REAL-TIME DYNAMIC PARTIAL ORDER PLANNING FOR MEMORY RECONSTRUCTION IN AUTONOMOUS VIRTUAL AGENTS

BY BHUVANA CHANDRA INAMPUDI

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Dr. Mubbasir Kapadia

and approved by

New Brunswick, New Jersey

January, 2017

ABSTRACT OF THE THESIS

REAL-TIME DYNAMIC PARTIAL ORDER PLANNING FOR MEMORY RECONSTRUCTION IN AUTONOMOUS VIRTUAL AGENTS

by BHUVANA CHANDRA INAMPUDI

Thesis Director: Dr. Mubbasir Kapadia

This paper introduces a novel approach to generate narratives from the memories of in-game agents. We propose an agent framework to accommodate memory and perception in virtual agents. This system extracts agent auto-biographic memories as multiple partial narratives, and generates possible complete narratives. We employ a novel narrative merging and extrapolation technique to generate unique complete narratives based on partial narratives of multiple agents. This is used to generate unique narratives for specific narrative constraints. To generate narratives for massive open-world scenes we introduce a novel algorithm that generates dynamic partial plans of large action spaces in real-time. These plans repairs as per the user actions, there by generating a unique narrative for every different user interaction. We conducted a comparative study of our dynamic planning technique with existing planning techniques. Also, we tested our agent framework for a complex environment to verify its robustness.

Acknowledgements

I would like to thank my adviser, Prof. Mubbasir Kapadia, without whose help and guidance I would not have achieved this. I would also like to thank Michael Dwyer, Nilay Chakraborty and Kuan Wang who helped me build and test this application. I would like to thank Prof. Norman Badler who gave me the opportunity to work at SIGLAB, where I connected with amazing people and learned a lot. Last but not least, I would also like to thank Dr.Lotzi Boloni, who provided assistance in helping me understand the Xapagy cognitive architecture.

Dedication

This work is dedicated to everyone who supported me through this endeavor.

Table of Contents

Chapter 1
Introduction

Video games these days are blurring the lines between virtual and real world by using extremely realistic graphics and exhaustive open-world environments. Despite all the above, very few games include strong narrative discourse that change according to player interactions. The existing techniques allows a game to either have a strong narrative with a selected decision points, or a dynamic narrative by sacrificing the possible user interactions.

Narratives for the present day games are pre-scripted and coded into the game. So, to make a game's narrative truly dynamic, content editors have to script narrative for every possible interaction for each step of the simulation. As it is very expensive and impractical to do that, we explore automated narrative generation techniques.

Partial order planners are being used to generate complex Narratives with a lot of user interaction. Since the plan time increases with the number of actions, using POP for open world narratives is not time efficient. Furthermore, if those massive game worlds are to accommodate free-form user interaction the plan should consider all the possible conflicts to the Narrative. This in turn increases the plan space and effort required from content editors to craft a complete narrative.

This thesis assumes the concepts of partial order planners as a basis and tackles the problem of generating automated narratives by introducing

1. An agent architecture to perceive events and record auto-biographic memories, discussed in section 3.2

2. An approach to generate narrative discourse from the partial memories of one or more agents (See section 10)

3. A dynamic planner that repairs the narratives according to user interactions, as discussed in *chapter 4*

1.1 Dynamic Planning

The motivation behind developing such a technique is to accommodates free-form user interactions in a generated narrative with minimum re-planning. Such an approach can be used to re-create any historic event and analyze how changes in various interactions would have effected that event, or to generate interactive games.

For such a system, the planner should be able to generate plans in real-time by using characters and actions that are relevant to that scene. The system should re-plan for any changes in the environment in real time, there by accommodating free-form user interactions. To validate or use such a planner the system architecture should be modular, robust and scalable to generate extensive narratives.

Though there hasn't been a single approach which tackles all the challenges, there has been attempts to address individual problems.[1] introduced Parameterized Behavior Trees to improve the modularity and re-usability of Behavior Trees. [17] used Interactive Behavior trees to incorporate free-form user interaction and conflict resolution. Decision trees can be used to generate believable by narratives, but they cannot practically support free-form user interaction.

To reduce the plan time, we introduced a heuristic to define related actions, thereby decreasing the plan space, thus decreasing the plan time. Inspired from the D* and Partial Order Planner, we can up with an approach to incorporate dynamic characteristic to the planner. This algorithm will repair the plan when a user interaction disrupts the current plot.

The plan times using the improved dynamic planner are compared against the plan times of a conventional partial planner with IBTs. The time of execution and the optimality of the solution are taken as the measures to quantify the benefits of the proposed solution.

1.2 Narrative Reconstruction from Agent Memories

This approach focuses on generating narratives from the past experiences/memories of the in-game characters. Using an approach like that, multiple possible narratives can be auto generated based on the past user interactions or a pre-determined plot, by extracting the memories from one or more virtual agents in the simulation environment.

For such a system to exist, one must first define an agent framework that allows agent to observe its surroundings and record the events that it perceives. Since the perception and memory generation in an agent are triggered by events, we call this Event-Based Agent Framework (**EBAF**). Also, the memory representation should be in a way such that, the narrative extraction can be done with out compromising the basic understanding of memory. For example, one cant store the reason for an event to happen in the memory as that should be left to the reasoning.

The proposed algorithm extracts the auto-biographic memories that are created by EBAF, and generates possible narratives. The proposed system is tested for robustness using multiple pre-authored narratives.

We describe the general architecture of the proposed approaches in chapter 3. Chapter 4 & 5 describes the implementation of the Dynamic Planner and Memory Reconstruction in detail, respectively. A comparative study of our Dynamic Planner is done in chapter 6.

Chapter 2

Related Work

There has been several works on manual authoring techniques for narrative generation. Work of [14] describes predefined behaviors, thus even small changes to the narrative result in monolithic work. Structure of story graphs [5] allows to author huge believable narratives with very little user interactions. In story graphs, the user interactions are facilitated in the form of choices at key points in the narrative.

Partial order planning is often used to auto generate narratives. [4] describes the basic algorithm of a partial order planner to generate partial plans. A comparative study between total-order and partial-order planners has been done by [16]. Techniques like D* are developed to incorporate make total-order planners dynamic. [13] describes about techniques to incorporate Anytime Dynamic nature in total ordering planners. [19] discusses a novel intent-driven planning technique to generative coherent narratives.

On the other hand [1] shows how Behavior trees can be parameterized to improve the modularity and re-usability of the narratives. This helps in re-using similar story arcs to generate rather complex narratives. [12] involves learning script-like narrative knowledge from crowd sourcing to generate narrative. [15] explores design issues of constructing a plot, creating AI characters, and using a director in an interactive storytelling environment. For implementing free-form user interactions [17] models an architecture with Behavior Trees. It also provides a conflict resolution algorithm in Interactive Behavior Trees. [7] describes a framework where virtual objects aid the user to accomplish a pre-programmed possible interaction. An event-centric framework for directing interactive narratives is shown in [22].

[8] presents a modular, and flexible platform for authoring purposeful human characters in a virtual environment. The frame work of [9] generates complicated behaviors

between interacting actors in a user authored scenario. Mubbasir Kapadia et. al. in [9] implemented a behavior authoring framework that provides domain control to user, for multi-agent simuations. [21] describes a framework for mitigating individual agent complexity while retaining agent diversity. The work of [14], provides a study for creating believable agents.

A survey of various approaches to quantify the interestingness of a narrative is provided by [11]. [18] is one of the early attempts to measure the interestingness of a computer generated plot. It compares ideal story tension graphs to the one that generated plot to find the interestingness. Although the framework in [10] can be used to quickly generate complex and believable narratives, the memory requirements increase exponentially as the number of possible actions increase.

Very few attempts have been made to introduce a memory model in virtual agents, which can also be used to generate Narrative discourse. ADAPT is an agent architecture [23] for designing and author- ing functional, purposeful human characters in a rich virtual envi- ronment. It is versatile platform to implement character animation and navigation functionalities. [24] discuss a event driven agent framework for steering in a crowd. [20] proposes an agent model in which some agents can be affected by peer agents, by introducing the concept of *'sphere of influence'*.

SPREAD [6], is a novel agent-based sound perception model that employs a discretized sound packet representation with acoustic features including amplitude, frequency range, and duration. Lotzi Boloni introduced a cognition architecture [3] where agent auto-biographic memories are stored as concept overlays, and reasoning in done using statistical approaches. He introduced a new pidgin language [2] which is used for Language translation. Though Xapi is primitive enough to generate a Narrative Discourse, we chose not to use Xapagy as its architecture does not support spatial reasoning, and quantifying properties is not necessary for our approach.

Chapter 3

Framework Overview

3.1 Dynamic Planner

To build an open world game that supports auto-narrative generation for non-player-characters (NPC) and free form user interaction for player controlled characters (PC), the design of the system should be modular, scalable and robust. To satisfy those constraints we introduce an architecture [see figure 3.1] that can be used by both NPCs and PCs. An in-depth explanation of the architecture is provided in the following sections of this paper.

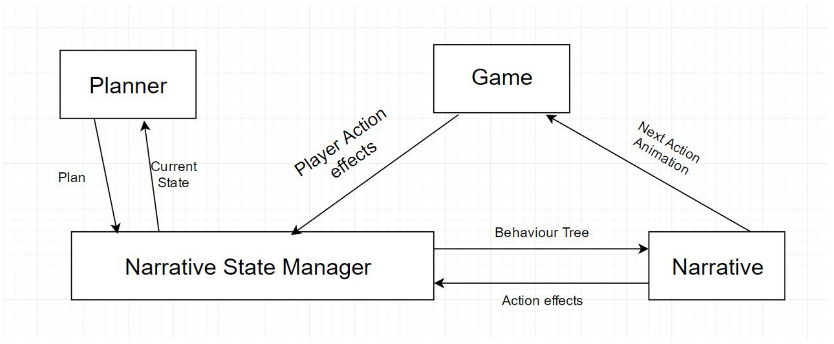


Figure 3.1: Frame work of the Dynamic Planner

3.1.1 Narrative State Manager

The Narrative State Manager(NSM) keeps track of the state of the game. It maintains the current plan/narrative, and state of the objects in the scene. Objects states are updated when either

1. An action in the Narrative plot is executed, or

2. User performs an interaction with the simulation environment

Whenever the Object states are updated, NSM checks if the current plan is still valid. If the plan is invalidated, NSM sends the current state to the Planner to repair the Narrative. Using the plan received from the Planner, NSM generates a Behavior Tree and sends it to Narrative component for execution.

3.1.2 Planner

As the name suggests, Planner module contains the logic to generate a partial plan given a current-state and the goal state. This game is a open world game which supports free-form user interaction, hence the planner should be able to provide partial plans for a massive environment in real-time. Chapter 4 describes the approach adopted to achieve that.

3.1.3 Narrative

Narrative module has the execution logic for a Behavior Tree. Given a Behavior Tree, the Narrative module execute the nodes and sends

1. The effects/state changes to the NSM, and
2. The related animation of that node to the Game UI

3.1.4 Game UI

Game UI displays the animations provided by the Narrative. It also provides the user interactions for the objects in the scene. The player can interact with the objects of the scene and the effects/state changes caused by that interaction will be sent back to NSM. The interactions supported by an Object in scene is determined by the affordances that object has. Since both NSM and Game UI depend on the affordances of the objects in scene, adding an affordance to the object will automatically create a new user interaction.

3.2 Event-based Memory Generation

To generate autobiographic memories from agent experiences and to extract Narratives from those memories, the architecture shown in figure 3.2 is employed.

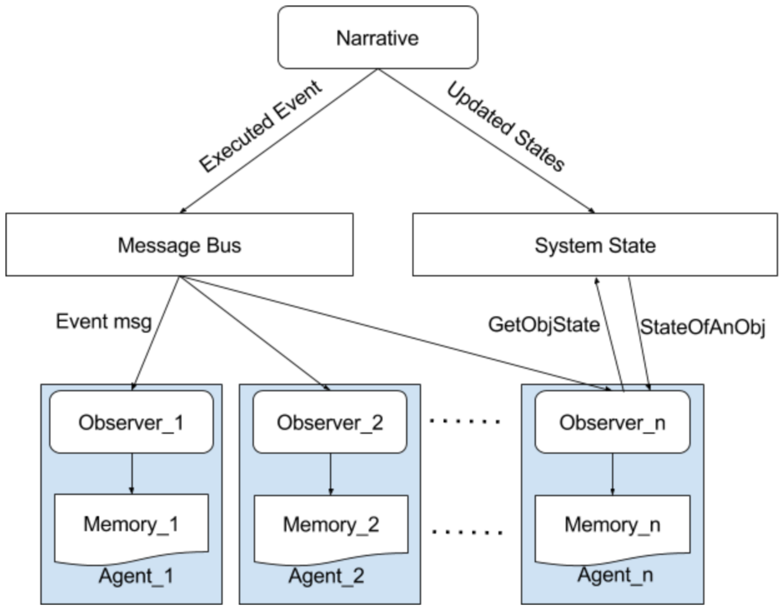


Figure 3.2: Frame work of the event-based architecture

3.2.1 Narrative Module

Given the Behavior Tree of the narrative that has to be simulated, Narrative module executes the tree nodes and sends

1. The executed event details (along with the time-stamp) to the Message Bus, and
2. The updated object states to the System State module

Narrative state sends a message for both start and end of an event.

3.2.2 Message Bus

When message bus receives event notifications from Narrative module, it broadcasts them to all agents. Notifications received in a simulation cycle are pushed into an array of messages. At the end of the simulation cycle all the messages in the message bus are published. Message bus is refreshed/cleared at the start of every cycle. So, at any given time, message bus contains notifications for the events happened at that simulation cycle.

3.2.3 System State

System state keeps track of all the object states in the simulation. Starting with the initial state of the system, the System State keeps updating the object states after every event execution. So, if queried, the System State module returns the state of an object at that point of time in the simulation.

3.2.4 Virtual Agent

Each virtual agent has a cognition module attached to it. Cognition includes perception, memory, reasoning etc. Though in our current system we implemented just the perception, memory and spatial reasoning, this architecture can be further extended to incorporate other elements of human cognition.

Observer Module handles the perception of a virtual agent. Perception can be visual, auditory, written etc. We just considered visual perception for our system. So, when a observer notices an event notification from the Message Bus, it verifies whether that event is related to the objects in its field of view. If it is, observer gets the state of the object from the System State and creates a memory with the object state, perception data and event notification. These memories are stored as Agent Auto-biographic Memories in the **Memory module**.

Chapter 4

Dynamic Planner

In the following sections we have made an attempt to describe the implementation of the dynamic planner. Below are some terminology that we will be using to describe our system.

Smart Object Objects in a scene that support a behavior or action. Smart Objects have affordances associated with them. (Ex: Door, Doctor, Gun etc.)

Affordance Affordance is an action that involves two objects (Affordee and Affordant). Affordee enables the action, and Affordant exerts that action. For example, Person Opens the Door. In this, Person and Door are smart objects, Open is an affordance. Door is the affordee of that action while Person is the affordant. Throughout this article, terms Affordance and Action are used interchangeably. Every affordance a has a set of Pre-Conditions and Effects ($< \{\Phi\}, \{\Omega\} >$). Pre-Conditions are required for the action to execute, and Effects are the state changes produced by the action after execution. Every affordance in our system also has a PBT [1] associated with it, that provides corresponding animation for the affordance.

Causal Link Two actions a_1 and a_2 have a causal link over a condition ϕ, if a pre-condition ϕ of a_2 is satisfied by executing a_1. In other words, executing a_1 produces an effect ϕ which allows a_2 to execute. An entry in causal link is denoted as $< a_1, \phi, a_2 >$.

Ordering Constraint An order constraint $a_1 \prec a_2$ between actions a_1 and a_2 indicates that a_1 should execute before a_2

4.1 Partial Order Planner

Since our algorithm is based on a Partial Order Planner, We introduced this section to discuss the basic properties and working mechanics of a partial order planer. Partial order planners generate partial order of actions i.e. they commit to ordering only when forced. In contrast,a total order plan provides ordering of all actions even if not necessary. So, a solution of a partial order planner can generate one or more total order plans.

Given a start state and a goal state, a partial planner gives the affordances that needs to be executed and the ordering constraints(if any) to reach the goal state. It does that by solving for each pre-condition in goal state by adding affordance that satisfy them. That is, for each precondition, action pair $< \phi, a >$, find an action a_0 whose effect satisfies the precondition in the pair. Then add all the preconditions of a_0 to the agenda. By repeating this process until there are no open preconditions left, partial planner finds a partial plan between start and end states. See [4] for a detailed description of the algorithm.

In case of a conventional Partial Order Planner the plan time increases exponentially with the plan space. So, for an open-world environment where there are numerous objects with multiple affordances, using a ordinary partial planner would be impractical.

4.2 Accelerated POP

We tried to make improvements to the existing partial planner so that it can be used online in open world games. We introduced a heuristic ***ActionRelationMap*** (ARM) to reduce the plan time. Given a condition ϕ, action-relation map gives all the possible actions that can satisfy that condition, i.e. $ARM[\phi] = \forall a \in \mathrm{A} s.t., \phi \in a \rightarrow \{\Omega\}$.

Algorithm 1 describes the construction of the action relation map. All the effects of the actions in action space Aare iterated in lines 2 & 3 of the algorithm, and $ARM$is populated in the following lines. If an action contains a condition in its effects, it implies that action can be used to satisfy the condition. By using ARMas a heuristic to search of possible actions for an agenda item $< \phi, a >$, the search space can be drastically reduced, there by reducing the plan times.

Since the possible actions remains constant for given Affordances and SmartObjects, $ARM$is calculated only when new a Affordance or a SmartObjects is added to the scene

Algorithm 1: Algorithm to construct the action-relation map

```
1 ConstructActionRealtions ()
2     foreach $a \in ActionSpace$ do
3         foreach $\phi \in a \rightarrow \{\Omega\}$ do
4             if $\phi \in ARM \rightarrow Keys$ then
5                 $ARM[\phi] = ARM[\phi] \cup a$
6             else
7                 $ARM[\phi] = a$
```

and stored. $ARM$is loaded once during the start of the simulation and used through out the simulation. This decreases the plan time furthermore.

4.3 Dynamic POP

All the existing techniques either preemptively plan for user interactions or replan the Narrative from scratch. Neither of those approaches will be efficient to update the Narrative in real time, when dealing real world simulation (as the number of objects and interactions will be prolific). To tackle that issue, we propose an algorithm which incorporates the concepts of D* algorithm [13] in Partial Order Planner. Below is some related terminology.

Running Causal Links A causal link $l < a_1, \phi, a_2 >$ is said to be a running causal link rl if action a_1 is running and a_2 is not yet executed.

Heuristic We are using the action relations (ARM) described in section 4.2 as the heuristic for the algorithm.

Over-Consistent Links A causal link will be in over-consistent state if the condition of the causal link is already present in the current state of the system.

Under-Consistent Links A running causal link is said to be under consistent if the condition is negated by the current state. A user action negating a running causal link will result in an under consistent link.

4.3.1 D-POP algorithm

The planner keeps executing on open conditions to be satisfied. Initial plan is generated using A-POP defined in section 4.2. Later, for every user action, NSM checks for any over consistent and under consistent states. If an in consistent state is found, it is added to the open conditions of the planner. In case of over consistency, *consistency propagation* is performed before the plan repair is done. Consistency propagation is discussed in detail in section 4.3.3

The Dynamic planner is implemented using the algorithm described in Algorithm 2. Agenda $\Phi_{open}$is a stack that contains all the open condition, action pairs. A is a set of all the actions currently in the plan, while O and L contains all the ordering constraints and causal links respectively.

The planner instantiates A, Φ_{open}, O and L with the a_{start} and a_{goal} states (see lines 2-5 of Algorithm 2). If $\Phi_{open}$is not empty, one open condition is removed from it and checked if any existing action satisfies the open condition (see Algorithm 3). If no existing affordance satisfies the open condition, the heuristic defined in section 4.2 is used to get an action from the *ActionSpace*(lines 10,11 in Algorithm 2). The suggested affordance is added and any conflict are resolved by calling **ResolveConflict** method (lines 12-16 in Algorithm 2). The preconditions of the newly added affordance are added to the open conditions (lines 17,18 in Algorithm 2). Causal link for the condition is added and the plan is again checked for conflicts

Conflict Resolution Conflict arises if the effect of an affordances $a$contradicts with the condition on a causal link $< a_1, \phi, a_2 >$. In such a case, the conflict is resolved by imposing a constraint such the $a$either happens before a_1 ($a \prec a_1$) or $a$happens after a_2 ($a_2 \prec a$)

4.3.2 Update for User Interactions

The dynamic planner repairs the plan for user interactions. Algorithm 5 determines whether an user action a_{ui}damages the consistency of the current narrative. If it does, the $\Phi_{open}$are updated the Algorithm 2 is used to repair the plan. The narrative is made inconsistent if

1. The user action satisfies a condition on causal link, resulting Over-Consistent links, or

2. It negates the condition on a causal link, thereby generating Under-Consistent links

Algorithm 5 checks for over consistent links in lines 3 through 8. The over consistency is propagated in the narrative by using consistency propagation algorithm discussed in section 4.3.3. Running causal links are verified for possible under-consistent links and open conditions are updated accordingly in lines 11-15.

4.3.3 Consistency Propagation

Over-consistent links can make some states in the narrative as redundant. For example, in figure 4.1, the User Action has made C_1 over-consistent. Hence, $Action_1$ is already satisfied by the User Action. So, $Action_2$ becomes redundant and is no longer required in the narrative. Removing redundant states from the Narrative is called Consistency Propagation. Algorithm 6 describes the logic we employed to find and remove the redundant states from the narrative (Note: ocActions in Algorithm 6 denotes the set of over consistent actions).

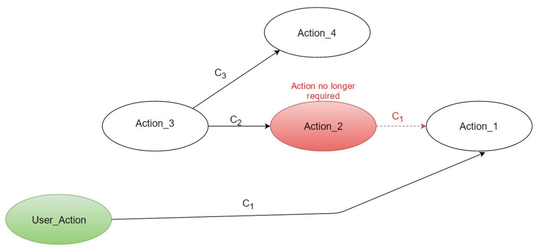


Figure 4.1: Example of an over consistent link leading to redundant states

Algorithm 2: DynamicPlanner()

```
begin
    $\Phi_{open} = \{< \phi, a_{goal} > |\forall\phi \in goal \rightarrow \{\Phi\}\}$
    $A = \{a_{start}, a_{goal}\}$
    $O = \{a_{start} \prec a_{goal}\}$
    $L = \emptyset$
    $isInitialPlanGenrated = false$
    repeat
        if $\Phi_{open} \neq \emptyset$ then
            $< \phi, a_c >=$ pop$(\Phi_{open})$
            if $!IsConditionAlreadySatisfied(\phi, a_s)$ then
                $a_s =$ SelectActionFromRelations$(\phi)$
                $A = A \cup a_s$
                $O = O \cup \{a_{start} \prec a_s\}$
                foreach $l \in L$ do
                    if $l\,is\,not\,executed$ then
                        $O =$ ResolveConflicts$(l, a_s, O)$
                foreach $\Phi \in a_s \rightarrow \{\Phi\}$ do
                    $\Phi_{open} = \Phi_{open} \cup \{< a_s, \Phi >\}$
            $O = O \cup \{a_s \prec a_c\}$
            $L = L \cup \{< a_s, \phi, a_c >\}$
            foreach $a \in A$ do
                if $a\,is\,not\,executed$ then
                    $O =$ ResolveConflicts$(< a_s, \phi, a_c >, a, O)$
        else
            $isInitialPlanGenerated = true$
        if $useraction\,a_{ui}\,\&\&\,isInitialPlanGenerated$ then
            UpdatePlannerSpace$(a_{ui})$
    until $forever$
```

Algorithm 3: Method to check if plan has an action that satisfies a condition

```
IsConditionAlreadySatisfied($\phi_c$, out $a_s$)
    foreach $a \in$ A do
        foreach $\phi \in a \rightarrow \{\Omega\}$ do
            if $\phi == \phi_c$ then
                $a_s = a$
                return true
    return false
```

Algorithm 4: Method to get action from action-relations map

```
SelectActionFromRelations($\phi$)
    if $ARM[\phi] \neq \emptyset$ then
        foreach $ainARM[\phi]$ do
            if $a \notin$ A then
                return $a$
```

Algorithm 5: Update planner space after user interaction a_{ui}

```
UpdatePlannerSpace($a_{ui}$)
    $A = A \cup a_{ui}$
    foreach $l \in L$ do
        if $l \rightarrow \phi \in a_{ui} \rightarrow \{\Omega\}$ then
            remove $l$from L
            $ocActions = ocActions \cup l \rightarrow \{a_1\}$
            foreach $\phi \in l \rightarrow a_2 \rightarrow \{\Phi\}$ do
                $\Phi_{open} = \Phi_{open} \cup \{< l \rightarrow a_2, \phi >\}$
    if $ocActions \neq \emptyset$ then
        PropagateConsistency(ocActions)
    foreach $rl \in RL$ do
        if $\neg rl \rightarrow \phi \in a_{ui} \rightarrow \{\Omega\}$ then
            remove $rl$ from RL
            foreach $\phi \in rl \rightarrow a_2 \rightarrow \{\Phi\}$ do
                $\Phi_{open} = \Phi_{open} \cup \{< rl \rightarrow a_2, \phi >\}$
```

Algorithm 6: Algorithm to propagate consistency in BT

```
PropagateConsistency(ocActions)
    foreach $a \in ocActions$ do
        if $a \notin \{L \leftarrow \{a_1\}\}$ then
            RemoveActionFromAffordancesAndConstarints($a$)
            foreach $l \in L$ do
                if $a \in l \leftarrow \{a_2\}$ then
                    $ocActions = ocActions \cup l \leftarrow \{a_1\}$
    if $ocActions \neq \emptyset$ then
        PropagateConsistency(ocActions)
    else
        return
```

Chapter 5

Narrative Generation from Agent Memories

In section 3.2 the general framework of our agent model is introduced. In the following chapters we explain how an agent memory is represented, how EBAF is used to generate memories and the implementation details of narrative discourse generation.

5.1 Memory Representation

The implementation is broadly divided into two parts viz., **Memory Generation** in agents and **Narrative Reconstruction** from the agents. Before starting with any algorithmic details, it is vital that we declare what a Memory object looks like. A Memory or 'A Memory Event' (m) has following data

- $memoryName$ - The memory name is a sentence of three words $'actor_1 - actionName - actor_2'$ (For example : John opens Door). So, an affordance can be extracted from a memory as each action has a unique *'actionName'*, and each smartObject has a unique *'name'*.
- *type* - A memory event can either be start event or an end event denoted by m_{start} and m_{end} respectively. By default, start events only have $startTime$ and end events only have $endTime$.
- $startTime$ - The time at which the event began.
- $endTime$ - The times at which the event has completed.
- *actorOneState(S_1)* - The state of actorOne noticed at the time of event. State of an object is a set of conditions $\{\phi\}$ related to that object at a given time.
- *actorTwoState(S_2)* - The state of actorOne noticed at the time of event.

- *perceptionData* - Data recorded by the observers (see section 3.2.4). Currently the system only recorded *visualPerceptionData*. It records the spatial information of the objects in the memory event at the time happened.

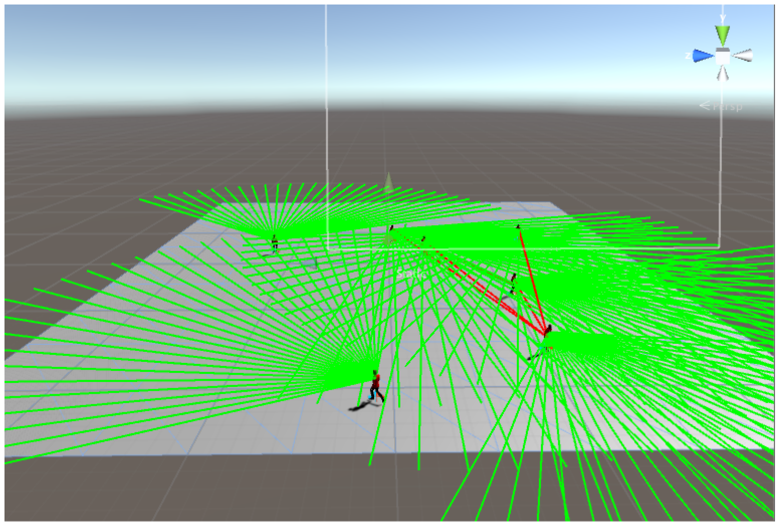
Since every affordance has a Behavior Tree associated with it, working with affordances indirectly means working with BehaviorTrees. So, memory objects have all the data that is required to create a behavior trees, which are used to represent the Narrative

5.2 Memory Generation

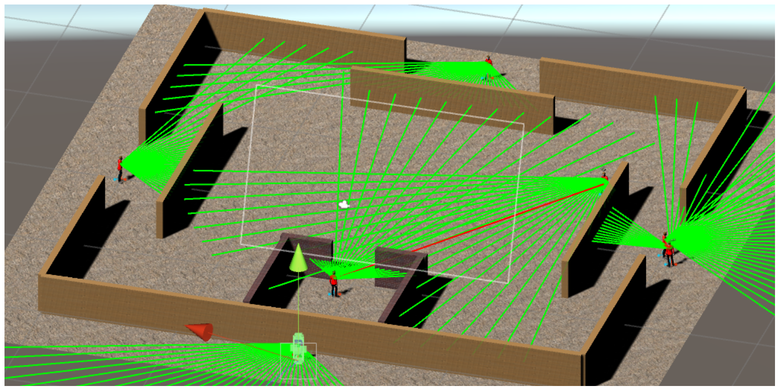
Every agent in the scene that has cognition has an observer attached to it. The observer constantly checks the objects in its field of view. Whenever an event notification appears in the MessageBus, observer verifies if any the object of that event are present in its field of view (See figure.5.1). If yes, then the observer creates a memory object by populating the spatial information and getting the object state from SystemState. All the memories of an agent are stored in its auto-biographic memory (M_{obs}). Pseudo-code for this is defined in Algorithm 7.

Algorithm 7: Method to generate autobiographic memories of an agent

```
1  RecordMemories()
2    repeat
3      $msgs = MsgBus \rightarrow \{msgs\}$
4      foreach $msg \in msgs$ do
5        foreach $Obj_{msg} \in msg \rightarrow \{objs\}$ do
6          if $Obj_{msg} \in \boldsymbol{ObjectsInFOV}$ then
7            $S_1 = \textbf{GetObjStateFromSystemState}(msg \rightarrow \{Obj_1\})$
8            $S_2 = \textbf{GetObjStateFromSystemState}(msg \rightarrow \{Obj_2\})$
9            $M_{obs} = M_{obs} \cup memory(msg, \textbf{SpatialInfo}, S_1, S_2)$
10   until $MsgBus.hasMsgs()$
```



(a)



(b)

Figure 5.1: The field of view of an observer is denoted by green lines. Red lines in figures (a) & (b) indicate that an object has entered the field of view of the observer.

5.3 Narrative Reconstruction

Generating possible narratives can be dived into three steps(see figure 5.2). First the selected agent memories are merged and start & end time are populated. Then, the consistency of the generated narrative is validated. If there are any inconsistencies, **dynamicPlanner** is used to extrapolate the missing parts of the narrative. These steps are discussed in detail following sections.

5.3.1 Merging Agent Memories

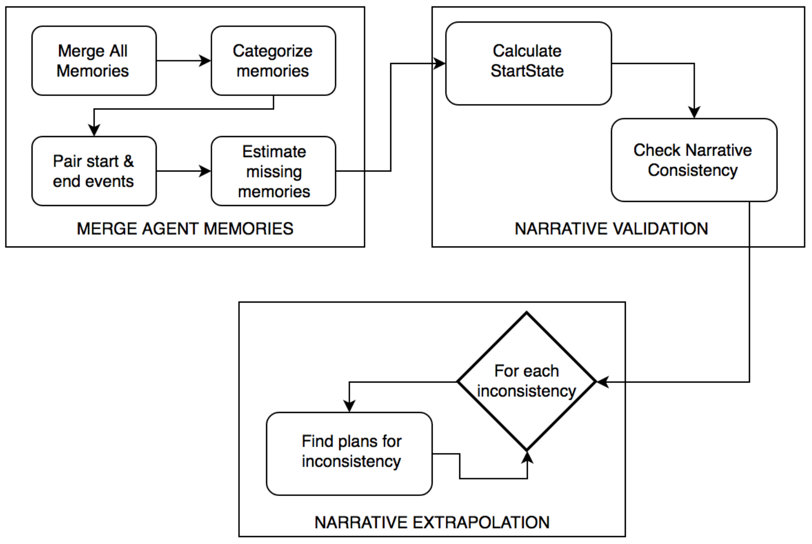


Figure 5.2: Overview of Narrative Reconstruction process

In this module, the memories from the selected agents are merged and the resulting start & end events are paired. For memories with no existing counter part, the corresponding event is estimated and then added.

As discussed in section 5.1, memory events are of two types

- $START_MEM$type - Memories that are created when an event is initiated.
- END_MEMtype - Memories that are created when viewing the completion of an event/action.

Having a respective start event for every end event is necessary as these pairs are used to validate the narrative in section 18.

Algorithm 8 is used for merging, pairing & estimation of memories. First, the memory traces of all selected agents ($SelectedObserverTraces$) are extracted and merged as shown in Algorithm 9. Then the Memories are sorted into start and end events (M_{start}& M_{end}respectively) based on the $type$of the memory event (see Algorithm 10). Figure 5.3 shows the step-by-step process of how the extracted memories are categorized using an example.

The pairing of the events is done in two steps. If the end event for a start memory is already present in the current memory collection, the start & end times for the events

Algorithm 8: Method to merge memories, pair events and estimate times

```
1 begin
2     MergeAgentMemories()
3     CategorizeMemories()
4     PairStartAndEndEvents()
5     Sort $M_{start}$ by $m_{start} \rightarrow startTime$
6     Sort $M_{end}$ by $m_{end} \rightarrow endTime$
```

Algorithm 9: Method to merge memories extracted from different agents

```
1 MergeAgentMemories()
2     foreach $M_{obs} \in SelectedObserverTraces$ do
3         foreach $m \in M_{obs}$ do
4             if $m \notin M$ then
5                 $M = M \cup m$
```

Algorithm 10: Method to classify memories into start and end events

```
1 CategorizeMemories()
2     foreach $m \in M$ do
3         if $m \rightarrow type == START\_MEM$ then
4             $M_{start} = M_{start} \cup \{m\}$
5         else
6             $M_{end} = M_{end} \cup \{m\}$
```

are populated using their counter part (See lines 4-7 of Algorithm 11). For memories whose counter-parts does not exist in the agent memories, an estimate for the time taken for that event is calculated using **EstimateTimeForMemory** function. The time estimate for an action is defined in that affordance definition. For example, *GoTo* action will have an estimation function described as $[\,distanceToTheLocation\,/\,agentVelocity\,]$. Also, an estimated memory (m_{Est}) is created for this, which will be used for Narrative Validation. Lines 9-18 of Algorithm 11 describe the implementation of the estimation,

Algorithm 11: Method to pair start and end events and populate start and end times. If the counter part of a memory does not exist, the time is estimated.

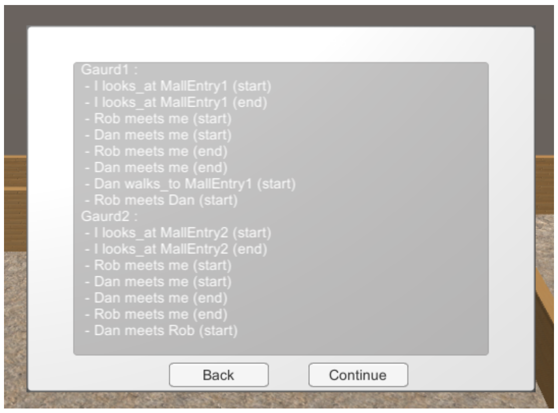
```
1  **PairStartAndEndEvents**()
2      foreach $m_{start} \in M_{start}$ do
3          if $m_{start} \rightarrow endTime == -1$ then
4              $m_{end} = \textbf{FindCounterPartForMemory}(m_{start})$
5              if $m_{end} \rightarrow Exists()$ then
6                  $m_{start} \rightarrow endTime = m_{end} \rightarrow endTime$
7                  $m_{end} \rightarrow startTime = m_{start} \rightarrow startTime$
8              else
9                  $timeForCompletion = \textbf{EstimateTimeForMemory}(m_{start})$
10                 $m_{start} \rightarrow endTime = m_{start} \rightarrow startTime + timeForCompletion$
11                 $m_{Est} = \textbf{CreateEstimatedEndEvent}(m_{end})$
12                 $M_{end} = M_{end} \cup \{m_{Est}\}$
13     foreach $m_{end} \in M_{end}$ do
14         if $m_{end} \rightarrow startTime == -1$ then
15             $timeForCompletion = \textbf{EstimateTimeForMemory}(m_{end})$
16             $m_{end} \rightarrow startTime = m_{end} \rightarrow endTime - timeForCompletion$
17             $m_{Est} = \textbf{CreateEstimatedStartEvent}(m_{end})$
18             $M_{start} = M_{start} \cup \{m_{Est}\}$
```

and figure 5.4 shows an example of the estimated event.

By sorting the start events (M_{start}) we get the temporal structure of the narrative. In section 18 we check whether the narrative deduced from the agent memories is complete/consistent.

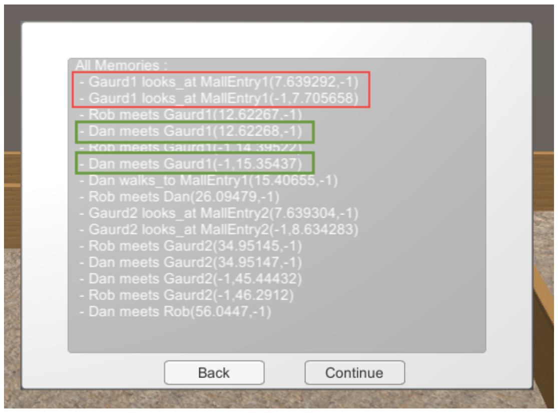
5.3.2 Narrative Validation

In **Narrative Validation** phase we check whether the narrative generated is consistent. Since the goal is to construct a Narrative on which a partial planner can work, this phase also have to generate causal links between the memory events. Also in order



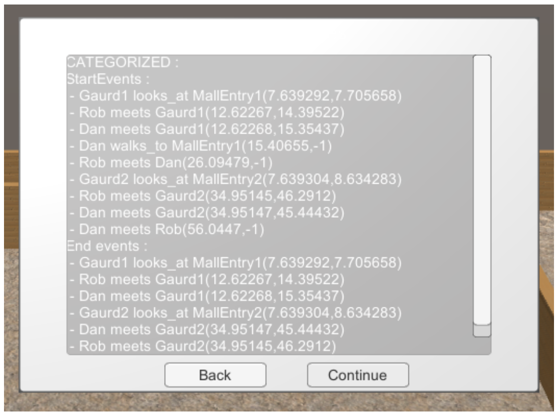
```
Gaurd1 :
 - I looks_at MallEntry1 (start)
 - I looks_at MallEntry1 (end)
 - Rob meets me (start)
 - Dan meets me (start)
 - Rob meets me (end)
 - Dan meets me (end)
 - Dan walks_to MallEntry1 (start)
 - Rob meets Dan (start)
Gaurd2 :
 - I looks_at MallEntry2 (start)
 - I looks_at MallEntry2 (end)
 - Rob meets me (start)
 - Dan meets me (start)
 - Dan meets me (end)
 - Rob meets me (end)
 - Dan meets Rob (start)
```

(a)



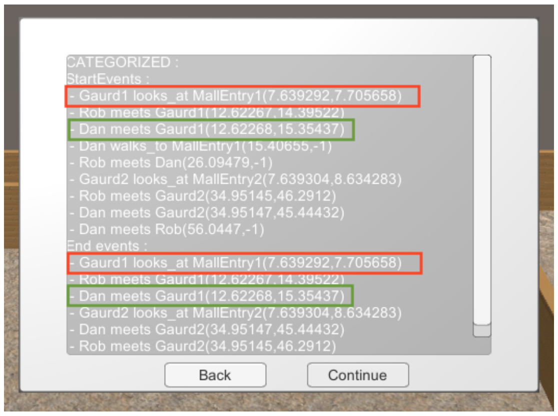
```
All Memories :
 - Gaurd1 looks_at MallEntry1(7.639292,-1)
 - Gaurd1 looks_at MallEntry1(-1,7.705658)
 - Rob meets Gaurd1(12.62267,-1)
 - Dan meets Gaurd1(12.62268,-1)
 - Rob meets Gaurd1(-1,14.39522)
 - Dan meets Gaurd1(-1,15.35437)
 - Dan walks_to MallEntry1(15.40655,-1)
 - Rob meets Dan(26.09479,-1)
 - Gaurd2 looks_at MallEntry2(7.639304,-1)
 - Gaurd2 looks_at MallEntry2(-1,8.634283)
 - Rob meets Gaurd2(34.95145,-1)
 - Dan meets Gaurd2(34.95147,-1)
 - Dan meets Gaurd2(-1,45.44432)
 - Rob meets Gaurd2(-1,46.2912)
 - Dan meets Rob(56.0447,-1)
```

(b)



```
CATEGORIZED :
StartEvents :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Dan walks_to MallEntry1(15.40655,-1)
 - Rob meets Dan(26.09479,-1)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Rob meets Gaurd2(34.95145,46.2912)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Dan meets Rob(56.0447,-1)
End events :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Rob meets Gaurd2(34.95145,46.2912)
```

(c)



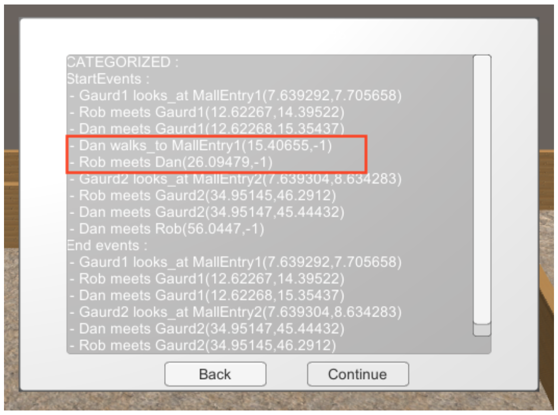
```
CATEGORIZED :
StartEvents :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Dan walks_to MallEntry1(15.40655,-1)
 - Rob meets Dan(26.09479,-1)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Rob meets Gaurd2(34.95145,46.2912)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Dan meets Rob(56.0447,-1)
End events :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Rob meets Gaurd2(34.95145,46.2912)
```

(d)

Figure 5.3: Figure (a) shows the memory events in the auto-biographic memory of agents. In figure (b) you can see that, start memories have no end time(= -1) and end memories have no start time. Figure (c) shows the events after categorizing into start and end events, and in figure (d) the existing memory events are paired and start & times are populated.

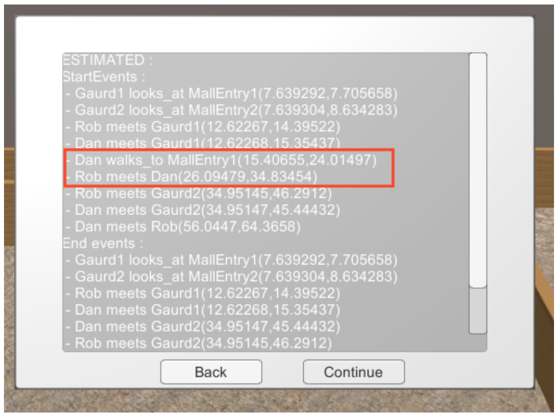
to simulate the narrative, the start state of the objects before the narrative execution has to be determined. Summarizing everything, this phase does the following things

- Check whether the Narrative is consistent.
- Derive possible start state of the simulation.
- Establish causal links between memory events.
- Determine inconsistencies in the Narrative, as inconsistent causal links.



```
CATEGORIZED :
StartEvents :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Dan walks_to MallEntry1(15.40655,-1)
 - Rob meets Dan(26.09479,-1)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Rob meets Gaurd2(34.95145,46.2912)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Dan meets Rob(56.0447,-1)
End events :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Rob meets Gaurd2(34.95145,46.2912)
```

(a)



```
ESTIMATED :
StartEvents :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Dan walks_to MallEntry1(15.40655,24.01497)
 - Rob meets Dan(26.09479,34.83454)
 - Rob meets Gaurd2(34.95145,46.2912)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Dan meets Rob(56.0447,64.3658)
End events :
 - Gaurd1 looks_at MallEntry1(7.639292,7.705658)
 - Gaurd2 looks_at MallEntry2(7.639304,8.634283)
 - Rob meets Gaurd1(12.62267,14.39522)
 - Dan meets Gaurd1(12.62268,15.35437)
 - Dan meets Gaurd2(34.95147,45.44432)
 - Rob meets Gaurd2(34.95145,46.2912)
```

(b)

Figure 5.4: The estimated times for the events in figure (a) are calculated and updated as shown in figure (b).

To calculate the start state of the system is to calculate the start state of all the objects in the system. This is calculated by determining the state of each object at their first occurrence($FirstOcc[Obj]$) in the narrative. Algorithm 12 describes the implementation details for calculating the start state of the narrative. First, the initial occurrence of each object in start memories is stored in a FirstOccurrences map ($FirstOcc$) [see lines 2-13]. If an end event happened before a start event, it means that the start state we assumed for that object is not actually the start state of that object as an event happened before that. So, we remove that object from the FirstOccurrences map [see lines 14-21]. Please note that we did not consider the estimated events while calculating the start state, as the estimated events do not the actual state of objects. Finally, in lines 22 & 23, we merge the object states to determine the start state of the narrative(Φ_{start}).

Now that we have determined the start state, we next check the consistency of the narrative and form the causal links. A narrative is inconsistent/incomplete if one or more events are missing in the discourse. In other words, the store is incomplete if the object state changes in between two consecutive events. For example, in two consecutive events E_1 and E_2, at the end of event E_1 if character *Rob* is at location L_1 and, before the start of E_2 he is at L_2 - then an inconsistency in the narrative arises between E_1 and E_2. The proposed algorithm verifies the consistency by updating the narratives

Algorithm 12: Method to calculate the possible start state of the narrative

```
1  CalculateStartState()
2      foreach $m_{start} \in M_{start}$ do
3          if $!m_{start} \rightarrow IsEstimated()$ then
4              $Obj_1 = m_{start} \rightarrow actor_1$
5              if $Obj_1 \notin FirstOcc$ then
6                  $FirstOcc[Obj_1] = m_{start}$
7              else if $m_{start} \rightarrow startTime < FirstOcc[Obj_1] \rightarrow startTime$ then
8                  $FirstOcc[Obj_1] = m_{start}$
9              $Obj_2 = m_{start} \rightarrow actor_2$
10             if $Obj_2 \notin FirstOcc$ then
11                 $FirstOcc[Obj_2] = m_{start}$
12             else if $m_{start} \rightarrow startTime < FirstOcc[Obj_2] \rightarrow startTime$ then
13                 $FirstOcc[Obj_2] = m_{start}$
       /* Remove the object if it appeared in an end event before it
          appeared in a start event */
14     foreach $m_{end} \in M_{end}$ do
15         if $!m_{end} \rightarrow IsEstimated()$ then
16             $Obj_1 = m_{end} \rightarrow actor_1$
17             if $(Obj_1 \in FirstOcc)$ && $(m_{end} \rightarrow endTime < FirstOcc[Obj_1] \rightarrow$
               $startTime)$ then
18                 Remove $FirstOcc[Obj_1]$ from $FirstOcc$
19             $Obj_2 = m_{end} \rightarrow actor_2$
20             if $(Obj_2 \in FirstOcc)$ && $(m_{end} \rightarrow endTime < FirstOcc[Obj_2] \rightarrow$
               $startTime)$ then
21                 Remove $FirstOcc[Obj_2]$ from $FirstOcc$
22     foreach $Obj \in FirstOcc \rightarrow Keys$ do
23         $\Phi_{start} = \Phi_{start} \cup \{FirstOcc[Obj] \rightarrow ObjectState\}$
```

state(Ω_{open}) by adding effects(Ω) and preconditions(Φ) of affordances, derived from end and start events respectively in a chronological manner(See Algorithm 15).

By now, it is evident that an affordance can be derived from a memory using the $memoryName$. A start-end memory pair have the same affordance, but while executing the start event we verify $\Omega_{open}$with preconditions(line 8 of Algorithm 15) and while executing end event $\Omega_{open}$is updated with the effects of that affordance(line ll of Algorithm 15).

In our algorithm Ω_{open} is defined as the set of action, effect pairs($\{< a, \phi >\}$) where a is the last executed action with ϕ as an effect. $\Omega_{open}$is initialized with the effects of the start affordance(a_{start}).

Updating with effects(Algorithm 13) : While updating the Ω_{open} with effects of an action,

- If the condition already exists in Ω_{open}, replace the action with the new action(lines 11-14).
- For every contradicting $< a, \phi >$ in Ω_{open}, remove it and replace it with the new action, effect entry(lines 6-10).
- If the condition is not present in Ω_{open}, create a new entry(line 16).

Updated narrative state(Ω_{open}) is useful to generate causal links and determine inconsistencies while verifying with action preconditions.

Verifying with preconditions(Algorithm 14) : The preconditions of an action are verified with current narrative state when a start memory is being processed.

- If a precondition Φ is present in the Ω_{open}, then a causal link is formed with that action over Φ (line 7).
- If a $< a, \phi >$contradicts with the Φ, then it means there is an inconsistency in the story. Hence an inconsistent link is added (line 11).
- If a $\Phi$is not present in the Ω_{open}, it means that a part of narrative that gives rise to that condition is missing. So, an inconsistency is added (line 15).

Finally if the number of inconsistencies($\#IL$) in more than 0, then the narrative is inconsistent. We generated agent memories for a sample narrative show in this *video*. As shown in figure 5.5, start state is derived and examples of consistent and inconsistent narratives are shown in figures 5.6 & 5.7 respectively.

Algorithm 13: Method to add effects to Narrative state

```
AddEffectsToNarrativeState($m_{end}$)
    $conditionExists = false$
    $a_e = m_{end} \rightarrow GetAffordance()$
    foreach $\Omega \in a_e \rightarrow \{\Omega\}$ do
        foreach $< a, \phi > \in \Omega_{open}$ do
            if $\Omega$ contradicts $\phi$ then
                $\phi = \Omega$
                $a = a_e$
                $conditionExists = true$
                break
            else if $\Omega == \phi$ then
                $a = a_e$
                $conditionExists = true$
                break
        if $!conditionExists$ then
            $\Omega_{open} = \Omega_{open} \cup < a_e, \Omega >$
        else
            $conditionExists = false$
```

5.3.3 Narrative Extrapolation

In this phase, each inconsistency generated by Algorithm 15 in **Narrative Validation** phase is fed to a planner to generate possible narratives. We use a slightly tweaked version of the dynamic planner defined in chapter 4 to generate the plans.

Algorithm 14: Method to check the consistency current narrative state(Ω_{open}) for a start memory and add causal links

```
1  UpdateCurrentNarrativeStateForStartMemory($m_{start}$)
2      $conditionExists = false$
3      $a_s = m_{start} \rightarrow GetAffordance()$
4      foreach $\Phi \in a_s \rightarrow \{\Phi\}$ do
5          foreach $< a, \phi > \in \Omega_{open}$ do
6              if $\Phi == \phi$ then
7                  $\mathrm{L} = \mathrm{L} \cup < a, \Phi, a_s >$
8                  $conditionExists = true$
9                  break
10             else if $\Phi$ contradicts $\phi$ then
11                 $IL = IL \cup < a, \Phi, a_s >$
12                 $\phi = \Phi$ $conditionExists = true$
13                 break
14         if $!conditionExists$ then
15             $IL = IL \cup < a_{start}, \Phi, a_s >$
16         else
17             $conditionExists = false$
```

Algorithm 15: Method to check the narrative consistency and form causal links

```
CheckNarrativeConsistency()
    $i = 0, j = 0$
    $isNarrativeConsistent = true$
    CalculateStartState()
    $a_{start}$ = CreateAffordaceWithEffects($\Phi_{start}$)
    repeat
        if $M_{start}[i] \rightarrow startTime < M_{end}[j] \rightarrow endTime$ then
            UpdateCurrentNarrativeStateForStartMemory($M_{start}[i]$)
            $i = i + 1$
        else
            AddEffectsToNarrativeState($M_{end}[j]$)
            $j = j + 1$
    until ($i < \#M_{start}$ && $j < \#M_{end}$)
    if $\#IL > 0$ then
        $isNarrativeConsistent = false$
```

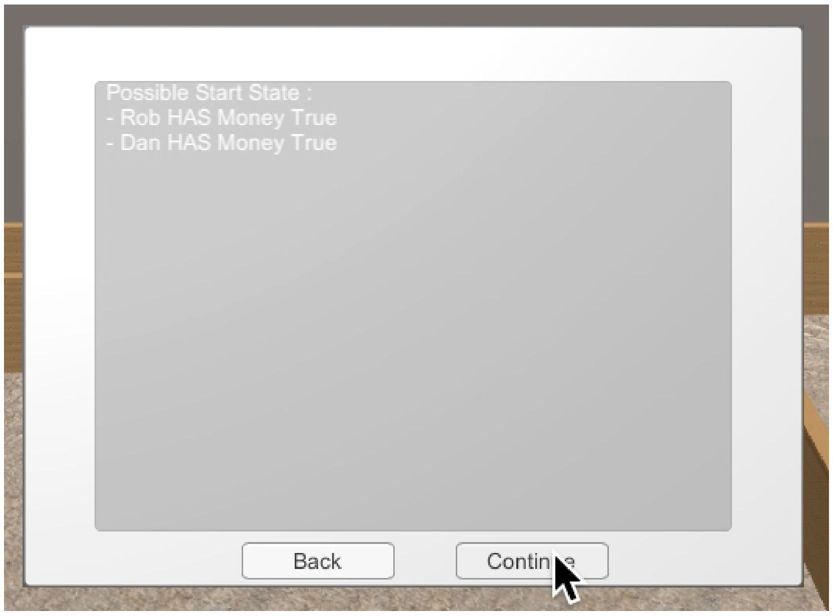

Possible Start State :
- Rob HAS Money True
- Dan HAS Money True

Figure 5.5: Displaying the start state (*full video*)

The dynamic planner defined in section 4.3 stops after finding a plan. But for our case, we tweak the algorithm to do an exhaustive search all possible plans. We populate an initial partial plan(π_i) by generating ordering constraints as shown in Algorithm 16. We deduce that an action a_1 happens before a_2 is a_2 starts after a_1 finishes(see line 8 of Algorithm 16).

For each inconsistency in IL, a set of possible partial plans(Π) is generated by executing the **dynamicPlanner** in plan space π(line 6 of Algorithm 17). Then the user selects of the plans(Narrative discourse) and the $\pi$is updated accordingly(see line 8 of Algorithm 17). This procedure is repeated until all the inconsistencies are resolved. Figure 5.8 shows an example where partial plans are generated for inconsistencies.

Algorithm 16: Method to generate ordering constraints(O) and populate affordances(A)

```
1 GenerateOrderingConstraints()
2     foreach $m_{start} \in M_{start}$ do
3         $a = m_{start} \rightarrow GetAffordance()$
4         $\mathrm{A} = \mathrm{A} \cup a$
5         $\mathrm{O} = \mathrm{O} \cup \{a_{start} \prec a\}$
6     foreach $m_{start} \in M_{start}$ do
7         foreach $m_{end} \in M_{end}$ do
8             if $m_{start} \rightarrow startTime > m_{end} \rightarrow endTime$ then
9                 $\mathrm{O} = \mathrm{O} \cup \{m_{end} \rightarrow GetAffordance() \prec m_{start} \rightarrow GetAffordance()\}$
```

Algorithm 17: Algorithm to generate possible complete narratives

```
1 begin
2     GenerateOrderingConstraints()
3     $\pi_i =< \mathrm{A}, \mathrm{L}, \mathrm{O} >$
4     $\pi = \pi_i$
5     repeat
6         $\Pi =$ DynamicPlanner($IL \rightarrow pop(), \pi$)
7         if $\Pi \neq \emptyset$ then
8             $\pi =$ SelectNarrativeFromUI($\Pi$)
9         else
10            Show("$No\,possible\,narratives\,found$")
11    until $\#IL > 0$
```

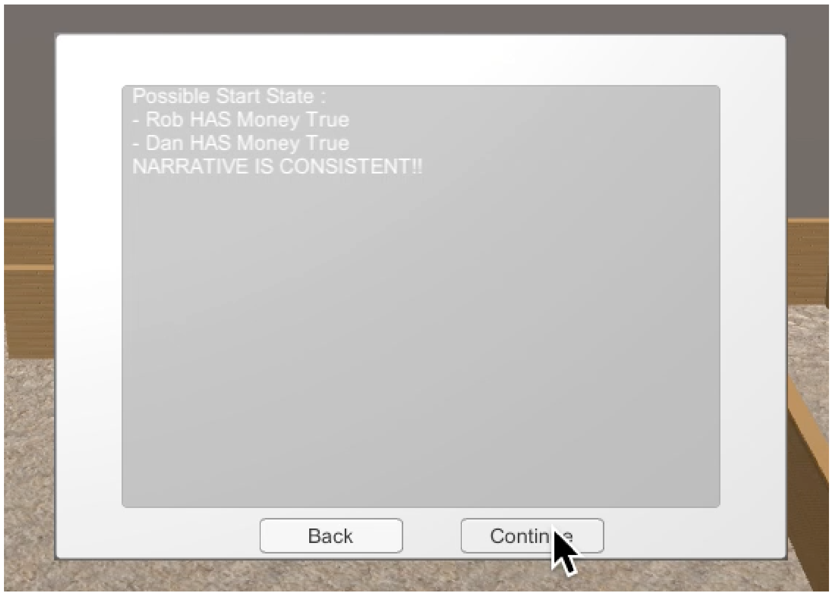

Possible Start State :
- Rob HAS Money True
- Dan HAS Money True
NARRATIVE IS CONSISTENT!!

Figure 5.6: Message if narrative is consistent (*full video*)

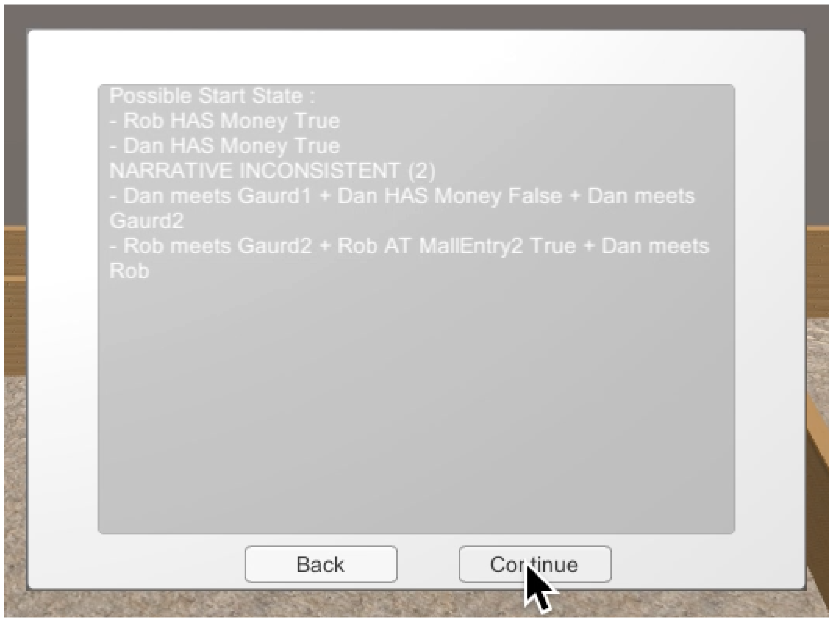
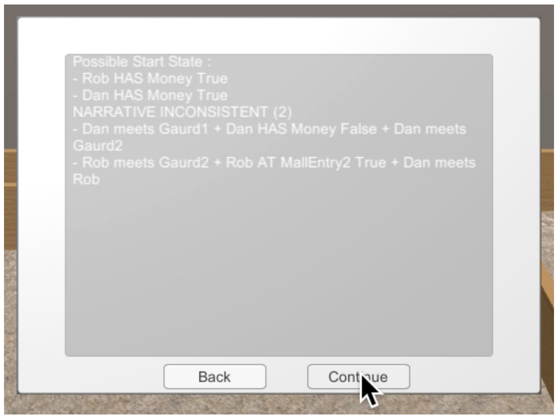
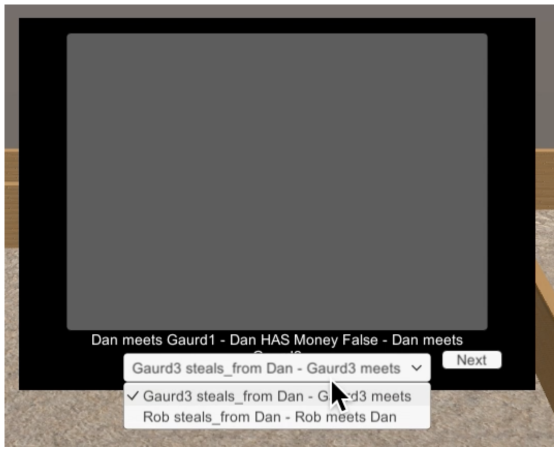

Possible Start State :
- Rob HAS Money True
- Dan HAS Money True
NARRATIVE INCONSISTENT (2)
- Dan meets Gaurd1 + Dan HAS Money False + Dan meets Gaurd2
- Rob meets Gaurd2 + Rob AT MallEntry2 True + Dan meets Rob

Figure 5.7: Example of inconsistent narrative (*full video*)



Possible Start State :
- Rob HAS Money True
- Dan HAS Money True
NARRATIVE INCONSISTENT (2)
- Dan meets Gaurd1 + Dan HAS Money False + Dan meets Gaurd2
- Rob meets Gaurd2 + Rob AT MallEntry2 True + Dan meets Rob

(a)

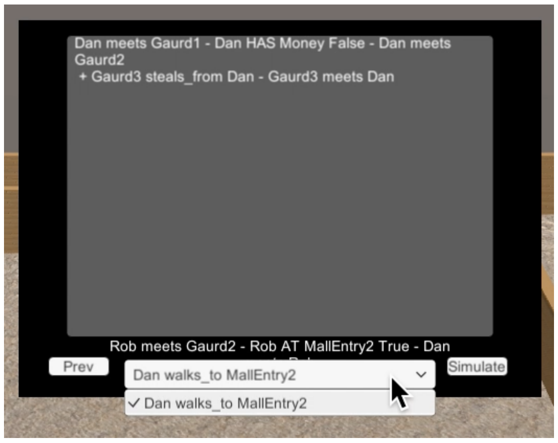


Dan meets Gaurd1 - Dan HAS Money False - Dan meets

Gaurd3 steals_from Dan - Gaurd3 meets

✓ Gaurd3 steals_from Dan - Gaurd3 meets

Rob steals_from Dan - Rob meets Dan



Dan meets Gaurd1 - Dan HAS Money False - Dan meets Gaurd2
+ Gaurd3 steals_from Dan - Gaurd3 meets Dan

Rob meets Gaurd2 - Rob AT MallEntry2 True - Dan

Dan walks_to MallEntry2

✓ Dan walks_to MallEntry2

(b) (c)

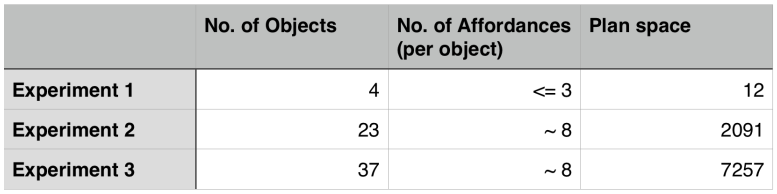
Figure 5.8: Figure (a) shows the inconsistencies generated by the **Narrative Validator**. Drop down in figure (b) shows the plans generated by **dynamicPlanner**. When a plan is selected, it is added and the next inconsistency is solved(see (c). The full video of demo is available *here*.

Chapter 6

Results

6.1 Comparative Study

The planner is tested for three different scenes with varying number of smart objects and affordances. The quantitative details of the experiments are shown in figure 6.1



	No. of Objects	No. of Affordances (per object)	Plan space
Experiment 1	4	<= 3	12
Experiment 2	23	~ 8	2091
Experiment 3	37	~ 8	7257

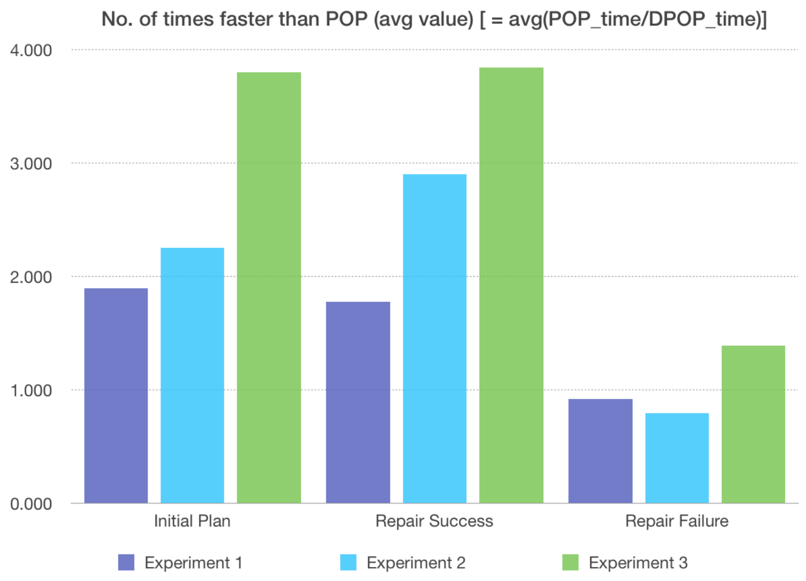
Figure 6.1: For homogenous objects, average of 10 observations(time in ms)

For each experiment, the data is collected for 3 cases; (i) Initial plan (ii) Repair/Replan success [when planner is able to repair the narrative] (iii) Repair/Replan failure [when planner is not able to repair the narrative]

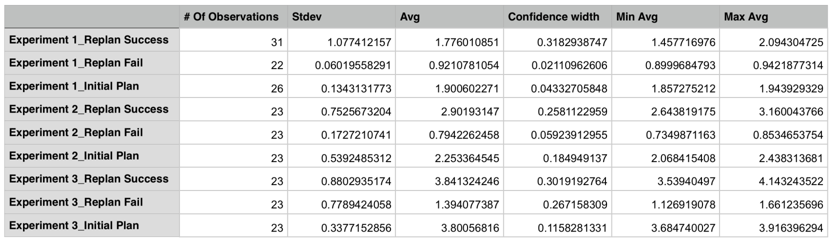
6.1.1 Accelerated POP

For initial tests, the speed-up planner is run in a scene with varying number homogenous objects and the times are compared against the normal planner. The tabulated results of the experiment are present in figure 6.3

The ***Initial Plan*** case of the experiments corresponds to the Accelerated POP. The A-POP displayed significant improvements over the general POP.The plan time improvements were exponential. Results of Experiment_3 implied that A-POP was 3.6 to 3.9 times faster than POP with 95% confidence (see figure 6.2).

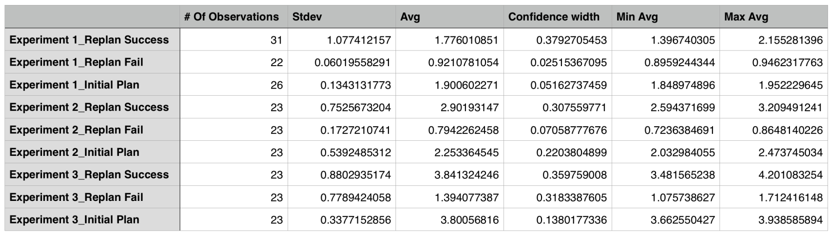


(a) Average for all expertiments



	# Of Observations	Stdev	Avg	Confidence width	Min Avg	Max Avg
Experiment 1_Replan Success	31	1.077412157	1.776010851	0.3182938747	1.457716976	2.094304725
Experiment 1_Replan Fail	22	0.06019558291	0.9210781054	0.02110962606	0.8999684793	0.9421877314
Experiment 1_Initial Plan	26	0.1343131773	1.900602271	0.04332705848	1.857275212	1.943929329
Experiment 2_Replan Success	23	0.7525673204	2.90193147	0.2581122959	2.643819175	3.160043766
Experiment 2_Replan Fail	23	0.1727210741	0.7942262458	0.05923912955	0.7349871163	0.8534653754
Experiment 2_Initial Plan	23	0.5392485312	2.253364545	0.184949137	2.068415408	2.438313681
Experiment 3_Replan Success	23	0.8802935174	3.841324246	0.3019192764	3.53940497	4.143243522
Experiment 3_Replan Fail	23	0.7789424058	1.394077387	0.267158309	1.126919078	1.661235696
Experiment 3_Initial Plan	23	0.3377152856	3.80056816	0.1158281331	3.684740027	3.916396294

(b) For 90% confidence



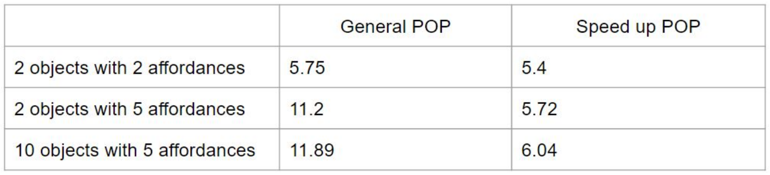
	# Of Observations	Stdev	Avg	Confidence width	Min Avg	Max Avg
Experiment 1_Replan Success	31	1.077412157	1.776010851	0.3792705453	1.396740305	2.155281396
Experiment 1_Replan Fail	22	0.06019558291	0.9210781054	0.02515367095	0.8959244344	0.9462317763
Experiment 1_Initial Plan	26	0.1343131773	1.900602271	0.05162737459	1.848974896	1.952229645
Experiment 2_Replan Success	23	0.7525673204	2.90193147	0.307559771	2.594371699	3.209491241
Experiment 2_Replan Fail	23	0.1727210741	0.7942262458	0.07058777676	0.7236384691	0.8648140226
Experiment 2_Initial Plan	23	0.5392485312	2.253364545	0.2203804899	2.032984055	2.473745034
Experiment 3_Replan Success	23	0.8802935174	3.841324246	0.359759008	3.481565238	4.201083254
Experiment 3_Replan Fail	23	0.7789424058	1.394077387	0.3183387605	1.075738627	1.712416148
Experiment 3_Initial Plan	23	0.3377152856	3.80056816	0.1380177336	3.662550427	3.938585894

(c) For 95% confidence

Figure 6.2: The data from all experiments is presented above. The metric for the data is *"Times Faster"*, i.e. D-POP is **'x'** *times faster* than POP [Formula : avg(POP_time/D-POP_time)]

6.1.2 Dynamic planner

In initial tests, the average of 26 observations in an action space size of 10, showed DPOP to be 10.9 percent faster than regular partial planner (see figure 6.7).



	General POP	Speed up POP
2 objects with 2 affordances	5.75	5.4
2 objects with 5 affordances	11.2	5.72
10 objects with 5 affordances	11.89	6.04

Figure 6.3: For homogenous objects, average of 10 observations(time in ms)

A detailed analysis is done for the experiments 1, 2 and 3 and the data is split in two cases. Case one where the D-POP tries to repair the plan and succeeds. Case two, when no plan exits between current state and goal state. For case one, D-POP showed similar improvements as A-POP. D-POP was 3.7 to 3.9 times faster than POP with 90% confidence (See figure 6.2). For case two, D-POP is almost as fast as POP(sometimes slower). This behavior is expected as D-POP does an exhaustive search just like POP. Detailed analysis is found in figure 6.2.

6.2 User Study

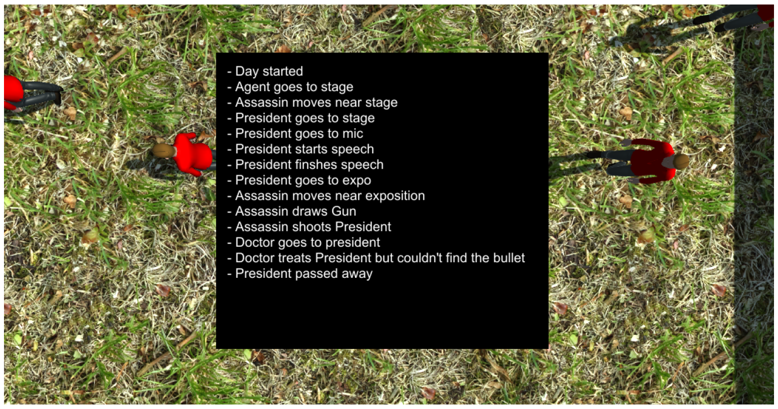
6.2.1 Hypotheses

There are two major hypotheses for our application

1. Dynamic Planner can reconstruct/repair the plan in real-time and faster than a partial planner. **Metrics** : Time for replan using POP and DPOP, Time for replan from scratch using POP and DPOP.

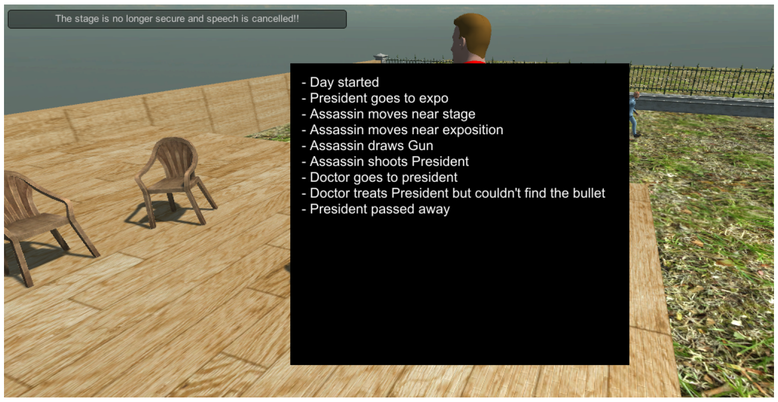
2. This framework could be used to generate real-world like narratives with free-from interactions. **Metrics** : How close the simulation is to the original event, complexity of the narrative.

In this user study, we did a quantitative analysis of the Dynamic Planner from the experimental results discussed in Results section. An in-game questionnaire is done to do a qualitative analysis of the system.



- Day started
- Agent goes to stage
- Assassin moves near stage
- President goes to stage
- President goes to mic
- President starts speech
- President finshes speech
- President goes to expo
- Assassin moves near exposition
- Assassin draws Gun
- Assassin shoots President
- Doctor goes to president
- Doctor treats President but couldn't find the bullet
- President passed away

(a) Initial narrative is auto-generated and displayed in journal



The stage is no longer secure and speech is cancelled!!

- Day started
- President goes to expo
- Assassin moves near stage
- Assassin moves near exposition
- Assassin draws Gun
- Assassin shoots President
- Doctor goes to president
- Doctor treats President but couldn't find the bullet
- President passed away

(b) UI invalidates the narrative and D-POP repairs it

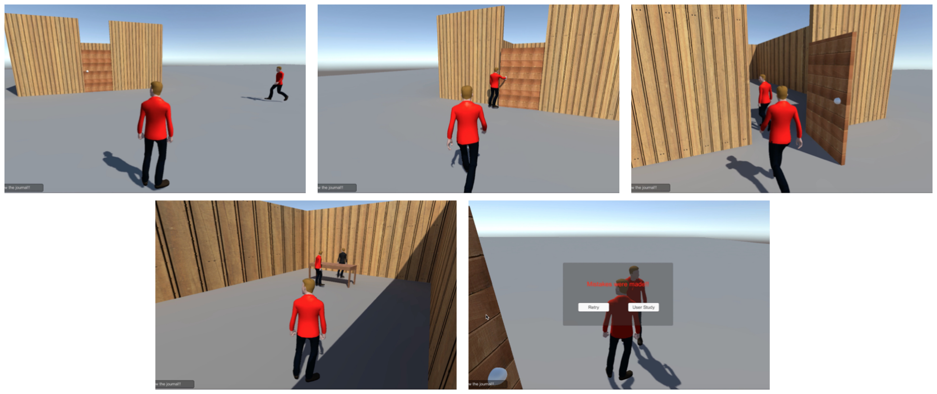
Figure 6.4: Screenshots from exp 3 demonstrating a case where narrative is repaired by D-POP

DPOP analysis

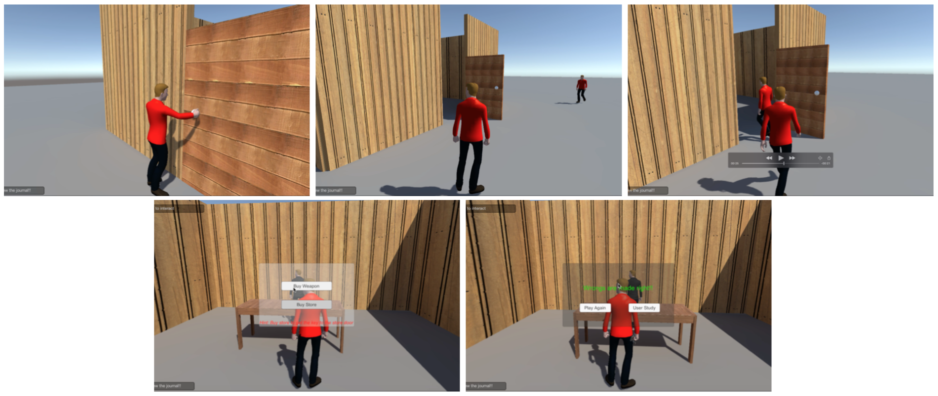
The following metrics have been collected from game plays of 23 people. The game environment consists of 11 affordances and 4 heterogeneous smart objects.

Time to plan from Scratch

For an experiment, the time take to generate initial plan was recorded for Dynamic POP and POP. During the initial plan, the speed-up planned discussed in section 4.1 is implemented. A comparative study is depicted in figure 6.6.



(a) When no user intervention, NPC opens the door, walks in and buys the weapon.



(b) Player opens the door before NPC. NPC walks in(without opening the door). Player buys the weapon before NPC and the narrative fails to repair.

Figure 6.5: 6.5a and 6.5b are two different cases of experiment 1

Time to repair plan

The dynamic planner mentioned in section 4.2 is evaluated by this metric. The time to repair a plan using DPOP is compared with the time to generate a partial plan from scratch using POP. The results are shown in figure 6.7

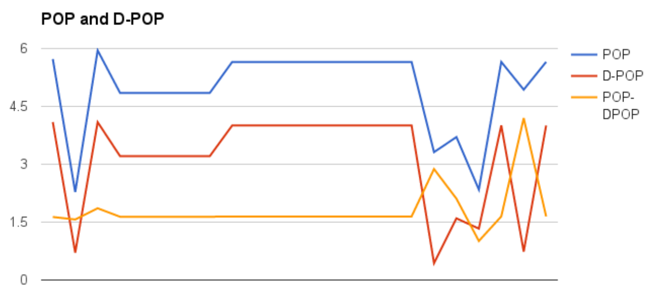


Figure 6.6: Time taken to generate a partial plan (in ms)

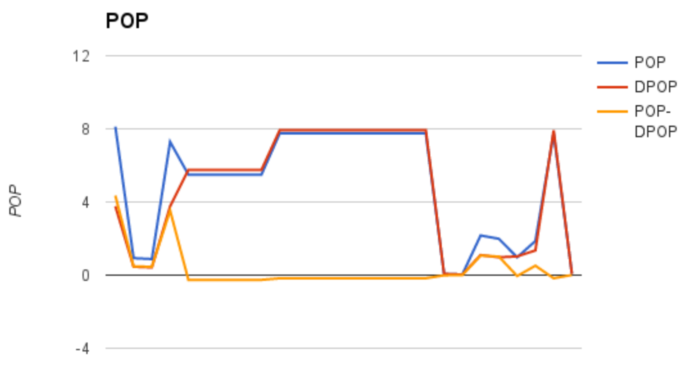
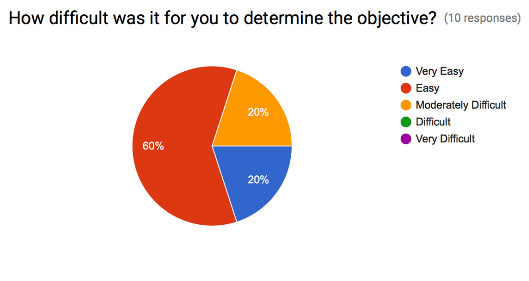
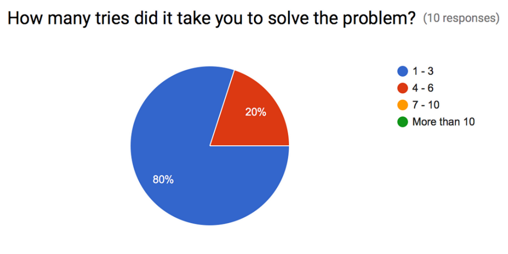


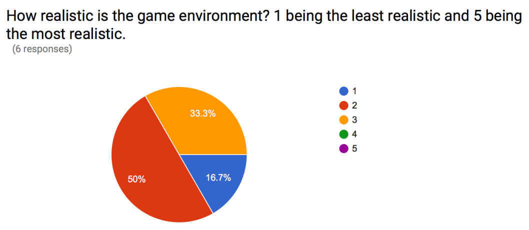
Figure 6.7: Time taken to repair the plan (in ms)



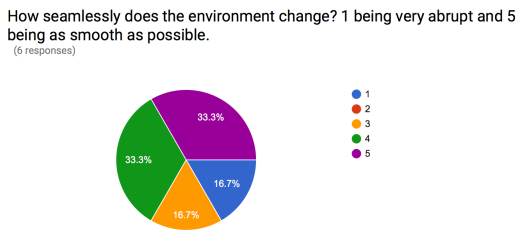
(a)



(b)



(c)



(d)

Figure 6.8: Player responses for the user study questionnaire

Chapter 7
Conclusion

7.1 Conclusion

We conclude by saying that, the techniques introduced in this paper to reconstruct narratives from multiple agent memories are tested for different scenarios and are proven robust. Though the agent architecture was introduced to support narrative recreation, this in itself can further be extended to implement a complete cognition architecture. As per our user study results so we have deduced that

- The A-POP and D-POP has shown improvements over POP for the test data. This system should be tested for more complex scenarios across different systems to get more diverse data
- From user study questionnaire, the complexity of the simulation can be further increased
- Also, from the questionnaire it can be deduced that Dynamic Planner is able to repair plans in real-time

7.2 Future Work

The agent architecture should further be tested for more complex scenarios and should be checked for its robustness to see if it can be used in real world scenarios like surveillance & crime solving. For the planner, the system used in this paper facilitates to prove the working and efficiency of the D-POP and accelerated POP, it could only do so for a predetermined number of affordances and smart objects. Hence we need a system that can procedurally generate smart objects and affordances so that universal statistics of a planner can be measured. The output(Narrative) quality of the planner can improved by using a better heuristics like interesting-ness or tension induced in the story if an action is added.

References

[1] Alexander Shoulson, Francisco M. Garcia, M. J. R. M. N. I. B. Parameterizing behavior trees. In *4th International Conference, MIG 2011, Edinburgh, UK, November 13-15, 2011. Proceedings* (2011).

[2] Boloni, L. A cookbook of translating english to xapi.

[3] Boloni, L. Xapagy cognitive architecture.

[4] David Poole, Alan Mackworth. Partial order planning.

[5] Gordon, A. S., van Lent, M., van Velson, M., Carpenter, P., and Jhala, A. Branching Storylines in Virtual Reality Environments for Leadership Development. In *Proceedings of the 16th Innovative Applications of Artificial Intelligence Conference (IAAI-04)* (San Jose, CA, 2004), AAAI Press, pp. 844–851.

[6] Huang, P., Kapadia, M., and Badler, N. I. Spread: Sound propagation and perception for autonomous agents in dynamic environments. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (New York, NY, USA, 2013), SCA '13, ACM, pp. 135–144.

[7] Kallmann, M., and Thalmann, D. Direct 3d interaction with smart objects. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 1999), VRST '99, ACM, pp. 124–130.

[8] Kapadia, M., Marshak, N., and Badler, N. I. ADAPT: The Agent Development and Prototyping Testbed. *IEEE Transactions on Visualization and Computer Graphics 99*, PrePrints (2014), 1.

[9] Kapadia, M., Singh, S., Reinman, G., and Faloutsos, P. A behavior-authoring framework for multiactor simulations. *Computer Graphics and Applications, IEEE 31*, 6 (nov.-dec. 2011), 45 –55.

[10] Kartal, B., Koenig, J., and Guy, S. J. User-driven narrative variation in large story domains using monte carlo tree search. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems* (Richland, SC, 2014), AAMAS '14, International Foundation for Autonomous Agents and Multiagent Systems, pp. 69–76.

[11] Li., B. *Learning Knowledge to Support Domain-Independent Narrative Intelligence.* PhD thesis, 2015.

[12] Li, B., Lee-urban, S., and Riedl, M. O. Crowdsourcing narrative intelligence. *Advances in Cognitive Systems 2* (2012).

[13] Likhachev, M., Ferguson , D., Gordon, G., Stentz , A. T., and Thrun, S. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)* (June 2005).

[14] Loyall, A. B. *Believable agents: building interactive personalities.* PhD thesis, Pittsburgh, PA, USA, 1997.

[15] Magerko, B., Laird, J. E., Assanie, M., Kerfoot, A., and Stokes, D. AI Characters and Directors for Interactive Computer Games. *Artificial Intelligence 1001* (2004), 877–883.

[16] Minton, S., Bresina, J. L., and Drummond, M. Total-order and partial-order planning: A comparative analysis. *CoRR abs/cs/9412103* (1994).

[17] Mubbasir Kapadia, Jessica Falk, F. Z. M. M. R. W. S. M. G. Computer-assisted authoring of interactive narratives.

[18] Pérez, R. P., and Ortiz, O. A model for evaluating interestingness in a computer–generated plot,. In *Proceedings of the Fourth International Conference on Computational Creativity* (Sydney, Australia, jun 2013), p. 131–138.

[19] Riedl, M. O., and Young, R. M. An intent-driven planner for multi-agent story generation. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1* (Washington, DC, USA, 2004), AAMAS '04, IEEE Computer Society, pp. 186–193.

[20] Schuerman, M., Singh, S., Kapadia, M., and Faloutsos, P. Situation agents: agent-based externalized steering logic. *Comput. Animat. Virtual Worlds 21* (May 2010), 267–276.

[21] Shoulson, A., and Badler, N. I. Event-centric control for background agents. In *ICIDS* (2011), pp. 193–198.

[22] Shoulson, A., Gilbert, M. L., Kapadia, M., and Badler, N. I. An event-centric planning approach for dynamic real-time narrative. In *Proceedings of Motion on Games* (New York, NY, USA, 2013), MIG '13, ACM, pp. 99:121–99:130.

[23] Shoulson, A., Marshak, N., Kapadia, M., and Badler, N. I. Adapt: the agent development and prototyping testbed. In *I3D* (2013), M. Gopi, S.-E. Yoon, S. N. Spencer, M. Olano, and M. A. Otaduy, Eds., ACM, pp. 9–18.

[24] Singh, S., Kapadia, M., Hewlett, B., Reinman, G., and Faloutsos, P. A modular framework for adaptive agent-based steering. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 141–150 PAGE@9.