Utrecht University

Formalizing Quest Systems For Analysis Of Playability And Diversity

Master's Thesis

The Netherlands, 2012

Author:
Bert Leemput
bert.leemput@gmail.com
ICA-3460738
Game and Media Technology

Supervisor:
Drs. Arno Kamphuis

Contents

Introduction

Current day computer games get increasingly more complex. This not only places a burden on game developers for creating more compelling graphics, but also on game designers who need to create more interesting content. As the amount of content increases, it becomes harder to maintain a certain level of quality. This is especially true for quest-based games, which contain massive amounts of content in the form of quests. Examples of such games are World of Warcraft and Guild Wars.

One facet that affects the quality of a game is playability. Certain choices, made by the game designer, can result in a player not being able to complete the game due to conflicting dependencies in the quest system. Such problems get increasingly more difficult to detect as the quest system grows larger. For example, the game World of Warcraft currently contains around 9834 quests[1]. For the game designer it is important to be able to automatically detect conflicting dependencies.

Another important facet of current day games is diversity. Players want to decide for themselves how they spend their time in a game. In most quest-based games the freedom of the player is limited to choosing whether or not he wants to accept a quest and in which order he wants to complete the desired quests. Therefore, it is important for the game designer to be able to analyse these types of player freedom.

In addition, different kinds of play style can affect the experience of the game. While some players rush through the content to complete the game as fast as possible, other gamers will take their time to complete every possible quest in the game. Due to the different play styles problems can occur. In the case of the rushing player it might occur that he doesn't encounter all parts of main story. This would result in a gap in the story which is undesirable.

In the first part of the paper we will discuss several techniques which will help game designers identify common problems that occur when working on a quest system. One example of such a problem is the occurrence of conflicting dependencies between quests. In such a situation the player will get stuck in the game because the dependencies for one or more quests can't be satisfied due to a construction error in the quest system. Another common problem that might occur, when working on a quest system that undergoes heavy and frequent changes, is the presence of redundant dependencies between quests. This will result in a needlessly complex quest system that will be harder to analyse by a game designer. Besides only detecting problems we also discuss algorithms that can help the game designer analyse the diversity of the quest system. One example of how the diversity of a quest system can be expressed, is by looking at the number of mandatory quests in the quest system. If a high percentage of quests is mandatory the player will not have much of a choice on deciding if he wants to accept a quest. The most likely result of skipping a quest is the problem that the player will get stuck.

The quest system used in the first part of the paper is very static. As a result every quest can be completed at some point in time. Therefore, if we need to reach a goal, we can only decide for each quest if it needs to be accepted or not. This changes in the second part of the paper, were we add choices to the quest system. This increases the diversity of the quest system by letting the player make choices that influence which quests are available to him. Therefore, the quest system becomes dynamic, which means that not all quests can be accepted/completed at some point in time. As a result the focus changes from "deciding if a quest should or shouldn't be completed as part of the goal" to finding the set of choices that lead to the desired goal.

[1] This number was found on the site www.wowhead.com which contains a database of the quests in World of Warcraft.

Goals

In the introduction we stated that quest based games have become increasingly more complex over the years. However, the methods of assessing the quality of the quest based content haven't evolved as much. Currently most game studios use testers in order to assess problems with a quest system. This is a very resource intensive way of testing the actual game. Furthermore, it does not guarantee that all scenario's have been tested. A better way to approach this problem is by automating the verification process of a quest system.

Furthermore, as a result of not having any tools to automatically test quest systems, we are also fairly certain that no tools exist to assess the quality of the quest system. In our opinion it could be interesting to provide game designers with tools that allow them to assess the quality of the content they create. One of the most interesting facets that can be assessed is the amount of freedom a player has in the game.

This leads us to the following formulation of our goals for this masters thesis:

1. Analysing quest systems for playability and diversity
2. Further formalization of quest graphs

The main goal of this masters thesis is the "Analysis of quest systems for playability and diversity". However, in order to be able to reach this goal we first need to have a formalization of a quest system. At the time of writing we haven't found a single formalization for quest systems. We therefore decided to add a sub-goal to our thesis, which is the "further formalization of quest graphs". In this case the "further"-part means that it is a continuation of the work described in the paper "A Framework for Formalizing Dynamic Quests" [vdW11].

In this paper van de Water defines a framework for formalizing dynamic quests. This includes a formalization of what a quest is. According to his findings a quest consists of three main parts: "prerequisites", "objectives" and "rewards". Furthermore, he explains that it is possible to make these parts dynamic, which means that they can change during the game. He continues by explaining that it is important to decide what the representation of a quest is when designing it. In his paper he states that there are two types of representations. The first type is the individual quest which is related to single player usage. The second type is the collective quest which is used for a group of players. The type of quest does not only influence the possibilities that a game designer has when it comes to making a quest dynamic. It also influences the possibilities of using a quest generator for generating dynamic content. Furthermore, he explains how his findings can be combined into a framework. This enables the possibility to check the viability of the quest system. Finally, he concludes his paper with a comparison between his framework and current state-of-the-art games. From the comparison he concludes that games can benefit from using a framework to define quest system.

Part I

Retrieving Information from a Quest Graph

Chapter 1

Introduction

In the global introduction we stated that current day computer games get increasingly more complex. As a result most games are provided with massive amounts of content in order to keep the players entertained. This is especially true for quest-based games which contain much content in the form of quests. Currently no tools, that we know of, are used to assess the quality and correctness of those quest systems. In this first part of the paper we are going to look at how we can validate the correctness of a quest system. Furthermore we will look at different metrics that can be used in order to assess the freedom a player has when interacting with the quest system.

Throughout the first part of the paper we will impose several limitations on the scope of our research. One of those limitations is the assumption that the order of accepting quests is the same order as completing quests. In reality these orders can be different. Furthermore, we assume that all quests on which a single quest depends need to be completed before that single quest can be completed. However, it is possible to create dependencies in a way that only a limited subset of dependencies needs to be completed before the latter single quest can be accepted. In the upcoming chapters we will explain the reasons for these and several less significant limitations in more detail.

Chapter 2

Definitions

In order to understand the upcoming chapters of this paper, it is important to have a basic understanding of the involved definitions. This chapter will explain the basic definitions. Throughout the paper new definitions will be introduced.

Quest
A quest is a task in a computer game which the player has to complete in order to make progress in the game. It is part of one and only one quest chain and a quest can have multiple prerequisites and objectives. In this paper we assume that we have explicit knowledge about to which quest chain a quest belongs. This means a quest can be a part of a quest chain, but it doesn't necessarily have to be.

Quest Chain
A quest chain is a chain of quests which are linked together by a game designer. In most games the game designer creates a quest chain in order to tell a small story in the game. However the game designer is free to link quests together based on a different criteria. In this paper we assume that a quest chain can have multiple quests to start the quest chain with. Also a quest chain can have multiple quests which serve as an end for the quest chain. The order of completing the quests defines which quest is the start and which is the end of the quest chain.

Quest Hub
A place in the game where the players can obtain multiple quests which are related by the story that is told in that quest hub. In most games a quest hub essentially is a town or outpost.

Prerequisite
A prerequisite is a requirement for a quest, which needs to be satisfied in order to be able to accept the quest. In this paper we distinguish between two types of prerequisites. The first type of prerequisites all have in common that they depend on another quest. Examples of this type of prerequisite are shown in Table 2.1.

Prerequisite	**Reason**
Quest Active Prerequisite	In order to accept a quest, another quest must be active.
Quest Completed Prerequisite	In order to accept a quest, another quest must be completed.
Item Prerequisite	In order to accept a quest an item is needed which is given as a reward for completing another quest.

Table 2.1: Quest related prerequisites

Please note that the quest active prerequisite is special because it allows players to accept a quest while the previous quest is not yet completed. When we look at the order in which we accept quests, we assume that the previous quest is in fact completed.

The second type of prerequisite is based upon a state of one or more entities. This means that such prerequisites can depend on anything ranging from the attributes of the current player to the game state. Examples of this type of prerequisite are shown in Table 2.2.

Prerequisite	**Reason**
Level	The player needs to be a certain level, in order for the player to accept the quest.
Reputation	A certain reputation with a faction is needed, in order to accept the quest.
Profession	A certain skill in a profession is needed, in order for the player to accept the quest.
Game State	The game needs to be in a certain state, in order for the player to accept the quest.

Table 2.2: Non-quest related prerequisites

A quest can have multiple prerequisites, so it is for example possible to have a quest with the prerequisites:

1. You must be level 5
2. Quest 1 must be completed.

In this paper we assume that when there are multiple prerequisites involved, they will be handled as an "and"-relationship. So you must meet both prerequisites in order to accept the quest. However, in games other relationships like the "or"-relationship are possible. The choice of only using "and"-relationship limits the freedom of the player, because now all prerequisites must be satisfied in order to accept a quest, while with an "or"-relationship this would not be necessary. We choose to limit our scope to only the "and"-relationship because it simplifies the analysis.

Objective
An objective is a part of a quest. It indicates what the player has to do in order to finish part of the quest. Some examples of objectives are shown in Table 2.3.

Objective	**Explanation**
Kill	The player must kill an amount of other players or NPC's in order to complete this objective.
Gather	The player must gather an amount of objects in order to complete this objective.
Escort	The player must escort an NPC in order to complete this objective.
Defend	The player must defend a certain area in order to complete this objective.
Locate	The player must locate an item, NPC or location in order to complete this objective.
Quest Completed	The player must complete a quest in order to complete this objective.

Table 2.3: Examples of objectives

Just like with the prerequisites, we assume that if a quest has multiple objectives they have an "and"-relationship, meaning that the player must complete all objectives. This results in a limitation on possible objectives for quests. With for example "or"- or "than"-relationships more versatile quests are possible. We choose not to investigate these relationships due to their more complex nature, but we believe that it is possible to integrate these relationships in the methods and algorithms proposed in this paper.

Chapter 3

Quest Dependency Graph

In the previous chapter we explained several game related definitions. In this chapter we are going to define what a Quest Graph is and how it can help game designers solve several problems related to quest systems. Because a quest system generally bares close resemblance to a dependency graph, we propose to create a dependency graph specialized for our needs, which we call a Quest Dependency Graph. Throughout the paper we will refer to the Quest Dependency Graph as the Quest Graph.

Figure 3.1 shows an example of a quest system. This system contains 21 quests which form five quest chains, each visualized with their own distinct color.

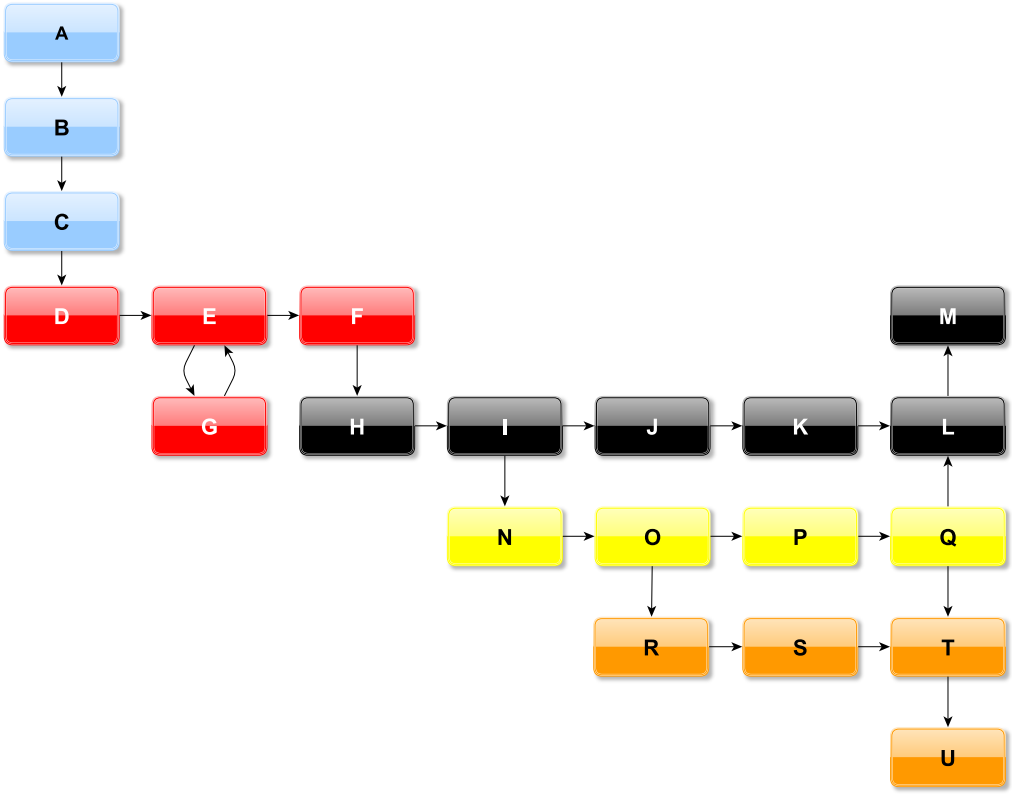

Figure 3.1: Example of a quest graph

3.1 The Elements of a Quest Graph

In this chapter we will explain the parts which form the quest graph and how they relate to each other.

3.1.1 Node

A graph consists of nodes so in this case it is only logical that each node represents a quest. However due to the different relationships between quests it is possible that a quest needs to be split into several parts. Furthermore, each quest chain can undergo a process that is called atomization which tries to make the structure of the quest chain more compact. The next chapters explain when and why the structure of the quest graph is altered. For now it is only important to understand that a node in the quest graph is not always a quest. As a result a node in the Quest Graph is defined as:

"A node in a quest graph can either represent an entire quest, part of a quest or a quest chain atom."

3.1.2 Edge

A dependency graph has edges which indicate the direction of the relationship between the nodes. This is also the case in the quest graph and we call such an edge an dependency link. The exact definition of a dependency link is explained in the section below.

Dependency Link

A dependency link is a representation of the prerequisites and objectives between two quests. Such a link is used to indicate in which order a player is allowed to accept quests. For example if quest B requires the player to gather 10 flowers which can only be obtained from quest A, then a link could be created between quests A and B. This would indicate that quest B has a dependency on quest A, meaning that a player first needs to complete quest A before he can start quest B.

Furthermore, a dependency link can either be "blocking" or "non-blocking". "Blocking" indicates that a link might possibly pose a barrier which the player cannot pass. Whereas the "non-blocking" link can always be passed. In the previous example the player obtains a certain amount of flowers from quest A. If that amount is lower than the amount needed in quest B then the player cannot ever finish quest B. This is an example of a blocking dependency link. If quest A would always provide enough flowers for quest B to be completed this link would be a non-blocking dependency link. So a dependency link is defined as:

"An dependency link represents one or more quest related dependencies. Each dependency is based upon a prerequisite or objective, and each dependency can be either blocking or non-blocking."

Table 3.1 explains which type of prerequisite results in which type of dependency link.

Please note that objectives are special because they can implicitly depend on one or more quests. In the case of the example above, the game designer has not explicitly mentioned where the player needs to obtain the flowers. However, in this paper we assume that this is an explicit dependency. This means that all quests providing an item that is needed in another quest will be directly linked to that quest. Just like with the prerequisites, this results in a limitation of the freedom of the player. All quests that provide the item need to be completed before the player can accept the quest which needs the items. However, in reality the player could obtain enough items by only completing a part of the quests. The reason why we directly link these quest is because we perform a

Prerequisite	**Dependency Link Type**
Quest Completed / Active Prerequisite	Non-blocking or blocking depending on how "Quest-Completed" is defined. It might be that in some game it is possible to complete a quest by only doing half of the objects.
Item Prerequisite	Non-blocking or blocking depending on the chance the player get the item that is needed. If a player always gets that item it's a non-blocking link.
Level	Neither because it is a state related prerequisite, which is not represented by a dependency link.
Reputation	Neither because it is a state related prerequisite, which is not represented by a dependency link.
Profession	Neither because it is a state related prerequisite, which is not represented by a dependency link.
Game State	Neither because it is a state related prerequisite, which is not represented by a dependency link.

Table 3.1: Prerequisites and their resulting dependency links

static analysis which means that we don't simulate the game. Therefore, we can only assure that the player has enough items if he completes all quests.

3.1.3 Connected Components

In a game it might appear that all quests depend on each other and therefore form one big quest graph. But after an investigation of current games we concluded that this is not the case. In addition, there are several reasons why it might not be a good idea to have all quests depend on each other at all:

1. It limits the freedom of a player
2. If one quest can't be completed, all quests that depend on it can also not be complete.

The most important reason for not letting all quests depend on each other is that it severely limits the freedom of the player (and game designer). If all quests are connected so that they form one quest graph, it would mean that a player needs to complete all quests in order to finish the game. This results in a game in which the player can't really choose which quests he wants to complete because if he skips one quest, he will be stuck. In most games the game designers give the player a sense of freedom by providing them with optional quest(chain)s. But because they are optional they must be disconnected from the other quest(chain)s. If they where not disconnected they would have dependency links, which indicate a dependency on the other quest chains and thus making them not optional.

The second reason is also important. If only one quest in the quest system can't be completed due to conflicting dependencies, all quests that depend on it can also not be completed. This effect will cascade through the entire graph and therefore will prevent the player from being able to play the game.

In order to prevent or minimize these problems we propose to use connected components. A connected component is a group of quests which all depend directly or indirectly on each other. Therefore, a connected component essentially is just a normal quest graph. Figure 3.2 shows an example of a quest graph with two connected components. In this simple example "Connected component 2" contains the optional quest. Note that it does not have any dependencies on the other component which contains the main quest system.

As a result of dividing the quest system into connected components, we enable the usage of optional quests in the quest graph. In addition, by dividing the quest graph into connected components we limit the cascading effect, which can occur if there are conflicting dependencies between quests, to only one connected component. While this does not entirely solve the problem, it at least provides the player with the freedom to continue playing the game by completing quests from another connected component.

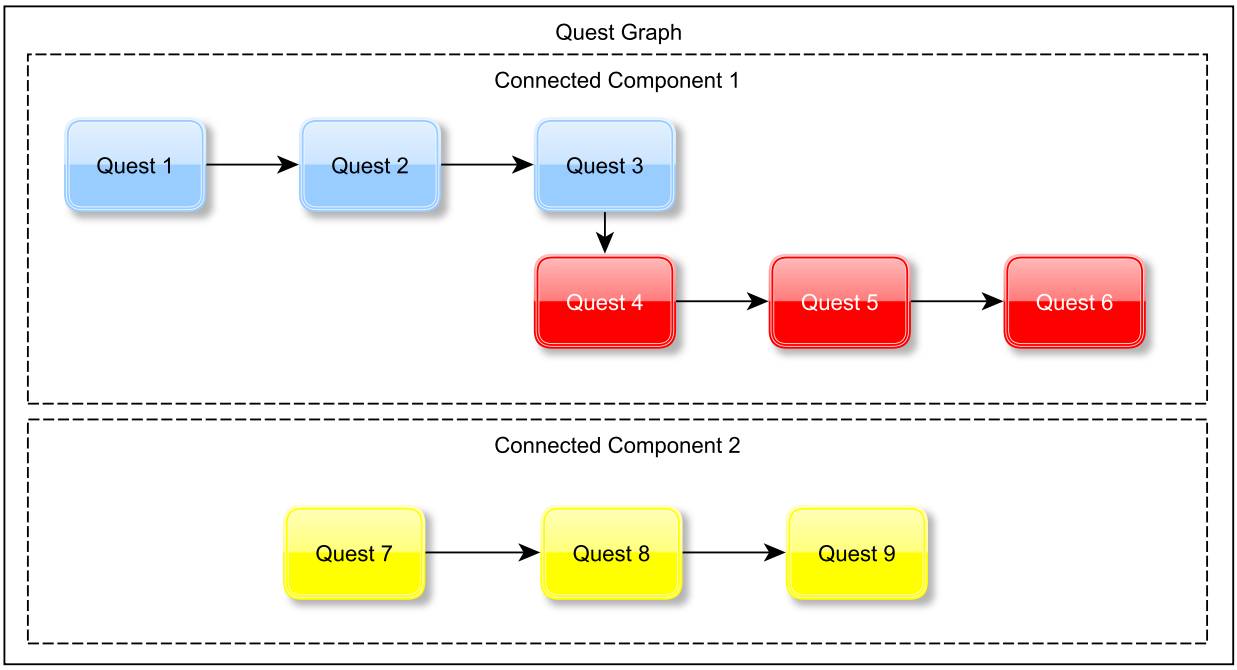

Figure 3.2: Example of a quest system with two connected components

3.2 Quest Relationships

In the previous section we explained what a quest graph is. In this section we will discuss which kind of relationships exist between quests and how they affect the quest graph. The relationships discussed in this chapter are taken from the paper "A Framework for Formalizing Dynamic Quests"[vdW11].

Reward as a Prerequisite
The first type of relationship we discuss is the "Reward as a Prerequisite"-relationship. In this relationship one quest (Quest B) depends on a reward that was given to the player upon completing another quest (Quest A). This results in a dependency link between the two quests. Like mentioned before, a dependency link can either be blocking or non-blocking. In this case it depends on the likelihood of the player obtaining the reward from Quest A. In a dynamic quest system it might occur that a player does a very bad job at completing the quest and therefore only gets a part of the reward. In which case it would result in a blocking dependency. Figure 3.3 shows a dependency from quest B on quest A. Please note that every quest that rewards the player with an item needed for another quest will be linked to the other quest.

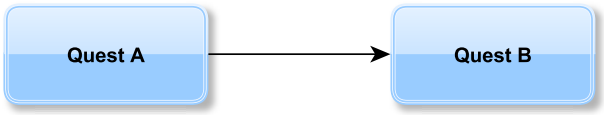

Figure 3.3: Example of a quest dependency. B depends on A.

Objective as a Prerequisite
The second type of relationship we discuss is the "Objective as a Prerequisite" in which one quest (Quest B) depends on the objective of another quest (Quest A). In the paper "A Framework for Formalizing Dynamic Quests"[vdW11] a good example of this type of relationship is given:

"A player needs to explore a forest for Quest A. Upon entering the forest the player hears the sound of a girl screaming, which triggers Quest B, which asks the player to investigate this situation."

In this example Quest B is triggered only when the player is completing Quest A. So if the player would walk into the forest without having Quest A active, it would not be triggered. Also notice that this type of relationship is combined with some kind of trigger. In this case it was the player who entered a specific area in the game that triggered the quest, but many other triggers are possible.

It is important to notice that this can result in either a blocking or a non-blocking dependency link, which depends on the chance that the quest is triggered. Just like the "Reward as a Prerequisite"-relationship this results in a dependency link from Quest A pointing to Quest B as shown in Figure 3.3.

Rewards as Objective
For the third type of relationship, one quest (Quest B) depends on the rewards given by another quest (Quest A). This relationship is almost the same as the "Reward as a Prerequisite", however in this case the reward is an implicit dependency. In the quest graph we treat both types as if they are the same.

This relationship between the two quests can result either a blocking or a non-blocking dependency link. Therefore this also results in a dependency link between Quest A and Quest B.

Please note that every quest, that rewards the player with the needed item, will be directly linked to Quest B. This means in this case that we act as if we can't accept Quest B until we have completed all the quests on which quest B depends. However, in reality we could in fact accept the quest without completing any of those quests. Unfortunately, we would not be able to complete the quest until enough quests, that provide the

needed item, have been completed. Therefore because the quest graph is a dependency graph we choose favor the dependency information over the minor inconsistency in the quest acceptance order.

Double-Linked Quests
The fourth type of relationship is basically a combination of the two previous types "Objective as a Prerequisite" and "Rewards as Objective". This results in a dependency link which can be blocking or non-blocking depending on the types links that form this relationships. Table 3.2 shows the combinations of both types and the resulting type of dependency link for this type.

Objective as a Prerequisite	**Rewards as Objective**	**Result: Double-Linked Quests**
Non-Blocking	Non-Blocking	Non-Blocking
Blocking	Non-Blocking	Blocking
Non-Blocking	Blocking	Blocking
Blocking	Blocking	Blocking

Table 3.2: Double-linked relationship combinations

Main / Sub Quests
The final type of relationship mentioned in the paper "A Framework for Formalizing Dynamic Quests"[vdW11], is the "main/sub quest"-relationship. In this situation one quest is the main quest and another quest is the sub-quest. In order to complete the main quest, the player first needs to complete the sub quest, which is part of the objectives of the main quest. This type of relationship is expressed by having a "Main Quest Active"-prerequisite in the sub quest and an "Sub-Quest Completed"-objective in the main quest. This results in the structure that is visualized in Figure 3.4.

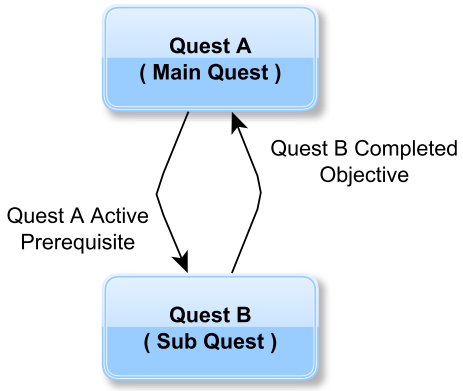

Figure 3.4: Example of a quest with a sub-quest

If we look at the structure of the "main/sub quest"-relationship, we notice that there is a conflict with the intended goals of the quest graph. One of the problems the quest graph solves, is the detection of conflicting dependencies between quests. These conflicting dependencies manifest themselves as a circular chain of quests. Each quest in that chain depends on the previous quest, resulting in a infinite cycle. If we look at the "main/sub quest"-relationship we notice that this type of relationship manifests itself as a cycle. However due to the nature of the relationship, it will never be a real cycle because we always meet the quest related prerequisites for Quest B.

To solve this problem we transform the structure of this relationship, so that we have exactly the same relationship but in a non-cyclic manner. We achieve this by splitting the main quest into two parts, a start and an end part. Figure 3.5 shows which parts of Quest A are included in the split version of the quest.

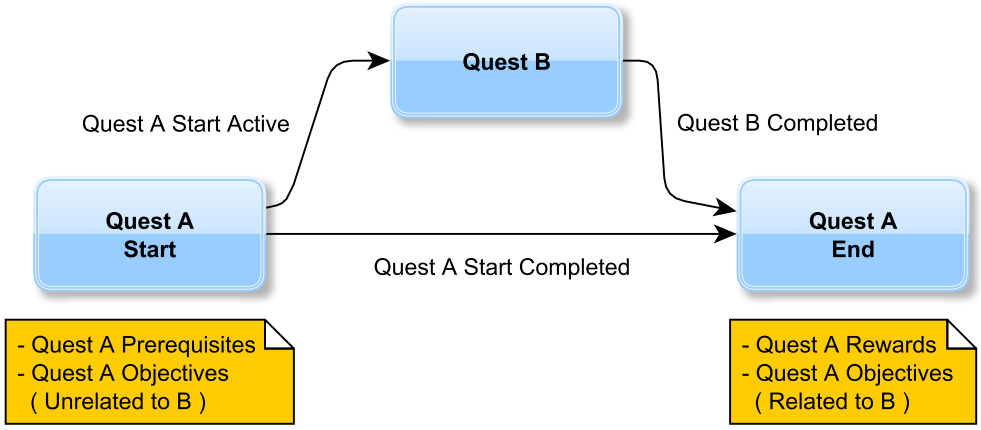

Figure 3.5: Transformed sub/main quest relationship

3.3 Quest Graph Information

From the quest graph we can obtain information which can help the game designer make decisions on improving the quest system. This chapter will explain which tools the game designer can use in order to retrieve the desired information. We will explicitly discuss methods and algorithms that are performed on quests, however in a quest graph a node can be a quest, part of a quest or a quest chain atom.

3.3.1 Detecting Redundant Links

Most quest systems contain a rather large amount of quests and therefore also contain a large amount of dependencies between the quests. While designing a quest system it might occur that one dependency is initially needed, but can be discarded at a later time. When this happens many times, the quest system can get cluttered with redundant dependencies. As the number of redundant dependencies increases it gets harder for a game designer to deduce what the intended flow of a part of the quest system is. This can get particularly bad if it looks like the player has much freedom, on deciding in which order he wants to complete quests, but in reality has no freedom at al.

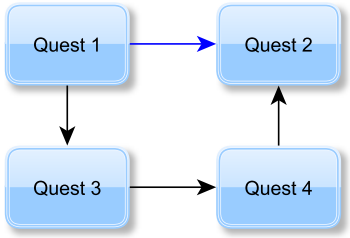

Figure 3.6: Example of a redundant link

Figure 3.6, shows an example of a quest chain with a redundant dependency (link), which is colored blue. If you look at the quest chain you will notice that there is only one way to complete the quest chain, namely in the order: quest 1, 3, 4 and finally quest 2. Also notice that the dependency on quest 1 by quest 2 is implicitly satisfied by the previously mentioned order. In other words, there is no way for the player to get to quest 2 without having to complete quest 1 first. Therefore, the link between quest 1 and quest 2 is redundant and if we would remove it, nothing would actually change with respect to the quest acceptance order. But visually the quest chain becomes an actual chain, which is much easier to understand when you look at it. In the section below we explain an algorithm that can be used to detect redundant links.

Redundant Link Detection Algorithm
In this section we will explain how redundant links can be detected. A link is redundant because the dependency is implicitly satisfied. If we put this statement in a graph oriented perspective it basically means:

"An edge is redundant if there exists another path from source to target"

The algorithm to detect all redundant edges in a quest graph is not complicated. We first remove the edge to check for redundancy from the quest graph. After which we check if there is an alternative path from the source to the target of that edge. If such a path is found the link is redundant. Then we put the edge back into the graph and we continue the algorithm for the other edges. This results in the algorithm that is shown below.

Algorithm DetectRedundantLinks(G)
Input. A quest graph G
Output. A set containing the redundant links

1. $R \leftarrow \emptyset$
2. **For** $\forall edge \in G_{edges}$
3. $\quad RemoveEdge(edge, G_{edges})$
4. $\quad$ **if** $Dijkstra(edge_{source}, edge_{target})$
5. $\quad\quad Add(edge, R)$
6. $\quad AddEdge(edge, G_{edges})$
7. return R

Please note that in our implementation we also check if the edge is removable because the transformation for the "main/sub quest"-relationship will create a redundant link. However, the link that would be redundant in that relationship is considered not to be redundant. The reason why this edge is not redundant, is because its function is to guarantee that the objectives of the main quest are completed before finishing the quest. These objectives are stored in the start part of the main quest, which means that the path via the sub-quest can't guarantee that these objectives are completed. The reason for this is that the sub quest is accepted when the start part of the main quest is active and therefore its objectives might not yet be completed. As a result the link can't be considered to be redundant. This is a perfect example of why it is up to the game designer to decide if a link is really redundant or not.

3.3.2 Quest Chain Atoms

Most quest systems are rather large and because of that the game designer can become overwhelmed by the number of elements, when looking at a visualization of such a quest system. The game designer will most likely need to work on only a few quest(chain)s at the same time, but this can be hard because the visualization is big and complex. In this paper we introduce the concept of Quest Chain Atoms to solve this problem. A quest chain atom can be described as a quest chain in which each quest maximally has one incoming and one outgoing dependency link. As a result of the structure of these links, the flow throughout the quest chain atom is fixed. Therefore, it will only have one order in which the quests can be completed which means that it preserves the structure of the original quest chain. By replacing (parts of) a quest chain with a quest chain atom, we simplify the (visualization of the) quest system.

Quest chain atoms are not only used to make the visualisation of the quest graph more compact, they can also be used to detect parts of the quest system that are complex. This can serve as a indication for the game designer that a part of the quest system needs investigation. Later in this section a more detailed example is given.

Figure 3.7 shows an example of a quest chain which contains 6 quests. When a game designer looks at this quest chain, he might not immediately realize that there is only one way a player can complete this chain. Please note that this chain contains a redundant link, which is colored blue. We will explain the relationship between quest chain atoms and redundant links at a later stage.

The reason why the game designer might not immediately notice that there is only one path, is because the quest chain is not as compact is it could be. When we atomize this chain it becomes more compact and as a result it is easier to see the flow through this chain. Figure 3.8 shows what the chain looks like when it is turned into several quest chain atoms. By the way we call this process atomization.

As a result of the atomization it is easier to see that there is only one way in which the player can complete this quest chain. But that's not all, if we combine this technique with the detection of redundant links we can make the quest chain even more compact.

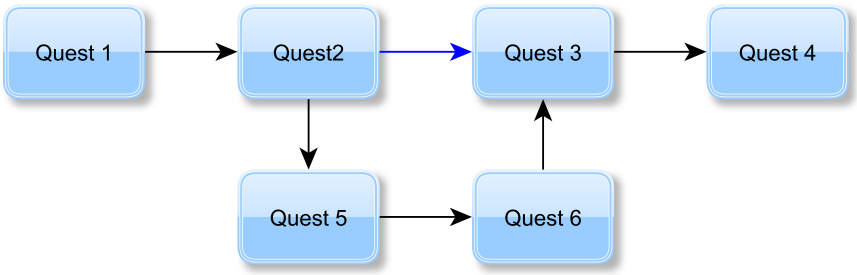

Figure 3.7: Example of a quest chain

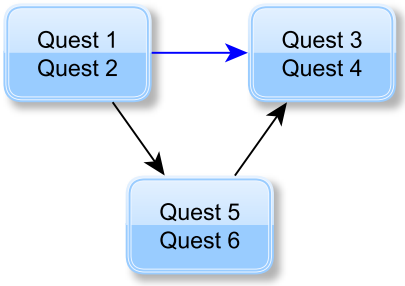

Figure 3.8: Example of the atomized quest chain

First we detect the redundant links, after which it is up to the game designer to decide if the link is really redundant. In this case we assume that the game designer decides to remove the redundant link. As a result we end up with a chain in which each quest only has one incoming and one outgoing link. If the game designer now looks at the chain it is immediately clear that this chain can only be traversed in one order. In addition, if we would now atomize the quest chain it would become one single quest chain atom, which is shown in Figure 3.9.

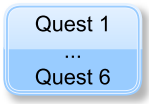

Figure 3.9: The atomized quest chain without redundant links

While in this case it still is relatively easy to see in which order the player can perform the quests, it becomes increasingly more difficult when there are more quest chains involved. In such a situation the structure of quest chain atoms can also serve as an indication of a poorly designed quest chain. To illustrate this, Figure 3.10 shows two quest chains. One chain consists of all the blue quests, while the other chain consists of the red quests. Again the redundant links are shown in blue.

If we would atomize these quest chains, we would end up with an atomized structure which is the same as the normal quest chain structure. The fact that the structure of the atomized quest chains is the same as the structure of the normal quest chains indicates that the original quest chains are very tightly coupled. This should be a trigger for the game designer to investigate these two quest chains. At this point it is best if he checks if there are redundant links in the quest chains. In this case there are three, and the game designer decides to remove them. After removing these links it becomes apparent that there is only one flow through the quest chains. At this point the game designer can consider to merge to two chains into one chain or to change the dependencies between the two chains.

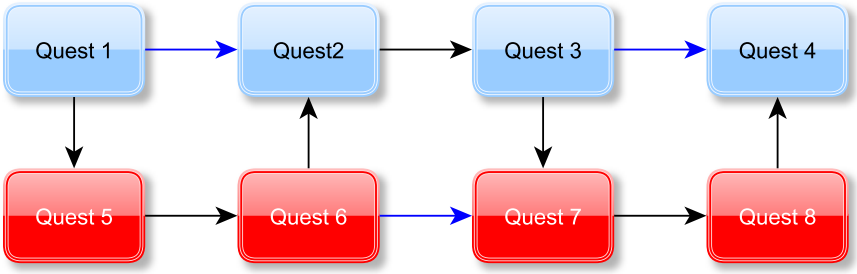

Figure 3.10: A more complex quest chain

The examples have shown that atomization simplifies the structure of a quest chain. This can help a game designer detect sections of the quest system which might need further investigation. By combining this technique with redundant link detection the game designer can optimize the quest chain (and its visualization) even further.

Please note that the "main/sub quest"-relationship does not influence the atomization process. In fact it captures the complexity involved in sub quests really well. This is due to the fact that each start and end part of the main quest will both be a separate atom.

Atomization Algorithm
In order to atomize a quest graph we first need to find which quests are the start of an atom. By first finding all starting points we can later retrieve the quest chain atoms in an efficient way. The algorithm for checking if a quest/node is the start of an atom is shown below. We check for three cases that result in a quest becoming a starting point for an atom:

1. A node has no incoming edges.
2. A node more than one incoming edges.
3. A node has exactly one incoming edge, but it is from a different quest chain.

Algorithm IsAtomStart(*node*)
Input. A QuestNode *node*
Output. A boolean indicating if this is the start of a Quest Chain Atom

1. $previousNode \leftarrow node_{previous}$
2. **if** $|node_{in-edges}| = 0$
3. return *true*
4. **if** $|node_{in-edges}| > 1$
5. return *true*
6. **if** $|node_{in-edges}| = 1$ and $node_{questchain} \neq previousNode_{questchain}$
7. return *true*
8. return *false*

Now that we can check if a node is the start of an atom, we can define the algorithm for creating the atoms. First we find all nodes that are the starting points of an atom. Then we scan forward until we encounter another node that is either the start of an atom or has multiple out edges. During the forward scan we put all the encountered nodes into a set. The nodes in the set are the nodes that belong to one quest chain atom. We repeat this process for all atom starting points in order to find all quest chain atoms.

Note that the algorithm doesn't describe how quest chain atoms can be linked, but this operation is not difficult. We check each first node of the quest chain atom for it's dependencies, after which we check in which atom the

dependency node is and we create an edge between them. The node on which we depend must be the end of a quest chain atom, so we can use this knowledge to patch the atoms together.

Algorithm AtomizeGraph(G)
Input. A graph G, that needs to be atomized
Output. A list with Quest Chain Atoms

1. $atoms \leftarrow \emptyset$
2. $atomStartnodes \leftarrow \emptyset$
3. **for** $\forall node \in G_{nodes}$
4. **if** $IsAtomStart(node)$
5. $Add(node, atomStartNodes)$
6. **for** $\forall atomStartNode \in atomStartNodes$
7. $currentNode \leftarrow atomStartNode$
8. **while** $currentNode \neq NULL$
9. $Add(currentNode, currentAtomNodes)$
10. **if** $|currentNode_{out-edges}| \neq 1$
11. break
12. $currentNode \leftarrow currentNode_{next}$
13. **if** $currentNode \in atomStartNodes$
14. break
15. $Add(currentAtomNodes, atoms)$
16. return $atoms$

3.3.3 Detecting Cycles

Another important piece of information that can be obtained from the quest graph, is the occurrence of cycles. A cycle is a situation in which circular dependencies between quests exist. If a cycle is present in a quest system, it is not possible for the player to accept/complete any of the quests that are involved in the cycle. This can lead to a situation in which a player gets stuck in the game, which is undesirable. The simplest example of an cycle is the situation in which two quest both depend on each other. Because this is a cycle which only involves two quests it is easy to detect, but bigger cycles are harder to spot. Therefore it is important for a game designer to be able to detect such a cycle, and resolve the cyclic dependencies. The section below explains how a cycle can be detected in a quest graph.

Cycle Detection Algorithm
Cycles can easily be detected in a quest graph. In our case we not only want to detect if there are cycles, but also which quests are involved in these cycles. We first clone each connected component, after which we try to destroy it completely. This can be achieved by recursively removing all nodes that have no incoming or outgoing edges from the connected component. If a connected component contains zero nodes after being destructed, it contains no cycle. If it contains one or more nodes, it contains a cycle. The algorithm for detecting cycles is shown below.

Algorithm DetectCycles(G)
Input. A graph G
Output. A list of graphs involved in a cycle

1. $componentsWithCycles \leftarrow \emptyset$
2. **for** $\forall component \in G_{connected-components}$
3. $\quad purgedGraph \leftarrow DestructFrontBack(component)$
4. $\quad$ **if** $purgedGraph_{nodes} > 0$
5. $\quad\quad Add(purgedGraph, componentsWithCycles)$
6. return $componentsWithCycles$

Algorithm DestructFrontBack(G)
Input. A graph G
Output. A graph with all nodes removed which have no incoming/outgoing edges

1. $clone \leftarrow G_{clone}$
2. $nodesRemoved \leftarrow true$
3. **while** $nodesRemoved$
4. $\quad nodesRemoved \leftarrow false$
5. $\quad$ **For** $\forall node \in G_{nodes}$
6. $\quad\quad$ **If** $|node_{in-edges}| = 0$ or $|node_{out-edges}| = 0$
7. $\quad\quad\quad RemoveNode(node, G)$
8. $\quad\quad\quad nodesRemoved \leftarrow true$
9. return $clone$

3.3.4 Shortest Path Detection

For a game designer it might be interesting to obtain the shortest path from one quest to another. Not only does the shortest path represent the minimal distance between quests, it also indicates the minimal effort the player needs to put into the game to progress from one quest to the other. By analyzing the shortest path the game designer can validate if for example all of the quests which tell the main plot are present. The shortest path can only be calculated if the start and end quests of the this path are both part of the same connected component.

One of the problems that we face when executing path queries, is that it is possible to create a query from which the start and end quests aren't connected. Figure 3.11 shows an example of such a situation, when we query from Quest 5 to Quest 7. There are two ways to deal with this problem. The first way is to assume that the user meant a different query, which in this example means that the user wanted to query from Quest 2 to Quest 7. However, this is not a good solution because it might not be clear to the user what is happening when we change the query. Therefore, a better approach is to just check if the two quests are connected, if they are connected then we can execute the query. If the quests are not connected, the user is informed of the badly configured query.

An example of a correct query on the same quest chain is shown in Figure 3.12. This time we search for the shortest path from Quest 4 to Quest 6. Because there is a path between the two quests, we can execute the shortest path query. Because we start at Quest 4, we know that Quest 1 and Quest 2 have been completed. So that leaves us with Quests 3, 4, 5 and 6 being the shortest path from quest 4 to quest 6.

Please note that the shortest path obviously doesn't include other connected components than the one that contains the start and end quest of the path.

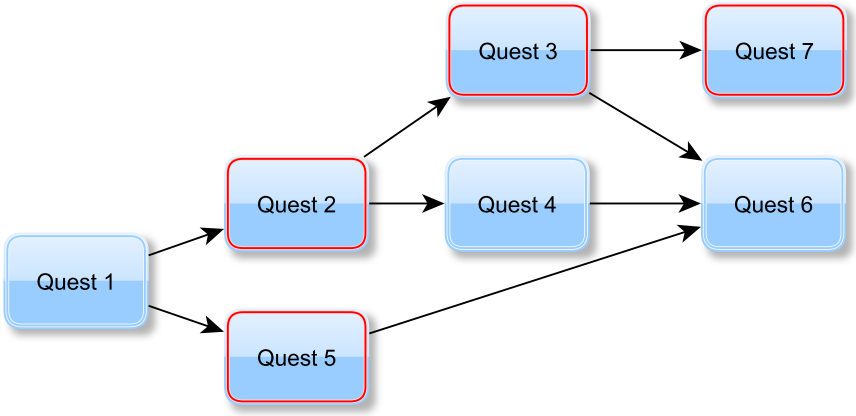

Figure 3.11: Incorrect query from quest 5 to quest 7

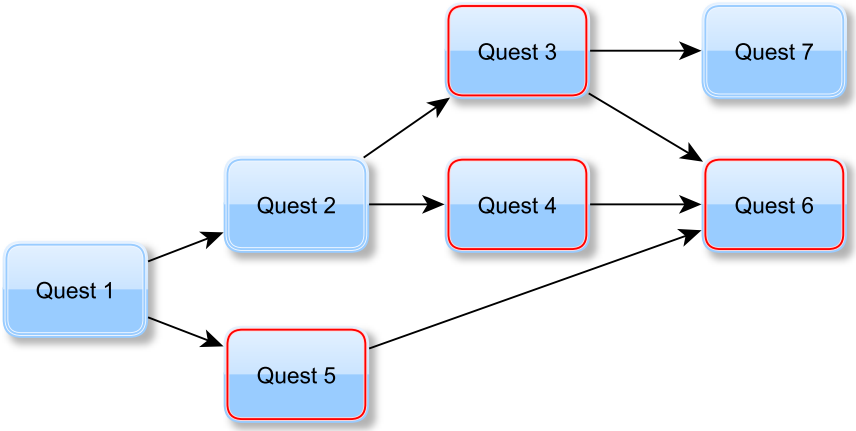

Figure 3.12: The shortest path from quest 4 to quest 6

Shortest Path Detection Algorithm
To detect the shortest path between to quests, we need to first remove all quests which occur before the "from"-quest. These quests are removed because the player already completed them and therefore they can't be part of the shortest path. Then we are going to backtrack from the "to"-quest. We backtrack until there are no more incoming edges. At this point we have found all quests which are part of the shortest path, namely all the nodes over which we backtracked. From this path we create an random topological order in order to have a valid shortest path. The algorithms for these operations are shown below. Please note that the "from"- and "to"-nodes are removed from the graph in order to ensure that they are in fact the first and last node on the path. The "Topological Ordering"-algorithm does not ensure this. Also note that this algorithm does not contain a check to verify that the start and end nodes of the query are connected. Such a check is trivial and can be performed by running Dijkstra's algorithm on the graph.

Algorithm ShortestPath(G, $fromNode$, $toNode$)
Requirements. Visited must be set to $false$ for all nodes and edges
Input. A graph G, a node $fromNode$ which is the start of the query, a node $toNode$ which is the end of the query
Output. The shortest path $path$ as a list of nodes

1. $RemoveBefore(fromNode, G)$
2. $graphClone \leftarrow BacktrackPath(toNode, G)$
3. $RemoveNode(fromNode, graphClone)$
4. $RemoveNode(toNode, graphClone)$
5. $path \leftarrow TopologicalOrdering(graphClone)$
6. $AddFirst(fromNode, path)$
7. $AddLast(toNode, path)$
8. return $path$

We propose to use a stack for the empty set that is defined in the first line of the "RemoveBefore"-algorithm. It how ever is possible to use a queue but that will give the algorithm slightly different properties.

Algorithm RemoveBefore($fromNode$, G)
Input. A node $fromNode$ for which we need to remove all nodes before it. A graph G.

1. $nodesToRemove \leftarrow \emptyset$
2. **for** $\forall inEdge \in fromNode_{in-edges}$
3. $\quad Push(inEdge_{source}, nodesToRemove)$
4. **while** $nodesToRemove \neq \emptyset$
5. $\quad node \leftarrow Pop(nodesToRemove)$
6. $\quad RemoveNode(node, G)$
7. $\quad$ **for** $\forall inEdge \in node_{in-edges}$
8. $\quad\quad Push(inEdge_{source}, nodesToRemove)$

Please note that the empty set defined in the first line of the "BacktrackPath"-algorithm creates a new instance of a quest graph. Furthermore we advise to use a stack for the set that is initialized in the second line of the algorithm.

Algorithm BacktrackPath($toNode$, G)
Requirements. Visited is set to $false$ for all nodes
Input. A node $toNode$ from which we start backtracking. A graph G.
Output. A graph containing the nodes of the path

1. $graphClone \leftarrow \emptyset$
2. $nodesToProcess \leftarrow \emptyset$
3. $Push(toNode, nodesToProcess)$
4. **while** $nodesToProcess \neq \emptyset$
5. $\quad node \leftarrow Pop(nodesToProcess)$
6. $\quad$ **if** $node_{visited} \neq true$
7. $\quad\quad node_{visited} \leftarrow true$
8. $\quad\quad AddNode(node, graphClone)$
9. $\quad\quad$ **for** $\forall previousNode \in node_{previous}$
10. $\quad\quad\quad$ **if** $previousNode_{visited} \neq true$
11. $\quad\quad\quad\quad Push(previousNode, nodesToProcess)$
12. Rebuild edges for $graphClone$ based upon edges in G
13. return $graphClone$

Please note that this and many other algorithms explained in this paper use the topological ordering algorithm that is described in "Introduction to Algorithms" [ea01]. This algorithm was originally described by Tarjan [Tar76].

3.3.5 Longest Path Detection

Just like the shortest path between two quests, there exists a longest path between those two quests. The longest path between quests can be seen as the maximal distance between those two quests. In terms of game play this means that the path contains all quests that the player can do before he ultimately must accept the quest at the end of the path. This can be an useful tool for the game designer to estimate how far the start and end quest are virtually apart. For example it might be possible that two quests need to be completed close after each other in order to tell a story without the player forgetting about the start of the story. By checking the longest path the game designer can guarantee that there is a maximal number of quests between the quests that needed to be close together.

With the longest path it is also possible to create a badly configured query. Figure 3.13 shows the same bad query, from quest 5 to quest 7, but now for the longest path. As was explained in the section on the shortest path, it is generally not a good idea to allow such queries. Therefore, it is best to detect such queries, by running Dijkstra's algorithm, and inform the user.

An example of a correct query, is a query from quest 4 to quest 6. If we compare this with the same query for the shortest path, we can see that the paths do not really differ much. The longest path query contains one extra quest, quest 7, that the player can complete. This is shown in Figure 3.14.

Please note that the longest path by default includes all other connected components in order to find the longest path. While we don't explain this in this paper it might be a good idea to enable the possibility exclude other connected components from this path. While it is possible that the quests can be completed in parallel, there might be other constraints which are not reflected in the quest graph. An example of such case is that the connected components are part of different regions of the game, making it unlikely for a player to complete the quests in parallel. For the game designer it might be interesting to see the longest path without the other connected components. Therefore it is up to the game designer to decide if the connected components are

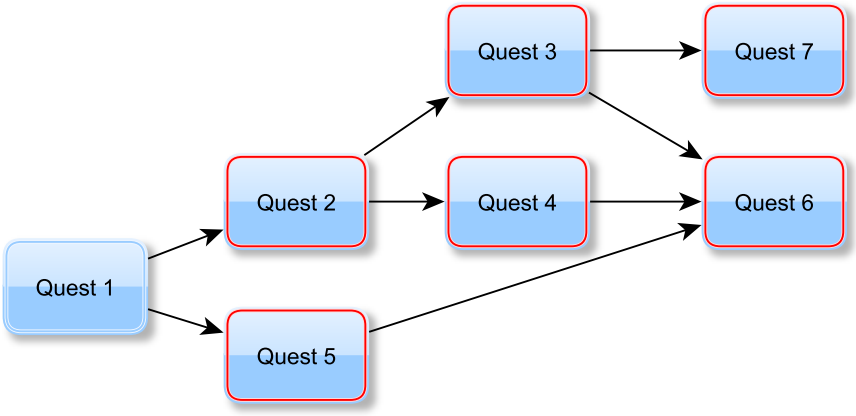

Figure 3.13: The longest path from quest 5 to quest 7

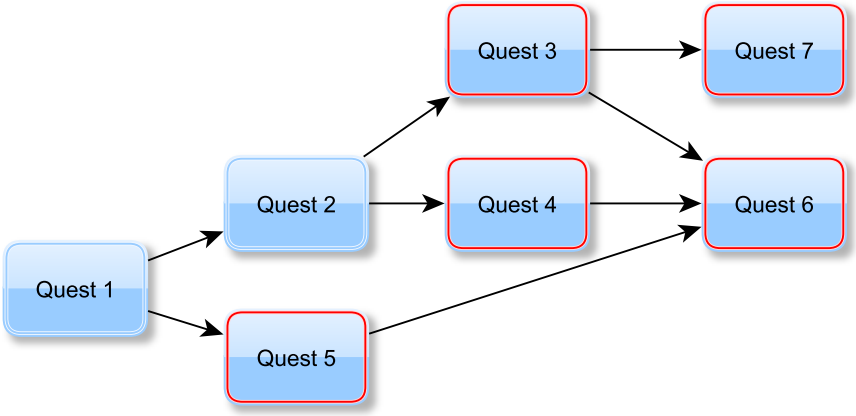

Figure 3.14: The longest path from quest 4 to quest 6

relevant or not.

Figure 3.15 shows an example of a quest graph which consists of three connected components. If we query from quest 1 to quest 4 and we exclude the other connected components we will end up a path that consists of quests 1 to 5. If we include the other connected components all of the other quests are on that path as well.

Longest Path Detection Algorithm
Just like with the "shortest path"-algorithm, we start by removing the quests that have been completed. This is accomplished by removing all quests before the "from"-quest. Besides removing the quests that are already completed, we must remove the quests that can't be completed without completing the "to"-quest. This can be accomplished by removing all quests which come after the "to"-quest. Now we only have the nodes for the longest path left. We remove the "to"-quest, and generate a topological ordering of the nodes on the longest path. We then add the "to"-quest to the end of the topologically generated path. This is done in order to ensure that the resulting path has the "to"-node as last node. Otherwise we could get a longest path with for example the "to"-node in the middle, which is clearly not the longest path. The same is true for the "from"-node which is also removed from the graph. The algorithms for these operations are shown below.

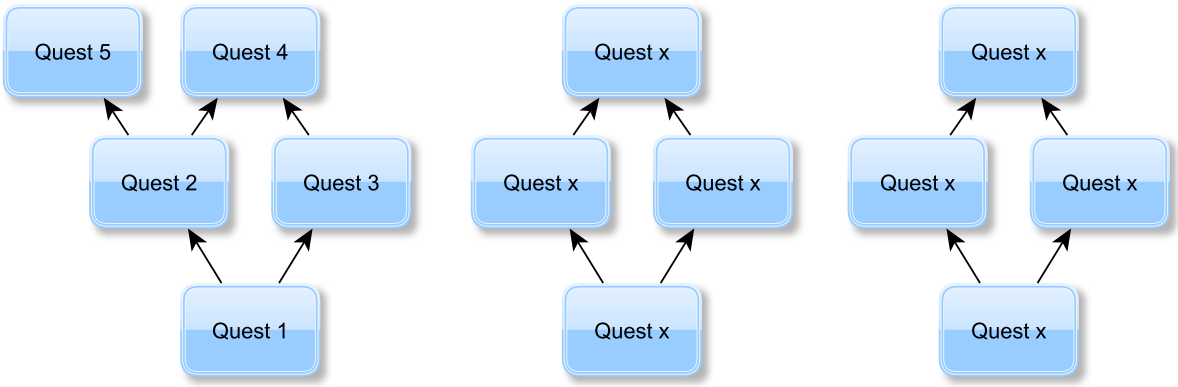

Figure 3.15: Multiple connected components can skew the results.

Algorithm LongestPath(G, $fromNode$, $toNode$)
Input. A graph G, a node $fromNode$ which is the start of the path, a node $toNode$ which is the end of the path
Output. The longest path $path$ as a list of nodes

1. $RemoveBefore(fromNode, G)$
2. $RemoveAfter(toNode, G)$
3. $RemoveNode(fromNode, G)$
4. $RemoveNode(toNode, G)$
5. $path \leftarrow ToplogicalOrdering(G)$
6. $AddFirst(fromNode, path)$
7. $AddLast(toNode, path)$
8. return $path$

Algorithm RemoveBefore($fromNode$, G)
Input. A node $fromNode$ for which we need to remove all nodes before it. A graph G.

1. $nodesToRemove \leftarrow \emptyset$
2. **for** $\forall inEdge \in fromNode_{in-edges}$
3. $\quad Push(inEdge_{source}, nodesToRemove)$
4. **while** $nodesToRemove \neq \emptyset$
5. $\quad node \leftarrow Pop(nodesToRemove)$
6. $\quad RemoveNode(node, G)$
7. $\quad$ **for** $\forall inEdge \in node_{in-edges}$
8. $\quad\quad Push(inEdge_{source}, nodesToRemove)$

Please note that the "RemoveAfter"-algorithm is not shown. This algorithm is basically the same as the "RemoveBefore"-algorithm, with the difference that it only removes all nodes at the outgoing end of an edge. In addition note that while it is not shown in the examples above, the longest path does include other connected components into the path.

3.3.6 Player Freedom - Mandatory Quest Ratio

A game designer must design a quest system in a way that it attracts the widest range of players. Nowadays most players like to play games which provide them with a fair amount of freedom. Therefore, it is important that the game designer has a notion on how much freedom a player actually has. We devised a simple metric that can be used to indicate how much freedom the player has in choosing which quests to accept. The metric will be calculated between two quests, because it uses path based analysis. We define the "Mandatory Quest Ratio" as:

$$mqr = \frac{|shortestpath|}{|longestpath|}$$

When the ratio is 1 the player has no freedom in deciding which quests he want to complete. The closer the ratio is to 0, the more freedom a player has when it comes to choosing quests.

In addition, the mandatory quest ratio can be used to analyze if the quest content satisfies the needs of different types of players. For most MMORPG's there basically are two types of players:

1. Finisher. A player that only finishes the minimal amount of quests to complete the game.
2. Achiever. A player which will complete every single quest in order to complete the game.

If the ratio is close to 1, it indicates that there is no freedom in choosing quests. This means that both the "finisher" and the "achiever", will need to complete all the quests. This might not be the desired experience for the "finisher", because he only wants to complete the minimal amount of quests in order to finish the game. However, for the "achiever" it makes no difference because he would complete all quests anyway. In the other case, when the mandatory quest ratio is low, the ratio indicates that the "finisher" will complete a much lower amount of quests than the "achiever". In such a scenario both players will be satisfied because they will have the freedom they desire.

We will now give two examples of this metric in action:

Example 1 A Mandatory Quest Chain

Example 2 A Quest Chain With Optional Quests

Example 1 - A Mandatory Quest Chain
For the first example we calculate the mandatory quest ratio on the static quest chain that is shown in Figure 3.16. This quest chain has 5 quests which are all mandatory in order to progress from quest 1 to quest 5. Therefore the shortest and longest path in this section are the same. The shortest path counts 5 quests and the longest path counts 5 quests. Therefore the mandatory quest ratio is 1 ($\frac{5}{5}$). So if the ratio is 1 the player has no freedom in deciding which quests he want to complete.

Example 2 - A Quest Chain With Optional Quests
For the second example we calculate the mandatory quest ratio for the quest chain that is shown in Figure 3.17. We assume that we must reach quest 8 in order to complete this quest chain. In this case the shortest path from quest 1 to quest 8 contains 5 quests, while the longest path contains 8 quests. So for this example the mandatory quest ratio is 0.625 ($\frac{5}{8}$), which indicates that the player has more freedom than with the previous example.

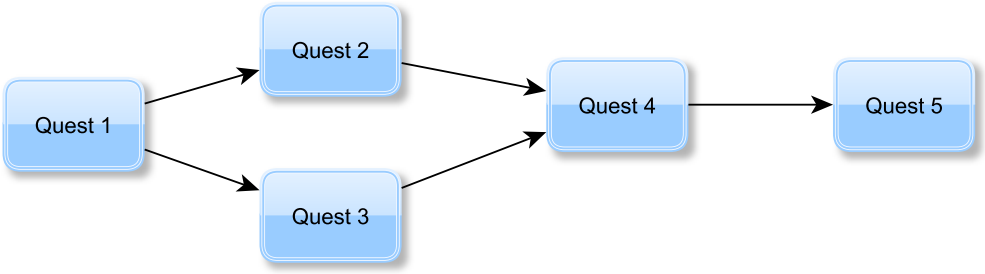

Figure 3.16: A mandatory quest chain

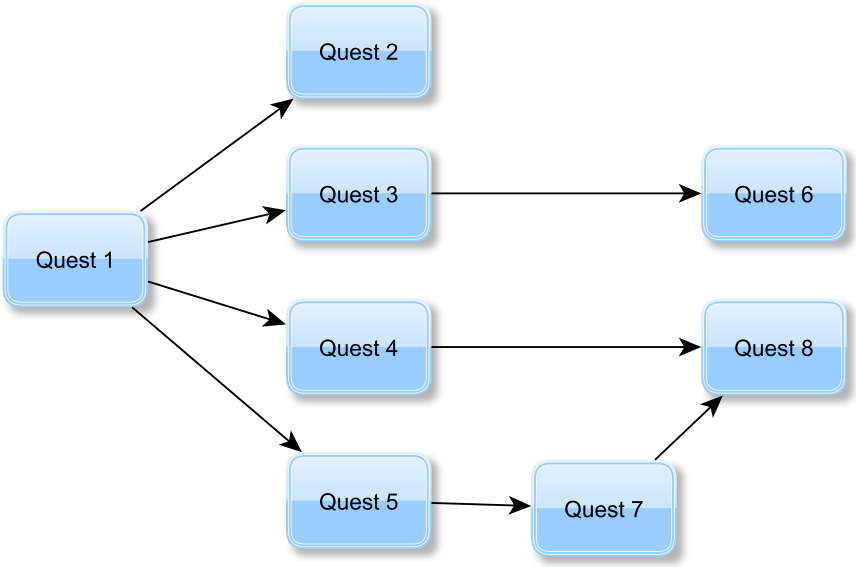

Figure 3.17: A quest chain with optional quests

3.3.7 Player Freedom - Quest Order

In the previous section we looked at how we can quantify the amount of freedom a player has when it comes to accepting quests. However that is only one facet that describes the freedom of the player. Another facet of freedom is the order in which the player can accept quests. For example some players like to first complete quests which involve killing NPC's while others enjoy the sheer excitement of exploring a new area. Therefore, it might be interesting for game designers to be able to quantify this type of freedom. In this section we explain several attempts at capturing this type of freedom.

For our first attempt to capture the quest acceptance diversity we attempted to count the number of possible quest acceptance orders. The section below explains how we define a quest order.

Quest order
A quest order is used to indicate a particular order in which it is possible to progress from one quest to another. Figure 3.18 shows an example of a quest chain with multiple quest orders when we assume that we want to progress from Quest 1 to Quest 4.

The quest orders for this quest chain are:

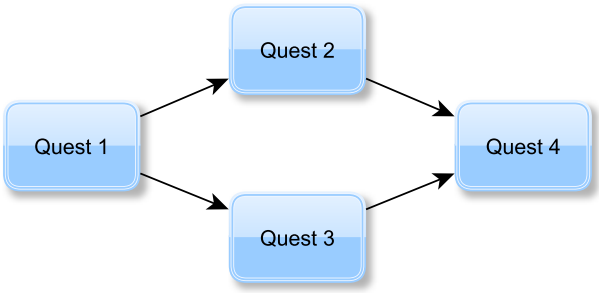

Figure 3.18: A simple quest chain

1. First Quest 1, then Quest 2, Quest 3 and finally Quest 4
2. First Quest 1, then Quest 3, Quest 2 and finally Quest 4

A player must first complete quest 1 and can then choose if he first wants to do quest 2 or 3. After which the player must complete quest 4 to complete the quest chain.

In a versatile quest chain there are many unique quest orders which can be used to reach the end of the quest chain. Therefore, the number of unique quest orders can serve as an indication of the freedom a player has in choosing in which order he can play through that quest chain.

In the previous example we have shown what a quest order looks like when we look at it from a quest chain point of view. However this doesn't mean that a quest order can only be used for quest chains. For example it is possible to define a quest order which starts at one quest chain and ends at another.

Unfortunately calculating the exact number of possible orders in a quest graph is a #P-complete problem, which is proven in a paper[BW91]. However, there are papers which describe how you can implement approximation algorithms which will give a decent indication of the total number of orders in a reasonable amount of time. Examples of such algorithms can be found in the paper "Using TPA To Count Linear Extensions"[ea10] or in the paper "Counting Linear Extensions of a Partial Order"[Hared]. However, we will not research one of these algorithms because they are complex and outside the scope of our research.

During the research on finding the total number of quest orders we also found the paper "On Computing The Number of Topological Orderings of a Directed Acyclic Graph" [ea05]. This paper proposes a method to count the number of quest orders by using combinatorial methods. However, it seems that the algorithm that is outlined in the paper is flawed. We used this algorithm on several different graphs and in some cases the count was off by a fairly high amount. While this paper did not solve our problem, it did give us an idea on another way to express a player freedom metric.

In the previously mentioned paper, the authors use intervals to partition the quest graph. It turns out these intervals can be used to give a good indication of the player freedom. For each quest in the quest graph an interval can be calculated which indicates at which step the quest can first be accepted and which is the latest possible step at which the quest must be accepted. By combining these intervals we can see how many quests can be active at a given time. We will explain how we can use these intervals to express the freedom of a player in the following three examples:

Example 1: A Static Quest Chain

Example 2: A Less Static Quest Chain

Example 3: A Complex Quest Chain

Example 1 - A Static Quest Chain
For the first example we analyze the quest chain that is shown in Figure 3.19. We see that this is a very static quest chain, and there is only one flow through this quest chain.

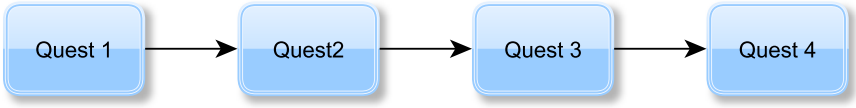

Figure 3.19: A static quest chain

If we would now extract the intervals from this quest chain we would end up with intervals which are shown in Table 3.3. The interval for each quest can easily be calculated with the equations shown below.

$$interval_{lowerbound} = |\prec quest| + 1$$

$$interval_{upperbound} = |quests| - |\succ quest|$$

The inclusive lower bound is the number of quests before the quest plus one. The inclusive upper bound is the total amount of quests in the part of the quest system we are analyzing minus the number of quests that come after the quest.

By summarizing the possibilities for each step we can capture the global variation at that step.

	Step 1	**Step 2**	**Step 3**	**Step 4**
Quest 1	√			
Quest 2		√		
Quest 3			√	
Quest 4				√
Total	1	1	1	1

Table 3.3: Quest intervals for quest chain of Figure 3.19

If we now plot the variation for each quest, we end up with the chart that is shown in Figure 3.20. It indicates that there is no variation for the player in this quest chain, because at each step the player only has one choice.

Example 2 - A Less Static Quest Chain
Now we do the same with a quest chain which is a bit less static. The chain that we will analyze this time also consists of 4 quests. Figure 3.21 shows this quest chain. Because we start with one quest and then enable three other quests, the quest chain provides the player with more freedom than the previous example. If we calculate the intervals for this quest chain, we end up with the intervals that are shown in Table 3.4. Just like before, we create an chart from these intervals which is shown in Figure 3.22.

Please note that the number of quests that is acceptable at each step must not be read as: “At step X there are N quest from which the player can choose.”, but rather as: “At step X the player could choose from n quests, if we would disregard his previous choices.”. This does not describe the exact freedom a player has, but it gives an indication on the players freedom, while it doesn’t involve any of the complex algorithms that are previously mentioned in the section on quest orders.

In this case we can clearly see that the player has more freedom in choosing quest orders. At the first step the player is forced to complete quest 1 first. But in steps 2, 3 and 4 the player can choose to complete quests 2,3 and 4 in any order.

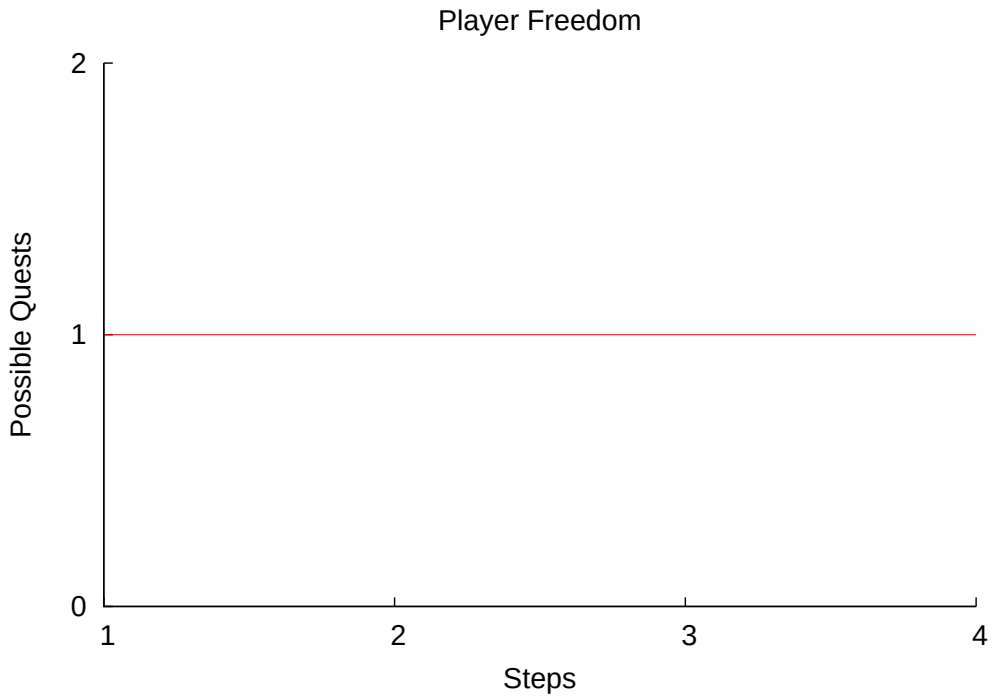

Figure 3.20: The freedom of a player in a static quest chain

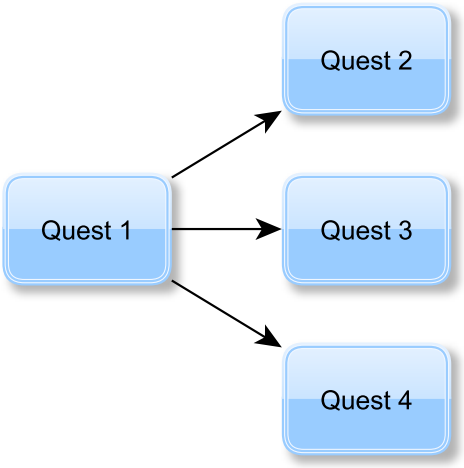

Figure 3.21: A less static quest chain

	Step 1	**Step 2**	**Step 3**	**Step 4**
Quest 1	√			
Quest 2		√	√	√
Quest 3		√	√	√
Quest 4		√	√	√
Total	1	3	3	3

Table 3.4: Quest intervals for quest chain of Figure 3.21

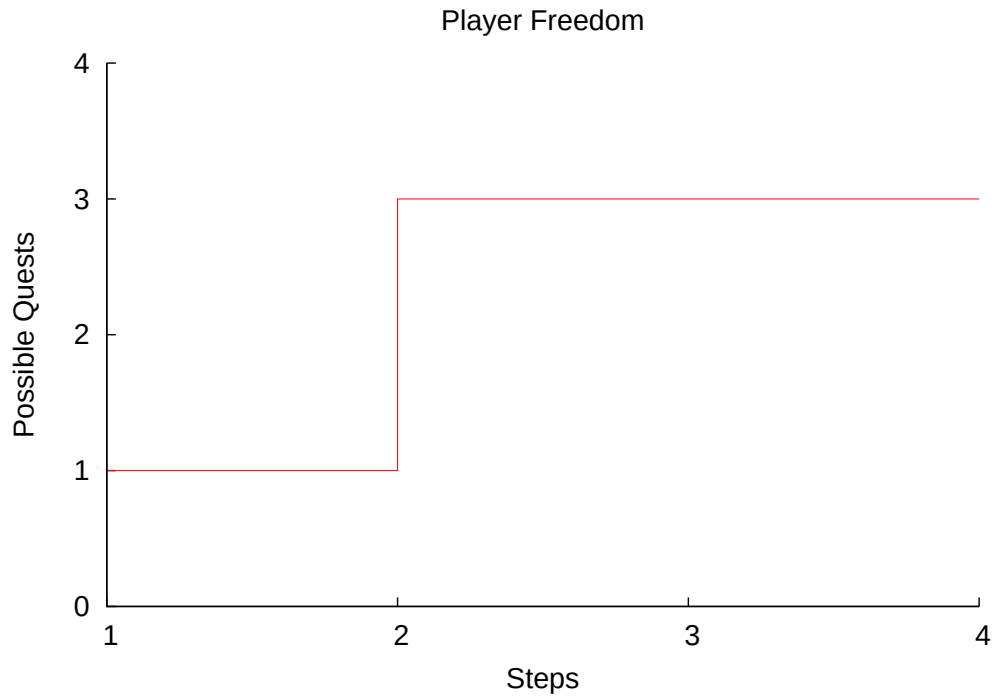

Figure 3.22: The freedom of a player in a less static quest chain

Example 3 - A Complex Quest Chain
The previous two examples have only shown cases with only a few quests and a single quest chain. In a real quest system the situations will be more complex. Therefore we will now show a more complicated example which consists of two quest chains, with multiple quests. Each quest chain has it's own distinct color. Because the two quest chains don't depend on each other, the player can decide which quest chain he wants to accept first. The quest system we just described is shown in Figure 3.23.

Step	1	2	3	4	5	6	7	8	9	10	11
Blue Chain											
Quest 1	√	√	√	√							
Quest 2		√	√	√	√	√	√	√	√		
Quest 3		√	√	√	√	√	√	√	√		
Quest 4		√	√	√	√	√	√	√	√	√	
Quest 5		√	√	√	√	√	√	√	√		
Quest 6				√	√	√	√	√	√	√	
Quest 7			√	√	√	√	√	√	√	√	
Quest 8								√	√	√	√
Red Chain											
Quest 1	√	√	√	√	√	√	√	√	√		
Quest 2		√	√	√	√	√	√	√	√	√	
Quest 3			√	√	√	√	√	√	√	√	√
Total	2	7	9	10	9	9	9	10	10	6	2

Table 3.5: Quest intervals for quest chain of Figure 3.23

In this case we can clearly see that the player has much freedom in choosing a quest. At the first step the player can only choose between the two first quests of both chains. However, this changes rapidly at the second step at which the player gains much freedom. At the next steps the amount of possible quests increase, and therefore the player freedom also increases.

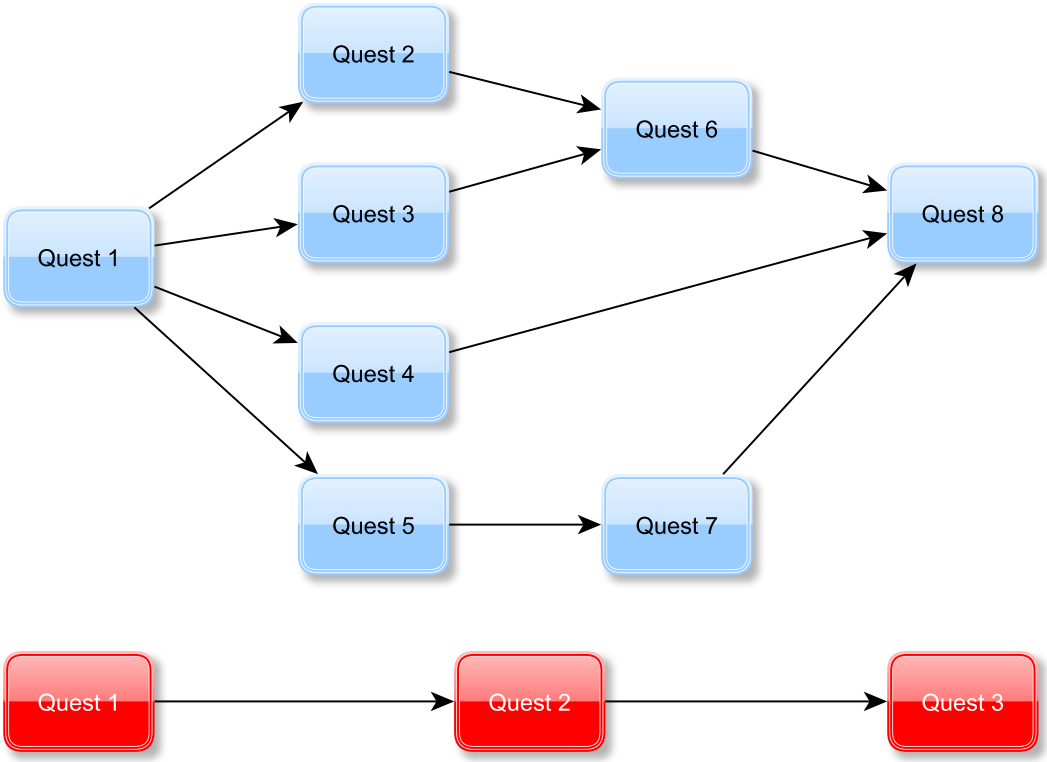

Figure 3.23: A complex quest system

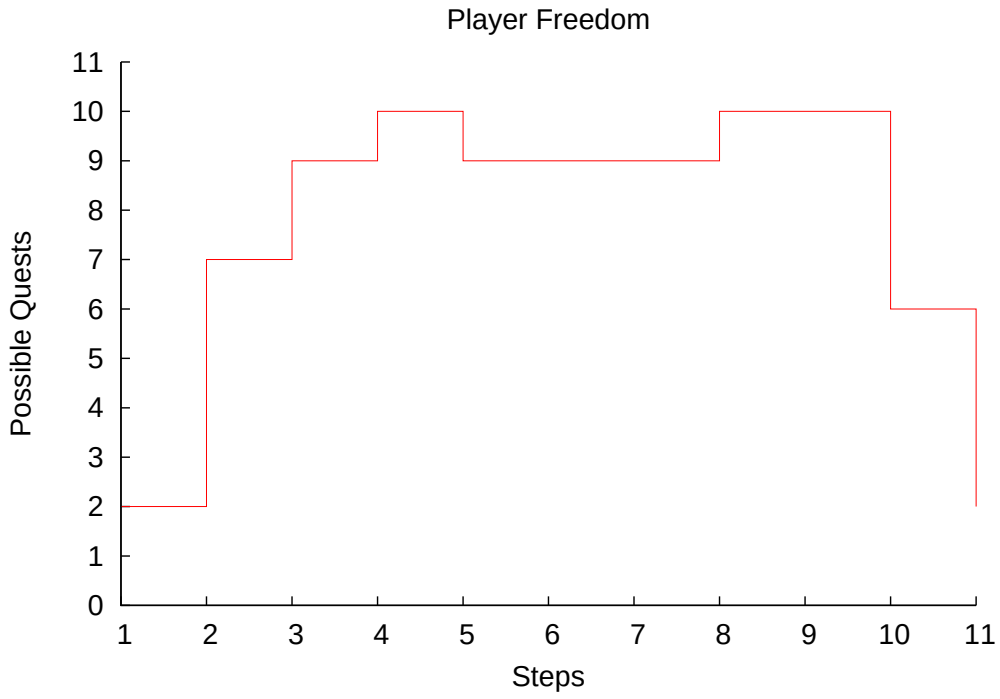

Figure 3.24: The freedom of a player in a more complex quest system

From this and previous charts we can conclude that the freedom of a player is a rather abstract metric. The general rule of thumb is that a higher amount of possible quests indicates more freedom. Also note that if there are multiple quest chains involved, the graph does not give any indication about the freedom in that chain. If we look at the red chain of Figure 3.24, we see a static chain. This is not reflected in the chart of Figure 3.24.

3.3.8 Player Freedom - Preventing Undesired Quest Orders

In the previous two sections we described how we can quantify the freedom of a player. But in some cases it is necessary that a game designer can limit the freedom of a player, because certain quests need to be performed in a certain order. In this section we explain how we can verify that a certain quest must be completed before another quest can be completed. If the quests belong to the same quest chain it is easy to verify that the order is correct, because the structure of the quest chain will express the order of the quests. However, if the quests are part of different quest chains or components it becomes harder to verify that the order is correct. Therefore, we propose a simple solution to verify if the quest order is correct. If we want to verify that *"Quest A must always be completed before Quest B"*, we need to perform the following steps:

1. First we calculate the quest intervals for both of the involved quests. This is explained in the section called "Player Freedom - Quest Order".
2. Then we check if the upper bound of Quest A is higher or equal to the lower bound of Quest B. If this is true there exists an order in which the player can complete Quest A before Quest B.

Chapter 4

Layers

In chapter 3 we explained that dividing the quest graph into connected components gives the player more freedom. However, connected components also create a new problem. In the quest system every connected component represents a part of the game, which would imply that the player can choose to play every part of the game at any given time. While this could be a viable scenario in a real game, it makes no sense when we are analysing the quest system. Therefore, we need information on how the separate connected components are related to each other. To illustrate why we need this information, imagine the situation in which the player continuously switches between game regions (with different requirements regarding the current level of the player) to complete quests. It would make more sense if the player finishes a region of the game first and then moves on to another region. However, because we only have information about connected components we cannot differentiate between different parts of the game. To enable this, we introduce the concept of layers, which help the game designer embed more information into the quest system.

Layers
By defining several types of layers in the quest system, we can group quest chains into logical groups. It is up to the game designer to specify what a layer represents. If he would choose to make layers depending on geographical location of the quests it could for example result in a system with the following layers:

1. *Quest Hub Layer*
   A layer which groups quest chains based upon their geographical location in the game.
2. *Region Layer*
   A layer which groups the quest-hub layers by their region.
3. *Continent Layer*
   A layer which groups the region layers by their continent.

Figure 4.1 shows such a configuration with some instances of each layer.

A layer can optionally contain extra information which is relevant to that layer. In the above mentioned example it is useful to indicate how a player is guided from quest hub to quest hub in a specific region.

As a result of using layers we provide the game designer with a tool to embed more information about the quest system into his analysis. This information can be used to extract more information from the quest system. In addition, because we divide the quest system into several pieces, we have effectively optimized all of the involved algorithms by making them only use a subset of all of the quests in the quest system.

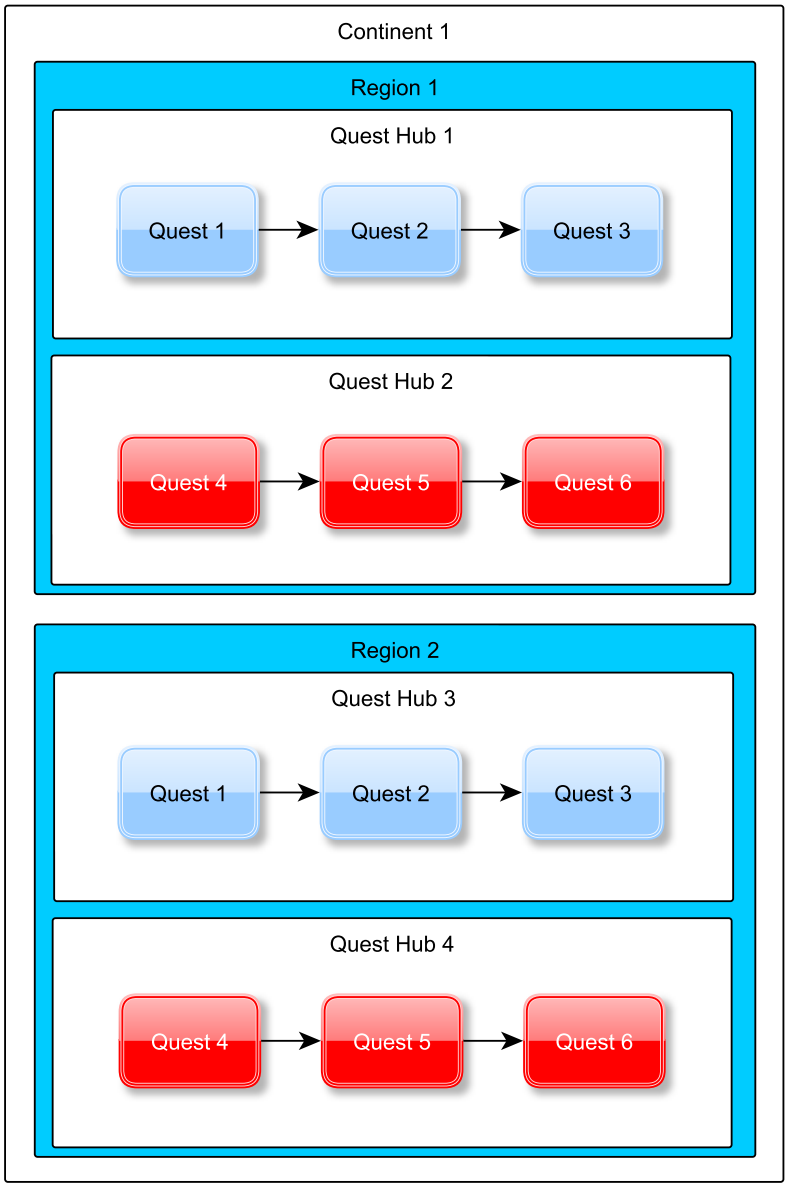

Figure 4.1: Example of a quest system with layers

Chapter 5

Case study: World of Warcraft - Cataclysm

In the previous chapters we explained how we can use a quest graph to analyse quest systems. In this chapter we will look at how the quest graph and the tools involved, can be used on a current day game. We will analyse the latest expansion of World of Warcraft by reconstructing the quest system from play sessions. In these play sessions we specifically analysed the region that is called "Mount Hyjal", which is one of the starting areas of the Cataclysm-expansion. Furthermore, the data that was obtained from the play sessions was partly complemented with data that was publicly available on the website www.wowhead.com. Besides information regarding the quest(chain)s, it also contains feedback from users which express their view on the quest system. Please note that the quest system structure we obtained for this case study is an approximation of the real quest system. Therefore, it is likely that there are errors in the quest system which influence our observations. Unfortunately this can't be prevented because we don't have access to the real data.

5.1 The Quest System

The quest system we obtained from our play sessions contains 136 quests which can be completed in the "Mount Hyjal"-region of the game. These quests are shown in Figures 5.2 to 5.7. Figure 5.1 shows how the parts of the quest system fit together. The quests which have a white color don't belong to a quest chain. All quests that have a color belong to a quest chain that consists of the quests with the same color.

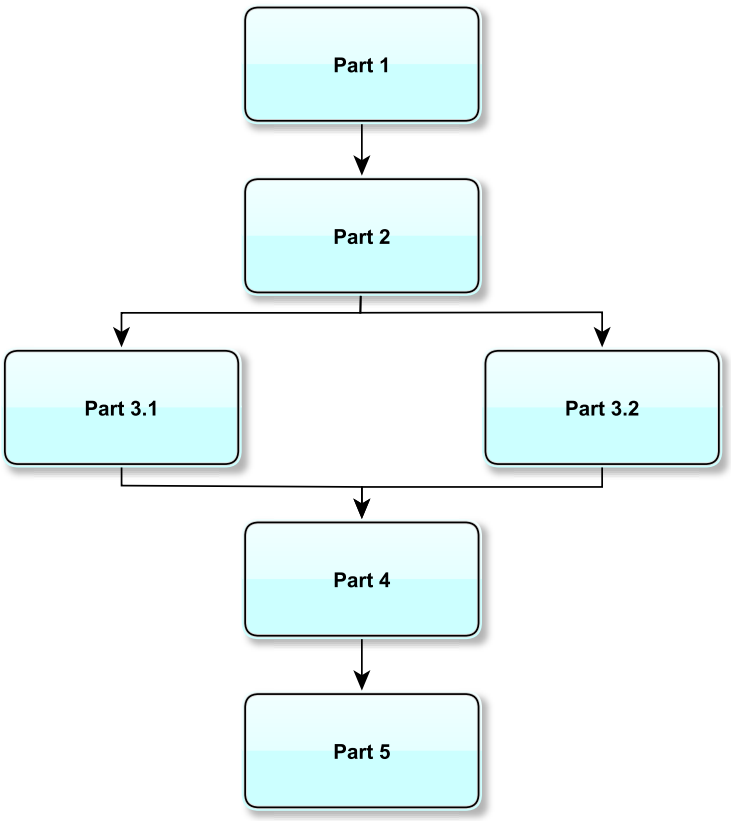

Figure 5.1: The global structure of the quest system

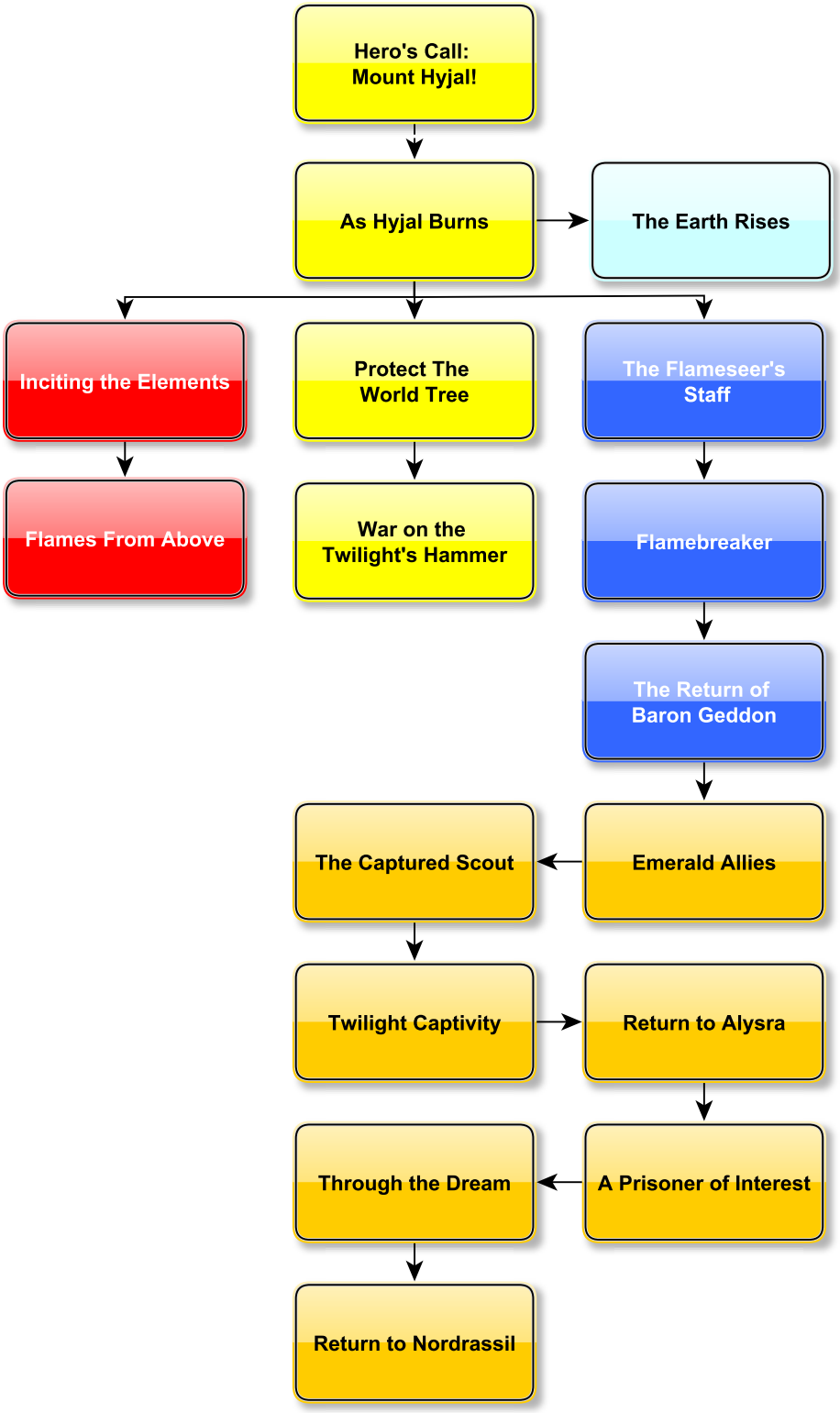

Figure 5.2: Part 1 of the Mount Hyjal quest system

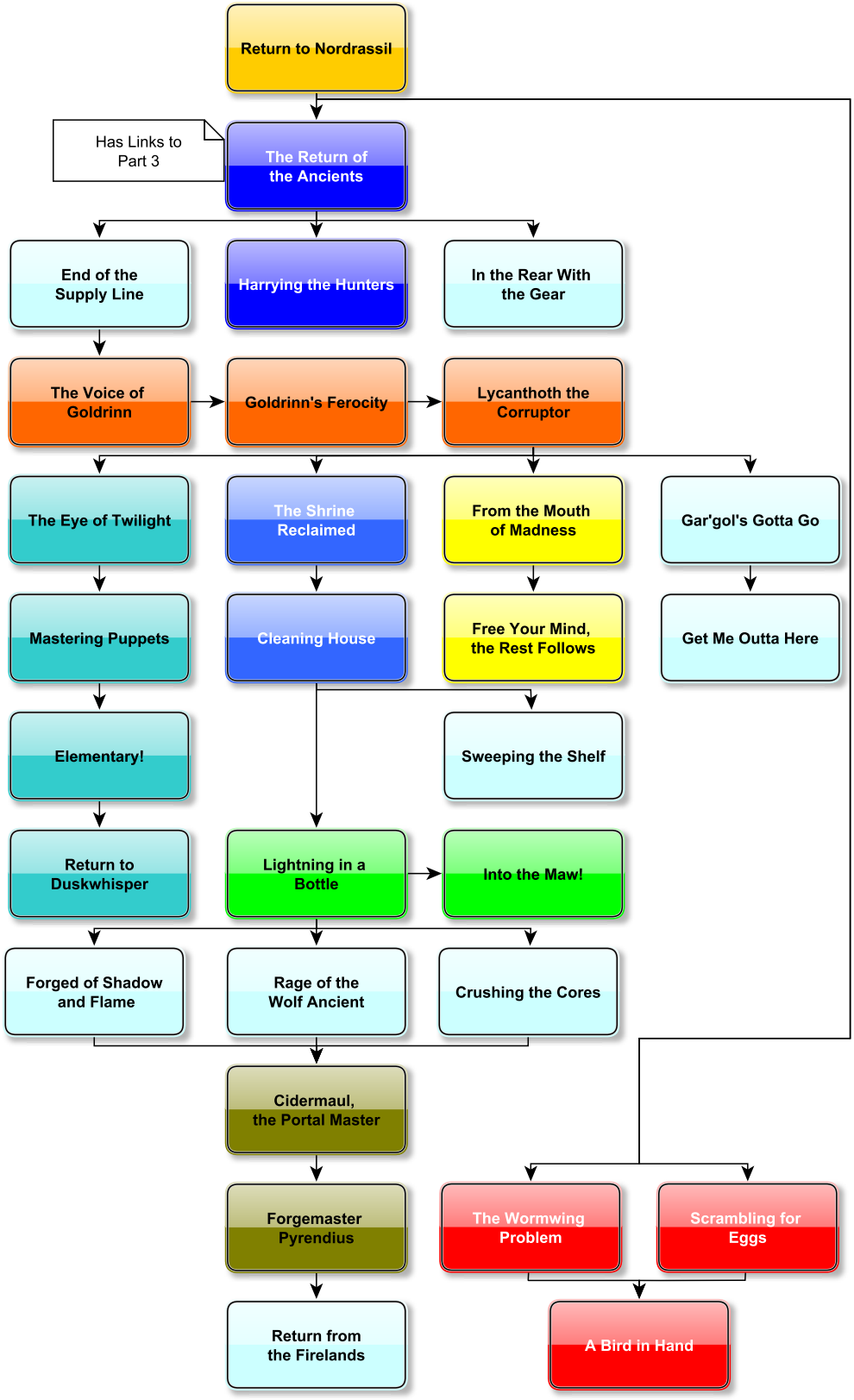

Figure 5.3: Part 2 of the Mount Hyjal quest system

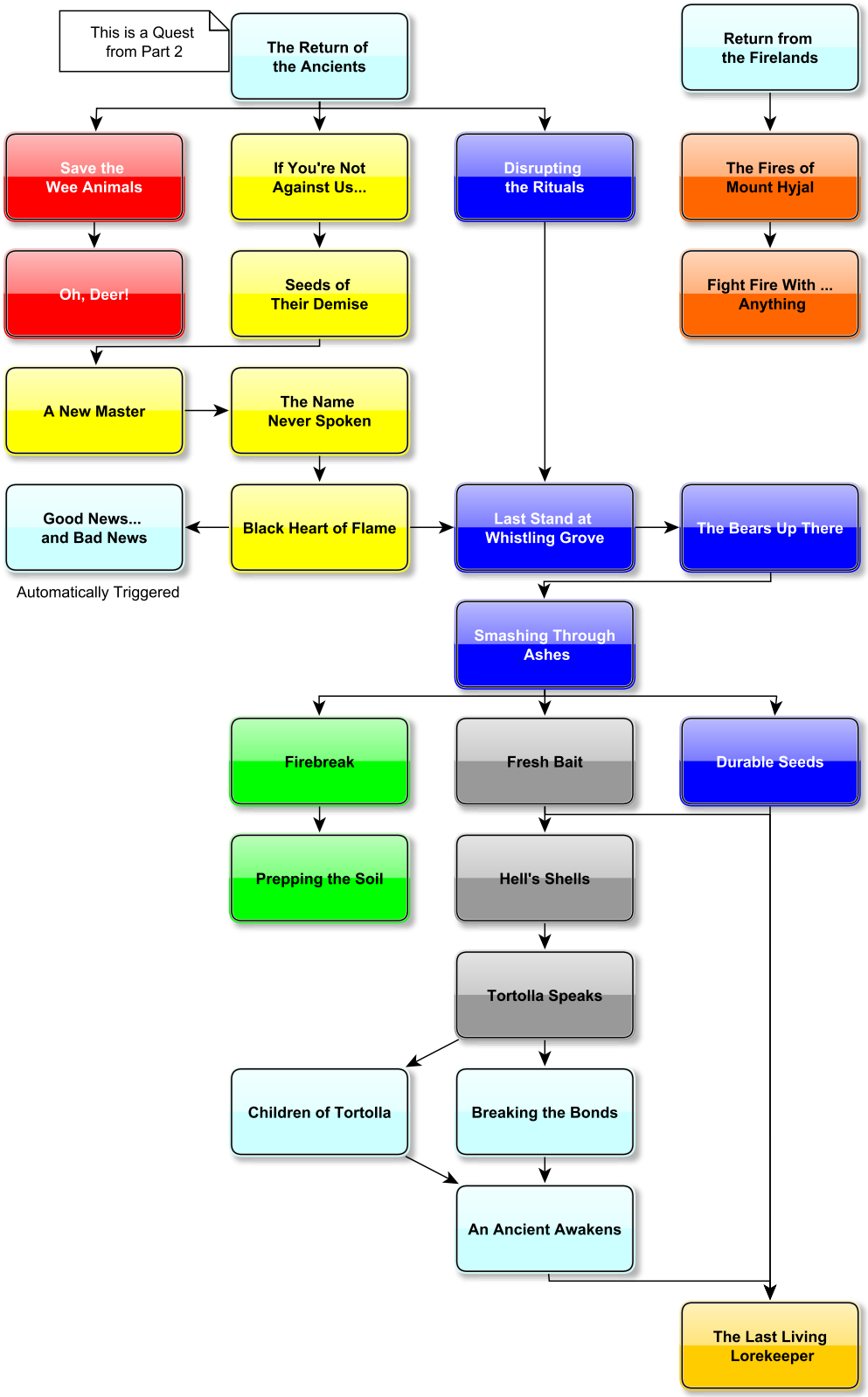

Figure 5.4: Part 3.1 of the Mount Hyjal quest system

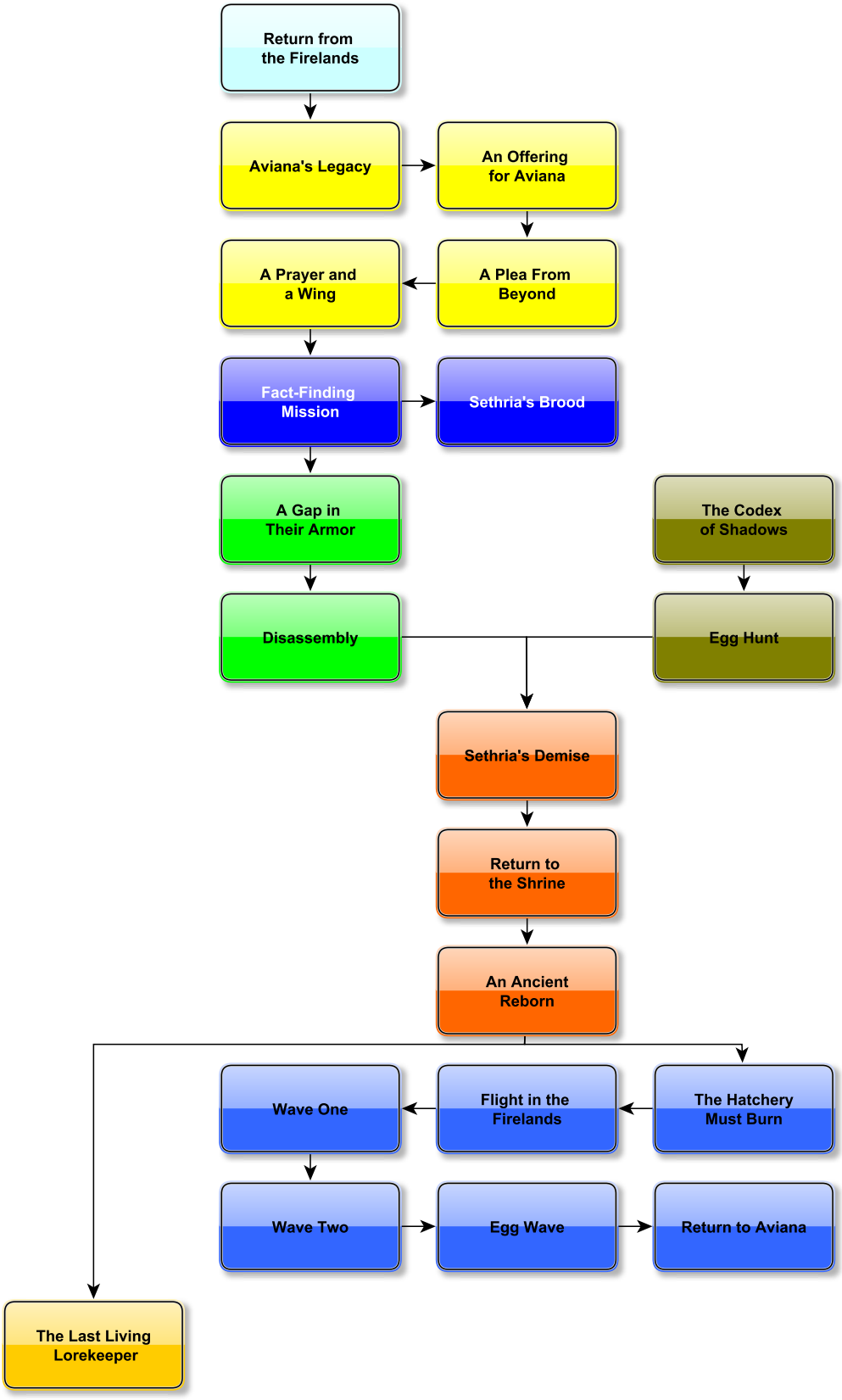

Figure 5.5: Part 3.2 of the Mount Hyjal quest system

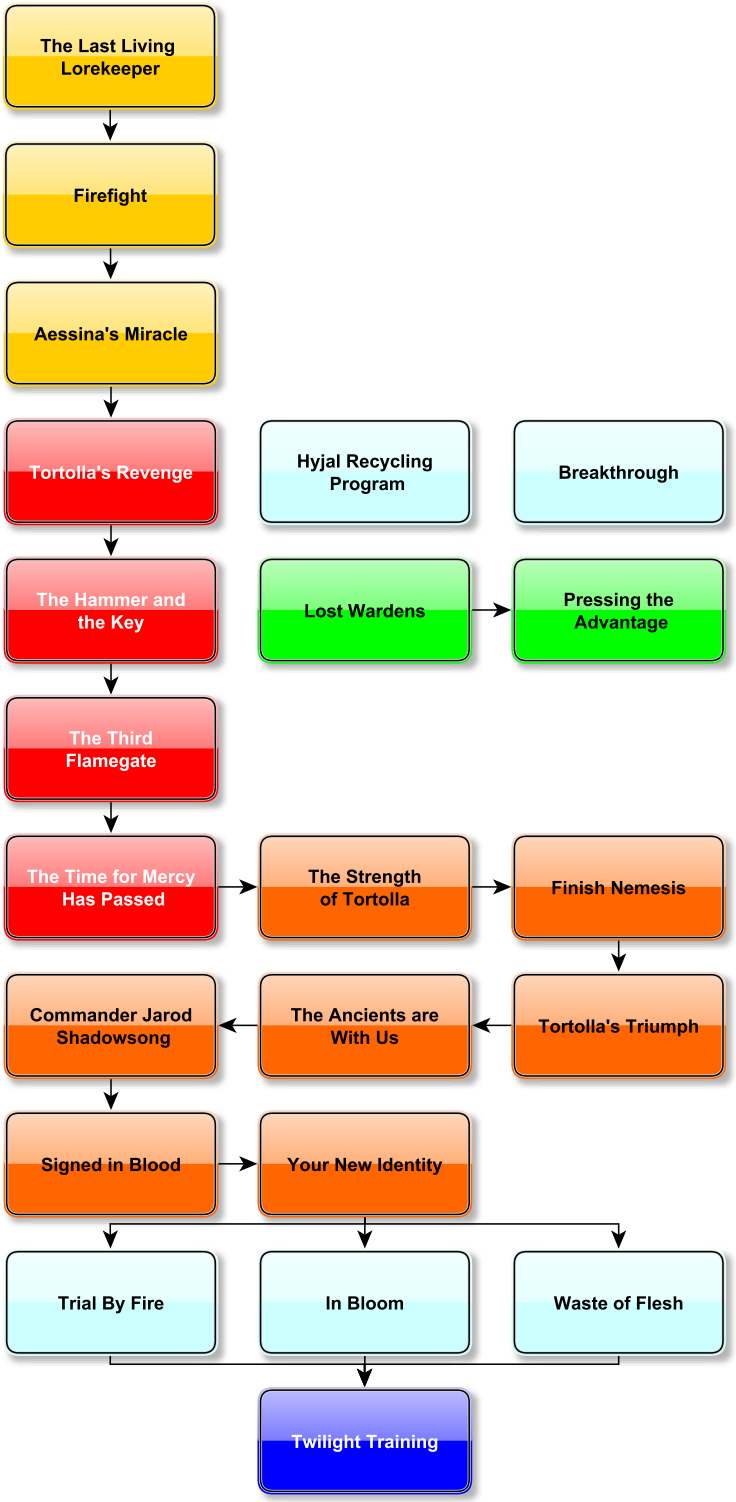

Figure 5.6: Part 4 of the Mount Hyjal quest system

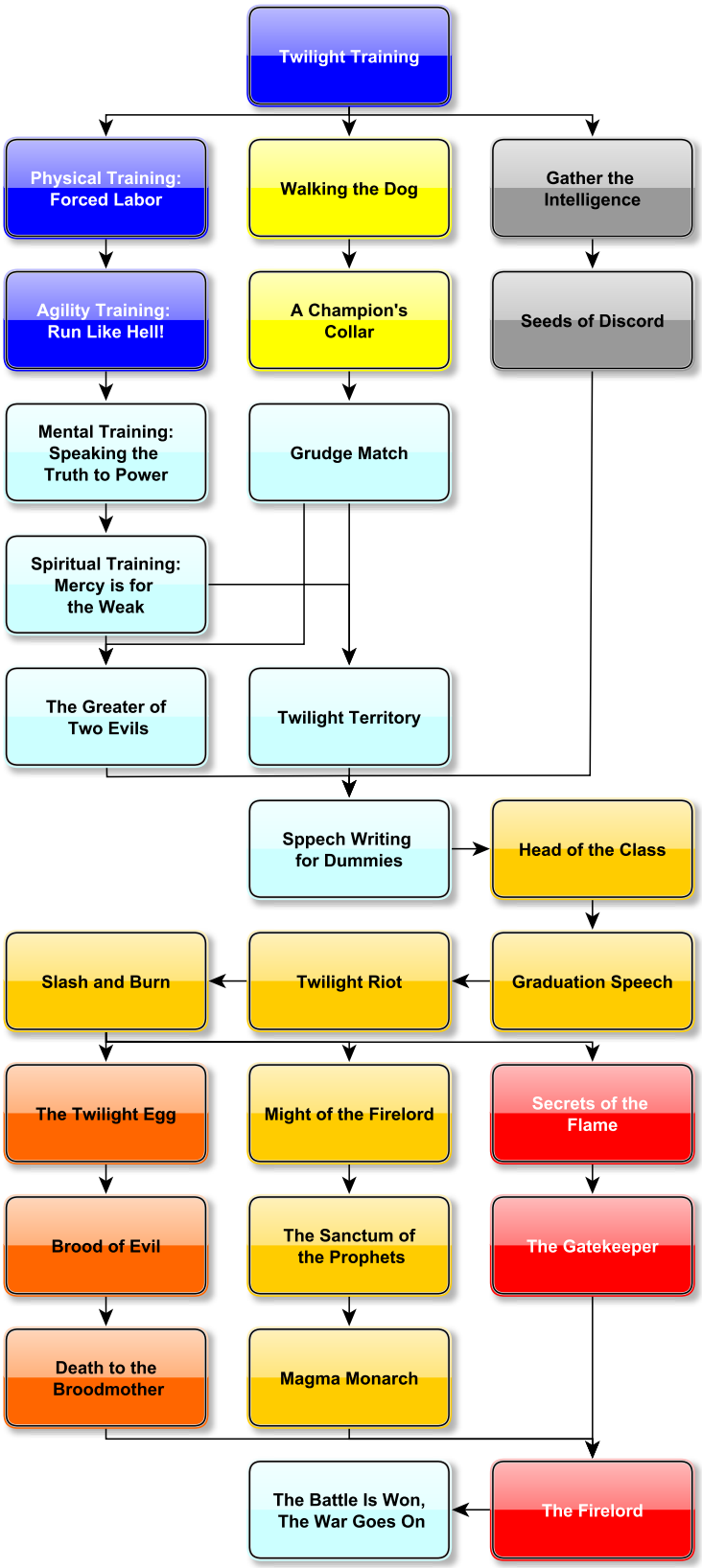

Figure 5.7: Part 5 of the Mount Hyjal quest system

5.2 Observations

In this section we will discuss the observations which we obtained from analyzing the quest graph. Again, we want to make it clear that these observations are based on the graph we reconstructed from several play sessions. Therefore, these observations might not reflect the actual game.

5.2.1 Quest Chains Are Not Well Defined

When we wrote the definition of a quest chain we expected it to be a universal definition. However, from the quest graph we can conclude that quest chains aren't well defined.

First and most importantly we expected that most quests would be part of a quest chain, which is not the case if we look at the graph. The white colored quests, which do not belong to any quest chain, occur relatively much in the quest graph. There are 26 occurrences of this type of quest on a total number of 136 quests.

Secondly, when we look at parts of the quest graph we see that there is a clear mismatch between the story told by the game designers and the structure of the quest chains which make op these stories. This could be an indication that the game designers didn't base the quest chain structure on the story. However, this would be strange because most of the quest chains in World of Warcraft are created around a story[1]. In addition we couldn't find another common factor, like for example geographical location, which is used to base the quest chain structure on.

Therefore, if we assume that the quest chains in World of Warcraft are based upon the story, we can observe two different "problems":

1. Some quests don't belong to a quest chain while they are part of a shared story.
2. Some quests are part of different quest chains while they share the same story.

Figure 5.8 shows two examples of the first problem. The blue quest chain consists of three quests according to the data obtained by wowhead. If we look at the titles of the quests, we observe that they are all "training"-quests. If we then look at Figure 5.8, we see that there are two white colored quests which are also "training"-quests. Furthermore, when we look at the story of both the blue and the white colored "training"-quests, we can see that they all tell parts of the same story[2]. Also note that all of the involved quests need to be completed at (or closely to) the same geographical location. Therefore, it would make more sense if the blue quest chain also included the two white colored "training"-quests. The same is true for the yellow quest chain. For this quest chain the player has to raise a pet hound. However, a later quest called "Grudge Match" asks the player to fight a raptor with the newly trained hound. Again, this is clearly part of the same story, so it would make sense to add the quest "Grudge Match" to the yellow quest chain.

Figure 5.9 shows an example of the second problem. The blue, green and orange quest chains are separate chains, but they all work to a common goal which is the "Demise of Sethria". Therefore, if we look at the story it would make more sense if the quest chains were merged into one quest chain. This would have no impact on the players experience, but it would make the quest system a bit more clear. However, it is up to the game designer to create the quest chains, so he might have a good reason for this structure. Unfortunately we couldn't find it.

As a result of the former two problems, we also notice that quest chains are rather short. In some cases this could occur because a story is broken in to several small quest chains, while in other cases the quest chains are

[1] This was observed in the play session and from playing the game for several years.
[2] The story for each quest can be found on wowhead.com.

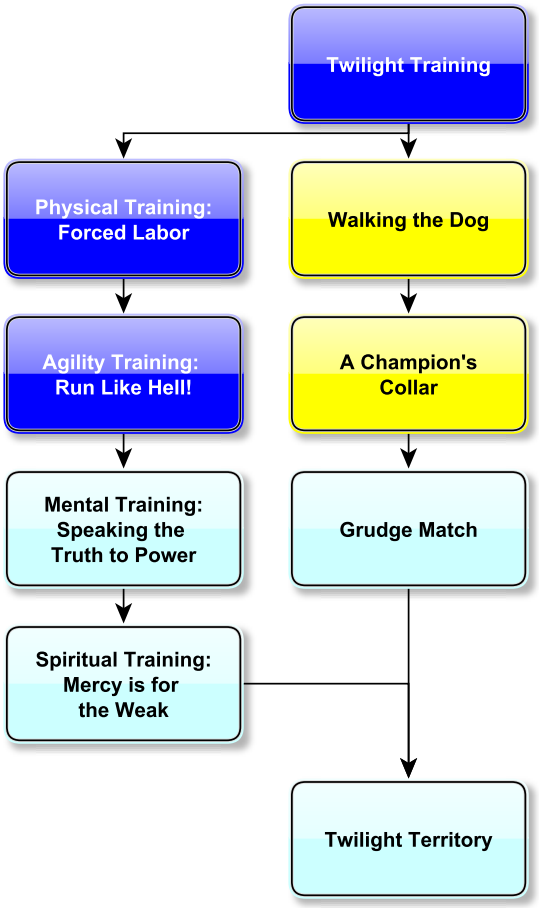

Figure 5.8: Incorrect mapping of story to quest chain

just only a few quests long. This makes us wonder why the quest chains are so short. Could it be that the quest chains would otherwise become to complex? Or could it be related to the players experience? Meaning that the short quest chains better capture the attention of the player than longer quest chains. At this point we find it hard to explain why Blizzard would use quest chains in this particular way.

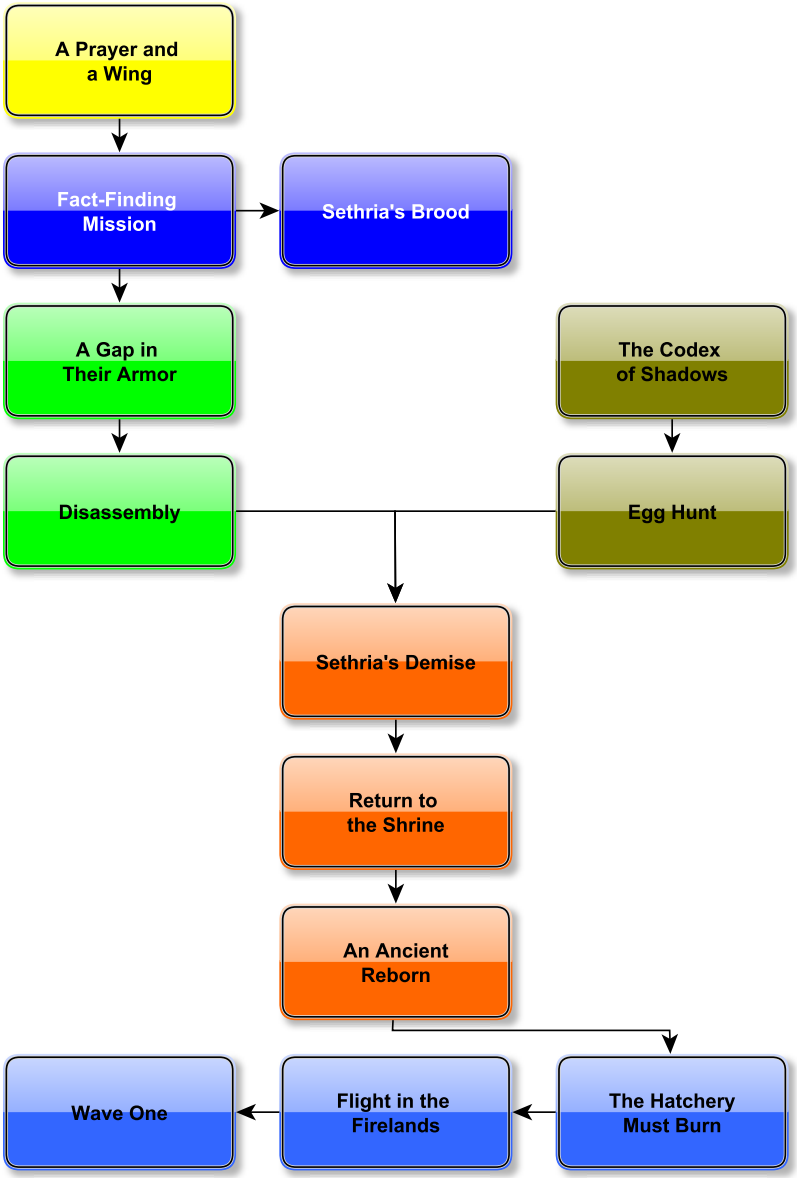

Figure 5.9: A story broken up into several small quest chains

5.2.2 Simplistic Quest Relationships

When we look at the quest structure we also notice that in World of Warcraft the "Quest Completed Prerequisite" and the "Level Prerequisite"[3] are the only prerequisites that are used. As a result of only using these prerequisites, a very straightforward quest system is created. We wonder if only these prerequisites were used in order to keep the quest system as simplistic as possible or that there are other motives for only using this limited set of prerequisites. It could be the case that the game designers at Blizzard would not approve of some of the types of prerequisites due to game play reasons. Another reason for not finding other quest relationships could be the limited scope of this case study. If this is the case, we can conclude that either diverse quest relationships are not a design goal of World of Warcraft or that they are just used very scarcely and in different parts of the world.

5.2.3 Incorrect Quest Order

During the case study we also stumbled onto one occurrence of a possible incorrect quest order. For the quest "Get Me Outta Here" the player needs to escort a NPC out of a cave. However when the player completes the quest "Elementary!", the hostile NPC's in the cave become neutral. Therefore, if the player choose to do complete the "Elementary!" quest first, he can escort the NPC out of the cave without a fight. While this can be a carefully considered scenario, it is more likely that the game designers didn't expect the player to use this quest order. While this type of problem doesn't limit the player in playing the game, it can break the experience of the game.

5.2.4 Layers

Previously we explained that layers can be used to divide the quest system in to separate groups of quests which share some kind of relationship. Initially we thought that we could use a layer to represent each geographical hub in the "Hyjal"-region of World of Warcraft. It turns out that there are around 15 to 20 hubs of this kind in the region. This is a big difference with what we expected, which was that there would be 4 to 5 main hubs. Due to the high number of hubs it makes no sense to create a layer for each hub, because it would seriously fragment the quest system and therefore make it harder to analyze it. We could however use the layers to map the relationships between the different regions in the game, but that is not really in the scope of this research. In addition we also observe that some quest chains go through different quest hubs. This conflicts with how we designed the layers, which is with the notion that a hub is set of complete quest chains. Which means that it is not possible that quest chains go through different instances of the same layer.

Therefore, we conclude that layers can in some cases simplify the quest system by breaking it up into several pieces, but that it will only work if we put constraints on how many hubs there will be in a region and on how the quest chains between these hubs interact. For World of Warcraft it wouldn't be a good idea to use a layer for the quest hubs.

5.3 Results

In the previous section we discussed the observations we made when we analyzed the quest graph of World of Warcraft. In this section we will use algorithms explained in section 3.3 on that quest graph in order to obtain more information. First we will look at the occurrence of redundant links. Then we will discuss the mandatory

[3] All quest have a minimum needed player level. The minimal needed level for a quest can be found on wowhead.

quest ratio which serves as an indication for how much freedom a player has when it comes to choosing which quest he may or may not want to do. And we will conclude by discussing the freedom of the player when it comes to choosing a quest order.

5.3.1 Redundant Links

Due to the simple quest system we expected not to find any redundant links. However, we managed to find one for the quest "The Last Living Lorekeeper". Figure 5.10 shows this redundant link. As we mentioned before, redundant links don't have to indicate a failure in quest design. It could be that this dependency exists for a very specific reason. We only mention this in order to show that redundant links do exist in current day quest systems.

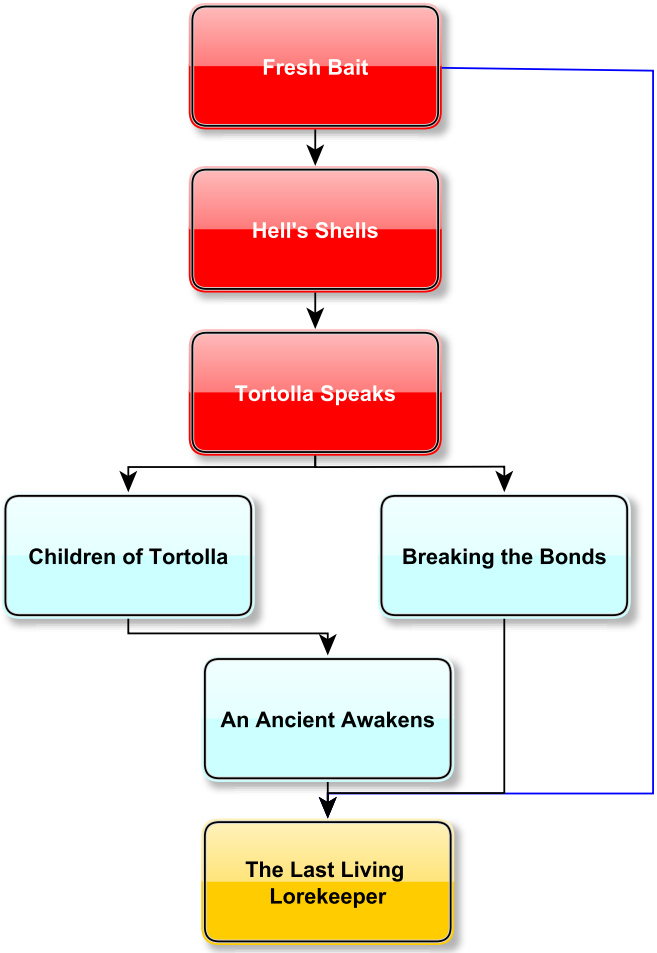

Figure 5.10: A redundant link in the quest chain

5.3.2 Mandatory Quest Ratio

From a game like World of Warcraft a player might expect that he has an enormous amount of freedom when it comes to choosing quests. However, if we look at the mandatory quest ratio for the "Hyjal"-region of the game,

we can only conclude that this is not the case. The shortest path from the first quest ("Hero's Call: Mount Hyjal!") to the last quest in this region ("The Battle Is Won, The War Goes On") contains 99 quests. The longest path for the same starting and end points contains 136 quests. As a result the mandatory quest ratio has a value of:

$$0.73 \approx \frac{99}{136}$$

The value of 0.73 indicates that 73% of the quests are mandatory in order to complete the main storyline of this area. Which to us is a clear indication that the player doesn't have much freedom when it comes to choosing the quests he wants to complete.

5.3.3 Quest Orders

In the previous section we concluded that a player does not have much freedom when it comes to choosing which quests he wants to complete. In this section we will analyze the possible orders in which the player can choose to complete the quests. First we will look at the orders with respect to the shortest path and then we will do the same for the longest path.

As mentioned before the shortest path contains 99 quests which need to be completed in order to complete the main story in "Hyjal". Figure 5.11 shows the graph with the total intervals for the quests on this path. Because the shortest path does not contain any optional quests, it shows the freedom of the player with respect to the main quest line very well.

From the figure we can conclude that at the first 15 steps the player does not have much freedom. For steps 15 to 55 the player gains an enormous amount of freedom, which can be explained by the fact that the player can complete parts 3.1 and 3.2 of the quest graph at the same time. This does imply that the player will need to do a quit high amount of travelling in order to obtain this freedom because the two parts need to be completed in different area's of the map. Then from step 55 to step 75, the player has no freedom. This is the part of the quest graph from the quest "The Last Living Lorekeeper" to "Your New Identity", which is basically one long chain of quests. After which the player gets a bit of freedom from steps 75 to steps 85, which is again lost at steps 85 to 90. For the last 9 quests the player gets some amount of freedom, which can be explained by the 3 quest chains that can be completed at the same time in order to reach the quest "The Firelord".

If we now compare Figure 5.12 for the chart of longest path with the chart of the shortest path, we will see quite a difference. The player will now complete all optional quests, which can be intertwined at all possible steps of the main quest chain. As a result, this effectively raises the freedom at each point in the chart and that in it turn results in a massive amount of freedom for the player. However, please note that much travelling can be needed in order to obtain this amount of freedom. For example, first performing all optional quests will let the player explore the whole map before starting the main quest chain. This will probably result in more travelling than if the player would do the optional quests when he encounters them during other quests. However, it is up to the player to decide how he wants to play the game and therefore this is a valid scenario.

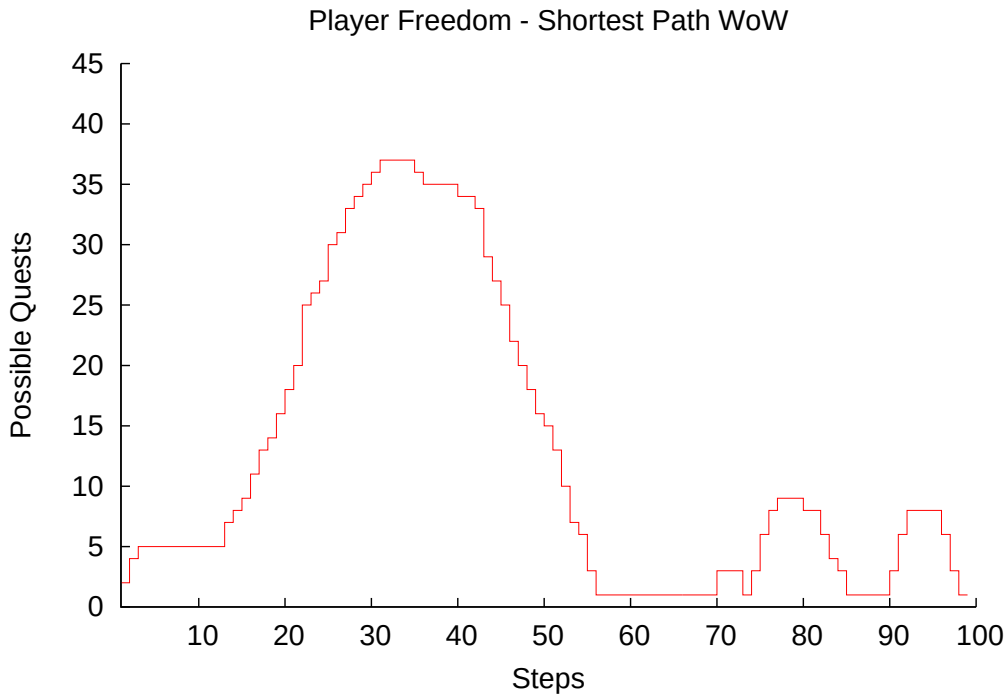

Figure 5.11: Freedom at the shortest path

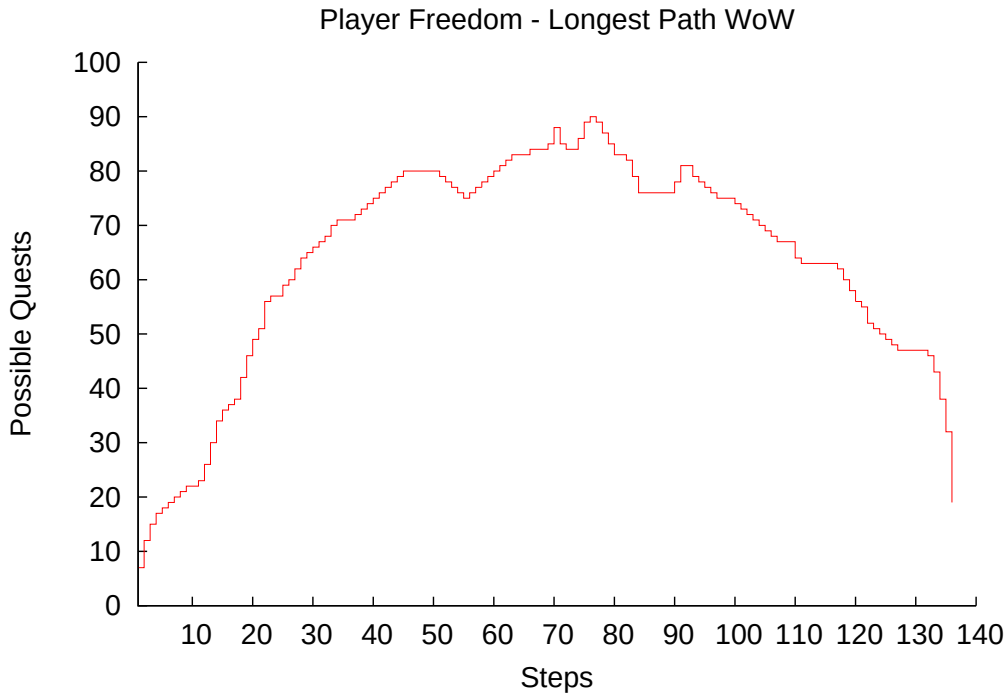

Figure 5.12: Freedom at the longest path

Part II

Improving Variation in a Quest Graph

Chapter 6

Introduction

In the first part of the paper we explained how a quest graph can be used to retrieve information from a quest system. This includes information about the freedom that a player has with respect to choosing which quests he wants to complete and in which order he wants to complete them. From our case study we concluded that quest systems can be very static. In case of World of Warcraft this meant that the player doesn't really have a choice when it comes to deciding which quests he wants to complete. This was due to the high amount of mandatory quests, which leads to the player getting stuck if he skipped one of those quests. Furthermore, the player did not have much influence on the order in which he can complete the quests. While these problems are interesting, they are the result of how game designer created a quest system. The problems can be diminished by adding more optional content to the game or making the main quest line shorter, therefore making less quests mandatory. In this part of the paper we will look at a different way to increase the freedom of the player.

Some game designers make a game more interesting by adding choices to the game. While choices don't give the player more influence on the order in which he completes quests, it does give the player a certain amount of control over how he wants to play the game. Most games base choices on the concept of "good versus evil". An example of such a game is Star Wars: The Old Republic. In that game performing only evil deeds will lead you down a slightly different path than only performing good deeds. Furthermore, based on the alignment of the player, some weapons may or may not be used.

If such a quest system is constructed well, it can help emerge the player into the game. Furthermore, using choices provides the game with a higher replay value because different choices lead to different game experiences. Obviously if the player would make the exact same choices, the experience with the game would be the same.

In this part of the paper we will look at how we can add choices to the quest system in order to make the game more versatile. We also propose to use "dynamic paths"[1] to provide the game with even more variation. While we don't discuss dynamic paths, it is very reasonable to assume that this construct will work. This is because the only difference between dynamic paths and choices is that choices will be decided by the player and that dynamic paths will be decided by the game engine. Therefore, in the remainder of the paper we will only discuss player-based choices because these provide the greatest amount of variation.

In part one of the paper we have explained that dependency links can either be blocking or non-blocking. We also stated that we treat all links as blocking because that makes it easier to handle the nodes of the graph. We explained that this imposes a slight error in the algorithms because non-blocking dependency links must be accepted at a given time but can be completed at any given time. So in fact this limitation, only using blocking links, enables us to treat the acceptance and completion of the quest as being the same thing. If a quest is accepted it will be immediately completed. In this part of the paper we still hold on to this assumption.

[1] Dynamic paths basically are choices which are made by the game engine based upon some kind of metric. An example of such choice could be one that gives the player different quests depending on the time of day. Another example of a dynamic path is one which gives a quest based upon the state of a region in the game. So if a region is in war, the player gets quests that will involve him in the war. But if the region is at peace then the player would get a different quest.

Furthermore, in this part of the paper we will ignore the fact that a quest can belong to a quest chain. From the case study we concluded that it is not evidently clear that quest chains are an integral part of a quest system. The absence of quest chains does not have any consequences for the quest graph in this part of the paper. The only algorithm that actually depends on quest chains is the "atomization"-algorithm. However, later on in the paper we will explain why atomization is considered outside the scope of this part of the research.

Finally two other concepts that are mentioned in the first part of the paper are disregarded in this part. We will ignore the fact that a quest system can be divided by using layers, because it is of no consequence for the quest graph. Furthermore we will also ignore the fact that a graph can have multiple connected components. While it still is important to consider multiple connected components, it actually has no real impact on the algorithms defined in this part of the paper. This is because a graph with multiple connected components is in fact just a set of graphs that is contained within one single graph. In order to keep things clear and simple we have chosen to only work on a single graph. Making the algorithms work on graphs with multiple connected component is rather straightforward.

In the following sections we will first explain the characteristics which we consider important when adding choices to the quest system. Then we will explain three different approaches to extending the quest graph to support choices. From these approaches we will select the approach which we consider to be the most appropriate. Finally, we will flesh out the details for that approach by explaining the algorithms used on this new version of the quest graph.

Chapter 7

Important Choice Characteristics

In this section we describe the characteristics we deem important for embedding choices in the quest graph. Some of these characteristics will have influence on gameplay, while others are more focused on the usability of the quest graph. We will globally explain each of these characteristic and the influence they have on choices. While the characteristics themselves are not complex, using them in conjunction with each other is.

Please note that throughout the remainder of the paper the term "merge point" is used. A merge point is a node in the graph in which two or more options of a choice converge. In the upcoming sections these nodes are often labeled with the name "Merge". However, not all merge points are labeled as such.

Furthermore, in order to prevent confusion, the quest graphs that do not represent any of the later discussed approaches are visualized with a grey colour. All graphs that are coloured differently are either related to the quest graph of part one or to an approach mentioned in this part of the paper.

7.1 Explicit Choices

One important characteristic to consider when defining the new version of the quest graph is the way in which we express the choices in the quest graph. There are two possible approaches for this problem:

1. Implicitly
   This can be accomplished by re-using prerequisites in a creative way. Figure 7.1 shows an example of an implicit choice. The prerequisite-approach discussed in chapter 8.1 will use prerequisites to implicitly define choice graphs.

2. Explicitly
   In order to explicitly express choices in a quest graph, we need new elements which will represent choices and/or options. Figure 7.2 shows a possible approach for explicitly defining choices by using new "Choice" and "Option" elements. In the chapter 8 the component and element approach will show different structures to explicitly define choices.

In our opinion the implicit approach is not a very suitable approach for adding choices and options to the quest graph. This is because it leans heavily on prerequisites, which become increasingly complex as choices become more complex. Furthermore, using prerequisites means that algorithms need to be able to detect choices from them. This would result in more preprocessing when for example the quest graph needs to be visualized. In order to handle the graph easily it should be transformed into an explicit form. Therefore it is better to just

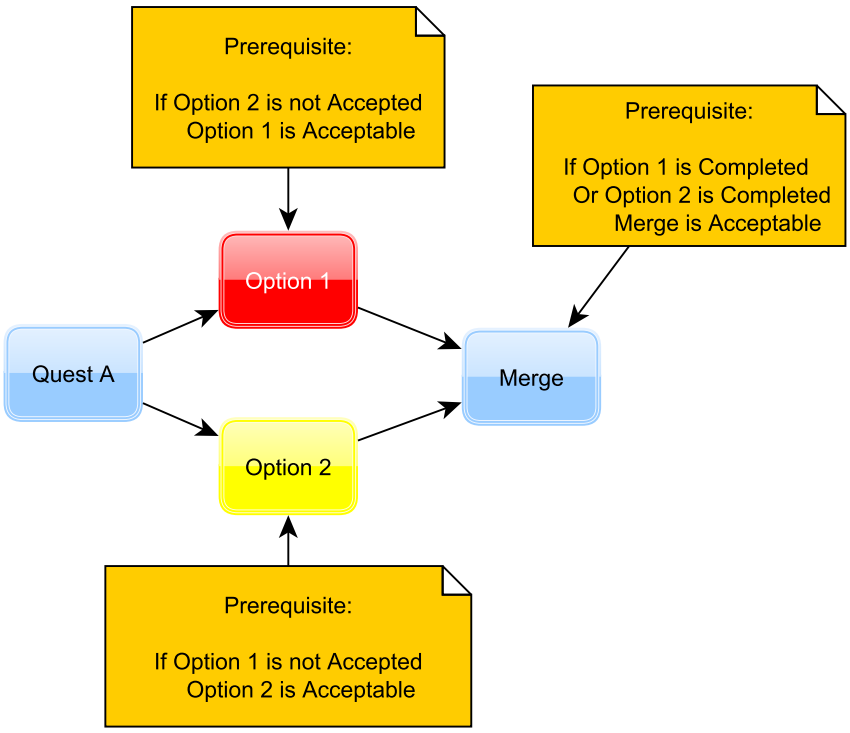

Figure 7.1: Example of a choice created with the elements of a normal quest graph

use an explicit form, which will prevent the need for the transformation. Implicit choices also have an upside, which is that the current quest graph can be used without any elements needing to change.

The explicit approach does not suffer from the previously mentioned problems and it can make choices less complex because limitations can be applied on the structure of the choice (and options). For example, it is wise to create a rule that states that an option can only have one in edge, which is a choice. A downside to using explicit choices is that the quest graph needs to handle new nodes, in the form of for example an "choice" or "option" node. In our opinion the benefits of using an explicit approach outweigh the downsides.

At this point it might seem that we have decided that an explicit approach will be used in order to add choices to the quest system. However, this is not the case. The goal of this section is to explain the differences between an explicit and implicit approach. Later on in chapter 9 we will select one of three approaches, which all enable different sets of characteristics.

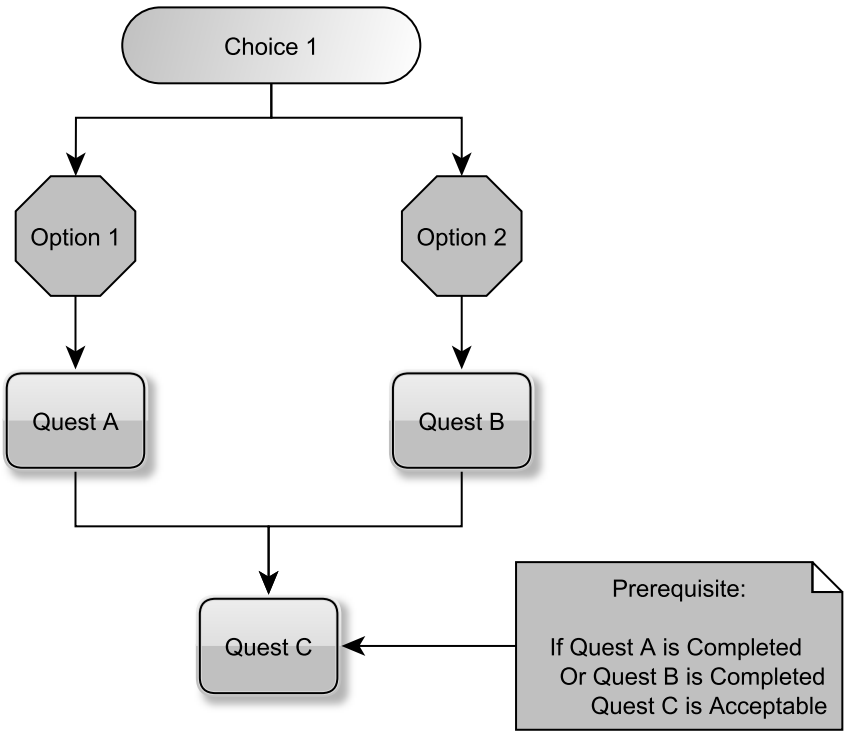

Figure 7.2: Example of a possible approach for using explicit elements

7.2 Consequences

In real life making a choice can be hard because it might have serious consequences. In some games however, you can slaughter entire villages without having to face the consequences of your actions. In our opinion it is useful to consider the possibilities of adding consequences to a choice. The most obvious and implicit consequence of a choice is that a player will take a different path through the quest graph. However other consequences might be added to the game to make the gameplay more interesting. A good example can be found in Star Wars: The Old Republic, in which the player has to make choices about good and evil. Certain choices will provide the player with light points, while others will give the player dark points. Based upon the alignment of the player, good or evil, some weapons may or may not be used.

In the previously mentioned example the consequences for being dark or light are not that severe, because for every dark weapon there is a light equivalent. A more interesting way of using consequences is to influence the game state with the combined choices of multiple players. For example it is possible to let the choices of the players influence the weather in an important region of the game. If the players dominantly make evil choices, the weather could be dark with rain and thunder. However, if the players dominantly make good choices the sky could be clear and the sun could be shining.

7.3 Altering Previous Choices

In some situations it might be desirable that a player can alter his previous choice. This can be especially useful for choices that are based on a moral dilemma of which the intent is not immediately clear. For example the

player can be okay with an “evil” choice that starts with terrorizing a village in order to make the villagers leave the village. When the quest is completed, the villagers will stay because they will stand their ground. The following quest will then ask the player to kill the entire village. The player might then decide that this was not his intention and he might decide to repent.

Figure 7.3 shows a quest system diagram which enables a player to alter his previous choice. If the player chooses the “evil” path, he can repent after one quest “Quest B” or continue on the path of “evil” by choosing not to repent.

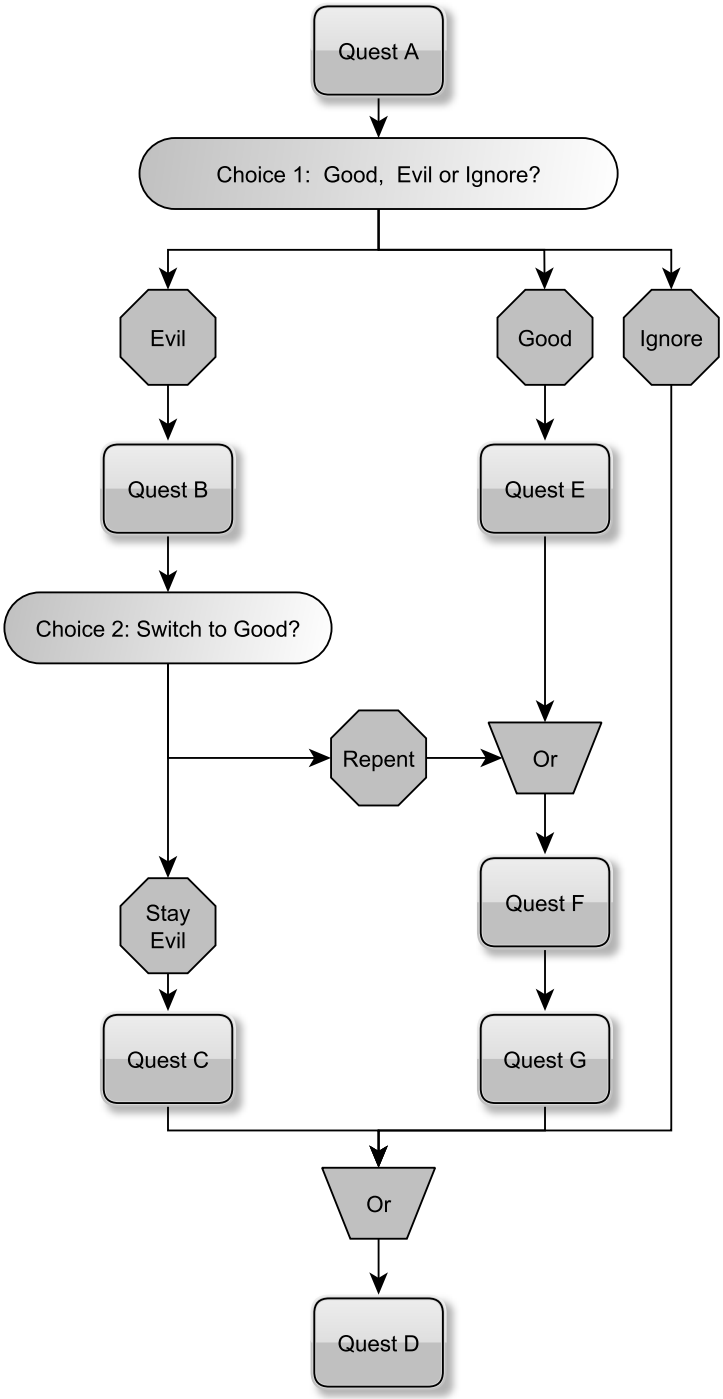

Figure 7.3: Example of a choice which allows a user to skip a choice or to change his previous choice

7.4 Skipping a Choice

In most cases making a choice means that the player will be lead down a different path. In our opinion the player should also be able to choose an option that indicates that he does not want to make the choice. This would mean that the player would skip the entire content of that choice and he would immediately be finished with that choice. Such an option would be perfect for increasing the freedom of player. The player can now choose not to complete the choice, but he can continue with the game.

An example of this concept is shown in figure 7.3. If a player chooses the “Ignore”-option, he decides not to complete any content of this choice and he continues with the rest of the game.

7.5 Skipping a Part of the Quest Graph

A more extreme version of skipping a choice is “skipping a part of the quest graph”. In some situations it could be desirable that a player can skip more than a single choice. For example if the player wants to skip a part of an area in the game such a feature would be necessary. Without this feature the player could potentially end up stuck.

Figure 7.4 shows an example of this concept. If the player choices the “Ignore”-option at “Choice 1”, he will immediately skip the content of “Choice 1” and part of the content of “Choice 2”. The player continues with ”Quest F”, which is still part of the second choice. This shows that skipping over a part of the quest graph can result in a merge point in the middle of another choice. This is in contrast with the normal situation in which an option will end in the final merge point of a choice which in this case would have been ”Quest D”. Consequently, this means that everything that goes over the boundary of a single choice falls in this category.

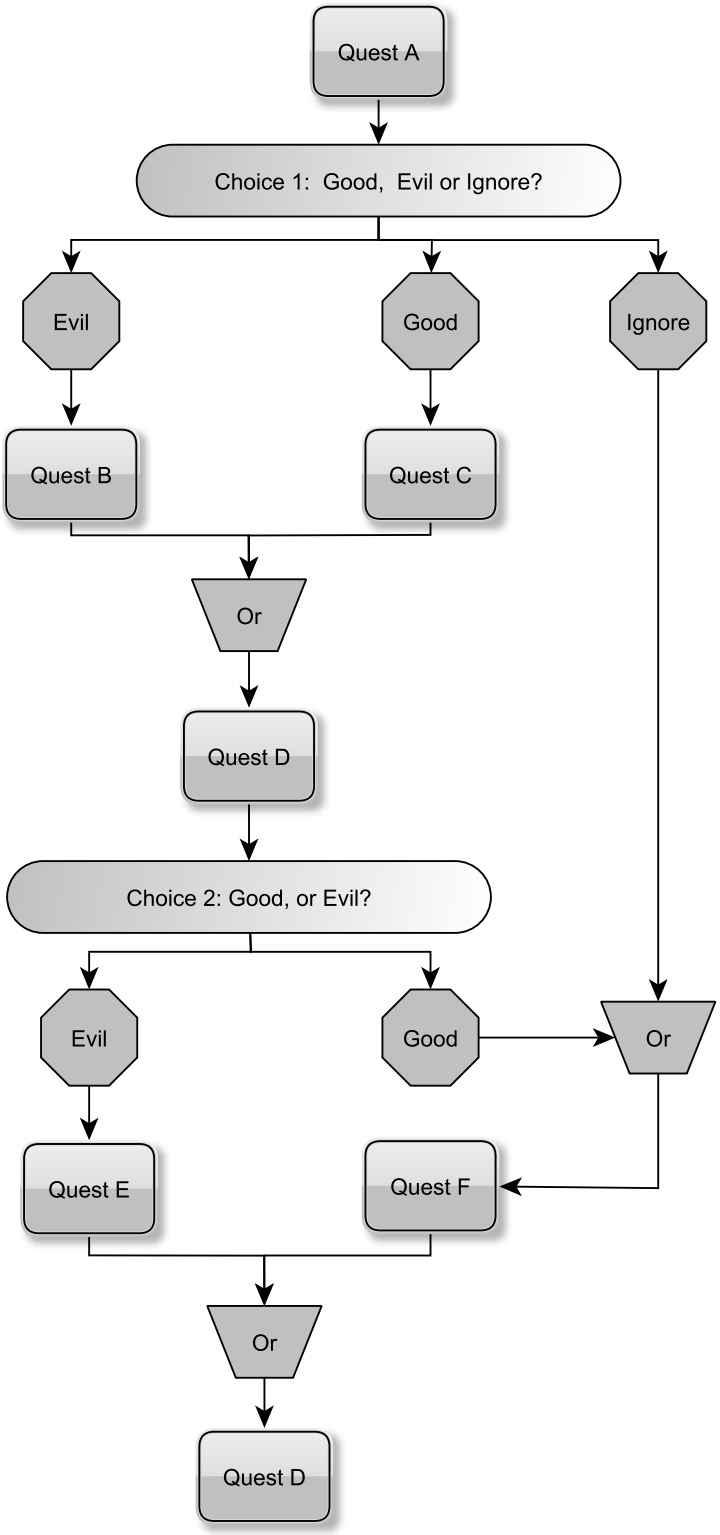

Figure 7.4: Example of a choice which allows a user to skip a part of the quest graph

7.6 Complexity

Some of the previously explained characteristics make the structure of a quest graph considerably more complex. Especially the usage of the "Skip a part of the quest graph"-characteristic will increase the complexity of the graph by allowing the game designer to cross the option and choice boundaries. Therefore, it won't be very surprising if we state that a quest graph will get even more complex if a game designer can combine two or more of these characteristics.

Because a picture says more than a thousand words, we show a quest graph that can be created with the previously explained characteristics in Figure 7.5. It is possible to argue over the validity of this contraption, but that itself is an indication of the complexity of the quest graph. Please note that we left out the prerequisite relationship types, "and" and "or", on purpose.

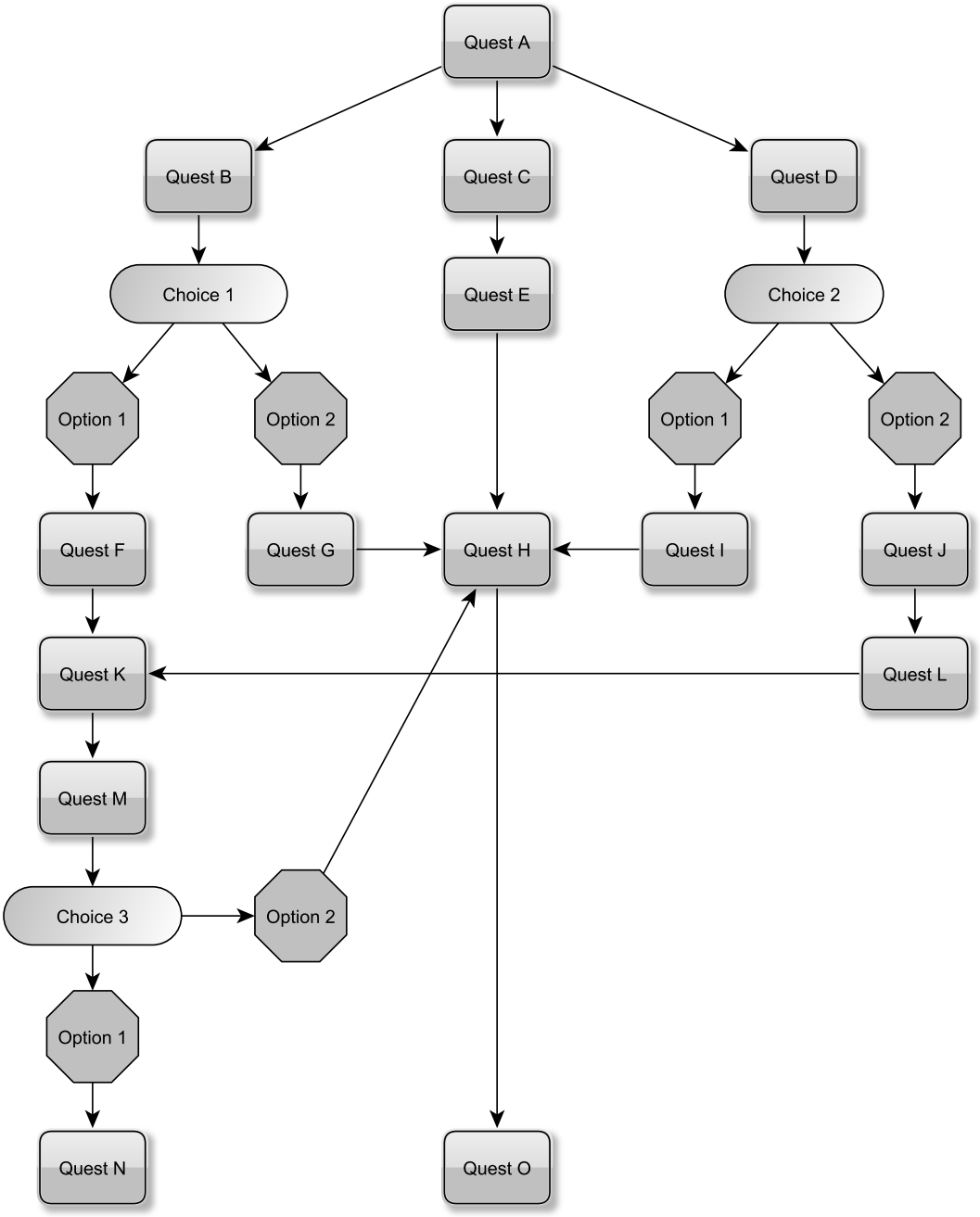

Figure 7.5: Complex quest graph with choices

Chapter 8

Different Approaches To Choices

The first step towards adding choices to the quest graph is to look at different approaches for integrating them. It is important to first explore the possibilities and limitations of several approaches before changing the quest graph. For some situations one approach might be easier to use than others. In chapter 9 the most suitable approach will be selected and it will be used in the remainder of this paper.

8.1 Using Prerequisites

With the prerequisites mentioned in the first part of this paper it is already possible to incorporate choices into the quest graph. This can be accomplished by creating rather complex prerequisite-expressions. Each expression checks the state of each option for the following: "if all other options for this choice aren't accepted, this option is acceptable". An example of this approach can be seen in Figure 8.1 where "Option 1" and "Option 2" are options for a choice. In this case the choice is located at the end of "Quest A". After choosing and completing one of the two options, the last quest of that option will fulfil the dependency for the merge quest. So in this case that means that the choice merges in the "Merge"-quest because it only need one of the two options in order for the quest to become acceptable.

This approach looks promising because it is built upon the basic building blocks that are used in a normal quest graph. However, there are also several limitations with this approach which make it undesirable for usage in a game. The following sections explain the characteristics and limitations of this approach.

Characteristics
The usability of this approach is assessed by the limitations on the previously explained characteristics. Table 8.1 shows that there are two limitations when using this approach. The first limitation is that choices are expressed implicitly and the second limitation is that it is not possible to add consequences to choices.

Characteristic	**Possible**
Explicit choice	No
Consequences	No
Skipping a choice	Yes
Skipping a part of the quest graph	Yes
Altering previous choice	Yes
Multiple start quests	Yes, but rather complex

Table 8.1: Characteristics of the prerequisite based approach

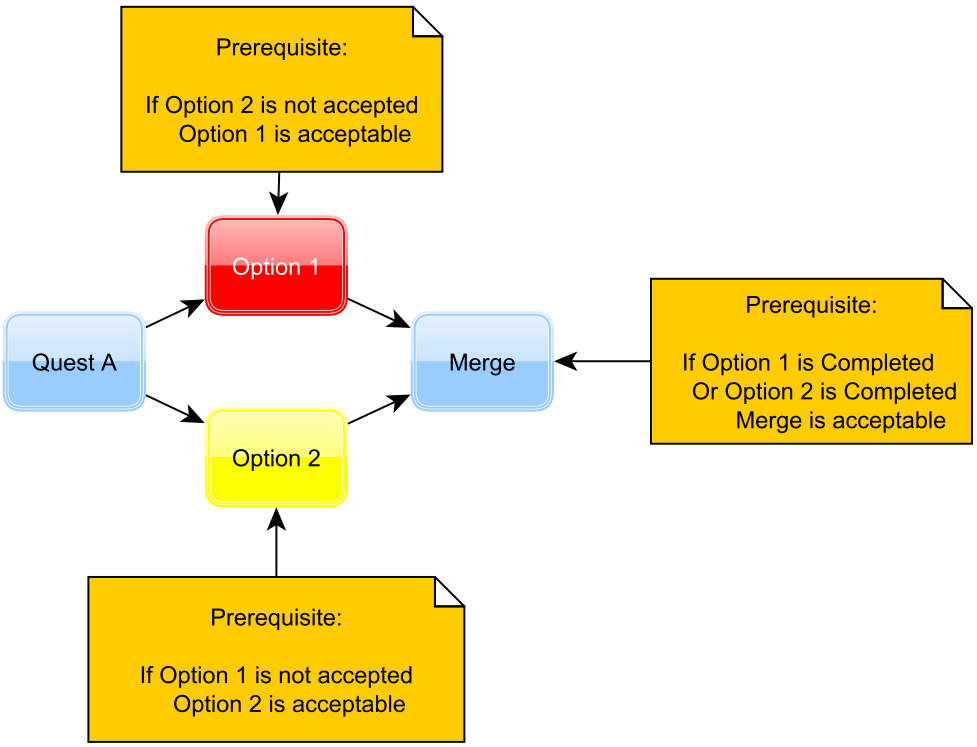

Figure 8.1: Example of creating choices via existing prerequisites

The next section will explain these limitations in more detail.

Explanation of Limitations
The first limitation, which indicates that choices are not explicit, is self-evident. As a result of this limitation it is not possible to add consequences to choices or options.

Therefore, the second limitation is the inability to use consequences. As a result of not having explicit choices and options, there is no suitable place to "put" these consequences. Off course it could be tempting to add the consequences to the first quest of each option. However, that will only work if the first quest of each option is not on the path of the other options, which is the case when an option points to the merge-point. This is the case when the player is provided with the possibility to skip a choice. As a result the consequences for skipping the choice will then be placed at the merge point. This leads to the problem that choosing another option will also result in facing these consequences. Figure 8.2 below shows an example of this scenario, in which the player would get two consequences instead of one if he chooses the first option. Please note that the merge node is the secondary option.

The limitation of not being able to add consequences to a choice will also effect some of the other characteristics, which are:

1. Skipping a part of the quest graph
2. Skipping a choice
3. Altering a previous choice

In some situations it might be viable to not have any consequences, but it would limit the usage of choices to only affect which quests a player can complete. However, the usage of consequences can lead to more creative

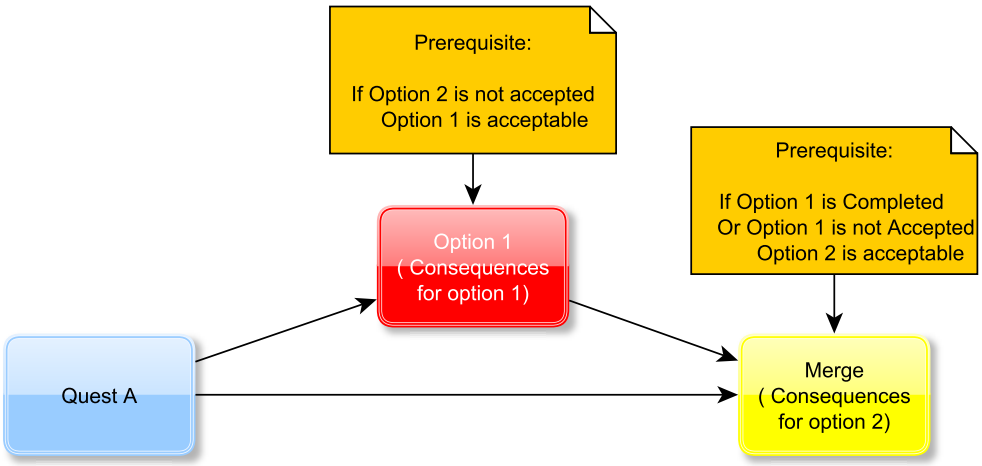

Figure 8.2: Example of naively adding consequences to the prerequisite based approach

scenarios.

A different limitation is that it will be "complex" to let a choice start with multiple quests. Figure 8.3 shows that it is hard, but not impossible, to have multiple start quests. It also shows that it is needed to add prerequisites to the "Option 1.1"-quest, which are almost the same prerequisites as for "Option 1". Also note that the prerequisites for the quest "Option 2" have become a bit more complex, because it is now needed to check if both of the starting quests of the first option haven't been accepted. In this case there are only two options, but it is possible to add many more options to a choice. In such a scenario the prerequisites will get increasingly complicated.

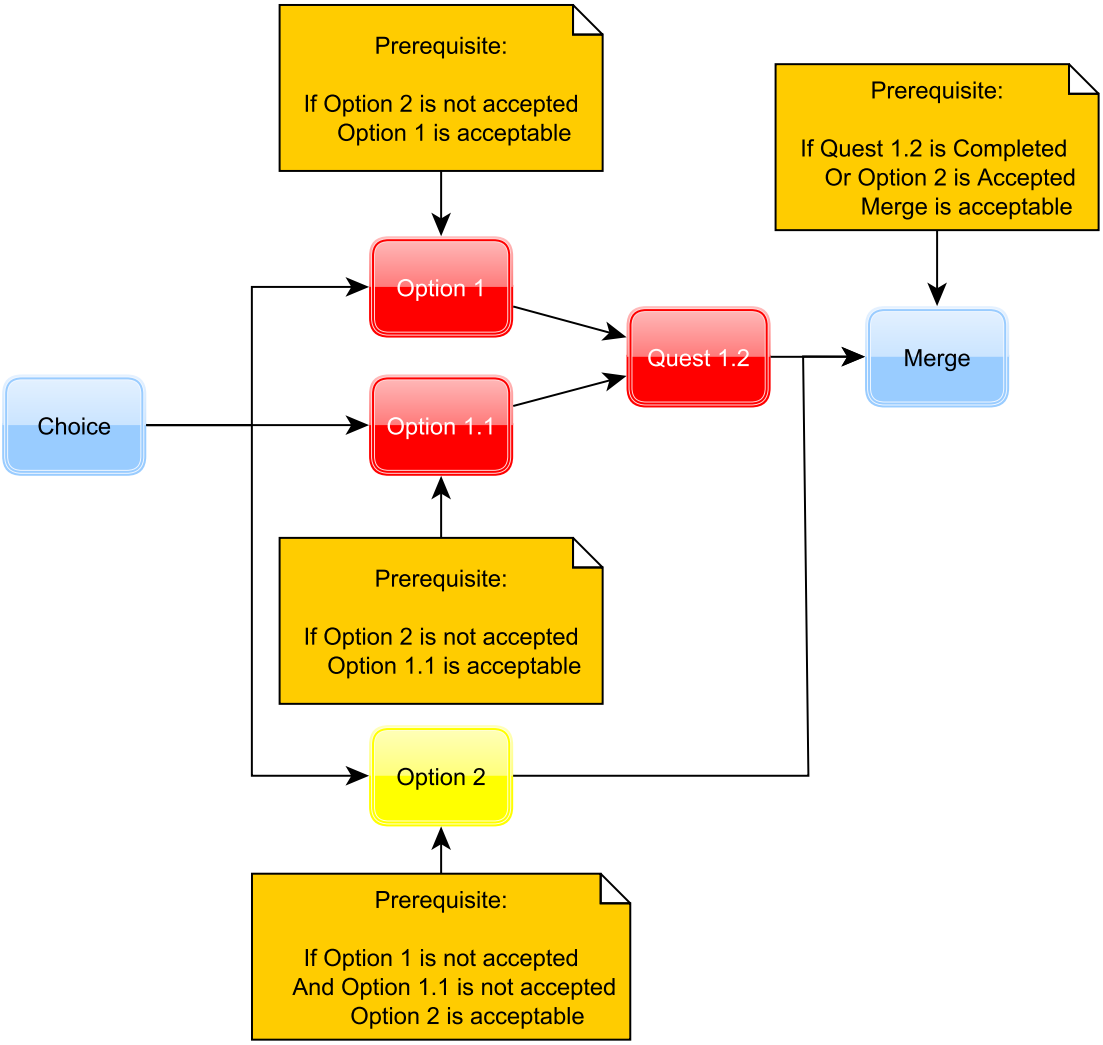

Figure 8.3: Example of complex prerequisites that enable multiple start quests

8.2 Using Components

Another approach to adding choices to a quest graph is by using components. For this approach we introduce a single new element, called a choice-component, which will encapsulate an entire choice. Furthermore, if a choice has any sub-choices they too will be contained within that same component. With this approach we need a new prerequisite called the "choice completed"-prerequisite, in order to be able to connect a choice component with a quest. Figure 8.4 shows an example of a component that contains a choice which can be made between quests A and B.

Characteristics
Table 8.2 shows which characteristics can be obtained if the component approach is used to extend the quest graph with choices. The next section will explain the limitations in more detail.

Limitations
The most important limitation with the component approach is that it limits its use to the boundaries of the component. This means that it is not possible to "skip a part of the quest graph" and that some scenarios for

Characteristic	Possible
Explicit choice	Yes
Consequences	Yes
Skipping a choice	Yes
Skipping a part of the quest graph	No
Altering previous choice	No
Multiple start quests	Yes

Table 8.2: Characteristics of the component based approach

"altering previous choices" are not possible. This shows that the encapsulation of a choice is a doubly edged sword. On one hand it nicely hides the complexities of the choice, but on the other hand it will limit the possibilities of a game designer when using these components. Figure 8.5 shows an example of a nested choice, which will let the user alter his previous choice. In this case it is not possible to implement this choice structure by using components, because it violates the boundaries of components. Quest F needs to be present in both components which is not valid.

However, in some cases it is possible to diminish this problem by dividing the choice into multiple components. But even then the result is sub-optimal. First of all, chaining choices will lead to constructions which allow the player to alter the choice for all previous options. An example of this situation can be seen in Figure 8.6 which shows us a chain of two components. The first component will let the player make the choice between good and evil, while the second component will let the player decide if he wants to alter his previous choice. In this case the game designer will have no way to provide the player with only the possibility to switch from bad to good.

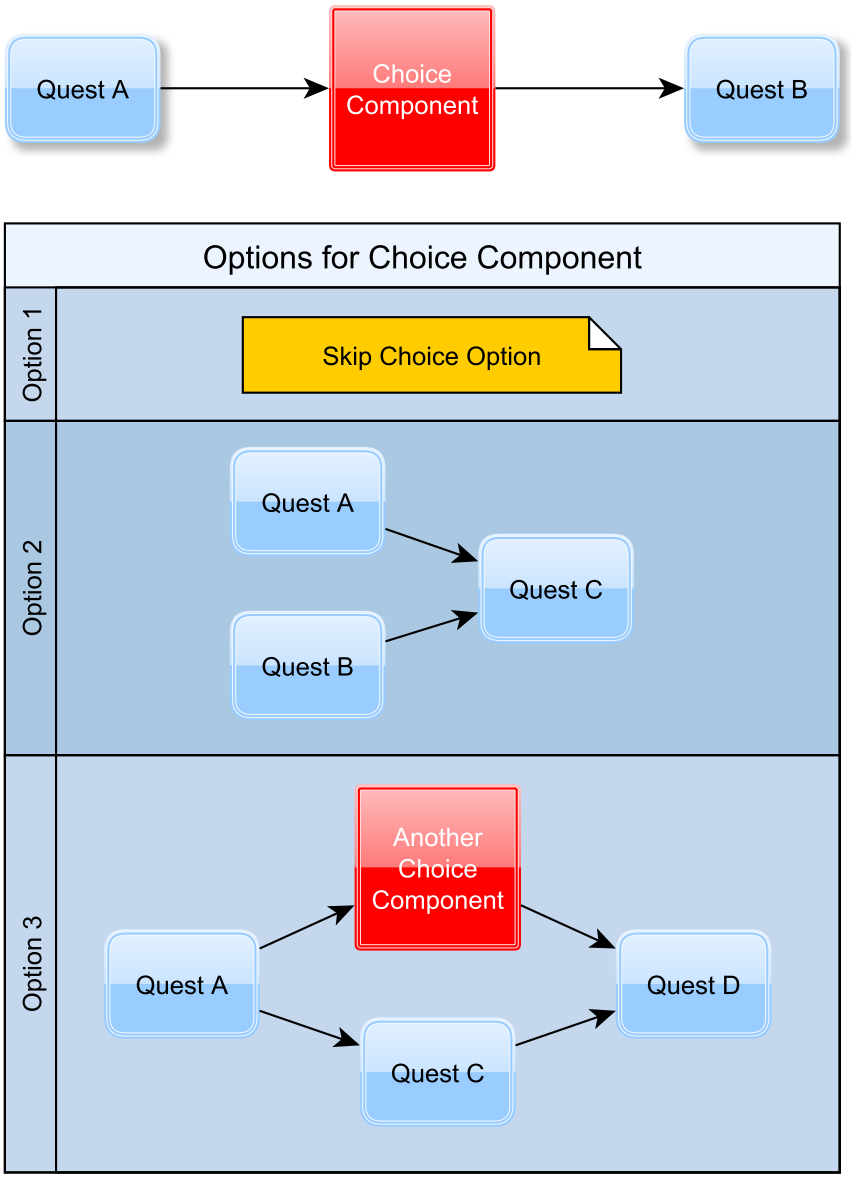

Figure 8.4: Example of adding choices via components

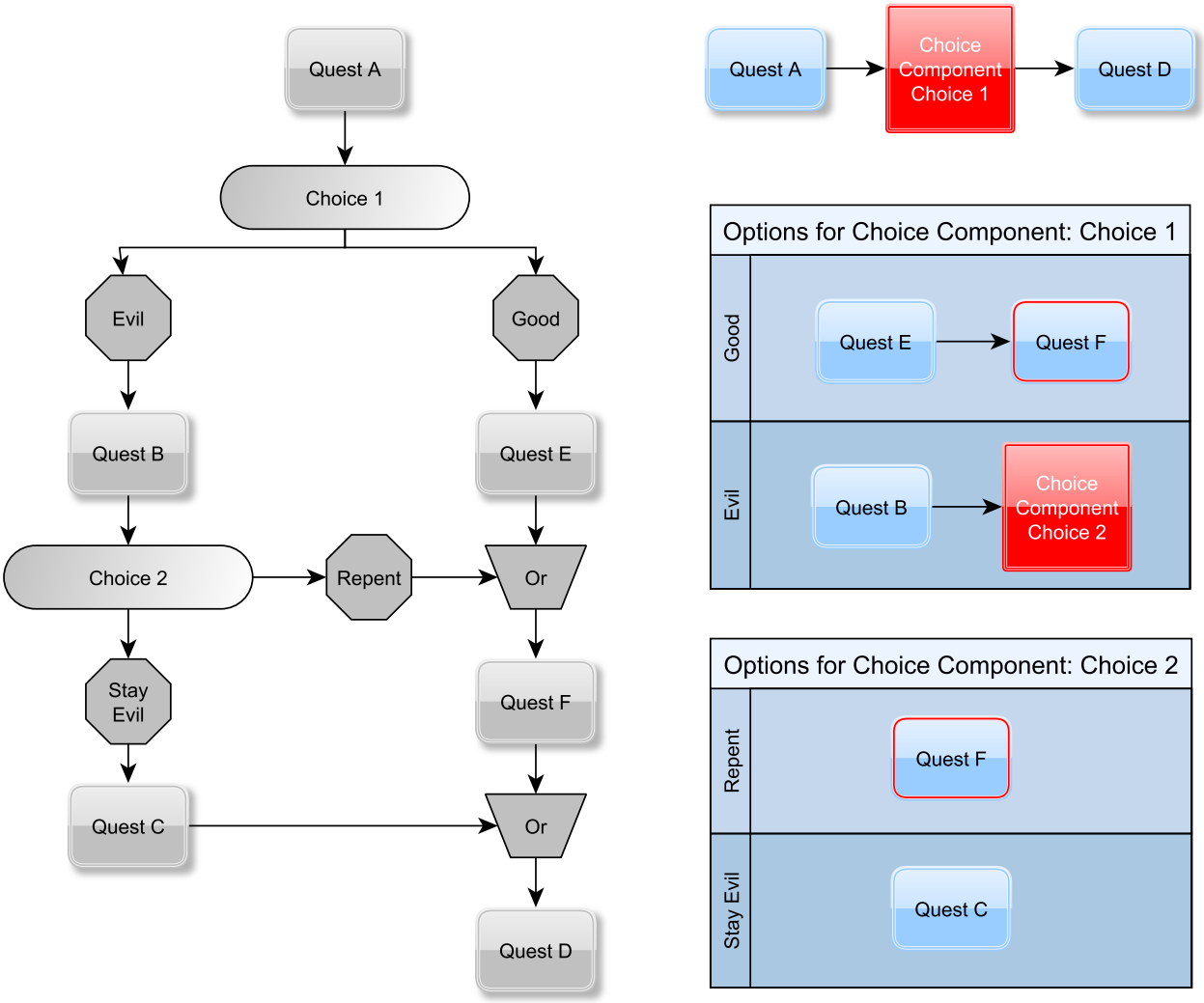

Figure 8.5: Example of the inability to alter a previous choice

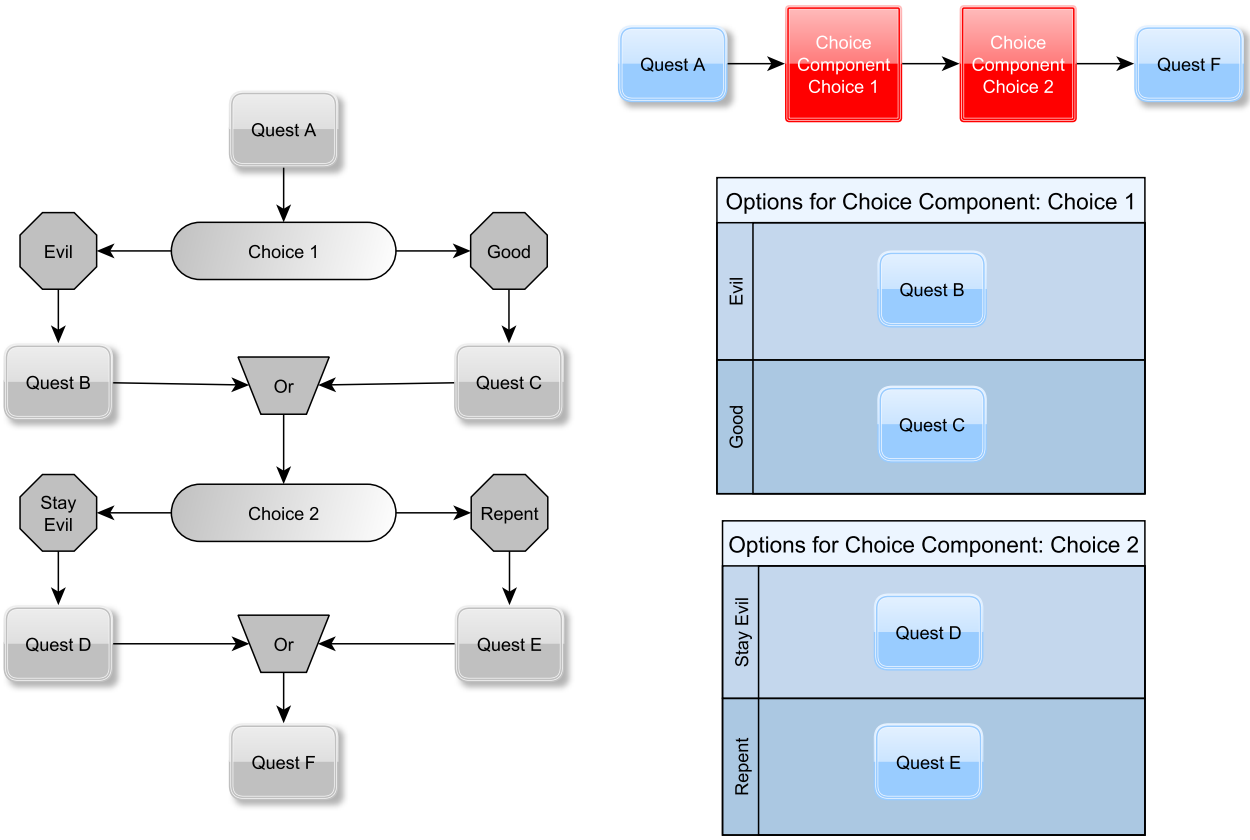

Figure 8.6: Example of chaining choice components to alter a choice

8.3 Using Elements

The last approach that will be discussed, is the element approach. With this approach new elements are created for both choices and options:

1. *Choice element*
   It indicates that at that node in the quest graph a choice needs to be made. A choice element can't depend on other choices or options, but solely on quests.
2. *Option element*
   It indicates a possible option for a choice. An option element can only depend on exactly one choice, this also means that options can't depend on other options.

In order to uses these new elements we will need to define how they can be connected in the graph. We decided that prerequisites are the best way to accomplish this. At this point it might seem that this approach is very much the same as the prerequisite based approach, because it also makes use of prerequisites. However, this is not the case because the element based approach contains explicit choices and options. Furthermore, the use of prerequisites is necessary in order to be able to put constrictions on the options and choices. It is for example possible to create a prerequisite for a choice that states:

Choice 1 acceptable = *(Quest A = completed* and *Player is level 5)*

The same is true for merge nodes who need prerequisites in order to be able to merge multiple options of a choice. So in a way prerequisites are the only way to create dependencies between elements in a quest graph, without losing the possibility to restrict the dependencies.

The choice element can use the existing prerequisites in order to indicate on which quests it depends. However, for the option element we need to create a new type of prerequisite, which is used to attach an option to a choice. Such a prerequisite is called a "choice available"-prerequisite. It's goal is to verify that an option can be chosen by checking, via the game engine, if no other option of the same choice was chosen.

We also need a new prerequisite for attaching a quest to an option. We call this prerequisite the "option chosen"-prerequisite. It checks, also via the game engine, if the option on which the quest depends was chosen by the player. This prerequisite can only be used between an option and a quest.

Figure 8.7 shows us an example of a quest graph which uses this approach. Please note that for this approach we will use an "or"-prerequisite relationship in order to merge different options from the same choice. In this case we show the "or"-relationship explicitly in order to prevent confusion. The prerequisite for "Quest H" in the example below would look like:

Quest H acceptable = *(Quest C = completed OR Quest D = completed OR Quest G = completed OR Option 3 = chosen)*

At this point, the option-elements might seem redundant. It is possible to implicitly define the options by letting all elements which come after the choice be "the options". However, from the prerequisite-based approach we learned that it would lead to the following problems:

1. It is not possible to add consequences in a correct way.
2. Constructing options that start with multiple quests will result in complex prerequisites.

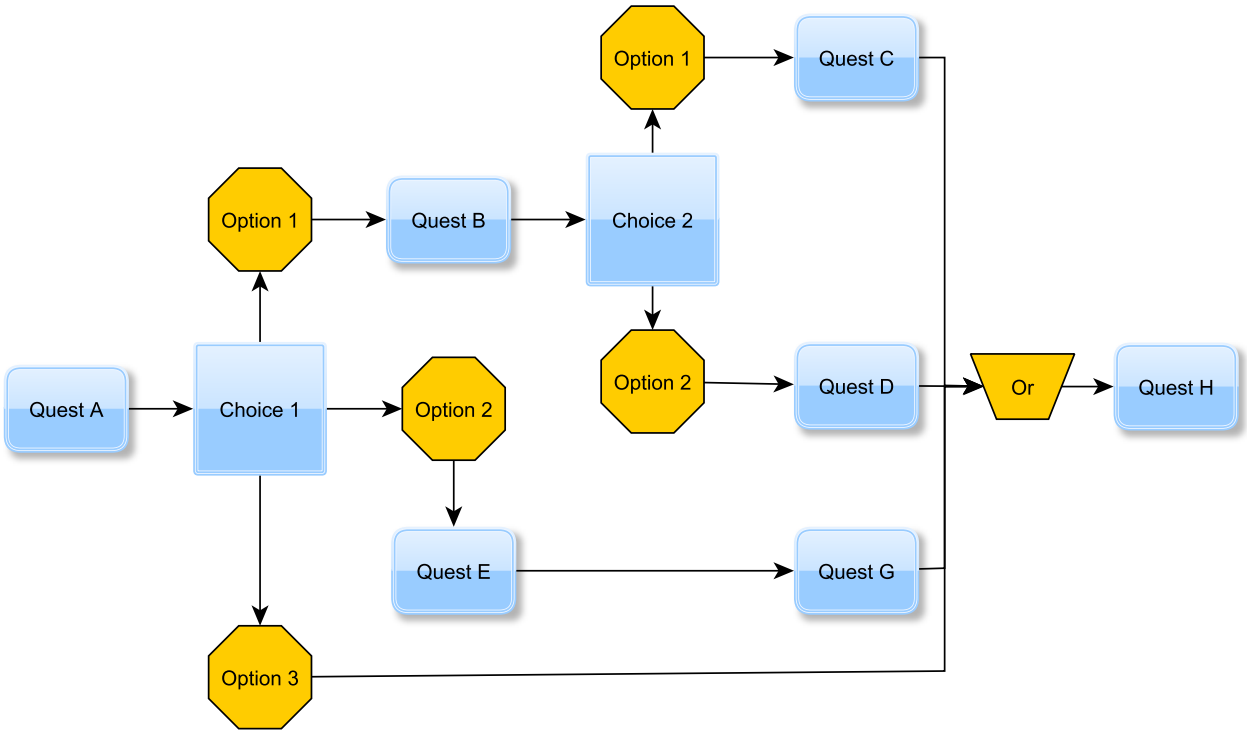

Figure 8.7: Example of adding choices via elements

The explicit option-elements are therefore used as an anchor to which all starting quests can be linked. And on which the consequences of an option can be stored.

Characteristics
Table 8.3 shows which characteristics can be obtained if the element approach is used to extend the quest graph with choices. The next section will explain the limitations in more detail.

Characteristic	Possible
Explicit choice	Yes
Consequences	Yes
Skipping a choice	Yes
Skipping a part of the quest graph	Yes
Altering previous choice	Yes
Multiple start quests	Yes

Table 8.3: Characteristics of the element based approach

Limitations
While this approach has all of the desired characteristics, there are some other limitations that are important to observe. The first limitation that is observed is that merge points aren't explicit. As a result a merge point can contain a high number of prerequisites. Of course this limitation can be lifted by explicitly adding merge points to a graph, but that would mean that we need more elements to represent the same graph. Furthermore, explicitly adding a merge-element would not negate the large number of prerequisites that are needed to attach quests to it.

Solutions that use this approach can get rather complex because there are many ways to mix the elements and dependencies. This results in the limitation that algorithms, like the ones defined in part one of the paper, will

get very complex for this approach. This could be a valid reason for not choosing this kind of approach.

Chapter 9

The Selected Approach

In this section we will explain why we consider the element approach the most suitable. Furthermore, we will discuss the impact on the algorithms that have been defined in the first part of the paper. Please note that from this point forward we will use the definition "the extended quest graph", which is defined as a quest graph that contains both the nodes from the normal quest graph and also the choice and option nodes from the element based approach.

9.1 Element based approach

We have selected the element-based approach because it supports all of the characteristics that we deem important for a quest graph with choices. The other two approaches have significant limitations which in our opinion are unacceptable.

In the remainder of this paper we will see that the flexibility that the element-based approach provides comes at a very steep price. Because this approach is very flexible, many different constructions are possible. This results in unclear boundaries between choices, which is not the case with for example the component approach. This means that scenarios can get very complex because there are many situations to consider.

The reason why the element approach was chosen in spite of these complexities is because we believe that it is better to start with an approach that can potentially support all desired characteristics, then to start with an approach that is limited from the start.

9.2 Impact on current algorithms

The impact of the addition of new elements to the quest graph will be severe. In the first part of the paper eight algorithms where discussed, which are used for:

1. Redundant link detection
2. Atomization
3. Cycle detection
4. Topological ordering

5. Shortest path
6. Longest path
7. Mandatory Quest ratio
8. Quest Order

In this section the impact of the new elements on these algorithms, with respect to the applicability and usability, will be discussed.

Differences
The normal quest graph is quite different from the extended quest graph. The most important difference is not the occurrence of the new choice and option elements. What is the most important difference is the fact that in the normal quest graph all quests can be accepted at some point in time. However, the extended quest graph does not have this characteristic. Choosing one option will lead to a certain path of quests, meaning that the paths of the other options have become unacceptable. This means that the algorithms cannot depend on this characteristic anymore.

Another important difference with the normal quest graph is that algorithms now need to be able to select the optimal option for a choice in different scenario's. This means that the algorithms need to be aware that an option needs to be chosen in order to obtain a valid path. This problem mainly effects the path algorithms because they allow a user to query for a specific path in the graph.

Redundant Links
The "redundant link"-algorithm will work correctly when used on an extended quest graph. The reason why it keeps working can be explained by looking at the new elements. These elements are connected in the same way as the elements that where introduced in the first phase of this paper. Therefore, all edges that are marked as a redundant link are either really redundant or a violation of the rules to which a choice or options must comply.

An example of a violation that results in a redundant link is shown in figure 9.1, the redundant link has a blue color. In this scenario an option depends on another option, which is illegal. An option may only depend on one other element which must be a choice-element. In this case "Option 2" also depends on "Option 1" which is not an choice.

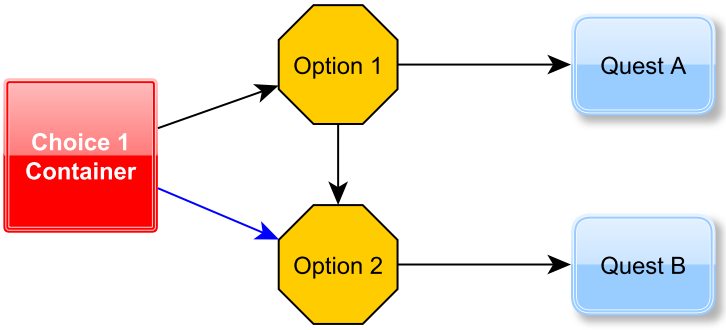

Figure 9.1: Invalid construction that results in a redundant link

Another example of a redundant link is shown in Figure 9.2. In this case the redundant link is not a validation of the rules for choices and options. Therefore, the link is redundant and will be detected accordingly.

Atomization
In the first part of the paper the concept of atomization was introduced as a way to simplify the graph. It can therefore be used to visually improve the graph by simplifying its structure. We would like to point out that the concept of atomization might change due to the introduction of choices. In the first part of the paper

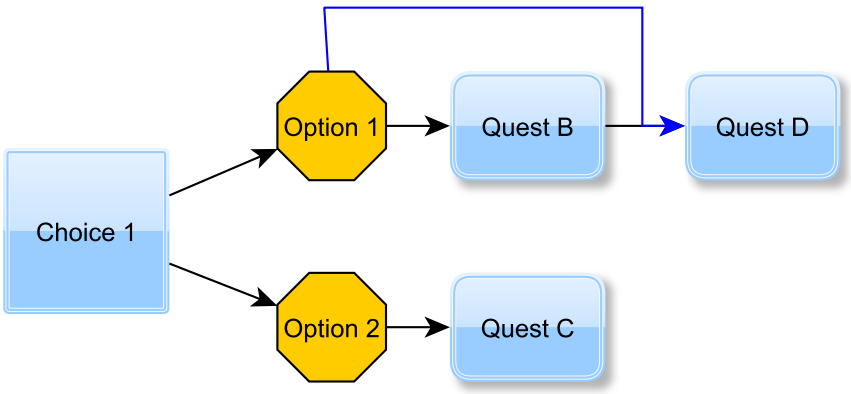

Figure 9.2: Valid construction that results in a redundant link

we used atomization to collapse chains of quests. For the extended quest graph it would be better to collapse choices. This means that the algorithms involved in the atomization process need to be changed. For this part of the paper atomization is considered to be outside the scope of our research, because we solely focus on the introduction and usage of choices in a quest graph.

Cyclic Detection
The "cycle detection"-algorithm defined in the first part of the paper will not work correctly on the extended version of the quest graph. This is because the algorithm destroys part of the quest graph in order to detect cycles. In the first part of the paper this problem was circumvented by cloning the quest graph. Later in Chapter 12 we will explain a more efficient way than the cloning of a graph. At this point it is important to understand that cycle detection need to be performed in a different way.

Topological Ordering
In the first part of the paper, the algorithm from Cormen et al.[ea01] was used to obtain a valid topological ordering of a path. This algorithm has to change slightly in order to be supported in this new version of the quest graph because the original implementation assumes that graph only contains nodes that are actually part of the topological ordering. However, this is not the case with the extended quest graph. Section 15 will explain this in more detail.

Path Algorithms
In the "Differences" section it was stated that algorithms need to be aware of the fact that making a choice influences which nodes are accessible in the quest graph. Because the algorithms from the first phase are unaware of the choices, it results in the problem that all quests in the graph will be visited by these algorithms. In order to make the algorithms work they need to be altered in a way that they will only visit quests that are accessible for that query. Furthermore, the algorithms need to be able to detect which option is best for a certain query type which can either be the shortest or the longest type.

Mandatory Quest Ratio
The Mandatory Quest Ratio is a simple metric that can be calculated by using the shortest and the longest path length. In the first part of the paper these values are calculated for a quest graph that is static. This means that the longest path is the shortest path plus all the optional quests.

Due to the introduction of choices the quest graph is not static anymore. It is likely that the shortest path will select different options than the longest path when a query is executed. This means that both paths can be very different and as a result it is illogical to compare both paths. So as a result the Mandatory Quest Ratio needs to be redefined.

In the new version of the quest graph the mandatory quest ratio is defined as: "the number of mandatory quests for a specific path in the quest graph". This means that a game designer can for example use the mandatory

quest ratio to check the difference between the shortest and longest path for a set of predefined chosen options. These predefined sets of options can be obtained with the "shortest"- and "longest path"-algorithms or can be created by hand by the game designer.

Quest Order
The chart that visualizes the quest orders consists of the intervals at which a quest can be accepted. Such an interval is calculated by using three values:

1. The number of nodes before the quest
2. The number of nodes after the quest
3. The total number of nodes in the graph

The counting approach will not work on the new version of the quest graph. First and foremost, the choice and option nodes will be added to the intervals, which will skew the results for the three previously mentioned values. Therefore, in order to calculate the quest intervals correctly, a normal quest graph needs to be extracted from the extended quest graph based on the choices that will be made. After we obtained the graph it is possible to use the algorithm from the first part of the paper to calculate the quest order.

9.3 Redefinition of a node

In the previous section we explained which algorithms are relevant for this part of the paper. The only algorithm that wasn't relevant for this part is the "atomization"-algorithm. As a result the definition of a node needs to be changed. Furthermore, the addition of new elements also changes the definition of a node. In part one a node was defined as:

"A node in a quest graph can either represent an entire quest, part of a quest or a quest chain atom."

In this part of the paper a node is defined as:

"A node in an extended quest graph can either represent an entire quest, a choice, an option or a container."

Please note that with this new definition sub quests also have been excluded from this part of the paper. However, because sub-quests are transformed into a graph friendly representation it still is possible to use them in this part of the paper.

Chapter 10

Scope Definition

Creating the algorithms that handle all of the previously mentioned characteristics is very hard. This is because some of the characteristics impose constructions that make the structure of the graph very complex. This problem gets increasingly more complex when we consider that a game designer can use combinations of these characteristics when working on a quest graph. Section 7.6 shows an example of a complex graph. We therefore concluded that this problem is too complex to solve all at once.

We therefore redefine the scope of the problem in a way that is more restrictive. By first finding a solution for a smaller set of characteristics, we will gain new insights into the problem. These insights can then later be used to get a better understanding of a more complex version of the problem.

We define the following boundaries for the first step towards finding a solution:

1. All options of a choice must merge in one point
2. "Or"-prerequisite relationships are only used to merge entire options
3. All nodes belonging to a choice must at least have one in-edge
4. Dependencies may not cross over choice or option boundaries

In the remainder of this chapter we will explain these boundaries in more detail. Furthermore, we will give examples that violate these boundaries and explain how this limits the possible constructions a game designer can use when creating a quest system. Please note that in the remainder of the paper we will show the prerequisite relationship nodes, the "and"- and "or"-nodes, explicitly in order to prevent confusion. However, in reality the relationship nodes are not present in the graph!

10.1 All options of a choice must merge in one point

The first restriction states that all options that belong to one choice must merge in one element. In most cases this element will be a quest but it is possible to merge in a choice. As a result of this restriction it becomes possible to subdivide the graph into small sub graphs, without changing the intent. Chapter 12 explains how this can be used to our advantage when querying the Quest Graph. Figure 10.1 shows a graph that adheres to this rule. The graph contains Choice 1 which is the main choice for the graph. The first option for Choice 1 contains an inner option, called Choice 2.

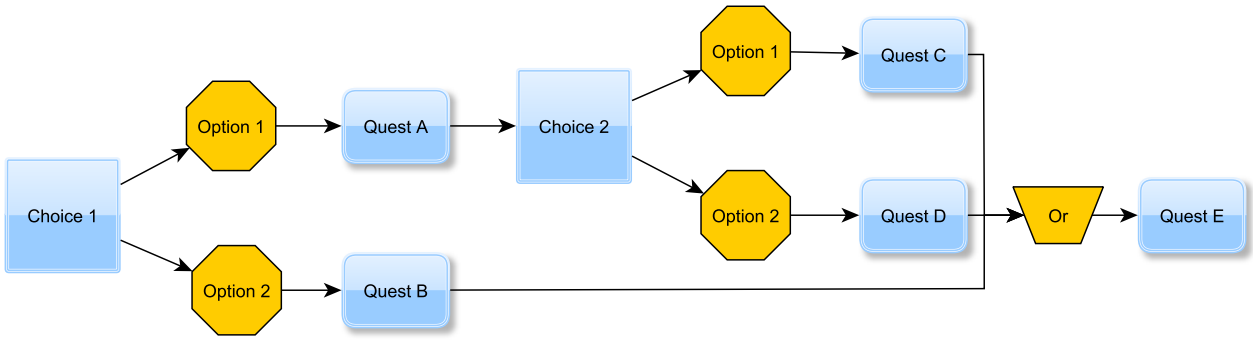

Figure 10.1: Example that adheres to the one merge point rule

Because Choice 2 is completely part of the first option of Choice 1, it is possible to replace the entire set of nodes that represents Choice 2 with another node. Figure 10.2 shows that replacing Choice 2 results in a graph that is more compact, therefore making it easier to understand. For now it is not important how and why the choice is replaced. This will be explained in detail in Chapter 12.

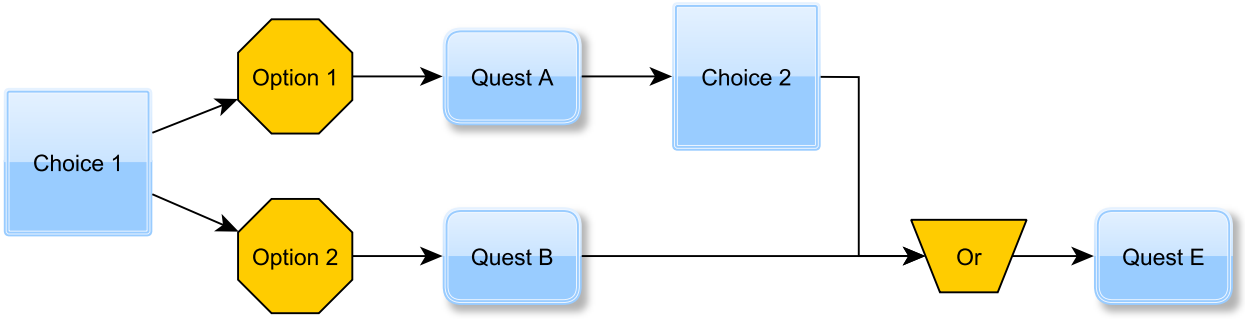

Figure 10.2: Example in which choice 2 is substituted with another node

If a choice violates this rule, we lose the property that we can replace a choice with another node. Figure 10.3 shows a graph that bares close resemblance to the graph used in the previous example. However it is slightly altered in order to make it violate the rule. Because both options of Choice 2 do not merge, the rule is violated which means that we cannot replace Choice 2 with another node. This is because it is unclear which of the nodes are part of Choice 1 and which of the nodes are part of Choice 2. Furthermore if we would assume that Quest C and Quest D belong to Choice 2, then the intent of the quest system would change if Choice 2 was replaced. This is because Quest C is not attached to Quest E in the original graph. However if we replace Choice 2, then the replacement node would be linked to Quest E. This would imply that Quest C is also linked to Quest E.

As a result of this limitation it is impossible for a game designer to create a choice that branches into two completely independent "sub quest systems". The quest systems will never be truly independent because in the near future there have to merge at some point. This means that a choice will never permanently effect the quest graph. The paths need to eventually merge which means that the effect of the choice is lost.

Please note that it is possible to introduce a fake node that will be used as a merge point. This can effectively make a "tree" of choices valid. But there are many other cases in which it is not possible to use a fake node. An example of such situation is shown in the previous example. If we would add a fake node, that is linked to Quest C and D, we would end up with the graph that is shown in Figure 10.4. However this graph is invalid because it violates a different constraint, which is the rule that states that dependencies may not cross over

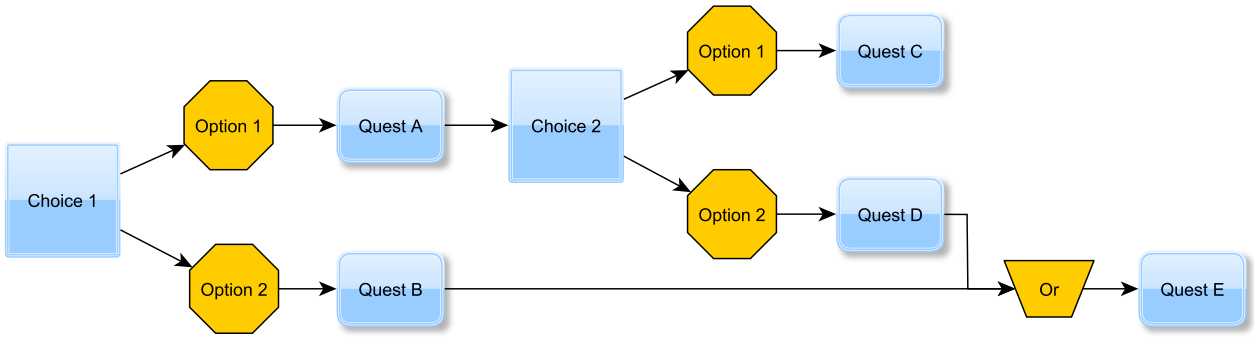

Figure 10.3: Example of rule violation. Quest c should be linked to quest e.

choice or option boundaries. This problem can't be solved without changing the structure of the graph and therefore changing the way the game designer intended the quest system to be used.

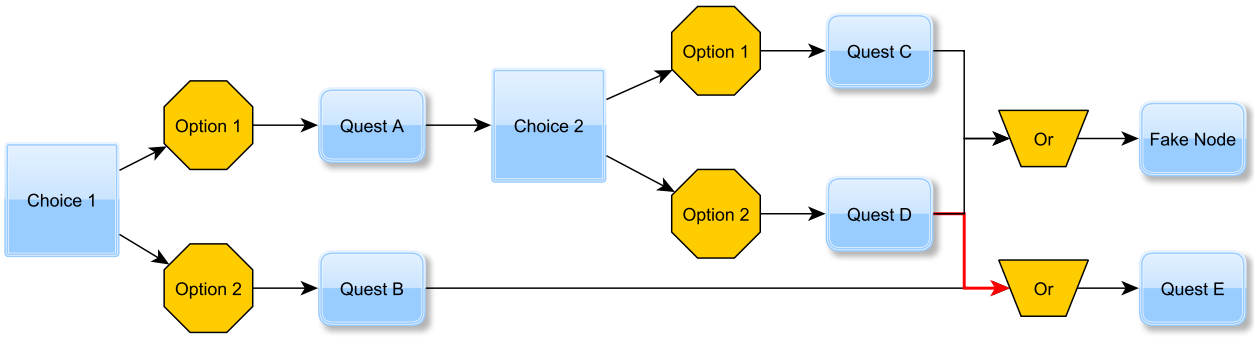

Figure 10.4: Incorrect example of a choice that uses a fake node to adhere to this rule.

10.2 "Or"-prerequisite relationships are only used to merge entire options

The second limitation states that "or"-relationships can only be used to merge entire options of a choice. This means that if a set of options merges, all the branches of those options need to merge in the same merge point. This limitation prevents the problems and removes complexities that arise when an option merges at multiple points. Please note that it is still valid to create optional paths. They are always explicitly part of a choice, because they never merge.

An important limitation of this rule is shown in Figure 10.5. It shows how multiple merge points can be used in order to create a mandatory common path between two options. Please note that "Path Option 1", "Path Option 2" and "Common Path" are in reality much longer and more complex paths. The first merge point in this graph is "Common Path", which allows the player access to this path if one of the two options is chosen. While this structure can be very useful for a quest system, it violates the rule which means that it cannot be used.

The previous example isn't very complicated, but it does introduce some problems. Because the options do not merge in one point, some of the nodes belong to multiple options. In this case the "common path" belongs to

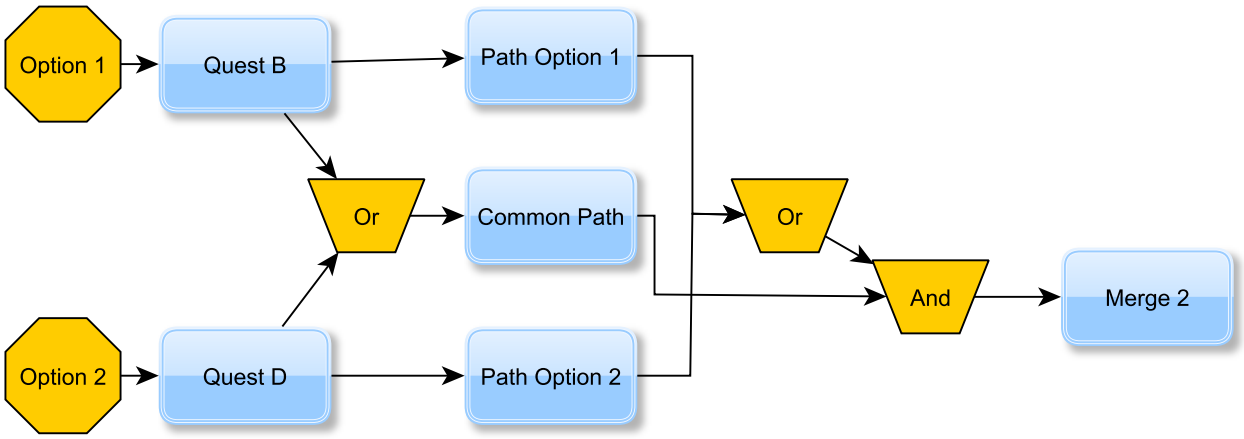

Figure 10.5: Example of an invalid choice with common path

both options. While it is not very complex in this situation it can get really ugly in more complex situations. In order to prevent these complexities this limitation was put in place.

There is another situation in which it is possible to use the "or"-relationship and that is when creating a construction that gives the player a choice to complete at least one, but potentially all quests before continuing with the main branch of the graph. This situation is shown in Figure 10.6.

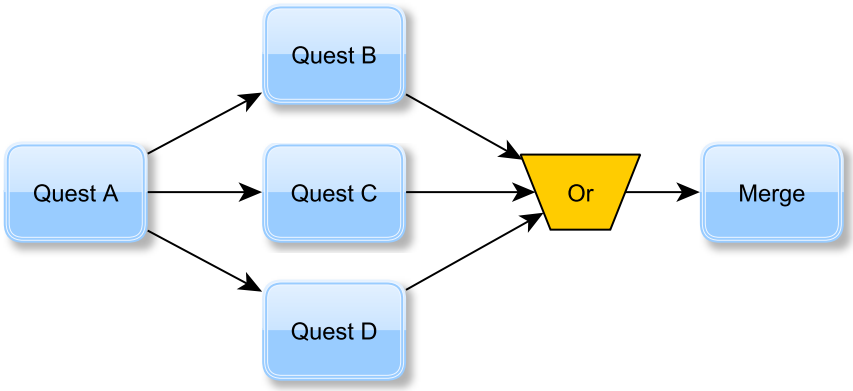

Figure 10.6: Example of an "or"-relationship used for quests

After completing Quest A it is possible to either complete Quest B, C or D and continue with the "merge"-node which will give the player access to the main branch. However, it is also possible that the player completes a subset of these quests before he continues on the main branch. This construction furthermore complicates the order of the quests because the quests that were not completed before continuing on the main branch, can be completed at any stage of the game.

10.3 All nodes belonging to a choice must at least have one in-edge

The third constraint on the structure of a choice is the limitation that all nodes that are part of a choice must at least have one in-edge. The option-elements automatically adhere to this rule because they need to be attached to the choice in order to be part of it. This limitation is put in place to prevent the problems that arise from quests that are mandatory for an option but are not directly connected to that option. An example of such a situation is shown in Figure 10.7.

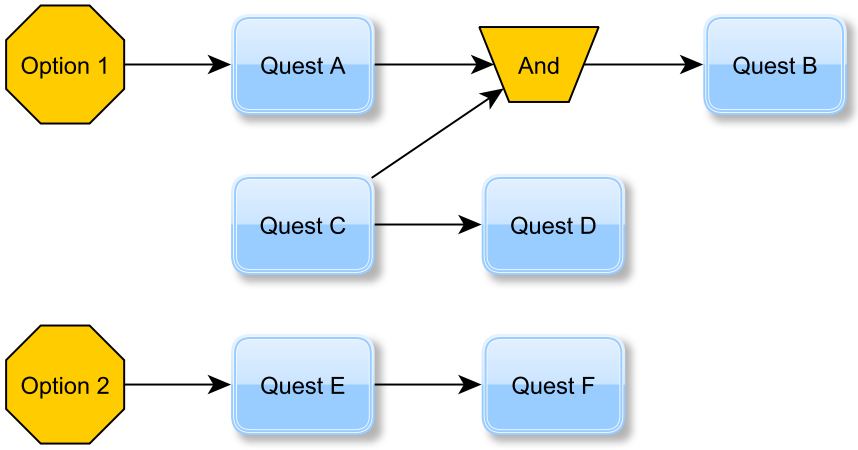

Figure 10.7: Example of a mandatory disconnected quest

Quest C technically is part of the first option, because it is needed in order to complete the entire path of that option. However, if we do not choose Option 1 it still is possible to complete Quest C, because it has no dependencies. This also means that it can be completed at any given time, because it is not limited by dependencies. The biggest problem with Quest C is the fact that it can be completed without having made any choice. Quest C can be completed before or after the choice is made. Therefore in some situations it will be part of the choice, while in other situations it isn't. This is a violation of the scope because is crosses the Choice boundaries. If the intent of the game designer is to make Quest C part of Option 1, then he should connect Quest C to Option 1.

While the previously explained violation is enough to not allow such a structure, it is interesting to look at the meaning of this construction if we explicitly create the dependencies between options 1 and 2 and quest C. In this case it is assumed that quest c cannot be accepted any time before the choice is made. Figure 10.8 shows what the graph would look like if this relationship was created explicitly.

It is interesting to see that Quest C has two prerequisites that are created with an "or"-relationship. This allows the player to accept Quest C either if Option 1 or Option 2 is chosen. However, the use of an "or"-relationship is a violation of the rules. This example has much in common with the example in the previous section. In a way Quest C can be seen as a "Mandatory Common Path". The big difference however is that only one of the options depends on that "Common Path".

For a game designer this limitation means that it is not possible to create "a quest that can be accepted at any given time, while it is connected one or more options of a choice". The fact that the quest can be accepted at any given time also indicates that the point at which this happens is not important. If it would have been important it would have dependencies on other quests, and we would not have been in this situation.

Therefore the game designer can work around this limitation by either explicitly connecting quest c with an

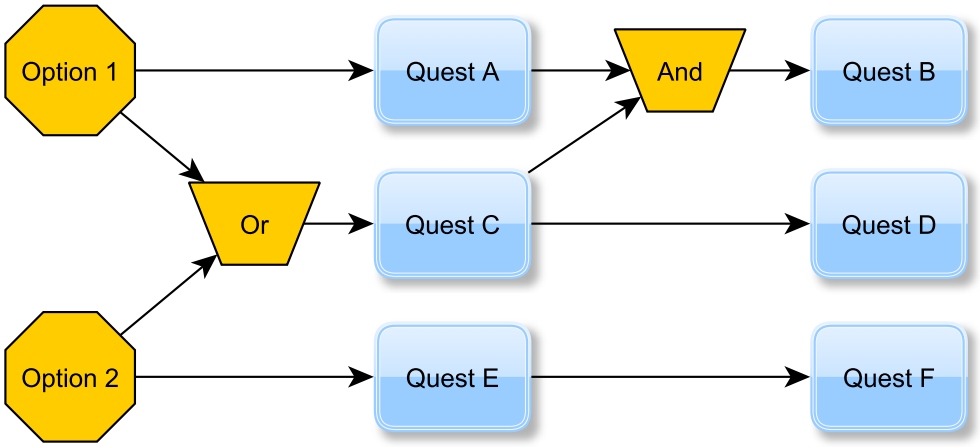

Figure 10.8: Example of explicit representation of the mandatory disconnected quest example

in-edge to an option, or to remove it from the choice and place it in parallel as is shown in Figure 10.9.

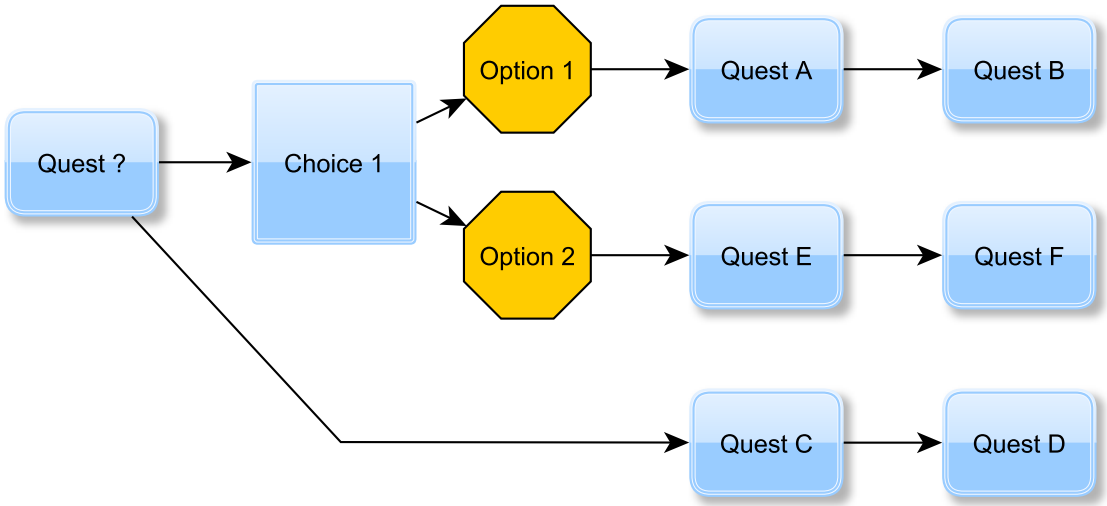

Figure 10.9: Example of parallel quests without mandatory disconnected quest

10.4 Dependencies may not cross over choice or option boundaries

The final constraint limits how dependencies can be defined for a choice. In this case it is illegal to let a dependency cross over a choice or option boundary. The most important reason for this decision is that it will prevent (implicit) conflicting dependencies.

Figure 10.10 shows an example of an implicit dependency. The game designer has decided that Quest B will give the player a reward in the form of Item X. However, this item is needed by Quest D, therefore creating

an implicit dependency between Quest B and Quest D. In the first part of the paper we explained that such a dependency will be explicitly added to the graph. So by restraining such a dependency, the game designer can automatically be notified that this is a violation of the rules.

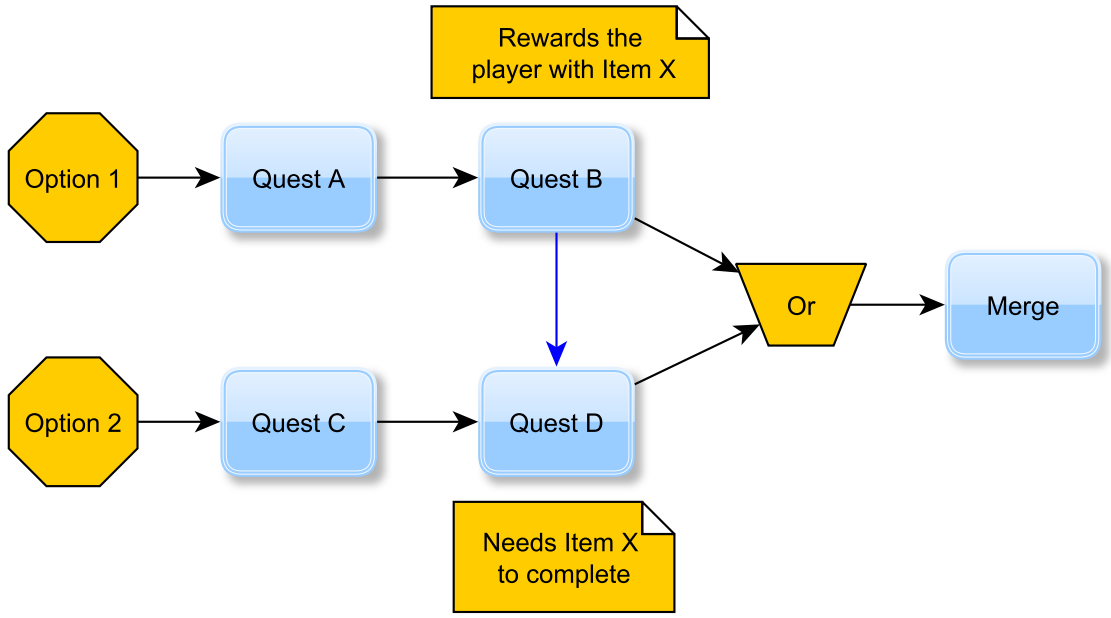

Figure 10.10: Example of an option boundary violation

This results in the limitation that if a certain quest needs an item, it needs to be provided in the same option of the same choice, because otherwise the rules are violated.

As a result of this limitation two important characteristics are considered illegal. These characteristics are "Skipping a part of a quest graph" and "Altering a previous choice". In our opinion this limitation has the most impact on the usability of the new version of the quest graph. It is very unfortunate that these constructions can't be used, but otherwise the graph would get too complicated to work with.

10.5 The consequence of the total set of limitations

In the previous sections we have explained how the scope of the extended quest graph is restricted. For each of those limitations we explained how the game designer is limited when creating a quest system. In this section we will explain what the consequences of the combined limitations are.

A consequence of the limitations is that we have constrained the element based approach in such a way that it effectively has become the component based approach. This is the result of the rule that states that "Dependencies may not cross over choice or option boundaries". However, there still is a little difference with the component based approach, and that is the explicit occurrence of option-elements in the quest graph. The component-based approach stores these options inside each component for each choice. However, in Section 12 we introduce the Choice Graph, which is an transformed quest graph, as a means to query the quest graph. This new graph is in fact structured in the same way as the component based approach graph.

So please be aware that the examples shown in the remainder of the paper might look like the component based

approach but that they are in fact are part of element based approach.

10.6 Impact on Usability

In many ways the scope definition limits which structures a game designer can use in order to create interesting quest systems. The most unfortunate limitation is the absence of a legal way to create the “skip a part of the quest graph” and ”altering a previous” choice characteristics. Also some other less complex constructions, like a shared path between two options, have been classified as a violation of these rules.

While it is important to look at the limitations of the current scope, it is also good to look at the new possibilities it brings. If these outweigh the limitations, then this scope is good for the initial approach. So what is possible with an extended quest graph and this scope?

First of all, it is now possible to embed choices into the quest graph, something that was not possible with the original quest graph. This can, depending on the game designer, provide the player with interesting game play. This is in the first place accomplished with providing the player with choices that lead to different quest paths. Another way to make the quest system/game more interesting is by adding consequences to choices. This can create interesting game play for the player that can challenge him in many different ways. It is for example possible to create choices that constantly challenge the player based upon moral dilemmas. However, the possibilities that consequences provide do not stop there. It is for example also possible to influence the game state with the combined set of choices of the players.

Furthermore even with the limited scope it still is possible to skip a choice. This will provide the player with at least the opportunity to skip some content without getting stuck. This was not possible in the original quest graph and is a big improvement in providing the player with more freedom.

We conclude that there are many constraints on what a game designer can do. However, there are also many things that a game designer previously couldn’t do. We therefore accepted this scope as a viable solution to the problem.

Chapter 11

Choice Structure Cases

Even with the scope that is defined in the previous chapter it is possible to create several different choice constructions. In this chapter we will look at these different constructions. First we explain the construction options involving a single choice. Then we continue with cases involving multiple choices to create specific quest structures.

11.1 One single choice results in one merge-point

The simplest situation involving a single choice contains only one merge-point. This means that every option for that choice merges in that merge-point. An example of this situation is shown in Figure 11.1. In that graph the merge point is "Quest C", which merges "Option 1" and "Option 2".

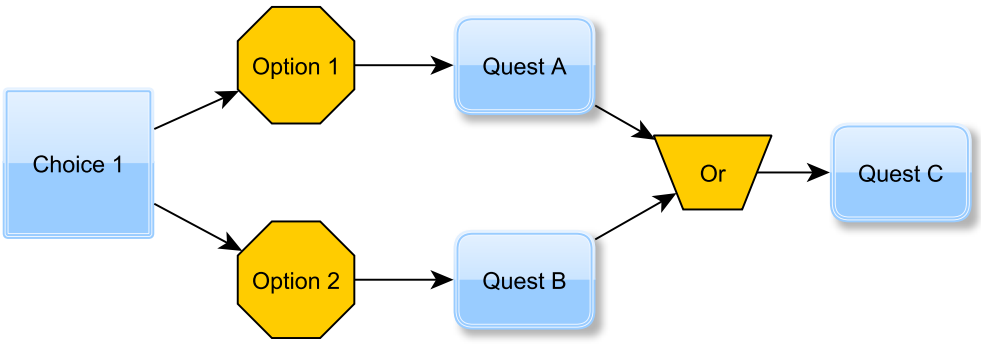

Figure 11.1: Example that shows a choice that merges in a single merge point

11.2 One single choice results in multiple merge-points

Another possibility is that one choice contains multiple merge-points. An example of such a situation is visualized in Figure 11.2. In this case there effectively are three merge points, which are “Quest E”, ”Quest F” and ”Quest G”. If there are multiple merge points, we can observe that there are subsets of options that merge in a merge-point before the final merge-point. It is important to observe the presence of these subsets because they will later be used to optimize path queries. In this example ”Quest E” and ”Quest F” both are merge-points with their own subsets of options.

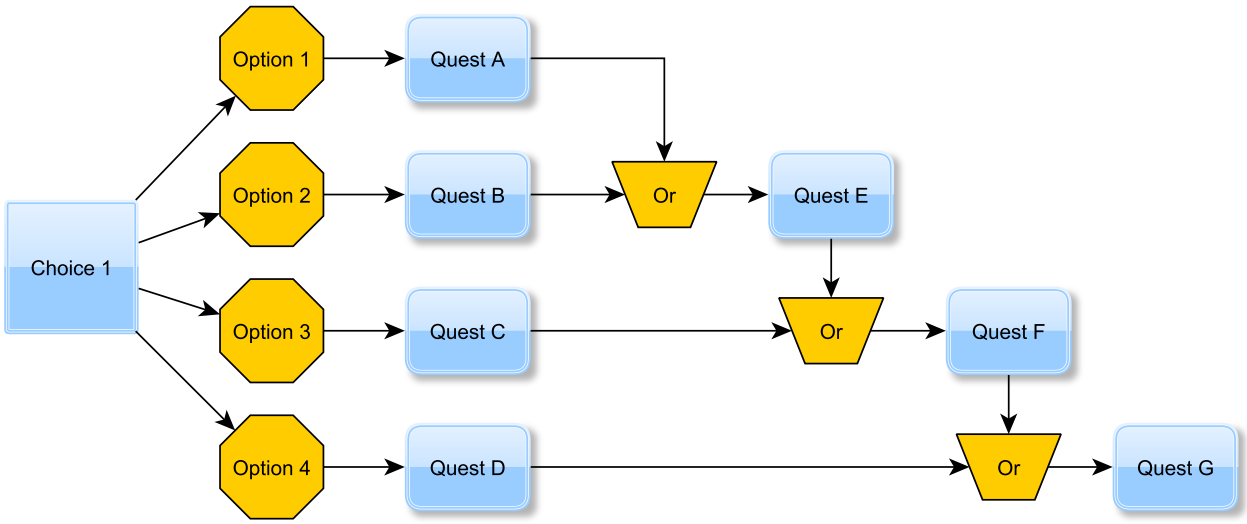

Figure 11.2: Example that shows a choice that merges in multiple merge points

11.3 Inner choice merges before the merge-point of the outer choice

A simple situation that involves multiple choices, is the nesting of choices that merge at different merge points. This means that the inner choice merges before the merge point of the outer choice. Figure 11.3 shows an example of this case. In this example “Choice 2” is the inner quest, which merges at ”Quest E” and ”Choice 1” is the outer quest which merges at “Quest F”.

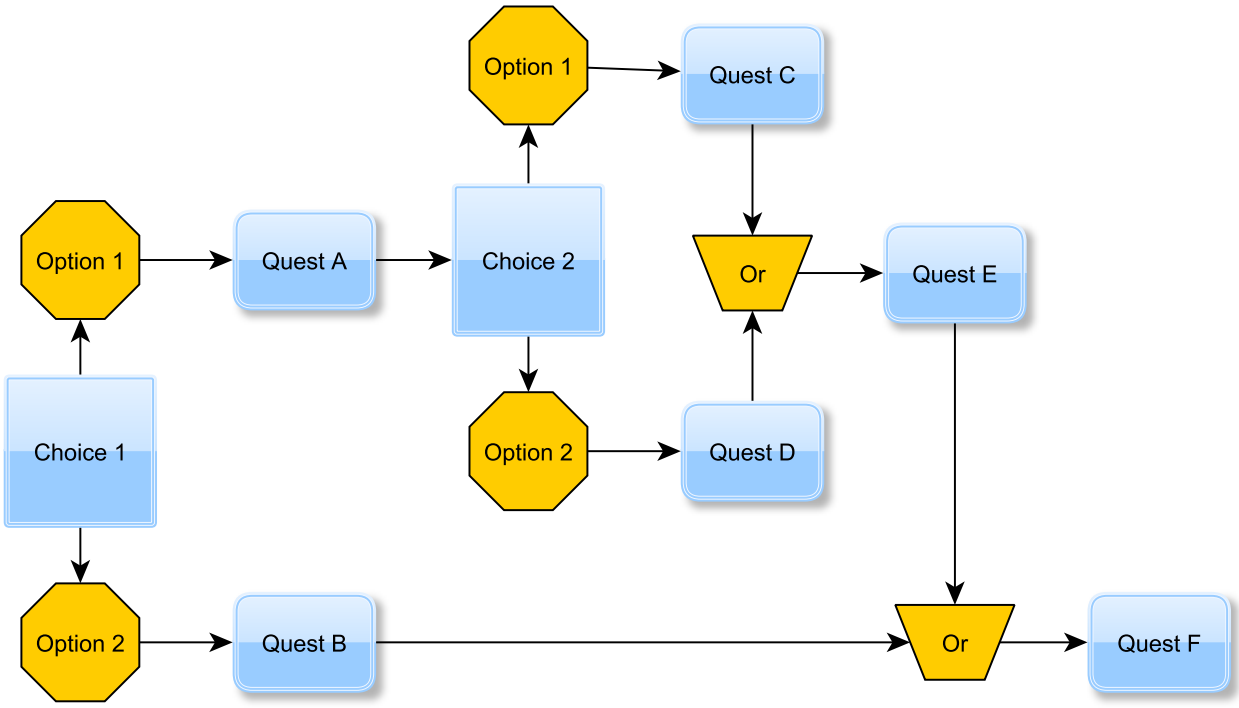

Figure 11.3: Example that shows an inner choice that merges before the outer merge point

11.4 Inner choices merge at the merge-point of the outer choice

A slightly more complicated version of the previous case is the case in which both the inner and the outer choice merge in the same merge point. If we look at Figure 11.4, that merge point would be "Quest F". It might seem that this case violates the rule that an inner-choice must be fully contained within the outer choice. However, that is not true because one could argue that the merge-point is not part of the choice. Therefore, this case is valid and it is important to remember that it can be used.

Another example of this case is the one that is shown in Figure 11.5. In this case there are three choices that are nested, which is still valid.

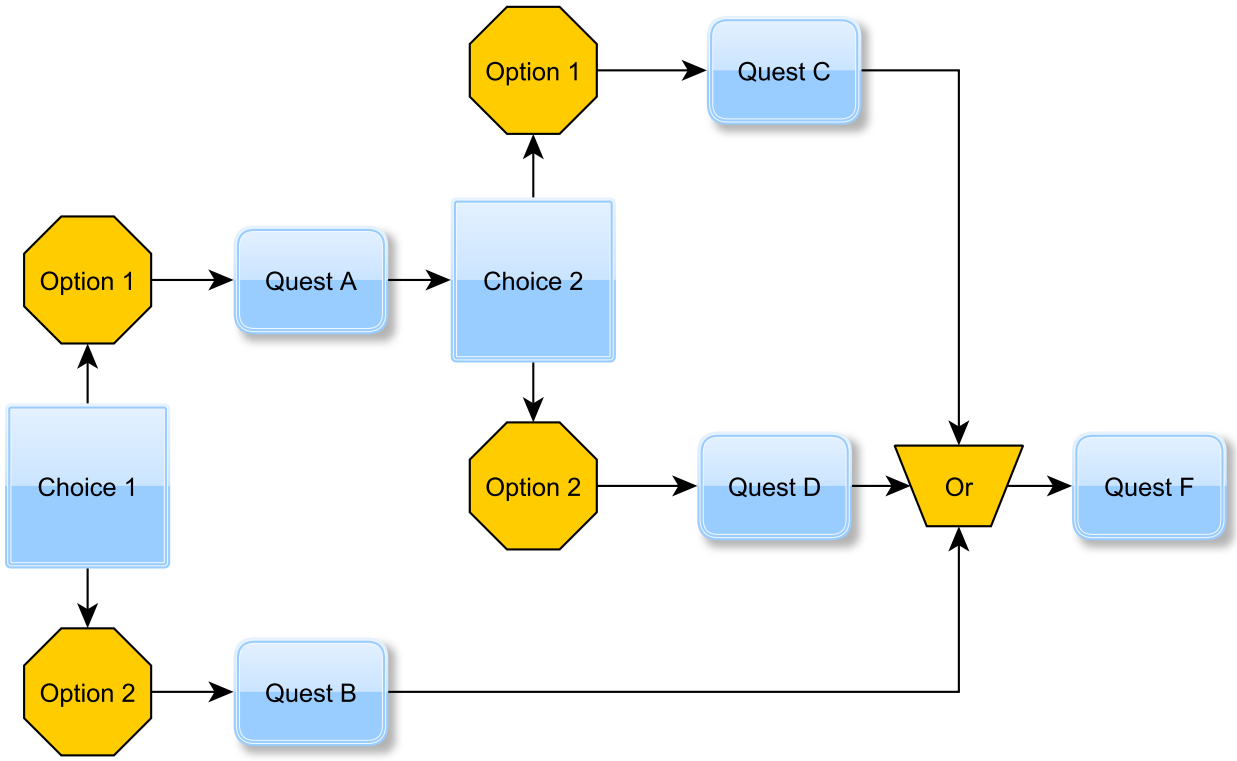

Figure 11.4: Example that shows a single inner choice that merges at the outer merge point

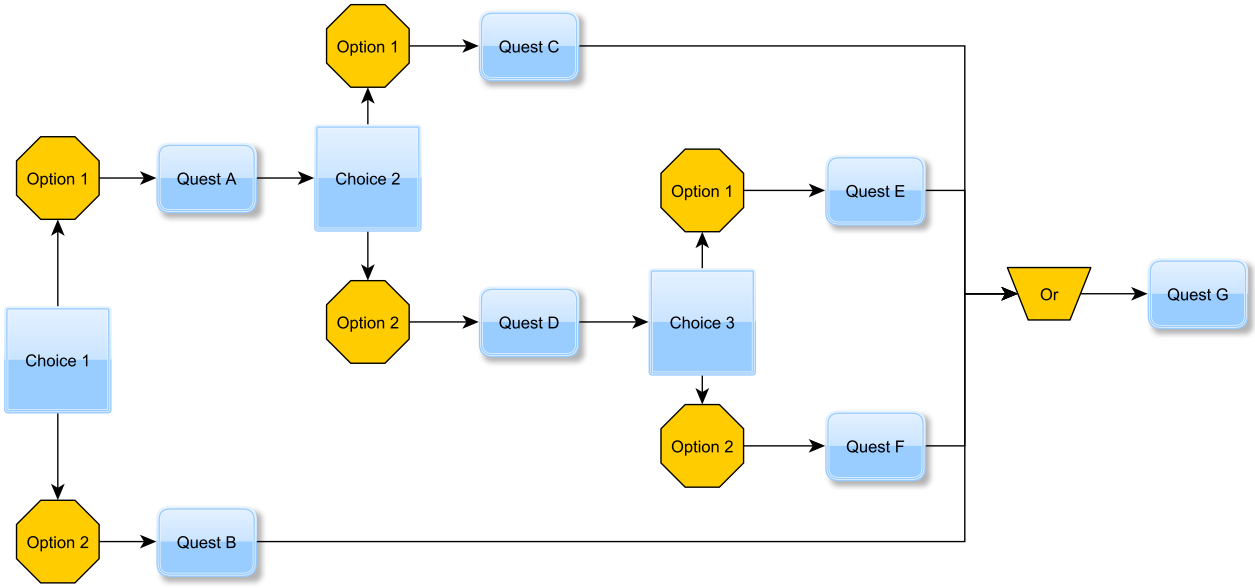

Figure 11.5: Example that shows multiple inner choices that merge at the outer merge point

11.5 Choices at the same depth can merge in the same merge point

In the previous case we looked at how multiple choices can effect each other on different levels. However, there also exists a special case which involves multiple choices on the same level. In that case the choices can merge in the same merge-point on the same level. An example of this case is shown in the Figure 11.6. Just as with case 4, this case is perfectly valid because the choices merge in one point and don't overlap.

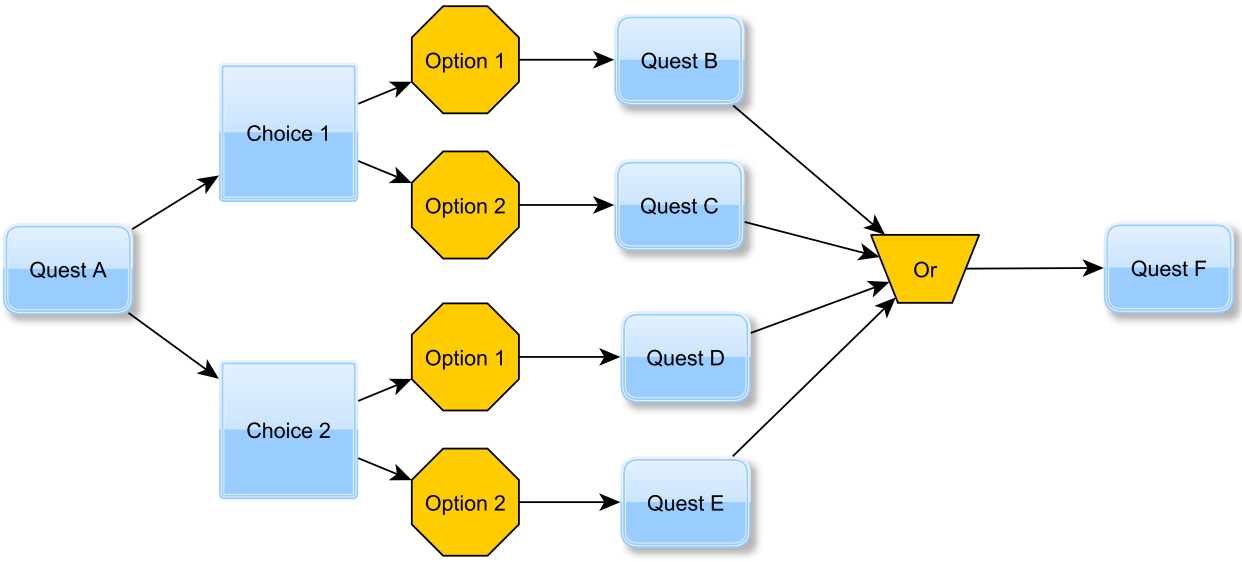

Figure 11.6: Example that shows that choices at the same level can merge in the same merge point

Chapter 12

The Choice Graph

If we look at the scope limitations we can imagine that a choice is a “black box”. This is a direct result from the rule that states that “all options of a choice must merge in one point” and the rule that states that “all inner choices must merge before or at the merge point of the outer choice”. The last rule is actually a rewritten version of the rule that states that “Dependencies may not cross over choice or option boundaries”. The most important aspect of a black box is that the internals of the box are not important. The only thing that is actually important is the fact that the black box does its job. This is an important concept to understand because this can be used to efficiently obfuscate the complexities that arise from the choices.

In this section we introduce a new graph, which we call a Choice Graph. The choice graph contains the same elements as a the extended Quest Graph, which are quests, choices and options. However, the choice graph also contains two new elements, which are the container and the dummy(-merge) element. The following sections will explain what these new elements are and how they are used.

12.1 Containers

A container is a “black box” replacement for an entire choice within a graph. This means that the part of the graph that represents the choice is removed from the graph and added to the container. Figures 12.1 to 12.3 show an example of a graph in which a choice is replace with a container. The first figure shows the extended quest graph, with one choice that merges in “Quest D”.

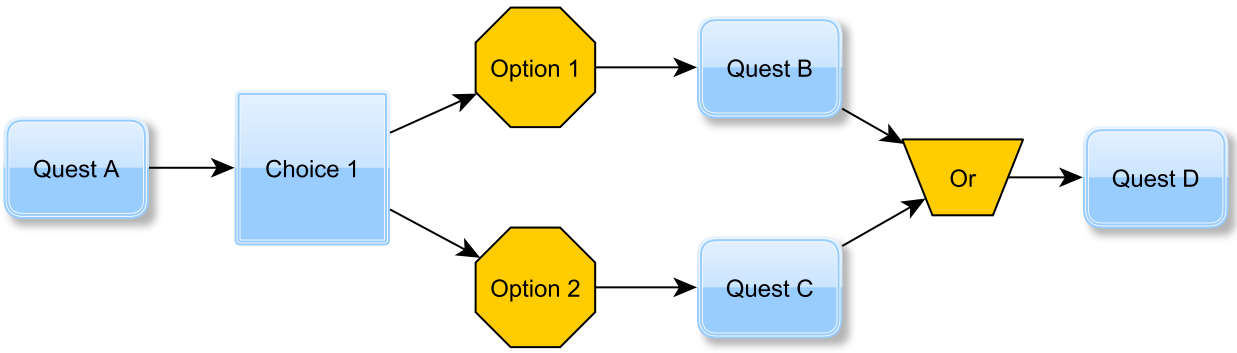

Figure 12.1: Example of a graph with a choice that can be replaced

Figure 12.2 shows what this graph looks like when the choice is replaced by a container. Please note that "Quest D" is not part of the container. The reason why "Quest D" is not part of the container can be found in case 5 of the previous chapter, which explains that it is possible that another choice exists that also merges in "Quest D". This also explains why we need the "dummy merge"-node to function as the fake merge point for the container. Figure 12.3 shows the graph that is contained within "Choice 1 Container". Also note that the choice is removed from the sub graph. The reason for removing the choice is, that effectively the container represents the choice, so it is no longer needed in the sub graph.

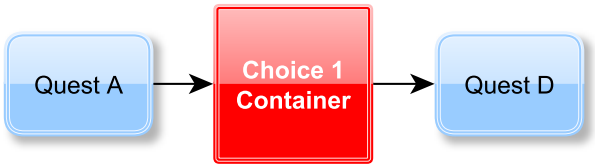

Figure 12.2: Example of the same graph with the choice replaced

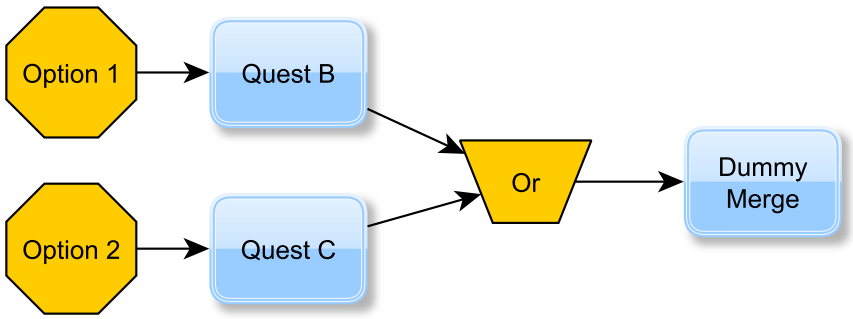

Figure 12.3: The nodes that are contained within the recreated container

The previous example shows how we can replace one choice with a container. However, it is possible that the graph that is stored in the container contains inner choices. So after creating a container it is important to also replace the containers that are contained within the sub graph. As a result it is possible that a container contains other containers which represent choices. Figures 12.4 to 12.6 show an example which shows a choice with an inner choice.

At the first level we have the graph which contains "Quest A", "Choice 1 Container" and "Quest F". At the second level, in "Choice Container 1", we have a sub graph which contains "Quests B and C" and "Choice 2 Container". At the third level, in "Choice 2 Container", we find the sub graph of choice 2.

So by introducing containers we have created a graph which consists of multiple layers. These layers can, as we will explain later, be used for optimizing the queries. But in order to do so, we need to be able to navigate through the containers. At this time we can only move from top to bottom via the containers and we can move horizontally via the edges between the nodes. However, we can't move from a container to its parent. Therefore, we need to add a pointer to the parent container, so that we can effectively move between layers vertically.

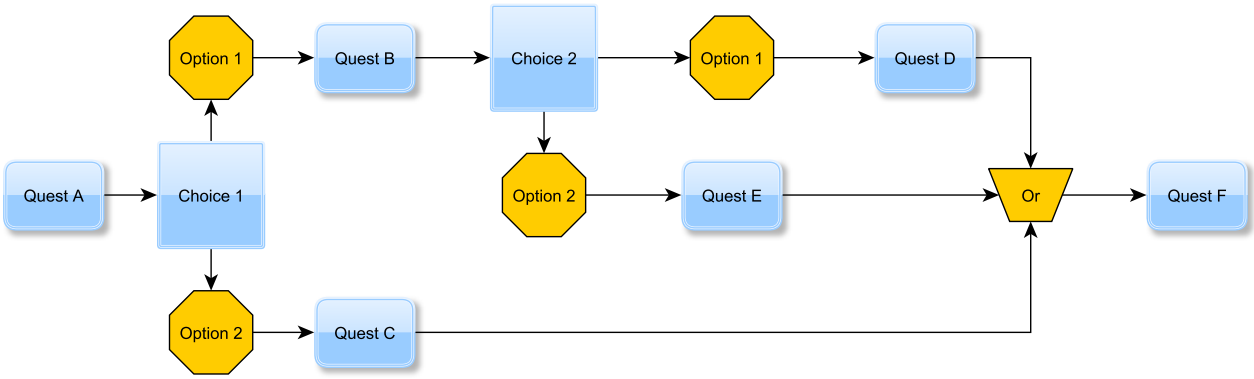

Figure 12.4: Example of a graph with an inner choice

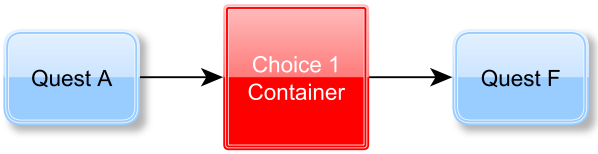

Figure 12.5: Example of a graph where the choices have been replaced with a container

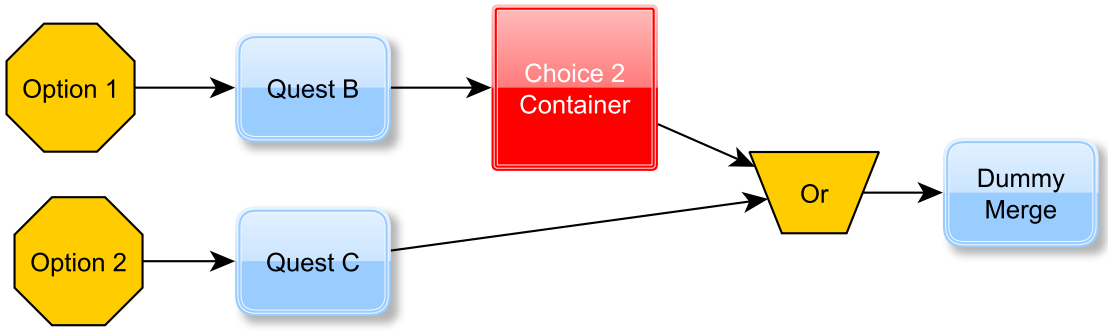

Figure 12.6: Example of the content of the container for choice 1

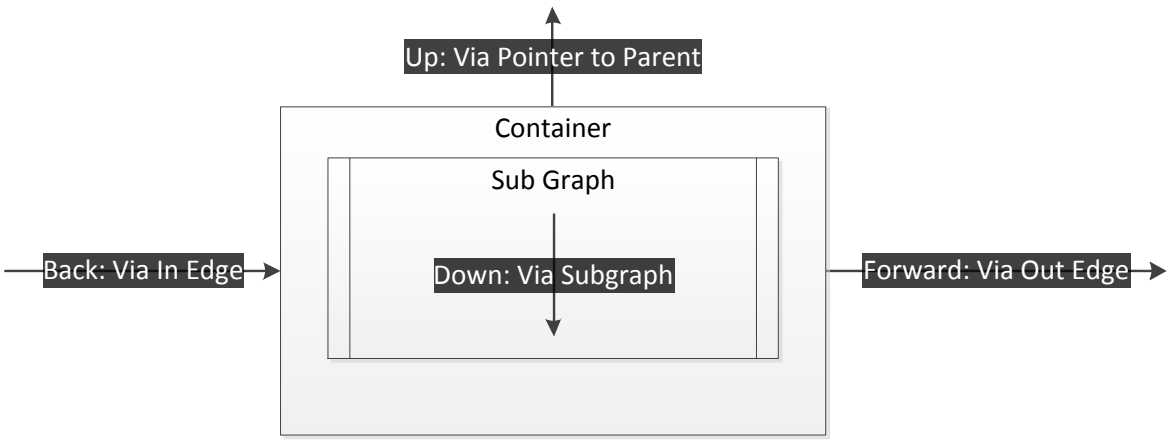

Figure 12.7: The pointer structure of a container

12.2 Using Containers

In the previous section we explained what a container is and how it can be navigated. In this section we are going to explain the influence the containers have on the structure of what originally was an extended quest graph. Then, we will explain how markers can be used to support querying the choice graph for paths.

The reason why we call the choice graph by that name, is that because it is constructed from an extended quest graph. This means that it in fact is a special version of an extend quest graph. However, if we look closely at the structure of this "graph", you will notice that in fact the graph has been turned into a tree as a result of the containerization. Each container, which functions as a node in the tree, stores a part of the total graph. Figure 12.8 shows the structure of a choice graph with three layers. The black arrows in the container represent the pointers to the parent containers, while the blue arrows indicate the pointers from container nodes within a sub graph to the corresponding container.

Please note that most parts of the original graph are contained within the sub graphs of the containers. However, at the root level we still have some quest nodes because they are not part of any choice and therefore are not placed within a container.

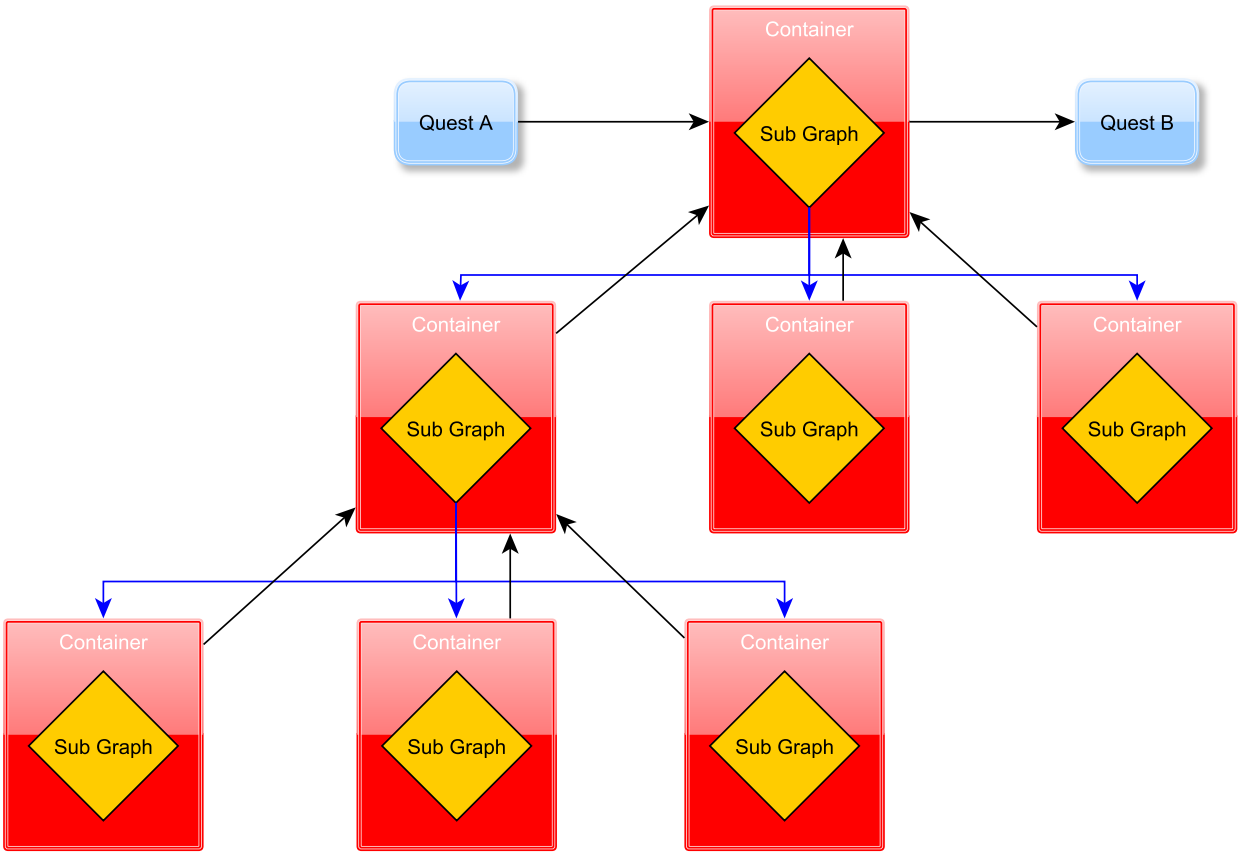

Figure 12.8: The structure of a choice graph

12.3 Markers

In order to use the Choice Graph, we need to be able to query it. In this section we introduce the concept of markers to address one major problem that arises from the path algorithms from the first phase. The biggest problem with those algorithms is that they have a destructive nature. Executing such an algorithm results in an unusable graph. In the first phase this was no problem because the graph was cloned. Then the algorithm would be executed on the clone, leaving the original graph intact. However, we decided to move away from the cloning based approach because it is very inefficient.

To solve this problem we first need to pinpoint the reason for the algorithms destructiveness. It turns out that the algorithms are destructive in order to assure that graph only contains nodes that are actually part of the result set of the query. It means that all nodes before the "from"-node and all nodes after the "to"-node have simply been removed.

Instead of actually removing the nodes we propose to use markers which are placed on each node. A marker contains data that can be used by algorithms. In the case of the original algorithms, the marker should contain information on the accessibility of the node. We will not provide the "marker"-algorithm for the normal quest graph because these algorithms can't be used on containers. Therefore, they are considered to be outside the scope of this research.

Luckily the concept of markers can be extended in order to run path algorithms on a container. There is a big difference between a normal quest graph and a graph that is part of a container. In the normal quest graph all nodes can be accessed at any time. However this is not the case for the graph that is part of a container because it contains options. When such a graph is queried, only one option will yield the desired result. This means that only the nodes that are on the path of that option are accessible for the query algorithm. The other nodes should not be accessible because they have been excluded by that choice. In the next section we explain how markers can be used in order to mark nodes with their respective accessibility depending on the chosen option.

12.4 Marking the accessible nodes in a container

In this section we will explain how we can use a marking approach to mark all accessible nodes for a query. We need the same "from"- and "to"-nodes that are going to be used to query the container. First we explain that it is not possible to extract the complete set of accessible nodes without markings by either backward or forward scanning. Then we will explain how we can use a combination of forward and backward markings in order to reach our goal.

The first thing that comes to mind when trying to find all accessible nodes is to use a backtracking algorithm. This is because backtracking algorithms have the property that they will find all mandatory quests but not the optional quests. This means that backtracking alone will not be enough to obtain all accessible nodes. Furthermore backtracking in a container is a bad idea because of a different reason. The problem with backtracking is that we will stumble on merge points. When we encounter such a node it is not clear which of the predecessor-nodes will lead to the desired option. An example of this situation is shown in Figure 12.9. At the dummy node we need to decide if we go towards "Quest F" or "Quest D" in order to reach the desired option. At this point things become complex, we could backtrack from each predecessor node in order to check if it is on the right path, but that would not be very efficient. From this we can conclude that scanning backwards is not the correct solution towards finding the correct set of accessible nodes.

As a result of the backtrack approach being too complex, we could turn towards a simple forward tracking approach. This simply recursively moves forward over all edges and yields all visited nodes as being accessible

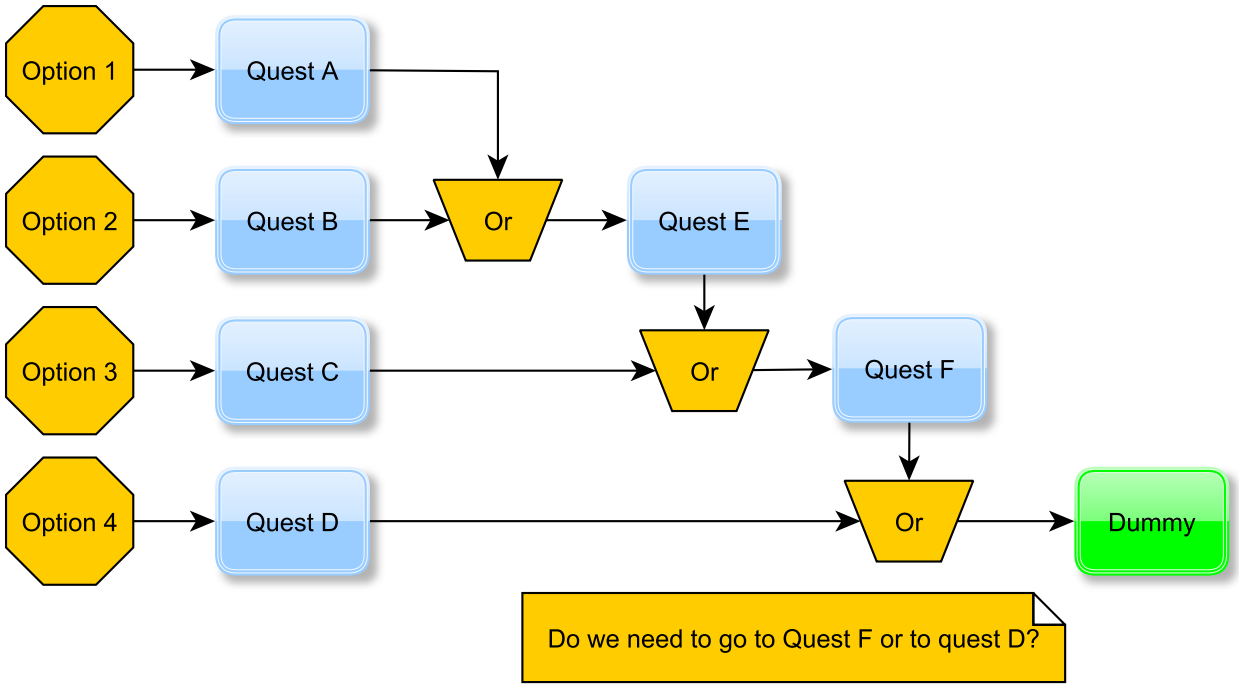

Figure 12.9: Backtracking is not a good idea. Which way should we go?!

nodes. This normally does not have the characteristic that it will find all mandatory nodes. However, due to the limitations imposed by the scope, the graphs contained in the containers are structured in such a way that a forward marking approach can find all mandatory nodes. Figure 12.10 shows a graph which we will use to explain why this is the case. If it was a normal graph we would not mark Quest G "accessible" because it is not on the path from Option 1 to Dummy. However, the scope contains a rule that states that "all nodes in a choice must at least have one in edge", which means that the situation shown in Figure 12.10 can't occur.

This means that a forward marking based approach will correctly visit all mandatory nodes. And because we use forward marking we will also visit all optional nodes.

However there is a problem that prevents us from using this approach. In the first part of the paper we explained that "and"-prerequisite relationships can be used to branch a path. A branched path needs to be handled differently than a normal path because all branches need to be completed before it is possible to continue on the main branch.

This means that there are two special cases to consider:

1. If we start a query in a branch
2. If we end a query in a branch

If the forward scan starts in a branch of an "and"-relationship, we will potentially miss nodes that are accessible. Figure 12.11 shows an example of this situation. If we forward scan from Quest C to Dummy, we obtain the set of nodes that is marked green as being accessible. However, our query starts at Quest C which means that we can only assume that Quest B and A are completed. The status of Quest D can't be inferred from the query, which means that in our opinion it is best to mark it as an accessible node, because it is needed in order to be able to accept Quest E.

A similar problem occurs when the forward scan algorithm ends in a branched path. But instead of missing

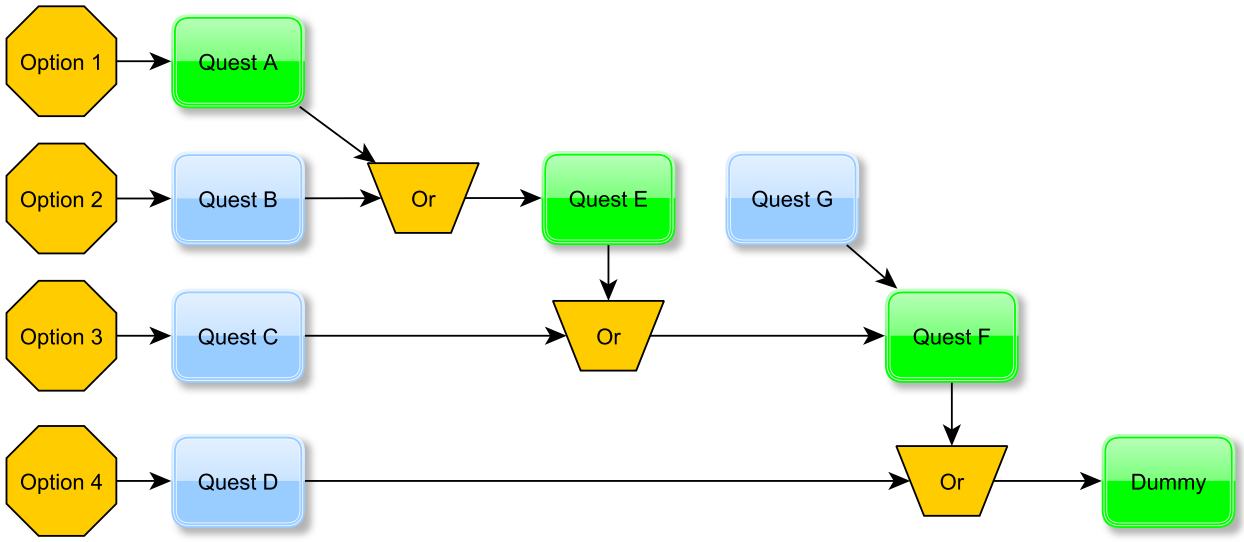

Figure 12.10: Impossible container graph structure. Quest g needs to have one in-edge

accessible nodes, it yields nodes that are not accessible. Normally we stop processing nodes when we encounter the “to”-node. However, because we end in a branched path, one of the other paths will continue to be processed. An example of this situation is shown in Figure 12.12.

In this example we forward scan the path from Quest A to Quest C. When we arrive at Quest B the path branches. The branch that starts with Quest C, will not be entirely processed because we only query to Quest C. However, the branch that starts with Quest D will entirely be processed because we don’t encounter the “to”-quest. This means that Quest E and Dummy are yielded as accessible nodes. This however is incorrect because they depend on Quest C. It is interesting to see that Quest D is marked as an accessible node, which is correct.

Both of these examples show that it is not possible to solve this problem by using a single forward scanning algorithm. We therefore constructed a different algorithm that will handle the previously explained problems correctly.

Forward & Backward Marking Algorithms
First we introduce two utility algorithms which are used for marking nodes. These algorithms are the “Mark-Forward” and ”MarkBackward”-algorithms. Both algorithms look very similar and that is no coincidence. As you might expect one of the algorithms uses forward marking in order to mark the nodes, while the other accomplishes the same but then by backward marking. Please note that the “markAction” parameter performs the actual marking. Furthermore, it is best to use a stack for the empty set defined in line one of the algorithm, because it will use a depth first marking strategy.

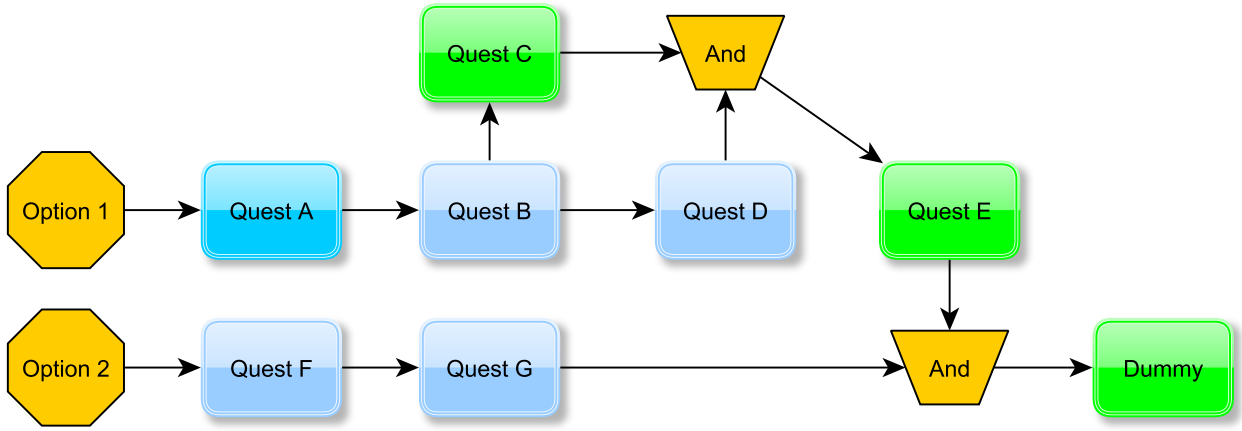

Figure 12.11: Forward scanning a query result leads to missing nodes if the query starts at a branch of the path

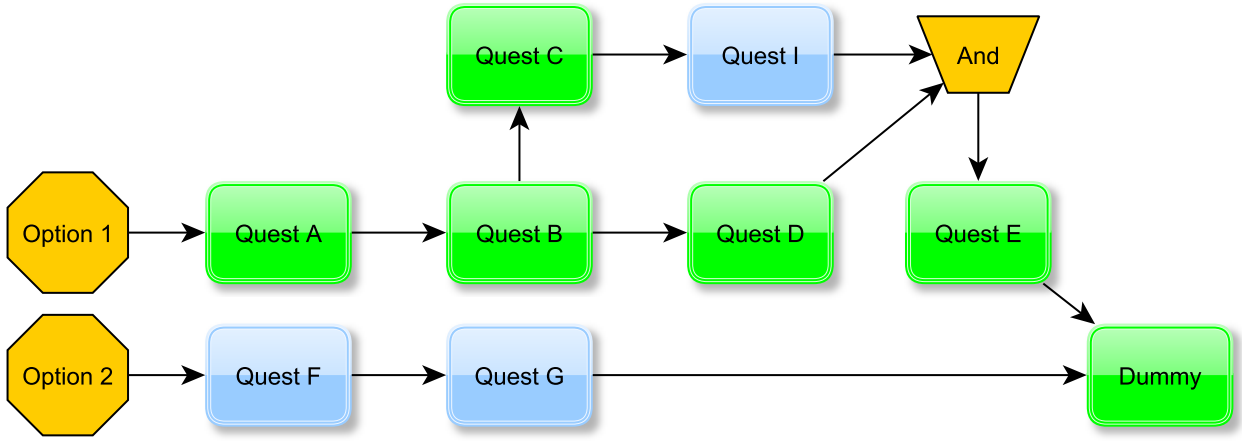

Figure 12.12: Forward scanning a query result leads to too much nodes if the query ends at a branch of the path

Algorithm MarkForward($fromNode, markAction$)
Input. $fromNode$, the node at which we start marking. $markAction$, the action to perform to mark the node

1. $S \leftarrow \emptyset$
2. $Push(fromNode, S)$
3. **While** $S \neq \emptyset$
4. $\quad currentNode \leftarrow Pop(S)$
5. $\quad$ Execute $markAction(currentNode)$
6. $\quad$ **For** $\forall outEdge \in currentNode_{out-edges}$
7. $\quad\quad Push(outEdge_{target}, S)$

Algorithm MarkBackward($fromNode, markAction$)
Input. $fromNode$, the node at which we start marking. $markAction$, the action to perform to mark the node

1. $S \leftarrow \emptyset$
2. $Push(fromNode, S)$
3. **While** $S \neq \emptyset$
4. $\quad currentNode \leftarrow Pop(S)$
5. $\quad$ Execute $markAction(currentNode)$
6. $\quad$ **For** $\forall outEdge \in currentNode_{in-edges}$
7. $\quad\quad Push(outEdge_{target}, S)$

The Container Marking Algorithm
The approaches discussed at the start of this chapter did not mark nodes, they just scanned forward or backward and yielded the results. With this new algorithm we will change the values of markers and not yield any results. The marker only contains a Boolean that indicates if the node is accessible or not. The initial value for that Boolean will be set to true, meaning that every node is acceptable. The main algorithm for marking a container for a query is shown below.

Algorithm MarkContainer($container, fromNode, toNode$)
Requirements. The accessible value for each node must be set to *true*.
Input. $fromNode$, the node at which we start marking. $markAction$, the action to perform to mark the node

1. $visitedOptions \leftarrow MarkBeforeInaccessible(container, fromNode)$
2. $unvisitedOptions \leftarrow \{\forall o : o \in container_{options} | o \notin visitedOptions\}$
3. **For** $\forall option \in unvisitedOptions$
4. $\quad ForwardMark(option, Action(\text{“}Accessible \leftarrow false''))$
5. $ForwardMark(fromNode, toNode, Action(\text{“}Accessible \leftarrow true''))$
6. $ForwardMark(toNode, Action(\text{“}Accessible \leftarrow false''))$
7. $toNode_{accessible} \leftarrow true$

In the first step of the algorithm we mark all nodes before the “from”-node as inaccessible. This is accomplished by calling the “MarkBeforeInaccessible”-algorithm, which will be explained later on in this section. At the second step we collect the options that have not been visited in the first step. These options will be used in steps 3 and 4 in order to mark them inaccessible. By forward marking the unvisited options we potentially have invalidated the marking data for nodes between the “from’- and “to”-node. This is because it is possible that one of those nodes can be reached from both a visited and an unvisited option.

In step five we restore the nodes between the “from”- and “to”-nodes, by forward marking them as accessible, starting at the “from”-quest. If we encounter the “to”-quest, the processing for that branches is stopped. However, it is possible that this strategy does not only mark the nodes between the “from”- and “to”-node. This is the case when the “to”-node is part of a branched path. In that case the algorithm will continue to mark the nodes that come after the “to”-node.

Therefore, in step six, we once again mark all nodes starting at the “to”-node. This time we will mark them inaccessible. Now the markings on the nodes in the container are almost correct. In the last step we set the “to”-node to be accessible, because it was marked inaccessible in the previous step. It is now up to the path algorithms to extract the correct path from the marked nodes.

We will now explain the “MarkBeforeInaccessible”- algorithm which is part of the algorithm to mark the nodes in a container. At the first step we check if the “from”-node is an OptionNode. If this is the case then only one option was visited, which means that we can return that node as the set of visited options. Then in the third step, if the “from”-node is not an OptionNode, we backward mark all nodes as being inaccessible. We reset the “from”-node to accessible because it was also marked inaccessible in the previous step. In steps 5 to 8, we check

which options have been visited, this can easily be accomplished because they are the only options that have been marked inaccessible at this point. We then return the set of visited options, so that they can be used in the main part of the algorithm.

Algorithm MarkBeforeInaccessible($container, fromNode$)
Input. $container$, the container in which we query. $fromNode$, the node at which the query starts

1. **If** $fromNode$ is Option
2. return $fromNode$
3. $MarkBackward(fromNode, Action(''Accessible \leftarrow false''))$
4. $fromNode_{accessible} \leftarrow true$
5. $visitedOptions \leftarrow \emptyset$
6. **For** $\forall option \in container_{options}$
7. Add *option* to *visitedOptions*
8. return *visitedOptions*

Chapter 13

Creating a Choice Graph

This chapter will explain how a choice graph can be created from an extended quest graph. The algorithms described in this section are created with the assumption that they will be used on a Choice Graph that adheres to the scope that is defined in chapter 10. Please note that the described algorithms indicate collections via an empty set symbol. In our opinion it is best to use stacks in most cases because they enable a depth first search strategy. If we prefer another type of set over a stack we will explain which type this is in the paragraphs that explain the algorithms.

The "GenerateChoiceGraph"-algorithm, which is shown below, is the main algorithm for transforming a quest graph into a choice graph. Because the algorithm changes the structure of the extended quest graph it is best to first clone it. On this clone we will then execute the "CreateChoiceGraph"-algorithm which will turn it into a choice graph.

Algorithm GenerateChoiceGraph(G)
Input. An extended quest graph QG
Output. A choice graph CG

1. $CG \leftarrow Clone(QG)$
2. $CreateChoiceGraph(CG)$
3. return CG

The "CreateChoiceGraph"-algorithm performs the actual transformation from an extended quest graph to a choice graph. The algorithm first searches for choices that can be replaced by using the "ScanForward"-algorithm. If a choice is found the "FindMergePoint"-algorithm is called which tries to find the node in which all options of the choice merge. The last step is to convert the entire choice into a container which is accomplished by calling the "CreateContainer"-algorithm.

Algorithm CreateChoiceGraph(CG)
Input. A quest graph that will be transformed into a choice graph CG

1. $S \leftarrow \emptyset$
2. **For** $\forall node \in CG_{Nodes}$
3. **If** $|node_{inedges}| = 0$
4. $Push(node, S)$
5. **While** $S \neq \emptyset$
6. $currentNode \leftarrow Pop(S)$
7. $choice \leftarrow ScanForward(currentNode, S)$
8. **If** $choice = NULL$
9. $continue$
10. $mergeNode \leftarrow FindMergePoint(choice)$
11. $CreateContainer(CG, choice, mergeNode)$
12. $Push(mergeNode, S)$

We previously explained that the "ScanForward"-algorithm is used to find choices that have not yet been transformed into containers. The algorithm performs a depth first search in order to find such choices. The algorithm will continue to search for a choice until one is found or until every node that comes after the starting node has been processed.

Algorithm ScanForward($node, S$)
Input. $node$, a node to start scanning from. S a stack that contains all nodes that need to be scanned in the future
Output. A choice node if one is found else NULL

1. **If** $node$ is a $ChoiceNode$
2. return $node$
3. $currentNode \leftarrow node$
4. **While** $currentNode \neq NULL$
5. $nodeOutEdges \leftarrow currentNode_{outedges}$
6. **If** $|nodeOutEdges| = 0$
7. return NULL
8. $currentNode \leftarrow nodeOutEdges_1$
9. **If** $|nodeOutEdges| > 1$
10. **For** $\forall edge \in nodeOutEdges_{2,3,\ldots,n-1}$
11. $Push(edge_{target}, S)$
12. **If** $currentNode$ is a $ChoiceNode$
13. return $currentNode$
14. return $NULL$

The "FindMergePoint"-algorithm attempts to find the node that is the merge point for a choice. The algorithm performs a depth first forward search in order to find the merge point. While it is performing this search it checks for each node if it is the merge point for the choice via the "IsMergePointFor"-algorithm. If the merge point it is found the algorithm will assume that the prerequisite relationships for that merge points are correct. In a later stage we will validate if the relationship, defined in the prerequisites of the merge point node, indeed is correct.

Algorithm FindMergePoint($node$)
Input. $node$, a ChoiceNode for which the merge point needs to be found
Output. A node in which $node$ merges, NULL otherwise

1. $S \leftarrow \emptyset$
2. $option \leftarrow node_{options_1}$
3. $Push(option, S)$
4. **While** $S \neq \emptyset$
5. $\quad currentNode \leftarrow Pop(S)$
6. $\quad$ **If** $IsMergePointFor(currentNode, choiceNode)$
7. $\quad\quad$ return $currentNode$
8. $\quad$ **For** $\forall edge \in currentNode_{out-edges}$
9. $\quad\quad Push(edge_{target}, S)$
10. return $NULL$

The "IsMergePointFor"-algorithm is a simple backtracking algorithm that marks all nodes that it visits as visited. After visiting all nodes in the set, it checks if all of the options for the choice node have been visited. If this is the case then the "orNode" is the merge point.

Algorithm IsMergePointFor($orNode, choiceNode$)
Requirements. $Visited$ must be set to $false$ for each node
Input. $orNode$ the node to check if it is the merge point. $choiceNode$, the choice for which we seek the merge point
Output. A boolean indicating if $orNode$ is the merge point for $choiceNode$

1. $S \leftarrow \emptyset$
2. $Push(orNode, S)$
3. **While** $S \neq \emptyset$
4. $\quad currentNode \leftarrow Pop(S)$
5. $\quad$ **If** $currentNode_{visited} = true$
6. $\quad\quad$ continue
7. $\quad currentNode_{visited} \leftarrow true$
8. $\quad$ **For** $\forall edge \in currentNode_{in-edges}$
9. $\quad\quad source \leftarrow edge_{source}$
10. $\quad\quad$ **If** $source$ is a $Option$
11. $\quad\quad\quad source_{visited} \leftarrow true$
12. $\quad\quad$ **If** $source \neq choiceNode$
13. $\quad\quad\quad Push(source, S)$
14. **For** $\forall option \in choiceNode_{Options}$
15. $\quad$ **If** $option_{visited} = false$
16. $\quad\quad$ return $false$
17. return $true$

The "CreateContainer"-algorithm is used to convert a choice into a container. In the first nine steps the choice and it's end-nodes are disconnected from the original graph. The set of nodes that represents the end-nodes of a choice is not the same set as all nodes that come before the merge point. The reason for this is that multiple choices can merge in one merge point as is shown in Section 11 in cases four and five. We therefore call the "FindEndNodesForOptions"-algorithm which searches for the nodes that represent the end-nodes of each option. Now that we have identified the nodes that make up the end of the choice it is possible to validate if the

merge point is configured correctly. Because a prerequisite in fact is an expression tree we can check if the part of the expression that is involved with this choice is correct. We will not provide an algorithm to accomplish this because it depends too much on the actual implementation.

Then in steps ten to fourteen we create and attach the container that will represent the current choice in the graph. However one problem still remains, the part of the quest graph that is disconnected in the first nine steps needs to be attached to the container. This is accomplished in step fifteen which calls the "CreateSubGraph"-algorithm in order to properly fix the graph. This means that all nodes that were part of the original graph will be transferred to the graph in the container. Please note that we do not explain the "CreateSubGraph"-algorithm because it is very straightforward.

At this point the graph of the container is valid which means that we can process it. We execute the "CreateChoiceGraph"-algorithm in order to transform the inner choices of the current choice/container into containers themselves. After converting all inner choices the container is almost completely processed. The only step left is to detect possible subsets of converging options in the graph of the container. This step is performed last because it is more efficient to do this after the containerization because it then has to visit a much lower number of nodes.

Algorithm CreateContainer($graph, choice, mergeNode$)
Input. $graph$, the QuestGraph that is turned into a ChoiceGraph. $choice$, the choice for which we are going to make a container. $mergeNode$ the node that is the merge point for $choice$

1. $choiceInNodes \leftarrow choice_{in-edges}$
2. $optionsEndNodes \leftarrow FindEndNodesForOptions(choice, mergeNode)$
3. "Validate if the endNodes for each option are correctly expressed in the prerequisite"
4. $RemoveInEdges(graph, choice)$
5. $RemoveOutEdges(graph, choice)$
6. **For** $\forall optionEndNodes \in optionsEndNodes$
7. **For** $endPoint \in optionEndNodesend - points$
8. $RemoveOutEdges(graph, endpoint)$
9. $RemoveNode(graph, choice)$
10. $container \leftarrow Container(choice, optionEndNodes)$
11. $AddNode(graph, container)$
12. **For** $\forall node \in choiceInNodes$
13. $AddEdge(graph, node, container)$
14. $AddEdge(graph, container, mergeNode)$
15. $CreateSubGraph(container, graph)$
16. $CreateChoiceGraph(container_{sub-graph})$
17. $FindSubSets(container)$

The "FindEndNodesForOptions"-algorithm is used to find the end nodes for each option. This are nodes on which the merge node depends. As we previously explained, not all predeccessing nodes are actually part of the set of end nodes because it is possible that multiple choices merge in the same merge point. We therefore perform a forward depth first search in order to find all the end nodes for each of the options. The reason why we detect such a set for each option is because we can use this information to validate the correctness of the prerequisite of the merge node.

Algorithm FindEndNodesForOptions($choice, mergeNode$)
Input. $choice$, the choice which contains the options for which we are going to find the end nodes. $mergeNode$ the node that is the merge point for $choice$
Output. A set of option-endpoints

1. $resultset \leftarrow \emptyset$
2. $options \leftarrow choice_{options}$
3. **For** $\forall option \in options$
4. $optionEndPoint_{Option} \leftarrow option$
5. $S \leftarrow \emptyset$
6. $Push(option, S)$
7. **While** $s \neq \emptyset$
8. $currentNode = Pop(S)$
9. **If** $currentNode = mergeNode$
10. $continue$
11. **For** $\forall outEdge \in currentNode_{out-edges}$
12. **If** $outEdge_{target} = mergeNode$
13. $Add(currentNode, optionEndPoint_{endpoints})$
14. **Else**
15. $Push(outEdge_{target}, S)$
16. $RemoveDuplicates(optionEndPoint_{endpoints})$
17. Add $optionEndPoint$ to $resultSet$
18. return $resultSet$

The last algorithm that is part of the creation process of the choice graph is the "FindSubSets"-algorithm. As the name implies this algorithm is used to find subsets of options that merge before the merge point. This information might not seem very useful at this point, but in chapter 17 we will use this information in order to optimize the query-process of the choice graph.

The algorithm first iterates over all nodes in the graph of the container. The nodes that have a prerequisite with an "or"-relationship are added to a stack. After a node is added to the stack the algorithm backtracks in order to find the options that lead to it. If the potential merge point leads to more than one option, as it should do according to the rules, then it is added to the set of subsets for the containers. This process will continue until all subsets have been found.

Algorithm FindSubSets(*container*)
Requirements. $Visited$ must be set to $false$ for each node
Input. *container*, the ContainerNode in which we are going to search for subsets

1. $container_{subsets} \leftarrow \emptyset$
2. $containerGraph \leftarrow container_{subgraph}$
3. **For** $\forall node \in containerGraph_{nodes}$
4. $S \leftarrow \emptyset$
5. **If** $ContainsOrRelationship(node)$
6. $Push(node, S)$
7. **While** $S \neq \emptyset$
8. $current \leftarrow Pop(S)$
9. **If** *current* is a $OptionNode$
10. $current_{visited} \leftarrow true$
11. **For** $\forall edgeincurrent_{in-edges}$
12. $Push(edge_{source}, S)$
13. $visitedOptions \leftarrow \{\forall o : o \in container_{options} | o_{visited} = true\}$
14. **If** $|visitedOptions| > 1$
15. $newSubset_{merge-point} \leftarrow node$
16. **For** $\forall option \in visitedOptions$
17. $Add(option, newSubset_{Options})$
18. $Add(newSubset, container_{subsets})$

Chapter 14

Detecting Scope Violations

In the previous section we explained how a choice graph can be constructed from an extended quest graph. The described algorithms assume that the extended quest graph adheres to the defined scope. If this is not the case the algorithm will fail or yield unexpected results. We therefore devote a section on explaining how these scope violations can be detected.

Please note that the construction algorithm for the quest graph does check if the merge point that is found for a choice has a proper "or"-relationship. Obviously an "and"-relationship is not allowed for merging different options.

14.1 All options merge in one point

The first scope limitation states that all options of a choice need to merge in one point. It is possible to violate this rule by not letting a subset of options from a choice merge. An example of this situation is shown in Figure 14.1. In this case Choice 2 doesn't merge as it supposed to.

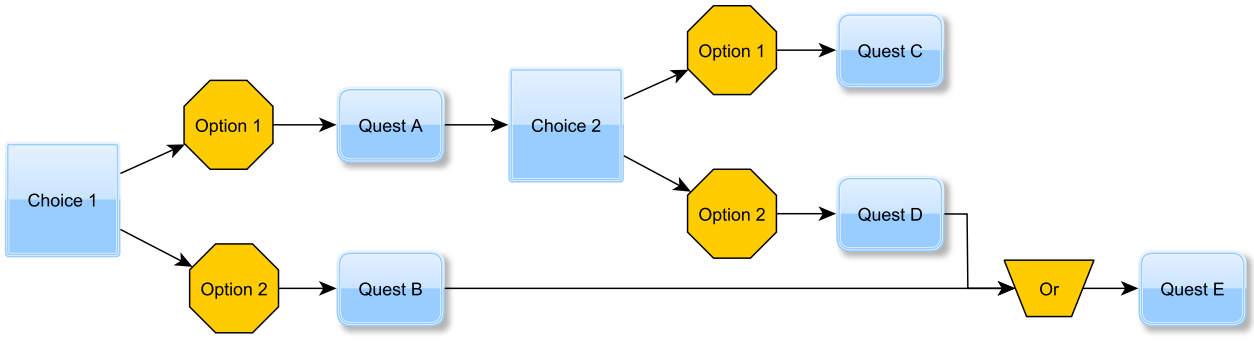

Figure 14.1: Example of a choice that does not merge

This problem can easily be detected when the extended quest graph is transformed into a choice graph. During this process the "FindMergePoint"-algorithm is called to find the merge point. If the choice does not merge at any point, then the algorithm will yield a null result. If a game designer would use a tool to create quest and choice graphs, then it is possible to provide him with information regarding this problem. While it is not possible to provide a solution for the problem, we at least are able to point the game designer to the choice that

causes the problems.

14.2 “Or”-relationships are only used to merge entire options

We also state that one rule demands that “or”-relationships are only used to merge entire options. This rule can be violated in two ways. First we can use the previously discussed common path approach, which uses two “or”-relationships to merge a choice. The first “or” however does not merge the entire choice and therefore is a violation of the rule. Figure 14.2 shows an example of this case. Option 1 and option 2 appear to merge in Common Path however there are outgoing edges from quests B and D that prevent the entire choice from merging.

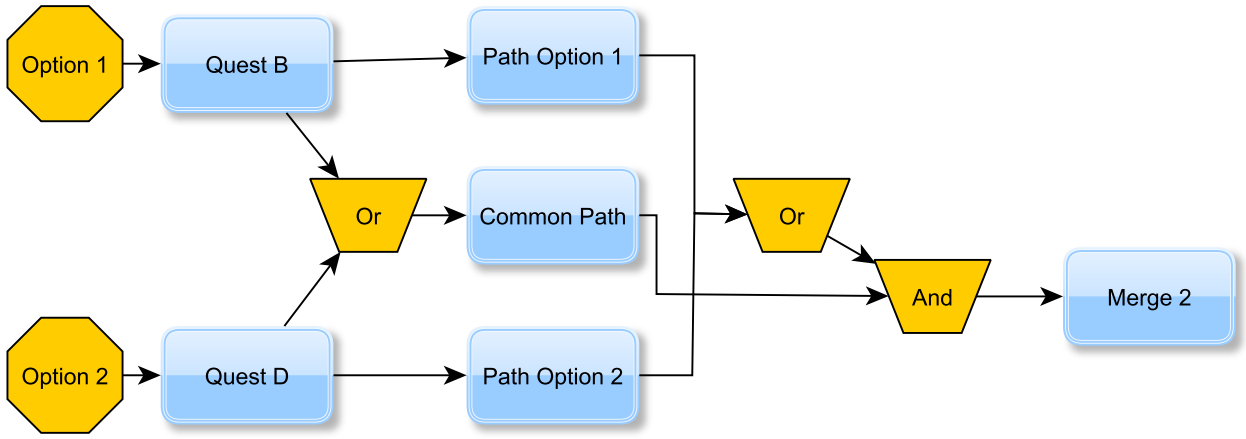

Figure 14.2: Example of a choice with a common path

This case can easily be detected by performing an extra check when the merge point for a choice is found. We can use a simple forward scan to check if all paths reach the merge node. If this is the case then the rule is not violated. However, if not all paths reach the merge node, it doesn’t automatically mean that this rule is violated. It might be the case that a path is optional, in which case it shouldn’t be connected to the nodes after the merge point. In order to assess that a path is actually optional, we first need to check if it doesn’t break the fourth rule that states that “Dependencies may not cross over choice or option boundaries”. This will be explained in Section 14.4 of the paper. If this is not the case, then we can assume that it actually is a proper optional path, which means that we can ignore it.

The second case in which an “or”-relationship can be misused is if it is not used in conjunction with a choice. In that case it is used to create a special relationship between quests as is shown in Figure 14.3. This can easily be detected by verifying that only the actual merge points contain an “or”-relationship. Such a step can be taken after the extended quest graph is transformed into a choice graph.

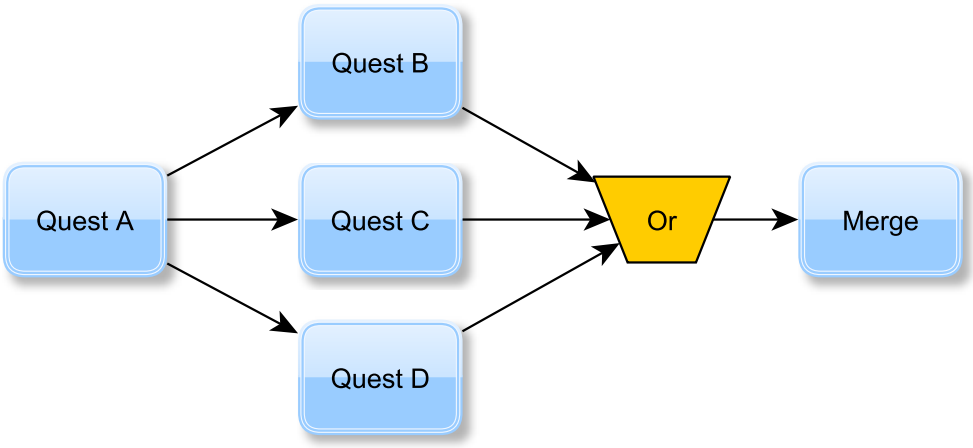

Figure 14.3: Example of a structure of quests with an “or”-relationship

14.3 All nodes belonging to a choice must at least have one in-edge

The third constraint is the one that is the most easy to verify. Each node that is part of a choice must at least have one in-edge. This can be verified before or after the choice graph construction by checking if all nodes in all containers have at least one in edge. If one of the nodes does not have one in-edge the rule is violated.

14.4 Dependencies may not cross over choice or option boundaries

In Section 14.2 we explained that we need to check optional paths for violations of the “dependencies may not cross over choice or option boundaries”-rule. This section will explain how such violations can be detected.

To verify if a path is actually an optional path we need to check that it does not merge at some point in the future with any possible path that originates from the merge point. An example of such case is shown in Figure 14.4. It can be verified by checking if the end nodes for the potential optional path, quest E and F in the figure, are reachable from the actual merge point, quest D. Dijkstra’s algorithm can be used in order to check if a path between D and both end nodes exists. In this case the construction is valid because there exists no path between the merge node and the end nodes.

Figure 14.5 shows a slight variation on the previous example. In this case the optional path is constructed incorrectly by merging the optional path with the main path. The figure shows that it is possible to use either an “and”- or an “or”-relationship in order to potentially violate this rule. If we assume that the violating relationship is of the type “and” then it is clear that this is a scope violation. As a result of the “and”-relationship the optional path has become mandatory which means that quest F must be completed in order to be able to accept quest H. This means that in this case the end points for the optional path are quests E and H. Quest H can be reached from quest D, which indicates that the rule has been broken.

If we assume that the violating relationship is of type “or” then a different problem emerges. In this case we

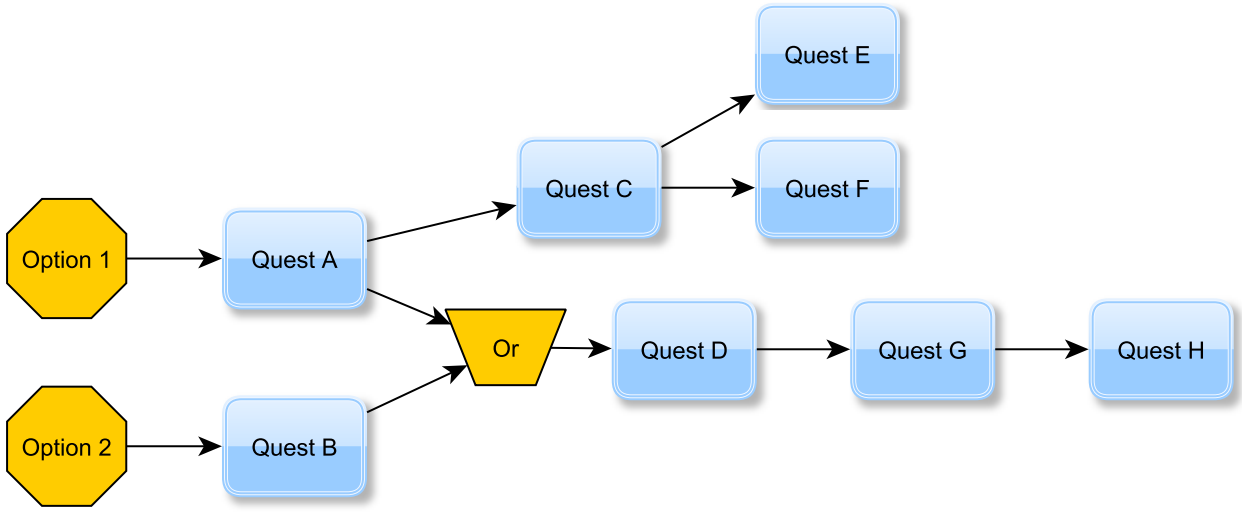

Figure 14.4: Example of a correctly structured optional path

don't violate the rule that states that "Dependencies may not cross over choice or option boundaries". However, we do violate the rule that states that "or"-relationships may only be used to merge entire options. At this point it is ambiguous how to proceed because we don't know the intentions of the game designer. It is up to the designer to decide which of the both "or"-relationships needs to be removed in order to solve this problem.

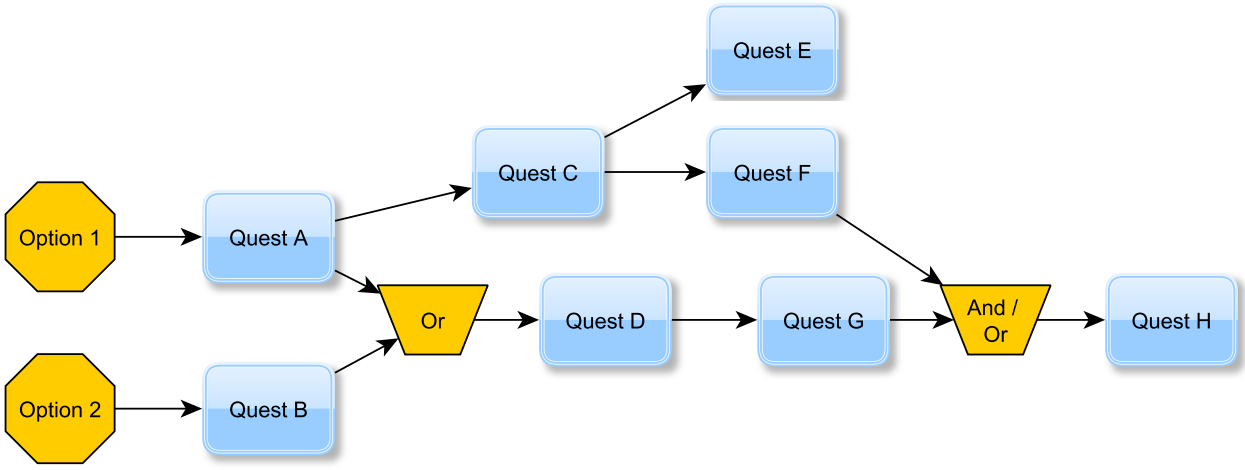

Figure 14.5: Example of an incorrectly structured optional path

Chapter 15

Creating a Topological Ordering

In the first part of the paper we stated that the "Topological Ordering"-algorithm, described first by Tarjan[Tar76], was used to create a topological ordering of a graph. This time however it is important to understand that the algorithm will be used on the graphs that are stored in the containers. This means that the algorithm needs to understand the markings which we explained Section 12.3. Therefore, the algorithm is slightly adapted in order to handle the accessibility attribute.

The main algorithm for creating topological orderings, "FindTopologicalOrdering", is shown below. First an empty set is created which will hold the ordered set of nodes. Then we obtain all accessible nodes with a zero out degree. These nodes are either extracted from the graph or from a set of accessible nodes. Using the set of accessible nodes is a small optimization, which we will explain later. In steps 3 and 4 we will then call the "Visit"-algorithm for each of those nodes. Finally we return the ordered set of nodes and the algorithm has been successfully executed.

Algorithm FindTopologicalOrdering($graph, accessibleNodes$)
Input. $graph$, the graph which contains the nodes. $accessibleNodes$, optional, the complete set of accessible nodes
Output. $ordering$, a ordered set of nodes that are ordered in a valid topological ordering

1. $ordering \leftarrow \emptyset$
2. $nodesZeroOutDegree \leftarrow GetNodesZeroOutDegree(graph, accessibleNodes)$
3. **For** $\forall node \in nodesZeroOutDegree$
4. $\quad Visit(node, ordering)$
5. return $ordering$

The "Visit"-algorithm is used to create the correct ordering of a graph. The algorithm first checks if the node was already visited by the "Topological Sorting"-algorithm. If this is not the case then the node will be visited. This means that first the node's state will be set to visited. Then we check if each node before the current node is accessible. If this is possible the "Visit"-algorithm will be called on those nodes. This means that this algorithm will recursively create the ordering, by first adding all nodes before the current node. Finally, in the last step the current node is added to the ordering.

Algorithm Visit($node$, $ordering$)
Requirements. $Visited$ must be set to $false$ for each node
Input. $node$, the node to visit. $ordering$ the current ordered set of nodes

1. **If** $node_{visited} = false$
2. $node_{visited} \leftarrow true$
3. **For** $\forall edge \in node_{in-edges}$
4. $inNode \leftarrow edge_{source}$
5. **If** $inNode_{accessible}$
6. $Visit(inNode, ordering)$
7. Add $node$ to $ordering$

A simple but important step of the main algorithm is the step in which we obtain the set of nodes with a zero out degree. Normally this set is obtained by iterating over all nodes in the graph and checking if each node has a zero out degree and is accessible. In our case we extended the algorithm, so that another set, the accessible nodes set, can be passed to the algorithm. This is a slight optimization because it allows the algorithm to check a smaller subset, of the nodes contained in the graph, for zero out degree. Because we will mostly use the "Topological Ordering"-algorithm with the query algorithms, it is perfectly possible to create such a set when we "handle" nodes during the query algorithm. However, if such a set is not provided, we will fall back to the original graph.

Algorithm GetNodesZeroOutDegree($graph, accessibleNodes$)
Input. $graph$, the graph which contains the nodes. $accessibleNodes$, optional, the complete set of accessible nodes
Output. $zeroOutSet$, a set of nodes with zero out edges

1. $zeroOutSet \leftarrow \emptyset$
2. **If** $accessibleNodes \neq \emptyset$
3. **For** $\forall node \in accessibleNodes$
4. **If** $HasZeroOut(node)$
5. $Add(node, zeroOutSet)$
6. return $zeroOutSet$
7. **Else**
8. **For** $\forall node \in graph_{nodes}$
9. **If** $node_{accessible}$ and $HasZeroOut(node)$
10. $Add(node, zeroOutSet)$
11. return $zeroOutSet$

For completeness, we will also explain the algorithm that checks if a node has a zero out degree. The algorithm is very simple. It checks if each of the nodes that comes directly after the current node is accessible by the algorithm. If one of the nodes is accessible the current node does not have a zero out degree.

Algorithm HasZeroOut($node$)
Input. $node$, the node to check for zero out edges
Output. A boolean indicating if $node$ has zero out edges

1. **For** $\forall edge \in node_{out-edges}$
2. $targetNode \leftarrow edge_{target}$
3. **If** $targetNode_{accessible}$
4. return $false$
5. return $true$

Please note that these algorithms have been slightly simplified. All of the involved nodes are checked if they are accessible. The boolean, that indicates the accessibility, is part of the marker that is explained in Section 12.3. The boolean "visited", is part of the marker for the "Topological Ordering"-algorithm. Each algorithm places its own marker data on a node. This is to ensure that something like a "visited"-value can't be used simultaneously by multiple algorithms.

Chapter 16

Detecting Cycles

In the first part of the paper we described how cycles can be detected in a normal quest graph. Using the same approach on an extended quest graph or on a choice graph is undesirable because cloning and destroying a graph is very inefficient. In this section we will show an adapted version of the algorithm that can detect cycles in an extended quest graph. We will however not discuss a method that can detect cycles in a choice graph. This is because it is important to know if cycles are present in an extended quest graph before building the choice graph.

The algorithm below shows how cycles can be detected in an extended quest graph. In this case we will use the visited markers to indicate if nodes have been visited.

Algorithm DetectCycles(G)
Requirements. All nodes must be marked as not visited.
Input. A graph to check for cycles G
Output. A set of nodes that is involved in a cycle

1. $nodesMarked \leftarrow true$
2. **while** $nodesMarked$
3. $\quad nodesMarked \leftarrow false$
4. $\quad$ **For** $\forall node \in G_{nodes}$
5. $\quad\quad$ **If** $|node_{in-edges}| = 0$ or $|node_{out-edges}| = 0$
6. $\quad\quad\quad node_{visited} \leftarrow true$
7. $\quad\quad\quad nodesMarked \leftarrow true$
8. $nodesInCycle \leftarrow \emptyset$
9. **For** $\forall node \in G_{nodes}$
10. $\quad$ **If** $node_{visited} = false$
11. $\quad\quad Add(node, nodesInCycle)$
12. return $nodesInCycle$

Please note that this algorithm is terribly inefficient due to the constant iteration over all nodes in the graph. In the original algorithm the nodes that are visited are immediately removed from the graph. This reduces the size of the graph with each iteration. This implementation does not share that same characteristic. However, it is not very hard to create an algorithm that has that characteristic. It is for example possible to add all nodes of the graph to a hashmap. Instead of iterating over the graph we will now iterate over the contents of the hashmap. Then when a node is to be visited we remove it from the hashmap. This way we have the same characteristic as the original algorithm but at a cost of more memory.

Chapter 17

Querying the Choice Graph

In order to execute a path query efficiently it is important to know which options need to be chosen in order to get from the start to the end of the query. So for both the shortest and longest path we need to find a way to detect the options that lead to that path. A naive approach would be to just extract all choices on the path and try to brute force the optimal solution. Fortunately, the previously introduced containers can be used to solve this problem.

In the next section we first explain the different types of queries that are encountered when a container is queried. Then we continue by explaining how these types of queries can be executed on a container for the shortest and longest path. After which we explain how we can query the choice graph by combing the results of multiple queries in order to construct the total path for a query. We conclude this chapter with a section on optimizing the queries by pre-calculating path data.

17.1 Possible Queries

In order to answer queries that are executed on a choice graph, we first need to consider in how many ways a container can be involved in a query. We have extracted the following four cases:

1. **The query completely contains the container**
   If the container is queried as if it is a single entity, we need to consider which of the options matches the query the best. This means that we need to find this option and the path that is related to that option in order to answer the query efficiently. Because there are multiple options available we need multiple queries to solve the problem.

2. **The query starts at another container but ends in the current container**
   If a query ends somewhere in the container, we basically are in the same situation as case 1. The only difference is that instead of all options being a potential answer, in this case only a subset of options can lead to the desired result. For an example look at Figure 17.1, if we query towards “Quest E”, we will only find “Option 1 and 2” being viable options.

3. **The query starts in the container but ends at another container**
   A totally different case is when the query starts in the current container and ends at another. For these cases there is no optimal option to find, because an option was already implicitly chosen. So we only need to find the path that leads out of the container. An example of this case is if we query from “Quest F” to the ”Dummy”-node, as shown in Figure 17.1.

4. **The query begins and ends in the same container**
   The last case is very similar to case 3. Instead of ending at the “dummy”-node this case will end somewhere before that node. This means that in this case we also won’t have to find an optimal option to choose, but only the path that leads towards the node at which the query ends. An example of this case is when we query from “Quest B” to ”Quest F”.

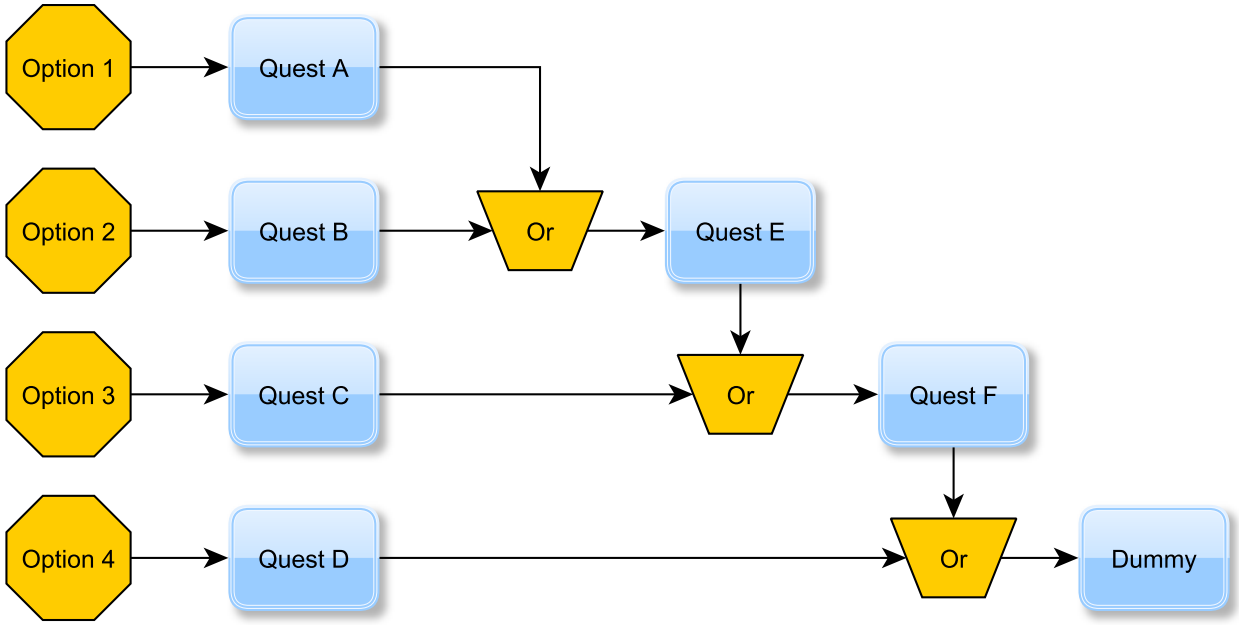

Figure 17.1: Part of a choice

Please note that we ignore the fact that a container can contain sub-containers, because that is not relevant at this point. The goal of this section is to explain which types of queries can be executed on a container.

17.2 Container Query Limitations

Queries executed on a container need to follow the same rules as the queries that are executed on the normal quest graph. Therefore, we remind you that it is illegal to query for a path from an optional quest towards any other quest that is not directly reachable from that quest. An example of such situation is shown in Figure 17.2. Executing a query from ”Quest E” to ”Dummy”, is not possible because they are not directly connect. Would there have been an edge between Quest E and Dummy, then the query would have been legal.

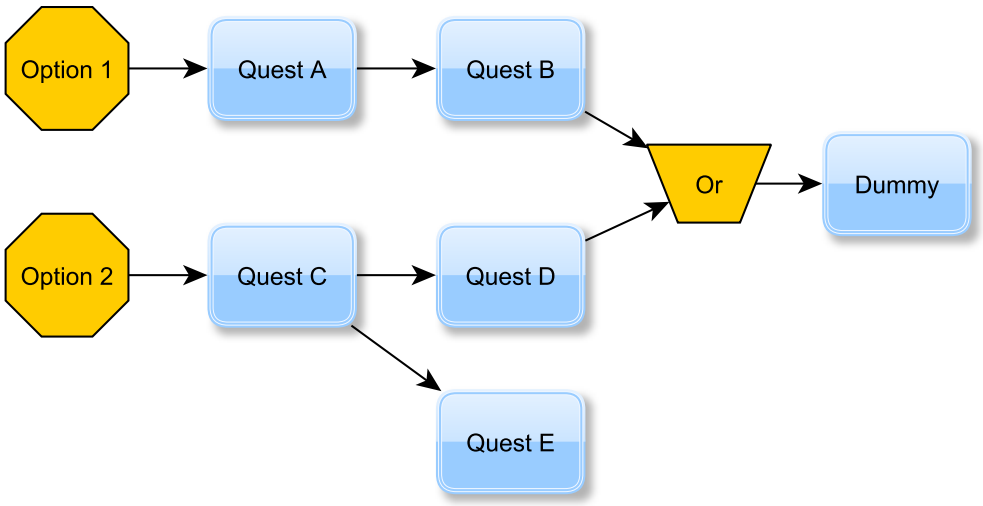

Figure 17.2: Example of a choice in which is illegal to query from quest E to any other quest

17.3 Solving Shortest Path Queries in Containers

In this section we will explain how we can extract the shortest path from a container that already has been marked with the algorithm mentioned in Section 12.3. After we have explained the algorithm, we will give concrete examples of how the queries, mentioned in Section 17.1, can be solved.

Shortest Path Algorithm
The "Shortest Path"-algorithm with markers bares close resemblance to the original "Shortest Path"-algorithm. The only differences are that now we check if the nodes are accessible and that we collect all the nodes on the path in order to make it more efficient to find the topological ordering of the shortest path. The new version of the algorithm is shown below. Please note that the empty set defined in line one of the algorithm in fact is a queue.

Algorithm FindShortestPath($graph$, $fromNode$, $toNode$)
Requirements. $Visited$ must be set to $false$ for each node
Input. $graph$, that contains the nodes for the query. $fromNode$, the node at which the query starts. $toNode$, the node at which the query ends
Output. A set of nodes that collectively form the shortest path

1. $Q \leftarrow \emptyset$
2. $nodesOnPath \leftarrow \emptyset$
3. $Add(toNode, Q)$
4. **While** $Q \neq \emptyset$
5. $currentNode \leftarrow Dequeue(Q)$
6. **If** $currentNode_{visited} = false$ and $currentNode_{accessible}$
7. $currentNode_{visited} \leftarrow true$
8. $Add(currentNode, nodesOnPath)$
9. **If** $currentNode \neq fromNode$
10. **For** $\forall inEdge \in currentNode_{in-edges}$
11. $inNode \leftarrow inEdge_{source}$
12. **If** $inNode_{visited} = false$ and $inNode_{accessible}$
13. $Add(inNode, Q)$
14. $fromNode_{accessible} \leftarrow false$
15. $toNode_{accessible} \leftarrow false$
16. $orderedPath \leftarrow FindTopologicalOrdering(graph, nodesOnPath)$
17. $AddFirst(fromNode, orderedPath)$
18. $AddLast(toNode, orderedPath)$
19. return $orderedPath$

Please note that we remove the "from"- and "to"-node from the set of accessible nodes. This is done in order to assure that they are in fact the first and last node of the topologically ordered path. Furthermore, note that the length of the path is not always equal to the number of nodes on the shortest path. If the shortest path contains "container"-nodes then the path will be much longer. We will now explain a naive approach that will calculate the length of the shortest path. This approach needs to calculate the length for each container at a lower level of the Choice Graph in order to find the correct path length. In a later section we will explain how pre-calculating sets of options will speed up the querying considerably. However let us first explain the naive approach.

Algorithm NaiveCalculateLength($path$)
Input. $path$, a set of nodes that represents a path for which we need to get the length
Output. An integer that equals the length of $path$

1. $length \leftarrow 0$
2. **For** $\forall node \in path$
3. **If** $node$ is a $Quest$
4. $length \leftarrow length + 1$
5. **If** $node$ is a $Container$
6. $innerPath \leftarrow$ "Find Shortest path in container (all options possible)"
7. $length \leftarrow length + NaiveCalculateLength(innerPath)$
8. return $length$

We first set the length to zero. We iterate over all nodes on the shortest path. If the node is a quest-node then

it has the length of one. So we add one to the length. If the node is a container, then we first need to find the shortest path in the container by calculating the length of each option. This means that if the options contain containers, these too need to be calculated. If we calculate multiple paths on the choice graph then the above mentioned approach is rather inefficient due to the duplicate calculations. What we don't show in this example is that you actually should extract the nodes from the container in order to get the nodes for the shortest path. In a later section we will explain how we can use pre-calculations in conjunction with path extraction algorithm in order to obtain a correct set of nodes for a path.

17.3.1 Solving the four cases

In this section we will explain how we can solve each of the four query cases by using the "Shortest Path"-algorithm in conjunction with the marker based approach.

The query completely contains the container

If the container is completely contained within the query, then all options for that container are considered to be a viable solution. This means that we need to perform shortest path queries for all of the available options. We can find the optimal option with the following approach:

Algorithm FindShortestPathInContainer(*container*)
Input. container, The container for which we want to find the shortest path
Output. The shortest path

1. $pathLength \leftarrow$ "maximal integer value"
2. $path \leftarrow NULL$
3. **For** $\forall options \in container_{options}$
4. $MarkContainer(container, option, container_{dummy})$
5. $shortestPath \leftarrow FindShortestPath(container_{subgraph}, option, container_{dummy})$
6. $length \leftarrow NaiveCalculateLength(shortestPath)$
7. **If** $length < pathLength$
8. $length \leftarrow pathLength$
9. $path \leftarrow shortestPath$
10. return *path*

We will illustrate this case with the example that is shown in the figures below. Figure 17.3 shows the content of a container, while figures 17.4 and 17.5 show the markings on the nodes, which are applied in steps two until four of the marking algorithm. If we would use the "Shortest Path"-algorithm on both those graphs we would end up with the result that "Option 1" is the shortest path.

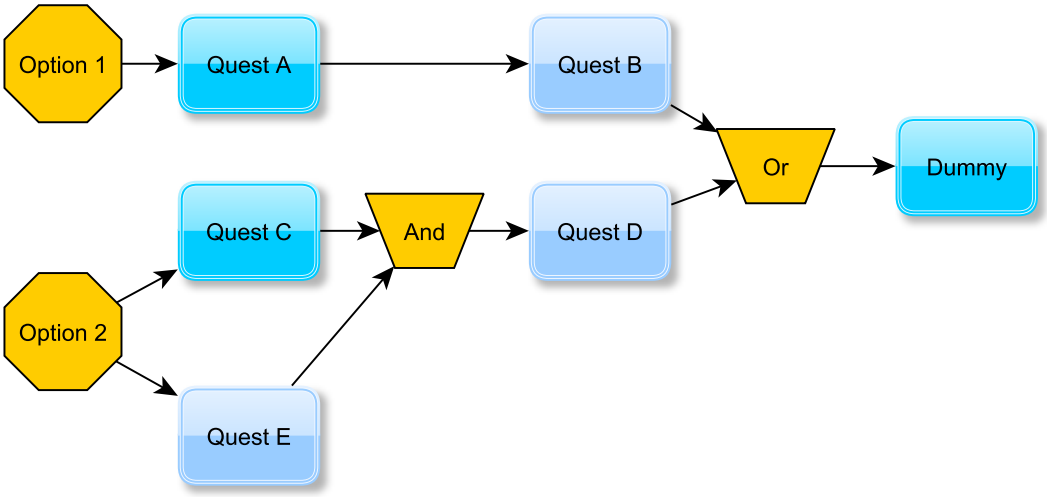

Figure 17.3: Content of a container for this example

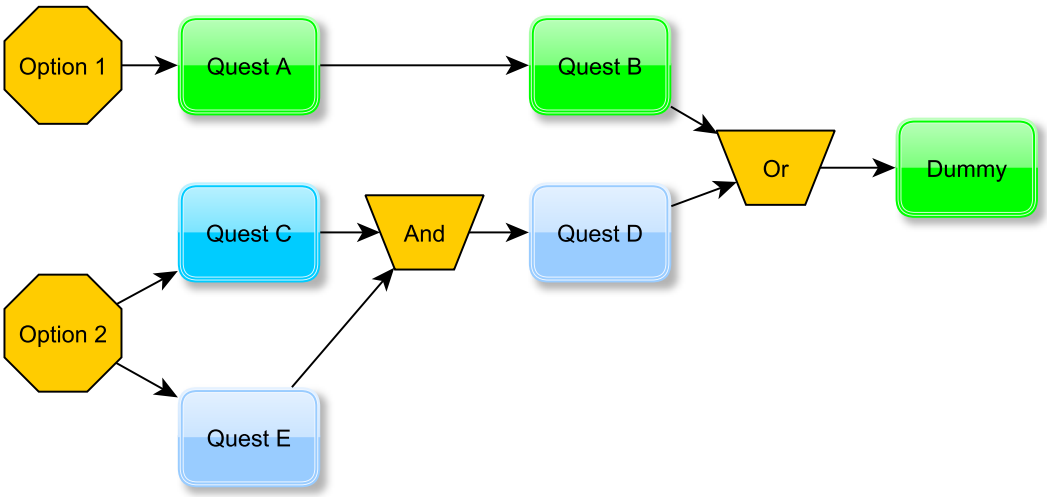

Figure 17.4: Marked path for option one

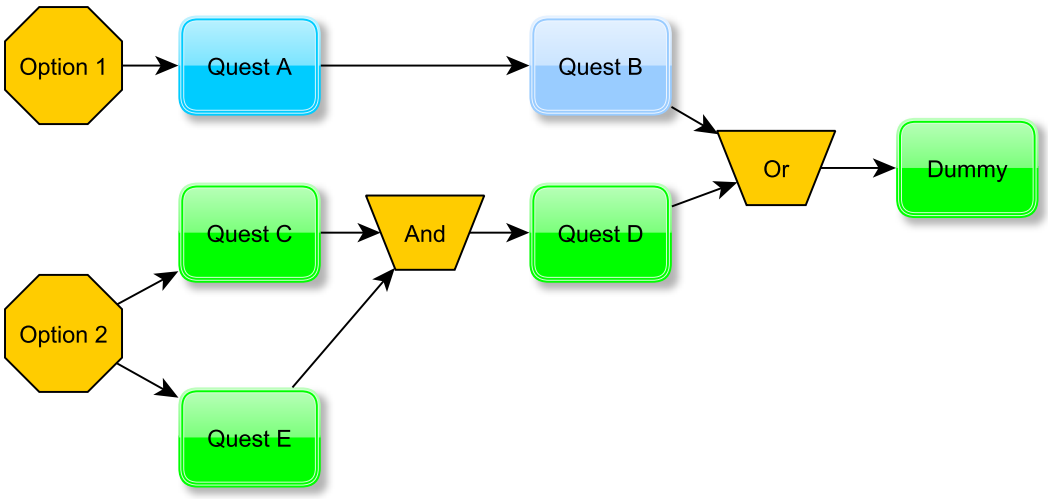

Figure 17.5: Marked path for option two

The query starts at another container but ends in the current container

The second case that we will explain is the case in which a query ends in the current container. As a result there are two possible cases to consider:

1. There is only one option available to that endpoint of the query. This can be seen in case 1 of Section 11. In such a case there is only one option so we automatically have the optimal solution.
2. There is a subset of options available to that endpoint of the query. This can be seen in case 2 of Section 11. In such a case we need to find which of the options of the subsets is the optimal solution. If there is a subset of possible options it must mean that there are one or more merge points of that subset before the final merge point of all options.

Fortunately these two cases do not have any influence on the approach used to find the shortest path. The algorithm has much similarities with the previous case. The only real difference is that we first need to backtrack from the "to"-node in order to obtain the possible options. The algorithm to find the shortest path in this case is shown below.

Algorithm FindShortestPathEndsInContainer(*container*, *toNode*)
Input. container, The container that is going to be queried. *toNode* the node in which the query ends
Output. The shortest path

1. $possibleOptions \leftarrow$ "Backtrack from $toNode$ to obtain the set
2. $pathLength \leftarrow$ "maximal integer value"
3. $path \leftarrow NULL$
4. **For** $\forall options \in possibleOptions$
5. $\quad MarkContainer(container, option, toNode)$
6. $\quad shortestPath \leftarrow FindShortestPath(container_{subgraph}, option, toNode)$
7. $\quad length \leftarrow NaiveCalculateLength(shortestPath)$
8. $\quad$ **If** $length < pathLength$
9. $\quad\quad length \leftarrow pathLength$
10. $\quad\quad path \leftarrow shortestPath$
11. return *path*

Figure 17.6 shows a graph on which we are going to query from the start of the container to "Quest G". According to the previously described algorithm we first backtrack in order to find the possible options. In this case "Option 2" and "Option 3" are reachable by backtracking.

Now that we know that only "Option 2" and "Option 3" are viable we calculate the shortest path for both these options. Figures 17.7 and 17.8 show what the graph looks like when we have marked it for both options. The "Shortest Path"-algorithm will find quests "G", "D", and "C" being on the shortest path for "Option 2".

"Option 3" has a shortest path of a different length. This path contains the quests "G", "F", "E" and "H". This means that "Option 2" is the optimal path from the start of the container to "Quest G".

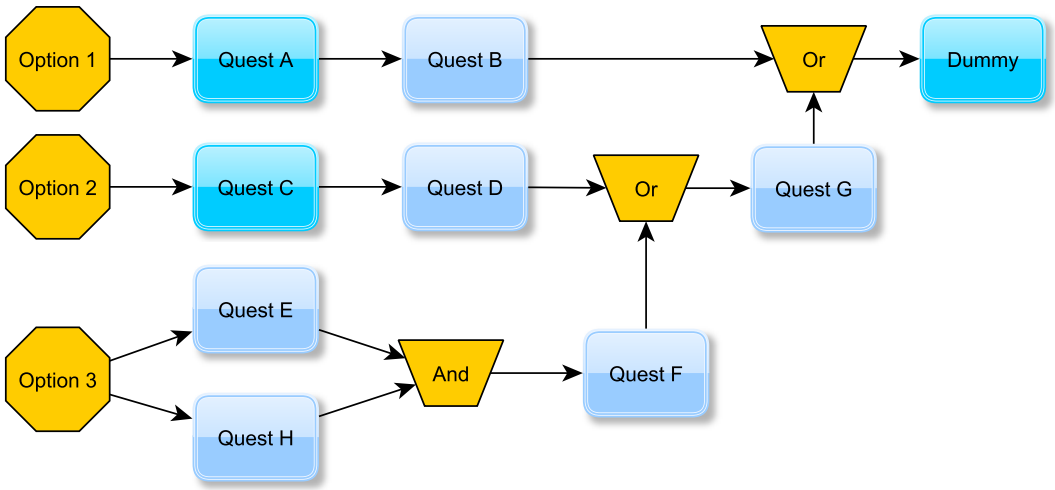

Figure 17.6: Content of a container for this example

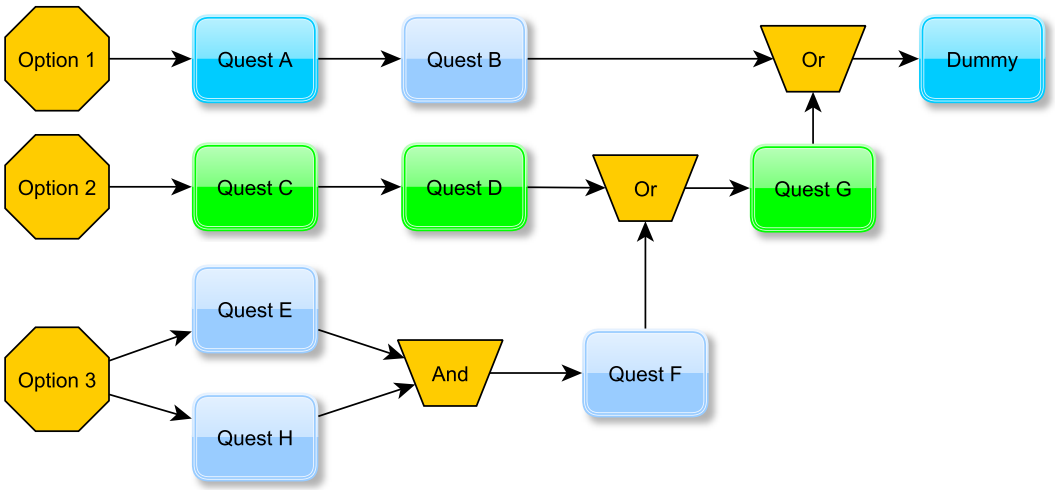

Figure 17.7: Marked path for option two

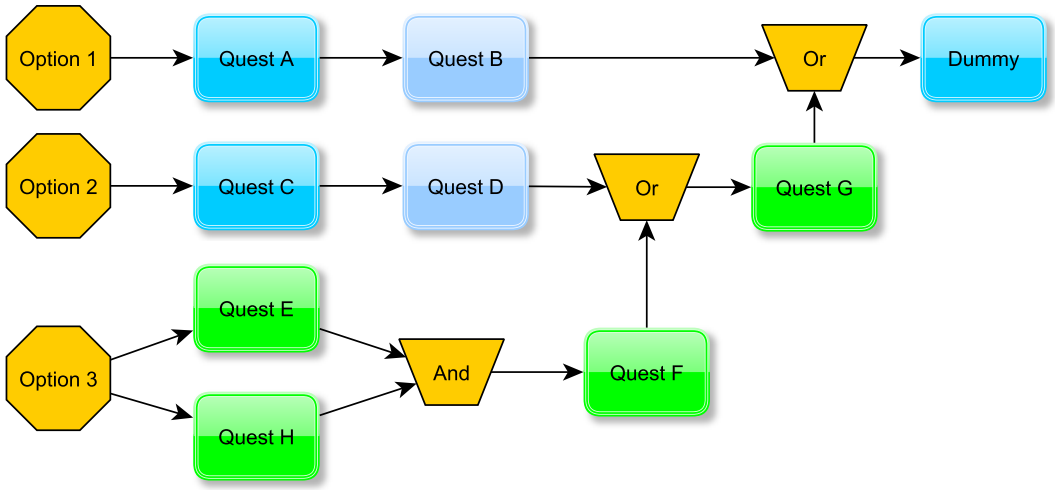

Figure 17.8: Marked path for option three

The query starts in the container but ends at another container

For the third case the query starts somewhere inside the container and ends at another container. In this case we don't know on forehand which option(s) will lead to the "from"-quest. Fortunately, the marking algorithm takes care of detecting which options lead to the "from"-quest, which means that we don't have to backtrack to find these options ourselves. Therefore it is pretty straightforward to query the container of this situation:

Algorithm FindShortestPathStartsInContainer($container$, $fromNode$)
Input. $container$, The container that will be queried. $fromNode$, the node at which the query starts
Output. The shortest path

1. $MarkContainer(container, fromNode, dummy)$
2. return $FindShortestPath(container_{subgraph}, fromNode, dummy)$

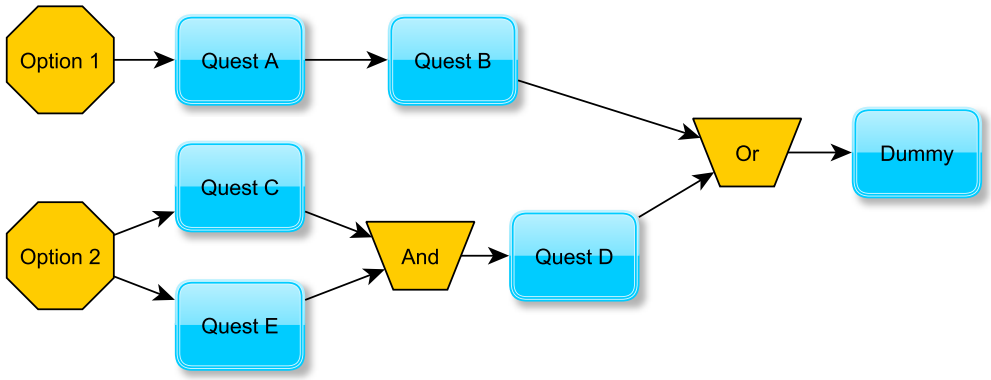

Figure 17.9: Content of a container for this example

Figure 17.9 shows an example container. In this container we will query from "Quest E" to the end of the container which is the "Dummy"-node. Figure 17.10 shows what this container looks like when it has been marked by the marking algorithm.

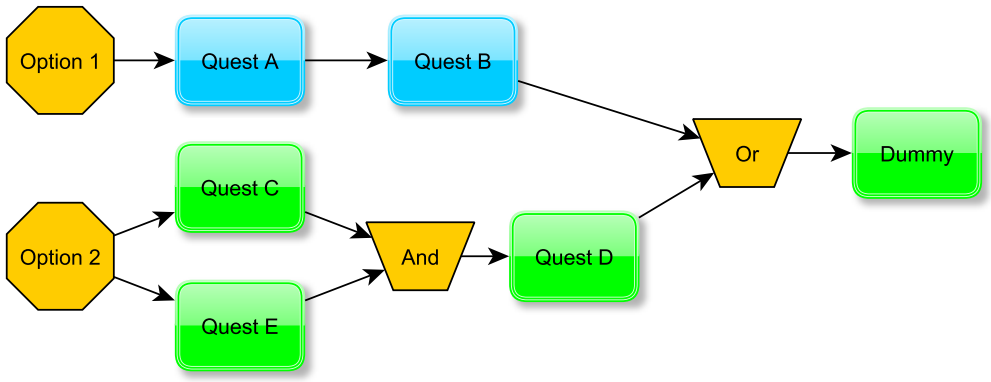

Figure 17.10: Marked path for query from quest e to dummy

The query begins and ends in the same container

The last case that occurs is the case in which we query from some node in the container to another node in the same container. This case is almost the same as the previous one, with the minor difference that the query does not end at the "dummy"-node but rather at another node before the "dummy"-node. Therefore, we use the same approach as with the previous case:

Algorithm FindShortestPathStartsInContainer($container$, $fromNode$, $toNode$)
Input. $container$, The container that contains the shortest path. $fromNode$, the node at which the query starts, $toNode$ the node in which the query ends
Output. The shortest path

1. $MarkContainer(container, fromNode, toNode)$
2. return $FindShortestPath(container_{subgraph}, fromNode, toNode)$

Figure 17.9 that was used to illustrate the previous case, will also be used to illustrate this case. However, this time we will query for the shortest path from "Quest C" to "Quest D". Figure 17.11 shows what the graph look like when it has been marked. The shortest path from "Quest C" to "Quest D" contains quests "C, D and E".

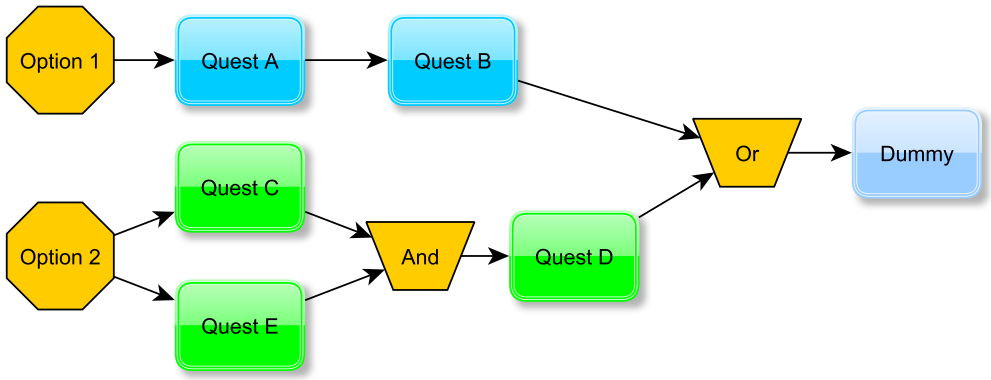

Figure 17.11: Marked path for query from quest c to quest d

17.4 Solving Longest Path Queries In Containers

In this section we will explain how we can solve longest path queries in a container that has already been marked with the marking algorithm from Section 12.3. In the remainder of this section we will explain the algorithm for obtaining the longest path.

Obtaining the longest path from a marked container is pretty straightforward. All nodes that have been marked are part of the longest path. This is in contrast with the shortest path where not all marked nodes are part of the resulting path. For the longest path we can't guarantee that the "from"- and "to"-node have a correct place in the topological ordering. Therefore, we manually place them at the beginning and end of the topological ordering.

Algorithm FindLongestPath($graph$, $fromNode$, $toNode$)
Input. $graph$, that contains the nodes for the query. $fromNode$, the node at which the query starts. $toNode$, the node at which the query ends
Output. A set of nodes that collectively form the longest path

1. $nodesOnPath \leftarrow \emptyset$
2. $fromNode_{accessible} \leftarrow false$
3. $toNode_{accessible} \leftarrow false$
4. **For** $\forall node \in graph_{nodes}$
5. **If** $node_{accessible} = true$
6. $Add(node, nodesOnPath)$
7. $orderedPath \leftarrow FindTopologicalOrdering(graph, nodesOnPath)$
8. $AddFirst(fromNode, orderedPath)$
9. $AddLast(toNode, orderedPath)$
10. return $orderedPath$

Therefore, we first set the from and to node as not accessible, thereby removing them from the topological ordering. Then we obtain the set of nodes that are accessible and pass it to the "FindTopologicalOrdering"-algorithm. This finds the ordered path between the "from"- and "to"-node. Finally we add the "from"-node to the ordering as the first node and we add the "to"-node to the ordering as the last node. This guarantees that the ordering is valid.

For the longest path we also need to be able to answer the four query types mentioned in Section 17.1. The algorithms to accomplish this are the same algorithms as mentioned in Section 17.3 that explains shortest path querying. The only actual difference is that we will now use the "Longest Path"-algorithm instead of the "Shortest Path"-algorithm. We will therefore not explain these cases again.

17.5 Querying the Choice Graph

In the previous sections we have explained how we can use markers in order to solve the shortest and longest path queries in a container. This was the first step towards solving the problem of querying the choice graph. We also explained in chapter 12 that the choice graph, in fact, is a tree. So when we perform a query we basically need to find a way in order to find the path from one container to another in the tree. The structure of the choice graph allows us to use any "Lowest Common Ancestor"-algorithm in order to find this path efficiently. The lowest common ancestor is the container which contains both the "from" and "to" node. This means that we can find the path in three efficient steps:

1. Find the path from the "from"-node to the container below the lowest common ancestor.
2. Find the path from the "to"-node to the container below the lowest common ancestor.
3. Find the path from the last "from"-container in step one to the last "to"-container in step two.

Figure 17.12 shows an example of a choice graph in which we query from "Quest A" to "Quest B". In this example "Container 1" is the lowest common ancestor. The containers with the blue borders are the containers on the path from the container of "Quest A" to the common ancestor. The containers with the green border are the containers on the path from the common ancestor to "Quest B".

Please note that it is possible that a query needs to be "solved" in the root level of the graph. This part of the graph is not part of any container. The same algorithms still apply, the only difference is that we will not find

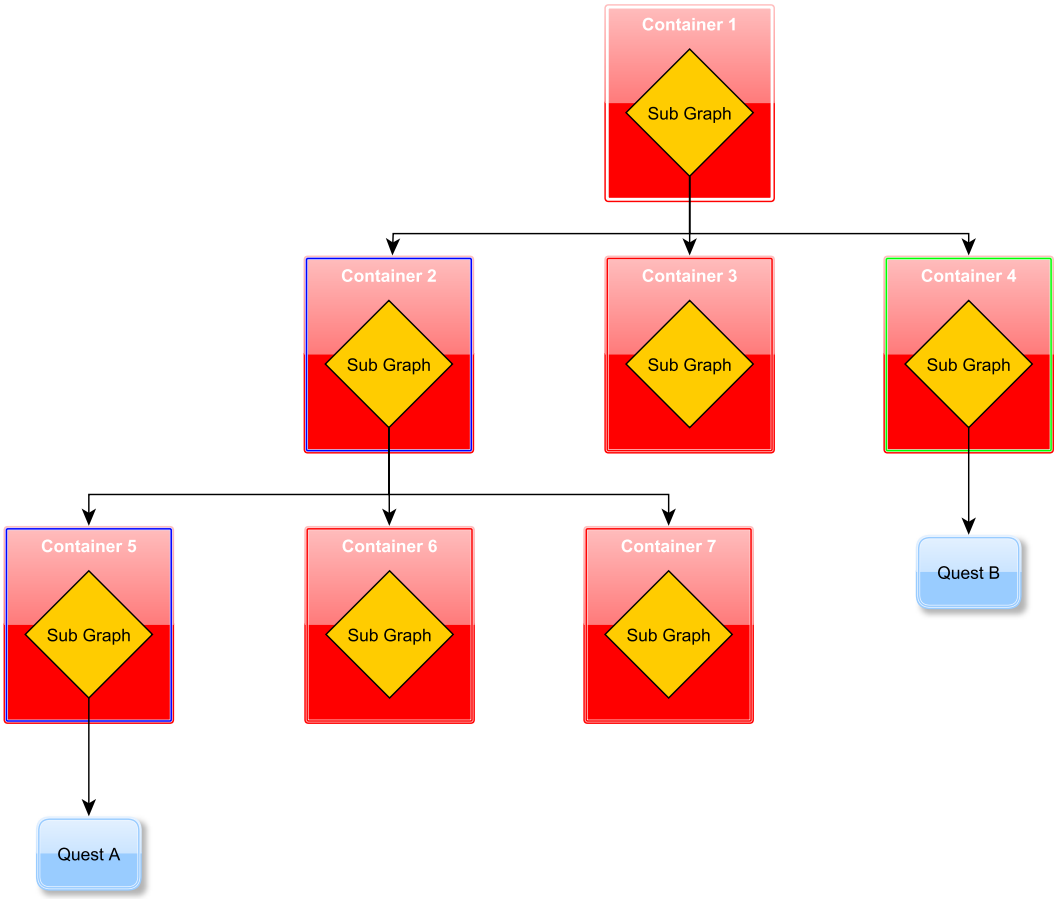

Figure 17.12: Example of the structure of a choice graph when querying from quest a to quest b

a lowest common ancestor, which is the indication that we merge at root level. It means that we need to use a different algorithm in the third step, namely a shortest path query for a normal graph. In order to keep the algorithms as simple as possible this is not shown.

When we discussed the "Shortest Path"-algorithm we explained that the resulting path will contain all the nodes on the path. This means that it also contains: options, containers and dummy-nodes. In a later section we will explain how we can extract a path that only contains quests from it.

The QueryChoiceGraph algorithm
At the start of this section we explained that we can obtain a path from the choice graph with three simple steps. In this section we will discuss the algorithm that combines those steps. This algorithm is shown below. First we obtain the containers from both the from and to node in order to get a handle on the tree-like structure of the choice graph. Then we use a lowest common ancestor algorithm to find the LCA in the choice graph. There are many algorithms to efficiently accomplish this, therefore we did not specify the "GetLowestCommonAncestor"-algorithm. After finding the LCA we can find the start and end part of the final resulting path. As a result of executing the "FindPathStart"- and "FindPathEnd"-algorithms we also obtain the nodes that are both part of the LCA. We use these nodes to obtain the middle section of the path by executing a query, of the fourth query type mentioned in Section 17.1. Finally we combine the parts of the path in order to obtain a complete path.

Algorithm QueryChoiceGraph($fromNode$, $toNode$, $pathAlgorithm$, $data$)
Input. $fromNode$, the node to start the query at. $toNode$, the query at which the query ends. $pathAlgorithm$, the path algorithm to execute. $data$, a data structure that contains pointers for each quest node to it's container
Output. The ordered path from the $fromNode$ to the $toNode$

1. $fromContainer \leftarrow GetContainer(fromNode, data)$
2. $toContainer \leftarrow GetContainer(toNode, data)$
3. $LCA \leftarrow GetLowestCommonAncestor(fromContainer, toContainer)$
4. $startPath \leftarrow \emptyset$
5. $startNodeInLCA \leftarrow FindPathStart(fromNode, fromContainer, LCA, pathAlgorithm, startPath)$
6. $endPath \leftarrow \emptyset$
7. $endNodeInLCA \leftarrow FindPathEnd(toNode, fromContainer, LCA, pathAlgorithm, endPath)$
8. $middlePath \leftarrow Find(pathAlgorithm, startNodeInLCA, endNodeInLCA)$
9. return $startPath + middlePath + endPath$

Find Path Start
The first step of the algorithm is to find the path from the "from"-node to the container below the lowest common ancestor. This can be accomplished with the following steps:

Algorithm FindPathStart($fromNode$, $fromContainer$, LCA, $pathAlgorithm$, $currentPath$)
Input. $fromNode$, the node to start the query at. $fromContainer$, the container which stores the $fromNode$. LCA, the container that is the lowest common ancestor for the query. $pathAlgorith$, the path algorithm to execute. $currentPath$, the current set of collected nodes that represent the path.
Output. The container node in the LCA

1. **If** $fromNode \in LCA_{nodes}$
2. return $fromNode$
3. **Else**
4. $currentNode \leftarrow fromNode$
5. $currentContainer \leftarrow fromContainer$
6. **While** $true$
7. $path \leftarrow Find(pathAlgorithm, fromNode, currentContainer_{dummy})$
8. $Append(path, currentPath)$
9. **If** $currentContainer_{parent} = LCA$
10. return $currentContainer$
11. **Else**
12. $currentNode \leftarrow currentContainer$
13. $currentContainer \leftarrow currentContainer_{parent}$

First we check if the "from"-node is contained within the lowest common ancestor. If this is the case then we need to find the path from the "from"-node, to the node found at the second step of the general algorithm.

If the node is not contained within the lowest common ancestor, then we first need to find the path from the current node (which in the first iteration is the "from"-node) to the end of the container. This is an example of the third query type mentioned in Section 17.1. Then we check if the parent for the current container is the lowest common ancestor, if this is the case then we stop iterating. We return the current container because we need to find the path from the current container to the node found at the second step of the general algorithm.

With this algorithm we will find quest nodes and entire container nodes, for which we haven't found the optimal option. This will be done at a later stage. Please note that in this step we will not find any options.

Find Path End
The second step of the algorithm is to find the path from the container below the lowest common ancestor to the "to"-node. The algorithm for this step is almost the same as for the previous step. The only difference is that instead of performing a forward search from a node in the container to the end, we execute a search for the beginning of the container to the a node in the container.

Algorithm FindPathEnd(*toNode*, *toContainer*, *LCA*, *pathAlgorithm*, *currentPath*)
Input. *toNode*, the node to start the query at. *toContainer*, the container which stores the *toNode*. *LCA*, the container that is the lowest common ancestor for the query. *pathAlgorith*, the path algorithm to execute. *currentPath*, the current set of collected nodes that represent the path.
Output. The container node in the LCA

1. **If** $toNode \in LCA_{nodes}$
2. return $toNode$
3. **Else**
4. $currentNode \leftarrow toNode$
5. $currentContainer \leftarrow toContainer$
6. **While** *true*
7. $path \leftarrow Find(pathAlgorithm, currentContainer, currentNode)$
8. $Prepend(path, currentPath)$
9. **If** $currentContainer_{parent} = LCA$
10. return $currentContainer$
11. **Else**
12. $currentNode \leftarrow currentContainer$
13. $currentContainer \leftarrow currentContainer_{parent}$

First we check if the "to"-node is contained in the lowest common ancestor. If this is the case we need to find the path from the node found in step 1 to the "to"-node of this step.

If this is not the case then we will get the path from the start of the container to the current node. In the first iteration the current node is the "to"-node. Then we do the same for each container at a higher level until we encounter a container who's parent is the lowest common ancestor. In this case the algorithm will use the second query type mentioned in Section 17.1.

The path obtained with this algorithm will yield quest, container and option nodes. It is important to extract the options because they indicate the best mandatory option that will lead to the "to"-quest. We will find the optimal options for the found containers at a later stage, just as we stated in the first step of the algorithm. First we will discuss how we can optimize the querying strategy.

17.6 Optimizing Queries

In the previous sections of this chapter, we explained how we can query the choice graph and obtain a path from that graph. However, the approach we previously described can be made more efficient by caching data. In this section we will explain how we can query the choice graph more efficiently by pre-calculating the optimal solutions for all query types. This means that we first need to look at what we can pre-calculate.

It turns out that there are two facets of each query type that can be pre-calculated:

1. The optimal option and length for a container with all possible answers
2. The optimal option and length for each subset of possible answers

Both facets can easily be calculated by iterating over the top level of the graph and recursively calculating the paths. The main algorithm for calculating the optimization cache is shown below. For each node in the choice graph we check if it is a container. If this is the case than it is possible to calculate optimization data. This is accomplished with the "CalculateContainerCache"-algorithm.

Algorithm CalculateChoiceCache($graph, algorithms, cacheData$)
Input. $graph$, the choice graph for which we calculate the cache. $algorithms$, the algorithms we want to optimize. $cacheData$, the set that contains the cache data

1. **For** $\forall node \in graph_{nodes}$
2. **If** $node$ is a $Container$
3. $CalculateContainerCache(node, algorithms, cacheData)$

The "CalculateContainerCache"-algorithm is used to calculate all data for one specific option/container. Before it starts the calculation process, it recursively calls itself on the nodes in the current container. As a result the lowest levels of the choice graph are calculated first which is optimal because they can't contain any containers. Then the algorithm calculates the optimal solution for each path type and for each possible set of options. This set therefore contains the solution to a query with all possible options available and the solutions for the subsets that have been calculated during the graph construction phase.

Algorithm CalculateContainerCache($container, algorithms, cacheData$)
Input. $container$, the container for which we calculate the cache. $algorithms$, the algorithms we want to optimize. $cacheData$, the set that contains the cache data

1. $choiceCacheData \leftarrow \emptyset$
2. **For** $\forall node \in container_{nodes}$
3. **If** $node$ is a $Container$
4. $CalculateContainerCache(node, algorithms, cacheData)$
5. **For** $\forall algorithm \in algorithms$
6. $CalculateOptimalOption(container, algorithm, choiceCacheData, cacheData)$
7. $CalculateOptimalSubsetOption(container, algorithm, choiceCacheData)$
8. $Add(choiceCacheData, cacheData)$

The "CalculateOptimalOption"-algorithm is used to calculate which of all possible options is the optimal solution for the current algorithm. The algorithm executes path queries for each option from the beginning to the end of the container. If all options have been queried the optimal solution is added to the dataset for the current option.

Algorithm CalculateOptimalOption($container, algorithm, choiceCacheData$)
Input. $container$, the container for which we calculate the optimal option. $algorithm$, the algorithm we want to optimize. $choiceCacheData$, the set that contains the cache data for a choice. $cacheData$, the set that contains the cache data.

1. **For** $\forall option \in container_{options}$
2. $currentOptimalPath \leftarrow \emptyset$
3. $currentLength \leftarrow algorithm_{no-value}$
4. $currentOption \leftarrow NULL$
5. $path \leftarrow Find(algorithm, option, container_{dummy})$
6. $length \leftarrow CountPathLength(path, algorithm, cacheData)$
7. **If** $MoreOptimal(path, length, currentOptimalPath, currentLength) = true$
8. $currentOption \leftarrow option$
9. $currentOptimalPath \leftarrow path$
10. $currentLength \leftarrow length$
11. $Add(option, algorithm, choiceCacheData_{all-options})$

The last algorithm that we discuss is the "CountPathLength"-algorithm. This very simple algorithm is used to calculate the actual length of a path. Because the cache data set was already filled in the previous steps we can simply look up the optimal length.

Algorithm CountPathLength($nodes, algorithm, cacheData$)
Input. $nodes$, the set of nodes that represent the path. $algorithm$, the algorithm for which we calculate the path. $cacheData$, the data set that contains all optimization data
Output. The actual length of the path

1. $length \leftarrow 0$
2. **For** $\forall node \in nodes$
3. **If** $node$ is a $Quest$
4. $length \leftarrow length + 1$
5. **If** $node$ is a $Container$
6. $optimalLength \leftarrow GetOptimalOption(container, algorithm, cacheData)_{length}$
7. $length \leftarrow length + optimalLength$
8. return $length$

Please note that we do not discuss the "CalculateOptimalSubSetOption"-algorithm. This is because the algorithm bares close resemblance to the "CalculateOptimalOption"-algorithm. The only differences are that a subset of options is available instead of all options and that the data needs to be added to a different part of the choice cache data set.

Now that we have defined a way to pre calculate the option data it is time to redefine a part of the querying strategy from the previous chapter. The algorithm below shows the optimized version of the "FindPathEnd"-algorithm. In the first version of the algorithm, it needed to find the optimal path by itself. Now it can use the data cache in order to immediately obtain the option that leads to the desired result. However, one backtrack is needed in order to find the set of possible options. This however is not as computationally intensive as the counting algorithms.

Algorithm FindPathEndOptimized($toNode$, $toContainer$, LCA, $pathAlgorithm$, $currentPath$, $dataCache$)
Input. $toNode$, the node to start the query at. $toContainer$, the container which stores the $toNode$. LCA, the container that is the lowest common ancestor for the query. $pathAlgorith$, the path algorithm to execute. $currentPath$, the current set of collected nodes that represent the path. $dataCache$, the data cache that contains the pre-calculated optimal data
Output. The container node in the LCA

1. **If** $toNode \in LCA_{nodes}$
2. return $toNode$
3. **Else**
4. $currentNode \leftarrow toNode$
5. $currentContainer \leftarrow toContainer$
6. **While** $true$
7. $possibleOptions \leftarrow Backtrack(currentNode)$
8. $optimalOption \leftarrow GetOptimalOption(currentContainer, pathAlgorithm, dataCache, possibleOptions)$
9. $path \leftarrow Find(pathAlgorithm, optimalOption, currentNode)$
10. $Prepend(path, currentPath)$
11. **If** $currentContainer_{parent} = LCA$
12. return $currentContainer$
13. **Else**
14. $currentNode \leftarrow currentContainer$
15. $currentContainer \leftarrow currentContainer_{parent}$

Normally the quest to container hashmap would be created during the calculation of the optimization cache. However, for simplicity we did not add this to the algorithms. This however is trivial to implement. In the "CalculateChoiceCache"-algorithm we need to add data to the hashmap if a node in the iteration is a quest. The same needs to be done in the "CalculateContainerCache"-algorithm, when iterating over all nodes.

17.7 Path Extraction

In the previous sections we explained how we can uses queries to effectively obtain paths from the choice graph. The problem with the path that we get from those algorithms is that it contains quest, option, dummy and container nodes, while ideally a path only should consist of quest nodes.

Therefore we will now explain how we can obtain a valid path from the total path. This can be accomplished with the algorithm that is shown below. We create a new set for the actual path that will only consist of quests. We start by iteration over the current path to handle each node. Because the nodes are of different types they need to be handled differently. Quest nodes can be added to the new path without any problems. However, the container nodes need to be processed in order to obtain the correct path. This is accomplished with the "ExpandContainer"-algorithm. All other nodes are simply ignored.

Algorithm ExtractPath($path, algorithm, cacheData$)
Input. $path$, the set of nodes that represent the path. $algorithm$, the algorithm for which we calculate the path. $cacheData$, the data set that contains all optimization data
Output. The actual path of Quests

1. $correctPath \leftarrow \emptyset$
2. **For** $\forall node \in path$
3. **If** $node$ is a $Container$
4. $ExpandContainer(node, algorithm, cacheData, correctPath)$
5. **If** $node$ is a $Quest$
6. $Add(node, correctPath)$
7. return $correctPath$

The "ExpandContainer"-algorithm is used to convert containers, with all options available, into correct quest paths. The algorithm uses the data cache in order to find the option that is optimal for the current query algorithm. Then the path is extracted from the container and processed by a recursive call to itself. This will eventually yield only quest nodes when it reaches to lowest layer of the choice graph.

Algorithm ExpandContainer($container, algorithm, cacheData, correctPath$)
Input. $path$, the set of nodes that represent the path. $algorithm$, the algorithm for which we calculate the path. $cacheData$, the data set that contains all optimization data. $correctPath$ the path that will eventually be the actual path

1. $option \leftarrow GetOptimalOption(container, algorithm, cacheData)$
2. $path \leftarrow Find(algorithm, option, container_{dummy})$
3. **For** $\forall node \in path$
4. **If** $node$ is a $Container$
5. $ExpandContainer(node, algorithm, cacheData, correctPath)$
6. **If** $node$ is a $Quest$
7. $Add(node, correctPath)$

With this last algorithm in place, it is now possible to actually obtain a set of topologically ordered quest based on a query that is executed on the choice graph.

Chapter 18

Mandatory Quest Ratio

In the Chapter 17 we explained how a choice graph can be queried. This provides the game designer with the possibility to check which options need to be chosen in order to complete the game as fast or as slow as possible. Another useful metric, that is discussed in the first part of the paper, is the mandatory quest ratio. In the first part of the paper it is used to indicate which ratio of the quests are mandatory. Such metric is also useful when used on the choice graph. However, due to the presence of choices and options the previously formulated definition of the mandatory quest ratio needs to be changed slightly.

The algorithm used in the first part of the paper calculates the metric by dividing the length of the shortest path by the length of the longest path. In this case the set of nodes used for the longest path contains all nodes that are on the shortest path and therefore is a super set of the shortest path. However, if we would calculate the shortest and longest path for the extended version of the quest graph, then the shortest path does not have to necessarily be a subset of the longest path. This is because it is very likely that the options chosen for the shortest path are not the same as the options chosen for the longest path. Therefore, it makes no sense to create a ratio that is based on two different unrelated paths in the quest graph. The solution is to redefine the way the mandatory quest ratio is used.

The root of the problem are the choices and options that are part of the quest system. If we could select the exact same options for both the longest and shortest path, then the mandatory quest ratio would be correct again. We therefore propose to use the following approach for obtaining the mandatory quest ratio. First a game designer needs to select the path that he wants to analyse for the mandatory quest ratio. Such a path can be obtained by using the shortest or longest path algorithm. Alternatively, it is possible that a game designer manually creates a path by specifying the set of selected options and the start and end node of the path.

One thing that we want to bring to attention is the difference in characteristics between selecting a path based on a shortest or longest path query.

1. Options from the shortest path
   If we would use the options from the shortest path then the longest path would result in “the longest path possible if all chosen options favor the shortest path”. A game designer can use this type of ratio in order to estimate what happens if a player makes all choices that lead to the shortest path, which can be related to moral choices or storytelling, but wants to complete as much quests as possible.
2. Options from the longest path
   If we would use the options from the longest path then the shortest path would be “the shortest path that can be taken if all choices are made in favor of the longest path”. A game designer can in this case, use the ratio to estimate what happens if a player makes all choices that lead to the longest path, but actually only wants to complete the smallest set of quests possible in order to finish the game.

In case the options are selected via a path algorithm an extra step needs to be taken. When a path is defined manually, we immediately have the set of selected options. However, this is not the case when we obtain a path via a query. Currently the algorithms do not store the options they have selected during the querying process. However, this can easily be accomplished by defining alternative versions of the path algorithms that store all the selected options. This means that the "FindPathEnd"-algorithm, which is defined in chapter 17, needs to be changed slightly in order to record the selected options. However, this alone is not enough to obtain the complete set of selected options. The final step is then to store all options that are selected while extracting the path.

At this point we obtained the path and the set of options for which we want to calculate the mandatory quest ratio. The last step left is to alter the previously two mentioned algorithms, the "FindPathEnd" and the path extraction algorithm, so that they can override the option selection process if a predefined option is selected for a choice. With these alternative versions of the algorithms in place it becomes possible to once again use the mandatory quest ratio to analyse the diversity of a quest graph.

Please note that making the adjustments to the algorithms is pretty straightforward. We will therefore not provide any concrete algorithms for the previously mentioned processes.

Chapter 19

Quest Order

In the first part of the paper we explained that one degree of freedom a player has is the amount of possible orders in which he can accept quests. Furthermore, we explained that finding all possible orders is a #P complete problem. We then defined our own algorithm that will help visualize the diversity of the quest orders. This is accomplished by calculating the interval for each quest. Each interval indicates the steps at which a quest can be accepted at the earliest and the latest. As we explained in our impact analysis in Section 9.2 it is not possible to use this approach on the extended quest graph or on a choice graph. This is because it is not possible to obtain the values that are needed to calculate the interval:

1. The number of nodes before the quest
2. The number of nodes after the quest
3. The total number of nodes in the graph

Luckily, if we use the algorithm differently it can still provide the game designer with useful information. While we used the algorithm on the entire graph in part one, the new approach is to use the algorithm on a path. This path can then be converted into a normal quest graph, which allows the utilization of the original algorithm. The algorithm used to converted an extracted path into a normal quest graph is shown below. Please note that the empty set defined in line one is a normal quest graph.

Algorithm ConvertToGraph($originalPath, ...$)
Input. $originalPath$, the set of nodes that represent the path.
Output. A quest graph containing clones of the path

1. $G \leftarrow \emptyset$
2. $path \leftarrow Clone(originalPath)$
3. **For** $\forall node \in path$
4. $\quad Add(node, G)$
5. **For** $\forall node \in path$
6. $\quad predecessorNodesEdges \leftarrow GetPredecessorNodesFrom(node, ...)$
7. $\quad$ **For** $\{\forall preNode : preNode \in predecessorNodes | preNode \in path\}$
8. $\quad\quad AddEdge(G, preNode, node)$
9. $\quad successorNodesEdges \leftarrow GetSuccessorNodesFrom(node, ...)$
10. $\quad$ **For** $\{\forall sucNode : sucNode \in successorNodes | sucNode \in path\}$
11. $\quad\quad AddEdge(G, node, sucNode)$
12. return G

First, we create a new graph to which clones of the path nodes are added. Then, for each of those nodes we are going to find all predecessor and successor nodes in order to recreate the edges between them. The specification for the "GetPredecessorNodesFrom"-algorithm is not complete. This is done on purpose because there are two ways in which we can obtain the predecessor and successor nodes. The first way is via the extended quest graph, the second is via the choice graph. For completeness will explain both of these approaches. However, we will only explain the algorithms for finding the predecessors and not for finding the successors.

First we discuss the algorithm that accomplishes this on the extended quest graph. Because the extended quest graph is not converted into a Choice Graph, all nodes are actually on one "level". This makes it very easy to find the actual node and extract the predecessors. Please note that it is possible that a predecessor is an Option node. We therefore call the "GetPredecessors"-algorithm which will obtain the correct set of quests that are the predecessors. This algorithm is shown later on in the paper.

Algorithm GetPredecessorNodesFromExtended($node, graph$)
Input. *node*, the node for which we seek the predecessors. *graph*, the extend quest graph.
Output. The set of predecessor nodes

1. $originalNode \leftarrow Find(node, graph)$
2. $inNodes \leftarrow \emptyset$
3. **For** $\forall edge \in originalNode_{in-edges}$
4. $\quad realPredecessors \leftarrow GetPredecessors(edge_{source})$
5. $\quad$ **For** $\forall realPreNode \in realPredecessors$
6. $\quad\quad Add(realPreNode, inNodes)$
7. return $inNodes$

The alternative approach is using the choice graph in order to obtain the predecessor nodes. In this case we first search for the container of the node in the container look up data. Then we look up the node in the container graph and we perform the same steps as with the previous algorithm.

Algorithm GetPredecessorNodesFromChoice($node, containerLookup$)
Input. *node*, the node for which we seek the predecessors. *containerLookup*, a set that stores a pointer for each quest to the container.
Output. The set of predecessor nodes

1. $container \leftarrow Lookup(node, containerLookup)$
2. $containerNode \leftarrow Find(node, container_{subgraph})$
3. $inNodes \leftarrow \emptyset$
4. **For** $\forall edge \in containerNode_{in-edges}$
5. $\quad realPredecessors \leftarrow GetPredecessors(edge_{source})$
6. $\quad$ **For** $\forall realPreNode \in realPredecessors$
7. $\quad\quad Add(realPreNode, inNodes)$
8. return $inNodes$

The last algorithm to explain is the "GetPredecessor"-algorithm. The version shown below is the one that can only be used on the choice graph. Basically it checks if the current predecessor node is a Quest. If this is the case than that is the only possible predecessor. However if the node is an Option, then we need to find all nodes on which the container depends. This way we will effectively remove the options and choices from the final graph.

Algorithm GetPrecessors($node, container$)
Input. $node$, the node that is the predecessor or an option. $container$, the container for $node$.
Output. The set of real predecessor nodes

1. $predecessors \leftarrow \emptyset$
2. **If** $node$ is a $Quest$
3. $\quad Add(node, predecessors)$
4. $\quad$ return $predecessors$
5. **If** $node$ is an $Option$
6. $\quad\quad parentContainer \leftarrow container_{parent}$
7. $\quad\quad$ **For** $\forall edge \in parentContainer_{in-edges}$
8. $\quad\quad\quad Add(edge_{source}, predecessors)$
9. return $predecessors$

This algorithm will not work for the extended version of the quest graph because we obviously can't go up a container. In that case we need to move backward, toward the choice. The in-edges for that choice are the real predecessors for the node.

As a result of the previously defined algorithms it is now possible to obtain a quest order for a path. This is accomplished by converting the path into a normal quest graph. On the new graph the original quest order algorithm will be executed. As a result we will obtain a correct quest order for that path.

Please note that it is important to consider which path is used to obtain he quest ordering. In the previous chapter it was mentioned that there are three possible ways to obtain a path. It can either be obtained by selecting the path by hand, using a shortest path query or using a longest path query. Furthermore, it is important to observe that a shortest path query will not always result in a quest order with the least diversity. Neither will the longest path always result in a quest order with the highest diversity. This is because the structure of the graph defines the diversity and not the length of the path.

Chapter 20

Conclusion

In this thesis we set out to assess the playability and diversity of quest systems. In order to achieve this goal we first needed to achieve a different goal which is to further formalize what a quest graph actually is.

The primary challenge we encountered was the way in which we should handle the relationships between quests. Ideally we would want to allow both "or"- and "and"-relationships between quests in order to build quest graphs. The "or"-relationship enables a game designer to allow the completion of only a limited subset of dependencies for a quest. However, if a "or"-relationship is used for quest, it becomes increasingly harder to estimate all the possible orders in which a player can complete quests. In order to prevent these complexities from occurring, the scope of the research in the first part was changed so that only the "and"-relationship is supported by the quest graph.

Another problem we encountered is the way in which dependencies on other quests are handled. If one quest depends on the completion of another quest the situation is clear. However, as soon as quests start to depend on for example items that are provided as a reward by another quest things become more complicated. In this case the situation gets even worse if the player needs to gather a certain number of items that are provided by multiple quests. In the end we decided to make all quests that provide the player with the needed item a dependency for the quest. This guarantees that the player can at some point complete the quest. However, in reality it is perfectly possible that only a limited subset of quests is needed in order to obtain the correct amount of items.

Despite the scope limitations we successfully formalized the quest graph and created several algorithms that will help a game designer analyse the diversity and playability of a quest system. Using these algorithms we analysed the "Cataclysm"-expansion for the game "World of Warcraft". As a result we can make several interesting observations about the quest system of "World of Warcraft" and about our formalization of the quest system. First of all we can conclude that quest chains are not well defined. If we assume that the quest chains in "World of Warcraft" are formed around the story behind the quests then we can observe two types of inconsistencies with respect to quest chains:

1. Some quests aren't part of any quest chain but are part of the same story
2. Some quests are part of different quest chains but share the same story

Furthermore, a fairly high amount of quests, 26 of a total of 136 quests, did not belong to any quest chain. This is a clear indication that quest chains are used rather casually and that they might not be an integral part of the quest system. In our opinion quest chains should only be used if they enable new gameplay. If they are only used as a tool to group quests then it is not really useful to make quest chains a part of the quest system at all.

From the case study we also noticed that the quest relationships used in "World of Warcraft" are rather

simplistic. Based on our play sessions we estimate that only the "Quest Completed Prerequisite" and the "Level Prerequisite" are used. We wonder why only these relationship types are used in a successful game like "World of Warcraft". Would it be the case that otherwise the quest system would get to complex to understand? Or would there be a game design related reason behind it? Further research is needed in order to find the answers to these questions.

Another observation from the case study is the applicability of layers. In our paper we propose to use the concept of layers to divide a quest system in to several parts. In our opinion it would be a good idea to make the lowest layer contain quest graphs based on the quest hubs in a region of the game. However, when we played "World of Warcraft" we noticed that this is not always possible. We therefore conclude that it is possible to use layers to divide the quest system into smaller parts but it would be wise to do this at a fairly high level like for example per region.

We also looked at the diversity of the quest system of "World of Warcraft". From the results of the case study we can conclude that the player does not have much freedom. This is because a fairly heigh percentage of quests, 73% to be exact, is mandatory. This indicates that a player has a high chance of getting stuck if he skips a quest. When we look at the results for the player freedom, when it comes to the order in which he accepts quests, we can see some interesting things. The player still has a relatively high amount of freedom when it comes to choosing the order for all the quest that are on the shortest path from the beginning of the region to the end. But that freedom is much lower than the freedom that is obtained when completing all quests. Furthermore, we like to note that much travelling will be needed for some of the quest acceptance orders.

In the second part of the paper we set out to increase the diversity of the quest system by adding choices to the quest system. First we needed to decide in which way choices should be added to the quest system. Eventually we decided to use the "Element"-approach because it can potentially provide game designers with the highest amount of design freedom. However, we noticed that this freedom comes at a steep price. If we want the quest system to express all of the desired characteristics then as a result the quest graph will become very complex. This results in the inability to adapt the algorithms from the first part of the research to work with the newly added choices. We therefore conclude that it is very hard to enable all of the characteristics that we deemed important. Ultimately we narrowed the scope of the "Element"-approach in order to become able to solve the problem.

While we where not able to add all of the desired characteristics to the quest graph, we did succeed in providing both the player and the game designer with more freedom than with the original quest graph. By adding choices to the quest system, we provided a game with a higher replay value. Furthermore, we enabled the usage of consequences which can be used to create more interesting game play than was previously possible. Finally, we enabled the usage of the "skipping of a choice"-characteristic which can help a game designer provide a player with the possibility to skip content without getting stuck.

From our total work we can conclude that quest systems despite their simple looking nature are quite complex. The problem with quest systems is that they, in most cases, form the bridge between the story and the content that is provided to the player. This means that formalizing and accessing a quest system is not only a technological problem but that it is also closely related to gameplay aspects. For example imposing a technologically unimportant limitation might result in a very important gameplay limitation. Therefore, it is important to create a good balance between those two domains.

As a result of the previously mentioned complexities it is important to keep in mind that it is best to tailor a quest graph and the accompanied algorithms to your specific situation. Therefore, if your implementation desires all of the characteristics that are expressed in our algorithms it is fine to use our algorithms. But if you need less it is probably better to simplify the algorithms accordingly. This will result in algorithms that run faster and are easier to understand.

In our opinion we have successfully shown that it is possible to analyse the diversity and playability of a quest system. However, we do regret that we did not get the chance to test our algorithms in real life situations. We made several attempts to obtain real life data from game studios. However, they were not able to comply with our requests. But even if they had, we still wouldn't be sure if that would have made a difference. This

is because their quest systems aren't designed with the same mindset as ours. Therefore, it is very likely that their version of a quest system will differ to much from our version, which will prevent a correct analysis. The case study performed in the first part of the paper did give us some interesting results about the game "World of Warcraft" but it can't be proven to be correct. This is a direct result of not being able to obtain any feedback from the game designers. This is very unfortunate because they could for example have explained the reasoning behind the limitations of the quest system of "World of Warcraft". Additionally the game designers are the ones that will have interesting ideas on how to extend the context of our current quest system formalization. We therefore consider this as future work.

Bibliography

[BW91] Graham Brightwell and Peter Winkler. Counting linear extensions. *Order*, 8:225–242, 1991. Springer Netherlands.

[ea01] Cormen et al. Introduction to algorithms. pages 549–552, 2001. MIT Press.

[ea05] Wing-Ning Li et al. On computing the number of topological orderings of a directed acyclic graph. *Congressus Numerantium*, pages 143–159, 2005. Winnipeg, Man. : Utilitas Mathematica Pub. Inc.

[ea10] Jacqueline Banks et al. Using tpa to count linear extensions. *arXiv:1010.4981v1*, 2010.

[Hared] Seth Harris. Counting linear extensions of a partial order. 2011, not published.

[Tar76] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6:171–185, 1976. Springer Berlin / Heidelberg.

[vdW11] Jens van de Water. A framework for formalizing dynamic quests. *Master Thesis, Department of Computer Science, Utrecht University*, 2011.