Shortest Walk in a Dungeon Graph

Christopher Durham, Chris Alvin
Furman University
Department of Computer Science
Greenville, SC, USA
chris.alvin@furman.edu

Abstract

With games like *No Man's Sky* and *Diablo IV*, procedural generation of content in games is ever-increasing. Heuristics are needed to assess the goodness of generated content. Using Nintendo's *The Legend of Zelda* as inspiration, we formalize the idea of a dungeon graph: a graph consisting of 'locked' edges in which 'keys' are acquired from specific nodes. We then introduce the *Shortest Dungeon Walk Problem* as well as a solution to this problem in the context of a dungeon graph and reduce Traveling Salesperson Problem in polynomial time to the *Shortest Dungeon Walk Problem* and conclude that *Shortest Dungeon Walk Problem* is NP-Complete. We then assess practical performance of the shortest walk algorithm using the first eight dungeons in the *Legend of Zelda*.

1 Introduction and Motivation

Procedural generation of levels in games has been well-documented (Smelik et al. 2010; Valtchanov and Brown 2012; Khalifa et al. 2016). Games such as *No Man's Sky* (Hello Games 2016) tout a procedurally generated universe, but often the reality of the generated content can be a bit stark. More recently, developers at Blizzard modified their dungeon generation algorithm in *Diablo IV* between Season 1 and Season 2 in response to player feedback. Their complaint: too much backtracking (Colp, Tyler 2023). In this work we propose a problem and solution that may be used as a heuristic for verifying procedurally generated graph-based dungeons in games.

We can represent a dungeon from the 1986 game *The Legend of Zelda* (Nintendo 1986) for the Nintendo Entertainment System (NES) as a sparse, undirected graph with locks and keys for those locks. We call this type of graph a *dungeon graph*. Our goal is to compute a shortest walk of a dungeon as a measure of the quality of the design and complexity of a dungeon (Bond 2022).

As an introduction to the *shortest dungeon walk problem* and our solution algorithm, we consider an example using the first dungeon from the *Legend of Zelda* as shown in Figure 1: the "Eagle Dungeon". The dungeon consists of 17 rectangular rooms (nodes in a dungeon graph); in Figure 1, six rooms hold keys, indicated with a key icon, which the player can obtain. The player may freely walk between rooms via doors (an edge in a dungeon graph) if a passageway is indicated; e.g., in Figure 1 the player may navigate from the bottom-left room to the bottom-middle without a key. However, as seen in Figure 1, the player may have to acquire and expend a key to navigate between two rooms separated by a locked door. For example, a key is needed to unlock the door between the bottom middle room and the singleton room directly to the north.

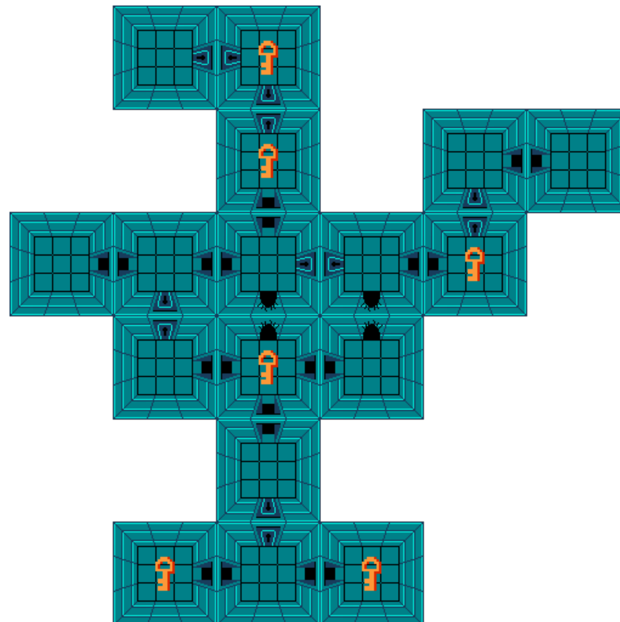

Figure 1: The "Eagle dungeon" from *Legend of Zelda*. Keys can be acquired in rooms with a key icon. Closed doors may be unlocked.

We share a few assumptions about dungeon navigation and the shortest dungeon walk problem. In the *Legend of Zelda*, players may navigate a dungeon by blasting through select walls using bombs. We assume an infinite number of bombs and complete knowledge of rooms, room contents, cracks in the wall (where bombs will be effective), etc. We also note that once a door is unlocked, it remains unlocked.

Copyright © 2024 by the authors.
This open access article is published under the Creative Commons Attribution-NonCommercial 4.0 International License.

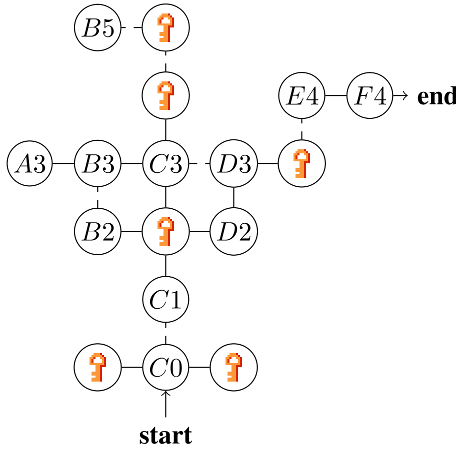

Figure 2: The “Eagle dungeon” as a dungeon graph. Dashed edges are locked doors; solid edges are unlocked doors. Nodes are labeled using a grid: top-to-bottom rows are 5 down to 0 while left-to-right columns are A through F.

For clarity in presentation of the “Eagle dungeon”, we use the graph in Figure 2 corresponding to the dungeon in Figure 1. In Figure 2, doors are edges: there are six locked doors indicated with dashed lines while unlocked doors are solid. We also label the six rooms (nodes) with keys; they are $B0$, $D0, C2, E3, C4, C5$.

Our goal is to compute a shortest walk of the dungeon. We note that there does not exist a shortest *path* with the dungeon in Figure 2 since the locked door $(C0, C1)$ must be traversed to advance to the rest of the graph. Thus, in order to unlock the door at $(C0, C1)$, the player must obtain a key from either room $B0$ or $D0$. Hence, shortest successful traversal of such a dungeon requires a player pursue a shortest walk and not a shortest path as it may be the case that rooms must be visited multiple times.

In order to compute a shortest walk, we use a brute force approach that takes as input a dungeon graph (such as Figure 2), a set of rooms with keys K, and a set of locked doors L. Each walk is constructed using an ordered subset of keys and an ordered subset of locked doors of the same magnitude. The walk then takes a shortest available path (i.e., excluding locked edges) to each key, and marks each edge as available in order. For example, a successful walk is constructed from the ordered key set $\{B0, C4, E3\}$ with the locked edge set $\{(C0, C1), (C3, D3), (E3, E4)\}$. This gives us a walk consisting of the ordered list of 12 rooms $[C0, B0, C0, C1, C2, C3, C4, C3, D3, E3, E4, F4]$ of length 11[1]. However, this is not a shortest walk of this dungeon.

A shortest walk can be constructed from the key set $\{B0, E3\}$ with the locked edge set $\{(C0, C1), (E3, E4)\}$. This gives us a walk $[C0, B0, C0, C1, C2, D2, D3, E3, E4,$ $F4]$ of length 9. Note that this walk happens to be a shortest path in the dungeon with a detour to get the key at $B0$. In a dungeon graph, there may be multiple unique shortest walks as is evident by opting for the key at $D0$ in the walk: $[C0,$ $\boldsymbol{D}0, C0, C1, C2, D2, D3, E3, E4, F4]$.

[1] Walk lengths are based on the number of edges traversed.

In this paper, we introduce the *Shortest Dungeon Walk Problem*, describe our graph-based representation of the problem in §2. In §3, we present a solution algorithm and prove its correctness. We show that the problem is NP-complete in §3.1. Finally, in §4, we demonstrate the efficacy of our solution with dungeon graphs from the *Legend of Zelda*.

2 Representation

While the origin of the *Shortest Dungeon Walk Problem* problem is from a video game, the problem and its solution is not limited in its application. We approach the shortest walk problem generally although the names of our constructs may seem specific to the problem. Finding a shortest walk begins with the underlying graph structure in which we encode rooms as nodes and doors as edges.

Definition 1 (Dungeon Graph)**.** *A* dungeon graph *is an undirected graph $G(R, D)$ consisting of a node-set of rooms R and an edge-set of doors $D \subseteq R \times R$.*

We define particular subsets of R and D in a dungeon graph G. We say $K \subseteq R$ is the set of rooms containing keys. Similarly, we say $L \subseteq D$ is the set of doors that are locked, thus prohibiting traversal from one room to another (unless a key is obtained). We may thus represent a dungeon graph with greater granularity as $G(R[K], D[L])$. For convenience, we define $K^C = R \setminus K$ and $L^C = D \setminus L$.

Since (un)lockability of doors is important, we introduce the concept of a position in a dungeon graph. When context is clear, we will refer to a *dungeon position* simply as a *position*.

Definition 2 (Dungeon Position)**.** *A* dungeon position *p in a dungeon graph $G(R, D)$ is a tuple $p = (r, k)$ with $r \in R$ and $k \in \mathbb{N}_0$ the current number of keys available to unlock doors.*

In order to compute a shortest walk in a dungeon graph, each movement may have side-effects (e.g., picking up a key, using a key to unlock a door, etc.). We describe each type of ‘move’ from one position to another in a dungeon graph.

Definition 3 (Dungeon Move)**.** *A* dungeon move *m is the transition from a dungeon position p in a dungeon graph $G(R[K], D[L])$ to a new position p' in $G(R[K'], D[L'])$ accounting for state changes induced by the move. A move is* valid *if and only if it matches one of the following cases.*

Case 1: Unrestricted movement. *Transition without using a key from a room r to an unlocked room r' that does not contain a key: $(r, k), G \rightarrow (r', k), G$ where $(r, r') \in L^C$ and $r' \notin K$.*

Case 2: Unlocking a locked door. *Consuming a key by moving from a room r to a room r' that does not contain a key: $(r, k), G(R, D[L]) \rightarrow (r', k'), G(R, D[L'])$ where:*

- $(r, r') \in L$,
- $r' \in K^C$,
- $k > 0$,
- $k' = k - 1$, *and*
- $L' = L \setminus \{(r, r')\}$.

Case 3: Unlocked door with key at the destination. *Moving from a room r to an unlocked room with key r' and picking up that key:* $(r, k), G\,(R[K], D) \rightarrow (r', k'), G\,(R[K'], D)$ *where:*

- $(r, r') \in L^C$,
- $r' \in K$,
- $k' = k + 1$, *and*
- $K' = K \setminus \{r'\}$.

Case 4: Locked door with key at the destination. *Transitioning from room r to room r' by unlocking, and picking up a key in r':* $(r, k), G\,(R[K], D[L]) \rightarrow (r', k), G\,(R[K'], D[L'])$ *where:*

- $(r, r') \in L$,
- $r' \in K$,
- $L' = L \setminus \{(r, r')\}$,
- $k > 0$, *and*
- $K' = K \setminus \{r'\}$.

A move in a dungeon graph only alters the underlying subsets of rooms and doors; for example, consuming a key by unlocking a door shifts a door from being locked to unlocked. Generally, a move m in a dungeon graph $G\,(R, D)$ results in a corresponding graph $G'\,(R', D')$. However, rooms and doors are static: it is always the case that $R = R'$ and $D = D'$. Our last definition formalizes a walk in a dungeon graph as a sequence of dungeon moves.

Definition 4 (Dungeon Walk)**.** *For a dungeon graph $G\,(R, D)$, a* dungeon walk *w is a sequence of valid* dungeon move*s from a start room $a \in R$ to a goal room $g \in R$. The length of w, denoted $|w|$, is the number of valid moves in the sequence.*

A *dungeon walk* can be uniquely identified by the rooms that it passes through, as there is only ever at most one valid *dungeon move* between two rooms. Unless otherwise specified, a *dungeon walk* starts with a position with zero keys and may end with a positive number of keys.

3 Algorithm

In this section, we define a bounded, brute-force algorithm to find a shortest walk in a dungeon graph from an input room to a final, goal room. This algorithm assumes that the walker possesses zero keys at the beginning of the walk.

Our solution assumes a general purpose graph shortest path algorithm is defined as $\Lambda(G, a, g)$ where $G = (N, E)$ is a graph composed of nodes N and edges E, a is the start node, and g is the final node; e.g., Dijkstra's shortest path algorithm is one such algorithm (Dijkstra 1959).

For a dungeon graph $G\,(R, D)$, Algorithm 1 computes a shortest walk from origin room $a \in R$ to goal room $g \in R$. We describe the algorithm in general before discussing details. Algorithm 1 is brute force as SHORTWALK checks every possible ordering of locked doors and every possible ordering of keys to unlock said doors. A shortest walk is constructed in BUILDWALK as the path from the start room to, in turn, each room with a chosen key, adding locked doors to the set of valid edges as we pick the key designated for use on that door.

We consider Algorithm 1 in detail. We begin with an initial, empty walk (Line 6) of infinite length (Line 7). Algorithm 1 then considers all permutations of keys (Line 10) and locked doors (Line 12) to accumulate a path. However, we note that a walk in a dungeon graph need not accumulate all keys in order to be a shortest walk. Therefore, while the loops on Line 10 and Line 12 search permutations of the sets, the outer loop (Line 8) ensures we search key and door sets of increasing size (starting with 0 keys).

Algorithm 1 Naive Shortest Dungeon Walk

1: **function** SHORTWALK($G\,(R[K], D[L])$, a, g)
2: G: Dungeon Graph
3: $a \in R$: starting room
4: $g \in R$: goal room
5:
6: $\lambda \leftarrow \emptyset$ ▷ Empty walk
7: $|\lambda| \leftarrow \infty$ ▷ Infinite length
8: **for all** $r \in [0 \ldots |K|]$ **do** ▷ For all rooms with keys
9: ▷ For all sequences of rooms with keys
10: **for all** $k \in r$-length permutations of K **do**
11: ▷ For all sequences of locked doors
12: **for all** $\ell \in r$-length permutations of L **do**
13: $w \leftarrow$ BUILDWALK(G, k, ℓ, a, g)
14: ▷ Save current shortest walk
15: **if** $|w| < |\lambda|$ **then**
16: $\lambda \leftarrow w$
17: **return** λ
18:
19: **function** BUILDWALK($G\,(R, D)$, k, ℓ, r_a, r_g)
20: G: Dungeon Graph
21: k: list of rooms with keys
22: ℓ: list of locked doors
23: $r_a \in R$: starting room
24: $r_g \in R$: goal room
25:
26: $w \leftarrow \emptyset$ ▷ w: walk accumulator
27: $r \leftarrow r_a$ ▷ r: current room
28: ▷ Seek a shortest path between each locked door
29: **for all** $i \in [0 \ldots n)$ **do** ▷ Half-open range
30: $\ell' \leftarrow \ell[0, \ldots, i-1]$
31: $k' \leftarrow k[i]$ ▷ Zero indexed
32: $w \leftarrow w + \Lambda((R, D + \ell'), r, k')$
33: ▷ Update r for next path between locked doors.
34: $r \leftarrow k'$
35: ▷ Append path from last room to goal
36: $w \leftarrow w + \Lambda((R, D + \ell), r, r_g)$
37: **return** w

The core of the shortest walk algorithm is an accumulation of shortest paths between rooms with keys and locked doors defined in BUILDWALK starting on Line 19. The loop starting on Line 29 sequentially unlocks the list of locked doors (ℓ) using the next key in the sequence (given by k). We then sequentially seek a shortest path between our current position and the next locked door (Line 32) with a modified version of the original graph accounting for unlocked doors. We note that if the path-finding algorithm (Λ) fails to find a path, likely due to disconnectivity due to locked edges, it returns an infinite length path. Last, on Line 36 we then finish the current walk by appending a shortest path from the current room to the final room.

Since Algorithm 1 considers all such path permutations

of keys and locked doors, we will find a shortest walk; we formalize this notion in the following lemma.

Lemma 3.1 (Correctness of Algorithm 1)**.** *If a shortest walk w exists from a start room $a \in R$ to an end room $g \in R$ in a dungeon graph $G\,(R[K], D[L])$, Algorithm 1 will identify a walk not longer than w.*

Proof. Let w be a shortest walk in G. Since a walk can be classified by the ordered set of keys K that it picks up and the ordered set of locked doors L that it unlocks, there are three potential cases for the relationship between $|K|$ and $|L|$:

1. $|\mathbf{K}| < |\mathbf{L}|$. In this case, more locked doors are unlocked than keys are collected. This constructs an invalid walk, as the dungeon move to unlock a door after using up all keys would be invalid.
2. $|\mathbf{K}| > |\mathbf{L}|$. In this case, more keys are collected than locked doors are unlocked. This walk is strictly longer than or equivalent to a walk which has K as the prefix with the same cardinality as L. By triangle inequality property, this walk will always require more valid moves to travel from a room r_1 to a room r_2 to a room r_3 than directly from r_1 to r_3, unless r_2 is on a shortest path from r_1 to r_3. For the case where r_2 is on a shortest path, we say K is allowed to omit a key from the ordered set if and only if the walk would also be valid on a dungeon graph that did not contain that key. Thus, this case reduces to $|K| = |L|$ with respect to a shortest walk.
3. $|\mathbf{K}| = |\mathbf{L}|$. In this case, the same number of keys are collected as doors are unlocked.

Thus, a shortest walk w must be identified by some K, L where $|K| = |L|$. Algorithm 1 constructs a walk (Line 13) for every permutation of keys (Line 10) and of doors (Line 12) of every length (Line 8), then chooses a shortest one out of those (Line 14). The constructed walk is a shortest available for that choice of keys and doors, as the shortest available path between each key is taken using Λ on Line 32 and Line 36.

As such, Algorithm 1 constructs walk w from its key and locked door sets K, L, and either produced w or some other walk not longer than w. Hence, Algorithm 1 produces a shortest walk. □

3.1 Complexity

In this section we consider the complexity of Algorithm 1 as well as the complexity class of the *Shortest Dungeon Walk Problem*.

Lemma 3.2 (Complexity of Algorithm 1)**.** *Let $G\,(R[K], D[L])$ be a dungeon graph, $a \in R$, and $g \in R$. Computing a shortest walk from a to g in G using Algorithm 1 is $O(K! \cdot L! \cdot K \cdot O(\Lambda))$, where $O(\Lambda)$ is the complexity of a shortest pathfinding algorithm Λ over G.*

Proof. The complexity can be constructed by multiplying the complexity of the loops in Algorithm 1: for all permutations of keys K on Line 10 ($O(K!)$), for all permutations of L doors on Line 12 ($O(L!)$), for all keys in the subset on Line 29 ($O(K)$), and computing a shortest path to that key on Line 32 ($O(\Lambda)$). □

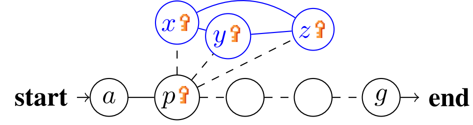

Figure 3: A dungeon graph to solve the Traveling Salesperson Problem for the connected subgraph $\{x, y, z\}$ (highlighted). Dashed edges are locked.

As such, this algorithm is combinatorially slow for non-trivial sizes of L and K. Our experimental analysis in §4 relies on the fact that dungeons are generally 'small' in practice and thus the sets L and K should be small as well.

Lemma 3.3 (Complexity Class of the *Shortest Dungeon Walk Problem*)**.** *The* Shortest Dungeon Walk Problem *is NP-Complete.*

Proof. We show that the Shortest Dungeon Walk Problem is NP-Complete by reducing from the Traveling Salesperson Problem (Cormen, Leiserson, and Rivest 1997, §36.5.5). That is, given a polynomial time solution for the Shortest Dungeon Walk Problem, we can solve the Traveling Salesperson Problem in polynomial time. We do so by converting the Traveling Salesperson Problem into the Shortest Dungeon Walk Problem. Our reduction is multi-step process for how to convert an arbitrary undirected graph H to a dungeon graph G.

Suppose $H(N, E)$ is an undirected graph: nodes N and edges E. We then begin constructing a dungeon graph $G\,(R, D)$ by adding all edges E to doors D. We then place keys in all the original nodes N resulting in dungeon graph $G\,(R[K], D[L])$ where $R = N$, $K = N$, $D = E$, and $L = \emptyset$. This first part of the construction is shown in blue in Figure 3: $N = \{x, y, z\}$.

We next add a pivot node p to G to serve as our entry and exit point into H; we add p to the set of rooms with keys ($K \leftarrow K + \{p\}$). We then add a locked door from p to every original room ($L \leftarrow \{(p, x) \mid x \in N\}$). The addition of room p and its outgoing doors can be observed in Figure 3.

We now add $|N| + 1$ rooms to R without keys ($R \leftarrow R + \{n_x \mid x \in [0 \ldots |N|]\}$). We then name one of these new non-key rooms as our start node a and one of the new non-key rooms as a goal node g. We draw an unlocked edge from a to p ($D \leftarrow D + \{(a, p)\}$) as access to the dungeon. Hence, we can now access the original graph H with the dungeon sub-walk $[a, p, x]$ for some $x \in N$.

For our exit path, we draw locked edges between each room in the dungeon that was not in the original graph H excluding a: $D \setminus (N \cup \{a\})$. The result is that each of these exit path rooms has degree 2 except for g, which has degree 1. In Figure 3, the exit path set of $|N|$ nodes consists of the three nodes, two of which are unlabeled and the last node ending in g. This construction guarantees a path between p and g consisting of $|N|$ locked edges. This concludes of construction of G.

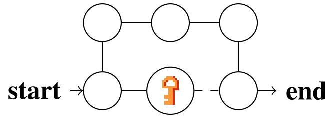

Figure 4: Counterexample of the "fewest keys" optimization

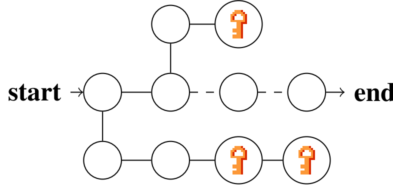

Figure 5: A counterexample of "smallest deviation" optimization

Solving the Shortest Dungeon Walk Problem for G now necessarily also solves the Traveling Salesperson Problem for H: traversing from a to p, then a TSP solution traversing H and ending up back at p, and concluding with the exit path nodes in G ending in g. We observe that the constructed graph G has a satisfactory number of keys. If the original graph has v nodes, our constructed graph has $v + 1$ keys: one at each original node, and one at p. v keys must be used to travel from p to g by construction. As such, exactly one edge from p to the original graph can be unlocked, and every original node must be visited to collect their key. The subset of the Dungeon Path between visits to p thus solves the Traveling Salesperson Problem.

We conclude that the Shortest Dungeon Walk Problem must be at least as hard as the Traveling Salesperson Problem. As the Traveling Salesperson Problem is NP-Complete (Cormen, Leiserson, and Rivest 1997, Thm. 36.15), the Shortest Dungeon Walk Problem must also be NP-Complete. □

3.2 Non-Total Optimizations

For procedural generation, we describe some possible optimizations that may not always find a shortest dungeon walk.

Fewest Keys. Is it possible that a walk that uses the fewest number of keys will be a shortest walk? A counterexample is shown in Figure 4 in which picking up and using a single key will reduce the walk from 5 rooms to 3 rooms. Hence, this optimization is non-total.

On-Demand Key Acquisition using Smallest Path Deviation. Consider an optimization where we compute a shortest path between start and goal node ignoring locked edges. Then, we modify the path to pick up enough keys. The simplest method of acquiring a key would be to seek the key which causes the smallest deviation from the current walk. Unfortunately, larger deviations can result in ultimately shorter walks when multiple keys need to be picked up. In Figure 5, the initial smallest deviation would pick up the top key at a cost of distance 4, but a second key needs to be acquired. Hence, the first of the bottom keys must be obtained at a cost of 6 for a total of distance 15 to walk to the last node. If, however, the walk acquired both keys from the bottom rooms, the total walk cost is 11. Thus, this optimization is non-total.

Table 1: Dungeon characteristics of the first 8 dungeons of *The Legend of Zelda* (Playthrough 1).

Dungeon	Key Count	Locked Door Count
1	6	6
2	4	3
3	5	4
4	4	5
5	7	6
6	5	5
7	3	4
8	4	4

Table 2: Graph characteristics of the first 8 dungeons of *The Legend of Zelda* (Playthrough 1).

Dungeon	Node Count	Edge Count	Avg. Connectivity
1	17	18	1.12
2	18	23	1.47
3	18	21	1.31
4	20	23	1.25
5	23	25	1.07
6	25	27	1.07
7	33	36	1.07
8	25	26	1.04

4 Experimental Analyses

We used the dungeons from *The Legend of Zelda* (Nintendo 1986) as a benchmark set of dungeons built in the dungeon room style to verify Algorithm 1 and gather empirical data. This data set gives us human-designed dungeons of a reasonable size for actual human play.

Background. We implemented Algorithm 1 in the Rust[2] programming language and benchmarked it using the Criterion statistical benchmarking harness[3] over the first eight dungeons of the first playthrough of *The Legend of Zelda*. Benchmarks were run on a 1.7 GHz Intel Xeon processor with 12 cores and 32 GB RAM. In order to optimize our algorithms, we note that the input graph is fixed. Hence, our first implementation step precomputes all calls to the shortest path function Λ using Dijkstra's algorithm (Dijkstra 1959). Our implementation is parallelized at the level of the key set permutations (Line 10) to scale with all available cores.

The characteristics of the first eight dungeons of *Legend of Zelda* is listed in Table 1. Dungeons 1 and 5 are the most difficult of the set to navigate, as each has 6 locked doors to consider. Dungeons 4 and 7 do not have enough keys for the number of locked doors; every other dungeon has at least as many keys as it has locked doors. (The *Legend of Zelda* allowed the player to carry excess keys from dungeons into other dungeons.) Characteristics of the corresponding graph for each dungeon is given in Table 2. We compute average connectivity of a graph as the average number of fully distinct paths between any two nodes (Beineke, Oellermann, and Pippert 2002). Dungeon 2 is the most connected with an average connectivity of 1.47, and Dungeon 8 is the least

[2] https://www.rust-lang.org/

[3] https://bheisler.github.io/criterion.rs/book/

Table 3: Benchmarking Algorithm 1; units are in ms and reported to three significant figures.

Dungeon	Mean	SD	Median	MAD
1	479	5.15	479	3.77
2	0.800	0.059	0.787	0.011
3	5.80	0.117	5.79	0.075
4	6.79	0.129	6.77	0.041
5	6030	25.8	6030	19.7
6	19.3	0.108	19.3	0.097
7	1.23	0.017	1.22	0.010
8	2.21	0.029	2.20	0.015

with 1.04.

Results. Table 3 shows the mean, standard deviation, median, and median absolute deviation duration of running Algorithm 1 on each of the eight dungeons. As the mean and median show a maximum difference of 2%, we can conclude that the samples taken fall into a symmetrical distribution. A maximum standard deviation of 7% additionally suggests a tight normal distribution. Figure 6 compares the mean durations on a log scale. We can see that Dungeons 1 and 5 clearly have the longest running times, correlating with their greater number of locks and keys than the other dungeons.

Figure 6b shows a exponential regression fit for the mean runtime of $15.7\,e^{0.29x}$. The experimental results confirm that the runtime of Algorithm 1 is exponentially related to the product of the number of locks and keys.

5 Related Applications

Our solution is to the Shortest Dungeon Walk Problem is similar to the forward algorithm for finding a shortest path in a petri net (ÖZKAN 2016) in that it relies upon a shortest path algorithm like Dijkstra's or Floyd-Warhsall. We go a step further than (ÖZKAN 2016) to show that this problem space is NP-Complete even in our simpler context of a game dungeon.

We envision this technique as an ideation tool for large dungeons as we have shown that, without optimization, a complete solution is limiting for iterative procedural generation of a dungeon For small dungeons, we have shown there is efficacy for the algorithm as a heuristic. Our discussion will focus on related applications. *The Legend of Zelda: Link's Awakening* (Nintendo 2019) offers a mode where the user can create a dungeon from pre-built rooms and upload it to share with other players. This algorithm offers a way dungeons could be grouped for presentation, but does not consider a shortest walk or other heuristics in measuring goodness of the resulting dungeon graph.

The level generation tool OBLIGE (Apted 2017) for classic DOOM generates levels by using a layout algorithm similar to dungeon layout, where prebuilt nodes are strung together by connections that may or may not be locked to make a level. This algorithm could serve to help orchestrate generation of more interesting levels where more of the level is required to be visited. That is, a shortest walk that avoids sections of a dungeon would influence a designer or procedural generation technique to add required components to

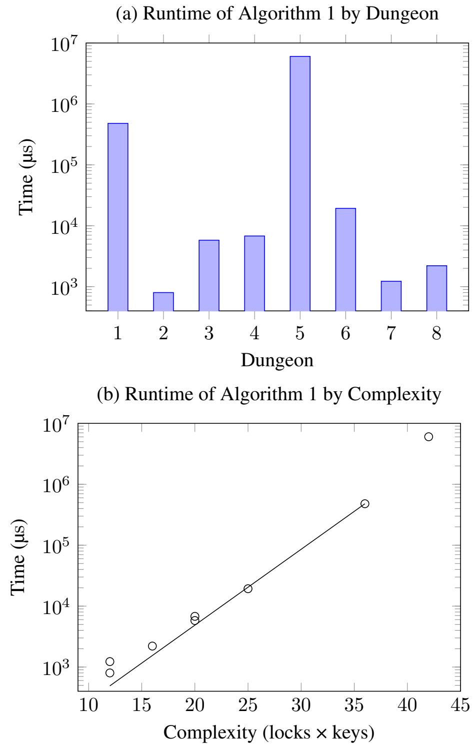

Figure 6: Runtime of Algorithm 1 on each of the *Legend of Zelda* dungeons.

avoid 'deserts' in a dungeon.

A Link to the Past Randomizer (Veetorp et al. 2017) offers a way to shuffle item locations around the map in *The Legend of Zelda: A Link to the Past*. Important items such as keys and unimportant items are redistributed throughout the game such that a new path has to be taken to complete the game. The 'logic' of the shuffle ensures that the game is still completable after the shuffle. This algorithm could be used as an input into the shuffle logic to help control where keys are located to create interesting dungeon layouts.

6 Conclusions

This work has formalized the representation of room-based dungeon graphs as seen in Nintendo's *Legend of Zelda*. We have proposed the Shortest Dungeon Walk Problem in the context of this dungeon graph-based representation. Given a dungeon graph, we have proposed an algorithm for computing a shortest walk and showed its correctness. Our main contribution is proving that the *Shortest Dungeon Walk Problem* is solvable but NP-complete. Last, we demonstrated the efficacy of our algorithm to a particular class of dungeon graphs from the *Legend of Zelda*. That is, the performance of solving the *Shortest Dungeon Walk Problem* for real dungeons completes in a reasonable amount of time despite the theoretical complexity thus facilitating shortest walk as a heuristic for procedural generation of graph-based dungeons.

References

Apted, A. 2017. OBLIGE Level Maker. Website. http://oblige.sourceforge.net/.

Beineke, L. W.; Oellermann, O. R.; and Pippert, R. E. 2002. The average connectivity of a graph. *Discrete Mathematics* 252(1):31 – 45.

Bond, J. G. 2022. *Introduction to Game Design, Prototyping, and Development*. Addison-Wesley Professional, 3rd edition.

Colp, Tyler. 2023. Diablo 4's season 2 patch fixes almost every problem i've had with it since launch. https://www.pcgamer.com/diablo-4s-season-2-patch-fixes-almost-every-problem-ive-had-with-it-since-launch/.

Cormen, T. H.; Leiserson, C. E.; and Rivest, R. L. 1997. *Introduction to Algorithms*. MIT Press.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Hello Games. 2016. No man's sky. http://www.hellogames.org/.

Khalifa, A.; Perez-Liebana, D.; Lucas, S. M.; and Togelius, J. 2016. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16.

Nintendo. 1986. The Legend of Zelda. Game Cartridge.

Nintendo. 2019. The Legend of Zelda: Link's Awakening for Nintendo Switch. Not yet published.

ÖZKAN, H. A. 2016. Shortest path algorithms for petri nets. *IU-Journal of Electrical Electronics Engineering* 16(2).

Smelik, R.; Tutenel, T.; de Kraker, K. J.; and Bidarra, R. 2010. Integrating procedural generation and manual editing of virtual worlds. In *Workshop on Procedural Content Generation in Games*, PCGames '10.

Valtchanov, V., and Brown, J. A. 2012. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12.

Veetorp; Karkat; Christos0wen; Smallhacker; and Dessyreqt. 2017. A Link to the Past Randomizer. https://alttpr.com.