DATAFLOW ANALYSIS ON GAME NARRATIVES

by
Peng Zhang

School of Computer Science
McGill University, Montréal

July 2008

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Modern computer games tend to provide complex narratives, ensuring both extended and interesting game play. Narratives are important not only in traditional adventure games and role playing games, but also in first person shooting games and strategy games. Even so, many narratives contain defects that reduce the quality of games, and so it is important to develop analysis or verification techniques. Unfortunately, formal verification attempts are few, and in practice primarily done by manual inspection and test-playing. Our research is based on a framework, named *Programmable Narrative Flow Graph* (PNFG) that provides a high-level language to represent narratives. Our approach is to apply high-level, formal analysis to narratives to find out their *winning paths*, as a first step forward deeper verification. We extend the PNFG framework by developing a generic *dataflow* analysis module, and implement several analyzers to gather high-level data on narrative behaviors. To improve the performance of our approach, we also designed several optimizations that reduce the size of the search space. Not all individual optimizations are effective on all narratives, but our final, fully-optimized strategy is able to analyze large narratives in relatively little time—an improvement of several orders of magnitude over the state-of-the-art in narrative analysis. This represents a practical and effective design for analyzing narratives that we hope can be the basic of real improvements in game development.

Résumé

Les jeux modernes pour ordinateurs ont tendance à apporter les structures narratives complexes, en affirmant un jeu à la fois entendu et intéressant. Les structures narratives sont importantes non seulement dans le jeu d'aventure et le jeu de rôle, mais aussi dans le jeu de tir à la première personne et le jeu de stratégie. Malgré tout, beaucoup de structures narratives possèdent des défauts qui réduisent la qualité des jeux, ainsi c'est important de développer la technique de analyse ou de vérification. Malheureusement, des tentatives de vérification formelle sont limitées, et dans la pratique effectuée avant tout par l'inspection manuelle et l'essai du jeu. Notre recherche est basée sur une structure applicative nommée *Programmable Narrative Flow Graph (PNFG) (Plan de Flux Narratif Programmable)* qui offre un langage de haut niveau à représenter des structures narratives. En tant que premier pas vers la vérification plue profonde, notre approche est à appliquer l'analyse formelle de haut niveau sur les structures narratives afin de trouver leurs *winning paths (chemins de la victoire)*. Nous prolongeons la structure applicative PNFG par l'élaboration d'un module d'analyse de flux de données génériques, mais aussi mettons en oeuvre plusieurs analyseurs à recueillir des données de haut niveau sur les comportements narratifs. Pour améliorer la performance de notre approche, nous avons également conçu plusieurs optimisations qui réduisent la taille de l'espace de recherche. Non toutes les différentes optimisations sont efficaces sur les structures narratives, mais notre stratégie finale et totalement optimisée est capable de analyser des structures narratives larges dans un délai relativement bref — une amélioration de plusieurs puissances de dix au-dessus l'état de la technique dans l'analyse narratif. Cela représente un pratique et une conception efficace pour analyser les structures narratives que nous espérons peuvent être la base de vraies améliorations de développement de jeu.

Acknowledgements

I would like to give my first thank to my supervisor, Clark Verbrugge. It is difficult to mention every effort he made in this research, but I appreciate his discussions, ideas, the review and modification of this thesis and other support throughout the research. Without his priceless help, my entire work would be impossible.

I also thank Chris Pickett and Félix Martineau for their contribution to the NFG and PNFG frameworks, and all the students in the "Modern Computer Games" (COMP521) course in McGill University, 2007, who wrote excellent benchmarks for the PNFG.

I would like to thank Dr. Laurie Hendren for her great course "Optimizing Compilers" (COMP 621). This research is a dataflow analysis approach on the PNFG framework, so her course provided lots of details about this techique and widened my eyes on this area.

I am also indebted to the people at Sable Reseach Group in McGill University: Gregory Prokopski who provides technical support for the entire lab, Haiying Xu and Dayong Gu who help me on LaTeX and other questions.

My parents provides a constant source of support. No matter what I am facing, you are always the spring of all my strength and spirit.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction and Contribution

1.1 Introduction

In recent years, the role that narratives play in games is becoming more and more important. Not only traditional role playing games and adventure games are in pursuit of longer and more complex narratives, but even strategy games, like "WarCraft3: Frozen Throne" [17], or first person shooting games, such as "Half-Life2" [14], are forced to look for more narratives to cater to the more demanding modern players.

Normally, narratives are written in either special-designed programming languages or scripting languages designed for general purposes rather than narratives [29], such as Python or Lua *etc.*. Like any other programs, narratives, especially complex ones, are highly like to contains flaws, and unfortunately, many of them are hard to identify until finally revealed by players.

As well as logical flaws, computer narratives can reach unplayable or unwinnable game states, or even unexpected behavior such as game crashes. Several general flaws are discussed in Section 2.1. To avoid these flaws, one possible step is to automatically analyze the narratives and check that certain game properties must or must not hold at any point during potential gameplay. Finding all the *winning paths* of the narrative, or sequences of game actions that results in winning the game, for instance, would not only verify that the narrative is at least playable, but would also be helpful in checking for potential game *deadlocks* or unsound states/moves. Our work has focused primarily on the problem of finding

the winning path.

Among various game genres, we base our study on the Interactive Fiction (IF) games. These games contain most fundamental narrative elements, for example, places, items, non-player characters (NPCs), actions, *etc.*, and also present the narrative in the simplest form of user interface — most IF games use pure text as their input/output, thereby giving us a clear image of the narrative itself. Since narratives are independent of the external elements, like graphics, the IF game is also suitable for designing narratives for other types of games, regardless their types and fancy functionalities.

Like programming bugs, flaws in narratives are hard to identify because there are so many formalisms to choose from with different trade-offs to represent narratives; and even in the same language, the representation of narratives varies by the game engine, though, hiding the narrative structure, and thus making it more difficulty to analyze. To avoid the informality issue, our research is built on a framework called *Programmable Narrative Flow Graph (PNFG)*, which provides a scripting language to represent narratives formally and at a high-level. By design, PNFG code is compiled into a Petri-Net representation, named *Narrative Flow Graph (NFG)* allowing for convenient examination and analysis of narratives at a low-level as well. Motivated by observations in previous work in this context that heuristically showed significant analyses benefit from using high-level game information [25], we focus our design on analyzing game in the PNFG language itself.

Our study starts by building a dataflow module for the PNFG framework to support formal *dataflow* analysis on high-level narrative codes. We analyze the PNFG structure and design a specialized *Control Flow Graph (CFG)* as the intermediate representation of the source code. The analysis module encapsulates all game states in two domains, elements of which are propagated through the game actions to find further reachable states. A generic design is provided which supports forward and backward dataflow analyses.

Based on this dataflow analysis module, we implement a series of analyses to find the winning paths. Since a winning path consists of connected sequences of actions, an *action summarization* analysis is designed. This consists of a backward analysis to compute the prerequisites for a successful trigger of the action, and a forward analysis to compute the necessary game state after the execution. Using the profiles that generated from the information given by these two analyses, we link the actions into a *Action Dependency Graph*

(ADG) respecting the causal requirements of each game action. This allows the game state space to be searched more efficiently, and we demonstrate the use of this technique in experiments using several optimized searching designs. We further develop three additional optimizations that reduce the searching cost by taking advantage of high-level game information available through our analysis framework.

1.2 Contributions

Specific contributions of our work include:

- Building a formal dataflow module for the PNFG framework. The new module, which is implemented in Java, is an extension of the PNFG, with a set of interfaces and abstract data structures. It provides fundamental dataflow features that permit analysis at a high-level.
- Implementing dataflow analyzers for PNFG. Actual analyzers are built on the new module. Concrete dataflow rules are implemented to complete various analyses, including profiling and winning path searching.
- Developing a new approach to finding the winning paths. Our set of analyses represents a novel approach to determine the winning path in the game state space. Significant efficiency gains are provided by basing our approach on information given by high-level PNFG analyzers.
- Applying analyses to large narratives and getting non-trivial, practiced results. We evaluate the performance of our analysis approach on a realistic benchmark suit and demonstrate order of magnitude improvements over previous, low-level approaches.

1.3 Organization

The rest of the thesis is organized as follows:

In the next chapter, we discuss interactive fiction games, representations and analyses of narratives. In Chapter 3, we give an overview of the NFG and PNFG framework by presenting core functionalities of the language. We then explain previous analysis approaches.

Chapter 4 shows our first step: building the dataflow analysis module on the PNFG framework. In Chapter 5 we describe how we use the new module to find out the winning path of the game, and several ways to improve the performance of the initial version of our new approach. Then all the experimental data regarding to our work, and the results analysis are shown and discussed in Chapter 6. We conclude in Chapter 7 and discuss our future work.

Chapter 2
Related Work

Narratives have been central to many computer game genres over the years, although formalization and analysis of narratives is much less common. Our research has focused on *Interactive Fiction (IF)* games, both to reduce the complexity and difficulty for analysis introduced by a more modern game interface, and due to their continued impact, evident in the structurally similar narrative designs of many current games in the Adventure, Role Playing, and even First Person Shooter genres.

IF is one of the oldest game genres, with William Crowther's "Colossal Cave Adventure" in 1975 recognized as the historical first IF game [27]. Thanks to its minimal technological requirements popularity in IF games boomed from that point, reaching its peak in the 1980s. In the 1990s, with the development and wide distribution of multimedia techniques, IF games largely faded out. Today, the IF game is minimally seen in terms of game community or commercial game market; nevertheless, people still view it as a legitimate literary art form [28], and some on-line communities on usenet newsgroups, like `rec.arts.int-fiction` and `rec.games.int-fiction` are still active (and sponsoring a yearly IF competition [2]). A complete history of IF games can be found in [27].

Although IF games are now marginal in the world of gaming, the core of IF, the narrative, thrives. Most modern, complex games include a large narrative component, increasingly so as the appeal of improved graphics and speed is less able to drive market interest. The interactive component of computer narratives also continues to be explored; narrative

is a main concern in the realm of Interactive Storytelling (IS), where the goal is to generate narratives using interactive computer systems. IS is very close to IF; however, different from the static, predefined world of most IF games, narratives in IS are dynamic, changeable while playing [15]. With IS, players may receive different experiences in each game play session [5, 9], making the story more interesting, replayable, and greatly reducing the resources required by developers. This more generally extends to game Artificial Intelligence (AI), in which narrative has been of interest to researchers for a long time. Young has posed a series of questions and concerns in relation to studies in this interdisciplinary field [43] such as what structures from AI research can best accommodate narrative representations, *etc.*. Sengers and Mateas review and demonstrate current combinations of narrative and AI approaches to a number of goals, including story database systems which can be indexed by stories, and story-understanding systems that are able to derive and reason about implicit information in narratives [37]. Other researchers have focused on applying AI methodologies more specifically to narrative generation. Gervás has built a framework for automated story generation in which both AI techniques and aesthetic issues are considered [18].

2.1 General narrative flaws

As games grow larger and more complex, narrative size and complexity is also increased. Unfortunately, an unavoidable consequence of such growth is that more flaws are also introduced by developers. As Lindley points out [22], most narrative flaws are related to unanticipated sequences of actions, and the greater the degree of freedom given to players the more possible scenarios must be considered, increasing the chance for narrative flaws. Adams describes several types of narrative flaw in real games [3]: “Deadlocks” are quite similar to the problem of the same name in multi-processing systems. In narratives deadlocks mainly result from the wrong topological order of required tasks; for example, asking the player to accomplish a task with some items he/she can only get in future chapters. If the designer fails to provide a way of solving deadlocks, such as giving the player an alternative item in the previous example, players will be unable to progress in the game. On the other hand, “Incorrect Victory Checks” and “Illogical Victory Checks” can also prevent

players from winning a certain level due to incorrect or incomplete winning conditions. In these situations, players are able to achieve their task by ways which are unexpected by the designers, and thus are not recognized by the game. A concrete example mentioned by Adams is from "Red Faction" [40]: in one mission of the game, the player is asked to destroy a particular computer in a space station, and then blow up the whole station. A shortcut exists where the player ignores the computer and tries to blow up the station directly — logically, the whole mission should be certainly accomplished this way as well. Unfortunately, the game does not treat the computer as destroyed even when the whole space station is reduced to ashes. Adams also considers a variety of other, less critical flaws that reduce the sense of immersion, or which create subtly undesirable properties.

Another common but severe flaw is "pointlessness" [41]. Once the player is definitely unable to win the game there is no logical sense in allowing gameplay to continue. Different endgame or failure solutions are possible of course, and telling a player they have lost does not have to always happen with great immediacy, but it should also not be overly-delayed, or players will experience frustration at spending time in a game which can no longer be successfully completed. An extreme example can be found in an early adventure game "The Hitchhiker's Guide to the Galaxy" [19]; at the beginning of the game the player has a one-time opportunity to pick up a specific item essential to eventually winning the game. If the player does not, however, the game does nothing until the very end, at which point it informs the player they have lost [3]. With game narratives that produce gameplay lasting weeks to months such problems are clearly vexing to players who invest significant time and effort under the impression the game is still winnable.

2.2 Characteristics of good narratives

A narrative without basic flaws is not the final goal. By its nature, a good narrative should also be "interesting enough". In his textbook on game design, Rabin, for instance, defines a great game as "*a series of interesting and meaningful choices made by the player in pursuit of a clear and compelling goal*" [34]. Although the term "interesting" is obviously subjective, Rabin also provides an insightful study on characteristics of narratives that reflect good quality. In consideration of the number of choices a player must make, the

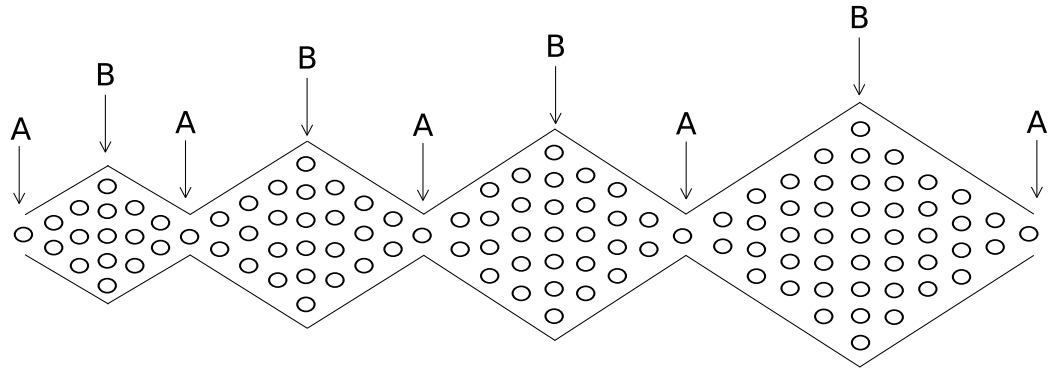

Figure 2.1 – A Series of Convexities with points of limited choice (A) and points of many choices (B), picture redrawn from [34]

term *convexity* is introduced. A convexity is an evolution of choices, starting from just one or quite a few choices, widening to a larger set, and finally returning to a single or limited set of choices. In fact an ideal game presents *iterated convexity,* a series of convex subregions connected by a reduced set of options, the chain itself forming an overall convex shape. A good narrative should have a chain of *convexities*, as depicted in Figure 2.1.

Loosely defined, convexity can be found in many narratives. For example, in Act 2 of Diablo II [16], before the final goal of the chapter (killing Duriel), the player is asked to accomplish several quests. Some of these tasks, like getting the Horadric Staff, are key quests, while others are optional, with the order of completion also not constrained. Reaching this point demonstrates the closure of the previous chapter's choices (end of one convexity), and opening up of a new set of choices (start of another convex region). In such structures the narrative gives the player enough choice, but not too much choice, improving immersion without over-saturating the player with information.

Formally defining convexity is more difficult. Martineau has done an initial investigation of the theory of convexities in the PNFG framework. In his Master's thesis, in each narrative, a game tree is built to represent the states and corresponding transitions, then the total number of states in each level is counted as the convexity [25]. Besides the overall convexity, he also calculates the convexity of the winning path. According to his experiments, patterns do exists in certain kind of narratives but a well shaped pattern like that of Figure 2.1 is rarely found.

Further work on "good" games has been investigated from psychological perspectives. Game enjoyment has been linked to the sense of "flow", and this correlated with various elements of game design [38]. Others have looked for game play features that imply certain kinds of enjoyment, such as suspense [11], or *Acousmêtre* [24]. Analyzing such abstract, emotional properties is beyond the scope of our investigation here, but is an interesting area for future work in improving narrative design.

2.3 Representations and analyses of narratives

Several high level frameworks have been proposed for representing and analyzing narratives, including logic-based prototypes. Reiter [35] provides a prologue implementation on a calculus-based language for narratives, and similarly, Kakas et al [21] achieve the same goal with their language named "$\mathcal{E}$". Other systems more directly represent the narrative structive; Petri-Nets, for example, have been used as a fundamental structure [7, 6, 33, 41]. In most of these frameworks, however, the formalism does not easily scale to larger games, making them quite hard to put into practice. For more practical purposes action-based scripting languages are typically used. The most popular frameworks, Inform [1] and TADS [36] have evolved for long time and are pretty mature today. There are many other frameworks used in the Interactive Fiction genre as well, primarily differing in narrative grammar design [23, 29, 36].

Narratives are best viewed as programs, and their analysis should be identical to verifying programs [35]. In reality, as underlined by Natkin and Vega, compared to the number of frameworks for narrative representation, techniques relating to analysis are far fewer, and due to the different concerns of frameworks, verification is more scattered as well [31]. Burg and Lang, for instance, analyze the chronology of narrative events using constraint-logic programming, successfully finding a chronologic conflict in the story "A Rose for Emily" by William Faulkner [8]. Lindley and Eladhari demonstrate another attempt to verify and improve narratives by using an approach mixing constraint-logic and object-oriented methods, used to encapsulate certain non-sequential structures of stories [22]. Szilas and Rety decompose the narrative to a finite state machine graph driven by tasks and then apply analyses on the resulting graph [39]. However, the initial conditions of

each action have to be fed manually rather than generated automatically—nearly an impossible mission for large narratives. Other research has applied analyses based on linear logic, checking the narrative complexity [13]. Systems specifically for authoring interactive storytelling systems have also been proposed, including features for helping ensure that dynamically created scenarios and predetermined structures are well balanced [10, 5, 26].

Chapter 3
Background

Our work serves as an extension to the PNFG framework. PNFG is a scripting language for narratives, with high-level PNFG code compiled down to low-level NFG form. In this chapter, we will first give a quick introduction to NFG in Section 3.1, and PNFG in Section 3.2 to draw a complete picture of the whole research. We will also review analyses on the PNFG framework in Section 3.3, including using the generic model checker and the high level analysis.

3.1 NFG

Narrative Flow Graph, or NFG [41], written in Java, is a 1-safe Petri-Net designed to formalize the representation of narratives. In NFG, a game consists of property states and actions. The former are depicted as *places* in the graph, while the latter are *transitions*. Tokens are assigned to the places if the states they indicate are true while playing the game, and removed if the states are false. As a 1-safe Petri-Net, each node can have at most one token at one time, and a transition is enabled to fire if all its incoming nodes are tokenized. When a transition is fired, tokens from incoming nodes are deleted and tokens are created in all to outcoming nodes of the transition. NFG execution rules are in general identical to standard Petri-Net operational semantics [30].

To interact with players, NFG is designed to be event-driven. The state of an NFG game does not change until a transition is fired, a process usually initiated due to user input,

otherwise the NFG interpreter will do nothing but wait for user commands. Specifically, at the beginning of a game, the NFG interpreter places a token in the special "idle" state, which is both required (consumed) by every action and restored when an action completes. If the action issued by player is ready to fire, tokens are moved and propagated until the graph reaches a steady state, that is, no more transitions can be fired. If the game does not end, the interpreter will return to the "idle" state and wait for the next command.

To keep the formalism simple, NFG does not model the mechanics of player interaction; for example, it cannot show how actions and choices are actually presented and resolved in a real, graphic game context. This simplifies analysis, but does imply that certain narrative designs cannot be represented. NFG, for instance, does not have a concept of real-time, even though timing is necessary in some narratives. The only timing issue in NFG is the number of user actions rather than real time intervals.

3.2 PNFG

Although NFG is an appropriate representation framework for game narratives, it runs as a 1-safe petri-net, and due to the characteristics of this kind of data structure, The required graph for significant narratives can get extremely large [41], and so is impractical for designers to directly write games in NFG. In order to provide a user-friendly, high-level representation, Programmable Narrative Flow Graph, PNFG [32], has been developed and also implemented in Java. PNFG provides a simple structure for narratives, and its syntax is close to other IF toolkits, like Inform. The primary advantage of PNFG is in keeping the presentation formal and easy to analyze, part of which is ensuring the PNFG code can always be compiled down to a well-structured NFG form.

3.2.1 Basic Structures & Statements

Although the PNFG language has relatively few basic constructs, it has the ability to express even quite complex narratives. To represent a narrative in PNFG, the following structures should be defined:

Objects Object is a fundamental element in PNFG. An object can be a place in the game

world, or an item to be used by the player. A "sub-type" of object is an object which can *contain* other objects; for convenience in interactive fiction design, these are termed "rooms".

– Rooms: A room is a special class of object. As displayed by its name, it represents a location in the game. The whole game world map is formed by connected rooms. By convention, a room has eight orientations that can be directed to other rooms (n, e, w, s and diagonals), an enter block containing statements to be executed just prior to the player arriving in the room, and an exit block to be executed when the player leaves. In addition, rooms acts like containers, too: a room can hold the player, or any other object.

  Since the player is also able to hold items, it is conceptually a container. Thus in PNFG, the player itself is described as a "room" as well. A special room "you" must be always defined in the code to represent the player.

  The PNFG also creates a default room called "offscreen", where all objects and items are placed as initialization. In addition, "offscreen" is an isolated room which the player has no way to get into, so it also acts as a trash for discarding items in the process of playing.

States States are declared within objects to describe binary properties of the objects. States are boolean typed and are initialized to false.

For each PNFG narrative, two special states are added by the framework: "game.win" and "game.lose" for detecting the final state of the game. As known by their names, "game.win" is set to indicate the player has won the game, and similarly, the losing state is indicated by "game.lose". However, no matter which of the two states is turned to "true", the game will be immediately terminated.

Counters Like states, counters are also associated with objects; a counter, however, have a defined value range to represent a countable property and may thus be incremented, decremented and set to specific values. Counters require code to explicitly maintain their values. In order to provide support to automatically increased variables in narratives, for example, the number of rounds elapsed, PNFG can also maintain a special type of counter named timer, which gets automatically incremented after every user

action is executed. In our analysis, timers will be treated as global counters.

Sets In general IF games, operations are often applicable to several different objects with little or no variantion in the code other than the object name. To reduce redundancy, PNFG provides *sets*. A set is defined as a group of objects or other sets, and elements of the set are referred to by an abstract set variable. Sets provide convenient syntactic sugar, allowing, for instance, a simple mechanism for iterating over a collection.

Actions Action is another fundamental element in PNFG aside from object. Like NFG, PNFG is also event-driven, and events are generated by the commands input by the player which are themselves bound to actions. Actions declared within rooms are *local* actions, which are only available if the player is in that room, while actions declared outside of any room declaration are global ones that can be executed at any time. Action bodies are composed of sequences of PNFG statements.

Threads Many narratives require an epilogue after each event or trigger a special event at a certain point in the game if a specific global game state is reached. In PNFG, this is done by *threads*. Despite the terminology, threads are not concurrent so much as ubiquitous: a thread is a code snippet that will be executed automatically after each move made by the player. This eliminates the need to interleave special event state checks throughout the code. Threads are conceptually just common extensions applied to each action; in analysis, threads are automatically traversed after triggering every action, simulating the real PNFG execution.

PNFG actions are expressed by operations that manipulate objects and states; the focus is on the evolution of the narrative state — only a single operation is provided for explicit I/O.

Output Statement Output statements are to print messages on the screen, the only means in PNFG for the game to communicate with the player. An output statement is created by simply declaring a string constant within quotation marks.

Move Statement Move statements change the containment relation of objects, permitting objects to be transferred from one room to another. The grammar of move statement is quite straightforward; the statement:

```
move X from A to B;
```

will move an object “X” from room “A” to “B”.

Set Statement Set statements provide designers the means to change the values of states. Statement:

```
+X.w;
```

will set state “w” of object “X” to be true, and a similar statement using “-” instead of “+” will set it to be false. One key issue for writing set statements is they need to be aware of the previous value of the state. In other words, in order to execute the statement above, “X.w” must be false initially, otherwise the NFG interpreter will just stall and be unable to act appropriately. This is a slight drawback of the underlying Petri-Net. “Safe” operators for set statements are thus provided as well. A statement is “safe” if it will not block the interpreter at any circumstances. In PNFG, a question mark after the operator will generate “safe” but more complex NFG code that ensures correct behavior irrespective of the object’s initial state. Another safe operator, “^”, toggles the value of a state, but has an even more complex Petri-Net structure.

Counter Statement Changing the value of a counter in PNFG is done by counter statements. In PNFG, three counter operators are supported, which are quite similar to common programming language: For a counter “c” of object “X”, statement

```
X.c++;
```

will increase its value by 1;

```
X.c--;
```

will decrease the value by 1; and

```
X.c = 1;
```

will directly assign value “1” to the counter. Counters are range-bounded, and cannot be increased (or decreased) beyond their maximum (minimum) value.

The flow control statements of PNFG are similar to most programming language but are simplified a lot:

Conditional Statement Branches are created by “if” statements. In PNFG, however, conditions can only be a logical test of either the location of an object, or the value of a state or counter.

In the code below, for instance, the first conditional executes only if the object “X” is in room “A”; the second only if the state “w” of object “X” is true and the last if the counter “c” of object “X” is set to the value 1.

```
1 if (A contains X) {
2   "X is in Room A"; // test of a place
3 }
4
5 if (X.w) {
6   "w is true"; // test of a state
7 }
8
9 if (X.c == 1) {
10   "c equals to 1"; // test of a counter
11 } else {
12   "c is not 1"; // else part
13 }
```

Loop Statement PNFG does not have real loops with unbounded repetition. The only type of loop in PNFG is simple syntactic sugar for replicating sections of code over multiple objects. In this way, loops are actually iterations over sets. To insert a loop in PNFG code, an auxiliary set and a temporary variable should be declared, for example:

```
1 stuff = {pen, ruler, eraser} // auxiliary set
2 for (stuff $s) { // s is a temporary variable
```

```
3   "$s is stationery.";
4 }
```

And the output of the code above will be:

```
pen is stationery.
ruler is stationery.
eraser is stationery.
```

3.2.2 PNFG execution

A complete execution of a PNFG game includes two phases: an initialization, and a main fetch-and-process cycle. As shown in the Figure 3.1, at the beginning of the game, the start node is the only node activated, containing initialization activities. Once moving out of the start node, the system reaches the "idle" state and enters the second phase, which is the core of the game playing. In the second phase, the interpreter begins to wait and process user commands: when a command is received, the appropriate action is executed. If at any point either "game.win" or "game.lose" is set to be true, the game is terminated. Otherwise, once the action body has completed, threads are triggered and automatic counters (timers) are updated. Finally, control is returned to "idle", waiting for another command to start another repetition of the cycle, continuing until the game terminates.

3.3 Analysis on Narratives

As mentioned in section 1.1, this research is focused on finding the winning path of narratives. Prior to this work, verification attempts have been made on both Petri-Net that the NFG based on(low level), and the PNFG semantics(high-level), and consideration of the limitations and problems encountered in those earlier efforts has guided the work presented here.

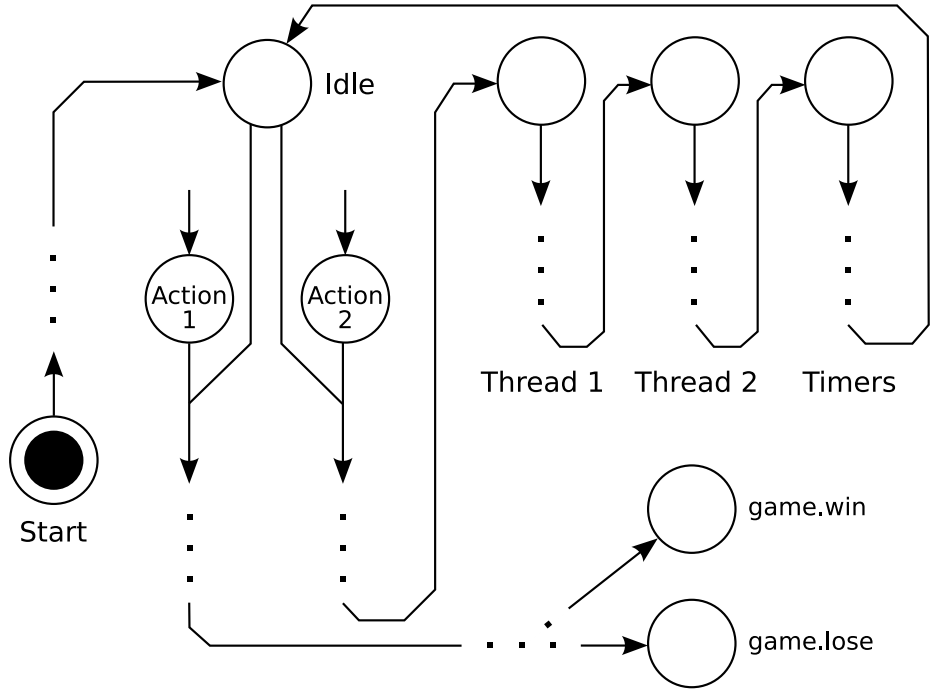

Figure 3.1 – The general NFG structure for a PNFG program

3.3.1 NuSMV Analysis

Previous work on PNFG verification was actually performed through analysis of the low-level NFG output from compiling a PNFG game. This low-level verification attempt was done using NuSMV [12], a symbolic model checker based on a Binary Decision Diagram (BDD) [4]. It is designed as an open architecture to be a generic verifier that has been applied to a wide variety of problem domains. To solve our problem, NFG Petri-Net code from the corresponding PNFG is translated to an acceptable format for NuSMV. The full game state in NFG level is fed to NuSMV and the goal is set to reaching the state "game.win". Full experimental results are available in [25].

According to the test data, NuSMV can be successful in finding the winning path of some narratives. Moreover, the result it produces is *optimal* — that is, it is always the shortest path. However, due to the representation characteristics of NFG code [41], the data passed to NuSMV is fairly large, even for small games that have no more than 10 steps in their solution; and it also requires a great amount of time and memory as well. For simple

narratives, NuSMV may complete in reasonable time on a modern machine, which has a dual core CPU and 4GB RAM, approximately several minutes to hours. Unfortunately the quickly scaling NFG problem space prevents NuSMV from solving narratives requiring more than six steps to win; in previous experiments, for instance, a NuSMV-based analysis of even a moderately-sized narrative, which requires about 10 steps to win, was unable to complete within two days. Details on the related experiments can be found in [25].

3.3.2 High Level Analysis

An alternative approach to verification is to use high-level analysis, basing verification on a custom heuristic solver [25]. As its description indicates, this approach operates in the PNFG level to take advantage of the information hidden in semantics. The design is actually more of a proof-of-concept, employing an unsound and incomplete strategy to ensure completion: for instance, global actions are ignored, since experience shows they are not required by the winning path of many narratives. Analysis at this high-level is much more coarse-grained, considering the actions in PNFG rather than transitions in the Petri-Net. As a result, the high-level analysis is much faster than NuSMV: searches in moderate narratives can be completed in minutes or even seconds. Full data is listed in [25].

For this approach, Martineau created a "backward" analysis to find out a winning path in the narrative. This proceeds by first finding all actions where the game is won (looking for "+game.win" statements) and searching inside the actions to find all branching conditions that must be passed through to reach those statements. The states being evaluated in these conditionals must be true in order to win the game, and so these states are then the object of a further recursive search. Chasing down the entire set of state dependencies produces a subset of actions that should contain the actual winning path. This smaller action set can then be subject to brute-force search.

Besides the condensed problem space, an advantage of this approach is in the way relationships among data are revealed by high-level information. Although the approach is fundamentally heuristic, it greatly accelerates the search by reducing the search width. On the other hand, because of the informality of the analysis, it cannot solve complex narratives. Furthermore, the solution it emits is not guaranteed to be the shortest, in the

right order, or even correct. The actual result is a “superset” of the winning path, which only guarantees actions in the winning set are all listed in the unordered result, assuming no global actions are required to win. However, the exciting improvement of this approach demonstrates the benefits of using high-level information, and encourages us to research further on high-level verification.

Chapter 4
Building the Dataflow Module

Although high level analyses have been applied to PNFG, they have been based on heuristics rather than a formal approach [25]. Such informality narrows down its usage and value of extending. Nevertheless such previous work encourages us to analyze narratives at a high level, and we are driven to seek a generic approach to formally traverse and gather information about PNFG code. In consideration of these requirements, *dataflow analysis* is the best candidate.

Dataflow analysis is a technique that can be applied to estimate the values of variables at different points in a program. This advanced technique in the area of compiler analysis is helpful for code optimization; the information, however, may be used to find the winning path too. Two directions may be used in dataflow: forward analysis is to get the information *after* the execution of a chunk of code, and backward analysis does the opposite work. Dataflow can be applied both *intra-procedurally* and *inter-procedurally*, where a procedure in common programming languages is roughly equivalent to an action in PNFG.

Distinct from common programming language like Java and C, PNFG is a user input driven scripting language. In Java and C, given a task, the program's behavior is predetermined, thus the routine of dataflow is clear. On the other hand, such properties do not exist in PNFG, since actions do not have any direct relationships and can be executed at any order according to the user's wishes. This unpredictability greatly widens the problem space of PNFG analysis. Some considerations, for example, the loop structure described in Section 3.2, also influenced the design and implementation of dataflow in the PNFG frame-

work, making it different from dataflow analysis on common languages. As an extension of the PNFG framework, the dataflow module should be generic so that it is not only suitable for our research, but also provides a base structure for further analyses.

In this chapter, we will describe our efforts and considerations when building the dataflow module from the ground up. First is the design and implementation of the Intermediate Representation and Control Flow Graph in Section 4.1; we then will discuss the structure of the abstract analyzers in Section 4.2, including the design of the flow set, and abstract code traversers.

4.1 The Control Flow Graph

In dataflow analysis, in order to gather data at certain points, instead of directly executing the source code, a special data structure called the *Control Flow Graph (CFG)* is used to traverse the code. A central idea of the CFG is that defines *basic blocks* of the code, each of which has exactly one entry point and one exit point, and it also creates forward/backward edges connecting these basic blocks so that the analyzer can traverse the graph. In PNFG actions are totally independent, and therefore one CFG is built for each corresponding action in the PNFG. All the other structures in PNFG do not need CFGs, because they cannot hold data operating statements.

4.1.1 The High-level Intermediate Representation

The first step of building the module is to complete the High-level Intermediate Representation (HIR) from the Abstract Syntax Tree (AST). HIR is an abstraction of source code, which generalizes and reforms the code in view of compiler requirements instead of the programmer, presenting a structural view of the source code.

To support the nature of PNFG code, the HIR is based on an aggregative class containing every element of the PNFG source code by its type, and providing functions to retrieve them, including:

Global Sets Since set declarations can be nested, they are "flattened" to contain only actual game objects.

Global Actions Contains the global actions only; local actions are recorded in the list of rooms described later.

Objects This array structure saves all game object declarations. Two auxiliary lists, one for Rooms, and the other for Items, are defined for further classification. Local actions are linked under the room they belong to.

Threads Threads are basically actions, thus they share the same structure as actions.

4.1.2 Basic Blocks

A basic block is a straight executed sequence of code, without any internal jumps, destinations of jumps, or branch codes in its body; in other words, a basic block has only one entry point and one exit point. Note that in PNFG only actions act as initial entry points — although PNFG supports "function" declarations, they are more like macros rather than real function definitions. The PNFG compiler automatically expands the code in-line instead of generating a function call. This simplifies the determination of the boundaries of basic blocks. In summary, in PNFG, an entry point for a basic block can be:

- Start point of an action;
- Merge point of an "if" statement;
- End point of a "for" statement.

and exit points are:

- End point of an action;
- Split point of an "if" statement;
- Start point of a "for" statement.

In the CFG, entry and exit points of basic blocks are embedded explicitly as extra nodes to increase awareness of switching environments. The CFG is built by recognizing statements and assigning corresponding nodes.

4.1.3 CFG nodes for actions

Each action has at least one basic block, and entry/exit points of the action correspond with the entry and exit of some basic blocks. The body of the action can be either one block

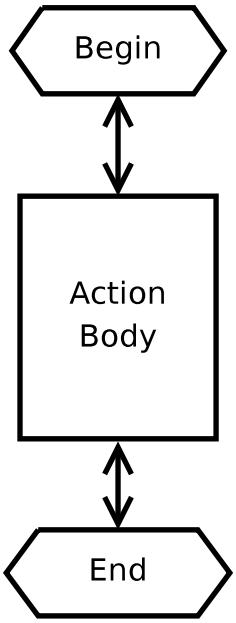

Figure 4.1 – The structure for an action in CFG, the upward arrows are used in backward analyses, and the downward ones are for forward analyses

if the action is completely straight, or multiple blocks otherwise. The structure for an action in CFG is shown in Figure 4.1.

4.1.4 CFG nodes for "if" statements

Each "if" statement creates two branches, splitting the CFG at the beginning of the statement and merging it afterwards. The basic structure for an "if" statement is shown in Figure 4.2. As in other programming languages, in PNFG, sometimes conditions are more than a simple boolean expression: conditions can be connected via boolean operators like "and" (`&&`) and "or" (`||`). To simplify condition evaluation when traversing, complex conditions are expanded in CFG building. According to the syntax of PNFG, only "and" and "or" are allowed to appear in compound condition expressions, and if the final value is determined by left hand side sub-expression, the other part is ignored. Also noteworthy is that the expanded code is required to evaluate each condition at most once to prevent redundant operation to unary operators like "++" and "−−". As a result, conditions connected with "and" are simplified as follows:

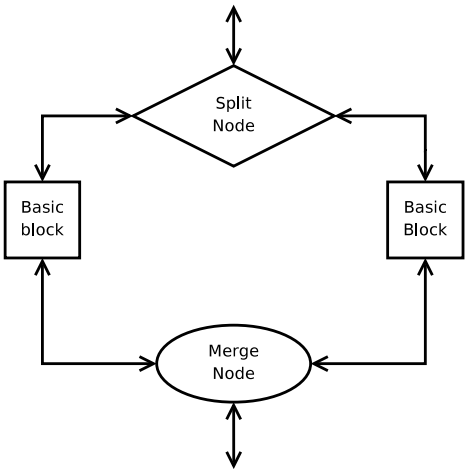

Figure 4.2 – The structure for an “if” statement in CFG

```
if (condition_A && condition_B) {
  then_statement list
} else {
  else_statement list
}
```

is converted to:

```
if (condition_A) {
  if (condition_B) {
    then_statement list
  } else {
    else_statement list
  }
} else {
  else_statement list
}
```

Similarly, a complex condition with “or”:

```
if (condition_A || condition_B) {
  then_statement list
```

```
3  } else {
4    else_statement list
5  }
```

is converted to:

```
1  if (condition_A) {
2    then_statement list
3  } else {
4    if (condition_B) {
5      then_statement list
6    } else {
7      else_statement list
8    }
9  }
```

Breaking down complex conditionals simplifies analysis. However, as seen above, part of the code is duplicated. Actions in PNFG are rarely longer than 30 lines, so this is not a serious problem.

4.1.5 CFG nodes for “for” statements

As described in Section 3.2, loops in PNFG are actually iterations over sets referenced by a temporary variable; in other words, the total number of loop iterations and values of the loop index variable in each iteration are pre-determined at compile time. Handling loops is thus mainly a process of loop elimination. We achieve this goal by unrolling the loop to a sequence of repeated loop bodies. A flattened loop is shown in Figure 4.3. In the CFG, not only is a pair of boundary nodes inserted for the start and end of the entire loop, but each iteration also has inner boundaries. The purpose of the outer boundary is to declare and discard the temporary loop variable, and the inner boundaries are responsible for updating its value. Although we get a longer CFG code, the CFG traverser gets much simpler because of the disappearance of loops.

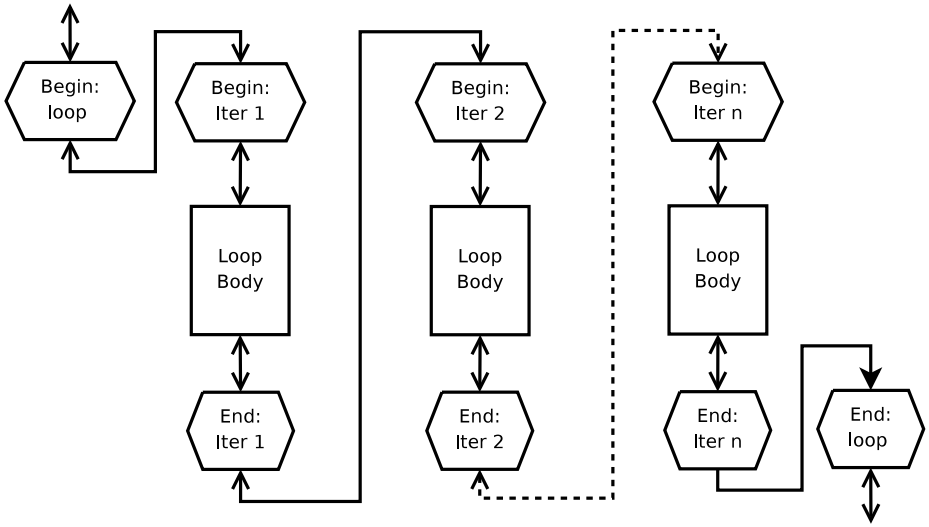

Figure 4.3 – The structure for an unrolled loop in CFG

4.1.6 CFG nodes for plain PNFG statements

Each plain PNFG statement, like an output statement, move statement or set statement, is embedded in one specific CFG node, and is inserted into the graph when building basic blocks. In our analysis on finding the winning path, we place primary concern on the data manipulation statements, while totally ignoring output statements. This is achieved by recognizing the type of the statements instead of directly removing them from the graph; in support of potential future work, they still exist in the CFG.

4.1.7 Algorithm for building the CFG

The CFG is built from top to bottom of the AST. In each level, all plain statements in the current level are inserted in the CFG; complex conditions and functions are also expanded. Structures for “if” and “for” statements are created, but branch and loop bodies are left unprocessed because they are at a lower level in the AST, and will be processed in the next pass. Such a procedure is repeated until we hit the bottom of the AST in a recursive traversal.

4.2 Building the Abstract Analysis Structure

Once the HIR and CFG are ready, we are able to traverse and analyze the code. Since our goal is to implement a universal dataflow analysis module, the next step is to build the abstract analysis, which serves as a generic CFG traverser and a skeleton for concrete analyzers.

4.2.1 The two domains

The first phase is to define the content and structure of the data we are interested in flowing through the PNFG code. As we are analyzing a game narrative and as others have observed [39], a game can be decomposed as a finite state machine, therefore, we should observe the change of the game state. In PNFG, changeable elements are:

- states of objects;
- counters of objects;
- timers;
- location of objects.

Considering that counters are just numerical values attributed to states of objects, and timers are actually automatic global counters, we integrate them all into the *state* domain. The range for elements in the state domain is shown in Figure 4.4. Besides all possible values in the middle, we place two extra values for merging: "$\top$", to indicate "can be any of the values" and "$\bot$" means "no information available".Locations are related to the whole object rather than any single property. They are organized into the *location* domain, whose value range is shown in Figure 4.5, and is quite similar to the state domain. All data in PNFG is statically declared, and thus both domains are finite.

4.2.2 Abstract flow set

The flow set is the structure holding game states, acting as a data container in the dataflow analysis. Fundamental operations to the two domains and the flow itself are defined in an abstract class, *AFlowSet*, and an interface *IFlowSet* is also extracted to provide a uniform interface with analyzers. Every concrete flow set is forced to inherit from this class

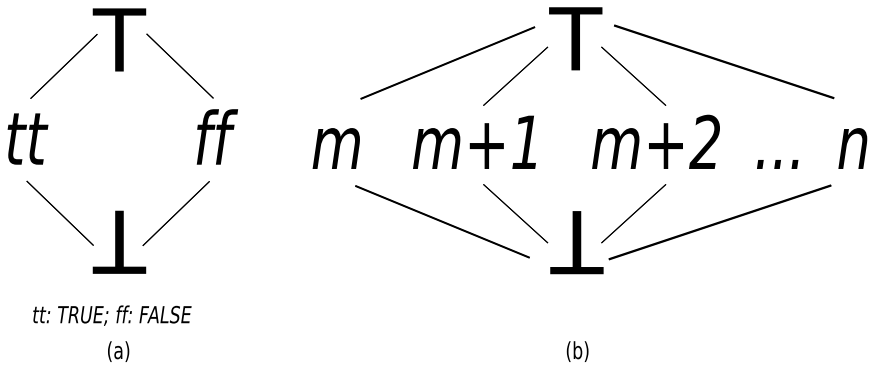

Figure 4.4 – The range of state domain, (a): boolean values; (b): numerical values (counter ranges)

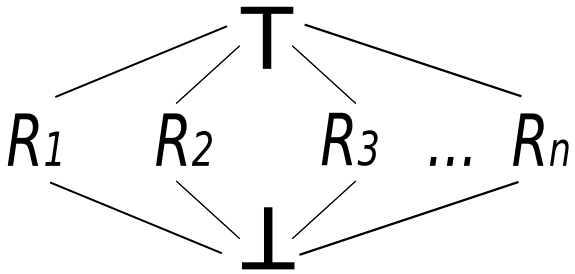

Figure 4.5 – The range of location domain, $R_1, R_2, R_3 \ldots R_n$ are rooms

and implement the interface. Primary abstract functions for child classes to implement are in two categories:

Initializations include setting the initial values for both state and location domains.

Merge Functions define the logical rule of merging two specific states or locations.

These functions are all labeled as protected, because they are only called internally, by the public initialization and merging functions. The primary public functions listed in the interface contains are:

Data Retrieve Functions return the value of a given state or location of an object to the user.

Data Store Functions give a new value to a state or location of an object according to corresponding PNFG statements.

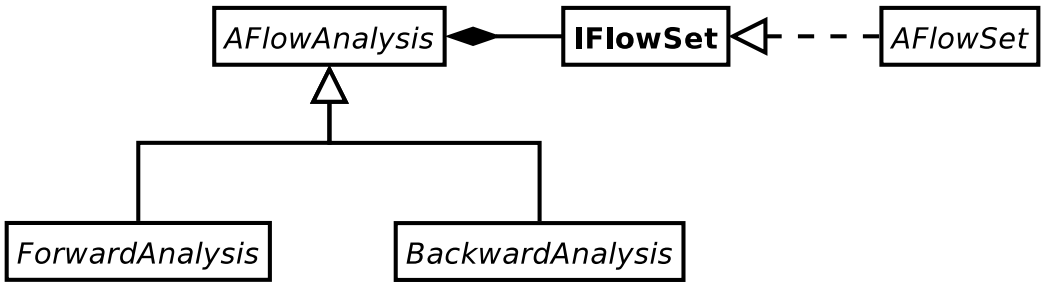

Figure 4.6 – The class hierarchy of abstract analyzers in UML diagram

Flow Set Merge Functions define the behavior of merging two flow sets with the help of internal merge functions.

Maintain Functions include creating a clone of the flow set, printing it out, and serializing the flow set.

4.2.3 Abstract analyzers

The abstract analyzers are actually CFG traversers without any flow set manipulation rules, which are left for concrete analyzers. Three analyzers are implemented according to the level of their abstraction. First of all, an abstract class *AFlowAnalysis* stands at the top of the hierarchy tree. It does not do anything about analysis, but implements common auxiliary functions which will be used by all analyzers, for example, functions to evaluate the value of an expression or a condition, change flow set according to a specific statement, and retrieve the value of a variable within a loop. The second level includes two abstract analyzers in opposite directions: the *ForwardAnalysis* and the *BackwardAnalysis*. They manage traversal of the code. As their names indicate, *ForwardAnalysis* traverses the CFG from the beginning to the end, while *BackwardAnalysis* work in reverse. Besides their directions, another notable difference of the two analyses is their handling of the conditions of "if" statements. In *ForwardAnalysis*, conditions can be evaluated, and branches are chosen by the evaluation result, but in *BackwardAnalysis*, since such evaluation is not defined in the merge point, it is ignored and both branches are traversed. Following both branches is only required in the *ForwardAnalysis* when the traverser cannot evaluate the condition from the current state. Figure 4.6 shows a UML class diagram of the abstract analyzers.

4.2.4 Creating a concrete analysis

The creation of a concrete analysis using our module consists of two steps:

1. Create a concrete flow set extending from *AbstractFlowSet*. Abstract functions like initialization and merging should be properly implemented.
2. Create a concrete analyzer by extending from either *ForwardAnalysis* or *BackwardAnalysis* according to circumstances. This requires implementation of functions for initialization, and most importantly, the function for flow set manipulation rules, which are automatically applied when corresponding CFG nodes are encountered.

More details on this topic will be discussed in next chapter, as we present our concrete set of analyses that attempt to locate the "winning path" in a PNFG narrative.

Chapter 5
Finding the Winning Path

The new dataflow module enables us to formally analyze the narrative at the PNFG level. We apply this new module to the problem finding the "winning path" in game narratives. Given a narrative, we analyze the PNFG code to determine a sequence of actions that would result in winning the game. To do this, we have several problems to overcome. First of all, we have to determine what data we need to achieve our goal, and how we collect it (Section 5.1). In Section 5.2 we discuss how we organize the data to form a clear view of the state machine, and to ease our search for the winning path. Finally, in Section 5.3 we present our drivers which can search out the winning path from the collected data.

5.1 The Action Summarization

Actions play an important role in PNFG narratives. As discussed in Section 2.3, a narrative can be viewed as a finite state machine, where the actions change the state. The winning path in this context, as shown by Martineau in [25], is a sequence of actions that drive the game state from the initial point to the final "game.win". In two other previous studies, Medler [26] and Natkin [31] analyze narratives by finding the logic relationships among actions. We thus approach finding the winning path by understanding the sequences of actions that may occur in the game. Writing a profile for each action to distinguish its behaviors with respect to changing the game state becomes our first step.

In PNFG, actions are independent and can be executed in any arbitrary sequences, and so a straightforward inter-procedural analysis will not scale well due to its non-context-free characteristics. Instead, intra-procedural analysis, which considers one single action each time, is a better solution. To profile changes in game state, two kinds of information are necessary: the prerequisites of an action — what must be true in order for an action to alter state, and the consequences of actions — what must be true after an action is executed. This approach allows actions to be linked together as a game state change sequence. Of course this requires an accuracy/time trade-off. On one side, the most accurate summarization of action can provide a more complete view of the transfer of game state but requires exploring and building the entire game state space. On the other extreme side, with no prerequisites and consequences information for actions, actions do not have any correlations, and thereafter can be followed by any other actions, making an extreme large search space for later verification. On the horn of the dilemma, we choose a conservative profile, generalizing all possible results to a pair of prerequisites and the consequences for each action. In this way we achieve a balance of searching space and time costs. This is performed by implementing a conservative flow analysis with the appropriate merging rule in our PNFG dataflow framework.

5.1.1 The common flow set

To use the dataflow analysis module, the first step is to create a concrete flow set. In finding the winning path, we implement a universal flow set class *CommonFlowSet* accommodating both the state and location domains. In the flow set, values of states and locations can be initialized as empty, all "$\top$", meaning any value within the range is possible, or they can assume some game state reached after the "start" node as part of analysis. It is also comes with a conservative merging rule. In both domains, merging two different values of the same key (state, counter, or object location) from two flow sets will definitely return a result of "$\top$" as shown in Table 5.1. The only exception is if either of the values is "$\bot$", then its value always overwrites others. For instance, given two flow set F_1 and F_2, and a state/object named "key", if:

$$In(key) = value_1 \in F_1$$

$$In(key) = value_2 \in F_2$$

and

$$value_1 ! = value_2$$

then after merging,

$$Out(key) = \top$$

Note that in the PNFG environment, most variables are global, and so they are associated with "$\top$" as initialization.

	$\top$	$value_1$	$value_2$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
$value_1$	$\top$	$value_1$	$\top$	$value_1$
$value_2$	$\top$	$\top$	$value_2$	$value_2$
$\bot$	$\top$	$value_1$	$value_2$	$\bot$

Table 5.1 – The merge rule of *CommonFlowSet*, for two domains.

5.1.2 Pre-condition Analysis

The pre-condition analysis is to find out the prerequisites for a *safe execution*. Safety, as described in Section 3.2, is one of the most important properties used in our analysis. The set statement and move statement in PNFG have strict requirements, and as is shown in previous section, in order to switch on state "w" for object "X" by "+X.w", "w" must be false beforehand, otherwise the whole system freezes. The same situation exists for move statements: to move object "X" from room "A" to "B" using the statement "move X from A to B", "X" *must* be in room "A". Pre-condition analysis is thus a backward analysis that determines the states of objects required for a given action to execute properly.

As an example of how the analysis works, consider the tiny "wizard" game adapted from Lindley's example narrative in [22]:

```
1 object dragon {
2   state {alive}
```

```
}

room village {
  (you,talk) {
    if (dragon.alive) {
      "The wizard tells you to kill the evil dragon.";
    } else {
      "You win";
      +game.win;
    }
  }

  (you,lair) {
    move you from village to lair;
    "You arrive in the lair";
  }
}

room lair {
  (you,attack) {
    if (lair contains dragon) {
      -dragon.alive;
      move dragon from lair to offscreen;
      "You've killed the evil dragon";
    }
  }

  (you,village) {
    move you from lair to village;
    "You leave the lair and head for the village.";
  }
}

//you start in the village
start {
  +dragon.alive;
  move you from offscreen to village;
```

```
40   move dragon from offscreen to lair;
41   "Type help for list of commands";
42   "You are in a village";
43   "A wizard is standing in front of you";
44 }
```

The CFG for the action (you, talk) along with the flow set at each point is shown in Figure 5.1. The flow set is initially empty, and starts at the end node. The first non-trivial node is the "merge" node, where in a backward analysis the flow unconditionally splits into two branches. In the "true" branch, there is only an single output statement, which is ignored in the analysis, thus the flow set is still empty when it reaches the "split" node with condition "dragon.live" = true. On the false path, it has a set statement "+game.win", thus requires "game.win" to be false before, and also the flow set is changed to:

{(game.win = false)}

As before the output statement is ignored. When we encounter the "split" node again, we have finished the traversal of both branches. When merging flow sets from branches, absent entries in one branch are assumed to be "$\top$". Thus the merged flow set is:

{(game.win = "$\top$")}

At the "begin" node, local constraints are inserted. For this action, it requires player to be in the room "village". With this addition we have the final pre-condition of the action.

According to our definition, the pre-condition for all the actions in "wizard" are:

(you, talk) in village:

{(game.win = $\top$); (you in village)}

(you, lair) in village:

{(you in village)}

(you, attack) in lair:

{(dragon.alive = $\top$); (dragon in $\top$)}

(you, village) in lair:

{(you in lair)}

The action "start" is executed only once at game initialization, and so it does not have pre-condition.

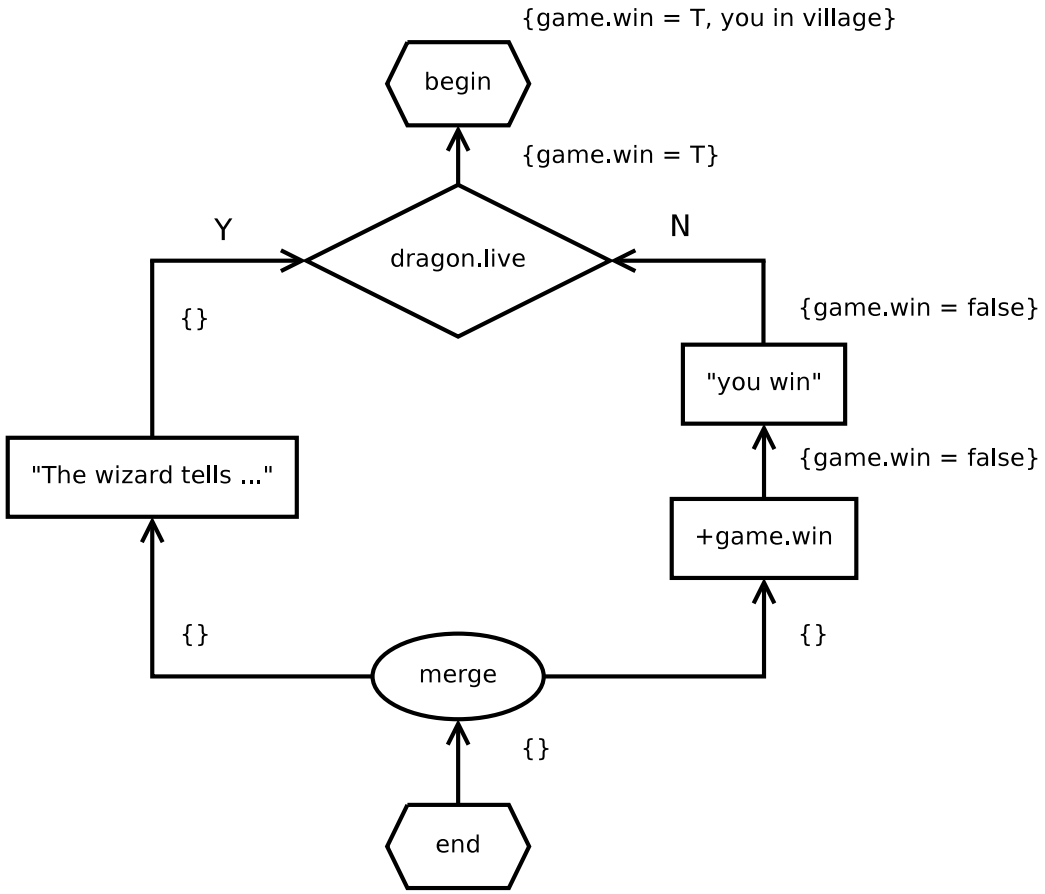

Figure 5.1 – The CFG and flow set at each point of action (you, talk) in village for pre-condition analysis

5.1.3 Post-condition Analysis

The post-condition analysis is quite similar to its pre-condition counterpart, but in the opposite direction. Its purpose is to compute the game state *after* the safe execution of an action. The use of the *CommonFlowSet*, and the same merging rule also make the analysis conservative. Another difference of this analysis is the use of a technique for improving flow analysis accuracy by taking advantage of how conditionals enforce certain states in each branch [42]: conditions in "if" statements are evaluated, so that it is always safe to claim corresponding values at the beginning of each branch.

Consider the same action, "(you, talk)" in village (Figure 5.2), as an example. At the

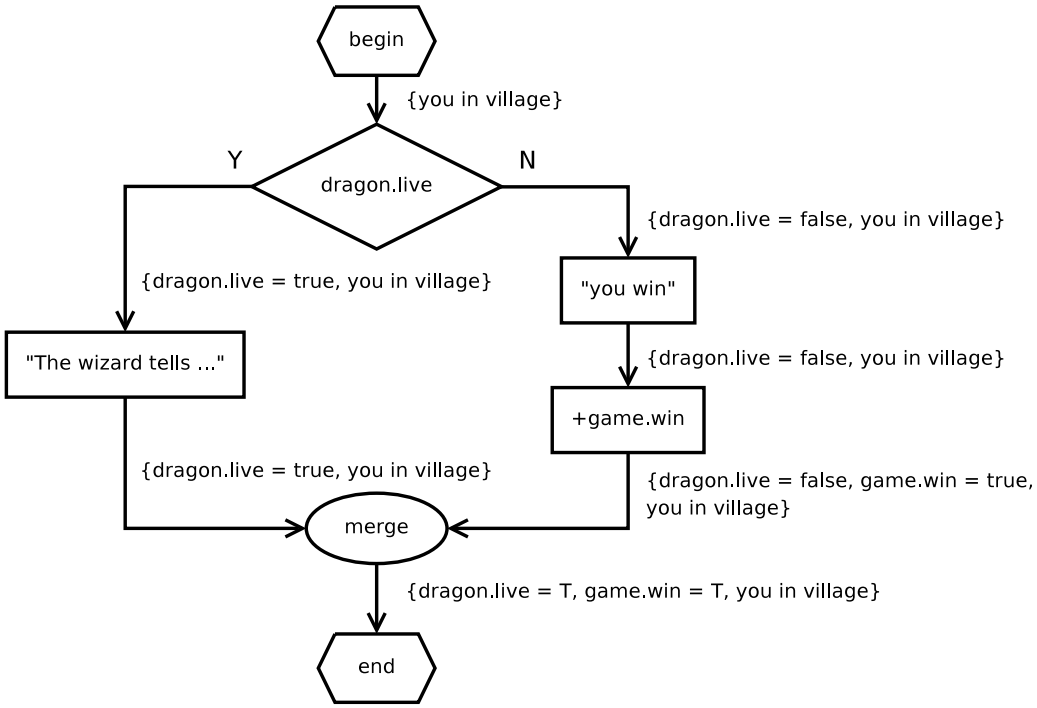

Figure 5.2 – The CFG and flow set at each point of action (you, talk) in village for post-condition analysis

beginning, the flow set is empty, except that since this action is scoped to being in the village, the constraint

{(you in village)}

is added. At the "split" node, since we know nothing about "dragon.live", no deterministic result can be made at this point, and both branches are traversed. On entering the "true" branch, however,

{(dragon.live = true)}

is added (and as before output statements are ignored). On the "false" branch, as well as adding

{(dragon.live = false)}

at the beginning of the branch, since the set statement sets "game.win" to be true,

{(game.win = true)}

is added, too. At the "merge" point, the location constraint is the same on both sides, and so is preserved. The dragon may be live or not, however, and similarly, the win state may

or may not be set, these are therefore merged to "⊤".

The post-conditions of all the actions in "wizard" are:

(you, talk) in village:

{(dragon.live = ⊤); (game.win = ⊤); (you in village)}

(you, lair) in village:

{(you in lair)}

(you, attack) in lair:

{(dragon.alive = ⊤); (dragon in ⊤); (you in lair)}

(you, village) in lair:

{(you in village)}

start

{(dragon.alive = true); (you in village); (dragon in lair)}

5.2 The Action Dependency Graph

The action summarization is basic material, from which we get some information. We now know actions have their requirements and can generate certain results, and so actions are not entirely independent at this stage with respect to changing the game state. However, the action summarization is a discrete analysis based on individual actions. A process is necessary to reveal the relationships among actions in terms of the pre- and post-conditions; we call this the "Action Dependency Graph" (ADG). By building the ADG it also creates the inter-procedural search space from our intra-procedural results. By basing the ADG on conservative action summarization we are able to ensure the ADG is conservative as well.

5.2.1 The algorithm

The ADG is a graph structure, where each node is an action, and if one action, namely A, can be safely executed after another action, namely B, a directed edge is added from B to A. The algorithm for building the graph is simple, for each action, we consider its post-condition, and then check the pre-conditions of all actions. If the pre-conditions of some actions *match* the post-conditions of A, then an edge is created between the two nodes.

The matching rule is defined as follows. A pair of pre-condition and post-condition are matched if for every state and location in the pre-condition, the corresponding entry in the post-condition has the same value, or "$\top$". The reason to include "$\top$" comes from the nature of the value in action summarizations. "$\top$" logically means it can be "any possible value" by the definition. Thus as a conservative analysis, a "may" connection should be established in this case if some prerequisite *might* be satisfied by a post-condition. Note that because it is impossible in PNFG to encounter a variable prior to its initialization, we never need to merge with "$\bot$".

Figure 5.3 shows the ADG for the "wizard" in which the location restraints are attached in square brackets before the action commands for local actions. Two special node "start" and "win" indicate the starting and ending of the game. Taking the action "(you, talk)" in village as an example for the third time, in the ADG it has two incoming edges, two outgoing edges and one self-loop. The pre-condition of this action is:

{(game.win = $\top$); (you in village)}

According to our matching rule, the "$\top$" is effectively ignored, and thus the only constraint for the action is the location of "you". Therefore in the post-condition results set, both "start" and "(you, village)" in lair are all possible followups. The same matching is checked from the inverse perspective as well. The post-condition of the action is:

{(dragon.live = $\top$); (game.win = $\top$); (you in village)}

which matches the pre-condition of both "(you, lair)" in village and "(you, talk)" itself. In the ADG, "win" is not a real action but a special node indicating termination of the game by winning. A special edge is therefore also added to the "win" node. All the other edges are created similarly, forming the entire ADG.

5.2.2 The State View of ADG

In this version of the ADG, actions are the primary focus, since it is valuable for us to trace the sequences of actions. However, in searching for the winning path, in many cases, as we are driving the finite state machine of the game, we are looking for the potential future steps given a certain game state. This inspire us to create another view of the graph, which is focused on game state rather than actions, and hence in which actions are edges

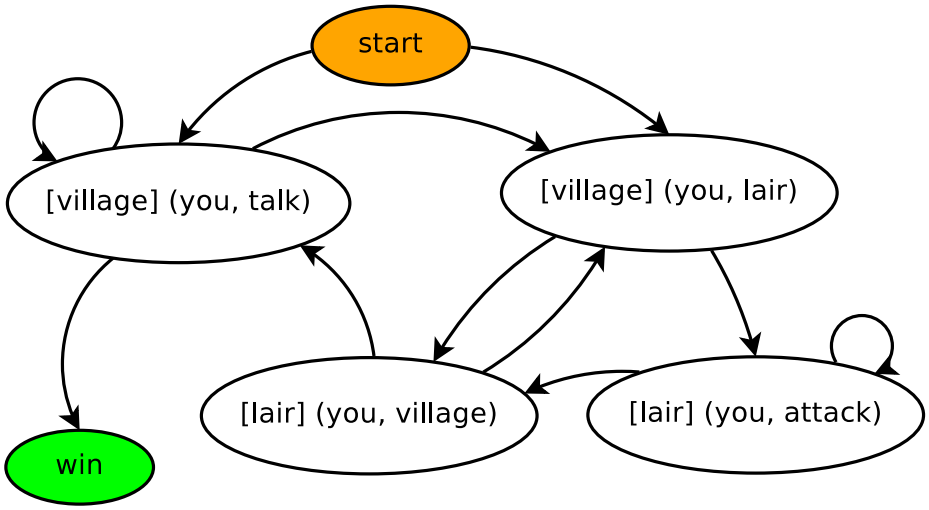

Figure 5.3 – Action Dependency Graph for "wizard"

instead of nodes. The algorithm is quite similar to the action view version: we start from the initial game state after the "start" action, if any pre-conditions of actions match the state, an edge labeled with the execution of the action is created from the current state to a new state, which is the post-condition of the chosen action. This process is continued until no more states are created. The matching rule is identical to the one in the first version. An example of this view of the ADG is shown in Figure 5.4, also for the game "wizard".

5.2.3 ADG as a search space

A primary role of the ADG is in acting as a search space to reduce the potential choices to avoid a brute-force search. In a brute-force approach every action can be followed by all the actions, and so conceptually the number of edges in the graph is $O(n^2)$. The ADG successfully shrinks the size of the graph, and heuristically the search space thereby, up to about 85%. Table 5.2 shows this comparison, in which the games being tested are from the benchmark set of the PNFG framework (more details will be given in Section 6.1).

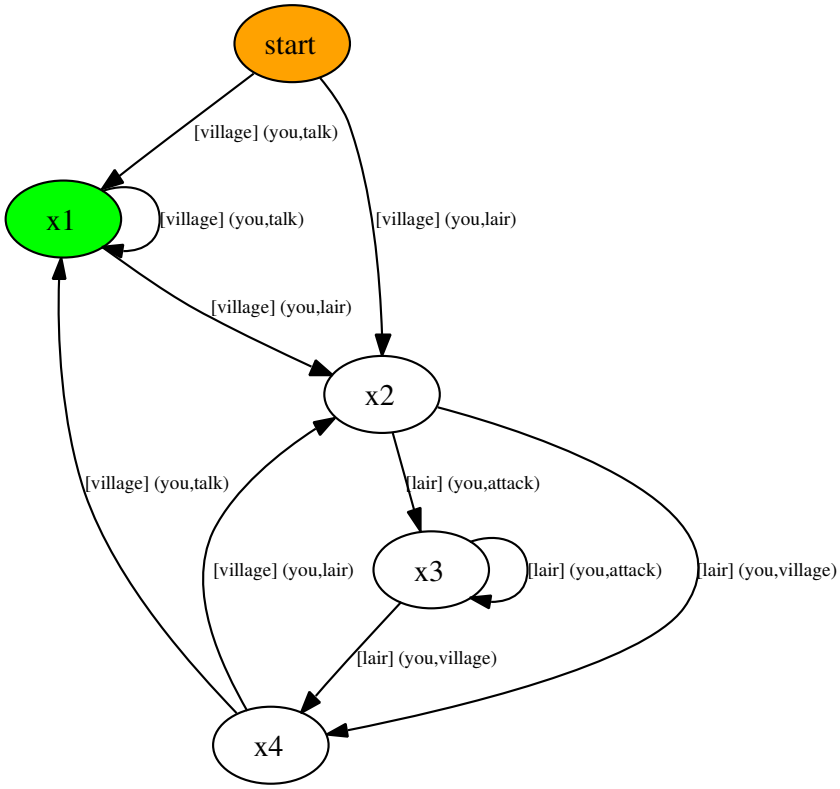

Figure 5.4 – State View of the Action Dependency Graph for "wizard"

	RTZ-task01 full	RTZ-task01 wbp	RTZ-task02	RTZ-task02 full
Actions	46	48	40	123
Nodes (Action View)	46	48	40	123
Edges (Action View)	645 (30.48%)	696 (30.21%)	417 (26.06%)	2301 (15.21%)
Nodes (State View)	44	46	37	117
Edges (State View)	595 (28.12%)	644 (27.95%)	370 (23.13%)	2155 (14.24%)

Table 5.2 – The size of two views of ADG, and the proportion to the size of brute force connectivity

5.3 The NFG Player

The winning path is a sequence of commands, and therefore an inter-procedural search is necessary for finding it through the ADG, which acts as the bridge between intra-procedural summarization and inter-procedural search. However, because of the conservativeness of our analyses, some action sequences (state changes in the state view) in the ADG are actually not allowed at run time. For example, in Figure 5.4, "start" directly jumping to the ending point "X1" is prohibited in reality. As well some actions may generate cycles, getting stuck in a game state: a sequence of the two move actions in the game "wizard" will move the player between "village" and "lair" back and forth without changing the game state in a meaningful way. These drawbacks of static analysis force us to make our search more dynamic, and in our solution, the *NFG player* is built to perform this work. The NFG player is a dynamic driver, and basically it has three responsibilities: to find potential commands in the ADG according to the game state, simulate playing the game action, and notice the feedback due to playing and update the game state accordingly. As the name implies, it works at the NFG level, but as it is just playing the commands in NFG as the PNFG code directs, all the work is done at a high level.

5.3.1 Basic algorithm

The NFG player uses the state view ADG and the pseudo code as shown in Algorithm 1. The basic idea of the NFG player is a depth first search of the ADG, pretty much like the save/load strategy of human players. It reads the game state, get the potential commands from the ADG and tries them one by one. If an action can drive the game to a new state it is considered to be potentially in the winning path, and the new game state is checked in the next repetition. If "bad things" like game loss or a previous state is encountered again, the game state is rolled back to pick another candidate. To make sure the algorithm terminates, a maximum search depth is defined. Every command added to the winning path increases the current search depth, and the algorithm quits with failure if the maximum search depth is reached on all branches, or with success in Line 9 if at least one winning path is found.

```
1  initialization;
2  while not reach MAX search depth do
3  |  get next command from ADG according to game state;
4  |  perform the command in NFG ;
5  |  if game.lose == true then
6  |  |  roll back game state;
7  |  else
8  |  |  if game.win == true then
9  |  |  |  output the winning path;
10 |  |  else
11 |  |  |  add command to winning path;
12 |  |  |  save current game state;
13 |  |  |  search depth++;
14 |  |  end
15 |  end
16 end
```

Algorithm 1: Basic algorithm of NFG player

5.3.2 Variants

Besides the basic version of the NFG player, several different versions of the NFG player are available as well, attempting to optimize the search. The structures explored fall within four categories:

Null move or cycle proof A *null move* is an execution of an action sequence without changing the game state. Although this is not a failure, it indicates an unnecessary action in the search and thus should be eliminated. Another consideration is cycles. A *cycle* is a repetition of game states in an action sequence. For example, after executing action "A", "B", and "C", if the game state returns to the initial state before "A" is triggered, a cycle is generated. Cycles are essentially larger null moves and should be removed as well. To detect cycles and null moves, the NFG player checks the game state after an execution, comparing to the previous state after each action in the action chain. If two states are identical, a cycle or a null move is found, and the current search is considered a failure, causing backtracking.

Bad states cache A *bad state* is not an "incorrect" part of the narrative, but a state that

cannot win the game. If from one state, all of its successors in the ADG have no positive results including only losing the game, null moves, or cycles back to a previous state, this current state is a "bad state" in the winning path search, and should be discarded. This feature is embedded in every variant of NFG player; however, we have an option to switch it off.

Potential commands preprocessing The NFG player's performance can be sensitive to the order of commands in the search list from the ADG. Ideally, if the "optimal" choices are always in the first places of the list, the player reaches the best performance. Of course we cannot define this order prior to searching. We can, however, sort them alphabetically by name, or randomize the list. Sorting will generate the same result for every run; however, it may or may not result in an efficient search, and although results from randomized lists may vary, they are likely to be encouraging.

The experimental results of variants of NFG players will be discussed in Section 6.2.

5.4 Optimizations

The NFG player successfully finds the winning paths out of most games in our benchmark set (details in Chapter 6). But there are still some games that cannot be solved in reasonable time. For example, the game "RTZ-task02 (full)" requires over three hours on average to complete the search on a high performance computer — compared to an entire weekend with no result by the previous low-level analysis by Pickett et al. [32], it is a great advance, though it could be faster still with the power of formal dataflow analysis. This drives us to the optimization phase of our study. All of our optimizations are also dataflow analyses to either filter the action list from the ADG used by the NFG player, or to prune the edges of the ADG to narrow down the search space from different aspects.

5.4.1 The Accurate Match

As we have noticed in our benchmark set, in many PNFG games, actions, especially local actions, tend to have implicit constraints as well as the explicit location constraint.

These constraints are in form of conditions from one of more “if” statements, with all the data manipulation embraced within a single branch. This can be found in our tiny example game “wizard” in Section 5.1, in the action “(you, talk)” in “village” and “(you, attack)” in “lair”. Obviously, if the current game state does not match these conditions, these actions should not be considered as candidates. Since such pointless actions will cause the NFG player to backtrack, filtering out these improper actions before passing them to the NFG player may speed up the search.

The accurate match is a dataflow analysis to recognize such actions, and precalculate their implicit constraints so that at runtime, actions whose constraints are conflict with the current game state can be filtered out to lower the rate of failure choice of the NFG player.

At the current stage of our study, we consider actions that have only one big “if” statement and without an “else” branch, which, from observation, encompass a large percentage of all actions. Implicit constraints are added to the pre-condition results; at runtime, a filter is used to convert these constraints in PNFG form to match the NFG state for comparison with the game state by the NFG player. Unfortunately, this optimization is not as effective as we expected. Details will be discussed in Chapter 6.

5.4.2 Useless Objects/Actions Analysis

As described by Rabin [34], an interesting game always provides more objects and actions than necessary, and this is an important consideration related to the convexity of narrative. It is a way to hide the winning path making it less obvious; another purpose is to please players with more reality in the game world. A widely used technique for such window-dressing in PNFG, as in many games, is to present a unique message about discovering the new room when a player first visits a room. In the future, a player returning to the same room will only get a plain message telling of his/her return. This feature is simply enabled and controlled by a state of the room, flipped after the first visit. Unfortunately, these objects and actions add complexity to the search space, and a lot of time can be wasted in verification due to exploring on these initial game room states, slowing down the performance greatly.

Although these features and items do have significance of existence in narratives, they

do not function in the winning path. Thus we name them "useless" objects/actions, and object states. In PNFG, at the current stage of our study, they are defined as below:

Useless State is a state of an object that is not read by any non-useless actions, or is referenced only to reassign itself.

Useless Object is related to the location domain only. It is an object whose location information is not read by any non-useless actions.

Useless Action is an action which does not change any non-useless states of objects nor move any non-useless objects.

The definitions are not complete though. Cyclic references are not always included. For example, if a pair of variables, namely "a" and "b", are referenced only to change the value of each other, they form a cyclic reference, and cannot be caught by our algorithm. However, this situation is rarely found in our benchmark set, and we could upgrade our algorithm if necessary when the benchmark set expands and requires this feature. Actually, the complete useless object/action set cannot be computed before the winning path is found, because all the objects/actions out of the winning path should be considered "useless" to win the game.

The definitions are recursive, and so it is necessary to compute a fix point of the useless set. We start the analysis by assuming everything is useless except default objects like "you", "offscreen" and "game.win", *etc.*, and diminish the set by finding non-useless objects/actions following the definitions until the set is stable. Since this analysis is performed before the building of the ADG, useless objects/actions are removed directly in the ADG.

The time needed for running the analysis depends on the size of narratives, but takes quite little, normally less than ten seconds on our test bed. Compared to the great time reduction in finding the winning path it brings, the overhead can be completely ignored. Details of our experiments will be shown in Chapter 6.

5.4.3 Winning Set

The heuristic analysis mentioned in Section 3.3 gives us another clue to prune the ADG. Recall that although the high level heuristic analysis cannot calculate the exact winning

path, it successfully creates a super set containing the winning path, and the size of the super set is fairly small. The only shortcoming is its efficiency, which is mostly because of the informality of the analysis. The winning set is actually a formal dataflow version of the same idea.

The winning set is a backward analysis similar to the heuristic analysis. The goal of the search is to prune the ADG to create a smaller search space for the NFG player. It starts with an empty set, and finds actions that can reach the winning state {game.win = true}. Once matched actions are found, they are inserted into the winning set; both the implicit and explicit constraints reaching the target state are recorded. For each newly added action, its constraint is set as a new target and another search is initialized. This process is continued until the winning set stops growing.

The winning set is computed before running the NFG player. Instead of removing edges from the ADG directly it acts as a filter when the NFG player is choosing new actions to perform: the player will consider actions in the winning set only. For our benchmark narratives, the winning set greatly reduces the number of choices, especially bad tries in the search, and improves the performance tremendously. Details of experimental results will be discussed in the next chapter.

Chapter 6
Evaluation

In this chapter we will show the experimental results we have due to our study on winning path searching. As our study is a search, the primary concern of our evaluation is the time elapsed and success of the search as well. After the introduction of our test environment and strategy in Section 6.1, the performance of variant NFG players is shown in Section 6.2, and performance of optimizations and options are shown in Section 6.3. In addition, we will also analyze and explain the interesting patterns we found in the result data at the end of the chapter.

6.1 Test Settings

Our basic test environment included a reasonable suite of benchmarks run on a modern machine, and measured in several ways.

6.1.1 Test cases

Our benchmarks originate from two sources:

PNFG framework examples selection Selected examples are adapted from famous IF or adventure games and other academic works on narrative. “Wizard” is a very small game from Lindley’s study [22], and “Cloak of Darkness” (cod for short), and versions of the first two chapters of “Return to Zork” (“RTZ” for short) are also used,

including fully ported and implemented versions with side-quests (RTZ-task01 (full), RTZ-task02 (full)), and a variant of the first chapter that ensures all tasks necessary to successfully complete later chapters have been performed (RTZ-task01 (wbp)) [20]. These benchmarks are rewritten in PNFG primarily to show the abilities and features of the framework. The size of benchmarks in this category varies from less than 100 lines to more than 1000 lines as shown in Table 6.1. Note that the enter/exit blocks of rooms are also counted as actions, because they also contains PNFG statements and can be triggered, though automatically. In addition, the game "wizard" is listed only as a comparison standard, we do not run any experiments on it because it is too simple to solve.

	wizard	cod	RTZ-task01	RTZ-task01	RTZ-task02
		full	full	wbp	full
PNFG Lines	61	535	563	583	1133
Actions	5	60	88	92	222
Objects	4	5	29	29	57
States	1	6	8	8	14
Steps to win	4	6	6	11	20

Table 6.1 – The properties of benchmarks in PNFG examples set

"The Three Little Pigs" adaptations selection As homework for the course "Modern Computer Games" (COMP521) provided in McGill University, 2007, students were asked to adapt the well-known fairy tale for the PNFG framework. Their games have to contain ten objects/rooms at minimum, at least three ways to win/lose, a non-trivial cycle, and other features, making them large and complex. The details are listed in Table 6.2 with the authors' IDs as their names.

6.1.2 Test environment

All of our experiments are run on a computer with the following configuration:

- CPU: AMD Althon 64 bit Dual Core 3800+
- RAM: 4GB
- Operating System: Ubuntu 7.10 with Linux Kernel 2.6.22

	csimon16	dpomer	dprykh	hsafad	mcheva	sdesja8
PNFG Lines	892	367	1472	387	775	775
Actions	58	58	104	88	62	110
Objects	23	18	17	22	22	19
States	8	3	21	16	4	33
Steps to win	8	4	13	12	26	17

Table 6.2 – The properties of benchmarks of "The Three Little Pigs" adaptations

– Java Virtual Machine: Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0)

6.1.3 Testing options

There are several ways to measure the efficiency of searching the winning path. Variations we consider are as follows:

First solution only Most non-trivial games have many possible command sequences that can reach a winning state. By default, the NFG player returns all solutions it finds within the maximum searching depth. Our approach is to focus on the time/cost to find the *first* solution. We have multiple reasons for our decision: first, the current state of our study are focused on *finding* the winning path, and not necessarily on analyzing, or comparing convexity, or other global game properties: one winning path is enough as long as it is certainly correct. Second, solutions are often extremely similar. As the examples in PNFG framework are all single-solution games, most of the results are quite similar, with only slight differences in the sequences of minor or redundant actions. The "three little pigs" adaptations can have multiple solutions, although, results are still easily classified into several categories. Exhaustively listing all solutions, including the mostly identical ones, is quite time consuming and as we witnessed in early experiments, tends to scale poorly, reducing the performance of all approaches to a basic function of number of solutions and search depth. A single solution output better measures the search cost without being as overwhelmed by the size of the solution itself.

Search depth The search depth is a factor not only defining the upper boundary of the

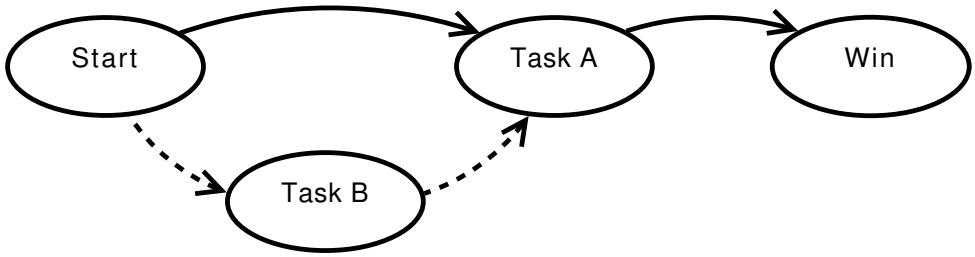

Figure 6.1 – The simple narrative with an optional task

search space, but also affecting the redundancy of the winning path. The "redundancy" basically refers to unnecessary moves that do not directly related to winning in the path. A larger depth tends to produce greater redundancy in the winning path by allowing more such moves or even cycles. For example, many games have optional tasks in form of a branch off of the main winning path, and completing these tasks will definitely result in a longer winning path. A single example is shown in Figure 6.1. The minimum steps to win is two: start, *task A*, and win, but it grows to three if the optional task, *task B*, is included. If we limit the search depth to three, and choose the *task B* branch, the first winning path found will be start, *task B*, *task As*, win. In out tests, we consider a range of search depths from *d* to *1.5d* for each narrative, where *d* is the length of the minimum winning path.

Repetition As mentioned in Section 5.3, the order of actions in the list influences the execution of the NFG player. To get an average performance for searching, we randomize the command list, and run the test multiple times with the same options. Tests on small (1–10 steps in the minimum winning path) and medium (11–15 steps) narratives are repeated 10 times, and tests on large (more than 15 steps) narratives are run 6 times. With a complex state space and random choice there can be great variance in search time and success; these values are chosen as a trade-off between confidence in results and practical testing time.

6.2 Results for Variant NFG Players

Analyses of the game narratives were done comparing the performance of four variations in the NFG players, as mentioned in Section 5.3.

"Normal" player basically is a brute-force approach.

"Null Move Removal" player removes single-step non-sense empty moves.

"Cycle Detection" player disable sequences containing cyclic state. Note that this strategy is a superset of the "null move removal" strategy, since null moves imply a trivial cycle.

"No Cache" player as well as cycle detection, this player also disables the bad-state cache trying to lower memory usage.

Table 6.3 provides a clear view of the abilities of the four NFG players.

	Null Move Removal	Cycle Detection	Bad State Cache
Normal			
Null Move Removal Player	✓		
Cycle Detection Player	✓	✓	✓
No Cache Player	✓	✓	

Table 6.3 – The abilities of PNFG players

Figure 6.2, Figure 6.3, Figure 6.4 and Figure 6.5 shows the average time of each NFG player on solving small narratives (1–10 steps to win) "dpomer", "RTZ task 01 (full)", "cod (full)" and "csimon16" respectively. The most notable fact from these figures is that the "normal" player can only solve the first game in a reasonable time; on the rest of the three narratives it takes much longer than its counterparts (a factor of over 200) and therefore its data is ignored in the rest of the figures. The first two narratives can be finished in less than 10 seconds by the players, so very few significant differences in performances can be detected. Caching helps a lot in "dpomer", but does not improve performance for "RTZ task 01". However, when solving the last two games, although they are just a little bit larger, the performance of the "null move removal" players varies greatly: although it is the best solution for "cod (full)" it becomes the worst by far in "csimon16". In short games cycles

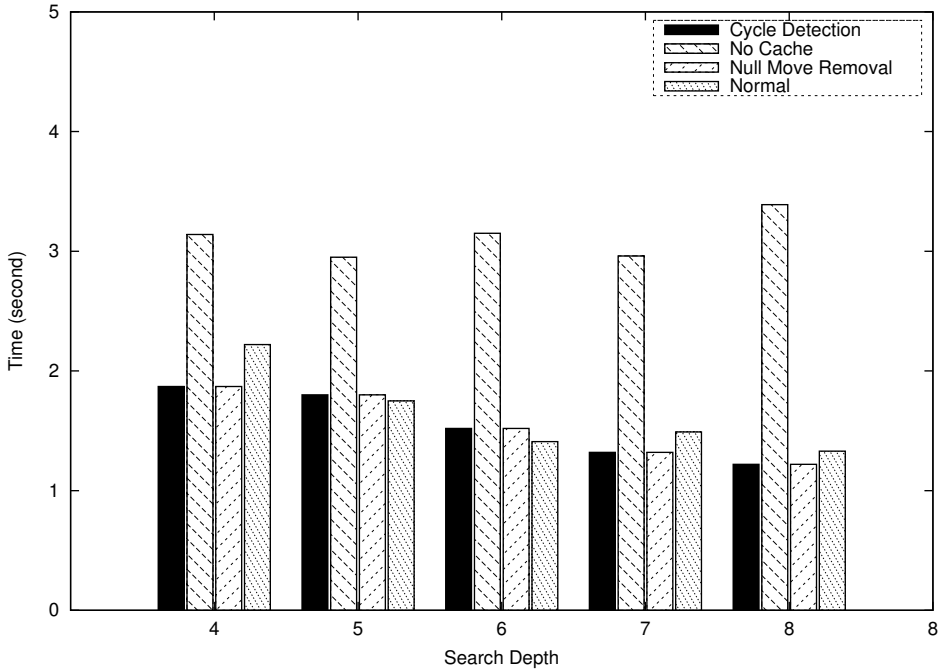

Figure 6.2 – NFG players on the game "dpomer"

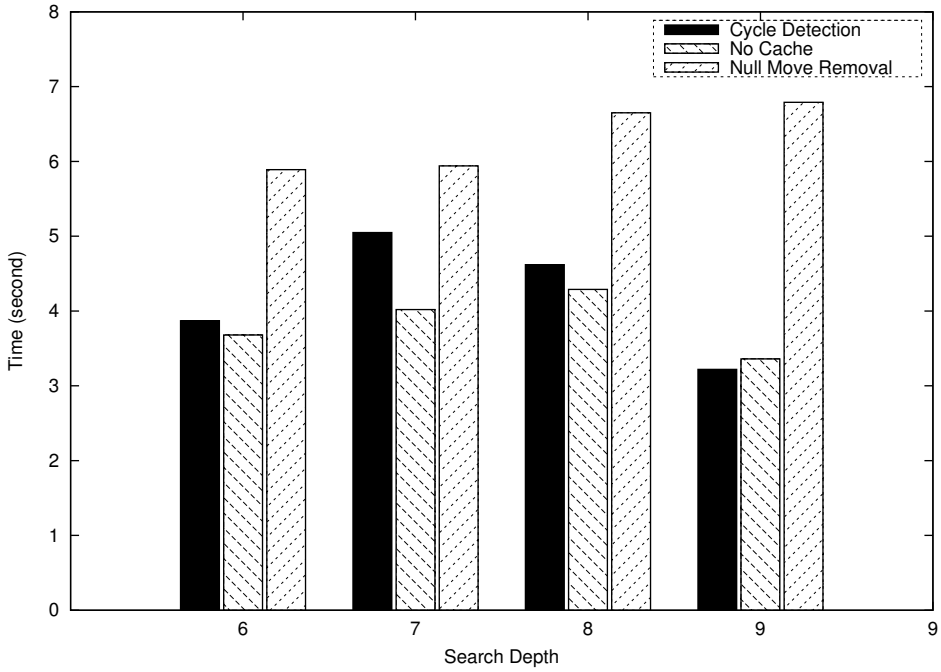

Figure 6.3 – NFG players on the game "RTZ task 01 (full)"

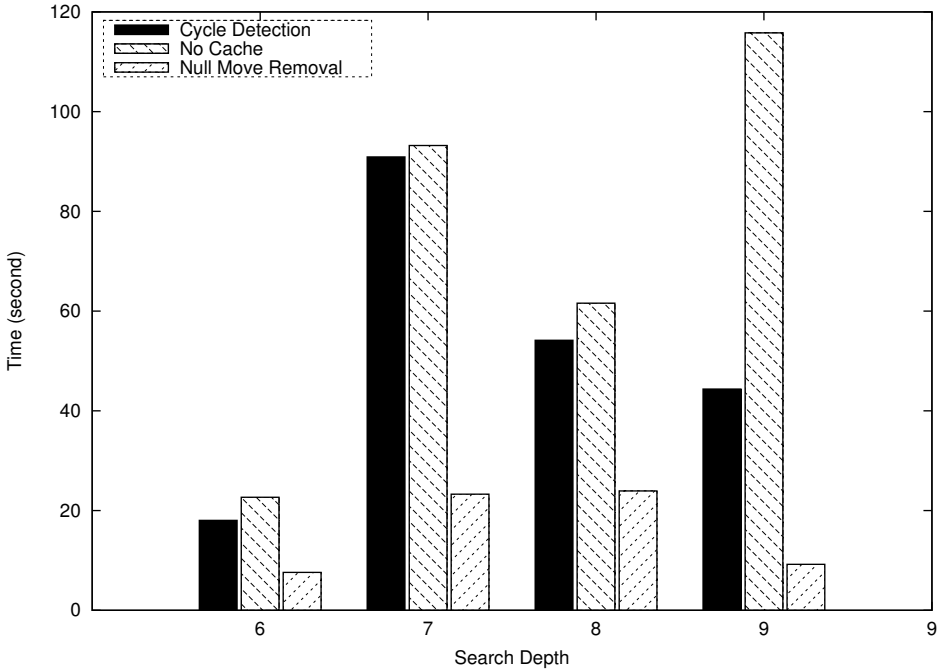

Figure 6.4 – NFG players on the game "cod (full)"

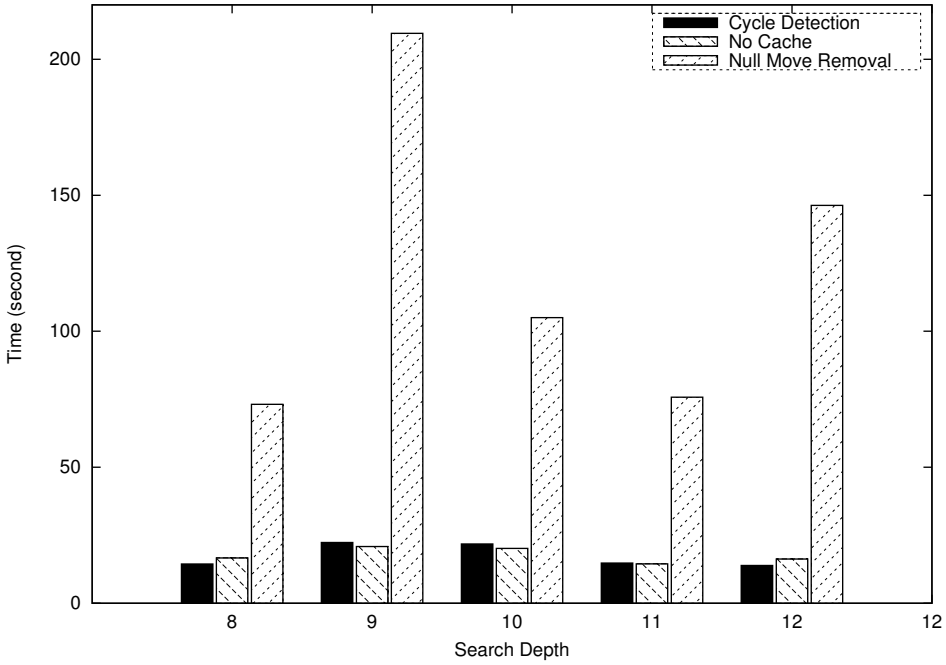

Figure 6.5 – NFG players on the game "csimon16"

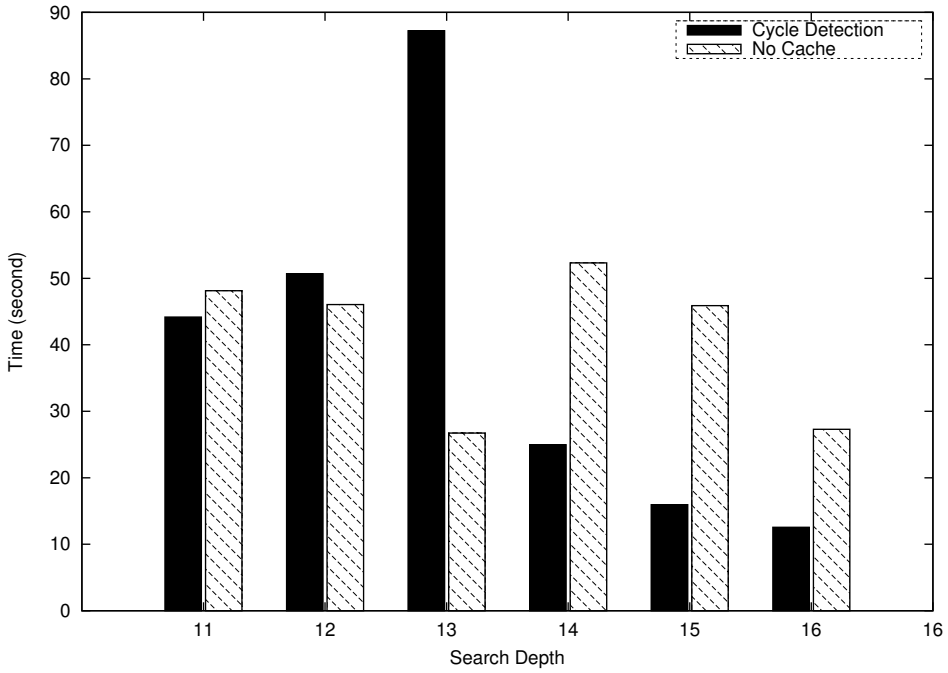

Figure 6.6 – NFG players on the game "RTZ task 01 (wbp)"

mainly consist of 1-step null moves, and thus a cheaper solution gives better tradeoff. The "cycle detection" players with/without cache are roughly similar in this test set.

Figure 6.6, Figure 6.7 and Figure 6.8 show the performance of NFG players on medium sized narratives (11–15 steps to win), "RTZ task 01 (wbp)", "hsafad" and "dprykh" respectively. On these benchmarks, not only the "normal" version but also the "null move removal" player is not able to finish in at least 1000 seconds and thus omitted in these figures. This is unsurprising as predicted above, the "null move removal" is not typically applicable to larger narratives. The time for the successful players returning the winning path is about 10 times longer than that of running on the small sized benchmarks, mainly due to the growth in the problem space. Again, in most cases, the performance of the last two players are at the same level; however, the player without a bad state cache fails to solve "dprykh", the most complex narrative in this set. By looking at runtime traces, the search encounters a flood of bad state rollbacks, blinding the player from finding the correct path.

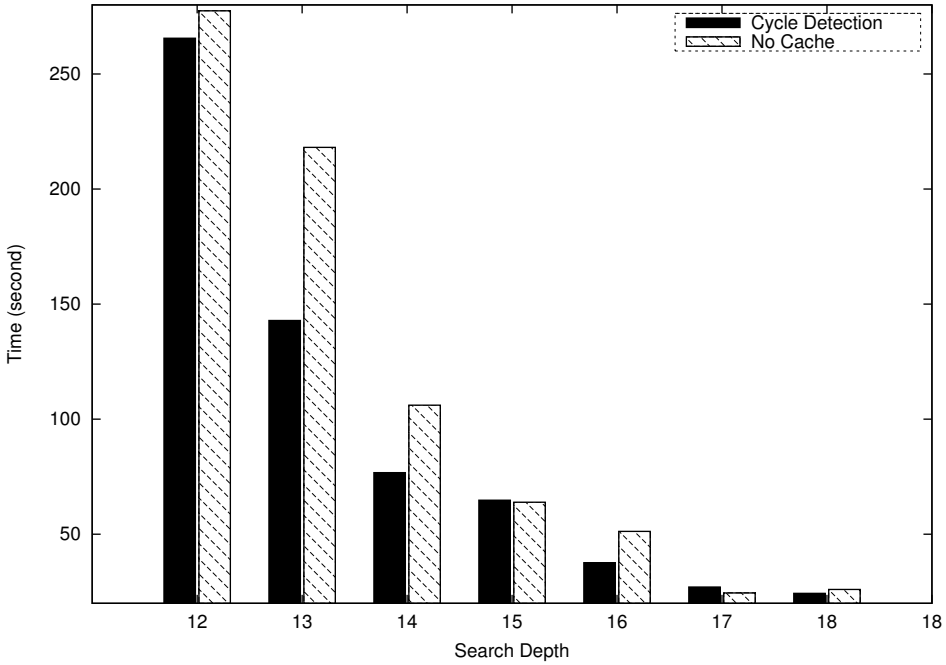

Figure 6.7 – NFG players on the game "hsafad"

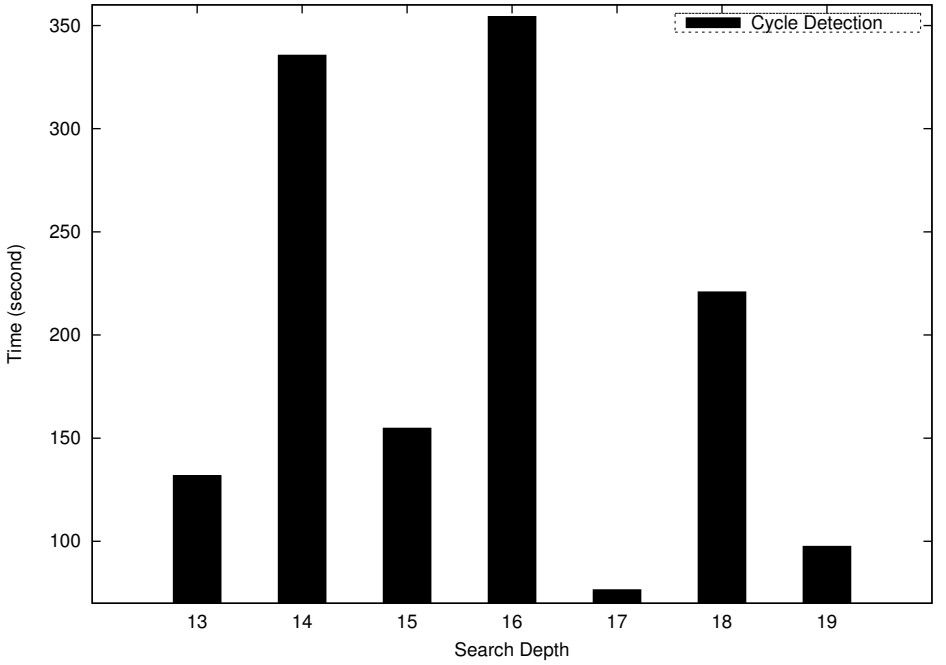

Figure 6.8 – NFG players on the game "dprykh"

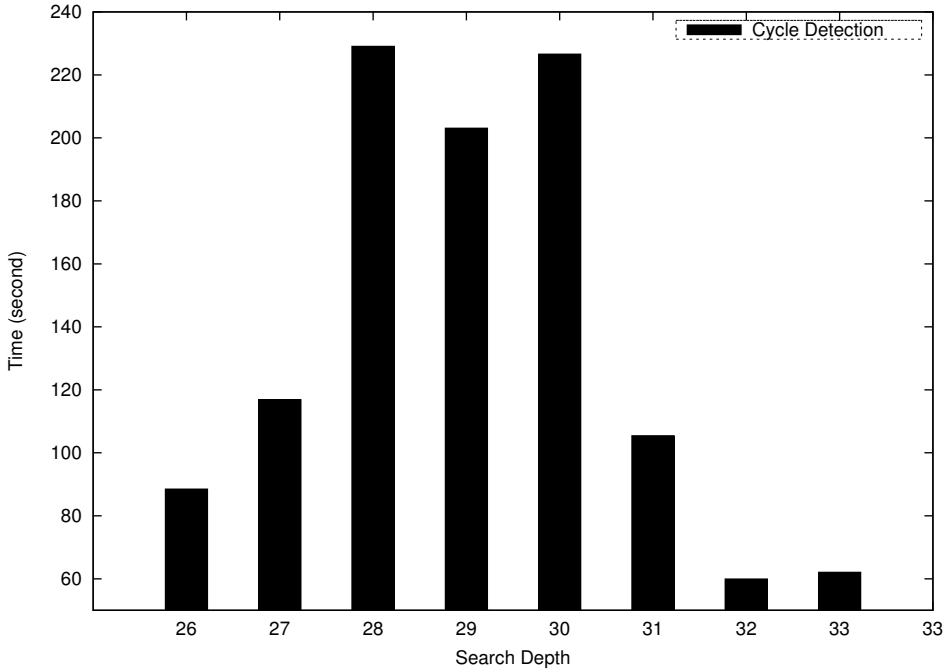

Figure 6.9 – NFG players on the game "mcheva"

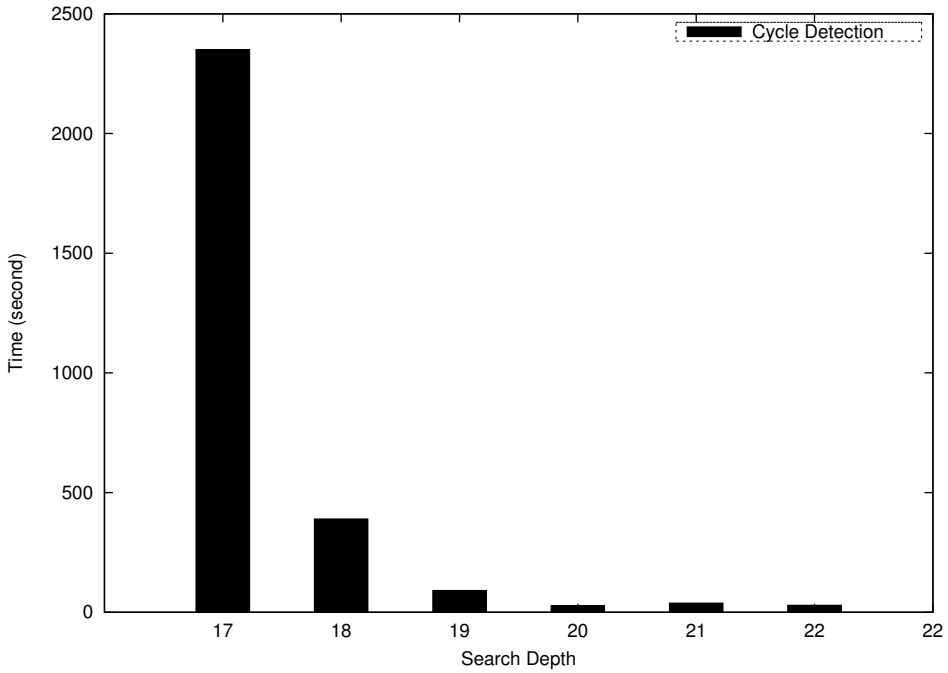

Figure 6.10 – NFG players on the game "sdesja8"

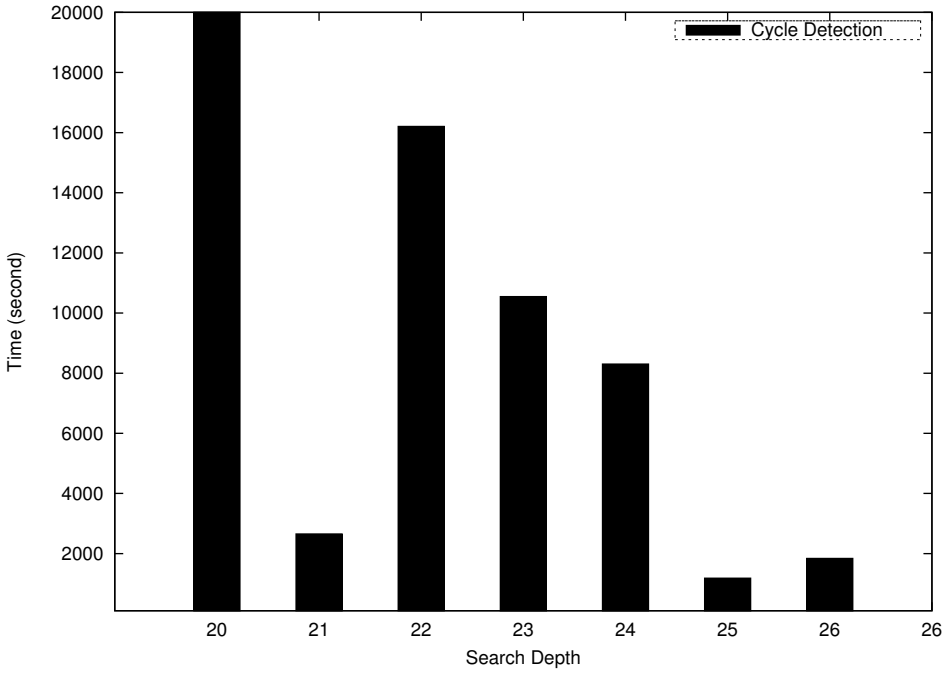

Figure 6.11 – NFG players on the game "RTZ task02 (full)". The no-cache player is not shown here since it has the same performance as the "cycle-detection" player

The performance of the players on large-sized benchmarks (more than 15 steps to win) is shown in Figure 6.9, Figure 6.10 and Figure 6.11 for "mcheva", "sdesja8" and "RTZ task02 (full)" respectively. At this size a bad state cache is essential and the "cycle detection" player with cache enabled is the only version that can solve the first two games. Interestingly, the third benchmark does not reach bad states, and so the no-cache player has identical performance to the version with cache enabled. Solutions for these games take much longer, as much as five hours to finish the "RTZ task02 (full)" at the minimal required search depth. However, this is still orders of magnitude improvement over earlier approaches [25, 32]. Moreover, its performance is still acceptable on the rest two games.

To find out the efficiency of the cache, we calculated its hit rate. One interesting thing found is that the cache only works on the two large narratives; on small and medium benchmarks, whose winning paths are able to be found by a quick search, the cache seldom takes effect, and actually, the size of the cache is 0 for all the other narratives in our benchmark set. On the contrary, some large narratives tend to create many bad states, and large hit rates

as well, which costs a great amount of time to traverse without the cache, preventing the player from finding the correct path in any reasonable search time. The other notable fact is that the cache size is also always 0 for "RTZ task 02 (full)" though it is a large-sized game. This is because the game is better designed by developers of the framework rather than the other two submitted as homework, it contains no real dead ends reachable within the maximum search depth tested.

The behavior of the analysis at different maximum search depths is also interesting. "sdesja8" and "hsafad" show that, counter-intuitively, performance of the analysis *improves* as the depth increase, despite the theoretical growth in search space. We discuss this in further detail at the end of this chapter.

Due to the performance of the variant PNFG players, we find the "cycle detection" player with "bad state cache" is the fastest and most generally useful version for most of our benchmarks; although "cycle detection" and "bad state cache" are both expensive operations in some cases, they are typically very worthwhile. Further experimentation below is base on this relatively better optimized player.

6.3 Results of Optimizations

Experimental data from optimizations applied to all of our benchmarks are listed from Figure 6.12 to Figure 6.21. To compare the performance and improvements, we choose the "cycle detection" NFG player with "bad state cache" as a baseline version, and refer to it as the "standard" in those figures.

6.3.1 Accurate match

Recall that the accurate match optimization reduces the search space by reducing the number of actions that can follow another action. Unfortunately, the solving time of accurate match is almost the same as, or even higher than the standard version in most cases except "mcheva" (Figure 6.19) and "RTZ task02 (full)" (Figure 6.21). Table 6.4 is the numerical statistics of a typical winning path search with accurate match on some games in our benchmark set. In the table, the "Choices Made" is the number of actions tried during

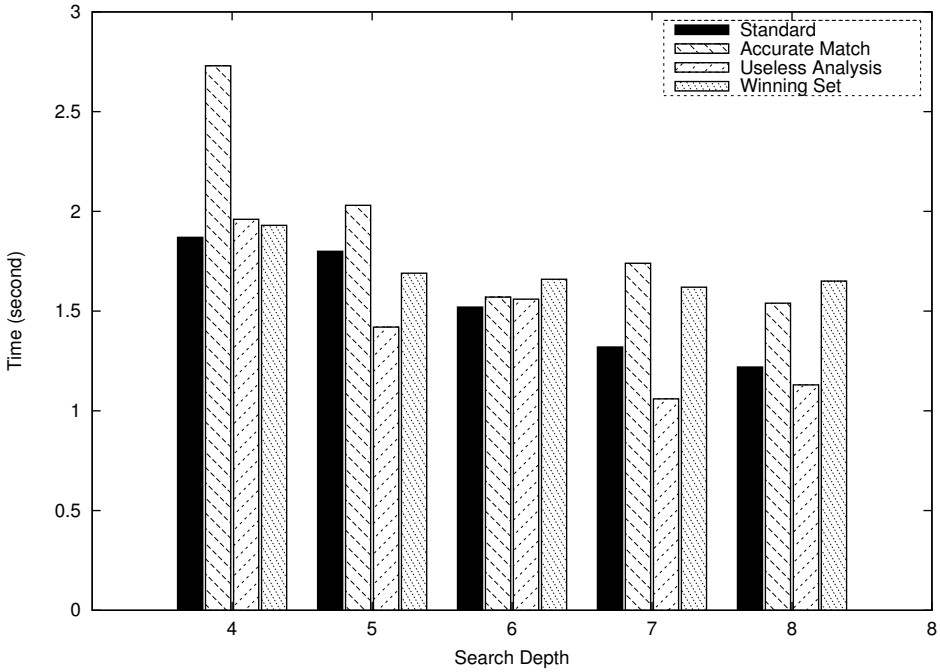

Figure 6.12 – NFG players with optimizations on the game "dpomer"

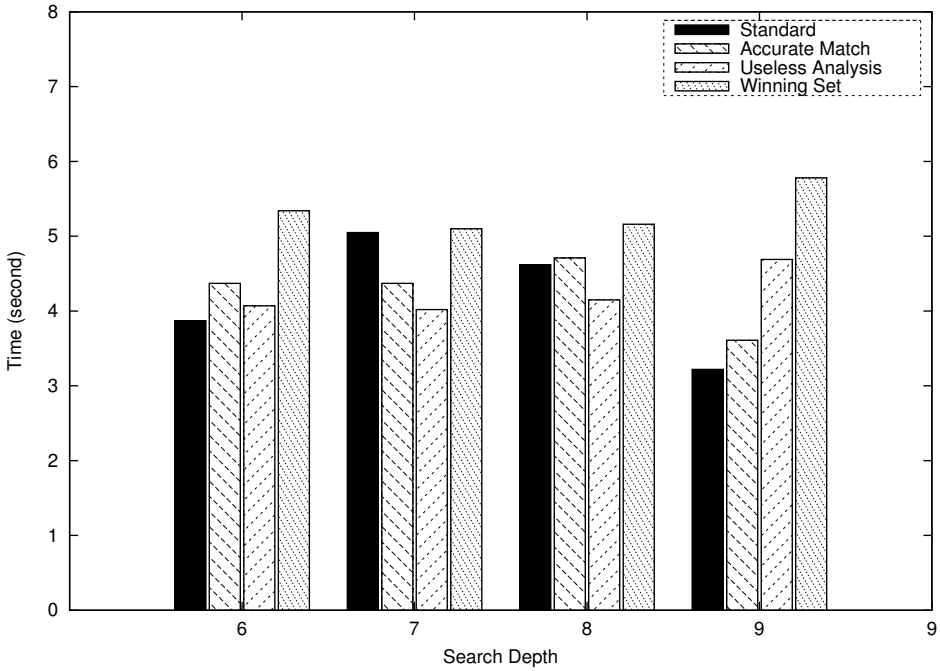

Figure 6.13 – NFG players with optimizations on the game "RTZ task 01 (full)"

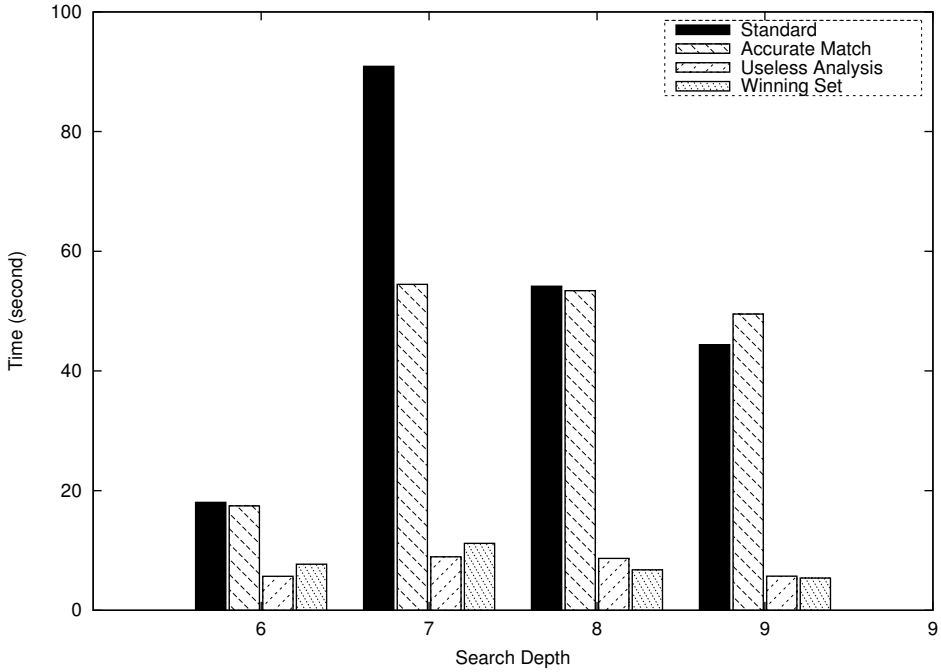

Figure 6.14 – NFG players with optimizations on the game "cod (full)"

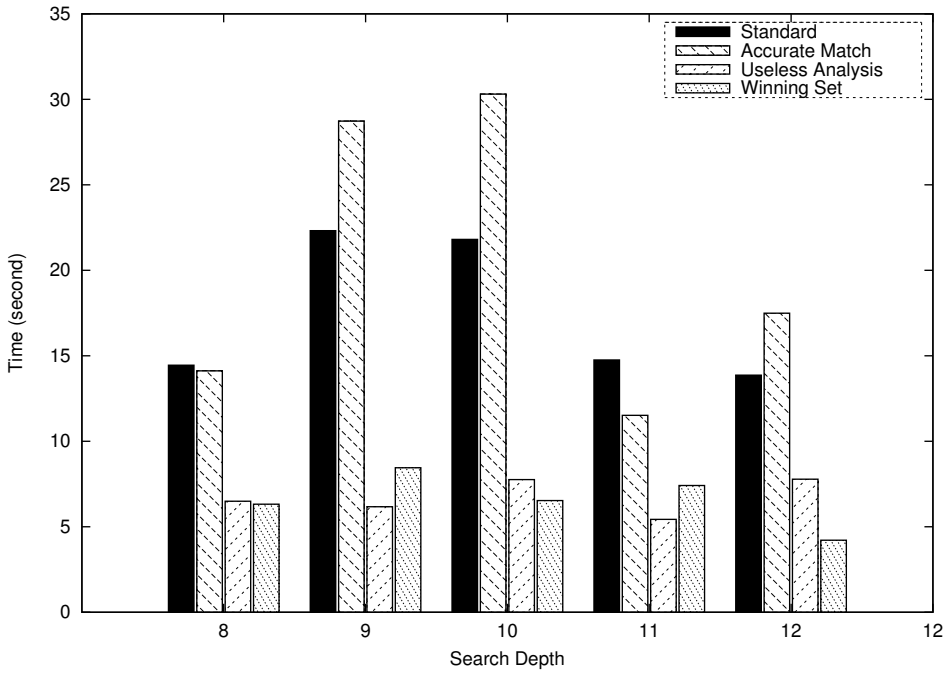

Figure 6.15 – NFG players with optimizations on the game "csimon16"

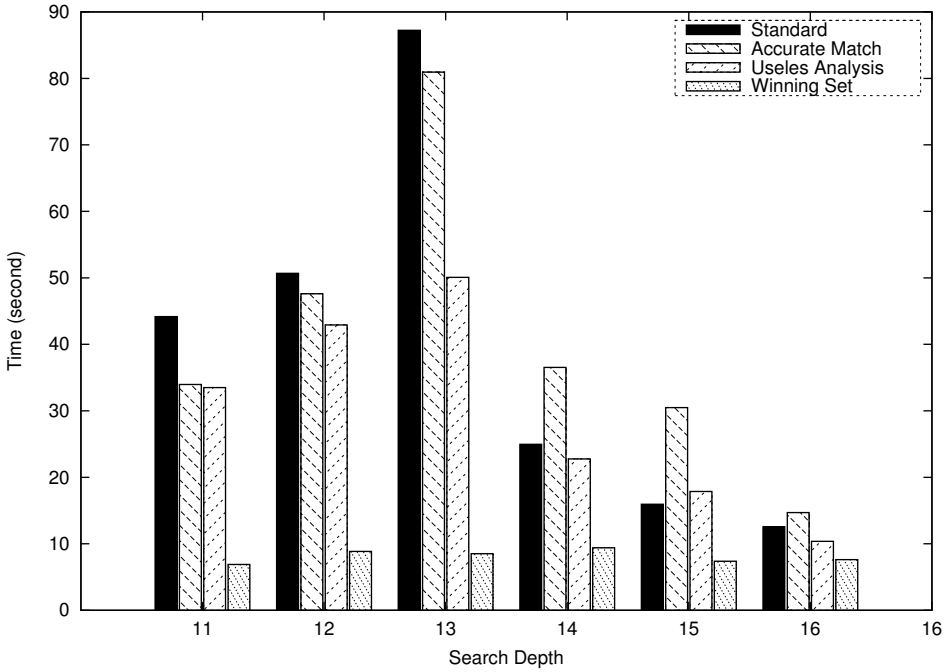

Figure 6.16 – NFG players with optimizations on the game "RTZ task 01 (wbp)"

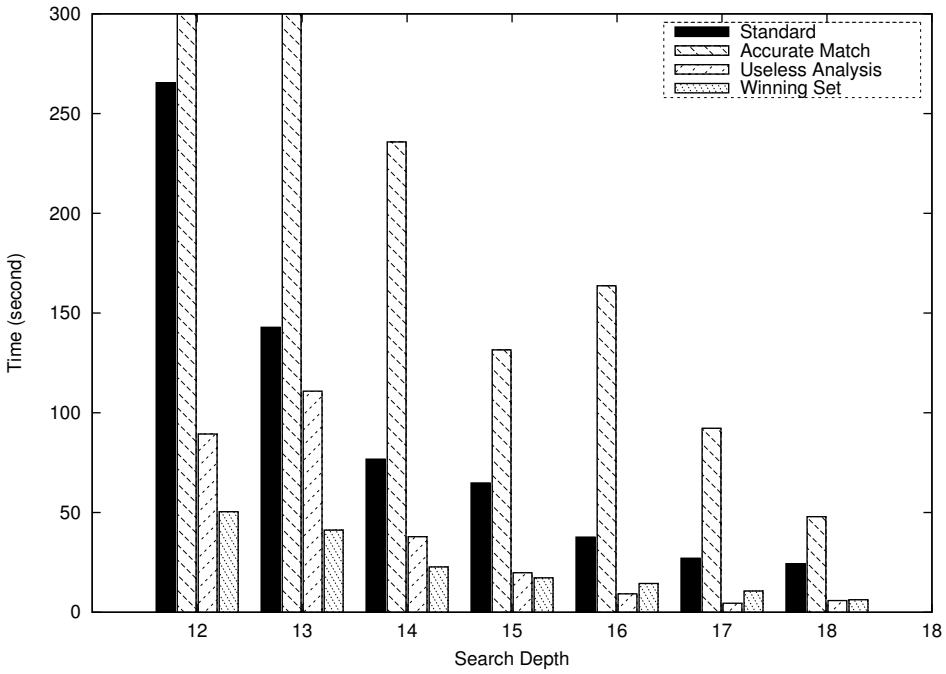

Figure 6.17 – NFG players with optimizations on the game "hsafad"

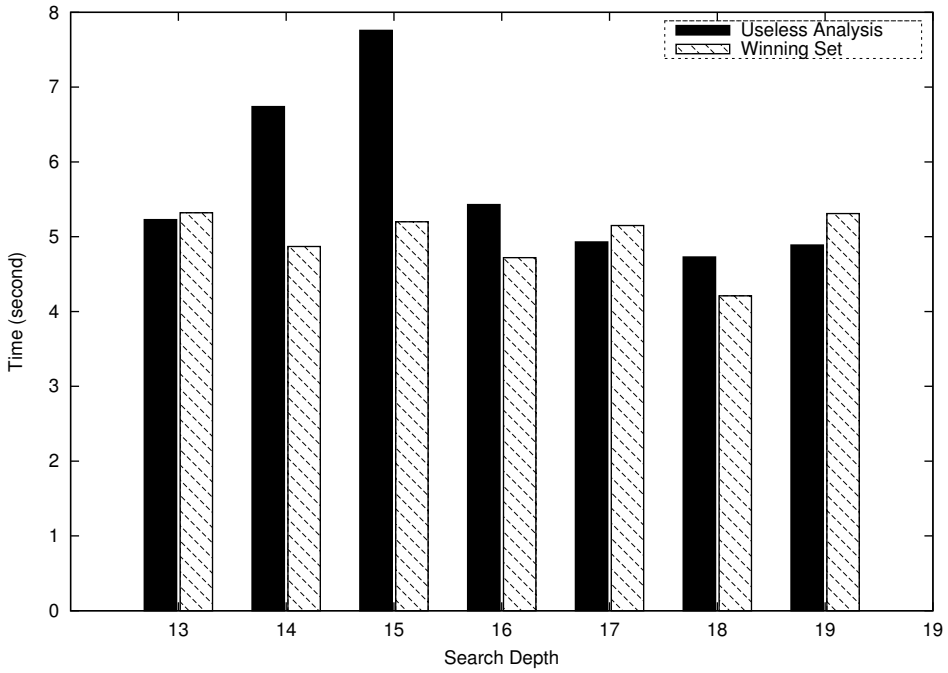

Figure 6.18 – NFG players with optimizations on the game "dprykh". The "Standard" performance is over 100 seconds (shown in Figure 6.8) and thus is omitted, the "Accurate Match" is also omitted because it does not support this game.

	cod	RTZ-task01	RTZ-task01	mcheva	RTZ-task02
	full	full	wbp		full
Choices Made	3860	55	90	25199	1424
Total Candidates	5873	106	204	46612	2966
Filtered Candidates	454	17	33	7168	576
	7.73%	16.04%	16.18%	15.38%	19.42%

Table 6.4 – The efficiency of the accurate match for a typical run on selected benchmarks

the search; the "Total Candidates" is the total size of command lists calculated for each choice; and the "Filtered Candidates" is the total number of actions filtered by the accurate match analysis. On average, less than 16% over all candidates are successfully filtered. According to the test results, this analysis is not as efficient as we expected, and this has two main causes. First of all, in many cases, although the game state does not have conflict with the explicit or implicit constraints of an action, it may not match other requirements of the actions which our analysis cannot detect. More seriously, such mismatches may lead

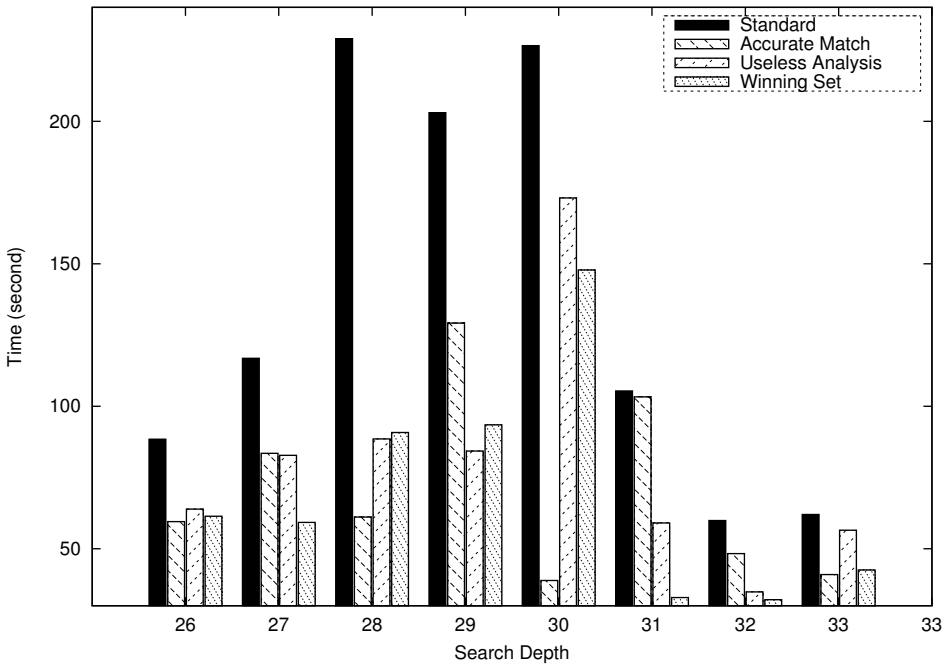

Figure 6.19 – NFG players with optimizations on the game "mcheva"

to rolling back of game state, a more expensive operation than repicking another command from the list. Thus although our analysis saves time on repicking, it cannot compensate for the time wasted on rolling back game state, making the analysis less effective.

6.3.2 Useless object/action

Although the ratio of useless objects/actions varies greatly among games, on average, we found about one quarter useless actions, and one third useless objects for games in our benchmark set, and a half of object state variables are useless as well. All these factors together reduce the size of ADG by more than a half accordingly as shown in Table 6.5. The performance of this optimization is closely related to the properties of games as well. For the game "cod (full)" which has a large reduction on the size of ADG (over 70%), Figure 6.14 shows a great decrease in searching; however, in Figure 6.16, for the game "RTZ task 01 (wbp)" which has about half of size reduction on the ADG, this optimization makes barely any difference from the standard version with few exceptions. The same pattern also

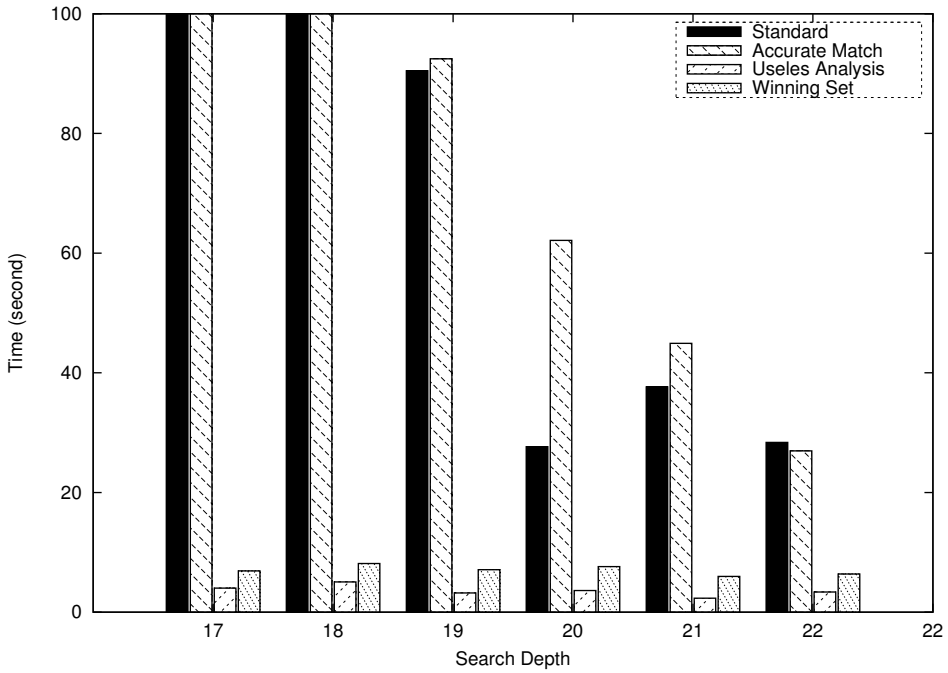

Figure 6.20 – NFG players with optimizations on the game "sdesja8", bars beyond the edge are over 400 for the "Accurate Match" and 2000 for the standard version

	cod	RTZ-task01	RTZ-task01	RTZ-task02
	full	full	wbp	full
Useless Actions	18	20	20	63
	30.00%	22.73%	21.73%	23.38%
Useless Objects	0	10	10	23
	0.00%	34.45%	34.45%	40.03%
Useless States	5	5	4	8
	83.33%	62.50%	50.00%	57.14%
Nodes in Reduced ADG	21	32	34	68
	56.76%	72.73%	73.91%	58.12%
Edges in Reduced ADG	322	280	313	913
	29.46%	47.06%	48.60%	42.37%

Table 6.5 – The number of useless objects/action, its overall proportion and the related size of the reduced ADG

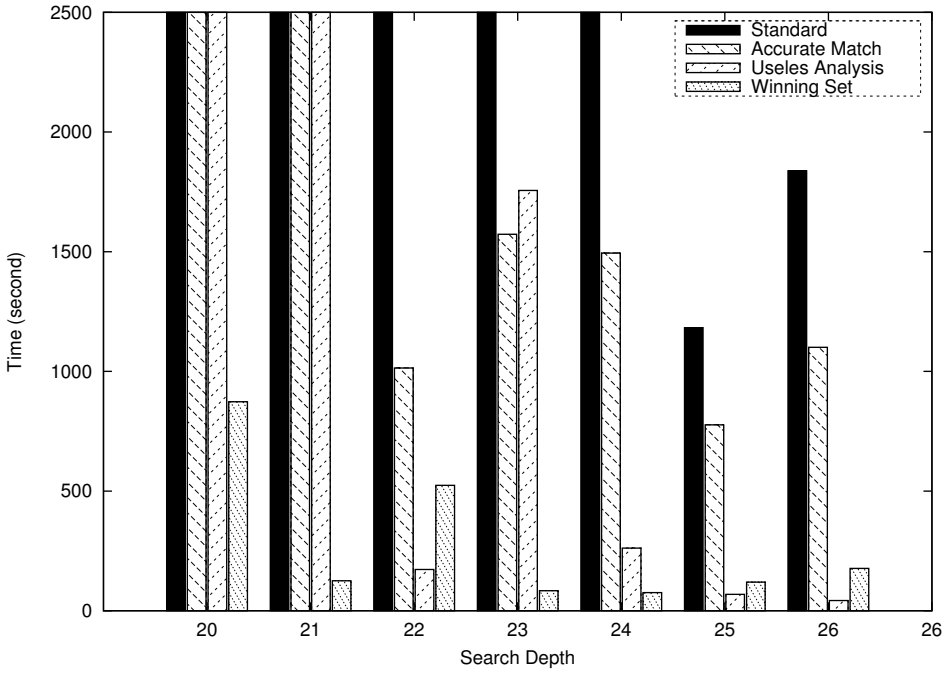

Figure 6.21 – NFG players with optimizations on the game "RTZ task02 (full)", bars beyond the top edge are over 8000 seconds

occurs on other benchmarks which are not shown in Table 6.5. It is possible that high-level game properties like convexity influence the number of useless objects/actions. Further investigation of this connection is left for future work.

6.3.3 Winning set

	cod	RTZ-task01	RTZ-task01	RTZ-task01	RTZ-task02
	full		full	wbp	full
Steps to win	6	5	6	11	20
Actions	37	55	88	92	222
Winning set	21	18	20	25	52
	56.77%	37.73%	22.73%	27.17%	23.42%

Table 6.6 – The size of the winning set, and how much the winning set covers the set of all actions

In general the winning set analysis effectively diminishes the number of potential ac-

tions; Table 6.6 shows over 75% of actions may not be required in the selected benchmarks. This tremendously narrows the problem space reducing the O(n^2) number of action pairings in the worst case to that of a comparable narrative only one quarter the size. Moreover, this optimization is more effective when applied to larger, more complex narratives. The main reason is that larger games tend to include more optional quests and more auxiliary actions to better reflect the real world in the game (as mentioned in Section 2.2), and thus most of these actions are excluded from the winning set. Though not analyzed directly, a relatively larger winning set implies a denser ADG, where actions have more than the necessary relationships between each other, and so will generate a larger, less precise winning set than a sparse ADG. Experiments on solving our benchmarks provide strong evidence of the value of this technique; this optimization has excellent effect on almost all of our games. For instance, Figure 6.14 and Figure 6.21 demonstrate on astonishing performance improvement on narratives with both large (imprecise) and small (precise) winning sets. The solving time is steadily around 10 – 50 seconds throughout both narratives and search depths, dramatically less than other approaches. In our future work, further analysis will be on the relationship between the size of the winning set, the ADG, and the convexity of the narrative.

6.3.4 Comparisons and final results

From the figures above, the "accurate match" is very close to the standard version and thus has the least overall improvement. Our "useless analysis" is in fact often useful, but it depends heavily on the properties of narratives, functioning well only if the calculated useless object/actions reach a certain ratio overall. The most impressive improvements are brought by the "winning set" approach, which seems to be uniformly extremely effective, almost regardless of the size of narratives. Although narratives may create big, fuzzy winning sets, the NFG player is often able to make good use of that information to choose actions effectively.

Other than performance improvements there is also an interesting similarity among all the search approaches, regardless of the variant of NFG players and the optimizations applied: for most of our moderate and large sized benchmarks, the solving time *decreases*

with a *growing* search depth. Some obvious examples are Figure 6.7, Figure 6.10, Figure 6.16, and Figure 6.21. At the minimum solution depth a search can turn quite long, but then with larger depths the NFG player becomes faster. Various factors contribute to this phenomenon. First, as described in Section 6.2, the games are designed to win, not to lose. Actually, in most cases games can be finished by trying everything that generates non-trivial results, such as wandering among rooms, changing the state of items, *etc.*; players rarely lose games unless intending to. Second, a depth-first search, as the NFG player uses, has rather obvious benefits in this respect. A deeper search depth is more likely to encounter a winning state if game actions tend to monotonically move the game toward a win. Finally, it is true that a search with a smaller maximum depth better approximates a search for an optimal solution; a deeper, but likely sub-optimal (longer) solution demonstrates the trade-off between optimality and time. Taking all these factors together, the "winning set" design, coupled with useless object/action analysis best satisfies our goal of improving the efficiency of narrative analysis.

Chapter 7
Conclusions

This work represents a detailed description of the dataflow analysis on PNFG framework. Many modern games, especially role playing games, and adventure game tend to have more and more complex narratives, and meanwhile testing these narratives becomes an urgent issue in game development. Although lots of frameworks are designed to formalize the creation of games, very limited studies and tools are available for verification. The PNFG is a good framework to represent narratives, with previous limited success in low-level game verification. Our study is a formal strategy to analyze the high-level PNFG code. We have created a generic dataflow module for this framework, and implemented several analyzers to flow through the narrative structure to gather information. Then with the help of such dataflow analyses, our study is able to find the winning path of narratives, as a demonstration of narrative verification. Finally, we have also applied our design to a non-trivial benchmark set and investigated performance of our technique.

The original PNFG framework lacks support of formal dataflow analysis, which is vital to our study. Our first milestone is to build a new module to offer the ability of dataflow analysis. We have designed the CFG to represent the structure of PNFG code, and implemented optimizations such as loop unrolling and conditional expansion in the CFG according to the features of the original language. Then we have defined two domains, the location domain and state domain for the flow set, both used as the data structure for dataflow analysis. Besides the flow set, several analyzers, in both the forward and backward directions, are implemented as well. Moreover, as a generic dataflow module, both the flow set and an-

alyzers are implemented as skeletons so that they can be instantiated with specific analysis rules easily, and thus extended for other purposes.

To find the winning path we start by profiling the actions using our dataflow analysis module. The profile contains the prerequisites to successfully trigger the action, calculated by a backward analysis, and the consequences after the execution, generated by a forward analysis. Although we use a conservative approach for correctness, they provide useful information to identify the actions, and introduce the possibility of discovering relationships among actions. The ADG represents this information, and is built by analyzing the profiles of actions. The ADG uncovers the connections between high-level actions, providing an ideal, pruned search space for the NFG player, our depth-first search engine on the ADG. Several variants of NFG players are considered as well, designed to either prevent noneffective moves, action cycles, or to cache bad game states. In addition to these variants, we have implemented three optimizations to improve the performance of searching; among them, the "accurate match" computes the implicit constraints of actions with a simple overall conditional structure; the "useless actions/object analysis" tries to find actions and objects that do not affect the game state except trivially, and the "winning set analysis" is a backward, inter-procedural analysis that calculates the subset of actions necessary to reach the winning state from the initialization point.

Our work is well-tested and evaluated. We have set up a benchmark set and run all the algorithms and analyses on them. From our results, with the ADG, the NFG players are able to find the winning path for all narratives we have, especially with the help of optimizations that monitor and eliminate action cycles. Although the "accurate match" optimization does not improve a lot, the "useless analysis" has a big impact on large but complex narratives, and the "winning set" tremendously reduces the search space for almost all kinds of narratives, almost always emitting results in the shortest time.

Our current study shows the feasibility and effectiveness of dataflow analysis applied to narrative verification. It also opens doors to lots of further work in this area. One interesting task is to better support the analysis of PNFG counters/timers which currently are treated as nothing but normal states. Many dataflow techniques from the program optimization world like numerical analysis and constant propagation are likely to be helpful with getting more accurate information.

As for the NFG player, the "accurate match" and "useless analysis" can both be improved. Due to the time budget, they are simplified in our work, and actually a more extensive implementation may further improve the performance. For example, in the "accurate match", only the actions with *one* big "if" statement are examined; actions with chains of "if" statements should also be taken into consideration to get more accurate constraints. In the "useless analysis", it is easy for useless objects and actions to be interwined, and improvements may be possible by greater focus on cyclic references.

Other than future work in design and implementation, another interesting direction is from the perspective of evaluation and results analysis. The PNFG framework has featured a measurement of several game metrics such as the number of actions, basic convexity, and so on. Considering the performance of our algorithms on different benchmarks, we believe that such properties of narratives affects their analysis efficiency, and furthermore, the difficulty in finding winning paths. For instance, the impact of "useless action/object analysis" is closely related to the number of redundant stuff in the game, which is also an important factor in the calculation of convexity; the size of the "winning set" is also an important property to convexity. By analyzing these relationships, we may expand the analysis of narratives to include useful quantitative measures for complexity and quality of narratives.

Bibliography

[1] Inform 7. http://www.inform-fiction.org/.

[2] The 14th annual interactive fiction competition. http://www.ifcomp.org/, 2008.

[3] Ernest Adams. The designer's notebook: Bad game designer, no twinkie! parts I–VIII. http://www.designersnotebook.com/Design_Resources/No_Twinkie_Database/no_twinkie_database.htm, 1998–2007.

[4] Sheldon B Akers. Binary decision diagrams. In *IEEE Transactions on Computers*, volume C-27, pages 509–516. 1978.

[5] Leandro Motta Barros and Soraia Raupp Musse. Introducing narrative principles into planning-based interactive storytelling. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 35–42, New York, NY, USA, 2005. ACM.

[6] Cyril Brom and Adam Abonyi. Petri nets for game plot. In *Adaption in Artificial and Biological Systems*, volume 3, 2006.

[7] Cyril Brom, Vít Sisler, and Tomás Holan. Story manager in 'Europe 2045' uses Petri nets. In *International Conference on Virtual Storytelling*, pages 38–50, 2007.

[8] Jennifer Burg and Sheau-Dong Lang. Using constraint logic programming to analyze the chronology in "a rose for Emily". *Computers and the Humanities*, 34:377–392, 2000.

[9] Marc Cavazza, Fred Charles, and Steven J. Mead. Interacting with virtual characters in interactive storytelling. In *AAMAS '02: Proceedings of the first international joint*

conference on Autonomous agents and multiagent systems, pages 318–325, New York, NY, USA, 2002. ACM.

[10] Marc Cavazza, Fred Charles, and Steven J. Mead. Interactive storytelling: from AI experiment to new media. In Donald Marinelli, editor, *International Conference on Entertainment Computing*, pages 1–8. Carnegie Mellon University, 2003.

[11] Yun-Gyung Cheong and R. Michael Young. A computational model of narrative generation for suspense. Technical Report WS-06-04, Association for the Advancement of Artificial Intelligence (AAAI), Boston, MA, USA, 2006. Computational Aesthetics: Artificial Intelligence Approaches to Beauty and Happiness.

[12] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 359–364, Copenhagen, Denmark, July 2002. Springer-Verlag.

[13] Frédéric Collé, Ronan Champagnat, and Armelle Prigent. Scenario analysis based on linear logic. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, page 1, New York, NY, USA, 2005. ACM.

[14] Valve Corporation. Half-life 2, 2004.

[15] Chris Crawford. *Chris Crawford on Interactive Storytelling*. New Riders Games, 2004.

[16] Blizzard Entertainment. Diablo II, 2000.

[17] Blizzard Entertainment. Warcraft3: The frozen throne, 2003.

[18] Pablo Gervás, Birte Lönneker-Rodman, Jan Christoph Meister, and Federico Peinado. Narrative models: Narratology meets artificial intelligence. In Roberto Basili and Alessandro Lenci, editors, *International Conference on Language Resources and Evaluation. Satellite Workshop: Toward Computational Models of Literary Analysis*, pages 44–51, Genova, Italy, 2006.

[19] Infocom. The Hitchhiker's Guide to the Galaxy, 1984.

[20] Infocom. Return to Zork, 1993.

[21] Antonis C. Kakas and Rob Miller. A simple declarative language for describing narratives with actions. *Journal of Logic Programming*, 31(1–3):157–200, 1997.

[22] Craig A. Lindley and Mirjam Eladhari. Causal normalisaton: A methodology for coherent story logic design in computer role-playing games. In *Computers and Games - Third Iternational Conference*, number 2883 in Lecture Notes in Computing Science, pages 292–307. Springer, Alberta, Canada, 2002.

[23] Brian Magerko. Story representation and interactive drama. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 87–92, 2005.

[24] Wendy Ann Mansilla and Bernhard Jung. Emotion and Acousmêtre for Suspense in an Interactive Virtual Storytelling Environment. *Proceedings of The Third Annual International Conference in Computer Game Design and Technology*, pages 151–156, 2005.

[25] Félix Martineau. PNFG: a framework for computer game narrative analysis. Master's thesis, McGill University, Montréal, Canada, jun 2006.

[26] Ben Medler and Brian Magerko. Scribe: A tool for authoring event driven interactive drama. In *Technologies for Interactive Digital Storytelling and Entertainment*, pages 139–150, 2006.

[27] Nick Montfort. *Twisty Little Passages: An Approach to Interactive Fiction*. The MIT Press, 2003.

[28] Nick Montfort. *IF Theory*, chapter Toward a Theory of Interactive Fiction. David Cornelson, 2004.

[29] Pablo Moreno-Ger, Iván Martínez-Ortiz, José Luis Sierra, and Baltasar Fernández-Manjón. Language-driven development of videogames: The e-game experience. In Richard Harper, Matthias Rauterberg, and Marco Combetto, editors, *ICEC*, volume 4161 of *Lecture Notes in Computer Science*, pages 153–164. Springer, 2006.

[30] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[31] Stéphane Natkin and Liliana Vega. A Petri net model for computer games analysis. *Int. J. Intell. Games & Simulation*, 3(1):37–44, 2004.

[32] Christopher J. F. Pickett, Clark Verbrugge, and Félix Martineau. (P)NFG: A language and runtime system for structured computer narratives. In *Proceedings of the 1st Annual North American Game-On Conference (GameOn'NA 2005)*, pages 23–32, Montréal, Canada, August 2005. Eurosis.

[33] Martin K. Purvis. Narrative structures for multi-agent interaction. In *2004 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2004)*, pages 232–238, Beijing, China, September 2004. IEEE Computer Society.

[34] Steve Rabin. *Introduction to Game Development*. Game Development Series. Charles River Media, 2005.

[35] Ray Reiter. Narratives as programs. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 99–108, San Francisco, 2000. Morgan Kaufmann.

[36] Michael J. Roberts. TADS: The Text Adventure Development System. `http://tads.org`, 1987–2005.

[37] Phoebe Sengers and Michael Mateas. Introduction to NI symposium. In *Narrative Intelligence Symposium, AAAI 1999 Fall Symposium Series*, pages 1–10, Cape Cod, North Falmouth, Massachusetts, 1999. Georgia Institute of Technology.

[38] Penelope Sweetser and Peta Wyeth. Gameflow: a model for evaluating player enjoyment in games. *Computers in Entertainment (CIE)*, 3(3):3–3, 2005.

[39] Nicolas Szilas and Jean-Hugues Rety. Minimal structures for stories. In *SRMC '04: Proceedings of the 1st ACM workshop on Story representation, mechanism and context*, pages 25–32, New York, NY, USA, 2004. ACM.

[40] THQ. Red Faction, 2001.

[41] Clark Verbrugge. A structure for modern computer narratives. In *Computers and Games: Third International Conference, CG 2002*, number 2883 in LNCS, pages 308–325. Springer-Verlag, 2003.

[42] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[43] R. Michael Young. Creating interactive narrative structures: The potential for AI approaches. Technical Report SS-00-02, Association for the Advancement of Artificial Intelligence (AAAI), 2000. Papers from the AAAI Spring Symposium.