

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Josef Hájiček**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Multiagent Narrative Planning**

Guidelines:

Within his individual project, the student examined single-agent domain-independent planners both for deterministic and probabilistic planning on a mock-up narrative planning problem. The master thesis builds on this knowledge.

- 1) Study literature in the field of deterministic multiagent planning and in the field of narrative planning.
- 2) Design a multiagent planning system for automated generation of narratives.
- 3) Implement designed system of multiagent narrative planning.
- 4) Implement interactive narrative in the form of a computer game using the implemented system.
- 5) Verify functionality of the implemented system.
- 6) Experimentally evaluate computational complexity of the proposed solution.

Bibliography/Sources:

Malik Ghallab, Dana Nau, Paolo Traverso, Automated Planning, Theory and Practice, Morgan Kaufmann Publishers, 2004. ISBN 1-55860-856-7

Partik Haslum, Narrative planning: Compilations to classical planning, Journal of Artificial Intelligence Research 44. 2012, pp 383-395.

Ronen I. Brafmana, Carmel Domshlak, On the complexity of planning for agent teams and its implications for single agent planning, Artificial Intelligence 198. 2013, pp. 52-71.

Mark O. Riedl, R. Michael Young, Narrative Planning: Balancing Plot and Character, Journal of Artificial Intelligence Research 39. 2010, pp. 217-268.

Diploma Thesis Supervisor: Ing. Antonín Komenda, Ph.D.

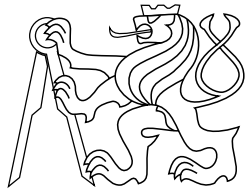
Valid until the end of the summer semester of academic year 2015/2016


doc. Ing. Filip Železný, Ph.D.
Head of Department




prof. Ing. Pavel Ripka, CSc.
Dean

Prague, March 26, 2015



CZECH TECHNICAL
UNIVERSITY IN PRAGUE
Department of Computer Science

Josef Hájíček

MULTIAGENT NARRATIVE PLANNING

Master's thesis

Apr 8, 2015

Supervisor: Ing. Antonín Komenda, Ph.D.

ACKNOWLEDGEMENT

I would like to thank my advisor, Ing. Antonín Komenda, Ph.D., for being helpful whenever it was necessary, and I would also like to thank my parents and my friends for their continued support.

DECLARATION

I hereby declare that this thesis is the result of my own work and all the sources I used are in the list of references, in accordance with the Methodological Instructions on Ethical Principles in the Preparation of University Theses.

Prague, Apr 8, 2015

ABSTRACT

This thesis presents a system for artificial generation of narratives with optional interactivity. First, the story domain, written in a custom meta-language, is compiled to PDDL. Then, any state of the art planning algorithm can be used to find characters' optimal actions. Character's prediction is modelled by Cognitive Hierarchy to ensure their believable behaviour. System ability to produce believable and interesting stories was verified experimentally.

KEYWORDS:

narratives, compilation, planning, multiagent planning, PDDL, computer games

ABSTRAKT

Tato práce představuje systém pro generování příběhů s možností interakce. Doména příběhu, zapsaná v mnou vytvořeném meta-jazyce, je nejprve zkompileovaná do PDDL jazyka. To umožňuje použít jakýkoliv state of the art plánovací algoritmus pro optimální generování akcí postav. Predikce postavy je modelována pomocí Kognitivní hierarchie k zajištění jejího uvěřitelného chování. Experimentálně bylo ověřeno, že systém je schopen produkovat uvěřitelné a zajímavé příběhy.

KLÍČOVÁ SLOVA:

příběhy, kompilace, plánování, multiagentní plánování, PDDL, počítačové hry

Contents

1	Introduction	1
2	State of the art of Narrative planning	3
2.1	Planning	3
2.2	STRIPS	3
2.3	PDDL	5
2.4	STRIPS to PDDL	5
2.5	PDDL Depot problem example	6
2.6	IPOCL planner	9
2.7	IMPRACTical	10
2.8	Cognitive Hierarchy	11
3	Direct approach through classical planning	13
3.1	The idea	13
3.2	The realization	14
3.3	Experiments	16
3.4	Discussion	18
4	Multiagent narrative planning system	19
4.1	System overview	19
4.2	Narrative cognitive definition language	20
4.3	Multiagent narrative planning algorithm	21
4.4	Conflicted actions	23
4.5	Plan merging	23
4.6	Planning	25
4.7	Compilation	26
4.8	Simulation	33
4.9	Stories, games and interactivity	33
4.10	Implementation	34

5 Experiments	35
5.1 Computational complexity	35
5.2 Survey	37
5.3 Uncovered shortcomings	43
6 Conclusion	45
A Testing machine configuration	49
A.1 Processor	49
A.2 RAM	49
A.3 OS	49
B CD content	51
C Running the system	53
C.1 Global requirements	53
C.2 Fast Downward compilation	53
C.3 Generating narratives	54
C.4 Editing the scenario	54

List of Figures

2.1	A PDDL action	5
2.2	PDDL domain file example	6
2.3	Depot instance example	7
2.4	PDDL problem file example	8
2.5	The example of story generated by IPOCL, (Riedl and Young, 2010)	9
2.6	Generated IMPRACTical narrative illustrated on a <i>Aladin</i> domain character co-operation, prediction of other character intent, chaining of commands and intent and unfulfilled intents (Teutenberg and Porteous, 2013)	10
2.7	Prisoner’s dilemma pay-off matrix	11
3.1	PDDL-rewritten kill action	15
3.2	Auxiliary cost action	15
3.3	Inheriting action	16
3.4	Illustration of the initial state	17
4.1	NCDL agents example	20
4.2	NCDL actions example	21
4.3	Conflict resolving skills	25
4.4	Simulations of rival’s <i>kill</i> action in the case when the preconditions are fulfilled; <i>kill</i> action itself was defined in Figure 4.2	27
4.5	Simulations of rival’s <i>kill</i> action in the case when the preconditions are not fulfilled; <i>kill</i> action itself was defined in Figure 4.2	27
4.6	Simulations of a rival’s action with the <i>unknown</i> flag/action	28
4.7	Compiled standard <i>kill</i> action for agent Kalam	29
4.8	NCDL kill contra escape definition	30
4.9	Kalam’s compiled kill contra escape action	31
4.10	Agent Kalam definition	31
4.11	Agent Kalam’s compiled goal to survive	32
4.12	The screenshot of the interface of the interactive mode	34

5.1	Compilation time for increasing number of agents and different problem sizes ($ A $ is the number of actions)	35
5.2	Runtime of the algorithm for increasing number of agents and different problem sizes ($ A $ is the number of actions)	36
5.3	Comparison of planning times of the <i>Aladin</i> domain	37
5.4	The believability and the attractiveness of the story #1	39
5.5	The believability and the attractiveness of the story #2	39
5.6	The believability and the attractiveness of the story #3	40
5.7	The believability and the attractiveness of the story #4	41
5.8	The believability and the attractiveness of the story #5	42
5.9	The believability and the attractiveness of the story #6	42

Introduction

Narratives play important role in the human history. Stories relate to all aspects of our culture, from literature and music to the modern media like movies and computer games. We believe that story generation can be automated with help of the artificial intelligence. To create stories, we have to have a system that creates story events and order them in a proper way. It can be modelled as trajectories in a space of possible states of the story (Porteous and Cavazza, 2009). It can be covered by area of automatic planing, or more particularly, narrative planning.

Most of the related works use some kind of multiagent planning techniques, where agents represent the characters of the stories, to create systems of artificial story generating. Riedl and Young introduced a concept of intent-based generation of a story, where each character was, in contrast to the previous works, framed by commitments generated based on the required results of the story. This approach generates story, where characters act more believable than in the previous works (Riedl and Young, 2010).

Believability is the most important criterion, which has to be fulfilled in every artificial narrative system, unlike the systems based on the game theory, where each character acts in the most ration way in relations to other characters' actions. Because humans, who are usually characters in the stories, are not capable of full rational behavior.

Our approach does not use multiagent planning directly to produce story, but only for a predictions of a behavior of other characters and then according to Cognitive Hierarchy (Camerer et al., 2004), we use these predictions to create suboptimal plans and finally we merge them to the final story.

According to Cognitive Hierarchy (CH) humans create strategies to reach their goals despite contradictory goals of other agents by predicting behavior of these entities and then they suppose that their strategy is the most sophisticated. These predictions of course include opponents' predictions, but humans are capable to do these recursion predictions only to a limited depth (cognitive level).

We show that CH is applicable to the more complex situations than one-actions games

and that limited cognitive level improves efficiency of story generation. We analyse complexity of our approach theoretically and practically and demonstrate its competitiveness with the narrative compilation for classical planning proposed by (Haslum, 2012).

Big motivation to design our system the way how it is designed is an intention to handle replacing of any subset of the automated agents by the human controlled agents without any disruption of the generated story (with assumptions of *human-rational* behavior of the humans agents). It is very important in the interactive story-telling system, like the computer games.

State of the art of Narrative planning

In this chapter we will introduce state of the art of narrative planning and tools on which these system are based. It means that first of all we will describe planning in artificial intelligence.

2.1 PLANNING

Planning is an activity in which we try to organize activities to achieve our goal. A result of planning is a plan. Planning is a fundamental approach to problem solving by intelligent entities, so it is widely studied in the research area of artificial intelligence.

There are many formalizations for automated planning tasks. We will focus on the STRIPS (Stanford Research Institute Problem Solver) and PDDL (Planning Domain Definition Language) which are classically used to efficiently solve STRIPS tasks.

2.2 STRIPS

A STRIPS instance is a quadruple $\langle P, I, A, G \rangle$, where

- P is a set of propositions or facts describing world state,
- $I \in 2^P$ is an initial world state containing all propositions which holds in the world initially,
- A is a set of grounded actions,
- $G = \langle N, M \rangle$ is a goal condition, where $N \in 2^P$ are variables which must hold in a goal state and $M \in 2^P$ are variable which must not hold in a goal state.

Formally, a state $C \in 2^P$ is a set of propositions and the transition between states is a function

$$\text{succ}: 2^P \times A \rightarrow 2^P.$$

Action a is a quadruple $\langle \text{pre}^+, \text{pre}^-, \text{add}, \text{del} \rangle$, where

- $\text{pre}^+(a) \in 2^P$ are positive preconditions
- $\text{pre}^-(a) \in 2^P$ are negative preconditions
- $\text{add}(a) \in 2^P$ are positive effects
- $\text{del}(a) \in 2^P$ are negative effects

and then

$$\text{succ}(C, \langle \text{pre}^+, \text{pre}^-, \text{add}, \text{del} \rangle) = (C \setminus \text{del}) \cup \text{add}.$$

The function *succ* is defined on C and $\langle \text{pre}^+, \text{pre}^-, \text{add}, \text{del} \rangle$ *iff*

$$\text{pre}^+ \subseteq C, \text{pre}^- \cap C = \emptyset.$$

Informally, we can say that action is applicable in state C if all positive preconditions are true in C and no negative preconditions are true in C . The result state C' is state C extended by positive effects and narrowed by negative effects.

The function *succ* can be extended to a sequence of action by a recursion relation

$$\begin{aligned} \text{succ}(C, []) &= C \\ \text{succ}(C, [a_1, a_2, \dots, a_k]) &= \text{succ}(\text{succ}(C, a_1), [a_2, \dots, a_k]) \end{aligned}$$

Then the sequence $[a_1, \dots, a_k]$ of actions is a valid plan for a STRIPS instance *iff*

$$\begin{aligned} \text{succ}(I, [a_1, \dots, a_k]) &= F \\ N &\subseteq F \\ M \cap F &= \emptyset. \end{aligned}$$

In the rest of the thesis we will often use *nop* action. It has no preconditions and no effects. Formally,

$$\text{pre}^+(\textit{nop}) = \text{pre}^-(\textit{nop}) = \text{add}(\textit{nop}) = \text{del}(\textit{nop}) = \emptyset$$

2.3 PDDL

PDDL (Planning Domain Definition Language) is a standard encoding language for classical planning tasks. There are many extensions, so most planners (automated systems for solving planning task) do not support full PDDL. However all planners support STRIPS subset, so a STRIPS task described by PDDL can be solved practically by any planner.

2.4 STRIPS TO PDDL

Because PDDL is a standard input format for planners and because PDDL allows to write STRIPS problem in more concise way, we will use STRIPS tasks written in PDDL for our planning domains.

STRIPS propositions can be written in PDDL as predicates. It means that initial state is defined as a list of grounded predicates and goal conditions are two list of predicates, positive and negative. Positive list is the set N and negative is the set M from the STRIPS definition.

The syntax of actions will be shown in an example later.

```

1 (:action action_X
2   :parameters
3     (?a - type_A ?b - type_B ?c - type_C)
4   :precondition
5     (and
6       (pred_1 ?a ?b)
7       (not (pred_2 ?b ?c))
8     )
9   :effect
10    (and
11      (pred_3 ?a ?b)
12      (not (pred_4 ?b ?c))
13    )
14 ))

```

Figure 2.1: A PDDL action

In the Figure 2.1 we can see an action *action_X* on the line 1. The line 3 contains parameters of the action. Each parameter has to have its type, in our case the parameter *a* has a type *type_A* and so on. The first precondition on the line 6 is a positive precondition and in the STRIPS definition it is an element of pre^+ . The second precondition on the line 7 is wrapped in a *not* operator and it means that it is a negative precondition. Effects are defined similarly. The effect on the line 11 is positive and effect on the line 12 is negative.

STRIPS does not define parameters, so the actions in PDDL are *grounded*. It means that planner creates one action for each combination of action's parameters. This is possible, because the number of instances of each type is finite.

2.5 PDDL DEPOT PROBLEM EXAMPLE

In this section we will introduce an example problem written in PDDL and we will explain the rest of the syntax and semantics of the PDDL.

DOMAIN

```

1 (define (domain depot)
2   (:requirements
3     :typing :action-costs)
4   (:types
5     gate box truck - object)
6   (:predicates
7     (box_at ?x - box ?y - gate)
8     (truck_at ?x - truck ?y - gate)
9     (truck_load ?x - truck ?y - box)
10    (empty_truck ?x - truck))
11
12   (:functions
13     (total-cost)
14     (time ?x - gate ?y - gate))
15
16   (:action load_truck
17     :parameters
18       (?t - truck ?g - gate ?b - box)
19     :precondition
20       (and
21         (box_at ?b ?g)
22         (truck_at ?t ?g)
23         (empty_truck ?t)
24       )
25     :effect
26       (and
27         (not (box_at ?b ?g))
28         (not (empty_truck ?t))
29         (truck_load ?t ?b)
30       )
31 )
32
33   (:action unload_truck
34     :parameters
35       (?t - truck ?g - gate ?b - box)
36     :precondition
37       (and
38         (truck_load ?t ?b)
39         (truck_at ?t ?g)
40       )
41     :effect
42       (and
43         (not (truck_load ?t ?b))
44         (empty_truck ?t)
45         (box_at ?b ?g)
46       )
47 )
48
49   (:action move_truck
50     :parameters
51       (?t - truck ?a - gate ?b - gate)
52     :precondition
53       (and
54         (truck_at ?t ?a)
55       )
56     :effect
57       (and
58         (not (truck_at ?t ?a))
59         (truck_at ?t ?b)
60         (increase (total-cost) (time ?a ?b))
61       )
62 ))

```

Figure 2.2: PDDL domain file example

The example in Figure 2.2 models a logistics problem with depots. A depot has several gates and there can be some boxes at each gate, which we have to transport to another gate. There is a truck which we can use to transport these boxes. A truck can carry only one box at a time. We want to minimize distance driven by the truck.

Figure 2.2 describes only the rules of the problem. It describes predicates and actions. The rest of the problem, the initial and goal state, is described in the *problem file*.

This partition is typical because we often have many problems taking place in the same environment.

We will go through the example and describe the PDDL in more details.

The lines 2 and 3 are requirements on the planner. *:typing* means that we use typed parameters and *:action-cost* means that we want to have defined cost for each action and optimize length of the generated plan.

The lines 4 and 5 contain types of objects we will use to describe states and actions. In our case the objects are gates, boxes and trucks.

The lines 6–10 describe predicates we will use to describe a state. *box_at* represents information at what gate what box is deposited. *truck_at* represents information about truck's position. *truck_load* represents information about truck's load. *empty_truck* represents information whether the truck is empty or not. Notice that each parameter of predicate has a type and thanks to *:typing* requirement, the planner checks these types and so there is smaller probability of making inconsistencies in the domain file. It also lowers the complexity, because the planner generates the grounded operators only for these types.

The lines 12, 13 and 14 contains functions. The functions are used to optimize cost of the plan.

The last distinct line is the line 60. It uses a function and it says one of the effect of the action to increase function *total-cost* by the function (*time ?a ?b*). Functions will be described later.

PROBLEM

The specific initial and goal state are described in the *problem file* in Figure 2.4.

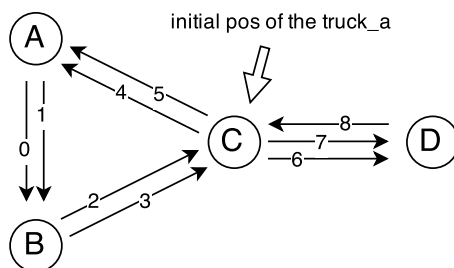


Figure 2.3: Depot instance example

In Figure 2.3, we can see a particular instance of the depot problem corresponding to the example in Figure 2.4. The arrows indicate original and goal destination of the boxes. Distances (cost of the *move_truck* action) between AB, AC, BC, CD are 10 and between AD, BD is 20. Arrow number represents box number.

The lines 4 to 10 contain all objects and their types. The lines 13 to 25 describe the initial world state. Lines 27 to 50 contains functions' initial values. Lines 55 to 63 describe

the goal state and the line 67 says that we want plan to minimize the value of the *total-cost* function, which represent the plan cost.

Solution returned from PDDL solver is a list of actions, where the truck moves in sequence *C,A,B,C,A,B,C,D,C,D* and at each gate one box is unloaded and another one loaded. It is easy to see that this plan is optimal. Of course, there are more plans with the same cost. But although it seems to be trivial task, it takes more than 20s to compute it on a computer with Intel Core i5 2.9 GHz using *Fast Downward* (Helmert, 2011) planner. STRIPS planning is well known PSPACE-complete problem.

```

1 (define (problem depot_task_0)           36 (= (time gate_a gate_d) 20)
2   (:domain depot)                       37 (= (time gate_d gate_a) 20)
3   (:objects                              38
4     gate_a gate_b gate_c gate_d         39 (= (time gate_b gate_d) 20)
5     - gate                             40 (= (time gate_d gate_b) 20)
6     truck_a                            41
7     - truck                             42 (= (time gate_c gate_d) 10)
8     b0 b1 b2 b3 b4                     43 (= (time gate_d gate_c) 10)
9     b5 b6 b7 b8                         44
10    - box)                              45 (= (time gate_a gate_a) 0)
11                                       46 (= (time gate_b gate_b) 0)
12 (:init                                  47 (= (time gate_c gate_c) 0)
13   (empty_truck truck_a)                48 (= (time gate_d gate_d) 0)
14                                       49
15   (truck_at truck_a gate_c)            50 (= (total-cost) 0)
16                                       51 )
17   (box_at b0 gate_a)                   52
18   (box_at b1 gate_a)                   53 (:goal
19   (box_at b2 gate_b)                   54   (and
20   (box_at b3 gate_b)                   55     (box_at b0 gate_b)
21   (box_at b4 gate_c)                   56     (box_at b1 gate_b)
22   (box_at b5 gate_c)                   57     (box_at b2 gate_c)
23   (box_at b6 gate_c)                   58     (box_at b3 gate_c)
24   (box_at b7 gate_c)                   59     (box_at b4 gate_a)
25   (box_at b8 gate_d)                   60     (box_at b5 gate_a)
26                                       61     (box_at b6 gate_d)
27   (= (time gate_a gate_b) 10)          62     (box_at b7 gate_d)
28   (= (time gate_b gate_a) 10)          63     (box_at b8 gate_c)
29                                       64   )
30   (= (time gate_b gate_c) 10)          65 )
31   (= (time gate_c gate_b) 10)          66
32                                       67 (:metric minimize (total-cost))
33   (= (time gate_c gate_a) 10)          68
34   (= (time gate_a gate_c) 10)          69 )
35

```

Figure 2.4: PDDL problem file example

2.6 IPOCL PLANNER

IPOCL (Intent-based Partial Order Causal Link) planner was introduced by (Riedl and Young, 2010). The IPOCL algorithm is based on a class of planning algorithms called Partial Order Causal Link (POCL) planners. The idea of a partial-order planner is to have a partial ordering between actions and only commit to an ordering between actions when forced.

IPOCL enriches the POCL by a frame of commitment. The purpose of the frame of commitment is to record a character's internal goal and the actions of the plan that the character will appear to perform during storytelling to achieve that goal. However, from the perspective of the audience, it is not enough to declare a character as having a goal. In order to make inferences about character intentions and plans, the audience must observe the characters to form and commit to goals (Riedl and Young, 2010).

There is a woman named Jasmine. There is a king named Jafar. This is a story about how King Jafar becomes married to Jasmine. There is a magic genie. This is also a story about how the genie dies. There is a magic lamp. There is a dragon. The dragon has the magic lamp. The genie is confined within the magic lamp. There is a brave knight named Aladdin. Aladdin travels from the castle to the mountains. Aladdin slays the dragon. The dragon is dead. Aladdin takes the magic lamp from the dead body of the dragon. Aladdin travels from the mountains to the castle. Aladdin hands the magic lamp to King Jafar. The genie is in the magic lamp. King Jafar rubs the magic lamp and summons the genie out of it. The genie is not confined within the magic lamp. The genie casts a spell on Jasmine making her fall in love with King Jafar. Jasmine is madly in love with King Jafar. Aladdin slays the genie. King Jafar is not married. Jasmine is very beautiful. King Jafar sees Jasmine and instantly falls in love with her. King Jafar and Jasmine wed in an extravagant ceremony. The genie is dead. King Jafar and Jasmine are married. The end.

Figure 2.5: The example of story generated by IPOCL, (Riedl and Young, 2010)

In Figure 2.5 we can see an example of a story generated by IPOCL system. The domain of the example appears in more papers about narrative planning, so we will also use it as benchmark for our final system.

We can see that IPOCL generates believable stories, but from our point of view, it has two shortcomings. The first one is time. Generating such story took more than 12 hours, which is too much for real-time systems like computer games. The second shortcoming is the centralized design. Because actions of all characters are planned in one planning process, there is no easy way how to replace some characters by human players. Another unfortunate property of IPOCL is related to the centralized design as well. IPOCL cannot

generate stories, where actions fail.

The problem with time can be solved by translating IPOCL to PDDL, as it has been shown by (Haslum, 2012). The time of the exemplified story was reduced by the translation and a planner for classical planning to 45 seconds.

2.7 IMPRACTICAL

IMPRACTical is an abbreviation of Intentional Multi-agent Planning with Relevant Actions, introduced by (Teutenberg and Porteous, 2013). The authors identified three reasoning process for generating believable stories: cooperation of two or more characters; characters prediction of other characters' actions; and the occurrence of chains of commands which propagate intent (Teutenberg and Porteous, 2013). The principle of this approach is to represent each character as an agent. Each agent uses narrative generator providing all possible believable actions. This allows to perform more general search of the action space. This search can use any heuristics used in classical planning.

1. Jafar falls in love with Jasmine
2. Aladdin falls in love with Jasmine
3. Aladdin travels to the mountain
4. Jafar travels to the mountain (*predicting*)
5. Aladdin slays Dragon
6. Aladdin pillages the magic lamp from Dragon
7. Jafar orders Aladdin to help marry
8. Aladdin summons Genie from the magic lamp
9. Aladdin commands Genie to help marry
10. Genie appears threatening to Jafar
11. Jafar orders Aladdin to slay the Genie
12. Jafar travels to the Castle
13. Genie casts a spell on Jasmine (*cooperating*)
14. Jafar marries Jasmine
15. Aladdin slays Genie

Figure 2.6: Generated IMPRACTical narrative illustrated on a *Aladin* domain character co-operation, prediction of other character intent, chaining of commands and intent and unfulfilled intents (Teutenberg and Porteous, 2013)

Figure 2.6 contains example of a story generated by the IMPRACTical. The total time of generating is about 2 seconds.

2.8 COGNITIVE HIERARCHY

In this section we will introduce the concept of Cognitive Hierarchy (Camerer et al., 2004), which is the core idea of our solution for narrative planning in this thesis.

EQUILIBRIUM

The best way how to deal with interchangeability of characters represented by a planner or by a human player, is to model the story as a non-cooperative game, where players' strategies generate a believable story. The most desirable state of any game is its equilibrium, mainly the *Nash equilibrium*. The Nash equilibrium is a set of strategies (one strategy for each player) where no player can improve his payoff without lowering payoffs of other players. We will show it on the example of the *Prisoner's dilemma* game.

	C	D
C	2/2	0/5
D	5/0	4/4

Figure 2.7: Prisoner's dilemma pay-off matrix

In Figure 2.7, we see a pay-off matrix of the Prisoner's dilemma game. It represent a situation, where two persons are accused of a crime. They are kept in separate cells and each of them has two options. To confess (C) or to deny (D). The pay-off matrix represents the numbers of years, which they will be sentenced to. If both confess, it will be mitigating circumstance and each of them will get two years in a prison. If both deny, each of them will get four year because there is evidence of their crime. If one confesses but the other deny, the first one will have mitigating circumstance and the second one will not, so the first one will be freed, but the second one will get five years. It's easy to see that Nash equilibrium is when both choose strategy deny (D), because whether the opponent plays (C) or (D), the best respond strategy is to play (D).

Although Nash equilibrium has many applications, it is not what we want. Because in Nash equilibrium all players behave fully rationally and suppose that the other players behave also fully rationally. But this is not how people behave in the real world because the human rationality is bounded (Chen, 2013).

COGNITIVE HIERARCHY

Cognitive Hierarchy (CH) brings another view on how people think when they are playing "games" in real life. According to the CH, humans create strategies to reach their goals

despite contradictory goals of other agents by predicting behavior of other entities and then they suppose that their strategy is the most sophisticated. These predictions include opponents' predictions, but humans are capable to do these recursion predictions only to a limited depth (cognitive level). Our system is based on this principle.

The example of the limited recursion is the "beauty contest" game. Players are asked to pick numbers from 0 to 100, and the player whose number is closest to $\frac{2}{3}$ of the average wins a prize. Equilibrium theory predicts each contestant will reason as follows: "Even if all the other players guess 100, I should guess no more than $\frac{2}{3}$ times 100, or 67. Assuming that the other contestants reason similarly, however, I should guess no more than $\frac{2}{3}$ times 67, or 45..." and so on, finally concluding that the only rational and consistent choice for all the players is zero.

However, when the game is played by humans, the average guess is typically between 20 and 35. Only, when the game is repetitively played in the same group, the average is approaching to 0 (Camerer et al., 2004).

MODEL

The CH model consists of iterative decision rules for k -players doing k depths of thinking. k -player always supposes that other players are j -players, where $0 \leq j < k$.

We assume that players doing $k \geq 1$ steps do not realize that others are using more than k steps of thinking. This is plausible because the brain has limits (such as working memory in reasoning through complex games) and also does not always understand its own limits. We also assume that people are overconfident and do not realize there are others using exactly as many thinking steps as they are. This is consistent with psychological evidence of persistent overconfidence about relative skill in many domains (Camerer et al., 2004).

The authors of (Camerer et al., 2004) use Poisson distribution for modelling opponents' level j . Another option described in the paper is

$$j = k - 1,$$

i.e., assuming that all opponents level is only one lower than our level. This model has almost the same properties as the model with Poisson distribution, but it is less computationally complex.

Because we need to adapt the CH to sequence games (in the paper authors consider only one round games), moreover to planning, which is a computationally hard task, we chose the $j = k - 1$ model in our system.

Direct approach through classical planning

In this chapter we will describe our first attempt to design multiagent narrative planning system. We used classical planning in a direct way and we only created a planning domain and a problem representing the narrative.

3.1 THE IDEA

As we said in the introduction, story is a sequence of story events. Result of a PDDL planner is a sequence of actions. So, the only thing which we have to do is to design a domain and a problem satisfying several condition.

- Every action is acted by story characters.
- Every character acts believable.
- Resultant story (plan) is interesting.

The first condition means that we do not want to have any *deus ex machina* events in the story. Any event of this type is suspicious for story observer and decreases the believability of the story. The second condition is similar. We do not want any *deus ex machina* acted by a story character. There must be motivation for every characters to act like they act, even if it goes against the story goal state. This is related to the last condition. Story where all characters has the same goal and cooperate to achieve this goal certainly meets first and second condition but a story like this is not interesting. Especially in computer games, where challenge for the player is a necessary part of the game. So, even if the world goal state is fixed and we want a plan that ends in that state, some players have to act against getting into this state.

Let us look at an example. Once upon a time there was a magic kingdom. A king of the kingdom had a beautiful daughter. There was a knight from a hostile kingdom. That is the initial world state. The goal state is that the king is dead and the princess is locked in the tower. Actions are *kill* and *lock*. One possible plan which can be generated by a planner is that the king kills himself and the princess locks herself in the tower. But this is certainly not believable behaviour. We rather want to get a plan in which the king locks his daughter in the tower and then he is killed by the knight. This example nicely shows that we cannot use the PDDL directly.

We want to preserve the PDDL structure in the manner such that there will be one domain with fixed actions, the initial and goal states will be specified in the problem file and the system will produce stories satisfying the above defined conditions for any combination of these states. Then there is only one way how to force the planner to generate such plans, which we want, and that is the action cost.

3.2 THE REALIZATION

Our domain describes several "pure" actions. Pure actions are actions as they are understood by humans. For example kill, travel, etc. These actions cannot be directly encoded to PDDL, as it will be explained later. Pure actions used in our domain are: *kill*, *travel*, *marry*, *leave city*, *take (conquer) city*, *take throne*, *kill by army*. The domain describes some *fantasy-middle age* world, where characters can kill each other, marry each other, occupy cities, usurp thrones etc.

As we said before, our task is to *rewrite* these actions to PDDL to generate believable stories using action costs. The principle is straightforward. An action has a high cost if it is not believable to be *played* by its character. Now we will go through some of these rewritten actions to show problems we have to deal with and the solutions.

The pure *kill* action has three parameters: killer (*?x*), victim (*?y*) and place (*?where*). To the PDDL-rewritten *kill* action (Figure 3.1), we had to add more information to calculate the right action cost. In our case it is the relation between the killer and the victim (*?q*) and the assassination skill of the killer (*?w*) and of the victim (*?e*). It is unlikely that character tries to kill his friend or someone with higher skill. The preconditions on the lines 8–16 are straightforward. The first four provide realistic behaviour of the action, e.g., (*alive_agent ?x*) and the rest provides binding of the parameters e.g., (*kill_skill ?x ?w*).

The preconditions on the line 18 and 19 are preconditions for the action cost. The PDDL (the version we use) does not allow to combine more costs in one action, so if we want to compose final cost from several simpler costs, we have two options. Manually prepare all possible combinations of these costs or use a "hack". Our solution is to have auxiliary actions, which we force to execute together with standard actions and its only effect is to increase the total cost. Example of such actions is in Figure 3.2. This cost

```

1 (:action kill
2   :parameters
3     (?x - agent ?y - agent
4     ?where - place ?q - rel_prop
5     ?w - kill_prop ?e - kill_prop)
6   :precondition
7     (and
8       (alive_agent ?x)
9       (alive_agent ?y)
10      (at_agent ?x ?where)
11      (at_agent ?y ?where)
12
13      (relation ?x ?y ?q)
14
15      (kill_skill ?x ?w)
16      (kill_skill ?y ?e)
17
18      (want_kill_cost_done ?q)
19      (can_kill_cost_done ?w ?e)
20      (transition_done)
21    )
22   :effect
23     (and
24       (dead ?y)
25       (killed_by ?y ?x)
26       (not (alive_agent ?y))
27
28       (not (want_kill_cost_done ?q))
29       (not (can_kill_cost_done ?w ?e))
30
31       (not (transition_done))
32       (cause_transition ?y)
33     )
34 )

```

Figure 3.1: PDDL-rewritten kill action

action express believability of killing between characters depending on their assassinating skill. It is not probable that a characters with bad such skill can kill a character with excellent skill. So, preconditions on the lines 18 and 19 (in the *kill* action) force planner to plan the cost actions and effects on the line 28 and 29 cancel predicates added by this actions.

```

1 (:action can_kill_cost_action
2   :parameters
3     (?x - kill_prop ?y - kill_prop)
4   :precondition
5     (and
6
7     )
8   :effect
9     (and
10      (can_kill_cost_done ?x ?y)
11      (increase (total-cost) (can_kill_cost ?x ?y))
12
13    )
14 )

```

Figure 3.2: Auxiliary cost action

Lines 20, 31 and 32 have similar function as lines 18 and 19. They force the planner to plan special action right after this action and before any other. Because the character *?y* was killed and because in the modelled world it is possible that the character had some noble titles, these titles have to be inherited. Line 20 block the action until such an inheriting is done and line 31 forces the inheriting action.

```

1 (:action inherit_titles                22      (forall (?t - title)
2   :parameters                          23      (when (has_title ?y ?t)
3     (?x - agent ?y - agent)           24      (not (has_title ?y ?t))
4   :precondition                          25      )
5     (and                                26      )
6     (cause_transition ?y)              27
7     (not (transition_done))            28      (forall (?a - armada)
8                                           29      (when (control_armada ?y ?a)
9     (dead ?y)                          30      (control_armada ?x ?a)
10    (alive_agent ?x)                    31      )
11    (heir ?x ?y)                        32      )
12  )                                       33      (forall (?a - armada)
13  :effect                                34      (when (control_armada ?y ?a)
14    (and                                  35      (not (control_armada ?y ?a))
15      (forall (?t - title)               36      )
16      (when (has_title ?y ?t)           37      )
17      (has_title ?x ?t)                 38
18    )                                     39      (not (cause_transition ?y))
19    )                                     40      (transition_done)
20  )                                       41      )
21  )                                       42  )

```

Figure 3.3: Inheriting action

In Figure 3.3, we can see the example of an inheriting action. In the effect, there is another possible construction of the PDDL, *forall* loop with *when* condition.

3.3 EXPERIMENTS

We wrote the whole domain in the way introduced in the previous section. Then we set initial and goal state. As a planner, we used the planning system *Fast Downward* (Helmert, 2011).

INITIAL STATE

There are four cities named Malaz, Quon Tali, Aren and Darujhistan. Malaz is the capital.

There are three characters named Sumar, Kalam, and Bugg. All of them are alive. Kalam has the title of a king. Bugg is the only heir of Kalam. Kalam and Bugg have friendly relation. Kalam and Sumar have neutral relation. Bugg and Sumar have neutral relation. Bugg has excellent assassinating skill, Kalam and Sumar has average assassinating skill. Kalam and Bugg are in Quon Tali and Sumar is in Aren.

There are two armies. Bonehunters are the strong one and Bridgeburners are the weak one. Bonehunters are controlled by Sumar and Bridgeburners by Kalam. Bonehunters are in Aren and Bridgeburners are in Malaz. (the situation is illustrated in Figure 3.4)

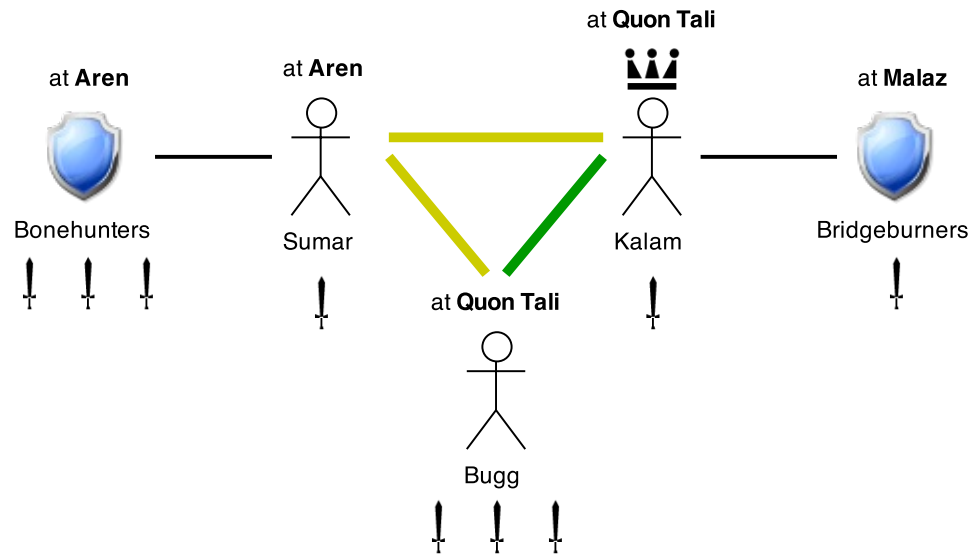


Figure 3.4: Illustration of the initial state

GOAL STATE

- Sumar is the king.

FOUND PLAN

1. Bugg leaves Quon Tali and go to Malaz (controlled by friend's army).
2. Sumar's army Bonehunters leaves city Aren.
3. Kalam leaves Quon Tali and go to Malaz (controlled by his army).
4. Bonehunters conquers Malaz (controlled by weaker Bridgeburners).
5. Sumar's army finds and kills Kalam.
6. Bugg inherits the king title.
7. Sumar's army finds and kills Bugg.
8. Sumar usurps the king title without any heir.
9. End. (elapsed time is 366s)

This plan was gained by using the *Fast Downward* (Helmert, 2011) planner and described domain. All auxiliary actions were removed.

3.4 DISCUSSION

This approach is not comfortable way to deal with narrative planning. It basically does not bring any useful tool for narrative planning and it leaves all work to a user because the user has to write the whole domain e.g. dealing with auxiliary actions for action costs. And more over, it does not satisfy the basic condition about easy replacement of a character by a humans player.

Using classical planning in narrative domain directly is not sufficient. It is necessary to chose another way how to deal with narrative planning.

Multiagent narrative planning system

In this chapter, we will describe our final narrative system and its implementation. We took the idea of Cognitive Hierarchy equilibrium and following the idea from section 2.8, we construct multiagent narrative planning system, where each story characters is represented by an agent. We create our own meta-language NCDL describing the world (domain) in PDDL-like way. This language can be partly compiled into PDDL, which provides us the planning power of the state of the art planners.

4.1 SYSTEM OVERVIEW

Our system works with two basic assumptions. The world is fully observable (in terms of planning) and the second assumption is that with given information about actions of other agents there is only one optimal plan to reach agent's goals.

The system is initialized by compiling NCDL (narrative cognitive definition language) to PDDL. NCDL is our meta-language describing an instance of Multiagent narrative planning system. The compilation is not complete. It means that a problem cannot be solved by one planning run, but instead the problem is solved iteratively by running the compiled PDDL domains and iterative generated PDDL problems. Each agent has its own domain file. The compilation is domain independent and it can handle any valid NCDL input.

The iterative algorithm starts by generating an empty plan for all agents. These are k -plans, where k refers to k -th level in the sense of CH and for the initial plans $k = 0$. Then the system for each agent iteratively generates plans of higher levels, i -plans. These plans are generated by a planner using compiled domain, generated plans with one lower level, $(i - 1)$ -plans (which corresponds to the model we described in the CH section) of all other agents and agent's goal. In the rest of the thesis we will refer to rivals of an agent as all

agents other than the agent. By this way the agent has information about actions of rivals and can react to these actions. Then the system chooses for each agent the first action from agent's k -plan (k matches the agent's cognitive level) and runs the simultaneous execution of these actions. Result of the executions represents the story events. These events are stored as parts of the result story. Then the world state is changed according to the result of the simulation and the whole described process starts again to generate the next part of the story.

4.2 NARRATIVE COGNITIVE DEFINITION LANGUAGE

Narrative cognitive definition language (NCDL) is a language describing input of our narrative multiagent planning system. Each instance of NCDL must contain:

- Actions with the same semantics as PDDL actions.
- Agents, where each agent has to have a name, a goal and a cognitive level.
- The initial world state.

Standard NCDL file contains additional data, but these additional data will be introduced during description of the compilation from NCDL to PDDL.

```

1 <agents>
2   <agent>
3     <name>sumar</name>
4     <level>3</level>
5
6     <goal>(alive sumar) -          5000 </goal>
7     <goal>(not (alive kalam)) - 5000 </goal>
8   </agent>
9
10  <agent>
11    <name>kalam</name>
12    <level>3</level>
13
14    <goal>(alive kalam) -          5000 </goal>
15    <goal>(not (alive sumar)) - 5000 </goal>
16  </agent>
17
18  <agent>
19    <name>tavore</name>
20    <level>3</level>
21
22    <goal>(alive tavore) -          5000 </goal>
23    <goal>(not (alive sumar)) - 5000 </goal>
24    <goal>(at tavore letheras) - 5000 </goal>
25  </agent>
26 </agents>

```

Figure 4.1: NCDL agents example

Figure 4.1 contains information about specific agents. Each agent must have a name, a cognitive level and goal(s) with its cost(s).

```

1 <actions>
2   <action>
3     <name>travel </name>
4     <cost>2 </cost>
5
6     <par>agent - player </par>
7     <par>from - place </par>
8     <par>to - place </par>
9
10    <precon>
11      <pred> alive </pred>
12      <val> agent </val>
13    </precon>
14    <precon>
15      <pred> at </pred>
16      <val> agent </val>
17      <val> from </val>
18    </precon>
19
20    <precon>
21      <pred> connected </pred>
22      <val> from </val>
23      <val> to </val>
24    </precon>
25
26    <effect>
27      <dir> NOT </dir>
28      <pred> at </pred>
29      <val> agent </val>
30      <val> from </val>
31    </effect>
32
33    <effect>
34      <pred> at </pred>
35      <val> agent </val>
36      <val> to </val>
37
38   </action>
39
40
41
42   <action>
43     <name>kill</name>
44     <cost>2</cost>
45
46     <par>agent - player </par>
47     <par>target - player </par>
48     <par>where - place </par>
49
50     <precon>
51       <pred> alive </pred>
52       <val> agent </val>
53     </precon>
54     <precon>
55       <pred> at </pred>
56       <val> agent </val>
57       <val> where </val>
58     </precon>
59     <precon>
60       <pred> at </pred>
61       <val> target </val>
62       <val> where </val>
63     </precon>
64
65     <effect>
66       <dir> NOT </dir>
67       <pred> alive </pred>
68       <val> target </val>
69     </effect>
70   </action>
71 </actions>

```

Figure 4.2: NCDL actions example

Figure 4.2 contains action definitions, the semantics is the same as in the PDDL. Predicates with `<dir>NOT</dir>` are negative predicates.

The initial world state consists of holding predicates facts like in the PDDL.

4.3 MULTIAGENT NARRATIVE PLANNING ALGORITHM

In this section, we will describe the multiagent narrative planning system in detail and in the next section we will describe its parts in detail.

The input of the the algorithm is

- Compiled NCDL domains (one for each agent)
- Agents

- Initial world state C
- Story length k
- Prediction length m
- Maximal cognitive level c

The output is a sequence of tuples of actions. One tuple for each time step. Each tuple contains one action for each agent.

```

1 story = [];
2 for i = 1 to k do
3   for a: agents do
4     generate 0-level empty plan for a;
5   end
6   for j = 1 to c do
7     for a: agents do
8       merge rivals' (j-1)-plans to consistent plans;
9       using compilation and classical planner generate j-level plan of length m
          given merged (j-1)-plans and current state C;
10    end
11  end
12  for each agent choose first action of their x-plan, merge them and save the result
      to C and add to the story;
13 end
14 return story;
```

Algorithm 1: Multiagent narrative planning algorithm

Each iteration of i in the first loop on the line 2 represent one time step. Because each agent execute one action at each time step, the number k is the length of the story, where each agent execute k actions.

The loop between lines 3 and 5 is initialization of agents' 0-plans. These plans contain only special *nop* action with no preconditions and no effects.

The main loop of the algorithm is on lines 6 to 11. As we said in the section 2.8 about Cognitive Hierarchy, we use model, where an agent supposes that all other agents has cognitive level one less than his level. It means that when an agent predicts to a certain depth k , he thinks that all other agents predict exactly to the depth $k - 1$. This principle is expressed on the line 9, where planning which result is a j -plan took as input $(j - 1)$ plans of all other agent. Generated plans have length m which is the length of agents' predictions and it means how many time steps the agents think ahead.

But before we can use plans from last iteration, we must merge them to be consistent. This happens on the line 8 and it will be explained later.

After we generate all plans, we can choose an action to be executed by each agent. This

action is the first action from his x -plan, where x is his cognitive level. These actions are merged again and the consistent result is added to the output story (line 12).

4.4 CONFLICTED ACTIONS

The natural exclusivity of actions is the biggest issue with a multiagent planning system. Because agents' goals are often antagonistic, their plans simultaneously executed can be also incompatible. This is a problem, because a story cannot contain such actions. For example: "Alice killed Bob while Bob married Catherine".

Formally, two actions A and B executed simultaneously are in conflict *iff* at least one of following is true.

1. $|\text{pre}^+(A) \cap \text{del}(B)| > 1$
2. $|\text{pre}^-(A) \cap \text{add}(B)| > 1$
3. $|\text{add}(A) \cap \text{del}(B)| > 1$

Informally, the condition 1 says that some fact must hold to execute action A but the action B removes this fact. The condition 2 says that some fact cannot hold to execute action A but the action B adds this fact. And the condition 3 says that the action A adds some fact and action B removes this fact concurrently.

The subsystem for detecting and solving these conflicted actions is an important part of the system. As there is no optimization in the process, it can be solved outside the planning process to lower the complexity of planning itself.

4.5 PLAN MERGING

The input of the merging algorithm is

- Agent t
- Agent t rivals' plans, all plans have the same length m
- The current world state C

The output is a set of consistent plans, one plan for each rival.

Algorithm 2 represents an unoptimized version of the merging algorithm. This unoptimized version is presented for better clarity.

On line 1 the algorithm iterates over all time steps m , because all plans have length m .

The loops on lines 2 and 3 iterate over all unique combinations of pairs of rivals.

On lines 4 and 5 the algorithm loads actions x and y according to the current time and rivals.

```

1  for  $i = 1$  to  $m$  do
2    for  $a = 1$  to number of rivals do
3      for  $b = a + 1$  to number of rivals do
4        action  $x = plans[i, a]$ ; // action of the rival  $a$  in the time  $i$ 
5        action  $y = plans[i, b]$ ;
6        if not succ'(C, x, t) then
7          for  $j = i$  to  $m$  do
8            |  $plans[j, a] = unknown$ ;
9          end
10       end
11       if not succ'(C, y, t) then
12         for  $j = i$  to  $m$  do
13           |  $plans[j, b] = unknown$ ;
14         end
15       end
16       if conflict(x, y) then
17         if resolve(x, y) == x then
18           for  $j = i$  to  $m$  do
19             |  $plans[j, b] = unknown$ ;
20           end
21         else
22           for  $j = i$  to  $m$  do
23             |  $plans[j, a] = unknown$ ;
24           end
25         end
26       end
27     end
28   end
29   for  $a = 1$  to number of rivals do
30     |  $C = succ(C, plans[i, a])$ ;
31   end
32 end
33 return plans;

```

Algorithm 2: Plan merging algorithm

The function $succ'(C, x, t)$ on the line 6 is similar to the function $succ(C, x)$ introduced in the STRIPS section (2.2). Unlike the $succ$ function, this function returns *true* or *false*. It returns *true* iff action's preconditions are fulfilled in the state C . But with one exception. Preconditions which contain agent t in their predicates can be skipped. This is because in merging algorithm we merge plans of rivals' of the agent t . The plan of the agent t is not present, because the result of merging is used to generate this plan.

Because the action cannot be executed (the lines 6 and 11), rival's plan is interrupted and we no longer consider it. Because of this, we replace, on the line 8, the rest of the rival's plan by the *unknown* action. This action has same semantic as *nop* action. It means no preconditions, no effects, no possible conflict with any other action. We choose *unknown* instead of *nop* only because of the readability.

On the line 16, the algorithm checks conflict between actions. If there is a conflict, the algorithm must solve it on the line 17. Now we have to extend the NCDL. Each pair of actions which can be possible in conflict must have stated skills which resolve conflicts between these action. We can see an example in Figure 4.3, which says that conflict between actions *escape* and *kill* wins the killer if its *kill_skill* is higher than the *escape_skill* of his victim. With equal skills the winner is the first agent. Specific skills can be replaced by special symbols *winner* and *fallen* which mean that agent playing action with the *winner* skill will always win.

The *resolve* function on the line 17 returns the action of the *winner*. It means that the plan of the other rival is interrupted and we no longer consider it during next time steps.

On the lines 29 and 30, the algorithm applies actions in the current time step i . Because there cannot be any conflicts between these action (conflicted actions were replaced by the *unknown* actions), the applying order does not matter.

```

1 <conflict>
2   <action_0>escape </action_0>
3   <action_1>kill   </action_1>
4
5   <skill_0>escape_skill </skill_0>
6   <skill_1>kill_skill   </skill_1>
7 </conflict>

```

Figure 4.3: Conflict resolving skills

4.6 PLANNING

After the merging rivals' plans to a consistent set, we can use them to plan a optimal plan (strategy) for a particular agent. The input for the planning of the k -plan of the agent a is:

- $(k - 1)$ -plans of all rivals of the agent a
- current state C

The output is the k -plan for the agent a .

The generated plan must meet several conditions:

- One action for each time step.
- Actions are consistent with rivals' actions.

- The plan leads to fulfilling of agent's goal for minimal plan cost.

PLAN STRUCTURE

To meet conditions introduced above, the plan will have to have following structure. The plan is divided by the time steps. In each step, the planner simulates rivals' actions in the current time step. This simulation ensures that the planner's inner state is actual and reflects effects of rivals' actions. Then planner plans one non-conflicted action for the agent and move forward in time.

After the last time step, the planner checks each agent's goal and if some goal is not fulfilled, the planner increases plan's cost by the value of the goal. This forces the planner to plan towards goals but not for price of exceeding the goal's value. The costs of the actions will still reflect the believability of their execution by the agent.

4.7 COMPILATION

In this section we will describe, on examples, each type of compiled action. For each agent, one domain is compiled, where all parameters are related to the agent and his rivals are replaced by a specific name representing a specific agent. The examples will be compiled from the example of the NCDL in Figure 4.2.

SIMULATION OF RIVAL'S ACTION

In Figures 4.4 and 4.5 we can see the way how we deal with simulations of rivals' actions. Simulating consists of two actions. The first action is used in the case when the preconditions of the action are fulfilled and the rival has not the *unknown* flag and the second one when the preconditions are not fulfilled and the rival has not the *unknown* flag. Unknown flag has same semantic as replacing of the actions by the *unknown* action. It says that rival's plan was interrupted and we no longer consider it. The way how the planner can interrupt rival's plan will be explain in the section about *contra* actions.

Parameters are the same in both actions. There are standard parameters of the simulated action and the parameter expressing current time step.

The lines 6 bind the parameter time to the current time value. The lines 7 check that the *unknown* flag is *false*. The lines 8 checks whether the rival has not been simulated in this time step yet. The lines 9 checks that the rival really *plays* the action in the current time step.

The lines 11 to 13 in the first action (Figure 4.4) checks that preconditions of the action are fulfilled and the lines 11 to 15 in the second action (Figure 4.5) check the opposite. We can see that the these lines are in the disjunction. An action containing disjunction

```

1 (:action run_rival_kill
2   :parameters
3     (?target - player ?where - place ?rival - player ?current - number)
4   :precondition
5     (and
6       (time ?current)
7       (not (unknown ?rival))
8       (not (pre_simulation_agent ?rival ?current))
9       (rival_action_kill ?rival ?target ?where ?current)
10
11      (alive ?rival )
12      (at ?rival ?where )
13      (at ?target ?where )
14    )
15   :effect
16     (and
17       (not (alive ?target ))
18
19       (pre_simulation_agent ?rival ?current)
20     )
21 )

```

Figure 4.4: Simulations of rival’s *kill* action in the case when the preconditions are fulfilled; *kill* action itself was defined in Figure 4.2

```

1 (:action not_prec_kill
2   :parameters
3     (?target - player ?where - place ?rival - player ?current - number)
4   :precondition
5     (and
6       (time ?current)
7       (not (unknown ?rival))
8       (not (pre_simulation_agent ?rival ?current))
9       (rival_action_kill ?rival ?target ?where ?current)
10
11      (or
12        (not (alive ?rival ))
13        (not (at ?rival ?where ))
14        (not (at ?target ?where ))
15      )
16    )
17   :effect
18     (and
19       (pre_simulation_agent ?rival ?current)
20       (unknown ?rival)
21     )
22 )
23 )

```

Figure 4.5: Simulations of rival’s *kill* action in the case when the preconditions are not fulfilled; *kill* action itself was defined in Figure 4.2

can be translated to several actions, where each of these actions has one element of the disjunction.

The effects of the first action are the same as effects of the simulated actions plus a predicate expressing that simulation has been done. The second action means that the rival cannot perform the action, the plan was interrupted so the effects are only the *unknown*

flag and the predicate expressing that simulation has been done.

```

1 (:action run_rival_unknown
2   :parameters
3     (?x - player ?current - number)
4   :precondition
5     (and
6       (time ?current)
7       (or
8         (unknown ?x)
9         (rival_action_unknown ?x ?current)
10      )
11    )
12  :effect
13    (and
14      (pre_simulation_agent ?x ?current)
15    )
16 )

```

Figure 4.6: Simulations of a rival's action with the *unknown* flag/action

The action in Figure 4.6 covers the last option at simulating rivals' actions. It covers both case, the first one with the *unknown* flag and the second one with the *unknown* action. *Unknown* action can appear in a plan as a result of the merging process.

AGENT'S ACTION

After simulation of all rivals, the planner has to generate one action for the agent.

In Figure 4.7 we can see a compiled *kill* action. Parameters are the same as in the NCDL action definition plus the parameter *?current* for the time. The predicate on the line 6 says that all rivals has been simulated already. The line 7 checks whether the generated plan will not exceed the length of prediction. The line 8 binds the time parameter to the current time. The lines 10 to 12 are implicit preconditions of the *kill* action.

The lines 14 to 22 are the most important in this example. They prevent the planner to plan actions which are in conflict with a rivals' action in the input. If the rival has *unknown* flag, we do not consider his plan, so we do not have to checks his actions. Otherwise we must ensure that rivals' actions are not in conflict. This happens on the lines 16 and 20. They contain enumeration of all conflicted actions. The line 22 indicates this enumeration of conflicted action must be presented for all rivals.

CONTRA ACTION

Blocking all conflicted actions is a rather strict model. If a rival wants to kill the agent, his only chance to defend is to break the preconditions of a rival's *kill* action or any action before (for example to run away before the attack). But if it happens in the first time step, the agent cannot play any defending action, because any defending action is, from definition, in the conflict.

```

1 (:action _kill
2   :parameters
3     (?target - player ?where - place ?current - number)
4   :precondition
5     (and
6       (pre_simulation_done)
7       (not (last_round_finished))
8       (time ?current)
9
10      (alive kalam )
11      (at kalam ?where )
12      (at ?target ?where )
13
14      (or
15        (unknown ?rival_A)
16        (not [conflicted actions of rival_A])
17      )
18      (or
19        (unknown ?rival_B)
20        (not [conflicted actions of rival_B])
21      )
22      ...
23    )
24   :effect
25     (and
26       (not (alive ?target ))
27
28       (not (pre_simulation_done))
29       (action_done)
30       (increase (total-cost) 2)
31     )
32 )

```

Figure 4.7: Compiled standard *kill* action for agent Kalam

Because of this, we came with a concept of *contra actions*. Contra action can interrupt a rival's plan directly. It means that the planner can skip checking of one rival's possible conflicted action and also skip this action in simulation.

In Figure 4.8 we can see definitions of the *kill contra escape* action in the NCDL. The line 3 says that the agent can play the *kill* action contra rival's *escape* action (line 2). The lines 6 and 7 determine what skills affect the probability of successful playing the contra action. It is not probability because the planner always plans the contra action as a successful action. But we want to preserve the principle that the planner should avoid planning actions with "low probability". We will show on the compiled contra action that we again use actions' costs for this purpose. Not listed skills express that probability of a successful contra action is 1.

The lines 3 and 4 contain optional parameters. These parameters must be always in pair and their purpose is to more specify the contra relation. An escape action can be certainly used as a contra action against the kill action. Someone wants to kill me but I can escape. But my escape action cannot stop the murder if I am not the victim. So, in this case the escape action can be contra against the kill action only if the victim of the kill action is the one who will escape.


```

1 <contra>
2 <action_target>escape </action_target>
3 <action_player>kill </action_player>
4 <par>agent </par>
5 <par>target </par>
6 <skill>escape </skill>
7 <skill>escape </skill>
8 </contra>

```

Figure 4.8: NCDL kill contra escape definition

In our example in Figure 4.8, it is said that the *target* parameter of agent's kill action must be the same as the *agent* parameter of the rival's escape action. The *agent* parameter always describes the agent who plays the action.

The example of compiled kill contra escape action is in Figure 4.9. The parameters are (a) parameters of the kill action, (b) parameters of rival's escape action (with *_r* suffix), (c) skills to express the cost of the action and (d) current time.

The preconditions are similar to preconditions in the compiled kill action. They differ in the line 8, where the standard action checks that all rivals were simulated but in the contra action we want to interrupt the rival's action. So the planner simulates all rivals except the one we want to interrupt.

The line 17 corresponds to additional parameters we introduced in the example in Figure 4.8. The line 18 checks that the rival really played the action we want to interrupt. The lines 20 and 21 bind the skill levels.

Lines 23 to 31 prevent possible conflict with another rivals. Actions of rival against who is played the contra action are, of course, excluded from these possible conflicts.

First part of the effects are effects of the kill action itself (line 35). The line 37 removes information about simulation of the rivals. The line 38 sets the *unknown* flag because the plan of the rival was interrupted. It is the same principle as replacing plan with *unknown* actions in the merging algorithm (section 4.5). The line 39 says that the action has been played in this time step.

And the last line 40 increases the cost of the plan. The *parameters* of the action cost are agents' skill and the function *escape_vs_escape__cost* representing the cost of the probability of the success.

GOAL

The last part of compilation is goal handling. In the Figure 4.10 we can see definition of the agent Kalam. It is a part of agents definition from the Figure 4.1. In the last example we will show how to compile Kalam's goal to our system.

For each goal, compilation creates two actions as we can see in Figure 4.11. The first action is planned when the goal is fulfilled, the second one when it is not. The preconditions are straightforward. The lines 5 and 20 check it that the plan is planned for all time steps

```

1 (:action _kill__escape
2  :parameters
3    (?target - player ?where - place
4     ?agent_r - player ?from_r - place ?to_r - place
5     ?levelX0 - level ?levelY0 - level ?current - number)
6  :precondition
7    (and
8      (almost_pre_simulation_done ?agent_r)
9      (not (last_round_finished))
10     (time ?current)
11
12     (alive kalam )
13     (at kalam ?where )
14     (at ?target ?where )
15     (not (married ?target ))
16
17     (= ?target ?agent_r)
18     (rival_action_escape ?agent_r ?from_r ?to_r ?current)
19
20     (skill_escape ?agent_r ?levelX0)
21     (skill_escape kalam ?levelY0)
22
23     (or
24       (unknown ?rival_A)
25       (not [conflicted actions of rival_A])
26     )
27     (or
28       (unknown ?rival_B)
29       (not [conflicted actions of rival_B])
30     )
31     ...
32
33  :effect
34    (and
35      (not (alive ?target ))
36
37      (not (almost_pre_simulation_done ?agent_r))
38      (unknown ?agent_r)
39      (action_done)
40      (increase (total-cost) (escape_vs_escape__cost ?levelX0 ?levelY0))
41    )
42 )

```

Figure 4.9: Kalam’s compiled kill contra escape action

```

1 <agent>
2   <name>kalam</name>
3   <level>3</level>
4
5   <goal>(alive kalam) -      5000 </goal>
6   <goal>(not (alive sumar)) - 5000 </goal>
7 </agent>

```

Figure 4.10: Agent Kalam definition

(in our case it is 10 steps). The lines 7 and 21 represent the goal. In the second action is this goal in the negative form. Both action has effect that confirms the goal was checked and the negative action increases the plan cost by the goal’s penalty. Because we minimize

plan cost, the planner is forced to plan towards fulfilment of the goals.

```

1 (:action goal_0_true
2   :parameters
3     ()
4   :precondition
5     (and
6       (time ten)
7       (alive kalam)
8     )
9   :effect
10    (and
11     (check_goal_0)
12    )
13 )
14
15 (:action goal_0_false
16   :parameters
17     ()
18   :precondition
19     (and
20       (time ten)
21       (not (alive kalam))
22     )
23   :effect
24     (and
25       (check_goal_0)
26       (increase (total-cost) 5000)
27     )
28 )

```

Figure 4.11: Agent Kalam’s compiled goal to survive

It is important to check the goals after the last time step because it is not sufficient to fulfil a goal in some time step but the agent must ensure that the goal will persist in the future. This future, of course, has to be finite and in our system is limited by input argument of the prediction length m .

AUXILIARY FUNCTIONS

Compiled domain contains more action than we just describes but these action has simple functionality, so we will describe them all in this section.

pre_simulate

Preconditions of this action only check that all rivals were simulated in the current time step. The effect is one and it adds predicate (*pre_simulation_done*) with this information.

almost_pre_simulate_[rival_A]

This action is very similar to the previous one with the difference that exists one for each rival and skips this rival in checking rivals’ simulations. The effect adds (*almost_pre_simulation_done [rival_A]*).

finish_round

Preconditions of the *finish_round* action are binding the current and the next time, checking that the action in the current time step was planned and the planner does not close the agent’s plan yet. Effects remove current time predicate and add next time predicate and remove predicate saying that agent’s action was already planned (*action_done*).

finish_last_round

This action is the same as the previous one with one addition effect, which adds the predicates closing agent's plan (*last_round_finished*).

Closing plan is important feature, because it forces planner to plan the non-nop actions before the nop actions, so the story is moving ahead.

__nop__after

The same action as the compiled nop action with the different that it can be planned only after closing a plan. Because this action has the lowest cost, planner will try to close the plan as soon as possible.

finish_game

It only checks handling all goals and adds predicate (*game_finished*), which always must hold in the planner's goal state.

COMPILATION SUMMARY

The compilation how we just describe it does not match the simulation model we use. In our model, the actions in one time step are executed simultaneously. But in the model used in planning compilation, the actions of all rivals are executed at first and then the actions of the agent. This solution was chosen, because planning compilation exactly corresponding to simultaneous model is much more slower that the current one.

4.8 SIMULATION

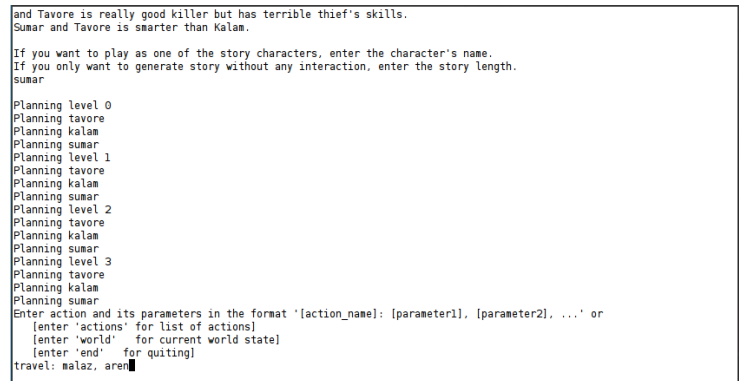
As we said before our system works with the assumption of a fully observable world. It is true in the meaning that all agents know the whole world state. But there is one information they don't know. It is the cognitive level of the rivals. This level determines from what plan the system will take the actions to the simulation (line 12 in Algorithm 1). Because of this, no agent knows what rivals will do.

Simulation itself is the same as the merging algorithm with the difference that the input are plans with the length 1 (it means only one action per agent) and instead of replacing unreachable actions by *unknown* action, it is replaced by the *nop* action.

4.9 STORIES, GAMES AND INTERACTIVITY

Our system is designed to be interactive. The only thing that it is needed to be done is to take user's input as the input of the simulation process.

We wanted to preserve the idea of the used model in our experimental game interface (Figure 4.12). At the beginning, the program shows the description (written by the author of scenario) of world, agents and agent’s goals. Then the user chooses if he (or she) want to only generate a story without any interaction (in this case the input is the number of time steps) or to play as one of story characters (the input is then the name of the character).



```

and Tavore is really good killer but has terrible thief's skills.
Sumar and Tavore is smarter than Kalam.

If you want to play as one of the story characters, enter the character's name.
If you only want to generate story without any interaction, enter the story length.
sumar

Planning level 0
Planning tavore
Planning kalam
Planning sumar
Planning level 1
Planning tavore
Planning kalam
Planning sumar
Planning level 2
Planning tavore
Planning kalam
Planning sumar
Planning level 3
Planning tavore
Planning kalam
Planning sumar
Enter action and its parameters in the format '[action_name]: [parameter1], [parameter2], ...' or
[enter 'actions' for list of actions]
[enter 'world' for current world state]
[enter 'end' for quitting]
travel: malaz, aren

```

Figure 4.12: The screenshot of the interface of the interactive mode

The output is always the same, it is a sequence of tuples, where each tuple represents one time step and contains one action for each agent. If a user plays as a character, he has to enter character’s action and it’s parameters each time step.

4.10 IMPLEMENTATION

The whole implantation is done in *python 3* (Python Software Foundation. Python Language Reference, version 3.3. Available at <http://www.python.org>). Because the complexity of merging and simulation is negligible in comparison to planning and the compilation from NCDL to PDDL runs only once at the beginning, we focus more on the form and the planning performance of the compiled PDDL.

PDDL domain files can grow to thousand lines because of enumeration of all conflicted action. But most of these actions are listed unnecessarily because any rival won’t play them. These rivals’ actions are unknown at the compilation time and they are part of the problem file. PDDL planners of course contains some preprocessor, which can recognize useless preconditions (like our conflicted actions which are not played by any rival) but this preprocessing costs a lot of computation power.

Our system tries to avoid this computation by its own preprocessing of compiled PDDL domain files before every planning task. The system analyses rivals’ actions and then removes remaining actions from conflicted actions’ preconditions from PDDL domains. It is done by keeping complete compiled domain for each agent and using the pruned (on unused rivals’ action) one for planning.

Experiments

In this chapter we will describe and evaluate the experiments performed on our system. In these experiments we focused on two main areas. The time complexity in the dependency on number of actions, numbers of agents etc. And the believability and the attractiveness of generated stories.

5.1 COMPUTATIONAL COMPLEXITY

To measure complexity of the compilation and efficiency of the proposed algorithm, we have prepared a synthetic planning domain. The domain allowed us to variably change the number of agents and/or coupling of their actions. Multiagent planning in general gets (exponentially) harder with increasing coupling as proven by (Brafman and Domshlak, 2008), therefore our experiments show the results both for coupled and decoupled problem variations.

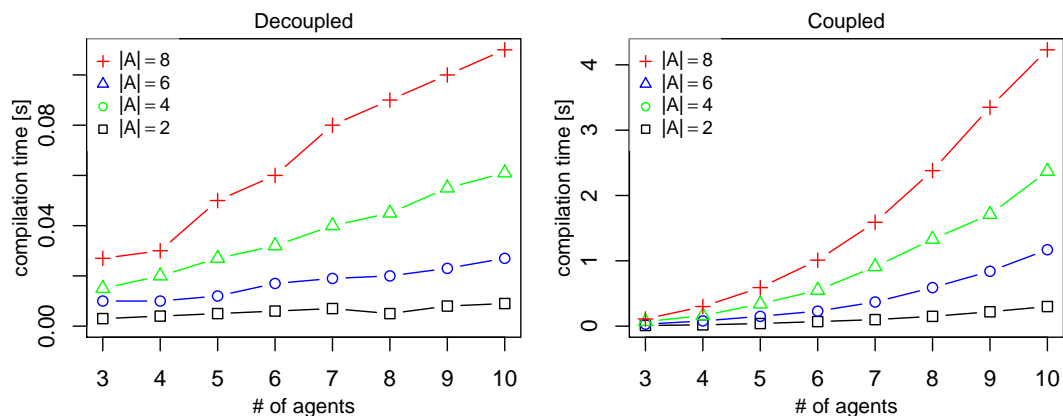


Figure 5.1: Compilation time for increasing number of agents and different problem sizes ($|A|$ is the number of actions)

The relation of compilation time and increasing number of agents is depicted in Figure 5.1. The trends show that the compilation time is roughly linear or polynomial to number of agents in decoupled version and exponential in the coupled problems. As the size of the problem increases, the compilation time increases accordingly.

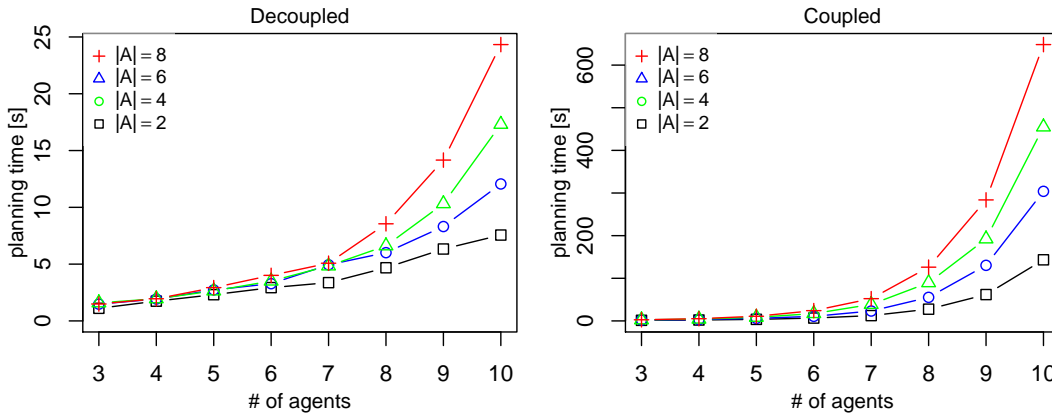


Figure 5.2: Runtime of the algorithm for increasing number of agents and different problem sizes ($|A|$ is the number of actions)

The efficiency of the proposed algorithm was analyzed for increasing number of agents as well. Figure 5.2 shows how is the planning runtime dependent on the number of agents. In the decoupled case, the trends show rather exponential growth with exception of the smallest problem with 2 actions per agent. In the coupled problems, the growth is clearly exponential which is an expected behavior considering results of (Brafman and Domshlak, 2008).

COMPARISON WITH THE STATE OF THE ART

In Figure 5.3 we can see the comparison between our Multiagent narrative planning system, Haslum's compiled IPOCL planner and the IMPRACTical system on the *Aladin* problem. Using cognitive level 3 is sufficient for generating stories comparable with other systems. It shows that finding our solution takes about 15 seconds and it is 3 times faster than Haslum's solutions but more than 7 times slower than IMPRACTical. But unlike others, our system was designed to be interactive in the form of computer game. But if we want to use these systems in this interactive way, the only possibility is the replanning. It means that each iteration would take the same time as one planning. But not in our system. Our system consists of these iterations implicitly. And in the *Aladin* problem one iteration takes only about 3 seconds, which is comparable with the IMPRACTical system.

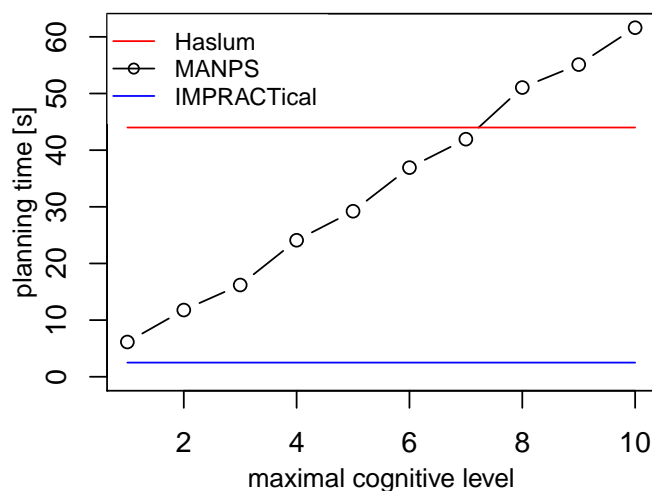


Figure 5.3: Comparison of planning times of the *Aladin* domain

5.2 SURVEY

The perception of narratives is very subjective among the people (Busselle and Bilandzic, 2008). We prepared a survey to assess the two most important properties of any narrative. The believability and the attractiveness.

The survey contains six stories generated by our system. For each story, each respondent has to answer 2 questions.

1. What do you think about the believability of the story...?
 - (a) Very believable
 - (b) Believable
 - (c) Implausible
 - (d) Very implausible

2. What do you think about the attractiveness of the Story...?
 - (a) Very interesting
 - (b) Interesting
 - (c) Boring
 - (d) Very boring

All stories were generated by the same domain and each agent had the same goals in each story. The differences between stories were agents' skills, agents' cognitive levels (expressed by agent's cleverness) and agents' initial position.

The domains contains 5 actions. (a) travel, (b) escape, (c) kill, (d) marry (only men can "play" this action), (e) do nothing.

At the beginning of each story, the respondent got story background, consisting of 2 paragraph. We can see the first one, which is identical for all stories, in Figure 5.1. The second paragraph differs in each and we will introduce it separately for each story.

There are 5 cities in the world of Wu. They are linearly connected in the sequence Malaz - Aren - Darujhistan - Quon Tali - Letheras. Sumar is man, who wants to marry Tavore. Tavore is a princess and she wants to marry Sumar. Kalam is knight of the king and his task is to prevent marriage between Tavore and Sumar.

Table 5.1: The common part of the story background of all generated narratives.

STORY #1

Sumar is at Malaz, Kalam at Darujhistan and Tavore at Letheras. Sumar is very good thief but poor killer. Kalam's skills are average and Tavore is really good killer but has terrible thief's skills. Sumar and Tavore is smarter than Kalam.

Generated story:

- Day 1:** Sumar travels from Malaz to Aren.
- Day 2:** Sumar travels from Aren to Darujhistan.
- Day 3:** Kalam tries to kill Sumar, but Sumar escapes from Darujhistan to Quon Tali.
- Day 4:** Sumar travels from Quon Tali to Letheras. Kalam travels from Darujhistan to Quon Tali.
- Day 5:** Kalam travels from Quon Tali to Letheras.
- Day 6:** Kalam tries to kill Sumar, but Sumar escapes from Letheras to Quon Tali.
- Day 7:** Kalam tries to marry Tavore, but Tavore kills Kalam. Sumar travels from Quon Tali from Letheras.
- Day 8:** Sumar marries Tavore.

The graphs in Figure 5.4 show that more than 75% of respondents answered that the story #1 is believable and interesting.

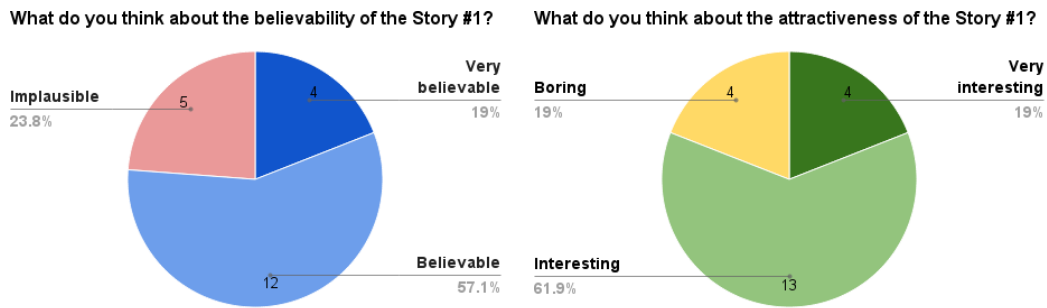


Figure 5.4: The believability and the attractiveness of the story #1

STORY #2

Sumar is at Malaz, Kalam at Darujhistan and Tavore at Letheras. Sumar is very good thief but poor killer. Kalam's skills are average and Tavore is really good killer but has terrible thief's skills. Kalam is smarter than Sumar and Tavore.

Generated story:

Day 1: Sumar travels from Malaz to Aren.

Day 2: Sumar travels from Aren to Darujhistan.

Day 3: Sumar tries to travel from Darujhistan to Quon Tali, but Kalam kills him.

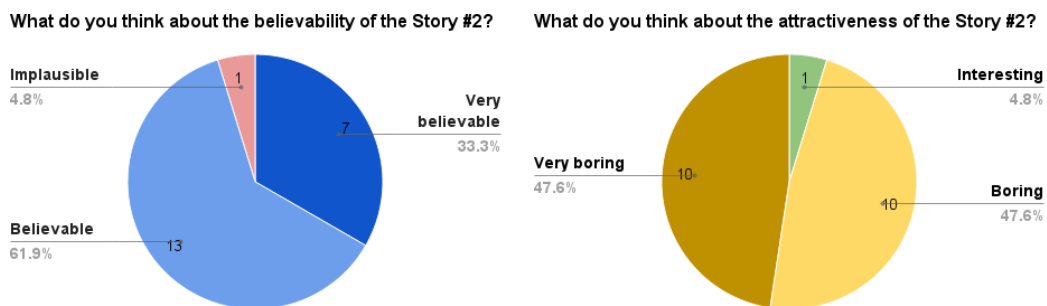


Figure 5.5: The believability and the attractiveness of the story #2

Figure 5.5 shows trend which occurs in some other generated stories too and it is the high believability but the low attractiveness.

STORY #3

Sumar is at Malaz, Kalam at Darujhistan and Tavore at Letheras.
 Sumar is very poor thief but very good killer. Kalam's skills are average
 and Tavore is really poor killer but has excelent thief's skills.
 Sumar and Tavore is smarter than Kalam.

Generated story:

Day 1: Sumar travels from Malaz to Aren.

Day 2: Sumar travels from Aren to Darujhistan.

Day 3: Kalam tries to kill Sumar, but Sumar kills Kalam first.

Day 4: Sumar travels from Darujhistan to Quon Tali.

Day 5: Sumar travels from Quon Tali to Letheras.

Day 6: Sumar marries Tavore.

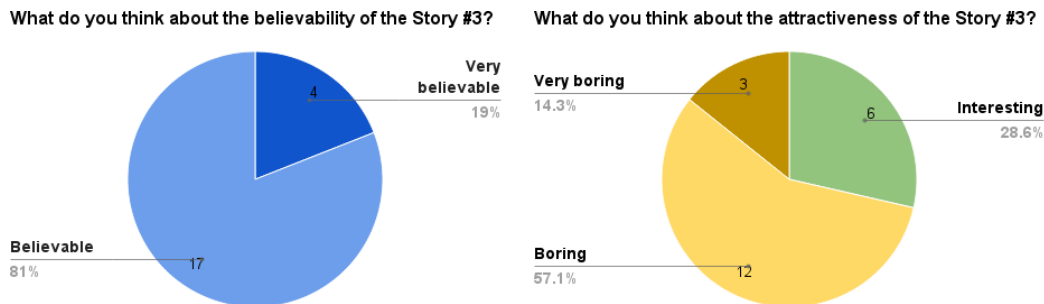


Figure 5.6: The believability and the attractiveness of the story #3

Although the length of the story #3 (Figure 5.6) is twice the length of the story #2, both of these stories have very similar results. The stories does not contains any story twist or story climax. It can be reason of the high believability but it is certainly the reason for the low attractiveness.

STORY #4

Sumar is at Malaz, Tavore and Kalam at Darujhistan.
 Sumar is very poor thief and poor killer. Kalam's skills are average
 and Tavore is really good killer and has excelent thief's skills.
 Tavore is smarter than Kalam and Sumar.

Generated story:

Day 1: Sumar travel from Malaz to Aren. Kalam tries to marry Tavore, but Tavore kills him.

Day 2: Sumar travel from Aren to Darujhistan.

Day 3: Sumar marries Tavore.

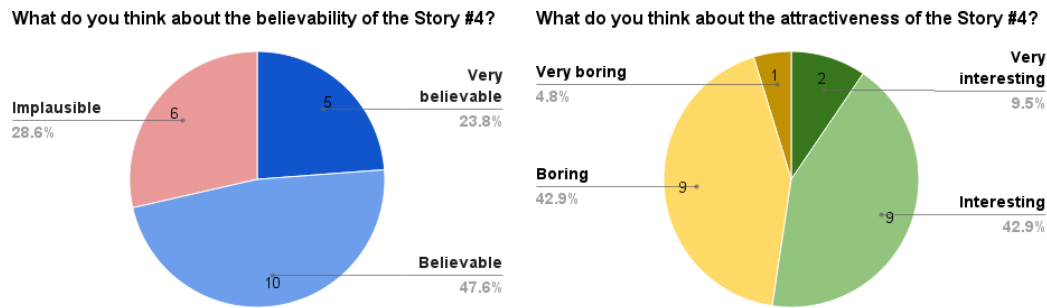


Figure 5.7: The believability and the attractiveness of the story #4

Figure 5.7 shows the big subjectivity in perception of the attractiveness of narratives. One half of respondents considers the story #6 interesting and the second one boring.

STORY #5

Sumar is at Malaz, Kalam at Darujhistan and Tavore at Letheras.
 Sumar is very poor thief and poor killer. Kalam's skills are average and Tavore is really good killer and has excelent thief's skills.
 Tavore is smarter than Kalam and Kalam is smarter than Sumar.

Generated story:

Day 1: Sumar travels from Malaz to Aren.

Day 2: Sumar travels from Aren to Darujhistan.

Day 3: Sumar tries to travel from Darujhistan to Quon Tali, but Kalam kills him.

According Figure 5.8 the story #5 is another good example of the high believability but the low attractiveness.

STORY #6

Sumar is at Malaz, Kalam at Darujhistan and Tavore at Letheras.
 Sumar is very good thief and excelent killer. Kalam's skills are average and Tavore is really poor killer and has terrible thief's skills.
 Kalam is smarter than Sumar and Tavore.

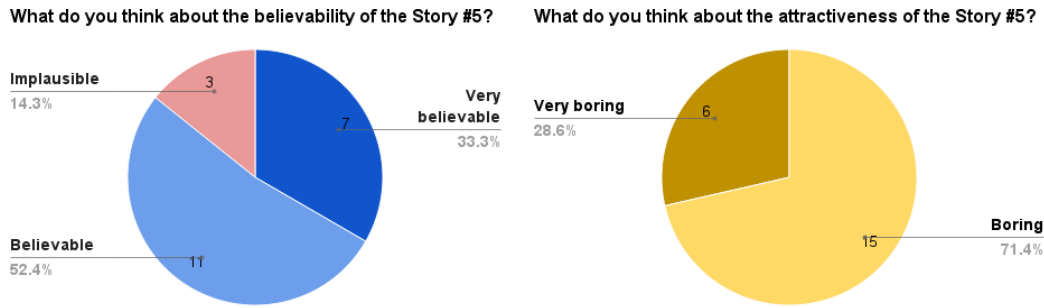


Figure 5.8: The believability and the attractiveness of the story #5

Generated story:

Day 1: Sumar travels from Malaz to Aren.

Day 2: Sumar travels from Aren to Darujhistan.

Day 3: Kalam travels from Darujhistan to Quon Tali. Sumar escapes from Darujhistan to Quon Tali.

Day 4: Kalam travels from Quon Tali to Letheras. Sumar escapes from Quon Tali to Letheras.

Day 5: Kalam tries to kill Sumar, but Sumar kills Kalam first.

Day 6: Sumar marries Tavore.

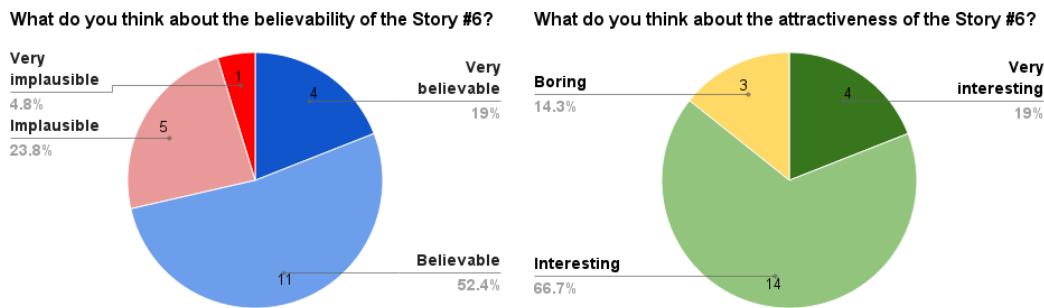


Figure 5.9: The believability and the attractiveness of the story #6

The story #6 is shorter than the story #1 but still has high attractiveness as well as the believability.

SURVEY CONCLUSION

The survey, in which answered 21 respondents, showed that even short narratives generated by our system and very simple domain can be both believable and interesting. Generation of one iteration (1 day) took about 35 s. This time is sufficient for computer games e.g.

dynamically generated side quest. And even if the story is not interesting, most of people still consider it believable, which is in the domain of computer games more important.

5.3 UNCOVERED SHORTCOMINGS

During experiments we have discovered two main shortcomings of the system. The first is mentioned at the end of the compilation section. The model used in our planner simulates rivals' actions first and then plans agent's action. The result can be that the planner plans such an action, which cannot be executed in the simultaneous model used in merging and simulation. If we skip these actions the result story remains consistent and believable.

Second one is a little bit more difficult. Because of detecting conflicted actions, user cannot use advanced expressions like *for each loop* or *when condition* in actions' effects definitions. The result is that user (the creator of the domain) is forced to use different actions for very similar concepts. This itself is not critical but it causes unbelievable behaviour of agents. We will explain it on an example.

Let us have world, where agents can kill each other and marry each other and we want to ensure that the marriage of killed agents is cancelled. It means that effects of the kill action has to contain an effect cancelling of the marriage. The planner has to know the partner of the killed victim to remove the correct marriage status from the world state. It means that the partner have to be bind in the action's preconditions. But this is not possible when the victim has no partner. The only solution is to have two variants of the kill action, one for single victim and one for married victim. The problem situation is following. Single agent A has prediction that the agent B wants to kill him. Agent A wants to be alive. It means that it has to interrupt the *kill_single* action of the agent B. One of the options is not to satisfy precondition of agent's B kill action. And because this version is limited only to single victims, agent A can interrupt the agent's B action by marrying someone.

But this kind of behaviour is not what we want in a believable story. Because we cannot, usually, save our lives just by marrying someone.

None of these phenomenons appears often and usually do not have any influence on the final story, but the removal of them would be good direction of future improvements.

FUTURE WORK

In the section 5.3 we described 2 problem which should be solved in some future extension of our system.

Because the time complexity of multiagent planning grows exponentially with growing number of agents and/or actions, the good direction for future improvement would be the optimizations of the compilation from NCDL to PDDL to lower the planning time and by this allow using our system in bigger domains.

CHAPTER 6

Conclusion

Our work showed that single-agent planning can be applicable for multiagent narrative planning. But before we can use classical planners, we have to transform narrative domain to suitable form covering all aspect of good story. This is the first part of our system, the compiler from NCDL to PDDL.

The second part is modelling the behaviour of other characters. We chose the Cognitive Hierarchy, which has been theoretically and experimentally proven to be good model describing human behaviour.

And the last part of this thesis cover the experimentally testing of our system on humans. In the survey containing several generated stories people had to rate the believability and the attractiveness of generated stories. The survey showed that our system is capable to generate believable and interesting narratives.

Bibliography

- Ronen I. Brafman and Carmel Domshlak. From one to many: Planning for loosely coupled multi-agent systems. In *Proceedings of ICAPS'08*, pages 28–35, 2008.
- R. Busselle and H. Bilandzic. Fictionality and perceived realism in experiencing stories: A model of narrative comprehension and engagement. 18:255–280+, 2008.
- Colin F. Camerer, Teck Hua Ho, and Juin-Kuan Chong. A cognitive hierarchy model of games. *The Quarterly Journal of Economics*, 119(3):861–898, 2004. URL <http://EconPapers.repec.org/RePEc:tpr:qjecon:v:119:y:2004:i:3:p:861-898>.
- Hubie Chen. Bounded rationality, strategy simplification, and equilibrium. *Int. J. Game Theory*, 42(3):593–611, 2013. doi: 10.1007/s00182-011-0293-7. URL <http://dx.doi.org/10.1007/s00182-011-0293-7>.
- Patrik Haslum. Narrative planning: Compilations to classical planning. *J. Artif. Int. Res.*, 44(1):383–395, May 2012. ISSN 1076-9757. URL <http://dl.acm.org/citation.cfm?id=2387933.2387941>.
- Malte Helmert. The fast downward planning system. *CoRR*, abs/1109.6051, 2011. URL <http://arxiv.org/abs/1109.6051>.
- Julie Porteous and Marc Cavazza. Controlling narrative generation with planning trajectories: The role of constraints. In IdoA. Iurgel, Nelson Zagalo, and Paolo Petta, editors, *Interactive Storytelling*, volume 5915 of *Lecture Notes in Computer Science*, pages 234–245. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-10642-2.
- Mark O. Riedl and R. Michael Young. Narrative planning: Balancing plot and character. *J. Artif. Int. Res.*, 39(1):217–268, September 2010. ISSN 1076-9757. URL <http://dl.acm.org/citation.cfm?id=1946417.1946422>.
- Jonathan Teutenberg and Julie Porteous. Efficient intent-based narrative generation using multiple planning agents. In Maria L. Gini, Onn Shehory, Takayuki Ito, and Catholijn M.

Jonker, editors, *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13, Saint Paul, MN, USA, May 6-10, 2013*, pages 603–610. IFAAMAS, 2013. ISBN 978-1-4503-1993-5. URL <http://dl.acm.org/citation.cfm?id=2485016>.

Testing machine configuration

A.1 PROCESSOR

Intel Core i5 520M, 2,4 GHz, 3 MB L3 Cache

A.2 RAM

4 GB DDR3 1066 MHz

A.3 OS

Kubuntu 13.10, 64-bit

APPENDIX **B**

CD content

The CD contains an electronic copy of this thesis. It also contains the Fast Downward planner and source files of our system in following structure:

- **/Fast-Downward-8bb82316af04** – the Fast Downward planner
- **/Cognitive_Kingdom** – python scripts of our system and testing scenario

Running the system

C.1 GLOBAL REQUIREMENTS

- Linux OS system

C.2 FAST DOWNWARD COMPILATION

First of all you have to build the Fast Downward planner. The planner has several dependencies:

- G++ (GCC) \geq 4.8
- python \geq 3.3
- bison
- flex

To compile the planner run:

```
1 cd Fast-Downward-8bb82316af04/src
2 ./build_all
```

To verify the functionality of the planner run:

```
1 cd Fast-Downward-8bb82316af04/src
2 ./plan ../benchmarks/trucks/domain.pddl ../benchmarks/trucks/p01.pddl
```

Then the penultimate line of the output should be:

```
1 Solution found.
```


C.3 GENERATING NARRATIVES

After successfully planner compilation you can start to generate narratives. Run:

```
1 cd Cognitive_Kingdom
2 [python] play.py scenarios/sc1/
```

And then follow the instructions in the system interface.

C.4 EDITING THE SCENARIO

The file

```
1 Cognitive_Kingdom/scenarios/sc1/game.cogk
```

describes the example scenario. Feel free to change any agents parameters like cognitive level, goals, skills or initial world state. The syntax and semantics of *.cogk* file is described in the compilation section (4.7).