan om de lengte van niet-onderscheidende planprefixen te verminderen. Ten eerste worden acties in de mogelijke plannen vervangen door meer onderscheidende acties. Bijvoorbeeld, wanneer de locatie van een item wordt gewijzigd, wordt de actie om dit item te nemen van zijn oorspronkelijke actie vervangen door een actie om dit item te nemen van de nieuwe locatie. Als deze nieuwe locatie alleen items bevat die nodig zijn om één van de doelen te bereiken en de mens deze locatie betreedt, kan zijn doel worden bepaald. Ten tweede worden bepaalde acties verhinderd zodat mensen gedwongen worden een alternatief, meer onderscheidend, plan te nemen. Door bijvoorbeeld een obstakel te plaatsen in de ruimte moet de persoon een alternatieve route nemen om de gewenste bestemming te bereiken. Onze benadering van GRD genereert een Actiegraaf door het uitvoeren van een breedte-eerste zoekstrategie in de zoekruimte van elk doel naar de initiële staat. De graaf wordt dan doorkruist om een door ons ontworpen afstandsmaat te berekenen. Deze metriek houdt er rekening mee dat de lengte van het hele plan, evenals de niet-onderscheidende planprefix, kan worden gewijzigd wanneer de omgeving wordt herontworpen. Onze Shrink-Reduce methode verkleint de plannen en vermindert vervolgens de niet-distinctieve voorvoegsels. Deze aanpak is minder rekenintensief dan een exhaustieve aanpak, maar kan niet garanderen dat er een oplossing wordt gevonden. De prestaties van beide methodes worden vergeleken op een testdomein met verschillend aantal doelen, variabelen en waarden, en op een realistisch keukendomein. Voor navigatieproblemen in een rooster blijkt onze methode om acties onmogelijk te maken effectiever te zijn dan een stand-der-techniek methode.

In plaats van het bereiken van één vooraf gedefinieerd doel, zou een persoon meerdere tussenliggende plannen kunnen uitvoeren. Daarom hebben we ook een Actiegraaf-aanpak ontwikkeld om te voorspellen welke acties een mens vervolgens zal uitvoeren. De knoopwaarden in de Actiegraaf worden geüpdatet op basis van waargenomen acties en de hoogst gewaardeerde knopen worden gebruikt als voorspelde acties. Vervolgens voert een robot een van de geëxtraheerde, d.w.z. voorspelde, acties uit als deze de mens niet belemmeren of vertragen. Experimenten hebben aangetoond dat onze aanpak acties nauwkeurig voorspelt, zelfs wanneer de plannen van meerdere subdoelen worden geobserveerd. Verder toonde een proof of concept in een keukenomgeving aan dat een robot in staat is om een persoon succesvol te assisteren.

De robot kan worden bijgestaan door de sensoren en actuatoren van een slimme omgeving. Het integreren van de toestanden en acties van Internet of Things (IoT) apparaten in de planner van een robot verhoogt echter de vereiste rekenkracht en kan zo de uitvoering van een taak vertragen. Bovendien moet de robot vaak herplannen door onverwachte acties van de persoon. In dit proefschrift wordt een hiërarchisch en continu Planning-in-the-Now framework ontwikkeld om herplannen efficiënter te maken. De informatie over de omgeving afkomstig van IoT sensoren wordt verwerkt in de back-end van de cloud en de robot wordt alleen geïnformeerd over de toestand die relevant is voor zijn huidige taakplan. De taken worden opgesplitst in subtaken en de eerste deeltaak wordt gepland en uitgevoerd vóór de volgende. Hierdoor kan de robot eerder in actie komen en kan de frequentie van de (her)planning worden gereduceerd.

Opdat een robot de bedoelingen van een mens kan herkennen en zijn eigen acties kan plannen, vereist ons werk een symbolische representatie van de handeling die geobserveerde agenten kunnen uitvoeren. Deze representatie wordt meestal manueel ontwikkeld, wat een tijdrovend en foutgevoelig proces kan zijn. Wij bieden daarom een techniek om uit ongelabelde beeldparen symbolische actiedefinities te maken. Beeldparen worden getransformeerd in staten en acties door de pixels die tegelijkertijd van waarde veranderen om te zetten in objecten en atomen. Door statische atomen te genereren wordt verhinderd dat ongeldige acties worden gegenereerd. De geproduceerde actiedefinities voor de Puzzle, Lights-Out en Towers of Hanoi (ToH) domeinen werden gebruikt in planningsproblemen met verschillende aantallen objecten.

Het onderzoek in dit proefschrift verbetert het vermogen van AI agenten om de intenties van een persoon te herkennen en autonoom hulp te bieden. In de toekomst zal Artificiële Intelligente de mens waarschijnlijk proactief bijstaan in alle aspecten van zijn leven. Verdere vooruitgang in de onderwerpen die in dit proefschrift aan bod komen zal de sleutel zijn om dit mogelijk te maken.

Summary

We are witnessing Artificial Intelligence (AI) increasingly becoming a part of our lives. Improvements to AI systems' ability to perceive, understand, predict and manipulate the environment are apparent within many domains. These domains include medical diagnosis, game playing, vehicle automation and robotic assistance. An artificial agent's ability to perceive the environment is enabled by sensors, such as those deployed on a robot and within a smart environment. These sensors produce data from which AI agents can derive information about the state of the environment, and consequently gain knowledge of which actions a human is performing. The actions of both non-artificial agents (e.g., humans) and artificial agents (e.g., robots) can be modelled symbolically. Symbolic models facilitate artificial agents with the ability to recognise a human's intentions and plan their own actions.

Symbolic-based approaches to intention recognition aim to recognise the goal and/or plan of the observee. For instance, in a kitchen environment, a goal could be to make breakfast, lunch or dinner and the plan will involve taking different items (e.g., bread and a plate) and using appliances (e.g., a toaster). Knowledge of a human's intentions permit a robot to plan and execute actions that help the human to achieve their goal sooner. Whilst executing its plan the robot can gain up to date information on the state of the environment from IoT sensors and command IoT actuators. By commanding IoT actuators, such as door mechanisms and light bulbs, a robot can complete tasks it would otherwise be incapable of accomplishing.

This PhD thesis aims to improve robots' ability to provide proactive assistance in dynamic smart environments. The first 3 chapters focus on three elements of intention recognition: Goal Recognition (GR), Goal Recognition Design (GRD) and action prediction. For each of these elements, our approach transforms the symbolic representation into an Action Graph. Action Graphs model the order constraints between actions. The creation method and the usage of the graph is specific to each element.

Our GR approach handles situations in which the definition of the initial world state is inaccurate. This occurs when an object's state is unknown or sensed/determined incorrectly. For instance, an item or human could be occluded, and thus its location unknown. After generating an Action

Graph from the symbolic model, the graph's nodes are labelled with their distance from each hypothesis goal. Subsequently, the goals' prior (uniform) probability is set. When an observation is received, the goal probabilities are updated based on either the distance the observed action's node is from each goal or the change in distance. The goals with the highest probability are returned as the set of candidate goals.

The aim of GRD is to reduce the number of actions that require observing before the human's true goal can be distinguished from the false goals. These action sequences are called non-distinctive plan prefixes. Two types of modifications are made to reduce non-distinctive plan prefixes. A first modification is that actions in the available plans are replaced with more distinctive actions. For instance, when an item's location is changed, the action to take that item from its original location is replaced with the action to take it from its new location. If this location only contains items required to reach one of the goals and the human accesses this location, their goal is predictable. A second modification is that certain actions are prevented/removed so that humans are forced to take an alternative, more distinctive, plan. For example, if an obstacle is created, humans must take an alternative route to reach their desired destination. Our approach to GRD generates an Action Graph by performing a Breadth First Search (BFS) backwards from each goal to the initial state. The graph is then traversed to calculate our novel distinctiveness metric. This metric takes into account that the length of the whole plan, as well as the non-distinctive prefix, could be altered when the environment is redesigned. Our Shrink-Reduce method shrinks the plans, then reduces the non-distinctive prefixes. It is less computationally expensive than our exhaustive approach but is not guaranteed to find the optimal solution. These approaches were compared using a test domain containing varying amounts of goals, variables and values, and a realistic Kitchen domain. Moreover, our action removal method is shown to be more efficient than a state of the art approach, at increasing the distinctiveness of various grid-based navigation problems.

Rather than achieving a single predefined goal, a human could perform multiple interleaving plans and their goal could be a subgoal of another goal. Therefore, we developed an Action Graph approach to predicting which actions a human will perform next. The Action Graph's node values are updated based on which actions are observed and the highest valued actions are extracted. Subsequently, a robot executes one of the extracted, i.e., predicted, actions if it does not impact the flow of the human by obstructing or delaying them. Experiments demonstrated that our approach adequately predicts actions, even when the plans of multiple interleaving subgoals are observed. Further, a simulated proof of concept in a kitchen environment showed a robot is able to successfully assist a person.

Whilst the robot is acting it can be assisted by the sensors and actuators of a smart environment. Nevertheless, incorporating the states and actions of Internet of Things (IoT) devices into the robot's onboard planner increases its computational load, and thus can delay the execution of a task. Moreover, tasks may be frequently replanned due to the unanticipated actions of humans. Therefore, we developed a Hierarchical Continual Planning in the Now framework to mitigate these inadequacies. IoT state knowledge is monitored by the cloud back-end and the robot is only informed about state relevant to its current task plan. Tasks are split-up into subtasks and the first subtask is planned and executed before the subsequent one. This enables the robot to start acting sooner and the frequency of (re)planning to be reduced.

For a robot to recognise a human's intentions and plan its own actions, our work requires a symbolic representation of the actions agents can execute. This representation is usually manually developed, which can be a time consuming and error prone process. We therefore provide a technique for creating symbolic action definitions from unlabelled image pairs. Image pairs are transformed into states and actions by converting the pixels that change value simultaneously into objects and atoms. Subsequently, static atoms are generated to prevent invalid actions from being created from the resulting action definitions. The produced action definitions for Puzzle, Lights-Out and Towers of Hanoi (ToH) domains were shown to be applicable to planning problems with different sized state spaces.

The work of this doctoral thesis improves artificial agents' ability to recognise a human's intentions and autonomously provide assistance. In the future, artificial agents are likely to be proactively assisting humans in all aspects of their lives. Further advances in the topics covered by this thesis will be key to enabling this.

1

Introduction

"*Attempts to 'organize' the field of Artificial Intelligence have never been wholly successful.*"
Nils J. Nilsson (1971)

The field of AI involves the development of systems that can perceive, understand, predict, and manipulate the environment [1]. Much of the early research within this field focused on creating machines that could act like humans, for instance, those that could pass the Turing Test [2]. Over time the definition of what AI is has expanded and AI concepts have been applied to many domains. In some ways machines have surpassed humans as they are much quicker at recalling and processing information. Today AI systems are capable of diagnosing medical conditions [3]; understanding and producing text [4, 5]; playing games [6]; driving vehicles [7], and providing robotic assistance [8]. Nevertheless, these systems have limitations and AI agents that fully perceive, think and act as humans still only exist in science fiction.

AI can enable a robot to recognise a human's intentions and provide assistance. An example scenario is depicted in Figure 1.1. The data from IoT sensors is processed to infer which actions the human is performing. In our scenario, the human is observed moving to the cupboard and taking coffee beans. This information enables the human's goal and future actions to be predicted. After predicting the human will make coffee, the robot creates and executes a plan that helps the human to realise this sooner, i.e., the robot fetches a cup. Whilst acting the robot acquires up to date state information from IoT sensors and commands IoT actuators to expand its

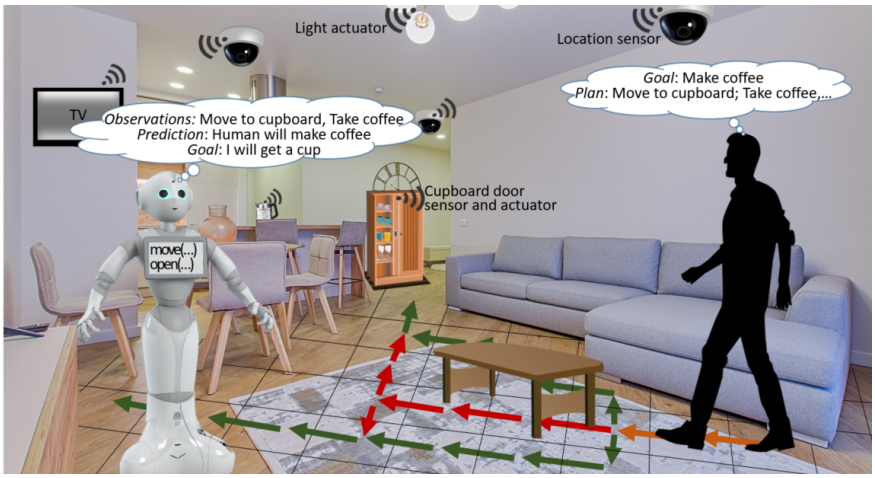


Figure 1.1: Example scenario. The arrows depict move actions. The placement of a table forces the human to take a more distinctive plan to reach the television or the cupboard, and thus their destination can be recognised after fewer actions have been observed.

manipulation capabilities. This doctoral thesis covers several topics that are key to enabling proactive robot assistance in dynamic smart environments.

To situate our research, this chapter presents a brief overview of the background information and history of this research area. Subsequently, a list of research questions with hypotheses, an outline of the research contributions, and a list of publications produced in preparation for this thesis are provided. Each of the chapters in the main body of this thesis corresponds to a key publication.

1.1 Symbolic Artificial Intelligence

AI systems can be broadly categorised as either data-driven or knowledge-driven [9, 10]. To train a model, data-driven approaches require a large dataset. In contrast, knowledge-driven approaches rely on logical descriptions [11, 12] of the world and how it can be manipulated. Knowledge-driven approaches require less data; however, the development of symbolic logic is often time consuming and error prone. This thesis primarily focuses on knowledge-driven methods. In the penultimate chapter, a data-driven approach is proposed to learn the symbolic representation.

Symbolic logic represents propositions that describe the state of the world. In 1847, Boole [11] defines a proposition as a true or false sentence that contains a subject and a predicate. He provides the example: "All men are mortal", which contains the subject "men" and the predicate "mortal". It can be symbolically represented by substituting the subject and predicate with classes, e.g., "All Xs are Ys" [11]. Although representations have evolved, for instance, in MacColl [13]'s work this example would be represented as X^Y, the concepts and terms are well established.

In later works, the notion of objects and actions, also called operators, were introduced [14, 15]. In STRIPS [15], an initial world state contains a set of atoms, each representing a proposition. For instance, `(mortal Alice)` is an atom, representing the proposition "Alice is mortal"; `mortal` is the name of a predicate which takes a person as an argument. Actions consist of objects (i.e., arguments), preconditions, delete effects and add effects. Continuing with the example, the `drink(Alice elixir)` action has the precondition `(mortal Alice)`, and will result in the deletion of `(mortal Alice)` and the addition of `(empty elixir)`. The closed world assumption [16] is made, thus everything that is not known to be true is false.

Symbolic actions model the behaviour of non-artificial agents, such as, humans, and artificial agents, such as, software, robots and IoT devices.

Modelling a human's actions help artificial agents to understand and predict the human's behaviour. Moreover, modelling their own actions endows agents with the ability to generate a plan, containing a set of actions that when executed modify the state of the environment. The Planning Domain Definition Language (PDDL), which is based on STRIPS, is now commonly developed to model an agent's behaviour [17–19]. It was originally designed for planning problems but is now also developed to define intention recognition problems.

1.2 Intention Recognition

The word intention is defined as "something that you want and plan to do" [20]. Symbolic-based intention recognisers have attempted to both recognise the goal of an observee (i.e., what the observee wants) and how they intend to achieve that goal (i.e., their plan). Such works are often inspired by concepts from psychology, particularly, the Theory of Mind (ToM) [21]. Agents are said to exhibit ToM capabilities, like humans, if they have the ability to infer knowledge that is not directly observable and exploit it to make predictions [21].

Knowledge of an agent's intentions is essential in numerous application areas. These include computer games in which non-playable characters must adapt to players' actions [22]; intelligent user help for human-computer interaction scenarios [23, 24]; offering humans energy saving advice [25]; robot sports playing (e.g., table tennis [26]); interfering, and thus preventing the intentions of computer network intruders [27, 28]; determining the location a human is navigating to (e.g., for airport security) [29], and proactive robot assistance [30–32].

1.2.1 Goal and Plan Recognition

The terms goal recognition and plan recognition are often use interchangeably. Schmidt et. al. provide the following definition: "plan recognition is to take as input a sequence of actions performed by an actor and to infer the goal pursued by the actor and also to organise the action sequence in terms of a plan structure" [33]. A set of hypothesis goals is usually predefined [34–36]. For instance, within a kitchen environment a human's goal could be to have made breakfast, lunch or dinner; and their plan involves taking different items (e.g., coffee, cup, milk, etc.) and using appliances [34]. If a human's plan is fully known, their goal can be derived; however, as there could exist multiple plans to reach a goal, it is not always possible to determine the human's plan from their goal. For example, when a human is observed taking bread and switching on a toaster, their goal is likely to be to have made breakfast; but whether their plan

will include making tea or making coffee remains unclear. Therefore, we consider goal recognition and plan recognition to be distinct but related challenges.

1.2.2 Goal Recognition Design

A human's goal often cannot be determined until their plan nears completion. This occurs when the plans to reach different goals are initially identical. In other words, the plans contain a non-distinctive prefix. The aim of Goal Recognition Design (GRD) [37–39] is to reduce the length of the non-distinctive plan prefixes by modifying the environment. For instance, in Figure 1.1 an obstacle has been placed to force humans into revealing the location they are navigating to sooner. This enables humans to be more promptly assisted or thwarted.

1.2.3 Action Prediction

Even after performing GRD, it is likely that parts of a human's plan will be predictable before their goal is recognisable. Moreover, GR and GRD assume that a human aims to reach one of the predefined goals; however, a human can perform any sequence of actions. These actions may only partially achieve a predefined goal and could appear in differing goals' plans. For example, when the human takes bread, they will likely want a plate; however, it is unclear whether they will make breakfast or lunch, or neither. Therefore, rather than attempt to recognise the human's entire plan, their next actions should be predicted.

1.3 Task Planning

Task planners search for a set of actions that when executed, transforms the initial world state into a desired goal state. The General Problem-Solving Program [14] was the first program to provide a generic solution to solving a planning problem; however, it could only solve basic problems due to combinatorial explosion. For a problem containing n propositions, there are 2^n states to search [1]. Thus, many researchers have since investigated how to reduce the time spent searching for a plan [19, 40, 41]. Moreover, rather than searching for a linear sequence of actions, partial-order planners were developed [42, 43].

Symbolic logic has its limitations, its representation of the world is often too abstract [44]. Further knowledge of the world is frequently required while planning and the planned actions need translating into executable instructions. For instance, it is difficult to encode continuous numerical information into symbolic logic. As a result, general-purpose planners that can be configured to call domain specific reasoners were de-

veloped [45, 46]. Domain specific reasoning has been performed, whilst searching for a plan, to determine which objects fit into a container [45], to schedule resources [46] and to discover if it is possible for a robot to reach a location [47].

1.4 Robot Planning

One of the first robots to perform symbolic task planning, nicknamed Shakey, was able to navigate and push objects [48, 49]. The software architecture consisted of 5 layers: i) a hardware connection layer, ii) low-level control programs, iii) actions (e.g., go-to) that enable basic behaviour, iv) task planning and v) monitoring. Each layer creates an additional layer of abstraction. This layered concept is still present in many, more recently developed, systems [47, 50]. Robotic planning systems interleave sensing, planning and acting, which is often referred to as continual planning [51, 52].

Although research within this field has greatly progressed, robots that are assisting humans in commercial, industrial and domestic environments are usually preprogrammed to perform a single task. The limitations of the available robotic systems are partly due to hardware constraints. If the robot acts alone and does not have the dexterity to perform certain actions (e.g., open a door), it can only complete a limited range of tasks. Moreover, the robot can only observe the parts of the environment that are within the scope of its onboard sensors.

1.5 Smart Environments

Smart environments are designed to perceive and control the environment; they contain IoT devices that are usually managed by cloud-based services. An IoT device is any object that is connected to the internet. These devices generate enormous amounts of data. Even within a single home there can be many IoT devices, for example, smart TVs, lights, doors, presence sensors, cameras, etc. These devices provide sensor data (e.g., the status of a door) and can been remotely controlled (e.g., a door opened). IoT middleware, such as DYAMAND [53], provides a standardised interface for communicating with heterogeneous IoT devices. Sensor data can subsequently be transformed into propositional representations that can be processed by symbolic task planners.

By integrating robots into smart environments, robots can gain further actuation, sensing and computational capabilities. Actuation and sensing capabilities are provided by IoT devices, and the cloud supplies computational resources. Embedding robots in smart environments is often referred to as the Internet of Robotic Things (IoRT) [8, 54]. This concept has been applied to assisting humans in smart factories [55], nursing homes [56],

and domestic houses [57, 58]. By employing IoT sensors, robots can become more context-aware, and thus gain greater autonomy.

1.6 Research Challenges

To act autonomously a robot requires the ability to recognise intentions; plan and execute actions, and adapt its task execution to unexpected state changes. This PhD aims to mitigate some of the current problems, gaps and disadvantages of applying symbolic AI planning techniques to proactive robot assistance. To frame the research challenges, this section mentions some related works. A more detailed discussion on the related works is provided in the chapter(s) that address the specific challenges. Our specific research questions and hypotheses are provided below.

Methods for intention recognition can be broadly categorised as data-driven and knowledge-driven (i.e., symbolic) methods [9, 10]. Data-driven approaches train a recognition model from a large dataset [10, 59–61]. The main disadvantages of this method are that often a large amount of labelled training data is required and the produced models often only work on data similar to the training set [62, 63]. We therefore opted to take a knowledge-driven approach.

Knowledge-driven approaches rely on a logical description of the actions agents can perform. They can be further divided into approaches that parse a library of plans (also known as "recognition as parsing"), and approaches that solve recognition problems, defined in languages usually associated with planning, i.e., "recognition as planning" [64]. Recognition as parsing tends to be fast and allows multiple concurrent (sub-)plans to be detected [65–67]. Nevertheless, a planning library containing all actions and their orderings must be produced a priori and cyclic plans are difficult to compile into a library [34]. Library-based approaches model plans using either hierarchical structures representing tasks (i.e., abstract actions) being decomposed into sub-tasks [65] or AND-OR trees [66, 68].

Recognition as planning is often viewed as more flexible [34, 69] as problems are expressed through planning languages (e.g., PDDL) rather than via a library of plans. Initial recognition as planning approaches were computationally expensive [34, 69], and although more recent approaches managed to reduce the computational cost [36, 70], recognising the plans of multiple interleaving (sub-)goals remained an open challenge.

Due to these disadvantages and advantages, we aim to combine the benefits of recognition as parsing with those of recognition as planning. This leads us to our first research question:

Research Question 1: *Can a structure similar to those created by library-based approaches be generated from a PDDL defined GR problem?*

The actions of PDDL defined problems contain preconditions and effects. Before an action can be executed, a state which is consistent with its preconditions must be reached by performing other actions. For each action, the actions that set its preconditions will need to be discovered; therefore, we hypothesise that:

Hypothesis 1: *An AND-OR graph structure that models the order constraints between actions can be created from a GR problem defined in PDDL. This process will have a time complexity of $O(n^2)$, where n is the number of actions.*

The already mentioned goal recognition as planning approaches assume that fluent atoms are known and correctly represented within the PDDL defined initial state. This is not always the case. For instance, the initial location of a human or object could be unknown. As these prior approaches are highly dependent of the initial state, their accuracy will deteriorate if the initial state is defined inaccurately. We specify that an inaccurately defined state, is state with incorrect or missing fluent atoms. GR with probabilistic, partially observable state knowledge and stochastic action outcomes has previously been investigated [71–73]; however, these systems require the probability of each state and action outcome to be known, and thus defined within the GR problem. Due to the identified limitation of prior works, we ask the following research question:

Research Question 2: *When the initial state is inaccurately defined, how can a goal recognition approach be prevented from suffering a major loss of accuracy?*

Ramírez and Geffner [34] and Pereira et al. [36] have created a dataset[1] consisting of 6313 GR problems split across 15 domains. This includes a Kitchen domain in which breakfast, lunch and dinner can be made by taking items and performing activities; a Logistics domain for transporting packages, and a Rovers domain for navigating a planet to collect rock samples. We will expand this dataset by creating problems with inaccurately defined initial states. This dataset will enable us to evaluate the following hypothesis on a wide range of domains:

[1] https://github.com/pucrs-automated-planning/goal-plan-recognition-dataset

Hypothesis 2: *By creating a structure that models the actions' order constraints, an observee's goal can be predicted to the same degree of accuracy when differing percentages of fluents have incorrect initial values. When all fluents are correct, this approach will have a recall and precision that is not, on average, worse than a current state of the art approach by* > 0.1. *This will be evaluated on the dataset consisting of 15 domains, and their corresponding problems.*

If the plans of different goals start with the same actions, recognising a human's goal will be challenging. Therefore, we aim to redesign the environment so that a human's goal can be recognised after fewer observations. Prior GRD approaches focused on removing the possibility of performing actions or/and constraining their order [37, 39, 74, 75]. For some domains, such as the Kitchen domain, these types of changes are not feasible; however, it is possible to modify this environment by changing the location of items. We therefore ask the following research question:

Research Question 3: *Can a configuration of an environment (e.g., a kitchen) that enables a observed agent's goal to be recognised after fewer observations be automatically discovered?*

In the original Kitchen domain created by Ramírez and Geffner [34], items do not have locations. We will expand this domain so that each item is in a cupboard, fridge or draw. This domain will be provided as input to our GRD approach. Our hypothesis is:

Hypothesis 3: *Modifications can be applied to a structure that addresses Research Question 1, to find a configuration that makes the goals within the Kitchen domain more distinctive by at least 1 action.*

Recognition as planning and GRD approaches assume the observed agent aims to reach one of the predefined hypothesis goals. This is not always the case. A human could perform a sequence of actions that leads to any reachable state. For instance in the Kitchen domain, rather than making breakfast a human could could just make a cup of tea; moreover, they could interleave the actions involved with making a cup of tea with those needed to make a cheese sandwich. Therefore, the human's next actions should be predicted.

Furthermore, knowledge of the human's next actions, could enable a robot to provide assistance. Applying recognition as planning to proactive robot

assistance was previously investigated by Freedman & Zilberstein [30] and Levine & Williams [31, 76]. Both these works focus on an agent performing a single goal. Moreover, the work by Freedman & Zilberstein calls the computationally expensive GR approach of Ramírez and Geffner [34]. Based on these challenges, our fourth research question is:

Research Question 4: *When provided with a PDDL defined GR problem as input, how can an observed agent's next actions be predicted and used by a robot to provide assistance?*

Library-based approaches to GR enable multiple interleaving subtasks to be recognised; therefore, the structure created to test Hypothesis 1 will act as a starting point for answering this question. We will investigate labelling each action with a value that increases when actions it is connected to are observed. After an observation has been processed, the action nodes whose values fall above a manually defined threshold can be selected as the predicted actions.

Experiments with the Kitchen domain by Ramírez and Geffner [34] will be performed. We chose this domain to prevent us from creating a domain that is designed to demonstrate only the advantages of our approach. This domain is also inline with ongoing research within our department [58]. We hypothesise that:

Hypothesis 4: *The process of updating the node values will have a worst case time complexity of $O(n)$, where n is the number of nodes. Moreover, when the highest valid predicted actions are executed by a robot, a simulated observed agent performs fewer actions.*

Robots often lack the capabilities required to perform certain tasks, for instance, the dexterity to open doors. By coupling a robot to a smart environment, the robot can gain state information that has been detected by IoT sensors and invoke the capabilities of IoT actuators. When preforming symbolic task planning, which device (or robot) will execute an action tends to be explicitly stated within the plan [77, 78]. As the number of IoT devices installed in smart environments is rising, adding all IoT devices explicitly to a planning problem greatly increases the planning time.

Moreover, symbolic planning languages (e.g., PDDL) often lack the expressively required to model the real-world. Therefore, several researches have investigated calling external reasoners during the search for a task

plan [45, 79]. We will investigate invoking an external reasoner in order to gain knowledge about if there is a device capable of executing an action available. Unfortunately, as external module calls increase the planning time it can be difficult to balance determining the correct state with keeping computational costs low [52].

Furthermore, when a state change that affects the robot's plan occurs, the robot must adapted its plan. As a robot coupled to a smart environment will receive many state updates, it may be forced to frequently replan. Due to these challenges, we formulated the following research question:

Research Question 5: *How can the state information and exogenous actuation capabilities of IoT devices be incorporated into a robot's continual planner without greatly increasing the computational cost and the frequency of (re-)planning?*

We aim to develop a framework in which IoT devices can be represented as a single object within a PDDL planning problem. The robot will leverage on a cloud-backend in order to reason about and communicate with IoT devices. By taking a hierarchical planning in the now approach in which only a subsection of the plan will be planned in detail, we aim to reduce the frequency of replanning.

The framework we develop will be tested with a Pepper (humanoid) robot within our department's smart house[2]. Moreover, we will evaluate the performance of the framework on a simulated coffee fetching scenario within a smart office environment. A comparison between our work and performing all planning upfront with all IoT devices represented in the PDDL problem will be preformed. Taking into account that the PDDL defined initial state will need to be generated, the amount of information that is likely to be provided to a task planner and the planning times reported by Dornhege et al. [45], we hypothesis that:

Hypothesis 5: *Our framework will enable a robot to start acting within 2 seconds of being provided with the goal for a simulated coffee fetching scenario. When the availability of different coffee machines changes, the robot will be less likely to have to replan than when everything is planned upfront.*

To validate the previously mentioned hypotheses, the PDDL defined actions of the observed agent and the robot will be manually written in PDDL.

[2]https://www.imec-int.com/en/homelab

Developing PDDL can be a time consuming and error prone process. Therefore, researchers have attempted to learn PDDL from pairs of images, showing the environment's state before and after an action has been executed. LatPlan [80, 81] employed a deep autoencoder to generate the action definitions, and reported that it took 4 hours for a task planner to generate a plan from the produced PDDL. A more compact representation is required to reduce the planning time. As a black-box approach was developed, the decisions made by their algorithms are likely to be unexplainable. Due to these disadvantages, our final research question is:

Research Question 6: *Can a more compact set of symbolic action definitions be learnt from pairs of images by taking a explainable (white-box) approach?*

An image simply contains (x,y) coordinates, i.e., pixels, and values. We will therefore investigate how pixels and values can be grouped together and assigned symbolic names (IDs) to form a set of PDDL objects. The value a location transitions from/to within an image pair, will form the starting point for an action's preconditions and effects. Further preconditions will be required to prevent invalid grounded actions being generated from the produced action definitions.

Asai and Fukunaga [81] generated the image pairs of Puzzle, Lights-Out and Towers-of-Hanoi domains (see Chapter 6 for a description) to evaluate their approach. As these datasets contain many pairs of images and planning with a manually created version of these domains would take very little time, we hypothesises that:

Hypothesis 6: *Generating action definitions will have a large computational cost but the produced PDDL will enable a task planner to find plans in less than 1 second for Puzzle, Lights-Out and Towers-of-Hanoi problems.*

1.7 Research Contributions

This doctoral thesis focuses on three core topics: intention recognition, robot assistance and the learning of symbolic action definitions. Each of these topics is key for enabling and improving proactive robot assistance. This section summarises the contributions, then introduces several of the assumptions and limitations of our work.

Intention Recognition *(Chapters 2, 3 and 4):*

We propose an Action Graph-based approach to three elements of

intention recognition: Goal Recognition (GR), Goal Recognition Design (GRD) and action prediction. An Action Graph models the dependencies (i.e., order constraints) between actions and is generated from a symbolically defined problem. The construction and usage of the graph is different for each of these elements.

Prior approaches to symbolic GR assumed that the initial world state is fully known, and thus is accurately represented. To remove this assumption, Action Graphs are constructed without considering the initial value of fluent atoms. When an action is observed, each hypothesis goal's probability is updated based on how far the observed action is from an action that achieves that goal. The goals with the equally highest probability are returned as the set of candidate goals.

The aim of GRD is to reduce the number of observations required to determine an observed agent's goal. This is achieved by modifying the initial state of the environment. The goal distinctiveness metrics calculated by prior approaches do not take into consideration that the total number of actions required to reach the goals, as well as the number in the non-distinctive plan prefix, could be altered during this process. We therefore introduce a new metric. To calculate and reduce this metric an Action Graph is generated by means of a Breadth First Search (BFS), which traverses backwards from each goal state to the initial state. To replace actions, we developed an exhaustive approach and an approach that shrinks the plans then reduces the non-distinctive plan prefixes, namely Shrink-Reduce. Exhaustive is guaranteed to find the minimal distinctiveness but is more computationally expensive than Shrink-Reduce.

Rather than completing a single predefined goal, a human could perform any sequence of actions. Therefore, we developed an approach that predicts a human's next actions. Each time an action is observed, the Action Graph's node values are updated and the action nodes with an above threshold value are extracted. These extracted actions form the set of predicted actions. Experiments with a Kitchen domain demonstrated that our approach adequately predicts actions, even when the plans of multiple interleaving subgoals are observed.

Robot Assistance *(Chapters 4 and 5):*

To enable proactive robot assistance, our system maps the predicted actions to a goal state for the robot. The robot then generates a plan that achieves this goal. It only executes its plan if it does not impact the flow of the human by obstructing or delaying them. A simulated

proof of concept demonstrated that a robot is able to successfully assist a person by executing actions on their behalf.

Whilst a robot is acting it can be assisted by the sensors and actuators of a smart environment. Nevertheless, incorporating the states and actions of IoT devices into the robot's onboard planner increases the computational load, and thus can delay the execution of a task. Moreover, IoT sensors enable more of the unanticipated actions of humans to be detected, and thus increase how frequently the robot must replan. Therefore, we developed a Hierarchical Continual Planning in the Now framework to mitigate these inadequacies. IoT state knowledge is monitored by a cloud-based service and the robot is only informed about state relevant to its current task plan. Tasks are split-up into subtasks and the first subtask is planned and executed before the subsequent one. This enables the robot to start acting sooner and the frequency of (re)planning to be reduced. The effectiveness of our framework was demonstrated by a test in a smart home and a simulator-based evaluation.

Learning Symbolic Action Definitions *(Chapter 6):*

To evaluate our first contributions the symbolic definitions of the agents' actions were manually developed. This can be a time consuming and error prone process. Therefore, we implemented an approach that transforms unlabelled pairs of images into symbolic action definitions. To evaluate these action definitions a task planner was invoked. Problems with large state spaces were solved using the action definitions learnt from smaller state spaces. Our approach was also evaluated on a dataset with varying amounts of missing image pairs.

Our goal recogniser and action predictor process observations. These observations are discrete symbolically represented actions. For example, the `move(0_0 1_0)` action represents that the observed agent has moved from grid location `(0,0)` to grid location `(1,0)`. Our work assumes that sensor data has already been transformed into symbolic actions. This transformation can be performed by mapping sensor data to symbols and, in some case, will require activity and object recognition [82]. The deployment of hardware, i.e., IoT devices and robots, and the integration of activity recognition algorithms are beyond the scope of our chapters on intention recognition and proactive assistance.

To recognise intentions and plan actions our work requires a symbolic representation of an agent's actions, along with their preconditions and effects. This symbolic representation must be created upfront. Moreover,

our continual planning framework requires the robot's and IoT devices' capabilities to be defined within an ontology.

To learn action definitions, pairs of images that show the state of the environment before and after an action has been executed are processed. These image pairs are created by a script, and thus contain a perfect representation of the environment's state. The learnt action definition can be utilised to generate a task plan, but currently this plan cannot be transformed into executable robot instructions.

1.8 Publications

The research produced during this doctoral thesis has been published in scientific journals, and presented at international conferences and workshops. These publications are listed below.

1.8.1 Journal Papers

[1] **H. Harman**, P. Simoens, *Action Graphs for Performing Goal Recognition Design on Human-Inhabited Environments*. Published in Sensors, 19 (12): 2741, 2019.

[2] **H. Harman**, K. Chintamani, P. Simoens, *Robot Assistance in Dynamic Smart Environments–A Hierarchical Continual Planning in the Now Framework*. Published in Sensors, 19 (22): 4856, 2019.

[3] **H. Harman**, P. Simoens, *Action Graphs for Proactive Robot Assistance in Smart Environments*. Published in Journal of Ambient Intelligence and Smart Environments, 12 (2), 2020.

[4] **H. Harman**, P. Simoens, *Cyclic Action Graphs for Goal Recognition Problems with Inaccurately Initialised Fluents*. Submitted to Knowledge and Information Systems, 2020.

1.8.2 Conference and Workshop Papers

[1] **H. Harman**, K. Chintamani, P. Simoens, *Architecture for Incorporating Internet-of-Things Sensors and Actuators into Robot Task Planning in Dynamic Environments*. Published in IEEE International Symposium on Robotics and Intelligent Sensors (IRIS), Ottawa, Canada , 2017.

[2] **H. Harman**, K. Chintamani, P. Simoens, *Action Trees for Scalable Goal Recognition in Robotic Applications*. Published in the Sixth ICAPS Workshop on Planning and Robotics (PlanRob), Delft, Netherlands, 2018.

[3] **H. Harman**, P. Simoens, *Solving Navigation-based Goal Recognition Design Problems with Action Graphs*. Published in AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR), Honolulu, USA, 2019.

[4] **H. Harman**, P. Simoens, *Generating Symbolic Action Definitions from Pairs of Images: Applied to Solving Towers of Hanoi*. Published in AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR), New York, USA, 2020.

[5] **H. Harman**, P. Simoens, *Action Graphs for Goal Recognition Problems with Inaccurate Initial States (Student Abstract)*. Published in Thirty-First AAAI Conference on Artificial Intelligence, New York, USA,

2020.

[6] **H. Harman**, P. Simoens, *Learning Symbolic Action Definitions from Unlabelled Image Pairs*. Accepted in International Conference on Advancements in Artificial Intelligence (ICAAI), London, UK, 2020.

References

[1] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Malaysia; Pearson Education Limited, 2016.

[2] A. Turing. *Computing Machinery and Intelligence*. Mind, 59(236):433, 1950.

[3] S. M. G. V. McKinney, S.M. et al. *International Evaluation of an AI System for Breast Cancer Screening*. Nature, 577:89—94, 2020.

[4] A. S. Lokman and M. A. Ameedeen. *Modern Chatbot Systems: A Technical Review*. In Proceedings of the Future Technologies Conference, FTC'18, pages 1012–1023, Cham, 2018. Springer International Publishing.

[5] L. Yao, N. Peng, R. Weischedel, K. Knight, D. Zhao, and R. Yan. *Plan-and-Write: Towards Better Automatic Storytelling*. In Proceedings of the AAAI Conference on Artificial Intelligence, AAAI'19, pages 7378–7385, 2019.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. *A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play*. Science, 362(6419):1140–1144, 2018.

[7] J. V. Brummelen, M. O'Brien, D. Gruyer, and H. Najjaran. *Autonomous Vehicle Perception: The Technology of Today and Tomorrow*. Transp. Res. Pt. C-Emerg. Technol., 89:384 – 406, 2018.

[8] P. Simoens, M. Dragone, and A. Saffiotti. *The Internet of Robotic Things: A Review of Concept, Added Value and Applications*. Int. J. Adv. Robot. Syst., 15(1), 2018.

[9] J. Rafferty, C. D. Nugent, J. Liu, and L. Chen. *From Activity Recognition to Intention Recognition for Assisted Living Within Smart Homes*. IEEE T. Hum.-Mach. Syst., 47(3):368–379, 2017.

[10] K. Yordanova, S. Lüdtke, S. Whitehouse, F. Krüger, A. Paiement, M. Mirmehdi, I. Craddock, and T. Kirste. *Analysing Cooking Behaviour in Home Settings: Towards Health Monitoring*. Sensors, 19(3), 2019.

[11] G. Boole. *The Mathematical Analysis of Logic*. Philosophical Library, 1847.

[12] J. McCarthy. *Concepts of Logical AI*, pages 37–56. Springer US, Boston, MA, 2000.

[13] H. MacColl. *Symbolic Logic and Its Applications.* Longmans, Green, 1906.

[14] A. Newell, J. C. Shaw, and H. A. Simon. *Report on a General Problem-Solving Program*. In Proceedings of the International Conference on Information Processing, pages 256–264, 1959.

[15] R. E. Fikes and N. J. Nilsson. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Artif. Intell., 2(3-4):189–208, 1971.

[16] R. Reiter. *On Closed World Data Bases*, pages 55–76. Springer US, Boston, MA, 1978.

[17] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. *PDDL - The Planning Domain Definition Language*. 1998.

[18] A. Coles, A. Coles, A. García Olaya, S. Jiménez, C. Linares López, S. Sanner, and S. Yoon. *A Survey of the Seventh International Planning Competition*. AI Mag., 33(1):83–88, 2012.

[19] M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, S. Sanner, and M. Editor. *The 2014 International Planning Competition: Progress and Trends*. AI Mag., 36(3):90–98, 2015.

[20] *INTENTION: Meaning in the Cambridge English Dictionary*. Accessed: 2019-12-24. Available from: https://dictionary.cambridge.org/dictionary/english/intention.

[21] D. Premack and G. Woodruff. *Does the Chimpanzee Have a Theory of Mind?* Behav. Brain Sci., 1(4):515–526, 1978.

[22] M. Fagan and P. Cunningham. *Case-Based Plan Recognition in Computer Games*. In International Conference on Case-Based Reasoning Research and Development, pages 161–170, Berlin, Heidelberg, 2003. Springer.

[23] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. *The LumièRe Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users*. In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, UAI'98, pages 256–265, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[24] J. Hong. *Goal Recognition Through Goal Graph Analysis*. J. Artif. Intell. Res., 15:1–30, 2001.

[25] W. S. Lima, E. Souto, T. Rocha, R. W. Pazzi, and F. Pramudianto. *User Activity Recognition for Energy Saving in Smart Home Environment*. In IEEE Symposium on Computers and Communication, ISCC, pages 751–757. IEEE, 2015.

[26] Z. Wang, A. Boularias, K. Mülling, B. Schölkopf, and J. Peters. *Anticipatory Action Selection for Human–Robot Table Tennis*. Artif. Intell., 247:399 – 414, 2017. Special Issue on AI and Robotics.

[27] C. W. Geib and R. P. Goldman. *Plan Recognition in Intrusion Detection Systems*. In Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01, volume 1, pages 46–55. IEEE, 2001.

[28] R. Mirsky, Y. Shalom, A. Majadly, K. Gal, R. Puzis, and A. Felner. *New Goal Recognition Algorithms Using Attack Graphs*. In Cyber Security Cryptography and Machine Learning, pages 260–278, Cham, 2019. Springer International Publishing.

[29] P. Masters and S. Sardina. *Cost-Based Goal Recognition in Navigational Domains*. J. Artif. Intell. Res., 64:197–242, 2019.

[30] R. G. Freedman and S. Zilberstein. *Integration of Planning with Recognition for Responsive Interaction Using Classical Planners*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 4581–4588. AAAI Press, 2017.

[31] S. J. Levine and B. C. Williams. *Watching and Acting Together: Concurrent Plan Recognition and Adaptation for Human-Robot Teams*. J. Artif. Intell. Res., 63:281–359, 2018.

[32] S. Lemaignan, M. Warnier, E. A. Sisbot, A. Clodic, and R. Alami. *Artificial Cognition for Social Human–Robot Interaction: An Implementation*. Artif. Intell., 247:45 – 69, 2017. Special Issue on AI and Robotics.

[33] C. Schmidt, N. Sridharan, and J. Goodson. *The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence*. Artif. Intell., 11(1):45 – 83, 1978. Special Issue on Applications to the Sciences and Medicine.

[34] M. Ramírez and H. Geffner. *Probabilistic Plan Recognition Using Off-the-Shelf Classical Planners*. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI'10, pages 1121–1126. AAAI Press, 2010.

[35] S. Sohrabi, A. V. Riabov, and O. Udrea. *Plan Recognition As Planning Revisited*. In Proceedings of the Twenty-Fifth International Joint Con-

ference on Artificial Intelligence, IJCAI'16, pages 3258–3264. AAAI Press, 2016.

[36] R. F. Pereira, N. Oren, and F. Meneguzzi. *Landmark-Based Heuristics for Goal Recognition*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 3622–3628. AAAI Press, 2017.

[37] S. Keren, A. Gal, and E. Karpas. *Goal Recognition Design*. In Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS, pages 154–162. AAAI Press, 2014.

[38] T. C. Son, O. Sabuncu, C. Schulz-Hanke, T. Schaub, and W. Yeoh. *Solving Goal Recognition Design Using ASP*. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI'16, pages 3181–3187. AAAI Press, 2016.

[39] R. Mirsky, K. Gal, R. Stern, and M. Kalech. *Goal and Plan Recognition Design for Plan Libraries*. ACM Trans. Intell. Syst. Technol., 10(2):14:1–14:23, 2019.

[40] A. Coles, A. Coles, A. García Olaya, S. Jiménez, C. Linares López, S. Sanner, and S. Yoon. *A Survey of the Seventh International Planning Competition*. AI Mag., 33(1):83–88, 2012.

[41] C. L. López, S. J. Celorrio, and Ángel García Olaya. *The Deterministic Part of the Seventh International Planning Competition*. Artif. Intell., 223:82 – 119, 2015.

[42] E. D. Sacerdoti. *The Nonlinear Nature of Plans*. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, volume 1 of *IJCAI'75*, pages 206–214, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

[43] A. Tate. *Generating Project Networks*. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, volume 2 of *IJCAI'77*, pages 888–893, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc.

[44] D. McDermott. *Robot Planning*. AI Mag., 13(2):55–79, 1992.

[45] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel. *Semantic Attachments for Domain-independent Planning Systems*. In Nineteenth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'09, pages 114–121. AAAI Press, 2009.

[46] K. Myers, S. F. Smith, D. W. Hildum, P. A. Jarvis, and R. de Lacaze. *Integrating Planning and Scheduling Through Adaptation of Resource Intensity Estimates*. In Sixth European Conference on Planning, 2014.

[47] D. Speck, C. Dornhege, and W. Burgard. *Shakey 2016 - How Much Does it Take to Redo Shakey the Robot?* IEEE Robotics and Automation Letters, 2(2):1203–1209, 2017.

[48] B. Raphael, R. O. Duda, R. E. Fikes, P. E. Hart, N. J. Nilsson, P. W. Thorndyke, and B. M. Wilber. *Research and Applications - Artificial Intelligence*. Technical report, Stanford Research Institute, 1971. Project 8973 Final Report From the Nilsson archives – SHAKEY papers.

[49] N. J. Nilsson. *Shakey The Robot*. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, 1984.

[50] M. Shanahan. *Reinventing Shakey*, pages 233–253. Springer US, Boston, MA, 2000.

[51] M. E. desJardins, E. H. Durfee, C. L. Ortiz, Jr., and M. J. Wolverton. *A Survey of Research in Distributed, Continual Planning*. AI Mag., 20(4):13, 1999.

[52] C. Dornhege and A. Hertle. *Integrated Symbolic Planning in the Tidyup-Robot Project.* In AAAI Spring Symposium: Designing Intelligent Robots, 2013.

[53] J. Nelis, T. Verschueren, D. Verslype, and C. Develder. *DYAMAND: DYnamic, Adaptive MAnagement of Networks and Devices*. In 37th Annual IEEE Conference on Local Computer Networks, pages 192–195, 2012.

[54] O. Vermesan, A. Bröring, E. Tragos, M. Serrano, D. Bacciu, S. Chessa, C. Gallicchio, A. Micheli, M. Dragone, A. Saffiotti, P. Simoens, F. Cavallo, and R. Bahr. *Internet of Robotic Things : Converging Sensing/Actuating, Hypoconnectivity, Artificial Intelligence and IoT Platforms*. In Cognitive Hyperconnected Digital Transformation : Internet of Things Intelligence Evolution, pages 97–155. River Publishers, 2017.

[55] J. Wan, S. Tang, Q. Hua, D. Li, C. Liu, and J. Lloret. *Context-Aware Cloud Robotics for Material Handling in Cognitive Industrial Internet of Things*. IEEE Internet Things J., 5(4):2272–2281, 2018.

[56] J. Nauta, C. Mahieu, C. Michiels, F. Ongenae, F. D. Backere, F. D. Turck, Y. Khaluf, and P. Simoens. *Pro-active Positioning of a Social Robot Intervening Upon Behavioral Disturbances of Persons with Dementia in a Smart Nursing Home*. Cogn. Syst. Res., 57:160 – 174, 2019.

[57] G. Amato, D. Bacciu, M. Broxvall, S. Chessa, S. Coleman, M. Di Rocco, M. Dragone, C. Gallicchio, C. Gennaro, H. Lozano, T. M. McGinnity, A. Micheli, A. K. Ray, A. Renteria, A. Saffiotti, D. Swords, C. Vairo, and P. Vance. *Robotic Ubiquitous Cognitive Ecology for Smart Homes*. J. Intell. Robot. Syst., 80(1):57–81, 2015.

[58] C. Mahieu, F. Ongenae, F. D. Backere, P. Bonte, F. D. Turck, and P. Simoens. *Semantics-Based Platform for Context-Aware and Personalized Robot Interaction in the Internet of Robotic Things*. J. Syst. Softw., 149:138 – 157, 2019.

[59] L. Amado, R. F. Pereira, J. Aires, M. Magnaguagno, R. Granada, and F. Meneguzzi. *Goal Recognition in Latent Space*. In International Joint Conference on Neural Networks, IJCNN, pages 1–8. IEEE, 2018.

[60] F. Bisson, H. Larochelle, and F. Kabanza. *Using a Recursive Neural Network to Learn an Agent's Decision Model for Plan Recognition*. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI'15, pages 918–924. AAAI Press, 2015.

[61] G. Singla, D. J. Cook, and M. Schmitter-Edgecombe. *Recognizing Independent and Joint Activities Among Multiple Residents in Smart Environments*. J. Ambient Intell. Humaniz. Comput., 1(1):57–63, 2010.

[62] P. C. Roy, S. Giroux, B. Bouchard, A. Bouzouane, C. Phua, A. Tolstikov, and J. Biswas. *A Possibilistic Approach for Activity Recognition in Smart Homes for Cognitive Assistance to Alzheimer's Patients*. In Activity Recognition in Pervasive Intelligent Environments, volume 4, pages 33–58, Paris, 2011. Atlantis Press.

[63] K. Yordanova, F. Krüger, and T. Kirste. *Context Aware Approach for Activity Recognition Based on Precondition-Effect Rules*. In IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops, pages 602–607. IEEE, 2012.

[64] S. Keren, R. Mirsky, and C. Geib. *Plan Activity and Intent Recognition Tutorial*, 2019. Available from: http://www.planrec.org/Tutorial/Resources_files/pair-tutorial.pdf.

[65] H. A. Kautz and J. F. Allen. *Generalized Plan Recognition*. In Proceedings of the Fifth AAAI National Conference on Artificial Intelligence, AAAI'86, pages 32–37. AAAI Press, 1986.

[66] C. W. Geib and R. P. Goldman. *A Probabilistic Plan Recognition Algorithm Based on Plan Tree Grammars*. Artif. Intell., 173(11):1101 – 1132, 2009.

[67] R. Mirsky, Y. K. Gal, and S. M. Shieber. *CRADLE: An Online Plan Recognition Algorithm for Exploratory Domains*. ACM Trans. Intell. Syst. Technol., 8(3), 2017.

[68] S. Holtzen, Y. Zhao, T. Gao, J. B. Tenenbaum, and S. Zhu. *Inferring Human Intent From Video by Sampling Hierarchical Plans*. In IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, pages 1489–1496. IEEE, 2016.

[69] M. Ramírez and H. Geffner. *Plan Recognition As Planning*. In Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence, IJCAI'09, pages 1778–1783, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[70] Y. E-Martin, M. D. R-Moreno, and D. E. Smith. *A Fast Goal Recognition Technique Based on Interaction Estimates*. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI'15, Buenos, Aires, Argentina, 2015. AAAI Press.

[71] P. Jiao, K. Xu, S. Yue, X. Wei, and L. Sun. *A Decentralized Partially Observable Markov Decision Model with Action Duration for Goal Recognition in Real Time Strategy Games*. Discrete Dyn. Nat. Soc., 2017.

[72] M. Ramírez and H. Geffner. *Goal Recognition over POMDPs: Inferring the Intention of a POMDP Agent*. In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, IJCAI'11, pages 2009–2014. AAAI Press, 2011.

[73] S. Yue, K. Yordanova, F. Krüger, T. Kirste, and Y. Zha. *A Decentralized Partially Observable Decision Model for Recognizing the Multiagent Goal in Simulation Systems*. Discrete Dyn. Nat. Soc., 2016.

[74] S. Keren, A. Gal, and E. Karpas. *Strong Stubborn Sets for Efficient Goal Recognition Design.* In Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS, pages 141–149. AAAI Press, 2018.

[75] C. Wayllace, P. Hou, and W. Yeoh. *New Metrics and Algorithms for Stochastic Goal Recognition Design Problems*. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI'17, pages 4455–4462. AAAI Press, 2017.

[76] S. J. Levine and B. C. Williams. *Concurrent Plan Recognition and Execution for Human-Robot Teams*. In Proceedings of the Twenty-Fourth International Conference on International Conference on Automated

Planning and Scheduling (ICAPS), ICAPS'14, pages 490–498. AAAI Press, AAAI Press, 2014.

[77] D. L. Kovacs. *A Multi-agent Extension of PDDL3.1*. In Proceedings of the Third Workshop on the International Planning Competition (IPC), pages 19–27, 2012.

[78] M. Brenner and B. Nebel. *Continual Planning and Acting in Dynamic Multiagent Environments*. Auton. Agents Multi-Agent Syst., 19(3):297–331, 2009.

[79] J. Buehler and M. Pagnucco. *A Framework for Task Planning in Heterogeneous Multi Robot Systems Based on Robot Capabilities*. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, AAAI'14, pages 2527–2533. AAAI Press, 2014.

[80] M. Asai and A. Fukunaga. *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary*. arXiv preprint arXiv:1705.00154, 2017.

[81] M. Asai and A. Fukunaga. *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary*. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, AAAI'18. AAAI Press, 2018.

[82] S. Tremblay, D. Fortin-Simard, E. Blackburn-Verreault, S. Gaboury, B. Bouchard, and A. Bouzouane. *Exploiting Environmental Sounds for Activity Recognition in Smart Homes.* In AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments, 2015.

2

Goal Recognition

"*Computers are good at following instructions, but not at reading your mind.*"
Donald E. Knuth (1984)

Cyclic Action Graphs for Goal Recognition Problems with Inaccurately Initialised Fluents

H. Harman & P. Simoens
Submitted to Knowledge and Information Systems

Goal recognisers attempt to infer an agent's intentions from a sequence of observed actions. This is an important component of intelligent systems that aim to assist or thwart actors; however, there are many challenges to overcome. For example, the initial state of the environment could be partially unknown, agents can act suboptimally and observations could be missing. Approaches that adapt classical planning techniques to goal recognition have previously been proposed but, generally, they assume the initial world state is accurately defined. In this chapter, a state is inaccurate if any fluent's value is unknown or incorrect. Our aim is to develop a goal recognition approach that is as accurate as the current state of the art algorithms and whose accuracy does not deteriorate when the initial state is inaccurately defined. To cope with this complication, we propose solving goal recognition problems by means of an Action Graph. An Action Graph models the dependencies, i.e., order constraints, between all actions rather than just actions within a plan. Leaf nodes correspond to actions and are connected to their dependencies via operator nodes. After generating an Action Graph, the graph's nodes are labelled with their distance from each hypothesis goal. This distance is based on the number and type of nodes traversed to reach the node in question from an action node that results in the goal state being reached. For each observation, the goal probabilities are then updated based on either the distance the observed action's node is from each goal or the change in distance. Our experimental results, for 15 different domains, demonstrate that our approach is robust to inaccuracies within the defined initial state.

2.1 Introduction

By observing the behaviour of an agent, artificially intelligent systems can attempt to determine the agent's intentions. Knowledge of an agent's intentions is essential in numerous application areas. These include computer games in which non-playable characters must adapt to players' actions [1]; intelligent user help for human-computer interaction scenarios [2, 3]; offering humans energy saving advice [4]; robot sports playing (e.g., table tennis [5]); interfering (and thus preventing) the intentions of computer network intruders [6, 7]; determine the location a human is navigating to (e.g., for airport security) [8], and to enable proactive robot assis-

tance [9–11]. Rather than developing domain specific intention recognition algorithms, a symbolic representation of the world and agents' actions can be provided as input to non-domain specific algorithms [12, 13].

Intention recognition can be split into several categories, namely, activity recognition [14–16], plan recognition [6, 17, 18], and goal recognition [19, 20]. Our work falls under the category of goal recognition (GR), in which the aim is to label a sequence of observations (e.g., actions) with which goal the observee is attempting to reach. For instance, when provided with a sequence of move actions, GR methods will attempt to select (from a predefined list) which location the agent is intending to reach. For the Kitchen domain by Ramírez and Geffner [20], the sequence of observed actions includes taking different items and using appliances (e.g., a toaster), and the returned classification indicates if the observee is likely to be making breakfast, dinner or a packed lunch. Goal and plan recognisers operate on discrete observations/actions, and thus assume that data streams have been preprocessed, e.g., sensor data have been processed by activity recognisers.

Our GR method aims to overcome several challenges. First, the defined initial world state could be inaccurate; for instance, if an item or agent (e.g., cup or human) is occluded, its location is indeterminable, and thus possibly defined incorrectly. Second, the observed agent could act suboptimally [20]; therefore, all plans (including suboptimal plans) are represented within the underlying structure generated by our approach. Third, actions could be missing from the sequence of observations [21], e.g., due to sensor malfunction or occlusions. Finally, an observation should be rapidly processed, so there is little delay in determining the observee's goal. The cited GR works have investigated handling suboptimal observation sequences and handling missing observations, but they do not consider inaccurate initial states.

We define the term inaccurate initial state as an initial state containing fluents (i.e., non-static variables) whose value is unknown (i.e., undefined) and/or incorrect (i.e., set to the wrong value). Inaccurate initial states have been handled by task planners [22, 23]. Moreover, GR with probabilistic, partially observable state knowledge and stochastic action outcomes has previously been investigated [24–26]; however, these systems require the probability of each state and action outcome to be known (and thus defined within the GR problem). GR with incomplete domain models, i.e., problems containing actions with incomplete preconditions and effects, have also been considered [27], but the initial state was assumed to be accurately represented. Our system makes no assumptions about the correctness of

the initial value assigned to a fluent.

In this chapter, we aim to answer two of the research questions that were presented in the introductory chapter. i) Can a structure similar to those created by library-based approaches be generated from a PDDL defined GR problem? ii) When the initial state is inaccurately defined, how can a goal recognition approach be prevented from suffering a major loss of accuracy?

To answer these questions, we investigate solving GR problems by means of an Action Graph, which models the order constraints between actions. Leaf nodes correspond to actions and are connected to their dependencies via operator nodes. Operator nodes include `DEP` (short for dependencies), `ORDERED-AND`, `UNORDERED-AND` and `OR` nodes. After transforming the action definitions and world model into an Action Graph, the Action Graph's nodes are labelled with their distance from each hypothesis goal, i.e., each goal the observee could be intending to achieve. Both these processes are performed offline. For each observation, the online process updates the goal probabilities based on either the distance the observed action's node is from each goal or the change in distance. Our distance measure is based on the number and type of nodes traversed to reach the node in question from an action node that results in the goal state being reached. The goal(s) with the highest probability are returned as the set of candidate, i.e., predicted, goals. An Action Graph does not contain a perfect representation of all plans; as mentioned by Pereira et al. [21], unlike task planning, this is not a requirement of GR. A conceptual overview of our system is provided in Figure 2.1.

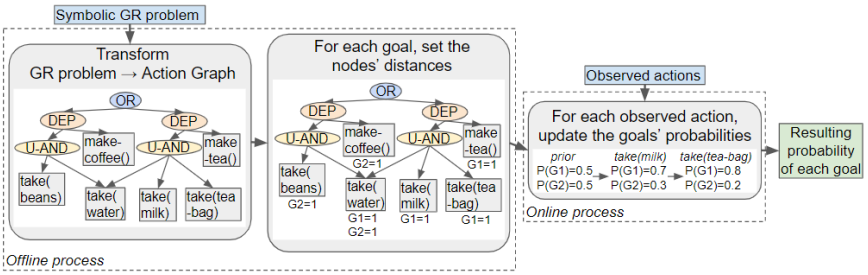


Figure 2.1: Conceptual overview of the goal recognition process described in this chapter.

The remainder of this chapter is structured as follows. Section 2.2 presents some background information. A formal definition of our Action Graph structure is provided in Section 2.3, and Section 2.4 describes the algorithm

that generates the Action Graph. Section 2.5 introduces our distance measure and how the nodes are labelled with their distance from each goal. The different goal probability update rules, that are executed when an observation is received, are described in Section 2.6. Our experimental results, discussed in Section 2.7, show that our GR method is unaffected by inaccuracies in the initial state.

2.2 Background

Symbolic task planning and goal recognition problems are often defined in Planning Domain Definition Language (PDDL) [28], a popular domain-independent language for modelling the behaviour of deterministic agents. A PDDL defined problem includes action definitions, objects, predicates and an initial state; an example of each is provided in Listing 2.1. Our GR approach transforms a PDDL problem into a multi-valued problem by running the converter of [29]. The multi-valued problem is then transformed into an Action Graph. This section first describes why a multi-valued representation is used. The task planning and GR (also known as inverse planning) problem definitions are provided, in the subsections, from a multi-valued problem perspective.

Listing 2.1: Example of a PDDL defined action, set of objects, set of predicates and initial state. Based on the International Planning Competition's (IPC's) Easy-IPC-Grid domain[1].

```
# Example action definition:
(:action move  :parameters (?1 ?2 - position)
  :precondition (and (at ?1) (not (at ?2)) (adjacent ?1 ?2) )
  :effect (and (at ?2) (not (at ?1)) )
)
# Example set of objects:
(:objects 1_1 1_2 - position)
# Example predicates:
(:predicates (at ?1 - position) (adjacent ?1 ?2 - position))
# Example initial state:
(:init (at 1_1) (adjacent 1_1 1_2) (adjacent 1_2 1_1) )
```

To create a concise, grounded representation of a problem, a PDDL defined problem is often converted into a multi-valued representation [29, 30]. This representation uses finite variables rather than boolean propositions. For example, rather than a `move(1_1 1_2)` action (which symbolises an agent moving from grid position `1_1` to `1_2`) removing the proposition `(at 1_1)` from the current state and inserting the proposition `(at 1_2)`, a variable, i.e., fluent, that represents the agent's location is changed from `(at 1_1)` to `(at 1_2)`. This enables a more concise representation of the problem to be produced, from which the relations between the different propositions

[1] http://www.icaps-conference.org/index.php/Main/Competitions

can be extracted. Moreover, a (grounded) action is only created, from an action definition, if its static preconditions appear in the PDDL defined initial world state. For example, to create the `move(1_1 1_2)` action, positions `1_1` and `1_2` must be adjacent. Which locations are adjacent can be statically defined; in other words, no action modifies which locations are adjacent. Further details on the benefits of this representation are given in [30].

2.2.1 Symbolic Task Planning

In symbolic task planning, a problem contains a single goal state, and task planners, e.g., Fast Downward [29], find the appropriate set of actions, i.e., a task plan, that can transform the initial world state into the desired goal state. The definition of a planning problem, its different components and its solution are provided below.

Definition 1. *Planning Problem:* A planning problem P can be defined as $P = (F, I, A, G)$, where F is a set of fluents, I is the initial state, G is a goal state, and A is a set of actions [31, 32].

Definition 2. *Fluent:* A fluent ($f \in F$) is a state variable.

When assigned a value, a fluent can be represented by a grounded predicate. Grounded predicates are also called atoms. For instance, `(at 1_2)` is an atom which denotes that the observed agent is at the position `1_2`.

Definition 3. *State:* A state contains all fluents, each of which is assigned a value.

The initial state (I) contains all fluents; whereas, the goal (G) could be a partial state, containing a subset of fluents. To transition between states, the value of fluents are altered by actions. An action is formally defined as follows:

Definition 4. *Action:* An action (a) is comprised of a name, a set of objects, a set of preconditions (a_{pre}) and a set of effects (a_{eff}). Preconditions and effects are composed of a set of valued fluents. Preconditions can contain `or` and `and` statements.

Action a is applicable to state s if the state is consistent with the action's preconditions. Applying action a to state s will result in state s', where $a_{eff} \subseteq s'$ and $\forall(f \in s', f \notin a_{eff}) : (f \in s)$

Definition 5. *Planning Problem Solution:* A solution to a planning problem is a sequence of actions $\pi = (a0, a1, ..., ai \in A)$ such that applying each action in turn starting from state I results in a state (s^i) that is consistent with the (partial) goal state G, i.e., $s^i \supseteq G$.

Planners search for the optimal solution to a planning problem. An optimal solution is the solution with the lowest possible cost. In our work the cost of an action is 1, and thus the cost of a plan is equivalent to its length.

2.2.2 Goal Recognition

Goal recognition is often viewed as the inverse of planning, as the aim is to label a sequence of observations with the goal the observed agent is attempting to reach. This section provides the formal definition of a GR problem, and describes the observation sequences and output of our GR approach.

Definition 6. *Goal Recognition Problem:* A GR problem is defined as $T = (F, I, A, O, \mathcal{G})$, where $\mathcal{G}$ is the set of all possible (hypothesis) goals and O is a sequence of observations [20].

Definition 7. *Observations:* O is a sequence of observed actions (observations), i.e., $O = (a1, a2, ..., ai \in A)$.

A completed sequence of observations with no missing actions can be applied to an initial state I to reach a goal state $G \in \mathcal{G}$. This sequence can also be incomplete, have missing observations or/and be suboptimal. An incomplete sequence of observations contains the first N actions that are required to reach a goal; in other words, the goal has not yet been reached. An action could be missing from anywhere within a (incomplete or complete) sequence of observations. Observations are suboptimal if any number of additional, unnecessary actions have been performed to reach the goal.

GR approaches attempt to select the real goal from the set of hypothesis goals $\mathcal{G}$. Our GR approach produces a probability distribution over the hypothesis goals, i.e., $\sum_{i=1}^{|\mathcal{G}|} P(G_i|O) = 1$. In other words, we aim to find the likelihood of a given observation sequence O under the assumption that the observee is pursuing a goal G_i, i.e., $P(O|G_i)$. The goal(s) with the highest probability are returned as the set of candidate goals $\mathcal{C}$. As goals can be equally probable, there can be multiple candidate goals, i.e., $|\mathcal{C}| \geq 1$. Nevertheless, we assume that there is only a single real goal. Note, our evaluation metrics (see Section 2.7.1) take into account that multiple goals could be returned.

2.3 Action Graph Structure and Formal Definitions

Action Graphs model the possible order constraints between actions by linking actions (dependants) to their dependencies. They are constructed of action nodes and operator nodes, namely, `DEP` (short for dependencies), `UNORDERED-AND`, `OR` and `UNORDERED-AND` nodes. Action nodes are always leaf

nodes and their dependencies are conveyed through their connections (via operator nodes) to other actions. Action Graphs are not designed as a planning mechanism. This section defines dependencies, provides a definition of an Action Graph, describes how the Action Graph structure links actions to their dependencies and briefly mentions related structures.

Definition 8. *Action's Dependencies* The set of dependencies of action $a \in \mathcal{A}$ is formally defined as: $D(a) = \{a' \mid (a'_{eff} \cap a_{pre}) \neq \emptyset\}$.

Definition 9. *Action's Dependant* Action a is a dependant of action a' if $a' \in D(a)$.

In other words, action a' is a dependency of action a if at least one effect of a' fulfils at least one of a's preconditions, i.e., $a' \in D(a)$ if $a'_{eff} \cap a_{pre} \neq \emptyset$. In that case, action a is called the dependant of the dependency a'. The order in which dependencies are likely to be observed can be conveyed by the nodes of an Action Graph.

Definition 10. *Action Graph* $AG = (N^O, N^A, E)$, where N^O is a set of operator nodes, N^A are action nodes and E are edges [2]. Operator nodes are of type `DEP`, `UNORDERED-AND`, `OR` and `ORDERED-AND` nodes, i.e., $N^O = (N^{OR}, N^{DEP}, N^{O\text{-}AND}, N^{U\text{-}AND})$. The root node is of type `OR`. All nodes (except the root) have a set of parents. All operator nodes (N^O) have a set of children, those children can be operator nodes or action nodes. N^A are leaf nodes.

The operator node types are described in the list below and depicted in Figure 2.2. The precedes operator $\prec$ denotes that the list of actions on the left precede (are dependencies of) the action on the right. Standard maths notation is used to denote if a set of actions is unordered or ordered, that is, curly brackets denote the actions are unordered and angle brackets show the actions are ordered [33]. Moreover, rather than writing *or* constraints as two statements, e.g., $a4 \prec a1$ OR $a5 \prec a1$, a shortened form is given, e.g., $or(a4, a5) \prec a1$.

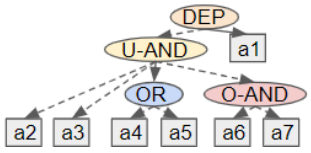


Figure 2.2: The different types of order constraints on actions that achieve `a1`'s preconditions, i.e., $\{a2, a3, or(a4, a5), \langle a6, a7 \rangle\} \prec a1$. Solid arrows point to the dependant and dashed arrows point to the dependencies. `UNORDERED-AND` is shortened to `U-AND` and `ORDERED-AND` to `O-AND`.

[2] N^x denotes a set of nodes that are of a specific type (x).

- `DEP` nodes indicate that an action's dependencies are performed before the action itself, e.g., $D(a1) \prec a1$. The second (i.e., last) child of a `DEP` node is the action node itself; the first child could be of any type.
- `UNORDERED-AND` nodes denote that different dependencies set different preconditions (and there are no order constraints on the dependencies), e.g., if $a2 \in D(a1)$, $a3 \in D(a1)$ and $(a1_{pre} \cap a2_{eff}) \neq (a1_{pre} \cap a3_{eff})$, then $\{a2, a3\} \prec a1$.
- `OR` nodes express the multiple (alternative) ways a precondition can be reached, e.g., if $a4 \in D(a1)$, $a5 \in D(a1)$ and $(a1_{pre} \cap a4_{eff}) = (a1_{pre} \cap a5_{eff})$, then $or(a4, a5) \prec a1$.
- `ORDERED-AND` nodes indicate there are order constraints between an action's dependencies. Such constraints are required when executing one dependency could unset the preconditions of another. For example, if $a6 \in D(a1)$, $a7 \in D(a1)$ and both $a6_{pre}$ and $a7_{eff}$ contain the same fluent but with different values, then $a6$ is performed before $a7$, i.e., $\langle a6, a7 \rangle \prec a1$. This is because an effect (fluent) of $a7$ is a precondition of $a1$ but to perform $a6$ that fluent must be assigned a different value. If these constraints are cyclic, e.g., $\{\langle a6, a7 \rangle, \langle a7, a6 \rangle\} \prec a1$, then the constraint is ignored; in other words, the dependencies are considered to be unordered.

Dependencies are actions, and thus they can also have dependencies. For example $a8$ could depend on $a9$, which depends on $a10$, i.e., $a10 \prec a9 \prec a8$. If an action has dependencies, its only parent is the `DEP` node linking it to its dependencies (and dependants). Thus, continuing with the example, the left child of $a8$'s parent `DEP` node is $a9$'s parent `DEP` node.

Cyclic dependencies can also occur, e.g., $a1$ could depend on $a2$ which depends on $a1$ (i.e., $...a2 \prec a1 \prec a2...$). This causes cycles to appear within the Action Graph. These cycles can also be caused by indirect dependencies, e.g., $...a2 \prec a3 \prec a1 \prec a2...$ in which $a1$ is a dependency of $a2$ and $a2$ is an indirect dependency of $a1$.

Our Action Graph structure does not contain states. An Action Graph only captures information about which actions fulfil each action's preconditions. This is similar to the structures of library-based intention recognition and planning approaches [34–37]. For instance, Goal-Plan trees contain (sub)goals with plans that can contain subgoals [36, 37]. Goal-Plan trees do not contain knowledge of the environment's current state. In particular, the Action Graph structure was inspired by the work of Holtzen et al. [34], who represented a library of plans as AND/OR trees. Differently,

our approach takes PDDL rather than a library of plans as input, and enables suboptimal and cyclic dependencies to be represented.

Moreover, the definition of a dependency is similar to causal links from Partial-Order Causal Link (POCL) planning [38]. Like dependencies, a causal link expresses that an action's preconditions are contained within another action's effects. Differently, POCL structures represent complete plans (to reach a single goal state from the initial state), edges rather than nodes are used to denote the order constraints and they can contain ungrounded actions. As GR does not require a completely valid plan, Action Graphs are simpler to construct than POCL structures.

2.4 Cyclic Action Graph Creation

Our goal recognition method creates an Action Graph, labels the nodes with their distance from each goal, then for each observation updates the goals' probability. This section describes how an Action Graph is generated from a GR problem. The modifications to the preprocessing step, which transforms a PDDL problem into a multi-valued problem, are described. Subsequently, the action insertion algorithm is detailed, followed by an example.

2.4.1 Preprocessing: Multi-Valued Problem Generation

This chapter only provides the details of the transformation, from a PDDL defined problem to a multi-valued problem, that are key to understanding our approach and that differ from [29]. A single goal statement is required by the converter of Helmert [29]; therefore, prior to calling the converter, a goal statement is created by placing all hypothesis goals ($\mathcal{G}$) into an `or` statement, i.e., $G = or(G_1, G_2, ..., G_{|\mathcal{G}|} \in \mathcal{G})$.

The converter's parameter, to keep all unreachable states, is set to true and, after parsing the PDDL, all groundings of the actions' effects are inserted into the initial state (I). This forces actions, and all fluents' values, to be inserted into the resulting representation even if the actions' fluent preconditions, and thus possibly the goals, are unreachable from the defined (original) initial state. For instance, if an agent's location is missing from I, e.g., because it is unknown, and no transition between an unknown and known location exists, then `move` actions would not be inserted as their preconditions can never be met. To prevent this, all `(at ?location)` groundings are inserted into I. Additional static atoms are not inserted into I; thus, continuing with our example, `move(1_1 1_2)` is only appended to the set of actions A if `(adjacent 1_1 1_2)` is declared in the defined initial state.

Our modifications expand the input provided to the translator but do not

directly alter the time complexity of the translator. Nevertheless, the translator will produce a greater number of actions, which are provided as input to our Action Graph creation method. Some of the additional actions could be unnecessary due to their preconditions being unreachable; however, there are likely to be few unreachable actions. The amount of unnecessary data the translator and our algorithms must process depends on quality of the PDDL code. For example, if a navigation problem contains an unreachable set of adjacent locations due to a mistake having been made, actions to move between (but not too) these locations will still be created. Without our modifications, the translator would remove these unreachable actions.

2.4.2 Inserting Actions Into an Action Graph

An Action Graph is initialised with an `OR` node as the root; then each action ($a \in A$) is inserted into the graph in turn by connecting it to its dependencies. Actions can be inserted in any order. Finally, the graph is adjusted so only the Goal Actions' parent `DEP` nodes are connected to the root. This process is detailed below. The pseudo-code for our Action Graph creation method is provided in Algorithm 2.1.

If an action has no dependencies, because either there are no actions that fulfil its preconditions or it has no preconditions, it is simply appended to the root's children (line 8). In all other cases, the root is linked to a new `DEP` node (lines 10 and 12). The `DEP` node's two children are set to an `UNORDERED-AND` node, proceeded by the action node itself (line 37). If this action node was already created, because it is a dependency of an already processed action, the action node's prior parents are moved to be the `DEP` node's parents (line 11).

The `UNORDERED-AND` node's children are set to one or more of the following: the action nodes (or parents) of the dependencies, `OR` nodes if there are multiple ways in which a precondition can be met, and/or `ORDERED-AND` nodes. `OR` nodes' are inserted by setting their children to the action nodes of the dependencies that set the same precondition(s) (line 23). If a dependency has dependencies, the corresponding child becomes the dependency's parent. This is because actions that have dependencies can only ever have a single parent, of type `DEP`. Note: if an operator would only have one child, the operator node is not inserted.

`ORDERED-AND` nodes indicate that there are order constraints on the dependencies themselves. This is detected by checking if a fluent has a value in a dependency's preconditions which is different in another dependency's effects (see Section 2.3); and thus the former dependency must be performed first (lines 29-31). If this constraint is bidirectional/cyclic, the `ORDERED-AND`

Algorithm 2.1 Action Graph creation

Data: $\mathcal{A}$ set of all actions, $\mathcal{G}$ set of hypothesis goals
Result: Action Graph

$root_n \leftarrow ORnode(\emptyset)$ ▷ Create `OR` node with no children
for each $a \in \mathcal{A}$ **do**
  $\mathcal{D}_a \leftarrow [\emptyset]$ ▷ 2D array - array per precondition
  $a_n = find(a)$ ▷ finds a's parent `DEP` node or finds/creates a's action node.
  for each $p \in a_{pre}$ **do** $\mathcal{D}_a$ += $\{a' \mid p \in a'_{eff}\}$ ▷ Get dependencies
  end for
  if $\mathcal{D}_a = [\emptyset]$ **then** ▷ e.g., because a has 0 preconditions
    $root_n.children$ += a_n ▷ append a's Action Node to root
  else
    $dep_n \leftarrow DEPnode(\emptyset)$ ▷ Create `DEP` node
    $dep_n.parents = a_n.parents$; $a_n.parents = \emptyset$
    $root_n.children$ += dep_n
    if $|\mathcal{D}_a| = 1$ **then** ▷ e.g., because a has 1 precondition
      if $|\mathcal{D}_a[0]| > 1$ **then**
        $dep_n.children = [ORnode(\mathcal{D}_a[0]), a_n]$ ▷ Node constructors will call the $find$ method on each action.
      else if $|\mathcal{D}_a[0]| == 1$ **then** $dep_n.children = [find(\mathcal{D}_a[0][0]), a_n]$
      end if
    else
      $u\text{-}and_n \leftarrow UANDnode(\emptyset)$ ▷ Create `UNORDERED-AND` node
      $OR_N = \emptyset$
      for each $D_p \in \mathcal{D}_a$ **do**
        if $|D_p| > 1$ **then**
          $or_n = ORnode(\mathcal{D}_p)$ ▷ Actions that set the same precondition are always the children of an (the same) OR node
          $OR_N.append(or_n)$
        else if $|D_p| == 1$; **then** $OR_N.append(find(D_p[0]))$
        end if
      end for
      for each $or_n \in OR_N$ **do**
        if $\{a' \mid a' \in or_n.children, a'$ is affected by any $a'' \in OR_n$ or $\{a'' \mid a'' \in or'_n.children, or'_n \in OR_N\}\} \neq \emptyset$ **and** $\mathcal{D}_a$ has no cycles **then**
          $o\text{-}and_n \leftarrow OANDnode(or_n, UANDnode($the OR_n nodes of affecting actions$))$ ▷ U-AND node is not created if it would have 1 child
          $u\text{-}and_n.children$ += $o\text{-}and_n$
        else if or_n not already decedent of $u\text{-}and_n$ **then**
          $u\text{-}and_n.children$ += or_n
        end if
      end for
    end if
    if $u\text{-}and_n.children > 1$ **then** $dep_n = (u\text{-}and_n, a_n)$
    else $dep_n.children = (u\text{-}and_n.children[0], a_n)$
    end if
  end if
end for

node is not inserted; instead, the dependencies become the children of the UNORDERED-AND node (line 33). Only the preconditions/effects of direct dependencies are checked, the algorithm does not check if a dependence's dependency could undo/unset a dependency's precondition. Performing this check would be computationally complex and a perfect representation of the plans to reach of the actions' effects is not required.

The ORDERED-AND node's children could also be of type UNORDERED-AND or OR (line 30). When multiple dependencies could unset a dependency's preconditions, an UNORDERED-AND is inserted as the child of the ORDERED-AND node's right-branch. Moreover, the dependencies that set the same precondition(s) of the dependant are grouped together as the children of an OR node (lines 20-27). Therefore, if one of these dependencies is affected by (or effects) another dependency, the OR node becomes the ORDERED-AND (or UNORDERED-AND) node's corresponding child. Without this feature, the graph's structure would become more complex, i.e., be of greater depth and/or breadth.

Inserting actions into an Action Graph has a time complexity of $O(n^2)$, where n is the number of grounded actions. This is because for each action all actions are checked to find the action's dependencies.

2.4.3 Identifying Goal Actions

An action is a Goal Action if its effects fulfil a goal's atoms, i.e., $a_{eff} \supseteq G$, where $G \in \mathcal{G}$. After all actions have been inserted, the root node's children are modified so that only the Goal Actions are attached to the root. If multiple actions are required to fulfil a goal, e.g., $(a1_{eff} \cup a2_{eff}) \supseteq G$, then an auxiliary Goal Action (a^x) is created. Auxiliary Goal Actions are linked to the multiple actions that fulfil the goal via a DEP node, e.g., $\{a1, a2\} \prec a^x$. They are connected to their dependencies, i.e., the goal's dependencies, in the same way as all other actions are.

Identifying and creating Goal Actions simplifies traversing the graph to find all nodes belonging to a single goal. All children, including indirect children, of a Goal Action's parent DEP node could appear in a plan (from any initial state) to reach the goal the Goal Action fulfils. Therefore, the graph can be traversed, in a depth-first or breadth-first manner, to find all the nodes, and thus actions, belonging to a goal.

2.4.4 Example

An example is provided, in this section, to demonstrate how our creation algorithm works for the grid-based navigation problem depicted in Figure 2.3. Figure 2.4 shows the Action Graph after each action, and its de-

pendencies, have been inserted. The four insertions, detailed below, were selected to show the different structural features of an Action Graph. A figure with all actions inserted into the graph would be unreadable, and thus is not provided.

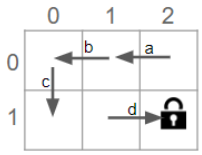


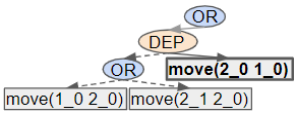
Figure 2.3: Example problem from the Easy-IPC-Grid domain. Before an agent can move to position (2,1), it must be unlocked. The lettered arrows indicate the actions inserted to produce the sub-figures of Figure 2.4. Based on the GR Easy-IPC-Grid problems developed by Ramírez and Geffner [39], based on a domain from the official IPC[1].

The example starts by inserting the goal action, `move(2_0 1_0)`. The preconditions of `move(2_0 1_0)` are met by executing one of two possible actions, i.e., $or($`move(1_0 2_0)`,`move(2_1 2_0)`$) \prec$ `move(2_0 1_0)`; therefore, it is inserted by connecting it to its dependencies via a `DEP` node and a `OR` node (see Figure 2.4a). Likewise, when `move(1_0 0_0)`, whose preconditions are reached by one of three actions (i.e., $or($`move(2_0 1_0)`,`move(1_1 1_0)`,`move(0_0 1_0)`$) \prec$ `move(1_0 0_0)`), is inserted, an `OR` node is created. As one of its dependencies has already been inserted, the appropriate child of the `OR` node is set to `move(2_0 1_0)`'s parent `DEP` node. This is shown in Figure 2.4b.

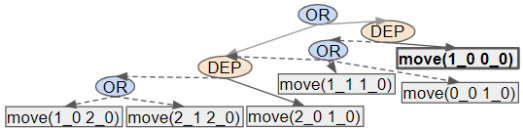
Inserting `move(0_0 1_0)` causes the graph to become cyclic (Figure 2.4c) because it depends on one of its dependants, i.e., `move(1_0 0_0)` $\prec$ `move(0_0 1_0)`. Figure 2.4d displays the graph after `move(1_1 2_1)` has been inserted. This action requires location `2_1` to be unlocked with `key1`, and thus its dependencies include `unlock` actions. As `unlock` actions' preconditions contain the location of the agent, they must be performed prior to the move actions required by the dependant. Therefore, an `ORDERED-AND` node is created during the insertion of `move(1_1 2_1)`, i.e., $\langle or($`unlock(2_1 2_0 key1)`,`unlock(2_1 1_1 key1)`$), or($`move(0_1 1_1)`,`move(2_1 1_1)`,`move(1_0 1_1)`$)\rangle \prec$ `move(1_1 2_1)`.

2.5 Node Distance Initialisation

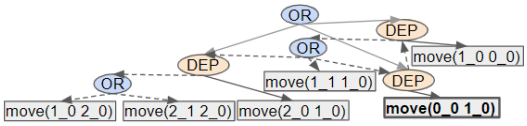
Each node has a set of distances associated with it, which indicate how far the node is from each goal, i.e., the number of `DEP` and `ORDERED-AND` nodes that must be traversed to get from the Goal Action's parent to the node in question. These distances are set by means of a breadth-first traversal



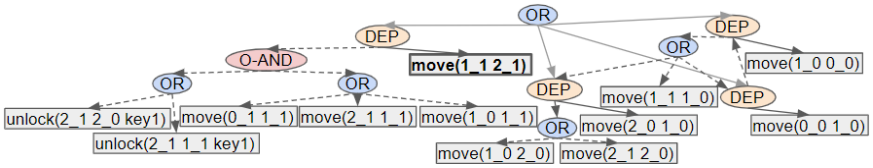
(a) The Action Graph after `move(2_0 1_0)` and its dependencies have been inserted.



(b) `move(1_0 0_0)` has been inserted into the Action Graph along with its dependencies. As `move(2_0 1_0)` was previously inserted, the new `OR` node was connected to it via its parent (`DEP`) node.



(c) `move(0_0 1_0)`'s dependency was already inserted into the graph. Connecting it to its dependency has caused a cyclic to occur.



(d) The order constrains on `move(1_1 2_1)`'s dependencies required an `ORDERED-AND` node to be inserted as performing a move action could undo a precondition of an action that unlocks position `2_1`.

Figure 2.4: Example of the steps taken to insert four actions, and their dependencies, into an Action Graph. This example has been simplified, i.e., in the original Easy-IPC-Grid problems, keys have different shapes. Solid pale arrows show the root node's connections (prior to the Goal Actions being discovered).

(BFT). An explanation of this algorithm is provided, followed by an example. The pseudo-code can be found in Appendix 2.A.

2.5.1 Node Value Initialisation Algorithm

During the BFTs, that start from each Goal Action's parent node, the current node's distance is set, the count (i.e., distance measure) is increased if the node is of type `DEP` or `ORDERED-AND`, and each of the node's children are pushed onto the BFT-queue. This distance measure provides an indication of how far each node is from each goal whilst attempting to minimise favouring shorter plans (see Section 2.6 for the calculations of the goal probabilities). The same node could be visited multiple times during a BFT; however, if the current distance/count is greater than or equal to

the node's already assigned distance, it is not reprocessed. As well as allowing the shortest distance to be assigned to each node, this prevents an endless loop from occurring when two actions depend on each other (e.g., $...a2 \prec a1 \prec a2...$).

As an action could appear in a plan multiple times, some nodes require multiple distances for the same goal; this is the case for the descendants of `ORDERED-AND` nodes' right branch. Therefore, a node contains a map for each goal, from the last traversed `ORDERED-AND` to the node's distance from the goal via the `ORDERED-AND` node. When the right branch of the `ORDERED-AND` node has been fully observed, the distance of the node, returned when calling a get distance method, will be the distance associated with that `ORDERED-AND` node. As the initial state is unknown and plans are not perfectly represented, the distances assigned to the left branch of `ORDERED-AND` nodes are not based on the depth of the right branch.

Labelling nodes with multiple distances per goal increases the worst case time complexity from $O(n^2)$ to $O(n^3)$, with respect to the number of actions. This is because each action in the graph could be a dependency of all other actions; thus, for all actions all other actions could be visited. When labelling the nodes with multiple values this process could be repeated n times. Therefore, to help minimise the number of nodes the BFTs traverse, when an `UNORDERED-AND` node is reached, its children's (including indirect children's) distance is not associated with the prior `ORDERED-AND` node(s). Developing this component greatly reduced ($\approx$halved) the run time of our experiments (Section 2.7.2.2) and had negligible impact on the accuracy of our approach.

The offline component of our system finishes by setting the prior probability of each goal. We chose to use a uniform prior probability as, since no actions have been observed, all goals are assumed to be equally likely.

2.5.2 Example

The Action Graph depicted in Figure 2.5 shows the resulting action nodes' distance from each goal for a simplified version of the Kitchen domain by Ramírez and Geffner [39]. In this example, there are two Goal Action nodes, namely, `pack-lunch()` and `make-dinner()`. By executing the BFTs described above, each node is labelled with their distance from each goal. This example will be used in Section 2.6, to demonstrate how an observation affects the goals' probability, and thus why this node value initialisation procedure has been implemented.

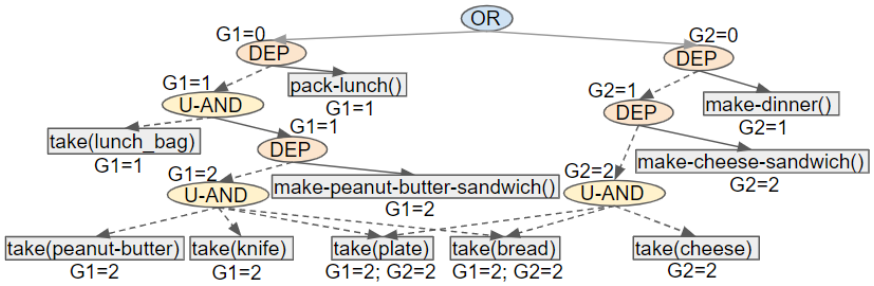


Figure 2.5: Example of an Action Graph with the action nodes labelled with their distance from each goal. `G1` represents the goal `(made_lunch)`, which is reached by performing the `pack-lunch()` action and `G2` represents `(made_dinner)`, which is reached by performing the `make-dinner()` action. This example is based on the Kitchen domain developed by Ramírez and Geffner [20] based on the work of Wu et al. [16]. To make this figure readable, it has been simplified, i.e., many nodes and edges are not included.

2.6 Updating the Goal Probabilities

When an action is observed, the probability associated with each goal is updated based on either its distance from the observed action or the difference between its distance from the prior observation and the current observation. These two update rules are described in turn along with their advantages and disadvantages. The experiments section presents results for both these update rules separately, as well as combined. The algorithms for update rules 1 and 2 are provided in Appendix 2.B. The pseudo-code for the rules combined is provided in Algorithm 2.2.

2.6.1 Update Rule 1: Distance From Observed Action

Each goal's probability is updated based on how close the goal is to the observed action and how unique the observation is to the goal. The probabilities of the goals closest to the observation are increased, whilst those furthest from the observation are decreased. If an observation only belongs to a single goal, that goal's probability is increased and all other probabilities are decreased. This is performed by multiplying each goal's probability by its distance from the observed action's node divided by the sum of all goals' distances (lines 8-11); then normalising the resulting values (lines 13-15). Note, if the observation is not within a plan to reach the goal G, 0 is returned by the *getDisFromGoal* method (line 9), so that $c(G) = 0$ and, as long as another goal's plan contains the action, its probability is reduced.

For the example shown in Figure 2.5, there are two goals, both with a prior probability of 0.5. When the `take(plate)` action is observed, the resulting

Algorithm 2.2 Update the goal probabilities.

Data: o^t observed action, o^{t-1} previously observed action, $\mathcal{G}$ set of hypothesis goals with current probability
Result: $\mathcal{G}$ set of hypothesis goals with updated probability

if $o^{t-1} \neq null$ **and** areConnectedViaDep/OAndNodes(o^t, o^{t-1}) **then**
  $\forall G \in \mathcal{G} : v(G) = P(G)$
  for each $G \in \{G' \mid G' \in \mathcal{G}, o^t.\text{hasDisFromGoal}(G')\}$ **do**
    $c(G) = \sigma(o^{t-1}.\text{getDisFromGoal}(G) - o^t.\text{getDisFromGoal}(G))$
    $v(G) = P(G)(1 + c(G))$
  end for
else
  for each $G \in \mathcal{G}$ **do**
    $c(G) = \frac{o^t.\text{getDisFromGoal}(G)}{\sum_{G' \in \mathcal{G}} o^t.\text{getDisFromGoal}(G')}$
    $v(G) = P(G)(1 + c(G))$
  end for
end if
for each $G \in \mathcal{G}$ **do**
  $P(G) = \frac{v(G)}{\sum_{G' \in \mathcal{G}} v(G')}$ ▷ probabilities sum to 1
end for
updateNodeDistancesIfo-andLeftBranchFullyObserved(o^t)

probabilities are unaltered as its node's distance to each goal is equal. More nodes must be traversed to reach `take(plate)` from `pack-lunch()`, than from `make-dinner()`. Nevertheless, the goal with a shorter plan was not favoured as the distance counter (see Section 2.5) was only increased when a `DEP` or `ORDERED-AND` node was traversed. If `take(knife)` is observed, the probability of making a pack lunch is increased, i.e., $P(\texttt{(made-lunch)}) = 0.67$ and $P(\texttt{(make-dinner)}) = 0.33$, as the observed action is unique to this goal.

The main disadvantage of this approach is that the probabilities of goals with shorter, strongly ordered plans are increased more than those with longer plans. Therefore, the list of returned candidate goals $\mathcal{C}$ often contains the goal(s) with a shorter plan. For instance, if an incomplete sequence of observations contains actions that approach both G1 and G2, whichever of these two goals has the shortest plan length will be returned as a candidate goal, the other will not be. The subsequent update rule aims towards mitigating this disadvantage.

2.6.2 Update Rule 2: Change in Distance From the Observed Actions

If the previous observation (o^{t-1}) and the current observation (o^t) are connected via a `DEP` or `ORDERED-AND` node, the goal probabilities are updated based on the change in distance, i.e., the difference between the goal's distance from the previous and current observations (lines 2-6 of Algorithm 2.2). To check if the observations are connected, an upwards traversal (in a depth-first manner) is performed, starting from the action node of o^{t-1}, to find a `DEP` or `ORDERED-AND` node whose right branch's child is the action node of o^t.

If the list of observations is not missing any actions, the change in distance will always be 1, 0 or -1. As an observation could be missed (e.g., due to sensor failure), our algorithm needs to account for the difference being within a wider range of values. A negative difference indicates the observee moved further from the goal; whereas, a positive difference indicates they moved closer. The logistic sigmoid function converts the difference into a value between 0 and 1 (i.e., σ from line 4); the goal's value is multiplied by this (line 5) then normalised (lines 13-15). If either observation does not belong to the goal, the value of $v(G)$ is equivalent to setting the result of the sigmoid function to 0; in other words, the difference is $-\infty$. This update rule results in the probability of the goal the observee is moving towards at the highest rate to be increased the most.

When this rule is used independently from the first rule, if the previous observation is null or the current and previous observations are not connected via a `DEP` or `ORDERED-AND` node (as detailed above), then $c(G) = 0.5$ for the goals dependent (or indirectly dependent) on the current observation and $c(G) = 0$ for all other goals. This prevents the goals that have shorter plans from being favoured; however, goals for which the observation appears in a (very) suboptimal plan are treated equally to those for which an optimal plan contains the observation.

In the example depicted in Figure 2.6, if the observee moves from position `2_1` to `1_1` then to `0_1`, the goal probabilities remain equal. As the distance to both goals is reduced at the same rate, the real goal is indiscernible. If the observee were to move vertically, and thus step towards one goal (and away from the other), the corresponding goal's probability is increased, e.g., observing `move(2_1 1_1)` then `move(1_1 1_0)` results in $P(G1) = 0.58$ and $P(G2) = 0.42$.

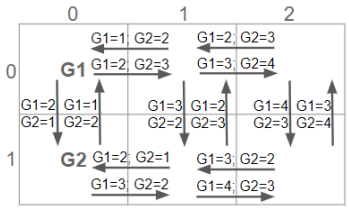


Figure 2.6: Action nodes' distance from each goal, represented on a depiction of a grid-based navigation environment. `G1` and `G2` are goals and arrows represent move actions. `G1` and `G2` have a prior probability of 0.5.

2.6.3 Processes Common to Update Rules 1 and 2

In both update rules, $1 + c(G)$ is calculated, rather than just $c(G)$, so that a goal's probability is never set to 0. If the probability of a goal were to be set to 0, it cannot be increased; thus, the heuristic would not be able to recover from receiving an incorrect (noisy) observation. These update rules, along with the graph's structure, enable our system to handle noisy observations as well as suboptimal plans and missing observations.

When an action has been observed, which operator nodes have been completed are updated by traversing up the graph, in a depth-first manner, from the observed action's node (line 16). `OR` nodes are set as completed if one of their children has been completed, `DEP` nodes are complete if the child connected to their right branch has been observed and `UNORDERED-AND` nodes are set as completed if all their children are. If a node is not set to completed, its parents are not traversed. When an `ORDERED-AND` node's left branch has been completed, the nodes attached to its right branch are informed so that their distance associated with that `ORDERED-AND` node is used when the next observation is received (as described in Section 2.5).

2.7 Experiments

Through experiments we aim to demonstrate the accuracy of our GR approach, after 10, 30, 50, 70 and 100 % of actions in a plan have been observed, on 15 different domains. This section describes the evaluation metrics, followed by the setup and results of the different experiments. A comparison between our different update rules and the goal completion heuristic, namely, h_{gc}, by Pereira et al. [19, 21] is provided, on a dataset containing GR problems with a known, and thus correctly defined, initial world state. Our method is then compared to h_{gc} on problems for which differing percentages of fluents have been set to incorrect values.

Pereira et al. [19, 21] recently improved the accuracy and computational time of GR by finding landmarks, i.e., states that must be reached to achieve a particular goal. After processing the observations, the resulting value of each goal ($G \in \mathcal{G}$) is based on the percentage of its landmarks that have been completed. h_{gc} takes a threshold value as a parameter. Any goals whose value is greater than or equal to the most likely goal's value minus the threshold are included in $\mathcal{C}$. When the threshold is 0.0, like our approach, only the most likely goal(s) are included in $\mathcal{C}$. Therefore, we present the results of their approach for 0.0 as the threshold. The compiled version of h_{gc}, provided by Pereira et al. [19], was ran during the experiments[3]. We provide a detailed comparison to h_{gc} as it has been very recently developed and was shown to outperform alternative methods; other approaches to GR will be discussed in the related work section (Section 2.8).

The dataset created by Ramírez and Geffner [20] and Pereira et al. [19][4] forms the basis for our experiments. Details on generating the lists of observations and the inaccurate initial states are provided in the setup sections specific to each experiment. A brief description of each domain is supplied in Appendix 2.C. Experiments were ran on a server with 16 GB of RAM and a Intel Xeon 3.10 GHz processor.

2.7.1 Evaluation Metrics

Our approach is evaluated on the number of returned candidate goals (i.e., $|\mathcal{C}|$) and standard classification metrics, namely, quality (sometimes referred to as accuracy), precision, recall and F1-Score [40, 41]. A definition for each of these is provided below. Subsequently, performance profiles, which are provided to show a comparison of the approaches' run-times, are introduced.

2.7.1.1 Classification Metrics Applied to Goal Recognition

Quality ($Q_{P,S}$), precision ($M_{P,S}$), recall ($R_{P,S}$) and F1-Score ($F1_{P,S}$) are provided in Equations 2.1, 2.2, 2.3 and 2.4 respectively. The definitions of TP, FP, FN and TN are provided, from a GR perspective, in Table 2.1. In these definitions G_P is the actual goal (ground truth) for problem P, $\mathcal{C}_{P,S}$ is the set of candidate goals returned by solution/approach S and $\mathcal{G}_P$ is the set of hypothesis goals. TP is 1 if the true goal is in the set of candidates or 0 if it is not; FN is the inverse of TP; FP is the number of returned candidates that are not the real goal, and TN is the number of goals correctly identified as not the real goal. For each metric, the average over all problems per

[3] https://github.com/ramonpereira/Landmark-Based-GoalRecognition/blob/master/goalrecognizer1.1.jar

[4] https://github.com/pucrs-automated-planning/goal-plan-recognition-dataset

domain is displayed in the results.

Table 2.1: Definitions of True Positive (TP), False Positive (FP), False Negative (FN) and True Negative (TN) results of solution/approach (S) on a goal recognition problem (P).

$TP = \begin{cases} 1, & \text{if } G_P \in \mathcal{C}_{P,S} \\ 0, & \text{otherwise} \end{cases}$	$FP = \begin{cases} \lvert\mathcal{C}_{P,S}\rvert - 1, & \text{if } G_P \in \mathcal{C}_{P,S} \\ \lvert\mathcal{C}_{P,S}\rvert, & \text{otherwise} \end{cases}$
$FN = \begin{cases} 0, & \text{if } G_P \in \mathcal{C}_{P,S} \\ 1, & \text{otherwise} \end{cases}$	$TN = (\lvert\mathcal{G}_P\rvert - 1) - FP$

$$Q_{P,S} = \frac{TP + TN}{TP + TN + FN + TP} \tag{2.1}$$

$$M_{P,S} = \frac{TP}{TP + FP} \tag{2.2}$$

$$R_{P,S} = \frac{TP}{TP + FN} \tag{2.3}$$

$$F1_{P,S} = \begin{cases} 2 * \frac{M_{P,S} * R_{P,S}}{M_{P,S} + R_{P,S}}, & \text{if } G_P \in \mathcal{C}_{\mathcal{P},\mathcal{S}} \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Prior goal recognition papers [19, 20, 42] defined the accuracy/quality as the number of times the actual goal appeared in the set of candidate goals, i.e., they did not take $|\mathcal{C}|$ into consideration when calculating accuracy. This resulted in approaches being reported as 100 % accurate, even when more than one candidate goal was returned. In our work, this is equivalent to recall ($R_{P,S}$). By using the definitions provided in this chapter, an approach can only have a quality of 1 (i.e., 100 %) if it always returns one candidate goal, i.e., the real goal.

2.7.1.2 Performance Profiles

The computation times (T) are presented in performance profiles, as suggested by Dolan and Moré [43]. This enables the results to be presented in a more readable format, and all datasets can be grouped into a single result to prevent a small number of problems from dominating the discussion. To produce the performance profile of an approach ($S \in \mathcal{S}$), i.e., of our Action Graph approach and of h_{gc}, the ratio between its run-time ($T_{P,S}$) and the quickest run-time for a problem ($P \in \mathcal{P}$) is calculated, as shown in Equation 2.5. Equation 2.6 calculates the percentage of problems an approach solved when the ratio is less than a given threshold, τ. When $\tau = 0$,

the resulting $P_S(\tau)$ of an approach is the percentage of problems it solved quicker than the other approach. How much τ must be increased for 100 % of problems to be solved depends on how far off the best approach that approach is.

$$\Gamma^T_{P,S} = \frac{T_{P,S}}{\min\left(T_{P,S} : S \in \mathcal{S}\right)} \tag{2.5}$$

$$P_S(\tau) = \frac{1}{|\mathcal{P}|}|P \in \mathcal{P} : \Gamma^T_{P,S} \leq \tau| \tag{2.6}$$

2.7.2 Goal Recognition with a Known Initial State

After describing the experiment setup, this section compares the computational time of our Action Graph approach to h_{gc}. The accuracy of these approaches, when ran on problems with accurate initial states, are then discussed. This includes GR problems where the observations are the first N % of actions in a plan and ones for which the observations are a random N % of actions in a plan (i.e., is missing observations).

2.7.2.1 Setup

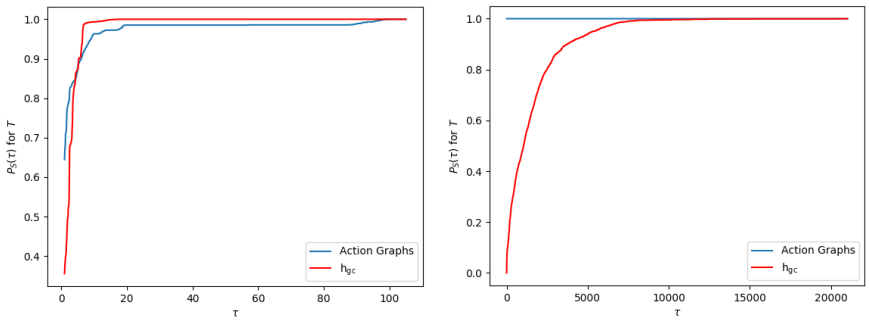
The GR problems in the original dataset[4] contain 10/30/50/70/100 % of observable actions in the plan to reach a goal; these observations/actions were selected at random. Therefore, for each of the original problems that contain 100 % of the observations, we generated GR problems by selecting the first 10, 30, 50, 70 and 100 % of observations. As a task planner (which is not guaranteed to find an optimal plan) was ran to create the problems produced by Pereira et al. [19], some observation sequences are suboptimal.

The accuracy results for our two goal probability update rules (described in Section 2.6), when ran independently and combined, are presented. In the results table these are named AG1 (i.e., the first update rule), AG2 (the second update rule) and AG3, which is the combination of the two rules (i.e., Algorithm 2.2).

2.7.2.2 Run-Times

The Action Graph heuristic took an average of 0.02 s to process all observations, whereas h_{gc} took 0.66 s. Labelling the nodes with their distance from the goals is computationally expensive; therefore, when the offline processing times (which includes the PDDL to Action Graph transformation steps) are included, Actions Graphs took an average of 2.38 s per problem. The performance profiles are displayed in Figure 2.7 and the results

per domain are shown in Table 2.2. Table 2.3 shows the average number of edges and nodes within the produced Action Graphs. These run-times and graph sizes were produced while processing the GR dataset containing the first N % of observations, which contains 2705 problems.



(a) Run-time comparison, which includes the graph initialisation (offline) time for our approach.

(b) Run-time comparison, which excludes the graph initialisation time for our approach.

Figure 2.7: Performance profiles comparing the recognition time of our Action Graph approach to the goal completion heuristic by Pereira et al. [19]. These values were produced using the dataset containing GR problems with the first N % of actions in a plan as observations.

When the whole process is included in the run-times, our approach GR out performed h_{gc} on 64 % of problems; however, the difference in run-time was greater for the problems our solution was slower at (than for the problems h_{gc} was slower on). This is indicated by how much τ must be increased before 100 % of problems were solved. At $\tau = 17.45$, h_{gc} solved all problems and at $\tau = 100.00$ all problems were solved by our approach. If only the online process is included, our approach solves 100 % of problems quicker than h_{gc}, and τ must reach 20426 before h_{gc} solves 100 %. Note: for h_{gc}, the landmarks could be discovered offline, and thus the online computational time reduced. This is only possible if the initial value of each fluent is known in advance.

Both the size and the structure of an Action Graph impact the run-times of our approach. Domains with relatively few actions have much shorter initialisation time (e.g., Kitchen, Intrusion-Detection and Campus). For larger domains, the structure of the graph had a greater impact as the run-time was affected by the number of times each node was visited. Our node labelling algorithm, which performs BFT, does not visit nodes if their already assigned distance is lower than the current distance. Moreover, an `UNORDERED-AND` node's children are not associated with the prior `ORDERED-`

Table 2.2: The run-times, in seconds, of our Action Graph approach, including and excluding the Action Graph initialisation (i.e., creation and node labelling) time, and h_{gc} [19] per domain. *ALL* is the total/average over all problems.

		Action Graphs (incl. graph initialisation)			Action Graphs (excl. graph initialisation)			h_{gc}		
Domain	$\|probs\|$	$\sum t$	$\bar{t}$	$\pm std$	$\sum t$	$\bar{t}$	$\pm std$	$\sum t$	$\bar{t}$	$\pm std$
Blocks-World	460	109.54	0.24	0.38	**0.27**	**0.00**	0.00	213.73	0.46	0.37
Campus	75	5.02	0.07	0.00	**0.01**	**0.00**	0.00	22.50	0.30	0.07
Depots	140	193.62	1.38	1.99	**0.15**	**0.00**	0.00	108.78	0.78	0.24
Driverlog	140	1964.08	14.03	17.56	**0.24**	**0.00**	0.00	96.03	0.69	0.49
DWR	140	243.17	1.74	2.49	**0.11**	**0.00**	0.00	116.04	0.83	0.26
Easy-IPC-Grid	305	1269.91	4.16	5.53	**16.64**	**0.05**	0.12	174.89	0.57	0.24
Ferry	140	124.39	0.89	1.48	**2.23**	**0.02**	0.03	53.17	0.38	0.15
Intrusion-Detection	225	12.64	0.06	0.00	**0.04**	**0.00**	0.00	85.84	0.38	0.10
Kitchen	75	3.79	0.05	0.00	**0.00**	**0.00**	0.00	20.73	0.28	0.07
Logistics	305	709.73	2.33	5.21	**6.39**	**0.02**	0.06	259.74	0.85	0.85
Miconic	140	520.12	3.72	5.75	**17.71**	**0.13**	0.25	120.04	0.86	0.66
Rovers	140	207.68	1.48	1.04	**0.13**	**0.00**	0.00	119.06	0.85	0.48
Satellite	140	108.35	0.77	0.84	**0.20**	**0.00**	0.00	133.58	0.95	0.69
Sokoban	140	92.30	0.66	0.42	**0.07**	**0.00**	0.00	125.74	0.9	0.32
Zeno-Travel	140	877.03	6.26	4.68	**0.96**	**0.01**	0.01	141.02	1.01	0.43
ALL	2705	6441.37	2.38	6.03	**45.15**	**0.02**	0.08	1790.89	0.66	0.51

`AND` node, and thus are visited fewer times than if no `UNORDERED-AND` node is traversed. As a result, for example, although the Action Graph of the Zeno-Travel domain contains more nodes than Driverlog's, its run-time is shorter.

We have identified two ways in which the total processing time of our approach could be reduced. Rather than calculating all the nodes' distances from the goals upfront, this process could be performed for just the observed actions; however, observations would be processed at a reduced rate. Second, the nodes' distances for each goal could be computed in parallel; as we envision this process being performed offline (thus the computational time of this is of lesser importance) and the performance gain would be hardware dependent, this was not implemented.

2.7.2.3 Results After Processing the First N % of Observations

Our Action Graph approach outperformed h_{gc} when 10 %, 30 % and 50 % of observations had been received; at 70 % and 100 % h_{gc} slightly outperforms our approach. As described in [21], the lower the number of observations the less likely it is that a landmark is observed; therefore, h_{gc} cannot disambiguate the goals. Figure 2.8 displays the average F1-Score, produced by AG3 and h_{gc}, at each percent of observations; Table 2.4 shows

Table 2.3: The average size of the Action Graph per domain. *ALL* is the average over all problems. The Nodes column provides the total number of action, `DEP`, `ORDERED-AND`, `UNORDERED-AND` and `OR` nodes.

Domain	Edges	Nodes	Actions	DEP	O-AND	U-AND	OR
Blocks-World	13994.16	1096.39	180.89	180.89	151.87	252.46	330.28
Campus	1656.00	420.00	123.00	123.00	25.00	14.00	135.00
Depots	43174.71	3338.00	440.29	440.29	410.00	732.29	1315.14
Driverlog	24999.57	3747.86	813.14	813.14	448.00	329.29	1344.29
DWR	37562.14	3679.14	489.29	489.29	481.43	750.14	1469.00
Easy-IPC-Grid	6595.90	3097.62	1034.82	1034.82	50.23	0.00	977.75
Ferry	15777.29	1190.29	233.86	233.86	162.57	170.14	389.86
Intrusion-Detection	294.33	230.33	102.33	92.33	0.00	34.67	1.00
Kitchen	102.00	71.00	34.00	12.00	0.00	18.00	7.00
Logistics	15595.05	4467.52	991.51	991.51	768.13	10.39	1705.98
Miconic	38783.71	2798.14	928.29	928.29	12.29	6.00	923.29
Rovers	5127.14	1966.43	535.71	517.43	232.14	220.71	460.43
Satellite	10427.29	2116.14	644.43	644.43	67.00	77.86	682.43
Sokoban	16895.14	4600.14	649.43	649.43	698.29	920.29	1682.71
Zeno-Travel	107562.00	6463.00	1104.00	1104.00	1098.00	914.29	2242.71
ALL	20497.93	2619.68	574.28	571.89	305.62	261.16	906.74

the results per domain for AG1, AG2, AG3 and h_{gc}. The ALL result is the average overall domains rather than problems, so that the result is not weighted towards the domains with the most problems.

Action Graphs have a low precision and recall for the Sokoban domain. GR problems for the Sokoban domain contain observations to navigate to and push two boxes to different locations. The actions for collecting and pushing one box were observed, before the second box was acted on. Whilst observing the actions to push the first box to its goal location, our Action Graph approach increased the probability of the appropriate goals (i.e., the goals the box was becoming closer to). When the observed agent started to navigate to the second box, the aforementioned goals' probability was decreased. Therefore, when the second box is pushed, any of the goals containing a location it is being pushed towards could appear in the set of candidate goals. In other words, the goal probabilities lose information about the first goal atom to be achieved. We considered increasing the probability of the goals with fully observed atoms; however, in problems from other domains some goals' atoms are sub-goals of another goal.

For the Kitchen domain, our Action Graph approach reduced the number of candidate goals significantly more than h_{gc} as few landmarks were observed. On the other hand, due to the structure of the produced graph, Action Graphs produced a low R and M for the Depots and Blocks-World domains. The plans for these domains are highly state dependent, which

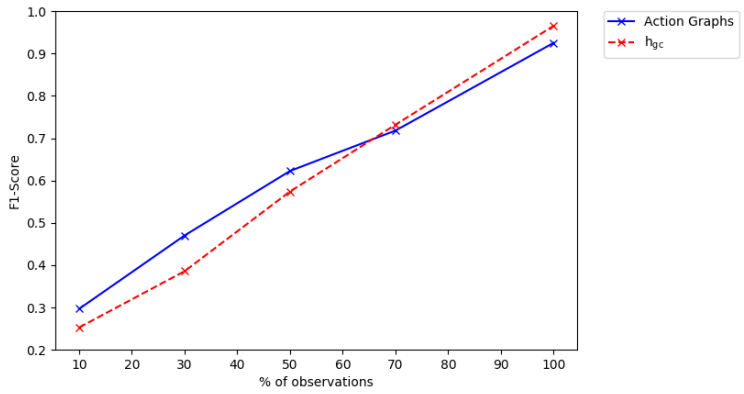


Figure 2.8: Average F1-Score, after the first 10, 30, 50, 70 and 100% of observations have been processed, for our Action Graph approach and h_{gc} by Pereira et al. [19].

Table 2.4: Accuracy results for the dataset containing the first 10 %, 30 %, 50 %, 70 % and 100 % of observations. AG1 is the first update rule, AG2 the second update rule and AG3 is the combination of the two rules (see Section 2.6).

			AG1				AG2				AG3				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
Blocks-World	20.28	10	9.65	0.52	0.52	**0.05**	20.01	0.05	**1.00**	**0.05**	9.65	0.52	0.52	**0.05**	**1.80**	**0.87**	0.10	**0.05**
		30	4.08	0.79	0.40	0.11	20.01	0.05	**1.00**	0.05	4.08	0.79	0.40	0.11	**1.22**	**0.91**	0.22	**0.18**
		50	1.99	0.89	0.36	0.20	20.01	0.05	**1.00**	0.05	1.99	0.89	0.36	0.20	**1.34**	**0.91**	0.32	**0.27**
		70	1.35	**0.94**	0.53	0.45	19.99	0.05	**1.00**	0.05	1.35	**0.94**	0.53	0.45	**1.27**	**0.94**	0.58	**0.49**
		100	**1.10**	**0.99**	0.90	**0.87**	19.91	0.05	**0.99**	0.05	**1.10**	**0.99**	0.90	**0.87**	1.36	0.98	1.00	0.86
Campus	2.00	10	**1.27**	**0.87**	**1.00**	**0.87**	2.00	0.50	**1.00**	0.50	**1.27**	**0.87**	**1.00**	**0.87**	**1.27**	0.60	0.73	0.60
		30	**1.00**	**1.00**	**1.00**	**1.00**	1.80	0.40	0.80	0.40	**1.00**	**1.00**	**1.00**	**1.00**	1.07	0.97	**1.00**	0.97
		50	**1.00**	0.93	0.93	0.93	1.13	0.07	0.13	0.07	**1.00**	0.93	0.93	0.93	**1.00**	**1.00**	**1.00**	**1.00**
		70	**1.00**	0.80	0.80	0.80	1.13	0.07	0.13	0.07	**1.00**	0.80	0.80	0.80	**1.00**	**1.00**	**1.00**	**1.00**
		100	**1.00**	0.87	0.87	0.87	1.13	0.13	0.20	0.13	**1.00**	0.80	0.80	0.80	**1.00**	**1.00**	**1.00**	**1.00**
Depots	8.86	10	5.71	0.40	0.68	**0.14**	8.86	0.11	**1.00**	0.11	5.71	0.40	0.68	**0.14**	**1.61**	**0.75**	0.21	0.13
		30	3.32	0.61	0.39	**0.22**	8.86	0.11	**1.00**	0.11	3.32	0.61	0.39	**0.22**	**1.71**	**0.76**	0.29	0.21
		50	1.96	0.79	0.54	0.41	8.86	0.11	**1.00**	0.11	1.96	0.79	0.54	0.41	**1.36**	**0.85**	0.50	**0.42**
		70	1.29	0.89	0.68	0.59	8.86	0.11	**1.00**	0.11	1.29	0.89	0.68	0.59	**1.25**	**0.93**	0.82	**0.72**
		100	1.14	0.98	**1.00**	0.96	8.86	0.11	**1.00**	0.11	1.14	0.98	**1.00**	0.96	**1.04**	**1.00**	**1.00**	**0.98**
Driver-log	7.14	10	3.93	0.49	0.71	0.20	7.04	0.16	**1.00**	0.15	3.93	0.49	0.71	0.20	**1.64**	**0.71**	0.29	**0.22**
		30	2.11	0.73	0.57	**0.38**	6.61	0.23	**1.00**	0.18	2.11	0.73	0.57	**0.38**	**1.21**	**0.80**	0.43	0.37
		50	1.82	0.77	0.61	0.51	6.61	0.23	**1.00**	0.18	1.79	0.79	0.64	0.53	**1.21**	**0.87**	0.64	**0.55**
		70	1.50	0.86	0.75	0.69	6.61	0.23	**1.00**	0.18	1.50	0.86	0.75	0.69	**1.18**	**0.91**	0.75	**0.71**
		100	1.11	0.96	0.93	0.89	5.54	0.34	**1.00**	0.24	**1.07**	**0.97**	0.93	**0.90**	1.21	**0.97**	**1.00**	**0.90**
DWR	7.29	10	3.00	0.61	0.57	0.21	7.29	0.14	**1.00**	0.14	3.00	0.61	0.57	0.21	**1.21**	**0.82**	0.43	**0.38**
		30	1.71	0.78	0.54	0.36	7.29	0.14	**1.00**	0.14	1.71	0.78	0.54	0.36	**1.14**	**0.87**	0.57	**0.54**
		50	1.25	0.84	0.54	0.43	7.29	0.14	**1.00**	0.14	1.25	0.84	0.54	0.43	**1.11**	**0.86**	0.54	**0.50**
		70	1.14	0.87	0.57	0.50	7.29	0.14	**1.00**	0.14	1.14	0.87	0.57	0.50	**1.11**	**0.89**	0.64	**0.59**
		100	**1.00**	**0.99**	0.96	**0.96**	7.29	0.14	**1.00**	0.14	**1.00**	**0.99**	0.96	**0.96**	**1.00**	0.98	0.93	0.93
Easy-IPC-Grid	8.36	10	1.64	0.75	0.30	**0.21**	5.80	0.34	**0.74**	0.12	**1.51**	**0.76**	0.26	0.20	3.62	0.58	0.51	0.20
		30	**1.43**	0.76	0.25	0.19	3.16	0.61	**0.52**	0.22	1.52	**0.77**	0.33	**0.27**	2.85	0.66	0.44	0.21
		50	1.59	0.76	0.33	0.28	1.85	0.79	**0.61**	0.46	**1.34**	**0.87**	**0.61**	**0.56**	2.70	0.69	0.51	0.32
		70	1.20	0.85	0.44	0.44	1.85	0.83	**0.72**	**0.58**	**1.10**	**0.89**	0.57	0.57	2.57	0.74	0.64	0.45
		100	1.07	0.97	0.87	0.87	**1.00**	**1.00**	0.98	0.98	**1.00**	0.98	0.90	0.90	**1.00**	**1.00**	**1.00**	**1.00**

Continued on next page...

... Table 2.4 continued.

			AG1				AG2				AG3				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
Ferry	7.57	10	6.54	0.26	**1.00**	0.19	7.57	0.13	**1.00**	0.13	6.54	0.26	**1.00**	0.19	**1.79**	**0.76**	0.54	**0.34**
		30	1.93	0.87	**1.00**	**0.70**	7.57	0.13	**1.00**	0.13	1.93	0.87	**1.00**	**0.70**	**1.29**	**0.89**	0.75	0.67
		50	1.25	**0.97**	**1.00**	**0.88**	7.57	0.13	**1.00**	0.13	1.25	**0.97**	**1.00**	**0.88**	**1.04**	**0.97**	0.89	**0.88**
		70	1.11	0.98	**1.00**	0.95	7.57	0.13	**1.00**	0.13	1.11	0.98	**1.00**	0.95	**1.00**	**0.99**	0.96	**0.96**
		100	1.07	**0.99**	**1.00**	**0.96**	7.57	0.13	**1.00**	0.13	1.07	**0.99**	**1.00**	**0.96**	**1.00**	**0.99**	0.96	**0.96**
Intrusion-Detection	16.67	10	**1.00**	**0.88**	0.07	0.07	6.27	0.68	**1.00**	**0.18**	**1.00**	**0.88**	0.07	0.07	1.91	0.86	0.31	0.16
		30	2.89	**0.88**	**1.00**	**0.41**	2.89	**0.88**	**1.00**	**0.41**	2.89	**0.88**	**1.00**	**0.41**	**1.62**	0.87	0.31	0.20
		50	1.51	**0.97**	**1.00**	**0.78**	1.51	**0.97**	**1.00**	**0.78**	1.51	**0.97**	**1.00**	**0.78**	**1.40**	0.96	0.89	0.70
		70	**1.04**	**1.00**	**1.00**	**0.98**	1.09	0.99	**1.00**	0.96	**1.04**	**1.00**	**1.00**	**0.98**	**1.04**	**1.00**	**1.00**	**0.98**
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Kitchen	3.00	10	**2.00**	**0.67**	**1.00**	**0.63**	**2.00**	**0.67**	**1.00**	**0.63**	**2.00**	**0.67**	**1.00**	**0.63**	2.87	0.38	**1.00**	0.38
		30	**1.40**	**0.87**	**1.00**	**0.80**	**1.40**	**0.87**	**1.00**	**0.80**	**1.40**	**0.87**	**1.00**	**0.80**	2.87	0.38	**1.00**	0.38
		50	**1.33**	**0.89**	**1.00**	**0.83**	**1.33**	**0.89**	**1.00**	**0.83**	**1.33**	**0.89**	**1.00**	**0.83**	2.87	0.38	**1.00**	0.38
		70	**1.33**	**0.89**	**1.00**	**0.83**	**1.33**	**0.89**	**1.00**	**0.83**	**1.33**	**0.89**	**1.00**	**0.83**	2.33	0.56	**1.00**	0.56
		100	**1.13**	**0.96**	**1.00**	**0.93**	**1.13**	**0.96**	**1.00**	**0.93**	**1.13**	**0.96**	**1.00**	**0.93**	1.93	0.69	**1.00**	0.69
Logistics	10.40	10	4.41	0.68	**1.00**	**0.31**	4.41	0.68	**1.00**	**0.31**	4.41	0.68	**1.00**	**0.31**	**2.80**	**0.69**	0.33	0.15
		30	2.69	**0.83**	**1.00**	**0.51**	2.72	**0.83**	**1.00**	0.50	2.69	**0.83**	**1.00**	**0.51**	**1.89**	0.78	0.33	0.22
		50	1.80	**0.92**	**1.00**	**0.66**	1.90	0.91	**1.00**	0.64	1.80	**0.92**	**1.00**	**0.66**	**1.72**	0.86	0.66	0.44
		70	1.67	**0.93**	**1.00**	**0.71**	1.74	**0.93**	**1.00**	0.68	1.67	**0.93**	**1.00**	**0.71**	**1.51**	0.91	0.80	0.58
		100	**1.00**	**1.00**	**1.00**	**1.00**	1.64	0.94	**1.00**	0.71	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Miconic	6.00	10	2.96	0.61	**0.82**	**0.31**	3.29	0.56	**0.82**	0.26	3.00	0.61	**0.82**	**0.31**	**2.00**	**0.67**	0.50	0.27
		30	1.61	0.88	**0.93**	0.69	1.75	0.82	0.82	0.53	1.61	0.85	0.86	0.62	**1.32**	**0.90**	0.86	**0.73**
		50	1.18	**0.97**	**1.00**	**0.92**	1.21	0.94	0.93	0.83	1.18	**0.97**	**1.00**	**0.92**	**1.07**	0.94	0.86	0.82
		70	1.07	**0.99**	**1.00**	**0.96**	1.18	0.96	0.96	0.89	1.07	**0.99**	**1.00**	**0.96**	**1.04**	0.98	0.96	0.95
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	1.0
Rovers	6.00	10	3.64	0.54	0.93	**0.32**	4.61	0.40	**1.00**	0.30	3.64	0.54	0.93	**0.32**	**2.00**	**0.65**	0.46	0.24
		30	1.61	**0.88**	**0.93**	**0.70**	2.25	0.77	**0.93**	0.62	1.61	**0.88**	**0.93**	**0.70**	**1.14**	**0.88**	0.71	0.66
		50	**1.11**	**0.96**	0.93	**0.88**	1.64	0.89	**1.00**	0.73	**1.11**	**0.96**	0.93	**0.88**	1.14	0.94	0.89	0.83
		70	**1.07**	**0.99**	**1.00**	**0.96**	1.18	0.97	**1.00**	0.91	**1.07**	**0.99**	**1.00**	**0.96**	1.11	0.97	0.96	0.93
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Satellite	6.43	10	4.00	0.51	**1.00**	0.39	6.25	0.19	**1.00**	0.16	4.00	0.51	**1.00**	0.39	**2.11**	**0.71**	0.68	**0.42**
		30	3.43	0.60	**1.00**	0.49	5.36	0.33	**1.00**	0.25	3.43	0.60	**1.00**	0.49	**1.96**	**0.76**	0.75	**0.51**
		50	1.43	0.90	0.89	0.71	2.21	0.81	**1.00**	0.64	1.43	0.90	0.89	0.71	**1.21**	**0.92**	0.86	**0.77**
		70	1.11	0.93	0.86	0.80	1.46	0.92	**1.00**	0.85	1.11	0.93	0.86	0.80	**1.04**	**0.96**	0.89	**0.88**
		100	**1.04**	**0.99**	**1.00**	**0.98**	1.11	0.98	**1.00**	0.95	**1.04**	**0.99**	**1.00**	**0.98**	1.07	**0.99**	**1.00**	0.96
Sokoban	7.14	10	**1.57**	**0.68**	0.18	0.12	7.00	0.15	**1.00**	0.15	**1.57**	**0.68**	0.18	0.12	2.36	0.63	0.43	**0.17**
		30	**1.14**	**0.77**	0.32	**0.30**	6.82	0.16	**0.96**	0.14	**1.14**	**0.77**	0.32	**0.30**	1.89	0.66	0.32	0.20
		50	**1.11**	0.81	0.39	0.38	6.82	0.16	**0.96**	0.14	**1.11**	0.81	0.39	0.38	1.21	**0.87**	0.64	**0.57**
		70	**1.04**	0.81	0.39	0.39	6.82	0.16	**0.96**	0.14	**1.04**	0.81	0.39	0.39	1.14	**0.91**	0.75	**0.70**
		100	1.04	0.84	0.50	0.50	6.64	0.18	0.93	0.13	1.04	0.84	0.50	0.50	**1.00**	**1.00**	**1.00**	**1.00**
Zeno-Travel	6.86	10	4.14	0.49	0.79	**0.32**	6.86	0.15	**1.00**	0.15	4.14	0.49	0.79	**0.32**	**1.50**	**0.73**	0.32	0.24
		30	2.96	0.66	0.71	**0.43**	6.86	0.15	**1.00**	0.15	2.96	0.66	0.71	**0.43**	**1.61**	**0.77**	0.54	0.38
		50	1.82	0.84	0.82	0.61	6.79	0.16	**1.00**	0.15	1.82	0.84	0.82	0.61	**1.36**	**0.89**	0.79	**0.70**
		70	1.14	0.96	0.93	0.86	6.79	0.16	**1.00**	0.15	1.14	0.96	0.93	0.86	**1.11**	**0.97**	0.96	**0.95**
		100	**1.00**	**1.00**	**1.00**	**1.00**	6.57	0.20	**1.00**	0.16	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
ALL	8.26	10	3.70	0.60	0.70	**0.29**	6.62	0.33	**0.97**	0.22	3.69	0.60	0.70	**0.29**	**2.03**	**0.69**	0.46	0.26
		30	2.22	**0.79**	0.74	**0.49**	5.69	0.43	**0.94**	0.31	2.23	**0.79**	0.74	**0.49**	**1.65**	**0.79**	0.57	0.43
		50	1.48	0.88	0.76	0.63	5.12	0.48	**0.91**	0.39	1.46	**0.89**	0.78	**0.65**	**1.45**	0.86	0.73	0.61
		70	**1.20**	0.91	0.80	0.73	4.99	0.50	**0.92**	0.44	**1.20**	**0.92**	0.81	0.74	1.31	0.91	0.85	**0.76**
		100	1.05	**0.97**	0.94	0.92	4.76	0.54	0.94	0.51	**1.04**	**0.97**	0.93	0.92	1.11	**0.97**	**0.99**	**0.95**

is not captured by the Action Graph structure. For instance, in a Blocks-World problem, picking up `blockA` requires the gripper to be empty by putting down all blocks (including `blockA`). The graph structure captures the dependencies of actions; however, does not account for the prior state(s) of the environment, e.g., the gripper could already be empty.

AG2 only outperformed AG1 on the Easy-IPC-Grid domain. In this domain

there are strong constraints on the order in which actions are performed and a false goal could be traversed on-route to the real goal. Therefore, for this domain, update rule 2 prevented the shortest plan being favoured and successfully increased the probabilities of the goals the observed agent was navigating towards. Nevertheless, this update rule could not determine the real goal for the majority of domains. This is because all goals, whose plans contain the observed action, were multiplied (increased) by the same amount when the current and previous observations were not connected via a `DEP` (or `ORDERED-AND`) node. All suboptimal plans are encoded in an Action Graph's structure; therefore, for many domains, all actions are included within a plan to reach any of the goals. Combining AG2 with AG1 increased the results of the Easy-IPC-Grid domain without greatly affecting the results produced for the other domains. The subsequent sections just show the results of AG3.

2.7.2.4 Missing Observations Results Discussion

Our Action Graph approach and h_{gc} were also ran on the 6313 GR problems[4] produced by Pereira et al. [19] and Ramírez and Geffner [20], that contain missing observations. These problems contain a random 10, 30, 50, 70 and 100 % of observations. The F1-Scores are depicted in Figure 2.9 and the table containing the results per domain can be found in Appendix 2.D

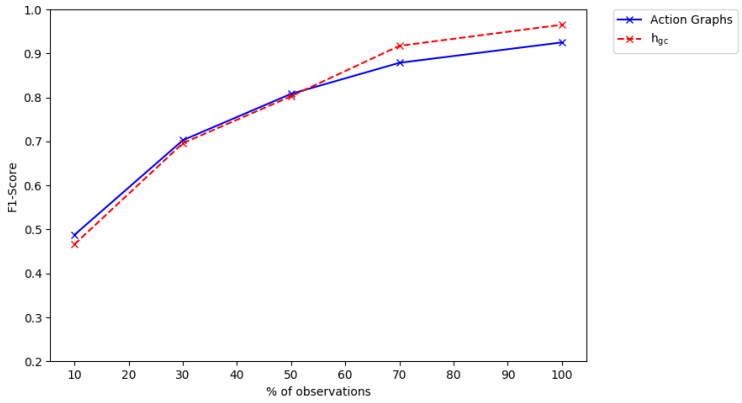


Figure 2.9: Average F1-Score produced by our Action Graph approach and h_{gc} by Pereira et al. [19] on the dataset containing missing observations (in other words when 10, 30, 50, 70 and 100 % of actions had been observed).

These results show a similar trend to the previous experiment, i.e., our approach produced a higher F1-Score than h_{gc} at 10 %, 30 % and 50 % of observations (and vice versa after 70 % and 100 % of observations). Both approaches perform better on the dataset containing missing observations,

than they did in the previous experiment. This is because each GR problem could contain observations that are close to the goal (due to random actions in a plan having been selected).

2.7.3 Goal Recognition with an Inaccurate Initial State

The main aim of our approach is to be able to perform GR when the initial state of the environment is defined inaccurately. A fluent's value could be incorrect if it is unknown, and thus incorrectly guessed, or an error has been made while determining the environment's state. Therefore, a dataset containing differing percentages, i.e., 10, 20, 40, 60, 80 and 100 %, of fluents set to incorrect values was produced. How this dataset was generated is discussed, followed by the results produced by our Action Graph approach and h_{gc}.

2.7.3.1 Setup

A dataset containing problems with varying amounts of fluents set to incorrect values was generated from the dataset containing the first N % of observations, i.e., that was used in the experiments of Section 2.7.2.3. For each problem, contained in the aforementioned dataset, 10, 20, 40, 60, 80 and 100 % of fluents were chosen at random and their value set to a randomly selected incorrect value. As there are elements of randomness, for each percentage of fluents, 5 problems were created. The changes that can be made to the initial state I were (manually) defined based on the actions' effects.

For instance, in a Zeno-Travel problem, containing 5 cities, 4 people, 3 aircraft and 2 fuel-levels, there are 10 fluents whose initial value can be altered. Each person can be at a city or in an aircraft; thus, a fluent indicating a person's location can be changed to one of the 7 alternative (incorrect) values. Each aircraft has two fluents associated with it, i.e., is in a city and has a fuel-level; both of these can be changed to an alternative value.

These state changes could cause some (or all) goals to be unreachable from the defined initial state (e.g., in the Sokoban domain, the robot could be unable to navigate to a location from which one of the boxes can be pushed) and the initial state itself could be invalid/contradictory (e.g., in the Blocks-World domain, `blockA`'s fluent could express the block is on the table and the gripper's fluent could indicate it is holding `blockA`). The changes made to the initial state are outlined in Appendix 2.C and a detailed table of possible changes can be found at https://doi.org/10.5281/zenodo.3889672.

2.7.3.2 Results Discussion

The accuracy of our approach was not affected by setting fluents in the initial state to incorrect values, whereas the accuracy of h_{gc} greatly reduced (see Figure 2.10 and Appendix 2.E). When 20 % of the fluents' were set to incorrect values, a large decrease in the accuracy of h_{gc} was observed, and as this percentage was increased, the accuracy further reduced. For several domains, i.e., Kitchen, Rovers and Intrusion-Detection, the resulting M and R of h_{gc} rose when 100 % of fluents (rather than 80 %) were incorrect. This is because at 100 %, for these domains, all goals were contained in the set of candidate goals, and thus the real goal was contained within $\mathcal{C}$. Other approaches to GR are also unable to handle inaccurate initial states because they attempt to find the plans/states that reach each goal from the defined initial world state. These are discussed further in the related work section.

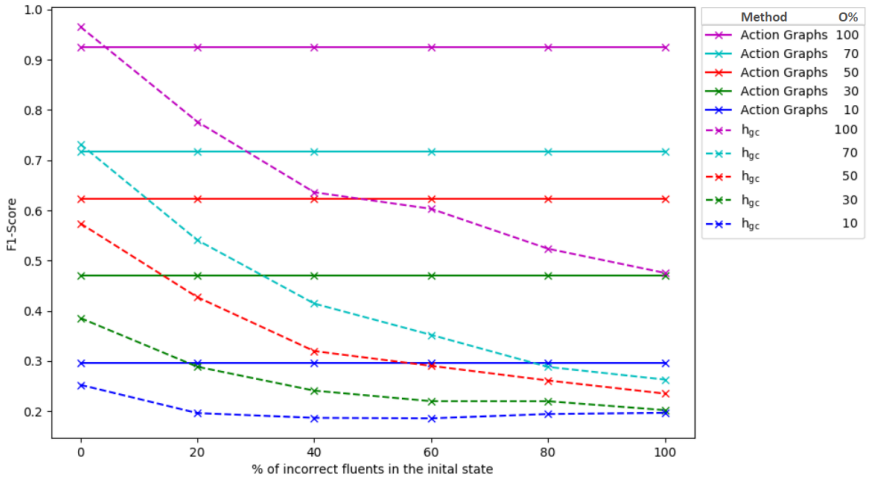


Figure 2.10: Graph showing the effect increasing the amount of incorrect fluents in the initial state had on the accuracy of our Action Graph approach and h_{gc} by Pereira et al. [19]. Our Action Graph approach is indicated by solid lines, the dash lines show the approach of Pereira et al. [19]. Each line colour indicates a different % of observations.

2.8 Related Work

Methods for intention recognition can be broadly categorised as data-driven and knowledge-driven (i.e., symbolic) methods [44, 45]. Data-driven approaches train a recognition model from a large dataset [45–48]. The main disadvantages of this method are that often a large amount of labelled training data is required and the produced models often only work on data

similar to the training set [49, 50]. Since our work belongs to the category of knowledge-driven methods, data-driven methods are not further discussed.

Knowledge-driven approaches rely on a logical description of the actions agents can perform. They can be further divided into approaches that parse a library of plans (also known as "recognition as parsing"), and approaches that solve recognition problems, defined in languages usually associated with planning, i.e., "recognition as planning" [51]. Our GR approach derives a graph structure, similar to those used by some recognition as parsing methods, from a PDDL defined (planning-based) GR problem. Recognition as planning is often viewed as more flexible and general because a library of plans is not required and cyclic plans are difficult to compile into a library [20]. We chose to transform a PDDL planning problem into an Action Graph to enable the goal probabilities to be updated quickly, all plans (including suboptimal plans) to be represented, cyclic plans to be expressed and inaccurate initial states to be handle. Our approach takes advantage of the fact that a perfect/complete representation of plans is not required to perform GR. In this section, recognition as parsing and recognition as planning approaches are discussed in turn.

2.8.1 Recognition as Parsing

In recognition as parsing, hierarchical structures are usually developed which include abstract actions along with how they are decomposed to concrete (observable) actions [35]. Several prior approaches have represented these hierarchical structures as AND/OR trees [34, 52]. As previously mentioned, our graph structure was inspired by these works. The recognition as parsing approaches, mentioned in this section, enable both the goal and plan of the observed agent to be recognised but do not mention handling invalid initial states or suboptimal plans.

Kautz et al. [35, 53] introduce a language to describe a hierarchy of actions. Based on which low level actions are observed, the higher level task(s) an agent is attempting to achieve is inferred. Their paper presents one of the earliest plan/goal recognition formal theories that aimed to handle simultaneous action execution, multi-plan recognition and missing observations.

A set of action sequence graphs is derived from a library of plans in [54]. This set is compared to an action sequence graph, created from a sequence of observations, to find the plan most similar to the observation sequence. Their approach was shown to perform well on misclassified (incorrect) sensor observations and missing actions; but to generate the library of plans a

planner is called, and thus a known initial state is required.

2.8.2 Recognition as Planning

Recognition as planning is a more recently proposed approach, in which languages normally associated with task planning, such as STRIPS [55] and PDDL [28], define the actions agents can perform (along with their preconditions and effects) and world states. In recognition as parsing there are usually only action definitions, whereas planning-based approaches allow for the inclusion of state knowledge, such as what objects are found within the environment and their locations.

In [20, 39], it was proposed to view goal recognition as the inverse of planning. To find the difference in the cost of the plan to reach the goal with and without taking the observations into consideration, a planner is called twice for every possible goal. Therefore, the performance would greatly deteriorate when exposed to inaccurate initial states. In [56], the work from [20] was extended, to find the joint probability of pairs of goals rather than a single goal. Their work aimed to handle multiple interleaving goals. Although initial approaches were computationally expensive as they required a task planner to be called multiple times [20, 39, 56], the latest advances in recognition as planning algorithms have greatly improved this [19, 42].

Plan graphs were proposed in [42]. A plan graph, which contains actions and propositions labelled as either true, false or unknown, is built from a planning problem and updated based on the observations. Rather than calling a planner, the graph is used to calculate the cost of reaching the goals. Our Action Graph structure differs greatly from a plan graph, as Action Graphs only contain actions and the constraints between those actions.

More recently Pereira et al. [19, 21] significantly reduced the recognition time by finding landmarks. Our experiments show a comparison to this approach. This work has been expanded to handle incomplete domain models [27], i.e., GR problems with incomplete preconditions and effects. In future work, we will explore applying our work to incomplete domain models.

2.9 Conclusion

Our novel approach to goal recognition aims to handle problems in which the defined initial state is inaccurate. An inaccurate initial state contains fluents whose value is unknown and/or incorrect. For instance, if an item or agent (e.g., cup or human) is occluded its location is indeterminable,

and thus possibly defined incorrectly. Our approach transforms a PDDL defined GR problem into an Action Graph, which models the order constraints between actions. Each node is labelled with the minimum number of `DEP` and `ORDERED-AND` nodes, traversed to reach it from each goal. When an action is observed, the goals' probability is updated based on either the distance the action's associated node is from the goals or, if the current and prior observation are connected via a `DEP` or `ORDERED-AND` node, the change in distance. Experiments proved that when the fluents have incorrect values in the initial state, e.g., because they are unknown or sensed/determined incorrectly, the performance of our approach is unaffected.

In future work, we intend to apply our Action Graph method to further challenges associated with symbolic GR. As well as the defined initial state being inaccurate, the domain model (i.e., action definitions) could be incorrect [27]. Therefore, we will experiment with adapting the Action Graph structure based on the order observations are received. To create a more compact structure, and thus reduce the computational time of this, we will investigate grouping related actions into a single node. For instance, in the Sokoban domain, the same actions can be performed on both `box1` and `box2`; therefore, actions, such as `push(box1 loc1 loc2)` and `push(box2 loc1 loc2)`, can be grouped into a single node. Moreover, we intend to apply our GR approach to problems in which either the observed agent has multiple goals, or multiple agents have individual or joint goals [57].

As developing the PDDL can be time consuming and challenging, researchers have attempted to replace this manual process, with deep learning methods [46, 58]. We will explore the potential of learning the Action Graph structure from pairs of images, and then converting the Action Graph into a PDDL defined domain. Which, subsequently, could be provided as input to task planners as well as goal recognisers.

Appendix 2.A Algorithm to Set the Nodes' Distance From Each Goal

The pseudo-code for the BFT that labels the nodes with their distance from each goal, described in Section 2.5, is provided in Algorithm 2.3.

Algorithm 2.3 Nodes' distance from each goal initialisation.

Data: A_g set of Action Goal nodes, Action Graph
Result: Action Graph with nodes' distance from each goal

```
1: for each a_g ∈ A_g do
2:     setNodeValueBFT(a_g.parent, 0, a_g.goal)
3: end for
4: function SETNODEVALUEBFT(node, startCount, G)
5:     queue = ∅
6:     queue.push(BftNode(node, startCount, null))
7:     while queue ≠ ∅ do
8:         currentNode, count, o-and = queue.pop()
9:         if currentNode.hasDisFromGoal(G, o-and) and
   currentNode.getDisFromGoal(G, o-and) ≤ count then continue
10:        end if
11:        currentNode.setDisFromGoal(count, G, o-and)
12:        if currentNode.type is DEPnode() or OANDnode() then
13:            count = count + 1
14:        end if
15:        for each child ∈ currentNode.children do
16:            if child.getDisFromGoal(G, o-and) ≤ count then continue
17:            end if
18:            if child.type is actionNode then
19:                child.setDisFromGoal(count, G, o-and)
20:            else
21:                if child.type is UANDnode() then
22:                    queue.push(BftNode(child, count, null))
23:                else if currentNode.type is OANDnode() and
   currentNode.children[1] = child then
24:                    queue.push(BftNode(child, count, currentNode))
25:                else
26:                    queue.push(BftNode(child, count, o-and))
27:                end if
28:            end if
29:        end for
30:    end while
31: end function
```

Appendix 2.B Algorithms for Updating the Goal Probabilities

The pseudo-code for update rules 1 and 2 (i.e., AG1 and AG2) are provided in Algorithms 2.4 and 2.5, respectively. These algorithms were described in Section 2.6.

Algorithm 2.4 Update rule 1 (AG1).

Data: o^t observed action, o^{t-1} previously observed action, $\mathcal{G}$ set of hypothesis goals with current probability
Result: $\mathcal{G}$ set of hypothesis goals with updated probability

for each $G \in \mathcal{G}$ **do**
  $c(G) = \frac{o^t.\text{getDisFromGoal}(G)}{\sum_{G' \in \mathcal{G}} o^t.\text{getDisFromGoal}(G')}$
  $v(G) = P(G)(1 + c(G))$
end for
for each $G \in \mathcal{G}$ **do**
  $P(G) = \frac{v(G)}{\sum_{G' \in \mathcal{G}} v(G')}$ ▷ probabilities sum to 1
end for
updateNodeDistancesIfo-andLeftBranchFullyObserved(o^t)

Algorithm 2.5 Update rule 2 (AG2).

Data: o^t observed action, o^{t-1} previously observed action, $\mathcal{G}$ set of hypothesis goals with current probability
Result: $\mathcal{G}$ set of hypothesis goals with updated probability

if $o^{t-1} \neq null$ **and** areConnectedViaDep/OAndNodes(o^t, o^{t-1}) **then**
  $\forall G \in \mathcal{G} : v(G) = P(G)$
  for each $G \in \{G' \mid G' \in \mathcal{G}, o^t.\text{hasDisFromGoal}(G')\}$ **do**
    $c(G) = \sigma(o^{t-1}.\text{getDisFromGoal}(G) - o^t.\text{getDisFromGoal}(G))$
    $v(G) = P(G)(1 + c(G))$
  end for
else
  $\forall G \in \mathcal{G} : v(G) = P(G)$
  for each $G \in \{G' \mid G' \in \mathcal{G}, o^t.\text{hasDisFromGoal}(G')\}$ **do**
    $v(G) = P(G)(1 + 0.5)$ ▷ i.e., $c(G) = 0.5$
  end for
end if
for each $G \in \mathcal{G}$ **do**
  $P(G) = \frac{v(G)}{\sum_{G' \in \mathcal{G}} v(G')}$ ▷ probabilities sum to 1
end for
updateNodeDistancesIfo-andLeftBranchFullyObserved(o^t)

Appendix 2.C Domains

The list below describes each domain in turn, and describes the fluents whose initial value was modified to generate the dataset for the experiments described in Section 2.7.3. Further details on the modified fluents are provided at https://doi.org/10.5281/zenodo.3889672. These GR domains were produced by [19, 20][4], based on the work of [16] and the IPC domains[1].

- `Blocks-World`: A hand stacks blocks on top of one another to create a tower.
  - A block can be placed on the table or another block, a block can become unclear, and the hand can be empty or holding a block.
- `Campus`: In the campus domain a university student navigates to different locations (e.g., the library, a cafe, ect.) to perform different activities.
  - The student's location can change and if an activity has been performed (or not) can be modified.
- `Depots`: In this domain, crates are relocated by trucks and hoists.
  - Crates can change location, a crate could be on another create or in a truck, and a hoist could be lifting a crate or available.
- `Driverlog`: Trucks are driven by different drives, so that objects can be relocated.
  - Trucks, drivers and objects can change location, an object could be in a truck, and a driver can be driving a truck.
- `DWR`: Robots and cranes relocate containers, which are piled on top of each other.
  - The location of a robot, which pile a container is in (and/or on top of), if a container has been loaded onto a robot and if a crane is holding a container can be changed.
- `Easy-IPC-Grid`: A robot navigates a grid to reach a goal location, and along the way must collect keys to unlock locations.
  - A location can be unlocked, and the location of the robot and if a key is being carried can be modified.
- `Ferry`: A ferry transports cars to different locations.
  - The ferry's location and if a car is at a location or on the ferry can be changed.
- `Intrusion-Detection`: An intruder accesses, modifies and downloads information from a computer system.
  - What information has been accessed, modified or downloaded can be changed.

- `Kitchen`: The observed agent takes different items and performs kitchen-based activities (e.g., makes toast). Note that: “activities" are not included in the list of observations.
  - Which items have been taken, what equipment has been used and what activities have been performed can be changed.
- `Logistics`: Packages are transported to different locations and airports by air planes and trucks.
  - Which airport, location, truck or airplane a package is at/in; a truck's airport/location, and an airplane's airport can be changed.
- `Miconic`: A lift takes different passages to there desired floor.
  - The floor the lift starts on, which passages are in the lift and which passages have been served can be changed.
- `Rovers`: A rover navigates a planet collecting rock samples, soil samples and images.
  - The location of the rover, if a camera has been calibrated, if the robots store is empty/full, what data has been collected, which data has been communicated and if a channel is free can be modified.
- `Satellite`: In the satellite domain, satellites take images in different modes and directions.
  - Which direction a satellite is pointing in, if the power is being availed, if the power is on, if the an instrument is calibrated and if an image has been taken can be modified.
- `Sokoban`: A robot must push two boxes to different locations.
  - The location of the robot and the boxes can be modified.
- `Zeno-travel`: People travel on aircraft, to reach different cities.
  - Which city/aircraft a person is in/at, which city an aircraft is in and an aircraft's fuel level can be changed.

Appendix 2.D Missing Observations Detailed Experimental Results

Our Action Graph approach to goal recognition (GR) was compared to the goal completion heuristic, namely, h_{gc}, by Pereira et al. [19, 27] on a dataset created by [20] and [19], which contains missing observations[5]. The results of this experiment are discussed in Section 2.7.2.4. A breakdown, per domain, of the number of candidate goals $|\mathcal{C}|$, quality/accuracy Q, recall R and precision M is provided in Table 2.5.

[5]https://github.com/pucrs-automated-planning/goal-plan-recognition-dataset

Table 2.5: Results of our Action Graph approach and h_{gc} by Pereira et al. [19], for the original dataset by [20] and [19], which contains missing observations; i.e., 10%/30%/50%/70% of observations picked at random.

			Action Graphs				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
Blocks-World	20.28	10	6.55	0.70	**0.78**	0.16	**1.26**	**0.93**	0.44	**0.38**
		30	2.11	0.91	**0.67**	0.42	**1.17**	**0.95**	0.56	**0.51**
		50	1.59	0.94	**0.72**	0.58	**1.13**	**0.96**	0.63	**0.59**
		70	1.28	0.97	0.83	0.74	**1.15**	**0.98**	**0.84**	**0.78**
		100	**1.10**	**0.99**	0.90	**0.87**	1.36	0.98	**1.00**	0.86
Campus	2.00	10	1.47	0.57	0.80	0.57	**1.13**	**0.80**	**0.87**	**0.80**
		30	1.20	0.63	0.73	0.63	**1.13**	**0.80**	**0.87**	**0.80**
		50	**1.07**	0.70	0.73	0.70	1.13	**0.87**	**0.93**	**0.87**
		70	1.13	0.80	0.87	0.80	**1.00**	**1.00**	**1.00**	**1.00**
		100	**1.00**	0.80	0.80	0.80	**1.00**	**1.00**	**1.00**	**1.00**
Depots	8.86	10	3.02	0.74	**0.83**	**0.46**	**1.31**	**0.83**	0.39	0.35
		30	1.63	0.90	**0.89**	**0.70**	**1.15**	**0.91**	0.67	0.61
		50	1.23	**0.96**	**0.95**	**0.89**	**1.11**	0.95	0.85	0.80
		70	1.14	0.98	**0.98**	**0.93**	**1.01**	**0.99**	0.94	**0.93**
		100	1.14	0.98	**1.00**	0.96	**1.04**	**1.00**	**1.00**	**0.98**
Driverlog	7.14	10	2.36	0.71	**0.63**	0.39	**1.29**	**0.80**	0.45	**0.41**
		30	1.42	0.84	**0.64**	**0.55**	**1.24**	**0.85**	0.60	0.52
		50	**1.24**	**0.91**	**0.80**	**0.76**	1.29	0.90	0.77	0.67
		70	**1.12**	0.94	0.86	0.81	1.24	**0.95**	**0.93**	**0.84**
		100	**1.07**	**0.97**	0.93	**0.90**	1.21	**0.97**	**1.00**	**0.90**
DWR	7.29	10	3.06	0.64	**0.71**	0.28	**1.20**	**0.80**	0.38	**0.35**
		30	1.32	0.87	**0.68**	0.56	**1.10**	**0.89**	0.64	**0.61**
		50	1.14	**0.93**	**0.80**	**0.74**	**1.06**	0.91	0.73	0.70
		70	**1.04**	0.94	0.81	0.79	1.05	**0.97**	**0.90**	**0.88**
		100	**1.00**	**0.99**	**0.96**	**0.96**	**1.00**	0.98	0.93	0.93
Easy-IPC-Grid	8.66	10	**1.50**	**0.85**	0.59	**0.51**	2.58	0.75	**0.67**	0.45
		30	**1.16**	**0.92**	0.75	**0.72**	1.65	0.89	**0.82**	0.69
		50	**1.05**	0.95	0.80	0.80	1.18	**0.96**	**0.91**	**0.87**
		70	**1.03**	0.96	0.84	0.84	1.07	**0.99**	**0.97**	**0.96**
		100	**1.00**	0.98	0.90	0.90	**1.00**	**1.00**	**1.00**	**1.00**
Ferry	7.57	10	3.80	0.62	**1.00**	0.46	**1.45**	**0.84**	0.64	**0.53**
		30	1.85	0.88	**1.00**	0.75	**1.15**	**0.94**	0.86	**0.81**
		50	1.39	0.94	**1.00**	0.86	**1.07**	**0.97**	0.94	**0.91**
		70	1.10	0.98	**1.00**	**0.96**	**1.00**	**0.99**	0.96	**0.96**
		100	1.07	**0.99**	**1.00**	**0.96**	**1.00**	**0.99**	0.96	**0.96**
Intrusion-Detection	16.67	10	1.83	0.92	**0.80**	0.56	**1.37**	**0.94**	0.74	**0.61**
		30	1.11	0.98	0.93	0.88	**1.03**	**0.99**	**0.95**	**0.94**
		50	**1.03**	**1.00**	0.98	0.97	**1.03**	**1.00**	**1.00**	**0.99**
		70	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Kitchen	3.00	10	**1.53**	**0.82**	**1.00**	**0.73**	3.00	0.33	**1.00**	0.33
		30	**1.20**	**0.93**	**1.00**	**0.90**	2.60	0.47	**1.00**	0.47
		50	**1.13**	**0.96**	**1.00**	**0.93**	2.60	0.47	**1.00**	0.47
		70	**1.13**	**0.96**	**1.00**	**0.93**	2.33	0.56	**1.00**	0.56
		100	**1.13**	**0.96**	**1.00**	**0.93**	1.93	0.69	**1.00**	0.69
Logistics	10.46	10	2.91	0.81	**1.00**	**0.57**	**2.01**	**0.83**	0.63	0.41
		30	1.42	**0.96**	**1.00**	**0.83**	**1.34**	0.94	0.86	0.75
		50	1.22	**0.98**	**1.00**	**0.90**	**1.21**	0.97	0.95	0.87
		70	**1.10**	**0.99**	**1.00**	**0.95**	**1.10**	**0.99**	0.97	0.93
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Miconic	6.00	10	2.54	0.70	**0.86**	0.41	**1.46**	**0.82**	0.69	**0.53**
		30	1.39	0.90	0.90	0.75	**1.15**	**0.97**	**0.98**	**0.91**
		50	1.17	0.94	0.92	0.84	**1.02**	**0.99**	**0.99**	**0.98**
		70	**1.01**	0.99	0.99	0.98	**1.01**	**1.00**	**1.00**	**0.99**
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Rovers	6.00	10	2.60	0.71	**0.92**	**0.49**	**1.82**	**0.75**	0.67	0.44
		30	**1.31**	**0.94**	**0.99**	**0.84**	1.36	0.88	0.82	0.72
		50	**1.06**	**0.99**	**0.99**	**0.96**	1.12	0.94	0.89	0.85
		70	**1.01**	**1.00**	**1.00**	**0.99**	1.05	0.99	**1.00**	0.98
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

Continued on next page...

... Table 2.5 continued.

			Action Graphs				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
		10	2.76	0.67	**0.87**	**0.47**	**2.18**	**0.71**	0.70	0.45
		30	1.52	0.87	**0.86**	0.71	**1.45**	**0.88**	**0.86**	**0.72**
Satellite	6.43	50	**1.24**	**0.93**	0.92	0.82	1.29	**0.93**	**0.94**	**0.84**
		70	1.08	0.97	0.95	0.91	**1.05**	**0.99**	**0.99**	**0.96**
		100	**1.04**	**0.99**	**1.00**	**0.98**	1.07	**0.99**	**1.00**	0.96
		10	**1.33**	**0.77**	0.39	0.31	2.10	0.70	**0.55**	**0.33**
		30	**1.10**	**0.82**	0.44	0.43	1.40	**0.82**	**0.58**	**0.48**
Sokoban	7.14	50	**1.06**	0.85	0.52	0.52	1.35	**0.87**	**0.71**	**0.61**
		70	**1.04**	0.87	0.58	0.58	1.08	**0.95**	**0.86**	**0.83**
		100	1.04	0.84	0.50	0.50	**1.00**	**1.00**	**1.00**	**1.00**
		10	3.02	0.65	**0.83**	**0.43**	**1.43**	**0.77**	0.45	0.34
		30	1.89	0.84	**0.88**	**0.66**	**1.40**	**0.87**	0.79	0.63
Zeno-Travel	6.86	50	1.23	**0.96**	**0.96**	**0.87**	**1.15**	0.92	0.82	0.76
		70	**1.01**	**1.00**	**0.99**	**0.98**	1.10	0.98	0.98	0.95
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
		10	2.68	0.73	**0.80**	**0.45**	**1.71**	**0.77**	0.62	**0.45**
		30	1.44	**0.88**	**0.83**	**0.69**	**1.36**	0.87	0.79	0.68
ALL	8.29	50	**1.19**	**0.93**	**0.87**	**0.81**	1.25	0.91	**0.87**	0.79
		70	**1.08**	**0.96**	0.91	0.88	1.15	0.95	**0.96**	**0.90**
		100	**1.04**	**0.97**	0.93	0.92	1.11	**0.97**	**0.99**	**0.95**

Appendix 2.E Inaccurate Initial State Detailed Experimental Results

As described in Section 2.7.3, experiments where performed in which varying amounts of fluents were set to incorrect values. A breakdown of the results, per domain, for these experiments are provided in Tables 2.6 and 2.7.

Table 2.6: Effect increasing the amount of incorrect fluents in the initial state had on the accuracy of our Action Graph approach and on h_{gc} [19] per domain.

			ANY %				20 %				40 %			
			Action Graphs				h_{gc}				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
		10	9.65	0.52	**0.52**	0.05	**3.24**	**0.81**	0.20	**0.09**	4.73	0.74	0.30	**0.09**
		30	4.08	0.79	**0.40**	**0.11**	**3.38**	**0.81**	0.26	0.10	4.95	0.74	0.37	**0.11**
Blocks-World	20.28	50	**1.99**	**0.89**	0.36	**0.20**	3.35	0.82	0.33	0.13	5.95	0.70	**0.43**	0.11
		70	**1.35**	**0.94**	**0.53**	**0.45**	3.42	0.82	0.46	0.23	5.20	0.74	0.52	0.16
		100	**1.10**	**0.99**	0.90	**0.87**	3.86	0.85	**0.97**	0.52	6.26	0.71	0.77	0.32
		10	1.27	**0.87**	**1.00**	**0.87**	1.07	0.57	0.60	0.57	**1.00**	0.47	0.47	0.47
		30	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.73	0.73	0.73	**1.00**	0.53	0.53	0.53
Campus	2.00	50	**1.00**	**0.93**	**0.93**	**0.93**	1.07	0.77	0.80	0.77	1.07	0.83	0.87	0.83
		70	**1.00**	0.80	0.80	0.80	1.07	**0.90**	**0.93**	**0.90**	1.13	0.47	0.53	0.47
		100	**1.00**	0.80	0.80	0.80	1.07	0.90	0.93	0.90	1.07	**0.97**	**1.00**	**0.97**
		10	5.71	0.40	**0.68**	**0.14**	**2.18**	**0.70**	0.25	0.12	2.71	0.66	0.32	0.10
		30	3.32	0.61	**0.39**	**0.22**	**2.07**	**0.71**	0.25	0.09	2.25	**0.71**	0.32	0.20
Depots	8.86	50	**1.96**	**0.79**	**0.54**	**0.41**	2.18	0.75	0.46	0.24	3.14	0.61	0.29	0.06
		70	**1.29**	**0.89**	**0.68**	**0.59**	1.79	0.79	0.46	0.33	2.71	0.69	0.46	0.25
		100	**1.14**	**0.98**	**1.00**	**0.96**	1.89	0.77	0.39	0.33	3.21	0.63	0.43	0.15

Continued on next page...

... Table 2.6 continued.

			ANY %				20 %				40 %			
			Action Graphs				h_{gc}				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
Driverlog	7.14	10	3.93	0.49	**0.71**	**0.20**	**1.14**	**0.76**	0.21	**0.20**	1.32	0.71	0.18	0.15
		30	2.11	0.73	**0.57**	0.38	1.29	0.78	0.36	0.30	**1.21**	**0.82**	0.50	**0.44**
		50	1.79	**0.79**	**0.64**	**0.53**	**1.36**	0.76	0.39	0.31	**1.36**	**0.79**	0.46	0.33
		70	1.50	0.86	**0.75**	**0.69**	**1.04**	**0.89**	0.64	0.64	1.50	0.79	0.50	0.45
		100	**1.07**	**0.97**	**0.93**	**0.90**	1.11	0.92	0.79	0.76	1.25	0.90	0.75	0.71
DWR	7.29	10	3.00	0.61	**0.57**	0.21	1.36	**0.77**	0.32	**0.29**	**1.29**	**0.77**	0.32	0.26
		30	1.71	0.78	**0.54**	0.36	**1.11**	**0.82**	0.39	**0.38**	1.32	0.72	0.14	0.13
		50	1.25	**0.84**	**0.54**	**0.43**	**1.14**	0.78	0.29	0.24	1.29	0.76	0.25	0.20
		70	**1.14**	**0.87**	**0.57**	**0.50**	**1.14**	0.82	0.43	0.43	1.25	0.81	0.43	0.35
		100	**1.00**	**0.99**	**0.96**	**0.96**	1.07	0.90	0.68	0.66	1.07	0.84	0.46	0.45
Easy-IPC-Grid	8.36	10	**1.51**	**0.76**	0.26	**0.20**	3.69	0.55	**0.48**	0.16	3.59	0.56	0.46	0.15
		30	**1.52**	**0.77**	0.33	**0.27**	2.98	0.65	**0.46**	0.21	2.93	0.61	0.33	0.10
		50	**1.34**	**0.87**	**0.61**	**0.56**	2.57	0.67	0.41	0.23	2.75	0.63	0.33	0.16
		70	**1.10**	**0.89**	**0.57**	**0.57**	2.61	0.71	0.52	0.33	2.67	0.69	0.49	0.29
		100	**1.00**	0.98	0.90	0.90	1.02	0.98	0.95	0.94	**1.00**	**1.00**	**1.00**	**1.00**
Ferry	7.57	10	6.54	0.26	**1.00**	0.19	1.57	0.73	0.29	0.16	**1.29**	**0.76**	0.25	**0.21**
		30	1.93	**0.87**	**1.00**	**0.70**	1.39	**0.87**	0.75	0.65	**1.14**	0.83	0.43	0.40
		50	1.25	**0.97**	**1.00**	0.88	**1.04**	**0.97**	0.93	**0.91**	1.11	0.92	0.75	0.71
		70	1.11	0.98	**1.00**	0.95	**1.00**	**0.99**	0.96	**0.96**	1.04	0.97	0.93	0.91
		100	1.07	**0.99**	**1.00**	**0.96**	**1.00**	**0.99**	0.96	**0.96**	**1.00**	**0.99**	0.96	**0.96**
Intrusion-Detection	16.67	10	**1.00**	**0.88**	0.07	0.07	1.31	0.85	0.02	0.01	1.93	0.83	**0.13**	**0.09**
		30	2.89	0.88	**1.00**	**0.41**	**1.04**	**0.89**	0.18	0.16	1.69	0.84	0.09	0.07
		50	1.51	**0.97**	**1.00**	**0.78**	**1.20**	0.90	0.36	0.31	1.44	0.86	0.13	0.09
		70	**1.04**	**1.00**	**1.00**	**0.98**	1.18	0.92	0.44	0.41	1.80	0.85	0.24	0.11
		100	**1.00**	**1.00**	**1.00**	**1.00**	1.04	0.97	0.76	0.73	1.42	0.86	0.16	0.14
Kitchen	3.00	10	**2.00**	**0.67**	**1.00**	**0.63**	2.13	0.49	0.80	0.41	**2.00**	0.44	0.67	0.30
		30	**1.40**	**0.87**	**1.00**	**0.80**	**1.40**	0.64	0.67	0.52	**1.40**	0.51	0.47	0.34
		50	**1.33**	**0.89**	**1.00**	**0.83**	1.60	0.58	0.67	0.47	1.53	0.51	0.53	0.33
		70	**1.33**	**0.89**	**1.00**	**0.83**	1.60	0.49	0.53	0.37	1.73	0.36	0.40	0.17
		100	**1.13**	**0.96**	**1.00**	**0.93**	1.47	0.80	0.93	0.74	1.73	0.53	0.67	0.38
Logistics	10.40	10	4.41	0.68	**1.00**	**0.31**	**1.72**	**0.79**	0.26	0.13	1.74	0.78	0.20	0.08
		30	2.69	**0.83**	**1.00**	**0.51**	1.67	0.80	0.30	0.19	**1.36**	**0.83**	0.30	0.20
		50	1.80	**0.92**	**1.00**	**0.66**	1.54	0.86	0.52	0.37	**1.48**	0.82	0.33	0.23
		70	1.67	**0.93**	**1.00**	**0.71**	1.41	0.89	0.66	0.47	**1.33**	0.88	0.56	0.42
		100	**1.00**	**1.00**	**1.00**	**1.00**	1.03	0.96	0.79	0.78	1.26	0.93	0.75	0.69
Miconic	6.00	10	3.00	0.61	**0.82**	**0.31**	**1.21**	**0.74**	0.32	0.29	**1.21**	0.68	0.14	0.10
		30	1.61	**0.85**	**0.86**	**0.62**	**1.21**	**0.85**	0.64	0.57	1.39	0.74	0.43	0.30
		50	1.18	**0.97**	**1.00**	**0.92**	**1.04**	0.91	0.75	0.75	1.11	0.82	0.50	0.46
		70	**1.07**	**0.99**	**1.00**	**0.96**	1.11	0.96	0.93	0.88	1.18	0.82	0.54	0.46
		100	**1.00**	**1.00**	**1.00**	**1.00**	1.11	0.98	**1.00**	0.95	1.61	0.90	**1.00**	0.82
Rovers	6.00	10	3.64	0.54	**0.93**	**0.32**	**1.46**	**0.66**	0.21	0.14	2.89	0.54	0.57	0.21
		30	**1.61**	**0.88**	**0.93**	**0.70**	1.75	0.68	0.43	0.31	3.07	0.55	0.68	0.31
		50	**1.11**	**0.96**	**0.93**	**0.88**	1.36	0.85	0.71	0.64	2.89	0.59	0.71	0.37
		70	**1.07**	**0.99**	**1.00**	**0.96**	2.07	0.74	0.75	0.50	2.82	0.59	0.68	0.36
		100	**1.00**	**1.00**	**1.00**	**1.00**	1.82	0.73	0.61	0.51	2.96	0.58	0.71	0.38
Satellite	6.43	10	4.00	0.51	**1.00**	**0.39**	1.57	0.69	0.32	0.23	**1.25**	**0.77**	0.43	**0.39**
		30	3.43	0.60	**1.00**	**0.49**	**1.29**	**0.74**	0.32	0.27	1.36	0.72	0.32	0.27
		50	1.43	**0.90**	**0.89**	**0.71**	**1.07**	0.85	0.57	0.55	1.18	0.83	0.57	0.52
		70	**1.11**	**0.93**	**0.86**	**0.80**	**1.11**	0.91	0.79	0.77	1.32	0.89	0.82	0.70
		100	**1.04**	**0.99**	**1.00**	**0.98**	1.11	0.97	0.96	0.92	1.46	0.91	0.96	0.80
Sokoban	7.14	10	**1.57**	**0.68**	0.18	0.12	2.36	0.63	**0.43**	**0.17**	2.39	0.60	0.32	0.09
		30	**1.14**	**0.77**	**0.32**	**0.30**	1.89	0.66	**0.32**	0.20	1.71	0.67	0.25	0.20
		50	**1.11**	0.81	0.39	0.38	1.21	**0.87**	**0.64**	**0.57**	1.14	0.81	0.39	0.34
		70	**1.04**	0.81	0.39	0.39	1.14	**0.91**	**0.75**	**0.70**	1.29	0.87	0.64	0.55
		100	1.04	0.84	0.50	0.50	**1.00**	**1.00**	**1.00**	**1.00**	1.04	0.96	0.89	0.88
Zeno-Travel	6.86	10	4.14	0.49	**0.79**	**0.32**	1.46	0.72	0.32	0.20	**1.14**	**0.73**	0.18	0.18
		30	2.96	0.66	**0.71**	**0.43**	1.68	0.70	0.32	0.21	**1.46**	**0.77**	0.43	0.31
		50	1.82	0.84	**0.82**	0.61	1.36	**0.88**	0.75	**0.68**	**1.29**	**0.88**	0.71	0.66
		70	**1.14**	**0.96**	**0.93**	0.86	1.18	0.95	**0.93**	**0.89**	**1.14**	0.91	0.75	0.71
		100	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	1.04	0.98	0.96	0.95

Continued on next page...

... Table 2.6 continued.

			ANY %				20 %				40 %			
			Action Graphs				h_{gc}				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
		10	3.69	0.60	**0.70**	**0.29**	**1.83**	**0.70**	0.34	0.21	2.03	0.67	0.33	0.19
		30	2.23	**0.79**	**0.74**	**0.49**	**1.68**	0.76	0.43	0.33	1.88	0.71	0.37	0.26
ALL	8.26	50	**1.46**	**0.89**	**0.78**	**0.65**	1.54	0.81	0.57	0.48	1.91	0.76	0.48	0.36
		70	**1.20**	**0.92**	**0.81**	**0.74**	1.52	0.85	0.68	0.59	1.87	0.75	0.57	0.42
		100	**1.04**	**0.97**	**0.93**	**0.92**	1.37	0.92	0.85	0.78	1.83	0.85	0.77	0.64

Table 2.7: Effect increasing the amount of incorrect fluents in the initial state had on the accuracy of h_{gc} [19], continued from Table 2.6.

			60 %				80 %				100 %			
			h_{gc}				h_{gc}				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
		10	5.71	0.69	**0.29**	0.07	5.00	0.73	0.27	0.07	**3.27**	**0.81**	0.22	**0.08**
		30	5.14	0.72	**0.29**	0.07	4.80	0.74	0.26	**0.08**	**3.85**	**0.78**	0.21	0.07
Blocks-World	20.28	50	6.21	0.68	**0.45**	**0.12**	4.98	0.74	0.36	0.10	**3.76**	**0.79**	0.26	0.10
		70	5.95	0.70	**0.43**	**0.14**	5.79	0.69	0.33	0.12	**4.03**	**0.78**	0.35	**0.14**
		100	6.48	0.69	**0.67**	**0.32**	5.96	0.71	0.58	0.27	**5.63**	**0.72**	0.54	0.26
		10	**1.00**	**0.53**	0.53	**0.53**	**1.00**	0.40	0.40	0.40	2.00	0.50	**1.00**	0.50
		30	1.07	**0.70**	0.73	**0.70**	**1.00**	0.47	0.47	0.47	2.00	0.50	**1.00**	0.50
Campus	2.00	50	1.07	0.50	0.53	0.50	**1.00**	**0.53**	0.53	**0.53**	2.00	0.50	**1.00**	0.50
		70	**1.00**	0.53	0.53	0.53	**1.00**	**0.60**	0.60	**0.60**	2.00	0.50	**1.00**	0.50
		100	1.27	**0.80**	0.93	**0.80**	**1.07**	0.57	0.60	0.57	2.00	0.50	**1.00**	0.50
		10	**2.79**	**0.68**	**0.46**	**0.13**	2.96	0.63	0.32	0.06	3.71	0.55	0.36	0.11
		30	2.68	0.66	**0.32**	**0.11**	2.96	0.63	0.29	0.07	**2.32**	**0.68**	0.18	0.07
Depots	8.86	50	2.39	**0.68**	0.25	**0.11**	**2.32**	0.67	0.18	0.08	3.11	0.61	**0.29**	0.06
		70	**2.00**	**0.73**	**0.29**	**0.11**	2.54	0.66	0.21	0.06	2.39	0.67	0.25	**0.11**
		100	**2.36**	**0.68**	0.18	**0.12**	2.64	**0.68**	**0.36**	**0.12**	2.93	0.63	0.32	0.07
		10	**1.54**	0.70	0.21	0.15	1.68	**0.71**	**0.32**	**0.20**	**1.54**	0.68	0.14	0.08
		30	**1.36**	**0.70**	0.14	0.07	1.54	**0.70**	**0.25**	**0.19**	1.79	0.68	**0.25**	0.15
Driverlog	7.14	50	**1.50**	**0.71**	**0.25**	**0.22**	1.79	0.68	**0.25**	0.16	**1.50**	**0.71**	**0.25**	0.13
		70	**1.46**	**0.83**	**0.61**	**0.51**	1.68	0.73	0.39	0.28	1.93	0.62	0.14	0.07
		100	**1.43**	**0.87**	**0.75**	**0.64**	1.57	0.77	0.46	0.36	1.89	0.74	0.54	0.35
		10	1.14	**0.76**	**0.21**	**0.16**	1.39	0.73	**0.21**	0.15	**1.11**	0.72	0.04	0.04
		30	**1.07**	0.74	0.11	0.07	1.11	**0.76**	**0.18**	0.16	1.21	0.74	**0.18**	**0.18**
DWR	7.29	50	**1.04**	**0.79**	**0.29**	**0.27**	1.18	0.76	0.25	0.25	1.32	0.73	0.18	0.12
		70	1.14	0.77	0.25	0.21	1.14	0.72	0.07	0.05	**1.07**	**0.79**	**0.29**	**0.27**
		100	**1.11**	**0.85**	**0.50**	**0.48**	1.18	0.82	0.46	0.40	1.29	0.78	0.32	0.27
		10	3.46	0.56	0.44	0.14	3.59	**0.57**	**0.49**	**0.16**	**3.34**	**0.57**	0.44	0.14
		30	2.62	0.64	0.33	0.13	2.49	**0.68**	**0.39**	**0.21**	**2.46**	0.64	0.25	0.11
Easy-IPC-Grid	8.36	50	2.66	**0.66**	**0.39**	**0.22**	2.59	0.64	0.31	0.15	**2.41**	**0.66**	0.28	0.13
		70	**2.51**	**0.69**	**0.44**	0.24	2.66	0.67	0.43	**0.26**	2.56	0.67	0.38	0.22
		100	**1.02**	**1.00**	**1.00**	**0.99**	1.10	0.98	0.97	0.92	1.10	0.99	**1.00**	0.95
		10	1.29	**0.76**	0.25	**0.20**	**1.25**	**0.76**	0.25	**0.20**	1.54	0.74	**0.29**	**0.20**
		30	**1.00**	**0.88**	**0.57**	**0.57**	1.11	0.83	0.46	0.43	1.18	0.82	0.43	0.38
Ferry	7.57	50	1.11	0.89	**0.64**	**0.61**	**1.07**	**0.90**	**0.64**	**0.61**	1.21	0.85	0.54	0.45
		70	1.11	**0.95**	**0.89**	**0.86**	**1.07**	0.90	0.68	0.64	1.14	0.86	0.57	0.52
		100	**1.00**	**0.98**	**0.93**	**0.93**	**1.00**	**0.98**	**0.93**	**0.93**	1.07	0.92	0.75	0.73
		10	3.49	**0.76**	0.29	**0.09**	8.33	0.51	0.58	0.08	**16.67**	0.07	**1.00**	0.07
		30	3.36	**0.76**	0.27	**0.07**	8.91	0.48	0.60	**0.07**	**16.67**	0.07	**1.00**	**0.07**
Intrusion-Detection	16.67	50	3.53	**0.75**	0.16	0.05	8.13	0.52	0.53	**0.08**	**16.67**	0.07	**1.00**	0.07
		70	4.31	**0.69**	0.20	0.05	8.27	0.49	0.42	0.06	**16.67**	0.07	**1.00**	**0.07**
		100	2.93	**0.78**	0.18	0.06	7.13	0.54	0.36	0.05	**16.67**	0.07	**1.00**	**0.07**

Continued on next page...

... Table 2.7 continued.

			60 %				80 %				100 %			
			h_{gc}				h_{gc}				h_{gc}			
Domain	$\|\mathcal{G}\|$	O%	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M	$\|\mathcal{C}\|$	Q	R	M
Kitchen	3.00	10	**2.13**	**0.44**	0.73	0.32	2.53	0.36	0.80	0.30	3.00	0.33	**1.00**	**0.33**
		30	2.27	0.40	0.73	0.32	**1.93**	**0.51**	0.73	**0.40**	3.00	0.33	**1.00**	0.33
		50	**1.93**	**0.47**	0.67	0.32	2.47	0.42	0.87	**0.39**	3.00	0.33	**1.00**	0.33
		70	**1.87**	**0.44**	0.60	0.32	2.13	0.36	0.60	0.28	3.00	0.33	**1.00**	**0.33**
		100	**1.87**	**0.44**	0.60	0.32	2.33	0.33	0.67	0.26	3.00	0.33	**1.00**	**0.33**
Logistics	10.40	10	**1.75**	**0.76**	0.15	0.06	1.79	**0.76**	0.18	0.07	2.11	0.75	**0.25**	**0.15**
		30	**1.75**	**0.78**	**0.26**	**0.12**	1.80	0.77	0.21	**0.12**	1.93	0.75	0.16	0.10
		50	1.75	0.77	0.20	0.13	**1.52**	**0.80**	0.25	**0.17**	1.89	0.77	**0.28**	0.14
		70	**1.41**	**0.82**	**0.28**	**0.22**	1.59	0.80	**0.28**	0.19	1.66	0.78	0.18	0.12
		100	**1.11**	**0.93**	**0.70**	**0.69**	1.28	0.90	0.59	0.53	1.43	0.89	0.64	0.54
Miconic	6.00	10	**1.36**	**0.70**	0.29	**0.23**	2.00	0.63	0.39	0.20	6.00	0.17	**1.00**	0.17
		30	**1.21**	**0.81**	0.54	**0.47**	1.68	0.65	0.29	0.20	6.00	0.17	**1.00**	0.17
		50	**1.32**	**0.77**	0.46	**0.39**	2.00	0.65	0.46	0.24	6.00	0.17	**1.00**	0.17
		70	**1.39**	**0.79**	0.57	**0.54**	2.64	0.61	0.64	0.29	6.00	0.17	**1.00**	0.17
		100	**1.96**	**0.84**	**1.00**	**0.72**	3.07	0.65	**1.00**	0.51	6.00	0.17	**1.00**	0.17
Rovers	6.00	10	**5.11**	**0.28**	0.89	**0.18**	5.82	0.18	0.96	0.16	6.00	0.17	**1.00**	0.17
		30	**4.96**	**0.28**	0.82	**0.18**	5.86	0.19	**1.00**	0.17	6.00	0.17	**1.00**	0.17
		50	**5.32**	**0.28**	**1.00**	**0.24**	5.86	0.18	0.96	0.16	6.00	0.17	**1.00**	0.17
		70	**4.82**	**0.33**	0.89	**0.22**	5.82	0.18	0.96	0.16	6.00	0.17	**1.00**	0.17
		100	**4.57**	**0.37**	0.89	**0.26**	5.71	0.19	0.93	0.16	6.00	0.17	**1.00**	0.17
Satellite	6.43	10	**1.57**	0.71	0.39	0.29	1.82	**0.74**	0.61	**0.42**	3.43	0.53	**0.79**	0.31
		30	**1.43**	0.69	0.25	0.20	**1.43**	**0.72**	0.36	**0.34**	3.25	0.56	**0.79**	**0.34**
		50	**1.36**	**0.81**	0.57	**0.51**	1.86	0.68	0.46	0.34	3.64	0.52	**0.86**	0.34
		70	**1.43**	**0.78**	0.54	**0.46**	2.07	0.66	0.50	0.34	3.46	0.51	**0.75**	0.32
		100	**1.96**	**0.83**	0.96	**0.75**	2.07	0.82	**1.00**	0.67	3.29	0.63	**1.00**	0.43
Sokoban	7.14	10	2.25	0.60	0.25	0.12	**1.75**	**0.68**	**0.29**	0.14	2.00	0.64	**0.29**	**0.16**
		30	1.75	0.70	0.39	0.27	**1.54**	**0.76**	**0.43**	**0.31**	1.93	0.65	0.29	0.15
		50	1.11	**0.87**	**0.61**	**0.55**	1.46	0.79	0.46	0.36	**1.07**	0.80	0.36	0.30
		70	1.18	**0.87**	**0.64**	**0.57**	1.21	0.85	0.57	0.54	**1.04**	0.79	0.29	0.26
		100	1.18	**0.95**	**0.89**	0.83	**1.07**	**0.95**	0.86	**0.84**	1.25	0.84	0.54	0.48
Zeno-Travel	6.86	10	**1.18**	**0.75**	**0.25**	**0.21**	1.43	0.72	**0.25**	0.19	1.36	0.72	0.21	0.17
		30	1.46	0.73	**0.32**	**0.24**	**1.11**	**0.75**	0.21	0.21	1.21	**0.75**	0.25	0.21
		50	1.29	0.84	**0.57**	0.49	**1.14**	**0.85**	0.54	**0.54**	1.18	0.81	0.39	0.36
		70	1.14	**0.92**	**0.79**	**0.73**	**1.11**	0.88	0.64	0.63	1.18	0.87	0.64	0.58
		100	1.04	**1.00**	**1.00**	**0.98**	**1.00**	0.96	0.89	0.89	1.21	0.96	0.96	0.87
ALL	8.29	10	**2.38**	**0.65**	0.38	**0.19**	2.82	0.61	0.42	**0.19**	3.81	0.53	**0.53**	0.18
		30	**2.21**	**0.68**	0.41	**0.24**	2.62	0.64	0.41	0.23	3.65	0.55	**0.53**	0.20
		50	**2.24**	**0.70**	0.47	**0.32**	2.62	0.65	0.47	0.28	3.65	0.57	**0.58**	0.22
		70	**2.18**	**0.72**	0.53	**0.38**	2.72	0.65	0.49	0.30	3.61	0.57	**0.59**	0.25
		100	**2.09**	**0.80**	0.75	**0.59**	2.55	0.72	0.71	0.50	3.65	0.62	**0.77**	0.41

References

[1] M. Fagan and P. Cunningham. *Case-Based Plan Recognition in Computer Games*. In International Conference on Case-Based Reasoning Research and Development, pages 161–170, Berlin, Heidelberg, 2003. Springer.

[2] J. Hong. *Goal Recognition Through Goal Graph Analysis*. J. Artif. Intell. Res., 15:1–30, 2001.

[3] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. *The LumièRe Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users*. In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, UAI'98, pages 256–265, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[4] W. S. Lima, E. Souto, T. Rocha, R. W. Pazzi, and F. Pramudianto. *User Activity Recognition for Energy Saving in Smart Home Environment*. In IEEE Symposium on Computers and Communication (ISCC), pages 751–757. IEEE, 2015.

[5] Z. Wang, A. Boularias, K. Mülling, B. Schölkopf, and J. Peters. *Anticipatory Action Selection for Human–Robot Table Tennis*. Artif. Intell., 247:399 – 414, 2017. Special Issue on AI and Robotics.

[6] C. W. Geib and R. P. Goldman. *Plan Recognition in Intrusion Detection Systems*. In Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01, volume 1, pages 46–55. IEEE, 2001.

[7] R. Mirsky, Y. Shalom, A. Majadly, K. Gal, R. Puzis, and A. Felner. *New Goal Recognition Algorithms Using Attack Graphs*. In Cyber Security Cryptography and Machine Learning, pages 260–278, Cham, 2019. Springer International Publishing.

[8] P. Masters and S. Sardina. *Cost-Based Goal Recognition in Navigational Domains*. J. Artif. Intell. Res., 64:197–242, 2019.

[9] R. G. Freedman and S. Zilberstein. *Integration of Planning with Recognition for Responsive Interaction Using Classical Planners*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 4581–4588. AAAI Press, 2017.

[10] S. Lemaignan, M. Warnier, E. A. Sisbot, A. Clodic, and R. Alami. *Artificial Cognition for Social Human–Robot Interaction: An Implementation*. Artif. Intell., 247:45 – 69, 2017. Special Issue on AI and Robotics.

[11] S. J. Levine and B. C. Williams. *Watching and Acting Together: Concurrent Plan Recognition and Adaptation for Human-Robot Teams*. J. Artif. Intell. Res., 63:281–359, 2018.

[12] C. Schmidt, N. Sridharan, and J. Goodson. *The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence*. Artif. Intell., 11(1):45 – 83, 1978. Special Issue on Applications to the Sciences and Medicine.

[13] M. Vilain. *Getting Serious About Parsing Plans: A Grammatical Analysis of Plan Recognition*. In Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI'90, pages 190–197. AAAI Press, 1990.

[14] L. Liao, D. Fox, and H. Kautz. *Extracting Places and Activities from GPS Traces Using Hierarchical Conditional Random Fields*. Int. J. Robot. Res., 26(1):119–134, 2007.

[15] S. Tremblay, D. Fortin-Simard, E. Blackburn-Verreault, S. Gaboury, B. Bouchard, and A. Bouzouane. *Exploiting Environmental Sounds for Activity Recognition in Smart Homes.* In AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments. AAAI-ATSE, 2015.

[16] J. Wu, A. Osuntogun, T. Choudhury, M. Philipose, and J. M. Rehg. *A Scalable Approach to Activity Recognition Based on Object Use*. In IEEE Eleventh International Conference on Computer Vision, ICCV, pages 1–8. IEEE, 2007.

[17] R. Mirsky, R. Stern, K. Gal, and M. Kalech. *Sequential Plan Recognition: An Iterative Approach to Disambiguating Between Hypotheses*. Artif. Intell., 260:51 – 73, 2018.

[18] S. Sohrabi, A. V. Riabov, and O. Udrea. *Plan Recognition As Planning Revisited*. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI'16, pages 3258–3264. AAAI Press, 2016.

[19] R. F. Pereira, N. Oren, and F. Meneguzzi. *Landmark-Based Heuristics for Goal Recognition*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 3622–3628. AAAI Press, 2017.

[20] M. Ramírez and H. Geffner. *Probabilistic Plan Recognition Using Off-the-Shelf Classical Planners*. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI'10, pages 1121–1126. AAAI Press, 2010.

[21] R. F. Pereira, N. Oren, and F. Meneguzzi. *Landmark-Based Approaches for Goal Recognition as Planning*. arXiv preprint arXiv:1904.11739, 2019.

[22] A. Cimatti and M. Roveri. *Conformant Planning Via Symbolic Model Checking*. J. Artif. Intell. Res., 13:305–338, 2000.

[23] H. Palacios and H. Geffner. *Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width*. J. Artif. Intell. Res., 35:623–675, 2009.

[24] P. Jiao, K. Xu, S. Yue, X. Wei, and L. Sun. *A Decentralized Partially Observable Markov Decision Model with Action Duration for Goal Recognition in Real Time Strategy Games*. Discrete Dyn. Nat. Soc., 2017.

[25] M. Ramírez and H. Geffner. *Goal Recognition over POMDPs: Inferring the Intention of a POMDP Agent*. In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, IJCAI'11, pages 2009–2014. AAAI Press, 2011.

[26] S. Yue, K. Yordanova, F. Krüger, T. Kirste, and Y. Zha. *A Decentralized Partially Observable Decision Model for Recognizing the Multiagent Goal in Simulation Systems*. Discrete Dyn. Nat. Soc., 2016.

[27] R. F. Pereira, A. G. Pereira, and F. Meneguzzi. *Landmark-Enhanced Heuristics for Goal Recognition in Incomplete Domain Models*. In Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS'19, pages 329–337. AAAI Press, 2019.

[28] D. McDermott. *The 1998 AI Planning System Competition*. AI Mag., 21(2):35, 2000.

[29] M. Helmert. *The Fast Downward Planning System*. J. Artif. Intell. Res., 26:191–246, 2006.

[30] M. Helmert. *Concise Finite-Domain Representations for PDDL Planning Tasks*. Artif. Intell., 173(5):503 – 535, 2009. Special Issue on Advances in Automated Plan Generation.

[31] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning: Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers, 1st edition, 2013.

[32] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. San Francisco: Elsevier, 2004.

[33] J. E. Rubin. *Set theory for the mathematician*. San Francisco (Calif.) : Holden-Day, 1967.

[34] S. Holtzen, Y. Zhao, T. Gao, J. B. Tenenbaum, and S. Zhu. *Inferring Human Intent From Video by Sampling Hierarchical Plans*. In IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, pages 1489–1496. IEEE, 2016.

[35] H. A. Kautz and J. F. Allen. *Generalized Plan Recognition*. In Proceedings of the Fifth AAAI National Conference on Artificial Intelligence, AAAI'86, pages 32–37. AAAI Press, 1986.

[36] J. Thangarajah, L. Padgham, and M. Winikoff. *Detecting & Exploiting Positive Goal Interaction in Intelligent Agents*. In Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS'03, page 401–408, New York, NY, USA, 2003. Association for Computing Machinery.

[37] P. H. Shaw, B. Farwer, and R. H. Bordini. *Theoretical and Experimental Results on the Goal-Plan Tree Problem*. In Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems, volume 3 of *AAMAS'08*, page 1379–1382, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[38] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[39] M. Ramírez and H. Geffner. *Plan Recognition As Planning*. In Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence, IJCAI'09, pages 1778–1783, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[40] G. Forman. *An Extensive Empirical Study of Feature Selection Metrics for Text Classification*. J. Mach. Learn. Res., 3(Mar):1289–1305, 2003.

[41] M. Hossin and M. Sulaiman. *A Review on Evaluation Metrics for Data Classification Evaluations*. International Journal of Data Mining & Knowledge Management Process, 5(2):1, 2015.

[42] Y. E-Martin, M. D. R-Moreno, and D. E. Smith. *A Fast Goal Recognition Technique Based on Interaction Estimates*. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI'15. AAAI Press, 2015.

[43] E. D. Dolan and J. J. Moré. *Benchmarking Optimization Software with Performance Profiles*. Math. Program., 91(2):201–213, 2002.

[44] J. Rafferty, C. D. Nugent, J. Liu, and L. Chen. *From Activity Recognition to Intention Recognition for Assisted Living Within Smart Homes*. IEEE T. Hum.-Mach. Syst., 47(3):368–379, 2017.

[45] K. Yordanova, S. Lüdtke, S. Whitehouse, F. Krüger, A. Paiement, M. Mirmehdi, I. Craddock, and T. Kirste. *Analysing Cooking Behaviour in Home Settings: Towards Health Monitoring*. Sensors, 19(3), 2019. Special Issue on Context-Awareness in the Internet of Things.

[46] L. Amado, R. F. Pereira, J. Aires, M. Magnaguagno, R. Granada, and F. Meneguzzi. *Goal Recognition in Latent Space*. In International Joint Conference on Neural Networks, IJCNN, pages 1–8. IEEE, 2018.

[47] F. Bisson, H. Larochelle, and F. Kabanza. *Using a Recursive Neural Network to Learn an Agent's Decision Model for Plan Recognition*. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI'15, pages 918–924. AAAI Press, 2015.

[48] G. Singla, D. J. Cook, and M. Schmitter-Edgecombe. *Recognizing Independent and Joint Activities Among Multiple Residents in Smart Environments*. J. Ambient Intell. Humaniz. Comput., 1(1):57–63, 2010.

[49] P. C. Roy, S. Giroux, B. Bouchard, A. Bouzouane, C. Phua, A. Tolstikov, and J. Biswas. *A Possibilistic Approach for Activity Recognition in Smart Homes for Cognitive Assistance to Alzheimer's Patients*. In Activity Recognition in Pervasive Intelligent Environments, volume 4, pages 33–58, Paris, 2011. Atlantis Press.

[50] K. Yordanova, F. Krüger, and T. Kirste. *Context Aware Approach for Activity Recognition Based on Precondition-Effect Rules*. In IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops, pages 602–607. IEEE, 2012.

[51] S. Keren, R. Mirsky, and C. Geib. *Plan Activity and Intent Recognition Tutorial*, 2019. Available from: http://www.planrec.org/Tutorial/Resources_files/pair-tutorial.pdf.

[52] C. W. Geib and R. P. Goldman. *A Probabilistic Plan Recognition Algorithm Based on Plan Tree Grammars*. Artif. Intell., 173(11):1101 – 1132, 2009.

[53] H. A. Kautz. *A Formal Theory of Plan Recognition*. PhD thesis, University of Rochester. Department of Computer Science, 1987.

[54] S. S. Vattam, D. W. Aha, and M. Floyd. *Case-Based Plan Recognition Using Action Sequence Graphs*. In Case-Based Reasoning Research

and Development, pages 495–510, Cham, 2014. Springer International Publishing.

[55] R. E. Fikes and N. J. Nilsson. *Strips: A New Approach to the Application of Theorem Proving to Problem Solving*. Artif. Intell., 2(3):189 – 208, 1971.

[56] J. Chen, Y. Chen, Y. Xu, R. Huang, and Z. Chen. *A Planning Approach to the Recognition of Multiple Goals*. Int. J. Intell. Syst., 28(3):203–216, 2013.

[57] D. H. Hu and Q. Yang. *CIGAR: Concurrent and Interleaving Goal and Activity Recognition*. In Proceedings of the Twenty-Third National Conference on Artificial Intelligence - Volume 3, AAAI'08, pages 1363–1368. AAAI Press, 2008.

[58] M. Asai. *Unsupervised Grounding of Plannable First-Order Logic Representation from Images*. In Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS'19, pages 583–591. AAAI Press, 2019.

3

Goal Recognition Design

"Humans are allergic to change. They love to say, 'We've always done it this way.' I try to fight that. That's why I have a clock on my wall that runs counter-clockwise."
Grace M. Hopper (1987)

Action Graphs for Performing Goal Recognition Design on Human-Inhabited Environments

H. Harman & P. Simoens
Published in Sensors, 19 (12):2741, 2019.

Goal Recognition (GR) is an important component of many context-aware and smart environment services; however, a person's goal often cannot be determined until their plan nears completion. Therefore, by modifying the state of the environment, the work of this chapter aims to reduce the number of observations required to recognise a human's goal. Two types of modifications are performed. i) Actions in the available plans are replaced with more distinctive actions. ii) Certain actions are prevented/removed so that humans are forced to take an alternative, more distinctive, route. In our solution, a symbolic representation of actions and the world state is transformed into an Action Graph. This transformation process is different to the previous chapter as, to perform goal recognition design, plans must be accurately represented. After the Action Graph has been created, it is traversed to discover the non-distinctive plan prefixes. These prefixes are processed to determine which actions should be replaced or removed. For action replacement, we developed an exhaustive approach and an approach that shrinks the plans then reduces the non-distinctive plan prefixes, namely Shrink–Reduce. Exhaustive is guaranteed to find the minimal distinctiveness but is more computationally expensive than Shrink–Reduce. These approaches are compared using a test domain with varying amounts of goals, variables and values, and a realistic Kitchen domain. Our action removal method is shown to increase the distinctiveness of various grid-based navigation problems, with a width/height ranging from 4 to 16 and between 2 and 14 randomly selected goals, by an average of 3.27 actions in an average time of 4.69 seconds, whereas a state-of-the-art approach often breaches a 10 minute time limit.

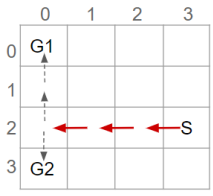
3.1 Introduction

Through the deployment of numerous IoT sensors, smart environments can attempt to recognise the goal of a human from the actions they perform, and thus become more context-aware. Despite recent advances in GR techniques [1, 2], a person's goal often cannot be determined until their plan nears completion. This is because the plans to reach different goals can initially be identical. Nonetheless, recognising a human's goal promptly is important in many situations. In environments where security is essential, such as airports [3], improved goal distinctiveness can allow

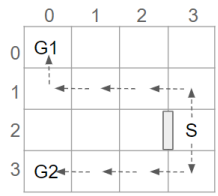
security personnel to intercept a person sooner. In a kitchen environment [4, 5], by minimising the number of observations required to recognise a human's goal, a robot can provide earlier assistance [6].

By redesigning an environment, our work aims to improve the distinctiveness of goals. We have identified two ways Goal Recognition Design (GRD) can affect human-inhabited environments. First, actions in the available plans can be replaced, e.g., by changing an item's location, the act of taking it from its first location is replaced with taking it from a different location. Second, the possibility of performing an action can be removed, e.g., a barrier or ornament can be placed to prevent a human from navigating between two positions. The resulting environment design potentially improves the accuracy of GR approaches, such as [7–9], and requires fewer distinct actions to be detected. In some cases, this reduction could lead to fewer, cheaper or less privacy invasive sensors being deployed in context-aware smart environments.

To clarify the principle of GRD, an example is provided. Figure 3.1 shows two potential goals. These goals could indicate the locations of the gates in an airport [11]. To recognise which goal a human is aiming to reach, their actions are observed; however, depending on which route is taken, initially the human's goal cannot be determined. At worst the plans to reach these goals have a non-distinctive prefix containing 3 actions, i.e., the Worst Case Distinctiveness (WCD) is 3. By placing an obstacle to prevent a person from moving between positions (3,2) and (2,2), the WCD of the environment is reduced to 0. In other words, after the environment



(a) The longest non-distinctive plan prefix from the start location (S), which is part of an optimal plan for the two goals (G1 and G2), is indicated by red arrows. The WCD for this environment is 3.



(b) By preventing the human from performing a single action, the goal of the human can be determined from their first action, i.e., WCD $= 0$.

Figure 3.1: In this grid-based navigation example, a human can move horizontally and vertically. There are multiple optimal plans to each of the goals, but for readability only a single plan to each goal is shown (indicated with arrows). On the left figure, the plans with the longest non-distinctive prefix are displayed. Reproduced from prior work [10].

is redesigned only 1 (discrete) action needs to be observed before the human's goal is discernible. The term goal recognition design and the WCD metric were introduced by Keren et al. [11].

GRD is a more complex problem than task planning. In task planning the aim is (normally) to find an optimal plan to reach a goal, whereas in GRD there are multiple goals defined and all optimal plans containing non-distinctive prefixes must be found before the environment can be redesigned. Current approaches to GRD [12–15] usually focus on removing the ability to perform actions or placing order constraints on the actions. To our knowledge, this chapter is the first to propose state changes that cause actions in the plans to be replaced, which could result in the length of a plan as well as its non-distinctive prefix changing. As the lengths of the plans can change, we propose a new metric, complementary to WCD, to measure the distinctiveness of an environment.

In this chapter, we aim to answer two of the research questions that were presented in the introductory chapter. i) Can a structure similar to those created by library-based approaches be generated from a PDDL defined GR problem? ii) Can a configuration of an environment (e.g., a kitchen) that enables a observed agent's goal to be recognised after fewer observations be automatically discovered?

Our novel approach to GRD transforms a problem defined in Planning Domain Definition Language (PDDL), a popular domain-independent language to model the behaviour of deterministic agents, into an Action Graph. Non-distinctive plan prefixes are extracted from the graph, and processed to determine which actions should be removed or replaced to increase the goals' distinctiveness. Two methods for selecting which actions to replace, namely exhaustive and a less computationally expensive method Shrink–Reduce, are introduced and compared to one another. An overview of our approach is provided in Figure 3.2.

Our approach is not applicable to domains in which cycles exist within the goals' plans, as Action Graphs are acyclic and no action is repeated within its structure. Moreover, the world is assumed to be fully observable; however, after redesigning the environment fewer actions tend to require observing. The algorithms are evaluated on a grid-based navigation domain, and on a Kitchen domain developed by Ramírez and Geffner [7] from the work of Wu et al. [16]. While a domain from [7] is used, a comparison to their approach is not provided as their approach performs GR and not GRD.

Section 3.2 provides an overview of related work. Section 3.3 states the

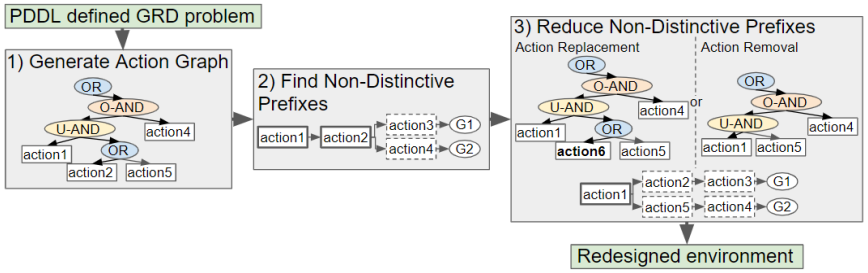


Figure 3.2: Conceptual overview of our novel approach to goal recognition design (GRD). An Action Graph is created from a GRD problem defined in PDDL. This Action Graph is traversed to identify non-distinctive prefixes. In the third step, modifications are performed to reduce the non-distinctive prefixes, leading to a redesigned environment with an increased goal distinctiveness.

formal definition of GRD problems and briefly mentions the GR methods that inspired our approach. The metrics to measure the distinctiveness of an environment are presented in Section 3.4. Section 3.5 describes the structure and construction of an Action Graph. The algorithm to find all the non-distinctive plan prefixes in an Action Graph is introduced in Section 3.6. Section 3.7 describes how the actions to replace are selected. How actions are removed to increase the distinctiveness is described in Section 3.8. Finally, we present experimental results in Section 3.9, by measuring the computational efficiency of our Shrink–Reduce action replacement heuristic and comparing our action removal method to a state-of-the-art approach [11].

3.2 Related Work

The term goal recognition design was coined by Keren et al. [11]. In their approach the WCD of a problem is calculated by transforming a GRD problem into multiple planning problems containing pairs of goals. An optimal plan, with the longest possible non-distinctive prefix, to each pair of goals is searched for. The longest non-distinctive plan prefix, across all joint plans, is the WCD. To reduce WCD an increasing number of actions are removed until either the WCD is 0, or the search space has been exhausted, in which case the best environment design discovered so far is returned. Their pruned-reduce method improves the computational efficiency by reducing the number of action combinations whose removal requires testing. We compare our solution to their pruned-reduce algorithm, and show that for navigation domains we have greatly reduced the time required to solve goal recognition design problems. Further to removing actions, their approach has been extended to action conditioning, in which

a partial ordering is enforced [15]. We do not investigate enforcing action ordering, as our focus is on human-inhabited environments, and it is difficult to force action order constraints on humans; but further to their work, we explore domains in which modifications to the state cause actions within a plan to be replaced and that result in the plan's length changing. Their approach has also been extended for non-optimal agents [17], and to determine where to place sensors within the environment [18]. In this chapter, we assume the agent/human is optimal and do not investigate sensor placement.

Wayllace et al. [12, 14] investigate goal recognition design involving stochastic action outcomes with Markov Decision Processes (MDPs). To calculate WCD, a MDP is created for each goal, the states that are common to pairs of goals are discovered and the Bellman equation is used to calculate the cost of reaching a state. To reduce the WCD their algorithm removes a set of actions, checks that the optimal cost to reach a goal has not been affected, and calculates the WCD to find out if it has been reduced. Their approach creates a MDP multiple times for each of the goals, which results in large computational costs. In our approach a single model (Action Graph) is created, which contains the actions to reach all goals, moreover it is only created once.

In [19], a new metric is introduced, namely expected-case distinctiveness (ECD), which is applicable to stochastic domains and solves a shortcoming of WCD. WCD only incorporates knowledge of the least distinctive pair of goals, rather than all goals' distinctiveness. To solve this, ECD is calculated recursively, starting from the actions applicable to the initial state and ending with the actions furthest from that state (i.e., that result in a goal state). The resulting ECD is the sum of all weighted plan lengths, with the weights based on prior probabilities of an agent choosing a certain goal. We also introduce a new metric which addresses the mentioned shortcoming of WCD, but for a deterministic setting. Moreover, our metric also accounts for the length of both the plan and non-distinctive prefix being altered during the environment design process.

Son et al. [13] propose an approach based on Answer Set Programming (ASP), as an alternative to PDDL, to reduce the computational cost. Their results only show a maximum of two actions being removed, which greatly limits how much WCD can be reduced. Our approach takes PDDL as input as we build on the work from Chapter 2, which performed well (i.e., in terms of computational time and accuracy) on goal recognition problems.

Planning libraries were provided as input to the goal recognition design

method by Mirsky et al. [20]. Further, they introduce plan recognition design, in which the aim is to make the plans, rather than goals, distinctive. They present a brute-force method, which gradually removes a higher number of rules from the planning library, and a constraint-based search method. Their constraint-based search attempts to remove different combinations of rules within the non-distinctive plans starting from the rules contained within the least distinctive plans. Like Mirsky et al. [20] and Keren et al. [15], we developed an exhaustive strategy and less computationally expensive method but, rather than preventing actions, removing rules or constraining the order actions must be performed in, our approach changes the state of the environment in such a way that the actions within plans are replaced by others (with the same effects), which could affect the length of the plans as well as the non-distinctive prefixes.

3.3 Background

To clarify the similarities and differences between symbolic task planning, goal recognition and goal recognition design problems, their formal definitions are provided below. The formal notation will be stated during the description of our GRD method. As our work is closely related to GR, this section also introduces the GR approaches that inspired our solution to GRD.

3.3.1 Formal Definition

Task planners (e.g., Fast Downward [21]) search for a sequence of actions (i.e., task plan) that changes the current world state into a desired goal state. Formally, a planning problem P can be defined as $P = (F, I, A, G)$, where F is a set of fluents, $I \subseteq F$ is the initial state, $G \subseteq F$ is a goal state, and A is a set of actions along with their preconditions $a_{pre} \subseteq F$ and effects $(a_{add} \subseteq F, a_{del} \subseteq F)$ [7, 11, 22, 23]. Actions add and delete fluents from states, and action a is applicable to the state s if $a_{pre} \subseteq s$. In this chapter, effects are denoted a_{eff}, fluents are interpreted as variables with values, and thus actions change the values of variables, e.g., change the value of a variable that represents a human's location, or change variables that indicate which items have been taken from cupboards from false to true.

GR is often viewed as the inverse of planning, i.e., $T = (F, I, A, O, \mathcal{G})$ where $\mathcal{G}$ is the set of all possible goals and O is a sequence of observations [7, 24]. Similarly, a GRD problem can be defined as $\mathcal{D} = (F, I, A, \mathcal{G})$ [11]. The aim of our GRD method is to find the world model (I) that maximises the distinctiveness of the goals $\mathcal{G}$, by reducing the length of the non-distinctive plan prefixes. This results in fewer observations ($|O|$) being required to determine which of the goals ($\mathcal{G}$) a human is intending to reach.

3.3.2 Goal Recognition

Activity recognition [16], plan recognition [25] and goal recognition [7, 8] can all be referred to as intention recognition; nevertheless, their aims slightly differ. Activity recognition labels sensor data with which activity (or action) a human is currently performing, e.g., switching on a kettle. While several previous works have used the terms plan and goal recognition interchangeably, we consider them to be distinct. A plan recogniser attempts to discover the sequence of actions (or possibly hierarchy of actions) a human is performing, including their possible future actions. GR methods aim to label a sequence of discrete observations (e.g., actions), with which (high-level) goal they belong to [26]. For instance, when provided with a sequence of `move` actions GR methods will attempt to select (from a predefined list) which location the agent is intending to reach. For the Kitchen domain by Ramírez and Geffner [7], the sequence of observed actions includes taking different items and performing activities (e.g., making toast), and the returned classification indicates if a person is making breakfast, dinner or a packed lunch.

Methods for GR can be broadly categorised as data-driven and knowledge-driven methods [4, 27]. Data-driven approaches train a recognition model from a large dataset [4, 28–30]. The main disadvantages of this method are that often a large amount of labelled training data is required, and the produced models often only work on data similar to the training set [31, 32]. Knowledge-driven approaches to GR rely on a logical description of the actions agents can perform. They can be further divided into approaches that search through a library of predefined plans (also known as "recognition as parsing" [25, 26, 33]) and approaches that solve a symbolic recognition problem, i.e., "recognition as planning" [2, 7, 26]. In our work, a graph structure is generated from a problem definition in PDDL; this graph structure is similar to those used by some recognition as parsing methods.

Recognition as parsing tends to be fast and allows multiple concurrent plans to be detected [33–37], but is often considered to be less flexible [7, 24] because a planning library containing all actions and their orderings must be developed a priori. A planning library is usually formulated in a hierarchical structure, which includes abstract actions along with how they are decomposed to concrete (observable) actions. The link between parsing text and hierarchical plan recognition structures was suggested by [38], and since then, many plan recognition as parsing algorithms have been developed, e.g., [33, 36, 39].

In [40], a Temporal AND-OR tree was constructed from a library of plans

to determine which objects a human will navigate to. Our method of representing a (human) agent's actions in an Action Graph is inspired by this AND-OR tree; however, the construction of our graph is considerably different. Moreover, their AND-OR tree only contains `ORDERED-AND` nodes, whereas Action Graphs include both `ORDERED` and `UNORDERED-AND` nodes as often actions do not need to be performed in a fixed order.

Recognition as planning is a more recently proposed approach, in which languages normally associated with task planning, such as PDDL, define the actions agents can perform (along with their preconditions and effects) and the world state. This enables a single set of action definitions to be written for task planning, GR and GRD. Moreover, whereas in recognition as parsing usually only actions are considered, planning-based approaches allow for the inclusion of state knowledge, such as what objects are found within the environment and their locations.

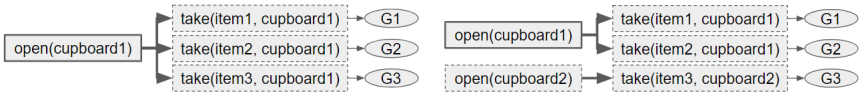
Early works, such as [7], were computationally intensive as a planner was called twice for every goal, to find the difference in the cost of the plan to reach the goal with and without taking the observations into consideration. Later advances in GR as planning algorithms greatly improved the computational efficiency [2, 8, 9]. Plan graphs were proposed in [9] to prevent a planner from being called multiple times. A plan graph, which contains actions and propositions labelled as either true, false or unknown, is built from a planning problem and updated based on the observations. Rather than calling a planner, the graph is traversed to calculate the cost of reaching the goals. More recently, the work presented in [2, 8] significantly reduced the recognition time by finding landmarks, i.e., states that must always be passed for a particular goal to be achieved.

3.4 Distinctiveness Metric

The WCD metric proposed in [11] only provides knowledge about the longest non-distinctive prefix for a set of goals $\mathcal{G}$, rather than considering the overall distinctiveness of the goals and the structures of the plans. In this section, several examples are provided to illustrate the shortcomings of the WCD metric and additional metrics are proposed.

Whilst redesigning the environment, the distinctiveness of some goals ($\mathcal{G}' \subset \mathcal{G}$) can be increased without affecting the WCD. We, therefore, propose finding the longest non-distinctive prefix for each goal and calculating the average length of these, namely the average distinctiveness (ACD). An example is provided to demonstrate the advantage of calculating ACD over WCD. Suppose there are three goals, all requiring a different item to be taken from the same cupboard. The plans are shown in Figure 3.3a, from

which it is clear that the WCD of these goals is 1. After redesigning the environment, by moving `item3` from `cupboard1`, G3 becomes fully distinctive (Figure 3.3b). The environment has arguably been made more distinctive but the WCD does not reflect this, i.e., it is still 1. The ACD of the initial environment is also 1, i.e., $(1 + 1 + 1)/3$, but after `item3` has been moved, the ACD is reduced to 0.67.



(a) WCD = 1, ACD = 1 (b) WCD = 1, ACD = 0.67

Figure 3.3: Example plans for 3 goals. The initial actions common to multiple goals (i.e., the non-distinctive prefixes) are indicated by boxes with solid borders. The remaining actions in the plans to the goals are indicated by boxes with dashed borders.

A second drawback of WCD (and also of ACD) is that these metrics only capture the lengths of the non-distinctive plan prefixes and not the structure of the plans. In other words, WCD lacks knowledge of the dependencies between actions, and thus the possible plan permutations. The more dependants a non-distinctive action has, the less distinctive that action is. If an action has one distinct dependant, then only one change is required to make it fully distinctive. When the state of the environment is redesigned, both the WCD and the ACD metrics could be reduced without the goals becoming necessarily more distinctive, or vice-versa. For this reason, we introduce modified versions of WCD and ACD, i.e., WCD_{dep} and ACD_{dep}.

To calculate these, if a non-distinctive action is required to fulfil multiple actions' precondition(s), that action is counted multiple times. More precisely, each action a in the longest non-distinctive prefix is counted C times, with C equalling the number of actions (including the goal itself) for which a is a dependency. Dependencies are defined as actions that set one or more of the dependant's preconditions, e.g., action 1 is said to be dependent on action 2 if action 2 fulfils one (or more) of action 1's preconditions, i.e., $a1_{pre} \cap a2_{eff} \neq \emptyset$. If multiple options exist, the longest list of dependencies is selected.

The calculation for ACD_{dep} is defined by Equation (3.1), in which $|p(G1, G2)|$ is the number of actions counted, as described above, in the longest prefix common to both $G1$ and $G2$. In the calculation of WCD_{dep}, the averaging operation is replaced by finding the maximum, see Equation (3.2). Note, the operator p is non-commutative: $|p(G1, G2)|$ is not necessarily

equal to $|p(G2, G1)|$. Algorithmic details on how p can be discovered are provided in Section 3.6. For clarification several examples in which WCD $\neq$ WCD_{dep}, and the reduction differs, are provided next.

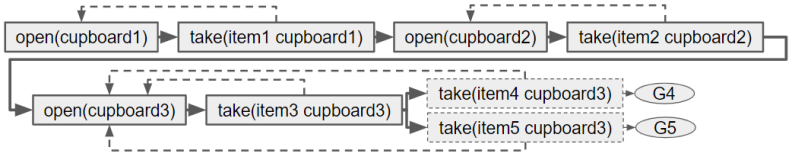
$$\text{ACD}_{\text{dep}}(\mathcal{G}) = \frac{\sum_{G1 \in \mathcal{G}} (\max_{G2 \in (\mathcal{G} \setminus G1)} |p(G1, G2)|)}{|\mathcal{G}|} \tag{3.1}$$

$$\text{WCD}_{\text{dep}}(\mathcal{G}) = \max_{G1 \in \mathcal{G}} (\max_{G2 \in (\mathcal{G} \setminus G1)} |p(G1, G2)|) \tag{3.2}$$

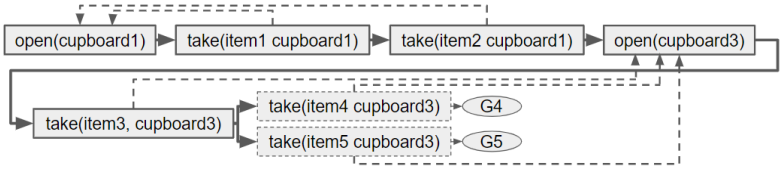
In the example shown in Figure 3.4a, each of the 2 goals requires taking `item1`, `item2` and `item3`. These items are all in different cupboards that must be opened before the item can be taken. Therefore, the WCD for this environment is 6. The WCD_{dep} is 7, as both these goals require a second (unique) item to be taken from `cupboard3`, so opening `cupboard3` is counted twice. If during the design process `item2` is moved into `cupboard1` the WCD is reduced to 5 but the WCD_{dep} is not reduced (Figure 3.4b). When items are moved into a single cupboard the plans, as well as non-distinctive prefixes, become shorter. Therefore, a GRD approach that aims to reduce WCD could simply move all items into the same cupboard. A better solution would be to put the items unique to a goal in a cupboard not required by any another goal since, depending on which plan permutation is selected, this enables the goal to be recognised after a single observation. Note, making plans shorter may be desired by the human; however, it is not the primary aim of our work.

In the same scenario, it is also possible that WCD_{dep} is reduced when WCD is not. Figure 3.4c shows the example after such a change occurs. If the items unique to both plans are moved into different cupboards (which are also not contained within the non-distinctive prefix), WCD_{dep} is reduced but WCD remains the same. This reduction reflects the fact that, due to the different plan permutations, the human's first action could now be distinctive (i.e., if they open `cupboard4` their goal must be G4).

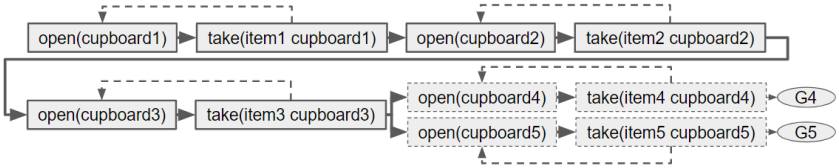
The non-distinctive prefix $p(G4, G5)$ does not necessary equal the non-distinctive prefix $p(G5, G4)$. For example, if `item4` is moved to `cupboard4` and `item5` remains in `cupboard3`, then $|p(G4, G5)| = 6$, whereas $|p(G5, G4)| = 7$ (i.e., the opening of `cupboard3` is counted twice). G4 is more distinctive than G5, as to maximise G5's distinctiveness one item's state needs to be changed, but modifying G4's plan will not increase the distinctiveness.



(a) $\text{WCD}_{\text{dep}} = 7$, $\text{WCD} = 6$



(b) After item2 has been moved to different cupboards. $\text{WCD}_{\text{dep}} = 7$, $\text{WCD} = 5$



(c) After item4 and item5 have been moved to different cupboards. $\text{WCD}_{\text{dep}} = 6$, $\text{WCD} = 6$

Figure 3.4: Examples in which $\text{WCD}_{\text{dep}} \neq \text{WCD}$, and there reductions differ when the state of the environment is changed. A single plan permutation is provided. The dashed arrows go from dependant to dependency.

3.5 Action Graphs for Modelling GRD Problems

Goal recognition design is performed by building an Action Graph, traversing the graph to find the non-distinctive prefixes, then either removing or replacing actions to reduce the length of these non-distinctive prefixes. This section provides details on the structural features of Action Graphs and on how such an Action Graph is derived from a problem specified in PDDL.

3.5.1 Structural Features

Action Graphs model the order constraints, i.e., dependencies, between actions. An Action Graph contains `OR`, `AND` and leaf nodes. Leaf nodes are also referred to as action nodes, as each one is associated with an action. `AND` nodes are split into two categories: `ORDERED-AND`, which denotes that all children are performed in order, and `UNORDERED-AND`, for which all children are performed in any order. For `OR` nodes one child is performed. The root node is always an `OR` node. Throughout this chapter, unless otherwise stated, the term parent(s) always refers to the direct parent(s) of a node, the same goes for child/children.

The term graph is stated, rather than tree, because action and ORDERED-AND nodes can have multiple parent nodes. Action Graphs are acyclic, i.e., do not contain any cycles. A sub-section of an example Action Graph is depicted in Figure 3.5.

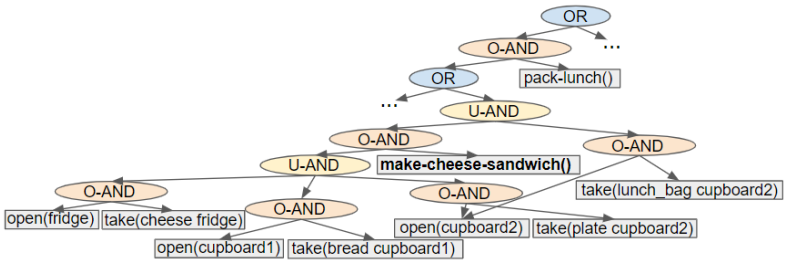


Figure 3.5: Example of an Action Graph. Arrows point towards the child node. O-AND stands for ORDERED-AND and U-AND is UNORDERED-AND. Figure only show a small sub-set of the actions in the complete Kitchen domain Action Graph.

3.5.2 Action Graph Creation

An Action Graph is generated from a GRD problem $\mathcal{D} = (F, I, A, \mathcal{G})$ by performing a Breadth First Search (BFS) backwards from each goal $G \in \mathcal{G}$ to the initial state I. This section first describes the construction of an Action Graph for unconstrained problems, i.e., problems in which all plan variations (including those longer than the optimal) are required to calculate the distinctiveness metrics. Subsequently, the optimisations made to the construction algorithm, that allow problems with constraints (i.e., navigation problems) to be handled, are detailed. Such problems have strict constraints on the order of actions and only the optimal plans are inserted into the Action Graph. The unconstrained and constrained Action Graph creation algorithms are demonstrated in the videos provided as supplementary material.

3.5.2.1 Unconstrained Action Graph Creation

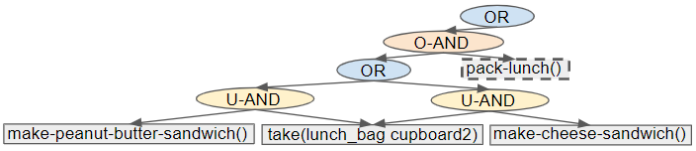
In many domains, e.g., a Kitchen domain, all plans (including sub-optimal plans) should be included, to calculate the distinctiveness. For example, the optimal plan for making breakfast includes making a cup of tea but there also exist longer plans, that for instance include making a cup of tea with milk or making a cup of coffee instead of tea. All these plan variations are required to calculate how distinctive making breakfast is from the alternative goals. These differences should not be inserted into the lists of hypothesis goals ($\mathcal{G}$), e.g., `(and (made_breakfast) (made_tea))`, `(and (made_breakfast) (made_coffee))`, etc., for a number of reasons. First, there

could be numerous different plans (e.g., in a large factory or hospital domain) to reach a goal; thus, it would be time consuming (and unnecessary) to add every plan variation as a separate goal in the list of hypothesis goals $\mathcal{G}$. Second, it may be more important to determine the high-level goal of a human (e.g., are they making breakfast or dinner). GRD should make these more distinctive rather than, for instance, increasing the distinctiveness between making breakfast with tea and making breakfast with coffee. Third, in some cases this would cause it to be impossible to reduce the WCD, e.g., the WCD would always be the length of the plan to make breakfast with tea as all its actions are within the plan to make breakfast including tea with milk.

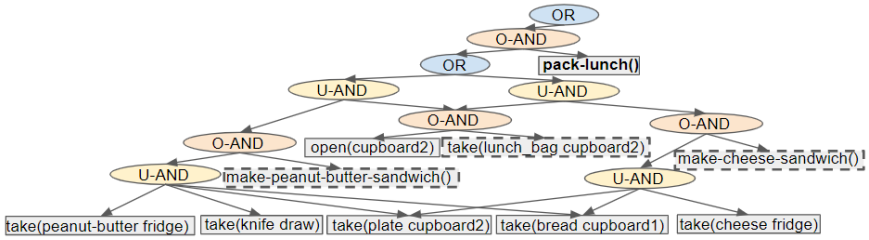
Performing BFS from a goal to the initial state enables an Action Graph to be built from the root downwards. In the resulting graph, (which has a tree-like structure) the actions closest to the root result in a goal atom being met and those at the furthest points from the root are applicable in the initial state (I). Figure 3.6 shows an example of the steps executed to insert the plans for a single goal, i.e., `(lunch_packed)`, into the graph.

An Action Graph is initialised with a `OR` node as the root, and each goal $G \in \mathcal{G}$ is processed in turn. Actions which result in a goal atom being reached ($a \in \mathcal{A}_g$ with $\mathcal{A}_g = \{a \mid a_{eff} \subseteq G\}$) are found and inserted into the graph along with their direct dependencies, as shown in Figure 3.6a. These goal actions are pushed onto a BFS queue to initialise it. For each action (a) in the queue our algorithm iterates over its dependencies, to insert the dependencies of these dependencies (e.g., Figure 3.6b,c) into the Action Graph. During this iteration, the action's (a) dependencies are pushed onto the queue. Actions and dependencies are processed in this manner because it is simpler to remove the action when none of its dependencies' dependencies can be inserted. Dependencies are actions; therefore, in the subsequent text we refer to a dependency's dependencies as an action's dependencies.

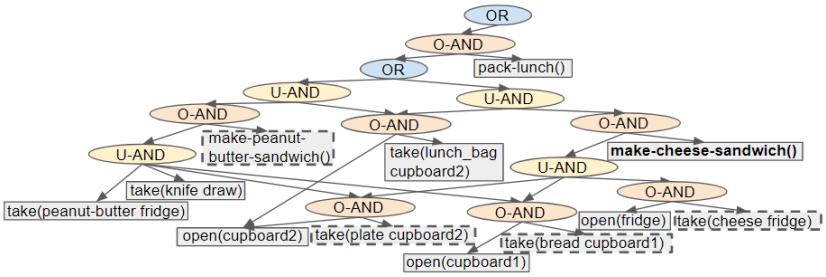
To insert an action's dependencies into an Action Graph multiple nodes are created, including an `ORDERED-AND` node and an action node for each dependency. If the action is a goal action, its action node is created and the `ORDERED-AND` node is appended to the root node's children (e.g., Figure 3.6a). Otherwise, the action itself is already in the graph, and the action's current parents become the `ORDERED-AND` node's parents (e.g., Figure 3.6b). The `ORDERED-AND` node's children are set to the dependencies followed by the action node itself. These dependencies can be a single action, a list of multiple actions or a set of alternatives (as actions can have equivalent effects, or their precondition could contain an `or` statement). If an action has a single



(a) The Action Graph creation starts by inserting the actions that result in the goal being reached (i.e., $\mathcal{A}_g = \{a \mid a_{eff} \subset G\}$), and their dependencies. The single goal action, `pack-lunch()` is inserted into the BFS queue.



(b) `pack-lunch()` is popped from queue and the algorithm iterates over its dependencies, i.e., `make-peanut-butter-sandwich()`, `make-cheese-sandwich()` and `take(lunch_bag cupboard2)`. These actions' dependencies are inserted into the Action Graph, and these three actions are appended to the queue. When `take(lunch_bag cupboard2)` is popped from the queue, as its single dependency is applicable to the initial state, the graph is not expanded and no actions are pushed onto the queue.



(c) The next item in the queue, `make-cheese-sandwich()`, is processed in the same way, i.e., its dependencies are iterated over to insert their dependencies.

Figure 3.6: Example of the steps taken to create an Action Graph, when the first goal is `(lunch_packed)`, which can be achieved by retrieving a lunch bag, and making either a cheese or peanut butter sandwich. Actions with a dashed border are currently in the BFS queue and the bold actions indicate the action which has just been popped from the queue. The algorithm continues until the queue is empty; only the first steps are provided to prevent the graph becoming unreadable.

dependency, a single action node is created and prepended directly to the `ORDERED-AND` node's children; e.g., in Figure 3.6b `take(lunch_bag cupboard2)` has a single dependency. When an action depends on multiple actions, an `UNORDERED-AND` node is prepended to the `ORDERED-AND` node's children. The `UNORDERED-AND` node's children are set to the dependencies; for example,

`make-cheese-sandwich()` in Figure 3.6b. For a set of alternatives, an `OR` node is inserted to indicate the different choices that can be made. In this case, the `OR` node is prepended to the `ORDERED-AND` node's children, and the different options (i.e., actions or `UNORDERED-AND` nodes) become the `OR` node's children (e.g., Figure 3.6a).

When an action that has no dependencies is reached, i.e., is applicable to the initial state ($\mathcal{A}_I = \{a \mid a_{pre} \subseteq I\}$), there is no need to further expand that branch of the graph (i.e., the action is not pushed on to the BFS queue). Moreover, if an action was previously processed, e.g., when adding the plans for the preceding goals, there is no need to re-insert its dependencies nor to insert the action into the queue. Branches for which the initial state cannot be reached are removed. This process finishes when the BFS queue is empty.

After all actions required to reach the first goal have been inserted, the same process is repeated for the subsequent goal. The order in which the goals are processed does not affect the resulting Action Graph. To insert all plan variations, no boundary on how sub-optimal a plan can be is set. For actions (and states) that can expand abundantly, such as in navigation domains, the constrained approach described in the next section (Section 3.5.2.2) should be executed.

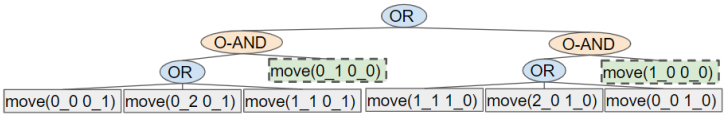
`UNORDERED-AND` nodes are also inserted when a goal containing an `and` statement, which results in more than one action being required to reach that goal, is detected. When this is detected an `ORDERED-AND` node is created and its children are set to a `UNORDERED-AND` followed by a placeholder/dummy action. The `UNORDERED-AND` node's children are set to the required actions (or their parent `ORDERED-AND` nodes if they have dependencies). This placeholder becomes the action required to reach the goal state, i.e., $a_{eff} = G$ where a is the placeholder. These placeholders simplify the graph traversal algorithms required to find and reduce the non-distinctive prefixes; they are ignored when calculating the ACD_{dep} and WCD_{dep} metrics.

3.5.2.2 Action Graph Creation for Navigation Problems

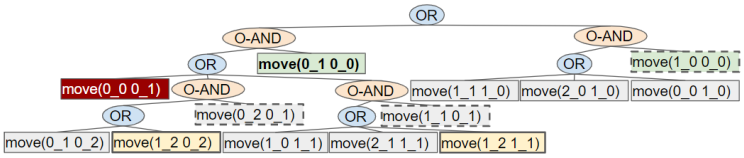
For navigation problems the graph creation process can be optimised by only finding the optimal plans. This section describes how a limit is set during the Action Graph creation. The modified version of the algorithm is provided in Appendix 3.A (Algorithm 3.4). For this domain, only a single permutation of a plan exists. Figure 3.7 shows an example of the steps executed to insert all the optimal plans for a single goal in a 3 by 4 grid-based navigation problem, as depicted in Figure 3.8.

Similar to before, all actions to reach a goal are inserted into the Action

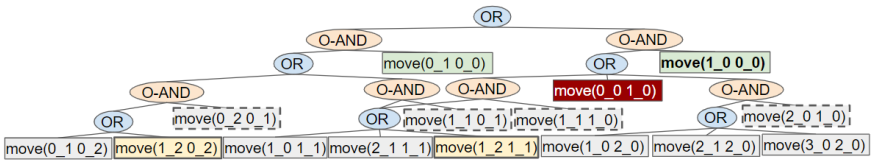
Graph by performing a BFS starting from actions whose effect (a_{eff}) result in the goal state (G), i.e., $\mathcal{A}_g = \{a \mid a_{eff} = G\}$. When an action that has no dependencies, i.e., is applicable to the initial state ($\mathcal{A}_I = \{a \mid a_{pre} \subseteq I\}$), is reached the length of the shortest plan is known; as this is the number of steps taken from the goal to reach that action (Figure 3.7d). Any actions not within an optimal plan, i.e., whose dependencies are further from the



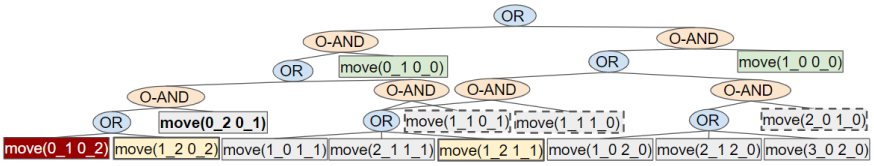
(a) The Action Graph creation starts by inserting the actions that result in the goal being reached (i.e., $\mathcal{A}_g = \{a \mid a_{eff} = G\}$), and their dependencies. The two goal actions, `move(0_1, 0_0)` and `move(1_0, 0_0)` are inserted into the BFS queue. If any of the goal actions are applicable to the initial state, a threshold on the number of allowed steps away from the goal action is set to 1.



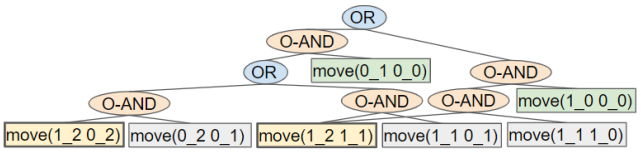
(b) `move(0_1, 0_0)` is popped from queue and the algorithm iterates over its dependencies, i.e., `move(0_0, 0_1)`, `move(0_2, 0_1)` and `move(1_1, 0_1)`. These actions also all have dependencies. All the dependencies of `move(0_0, 0_1)` have already been inserted into the graph, and connecting it to its dependencies would result in its dependencies being further from the goal (than their current placement), so the action (`move(0_0, 0_1)`) is removed. For the remaining actions, i.e., `move(0_2, 0_1)` and `move(1_1, 0_1)`, their dependencies are successfully inserted into the Action Graph, and these two actions are appended to the queue.



(c) The next item in the queue, `move(1_0, 0_0)`, is processed in the same way. `move(1_1, 1_0)`'s dependencies already exist in the Action Graph, therefore it can be linked to these without the need to create any action nodes.



(d) As a dependency of `move(0_2, 0_1)` is applicable to the initial state, the threshold on the number of actions required to reach the goal is set to 2. Adding the dependencies for `move(0_1, 0_2)` would result in the threshold being breached, therefore this action is removed. No further actions are inserted into the queue.



(e) The items in the queue continue to be processed as described in the previous steps until the queue is empty. This figure shows the resulting Action Graph, after a single goal has been inserted.

Figure 3.7: Example of the steps taken to create an Action Graph, when the initial state is `(human-at 1_2)` and the first goal is `(human-at 0_0)`. The layout of the environment is shown in Figure 3.8. Actions which result in the goal state being reached are highlighted in green; yellow boxes with a thick border indicate actions applicable to the initial state; red indicates actions that are being removed; actions with a dashed border are currently in the BFS queue and the bold actions indicate the action which has just been popped from the queue. Reproduced from our prior work [10].

goal than the current limit, are removed from the graph; if there are no actions with equivalent effects, actions that are dependent on the action being removed are also removed.

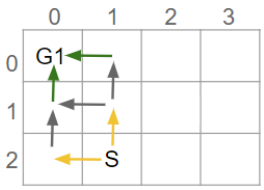


Figure 3.8: A depiction of the grid-based navigation problem, that used to describe the Action Graph creation algorithm. The arrows indicate the actions within the produced Action Graph. *S* is the human's start location and *G*1 is a goal.

Actions are also removed when linking it to its dependencies would cause a cycle (and thus sub-optimal plans) to occur. If all the dependencies of an action have been inserted into the graph while processing the current goal, then connecting it to its dependencies would create a cycle within the graph. Therefore, the action is removed. For example, the `move(0_0 0_1)` action in Figure 3.7b.

When the BFS queue is empty, all optimal plans to reach the first goal are contained within the Action Graph; an example is shown in Figure 3.7e. When inserting the actions that lead to the subsequent goals, the algorithm also checks if an action was inserted when processing a previous goal. If so, there is no need to (re-)insert the action's dependencies (nor insert the

action into the queue). The limit on the number of actions to reach the goal is set to the current action's distance from the goal plus its distance from the initial state, which is discovered by traversing the graph (depth-first).

3.6 Find All Non-Distinctive Plan Prefixes

Once an Action Graph has been created, the non-distinctive plan prefixes are discovered, and the ACD_{dep} and WCD_{dep} are calculated. These prefixes are then iterated over to increase the distinctiveness of the goals. In our approach finding non-distinctive plan prefixes involves two steps: (1) labelling which nodes in the graph belong to which goals and (2) for each of the goals traversing the graph to find the actions that also belong to another goal's plan (as shown in Algorithm 3.1). This section describes these two steps.

3.6.1 Label Which Nodes Belong to Which Goals

Each goal action $\mathcal{A}_g = \{a \mid a_{eff} = G \in \mathcal{G}\}$ that has dependencies, has an `ORDERED-AND` node as its only parent. All children of this `ORDERED-AND` node, including non-direct children, belong to the same goal as the goal action. Therefore, by performing a depth first traversal starting from these `ORDERED-AND` nodes, all nodes are labelled with which goals they belong to. This step is performed to make traversing the graph to extract the non-distinctive prefixes easier and more efficient. These labels will also be utilised by the algorithms for increasing the distinctiveness of goals.

3.6.2 Extract Non-Distinctive Prefixes

To extract the non-distinctive prefixes, goal actions are iterated over (lines 6–9 Algorithm 3.1) and a depth first traversal starting from each of their parent `ORDERED-AND` nodes is performed, to find the actions (and their dependencies) that are also in another goal's ($G2$'s) plan. During this depth first traversal what steps are performed depends on the node's type (e.g., action, `OR`, etc.) and whether that node belongs to the second goal ($G2$).

- If the current node is an action node and it belongs to $G2$ (line 15), then it is simply appended to the list of non-distinctive actions (i.e., the prefix). Actions that do not belong to $G2$ are ignored.
- When the node is of type `OR` (line 17), each of its children are iterated over to find the one containing the most actions belonging to $G2$. Only one of its branches are required within a plan, and so to calculate the metrics only the longest non-distinctive prefix is required. The non-distinctive actions not within the branch containing the most actions belonging to $G2$, are inserted into the list of all non-distinctive prefixes, as these actions should also be processed by

Algorithm 3.1 Extract non-distinctive prefixes (depth-first traversal).

```
> Inputs: the list of goals and the Action Graph
> Output: list containing all non-distinctive plan prefixes
function GET_NON_DISTINCTIVE_PREFIXES(𝒢, graph)
    allPrefixes = []                                          ▷ global variable
    for G1 ∈ 𝒢 do
        A_n = graph.get_action_nodes(a ∈ {a' | a'_eff = G1})
        for G2 ∈ (𝒢 \ G1) do
            for a_n ∈ A_n do
                GET_NON_DISTINCTIVE_PREFIX_RECURSE(a_n.parent, G2, prefix = [ ])
                allPrefixes.append(prefix)
            end for
        end for
    end for
    sort(allPrefixes)                                         ▷ longest first
end function
function GET_NON_DISTINCTIVE_PREFIX_RECURSE(node, G2, prefix)
    if node is action node and node.belongsTo(G2) then
        prefix.append(node)
    else if node is OR node then ▷ Get longest non-distinctive prefix, all others are appended to allPrefixes
        longestPrefix = GET_LONGEST_NON_DISTINCTIVE_PREFIX(node.children, G2)
        prefix.append_all(longestPrefix)
    else if node is ORDERED-AND and node.belongsTo(G2) then
        prefix.append_all(node.get_all_leaves_dfs())
    else
        for child ∈ node.children do
            GET_NON_DISTINCTIVE_PREFIX_RECURSE(child, G2, prefix)
        end for
    end if
end function
function GET_LONGEST_NON_DISTINCTIVE_PREFIX(children, G2)
    longestPrefix = []
    for child ∈ children do
        prefix = []
        GET_NON_DISTINCTIVE_PREFIX_RECURSE(child, G2, prefix)
        if |prefix| > |longestPrefix| then
            allPrefixes.append(longestPrefix)
            longestPrefix = prefix
        else
            allPrefixes.append(prefix)
        end if
    end for
    return longestPrefix
end function
```

algorithms that attempt to minimise the ACD_{dep}/WCD_{dep}.

- If an `ORDERED-AND` node that belongs to $G2$ is encountered (line 20), then all the nodes' children (including non-direct children) also belong to $G2$. Therefore, a depth first traversal is performed to insert all the actions that are descendants of the `ORDERED-AND` node into the list of non-distinctive actions. As the left branch of an `ORDERED-AND` node must be performed before the right branch, actions are inserted in the order they must be performed. During this traversal, `OR` nodes are handled using the method mentioned above.
- In all other cases, the algorithm recurses over all the node's children (lines 23–25).

The longest non-distinctive prefix discovered for each pair of goals (lines 6–9) is passed to Equations 3.1 and 3.2 (from Section 3.4) to calculate ACD_{dep} and WCD_{dep}.

3.7 Performing Action Replacement to Reduce ACD

Modifications applied to the world state I can cause actions in a plan to be replaced by more distinctive actions, and thus reduce ACD_{dep}. For example (as shown in Figure 3.3), if the state of the environment is modified by moving `item3` from `cupboard1` to `cupboard2`, in all plans opening `cupboard1` then taking `item3` is replaced by opening `cupboard2` then taking `item3`. In our approach, part of the Action Graph is replaced by another sub-graph. This section describes how the list of actions and their replacements is formulated, followed by two methods for selecting which actions to replace, i.e., exhaustive and our less computationally expensive method Shrink–Reduce.

3.7.1 Defining Modifications

This section describes how the possible modifications are defined; the process that creates the replacement sub-graphs; and how these modifications are applied to the Action Graph by the exhaustive and Shrink–Reduce algorithms. The modifications applied during the environment design process are generated from additional PDDL action definitions. For instance, the `move-item-state-modification(?item ?container1 ?container2)` action definition, along with its preconditions and effects, is filled in with all combinations of items and containers that are stated in a PDDL defined problem. The modifications (actions) not applicable to the initial state are filtered out. An example definition is provided in Appendix 3.B.

Our action replacement algorithms start by creating a map, that maps action nodes to their replacement graphs. For each modification, a list of

actions affected by the modification is extracted. An action is affected by a modification if its preconditions (a_{pre}) contains a variable of which the value is different in the modification's effects (m_{eff}). Subsequently, a replacement sub-graph is generated by first applying a modification to the initial state I, then creating a (separate) Action Graph for a goal set to an affected action's effects (a_{eff}). The resulting map contains a list of actions along with their replacement sub-graphs and corresponding modifications. An example of this is visualised in Figure 3.9.

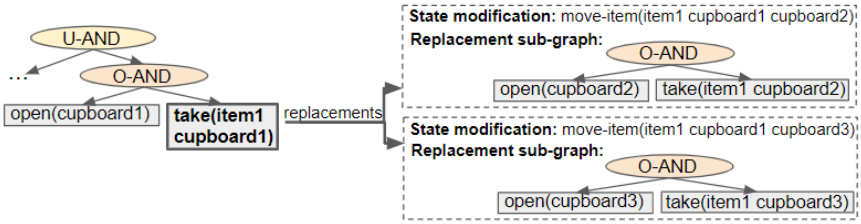


Figure 3.9: Example of an action currently in the Action Graph, i.e., `take(item1 cupboard1)`, which can be affected by 2 state modification actions. Thus, it can be replaced, by swapping its parent `ORDERED-AND` node with one of its replacement sub-graphs.

Changes are made to the graph by swapping the action node(s) (or, if the action has dependencies, its parent `ORDERED-AND` node) with the appropriate sub-graph(s). If a replacement is selected and the corresponding modification affects other actions, they will also be replaced with their equivalent replacements. Actions are not repeated within the Action Graph; thus, nodes' parents and children are altered if an action in the replacement already exists (rather than creating a new action node).

3.7.2 Exhaustive

The exhaustive approach iteratively applies each modification to the Action Graph, then pairs of modifications, then triplets, etc. After a set of action nodes in the Action Graph has been replaced with their corresponding sub-graphs, the ACD_{dep} is re-calculated. The set of modifications resulting in the lowest ACD_{dep} is returned. These modifications can be applied to I to produce the resulting environment design. While this algorithm is guaranteed to find the best possible solution, its computational time can become exceedingly high, especially for problems with large state spaces. Therefore, a maximum size for the set of modifications (N) is provided as a parameter. The pseudo-code for this process is provided in Appendix 3.C.

3.7.3 Shrink–Reduce

Our Shrink–Reduce heuristic is a two step process: (1) Shrink all plans and (2) reduce the length of the non-distinctive prefixes. The pseudo-code can be found in Algorithms 3.2 and 3.3. A video showing the Shrink–Reduce process, for an example problem, has been provided as supplementary material. In this section their details are described in turn, then an example is provided. This method has been developed as it is less computationally expensive than an exhaustive search; however, it is not guaranteed to find the solution with the lowest ACD_{dep}. Moreover, our Shrink–Reduce method was designed for problems containing plans with multiple permutations.

3.7.3.1 Shrink Plans

Shrinking the plans makes it possible to increase the distinctiveness of the environment by performing a single replacement, as afterwards there will exist unused actions that can replace parts of the non-distinctive prefixes. In other words, if there are x variables (e.g., items) that can have y different values (e.g., locations), after changing all variables (that can change value) to 1 value, there are some (e.g., $|y| - 1$) unused values that can be utilised to increase the distinctiveness. Without this step it is more difficult to reduce the non-distinctive prefixes, as a single modification often does not affect

Algorithm 3.2 Shrink plans.

> **Inputs**: list of goals, the Action Graph and map of action node to their possible replacements (e.g., Figure 3.9)
> **Output**: Action Graph containing the shrank plans.

function SHRINK_PLANS($\mathcal{G}, graph, actionsAndReplacements$)
  for $G \in \mathcal{G}$ **do**
    $allPlans = graph.get\_all\_plans(G)$
    $A_r = get\_already\_replaced\_actions(allPlans)$
    for $a \in \{a' \mid a' \in allPlans, a' \in actionsAndReplacements.keys\}$ **do**
      for $r \in actionsAndReplacements[a]$ **do**
        if $|set(A_r, r.actions)| < |set(A_r, get\_preceding\_actions(a))|$ **then**
          $graph.replace\_action(a, r)$
          $A_r.append\_all(r.actions)$
          break
        end if
      end for
      if no $r \in actionsAndReplacements[a]$ has been appended to A_r **then**
        $A_r.append\_all(get\_preceding\_actions(a))$
      end if
    end for
  end for
end function

ACD_{dep}. The process described in this section is performed on each goal $G \in \mathcal{G}$ in turn.

All actions in the plans to reach a goal are extracted from the graph (Algorithm 3.2 line 3) by performing a depth-first search starting from the goal action's ($a_{eff} = G$) parent (`ORDERED-AND` node). Actions inserted when processing a preceding goal are discovered (i.e., A_r from line 4), to prevent increasing the length of a previously shortened plan. Subsequently, all the actions in the plans that can be replaced are iterated over (line 5). If replacing an action results in the plan being shortened (line 7), the replacement is applied to the Action Graph (line 8) and all actions in the replacement's sub-graph are appended to A_r (line 9); otherwise the original action (and its dependencies including indirect dependencies) are appended to A_r (line 14). This check is performed by comparing the number of elements in the

Algorithm 3.3 Reduce non-distinctive prefixes

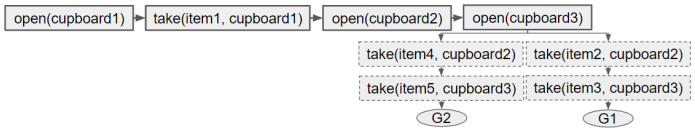
> **Inputs**: list of goals, the Action Graph and map of action node to their possible replacements (e.g., Figure 3.9)
> **Output**: list of modifications (actions) to produce the resulting environment design

1: **function** REDUCE_NON_DISTINCTIVE_PREFIXES($\mathcal{G}, graph, actionsAndReplacements$)
2: $prefixes = GET\_NON\_DISTINCTIVE\_PREFIXES(\mathcal{G}, graph)$
3: $lowestAcd = calculate\_acd()$
4: **for** $p \in prefixes$ **do**
5: **for** $replaceableAction \in getReplaceableActionsThatAffectThePrefix(actionsAndReplacements, p)$ **do**
6: **for** $r \in actionsAndReplacements[replaceableAction]$ **do** ▷ Iterate over an action's replacements
7: $graph.replace\_action(a, r)$
8: $allPrefixes = GET\_NON\_DISTINCTIVE\_PREFIXES(\mathcal{G}, graph)$
9: $p' = allPrefixes.get\_equivalent(p)$ ▷ i.e., whose goals match
10: $acd = calculate\_acd()$
11: **if** $|p'| < |p|$ **and** $acd < lowestAcd$ **then**
12: $lowestAcd = acd$
13: **break**
14: **else**
15: $graph.undo\_replacement(a, r)$
16: **end if**
17: **end for**
18: $prefixes.remove\_equivalent(p)$ ▷ i.e., whose goals match
19: **end for**
20: **end for**
21: **return** $graph.extract\_modifications()$
22: **end function**

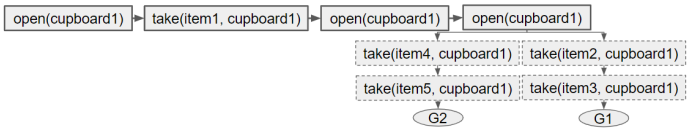
set containing A_r plus the original action and its dependencies (including dependencies' dependencies), with the number of elements in the set containing A_r plus the replacement's actions. During this check, repeated actions are ignored. Performing this comparison with A_r, rather than the full plan, increases the amount the plans are shrunk (e.g., if `item1` has already been moved to `cupboard2`, all items should be moved to `cupboard2`; rather than another cupboard that appears elsewhere in the plan). This process is likely to increase the ACD_{dep}; however, in the next step it will be reduced.

3.7.3.2 Reduce Non-Distinctive Prefixes

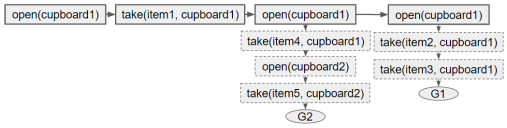
The second step attempts to reduce the length of the longest non-distinctive prefix for each pair of goals (e.g., $p(G1, G2)$ and $p(G2, G1)$). For each prefix, this step iterates over the replaceable actions (line 5 of Algorithm 3.3); a replaceable action appears anywhere in the remainder of the plan (that the prefix belongs to), provided that at least one of its dependencies is within the prefix and the action itself is not in the prefix. An action is replaced by



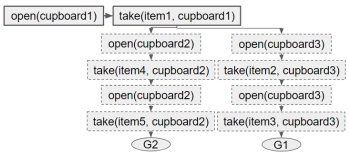
(a) Plan for G1 and G2, prior to shrinking. $|p(G1, G2)| = 4$, $|p(G2, G1)| = 4$, $\text{ACD}_{\text{dep}} = 4$



(b) Plan for G1 and G2, after shrinking. $|p(G1, G2)| = 4$, $|p(G2, G1)| = 4$, $\text{ACD}_{\text{dep}} = 4$



(c) Plan for G1 and G2, after first change made by the reduction step. $|p(G1, G2)| = 4$, $|p(G2, G1)| = 3$, $\text{ACD}_{\text{dep}} = 3.5$



(d) Plan for G1 and G2, after both non-distinctive prefixes have been processed. $|p(G1, G2)| = 2$, $|p(G2, G1)| = 2$, $\text{ACD}_{\text{dep}} = 2$

Figure 3.10: Example of shrinking the plans, then reducing the non-distinctive prefix. Only a single plan permutation is displayed. Rather than showing arrows to indicate when a non-distinctive action is counted multiple times to calculate $|p(G1, G2)|$ and $|p(G2, G1)|$ (as the dependency is required by multiple actions), these actions (dependencies) are repeated.

each of its possible replacements in turn (line 7), until the prefix has been shortened and the ACD_{dep} reduced (lines 11–13), or all its replacements have been processed. If a replacement does not shorten the prefix and reduce ACD_{dep} the alteration is undone (line 15). During this process the currently applied modifications are tracked and, once the non-distinctive prefixes have been processed, they are returned (line 21). This list of modifications (i.e., actions) can be applied to the initial state I to get the resulting environment design.

3.7.3.3 Example of Performing Shrink–Reduce

To clarify why this two step process is performed an example is provided, and depicted in Figure 3.10. The Action Graph of Figure 3.10a is displayed in Figure 3.11. In this example, there are 2 goals (G1 and G2) and to reach these goals different items must be taken from cupboards. The state of the environment can be altered by changing the locations of items. Prior to redesigning the environment, all available cupboards must be opened to reach either goal (Figure 3.10a). Therefore, it is impossible to alter the ACD_{dep} (i.e., the length of the non-distinctive prefixes) with only a single modification. After shrinking the plans, i.e., moving all items to single cupboard (Figure 3.10b), a single modification can be applied to reduce ACD_{dep}; for instance, as shown in Figure 3.10c, `item5` can be moved to `cupboard2`. The resulting plans and non-distinctive prefixes, after all actions in both non-distinctive prefixes (i.e., $p(G1, G2)$ and $p(G2, G1)$) have been reduced, are depicted in Figure 3.10d.

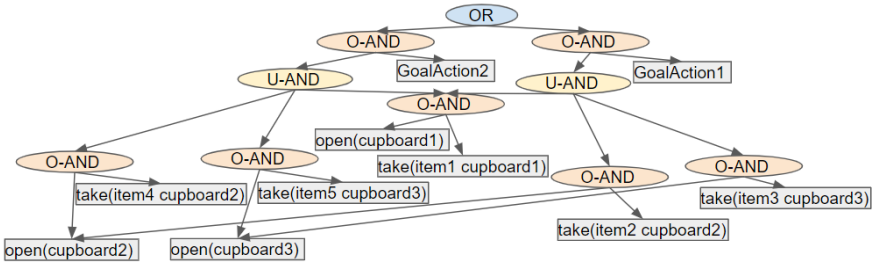


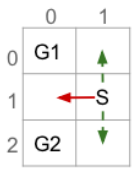
Figure 3.11: Action Graph for the example plans shown in Figure 3.10a. `GoalAction1` results in $G1$ being achieved and `GoalAction2` achieves $G2$.

3.8 Performing Action Removal to Reduce ACD

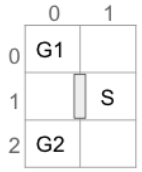
By removing the possibility of performing an action and, consequently, modifying the state of the environment, the goals $\mathcal{G}$ can be made more distinctive (i.e., the ACD_{dep} reduced). This type of state modification is applied to a navigation domain as it is the only human-inhabited environment, we can think of, in which it is feasible to remove actions (e.g., by

placing obstacles). As mentioned above, for such a domain only the optimal plans are contained within the Action Graph, and thus during the action removal process, the cost of the optimal plan is not increased. Rather than simply exhaustively removing parts of the graph, a much less computationally expensive algorithm has been developed. This section first provides an overview of how the non-distinctive prefixes are processed, before providing the details. In Figures 3.12–3.14, taken from our previous paper [10], some simple examples are provided to illustrate how our approach works. An example is also provided in a video supplied as supplementary material. Our approach is applicable to environments of any size with any number of goals.

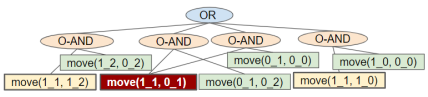
The list of non-distinctive plan prefixes is sorted, most costly first, so that the worst is processed first. In turn each prefix is taken from the list and its actions iterated over to discover if: (1) all goals an action belongs to have a (unique) alternative action; (2) a sub-set of the goals the non-distinctive prefix belongs to have an alternative; or (3) all the non-distinctive prefix's goals have an action with an non-unique alternative. These points are expanded on below. The actions are iterated over in order because if an action at the start of the prefix can be removed, the remainder of the prefix is no-longer valid. If the algorithm were to start from the last action in a plan containing the non-distinctive prefix, it would be more difficult to reduce the distinctiveness (i.e., more actions would require removing).



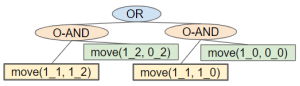
(a) Example of an environment with a single non-distinctive plan prefix containing 1 action, for which each goal has an alternative action.



(b) As each goal has an alternative, the non-distinctive action can be removed, resulting in an ACD of 0.



(c) The action graph for the example environments. The action which is removed is shown highlighted (white text with red background).



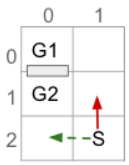
(d) The Action Graph after the ACD has been reduced.

Figure 3.12: Example GRD problem, in which both goals have an alternative to the plan(s) containing the non-distinctive prefix. In all example environments a longest non-distinctive plan is indicated with solid red arrows, and the alternative action(s) by a green dashed arrow.

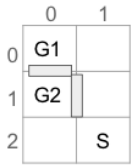
A goal has an alternative action if an action (or if the action has dependencies, its single `ORDERED-AND` parent) has an `OR` node as a direct parent and another of the `OR` node's children belong to that goal. If so, an action can be removed without causing the goal to become unreachable. Moreover, the alternative cannot belong to any of the other goals the (non-distinctive) action belongs to as removing this action would have no effect on the goals' distinctiveness. If all the goals with a plan containing the non-distinctive action have an alternative, the action is removed. An example is provided in Figure 3.12.

After checking all actions in a non-distinctive prefix, if only a subset of the goals have alternative actions, the action(s) directly after the non-distinctive prefix in their plans is removed. As a result, those goals can only be reached by their alternative (possibly more distinctive) plan(s). An example is shown in Figure 3.13. Otherwise, when all the non-distinctive prefix's goals have an action (in the non-distinctive prefix) with an non-unique alternative, i.e., the alternative(s) belongs to more than one of the action's

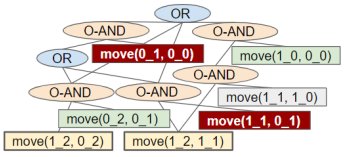
(a) Both goals have an alternative to the 1st action in the non-distinctive prefix but the alternative, `move(1_2 0_2)`, also belongs to both goals; so is only taken into consideration as a last resort. G1 has an alternative to the 2nd action, whereas G2 does not. So, the next action in G1's plan, containing the prefix, is removed.



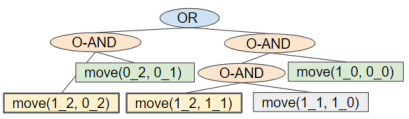
(b) After removing an action, `move (1_2 1_1)` is still a non-distinctive action; thus, is evaluated on the next iteration. This time G2 is the only goal with an alternative action.



(c) The environment after all non-distinctive plan prefixes have been evaluated.



(d) Initial Action Graph.



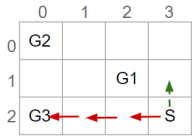
(e) The Action Graph after the action removal process has completed.

Figure 3.13: Example of ACD reduction, when only a subset of the goals have an alternative action, to an action in the worst non-distinctive plan prefix.

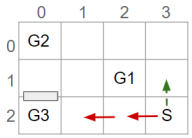
goals, the next (distinctive) action(s) for one of the goals is removed (see Figure 3.14c–d). This prevents the non-distinctive prefix being valid for that goal, thus making it more distinctive. Our action removal method always checks that the action removed will not interfere with any of the other goals, which do not have an alternative, to prevent them from becoming unreachable.

Once an action has been removed, which nodes belong to which goal is re-evaluated (see Section 3.6); and the actions in the non-distinctive prefix, prior to any removed action, are checked to see if they should be inserted into the list of non-distinctive plan prefixes (e.g., Figure 3.13a–b).

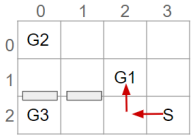
Actions, that are not the last action in the prefix, should be re-processed as it may be possible to further reduce the length of the non-distinctive prefix (e.g., Figure 3.14a–c). The last action in the prefix will not be processed again, if it is still a non-distinctive action then the ACD will not be reduced to 0. An example of an environment in which the ACD cannot be reduced to 0, and the steps our algorithm performs, is shown in Figure 3.14.



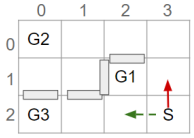
(a) Initial environment. Only G2 has an alternative action, for any action in the non-distinctive plan prefix, so the next action in G2's plan which contains the non-distinctive prefix is removed.



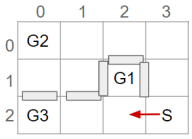
(b) After the 1st action has been removed the second to last action in the original non-distinctive plan prefix is still a non-distinctive action; thus, is evaluated in the next iteration.



(c) After the 2nd action has been removed the longest non-distinctive prefix belongs to G1 and G2. Neither G1 nor G2 have a unique alternative action. So, the next actions(s) in 1 of the goal's plans are removed. In this case, as G1 has no further actions, G2's next actions are removed.



(d) Two actions have been removed, since there were two possible next actions to reach G2 (discovered by detecting an `OR` node in the Action Graph).



(e) G2 is now distinctive, but there still exists a non-distinctive plan prefix for G1 and G3.

Figure 3.14: Example of a 3 goal environment, in which WCD (and thus ACD) cannot be reduced to 0.

3.9 Experiments

Through experiments we aim to demonstrate the scalability and performance of our goal recognition design approach. First, Shrink–Reduce is compared to the exhaustive search on problems generated from a scalability testing domain we developed. Second, the results of running Shrink–Reduce and exhaustive on a Kitchen domain are presented. Finally, our action removal method is compared to the pruned-reduce method by Keren et al. [11], on grid-based navigation problems of various sizes.

3.9.1 Action Replacement Scalability Experiments

The purpose of experiments in this section is to demonstrate the performance of our Shrink–Reduce method, and compare this to the exhaustive approach. The results show the computation time, ACD_{dep} reduction, WCD_{dep} reduction and number of states modifications required to get from the provided to the designed world model. As the plans' lengths can be altered during this process, the average change in length is also mentioned. The experiment setup is described, before presenting and discussing the results.

3.9.1.1 Setup

For these experiments, a test domain was developed that contains definitions for the following actions: `open(?container)`, `take(?item ?container)` and multiple goal actions, e.g., `goal1()`. Each goal action was generated by randomly selecting between 3 to 10 items that require taking before the goal action can be performed, i.e., its preconditions are set to a list of `(taken ?item)` fluents, and its effects were set to a single fluent e.g., `(goal1_reached)`. These actions definitions are representative of definitions suitable for many application domains, e.g., a kitchen, factory or hospital. The location of each item was also selected randomly. The location of these items can be modified during the environment design process. For example, if `open(cupboard1)` and `take(item1 cupboard1)` are in a plan, they can be replaced by `open(cupboard2)` and `take(item1 cupboard2)`.

Three datasets were created from this test domain, (1) with a differing number of variables (i.e., items) whose value can modified, (2) an increasing number of values the variables can be set to (i.e., cupboards the items can be in), and (3) a varying number of goals. When one of these amounts is changed the others are fixed, i.e., there are 15 variables, 5 values, and 5 goals. As the datasets contain an element of randomness, for each variation 5 problems were created and the average result is presented. Experiments were ran on a server with 11 GB of RAM and a Intel Xeon CPU 2.27 Ghz processor. All graph's error bars show the minimum and maximum result.

The exhaustive approach was ran for varying values for the maximum number of changes that can be applied (N), i.e., 1, 2, 3 and 4. In the results these are named *exhaustive1*, *exhaustive2*, *exhaustive3* and *exhaustive4*, respectively. At $N = 5$, the exhaustive search can take 2 days to complete a single (relatively large) problem. Therefore, for the scalability experiments N was set to a maximum of 4. For the experiment presented in the subsequent section (Section 3.9.2), as a domain with fewer goals and possible modifications was used, no limit was set for N.

The ACD_{dep} and WCD_{dep} reductions are presented in performance profile graphs, as suggested by Dolan and Moré [41]. This enables the results to be presented in a more readable format, and all datasets can be grouped into a single result to prevent a small number of problems dominating the discussion. For run-times, graphs showing the individual data-points are provided so that the exponential increase in run-time when the problem size is increased can be clearly seen. To produce the performance profile of an approach ($S \in \mathcal{S}$) the ratio between its result ($D_{P,S}$) and the best solution for a problem ($P \in \mathcal{P}$) is calculated, as shown in Equation (3.3). Equation (3.4) calculates the percentage of problems an approach solved when the ratio is less than a given threshold, τ. When the $\text{ACD}_{\text{dep}}/\text{WCD}_{\text{dep}}$ reduction is 0 ($D_{P,S} = 0$), the ratio is set to infinity (as the approach failed to provide a solution). As approaches can produce the same reduction, the sum of all approaches' performance at $\tau = 1$ does not necessarily equal 0.

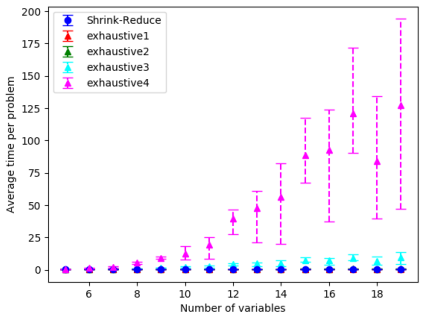
$$R_{P,S} = \begin{cases} \frac{\max(D_{P,S}:S\in\mathcal{S})}{D_{P,S}} & D_{P,S} \neq 0 \\ \infty & \textit{otherwise} \end{cases} \tag{3.3}$$

$$P_S(\tau) = \frac{1}{|\mathcal{P}|}|P \in \mathcal{P} : R_{P,S} \leq \tau| \tag{3.4}$$

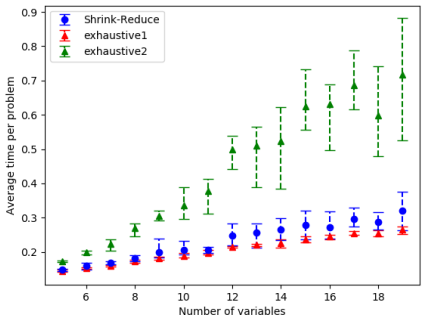
3.9.1.2 Results and Discussion

The run-time and number of required state changes, for an increasing number of variables, values and goals, are shown in Figures 3.15–3.17, respectively. The ACD_{dep} and WCD_{dep} reduction comparisons are shown in Figure 3.18 (separate results for each dataset are provided in Appendix 3.E.1). A similar trend was observed in each of these configurations. The computational times for the different approaches, followed by the increase in goal distinctiveness, are discussed.

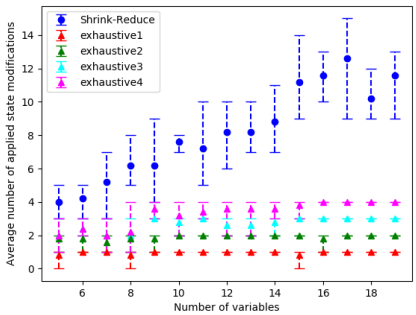
As the size of the problem (i.e., number goals, values or variables) was increased, there was an exponential increase in the average time taken by each of the approaches. The outliers of this trend (e.g., in Figure 3.15a at 18



(a) Average time, in seconds, to complete a GRD problem.



(b) Average time to complete a GRD problem, for the three least time consuming approaches.

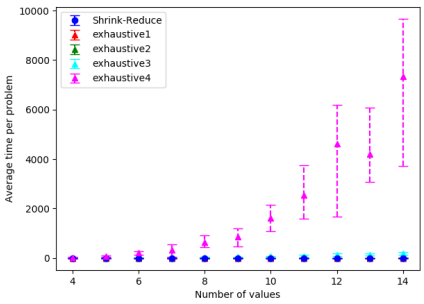


(c) Average number of state changes in the solution.

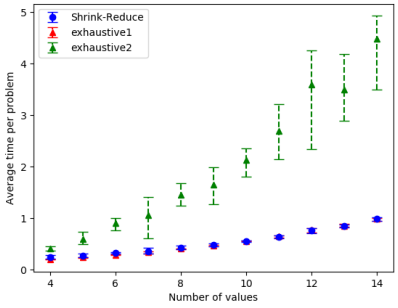
Figure 3.15: Results for an increasing number of variables.

variables) are due to the nature of generating data with randomness, e.g., if a variable is not within a plan to any of the goals, the actions associated with that variable are not inserted into the graph, and thus not processed when increasing the distinctiveness of goals. Our Shrink–Reduce method finished in less time than exhaustive, except for exhaustive1 (i.e., $N = 1$). Having said that, exhaustive1 was inferior at increasing the distinctiveness of the goals.

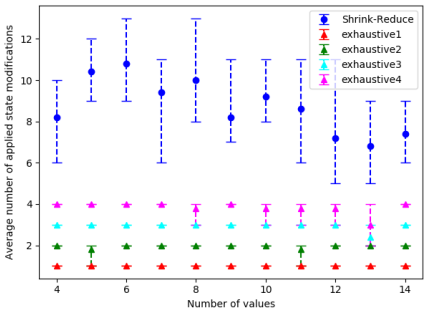
For exhaustive, higher values of N allow ACD_{dep} (Figure 3.18a) and WCD_{dep} (Figure 3.18b) to be reduced to lower values. If exhaustive performs all possible combinations of state changes, it is guaranteed to find the best environment design. Nevertheless, our Shrink–Reduce approach often made the goals more distinctive than exhaustive4, but it also performed a large number of state changes (Figures 3.15c, 3.16c and 3.17c) and often (slightly) increased the average plan length (Appendix 3.E.1).



(a) Average time, in seconds, to complete a GRD problem.



(b) Average time to complete a GRD problem, for the three least time consuming approaches.

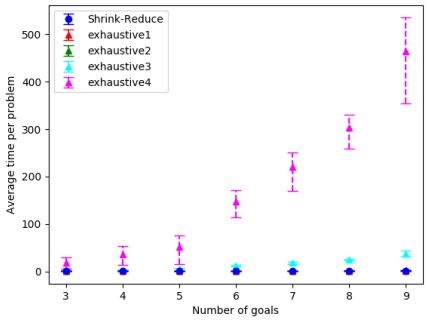


(c) Average number state changes in solution.

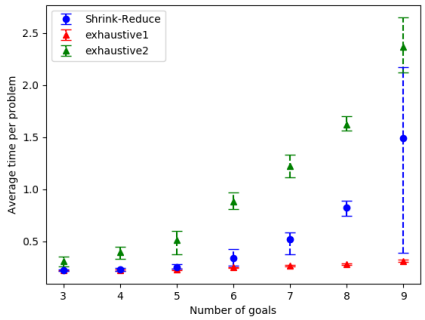
Figure 3.16: Results for an increasing number of values.

As stated earlier, and reiterated by these results, ACD_{dep} is sometimes reduced without any reduction in WCD_{dep}; moreover, it is possible that WCD_{dep} is increased when ACD_{dep} is decreased (e.g., exhaustive1 Figure 3.24b). At $\tau \geq 2$, the percentage of problems solved by exhaustive4 is higher than Shrink–Reduce. This shows that on the problems Shrink–Reduce did not have the highest ACD_{dep} reduction it is further from the best approach, than exhaustive4 is when it did not produce the best result.

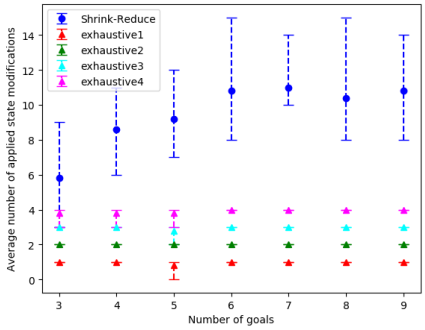
With some alternative GRD methods, such as [11], it would be more difficult to calculate ACD_{dep} and WCD_{dep}. In [11], a task planner is called to find a plan to reach a pair of goals, this joint plan contains the longest non-distinctive prefix. Our distinctiveness metrics could be calculated by searching joint plans for actions that have dependencies (i.e., should be counted multiple times); however, this would be computationally expensive and, as multiple prefixes could be of equal size, multiple joint plans (for a single pair of goals) may require processing. Methods that create graph or tree structures, such as MDPs [12] or planning libraries [20] could



(a) Average time, in seconds, to complete a GRD problem.



(b) Average time to complete a GRD problem, for the three least time consuming approaches.



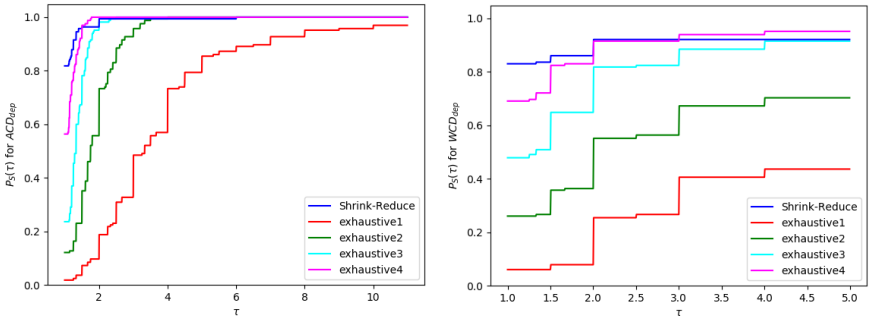
(c) Average number state changes in the solution.

Figure 3.17: Results for an increasing number of goals.

employ a similar method to Action Graphs. Wayllace et al. [12] perform stochastic GRD with MDPs, that each contain a single terminal (goal) state; these MDPs can be traversed to find the non-distinctive prefixes, include those actions that should be counted multiple times. In future work, we intend to investigate stochastic GRD in more detail.

3.9.2 Action Replacement Applied to a Kitchen Domain

The Kitchen domain contains the actions and world model required for a human to make several meals, i.e., breakfast, a packed lunch and dinner. Ramírez and Geffner [7] created the original version of the Kitchen domain for their work on goal recognition, based on the work by Wu et al. [16]. Moreover, similar problems have been utilised by several related works, e.g., the cooking problem by Yordanova et al. [4]. This experiment is included to demonstrate our approach applied to a realistic domain containing more action definitions (than our scalability testing domain). Rather



(a) Performed based on ACD_{dep} reduction. (b) Performed based on WCD_{dep} reduction.

Figure 3.18: Results for all action replacement experiments. When τ is further increased, there is no change in $P_s(\tau)$; unless $\tau = \infty$, then $P_S(\tau) = 1$ for all solutions. The line corresponding to Shrink–Reduce starts the highest, the next line down is exhaustive1.

than just a goal action preceded by a list of take item actions, a more complex graph (plan) structure was produced. Further, fewer goals, variables and values were contained within the kitchen goal recognition design problem, therefore running exhaustive to find the best possible solution was feasible.

3.9.2.1 Setup

We extended the original Kitchen domain by Ramírez and Geffner [7] with action definitions and the state knowledge required to take items from cupboards, a fridge and a drawer; the newly defined actions are `open(?container)`, `close(?container)` and `take(?item ?container)`. The initial state of the environment is set so that all food items (that don't need to be stored in the fridge) are in cupboard1, equipment (e.g., cup and plate) is in cupboard2 and all utensils (e.g., spoon and knife) are in the drawer. In this dataset there are 3 hypothesis goals: $\mathcal{G}$ ={`(made_breakfast)`, `(lunch_packed)` and `(made_dinner)`}. These goals require multiple items to be taken and other actions to be performed, e.g., part of the plan to make breakfast involves taking bread and making toast. Each goal can be achievable through multiple plans, e.g., for `(lunch_packed)` to be reached a person must always perform the `take(lunch_bag)` action but has the option of either performing the `make-peanut-butter-sandwich()` or `make-cheese-sandwich()` action. Information on the plans producible from this domain are provided in Appendix 3.D.

Like the previous experiment, the location of items is modified during the environment design process. As it is likely that a human would not want

items within the fridge or drawer to be changed (e.g., milk taken out the fridge, or cereal put in the drawer), PDDL `not` statements have been inserted into the state modification action to prevent this. The PDDL definition for this action is provided in Appendix 3.B.

3.9.2.2 Results and Discussion

For the Kitchen domain, the initial $WCD_{dep} = 14$ and $ACD_{dep} = 11.00$. Both our Shrink–Reduce and exhaustive approaches successfully increase the distinctiveness of the goals. Exhaustive (although slower) made the goals more distinctive than Shrink–Reduce. Shrink–Reduce reduced WCD_{dep} to 12 and ACD_{dep} to 10.00, whereas exhaustive reduced WCD_{dep} to 11 and ACD_{dep} to 9.33. This illustrates that, for environments with fewer state changes, the exhaustive algorithm should be executed.

Both approaches reduced the WCD (distinctiveness metric by Keren et al. [11]) by 1 action. WCD_{dep} was lowered more than WCD as the number of repeated `open` actions (that are dependencies for `take` actions) was also reduced (as explained in Section 3.4). This difference indicates that depending on which plan permutation is selected by a human, their goal can now be determined sooner. As demonstrated by these results, for problems with multiple plan permutations, WCD_{dep} can provide more insight into the distinctiveness of an environment than just providing WCD.

To produce the resulting environment designs of either approach, 3 state modifications need to be applied to the initial state (I). The list of required modifications are provided in Table 3.1. Both approaches proposed moving `bread` from `cupboard1` to `cupboard2`, this results in `cupboard1` only needing to be opened when making breakfast, and not when making dinner or a packed lunch. Therefore, if a human opens `cupboard1`, their goal is known. Moving the `water-jug` and/or `cup` to `cupboard1` decreases ACD_{dep} as these items are only required to make breakfast. Shrink–Reduce moved the `bowl` instead of the `cup` into `cupboard1`. This increased the distinctiveness between making dinner (which requires the `bowl`) and making a packed lunch (which does not require the `bowl`). After moving the `bowl`, changing the `cup`'s location would not have altered the distinctiveness (as `cupboard1` requires opening for both making breakfast and dinner).

During some initial tests we enable items to be moved into/from the fridge and drawer. This resulted in a lower ACD_{dep} (than provided in Table 3.1) for the redesigned environment produced by both methods. Nonetheless, some changes, e.g., moving the bread into a drawer, are undesired by humans. In many scenarios, there will be a trade-off between making undesired changes and increasing the goals' distinctiveness.

Table 3.1: List of state change actions returned by the Shrink–Reduce and Exhaustive methods, that when applied to the initial state (I) result in the designed environment. The initial $\text{WCD}_{\text{dep}} = 14$, $\text{ACD}_{\text{dep}} = 11.00$, $\text{WCD} = 7$.

Shrink–Reduce: $WCD_{dep} = 12$, $ACD_{dep} = 10.00$, $WCD = 6$	**Exhaustive:** $WCD_{dep} = 11$, $ACD_{dep} = 9.33$, $WCD = 6$
move-item(bread cupboard1 cupboard2)	move-item(bread cupboard1 cupboard2)
move-item(water-jug cupboard2 cupboard1)	move-item(water-jug cupboard2 cupboard1)
move-item(bowl cupboard2 cupboard1)	move-item(cup cupboard2 cupboard1)

3.9.3 Action Removal Experiments

The action removal experiments, presented in this section, aim to show: (1) How well our approach scales as the number of goals and grid size of a navigation domain are increased, (2) how much the ACD and WCD is reduced, and (3) how many actions are removed. For this domain $\text{ACD} = \text{ACD}_{\text{dep}}$ and $\text{WCD} = \text{WCD}_{\text{dep}}$, as actions are strictly ordered. Our Action Graph approach is compared to the pruned-reduce method by Keren et al. [11], by running both approaches on a dataset we generated, containing problems with randomly selected initial and goal locations. This section first describes the experiment setup, then discusses the results.

3.9.3.1 Setup

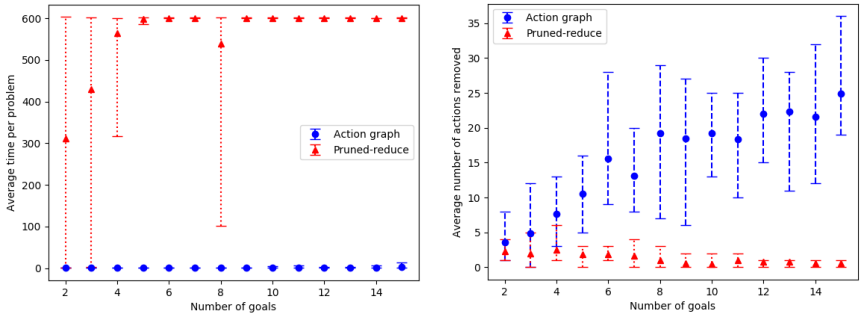
Two grid-based navigation datasets containing GRD problems were created. The first contains problems with an 8 by 8 grid and a varying number of goals (destinations). For each number of goals, 8 problems were generated by randomly selecting a start location and the goals' locations. In total the dataset contains 112 problems.

The second dataset consists of problems with differing grid sizes, for all problems both the width and the height are equal. For each grid size 8 problems, with a random start location and three random goal locations, were generated. In total this dataset contains 56 GRD problems.

Experiments were ran on our department's server with 3 GB of RAM and a Dual Core AMD Opteron 2 Ghz processor. A timeout of 10 minutes per GRD problem was set for all experiments. The whole process, including converting the PDDL into an Action Graph is included in the run-times for our approach. The output of pruned-reduce does not state the ACD (as this is our new metric), so to calculate this the resulting environment design is passed to our algorithm's ACD calculation. As before, all graph's error bars show the minimum and maximum result.

3.9.3.2 Results and Discussion

Experiments were ran on both datasets: with varying amounts of goals and with an increasing grid size. The corresponding results, showing the average time and number of actions removed, are provided in Figures 3.19 and 3.20; and the WCD reduction and ACD reduction comparisons are shown in Figure 3.21 (separate results for the datasets are provided in Appendix 3.E.2).



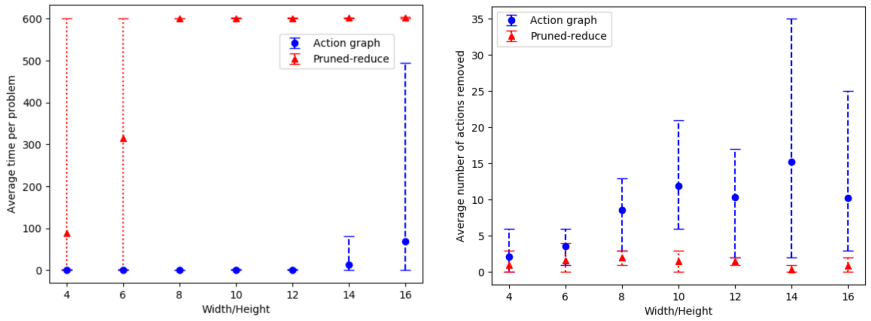
(a) Average time, in seconds, per GRD problem.

(b) Average number of actions that were removed.

Figure 3.19: Results for an increasing number of goals. The results of our Action Graph approach are indicated by blue circles, pruned reduce [11] is shown with red triangles. All times are given in seconds.

For the majority of problems, within the varying number of goals dataset, our Action Graph approach took less than 1 second to perform GRD. In contrast, pruned-reduce hit the timeout for the majority of problems (Figure 3.19a). The execution time (i.e., minimum and maximum) varies between different problems as for some problems removing a small number of actions reduces WCD to 0; whereas, for others WCD can only be partially reduced. Furthermore, the optimal plans within the different problems differ in length (longer plans take more time to find and iterate over). The same trends are observed in the results for an increasing grid size (Figure 3.20).

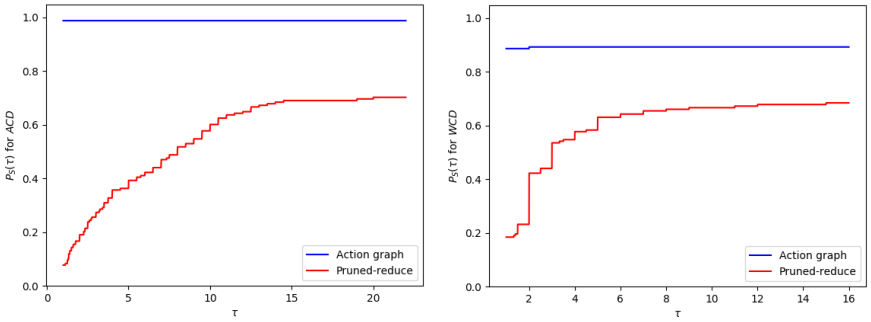
Our approach successfully managed to reduce WCD more than pruned-reduce. Pruned-reduce attempts to remove an increasing number of actions; thus, as shown in Figure 3.19b, did not attempt to remove many actions before the timeout was reached. This resulted in pruned-reduce not reducing the WCD as much as our approach (Figure 3.21b), and there were more problems for which pruned-reduce did not manage to reduce the WCD. This was also observed in the ACD reduction (Figure 3.21a). The



(a) Average time, in seconds, per GRD problem.

(b) Average number of actions that were removed.

Figure 3.20: Results for an increasing grid size. The results of our Action Graph approach are indicated by blue circles, pruned-reduce [11] is shown with red triangles. All times are given in seconds.



(a) Performance profile of ACD reduction.

(b) Performed profile of WCD reduction.

Figure 3.21: Results for comparing our Action Graph approach to pruned-reduce on all action removal experiments.

aim of pruned-reduce is to reduce WCD (not ACD); thus, it performs worse at reducing ACD than WCD.

For several problems, our approach reduced ACD but not WCD. This indicates that although the least distinctive goal could not be made more distinctive, some goals could be. Both metrics provide an important insight into the distinctiveness of an environment; thus, in future GRD experiments we propose providing both.

These experiments proved that our Action Graph approach is more scalable than a current state-of-the-art approach on grid-based navigation problems. Our approach has managed to reduce the WCD of 168 grid-based

navigation environments, with various numbers of goals and grid sizes, from an average of 6.29 (ACD = 4.69) actions to 3.02 (ACD = 1.91) actions, in an average of 4.69 s per problem.

3.10 Conclusions

As the plans to reach different goals can start with the same actions, a human's goal often cannot be recognised until their plan nears completion. By redesigning an environment, our work enables the goal of a human to be recognised after fewer observations. This is achieved through transforming a PDDL defined Goal Recognition Design (GRD) problem into an Action Graph, by means of a Breadth First Search (BFS) from each of the goal states to the initial world state. The non-distinctive plan prefixes are then extracted to calculate how distinctive the goals are, i.e., the WCD_{dep} and ACD_{dep}. Subsequently, these prefixes are processed to determine which actions can be replaced or removed. Our Shrink–Reduce method replaces actions by first shrinking the plans, then reducing the non-distinctive prefixes. Shrink–Reduce is less computational expensive than an exhaustive approach; however, when ran on Kitchen domain Shrink–Reduce only reduces the ACD_{dep} by 1, whereas exhaustive reduces ACD_{dep} by 1.67. Our action removal method is shown to increase the distinctiveness of various grid-based navigation problems, with a width/-height ranging from 4 to 16 and between 2 to 14 randomly selected goals, by an average of 3.27 actions in an average time of 4.69 s; whereas, a state-of-the-art approach, namely, pruned-reduce [11], often breaches a 10 minute time limit.

Action Graphs are acyclic and no actions are repeated; therefore, domains in which an action has different dependencies under varying circumstances are not currently solvable by our approach. For instance, if to reach one goal a human must retrieve `item1` before taking `item2`, but to reach another goal the human must take `item3` before `item2`, the action to retrieve `item2` has different dependencies. For many domains, e.g., the Kitchen domain, this strict ordering is not required, as humans are unlikely to adhere to a fixed ordering of actions, but for other domains this may be required. In future work, we will consider enabling actions to be repeated within an Action Graph. For domains such as the barman domain [42], in which different cocktails are created, a combination of our unconstrained method (as ingredients can be added in any order) and constrained method (as grasping and leaving a shot requires an optimal plan) will be required.

In future work, we intend to apply our Action Graph approach to other, closely related, research domains. For instance, GRD with stochastic ac-

tions [14, 43], plan recognition design [20], goal recognition [8] and task planning (e.g., plan legibility [44, 45]). Future experiments will hopefully demonstrate how Action Graphs can help all agents in collaborative smart environments to become more aware of each other's intentions.

Further to contextual-awareness in human-inhabited environments, our work is applicable to numerous application areas. In human computer interaction scenarios, offering a user assistance [46, 47] can benefit from recognising the user's goal sooner. In video game development [48] often the world is designed so the player's goal can be recognised, thus enabling the non-playable characters to assist or thwart them; moreover these characters' plans can also be modelled as an Action Graph. Action Graphs could allow robots to learn from humans. For instance, the graph could be built from observations or their structure adjusted to match the order humans perform actions. We hope this chapter will inspire researchers in these domains, to incorporate our Action Graph approach into their work.

Supplementary Materials: The following are available at https://www.mdpi.com/1424-8220/19/12/2741/s1 Video 1: Solving GRD Problems with Shrink–Reduce, Video 2: GRD for Navigation Domains.

Appendix 3.A Breadth First Search for Constrained Problems

Action Graphs are created by running a BFS search from each goal state to the initial state. Pseudo-code to create the Action Graph of navigation problems is shown in Algorithm 3.4. For such problems only the optimal plans for each goal are inserted into the graph. For other domains, in which all plan variations (including sub-optimal plans) are inserted into the graph, the limit variable is not set and goals may contain multiple atoms (e.g., line 6 becomes $a \in \{a' \mid a'_{eff} \subseteq G\}$).

Algorithm 3.4 BFS for generating an Action Graph containing optimal plans

> **Inputs**: list of goals, list of actions and the initial state
> **Output**: the Action Graph

function GENERATE_ACTION_GRAPH($\mathcal{G}$, A, I) ▷ Inputs: list of goals, list of actions and the initial state
    graph = ActionGraph()
    for each $G \in \mathcal{G}$ **do**
        $limit = -1$
        $q = \emptyset$
        for $a \in \{a' \mid a'_{eff} = G\}$ **do**
            $graph.insert\_action\_and\_deps(a, \{a' \mid a_{pre} = a'_{eff}\})$ ▷ Insert action and its dependencies
            if $a_{pre} \subset I$ **then**
                $limit = 1$
            else
                $q.push\_back(a)$ ▷ Initialise the BFS queue, but only if a was not already in the graph
            end if
        end for
        while $q \neq \emptyset$ **do**
            $a = q.pop()$
            for $d \in \{a' \mid a_{pre} = a'_{eff}\}$ **do**
                if $d_{pre} \subseteq I$ **or** any $\{d' \mid d_{pre} = d'_{eff}\}$ have been inserted when processing previous goal(s) **then**
                    $limit =$ max distance from goal
                else if $distance(G, d) = limit$ **or** all $\{d' \mid d_{pre} = d'_{eff}\}$ inserted when processing G **then**
                    $graph.remove(d)$ ▷ Remove action and dependant actions whose branches will not reach I
                else
                    $graph.insert\_action\_and\_deps(d, \{d' \mid d_{pre} = d'_{eff}\})$
                    $q.push\_back(d)$
                end if
            end for
            if a_{pre} not met by actions within the graph **then** ▷ failed to insert an action to reach ≥ 1 of a_{pre}
                $graph.remove(a)$
            end if
        end while
    end for
end function

Appendix 3.B Example Action Definition for Action Replacement

The modifications to the state of the environment, that can be made during the action replacement process, are defined as PDDL actions. In the example provided in Listing 3.1, an object (e.g., cup) is moved from one container (e.g., a cupboard) to another. Objects cannot be removed from the fridge or drawer (i.e., `(not (= ?s fridge))`, `(not (= ?s drawer))`), and no object can be moved into the fridge or drawer (i.e., `(not (= ?g fridge))`, `(not (= ?g drawer))`).

Listing 3.1: Example PDDL actions for action replacement.

```
(:action move-item-state-modification
    :parameters (?obj - object ?s - container ?g - container)
    :precondition (and
        (in ?obj ?s) (not (in ?obj ?g))
        (not (= ?s fridge))  (not (= ?g fridge))
        (not (= ?s drawer))  (not (= ?g drawer))
    )
    :effect ( and
        (in ?obj ?g) (not (in ?obj ?s))
    )
)
```

Appendix 3.C Exhaustive Algorithm for Reducing ACD

The exhaustive action replacement algorithm iteratively applies each modification to the Action Graph, then pairs of modifications, then triples, etc. The pseudo-code is shown in Algorithm 3.5.

Appendix 3.D Plans Generated From the Kitchen Domain

During the experiments, our action replacement algorithms were ran on a Kitchen domain, to increase the distinctiveness of the plans for making breakfast, a packed-lunch and dinner. In this Appendix we provide further details on the plans producible from this domain. This domain was originally developed by Ramírez and Geffner [24]; we expanded it to include container objects, an open container action definition and a take item from container action definition. A single permutation of the longest plan to each of the goals within the Kitchen domain are provided, in Table 3.2. There also exits plans to:

- `make-dinner()` with either just `make-salad` or `make-cheese-sandwich`
- `make-breakfast()` with `make-coffee()` instead of `make-tea()`. In addition, the option of making tea without milk and sugar or with just sugar.
- `pack-lunch()` with `make-cheese-sandwich()` instead of `make-peanut-butter-sandwich()`

Algorithm 3.5 Reduce ACD_{dep}: Exhaustive

> **Inputs**: maximum number of modification, action graph and list of actions to modify the initial state
> **Output**: list of modifications (actions) to produce the resulting environment design

```
1: function EXHAUSTIVE(N, graph, stateModifications)
2:     lowestAcd = graph.calculate_acd()
3:     for n = 1; n < N; n++ do
4:         mods = stateModifications[: n]
5:         do
6:             if isNotValid(mods) then    ▷ if mods contains changes that affect the same variable:
7:                 continue
8:             end if
9:             graph.apply_all_modifications(mods) ▷ i.e., replaces actions affected by the modifications with their replacements
10:            acd = graph.calculate_acd()
11:            if acd < lowestAcd then
12:                lowestAcd = acd
13:                bestMods = mods
14:            end if
15:            graph.undo_all_modifications(mods)
16:        while next_combination(stateModifications, n, mods)
17:    end for
18:    return bestMods
19: end function
```

Appendix 3.E Detailed Results Graphs

In the experiments section the performance profiles for the reduction in ACD_{dep} and WCD_{dep} are shown. In this Appendix graphs showing the average reduction for each dataset are provided independently. These provide further detail on the performs of the different approaches. The graphs for the action replacement experiments are shown, followed by the action removal graphs. All error bars show the minimum and maximum result.

Appendix 3.E.1 Action Replacement

Our action replacement experiments compare our Shrink–Reduce and exhaustive methods. The exhaustive approach was ran for varying values of N, i.e., 1, 2, 3 and 4; which are named exhaustive1, exhaustive2, exhaustive3 and exhaustive4 in the results. A test domain was developed containing actions to open containers and take items; the state of the environment can be modified by changing which container items are in. Experiments were ran on datasets with a varying number of variables (items), values (containers) and goals. The corresponding detailed results, showing the

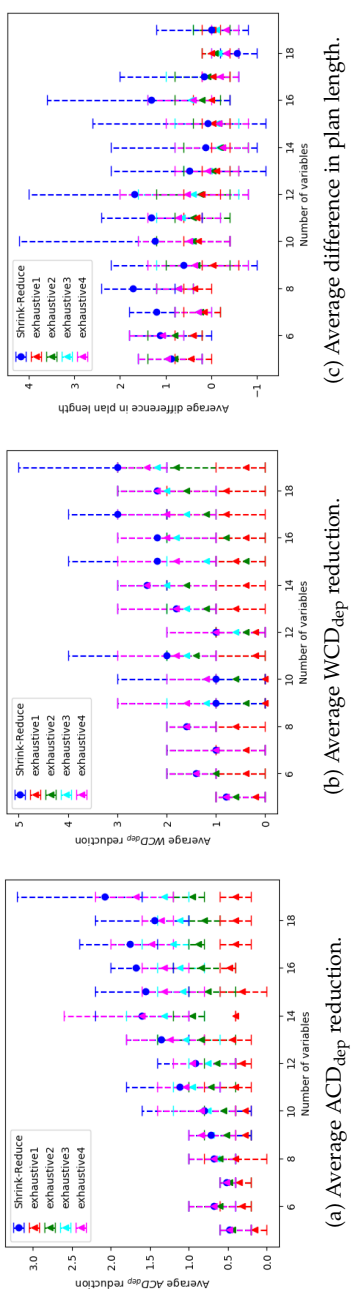
Table 3.2: Longest plans for the goals from the Kitchen domain.

(made_breakfast)	**(packed_lunch)**	**(made_dinner)**
open(cupboard1)	open(cupboard1)	open(cupboard1)
open(cupboard2)	open(cupboard2)	open(fridge)
open(drawer)	open(drawer)	open(cupboard2)
open(fridge)	open(fridge)	take(salad-tosser cupboard2)
take(kettle)	take(peanut-butter fridge)	take(bowl cupboard2)
take(cloth)	take(bread cupboard1)	take(plate cupboard2)
take(tea-bag cupboard1)	take(knife drawer)	take(dressing fridge)
take(sugar cupboard1)	take(plate cupboard2)	take(cheese fridge)
take(cereal cupboard1)	take(lunch-bag cupboard2)	take(bread cupboard1)
take(bread cupboard1)	make-peanut-butter-sandwich()	make-salad()
take(water-jug cupboard2)	pack-lunch()	make-cheese-sandwich()
take(cup cupboard2)		make-dinner
take(bowl cupboard2)		
take(knife drawer)		
take(spoon drawer)		
take(butter fridge)		
take(milk fridge)		
make-cereal()		
make-toast()		
boil-water()		
make-tea()		
use(toaster)		
make-toast()		
make-buttered-toast()		
make-breakfast()		

average ACD_{dep} reduction, WCD_{dep} reduction and the impact (difference) in average plan length the resulting environment design has, are provided in Figures 3.22–3.24. As the datasets contains an element of randomness, all points on the graphs shown the average result for 5 problems.

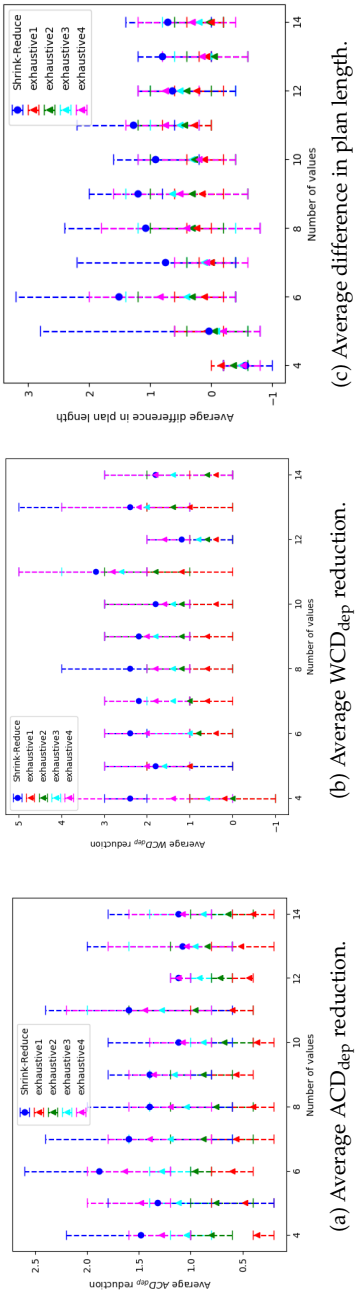
Appendix 3.E.2 Action Removal

The action removal experiments compare our approach to the pruned-reduce method by Keren et al. [11]. A grid-based navigation domain was provided as input. The average ACD and WCD reduction for an increasing number of goals and varying grid sizes (with equal width and height) are provided in Figures 3.25 and 3.26. As the datasets contains an element of randomness, all points on the graphs shown the average result for 8 problems.



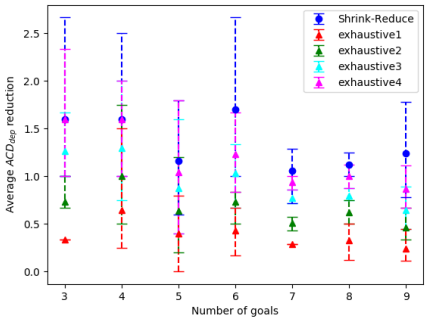
(a) Average ACD_{dep} reduction. (b) Average WCD_{dep} reduction. (c) Average difference in plan length.

Figure 3.22: Results for an increasing number of variables.

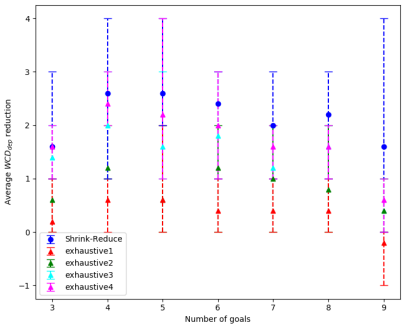


(a) Average ACD_{dep} reduction. (b) Average WCD_{dep} reduction. (c) Average difference in plan length.

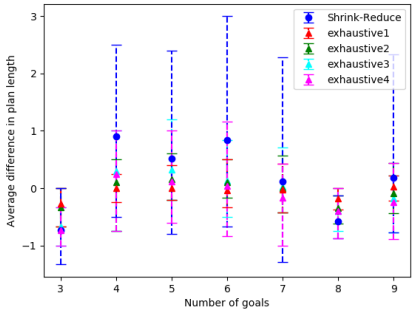
Figure 3.23: Results for an increasing number of values.



(a) Average ACD_{dep} reduction.

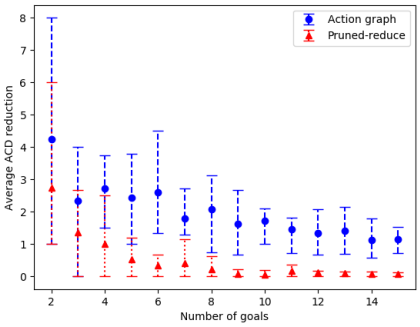


(b) Average WCD_{dep} reduction.

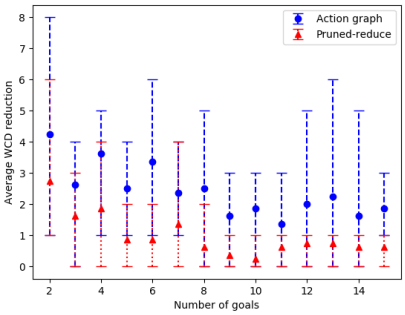


(c) Average difference in plan length.

Figure 3.24: Results for an increasing number of goals.

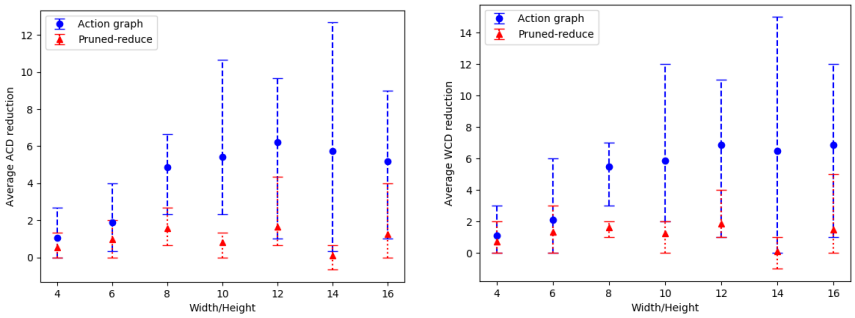


(a) Average ACD reduction.



(b) Average WCD reduction.

Figure 3.25: Results for an increasing number of goals. The results of our Action Graph approach are indicated by blue circles, pruned reduce [11] is shown with red triangles.



(a) Average ACD reduction.

(b) Average WCD reduction.

Figure 3.26: Results for an increasing grid size. The results of our Action Graph approach are indicated by blue circles, pruned-reduce [11] is shown with red triangles.

References

[1] P. Masters and S. Sardina. *Cost-Based Goal Recognition in Navigational Domains*. J. Artif. Intell. Res., 64:197–242, 2019.

[2] R. F. Pereira, N. Oren, and F. Meneguzzi. *Landmark-Based Approaches for Goal Recognition as Planning*. arXiv preprint arXiv:1904.11739, 2019.

[3] Y. Zhu, K. Zhou, M. Wang, Y. Zhao, and Z. Zhao. *A Comprehensive Solution for Detecting Events in Complex Surveillance Videos*. Multimed. Tools Appl., 78(1):817–838, 2019.

[4] K. Yordanova, S. Lüdtke, S. Whitehouse, F. Krüger, A. Paiement, M. Mirmehdi, I. Craddock, and T. Kirste. *Analysing Cooking Behaviour in Home Settings: Towards Health Monitoring*. Sensors, 19(3), 2019.

[5] U. Naeem, J. Bigham, and J. Wang. *Recognising Activities of Daily Life Using Hierarchical Plans*. In Smart Sensing and Context, pages 175–189, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[6] R. G. Freedman and S. Zilberstein. *Integration of Planning with Recognition for Responsive Interaction Using Classical Planners*. In Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 4581–4588, 2017.

[7] M. Ramírez and H. Geffner. *Probabilistic Plan Recognition Using Off-the-Shelf Classical Planners*. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI'10, pages 1121–1126. AAAI Press, 2010.

[8] R. F. Pereira, N. Oren, and F. Meneguzzi. *Landmark-based Heuristics for Goal Recognition*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 3622–3628. AAAI Press, 2017.

[9] Y. E-Martin, M. D. R-Moreno, and D. E. Smith. *A Fast Goal Recognition Technique Based on Interaction Estimates*. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI'15. AAAI Press, 2015.

[10] H. Harman and P. Simoens. *Solving Navigation-Based Goal Recognition Design Problems with Action Graphs*. In AAAI Workshops on Plan, Activity, and Intent Recognition (PAIR-19), 2019.

[11] S. Keren, A. Gal, and E. Karpas. *Goal Recognition Design*. In Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS, pages 154–162. AAAI Press, 2014.

[12] C. Wayllace, P. Hou, W. Yeoh, and T. C. Son. *Goal Recognition Design with Stochastic Agent Action Outcomes*. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI'16, pages 3279–3285. AAAI Press, 2016.

[13] T. C. Son, O. Sabuncu, C. Schulz-Hanke, T. Schaub, and W. Yeoh. *Solving Goal Recognition Design Using ASP*. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI'16, pages 3181–3187. AAAI Press, 2016.

[14] C. Wayllace, S. Keren, W. Yeoh, A. Gal, and E. Karpas. *Accounting for Partial Observability in Stochastic Goal Recognition Design: Messing with the Marauder's Map*. Proceedings of the 10th Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP), pages 33–41, 2018.

[15] S. Keren, A. Gal, and E. Karpas. *Strong Stubborn Sets for Efficient Goal Recognition Design.* In Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS, pages 141–149. AAAI Press, 2018.

[16] J. Wu, A. Osuntogun, T. Choudhury, M. Philipose, and J. M. Rehg. *A Scalable Approach to Activity Recognition Based on Object Use*. In Proceedings of the Eleventh IEEE International Conference on Computer Vision, pages 1–8. IEEE, 2007.

[17] S. Keren, A. Gal, and E. Karpas. *Goal Recognition Design for Non-Optimal Agents*. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI'15, pages 3298–3304. AAAI Press, 2015.

[18] S. Keren, A. Gal, and E. Karpas. *Goal Recognition Design with Non-Observable Actions*. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI'16, pages 3152–3158. AAAI Press, 2016.

[19] C. Wayllace, P. Hou, and W. Yeoh. *New Metrics and Algorithms for Stochastic Goal Recognition Design Problems*. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI'17, pages 4455–4462. AAAI Press, 2017.

[20] R. Mirsky, K. Gal, R. Stern, and M. Kalech. *Goal and Plan Recognition Design for Plan Libraries*. ACM Trans. Intell. Syst. Technol., 10(2):14:1–14:23, 2019.

[21] M. Helmert. *The Fast Downward Planning System*. J. Artif. Intell. Res., 26:191–246, 2006.

[22] M. Ghallab, D. Nau, and P. Traverso. *Part I - Classical Planning*. In Automated Planning, The Morgan Kaufmann Series in Artificial Intelligence, pages 17 – 18. Morgan Kaufmann, Burlington, 2004.

[23] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning: Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers, 1st edition, 2013.

[24] M. Ramírez and H. Geffner. *Plan Recognition As Planning*. In Proceedings of the Twenty-First International Joint Conference on Artifical Intelligence, IJCAI'09, pages 1778–1783, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[25] C. W. Geib and R. P. Goldman. *Plan Recognition in Intrusion Detection Systems*. In Proceedings of the DARPA Information Survivability Conference and Exposition II., volume 1 of *DISCEX'01*, pages 46–55 vol.1, 2001.

[26] S. Keren, R. Mirsky, and C. Geib. *Plan Activity and Intent Recognition Tutorial*, 2019. Accessed: 2019-25-03. Available from: http://www.planrec.org/Tutorial/Resources_files/pair-tutorial.pdf.

[27] J. Rafferty, C. D. Nugent, J. Liu, and L. Chen. *From Activity Recognition to Intention Recognition for Assisted Living Within Smart Homes*. IEEE T. Hum.-Mach. Syst., 47(3):368–379, 2017.

[28] G. Singla, D. J. Cook, and M. Schmitter-Edgecombe. *Recognizing Independent and Joint Activities Among Multiple Residents in Smart Environments*. J. Ambient Intell. Humaniz. Comput., 1(1):57–63, 2010.

[29] F. Bisson, H. Larochelle, and F. Kabanza. *Using a Recursive Neural Network to Learn an Agent's Decision Model for Plan Recognition*. In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, IJCAI'15, pages 918–924. AAAI Press, 2015.

[30] L. Amado, R. F. Pereira, J. Aires, M. Magnaguagno, R. Granada, and F. Meneguzzi. *Goal Recognition in Latent Space*. In Proceedings of the International Joint Conference on Neural Networks, IJCNN, pages 1–8. IEEE, 2018.

[31] P. C. Roy, S. Giroux, B. Bouchard, A. Bouzouane, C. Phua, A. Tolstikov, and J. Biswas. *A Possibilistic Approach for Activity Recognition in Smart Homes for Cognitive Assistance to Alzheimer's Patients*, pages 33–58. Atlantis Press, Paris, 2011.

[32] K. Yordanova, F. Krüger, and T. Kirste. *Context Aware Approach for Activity Recognition Based on Precondition-Effect Rules*. In IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pages 602–607. IEEE, 2012.

[33] C. W. Geib and R. P. Goldman. *A Probabilistic Plan Recognition Algorithm Based on Plan Tree Grammars*. Artif. Intell., 173(11):1101 – 1132, 2009.

[34] H. A. Kautz and J. F. Allen. *Generalized Plan Recognition*. In Proceedings of the Fifth AAAI National Conference on Artificial Intelligence, volume 86 of *AAAI'86*, pages 32–37. AAAI Press, 1986.

[35] H. A. Kautz. *A Formal Theory of Plan Recognition*. PhD thesis, University of Rochester, Department of Computer Science, 1987.

[36] R. Mirsky, Y. K. Gal, and S. M. Shieber. *CRADLE: An Online Plan Recognition Algorithm for Exploratory Domains*. ACM Trans. Intell. Syst. Technol., 8(3):45:1–45:22, 2017.

[37] D. Avrahami-Zilberbrand and G. A. Kaminka. *Fast and Complete Symbolic Plan Recognition*. In Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, IJCAI'05, pages 653–658, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[38] M. Vilain. *Getting Serious About Parsing Plans: A Grammatical Analysis of Plan Recognition*. In Proceedings of the Eighth National Conference on Artificial Intelligence, volume 1 of *AAAI'90*, pages 190–197. AAAI Press, 1990.

[39] F. Kabanza, J. Filion, A. R. Benaskeur, and H. Irandoust. *Controlling the Hypothesis Space in Probabilistic Plan Recognition*. In Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI'13, pages 2306–2312. AAAI Press, 2013.

[40] S. Holtzen, Y. Zhao, T. Gao, J. B. Tenenbaum, and S. Zhu. *Inferring Human Intent From Video by Sampling Hierarchical Plans*. In IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, pages 1489–1496. IEEE, 2016.

[41] E. D. Dolan and J. J. Moré. *Benchmarking Optimization Software with Performance Profiles*. Math. Program., 91(2):201–213, 2002.

[42] C. L. López, S. J. Celorrio, and Ángel García Olaya. *The Deterministic Part of the Seventh International Planning Competition*. Artif. Intell., 223:82 – 119, 2015.

[43] S. Keren, L. Pineda, A. Gal, E. Karpas, and S. Zilberstein. *Redesigning Stochastic Environments for Maximized Utility*. In AAAI Workshops on Plan, Activity, and Intent Recognition (PAIR-17), 2017.

[44] T. Chakraborti, A. Kulkarni, S. Sreedharan, D. E. Smith, and S. Kambhampati. *Explicability? legibility? predictability? transparency? privacy? security? the emerging landscape of interpretable agent behavior*. In Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS'19, 2019.

[45] A. D. Dragan, K. C. Lee, and S. S. Srinivasa. *Legibility and Predictability of Robot Motion*. In Proceedings of the 8th ACM/IEEE International Conference on Human-robot Interaction, HRI '13, pages 301–308, Piscataway, NJ, USA, 2013. IEEE Press.

[46] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. *The LumièRe Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users*. In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, UAI'98, pages 256–265, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[47] J. Hong. *Goal Recognition Through Goal Graph Analysis*. J. Artif. Intell. Res., 15:1–30, 2001.

[48] M. Fagan and P. Cunningham. *Case-Based Plan Recognition in Computer Games*. In International Conference on Case-Based Reasoning Research and Development, pages 161–170, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

4

Proactive Robot Assistance

"*A robot may not injure a human being or, through inaction, allow a human being to come to harm.*"
Isaac Asimov (1942)

Action Graphs for Proactive Robot Assistance in Smart Environments

H. Harman & P. Simoens

Published in Journal of Ambient Intelligence and Smart Environments, 12 (2), 2020.

Our Goal Recognition (GR) approach assumed that the observed agent aimed to achieve one of the predefined hypothesis goals; however, this will not always be the case. Rather than aiming to reach one of the predefined goals, a human could perform any sequence of actions. Therefore, based on what actions are observed, our work aims to predict the actions a human is likely to perform next. In this chapter, a different method for deriving an Action Graph from a problem defined in PDDL is presented. The value of each node is initialised to 0. Then, when an action is observed, the node values are updated and the highest valued action nodes are extracted. Subsequently, a robot executes one of the extracted, i.e., predicted, actions if it does not impact the flow of the human by obstructing or delaying them. Our Action Graph approach is applied to a Kitchen domain.

4.1 Introduction

An increasing number of robots are being deployed in smart environments to work alongside and assist humans in their daily activities. Rather than a human explicitly stating their goal and conversing with the robot to determine how they will achieve that goal, our work aims to automatically predict what actions the human will perform next (e.g., open a door, switch on an appliance or take an item from a cupboard) and determine which of those actions could be executed by a robot. Figure 4.1 shows a conceptual overview.

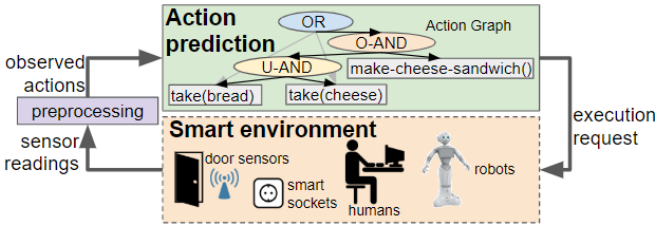


Figure 4.1: Conceptual overview.

There are many challenges to predicting the actions of humans. For instance, people might pursue multiple goals concurrently, e.g., while wait-

ing for the kettle to boil to make a cup of tea, a person could hang-up their washing. Further, goals are not always well-defined: a prepared cup of tea could be a goal, or a subgoal to be realised as part of the goal to prepare breakfast.

Our proposed system aims to adhere to the following constraints. First, the robot should have minimal impact on the action flow of the observed agent, e.g., not attempt to take an item from a cupboard at the same time as the human. Second, the system should cope with invalid and missing observations caused by noisy erroneous sensor readings and the human making mistakes, e.g., opening the wrong cupboard. Moreover, due to privacy concerns and humans preferring non-intrusive sensors [1], some actions, e.g., navigation actions, are unobservable. Third, the algorithms should work online, i.e., update the predictions each time an action is observed.

In this work, actions are discrete and the term observation refers to an action that has already been derived from raw (continuous) sensor data and low-level activity recognition algorithms, such as the work by Tremblay et al. [2]. The deployment of hardware (i.e., IoT devices and robots) and the integration of activity recognition algorithms are beyond the scope of this chapter.

In this chapter, we aim to answer two of the research questions that were presented in the introductory chapter. i) Can a structure similar to those created by library-based approaches be generated from a PDDL defined GR problem? ii) When provided with a PDDL defined GR problem as input, how can an observed agent's next actions be predicted and used by a robot to provide assistance?

The main contributions of this chapter are algorithms for action prediction and for deciding which of the predicted actions should be executed by a robot. The action prediction aspect is performed by an Action Graph, a graphical structure that models the dependencies between the actions an observed (human) agent can perform. An Action Graph is automatically generated from a problem defined in Planning Domain Definition Language (PDDL). When an observation is received, its node values are updated and the action nodes whose values are above a threshold are extracted, i.e., the predicted actions. The predicted actions are then mapped to a goal state before sending them to a robot. By running a classical task planner, the robot generates a task plan containing a set of actions to reach the goal state from its current state. The robot then decides to pursue the goal based on how short its plan is and how far the human is from per-

forming the predicted action.

The remainder of this chapter is structured as follows. A summary of related work is presented in Section 4.2. In Section 4.3, the action prediction problem is formalised and the Kitchen domain is introduced. Section 4.4 outlines the structural features of an Action Graph and how it is generated. The node value update rules are detailed in Section 4.5 and the extraction of the predicted actions in Section 4.6. How the robot decides whether to execute an action (or not) is presented in Section 4.7. Section 4.8 discusses the results of our action prediction experiments and Section 4.9 presents a proof of concept detailing which actions the robot executes.

4.2 Related Work

Within the literature, studies on recognising the intentions of an observed agent interchange various terms and these often have different definitions, e.g., goal, plan, intention and activity recognition [2–4]. In this section some of the existing approaches to intention recognition are introduced, in particular the focus is on those which have similar (hierarchical) structures or take PDDL as input. Intention recognition has been applied to different application domains including computer gaming [5], human-computer interaction [6], energy saving [7] and social robotics [8]. As our work focuses on proactive robotic assistance, other works that integrate intention recognition with robot response are also discussed.

4.2.1 Intention Recognition

Methods for intention recognition can be broadly categorised as data-driven and knowledge-driven methods [9, 10]. Data-driven approaches train a recognition model from a large dataset [10–13]. The main disadvantages of this method are, that often a large amount of labelled training data is required and the produced models often only work on data similar to the training set [14, 15]. Since our work belongs to the category of knowledge-driven methods, data-driven methods are not discussed further.

Knowledge-driven approaches rely on a logical description of the actions agents can perform. They can be further divided into approaches that search through a library of predefined plans (also known as "recognition as parsing") and approaches that solve a symbolic recognition problem, i.e., "recognition as planning" [16]. These are discussed in turn.

4.2.1.1 Recognition as Parsing

Recognition as parsing tends to be fast and allows multiple concurrent plans to be detected [17–19], but is often considered to be less flexible [3, 20] because a planning library containing all actions and their orderings must

be produced a priori. Hierarchical structures are usually developed, which include abstract actions that can be decomposed into concrete (observable) actions.

In [21], a Temporal AND-OR tree was constructed from a library of plans to determine which objects a human will navigate to. Our method of representing a human's actions in an Action Graph is inspired by this AND-OR tree; however, the construction of our graph is considerably different (i.e., Action Graphs are derived from PDDL) and our node value update rules differ. Moreover, their AND-OR tree only contains `ORDERED-AND` nodes, whereas Action Graphs include both `ORDERED` and `UNORDERED-AND` nodes as often actions are not strictly ordered.

A set of action sequence graphs is derived from a library of plans in [22]. This set is compared to an action sequence graph created from a sequence of observations to find the plan most similar to the observation sequence. Their approach was shown to perform well on misclassified (incorrect) sensor observations and missing actions, but they did not investigate multiple interleaving goals.

Kautz et al. [17, 23] considered many of the features (e.g., multiple goals, partial plans) we believe are key for goal recognition and action prediction to be deployed within a smart environment. They introduce a language to describe a hierarchy of actions. Based on which low level actions are observed, the higher level task(s) an agent is attempting to achieve is inferred. Their work describes a formal theory and as far as we are aware was not implemented.

4.2.1.2 Recognition as Planning

Recognition as planning is a more recently proposed approach, in which languages normally associated with task planning, such as PDDL, define the actions agents can perform (along with their preconditions and effects) and world states. This enables a single set of action definitions to be written in a standard language that can be utilised for both action prediction and robot task planning. Moreover, whereas in recognition as parsing there are usually only action definitions, planning-based approaches allow for the inclusion of state knowledge, such as what objects are found within the environment and their locations.

In [3, 20], it was proposed to view goal recognition as the inverse of planning. In symbolic task planning, a problem is formulated in terms of an initial and goal state, and task planners find the appropriate set of actions (i.e., a task plan) that changes the current state into the desired goal state [24]. In goal recognition, the initial state, multiple hypothesis goal states, a set

of actions and a list of observations are provided as input, and intention recognisers find the most probable goal and/or plan. The approach by Ramírez & Geffner [3] handles suboptimal plans, but cannot cope with multiple interleaving goals and does not include action prediction. As a planner needs to be called twice for every possible goal, to find the difference in the cost of the plan to reach the goal with and without taking the observations into consideration, this approach is computationally expensive. In [25], the work of Ramírez & Geffner [3] was extended to find the joint probability of pairs of goals rather than a single goal. Their work aims to handle multiple interleaving goals but also suffers from large computational costs. Although initial approaches were computationally heavy as they required a task planner to be called multiple times [3, 20, 25], the latest advances in recognition as planning algorithms have greatly improved this [4, 26].

Plan graphs were proposed in [26], to prevent a planner from being called multiple times. A plan graph, which contains actions and propositions labelled as either true, false or unknown, is built from a planning problem and updated based on the observations. Rather than calling a planner, the graph is used to calculate the cost of reaching the goals. Our Action Graph structure differs greatly from a plan graph as Action Graphs only contain actions and the constraints between those actions.

More recently Pereira et al. [4] significantly reduced the recognition time by finding landmarks, i.e., states that must be passed for a particular goal to be reached. As landmarks rather than all actions are reasoned on, action prediction cannot easily be performed. Moreover, neither [26] nor [4] investigate multiple interleaving plans.

As far as we are aware, our work is the first to take a recognition as planning approach to solve multiple concurrent subgoal action prediction problems, by deriving a graph structure similar to those used by some recognition as parsing approaches from a PDDL defined problem. Moreover, the majority of work on recognition as planning focuses on goal recognition and not on how the observed agent will achieve their aim, e.g., these approaches recognise that a human's goal is to make breakfast but not if they will make tea or coffee to reach that goal, whereas our approach predicts what actions the observed agent will perform next.

4.2.2 Proactive Robot Assistance

A robot should assist a human by timely executing an action that helps the human to achieve their goals. In factory and assembly environments the goal and/or plan is often known upfront. Johannsmeier et al. [27] create a

high level (team) plan for human and robot workers offline, and map this to robot specific hardware at run-time. Nonetheless, there are many situations in which the plan and goal are not known to the system in advance. Further, several recent studies [28, 29] show that humans prefer robots that are proactive, i.e., a robot which autonomously assists a person whenever the robot is able to. Like our approach, the works discussed below process observations to determine part (or potentially all) of the observed agent's plan, which the observer can then execute.

In [30, 31], planning networks are derived from causal links extracted from the PDDL problem description. These networks contain the decisions that will be taken by the robot and human (i.e., team plans). They are created offline due to large computational costs, and then processed by an online look-up procedure, which permits a robot to assist a human. Their approach focuses on what uncontrollable decisions a human makes and what action(s) the robot should plan in response. Their work does not aim to prevent the robot from disrupting the flow of a human. Whereas, our work aims to achieve this by only allowing the robot to perform a predicted action if, in comparison, the robot's plan is short and the human is far from completing the prediction.

By contrast, in [32] the most probable goals are determined by running the already cited work on goal recognition as planning [3]. A list of the most likely propositions is then extracted; these propositions are set as the robot's goal. As the goal recogniser [3] calls the planner twice per goal, this approach is computationally expensive. Similar to our work, Levine & Williams [30] and Freedman & Zilberstein [32] take ideas from symbolic task planning as a starting point for their prediction algorithms, but they focus on an agent performing a single goal. Our work also considers multiple, interleaving and partially completed goals (i.e., subgoals).

4.3 Problem Statement

An Action Graph is created from an action prediction problem defined in PDDL. This section introduces the concepts of PDDL and Domain Transition Graphs (DTGs), an intermediate representation that is processed during creation of an Action Graph. A description of the concrete smart kitchen scenario our work aims to address is also provided.

4.3.1 Problem Formalisation

Formally, an intention recognition problem can be defined as $T = (F, I, A, O, \mathcal{G})$, where F is a set of atoms, $I \subseteq F$ is the initial state, A is a set of actions, $\mathcal{G}$ is the set of all possible goals and O is the sequence of observed actions [3]. Actions contain preconditions $a_{pre} \subseteq F$ and effects a_{eff}, which

includes add effects $a_{eff+} \subseteq F$, e.g., `(open cupboard)`, and delete effects $a_{eff-} \subseteq F$, e.g., `(not (open cupboard))`. Our action prediction method produces $\mathcal{A}$, which contains the actions the observee is likely to perform next.

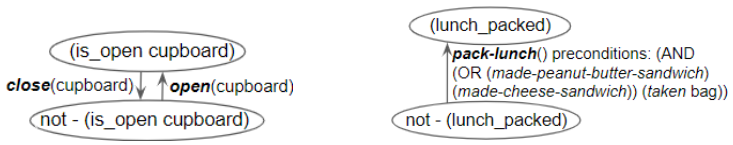
Intention recognition is often viewed as the inverse of planning. When invoked, planners translate PDDL into structures, e.g., DTGs, that can be efficiently searched [24, 33]. DTGs are created from a planning problem, i.e., $P = (F, I, A, G)$ [34, 35]. G is a goal state, specified in PDDL using *or* and *and* statements.

Each variable has its own DTG, in which the nodes contain a variable's possible values. The edges are called transitions and describe what actions (and/or axioms) are required to move between values. If there are multiple ways (plans/actions) to transition between two values, the corresponding edge will have multiple labels.

4.3.2 Running Example: Smart Home Kitchen

Throughout this chapter, examples from the Kitchen dataset are provided to help describe our approach. This dataset was created by Ramırez et al. [3, 20] based on the work by Wu et al. [36], but has been extended to include `open(?container)`, `close(?container)` and `take(?item ?container)` actions. In this dataset there are 3 hypothesis goals: $\mathcal{G}$ ={`(made_breakfast)`, `(lunch_packed)` and `(made_dinner)`}. These goals require multiple items to be taken and other actions to be performed, e.g., part of the plan to make breakfast involves taking bread and making toast. Each goal can be achieved by multiple plans, e.g., for `(lunch_packed)` to be reached a person must always perform the `take(lunch_bag)` action and either perform the `make-peanut-butter-sandwich()` or `make-cheese-sandwich()` action.

A subset of the DTGs created from a Kitchen problem are depicted in Figure 4.2. In this problem, variables transition between true and false values. As making a packed lunch has multiple possible plans, the transition to the value `(lunch_packed)` being true has (at least) two transition labels. Details on the plans producible from this domain are provided in Appendix 4.A.



(a) Open cupboard. (b) Pack lunch.

Figure 4.2: Example DTGs taken from a Kitchen problem.

4.4 Action Graphs

Action prediction is performed by building an Action Graph, updating the value of the graph's nodes when an observation (i.e., action) is received, and then extracting the highest valued actions and their dependencies. This section provides details on the structural features of an Action Graph and on how it is derived from a PDDL defined problem.

4.4.1 Structural Features

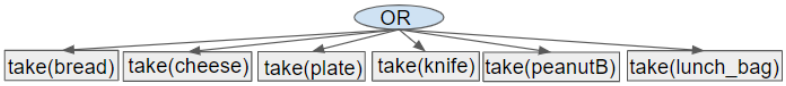
Action Graphs model the order constraints, i.e., dependencies, between actions. Dependencies are defined, in this chapter, as actions that set one or more of the dependant's preconditions. For instance, action 1 is said to be dependent on action 2 if action 2 fulfils one (or more) of action 1's preconditions. An Action Graph contains `OR`, `ORDERED-AND`, `UNORDERED-AND` and leaf nodes. Leaf nodes are also referred to as action nodes, as each one is associated with an action. `ORDERED-AND` denotes that its children are performed in order, and `UNORDERED-AND` denotes its children can be performed in any order. For `OR` nodes, one or more of its children can be performed. The root node is always an `OR` node, and will receive a new child for every action inserted into the graph. Throughout this chapter, unless otherwise stated, the term parent(s) always refers to the direct parent(s) of a node, the same goes for child/children.

The term graph is used, rather than tree, because action and `ORDERED-AND` nodes can have multiple parent nodes. Action Graphs are acyclic, i.e., do not contain any cycles; actions have a fixed set of dependencies, and if an action 1 depends on action 2, action 2 cannot depend on action 1.

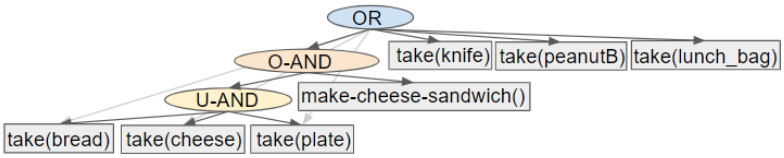
4.4.2 Creation

DTGs [24] are generated from a planning problem $P = (F, I, A, G)$ and not from an action prediction problem $T = (F, I, A, O, \mathcal{G})$. Therefore, a goal state G is created by placing all elements in the set of hypothesis goals ($\mathcal{G}$) in an `or` statement. For each DTG, the DTG's transition labels are iterated over to extract actions along with their dependencies, and insert them into the Action Graph. The Action Graph construction steps described in this section are illustrated in Figure 4.3. For simplicity, open/close actions are omitted.

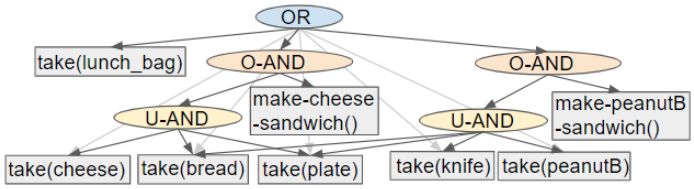
Actions associated with transition labels that do not have any preconditions, and therefore are assumed to have no dependencies, are appended as children of the root node, e.g., `take(bread)` shown in Figure 4.3a. These nodes may gain additional parent(s) when subsequent actions, which they are dependencies of, are inserted into the graph.



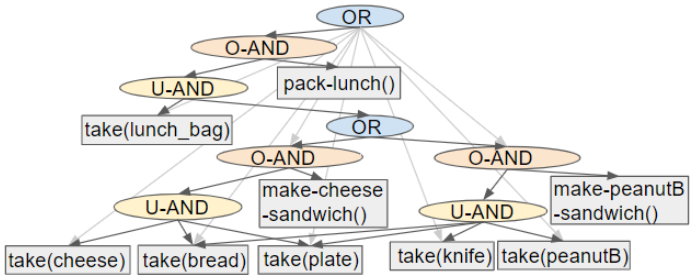
(a) Actions without any dependencies are appended to the root node's children.



(b) The `make-cheese-sandwich()` action has three dependencies which are set as an `U-AND` node's children. The `U-AND` node followed by the `make-cheese-sandwich()` action node are set as an `O-AND` node's children.



(c) The `make-peanutB-sandwich()` action is inserted in the same way as the `make-cheese-sandwich()` action.



(d) The `pack-lunch` action has a dependency on `take(lunch_bag)` and either the `make-peanutB-sandwich()` or `make-cheese-sandwich()` actions. Therefore, as well as inserting a new `U-AND` and `O-AND`, an `OR` node is also created.

Figure 4.3: Figures showing some of the steps performed during the creation of the Action Graph for the Kitchen domain. To simplify our examples `open` and `close` actions are excluded, i.e., the original unmodified Kitchen problem [3] is used. For readability, arrows from the root node to nodes that have another parent have been put in light grey. O-AND stands for `ORDERED-AND` and U-AND is `UNORDERED-AND`. Some action names have been shortened, e.g., `peanut-butter` to `peanutB`.

Actions associated with transition labels with preconditions are inserted after all of their dependencies. To perform this insertion several nodes are created including: the action node itself; if the action has more than one dependency an `UNORDERED-AND` node containing all the dependencies as its children; and an `ORDERED-AND` node whose children are the `UNORDERED-AND`

node (or single dependency) followed by the action node. The ORDERED-AND node is appended as a child of the root node. Figures 4.3b and 4.3c show the Action Graphs after this process has been performed on the make-cheese-sandwich() and make-peanut-butter-sandwich() actions, respectively.

If an action has dependencies, it will only ever have one parent, of type ORDERED-AND. Therefore, when an action has dependencies, its parent ORDERED-AND node is used to connect it to its dependants.

For transitions with multiple labels (e.g., the DTG in Figure 4.2b), an OR node is inserted into the Action Graph. This node is appended to the (UN)ORDERED-AND node's children and the multiple possible dependencies set as the OR node's children. An example is provided in Figure 4.3d in which the pack-lunch() action requires either make-peanut-butter-sandwich() or make-cheese-sandwich() to be performed first.

Each action node may, optionally, have a list of reverse actions. Two actions are the reverse of each other if one sets an atom a_{eff+} and the other deletes it a_{eff-}. For example, close(cupboard) is the reverse of open(cupboard). Reverse actions are identified by checking if a DTG has opposite edges between the same two values, e.g., the DTG depicted in Figure 4.2a. Details on the importance of reverse actions are provided in Section 4.5.2.

The Action Graph generation process finishes when all actions have been inserted and reverse actions have been identified. Action Graphs are quick to generate (see Experiment in Section 4.8) but the process will typically be performed offline. If new actions become available, e.g., because the human installs a new IoT device in their house, these new actions can be inserted into the graph by executing the method described above. The size of the produced graph depends on the number of actions and what constraints those actions have. Readers familiar with Fast Downward (FD) [24] will know that DTGs also contain transitions labelled with axioms; axioms are not inserted into the Action Graph.

4.5 Node Value Updates

Every time the environment observes the human performing an action, the Action Graph's node values are updated according to the algorithms described in this section. In this section, the term AND node refers to either a UNORDERED-AND or a ORDERED-AND node.

4.5.1 Updating Node Values Based on Observations

Each node has an initial (minimum) value of 0 and a maximum value of 1. When an observation $o \in O$ is received, the value of the corresponding action node is set to 1. Subsequently, a value update procedure is executed,

which contains an upward and a downward pass. The upward pass traverses the Action Graph from the observed action's node to the root node. Then the downward pass updates all node values in a depth first trajectory starting at the root. The purpose of this value update process is to increase the value of the actions, which are most likely to be performed next.

Pseudo-code of the upward pass is shown in Algorithm 4.1. After setting the value of the observed action's node to 1 (line 3), the parent nodes are updated recursively (lines 13). If the observed action has a reverse action, the reverse action's value is reset (see Section 4.5.2). It does not matter in which order a node's parents are updated. During the upwards recursion, the argument of the method call in line 1 will always be of node type `OR`, `UNORDERED-AND` or `ORDERED-AND`.

Algorithm 4.1 Update node value upwards

```
1:  function UPDATE_NODE_VALUE_UPWARDS(node)
2:      if node is an action node then                          ▷ The observed action
3:          node.value = 1.0
4:          for each r in node.reverseActions do r.RESET_NODE_VALUE() end for
    ▷ See Section 4.5.2
5:      else if node is an OR node then
6:          node.value = max(children.values)                   ▷ Maximum value of node's
    children
7:      else if node is an UNORDERED-AND node then
8:          node.value = mean(v(c))   ∀c ∈ node.children        ▷ Mean of node's children
9:      else if node is an ORDERED-AND node then
10:         U ← c   ∀c ∈ node.children after last child with node value 1
11:         node.value = mean(v(c))   ∀c ∈ U    ▷ Mean of children after (and including)
    1st observed child
12:     end if
13:     for each parent in node.parents do
            UPDATE_NODE_VALUE_UPWARDS(parent) end for
14: end function
15: function v(node)
16:     if node is an action node and node.value < 1.0 then return 0.0
17:     else return node.value end if
18: end function
```

- If the argument is a node of type `OR`, its value is set to the maximum value of its children (line 6) because, by the nature of the node itself, its value should not depend on the number of children it has.
- If the argument is a node of type `UNORDERED-AND`, its value becomes the mean of its children's values. We considered calculating product, rather than mean, but with product the size of the sub-trees has a much larger effect on the probability of a goal, i.e., strongly favours

shorter plans. To calculate the mean, a value of 0 replaces the actual value of unobserved action nodes (see lines 15-18). If the algorithm were to use the actual node value, node values would be increased too rapidly which causes a high rate of false positive predictions on subgoal action prediction problems.

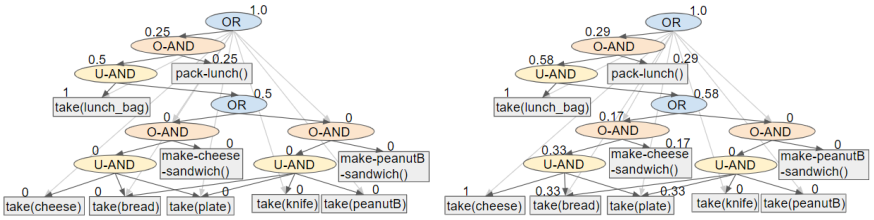
- If the argument is a node of type `ORDERED-AND`, its value becomes the mean value of its children which come after (and including) the last child whose value is 1. Like `UNORDERED-AND`, an `ORDERED-AND` node's value is based on how completed its children are. Moreover, due to the order constraints imposed on `ORDERED-AND` nodes' children, we can assume that the left most branch has been completed if the right branch has been observed. This improves Action Graphs' ability to handle missing observations.

After Algorithm 4.1 returns, a downward pass, shown in Algorithm 4.2, is performed. Starting from the root node, this recurs over all nodes in a depth-first manner. Direct children of `AND` nodes are assigned the value of their parent if that value is larger. The direct children of `OR` nodes are not updated.

Algorithm 4.2 Update node value downwards (depth-first)

```
1: function UPDATE_NODE_VALUE_DOWNWARDS(node)
2:     if node is an ORDERED-AND node or node is an UNORDERED-AND node then
3:         for each child in node.children do
               child.value = max(child.value, node.value) end for
4:     end if
5:     for each child in node.children do
               UPDATE_NODE_VALUE_DOWNWARDS(child) end for
6: end function
```

To demonstrate how these algorithms affect the nodes' values, an example is provided and depicted in Figure 4.4. When processing the first observation, namely, `take(lunch_bag)`, the only action node whose value is affected is `pack-lunch()` as it is connected to `take(lunch_bag)` via `AND` nodes. At this point determining if the person is making a cheese or peanut butter sandwich is impossible. When `take(cheese)` is observed, the values of the `pack-lunch()` action and all the actions associated with making a cheese sandwich are increased. `make-cheese-sandwich()` node's value is increased by 0.17, whereas `pack-lunch()` node's value is increased by 0.04. This is because the further (i.e., number of nodes that must be traversed) an action is from the observed action the less its value is increased (as the human could be aiming to achieve a subgoal).



(a) Node values after `take(lunch_bag)` has been observed

(b) Node values after `take(cheese)` has been observed.

Figure 4.4: Figures showing the node values after they have been updated by Algorithms 4.1 and 4.2.

4.5.2 Node Value Updates with Reverse Actions

If the state set by an observed action is reversed, the nodes whose values were increased when the action was observed should be decreased. For example, closing a cupboard is the reverse of opening that cupboard and reduces the chance of an item being taken from the cupboard, and thus any action which requires those items. Moreover, when a node value is set to 1 the action prediction algorithms will not consider this action as a plausible candidate for future actions. This is not a valid assumption in real-world domains, where people can, e.g., open a cupboard multiple times. When an action is observed, the first step (i.e., line 4 of Algorithm 4.1) resets the value of the reverse action nodes. If the node being reset is the child of an `AND` node, all its non-observed siblings must also be reset to zero. Pseudo-code and an example is provided in Appendix 4.B.

After setting the reverse action node's value to 0, each of its parents are iterated over. If the parent is an `AND` node, its children that have not yet been observed (and have not already been reset) also need to be reset. An action node is simply reset by setting its value to 0. For children that are `OR` nodes, their value becomes the maximum of their children because during the downwards pass (Algorithm 4.2) their value was updated but their children's values were not. For `AND` nodes the children are recursively reset, as they were also updated when the reverse action was observed. Once all of its children have been reset, the `AND` node's value is then updated to the mean value of its children, and the algorithm continues to recurs upwards until an `OR` node, e.g., the root, is reached.

4.6 Action Prediction with Action Graphs

Action prediction is performed by finding action nodes with a value greater than threshold θ and extracting their dependencies, including the depen-

dencies' dependencies. Dependencies are extracted because an action can only be performed if its dependencies are executed first. This process results in a map $\mathcal{A}$, which maps each action node with a value greater than θ to a list of its dependencies.

Dependencies are extracted by traversing depth-first starting from an OR DERED-AND node (i.e., the parent of a node with dependencies), and appending all actions nodes encountered to the dependency list. For OR nodes, only the most likely child is traversed, and for UNORDERED-AND nodes, the children are first sorted, highest value first. Note, the action (key) itself is also appended to the end of the dependency list during this depth-first traversal.

If an action node (a) in the map's keys is in the dependency list of another action in the map, then the latter dependency list will contain all elements in the dependency list of a. Therefore, the entry with key a is removed from the map ($\mathcal{A}$). This reduces the amount of unnecessary data; however, it is still possible (and very likely) for some actions to appear in multiple dependency lists.

Actions which are attached to an OR node (not including the root) that have a lower value than another action attached to the same OR node are removed from the predicted actions. This is because OR branches often contain similar actions, and thus when one branch is above the threshold so are the others, e.g., making a cup of coffee and making a cup of tea require many similar actions. The map of predicted actions is recreated every time an observation is received, thus removed actions can reappear in the list during subsequent iterations.

4.7 Proactive Robot Assistance

A robot can proactively assist a human by executing one of the predicted actions. The decision about which action to execute is based on the value of the predicted action and on how long it would take the robot or human to execute that action. An overview of this process is depicted in Figure 4.5. This section introduces which predicted action is selected, what information is sent to the robot and the robot's decision making process.

The Action Predictor checks if a robot can execute one of the predicted actions ($\mathcal{A}$). Starting with actions in the highest valued action's (key's) dependency list, each action is checked to see if it appears in a predefined list of actions the robot is capable of executing. If it does, an execution request is sent to the robot and if the robot chooses to execute that request, the iteration finishes. Otherwise, the iteration continues until all predicted

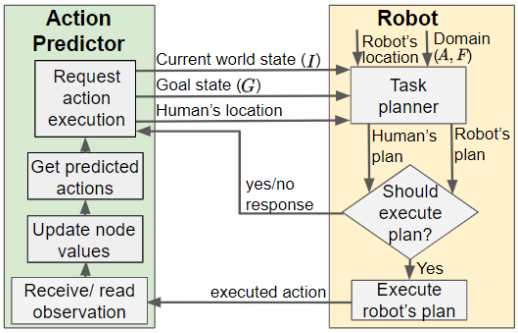


Figure 4.5: Overview of the processes the Action Predictor and robot perform.

actions (and their dependencies) have been processed.

The robot decides to execute the request by generating task plans for itself and the human with the task planner of [37]. As introduced in Section 4.3, this planner solves a planning problem $P = (F, I, A, G)$. The world state I is provided by the Action Predictor, which maintains an updated world model by applying observed actions to this model. The predicted action is converted to a goal state G, which is determined from the action's effects (a_{eff}). When the robot receives a request it will call the planner twice, once from the perspective of itself and once from the perspective of the human (i.e., as if the robot is at the human's location in the world model I). The length of the plans, i.e., number of actions in the plans, produced by the task planner are provided to Equation 4.1.

The robot decides to execute the request by comparing the length of its own plan with the human's plan, see Equation 4.1. In this equation l_r is the length of the robot's plan, l_h is the length of the human's plan, w is a weight factor ($0 \leq w \leq 1$) and σ is the sigmoid function that maps the plan length to a value between 0 and 1. The length of the robot's plan is converted to a negative value, so that the longer the robot's plan the lower the result (i.e., inverse sigmoid). If the result of Equation 4.1 is above threshold β the robot will execute its plan.

$$\gamma(w) = (\sigma(-l_r) \times w) + (\sigma(l_h) \times (1 - w)) \tag{4.1}$$

This decision rule is designed to help prevent the robot from obstructing or delaying the human. For example, if the human is about to take an item from a cupboard (i.e., l_h is small), the robot should not obstruct them by attempting to take an item from the same cupboard. We also assume the

human is friendly towards the robot, i.e., the robot will announce its plan and the human will not perform the actions within the robot's plan. Thus, the robot should not force the human to wait for it to execute a long plan.

Actions that are absent from the Action Graph, as they are not defined in the action prediction problem the graph was generated from, could still be executed by the robot. This is because the robot's own set of actions, defined in PDDL, could include additional (unobservable) actions.

4.8 Experiments: Action Prediction Accuracy

The experiments in this section aim to show how accurately Action Graphs predict a person's actions. This section describes the setup, then reports the action prediction results. The action prediction results include the accuracy for when a simulated human is pursuing one of the hypothesis goals, a subgoal and multiple (interleaving) subgoals.

4.8.1 Setup

Our Action Graph based method for action prediction was tested on the Kitchen domain, introduced in Section 4.3.2. When running the task planner [37] to create list of observations (i.e., actions) and the ground truth, a `move(?s ?g)` action definition was also included; in the original dataset spatial layout was not accounted for. This action enabled rational plans to be generated, i.e., the simulated human takes all items from the closest location before moving to the next location, instead of re-visiting the same location multiple times. The layout of the modelled environment is depicted in Figure 4.6. Unless otherwise stated, for each experiment the human starts from the kitchen entrance.

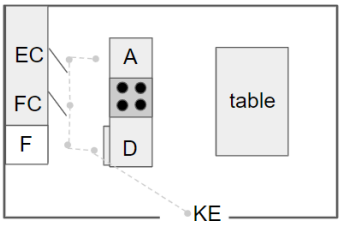


Figure 4.6: The layout of the environment used for the simulated experiments. EC is equipment cupboard, FC food cupboard, F fridge, D drawer, A appliance (e.g., toaster and kettle) and KE is kitchen entrance. There are waypoints in front of containers and at the entrance to the kitchen, the dashed lines indicate how the locations are linked for the planning of move actions. This environment is based on our HomeLab[1].

[1]https://www.imec-int.com/en/homelab

Statements, e.g., `(not(open fridge))`, were appended to the planner's goal to force all containers to be closed before the simulation finished, without these statements the task plans would not have contained `close(?container)` actions. All actions in this domain, except move actions, are assumed to be observable by mapping sensor readings to observations or by applying activity recognition algorithms such as [2].

The Action Graph creation for the Kitchen problem, including transforming the PDDL to DTGs, only takes an average of 0.13 seconds on a virtual machine with 8 GB RAM and 2 CPUs (2.90 GHz). Having said that, Action Graphs can be created offline; the algorithms for traversing the graph to update the node values and find the predicted actions must be performed online. Our DTGs to Action Graph transformation method has a time complexity of $O(n^2)$ as for each action (not applicable to the initial state) all actions are iterated over, to find the actions' dependencies. Algorithms 4.1, 4.2 and 4.3 have a linear (worst-case) time complexity, i.e., $O(n)$ with respect to the number of actions, as these algorithms traverse the graph (which has a tree-like structure) without traversing the same edge twice.

The results, in this section, report the number of correctly predicted (i.e., true positives) and incorrectly predicted (i.e., false positives) actions after observing the first 10, 30, 50, and 70 % of the actions in the plan produced by the task planner. Already observed actions are not included in the comparison, and move actions are not accounted for since they are assumed unobservable. Experiments were repeated for various values of θ, which is the minimum value of an action node to be included in the list of predicted actions.

4.8.2 Hypothesis Goal Action Prediction

In the Kitchen dataset, there are three hypothesis goals: $\mathcal{G}$ ={ `(made_breakfast)`, `(lunch_packed)`, `(made_dinner)`}. There are multiple options a human could choose to achieve each of these goals:

- `(made_breakfast)` by making coffee or tea;
- `(made_dinner)` by either making a salad or a cheese sandwich, or both;
- `(lunch_packed)` by either making a peanut butter or cheese sandwich.

To produce the plans of these 7 options, the goal provided to the task planner was set to an `and` statement, e.g., `(and ( (made_coffee) (made_breakfast)))`. The planner [37] is not guaranteed to find the optimal plan; thus, we also inserted `not` statements into the goal to prevent the human taking additional items, e.g., to prevent the human taking a tea bag when

they are making coffee. The results for action prediction when the simulated agent is performing each one of these options are presented in Table 4.1.

Table 4.1: Number of true positives (TP) and false positives (FP) for different goals (and plans) when the first 10, 30, 50 and 70 % of actions have been observed. $|a|$ is the total number of observable actions in the plan, and $|O|$ the number of processed observations.

			0.65		0.75		0.85		0.95	
Goal	$\|a\|$	$\|O\|$	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
`(made_breakfast)` including: `(made_coffee)`	28	3	0	0	0	0	0	0	0	0
		8	7	1	4	0	4	0	0	0
		14	11	3	9	3	9	2	4	0
		20	6	6	6	5	4	4	0	2
`(made_breakfast)` including: `(made_tea)`	27	3	0	0	0	0	0	0	0	0
		8	7	1	4	0	4	0	0	0
		14	10	7	5	4	5	3	1	2
		19	6	7	6	6	4	4	1	2
`(lunch_packed)` including: `(made_peanut_butter_sandwich)`	15	2	0	0	0	0	0	0	0	0
		5	4	3	4	0	0	0	0	0
		8	2	5	2	1	2	1	2	0
		11	2	10	0	3	0	1	0	1
`(lunch_packed)` including: `(made_cheese_sandwich)`	12	1	0	0	0	0	0	0	0	0
		4	3	0	0	0	0	0	0	0
		6	2	3	2	0	2	0	1	0
		8	2	7	0	5	0	3	0	0
`(made_dinner)` including: `(made_cheese_sandwich)` and `(made_salad)`	14	1	0	0	0	0	0	0	0	0
		4	3	0	0	0	0	0	0	0
		7	3	6	1	3	1	0	1	0
		10	2	6	2	4	0	4	0	0
`(made_dinner)` including: `(made_cheese_sandwich)`	11	1	0	0	0	0	0	0	0	0
		3	0	0	0	0	0	0	0	0
		6	2	3	2	0	2	0	0	0
		8	1	8	0	5	0	3	0	0
`(made_dinner)` including: `(made_salad)`	6	1	0	0	0	0	0	0	0	0
		2	2	0	0	0	0	0	0	0
		3	1	0	1	0	1	0	0	0
		4	1	0	0	0	0	0	0	0

Both lower values of θ and higher numbers of observations ($|O|$) resulted in more predictions being made. At $\theta = 0.65$ more true positives were returned in comparison to higher values of θ, but also a higher number of false predictions were made. On average the number of false positives reduced more than the number of true positives as θ is increased. This shows that at least some of the true positive actions have a higher value than those that were falsely identified.

We considered increasing the value of θ as more observations are observed but this would have little effect on which actions the robot executes. The list of predicted actions is sorted (highest value first) prior to checking if the robot can execute each action in turn; therefore, the robot is likely to execute actions correctly even when false positives are included within these

predictions. Moreover, in all presented experiments no predictions were made when 10 % of observations are provided. When such a small number of observations are provided it is unclear what the human's intentions are, thus a robot should not start acting.

Even with a high threshold value $\theta = 0.85$, the number of false positives in relation to the plan length $|a|$ remains high for three options: 1) `(lunch_packed)` including `(made_cheese_sandwich)`, 2) `(made_dinner)` including `(made_cheese_sandwich)` and 3) `(made_dinner)` including `(made_cheese_sandwich)` and `(made_salad)`. The main reason is that the plans to make a peanut butter sandwich and a cheese sandwich are very similar; therefore, the predicted actions are the union of actions required to make both types of sandwich.

4.8.3 Single Subgoal Action Prediction

To demonstrate how well our approach predicts actions of subgoals, plans (ground truth) for 6 subgoals were created using the method described in Section 4.8.1, and the accuracy of the predictions was recorded after 10, 30, 50 and 70 % of actions had been observed. No modifications were made to the list of hypothesis goals, which is processed to transform PDDL into DTGs.

Table 4.2: Action prediction for the subgoals, found in the Kitchen domain.

			0.75		0.85		0.95	
Goal	$\|a\|$	$\|O\|$	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
`(made_coffee)`	15	2	0	0	0	0	0	0
		5	7	0	0	0	0	0
		8	5	2	5	2	2	0
		11	3	3	3	2	3	0
`(made_peanut_butter_sandwich)`	13	1	0	0	0	0	0	0
		4	0	0	0	0	0	0
		7	3	1	3	0	0	0
		9	2	1	2	1	2	0
`(made_buttered_toast)`	12	1	0	0	0	0	0	0
		4	0	0	0	0	0	0
		6	4	0	4	0	0	0
		8	2	4	2	3	0	0
`(made_tea)`	11	1	0	0	0	0	0	0
		3	0	0	0	0	0	0
		6	3	0	3	0	0	0
		8	2	0	2	0	0	0
`(made_cheese_sandwich)`	10	1	0	0	0	0	0	0
		3	0	0	0	0	0	0
		5	2	0	2	0	0	0
		7	1	3	1	0	1	0
`(made_salad)`	6	1	0	0	0	0	0	0
		2	0	0	0	0	0	0
		3	1	0	1	0	0	0
		4	0	0	0	0	0	0

The results, presented in Table 4.2, show a similar trend as the results in the previous section. When (non-distinctive) actions belonging to multiple goals are observed, often actions belong to both goals will appear in the predicted actions of a single goal. For example, the false positives returned for `(made_buttered_ toast)` are contained within the plan to `(made_peanut_butter_sandwich)`, both these goals have similar actions, i.e., require the human to retrieve bread, retrieve a knife and open the fridge.

In the case of `(made_coffee)`, the 2 false positives (for $\theta = 0.85$), and all actions required to make coffee, are contained within the plan to reach `(made_breakfast)`. The more observed a plan is the more likely the human is to be aiming to achieve its goal, thus the more likely they are to perform other actions in that plan. When θ is increased to 0.95 all the predicted actions are correct, as those actions further away (in the Action Graph structure) from the observed actions have a lower value.

4.8.4 Multiple Interleaving Subgoals

In this section, results are presented for when a simulated human is interleaving actions to complete 2 subgoals. Subgoals were chosen, rather than the original hypothesis goals, as there are a higher number of subgoals, allowing more combinations to be tested. Moreover, if multiple of hypothesis goals were pursued, nearly all actions would be required, making it trivial to correctly predict actions.

When a human will switch between plans is non-deterministic; therefore, the results for 2 different methods of selecting when to switch between the plans are provided. In the first approach, a single call to the planner was performed with the goal set using an `and` statement, e.g., `(and (made_coffee) (made_salad))`.

For the second approach, the plans for both subgoals were created independently, then interleaved based on the probability (p) of a human switching between the two plans. If this probability is set to 1, one action from each of the plans will be selected in turn (i.e., alternating); if one plan is longer, any remaining actions are appended to the end. Actions are not repeated: if an action has already been executed because it was part of the other plan, the subsequent action is selected. The results show the average of 5 runs for when the probability of switching plans was set to 1, 0.75, 0.5, 0.25 and 0. As before, the predictions for when 10, 30, 50 and 70% of the actions have been observed are compared to the real (interleaving) plan. The average results are shown in Table 4.3 and results for each individual pair of subgoals are provided in Appendix 4.C.

There is a large amount of variation in the results for the different ways

Table 4.3: Action prediction for combinations of 2 plans that achieve subgoals, found in the Kitchen domain. θ is set to 0.85.

			Planner		$p = 1$		$p = 0.75$		$p = 0.5$		$p = 0.25$		$p = 0$	
Goal	$\lvert a \rvert$	$\lvert O \rvert$	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
Average	17.93	1.93	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		5.47	0.80	0.00	0.67	0.07	0.57	0.07	1.10	0.17	1.32	0.25	2.07	0.40
		9.33	2.27	1.07	4.13	0.73	3.49	0.84	3.38	0.97	2.28	1.04	1.00	1.40
		12.60	2.40	2.60	2.73	2.00	2.69	2.08	2.57	2.09	2.30	2.32	1.07	1.40

of interleaving two goals. In general, when 50 % and 70 % of actions had been observed, the number of true positive predictions decreases when the probability of switching plans (p) is reduced. With lower switching probabilities, most of the observations are from only one of the goals, hence the actions from the other goal cannot be predicted. For 30 % of observations the number of true positives often decreases when p is increased, as fewer actions from a single goal are observed, making it more challenging to make any predictions.

4.9 Experiments: Robotic Assistance

In this section, how the robot decides which of the predicted actions it can execute on behalf of the human is evaluated. The effect changing the values of β, w and θ, and of receiving invalid observations, has on which actions the robot executes is discussed.

4.9.1 Setup

The simulated robot is capable of all `take(?item)` and `take(?item, ?container)` actions. How many actions were (correctly) taken over by the robot is discussed, as well as if the human was temporarily blocked from performing an action, e.g., because they had to wait for the robot to take an item.

The simulated human's plan may be affected by the robot. When the robot changes the state of the environment and a precondition of the human's next action is no longer met, a new plan is produced for the human. It can also happen that the next action the human wants to execute is in the robot's current plan. If so, this action is skipped and the human attempts to proceed with the next action in its plan, without replanning.

Simulations were ran for the three hypothesis goals: made-breakfast, made-dinner and packed-lunch. The resulting action sequence for making breakfast is shown in Figure 4.7. For making dinner and making lunch, the action sequences are provided in Appendix 4.D. To produce these results, θ was set to 0.85, β to 0.45 and w to 0.5.

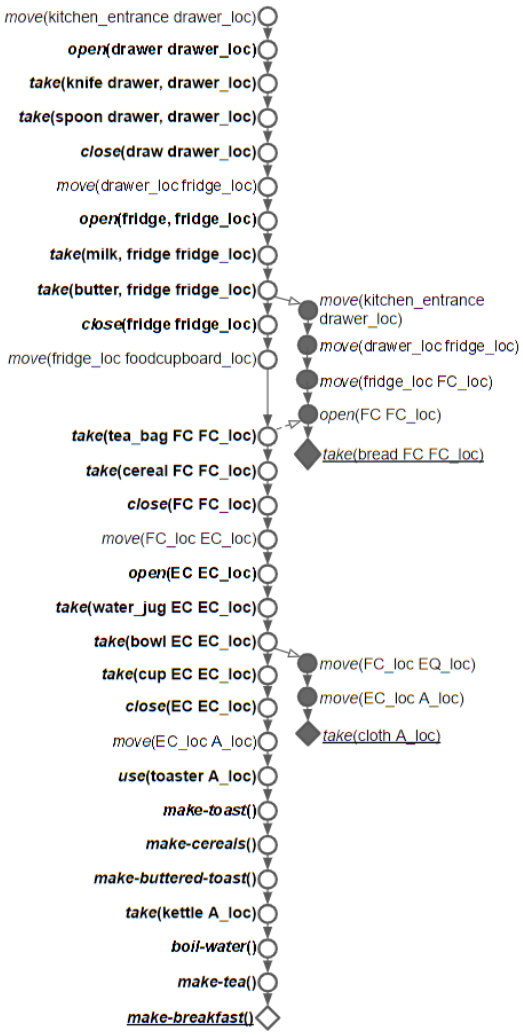


Figure 4.7: The human action sequence when their goal is to make breakfast (left) and the actions executed by the robot (right) when its initial location is `kitchen_entrance`. θ is set to 0.85, $\beta = 0.45$ and $w = 0.5$. Actions in bold are observable. Thick arrows pointing right indicate when the robot started execution, and dashed arrows when the human waited for the robot before executing an action.

4.9.2 Results and Discussion

To make breakfast, the human requires 32 actions without robotic assistance and 29 actions with robotic assistance. The robot correctly opened the food cupboard, took the bread and took a cloth. The robot did not perform any incorrect actions, i.e., no actions were executed that were not

in the human's original plan. During this experiment the human waited for the robot to open the food cupboard as it took a while for the robot, i.e., 3 move actions, to reach its location. An impatient human could have opened the cupboard themselves, enabling the robot to then just take the bread, or continued with actions later in their plan. The human's wait times were more prominent when the robot assisted the human to make dinner and pack lunch. Because the plans to reach these two goals are short, the human could not perform any further actions.

4.9.2.1 Changing β

The threshold β controls if the robot should execute the request or not based on the length of its plan and the plan it generates for the human to perform that request, see Equation 4.1. When β is increased to 0.5, the robot does not execute any actions, unless its starting location is adjusted in order to shorten its plan. When the human was making breakfast and the robot started near the fridge, the robot correctly retrieved bread, but then incorrectly retrieved sugar. When making breakfast the human has the option of making a coffee with sugar, and as the actions to make tea and coffee are similar, the actions in the plan to make coffee also had high values. For making lunch and dinner, when the robot is placed near the cupboard, the robot correctly opened the equipment cupboard and took a plate.

When β was decreased to 0.4 and the human was making breakfast (with the original initial locations), the robot retrieved the bread and a bowl and incorrectly retrieved sugar. Prior to the robot starting to get a bowl, the human had opened the equipment cupboard in which the bowl is stored, and both the human and robot took items from the equipment cupboard simultaneously. In a real-world setting, such collisions must be prevented, e.g., by setting β to a higher value. For making dinner and making lunch, lowering the value of β had no effect on which actions the robot executed.

4.9.2.2 Changing w

By changing w the robot's decision can be either weighted towards the robot's or human's plans' length. When weighted towards the robot's plan (i.e., $w = 0.6$), the robot did not execute any actions for any of the goals due to how long its initial plan was. Lowering w had the same effect as decreasing β.

4.9.2.3 Changing θ

Only action nodes with a value above the threshold θ are returned by the action predictor and are thus possible candidates for being executed by the robot. For making breakfast, when θ was decreased to 0.7 the robot

correctly executed the following actions: `open(foodcupboard)`, `take(bread foodcupboard)`, `open(equipmentcupboard)`, `take(bowl equipmentcupboard)` and `take(cloth)`. This resulted in the human only performing 27 actions, instead of 32 without robotic assistance. When θ was reduced further, the robot executed incorrect actions, i.e., false positive predictions, as it started acting before enough observations to determine the human's true intentions had been received. At $\theta = 0.6$, the robot executed 2 incorrect actions (i.e., retrieved peanut butter and sugar), and at $\theta \leq 0.4$ three incorrect actions were executed. Increasing θ reduces the number of actions executed by the robot and at $\theta = 0.9$ the robot did not execute any actions.

At $\theta \geq 0.25$ and $\theta \leq 0.85$, and when the human's goal was to make dinner or pack lunch, the robot correctly opened the equipment cupboard and took the plate for the human. When θ was lowered to 0.2, the robot also correctly opened the food cupboard and took the bread. Lowering θ further resulted in the robot performing incorrect actions as the robot acted too soon.

4.9.2.4 Noisy Observations

As sensors may provide erroneous readings and humans can make mistakes (e.g., take the wrong item or open the wrong cupboard) experiments were performed with the simulated human deviating from the plan produced by the task planner. Before the human performs an action in its plan, with a predefined probability a random (invalid) action was performed. This invalid action could be any action which is not in the person's plan and that has not already been observed; thus, if all invalid actions have already been observed, no further invalid actions can be observed.

Multiple experiments were performed for a range of probabilities; during these experiments the robot did not perform any incorrect actions. For `(packed_lunch)` and `(made_dinner)` the robot correctly opened the equipment cupboard and took a plate, and for `(made_breakfast)` opened the food cupboard, took bread and sometimes either opened the equipment cupboard and took a bowl or took a cloth. The `take(bread)` action belongs to all three goals, so no matter which action is observed its value is increased. Taking a plate is required for making salad, making a cheese sandwich and making a peanut butter sandwich; taking a bowl is required for making salad and cereal, and taking a cloth required for both making tea and making coffee. Unintentionally, the robot has performed actions which are non-distinctive, i.e., actions belonging to multiple goals' plans, thus small amounts of noisy have very little effect on the results. In future work, we will investigate explicitly extracting non-distinctive actions for the robot to execute.

4.10 Conclusion and Future Work

When a robot is integrated into a smart environment, it can execute actions that help a human to achieve their goals sooner by predicting the human's next actions. Our approach applies techniques from classical planning to transform a PDDL-defined problem into an Action Graph. Action Graph node values are updated based on the observations received. Experiments show Action Graphs adequately predict actions, even when the plans of multiple interleaving subgoals are observed. Furthermore, a simulated proof of concept demonstrated that by predicting a person's actions a robot is able to successfully assist a person by executing actions on their behalf.

We see several directions for future work. First, our approach favours the actions with shorter dependency lists due to the node value update algorithm calculating the mean value, e.g., `make-tea()` has fewer dependencies than `make-coffee()`. Thus, when an action common to both their dependency lists is observed, the value of `make-tea()` is increased more than the value of `make-coffee()`. Applying biases to nodes with more dependencies and weighting actions that are more unique to a plan [4] will be investigated to potentially improve the accuracy of our method. Nevertheless, for predicting the actions to reach subgoals (as well as goals), favouring actions with fewer dependencies is the desired behaviour for our system.

Second, how β and w are affected by differing action durations will be investigated. Moreover, β could be automatically adjusted based on how a human reacts to which actions the robot executes. For instance, if the robot continuously obstructs the human, β should be increased. If both the human's and robot's plans are short, increasing w will increase the chance of the robot performing an action; for long plans increasing w will have the opposite affect. Therefore, the value of w could be learnt for each domain, by measuring the length of the plans within the domain as well as if the robot acted without obstructing the human

Performing a statistical analysis of the effect changing the Action Graph's structure has on the optimal choice for θ will be investigated. These structural changes include changing the number of dependencies each action has, the types of nodes used to connect actions to their dependencies and how unique the dependencies are. The optimal value of θ depends on which of these dependencies have been observed, i.e., if the observed dependency is unique to a correct action, θ can be set to a much lower value than if the observed dependency also belongs to a incorrect action. Moreover, such an analysis must consider the trade-off between false positive and true positive predictions.

Fourth, we intend to prevent the robot obstructing the human when it is appointed an independent goal [32, 38]. To do this we will consider reasoning about the constraints between the person's and robot's plans; for example, if the person is handling an item, the robot cannot handle it at the same time. Furthermore, the person's plan may unintentionally contain actions that assist the robot, e.g., opening a door; therefore, the robot could conserve time and energy by allowing the human to perform certain actions first.

Last, our experiments assumed a rational human (i.e., followed plans produced by a task planner), and all human and robot actions took the same constant time. In future work, real-world tests will be conducted in our HomeLab[1], where we will deploy sensors, e.g., cupboard door sensors, cameras to detect items being take from in cupboards, smart electric sockets, audio sensors for recognising activities [2] etc.

Appendix 4.A Kitchen Domain

The AND-OR trees in Figures 4.8, 4.9 and 4.10 represent the plans found within in the Kitchen domain. To make the diagrams readable the actions to reach each hypothesis goal have been put in separate figures and `close(?container)` actions have not been included. Moreover, in the real Action Graph no action is repeated, nodes have multiple parents and only a single graph is produced. These figures provide insight on the complexity of the domain, for full details the original Kitchen domain produced by Ramırez et. al [3] can be download from https://sites.google.com/site/prasplanning/file-cabinet.

Appendix 4.B Reverse Actions

As explained in Section 4.5.2, when an action is observed, its reverse actions are reset along with any action whose value was increase when the reverse action was observed. The algorithm that performs this reset procedure is provided in Algorithm 4.3.

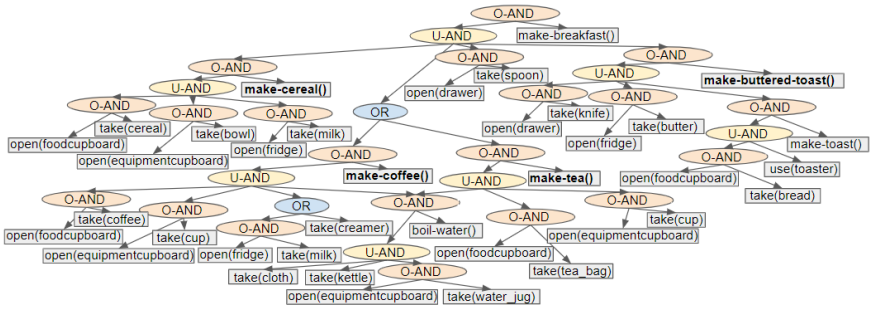


Figure 4.8: The plan to make breakfast including making tea, represented as an AND-OR tree. The final actions in the subgoal's plans, used for our experiments, has been put in bold. Some action names have been shortened, e.g., `take(bread foodcupboard)` to `take(bread)`

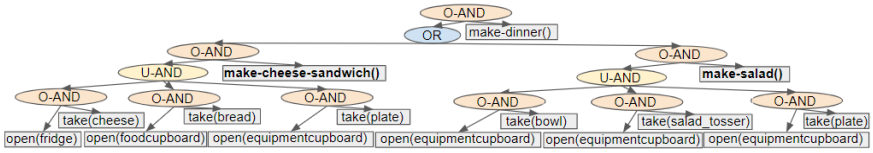


Figure 4.9: The two possible plans for making dinner represented as an AND-OR tree.

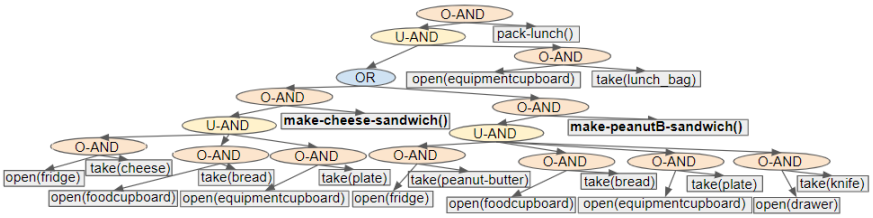


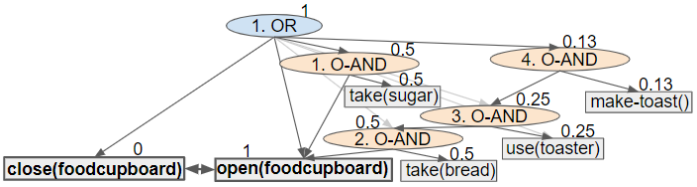
Figure 4.10: The two possible plans for making lunch represented as an AND-OR tree.

An example of how the process works is depicted in Figure 4.11. After observing the action `open(food-cupboard)`, the value of its action node is set to 1 and then the reset algorithm is called on its reverse action node. As `close(foodcupboard)` has not yet been observed, no reset is required. Applying the node value update algorithms, described in Section 4.5.1, results in the nodes values that are shown in Figure 4.11a. The updated node values after the next observation, i.e., `take(sugar)`, was received are shown in Figure 4.11b.

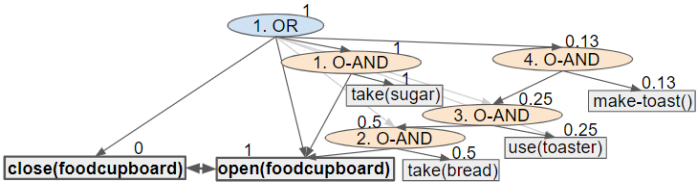
Algorithm 4.3 Reset node value

```
 1: function RESET_NODE_VALUE(node)
 2:     if node is action node and node.value ≠ 1 then return
 3:     else if node is action node then node.value = 0 end if
 4:     for each parent in node.parents do
 5:         if (parent is AND node) then
 6:             RESET_CHILDREN(parent)
 7:             RESET_NODE_VALUE(parent)
 8:         else parent is OR node
 9:             parent.updateValue()                    ▷ max value of children
10:         end if
11:     end for
12: end function
13: function RESET_CHILDREN(node)
14:     for each child in node.children do
15:         if child.value ≠ 1 and child has not been reset then
16:             if child is action node then
17:                 child.value = 0
18:             else if child is OR node then
19:                 child.updateValue()                 ▷ max value of children
20:             else
21:                 child.RESET_CHILDREN(child)
22:             end if
23:         end if
24:     end for
25:     node.updateValue()                          ▷ mean of children update rules
26: end function
```

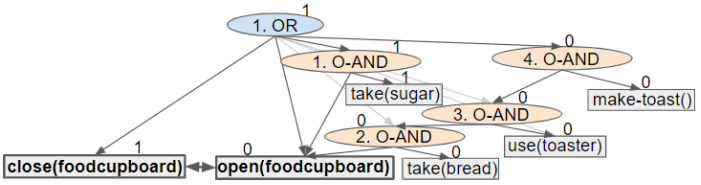
The next observation to be received, i.e., `close(foodcupboard)`, causes the `open(foodcupboard)` action node to be reset (line 3). The algorithm iterates over each of its parents, i.e., `OR` node 1, `ORDERED-AND` node 1 and `ORDERED-AND` node 2 (line 4-11). As another child of the `OR` node has a value of 1, its value is not changed. `ORDERED-AND` node 1 also remains unaltered as its subsequent child, `take(sugar)`, has already been observed. The algorithm recurs over the subsequent children of `ORDERED-AND` node 2 (lines 6 and 13-26), resulting in `use(bread)` being reset (line 17). The `ORDERED-AND` itself is then updated to the mean of its children (line 25) and its parents are iterated over (line 7 and 1-12). The process also occurs for `ORDERED-AND` nodes 3 and 4, resulting in both the `use(toaster)` and `make-toast()` action nodes being reset. After resetting the reverse action, the upwards and downwards procedures are performed. In our simple example the observed action's node (`close(foodcupboard)`) only has a single `OR` parent, i.e., the root node, whose value is already 1; therefore, no values require updating. The final node values are shown in Figure 4.11c.



(a) Node values after `open(foodcupboard)` was observed.



(b) Node values after `take(sugar)` has been observed.



(c) Node values after observing `close(foodcupboard)`.

Figure 4.11: Action graphs to show the effect of reverse actions. Double ended arrows indicate two actions are the reverse of each other.

Appendix 4.C Experiments: Two Subgoals

Our action prediction system was tested on problems in which a simulated human aimed to achieve two subgoals. The average result is provided in the main body of this manuscript, see Section 4.8. Table 4.4 shows the result for each pair of subgoals.

Appendix 4.D Experiments: Robot Assistance

The robot assistance experiment is discussed in Section 4.9. In this experiment the simulated human performed the actions to reach a goal (e.g., made dinner) and the robot assisted them by executing several of the predicted actions. For when the human's goal is to make dinner and to make lunch, the action sequences (of the human and the robot) are shown Figures 4.12 and 4.13, respectively.

Table 4.4: Action prediction for combinations of 2 plans that achieve subgoals found in the Kitchen domain. θ is set to 0.85.

			Planner		$p = 1$		$p = 0.75$		$p = 0.5$		$p = 0.25$		$p = 0$	
Goal	$\|a\|$	$\|O\|$	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
(made_coffee) and (made_buttered_toast)	23	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		7	4	0	3	0	1.4	0.0	3.6	0.4	2.6	0.8	4	2
		12	1	2	8	2	7.6	1.8	5.0	2.0	5.0	1.2	0	3
		16	1	5	4	5	4.2	5.0	4.8	4.6	4.2	4.8	0	3
(made_coffee) and (made_peanut_butter _sandwich)	22	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		7	0	0	3	0	1.8	0.0	4.2	1.2	3.0	1.2	4	2
		11	7	1	7	0	6.2	1.0	6.4	1.6	2.4	2.8	0	3
		15	5	5	4	4	4.0	4.0	3.8	2.8	2.4	3.2	0	3
(made_tea) and (made_buttered_toast)	21	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		6	4	0	0	0	0.0	0.0	1.2	0.0	0.6	0.0	3	0
		11	2	2	6	1	4.4	0.8	2.6	1.4	1.4	1.4	0	2
		15	2	2	4	3	3.8	2.4	2.8	2.6	3.4	3.4	3	4
(made_tea) and (made_peanut_butter _sandwich)	20	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		6	0	0	0	0	0.0	0.0	1.8	0.0	1.8	0.0	2	0
		10	2	1	2	0	3.8	0.0	2.4	0.0	0.6	1.0	0	2
		14	0	2	3	1	2.6	1.4	2.6	1.4	1.6	3.2	2	6
(made_coffee) and (made_cheese_sandwich)	19	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		6	0	0	0	0	0.8	0.0	1.2	0.0	3.4	1.2	5	2
		10	6	2	6	2	5.8	2.8	6.0	1.2	4.6	2.0	1	3
		13	4	7	3	6	3.6	6.6	3.4	6.0	3.0	4.8	0	3
(made_coffee) and (made_salad)	19	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		6	0	0	0	1	0.8	1.0	0.6	1.0	3.2	0.6	6	0
		10	0	0	5	1	2.0	1.0	5.0	1.0	3.2	1.0	4	1
		13	4	3	4	1	4.0	1.0	3.6	1.0	3.4	1.2	1	2
(made_salad) and (made_buttered_toast)	18	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	0	0	0	0	0.4	0.0	0.2	1.0	0.2	0.0	0	0
		9	0	0	2	2	2.6	1.4	2.8	4.0	1.6	1.0	0	0
		13	3	4	3	4	2.6	4.0	2.4	0.0	2.8	4.0	3	4
(made_tea) and (made_cheese_sandwich)	17	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	0	0	0	0	0.0	0.0	0.8	0.0	1.2	0.0	3	0
		9	0	3	1	0	0.8	1.2	1.2	1.8	1.4	1.2	0	2
		12	3	4	3	3	2.8	2.4	2.8	3.4	2.6	2.8	0	2
(made_coffee) and (made_tea)	17	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		9	4	2	5	0	4.4	0.0	4.4	0.8	3.0	1.2	3	2
		12	2	2	3	0	2.6	0.0	2.8	0.8	2.4	1.2	2	2
(made_cheese_sandwich) and (made_buttered _toast)	17	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	0	0	0	0	0.4	0.0	0.4	0.0	1.6	0.0	2	0
		9	4	1	4	1	3.4	1.0	4.0	1.0	1.8	1.2	2	2
		12	2	1	2	1	2.0	1.0	2.0	1.0	1.2	1.4	0	2
(made_peanut_butter _sandwich) and (made_buttered_toast)	17	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	4	0	4	0	1.6	0.0	1.6	0.0	0.8	0.0	0	0
		9	4	1	4	1	4.0	1.0	4.0	0.8	3.4	0.8	1	1
		12	2	1	2	1	2.0	1.0	2.0	1.0	1.6	1.2	0	2
(make_peanut_butter _sandwich) and (made_salad)	16	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	0	0	0	0	0.2	0.0	0.0	0.0	0.2	0.0	0	0
		8	1	1	3	1	0.6	0.2	1.0	0.0	1.2	0.0	0	0
		11	2	3	2	1	2.0	0.4	1.6	0.8	1.8	0.8	2	0
(made_tea) and (made_salad)	15	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		8	0	0	4	0	2.0	0.0	1.4	0.0	1.6	0.0	1	0
		11	3	0	2	0	2.2	0.4	2.0	0.4	1.6	0.8	0	2
(made_cheese_sandwich) and (made_peanut_butter _sandwich)	15	2	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		5	0	0	0	0	0.8	0.0	0.8	0.0	1.2	0.0	2	0
		8	3	0	3	0	3.0	0.0	2.8	0.0	2.6	0.0	3	0
		11	1	0	1	0	1.0	0.0	1.0	0.0	1.0	0.6	1	1
(made_cheese_sandwich) and (made_salad)	13	1	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
		4	0	0	0	0	0.4	0.0	0.2	0.0	0.0	0.0	0	0
		7	0	0	2	0	1.8	0.4	1.8	0.4	0.4	0.8	0	0
		9	2	0	1	0	1.0	1.6	1.0	2.4	1.6	2.0	2	0

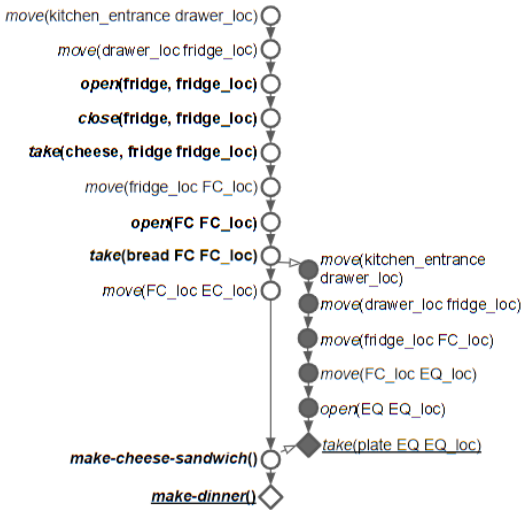


Figure 4.12: The order the human performs actions when their goal is to make dinner (left) and the actions executed by the robot (right) when its initial location is `kitchen_entrance`. With the following parameters: $\theta = 0.85$, $\beta = 0.45$, $w = 0.5$.

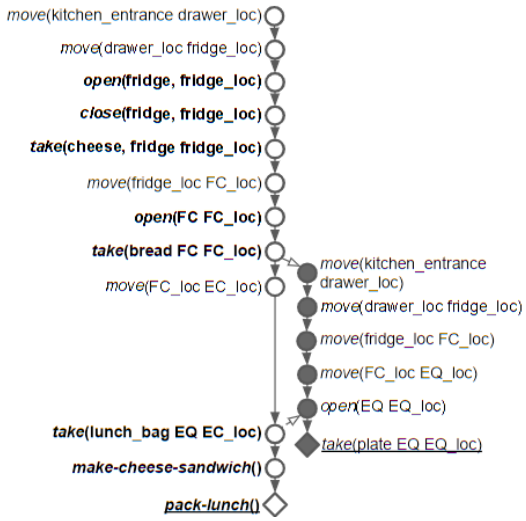


Figure 4.13: The order the human performs actions when their goal is to make a pack lunch (left) and the actions executed by the robot (right) when its initial location is `kitchen_entrance`. With the following parameters: $\theta = 0.85$, $\beta = 0.45$, $w = 0.5$.

References

[1] N. Roy, A. Misra, and D. Cook. *Ambient and Smartphone Sensor Assisted ADL Recognition in Multi-Inhabitant Smart Environments*. J. Ambient Intell. Humaniz. Comput., 7(1):1–19, 2016.

[2] S. Tremblay, D. Fortin-Simard, E. Blackburn-Verreault, S. Gaboury, B. Bouchard, and A. Bouzouane. *Exploiting Environmental Sounds for Activity Recognition in Smart Homes.* In AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments, 2015.

[3] M. Ramírez and H. Geffner. *Probabilistic Plan Recognition Using Off-the-Shelf Classical Planners*. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI'10, pages 1121–1126. AAAI Press, 2010.

[4] R. F. Pereira, N. Oren, and F. Meneguzzi. *Landmark-Based Heuristics for Goal Recognition*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 3622–3628. AAAI Press, 2017.

[5] M. Fagan and P. Cunningham. *Case-Based Plan Recognition in Computer Games*. In International Conference on Case-Based Reasoning Research and Development, pages 161–170, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[6] J. Hong. *Goal Recognition Through Goal Graph Analysis*. J. Artif. Intell. Res., 15:1–30, 2001.

[7] W. S. Lima, E. Souto, T. Rocha, R. W. Pazzi, and F. Pramudianto. *User Activity Recognition for Energy Saving in Smart Home Environment*. In IEEE Symposium on Computers and Communication (ISCC), pages 751–757. IEEE, 2015.

[8] R. G. Freedman, Y. R. Fung, R. Ganchin, and S. Zilberstein. *Towards Quicker Probabilistic Recognition with Multiple Goal Heuristic Search*. In AAAI Workshops on Plan, Activity, and Intent Recognition (PAIR-18), 2018.

[9] J. Rafferty, C. D. Nugent, J. Liu, and L. Chen. *From Activity Recognition to Intention Recognition for Assisted Living Within Smart Homes*. IEEE T. Hum.-Mach. Syst., 47(3):368–379, 2017.

[10] K. Yordanova, S. Lüdtke, S. Whitehouse, F. Krüger, A. Paiement, M. Mirmehdi, I. Craddock, and T. Kirste. *Analysing Cooking Behaviour in Home Settings: Towards Health Monitoring*. Sensors, 19(3), 2019.

[11] G. Singla, D. J. Cook, and M. Schmitter-Edgecombe. *Recognizing Independent and Joint Activities Among Multiple Residents in Smart Environments*. J. Ambient Intell. Humaniz. Comput., 1(1):57–63, 2010.

[12] F. Bisson, H. Larochelle, and F. Kabanza. *Using a Recursive Neural Network to Learn an Agent's Decision Model for Plan Recognition*. In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, IJCAI'15, pages 918–924. AAAI Press, 2015.

[13] L. Amado, R. F. Pereira, J. Aires, M. Magnaguagno, R. Granada, and F. Meneguzzi. *Goal Recognition in Latent Space*. In Proceedings of the International Joint Conference on Neural Networks, IJCNN, pages 1–8. IEEE, 2018.

[14] P. C. Roy, S. Giroux, B. Bouchard, A. Bouzouane, C. Phua, A. Tolstikov, and J. Biswas. *A Possibilistic Approach for Activity Recognition in Smart Homes for Cognitive Assistance to Alzheimer's Patients*, pages 33–58. Atlantis Press, Paris, 2011.

[15] K. Yordanova, F. Krüger, and T. Kirste. *Context Aware Approach for Activity Recognition Based on Precondition-Effect Rules*. In IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pages 602–607. IEEE, 2012.

[16] S. Keren, R. Mirsky, and C. Geib. *Plan Activity and Intent Recognition Tutorial*, 2019. Available from: http://www.planrec.org/Tutorial/Resources_files/pair-tutorial.pdf.

[17] H. A. Kautz and J. F. Allen. *Generalized Plan Recognition*. In Proceedings of the Fifth AAAI National Conference on Artificial Intelligence, volume 86 of *AAAI'86*, pages 32–37. AAAI Press, 1986.

[18] C. W. Geib and R. P. Goldman. *A Probabilistic Plan Recognition Algorithm Based on Plan Tree Grammars*. Artif. Intell., 173(11):1101 – 1132, 2009.

[19] R. Mirsky, Y. K. Gal, and S. M. Shieber. *CRADLE: An Online Plan Recognition Algorithm for Exploratory Domains*. ACM Trans. Intell. Syst. Technol., 8(3), 2017.

[20] M. Ramírez and H. Geffner. *Plan Recognition as Planning*. In Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence, IJCAI'09, pages 1778–1783, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[21] S. Holtzen, Y. Zhao, T. Gao, J. B. Tenenbaum, and S. Zhu. *Inferring Human Intent From Video by Sampling Hierarchical Plans*. In IEEE/RSJ

International Conference on Intelligent Robots and Systems, IROS, pages 1489–1496. IEEE, 2016.

[22] S. S. Vattam, D. W. Aha, and M. Floyd. *Case-Based Plan Recognition Using Action Sequence Graphs*. In Case-Based Reasoning Research and Development, pages 495–510, Cham, 2014. Springer International Publishing.

[23] H. A. Kautz. *A Formal Theory of Plan Recognition*. PhD thesis, University of Rochester, Department of Computer Science, 1987.

[24] M. Helmert. *The Fast Downward Planning System*. J. Artif. Intell. Res., 26:191–246, 2006.

[25] J. Chen, Y. Chen, Y. Xu, R. Huang, and Z. Chen. *A Planning Approach to the Recognition of Multiple Goals*. Int. J. Intell. Syst., 28(3):203–216, 2013.

[26] Y. E-Martin, M. D. R-Moreno, and D. E. Smith. *A Fast Goal Recognition Technique Based on Interaction Estimates*. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI'15. AAAI Press, 2015.

[27] L. Johannsmeier and S. Haddadin. *A Hierarchical Human-Robot Interaction-Planning Framework for Task Allocation in Collaborative Industrial Assembly Processes*. IEEE Robotics and Automation Letters, 2(1):41–48, 2017.

[28] Y. Zhang, V. Narayanan, T. Chakraborti, and S. Kambhampati. *A Human Factors Analysis of Proactive Support in Human-Robot Teaming*. In IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, pages 3586–3593. IEEE, 2015.

[29] J. Baraglia, M. Cakmak, Y. Nagai, R. P. Rao, and M. Asada. *Efficient Human-Robot Collaboration: When Should a Robot Take Initiative?* Int. J. Robot. Res., 36(5-7):563–579, 2017.

[30] S. J. Levine and B. C. Williams. *Concurrent Plan Recognition and Execution for Human-Robot Teams*. In Proceedings of the Twenty-Fourth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'14, pages 490–498. AAAI Press, AAAI Press, 2014.

[31] S. J. Levine and B. C. Williams. *Watching and Acting Together: Concurrent Plan Recognition and Adaptation for Human-Robot Teams*. J. Artif. Intell. Res., 63:281–359, 2018.

[32] R. G. Freedman and S. Zilberstein. *Integration of Planning with Recognition for Responsive Interaction Using Classical Planners*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 4581–4588. AAAI Press, 2017.

[33] P. Jonsson and C. Bäckström. *State-Variable Planning Under Structural Restrictions: Algorithms and Complexity*. Artif. Intell., 100(1):125 – 176, 1998.

[34] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and practice*. San Francisco: Elsevier, 2004.

[35] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning: Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers, 1st edition, 2013.

[36] J. Wu, A. Osuntogun, T. Choudhury, M. Philipose, and J. M. Rehg. *A Scalable Approach to Activity Recognition Based on Object Use*. In Proceedings of the Eleventh IEEE International Conference on Computer Vision, pages 1–8. IEEE, 2007.

[37] C. Muise, S. Vernhes, and P. Florian. *PDDL Solver in the Cloud*, 2018. Accessed: 2018-06-06. Available from: https://bitbucket.org/planning-researchers/cloud-solver.

[38] M. Cirillo, L. Karlsson, and A. Saffiotti. *Human-Aware Task Planning: An Application to Mobile Robots*. ACM Trans. Intell. Syst. Technol., 1(2):15:1–15:26, 2010.

5

Robot Assistance in Dynamic Smart Environments

"Computer technology offers the possibility of incorporating intelligent behavior in all the nooks and crannies of our world. With it, we could build an enchanted land."
Allen Newell (1976)

Robot Assistance in Dynamic Smart Environments—A Hierarchical Continual Planning in the Now Framework

H. Harman, K. Chintamani & P. Simoens
Published in Sensors, 19(22):4856, 2019.

In the previous chapter, the robot's plan only contained actions the robot itself could execute. This greatly limits the range of tasks that can be completed. By coupling a robot to a smart environment, the robot can sense state beyond the perception range of its onboard sensors and gain greater actuation capabilities. Nevertheless, incorporating the states and actions of Internet of Things (IoT) devices into the robot's onboard planner increases the computational load, and thus can delay the execution of a task. Moreover, tasks may be frequently replanned due to the unanticipated actions of humans. Our framework aims to mitigate these inadequacies. In this chapter, we propose a continual planning framework which incorporates the sensing and actuation capabilities of IoT devices into a robot's state estimation, task planing and task execution. The robot's onboard task planner queries a cloud-based framework for actuators, capable of the actions the robot cannot execute. Once generated the plan is sent to the cloud backend, which will inform the robot if any IoT device reports a state change affecting its plan. Moreover, a Hierarchical Continual Planning in the Now approach was developed in which tasks are split-up into subtasks. To delay the planning of actions that will not be promptly executed, and thus to reduce the frequency of replanning, the first subtask is planned and executed before the subsequent subtask is. Only information relevant to the current (sub)task is provided to the task planner. We apply our framework to a smart home and office scenario in which the robot is tasked with carrying out a human's requests. A prototype implementation in a smart home and simulator-based evaluation results are presented to demonstrate the effectiveness of our framework.

5.1 Introduction

The deployment of mobile and dexterous robots is being envisioned by the research community in an expanding range of environments. For instance, assistance and companion robots for elderly at home, and service robots in office buildings. These robots work alongside humans, and therefore must be able to adapt their task execution to unexpected changes in the environment's state.

Instead of hard coding sequences of actions to accomplish a given task,

a more generic approach is symbolic task planning, an artificial intelligence technique that can be applied to real-world robotics [1]. A task is formulated as a desired goal state, and planners autonomously find the appropriate set of actions (i.e., a task plan) to move from the current state to the desired goal state. Symbolic task planners are ran by continual planners, for example, [2–5], which interleave sensing, planning and acting. The (re)planning phase is only performed when a state change that affects the plan is detected.

An increasing number of Internet-of-Things (IoT) devices with sensing and actuation capabilities are being installed, resulting in smart environments [6]. Embedding robots in smart environments, a concept referred to as the Internet-of-Robotic-Things [7], can extend the scope of tasks a robot's continual planner can handle in two ways. First, the pervasive sensors provide valuable information, which can be incorporated in the planner's estimate of the current world state. Second, actuation devices expand the set of available actions. For instance, imagine a robot that is able to drive around a house but cannot open doors, for example, because it lacks arms or is carrying an object. If a door on its planned path is closed, a robot that is not coupled to the smart environment will only discover the infeasibility of its current plan when the door is within the perception range of its own sensors. Whereas, an IoT door sensor could immediately notify the robot of the door being closed. If the door is also equipped with an remotely controllable door pump, the robot can request a smart home system to open the door. Otherwise, if there are no alternative routes available, a closed door would simply result in the robot failing to accomplish its task.

In this chapter, we introduce our Hierarchical Continual Planning in the Now framework, which aims to i) enable a robot's planner to incorporate the state information and exogenous (off-board) actuation capabilities provided by IoT sensors and actuators; and ii) reduce the frequency of and time spent (re)planning tasks in dynamically changing environments. In other words, as mentioned in the introductory chapter, we aim to answer the following research question : How can the state information and exogenous actuation capabilities of IoT devices be incorporated into a robot's continual planner without greatly increasing the computational cost and the frequency of (re-)planning? Representing all devices directly within a symbolic task planner's state, together with the sensor information and the actuation capabilities they provide, would cause unreasonable planning times. Therefore, in our framework, cloud-based state monitors inform the robot of IoT-sensed state changes that are relevant to its current task plan. Moreover, during the planning phase, remote devices are abstractly rep-

resented and calls to an external (ontology-based) module check if one or more devices are capable of executing an action.

Actions further along in the planning horizon are more likely to become unexecutable or unnecessary due to a unexpected change in the world's state. State knowledge and actions are therefore split-up into a hierarchical structure. A single branch is planned and executed before the subsequent branch. This reduces how far in advance a detailed plan is generated and enables a robot to replan subsections of its task plan.

The remainder of this chapter is structured as follows. A summary of related work is provided in Section 5.2. Background information on the concept of hierarchical planning in the now is provided in Section 5.3. Section 5.4 provides an overview of the framework architecture. Details on the different components are described in Section 5.5. Section 5.6 introduces how a engineer can design the hierarchy of state knowledge and actions. Finally, in Sections 5.7 and 5.8 we evaluate our work through real world tests and simulated experiments.

5.2 Related Work

This section discusses several strands of related work. First, approaches that perform planning within a smart environment are introduced. This is followed by a discussion on how robots can reason on the capabilities of themselves and IoT sensors and actuators. The third subsection outlines how domain specific reasoning has been integrated into symbolic task planners. Finally, methods that reduce the amount of planning performed upfront are described.

5.2.1 Planning in Smart Environments

As explained in the introduction, the sensing and actuation capabilities provided by a smart environment can extend both the robot's perception range and ability to manipulate the environment. Below, only works that incorporate IoT sensor and/or actuator devices into planners are discussed. For a broader survey on the integration of IoT and robotics, we refer to [8].

In [9], data from off-board RGB-D cameras allows a robot to create a plan that avoids congested areas. When a sensor nearby the robot detects congestion, the robots within the local area attempt to adapt their plan. If unsuccessful, a central scheduler recalculates the robots' plans by reasoning on an ontology containing all robots, along with the tasks they can fulfil.

When using symbolic task planning in multi-agent robotics, which device (or robot) will execute an action tends to be explicitly stated within the plan, for example, MA-PDDL [10] and MAPL [11]. However, the num-

ber of IoT devices installed in smart environments is rising, and adding all these devices explicitly to the planning problem increases the planning time. This causes unreasonable delays before task execution can begin. Further to this, devices may become unavailable, causing the plan to fail; and new devices may be discovered, which could allow a less costly plan to be produced.

In [12], plans containing abstract services are generated. These, abstract services, are mapped to actual devices during run-time. As the plan contains abstract services, replanning is not required when services appear and disappear. We aim to incorporate this concept into robot task planning by adding an abstract remote device to the planning problem. As we use a mobile robot, some knowledge about which device the robot should interact with is still required during part of the planning process. For example, if the robot requires an IoT coffee machine, it needs to know the location of the coffee machine.

The PEIS Ecology middleware [13–15] provides robots with the ability to discover what heterogeneous smart devices are available and subscribe to their capabilities for assistance with executing a task. When a device becomes unavailable, the ecology is automatically reconfigured to use an alternative device. Like PEIS, in our system a robot is responsible for planning its own task; however, rather than a robot directly communicating to devices, they communicate via a central server, which reasons about devices and their capabilities.

5.2.2 Capability Reasoning

Robots come with many different capabilities, for example, some may have the dexterity to open doors while others have no effectors for object manipulation [16]. A similar diversity in sensing and actuation capabilities is observed across smart environments. Some capabilities may be provided by more than one device. For instance, a door can be opened by another robot or by a nearby human. Moreover, the available functions of a device may evolve over time and, for example, become temporarily unavailable due to a malfunction. Therefore, during the planning process, the robot needs to know that there is at least one device (possibly itself) capable of performing an action for a given set of parameters (e.g., an open door action can be performed on door 1 but not on door 2); otherwise, the action should not be planned. We propose calling an external module, which queries the cloud for capable devices. The cloud reasons on an ontology about the capabilities of devices, and monitors their status.

Numerous robotic and IoT ontologies exist for reasoning about devices and

there capabilities, such as IoT-O [17, 18] and KnowRob's SRDL [19]. These describe "things" and their relationships in the Web Ontology Language (OWL). Rather than coming up with another new ontology, our framework makes use of the KnowRob ontologies [20], which are part of the RoboEarth project. KnowRob enables robots to be described in detail (e.g., their software and hardware components) and each robot can be linked to one or more capabilities. As these capabilities do not include any parameters, we expanded this ontology (see Section 5.5.2.2).

In the RoboEarth project, if a robot has the required capabilities, an action recipe is downloaded onto the robot and tailored to its specific hardware to create an executable plan. Furthermore, a task plan is produced based on the most likely locations of objects. If the most likely location is incorrect, the next most probable location is used. When the object is detected, information is sent back to the cloud to update the probabilities. We aim to allow robots to execute tasks even if network connectivity is lost; therefore, in our system a robot is responsible for performing its own planning. Although a robot will be limited to its onboard capabilities, some tasks can still be accomplished.

In [21], semantic maps are queried to enrich the planner's initial state with additional knowledge about the environment. Providing the planner with all knowledge derived from the semantic map would be computationally expensive. Therefore, they only include knowledge relevant to the current context, i.e., concepts stated within the goal. Moreover, "semantic-level planning" is performed to prune the initial state of unnecessary information. Similar to this approach, our framework only provides relevant information to the planner; however, our approach focuses on the splitting-up of actions and state into subtasks to reduce the amount to knowledge required by a task planner. Our approach also performs planning in the now, thus reducing the planning time required before the robot starts executing its task.

Köckemann et al. [22] integrated a constraint-based planner with an ontology to avail the capabilities of IoT devices. To accomplish this, they enabled ontology queries to be written within the planning language. Further, they also developed a method to generate the queries automatically from statements within the domain definition. Their examples and experiment focus on querying sensors. Similarly, we integrate a symbolic planner with an ontology but we focus on reasoning on actuation capabilities. We also develop a continual planning framework that enables the planner to be run onboard a robot, and the cloud to monitor IoT devices and reason on their capabilities.

5.2.3 Planners with External Reasoners

Often the (re)planning phase of a continual planner is performed by a classical symbolic (task) planner, for example, Fast Downward (FD) [23]. These planners take a symbolic representation of an agent's behaviour and the world state as input, and search for a set of actions that will transform the initial state into the goal state. A popular symbolic language is Planning Domain Definition Language (PDDL). A PDDL problem definition usually consists of a problem file and a domain file. The problem file contains an initial (current) state and a goal state, both expressed using predicates, fluents and objects. Actions, along with their conditions and effects, are defined in a domain file.

As PDDL is restrictive in what can be represented, several works, such as TFD/M [16, 24], enable external modules to be called during planning. These modules incorporate low-level domain specific reasoning into high-level symbolic planning to allow for better (i.e., more accurate and detailed) state estimation. For example, an external module could be a motion planner to discover if it is possible for a robot to reach a location [3] or a reasoner which determines if an object will fit into a container based on its shape and size [24]. Unfortunately, as external module calls increase the planning time, it can be difficult to balance determining the correct state with keeping computational costs down [4]. In our framework, which expands on the work by Dornhege and Hertle [4] and Speck et al. [3], an external module is called to reason on the capabilities of IoT actuators.

Checking the feasibility of plans with additional domain-specific algorithms has also been previously investigated for Hierarchical Task Network (HTN) planners. In HTNs, a domain file contains a set of compound and primitive (executable) tasks along with methods, which describe how to decompose each compound task into subtasks. Subtasks can consist of both primitive tasks and compound tasks (which will be further decomposed). HTN planners have been combined with algorithms for geometric task planning [25] and for resource scheduling [26]. When the lowest level of the HTN is reached, the respective algorithm is invoked and, if unsuccessful in finding a solution, backtracking is performed.

5.2.4 Interleaving Planning and Execution

Planning methods usually generate the full plan upfront before starting execution. Unexpected changes in the state of the environment can cause the plan to become infeasible; therefore, the effort to generate the full plan is wasted. As well as causing a cost to the robot itself, replanning can be costly to the system as a whole due to resources having been assigned to

the plan (e.g., the use of external devices). Furthermore, additional state may become apparent during the execution of the plan, which can enable a more cost efficient plan to be generated. In this case, replanning is required to take this additional state information into consideration. To resolve these issues, scholars [27–31] have suggested planning short term actions in more detail than those later in the planning horizon.

Sukkerd et al. [27] propose multi-resolution temporal planning using a Markov Decision Process (MDP). Actions up until a given time are planned in detail; and those later in the planning horizon are planned at a coarser level of detail. Replanning of the coarser sections is performed with up to date state observation to generate a detailed plan. This approach enables distant actions to be taken into consideration when generating the detailed, short-term plan.

In Belief-Desire-Intention (BDI) approaches [32–34], an agent selects its intentions (i.e., plan) from a set of desires based on its beliefs about the current world state. Traditionally, the desires are a predefined library of plans [35]; however, BDI has also been combined with planners. For instance, Mora et al. [33] first select an intention and then plan the necessary actions. Besides actions, the plan can also contain intentions, which require further refinement. An agent revises its beliefs and triggers replanning when necessary.

Instead of time determining which actions are planned in detail, one can use the structure of hierarchical plans. HTN's task decomposition can determine which high-level actions to plan in detail. Hierarchical Planning in the Now (HPN) recursively plans and executes actions [28]. An abstract plan is formulated, and the first composite action is planned and executed before decomposing the subsequent composite action. The authors use a hierarchical planning language in which the actions must be labelled with the level of abstraction, and the complete world state is always reasoned on. This has been expanded with belief states, namely, Belief Hierarchical Planning in the Now (BHPN), and planning using continuous state information (e.g., position) [29].

As our system builds on many of the concepts from HTN planners, it has many of the same benefits and disadvantages. Although HTN planners do not always guarantee an optimal plan [36], the use of compound tasks enables HTN planners to be faster [25] and simpler [37] than classical symbolic planning. HTNs are also claimed to represent tasks in a way more closely related to how humans think about problems [25, 38]. Unfortunately, there is no standard planning language for HTNs and recent works,

e.g., [39], attempt to transform a HTN planning problem into a classical (PDDL) planning problem to apply more advanced heuristics. Therefore, in our work, PDDL is split-up into a hierarchy.

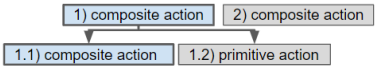
In order to maximise the developer's flexibility and to ensure compatibility with various types of smart environment, our planning framework differs from HPN [28] and BDI [33] systems in a number of ways. First, our approach splits-up both the state knowledge and actions into different layers; therefore, only knowledge relevant to the current layer and context is reasoned on. Second, any planner, including those using a different representation of the world, can be used at each layer. Third, by planning with different domain files at each layer, domains can be re-used in different scenarios and this prevents a single extremely large domain file from being developed. Finally, we also focus on how the planner can use IoT devices found within a smart environment and replan when a state change that causes the robot's plan to be invalidated is detected.

5.3 Background: Hierarchical Planning in the Now

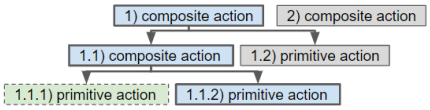
Our framework enables hierarchical task plans to be monitored, so that replanning is triggered on a sub-branch of the plan when IoT sensors detect unanticipated state changes. This background section describes the general concept of Hierarchical Planning in the Now (HPN), as introduced by Kaelbling and Lozano-Pérez [28]. In HPN, the initial task plan, generated by the planner, contains a sequence of actions that can be primitive or composite. Each of these actions will be executed in turn. The execution of a composite action will generate a sub-plan, which again will be comprised of primitive and/or composite actions.

This concept is depicted in Figure 5.1. In this figure, the initial plan consists of a sequence of two composite actions (Figure 5.1a). The first composite action produces a sub-plan composed of a composite and a primitive action (Figure 5.1b). As the sub-plan's first action is also a composite action, it is decomposed and a plan containing only primitive actions is produced (i.e., Figure 5.1c). Once these primitive actions have been executed, the effects of composite action 1.1 are reached, enabling the next action within the second level to be executed (Figure 5.1d). This continues until all actions have been completed (Figure 5.1e–f).

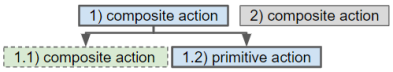
(a) Initial plan.



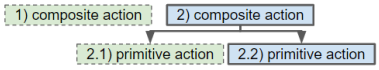
(b) As the first action is a composite action, when it is executed, a subtask plan is created.



(c) The first action of the sub-plan is also a composite action, so executing it generates another sub-plan. As this only contains primitive actions, no further task decomposition is required.



(d) The first action in the sub-plan has finished, allowing the second action to be executed.



(e) Once the first composite action in the initial plan has finished, the subsequent action is executed. This causes another plan to be generated.

(f) The execution of the plan has completed.

Figure 5.1: Grey boxes (with thin solid borders) indicate actions which have not yet been executed; blue (thick solid borders) shows the actions currently being executed and green (dashed borders) indicates executed actions. These figures describe the general concept of hierarchical planning in the now, introduced by [28].

5.4 Framework Overview

Our framework consists of components that are deployed onboard a robot and in the cloud. An conceptual overview of the framework is visualised in Figure 5.2. This sections explains which components are deployed onboard the robot and which are run in the cloud. Further details, about these components, are provided in the subsequent section. Communication with the IoT devices of a smart environment is abstracted by an IoT middleware, namely, DYnamic, Adaptive MAnagement of Networks and Devices (DYAMAND) [40]. DYAMAND enables all heterogeneous IoT sensors to communicate using a standard format and IoT actuators can be commanded by sending JSON strings (via a central-server) over HTTP.

The main components onboard a robot are the Continual Planner and a knowledge base. The Continual Planner performs three processes: it composes the PDDL files based on the information stored in the knowledge base, it generates a task plan containing composite and primitive actions and it executes that plan. Running the planner on the robot allows it to continue operating when (wireless) network connectivity towards the cloud drops but requires limiting the amount of information transmitted from

the smart environment to the robot. Sending the raw data streams of all IoT sensors in the environment would incur prohibitively large network bandwidth and associated battery drain. Moreover, the robot would lack the required computational resources to process raw data, especially if rich data from sensors such as cameras or microphones [41, 42] is involved.

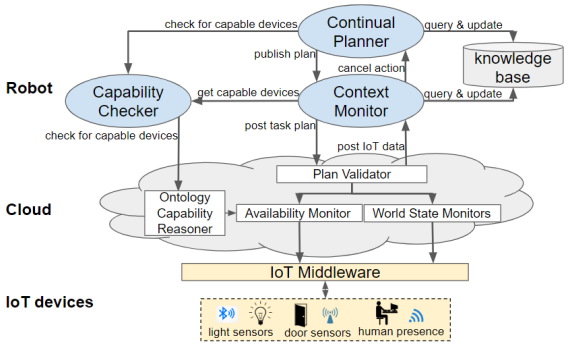


Figure 5.2: During planning, external module calls check if any device is capable of performing an action by calling a service running in the Capability Checker, which queries the cloud for capable devices and caches the results. The Context Monitor queries this when a plan has been generated, to know which device(s) should perform which action. The plan (containing the list of actual device) is sent to the cloud, which is monitoring the state of the environment and the devices. Any Internet of Things (IoT) data relevant to the robot's plan is sent back to the robot and inserted into the robot's knowledge base. If this information will cause the robot's plan to fail, its current action is cancelled forcing the state to be (re-)estimated and replanning to be performed.

During the task planning phase, the Continual Planner will check via the Capability Checker if a remote (IoT) device is capable of performing an action. The Capability Checker queries the cloud-based Ontology Capability Reasoner for capable devices and caches the result. All reasoning over IoT actuators and their capabilities is thus delegated to the back-end components of the framework.

A robot, via its onboard Context Monitor, communicates its current hierarchical task plan to the cloud back-end. In the cloud, World State Monitors process the data produced by IoT sensors and the Availability Monitor tracks the availability of IoT actuators. The cloud informs the robot of unanticipated state changes affecting its plan and, subsequently, the robot decides if replanning is necessary. Rather than recreating the whole hierarchical task plan, replanning is triggered at the level containing the action(s) that will fail. As only a small subsection of the plan is replanned, the time

spent planning is greatly reduced in comparison to planning everything upfront. If replanning fails, the composite action becomes invalid. This will trigger its parent's (i.e., a higher-level's) state estimation and replanning.

5.5 Incorporating IoT into Continual Planning

Our continual planning component, shown in Figure 5.3, is an expanded version of the framework by Dornhege & Hertle [4]. The original framework loops through the phases of populating a PDDL problem by calling State Estimators, running the TFD/M task planner and then calling the appropriate Action Executor plug-ins to translate actions into executable robot instructions. We have expanded this framework to incorporate IoT sensed information, utilise the capabilities of IoT actuators, produce a hierarchical task plan and monitor that task plan. This section first discusses how the PDDL files are formulated. Second, the capability checking process, which is performed during the search for a plan, is described. Subsequently, the context monitoring and replanning processes are introduced. In the final subsection, the action execution is detailed.

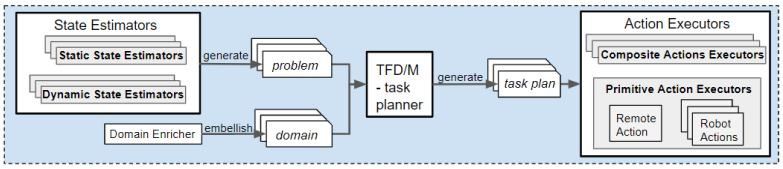


Figure 5.3: The continual planner calls the State Estimators (i.e., Static State Estimators and Dynamic State Estimators) and Domain Enricher to generate the PDDL problem and domain based on observations from the robot and smart environment. The TFD/M planner generates a task plan that contains primitive actions, to be executed by the robot and IoT actuators and composite actions. Composite action executors contain an instance of the the continual planner, which is configured with the relevant State Estimators, Action Executors and domain file.

5.5.1 Generating a Planning Problem

In the first phase a planning problem is generated by a set of State Estimators, which populate a PDDL problem file and a Domain Enricher, which populates a PDDL domain file. These files can then be provided to a task planner (in our implementation, this is TFD/M) to produce a task plan containing both primitive and composite actions. Each action will be executed by its corresponding Action Executor (see Section 5.5.4.2). Which State Estimators, Action Executors and domain file should be loaded, is stated within a configuration file. Each composite action, i.e., branch of the hierarchy, can be configured with its own domain file and instances of

State Estimators and Action Executors. State Estimators and the Domain Enricher are detailed in this section.

5.5.1.1 State Estimation

The PDDL problem file contains a goal state and the current state. The problem file is populated by calling a set of State Estimator plug-ins, each of which implements a `fillState` method. State Estimators query the most recent state either directly from the robotic middleware (e.g., to gain the robot's position) or from the knowledge database and transform this information into PDDL statements. The knowledge base, implemented in MongoDB, stores information on the current state and the goals. Updates to the knowledge database come from external IoT devices via the Context Monitor and from the result returned by Action Executors.

Table 5.1: Description of the different State Estimators.

State Estimator	Description
`object_state`	The object_state estimator is configured with a list of objects (given as arguments at initialisation or added later by calling a method) that the planner requires the state of. The objects' state is read from the robot's knowledge base and the relevant statements are added to the planner's current state. A statement is relevant if the predicates and object types, contained within the statement, have been defined in the domain file. When one of its arguments is "request", then all requests sent (by the cloud) to the robot are inserted into the PDDL problem file. This enables a human to request the robot's assistance at any point in time.
`locations`	This Static State Estimator inserts the waypoints, which are within the robot's knowledge base, into the PDDL problem. Waypoints are written in the format <*ID*>_<*roomID*> (e.g., `doorway1.2_room1.1`) and those with matching <ID>s (e.g., locations either side of doorways) are set as being in-line.
`robot_pose`	Obtains the robot's position from odometry and localisation. If the position is equivalent to a location that has previously been added to the state, the robot is assigned to the location using the `at-base` PDDL predicate. If the robot is at an unknown location, a new location is created and inserted into the PDDL problem file. Based on the work of Speck et al. [3].
`sensed_-obstacles`	Adds PDDL statements for any identified obstacles on the robot's path. Identification can come from running object recognition algorithms on a robot's RGB camera or directly from the cloud (e.g., closed door). Obstacles that have been acted on and are no longer obstacles, are removed from the problem.
`goal_creator`	Sets the robot's goal. This Static State Estimator is used at the highest level of the hierarchy. Further details are specific to the experiments that were ran (see Sections 5.7 and 5.8).

We categorise State Estimators as either static or dynamic. Static State Estimators are called once, and insert immutable state information such as waypoints and immovable objects. Dynamic State Estimators are called at the start of each iteration to update the problem file; for instance, to reflect a change in the current state or to add a new goal. Table 5.1 presents an overview of the different State Estimators in our system; further State Estimators can easily be added.

5.5.1.2 Defining Actions and Devices

The PDDL domain file contains the set of action definitions from which the TFD/M planner generates a task plan. In traditional approaches, this list is predefined with a fixed set of actions and the same domain file is reasoned on in every iteration. In realistic environments, this list can quickly grow in size, for example, for each type of object a robot may encounter different manipulation actions could be defined.

Instead of always providing an exhaustive and predefined list of actions to the TFD/M planner, a Domain Enricher is called. The Domain Enricher analyses the PDDL problem, obtained after calling the State Estimators and expands it with actions that are relevant to the defined entities/objects. Starting from a minimal domain file, only containing the most elementary actions a mobile robot can execute (e.g., for a navigation layer this includes actions to drive around the environment and inspecting objects), further actions are inserted in subsequent planning iterations. For instance, if a closed door is detected on the robot's path, the `sensed_obstacles` State Estimators will add a door to the problem file and the Domain Enricher will insert the open door action definition, which is shown in Listing 5.1. Similarly, if the robot discovers a box, a `push_box` action will be inserted into the domain file. These actions are obtained by requesting all actions that can be performed on the object type, for example, objects of type door, from the cloud (which, for each object type, has a file containing the set of action definitions).

The actions provided by IoT devices are also defined in the domain file. Since the continual planner must be able to request a device to execute an action (e.g., the robot itself, a door pump, an IoT light switch or another robot), one solution would be to declare all individual devices as objects in the PDDL problem file. Due to the resulting expansion of the problem file, this solution would lead to long planning times. We recall the reader that frequent replanning might be required in dynamic smart environments. Rather than defining all IoT devices in the PDDL problem file, we introduce a single "remote" object that represents all remote devices.

Listing 5.1: The `open_door` action definition.

```
(:durative-action open_door
 :parameters (?r - device ?s - location ?g - location ?d - door)
 :duration (= ?duration 5000)
 :condition ( and
    (at start (object-is-in-path ?d ?s ?g))
    (at start ([checkCanOpenDoor open_door ?r ?d]))
  )
  :effect ( and  (at end (not (object-is-in-path ?d ?s ?g)) )  )
)
```

5.5.2 Capability Checking

The availability of remote devices, and thus the actions they can perform may change over time. An IoT device may become unavailable for numerous reasons, for example, because it is being used by another service (or human), is under maintenance, is low on battery or has a degraded network connection. As some actions can be performed by more than one device, this does not necessarily mean the action cannot be executed. For instance, a door can be opened by an electric pump, another robot or a nearby human; and a cup of coffee can be fetched from another location if the nearest machine has run out of coffee beans.

During the search for a plan, the TFD/M planner will evaluate the conditions of an action many times with different argument combinations. It is difficult to formally represent the knowledge that is needed to reason on the availability of an action in PDDL. Therefore, our framework provides an external module, which in turn leverages an ontology-based reasoner, deployed in a cloud back-end. In this section, the robot's capability checking module and Capability Checker are described, followed by the cloud's Ontology Capability Reasoner.

5.5.2.1 Capability Checking Module

The TFD/M task planner of Dornhege et al. [24] enables external modules to be called during the planning phase. These enable domain-specific code to be executed. Inside the PDDL domain file we define a list of external module definitions. Like predicates, these are stated within the conditions of actions and are checked during the search for a plan, but rather than being compared to the world model, external code is called. In our case, this is a single capability checking module.

The first two arguments in the capability checking condition are always the name of the action and a device, i.e., the name of the local robot or "remote". The name of the action is given so that, after a plan has been found, a planned remote action can be associated with the discovered (capable) remote device(s). The number and type of the remaining argu-

ments are specific to the action. For instance, in the example of Listing 5.1, `checkCanOpenDoor` has a parameter of type `door`; whereas `checkCanFillObject` would have two parameters, for example, a `cup` and `coffee`. If the name of a robot is passed (as the second argument), the capability checking module will evaluate based on the robot's current capabilities. If "remote" is passed, the module will communicate with the Capability Checker, which queries the cloud for a list of capable devices and caches that list. The cache is queried for subsequent calls to speed up the process. Once a branch of the plan has been executed, the cache is cleared for actions within that branch (layer). This allows any newly available devices to be reasoned on in subsequent branching. In the cloud back-end, ontology reasoning determines the list of remote devices that can execute an action.

5.5.2.2 Ontology-Based Reasoning

To decouple the availability of an action from the availability of a specific device, the requirements of each action are modelled as a set of capabilities and each device has a number of capabilities. These relationships, depicted in Figure 5.4, are inspired by the SRDL ontology [19]. We have expanded the ontology by allowing capabilities to have requirements (e.g., the `OpenDoorCapability` can require a door to be specified) and an associated cost. The cost allows the planner to select the least costly device to perform an action. Currently this is a static (manually defined) value; in the future we envision more intelligent reasoning and learning being utilised. Moreover, we created a device class which represents simple devices, such as electronic doors, IoT lifts and smart coffee machines, as well as (mobile) robots defined in SRDL [19].

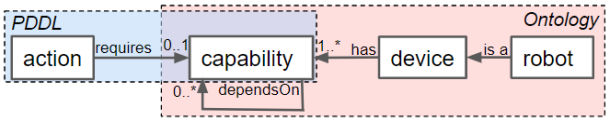


Figure 5.4: Outline of what we define action, capability, device and robot as. PDDL actions require a capability and devices along with what they are capable of are defined in an ontology.

Upon receiving the request for capable devices, the ontology is queried for devices with a capability of the required type, for example, for all devices that have a capability of type `OpenDoorCapability`. The `OpenDoorCapability` has a "canOpenDoor" property matching a specific door; if this property is missing the device (or human) is assumed to be able to open all doors. The Ontology Capability Reasoner also queries the IoT Context Monitor to check the device is available (e.g., is not down for maintenance). All

capable devices, along with the costs associated with the capability, are returned to the robot.

5.5.3 Context Monitoring

After a (hierarchical) task plan has been formulated, context monitoring and plan execution (see Section 5.5.4) occur in parallel. Context monitoring is the process of continuously evaluating the sensor data received from IoT devices, as well as the status of the devices, and identifying any possible state changes that affect the current plan. If replanning is necessary, this is performed at the layer of the hierarchy affected by the state change. The robot's onboard Context Monitor is discussed, followed by the cloud-based Plan Validator and IoT context monitors (i.e., Availability Monitor and World State Monitors).

5.5.3.1 Robot's Onboard Context Monitor

The Context Monitor, running onboard the robot, is the component responsible for keeping track of the hierarchical task plan and triggering the replanning of a branch of the plan. When a plan has been generated, it replaces all references to the generic "remote" object with a list of actual devices, which it obtains from the Capability Checker. The full hierarchical task plan, containing the device list(s), is then announced to the cloud-based Plan Validator. The Context Monitor will receive relevant IoT state updates from the cloud.

Although performing planning in the now restricts how far in the future a detailed plan is produced, state changes can still cause parts of the plan to become infeasible. The robot itself may detect that its current action will/has failed, for example, a box is blocking its path when executing a `drive_base` action; or an IoT device may detect a state that will cause one or more of the robot's planned actions to fail, for example, a door closes. In both these cases the robot needs to replan, to either bypass the issue (e.g., find a different route) or solve it (e.g., open the door). When a state change that violates one or more preconditions of a planned action occurs, the state change is inserted into the robot's knowledge base and the executing action from the affected layer is stopped, along with its subtasks. For instance, in Figure 5.1c, if primitive action 1.2 will fail, composite action 1.1 (as it is currently being executed) is stopped. Thus, primitive action 1.1.2 is also stopped. This will trigger composite action 1 to replan its subtask.

In other situations, a remote device may become unavailable. If there is an alternative device available and all necessary dependencies are still met, no replanning is required. This is because the actual device that will execute a remote action is only resolved by the continual planner when the execution

of a remote action is started.

Even when an alternative device is present, it might still be necessary to trigger a replanning. This is the case if another action depends on the choice of the remote device. For instance, in the plan: (1) `move_to_object( rob1 remote)`, (2) `hand_over_object(rob1 cup remote)`, (3) `fill_object( remote cup coffee)`, where "remote" will be replaced by [`coffee_machine1`, `coffee_machine2`], the actions prior to the remote action (i.e., `fill_object`) mention the remote device. Therefore, if the execution of one of these actions has started and the initially selected remote device (i.e., first device in the list) malfunctions, replanning is required to enable the robot to move to the alternative coffee machine. Note: if remote actions have dependencies, remote actions that are executed by different devices should be planned within different branches of the hierarchy (e.g., the robot should move to and use one device before planning its navigation to and use of, a second device).

5.5.3.2 Plan Validator and IoT Context Monitor

The Plan Validator communicates with robots and keeps a list of each robot's plan and the objects/devices whose state could affect that plan. The IoT context monitors, i.e., an Availability Monitor and set of World State Monitors, receive messages from the IoT middleware and provide the object specific information the robot requires when its plan is affected by a state change. This section describes the Plan Validator, followed by the Availability Monitor and World State Monitors.

The Plan Validator receives the (JSON) messages sent by one or more robots' Context Monitors. These contain the robot's IP address and hierarchical task plan in which "remote" has been substituted with the list of capable devices. The Plan Validator parses the task plan. Then queries the IoT context monitors for the set of objects (e.g., doors, cups, boxes, etc.) and devices, whose state could affect the robot's planned actions. If the current state of an object or device already affects the robot's plan, the robot is instantly informed. To receive the state changes, the Plan Validator registers itself as an observer of the Availability Monitor and each of the World State Monitors (which are observables). When informed that an object/device has changed state, the Plan Validator notifies the appropriate robots. The notifications, sent to the robot(s), contain which action the state change affects and the information provided by the monitors. Although the Plan Validator handles multiple robots' plans, resource negotiations are beyond the scope of this chapter.

When initialised, the Availability Monitor and World State Monitors query

the ontology for the list of objects they should monitor the state of, that is, the Availability Monitor gets all objects of type device and, for example, a Door World State Monitor gets all objects of type door. To determine which objects a robot should know the state of a default method is provided, which simple checks the monitor's list of objects against the robot's planned actions' parameters and returns the matches. In some cases, some domain specific processing is required. For example, the Door World State Monitor checks the plan for `drive_base` actions and returns a list of doors that sit between two locations the robot plans to drive between (these locations are also stated in the ontology).

When the IoT middleware (i.e., DYAMAND), announces a state change (i.e., posts a JSON string, which contains the name of the object/device and its state) the relevant monitor consumes that message. The monitor then provides the Plan Validator with the information that should be sent to the robots, whose plans could be affected by the change. In the case of the Availability Monitor, this information just contains the name of the device and its availability (i.e., a boolean value). For the World State Monitors, this includes the objects and PDDL statements that should be inserted into the problem file. For instance, the Door World State Monitor provides a PDDL statement declaring which locations the door blocks the robot from driving between, for example, `(object-is-in-path door1.1 doorway1.1_room1.1 doorway1.1_room1.2)`.

5.5.4 Plan Execution

Each action, defined in the PDDL domain hierarchy, is linked to an Action Executor, a plug-in containing the logic and low-level instructions to be executed. These plug-ins implement an `execute()` method, which is called by the continual planner. The workings of this method depend on whether the Action Executor is categorised as a Primitive Action Executor or a Composite Action Executor. Examples of both types are provided in Appendix 5.A. Figure 5.5 illustrates how these interact with the other components of our framework.

5.5.4.1 Primitive Action Executors

Primitive Action Executors perform the lowest-level actions, for example, `open_door`, `drive_base`, `request_lift`. These include Local Action Executors and a single Remote Action Executor. Local Action Executors interact with the robot's actuators and sensors through a robot middleware, for example, Robotic Operating System (ROS) [43]. For example, the `drive_base` action will be executed by the `DriveBaseActionExecutor`, which, after querying the knowledge base for the geometric position of the symbol-

ically represented location, interacts with the `move_base` ROS ActionLib to command the robot to drive to a specific position.

Any actions involving remote (off-board) devices are delegated to a single Action Executor that acts as a proxy for all IoT actuators. When remote action execution is required, the `RemoteActionExecutor` will retrieve the (least costly) device to execute the action from the Context Monitor. The action request is then sent to the cloud back-end (via a blocking HTTP post), where it is redirected to the appropriate actuator service via the IoT middleware.

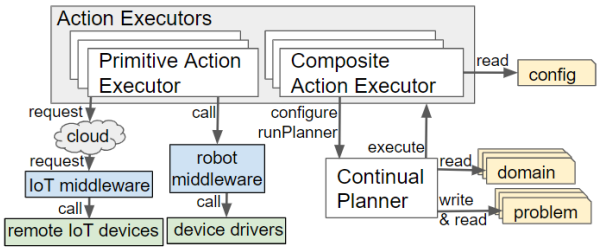


Figure 5.5: Functionality of the types of Action Executors. Grey boxes indicate types of plug-ins, white indicates classes/instances and yellow shows files.

5.5.4.2 Composite Action Executor

Further planning is required for high-level (abstract) actions in order to create a plan containing only primitive actions. When invoked, a Composite Action Executor will start a new instance of the continual planner and configure it with the necessary domain file, State Estimators and Action Executors. If the resulting plan also contains composite actions, the procedure is repeated. Using this recursion, the execution of a composite action, ultimately, results in a set of calls to Primitive Action Executors. We provide a base `CompositeActionExecutor` class, which performs the action execution and includes 4 overridable methods: i) to load the correct plug-ins and domain file (which, by default, are read from the configuration file), ii) set the goal state, iii) handle successful completion and iv) handle failure.

A threshold on the number of sense-plan-act iterations a composite action will perform can be set to prevent endless attempts. Performing multiple iterations enables any additional (evolving) state to be taken into consideration. For instance, if a human crosses the path of the robot, the robot's drive action might temporarily fail. By preforming a sense-plan-act iteration, the robot can continue driving after the human has moved. When the threshold is reached, the cloud back-end is notified. This enables the task

to be assign to an alternative robot. In the future, this can enable the system to learn about how successful composite actions are likely to be given the current context.

5.6 Designing the Hierarchy for Smart Environments

An important aim of our framework is to reduce the frequency of replanning and, when replanning is required, reduce the time spent replanning. Applying principles of hierarchical planning, the domain knowledge is split-up across multiple re-usable domain files, each containing action definitions written in PDDL. Moreover, only knowledge relevant to the current context is inserted into the PDDL problem file. Actions in higher levels of the hierarchy are more generic and abstract the details of the lower level actions. This section describes the design process performed to create the layers of the hierarchy for our experiments.

First, an example scenario set in a smart home is introduced, followed by the drawbacks of not using a hierarchy. The last three subsections describe the three ordered steps an engineer could take to design the hierarchy: (i) splitting up key concepts, (ii) separating repeated blocks and (iii) reducing unused state knowledge. The actions contained within each domain file, for each design step and a plan created using the domain file(s) are shown in Figures 5.6, 5.7, 5.8 and 5.9. To improve legibility for the reader, figures show only the actions relevant for our discussion. The full plans are provided in Appendix 5.B. As previously mentioned, a description of all composite and primitive actions in our final domain files is given in Appendix 5.A.

5.6.1 Example Scenario

A smart house, with two floors and an IoT-controllable lift, is notified via a bed sensor that a child steps out of bed when they are supposed to be sleeping. The child may have woken up because, for example, it is too hot, they want the night light on or they are thirsty. Initially the robot is at its charging location and, when the child gets out of bed, the robot is instructed to complete the child's request. The robot must drive to the bedroom, find out what the child wants, carry out their request, which is to switch on a night light and then return to the nearest charging station to await further instruction. Below, this scenario is worked out.

Moreover, to show the applicability of our system to different situations, our domain files also contain actions needed to guide a visitor and to fetch a cup of coffee. As well as being applicable to a smart home, these are applicable to a smart office building.

5.6.2 Baseline: All Actions and State Within a Single Level

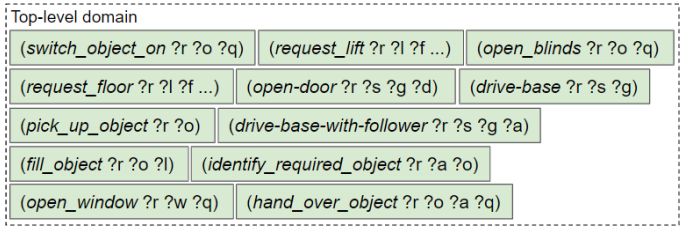
Without a hierarchy, a planner would need to plan in terms of very specific low-level actions, a few examples are shown in Figure 5.6. When the goal is provided to the planner, the entire plan is generated. Therefore, if a state change causes the plan to become invalid (e.g., a device becomes unavailable or a door in the robot's path closes), the robot must replan the entirety of its plan. As the state may change again prior to the affected action being executed, the time spent (re)planning may be deemed unnecessary. For our example scenario, the Listing in Figure 5.6 shows the plan generated by the TFD/M planner. In the generated plan, after discovering the child's request, the robot starts driving towards the location of its charging point (action 9) before performing the user's request to switch on the night light (action 10). More natural behaviour would be to switch the night light on first. This could have been amended by creating an atom that prevents the robot from driving while fulfilling a request; however, we opted to not add this atom as to fulfil a request the robot might need to drive.

5.6.3 Splitting up Key Concepts

The main disadvantage of the above approach is the requirement to replan when a different request is made or the state of the environment changes. The chance of replanning can be reduced by adding a layer of abstraction onto the primitive actions. At a high-level the scenario has three main subtasks: (i) discovering what the request is, (ii) performing the request and (iii) returning to the charging point. Hence, a first step towards the design of our hierarchy is to create two levels, with the three generic actions in the top level and specific actions in a primitive domain. The resulting domain files are represented in Figure 5.7 alongside the actions that would be executed, i.e., the composite and primitive actions that were planned by running our framework.

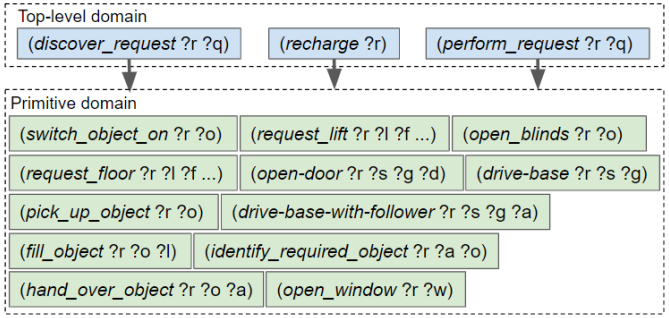
5.6.4 Separate Repeated Blocks

There are sets of (composite and/or primitive) actions which will be planned by many different composite actions, for example, navigation and object manipulation. We separate these actions into different domain files, to allow them to be used by multiple composite actions. In our example (see Figure 5.8), we created separate domain files for actions related to object interaction and actions that enable the environment to be navigated. By performing this separation, the range of tasks the robot can perform can be expanded without having to either repeatedly re-write the PDDL for navigation or continually add actions to a single increasingly complex domain file.



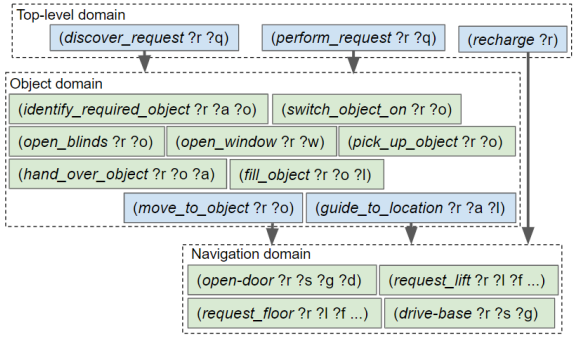

```
1:(drive_base rob1 w1.1_r1.0 d1.3_r1.0)
...
4:(request_lift lift0_device lift0 f1 ...)
5:(drive_base rob1 d1.6_r1.2 d1.6_lift0loc)
6:(request_floor lift0_device lift0 f2 ...)
7:(drive_base rob1 d1.6_lift0loc d2.6_r2.2)
8:(identify_required_object rob1 human1 request1)
9:(drive_base rob1 d2.6_r2.2 d1.6_lift0loc)
10:(switch_object_on plug_device night_light1 request1)
11:(request_floor lift0_device lift0 f1 ...)
...
15:(drive_base rob1 d1.3_r1.0 w1.1_r1.0)
```

Figure 5.6: All actions definitions are contained within a single domain file. When planning everything upfront, the planner must make an assumption on the most likely request and include all actions needed to fulfil that request. In all figures the following words are abbreviated: doorway is shortened to d, waypoint to w, room to r and floor to f. In all proceeding figures blue boxes (with downwards arrows) indicate composite actions and green is used for primitive actions.



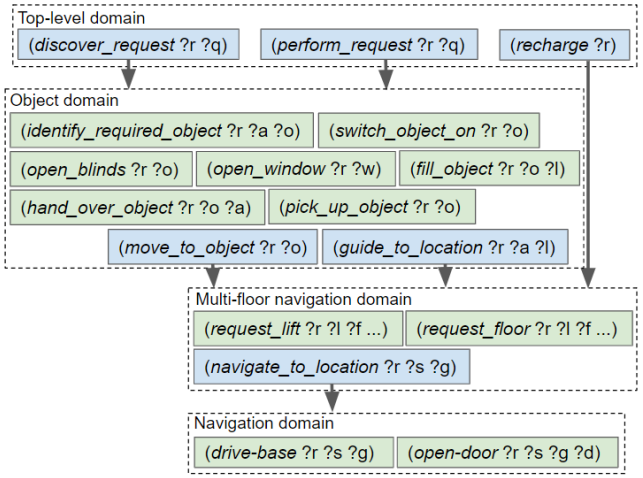

```
1:(discover_request rob1 request1 human1)
 1.1:(drive_base rob1 w1.1_r1.0 d1.3_r1.0)
 ...
 1.6:(request_f lift0_device lift0 f2 ...)
 1.7:(drive_base rob1 d1.6_lift0loc d2.6_r2.2)
 1.8:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
 2.1:(switch_object_on remote light1)
3:(recharge rob1)
 3.1:(drive_base rob1 d2.6_r2.2 d1.6_lift0loc)
 3.2:(request_floor lift0_device lift0 f1 ...)
 ...
 3.6:(drive_base rob1 d1.3_r1.0 w1.1_r1.0)
```

Figure 5.7: Action definitions split-up into two domain files; and executed actions when the state and actions are split up by key concepts.




```
1:(discover_request rob1 request1 human1)
 1.1:(move_to_object rob1 human1)
  1.1.1:(drive_base rob1 w1.1_r1.0 d1.3_r1.0)
  ...
  1.1.6:(request_f lift0_device lift0 f2 ...)
  1.1.7:(drive_base rob1 d1.6_lift0loc d2.6_r2.2)
 1.2:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
 2.1:(switch_object_on plug_device light1)
3:(recharge rob1)
 ...
 3.6:(drive_base rob1 d1.3_r1.0 w1.1_r1.0)
```

Figure 5.8: Action definitions split-up into three domain files. Executed actions when repeated groups of actions are separated.




```
1:(discover_request rob1 request1 human1)
 1.1:(move_to_object rob1 human1)
  1.1.1:(navigate_to_loc rob1 r1.0 r1.2)
   1.1.1.1:(drive_base rob1 w1.1_r1.0 d1.3_r1.0)
   ...
  1.1.2:(request_lift lift0_device lift0 f1 ...)
  1.1.3:(navigate_to_loc rob1 r1.2 lift0loc)
   1.1.3.1:(drive_base rob1 d1.6_r1.2 d1.6_lift0loc)
  1.1.4:(request_f lift0_device lift0 f2 ...)
  1.1.5:(navigate_to_loc rob1 lift0loc r2.2)
   1.1.5.1:(drive_base rob1 d1.6_lift0loc d2.6_r2.2)
 1.2:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
 2.1:(switch_object_on plug_device light1)
3:(recharge rob1)
 3.1:(navigate_to_loc rob1 lift0loc r2.2)
   3.1.1:(drive rob1 d2.6_r2.2 d1.6_lift0loc)
 3.2:(request_floor lift0_device lift0 f1 ...)
 3.3:(navigate_to_loc rob1 lift0loc w1.1_r1.0)
  ...
  3.3.4:(drive_base rob1 d1.3_r1.0 w1.1_r1.0)
```

Figure 5.9: Domain files and executed actions for when we have split-up knowledge about what floor the robot is on. The fully hierarchical approach.

5.6.5 Reducing Unused Knowledge

When the number of objects within a problem file is increased, an exponential increase in the planning time is observed. This is due to the state explosion problem [44, 45]. The fewer objects included within the PDDL problem, the fewer possible states and therefore state transitions, the planner has to search through. We aim to minimise the number of objects within the problem by only inserting those relevant to the robot's current context or are likely to soon appear within the robot's plan. This is achieved by splitting up more distant and loosely related objects. For our example, we split the navigation state knowledge up based on what floor of a building the robot is currently on. If the robot is on floor 1, it does not need to know the details (e.g., rooms and waypoints) of floor 2 until it reaches that floor. The domain and resulting plan are shown in Figure 5.9.

To provide an alternative example, which uses the types of objects rather than their location, a factory domain is described. In a smart furniture factory, a robot could be requested to construct a table and a chair. When making a table the robot does not need to know the state of the parts and tools required to make the chair. Thus, rather that reasoning over all objects and actions, the construction of the chair and table can be split into two separate branches.

Further splitting up of our hierarchy could be performed (e.g., navigation separated by room), however, as each level needs to be planned adding more levels causes unnecessary additional computational costs. If there is no clear separation and two actions/sub-goals are greatly dependent on the state of an object, these actions should not be split-up. Initially, designing the hierarchy may take more time than writing a single PDDL domain file but these layers are reusable.

5.7 Proof of Concept in a Smart Home

We developed a proof of concept in our HomeLab (https://www.imec-int.com/en/homelab) (smart house) with a Pepper (humanoid) robot. The HomeLab is a two-story house (600 m^2) equipped with sensors and a home automation system, which communicates via DYAMAND [40]. The components onboard the robot were developed as nodes for the Robot Operating System (ROS) [43]. The robot used in our proof of concept has very limited dexterity, i.e., is unable to carry or manipulate objects. For robots with greater dexterity, rather than just navigating the environment and commanding IoT devices, further tasks could be included, for example, the option of making and delivering a drink (see Section 5.8.2 for a simulated coffee fetching scenario).

A similar scenario to that described in Section 5.6.1 was setup. In this experiment the child is located in a room on the same floor as the robot and the robot requires the room door to be opened and the room light to be switched on before it can enter the room. This experiment shows how IoT devices enable a robot to complete a task it would otherwise be incapable of. Photographs alongside a description of the actions the robot is performing are shown in Figure 5.10 and videos have been provided as supplementary material. This section describes the steps taken by our framework, for when the hierarchical structure of Figure 5.9 was provided as input and the robot's goal was set to `(completed request1)`.

Initially, the robot's `navigate_to_location` planning branch only knew the static map (walls and waypoints) and thus generated a plan containing two `drive_base` actions, see Listing 5.2. The full (hierarchical) task plan was submitted by the robot's Context Monitor to the Plan Validator in the cloud. The cloud (i.e., the Door State Monitor and Light State Monitor) reasoned on the path in the plan and realised the robot required the room door to be opened and the room light to be switched on. As this state invalidated the current plan, the Plan Validator sent this information to the robot's Context Monitor. The Context Monitor preempted (i.e., stopped) the `drive_base` action although the robot by itself reported no failures (since it had not arrived at the closed door yet).

Listing 5.2: Plan prior to the robot being informed that the care door is closed and the light is off.

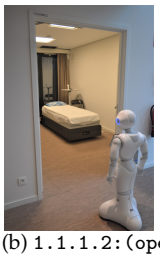
```
1:(discover_request rob1 request1 human1)
 1.1:(move_to_object rob1 human1)
  1.1.1:(navigate_to_location rob1 room1 room2)
   1.1.1.1:(drive_base rob1 doorway0_room1 doorway1_room2)
   1.1.1.2:(drive_base rob1 doorway1_room1 doorway1_room2)
 1.2:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
```

Only the `navigate_to_location` branch of the plan was modified by this cancellation. At the start of a new iteration of the continual planner, the `sensed_obstacles` State Estimator inserted the door and light objects into the PDDL problem. As replanning was required, the Domain Enricher immediately loaded the `open_door` and `switch_room_light_on` action definitions.

Using the Capability Checker, the robot's planner determined that the robot itself did not have the capability to open doors or switch on lights. Therefore, without the assistance of IoT actuators the robot would be unable to complete its task. In our ontology, there are two devices capable of opening the door: the IoT enabled door actuator and a human. Once a plan

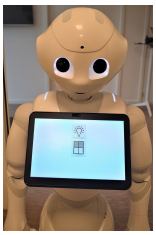
The robot navigates to the child, located in the care room (Listing 5.2). When it receives a notification, that the door is closed and the light is off, the `navigate_to_location` plan branch is replanned. The new plan (Listing 5.3) contains actions to open the door and switch on the light.

(a) `1.1.1.1:(drive_base rob1 d0_r1 d1_r1)`



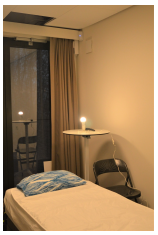
Before entering the room, the robot opens the door and switches on the light. These remote actions are sent to the relevant devices via DYAMAND.

(b) `1.1.1.2:(open_door remote d1_r1 d1_r2 care_room_door)` and `1.1.1.3:(switch_room_light_on remote d1_r1 d1_r2 r2_light)`



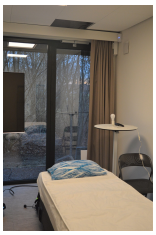
The robot asks the child what request they would like performing; and the child response by clicking the relevant button on the robot's tablet.

(c) `1.2:(identify_required_object rob1 human1 request1)`



As the child asked for the night light to be switched on the robot invokes a remote action, which gets sent to the plug device via DYAMAND. The light by the bed switches on.

(d) `2.1:(switch_object_on remote light1)`



If the child asked for the blinds to be opened the robot invokes a remote action, which gets sent to the blinds via DYAMAND. The room blinds open.

(e) Alternative request: `2.1:(open_blinds remote blind1)`

Figure 5.10: Real world tests. The full list of PDDL executed actions are given in Listings 5.3 and 5.4. Doorway has been shortened to d and room to r.

(i.e., Listing 5.3) had been generated, the Context Monitor retrieved the devices that can perform the remote actions from the Capability Checker. Subsequently, it replaced the "`remote`" objects in the plan with an ordered list of these devices, then sent the plan to the cloud. Capable devices are ordered by their static cost, which was manually defined in the ontology (i.e., the human's capabilities were set as more costly than the IoT devices'). If a device within the plan becomes unavailable, the cloud informs the robot, which would then use the alternative device. Since it is less costly to request the IoT enabled door actuator to perform the action, the `RemoteActionExecutor` asked this device to perform the `open_door` ac-

tion. These steps are the same for the room light but the remote action request was sent to the IoT light switch.

Listing 5.3: Plan after the robot had been informed that the care door is closed and the light is off.

```
1:(discover_request rob1 request1 human1)
 1.1:(move_to_object rob1 human1)
  1.1.1:(navigate_to_location rob1 room1 room2)
   1.1.1.1:(drive_base rob1 doorway0_room1 doorway1_room2)
   1.1.1.2:(open_door remote doorway1_room1 doorway1_room2 care_room_door
       ↪)
   1.1.1.3:(switch_room_light_on remote doorway1_room1 doorway1_room2
       ↪room2_light)
   1.1.1.4:(drive_base rob1 doorway1_room1 doorway1_room2)
 1.2:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
```

Once the robot entered the care room the child pressed a button on its tablet, which identified the request they wanted performing. The `perform_request` action was then executed. Depending on the child's request this either generated a sub-plan to switch on the night light (see Listing 5.4) or to open the blinds. During the experiments, both these actions were performed by remote devices.

Listing 5.4: The robot's plan after the execution of `perform_request` had begun.

```
2:(perform_request rob1 request1)
 2.1:(switch_object_on remote night_light1)
```

5.8 Simulated Experiments

Through simulated experiments we aim to demonstrate (i) how much representing all devices as a single "remote" device has reduced the number of calls from the TFD/M planner to the Capability Checker; (ii) our framework's ability to trigger replanning when a device becomes unavailable, (iii) the effect increasing the number of objects in the environment has on the planning time and (iv) the effect increasing the number of layers in the PDDL hierarchy has on the planning time. All these experiments use an environment simulated with Gazebo, this is shown in Figure 5.11. The State Estimators and Action Executors used are described in Table 5.1 and Appendix 5.A, respectively; the goal the `goal_creator` sets is specific to each experiment. Each subsection describes the experiment set-up, followed by the results.

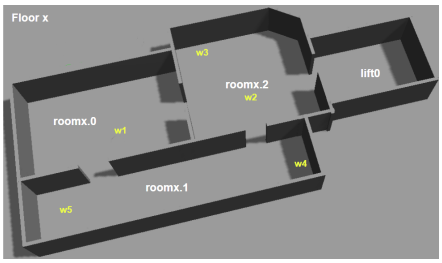


Figure 5.11: Gazebo simulated world used for the experiments. We use an extended version of the world originally created by Speck et al. [3]. Waypoints (e.g., `w1`) that are used during the different experiments have been indicated on the map. For all experiments there are also waypoints on either side of doorways.

5.8.1 Exp. 1: Single "remote" Object to Represent all Devices

In this section, experimental results are presented to show how including all remote devices as separate objects in the PDDL problem compares to representing the remote devices as a single object. The number of calls from the TFD/M planner to the capability checking module, for a varying number of devices, are provided in Table 5.2. The total number of states generated by TFD/M, during its search for each (sub)plan, is also provided. Further details on TFD/M are provided in [24, 46].

5.8.1.1 Set-Up

In this experiment the robot is tasked with simply navigating to another room on the same floor (i.e., from w1 in room1.0 to room1.2). Along its way, the robot must open a door and switch on a light. An initial plan without involving doors and lights is generated and sent to the Plan Validator, which informs the robot that the door is closed and the light is off. This causes the robot to replan and thus create a plan that includes opening the door and switching on the light.

Listing 5.5: Actions executed when a hierarchical planning is performed using a single remote device.

```
1:(navigate_to_location rob1 room1.0 room1.2)
  1.1:(drive_base rob1 waypoint1.1_room1.0 doorway1.3_room1.0)
  1.2:(open_door remote doorway1.3_room1.0 doorway1.3_room1.2 door1.3)
  1.3:(switch_room_light_on remote doorway1.3_room1.0 doorway1.3_room1.2
      ↪light1.2)
  1.4:(drive_base rob1 doorway1.3_room1.0 doorway1.3_room1.2)
```

The list of executed actions when there was a single "remote" object is shown in Listing 5.5. When all devices were stated within the problem,

the same plan was produced but instead of "remote" the actual device name was stated in the plan. For this experiment, we simple increased the number of devices in the ontology and, when all devices were represented, these were also inserted into the PDDL problem file.

5.8.1.2 Results and Discussion

The results of this experiment are presented in Table 5.2. When all devices were stated in the problem, as the number of devices is increased so did the number of calls to the capability checking module and the number of generated states. Whereas, when a single remote object was stated no increase was observed. For each of these experiments, the onboard Capability Checker only performed 2 HTTP requests to the cloud-based Capability Reasoner, i.e., one for each of the remote actions (the `open_door` and the `switch_room_light_on` actions), as the Capability Checker caches the list of capable devices.

Table 5.2: Comparison of adding all devices to the PDDL problem versus using a single object to represent all remote devices. "Generated States" is the aggregated number of states TFD/M generated during its search for the (sub)plans.

	All Devices Represented					**Single Device Represented**
Number of Devices	**5**	**10**	**15**	**20**	**25**	**Any**
Calls to capability module	365	680	1005	1520	1850	56
Generated States	749	1035	1346	1913	2241	349

5.8.2 Exp. 2: IoT Device Failure and Plan Quality

Our second set of simulated experiments employed a scenario in which an IoT actuator, that a robot must navigate to before using, became unavailable. The quality of the executed plan (i.e., number of primitive actions) and the time spent planning are discussed. All experiments were ran on a virtual machine with 9 GB of RAM and an Intel i7 2.9 GHz CPU. The scenario also demonstrates the applicability of our designed hierarchy to a different situation, i.e., a smart office environment. Further, these experiments demonstrate that our framework is able to monitor both hierarchical and non-hierarchical plans.

5.8.2.1 Set-Up

A smart office environment has two floors with identical floor plans, as shown in Figure 5.11. Each floor is equipped with an IoT coffee machine, which is located at waypoint `w4`. The robot started on the first floor, at location `w2` and was tasked with delivering a cup of coffee to a human. Specifically, the robot's goal was `(and (is-full cup1 coffee) (has-object human1`

cup1)). This task involved the following high-level steps: (i) getting a cup from location w3 on floor 1, (ii) asking one of the machines to fill the cup and (iii) delivering the cup to a human, located at w5 on the second floor.

We evaluated our framework on three possible situations: (i) the coffee machines always being available, (ii) the coffee machine on the first floor becoming unavailable and (iii) the coffee machine on the second floor becoming unavailable. The produced plans, for when neither coffee machine malfunctions, are provided in Appendix 5.C. For the last two situations, we disable the coffee machine just before the robot reached it, i.e., when the robot was executing the drive-base action that would have resulted in it reaching the coffee machine. This was simulated by sending a message to our cloud back-end, informing it about the coffee machine's unavailability, in other words, we acted as the IoT middleware.

The experiment was ran for our hierarchical approach and for when everything is planned upfront (referred to as "No hierarchy" in the results table). In both cases, our framework informed the robot about the coffee machine malfunctioning and, when necessary, replanning was triggered. Only the highest level of the hierarchy differs from the previous experiments. For this experiment, it contains three composite actions: get_object(?robot ?object), go_fill_object(?robot ?object ?filling), give_object_to_agent(?robot ?object ?agent). These action decompose by reusing the object domain (from Figure 5.9) and its subsequent layers. For the hierarchical planner, a single remote object represents all IoT devices in the environment, i.e., the lift and the two coffee machines.

For planning everything upfront, all primitive actions are contained within a single domain file. As the planner requires the location of all the coffee machines upfront, the single remote object cannot be used. Only the 3 remote devices required for the scenario are included within the problem. As shown in Section 5.8.1, adding more devices would slow down the planning due to an increase in the number of calls to the capability reasoning module.

5.8.2.2 Results and Discussion

The results, shown in Table 5.3, present the total planning time, planning time before the first primitive action was executed and number of executed primitive actions. All times are given in seconds and are the average of 5 runs. The total planning time was longer when a hierarchy was used, than without a hierarchy. This is due to the overhead of starting multiple (i.e., 13) instances of the task planner. Nevertheless, with a hierarchy, the robot started acting sooner because the robot, initially, only decom-

posed the first branch of its plan (see Appendix 5.C, Listing 5.12) and the navigation layer only reasoned on the waypoints located on the robot's initial floor (rather than both floors).

Table 5.3: Comparison of planning times and number of executed primitive actions for the fetching coffee scenario. t is the total planning time, t' planning time before the first primitive action is executed, e is number of executed primitive actions. All times are given in seconds and are the average of 5 runs. "No hierarchy" was not affected by coffee machine 1 breaking and "Hierarchical" was not affected by coffee machine 2 breaking.

	No Device Breaks		**Coffee Machine 1 Breaks**		**Coffee Machine 2 Breaks**	
	Hierarchy	**No Hierarchy**	**Hierarchy**	**No Hierarchy**	**Hierarchy**	**No Hierarchy**
t	11.30	5.27	13.68	-	-	8.21
t'	3.68	5.27	3.59	-	-	5.09
e	19	15	20	-	-	32

When neither coffee machine broke down, our hierarchical approach produced a plan of worse quality, that is, more primitive actions were executed, than the approach without a hierarchy. This was due to the coffee machine on floor 1 being selected, which produced a suboptimal plan. In a different situation/environment this difference could have been even larger. We see two options for improving this in future work. As the cloud's Plan Validator knows what the robot's plan is, the cost of a device executing an action could incorporate knowledge about the current and goal states of the robot. Alternatively, as the robot is informed about the location of both devices, it could create a plan to use each of the devices and select the shortest one. Both these approaches require further investigation.

When the coffee machine on floor 1 became unavailable, with our hierarchy the robot replanned the layer containing the remote device and in total executed 20 primitive actions (see Table 5.3). Planning without a hierarchy was not affected by coffee machine 1 breaking down as its plan did not contain that device. In contrast, for planning without a hierarchy, when the coffee machine on floor 2 became unavailable, the robot executed 32 actions as it navigated back to floor 1. These results demonstrate that even though planning everything upfront guarantees completeness, in dynamic environments it can be beneficial to deploy a hierarchical approach.

Moreover, if the coffee machine becomes unavailable and the robot has not started to execute the `move_to_object(robot1, remote)` action (i.e., is still fetching the cup), no replanning is performed by our hierarchical method. In contrast, for planning everything upfront, replanning is still required.

Furthermore, if the coffee machine becomes available again, replanning was pointless. This is also applicable to any state change that may occur, for example, the planned cup being used by another agent, a door being closed, the human moving location and so forth.

5.8.3 Exp. 3: Comparison with Planning Everything Upfront

In order to compare the computational times of planning everything upfront and our hierarchical method, we present the planning times for differing environment complexities. By incrementing the number of floors in our scenario, we are able to increase the complexity of the environment at a steady rate, in other words, the total number of object rises linearly. These objects include: waypoints, doors and all IoT devices. As an example, a floor is displayed in Figure 5.11.

5.8.3.1 Set-Up

For this experiment the robot navigated from its start location (i.e., `waypoint1.1_room1.0`) to a room on a different floor, which was chosen at random. The plans produced when the randomly selected room is `room2.2`, for planning everything upfront and for hierarchical planning are shown in Listings 5.6 and 5.7, respectively. We selected 3 random different rooms and ran each experiment 5 times; thus, each point on the results graph is the average of 15 runs.

The planning everything upfront approach takes, as input, a single domain file containing the actions required to navigate a multi-floor environment, i.e., `drive_base`, `request_lift`, `request_floor` and `open_door`. The State Estimators populate the problem file with the initial state for the following objects: a lift, the local robot, all remote devices (e.g., IoT lift and IoT doors), doors, floors, rooms and waypoints; and the goal is set to the required room, i.e., `(exists (?w - waypoint) (and (at-base ?w rob1) (in-room ?w <the required room>)))`.

Listing 5.6: Example task plan, produced when all planning is performed upfront.

```
1:(drive_base rob1 waypoint1.1_room1.0 doorway1.3_room1.0)
2:(drive_base rob1 doorway1.3_room1.0 doorway1.3_room1.2)
3:(request_lift lift0_device lift0 floor1 floor1 rob1 room1.2 lift0loc)
4:(drive_base rob1 doorway1.3_room1.2 doorway1.6_room1.2)
5:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
6:(request_floor lift0_device lift0 floor2 rob1 lift0loc)
7:(drive_base rob1 doorway1.6_lift0loc doorway2.6_room2.2)
```

For our hierarchical approach, we opted to reuse the multi-floor navigation domain and navigation domain (shown in Figure 5.9). The goal for the

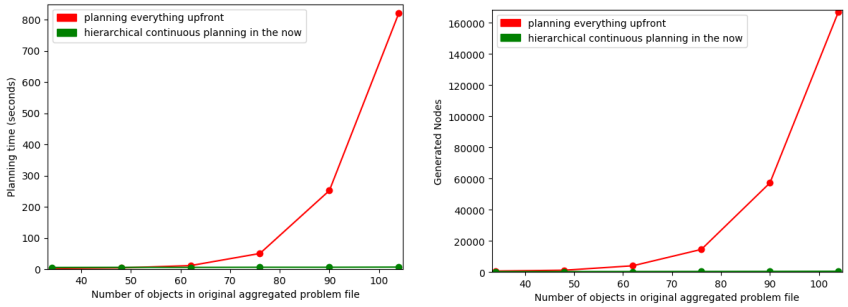
multi-floor navigation layer is set to the required room, i.e., `(at-location rob1 <the required room>)`. This layer contains state for rooms, floors, lifts, the local robot and a lift device. As shown in Listing 5.7, this will plan the use of the `navigate_to_location` composite action, which uses the navigation domain. The navigation problem file contains the objects (i.e., rooms, doors, the local robot, door devices and waypoints) and state for a single floor. Its goal is set to, for example, `(exists (?w - waypoint) (and (at-base ?w rob1) (in-room ?w <the required room>)))`.

Listing 5.7: Example task plan, produced when hierarchical planning is performed.

```
1:(navigate_to_location rob1 room1.0 room1.2)
  1.1:(drive_base rob1 waypoint1.1_room1.0 doorway1.3_room1.0)
  1.2:(drive_base rob1 doorway1.3_room1.0 doorway1.3_room1.2)
2:(request_lift lift0_device lift0 floor1 floor1 rob1 room1.2 lift0loc)
3:(navigate_to_location rob1 room1.2 lift0loc)
  3.1:(drive_base rob1 doorway1.3_room1.2 doorway1.6_room1.2)
  3.2:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
4:(request_floor lift0_device lift0 floor2 rob1 lift0loc)
5:(navigate_to_location rob1 lift0loc room2.2)
  5.1:(drive_base rob1 doorway2.6_lift0loc doorway2.6_room2.2)
```

5.8.3.2 Results and Discussion

As we increase the complexity of the environment there is an exponential increase in the planning time, this is shown in Figure 5.12a. Figure 5.12b shows the number of states TFD/M generates as this is deterministic and thus is not hardware dependent and does not vary between runs. For simpler environments, for example, less than 48 objects (split in 3 when our hierarchy is used), planning everything upfront performs slightly better than our hierarchical planner. This is due to the overhead of starting multiple instances of the planner in order to produce the layered plan. As planning everything upfront resulted in a much steeper exponential increase in time, our hierarchical approach performs substantially better in more complex environments. Our hierarchical planner only inserted objects on the robot's current floor into the problem of the navigation layer, allowing the planner to reason over less unnecessary information (and thus fewer states). This splitting up is also applicable to many other domains, for example, a factory domain, in which the knowledge about the different items being constructed can be split-up or a kitchen environment for which different meals (such as, making coffee or making a cheese sandwich) can be split-up.



(a) Planning time

(b) Number of generated states.

Figure 5.12: Planning times for planning everything upfront (red line, which ends in the top right corner) versus using a hierarchy (green line). For each result three random rooms were selected as goal locations and all experiments were ran 5 times.

5.8.4 Exp. 4: Varying the Number of Levels in the Hierarchy

How the PDDL actions and state knowledge are split-up has a large impact on the planning time. In comparison to planning everything upfront, our approach decreases the planning time for large state spaces. In contrast, for small state spaces our approach increases the planning time. Experiments presented in this section show how much of an impact adding more layers has on the computational cost.

5.8.4.1 Set-Up

To show results for when deploying a hierarchy produces longer and shorter total planning times we chose two environment complexities: (i) 70 objects split across 4 floors and (ii) 130 objects split across 8 floors. These experiments use the scenario and each of the actions and state knowledge splits described in Section 5.6, that is, the robot finds out what the child's request is, then accomplishes that request (which is to switch on a night light). This scenario could also be applied to fulfilling requests in a multi-storey office environment or a smart hotel. Moreover, rather than splitting the knowledge by floor, in a different scenario, we could have split by, for example, the objects required for different tasks. We choose floors for our experiments as it is reasonable that the the total number of object rises linearly when the number of floors is increased and to not overcomplicate this chapter with many different hierarchies.

For these experiments, the State Estimators populated the problem file with the initial state for the following objects: a lift, the local robot, all

remote devices (i.e., an IoT lift, all IoT doors), doors, floors, rooms and waypoints. Both the robot and child are located on different floors. As before, all experiments were ran 5 times. Each experiment employed a set of domain files and produced the corresponding plan, described in Section 5.6 (and Appendix 5.B). The "DF1" experiment used one domain file, containing all actions and thus everything was planned upfront (i.e., Figure 5.6). For "DF2" the two domain files and the executed actions are shown in Figure 5.7; the domain files and executed actions of "DF3" and "DF4" are shown in Figures 5.8 and 5.9, respectively.

5.8.4.2 Results and Discussion

As shown in Table 5.4, for 70 objects (4 floors) the more we split-up the actions and state the longer the total planning time. On the other hand, when our fully hierarchical approach (i.e., DF4) was ran, the first primitive action was executed sooner than when everything was planned upfront (i.e., DF1). When there were 130 objects (8 floors) the total planning time was shorter for our fully hierarchical approach. This was especially true for the amount of time spent planning before the first primitive action was executed.

Table 5.4: The impact increasing the number of layers has on the number of states and the planning time, when there are 70 and 130 objects in the original aggregated PDDL problem file. t is the total planning time in seconds, t' the planning time before 1st primitive action was executed and s is number of generated states.

	70 Objects				**130 Objects**			
	DF1	**DF2**	**DF3**	**DF4**	**DF1**	**DF4**	**DF3**	**DF4**
t'	10.66	6.05	7.10	6.14	77.07	21.82	15.64	6.52
s	2537	1126	1034	347	13,045	5905	4684	413
t	10.66	12.46	13.95	14.93	77.07	53.89	47.70	16.33

5.9 Conclusions and Future Work

In this chapter, we presented our Hierarchical Continual Planning in the Now framework, which aims to improve a robot's ability to act in dynamic environments with the assistance of IoT devices. IoT state knowledge is monitored by components in a cloud-backend and the robot is only informed about state relevant to its current task plan. During the planning phase, a single remote device object represents all remote (IoT) actuators; therefore, the robot's plan is less likely to require replanning when a devices becomes unavailable. Further, our experiments showed that when the environment contained 10 IoT devices, the number of calls to the external capability checking module decreased by 92 %.

World state further in the planning horizon is harder to predict and more likely to change. Therefore, tasks are split-up into subtasks and the first subtask is planned and executed before the subsequent one. Planning time increases exponentially as the number of objects increases [44, 45]; thus, to keep planning times tractable, only state relevant to the robot's current context is inserted into the planning problems. In our coffee bot scenario, these features enabled the robot to start acting after 3.68 s rather than 11.30 s. Conversely, planning everything upfront guarantees completeness and, when no replanning is required, the plan is likely to be more optimal than when a hierarchy is used. Nevertheless, our framework reduces the need for a robot to frequently replan and when replan is required, only a single branch (i.e., subtask) is replanned.

We see several directions for future work. First, our approach may lead to suboptimal plans. For example, if a robot is tasked with fetching a cup of coffee it would plan three distinct subtasks: 1. collecting a cup, 2. filling the cup using a coffee machine and 3. delivering the coffee. Which cup is more efficient for the robot to collect may depend on the location of the coffee machine but currently this is not taken into consideration while collecting the cup. In the future, we will investigate how information about the subsequent composite actions can be integrated within the composite action currently being executed.

Second, we make a worst-case assumption during planning, that is, that the world will constantly change in heterogeneous ways and these changes will continuously cause the robot's task plan to become invalid. In the future we will investigate applying the "just in time" concept to our planner, which will minimise the time between executing each branch of a plan. This entails planning a subsequent branch while the current branch is still being executed.

Third, we intend to evaluate running a probabilistic planner (e.g., [47]) to handle uncertainty in the outcome of actions. In particular we think this could be beneficial to our lower-level actions, such as navigation and object manipulation. Combining observations from noisy IoT and on-board sensors could help determine the probability of the environment's state, for example, the robot's or object's position.

Fourth, our architecture could be extended to multi-robot systems requiring conflict resolutions, that is, in which multiple robots require the same resources to complete their plans. One possible solution would be to expand our cloud-based Plan Validator. This component receives a copy of all robots' plans; therefore, could search for conflicts and inform robots about

restrictions they must adhere to. Resolving conflicting states without a centralised planner is likely to result in a suboptimal solution and if the robot cannot adhere to the provided restrictions, its planning will fail. Moreover, a robot could unset the state required by another robot, for example, close a door the another robot has just opened. How such challenges can be overcome by our system requires further investigation. For instance, the Plan Validator could contain a planning component that attempts to merge the robots' plans, then informs the robots of the modifications made to their plans.

Last, the framework could incorporate the functionality to learn about what actions/tasks have failed and what the state of the environment was at that time. As humans tend to have daily routines, this could include knowledge about what time of day the action was executed. This knowledge can be processed to tune the cost of actions (e.g., increase the cost of actions likely to fail) and to improve the state estimation.

Merging IoT sensors and actuators and robotics is a promising trend in distributed robotics. We hope that our work may inspire other researchers to further work in this emerging domain.

Supplementary material: The following are available online at https://www.mdpi.com/1424-8220/19/22/4856/s1, Video S1: planning in a smart home–open blinds, Video S2: planning in a smart home–switch on light

Appendix 5.A Description of Actions

A description of the composite and primitive actions, used for our experiments, are provided in this section.

Appendix 5.A.1 Composite Action Description

The Composite Action Executors are described in Table 5.5.

Appendix 5.A.2 Primitive Action Description

The Primitive Action Executors are described in Table 5.6.

Table 5.5: Description of composite actions.

Composite Action	Description	Example Goal(s) for Lower-Level
`discover_request`	The robot must move to the human who has a request and ask them what they want. Uses the object domain file to generate a plan.	`(not (is-unknown request1))`
`perform_request`	The goal, which is read from the robot's knowledge base, of its task planner instance will depend on the request. The effect of this action is: `(is-completed ?request)`. In our example scenarios the object domain is used; however, a different domain could be used for differing requests.	Depends on what the request is. e.g., `(is-on night_light)`
`recharge`	Robot needs to navigate to the nearest waypoint that is a charging location. Uses the multi-floor navigation domain.	`(at-location rob1 waypoint1_room1.1)`
`move_to_object`	Gets the location (i.e., room or waypoint) of the object (e.g., human) from the knowledge base. Creates an abstract plan for the robot to navigate to the correct floor and room using the multi-floor navigation domain.	`(at-location rob1 waypoint2_room2.0)` or `(at-location rob1 room1.1)`
`guide_to_location`	This is similar to `move_to_object` but an agent (e.g., human) is following the robot to the location. Creates a plan for the robot to navigate to the correct floor and room, using the multi-floor navigation domain.	`(at-location rob1 waypoint2_room2.0)` or `(at-location rob1 room1.1)`
`navigate_to_location`	Its plan will contain actions for the robot to navigate to from a specific waypoint to a location on the same floor as the robot. Uses the navigation domain.	`(at-base rob1 doorway1.1_room1.1)` or `(exists (?w - waypoint) (and (at-base ?w rob1) (in-room ?w room1.1) )`

Table 5.6: Description of primitive actions.

Primitive Action	Description
`identify_-required_object`	This ask is planned when there is an unknown PDDL object (e.g., a request), which the robot requires another agent (i.e., human or remote device) to identify. Its Action Executor looks up what type of object should be identified from the knowledge base and displays the relevant options to the agent.
`drive_base`	Calls the move_base ROS ActionLib to allow the robot to navigate the environment. The robot can drive between two locations that are `in-line` or are in the same room. Based on Speck et al.'s work [3].
`pick_up_object`	Allows the robot to pick up an object. The Action Executor will determine the details of how to pick up the specific object.
`hand_over_object`	Enables the robot to give an object (e.g., cup) to another agent (e.g., IoT coffee machine or human).
`open_blinds` `open_window`	For all remaining actions in this table, the robot used in our experiments is not capable of performing them; therefore, the action is executed by the `RemoteActionExecutor`. In our smart home blinds and windows are IoT enabled.
`fill_object`	Fills an object (e.g., cup) with something (e.g., coffee).
`switch_object_on`	Used to plan the switching on of an object (e.g., a night light).
`request_lift`	This action can only be executed when the robot is inside the room with the entrance to the lift. The robot used in our experiments is not capable of pressing the lift button.
`request_floor`	When the robot enters the lift, it request this action is preformed, to get to the correct floor.
`switch_on_room_light`	This action, and the `open_door` action, are added to the navigation domain when required, i.e., when a room the robot is about to drive into is dark.
`open_door`	Enables the robot to open (IoT enabled) doors which are blocking its path.

Appendix 5.B Example of the Executed Actions at Varying Hierarchical Splits

The plans produced by our framework, when provided with the PDDL files for each of the design steps (described in Section 5.6), are displayed in this section. To produce these, the `goal_creator` set the robot's goal to `(is-recharging rob1)`. When all actions are contained within the a single domain file, a derived rule is defined, which states that if the robot is at a charging location (e.g., `waypoint1.1_room1.0`) and all its requests are completed, then `(is-recharging rob1)` is true. When a hierarchy is used, a condition of the `recharge` action is set to `(not (has-incomplete-requests))`; `has-incomplete-requests` is a derived rule that checks all requests are completed. We chose to do this so that when a new request

is sent to the robot, if it is recharging, the `recharge` action is cancelled. This causes a replanning, which results the robot creating a plan to accomplish the new request and return to a charging location. As mentioned in Table 5.1, the `object_state` creator inserts requests into the PDDL problem file.

Appendix 5.B.1 No Splitting

When all action are included in a single domain file (see Figure 5.6) the plan in Listing 5.8 is produced. A precondition of the `switch_object_on` is that the request has been identified as the correct type, i.e., `(is-switch-on-request ?request)`; and one of its effects is set to `(is-completed ?request)`.

Listing 5.8: When planning everything upfront, the planner must make an assumption on the most likely request and plan all actions needed to fulfil that request.

```
1:(drive_base rob1 waypoint1.1_room1.0 doorway1.3_room1.0)
2:(drive_base rob1 doorway1.3_room1.0 doorway1.3_room1.2)
3:(drive_base rob1 doorway1.3_room1.2 doorway1.6_room1.2)
4:(request_lift remote lift0 floor1 floor1 rob1 room1.2 lift0loc)
5:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
6:(request_floor remote lift0 floor2 floor1 rob1 lift0loc)
7:(drive_base rob1 doorway1.6_lift0loc doorway2.6_room2.2)
8:(identify_required_object rob1 human1 request1)
9:(drive_base rob1 doorway2.6_room2.2 doorway1.6_lift0loc)
10:(switch_object_on remote night_light1 request1)
11:(request_floor remote lift0 floor1 floor2 rob1 lift0loc)
12:(drive_base rob1 doorway1.6_lift0loc doorway1.6_room1.2)
13:(drive_base rob1 doorway1.6_room1.2 doorway1.3_room1.2)
14:(drive_base rob1 doorway1.3_room1.2 doorway1.3_room1.0)
15:(drive_base rob1 doorway1.3_room1.0 waypoint1.1_room1.0)
```

Appendix 5.B.2 Splitting up Key Concepts

As show in Figure 5.7, the main tasks the robot can perform can be abstracted, enabling the plans of Listing 5.9 to be produced. When the `discover_request` action is executed, its Composite Action Executor will use the "primitive domain" to decompose this action into primitive actions. After `discover_request` has been executed, `perform_request`, then `recharge`, are executed by also creating plans containing primitive actions. This split result in the robot being less likely to have to replan, and enables the creation of a (high-level) human understandable plan.

Listing 5.9: Executed action when the state and actions are split up by key concepts.

```
1:(discover_request rob1 request1 human1)
 1.1:(drive_base rob1 waypoint1.1_room1.0 doorway1.3_room1.0)
 1.2:(drive_base rob1 doorway1.3_room1.0 doorway1.3_room1.2)
 1.3:(drive_base rob1 doorway1.3_room1.2 doorway1.6_room1.2)
 1.4:(request_lift remote lift0 floor1 floor1 rob1 room1.2 lift0loc)
 1.5:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
 1.6:(request_floor remote lift0 floor2 floor1 rob1 lift0loc)
 1.7:(drive_base rob1 doorway1.6_lift0loc doorway2.6_room2.2)
 1.8:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
 2.1:(switch_object_on remote night_light1)
3:(recharge rob1)
 3.1:(drive_base rob1 doorway2.6_room2.2 doorway1.6_lift0loc)
 3.2:(request_floor remote lift0 floor1 floor2 rob1 lift0loc)
 3.3:(drive_base rob1 doorway1.6_lift0loc doorway1.6_room1.2)
 3.4:(drive_base rob1 doorway1.6_room1.2 doorway1.3_room1.2)
 3.5:(drive_base rob1 doorway1.3_room1.2 doorway1.3_room1.0)
 3.6:(drive_base rob1 doorway1.3_room1.0 waypoint1.1_room1.0)
```

Appendix 5.B.3 Separate Repeated Blocks

In many domains, the robot will be required to navigate the environment. Therefore, as shown in Figure 5.8, navigation-based actions can be split into a separate domain file. The example hierarchical plan is shown in Listing 5.10.

Listing 5.10: Executed action when repeated groups of actions are separated.

```
1:(discover_request rob1 request1 human1)
 1.1:(move_to_object rob1 human1)
  1.1.1:(drive_base rob1 waypoint1.1_room1.0 doorway1.3_room1.0)
  1.1.2:(drive_base rob1 doorway1.3_room1.0 doorway1.3_room1.2)
  1.1.3:(drive_base rob1 doorway1.3_room1.2 doorway1.6_room1.2)
  1.1.4:(request_lift remote lift0 floor1 floor1 rob1 room1.2 lift0loc)
  1.1.5:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
  1.1.6:(request_floor remote lift0 floor2 floor1 rob1 lift0loc)
  1.1.7:(drive_base rob1 doorway1.6_lift0loc doorway2.6_room2.2)
 1.2:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
 2.1:(switch_object_on remote night_light1)
3:(recharge rob1)
 3.1:(drive_base rob1 doorway2.6_room2.2 doorway1.6_lift0loc)
 3.2:(request_floor remote lift0 floor1 floor2 rob1 lift0loc)
 3.3:(drive_base rob1 doorway1.6_lift0loc doorway1.6_room1.2)
 3.4:(drive_base rob1 doorway1.6_room1.2 doorway1.3_room1.2)
 3.5:(drive_base rob1 doorway1.3_room1.2 doorway1.3_room1.0)
 3.6:(drive_base rob1 doorway1.3_room1.0 waypoint1.1_room1.0)
```

Appendix 5.B.4 Reducing Unused Knowledge

Rather that reasoning about the whole environment in detail, this information can be split-up. As shown in Figure 5.9, the "multi-floor navigation domain" enables the robot to plan the high-level steps of its navigation (i.e., request a lift to get to the correct floor and navigate to/from the lift from either its initial location or its desired location), and the "navigation domain"

performs the detailed steps required to navigate a single floor. An example hierarchical plan is shown in Listing 5.11.

Listing 5.11: Executed action when distant objects are split-up. The fully hierarchical approach.

```
1:(discover_request rob1 request1 human1)
 1.1:(move_to_object rob1 human1)
  1.1.1:(navigate_to_location rob1 room1.0 room1.2)
   1.1.1.1:(drive_base rob1 waypoint1.1_room1.0 doorway1.3_r1.0)
   1.1.1.2:(drive_base rob1 doorway1.3_room1.0 doorway1.3_room1.2)
   1.1.1.3:(drive_base rob1 doorway1.3_room1.2 doorway1.6_room1.2)
  1.1.2:(request_lift remote lift0 floor1 floor1 rob1 room1.2 lift0loc)
  1.1.3:(navigate_to_location rob1 room1.2 lift0loc)
   1.1.3.1:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
  1.1.4:(request_floor remote lift0 floor2 floor1 rob1 lift0loc)
  1.1.5:(navigate_to_location rob1 lift0loc room2.2)
   1.1.5.1:(drive_base rob1 doorway1.6_lift0loc doorway2.6_room2.2)
 1.2:(identify_required_object rob1 human1 request1)
2:(perform_request rob1 request1)
 2.1:(switch_object_on remote night_light1)
3:(recharge rob1)
 3.1:(navigate_to_location rob1 lift0loc room2.2)
  3.1.1:(drive_base rob1 doorway2.6_room2.2 doorway1.6_lift0loc)
 3.2:(request_floor remote lift0 floor1 floor2 rob1 lift0loc)
 3.3:(navigate_to_location rob1 lift0loc waypoint1.1_room1.0)
  3.3.1:(drive_base rob1 doorway1.6_lift0loc doorway1.6_room1.2)
  3.3.2:(drive_base rob1 doorway1.6_room1.2 doorway1.3_room1.2)
  3.3.3:(drive_base rob1 doorway1.3_room1.2 doorway1.3_room1.0)
  3.3.4:(drive_base rob1 doorway1.3_room1.0 waypoint1.1_room1.0)
```

Appendix 5.C Plans Produced during the Simulated Coffee Fetching Experiment (Exp. 2)

In the experiments described in Section 5.8.2, a robot is tasked with fetching a coffee for a human. The plans produced when neither coffee machine malfunctions are presented in this appendix. The actions planned when the first primitive action was executed and all executed actions, when our hierarchy was used, are provided in Listings 5.12 and 5.13, respectively. When everything is planned upfront the plan of Listing 5.14 is produced.

Listing 5.12: Action planned before the first primitive (`drive_base`) action was executed. This plan is from the coffee bot experiments, for when hierarchical planning is performed.

```
1:(get_object rob1 cup1)
  1.1:(move_to_object rob1 cup1)
    1.1.1:(navigate_to_location rob1 room1.2 waypoint1.3_room1.2)
      1.1.1.1:(drive_base rob1 waypoint1.2_room1.2 waypoint1.3_room1.2)
  1.2:(pick_up_object rob1 cup1)
2:(go_fill_object rob1 cup1 coffee)
3:(give_object_to_agent rob1 human1)
```

Listing 5.13: Actions executed, for the coffee bot scenario, when hierarchical planning is performed.

```
1:(get_object rob1 cup1)
  1.1:(move_to_object rob1 cup1)
    1.1.1:(navigate_to_location rob1 room1.2 waypoint1.3_room1.2)
      1.1.1.1:(drive_base rob1 waypoint1.2_room1.2 waypoint1.3_room1.2)
  1.2:(pick_up_object rob1 cup1)
2:(go_fill_object rob1 cup1 coffee)
  2.1:(move_to_object rob1 remote)
    2.1.1:(navigate_to_location rob1 waypoint1.3_room1.2 waypoint1.3
        ↪_room1.1)
      2.1.1.1:(drive_base rob1 waypoint1.3_room1.2 doorway1.5_room1.2)
      2.1.1.2:(drive_base rob1 doorway1.5_room1.2 doorway1.5_room1.1)
      2.1.1.2:(drive_base rob1 doorway1.5_room1.1 waypoint1.4_room1.1)
  2.2:(hand_over_object rob1 cup1 remote)
  2.3:(fill_object remote cup1 coffee)
  2.4:(pick_up_object rob1 cup1)
3:(give_object_to_agent rob1 human1)
  3.1:(move_to_object rob1 human1)
    3.1.1:(navigate_to_location rob1 room1.0 room1.2)
      3.1.1.1:(drive_base rob1 waypoint1.4_room1.1 doorway1.5_room1.1)
      3.1.1.2:(drive_base rob1 doorway1.5_room1.1 doorway1.5_room1.2)
    3.1.2:(request_lift remote lift0 floor1 floor1 rob1 room1.2 lift0loc)
    3.1.3:(navigate_to_location rob1 room1.2 lift0loc)
      3.1.3.1:(drive_base rob1 doorway1.5_room1.2 doorway1.6_room1.2)
      3.1.3.2:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
    3.1.4:(request_floor remote lift0 floor2 rob1 lift0loc)
    3.1.5:(navigate_to_location rob1 lift0loc waypoint2.5_room2.1)
      3.1.5.1:(drive_base rob1 doorway2.6_lift0loc doorway2.6_room2.2)
      3.1.5.1:(drive_base rob1 doorway2.6_room2.2 doorway2.5_room2.2)
      3.1.5.1:(drive_base rob1 doorway2.5_room2.2 doorway2.5_room2.1)
      3.1.5.1:(drive_base rob1 doorway2.5_room2.1 waypoint2.5_room2.1)
  3.2:(hand_over_object rob1 cup1 human1)
```

Listing 5.14: Actions executed, for the coffee bot scenario, when all planning is performed upfront.

```
1:(drive_base rob1 waypoint1.2_room1.2 waypoint1.3_room1.2)
2:(pick_up_object rob1 cup1)
3:(drive_base rob1 waypoint1.3_room1.2 doorway1.6_room1.2)
4:(request_lift lift0_device lift0 floor1 floor1 rob1 lift0loc)
5:(drive_base rob1 doorway1.6_room1.2 doorway1.6_lift0loc)
6:(request_floor lift0_device lift0 floor2 rob1 lift0loc)
7:(drive_base rob1 doorway2.6_lift0loc doorway2.6_room2.2)
8:(drive_base rob1 doorway2.6_room2.2 doorway2.5_room2.2)
9:(drive_base rob1 doorway2.5_room2.2 doorway2.6_room2.1)
10:(drive_base rob1 doorway2.6_room2.1 waypoint2.5_room2.1)
11:(hand_over_object rob1 cup1 coffee_machine2 waypoint2.5_room2.1)
12:(fill_object coffee_machine2 cup1 coffee)
13:(pick_up_object rob1 cup1 coffee_machine2 waypoint2.5_room2.1)
14:(drive_base rob1 doorway2.5_room2.1 waypoint2.5_room2.1)
15:(hand_over_object rob1 cup1 human1)
```

References

[1] M. Cashmore and D. Magazzeni. *ICRA 2017 Tutorial on AI Planning for Robotics*, 2017. Accessed: 2018-01-03. Available from: http://kcl-planning.github.io/ROSPlan/tutorials/tutorialICRA2017.

[2] S. Knight, G. Rabideau, S. Chien, B. Engelhardt, and R. Sherwood. *Casper: Space Exploration Through Continuous Planning*. IEEE Intell. Syst., 16(5):70–75, 2001.

[3] D. Speck, C. Dornhege, and W. Burgard. *Shakey 2016 - How Much Does it Take to Redo Shakey the Robot?* IEEE Robotics and Automation Letters, 2(2):1203–1209, 2017.

[4] C. Dornhege and A. Hertle. *Integrated Symbolic Planning in the Tidyup-Robot Project.* In AAAI Spring Symposium: Designing Intelligent Robots, 2013.

[5] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carreraa, N. Palomeras, N. Hurtós, and M. Carrerasa. *ROSPlan: Planning in the Robot Operating System*. In Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'15, pages 333–341. AAAI Press, 2015.

[6] L. Atzori, A. Iera, and G. Morabito. *The Internet of Things: A Survey*. Comput. Netw., 54(15):2787 – 2805, 2010.

[7] O. Vermesan, A. Bröring, E. Tragos, M. Serrano, D. Bacciu, S. Chessa, C. Gallicchio, A. Micheli, M. Dragone, A. Saffiotti, P. Simoens, F. Cavallo, and R. Bahr. *Internet of Robotic Things : Converging Sensing/Actuating, Hypoconnectivity, Artificial Intelligence and IoT Platforms*. In Cognitive Hyperconnected Digital Transformation : Internet of Things Intelligence Evolution, pages 97–155. River Publishers, 2017.

[8] P. Simoens, M. Dragone, and A. Saffiotti. *The Internet of Robotic Things: A Review of Concept, Added Value and Applications*. Int. J. Adv. Robot. Syst., 15(1):1729881418759424, 2018.

[9] P. Papadakis, C. Lohr, M. Lujak, A. Karami, I. Kanellos, G. Lozenguez, and A. Fleury. *System Design for Coordinated Multi-Robot Assistance Deployment in Smart Spaces*. In Second IEEE International Conference on Robotic Computing, IRC, pages 324–329. IEEE, 2018.

[10] D. L. Kovacs. *A Multi-agent Extension of PDDL3.1*. In Proceedings of the Third Workshop on the International Planning Competition (IPC), pages 19–27, 2012.

[11] M. Brenner and B. Nebel. *Continual Planning and Acting in Dynamic Multiagent Environments*. Auton. Agents Multi-Agent Syst., 19(3):297–331, 2009.

[12] J. Bidot and S. Biundo. *Artificial Intelligence Planning for Ambient Environments*. In Next Generation Intelligent Environments, pages 195–225. Springer New York, New York, NY, 2011.

[13] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B. S. Seo, and Y. J. Cho. *The PEIS-Ecology Project: Vision and Results*. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 2329–2335, 2008.

[14] R. Lundh, L. Karlsson, and A. Saffiotti. *Autonomous Functional Configuration of a Network Robot System*. Robot. Auton. Syst., 56(10):819 – 830, 2008. Network Robot Systems.

[15] M. Broxvall, M. Gritti, A. Saffiotti, B.-S. Seo, and Y.-J. Cho. *PEIS Ecology: Integrating Robots into Smart Environments*. In Proceedings of the IEEE International Conference on Robotics and Automation, ICRA, pages 212–218. IEEE, 2006.

[16] J. Buehler and M. Pagnucco. *A Framework for Task Planning in Heterogeneous Multi Robot Systems Based on Robot Capabilities*. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, AAAI'14, pages 2527–2533. AAAI Press, 2014.

[17] N. Seydoux, K. Drira, N. Hernandez, and T. Monteil. *IoT-O, a Core-Domain IoT Ontology to Represent Connected Devices Networks*. In Knowledge Engineering and Knowledge Management, pages 561–576, Cham, 2016. Springer International Publishing.

[18] S. R. Fiorini, J. Bermejo-Alonso, P. Gonçalves, E. P. de Freitas, A. O. Alarcos, J. I. Olszewska, E. Prestes, C. Schlenoff, S. V. Ragavan, S. Redfield, et al. *A Suite of Ontologies for Robotics and Automation [Industrial Activities]*. IEEE Robot. Autom. Mag., 24(1):8–11, 2017.

[19] L. Kunze, T. Roehm, and M. Beetz. *Towards Semantic Robot Description Languages*. In Proceedings of the IEEE International Conference on Robotics and Automation, ICRA, pages 5589–5595. IEEE, 2011.

[20] L. Riazuelo, M. Tenorth, D. Di Marco, M. Salas, D. Gálvez-López, L. Mösenlechner, L. Kunze, M. Beetz, J. D. Tardós, L. Montano, and J. M. M. Montiel. *RoboEarth Semantic Mapping: A Cloud Enabled Knowledge-based Approach*. IEEE Trans. Autom. Sci. Eng, 12(2):432–443, 2015.

[21] C. Galindo, J.-A. Fernández-Madrigal, J. González, and A. Saffiotti. *Robot Task Planning Using Semantic Maps*. Robot. Auton. Syst., 56(11):955 – 966, 2008. Semantic Knowledge in Robotics.

[22] U. Köckemann, M. Alirezaie, L. Karlsson, and A. Loutfi. *Integrating Ontologies for Context-based Constraint-based Planning*. In Tenth International Workshop on Modelling and Reasoning in Context (IJCAI-MRC), pages 22–29, 2018.

[23] M. Helmert. *The Fast Downward Planning System*. J. Artif. Intell. Res., 26:191–246, 2006.

[24] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel. *Semantic Attachments for Domain-independent Planning Systems*. In Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'09, pages 114–121. AAAI Press, 2009.

[25] R. Lallement, L. De Silva, and R. Alami. *HATP: An HTN Planner for Robotics*. In Proceedings of the Second Workshop on Planning and Robotics (PlanRob), 2014.

[26] K. Myers, S. F. Smith, D. W. Hildum, P. A. Jarvis, and R. de Lacaze. *Integrating Planning and Scheduling Through Adaptation of Resource Intensity Estimates*. In Proceedings of the Sixth European Conference on Planning, 2014.

[27] R. Sukkerd, J. Cámara, D. Garlan, and R. Simmons. *Multiscale Time Abstractions for Long-range Planning Under Uncertainty*. In Proceedings of the Second International Workshop on Software Engineering for Smart Cyber-Physical Systems, SEsCPS '16, pages 15–21, New York, NY, USA, 2016. ACM.

[28] L. P. Kaelbling and T. Lozano-Pérez. *Hierarchical Task and Motion Planning in the Now*. In Proceedings of the IEEE International Conference on Robotics and Automation, ICRA, pages 1470–1477. IEEE, 2011.

[29] L. P. Kaelbling and T. Lozano-Pérez. *Implicit Belief-Space Pre-images for Hierarchical Planning and Execution*. In Proceedings of the IEEE International Conference on Robotics and Automation, ICRA, pages 5455–5462. IEEE, 2016.

[30] N. Ramoly, A. Bouzeghoub, and B. Finance. *Context-aware Planning by Refinement for Personal Robots in Smart Homes*. In Forty-Seventh International Symposium on Robotics (ISR), pages 1–8. VDE, 2016.

[31] M. Martínez, F. Fernández, and D. Borrajo. *Planning and Execution Through Variable Resolution Planning*. Robot. Auton. Syst., 83:214 – 230, 2016.

[32] A. S. Rao, M. P. Georgeff, et al. *BDI Agents: From Theory to Practice.* In First International Conference on Multiagent Systems, volume 95 of *ICMAS*, pages 312–319. AAAI Press, 1995.

[33] M. C. Mora, J. G. Lopes, R. M. Viccariz, and H. Coelho. *BDI Models and Systems: Reducing the Gap*. In Intelligent Agents V: Agents Theories, Architectures, and Languages, pages 11–27, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[34] A. Pokahr, L. Braubach, and W. Lamersdorf. *A Goal Deliberation Strategy for BDI Agent Systems*. In Multiagent System Technologies, pages 82–93, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[35] F. Meneguzzi and L. De Silva. *Planning in BDI Agents: A Survey of the Integration of Planning Algorithms and Agent Reasoning*. The Knowledge Engineering Review, 30(1):1–44, 2015.

[36] I. Georgievski and M. Aiello. *HTN Planning: Overview, Comparison, and Beyond*. Artif. Intell., 222:124 – 156, 2015.

[37] R. C. Cardoso and R. H. Bordini. *A Multi-agent Extension of a Hierarchical Task Network Planning Formalism*. Advances in Distributed Computing and Artificial Intelligence Journal (ADCAIJ), 6(2), 2017.

[38] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. *SHOP2: An HTN Planning System*. J. Artif. Intell. Res., 20:379–404, 2003.

[39] D. Höller, P. Bercher, G. Behnke, and S. Biundo. *A Generic Method to Guide HTN Progression Search with Classical Heuristics*. In Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS'18. AAAI Press, 2018.

[40] H. Vandaele, J. Nelis, T. Verbelen, and C. Develder. *Remote Management of a Large Set of Heterogeneous Devices Using Existing IoT Interoperability Platforms*. In Internet of Things. IoT Infrastructures, pages 450–461, Cham, 2016. Springer International Publishing.

[41] W. Chamberlain, J. Leitner, T. Drummond, and P. Corke. *A Distributed Robotic Vision Service*. In Proceedings of the IEEE International Conference on Robotics and Automation, ICRA, pages 2494–2499. IEEE, 2016.

[42] D. Sprute, A. Pörtner, R. Rasch, S. Battermann, and M. König. *Ambient Assisted Robot Object Search*. In Enhanced Quality of Life and Smart Living, pages 112–123, Cham, 2017. Springer International Publishing.

[43] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. *ROS: an open-source Robot Operating System*. In ICRA Workshop on Open Source Software, volume 3, page 5, 2009.

[44] J. Buehler and M. Pagnucco. *Planning and Execution of Robot Tasks Based on a Platform-independent Model of Robot Capabilities*. In Proceedings of the Twenty-first European Conference on Artificial Intelligence, ECAI'14, pages 171–176, Amsterdam, The Netherlands, 2014. IOS Press.

[45] A. Hornung, S. Böttcher, J. Schlagenhauf, C. Dornhege, A. Hertle, and M. Bennewitz. *Mobile Manipulation in Cluttered Environments with Humanoids: Integrated Perception, Task Planning, and Action Execution*. In IEEE-RAS International Conference on Humanoid Robots, pages 773–778. IEEE, 2014.

[46] P. Eyerich, R. Mattmüller, and G. Röger. *Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning*. In Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'09, pages 130–137. AAAI Press, 2009.

[47] N. Ye, A. Somani, D. Hsu, and W. S. Lee. *DESPOT: Online POMDP Planning with Regularization*. J. Artif. Intell. Res., 58:231–266, 2017.

6

Learning Symbolic Action Definitions

"*If making stacks of blocks seems insignificant – remember that you didn't always feel that way.*"
Marvin Minsky (1988)

Learning Symbolic Action Definitions from Unlabelled Image Pairs

H. Harman & P. Simoens

Accepted in International Conference on Advancements in Artificial Intelligence (ICAAI), 2020.

To evaluate the algorithms described in the previous chapters, the symbolic definition of the agents' behaviour was manually developed. This can be a time consuming and error prone process. Therefore, the work presented in this chapter transforms unlabelled pairs of images, namely, transitions, into reusable action definitions. These transitions show the state before and after an action has been executed. Each action definition consist of a set of parameters, effects and preconditions. During the experiments, the action definitions for Puzzle, Lights-Out and Towers-of-Hanoi domains were generated by our approach. To evaluate the learnt action definitions, initial and goal states were generated and a task planner invoked. Problems with large state spaces were solved using the action definitions learnt from smaller state spaces. On average the task plans contained 5.46 actions and planning took 0.06 seconds. Moreover, when 20 % of transitions were missing, our approach generated the correct number of objects, action definitions and plans 70 % of the time. Further research is required to apply the work presented in this chapter to the domains presented in the earlier chapters.

6.1 Introduction

Symbolic models of an agent's behaviour are required by tasks planners and goal/plan recognisers. To achieve a given goal, task planners [1, 2] find a sequence of discrete actions. In contrast, plan and goal recognisers [3–5] attempt to infer an observed agent's goal and/or plan from a sequence of observations.

Rather than enumerating all possible actions an agent can perform, symbolic action definitions are developed. The signature of an action definition consists of a name and a parameter list, and its body contains preconditions and effects. Action definitions are grounded (i.e., transformed into actions) by providing objects as arguments. The aim of our work is to automatically generate generic (i.e., reusable) action definitions from unlabelled pairs of images, namely, transitions, which show the state before and after an action has been executed.

Symbolic action definitions are usually manually written. This can be a

burdensome task and often requires domain specific knowledge [2, 6]. Therefore, methods that attempt to learn the preconditions and effects of actions have been proposed [7, 8] but, until recently, some symbolic knowledge was still necessary. Transitions were provided to LatPlan [9], which employed a deep autoencoder to generate the action definitions. Nevertheless, to generate symbolic actions (i.e., PDDL), all transitions were required, and planning took a considerable length of time. Moreover, deep-learning approaches often have a lengthy training time, and the decisions made by the algorithms are likely to be unexplainable. Therefore, as mention in the introductory chapter, we aim to address the following research question: Can a more compact set of symbolic action definitions be learnt from pairs of images by taking a explainable (white-box) approach?

In this chapter, objects are discovered by finding pixels that always change value simultaneously; both the location of these pixels and their values are defined as objects. Subsequently, the transitions are transformed into actions by generating predecessor and successor states from the images. These states enable the actions' preconditions and effects to be determined. Actions are then converted into a set of action definitions, which can be provided to task planners and goal/plan recognisers. Our learnt action definitions are evaluated by calling a task planner on a generated initial and goal state.

Seemingly simple problems can contain thousands of transitions. For instance, a 3 by 3 puzzle problem (see Figure 6.1) contains 362 880 states and 967 680 transitions. Therefore, we do not assume that all transitions are present in the image dataset. As a result, because there is no way to determine if an unseen transition is invalid or just missing, there are likely to be domains our method does not create valid action definitions for. Nevertheless, the decisions made by our method are explainable, and thus to handle these cases alterations can be made to our state and precondition generation algorithms. Like images, many other sources of sensed data simply contain locations with values; thus, the presented processes can be applied to alternative data sources. For simplicity, all objects (within an image) are assumed to be rectangular, no objects are occluded, and images contain no noise.

6.2 Problem Formalisation

Task planning and goal recognition problems are often defined in Planning Domain Definition Language (PDDL) [10]. A PDDL defined problem includes objects, predicates, action definitions, and states. Our approach takes a set of transitions as input and transforms them into PDDL. This

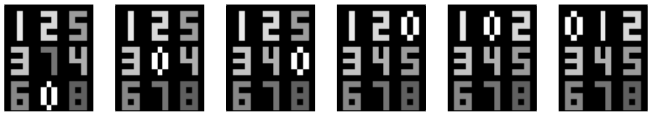


Figure 6.1: Example plan for solving a Puzzle problem. The initial state is on the left and the goal is on the right. Tiles can only be swapped with the clear tile (i.e., the 0) if they are adjacent to it. All example plans were produced using the action definitions created by our approach. The transitions, which were provided as input, are based on the work of Asai and Fukunaga [9].

section provides a definition for a transition and for the components of a PDDL defined problem. Subsequently, the example domains are introduced.

Definition 11. *Transitions:* A transition ($t \in T$) is a pair of unlabelled images, i.e., a predecessor image (t_{pi}) and a successor image (t_{si}).

Definition 12. *Objects:* There are two types of objects in our system, *locations* (L) and *image objects* (I). Locations are the areas of an image which can change value. The values these locations can be set to, are called image objects. A location can also, optionally, have a *clear* value, which indicates that no image object is at that location. An image object's value is a location. We use the term "value" as "state" describes the entire environment's state (i.e., all object's values).

Definition 13. *Predicates and Atoms:* A predicate consist of a name and a typed parameter list. For instance, when grounded, the `(at ?1 - location ?2 - image_object)` predicate indicates an image object is at a location. An atom is a grounded predicate, and can be fluent or static.

Definition 14. *States:* A state consists of fluent atoms, which represent all objects' values.

Definition 15. *Actions:* Each action ($a \in A$) can be cast to a transition ($t \in T$), and vice versa. An action modifies a subset of the fluents within a predecessor state (a_{ps}) to reach a successor state (a_{ss}). It is comprised of a name, arguments, preconditions (a_{pre}) and effects (a_{eff}). Preconditions include known preconditions, containing the fluent atoms the action modifies, and possible preconditions, which include both static and fluent atoms. Effects contain fluent atoms, denoting the modified objects' value after the action has been executed.

Definition 16. *Action definitions:* An action definition is an ungrounded action.

We aim to automatically generate a set of generic action definitions that can be provided to task planners, along with an initial and goal state.

Throughout this chapter, examples from a Towers of Hanoi (ToH) domain are provided. In this domain, there are three towers and four discs of differing sizes. Larger discs cannot be placed on smaller discs. An example is provided in Figure 6.2. Our experimental results also show our approach successfully learns the action definitions for Puzzle domains (e.g., Figure 6.1) and Lights-Out domains (e.g., Figure 6.3). For all domains, the transitions were created from the work of Asai and Fukunaga [9].

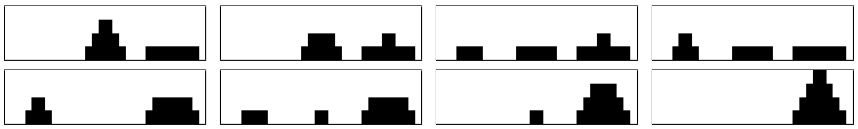


Figure 6.2: Example of a plan for solving a ToH problem. The initial state is shown in the top-left and the goal state in the bottom-right.

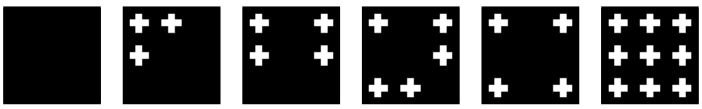


Figure 6.3: Example plan for solving a Lights-Out problem. When a location is pressed, it and its vertically and horizontally adjacent locations change value (i.e., switch on/off).

6.3 Discover Objects

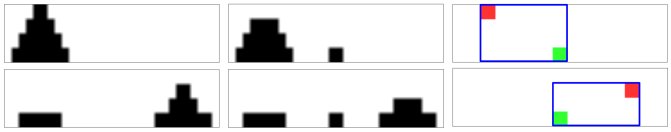
Our system starts by discovering the locations (L) and image objects (I). Although we do not assume the set of transitions is complete, for this step to work the transitions must include multiple instances of each location changing state and each image object (e.g., disc) must change state at least once. All objects are provide with a unique ID to enable them to be transformed to/from PDDL.

6.3.1 Discover Locations

A location has a (x, y) position, a width and a height, and encompasses a group of pixels that change values simultaneously. The algorithm that discovers the locations is described below.

Each transition is processed in turn. The area of the images that changes is discovered by finding the minimum and maximum x, and minimum and maximum y of the pixels that change value ($\forall(x, y) : t_{si}[x, y] - t_{pi}[x, y] \neq 0$). This area is then checked to see if it overlaps any of the previously

discovered areas. If so, the overlapping area is taken and shrank to cover just the pixels (within the overlap) that change value (e.g., as depicted in Figure 6.4). These areas, as well as the original area, are appended onto the list of discovered areas.



(a) Predecessor (left) and successor (middle) images of two transitions. The difference between the predecessor and successor is represented in the right images; red represents positive pixel values and green shows negative values. The blue box incorporates all pixels whose value is non-zero, i.e., whose value has changed.

(b) The changed image area, shared by the two transitions.

Figure 6.4: Two transitions and their overlapping changed image area.

The discovered areas are then iterated over, starting with the smallest, to create the set of locations (L). If the image area does not overlap an already created location ($l \in L$), it is inserted into the set of locations. If the image area only overlaps a single previously created location ($l' \in L$) and it fully encompasses that location, it is appended to the set of locations and the location (l') is removed. For the ToH domain, this procedure results in the locations depicted in Figure 6.5.

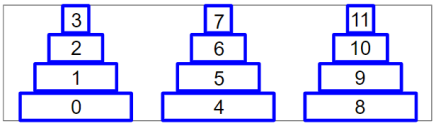


Figure 6.5: The discovered location definitions for a ToH domain.

The resulting locations are provided with a list of possible values. If a location always transitions to/from a certain value, that value becomes its *clear* value; `clear` is the name of a predicate. If a location only ever has one of two values, one of those values is randomly selected as the clear value. The locations' remaining possible values are image objects.

6.3.2 Discover Image Objects

Image objects are extracted from the transitions. For each transition, the locations which encompass the pixels that change are discovered (e.g., Figure 6.6). If a location's value matches its clear value, it is ignored. Oth-

erwise, the image is cropped to the changed pixels, contained within the location. These pixels make up an image object. If a location's centre and an image object's centre do not match, the location will also contain an offset (per image object). For the example ToH domain, this results in 4 image objects (i.e., black rectangles) being discovered.

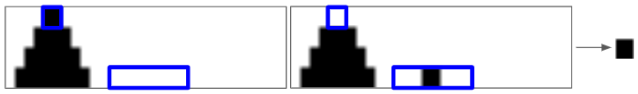


Figure 6.6: Predecessor (left) and successor (middle) images of a transition. Blue boxes indicate the locations that encompass the changed pixels. The discovered image object is shown on the right; duplicate image objects are ignored as they are equivalent.

6.4 Generate States

This section describes how a state (i.e., set of fluent atoms) is generated from an image. Further to performing this on the predecessor and successor images of the transitions, the initial and goal image of a planning problem, and the observations from a goal/plan recognition problem, can be transformed into PDDL states by running this process.

Fluent atoms are created from an image by iterating over the location definitions, and determining which image object is at that location. To prevent the smallest image object from matching all occupied locations, image objects are sorted by area, largest first. This process results in a state containing groundings of the predicates `(at ?1 - location ?2 - image_object)` and `(clear ?1 - location)`.

6.5 Generate Action Definitions

The predecessor (a_{ps}) and successor (a_{ss}) states of an action are generated by the process described in the previous section. From these states each action's effects (a_{eff}) and preconditions (a_{pre}) are determined. Preconditions include known preconditions and possible preconditions. The known preconditions are the fluents that change value when the action is executed; and the effects are their resulting values. All fluent atoms not in the set of known preconditions are set as the possible preconditions. The use of possible preconditions was inspired by the works on goal recognition and planning in incomplete domains [2, 5]. Before transforming actions into action definitions, static atoms are created and the set of possible preconditions is reduced. These static atoms prevent invalid actions being produced from the resulting action definitions.

6.5.1 Linked Locations

The first static atoms to be created link together the locations that change value simultaneously. For instance, in the transition depicted in Figure 6.6, locations `loc_3` and `loc_4` change value. Therefore, these locations are linked by a static atom, i.e., `(linked_2 loc_3 loc_4)`. The index within the atom name indicates the number of objects; it is used because predicates have a fixed sized parameter list and differing numbers of locations can change value simultaneously. This atom is appended to the corresponding action's preconditions. Creating these atoms prevents invalid actions being generated from the action definitions produced for domains in which only certain locations can change value simultaneously.

6.5.2 Finding Locations' Dependencies

A location's ability to change value could depend on the state of another object (i.e., location or image object). For instance, in the ToH problem (Figure 6.5), for `loc_2` to change state, `loc_1` must be occupied and `loc_3` must be clear. Therefore, `loc_2` depends on `loc_1` and `loc_3`, i.e., the static atom `(depends_on_3 loc_2 loc_1 loc_3)` is required. This section details how this atom is created.

For each location ($l \in L$), the actions that result in each of its possible values are found. If a location always transitions between clear and another value, for this process only, the location has two possible values, clear and unclear. From the actions which set the same value for a location (l^v), e.g., all the actions that set `loc_4` to clear, the common possible preconditions are extracted. These extracted atoms become the preconditions of the location (l^v_{pre}).

The location's preconditions are reduced by removing the preconditions that are preconditions of other preconditions. For each precondition ($p \in l^v_{pre}$), the actions that set the precondition are discovered (e.g., left side of Figure 6.7). The intersection of these actions' predecessor atoms and the location's preconditions (l^v_{pre}) is taken (e.g., right side of Figure 6.7). The resulting atoms of the, equally, largest sets are removed from the location's preconditions (l^v_{pre}); the precondition (p) these belong to is not removed. The arguments of the remaining preconditions become location's dependencies (D^l), and thus the arguments of a `depends_on` atom. These atoms are appended to the actions', which set the location's (l's) value, possible preconditions. Note, locations can depend on image objects as well as other locations, e.g., when `loc_3` changes state `image_14` is always at `loc_2`, thus `loc_3` depends on `loc_2` and `image_14`.

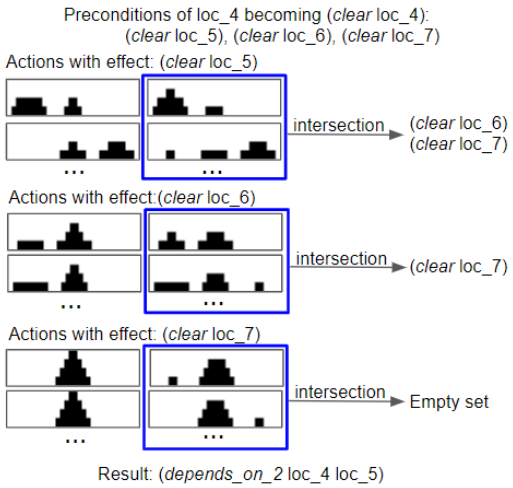


Figure 6.7: Whenever `loc_4` becomes clear, `(clear loc_5)`, `(clear loc_6)` and `(clear loc_7)` are true. For each of these atoms, the actions whose effects include the atoms are shown on the right; actions are depicted by (horizontally adjacent) predecessor and successor images. Taking the intersection of `loc_4`'s preconditions and the predecessor states of actions with `(clear loc_5)` as an effect, results in `[(clear loc_6), (clear loc_7)]`. As this is the largest set of resulting atoms (see right column), `(clear loc_6)` and `(clear loc_7)` are removed from `loc_4`'s preconditions; and `(clear loc_5)` is set as irremovable. Thus, loc_4 is said to depend on `loc_5`, i.e., `(depends_on_2 loc_4 loc_5)`.

6.5.3 Finding Image Objects' Dependencies

Image objects also require dependencies to, for example, prevent a disc from being placed on a smaller disc. The process described in this section, is performed on each image object.

The actions ($A^i \subseteq A$) that change the image object's (i) value are discovered and for each of these actions ($a \in A^i$), a set of dependencies (D^i) is created. The locations ($L^c \subseteq L$) that change state (i.e., locations mentioned in a_{eff}) are extracted and, for each location ($l \in L^c$), their dependencies (D^l) are iterated over. The image objects that are either at the location of a dependency ($d \in D^l$) or are a dependency ($d \in D^l$) themselves are inserted into the new dependency set (D^i). If none of the location's dependencies (D^l) have an image object (i.e., they are all clear), then the location (l) itself is inserted into the dependency set (D^i). After processing all locations that change state, a `depends_on` atom is created from D^i. The arguments of the `depends_on` atom are the image object (i) itself, followed by, the set of dependencies (D^i). This atom is inserted in the action's (a's) possible

preconditions. An example is provided in Figure 6.8.

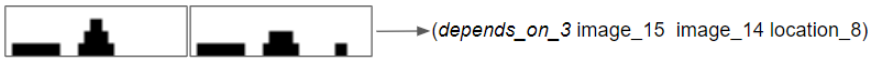


Figure 6.8: In this transition, `image_15` moves from `loc_6` to `loc_8`. `loc_6` depends on `loc_5` and `loc_7`. As `image_14` is at `loc_5` and `loc_7` is clear, `image_14` is appended to image object's dependencies (D^i). `loc_8` depends on `loc_9`, and as `loc_9` is empty, `loc_8` is append to D^i.

6.5.4 Remove Unrequired Possible Preconditions

So far the preconditions of the actions have not been reduced, and thus contain the entire state space. An action ($a \in A$) only requires, as its preconditions (a_{pre}), the atoms of the objects that change value and the atoms of the objects those objects depend on. Therefore, all other fluent atoms are removed from the action's possible preconditions. Possible preconditions are used because when transitions are missing, additional `depends_on` atoms could be created, and thus appear in the possible preconditions (along with objects' fluent atoms). In future work, this set of possible preconditions can be provided to goal recognisers [5] and task planners [2] designed for incomplete domains.

6.5.5 Generate Action Definitions

Action definitions are generated from the set of actions (A). These are generated by iterating over all actions, checking if a grounding of an already generated action definition is equivalent to that action (i.e., same arguments, preconditions and effects), and if not, creating a new action definition. A new action definition is created by ungrounding the action, i.e., replacing actual objects (e.g., `loc_1`, `image_12`) with typed parameters (e.g., `?l - location`, `?i - image_object`).

6.6 Handling Large State Spaces

For large state spaces, we propose generating the action definitions from a smaller state space (e.g., a 2 by 2 puzzle). Then, using the knowledge gained, create the objects and static atoms of a larger state space (e.g., a 3 by 3 puzzle) from the transitions applicable to a single state. This is currently only possible for domains containing objects of equal size and whose adjacent locations are linked. If these conditions are not met, the method described in the previous sections is invoked.

The size of the objects is discovered by finding the smallest overlapping area of a change. This is performed using the already described process but only the transitions from a single state are processed. To create all locations (L), the algorithm iterates over the pixels of a single image, using the

size of the objects plus the gap between objects as a step size. The value of these locations (in the image), are set as the image objects (I). Note, the gap between locations is set using the number of objects in the linked atoms and the size of the changed image area. Static atoms are then created. If in the original problem the `linked` atoms contain two objects, then for every location a `linked` atom is created for each of its adjacent locations (e.g., `(linked_2 loc_0 loc_1) (linked_2 loc_0 loc_3)`). Otherwise, a single `linked` atom is created for each location, to link it to all its adjacent locations (e.g., `(linked_3 loc_0 loc_1 loc_3)`).

6.7 Experiments and Discussion

This section discusses the action definitions that were produced by our approach, the time taken to generate the actions definitions, the time it takes a task planner to find a plan and how missing transitions affected the production of PDDL. Experiments were ran on server with Intel Xeon 2.27 GHz CPU and 11 GB of RAM.

6.7.1 Action Definitions

This section describes the action definitions produced for the ToH, Puzzle and Lights-Out domains and mentions the minimum number of transitions and state space required to produces these action definitions. The full sets of action definitions are available at: https://doi.org/10.5281/zenodo.3595871.

6.7.1.1 Towers of Hanoi Domains

For the ToH domains containing 3 or more discs, 6 action definitions were generated. One for when the two modified locations are within the middle of the towers (8 parameters); two for when one modified location is at the bottom of the image and the other is in the middle (i.e., a block being moved to a bottom location and a block being moved from a bottom location) (6 parameters); one for when both locations are at the bottom (4 parameters); and two to handle a block being moved to/from a top location (5 parameters). To create these action definitions, a minimum of 3 towers and 3 discs is required. Due to the complexity of this domain (i.e., many `depends_on` atoms being required), nearly all transitions must be provided as input.

6.7.1.2 Puzzle Domains

For the Digital, Mandrill and Spider puzzle domains, a single correct action definition was created. The minimum Puzzle size to generate this action definition is 2 by 2. A 1 by 2 Puzzle results in the creation of unnecessary `depends_on` atoms.

For a 3 by 3 puzzle domain, only 12 transitions are required as input; however, these 12 transitions must be carefully selected. Each location must be acted on at least twice; the location must either transition to clear or not-clear in both transitions and each of the remaining locations must have a different value for both transitions. The likelihood of 12 randomly selected transitions meeting these requirements is less than 2 %.

6.7.1.3 Lights-Out Domains

For Lights-Out, 4 action definitions were generated for a 2 by 2 domain and 15 actions were generated from a 3 by 3, or larger, domain. In these domains, either 3, 4 or 5 locations can change state simultaneously; the action definitions include (for each of these values) handling when no lights are on, when 1 light is on, when 2 are on, etc. The action definitions of a 3 by 3 Lights-Out domain can be generated from 21 selected transitions.

6.7.2 Time Complexity

The most time complex aspects of our implementation are: discovering the locations and creating the locations' `depends_on` atoms. Discovering locations has a time complexity of $O(n^2)$ and creating the locations' dependencies is $O(lno)$; where $n = |T|$, $l = |L|$ and $o = |L| + |I|$. Increasing the number of image pixels also impacts the location discovery time. For instance, to generate the action definition of a Puzzle problem with 10*12 pixels it takes 0.64 seconds, whereas 43*43 pixels takes 9.83 seconds. The action definition generations times for various domains are provided in Table 6.1. All times are the average of 5 runs and are in seconds. For the 3 by 3 Digital and Lights-Out domains the minimum number of transitions are provided as input because processing all transitions, i.e., 967 680 transitions for Digital and 4608 for Lights-Out, would be computationally expensive.

Table 6.1: Action definition generation results. $|T|$ is the number of transitions that were provided as input, $|L|$ the number of discovered locations, $|I|$ the number image objects and $|A^D|$ the number of generated action definitions.

Domain	Config	Resolution	$\|T\|$	$\|L\|$	$\|I\|$	$\|A^D\|$	Avg Time (s)
Digital	2 by 2	10*12	48	4	3	1	0.64 ± 0.00
Digital	3 by 3	15*18	*12*	9	8	1	0.37 ± 0.01
Mandrill	2 by 2	43*43	48	4	3	1	9.83 ± 0.06
Spider	2 by 2	57*57	48	4	3	1	17.15 ± 0.10
Lights-Out	2 by 2	10*10	64	4	1	4	1.51 ± 0.00
Lights-Out	3 by 3	15*15	*21*	9	1	15	1.47 ± 0.02
ToH	2 discs	36*8	24	6	2	3	0.89 ± 0.00
ToH	3 discs	48*12	78	9	3	6	12.04 ± 0.05
ToH	4 discs	60*16	240	12	4	6	115.38 ± 1.55
ToH	5 discs	72*20	726	15	5	6	1012.99 ± 6.08

After using a 2 by 2 Digital puzzle to generate the actions definitions, the objects and static atoms of 3 by 3 Puzzle (i.e., Digital, Mandrill or Spider) domains were discovered in an average of 1.25 seconds. For Lights-Out, the objects and static atoms of a 4 by 4 domain were created in 4.48 seconds.

6.7.3 Task Planning Results: No Missing Transitions

During the experiments, initial and goal states were generated from two images and these states, along with the static atoms, objects and action definitions, were provided to a task planner, i.e., Fast Downward (FD) [1]. 100 goal and initial images were selected at random from a set of all possible images. No goal image and initial image pairing is repeated, and the problem could be unsolvable (i.e., the goal unreachable). Each experiment was ran 5 times; the average total time (and standard deviation), reported by FD, is provided in Table 6.2.

The longest time it took FD to generate a plan was 0.71 seconds (on average 0.06 seconds were taken). The state of the art approach, LatPlan [9, 11], reported that it took 4 hours for an adapted version of FD to solve a generated PDDL problem. Thus, the PDDL produced by our approach is more efficient.

Table 6.2: Time spent planning. $|probs|$ is the number of problems (i.e., goal and initial images) and $\overline{|A^P|}$ is the average number of actions in the produced plans. In the two disc ToH domain, there are only 72 unique problems.

Domain	**Config**	$\|probs\|$	$\overline{\|A^P\|}$	**Avg Time (s)**	**Total Time (s)**
Digital	2 by 2	100	1.44 ± 1.93	0.00 ± 0.00	0.35
Digital	3 by 3	100	11.22 ± 11.17	0.47 ± 0.23	46.87
Mandrill	2 by 2	100	1.49 ± 1.89	0.00 ± 0.00	0.35
Spider	2 by 2	100	1.48 ± 1.89	0.00 ± 0.00	0.35
Lights-Out	2 by 2	100	2.11 ± 0.92	0.00 ± 0.00	0.42
Lights-Out	3 by 3	100	4.59 ± 1.50	0.02 ± 0.00	2.39
Lights-Out	4 by 4	100	0.12 ± 0.68	0.24 ± 0.03	23.66
ToH	2 discs	72	2.00 ± 0.82	0.00 ± 0.00	0.24
ToH	3 discs	100	4.19 ± 2.01	0.00 ± 0.00	0.47
ToH	4 discs	100	8.57 ± 3.99	0.01 ± 0.00	0.96
ToH	5 discs	100	16.52 ± 6.92	0.02 ± 0.00	2.32

After finding a plan, the planned actions were translated into images. The first actions effects were applied to the initial image, then the subsequent action's effects were applied to the resulting image and so on. `(clear ?location)` effects were applied to the image by setting the pixels, corresponding to the location, to the location's clear state. `(at ?location ?image -object)` effects were applied by setting the corresponding pixels to the corresponding image object. For each experiment, this took less than 0.01 second.

6.7.4 Task Planning Results: Missing Transitions

Experiments were ran for various percents of missing transitions. For each percentage, 5 sets of transitions were randomly selected and as before, 100 goal and initial state images were selected. The results are presented in Table 6.3.

Table 6.3: Accuracy for identifying locations (L_c), image objects (I_c), action definitions (A_c^D) and the plan when differing percentages of transitions are missing (m%). C is the ratio of times a plan was successfully created. A_c^P shows how often a successfully created plan's length matches the actual plan's length and D is the average length difference.

Domain	**Config**	m%	L_c	I_c	A_c^D	C	A_c^P	D
Puzzle	2 by 2	70	1.00	1.00	1.00	1.00	1.00	0.00
		80	0.80	0.80	0.80	0.53	1.00	0.00
		90	0.13	0.13	0.13	0.04	1.00	0.00
Lights-Out	2 by 2	20	0.60	1.00	0.60	0.97	0.82	-0.21
		40	0.00	1.00	0.00	0.93	0.53	-0.56
		60	0.40	1.00	0.40	0.96	0.72	-0.26
		80	0.40	1.00	0.60	0.95	0.81	-0.23
ToH	5 discs	20	0.60	1.00	1.00	1.00	0.93	1.08
		40	0.20	1.00	0.60	1.00	0.74	1.53
		60	0.00	0.20	0.00	0.79	0.23	-4.73
		80	0.00	0.60	0.00	0.31	0.27	4.01

For Puzzle problems, when $\leq$70 % of transitions were missing, the correct action definitions and plans were generated. At 80 % the performance greatly deteriorated; plans were only generated 53 % of the time. Nevertheless, when the plan was generated, it was the same length as the plan produced when no transitions are missing.

When 20 % of transitions were missing, Lights-Out was unsolvable 3 % of the time and for 18 % of solved problems, the plan was an incorrect length. Most incorrect plans were shorter than the actual plan length. This is because some locations were not identified, and thus not included within the initial and goal states. For ToH, the discovered plan was often longer because additional `depends_on` atoms (and action definitions) were generated.

6.8 Related Work

Many prior works [7, 12–17] have attempted to learn the actions' preconditions and effects when provided with some symbolic knowledge. In ARMS [7], the initial state, goal state, predicates and a plan, containing actions along with the objects that change state (i.e., the action signature), are provided as input. Their work attempts to guess the action model (i.e., the actions' preconditions and effects) that best matches the plan. Similarly, SLAF [14] enables the actions effects and preconditions to be discovered from a sequence of actions and partially observable states; and the action

model is updated online. LOCM [15] also takes plans (and partial plans) as input but does not require the predicates, initial state or goal state. Nevertheless, all these approaches require a domain engineer to provide some symbolic information.

Previous methods of transforming unlabelled image pairs into symbolic models, such as LatPlan [9] and the work of Amado et al. [18], involved an deep autoencoder and, for the production of PDDL, required all transitions. Both AMA_1 [9] and [18] perform a bitwise comparison of pairs of encoded images to determine the actions' effects. They do not attempt to determine what objects are present. Moreover, training an deep autoencoder can be computationally expensive, and when action definitions are produced from the actions, they are likely to be problem (as well as domain) specific, i.e., only work on problems containing the same objects.

LatPlan's AMA_2 [9] only required a subset of transitions; however, does not produce PDDL, and thus was incompatible with existing planners. Moreover, during the training phase, AMA_2 requires examples of (possibly) invalid states. LatPlan's AMA_1 has also been expanded to work with learnt predicates [19]. This greatly reduced the planning time; however, the objects the images contain were assumed to be known.

6.9 Conclusion

Our work transforms unlabelled pairs of images into PDDL. Locations and image objects are discovered by finding the pixels that change value simultaneously, which enables the images to be transformed into symbolically represented states. Actions, containing predecessor and successor states, are generated from transitions. The atoms of the predecessor and successor states that change are set as the known preconditions and effects of an action. All other atoms form the action's set of possible preconditions. Subsequently, static atoms are created, including atoms that link together locations that change state simultaneously, and atoms that express the locations' and image objects' dependencies. These static atoms are used to reduce the actions' possible preconditions. Actions are then converted into action definitions, which can be processed by goal/plan recognisers and task planners. The produced action definitions for Puzzle, Lights-Out and ToH domains were evaluated by calling a task planner.

In future work, we will evaluate the produced PDDL using goal/plan recognition approaches. Moreover, how well goal recognition design approaches [20–22] perform on our learnt PDDL will be assessed; these methods modify the PDDL to reduce the number of observations required to determine the observee's goal. We also intend to enhance the image pro-

cessing method currently integrated into our approach. Rather than cropping all objects into rectangles, a higher resolution boundary should be created. Furthermore, processing video rather than pairs of images will be investigated.

References

[1] M. Helmert. *The fast downward planning system*. J. Artif. Intell. Res., 26:191–246, 2006.

[2] C. Weber and D. Bryce. *Planning and Acting in Incomplete Domains*. In Proceedings of the Twenty-First International Conference on International Conference on Automated Planning and Scheduling, ICAPS'11, pages 274–281. AAAI Press, 2011.

[3] H. A. Kautz and J. F. Allen. *Generalized Plan Recognition*. In Proceedings of the Fifth AAAI National Conference on Artificial Intelligence, AAAI'86, pages 32–37. AAAI Press, 1986.

[4] M. Ramírez and H. Geffner. *Probabilistic Plan Recognition Using Off-the-shelf Classical Planners*. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI'10, pages 1121–1126. AAAI Press, 2010.

[5] R. F. Pereira, A. G. Pereira, and F. Meneguzzi. *Landmark-Enhanced Heuristics for Goal Recognition in Incomplete Domain Models*. In Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS'19, pages 329–337. AAAI Press, 2019.

[6] S. Kambhampati. *Model-lite Planning for the Web Age Masses: The Challenges of Planning with Incomplete and Evolving Domain Models*. In Proceedings of the Twenty-Second National Conference on Artificial Intelligence, volume 2 of *AAAI'07*, pages 1601–1604. AAAI Press, 2007.

[7] Q. Yang, K. Wu, and Y. Jiang. *Learning action models from plan examples using weighted MAX-SAT*. Artif. Intell., 171(2):107 – 143, 2007.

[8] D. Aineto, S. Jiménez, and E. Onaindia. *Learning STRIPS action models with classical planning*. In Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS'18. AAAI Press, 2018.

[9] M. Asai and A. Fukunaga. *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary*. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, AAAI'18. AAAI Press, 2018.

[10] D. McDermott. *The 1998 AI planning system competition*. AI Mag., 21(2):35, 2000.

[11] M. Asai and A. Fukunaga. *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary*. arXiv preprint arXiv:1705.00154, 2017.

[12] Q. Yang, K. Wu, and Y. Jiang. *Learning Action Models from Plan Examples with Incomplete Knowledge*. In Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'05, pages 241–250. AAAI Press, 2005.

[13] T. J. Walsh and M. L. Littman. *Efficient Learning of Action Schemas and Web-service Descriptions*. In Proceedings of the Twenty-Third National Conference on Artificial Intelligence, volume 2 of *AAAI'08*, pages 714–719. AAAI Press, 2008.

[14] E. Amir and A. Chang. *Learning partially observable deterministic action models*. J. Artif. Intell. Res., 33:349–402, 2008.

[15] S. N. Cresswell, T. L. McCluskey, and M. M. West. *Acquiring planning domain models using LOCM*. The Knowledge Engineering Review, 28(2):195–213, 2013.

[16] K. Mourão, L. Zettlemoyer, R. P. A. Petrick, and M. Steedman. *Learning STRIPS Operators from Noisy and Incomplete Observations*. In Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI'12, pages 614–623. AUAI Press, 2012.

[17] B. Bonet and H. Geffner. *Learning First-Order Symbolic Planning Representations from Plain Graphs*. arXiv preprint arXiv:1909.05546, 2019.

[18] L. Amado, R. F. Pereira, J. Aires, M. Magnaguagno, R. Granada, and F. Meneguzzi. *Goal Recognition in Latent Space*. In 2018 International Joint Conference on Neural Networks, IJCNN'18, pages 1–8, 2018.

[19] M. Asai. *Unsupervised Grounding of Plannable First-Order Logic Representation from Images*. arXiv preprint arXiv:1902.08093, 2019.

[20] S. Keren, A. Gal, and E. Karpas. *Goal Recognition Design*. In Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS'14, pages 154–162. AAAI Press, 2014.

[21] S. Keren, A. Gal, and E. Karpas. *Strong Stubborn Sets for Efficient Goal Recognition Design*. In International Conference on Automated Planning and Scheduling, ICAPS'18, pages 141–149. AAAI Press, 2018.

[22] H. Harman and P. Simoens. *Action Graphs for Performing Goal Recognition Design on Human-Inhabited Environments*. Sensors, 19(12):2741, 2019.

7

Conclusions and Future Research

"We can only see a short distance ahead, but we can see plenty there that needs to be done."
Alan M. Turing (1950)

Symbolic models of an agent's behaviour facilitate artificial agents with the ability to recognise a human's intentions and plan their own actions. In this thesis, intention recognition includes determining which goal the human is pursuing, forcing the human to reveal their goal sooner and predicting which actions the human might perform next. These predictions enable a robot to create and execute a task plan, and thus provide proactive assistance. Moreover, when the robot is integrated into a dynamic smart environment, our framework reduces the amount of information the robot's planner must reason on and decreases the frequency of (re)planning. This chapter critically discusses the research questions and hypothesis that were listed in the introduction, and provides some possible future research directions.

Each chapter of the thesis contributes a component that is key to enabling proactive robot assistance in dynamic smart environments. In Chapter 2, we introduced an Action Graph approach to Goal Recognition (GR), which is able to handle the initial world state being inaccurately modelled. To determine the human's goal after fewer observations, as described in Chapter 3, our Goal Recognition Design (GRD) method modifies the initial state of the environment. Moreover, we introduced a distinctiveness metric that accounts for changes in the number of actions in the whole plan, as well as the non-distinctive prefix. Chapter 4 described our action prediction approach. When an observation is received, an Action Graph's node values are updated and the highest valued actions are extracted. These actions are mapped to a goal, and thus enable a robot to generate a task plan and execute actions on behalf of a human. In Chapter 5, we presented our Hierarchical Continual Planning in the Now framework. This framework improves a robot's ability to act in a dynamically changing smart environment. All these components require a symbolic model of an agent's behaviour; therefore, in Chapter 6, we proposed a method that automatically generates symbolic action definitions from unlabelled pairs of images.

7.1 Research Question Discussion

This PhD aimed to answer the six research questions that are stated in the introduction. In this section, each research question and the hypothesis related to the question is repeated, then discussed. This includes discussing if the hypothesis is valid or not, and how applicable our work is to the real-world.

Research Question 1: *Can a structure similar to those created by library-based approaches be generated from a PDDL defined GR problem?*

Hypothesis 1: *An AND-OR graph structure that models the order constraints between actions can be created from a GR problem defined in PDDL. This process will have a time complexity of $O(n^2)$, where n is the number of actions.*

In Chapters 2, 3 and 4 we presented different approaches for generating an Action Graph from an PDDL defined problem. The reasoning behind these differences is due to the specific challenge each chapter looked at addressing. Our GR approach (Chapter 2) took advantage of plans not needing to be flawlessly represented to perform GR. Actions were simply inserted into the graph in turn, by connecting the action to its dependencies.

In contrast, GRD and action prediction require the plans to be correctly represented. Our GRD method took a goal driven approach, that is, a breadth first traversal from each goal to the initial state was performed to insert the actions. This is because only the optimal plans for the navigation domain should be inserted. Furthermore, the initial state could contain additional atoms which can cause the traversal from the initial state to take much longer than traversing from the goal state.

Our action prediction method required all actions, whose preconditions are reachable, to be inserted into the Action Graph. Therefore, only when an action's preconditions were consistent with the initial state or/and other actions within the graph set its preconditions, was the action inserted. Unlike our GR and GRD approach, Goal Actions are not identified or created. This is because the observed agent could perform any sequence of actions rather than a sequence that reaches one of the hypothesis goals.

The Action Graph is an AND-OR graph structure similar to those developed for library-based recognition approaches. Moreover, each of our construction approaches takes $O(n)$, where n is the number of actions. Therefore, our first hypothesis was shown to be valid.

Research Question 2: *When the initial state is inaccurately defined, how can a goal recognition approach be prevented from suffering a major loss of accuracy?*

Hypothesis 2: *By creating a structure that models the actions' order constraints, an observee's goal can be predicted to the same degree of accuracy when differing percentages of fluents have incorrect initial values. When all fluents are correct, this approach will have a recall and precision that is not, on average, worse than a current state of the art approach by > 0.1. This will be evaluated on the dataset*

consisting of 15 domains, and their corresponding problems.

Our GR method inserted actions into an Action Graph without taking into consideration the initial value of fluents. This enabled our approach to be unaffected by fluents with incorrect initial values. Moreover, our approach outperformed a state of the art approach, namely, h_{gc} [1], when 10, 30 and 50 % of actions had been observed. At 70 and 100 % the recall of our approach was worse than h_{gc}'s by 0.04 and 0.06, respectively, and the precision by 0.02 and 0.03, respectively. These experiment results validate our hypothesis.

During the experiments, observations, i.e., symbolically represented actions, were read from a text file. We assumed that real-world sensor data can be transformed into discrete symbolic actions. For many domains this will be possible. For instance, an environment can be split-up into a grid to enable move actions to be observed. Nevertheless, some actions will be more challenging to observe. For example, for the Kitchen domain it could be difficult to determine which items a human is taking as the items may be occluded. Furthermore, humans can act irrationally. Although our work will handle some suboptimality/irrationality, real-world experiments are required to fully evaluate the applicability of our approach.

Research Question 3: *Can a configuration of an environment (e.g., a kitchen) that enables a observed agent's goal to be recognised after fewer observations be automatically discovered?*

Hypothesis 3: *Modifications can be applied to a structure that addresses Research Question 1, to find a configuration that makes the goals within the Kitchen domain more distinctive by at least 1 action.*

To recognise a human's goal after fewer observations, we developed an approach that modifies the Action Graph by swapping actions with their alternatives. For instance, the actions for taking bread from cupboard1 were swapped with the actions for taking bread from cupboard2. To measure the distinctiveness of the goals we introduced a new metric, ACD_{dep}. This metric calculates the average rather than the worst case distinctiveness, and takes into account the structure of the plans. Our Shrink–Reduce method successfully made the goals of a Kitchen domain more distinctive

by an ACD_{dep} of 1, and our Exhaustive approach reduced ACD_{dep} by 1.67. These results validated our hypothesis.

To improve the acceptability of the designs, our work allows the human to specify which objects should not be moved. In our experiments, we did not allow any items to be taken from the fridge or drawer. Nevertheless, if the human prevents too my items from being moved, the goals will not be able to be made more distinctive. Furthermore, real-world experiments are required to discover how accepting humans are of the re-designed environment. Until the benefit of the redesign can be witness by the human, they are unlikely to accept the modifications. For the kitchen environment, this will first require the improvement of a robot's ability to act autonomously. Our GRD approach could also be applied to gaming and virtual reality environments. This could enable the player/user's intentions to be recognised sooner.

Research Question 4: *When provided with a PDDL defined GR problem as input, how can an observed agent's next actions be predicted and used by a robot to provide assistance?*

Hypothesis 4: *The process of updating the node values will have a worst case time complexity of $O(n)$, where n is the number of nodes. Moreover, when the highest valid predicted actions are executed by a robot, a simulated observed agent performs fewer actions.*

Our action prediction method created an Action Graph and assigned all nodes an initial value of 0. When an action was observed, the nodes values were updated and the highest valued action nodes were extracted. This update procedure has a time complexity of $O(n)$. The highest valued predicted actions were then mapped to a goal and sent to a robot. The robot created a plan, and executed that plan if in comparison its plan was short and the human was far from completing the prediction. When the simulated observed agent made breakfast, which without assistance required 32 actions, the agent performed up to 5 fewer actions with the robot's assistance. For both making a pack-lunch and making dinner 4 fewer actions were performed by the observed agent when the robot assisted them. These experiments validate our hypothesis.

Our work only involved simulated experiments. To perform real-world experiments, further developments are required. As well as the already

mentioned challenge of determining which actions/activities a human is performing, the robot may lack certain capabilities. For instance, robots often lack the dexterity to open cupboards/drawers and pick-up objects; and those that have the dexterity, often make mistakes (e.g., take multiple attempts to open cupboards and drop objects). The trade-off between the benefits of a robot providing assistance and the negative impact mistakes will have, will need to be carefully considered before a person is provided with a proactive robot.

Research Question 5: *How can the state information and exogenous actuation capabilities of IoT devices be incorporated into a robot's continual planner without greatly increasing the computational cost and the frequency of (re-)planning?*

Hypothesis 5: *Our framework will enable a robot to start acting within 2 seconds of being provided with the goal for a simulated coffee fetching scenario. When the availability of different coffee machines changes, the robot will be less likely to have to re-plan than when everything is planned upfront.*

Our framework incorporated IoT devices into a robot's continual planner. All IoT devices were represented as a single object within the planning problems, which enabled fewer calls to an external capability checking module to be performed. Moreover, a hierarchical planning in the now approach was developed so that only a subsection of the the plan was planned in detail. This enabled the robot to re-plan less frequently. Our coffee-fetching experiment demonstrated our hierarchical approach can enable a robot to start acting sooner that when everything is planned upfront. Nevertheless, the hypothesis time was not achieved as the simulated robot started acting after 3.68 seconds rather than in under 2 seconds. This was partly due to the experiment set-up. All components were ran on a virtual machine; this caused the machine to operate slowly. Further work on reducing planning time is required.

Research Question 6: *Can a more compact set of symbolic action definitions be learnt from pairs of images by taking a explainable (white-box) approach?*

Hypothesis 6: *Generating action definitions will have a large computational cost but the produced PDDL will enable a task planner to find plans in less than 1*

second for Puzzle, Lights-Out and Towers-of-Hanoi problems.

In Chapter 6, a method for learning action definitions from pairs of images, namely, transitions, was presented. Objects were discovered by finding pixels that always change value simultaneously; both the location of these pixels and their values were defined as objects. Subsequently, the transitions were transformed into actions by generating predecessor and successor states from each image pair. These states enabled the actions' preconditions and effects to be determined. Actions were then converted into a set of action definitions. The maximum time it took a planner to solve Puzzle, Lights-Out and Towers-of-Hanoi problems using our learnt action definitions was 0.71 seconds, and the average was 0.06 seconds. Therefore, our hypothesis is valid.

Furthermore, as reported by Asai & Fukunaga [2, 3], a problem generated by the state of the art approach, Latplan, takes 4 hours for a task planner to solve. Thus, our work demonstrates a significant improvement. Nevertheless, the current applicability of these approaches is limited. In particular, the object discovery aspect of our work requires improving due to its large computational cost and its inability to handle occlusion. Expanding these approaches to other sources of data, e.g., real-world videos, is likely to be challenging and require the use of object and activity recognisers.

7.2 Future Work

In relation to this work, there are several open research challenges that require investigating further. This section highlights some of these challenges.

Real World Applicability

Although there is a considerable amount of research on symbolic AI algorithms, often the link between this research and the real world is missing. The intention recognition algorithms and planning framework of this doctoral thesis aspired to solve some of the challenges associated with deploying symbolic AI methods in the real world. Our GR method is able to handle the initial world state being partially unknown, and thus incorrectly modelled. When the human does not have a clearly defined goal, our action predictor can be ran to predict their next actions. Both these intention recognition methods aimed to handle observation sequences containing incorrect and missing actions. Moreover, our planning framework was designed to reduce the impact dynamic environments have on a robot's ability to function.

In contrast, the challenge of translating the action sequences produced from learnt PDDL into executable instructions remains unaddressed. The actions contained within these sequences consist of a unintelligible name and object list, e.g., `action_1(loc_1 loc_2 image_3)`. The link between this symbolic action and a robot's low-level control programs is missing. One step towards improving the understandability of these actions could be to perform object recognition on the learnt image objects and combined this with semantic reasoning. This could also help towards verifying the validity of the produced PDDL. Currently, these action sequences are translated into images that could then be interpreted, and thus executed, by a human. Note, although the actions and their definitions are unintelligible to robots and non-technical humans, why each action definition and its preconditions/effects where created by our system can be explained.

Stochastic Domains and Action Costs

Throughout this PhD research, domains were deterministic and actions had a uniform cost. Stochastic languages, such as PPDDL [4] and RDDL [5], enable uncertainty to be modelled. For instance, uncertainty due to noisy sensor data and in the outcome of actions (e.g., if the robot's wheels slip, the robot might not be entirely sure about its location). Moreover, actions can vary in how costly they are to execute, specifically, vary in their duration and resource requirements.

Each component of this thesis could be expanded to take probabilistic action outcomes and costs into account. Our Action Graph structure could be adjusted to enable stochastic information to be represented. This is achievable through the creation of probabilistic `OR` nodes for which the edges linking them to their children are labelled with probabilities. The node value initialisation procedure, and consequently the goal probabilities, will also need to take this uncertainty into account. As the aim of GRD is to improve how soon the observee's goal can be recognised, the actual action duration could be provided to the distinctiveness metric calculation, and our Shrink-Reduce method modified to first reduce the most costly non-distinctive prefix rather than the longest. Moreover, both action costs and probabilistic planning could be integrated into our continual planner. During the planning phase, a probabilistic planner, such as DESPOT [6], can be called. The execution and monitoring components could record, and thus learn, how costly an action was to execute and update the probabilities of the action's possible outcomes.

Plan Legibility and Predictability

Works on GRD could lead to artificial agents having improved plan legibility and predictability [7, 8]. An agent's actions are legible if an observer

can determine the agent's goal and are predictable if the agent's plan is recognisable. This of particular importance in robot task planning, so that a human can recognise the robot's intentions. Rather than finding non-distinctive plans, our GRD approach could be applied to finding the most distinctive and thus most legible plans.

Multiple Agents with Multiple Goals

Multiple humans and multiple robots could be acting within the same environment. These agents could be working independently and/or collaboratively. Multi-agent and multi-goal situations are only partially handled by our work.

When determining an observee's goal, the set of observations could include actions from any number of agents; however, it is assumed that a single goal is being pursued. In many situations this is a reasonable assumption to make as which human performs which action could be identifiable and humans often pursue a single goal. For example, a human can only navigate to a single location at a time. Our action prediction approach was shown to be applicable when multiple goals are pursued.

In our continual planning framework, a robot creates its own plan without considering the plans of other robots or humans. This framework's cloud-based Plan Validator receives a copy of each robot's plan, and thus in future work, could detect conflicts and invoke resource negotiations. Moreover, this component can be linked to our goal recogniser and action predictor, so that a robot pursuing an independent goal can adjust its plan in accordance with the human's future actions. This includes preventing the robot from interrupting the human, and the human's actions unintentionally assisting the robot to complete its task. For instance, if it is predicted that the human will soon open a door within the robot's path, rather than attempt to open that door, the robot can wait for the human to open it.

Combining Symbolic and Subsymbolic Methods

Within the field of AI, further approaches that combined symbolic and subsymbolic (e.g., deep-learning) methods are starting to emerge. Subsymbolic methods employ Neural Networks (NNs) to learn about unstructured data in a supervised or unsupervised manner. For instance, an autoencoder learns a, usually reduced, representation of a dataset using unsupervised learning. Deep autoencoders have been applied to generating symbolic logic, including PDDL [3, 9]. Nevertheless, NN-based approaches are black-boxes, that is, the decisions they make are unexplainable [10]. Integrating symbolic logic into subsymbolic methods could improve their explainability and facilitate the development of a more general purpose NN.

AI researchers have been inspired by many different disciplines (e.g., biology, physics, psychology and science fiction). Conversely, such disciplines are reaping the benefits of recent advancements in AI. It will be interesting to see how different research areas are combined and expanded in future pursuits for artificial intelligence. This doctoral thesis hopefully inspires others to contribute to this field of research.

References

[1] R. F. Pereira, N. Oren, and F. Meneguzzi. *Landmark-Based Heuristics for Goal Recognition*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, pages 3622–3628. AAAI Press, 2017.

[2] M. Asai and A. Fukunaga. *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary*. arXiv preprint arXiv:1705.00154, 2017.

[3] M. Asai and A. Fukunaga. *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary*. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, AAAI'18. AAAI Press, 2018.

[4] H. L. Younes and M. L. Littman. *PPDDL1.0: The language for the Probabilistic Part of IPC-4*. In Proceedings of the International Planning Competition, 2004.

[5] S. Sanner. *Relational Dynamic Influence Diagram Language (RDDL): Language Description*. 2010.

[6] N. Ye, A. Somani, D. Hsu, and W. S. Lee. *DESPOT: Online POMDP Planning with Regularization*. J. Artif. Intell. Res., 58:231–266, 2017.

[7] T. Chakraborti, A. Kulkarni, S. Sreedharan, D. E. Smith, and S. Kambhampati. *Explicability? Legibility? Predictability? Transparency? Privacy? Security? The Emerging Landscape of Interpretable Agent Behavior*. In Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS'19, 2019.

[8] A. D. Dragan, K. C. Lee, and S. S. Srinivasa. *Legibility and Predictability of Robot Motion*. In Proceedings of the 8th ACM/IEEE International Conference on Human-Robot Interaction, HRI '13, pages 301–308, Piscataway, NJ, USA, 2013. IEEE Press.

[9] L. Amado, R. F. Pereira, J. Aires, M. Magnaguagno, R. Granada, and F. Meneguzzi. *Goal Recognition in Latent Space*. In International Joint Conference on Neural Networks, IJCNN'18, pages 1–8, 2018.

[10] W. Samek and K.-R. Müller. *Towards Explainable Artificial Intelligence*, pages 5–22. Springer International Publishing, Cham, 2019.