The Thirty-First International Florida
Artificial Intelligence Research Society Conference (FLAIRS-31)

# Non-Linear Quest Generation

**Alex Stocker**
Bradley University
astocker@bradley.edu

**Chris Alvin**
Bradley University
calvin@bradley.edu

## Abstract

This paper presents a method for generating game quests supporting concurrency and user-choice. We formalize the notion of a quest in the form of a directed hypergraph and algorithms for generating such quests over a user-defined set of verbs and nouns. Our experimental results demonstrate the complexity and diversity of the quest space. We then compare the richness of our generated quests with a corpus of existing quests.

## 1 Introduction

We describe an ideation technique for non-linear quest generation supporting concurrency in player activities, parallelism for user choice of activity, and dependence between activities. We accomplish this goal using a many-to-one hypergraph (Berge 1989) as our representative structure. Our technique operates in four steps. We first construct a set of 'actions' using noun-verb pairings. Given this set of actions, we construct a linearization that captures dependence among the action set. Then, we construct a directed hypergraph where hypernodes indicate concurrency of activities. Last, we may replace one-to-one edges with parallel sub-hypergraphs. The result is a set of hypergraphs that can be analyzed for game quest structures.

As an example generation, we begin with a set of colloquially interpreted verbs (e.g., talk, collect, etc.) and a set of nouns (e.g., rope, ogre, etc.). Using the set of verbs and nouns, we construct individual *action*s consisting of a verb and noun; Figure 1 has five sample actions. We then construct a linear ordering of the actions. To do so, we heed any explicit dependencies specified by the user; such a dependence is indicated as a dashed edge between "Trap Ogre" occurring before "Return To Elder" as shown in Figure 1. These ordered actions correspond directly to the nodes in a directed hypergraph (Berge 1989). We then construct the set of hyperedges by traversing the linear ordering of nodes in reverse: we select a consequent node and a set of antecedent nodes. In Figure 1, there are two resulting one-to-one edges and a single two-to-one hyperedge. We interpret this hyperedge as allowing the player to complete action "Collect Marionberry" and "Collect Rope" in any order. The resulting hypergraph representation of the quest is a single hyperpath

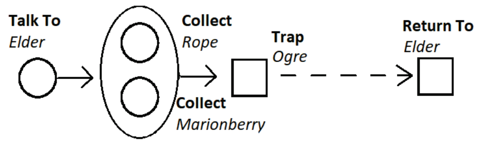

Figure 1: Phase 1 of Quest Generation: Base Quest

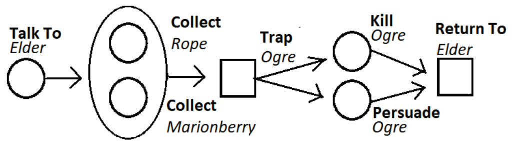

Figure 2: Phase 2 of Quest Generation: Adding Parallelism

(Berge 1989) and serves as the required set of actions which a user must complete.

The second phase of quest generation adds parallel paths to a linear quest (e.g. Figure 1 into Figure 2). For each one-to-one edge in the Phase 1 hypergraph, we may we insert parallel paths; for example, between "Trap Ogre" and "Return To Elder" in Figure 2 we inject two distinct hyperpaths. What results is the quest in Figure 2 that provides concurrency among actions and path selection. The rest of this paper will define explicitly our quest generation technique.

## 2 Preliminaries

**Actions.** Players perceive quests as a sequence of isolated steps. A single step often involves an 'action' that consists of, in simplest form, a verb applied to a noun. For example, given the verb 'talk' and the noun 'Elder,' we may construct the action "Talk to Elder" as shown in Figure 2. We informally refer to the terms *verb* and *noun* as the English-language based interpretation and intuition of those terms. For a verb $v$ and noun $n$, an *action* is a pair $\langle v, n\rangle$ a player may carry out as a step to complete a quest.

Some verb-noun pairings are nonsensical according to their interpretation in the English language (e.g. verb 'steal' and noun 'Elder'). We define two classes for both verbs and nouns: $C = \{living, static\}$. 'Steal' is an example of a static verb since we will not 'steal' a living noun (although we might 'kidnap') whereas a living verb like 'talk' may be

Table 1: Sample Equivalence Relation of Verbs

| Representative Verb | Synonyms |
|---|---|
| kill | { assassinate, execute, murder } |
| steal | { take, thieve, pilfer} |
| convince | { persuade, sway} |

applied to a living noun. We further divide the classes of static nouns into subclasses $\{object, location\}$ because it does not make sense to 'steal' a place. Classes of verbs and nouns prohibit meaningless combinations (e.g., "Steal Elder") while allowing other combinations (e.g. "Steal book"). We define $\mathsf{constr}\,(v, n)$ to return true or false whether verb $v$ may be combined with noun $n$ into an allowable action.

Without context, "Convince Elder" and "Persuade Elder" can be interpreted as synonymous actions. For *verbs* $v_1, v_2$ and *nouns* $n_1, n_2$, we say $v_1 \equiv v_2$ (resp. $n_1 \equiv n_2$) if the verbs (resp. nouns) are colloquially equivalent. Two actions $\langle v_1, n_1\rangle \equiv \langle v_2, n_2\rangle$ if and only if $v_1 \equiv v_2$ and $n_1 \equiv n_2$.

We may then construct a *verb equivalence relation* consisting of a representative set of verbs and the corresponding synonyms; see Table 1. For a verb equivalence relation $V$, we acquire the representative set of verbs from $V$ using function $\mathsf{rep}\,(V)$. The verb equivalence relation will guarantee generation of unique actions.

**Action Sets and Linear Ordering.** Structurally, a viable quest is a directed acyclic graph (DAG) with single sink node acting as the goal action of the quest. We say an *action set* $A$ is a DAG of actions and an *action set linear ordering* (or *linearization*) $L_A$ is a topological ordering of an action set $A$. For an action set $A$, $\mathcal{L}_A$ refers to the collection of linearizations. We note $|\mathcal{L}_A| \in [1, |A|!\,]$.

**Action Sets and Linear Ordering.** For an action set $A$, we use a *directed hypergraph* (Berge 1989) data structure where a *hypernode* (a set of nodes) captures the concurrency of actions and a *directed hyperedge* captures the sequential nature of actions. In our model, hyperedges consist of a set of antecedent nodes and a single consequent node: a many-to-one relationship. For example, there is a single hyperedge in Figure 2 with two actions ("Collect Rope" and "Collect Marionberry") in the antecedent set and "Trap Ogre" as the consequent action. We call our model an *action hypergraph*.

**Definition 1.** *An* action hypergraph *is a directed hypergraph* $H_A\,(N, E)$ *where each* $n \in N$ *corresponds to an action* $a \in A$ *and each directed hyperedge* $S \xrightarrow{f} t \in E$ *where* $S \subseteq N$ *is the* antecedent set *of actions,* $t \in N$ *is the* consequent *action, and each hyperedge is labeled as a user-defined dependence by* $f : E \to \{true, false\}$.

Let $A$ be an action set. A *quest* is an acyclic action hypergraph $H_A\,(N, E)$. Since a quest corresponds to an acyclic many-to-one directed hypergraph, we may analyze the corresponding structure using various measures. We refer to the *length* of a quest as the number of actions from source action to the sink action. The minimum (resp. maximum) length refers to the number of actions in the shortest (resp. longest) path. We call the number of non-one-to-one hyperedges in a quest the *concurrency count* and the *concurrency factor* is the measure of concurrency count divided by the minimal length. The concurrency factor is a measure that permits us to compare the relative amount of concurrency in quests with respect to the minimum number of actions required to complete a quest. Last, we say the *path complexity* is the number of possible paths in a quest's action hypergraph.

**Algorithm 1** Quest Generation

```
Require: $B_L$: Global quest length upper bound,
 1:     $B_A$: Global antecedent upper bound,
 2:     $B_P$: Global parallel depth upper bound,
 3:     $B_W$: Global number of parallel paths upper bound,
 4:     $AG_{V,N}$: Action Generator
 5: function MAIN($AG_{V,N}$)
 6:     $A \leftarrow AG_{V,N}$.GENUNIQUE($B_L$)
 7:     return QUESTGEN($A, AG_{V,N}, 0$)
 8: function QUESTGEN($A$, $AG_{V,N}, b$)
 9:     $L_A \leftarrow$ LINEARIZATIONGEN($A$)
10:     $H_A \leftarrow$ BASEQUESTGEN($L_A$)
11:     if $b < B_P$ then PATHS($H_A, AG_{V,N}, b + 1$)
12:     return $H_A$
```

## 3 Action Hypergraph Construction

In this section we describe our recursive procedure for constructing quests via action hypergraphs; we briefly consider the MAIN function in Algorithm 1. The quest search space is vast. Hence, Algorithm 1 defines upper bound constants allowing the user to refine the quest space and limit features of resulting quests: length ($B_L$), concurrency ($B_A$), parallel depth ($B_P$), and the number parallel paths ($B_W$). The first step in the procedure (Line 10) involves constructing an acyclic action hypergraph sans parallel paths we call a *base quest*. If requested (Line 11), we recursively construct parallel hyperpaths in the base quest.

**Generating Actions and Ordering.** For simplicity in communicating our algorithms, we introduce an *Action Generator*, $AG_{V,N}$: a factory-based data structure that generates actions on-demand based on a verb equivalence relation $V$ and a set of nouns $N$. We treat the action generator as the single input structure in our algorithms starting with Algorithm 1 on Line 5.

We construct an arbitrary action set in an action generator using class-based constraints on each verb-noun pairing as $\{\langle v, n\rangle \mid v \in V \wedge n \in N \wedge \neg\mathsf{constr}\,(v, n)\}$. Similarly, $\{\langle v, n\rangle \mid v \in \mathsf{rep}\,(V) \wedge n \in N \wedge \neg\mathsf{constr}\,(v, n)\}$ generates a unique action set because they are constructed using representative set of verbs in $V$; we do so using GENUNIQUE in Algorithm 1 on Line 6.

Given a set of actions $S$, we introduce the user-defined dependencies thus creating an action set DAG $A$. For quest generation purposes, we are not interested in a single linearization of the action set. Experimentally (§4), we generate a diverse subset of all linearizations, $\mathcal{L}_A$ (Varol and Rotem 1981), by choosing maximally distant linearizations using the Levenshtein string metric.

**Base Quest.** Our goal is to generate an acyclic action hypergraph (such as Figure 1) as defined in Algorithm 2.

**Algorithm 2** Base Quest Generation

```
Require: $L_A$: Linearization
1: function BASEQUESTGEN($L_A$)
2:     $H_A$ : Action Hypergraph
3:     $H_A$.ADDNODES($A$)
4:     for $i \leftarrow |L_A|$ downto 1 do
5:         if $\neg L_A$.HASEDGE($i-1, i$) then
6:             $i_a \leftarrow i - 1$
7:             if $H_A$.HASANTECEDENT($i$) then
8:                 $i_a \leftarrow$ MIN ($H_A$.EDGES($i$).$antecedent$) $- 1$
9:             $|ante| \leftarrow$ MIN(RANDOM $(1, B_A)$ , $i_a$)
10:            $ante \leftarrow L_A[i_a - |ante|, i_a]$
11:            $H_A$.ADD($ante \overset{false}{\rightarrow} L_A[i]$)
12:        else $H_A$.ADD($L_A[i-1] \overset{true}{\rightarrow} L_A[i]$)
13:    return $H_A$
```

*Nodes.* Given a linearization $L_A$, we add the corresponding nodes to an action hypergraph $H_A$ (Line 3, Algorithm 2).

*Hyperedges.* In Algorithm 2, we traverse $L_A$ from sink node to source node (Line 4). If the user defined a dependence between two adjacent nodes in $L_A$ (HASEDGE on Line 5), we add a corresponding edge to $H_A$ with a $true$ annotation. Otherwise, for the node at index $i$ as consequent, we construct a hyperedge and ensure an acyclic hypergraph by choosing a proper antecedent set. We default to an index of a preceding node $i - 1$ (Line 6), but on Line 7 the node at index $i$ may be an antecedent in some other hyperedge. If it is, on Line 8, using function EDGES in $H_A$, we acquire the set of all hyperedges for which the node at index $i$ is an antecedent. Of this hyperedge set we choose the smallest invalid antecedent node index; subtracting 1 results in the largest, valid antecedent index for our new hyperedge. We then choose a random-sized antecedent set size (Line 9) bounded by user-defined $B_A$ and the number of remaining valid antecedent nodes indicated by $i_a$. Last, on Line 11, we construct the hyperedge with $false$ annotation indicating a constructed hyperedge. Precise construction of hyperedges results in an acyclic base quest (Stocker and Alvin 2018).

For each linearization, Algorithm 2 results in a single corresponding hypergraph. To avoid undue complexity, we have presented a simplified, one-to-one algorithm. Instead of random antecedent sizes presented in Algorithm 2, we construct and report in §4 all possible action hypergraphs.

**Parallel Hyperpaths.** Our goal is to now add unique choices via parallelism. In Algorithm 3, we take as input an action hypergraph ($H_A$), an action generator ($AG_{V,N}$) and the recursive bound for the depth of parallelism ($b$). The goal of PATHS is to determine if and where to insert parallel paths in $H_A$. To maintain the dependent structure of the linear set of actions $A$ in $H_A$, we generate and insert parallel paths only between (one-to-one) edges (Line 2) regardless of annotation. For each edge $a \rightarrow c$ on Line 3, we may add parallel paths; experimentally (§4), we generate both options. If we generate parallel paths, we call GENPATHS (Line 4) and on Line 5 insert each distinct path $p$ between $a$ and $c$ into $H_A$.

GENPATHS in Algorithm 3 recursively generates a set of unique parallel paths $P$; $|P|$ is bounded by $B_W$ (Line 8). We ensure uniqueness in each path by having at least one unique action (Line 8) by fixing a noun ($n \in N$) and using it to generate a representative set of actions $A_R$ based on the representative set of valid verbs, $\texttt{rep}(V)$ using the GENREP function of the action generator. For each parallel path (Line 9), we construct a set of actions $A_P$ consisting of a unique representative action $a_R$ and another set of unique actions (Line 10). For the action set $A_P$, Line 11 calls QUESTGEN to recursively generate a base quest with deeper parallelism as indicated by the increasing variable $b$.

**Algorithm 3** Parallel Path Generation

```
Require: $H_A$: Base Action Hypergraph, $AG_{V,N} B_L$: Action Generator, $b$: Current Parallelism Depth
1: procedure PATHS($H_A$, $AG_{V,N}$, $b$)
2:     for all $a \rightarrow c \in H_A$ do                ▷ one-to-one edges
3:         if RANDOM(true, false) then
4:             for all $p \in$ GENPATHS($AG_{V,N}, b$) do
5:                 $H_A$.INSERT($a, p, c$)
6: function GENPATHS($AG_{V,N}, b$)
7:     $P \leftarrow \emptyset$
8:     $A_r \leftarrow AG_{V,N}$.GENREP($n \in N, B_W$)
9:     for all $a_r \in A_R$ do
10:        $A_P \leftarrow \{a_R\} \cup AG_{V,N}$.GENUNIQUE($B_L$)
11:        $P \leftarrow P \cup \{$QUESTGEN($A_P, AG_{V,N}, b$)$\}$
12:    return $P$
```

Table 2: Characteristics of 82 Quests from *Skyrim*

| Characteristic | Min | Mean (Std.Dev.) | Median (IQR) | Max |
|---|---|---|---|---|
| **Action Count** | 3 | 11.57(6.71) | 10 (8.5) | 38 |
| **Minimal Length** | 3 | 10.09 (5.11) | 9 (7) | 29 |
| **Maximal Length** | 3 | 10.71 (5.55) | 9.5 (7) | 29 |
| **Concurrency Count** | 0 | 0.17 (0.41) | 0 (0) | 2 |
| **Concurrency Factor** | 0 | 0.01 (0.03) | 0 (0) | 0.17 |
| **Path Complexity** | 1 | 3.09 (10.13) | 1 (1) | 90 |

We refer to Figure 1 into Figure 2 as a simple example of parallel path generation. In Figure 1, we select the dashed edge between square actions "Trap Ogre" and "Return to Elder". With noun 'Ogre', we generate two paths with representative actions "Kill Ogre" and "Persuade Ogre", inserting both paths into the hypergraph between the original edge. See (Stocker and Alvin 2018) for more complex sub-quest and sub-parallelism examples.

## 4 Experimental Results

**Experimental Setup.** Fundamentally, our quest synthesis algorithm is based on a set of nouns, verbs, and user-defined dependencies among actions, if any. Each parameter described in §2 and §3 can be tuned. Each execution of our quest synthesis algorithm takes these factors into account.

**Characteristics of Existing Quests.** As a contrast to our technique, we analyzed a corpus of 82 of the approximately

400 quests from *The Elder Scrolls V: Skyrim* (*Skyrim*); see Table 2. While *Skyrim* is only one AAA game, it is a good representation of RPGs as it boasts concurrency and parallelism in quests commensurate with other AAA titles. The distribution describing the number of actions in *Skyrim* quests is slightly skewed having more 'middle'-range quests with around 11 actions. The mean and median of path complexity characteristic describe a strongly skewed distribution indicating little to no choice intra-quest. We also conclude that the ability for a player to pursue concurrent activities intra-quest in *Skyrim* is limited. We conclude that most quests in *Skyrim* are strictly linear, but recognize that choice is offered via switching back and forth among multiple quests.

**Quest Generation.** We generated a sample of 590 quests using a range of 4-13 actions without any parallelism (thus path complexity of 1). The concurrency count mean was 3.81 (std. dev. 2.01) and median 4 (IQR 3) with concurrency factor mean 0.40 (std. dev. 0.15) and median 0.41 (IQR 0.2). We computed a 99% confidence interval of the concurrency factor ($0.40 \pm 0.02$) and thus conclude our technique guarantees strong concurrency in quests compared to existing games (Table 2) even for quests with few actions.

For quests with parallelism, we wish to show that with a limited number of actions and levels of parallelism we achieve a large number of quests. We restrict our base quest to between 3 to 5 actions and parallel actions from 6 to 12 actions. We also restrict a maximum of 2 parallel paths inserted into the base quest, parallel depth 2, and limit of 3 antecedent actions per hyperedge. In the extreme case of 5 base quest actions and 12 parallel actions, we generated 612,000 quests in 344 seconds. The resulting quests had a path complexity mean of 25.0 indicating an average of 25 different ways to complete the resulting quest. Interestingly, the mean minimum length was 7.81 compared to maximum length 12.01 indicating some paths are expeditious and some are labored. For concurrency factor, we observed 0.20 in this experiment and refer to this result as indicating strong concurrency. We note that with our algorithms, parallelism is constructed between one-to-one edges. That is, with increased parallelism, there is bound to be less concurrency (contrasting 0.40 from our earlier experiment). We note that this experiment is restricted due to the exponential growth in the space, but with other experimental constraint configurations we arrive at the same conclusions.

## 5 Related Work

Some approaches (Doran and Parberry 2011) share the fact that only linear, single path quests are generated, having no parallelism or concurrency. The technique developed by Sullivan, et al. (Sullivan, Mateas, and Wardrip-Fruin 2009) takes in user input phrases like locations or objects and produces related concepts from a database rather than a sequence of actions for a player to perform. This provides ideas, but leaves quest structure to the quest designer.

Our work is in the spirit of Dormans (Dormans 2010) who describes 'mission' (quest) generation using generative graph grammars. Both Dormans' and our techniques generate parallel paths, but we choose to represent quests using a richer structure: the hypergraph. Although (Dormans 2010) discusses real and generated examples, they do not discuss the resulting space of quests. (Doran and Parberry 2011) generates quests given the motivation of the originating NPC, a set of rules, and a categorized collection of verbs. The motivation restricts the quest to one path (set of terminals from grammar-based generation) and limits the usable actions. Strict grammar-based generation (using graph grammar or otherwise) does not guarantee particular plotpoints in a resulting quest. Our approach defines a natural means of using a linearization as a set of discrete plot points prior to sub-quest generation. Contrasting existing techniques, we describe a method of quest construction using only a collection of categorized verbs and nouns as input and automatically provide structure. In addition, our directed hypergraph structure facilitates unique player experiences through concurrency and path selection while maintaining any specified dependencies. We feel an appropriate generation scheme would strike a balance between the rule-based approach of generative graph grammars and our structure-driven approach.

## 6 Conclusions

We have formalized a representation of a game quest supporting concurrency and parallelism as an acyclic directed hypergraph. We have presented an effective and efficient technique for generating such quests. Last, we demonstrated the efficacy of our approach comparing to existing quests in a AAA commercial game.

## 7 Acknowledgments

This work was supported in part through a Special Emphasis Award from Bradley University as well as the Bradley University College of Liberal Arts and Sciences.

## References

Berge, C. 1989. *Graphs and hypergraphs*, volume 45. North-Holland Mathematical Library; ELSEVIER SCIENCE PUBLISHERS B.V.

Doran, J., and Parberry, I. 2011. A prototype quest generator based on a structural analysis of quests from four mmorpgs. In *Workshop on Procedural Content Generation in Games*. ACM.

Dormans, J. 2010. Adventures in level design: Generating missions and spaces for action adventure games. In *Workshop on Procedural Content Generation in Games*, PCGames '10, 1:1–1:8. New York, NY, USA: ACM.

Stocker, A., and Alvin, C. 2018. Non-linear quest generation. http://hilltop.bradley.edu/ calvin/papers/flairs-2018-full.pdf.

Sullivan, A.; Mateas, M.; and Wardrip-Fruin, N. 2009. Questbrowser: Making quests playable with computer assisted design. In *In Proceedings of the Digital Arts and Culture Conference*.

Varol, Y. L., and Rotem, D. 1981. An algorithm to generate all topological sorting arrangements. *Comput. J.* 24(1):83–84.