MIDTERM Is a Deterministic Technique to Exit Recursive Mazes

Charles Bouillaguet
LIP6 laboratory, Sorbonne Université, Paris, France

Orel Cosseron
Independent researcher, Paris, France

Abstract

A recursive maze is a maze that contains copies of itself (that themselves contain copies of themselves, etc.). To exit the maze or reach the goal, each recursive block that has been entered must be exited. These once-popular puzzles are difficult to solve by hand, and this begs for an algorithmic solution.

It has been observed many times in the past that a recursive maze can be represented by a deterministic pushdown automaton. Finding a path, possibly the shortest, that leads to an exit therefore reduces to finding a word in a context-free language described by such an automaton. The problem is well-known to be decidable, and there is a classical algorithm for this task. We present a new algorithm, MIDTERM, with improved complexity compared to existing solutions.

MIDTERM improves on a previous attempt called LONGTERM (Obviously Not a Good Technique to Exit Recursive Mazes).

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Theory of computation → Graph algorithms analysis

Keywords and phrases Recursive maze, pushdown automaton, reachability, context-free grammar, graph rewriting

Digital Object Identifier 10.4230/LIPIcs.FUN.2026.9

Related Version *Extended full version (with proofs and more funny "details")*: `https://hal.science/hal-05517329`

Acknowledgements We thank Ambroise Fleury for involuntarily suggesting this work, and the anonymous reviewers for their constructive feedback as well as for spotting many, many typos.

1 Introduction

A *recursive maze* is a maze that contains copies of itself. As the adage goes, a picture is worth a thousand words, and the concept is therefore best illustrated by Figures 1 and 2. Recursive mazes have been invented by Mark J. P. Wolf and the first one was published in 1996 under the name "*fractal maze*" (*Extropy* #17, p. 67, available in the internet archive). Similar puzzles later appeared both in print and online. Recursive mazes are notoriously difficult to solve by hand, without taking notes and/or using an *ad hoc* computer program to track one's position in the maze. Several such program exist, notably to play in a web browser[1].

At first glance, a recursive maze looks like the floor plan of an integrated circuit, that itself contains "boxes" representing inner integrated circuits. There are *wires* connecting the "outer" IO pins (on the edge of the circuit) to the "inner" IO pins of the boxes contained inside. The boxes are often distinguished with capital letters. There are also special locations, such as ⊕ and ⊖ in Fig. 1, or the starting point ("you are here") in Fig. 2. The aim of

[1] For instance `https://amurielagi.github.io/fractal-maze/` or [shameless advertisement] `https://orelcosseron.github.io/fractal-maze/`.

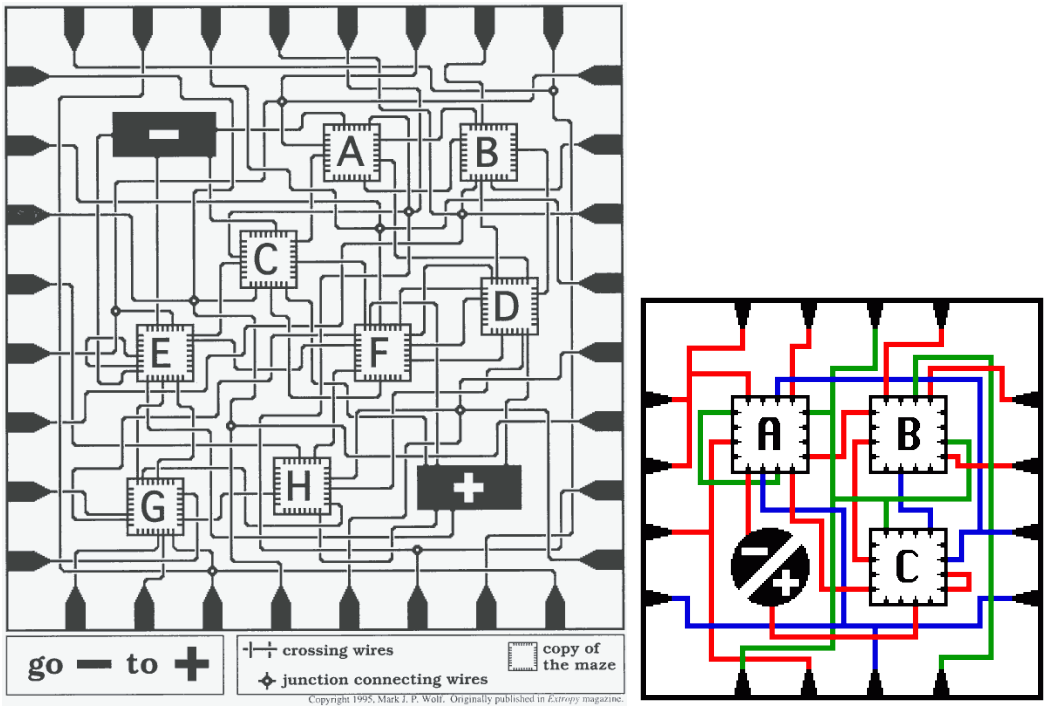


Figure 1 Two early recursive mazes designed by Mark J. P. Wolf. First published in Extropy #17, Fall 1996, page 67 (left) and in the "Bogglers" column of the June 2004 issue of Discover Magazine, page 86 (right). See also the book [13] by the same author.

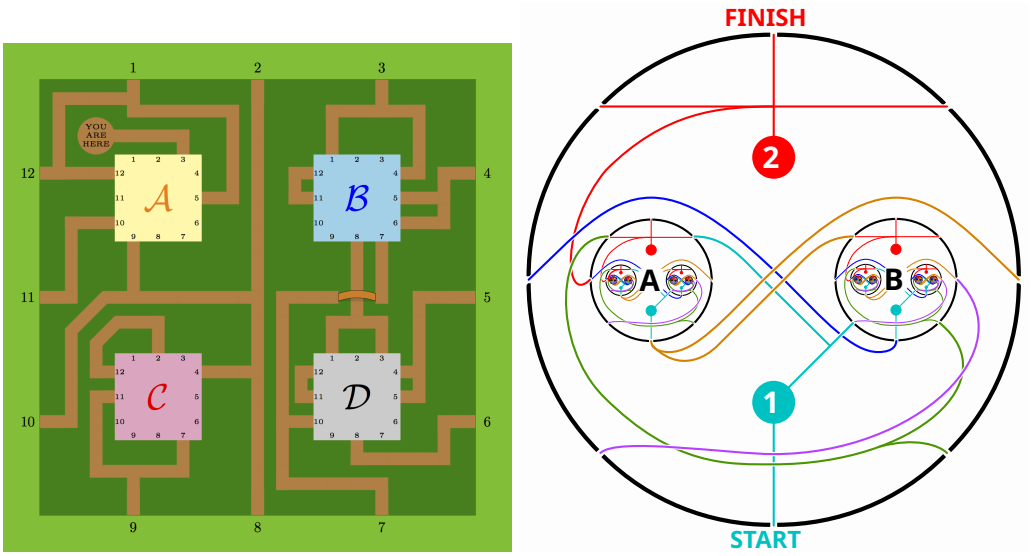


Figure 2 Two other recursive mazes designed by Mike Earnest (published on July 2016 as a StackExchange puzzle, left) and Tristan "Siggy" Miller (published on October 2010 on the Skeptic's Play blog, right).

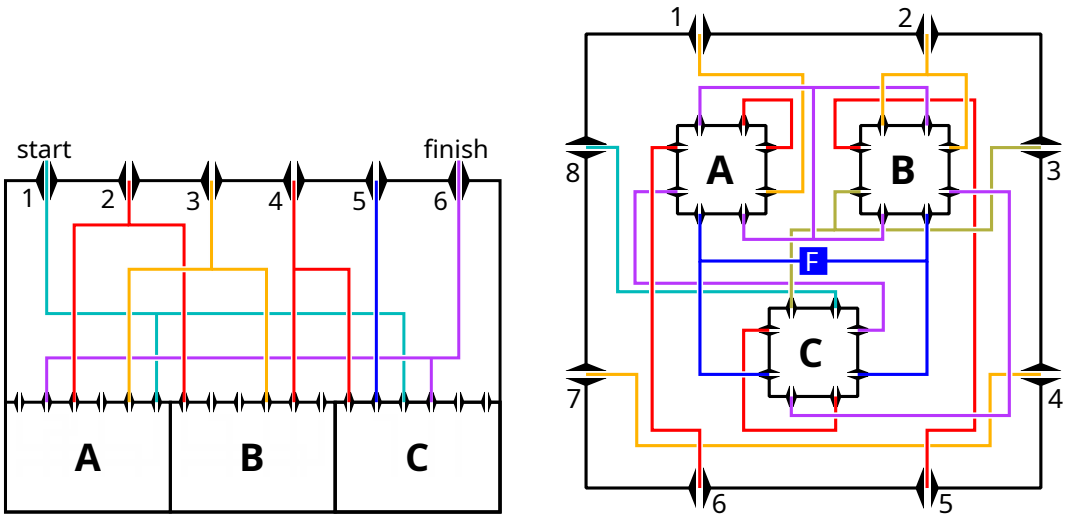
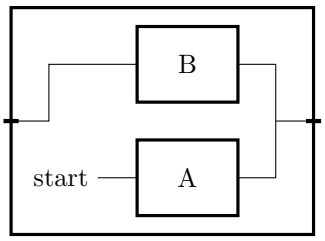


Figure 3 Two unsolvable recursive mazes by Tristan "Siggy" Miller (published on October 2023 on the Freethought blogs). On the right, the goal is to reach the "F" from any outer IO pin. These mazes are unsolvable with the normal rules, but what if one allows transfinite moves?

the puzzle consists in following the wires from point A to point B: from $\ominus$ to $\oplus$ in Fig. 1, from the starting point to the outside or to the finish line in Fig. 2. The trick is that, when following a wire, we may hit one of the inner boxes. In this case, we "enter" the box, which is a copy of the whole maze. Each entered box must be exited, by reaching one of its outer IO pin. This is where the real fun begins. For one, exiting may not be possible at all. Consider for instance the innocent-looking maze:



From the starting point, one has to enter A (from the left). But from there, the only option is to follow the wire that leads to B, and enter on the left. From there, we are stuck in an infinite loop of entering B, and there is no way that we can exit the maze.

It follows that simple graph-search techniques are not sufficient to decide if there is a way to exit the maze, because the graph that consists of all possible paths is infinite (even though it has a regular structure). Sometimes it is easy to determine that the maze is unsolvable, for instance if the available paths always enter a recursive box (without any option to exit). This is clearly the case on the left maze of Fig. 3, but what about the one on the right? Because of the wire between pins 4 and 7, there *is* potentially a way to exit a recursive box. More precisely, any box entered on pin 4 or 7 can be exited by the opposite pin (and that's it). So the only way to reach the goal would be to exit C by pin 4 or 7... but this requires C to be entered by pin 7 or 4, in which condition the goal was already reachable. Therefore, the target cannot be reached.

When the maze is known to be solvable (i.e., there is a finite path from the start to a target), then a breadth-first-search will find it. It will even find the shortest possible one. However, the number of explored nodes might be exponential in the size of the maze.

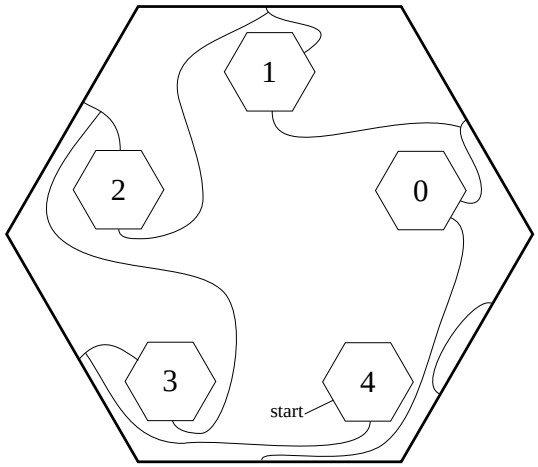


Figure 4 The "power-of-two" maze.

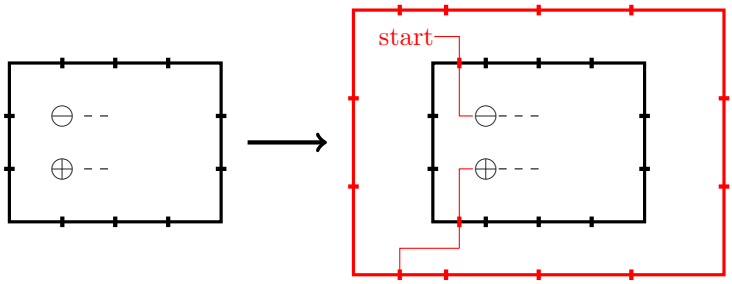


Figure 5 Converting a maze from "reaching an internal goal" to "getting out". The new red maze contains all the content of the original maze plus an extra recursive box that connects "start" to $\ominus$.

And there is worse: the length of a shortest path can itself be exponential in the size of any reasonable description of the maze. Consider for instance the maze shown in Fig. 4. A moment's thought will reveal that there is a path from the north-east to the south that goes through 0. Then, there is a path from the north to the south that enters 1, goes through 0, exits 1 and go through 0 again. Then there is a path from the north-west that enters 2, goes through 1, exits 2 and goes through 1 again, etc. Generalizing this construction to an n-gon yields a family of recursive mazes where the only way to exit makes an exponential number of steps in the size of the maze.

We first take one aspect of the problem off the table: out of the two possible goals (getting out of the maze, or reaching $\oplus$ starting from $\ominus$), only getting out is interesting. More precisely, every maze where the goal is to reach a specific wire can be converted into another maze where the goal is to exit. Fig. 5 shows how. Interestingly, we do not know how to make the conversion the other way around.

In this article, we consider old and new algorithms to decide, in polynomial time, if the maze can be exited or not. All these algorithms produces (in polynomial time) a description of a solution path, even if it is of exponential length. We compare their complexities, expressed in terms of various characteristics of the maze: the number of wires W, the number of recursive boxes B and the number E of possible ways to enter a recursive box (8 in the maze on the left of Fig. 6).

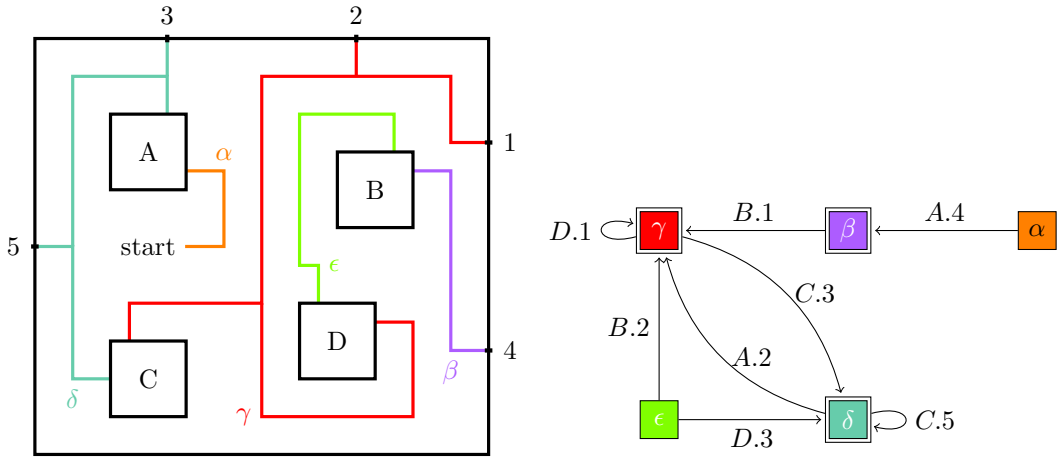


Figure 6 A simple maze of our own design (left) and its "dual" representation (right). Wires with a double-box are "accepting" (connected to an outer IO pin).

After presenting some context, we review in section 4 the classical technique that decides emptiness of the context-free language associated to a deterministic pushdown automaton built from the description of the maze. It runs in time $\mathcal{O}\left(W^2EB\right)$. In section 5, we review improved algorithms to decide reachability of arbitrary states in pushdown automata, that yield a faster decision procedure with complexity $\mathcal{O}\left(\mathcal{W}^3\right)$. Finally, in section 6 we discuss a new algorithmic technique to compute equivalence classes of wires inside a recursive maze. In section 7 we discuss the efficient implementation of this procedure, and show a version that terminates in time $\mathcal{O}\left(WB + E\alpha(W)\right)$, where α is the inverse-Ackermann function, improving the state of the art. We discuss how to bring this down to $\mathcal{O}\left(E \log E\right)$, but this seems to either require: a) cheating, or b) randomization. Lastly, in section 8, we discuss how to actually construct an exit path (this was the whole point of the paper, isn't it?).

"Solving" recursive mazes in linear time, e.g. $\mathcal{O}\left(W + E + B\right)$ is left as an intriguing open problem.

2 Topological representation of recursive mazes

A recursive maze has several parameters: the number B of recursive boxes (from 2 to 8 in the given examples), the number of output pins, the number W of wires. As a convention, inner boxes are identified by capital letters (A, B, C, ...), outer IO pins are referenced by numbers and wires are referenced by Greek letters. An inner IO pin (on a recursive box) is referenced by the identifier of the box and that of the pin. For example, on Fig. 3 (left), "1" is connected to "A.6" and "C.3". An *endpoint* is either an outer pin, or an inner pin. A wire connects a set of endpoints. For instance, the wires of the simple maze shown on the left of Fig. 6 are:

$$\{\texttt{start}, A.4\}, \{4, B.1\}, \{1, 2, C.3, D.1\}, \{3, 5, A.2, C.5\}, \{D.2, B.2\}$$

This completely represents the maze, and can easily be encoded as a text file. The size of this text file is a good measure of the size of the maze. Note that a recursive maze is therefore a hypergraph whose vertices are the endpoints. It even seems legitimate to consider this as "the" abstract description of the maze. Note that in principle each endpoint must appear in at most one wire, and each outer IO pin must appear in exactly one wire for the maze to be well-formed. This implies that $E \leq BP$ and $W \leq P$.

An alternate, "dual" description is also useful (Fig. 6, right): it is a directed graph whose nodes are the wires of the maze, and where there is a directed edge $\alpha \xrightarrow{X.i} \beta$ if it is possible to go from wire α to wire β by entering the recursive box labelled X using its pin labelled i. As before, E denotes the number of edges of this graph. The dual representation graph can be assembled in essentially linear time from the previous "hypergraph" representation:

- First, associate each outer pin to the wire that contains it.
- Initialize an empty directed graph G on the wires.
- Then, for each wire ϕ, do:
  - For each inner pin $X.i$ in ϕ, add an edge labelled by $X.i$ from ϕ to the wire containing the outer pin i (if it does not already exist).

It follows from the well-formedness conditions that all edge labels are distinct in the dual representation graph. The original "hypergraph" representation can be reassembled from the dual representation graph, under the assumption that the initial location (`start`) is always incident to the wire α. An edge $\alpha \xrightarrow{X.i} \beta$ means that the outer IO pin i belongs to wire β, and that $X.i$ belongs to wire α. By convention, the starting point (`start`) belongs to wire α.

3 Recursive mazes as pushdown automata

Navigating recursive mazes in one's head is difficult because it is necessary to keep track of all the boxes that have been entered (and by what pin).

Suppose a sequence of instructions is given by the player to some kind of game engine. These instructions can be of three kinds: "from the current location, enter box X using its pin i" (encoded as $X.i$) or "from the current location, leave the current box X through the outer pin i" (encoded as $\overline{X.i}$) and finally "I have solved the problem" (`done`). For instance, this is a sequence of instructions for the the simple maze of Fig. 6 (that presumably exits):

$$A.4; B.1; \overline{B.2}; D.3; C.5; \overline{C.3}; \overline{D.1}; \overline{A.2}; \texttt{done}$$

We note that $\overline{X.i}$ could be more compactly encoded as just "i" (we do not need to know that we are inside box X to just take the exit). However, this more verbose encoding enables us to define the *mirror* operation on instructions (also denoted with an overline): $\overline{X.i}$ is the mirror of $X.i$, and vice-versa so that $\overline{\overline{X.i}} = X.i$. Applying a (legal) instruction then its mirror is a no-op (the mirror cancels the effect of the first instruction). Lists of instructions can also be mirrored, by mirroring each instruction individually and reversing their order: $\overline{A.4; B.1; \overline{B.2}} = B.2; \overline{B.1}; \overline{A.4}$. Applying a sequence of legal instructions then its mirror is a no-op.

There is a way to efficiently check if a sequence of instructions is valid, i.e. if the desired moves are actually feasible. The validity of a move depends on the last box that was entered (for instance $\overline{A.2}$ is valid inside A but not inside B, C and D). So it is required to track the current box, as boxes are entered and exited. The analogy with the call stack of a recursive function is quite obvious. This also strongly suggests some kind of stack-augmented automaton. This brilliant observation has certainly been made many times, by many people independently (including us), but a notorious and early statement is due to Peter Shor (of quantum polynomial factoring fame) in 2012, as a comment to a question on StackExchange). He writes:

> *Can't you represent any fractal maze as a pushdown automaton, where the stack corresponds to the sequence of submazes that you're in? Then the solubility question would turn into the emptiness problem for context-free languages, which is decidable.*

Making this intuition precise is easier with the dual representation graph of recursive mazes. We assume that there is an initial wire (denoted as α) where the player starts. The position of the player consists of a current wire and a stack of entered boxes. We write such a position as "wire | stack" or "$\frac{\text{wire}}{\text{stack}}$". The stack is initialized with the identifier of the "main" box that contains everything, which is denoted as $\bot$. Moving between wires pushes box identifiers on the stack (when a box is entered), or pops them (when a box is left). Entering box X is always possible (if there is a wire leading to it of course), but exiting box X only makes sense if we are currently inside it. The possible transitions from a given state are thus entirely determined by the top of the stack. If there is an edge $\phi \xrightarrow{X.i} \psi$ in the dual representation of the maze, then the player may move between positions $\phi|S$ and $\psi|X;S$ (both ways) by issuing $X.i$ or $\overline{X.i}$.

We can therefore describe a deterministic pushdown automaton (DPDA) for the maze, that accepts by empty stack. Suppose the set of outer pins is $\mathcal{P}$ (think $1, \dots, n$), the set of recursive boxes is $\mathcal{B}$ (think $\bot, A, B, C, \dots$) and the set of wires is $\mathcal{W}$ (think $\alpha, \beta, \dots$).

- The set of *states* of the DPDA is $Q := \mathcal{W} \cup \{\texttt{goal}\}$.
- The *input alphabet* is $\Sigma := \mathcal{B} \times \mathcal{P} \cup \overline{\mathcal{B} \times \mathcal{P}}$.
- The *stack alphabet* is $\Lambda := \mathcal{B}$.
- The *start state* is a designated wire $\alpha \in \mathcal{W}$ (part of the specification of the maze).
- The *initial stack symbol* is $\bot$ (the global box).
- The *transition function* $f : Q \times \Sigma \times \Lambda \to Q \times \Lambda^*$ is defined as follows. For every edge $\phi \xrightarrow{X.i} \psi$ in the dual representation graph and for any box $Y \in \mathcal{B}$, $f(\phi, X.i, Y) = (\psi, XY)$ and $f(\psi, \overline{X.i}, X) = (\phi, \emptyset)$. If the goal is to exit the maze, additionally set $f(\omega, \texttt{done}, \bot) = (\texttt{goal}, \emptyset)$ for all wires ω adjacent to an outer IO pin. If the goal is to reach a specific wire ω, set $f(\oplus, \texttt{done}, \bot) = (\texttt{goal}, \emptyset)$.

This pushdown automaton accepts all sequences of instructions that solve the maze. For instance, the following is a legal sequence of transitions in the simple maze of Fig. 6 (that exits the maze):

$$\frac{\alpha}{\bot} \xrightarrow{A.4} \frac{\beta}{A\bot} \xrightarrow{B.1} \frac{\gamma}{BA\bot} \xrightarrow{\overline{B.2}} \frac{\epsilon}{A\bot} \xrightarrow{D.3} \frac{\delta}{DA\bot} \xrightarrow{C.5} \frac{\delta}{CDA\bot} \xrightarrow{\overline{C.3}} \frac{\gamma}{DA\bot} \xrightarrow{\overline{D.1}} \frac{\gamma}{A\bot} \xrightarrow{\overline{A.2}} \frac{\delta}{\bot} \xrightarrow{\texttt{done}} \underline{\texttt{goal}}$$

As a consequence, the sequences of instructions that exit the maze form a (deterministic) context-free language. We note that there are off-the-shelf software libraries to test reachability of configurations in pushdown automata such as Moped [8] or PDAAAL [6] (just to mention a few) that can therefore be used to solve recursive mazes.

4 Off-the-shelf algorithms to exit recursive mazes

Given a recursive maze, the "dual representation" graph can be built in linear time, and from there we have derived the context-free language of valid sequences of instructions. It is easy to write a context-free grammar (CFG) for this language. Emptiness of the language can be decided in time linear in the size of the grammar (it seems this was the solution implicitly suggested by Peter Shor). We now describe the classic algorithm that does this.

The non-terminal symbols are L (the language itself) and $[\alpha X \beta]$ for $\alpha, \beta \in \mathcal{W}$ and $X \in \mathcal{B}$. Intuitively, $[\alpha X \beta]$ is the set of sequences of instructions that move from wire α to wire β while exiting the box X. The derivation rules are as follows. For the whole language, we need to exit the maze, so we need to reach the outside of the starting box (popping $\bot$), from the starting wire α.

- $\mathrm{L} ::= [\alpha \bot \omega]$ (for all wire $\omega \in \mathcal{W}$)

Then for each edge $\phi \xrightarrow{X.i} \psi$ in the dual representation of the maze:

- $[\psi X \phi] ::= \overline{X.i}$
- $[\phi Y \omega] ::= X.i\ [\psi X \chi]\ [\chi Y \omega]$ (for all wires ω, χ and all boxes $Y \in \{\bot\} \cup \mathcal{B}$)

So, there are BW^2 non-terminals, and $1 + BW^2$ derivation rules per edge in the dual representation of the maze. The usual algorithm to decide emptiness operates by iteratively marking variables that produce a string of terminal symbols and/or marked variables. It can be implemented to run in constant time per derivation rule. This decides if a recursive maze can be exited in time $\mathcal{O}\left(W^2EB\right)$.

5 Getting out of recursive mazes faster with inter-dimensional wormholes

There is a more efficient technique that can be better suited to laaaaaaaaaaaarge recursive mazes. It is heavily inspired by algorithms designed almost simultaneously by Bouajjani, Esparza and Maler [1] on the one hand and Finkel, Willems and Wolper [3] on the other hand. These algorithms decide reachability properties in pushdown systems, in the context of model checking problems that could admit such a representation.

The main idea behind this algorithm is simple. If there are two edges $\alpha \xrightarrow{X.i} \beta \xleftarrow{X.j} \gamma$ in the dual representation graph, then it is possible to move from α to γ as follows: starting from wire α, enter the recursive box X using its i-th pin (this lands on wire β inside X), then leave X using its j-th pin (this lands on wire γ outside of X). Of course, the return trip is also possible.

More generally, if there is a sequence of instructions S that allows the player to travel from wire β to wire γ while exiting box X (popping X from the stack), then $X.i;S$ is a sequence of instructions that allows the player to travel from α to γ while leaving the stack unchanged. This is the main idea underlying the construction of the context-free grammar given above.

We therefore define a binary relation $\equiv$ between wires, where $\alpha \equiv \gamma$ if and only if there is a box X and two pins i, j such that the two edges $\alpha \xrightarrow{X.i} \beta \xleftarrow{X.j} \gamma$ exist in the dual representation graph. In this case we say that there is a *local wormhole* between α and γ. Note that $\equiv$ is symmetric, but not necessarily reflexive nor transitive. We then consider the following sequence of pairs of wires:

$$\begin{aligned}\Gamma_0 &= \{(\alpha,\beta) \mid \alpha \equiv \beta\} \cup \{(\omega,\omega) \mid \omega \in \mathcal{W}\}\\ \Gamma_{i+1} &= \Gamma_i \cup \{(\alpha,\gamma) \mid \exists\beta.\ (\alpha,\beta) \in \Gamma_i, (\beta,\gamma) \in \Gamma_i\}\\ &\qquad \cup \{(\alpha,\delta) \mid \exists\beta.\ \exists\gamma.\ (\alpha,\beta) \in \Gamma_i, \beta \equiv \gamma, (\gamma,\delta) \in \Gamma_i\}\end{aligned}$$

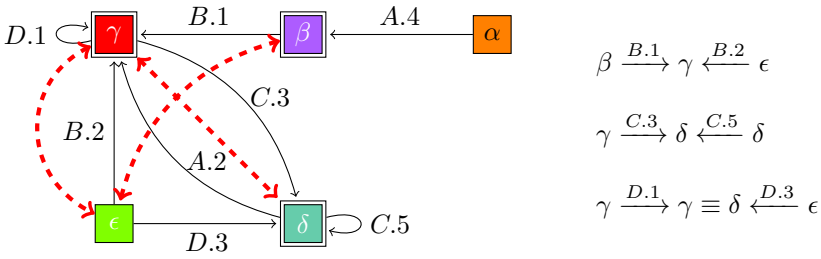
The sequence Γ_i is increasing (with respect to set inclusion) and upper-bounded by $\mathcal{W} \times \mathcal{W}$, therefore it converges (in a finite number of steps) to a limit denoted as Γ_∞. We write $\alpha \equiv_i \beta$ (resp. $\alpha \overset{*}{\equiv} \beta$) as syntactic sugar for $(\alpha,\beta) \in \Gamma_i$ (resp. Γ_∞).

▶ **Lemma 1.** *$\overset{*}{\equiv}$ is the smallest equivalence relation such that $\alpha \xrightarrow{X.i} \beta \overset{*}{\equiv} \gamma \xleftarrow{X.j} \delta \implies \alpha \overset{*}{\equiv} \delta$.*

Proof. By intimidation: if this is not obvious to the reader, then they better stop reading *RIGHT NOW*! (otherwise, the argument is that the Γ_i are constructed to this purpose; $\overset{*}{\equiv}$ is the least fixed-point of the increasing function on pairs of wires that guarantees the wanted properties). ◀

The idea is that there is a sequence of instructions that takes the player from α to δ while leaving the stack unchanged whenever $\alpha \overset{*}{\equiv} \delta$. We then say that there is a *transitive wormhole* between α and δ.

Most games that require the player to explore a large map have some built-in mechanism to "fast-travel" between some previously visited locations. Recursive mazes lack this mechanism, and that is really unfortunate. For this reason, we suggest to extend the game with the ability to fast-travel through wormholes (and we bet that this could relieve players of a considerable amount of frustration). We extend the usual semantic of the dual representation graph with these wormholes, using dashed edges:



Note that the wormhole between γ and ϵ is transitive ($\gamma \not\equiv \epsilon$ but $\gamma \overset{*}{\equiv} \epsilon$). From this, we observe that $\alpha \xrightarrow{A.4} \beta \overset{*}{\equiv} \gamma \xleftarrow{A.2} \delta$, so that $\alpha \overset{*}{\equiv} \delta$ and the maze can be exited from the starting wire.

The algorithm described in [1, 3] essentially consists in explicitly computing $\overset{*}{\equiv}$ by iterating the recursive construction starting from $\equiv$ until a least fixed point has been reached. This algorithm has complexity cubic in the number of states of the automaton. In our context, this leads to an algorithm with complexity $\mathcal{O}\left(W^3\right)$, which is already better than constructing the associated context-free grammar.

This algorithm adds all the possible wormhole edges to the dual representation graph (at most W^2 of them). Each edge can be labelled with a unique identifier. Local wormhole edges can additionally be labelled with a sequence of *two* instructions. Transitive wormhole edges can be labelled with a tuple of (instruction, wormhole identifier, instruction). All wormhole edges can be labelled with a length. Therefore, not only it is possible to find a shortest path from the starting point to an exit (using only wormhole edges), but it is possible to emit a description of this path in polynomial time, even though the path may have exponential length, by printing the labels of the crossed wormhole edges.

6 Getting out of recursive mazes even faster by collapsing wormholes

The main contribution of this paper is that, in the case of recursive mazes, it is possible to do better than [1, 3]. Our starting point is that the reversible nature of all movements makes it possible to exploit equivalence classes of wires. If $\alpha \overset{*}{\equiv} \beta$, then we are going to consider that α and β are "the same wire", and as far as reachability is concerned no information is lost by doing so.

This is different from the general problem tackled by [1, 3] of deciding reachability of a given state in an arbitrary pushdown automaton. The main difference is that in a general PDA there may be a sequence of transitions from a state q to another state q' that leaves the stack unchanged, while there might not be a sequence of transitions from q' to q that leaves the stack unchanged. In recursive mazes, we now that the latter exists.

On this occasion, we cannot resist the urge of a little digression. We see a striking analogy with abstract quantum computational models: applying a unitary transformation to the state of a quantum system is reversible, *just like moves in a recursive maze*. This sheds new light to the intervention of Peter Shor mentioned in section 3 – we are not so naive as to believe that it is a mere coincidence. Therefore, we plan to push forward the idea that recursive mazes are ideally suited as a computational model for future quantum computers (although we are going to leave that aspect aside for future work).

Back to the problem at hand, this makes it possible for us, instead of explicitly building a clique between equivalent wires as in [1, 3], to build a (much smaller) *spanning tree*: this preserves reachability properties. The same technique is used in the context of sparse matrix factorization [4] to construct the elimination tree of the matrix A^tA, given a sparse matrix A and without explicitly forming A^tA that might be much denser. Each row of A with k non-zero entries creates a clique of size k in the graph whose adjacency matrix is A^tA. This clique is replaced by a path that preserves reachability.

Our proposed solution is also inspired by the "edge-contracting" probabilistic algorithm of Karger for min-cut [7]. The core idea of our proposal is simple: when a local wormhole $\alpha \xrightarrow{X.i} \beta \xleftarrow{X.j} \gamma$ is identified, we add an edge $\alpha \leftrightarrow \gamma$ to the dual representation graph, and we immediately contract it. This collapses the two nodes that are the extremities of the wormhole into one and deletes the newly added edge. The reader may wonder what was the point of adding add edge just before deleting it. So, OK, let's not add it, and instead directly "merge" the two equivalent wires. This operation creates a *supernode* labelled with a set of wires ($\{\alpha, \gamma\}$ if starting from α and γ), as opposed to a normal node labelled with a single wire. All edges originating from (resp. pointing towards) the original nodes now originate in (resp. point to) the new supernode. The two edges forming the local wormhole now have the same extremities and the same box attribute, so we remove one of them. See Fig. 7 for an illustration. The underlying idea is that all wires inside of a supernode are known to be in relation by $\overset{*}{\equiv}$. Our algorithm consists in iterating this graph rewriting process as long as possible.

After a draft of this paper was written, we discovered that the same operation has been called "folding" in [9, 10, 11], in quite a different context – profinite topologies on a free group, if you can believe it. The main properties of this process have already been proved in these works, and because of space constraints we will not repeat them: wormholes can be contracted in any order, and at the end we get exactly the equivalence classes of $\overset{*}{\equiv}$. The intuition behind this is that contracting a wormhole may create new wormholes but not destroy preexisting ones (other than the one being contracted, of course). This is because a wormhole requires two similarly-labelled edges pointing to the same node ; contracting a wormhole can only bring together the endpoints of edges previously pointing to different nodes, not take them apart.

Starting from the dual representation graph of a maze, we run the contraction process until death ensues. To check if it is possible to exit the maze, it is sufficient to examine the wires in the equivalence class of the starting point, and test if one of them is connected to an outer IO pin. When it is the case, an exit path can be built explicitly (but how to do so depends on implementation details discussed below).

7 Efficient deterministic implementations

We now discuss the efficient implementation of the graph rewriting process discussed above. We assume that everything is represented as word-sized integers: wires are numbered from 0 to $W-1$, boxes are numbered from 0 to $B-1$, etc. The input is the dual representation

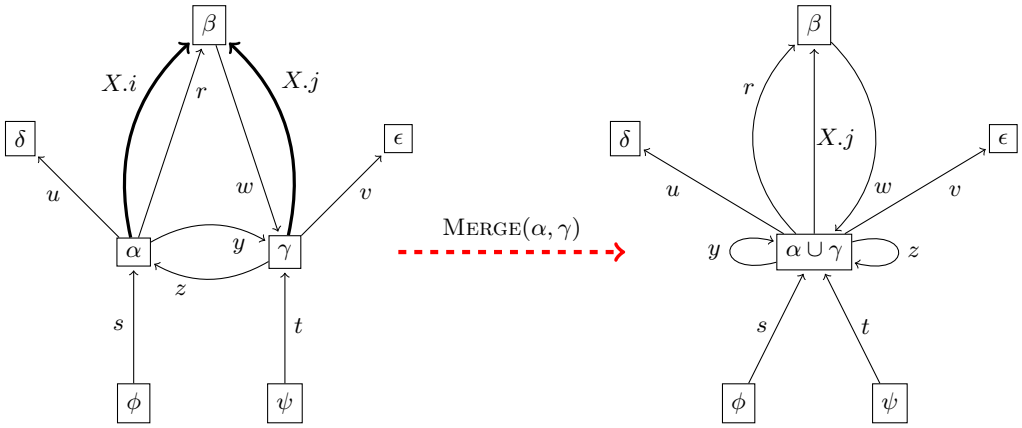


Figure 7 Illustration of the graph contraction process. Edge labels in lower-case letters are arbitrary.

graph, provided as adjacency lists, i.e. as an array `G` such that `G`$[\omega]$ is the (linked) list of edges coming out of node ω. The edge $\alpha \xrightarrow{X.i} \beta$ is represented by the triplet (X, i, β), where the endpoint is explicit and the starting point is implicit. We therefore define the "type" `edge` ::= `box` × `pin` × `wire` (these are actually 3 integers, but this spells out their purpose more explicitly). The graph is therefore provided as an `edge list array`, where the edges coming out of wire ω are given in `G`$[\omega]$.

Tuples, also known as records or `struct`, can be (de)constructed in constant time. We make abundant use of linked lists, and we employ the following notations:

- `[]` denotes the empty list (a `NULL` pointer).
- $L \leftarrow$ `head` :: `tail` denotes the operation that constructs a new linked L list whose first element is `head`, followed by the linked list `tail` (this means that a new cell must be acquired, `head` must be copied into it, and a pointer to `tail` must also be stored).
- `head` :: `tail` $\leftarrow L$ denotes the "deconstruction" operation: it fetches the data item `head` contained in the first entry of the list, and grabs a pointer `tail` to the remaining items.

(De)constructing a list runs in constant time, and so does accessing any location in an array. The observant reader may have noticed that our notations are inspired from functional languages from the ML family.

We keep a list (`pending`) of wormholes awaiting contraction, and process them one at a time, adding newly discovered wormholes to the list. We must however navigate around two problems:

1. The same pair of wires can be merged by two (or more) wormholes (as shown in Fig. 8, left). In our implementation, the pending list may contain several wormholes merging the same pair of nodes.
2. The number of wormholes can be quadratic in the size of the maze, even if it has only $E = \mathcal{O}(W)$ edges (see Fig. 8, right). However, not all of them can be collapsed: some will vanish into thin air when *others* are collapsed.

When processing the next wormhole, we check if the two wires have already been merged and only proceed when it is not the case (this solves the first problem). In addition, when a wormhole $\alpha \xrightarrow{X.i} \beta \xleftarrow{X.j} \gamma$ is discovered and added to the list, we immediately remove one of the two edges from the graph – they would become identical later when the wormhole actually gets contracted and one of them would be removed anyway, so why delay the inevitable?

Figure 8 Two wires can be merged by several wormholes (left). There can be a quadratic number of wormholes (right).

With this modification, only E wormholes can be added to the list. We implement this as follows: we look at each edge of the initial graph in sequence: either it creates a wormhole with another "hot" edge (we register the wormhole and the edge under scrutiny is sucked into the void), or it does not (we mark the edge as "hot").

Inside each supernode, one node is chosen as "supernode leader". The supernode is hosted by the leader node in the original graph. Nodes that have been merged into larger supernode are not actually removed from the graph ; they remain there but become *followers* (as opposed to their leader).

When two (super)nodes are merged, one of them is in fact "absorbed" into the other one. Hot edges pointing to the absorbee either create a wormhole and are dropped, or become hot edges pointing to the absorber. The actual issue is to detect the new wormholes that are created by the displacement of edges.

A major difference with the process suggested by Fig. 7 is that we only move the targets of edges, not their origins. This does not affect the process, at least not until an actual exit path has to be constructed. This issue is discussed in section 8.

7.1 Dense mazes

We first describe a simple implementation tailored for the case of *dense* mazes, where $E \approx WB$. We use a `hot` array of dimension $W \times B$ such that `hot`$[\beta, X]$ is either `none` or `some`(α, i), in which case there is a "hot" edge $\alpha \xrightarrow{X.i} \beta$ in the graph.

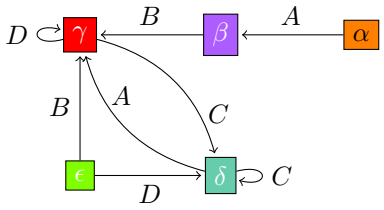
When the algorithm starts, each node in the graph corresponds to a wire of the maze. Then nodes will be merged into supernodes. Maintaining the content of supernodes turns out to be once-in-a-lifetime chance for us to use the celebrated "inverse-Ackermann" disjoint-set data structure (also known as Union-Find). This uses the two arrays `parent` and `rank`.

The pseudo-code of our solution in shown as Algorithm 1. We believe that it is simple enough to be implemented in assembler by a motivated undergrad. Its step-by-step execution on the simple maze of Fig. 6 is shown in Fig. 9. When it terminates, equivalence classes of $\overset{*}{\equiv}$ have been determined. This easily allows to test if the maze can be exited.

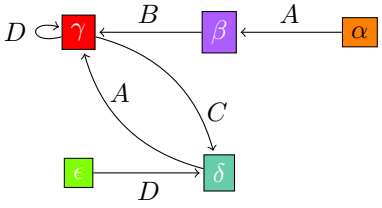
It is clear that AddEdge runs in constant time, and that Merge runs in $\mathcal{O}(B)$ operations. The setup phase of CollapseMaze requires $\mathcal{O}(WB + E)$ operations. The number of iterations of the **while** loop is at most E (each new wormhole added to `pending` destroys an edge). Merge can only be invoked $W-1$ times. So the total time spent in Merge is $\mathcal{O}(BW)$ and the total time spent in calls to Find is $\mathcal{O}(E\alpha(W))$, where α is the inverse-Ackermann function (thanks to the celebrated amortized analysis of [12]). So, the final complexity of CollapseMazeDense is $\mathcal{O}(BW + E\alpha(W))$. If $E \approx W^2$ and $W \approx B$, then this gives a nearly-quadratic algorithm, improving upon the known cubic one presented in section 5.

Algorithm 1 Algorithm to collapse dense recursive mazes.

```
1:  function FIND(α)
2:      if parent[α] ≠ α then                                   ▷ path compression
3:          parent[α] ← FIND(parent[α])
4:      return parent[α]

5:  procedure ADDEDGE(α, X, i, β)
6:      if hot[β, X] = some(γ, j) then
7:          pending ← (α, X, i, β, j, γ)) :: pending            ▷ detected α -X.i-> β <-X.j- γ
8:      else
9:          hot[β, X] ← some(α, i)

10: procedure MERGE(σ, π)
11:     if rank[π] < rank[σ] then                               ▷ UNION
12:         tmp ← π, π ← σ, σ ← tmp                             ▷ Union-by-rank: σ is absorbed in π
13:     parent[σ] = π
14:     if rank[σ] = rank[π] then
15:         increment rank[π]
16:     for 0 ≤ X < B do                                        ▷ Transfer hot edges
17:         if hot[σ, X] ≠ none then
18:             some(ω, i) ← hot[σ, X]                          ▷ looking at ω -X.i-> σ
19:             ADDEDGE(ω, X, i, π)                             ▷ replace by ω -X.i-> π

20: procedure COLLAPSEMAZEDENSE(W, B, G)
21:     pending ← []                                            ▷ Setup
22:     for 0 ≤ α < W do
23:         parent[α] ← α                                       ▷ α is a supernode by itself
24:         rank[α] ← 0
25:         for 0 ≤ X < B do
26:             hot[α, X] ← none
27:         for each (X, i, β) in G[α] do                       ▷ prime the wormhole pump
28:             ADDEDGE(α, X, i, β)
29:     while pending ≠ [] do                                   ▷ main loop : process known wormholes
30:         (α, X, i, β, j, γ) :: tail ← pending                ▷ popped α -X.i-> β <-X.j- γ
31:         pending ← tail
32:         σ ← FIND(α)
33:         π ← FIND(γ)
34:         if σ ≠ π then                                       ▷ already merged?
35:             MERGE(σ, π)                                     ▷ merge nodes containing α and γ
```



original graph

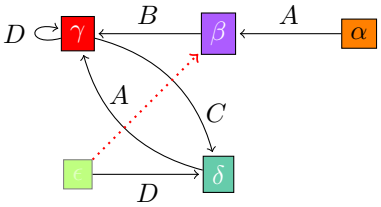


pending:

$\gamma \xrightarrow{C} \delta \xleftarrow{C} \delta$

$\beta \xrightarrow{B} \gamma \xleftarrow{B} \epsilon$

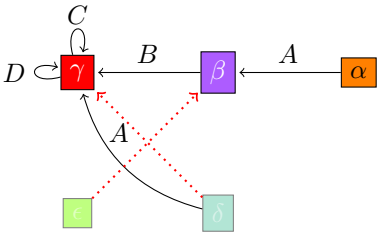
before the main loop



pending:

$\gamma \xrightarrow{C} \delta \xleftarrow{C} \delta$

$\beta \xrightarrow{B} \gamma \xleftarrow{B} \epsilon$

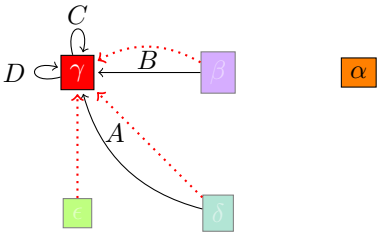


pending:

$\gamma \xrightarrow{C} \delta \xleftarrow{C} \delta$

$\beta \xrightarrow{B} \gamma \xleftarrow{B} \epsilon$

$\epsilon \xrightarrow{D} \gamma \xleftarrow{D} \gamma$



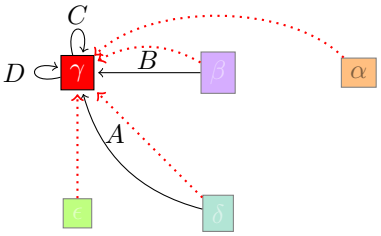
pending:

$\beta \xrightarrow{B} \gamma \xleftarrow{B} \epsilon$

$\gamma \xrightarrow{C} \delta \xleftarrow{C} \delta$

$\epsilon \xrightarrow{D} \gamma \xleftarrow{D} \gamma$

$\alpha \xrightarrow{A} \gamma \xleftarrow{A} \delta$



pending:

$\beta \xrightarrow{B} \gamma \xleftarrow{B} \epsilon$

$\gamma \xrightarrow{C} \delta \xleftarrow{C} \delta$

$\epsilon \xrightarrow{D} \gamma \xleftarrow{D} \gamma$

$\alpha \xrightarrow{A} \gamma \xleftarrow{A} \delta$

Figure 9 Step-by-step illustration of Algorithm 1. The plain edges are "hot". The dotted edges represent the supernode leaders (for non-trivial supernodes).

7.2 Sparse mazes

When E is much less than BW, several aspects of the algorithm could be made more efficient. For instance, MERGE requires B operations to inspect B potential edges, even if just a few are present. We can use a lightweight variant of the "sparse array" technique abundantly used in very early sparse matrix codes and rediscovered 20 years later [2] in a different context. To efficiently iterate on the hot incoming edges to a node, we also use an auxiliary array `hotpot` where `hotpot`$[\omega]$ contains a list of X's such that `hot`$[\omega, X] \neq$ `none`.

Also, when merging two nodes, we could absorb the "smaller" one (with less edges to process) into the "larger" one. Doing all the work on the smaller of the two nodes minimizes the total amount of work (this idea is common ; for instance it is at work in Hopcroft's DFA minimization algorithm [5]). Unfortunately, this breaks down the guarantees offered by the disjoint set data structure. We therefore make do with a more naive solution: we eagerly update all nodes when two sets are joined. All nodes have a pointer (`leader`) to the leader node in their supernode (possibly themselves). Supernode leaders have a list (`followers`) of nodes in their supernode. The *size* of a (super)node is the number of nodes it contains plus the number of incoming edges. The algorithm maintains the sizes at all time (in an array `size`), as new wormholes are discovered and (super)nodes are merged. When two (super)nodes are merged, to form the resulting supernode, the smaller one (by `size`) is "absorbed" into the larger one. More precisely:

- The `leader` pointer of all nodes in the smaller (super)node is set to the larger supernode.
- Hot edges pointing to the smaller supernode either create a wormhole and are dropped, or become hot edges pointing to the larger supernode.
- The size of the larger supernode is augmented accordingly.

ADDEDGE' still runs in constant time, and MERGE runs in time dominated by the `size` of the smaller supernode: it takes constant-time per wire in the smaller supernode, plus constant time per hot edge pointing to the smaller supernode (thanks to the `hotpot` array) – and not all edges are hot, hence the upper-bound.

The number of iterations of the **while** loop is again at most E and it does a constant number of operations per iteration, excluding calls to MERGE. It thus remains to control the total time spent in MERGE – and this is the tricky part. As the algorithm progresses, the sequence of merges can be represented by a binary forest. The wires of the maze are the leaves, and each call to MERGE creates a new root whose two children are preexisting roots. It represents a new supernode containing all the wires associated to the leaves in the subtree. Each supernode is labelled by its size (as defined above). When two roots of labels $x \leq y$ are joined, the result has label $x + y$, and we are "charged" x operations (the smaller size).

Let $\mathcal{F}$ denote an arbitrary merge forest built from the dual representation graph of a recursive maze. Its W leaves are labelled with the size of the wires (one plus the number of incoming edges). Let S denote the sum of sizes: if the dual representation graph has E edges, then $S \leq W + E$. Let $C(\mathcal{F})$ denote the total amount "charged" by this merge forest (i.e., an upper-bound on the total number of operations spent in MERGE during the process). Then we have:

▶ **Lemma 2.** $C(\mathcal{F}) \leq S \log_2 S$.

The (very standard) proof is given in the full version. Therefore, the total running time of COLLAPSEMAZESPARSE is $\mathcal{O}(BW + E \log E)$. At first glance, this could seem like a *decremental* improvement compared to COLLAPSEMAZEDENSE. However, the *only* part that requires BW operations is the initialization of the `hot` array. We therefore see two ways to cheat our way out of this embarrassing situation.

Algorithm 2 Algorithm to collapse sparse recursive mazes.

```
1: procedure AddEdge'(α, X, i, β)
2:     if hot[β, X] = some(γ, j) then
3:         pending ← (α, X, i, β, j, γ)) :: pending          ▷ detected α —X.i→ β ←X.j— γ
4:     else
5:         hot[β, X] ← some(α, i)
6:         hotpot[β] ← X :: hotpot[β]                        ▷ Register non-empty location

7: procedure Merge'(σ, π)
8:     if size[σ] > size[π] then
9:         tmp ← σ, σ ← π, π ← tmp                           ▷ union-by-size: now size[σ] ≤ size[π]
10:    size[π] ← size[π] + size[σ]
11:    for each ω ∈ followers[σ] do
12:        leader[ω] ← π
13:        followers[π] ← ω :: followers[π]
14:    for each X ∈ hotpot[σ] do                             ▷ sparse iteration
15:        some(ω, i) ← hot[σ, X]                            ▷ looking at ω —X.i→ σ
16:        AddEdge'(ω, X, i, π)                              ▷ replace by ω —X.i→ π

17: procedure CollapseMazeSparse(W, B, G)
18:    pending ← []                                          ▷ Setup
19:    for 0 ≤ α < W do
20:        size[α] ← 1
21:        leader[α] ← α                                     ▷ α is a supernode by itself
22:        followers[α] ← α :: []
23:        hotpot[α] ← []
24:        for 0 ≤ X < B do
25:            hot[α, X] ← none                              ▷ potential hot-spot
26:        for each (X, i, β) in G[α] do                     ▷ prime the wormhole pump
27:            AddEdge'(α, X, i, β)
28:            increment size[β]
29:    while pending ≠ [] do                                 ▷ main loop : process known wormholes
30:        (α, X, i, β, j, γ) :: tail ← pending              ▷ popped α —X.i→ β ←X.j— γ
31:        pending ← tail
32:        if leader[α] ≠ leader[γ] then                     ▷ already merged?
33:            Merge'(leader[α], leader[γ])                  ▷ merge nodes containing α and γ
```

We initially need the `hot` array to be filled with the `none` value. If the array was *provided* to us in this state, then everything would be perfect. CollapseMazeSparse would run in time $\mathcal{O}(E \log E)$, mission accomplished. And although this is *a bit* cheating, it is not too bad because we could easily *restore* the `hot` array to its initial state with only a constant factor increase in the running time (just store the indices of all modified locations and restore them to `none` at the end of the algorithm). It follows that if a long sequence of mazes had to be solved in sequence, then a single (large enough) `hot` array could be initialized *once* and used several times, so the cost of its initialization could be amortized over an (arbitrarily long) sequence of uses.

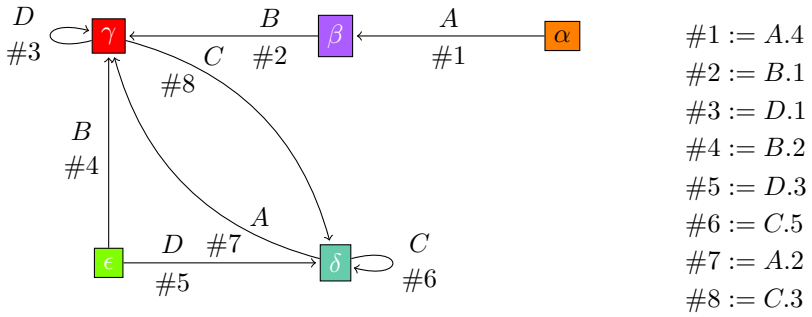


Figure 10 Extended dual representation graph corresponding to the simple maze of Fig. 6.

- A good-natured reviewer suggested that "*the requirement that the array is initialized with* `none` *is not unreasonable at all. All you really need is the model of computation to provide* any *uniform representation of the initial memory and then just call that your* `none` *value*". We appreciate the empathy. But unlike this sickeningly well-meaning stance, we believe the reality is quite different: no existing hardware architecture that can run compiled C programs offers "free" memory initialization.
- Initializing the `hot` array annoys us. Wo we could simply get rid of it and use a hash table instead. It will ever only hold E entries, and we have a time budget to initialize an array of size $E \log E$, so that we can keep the fill ratio of this hash table comfortably low. However, this requires some kind of randomization and we leave it as an open problem.

8 Construction of an exit path

All wires in an equivalence class of $\overset{*}{\equiv}$ are mutually reachable (without modifying the stack). In this section we show how to augment Algorithms 1 and 2 to provide the computation of such paths without asymptotically increasing their complexity.

In the dual representation graph, an edge $\alpha \xrightarrow{X.i} \beta$ is labelled with an instruction ("$X.i$"). This describes a feasible move in the maze (and a valid transition in the pushdown automaton that pushes X).

We extend this as follows: each edge is labelled with a *box attribute* (X – what recursive box is entered by taking this edge) and the *identifier of an instruction sequence* (that describes what must be done to actually cross the edge). We write these identifier with a "sharp" (e.g. #42). There is also a global *context* that maps instruction sequence identifiers to actual sequences of instructions and instruction sequence identifiers. This time, an edge $\alpha \xrightarrow[\#42]{X} \beta$ means that the sequence of instruction #42 takes the player from wire α to wire β, while entering box X. So, before the graph rewriting process even starts, we transform the dual representation graph into the *extended dual representation graph* shown in Fig. 10.

We then perform the same graph rewriting process, except that we now take care of instruction sequence identifiers as shown in Fig. 11. All the edges incident to the supernode that gets "absorbed" gets a new instruction sequence identifier. In the figure, the sequence of instruction $\#\mathcal{K} := \mathcal{I}; \overline{\mathcal{J}}$ travels from α to γ. Edges incoming into α are now incoming into γ, therefore $\#\mathcal{K}$ must be added after their instruction sequence. Edges pointing out of α now point out of γ, so $\overline{\#\mathcal{K}}$ must be added before their instruction sequence. We finally record the "spanning" edge $\alpha \xrightarrow{\#\mathcal{K}} \gamma$.

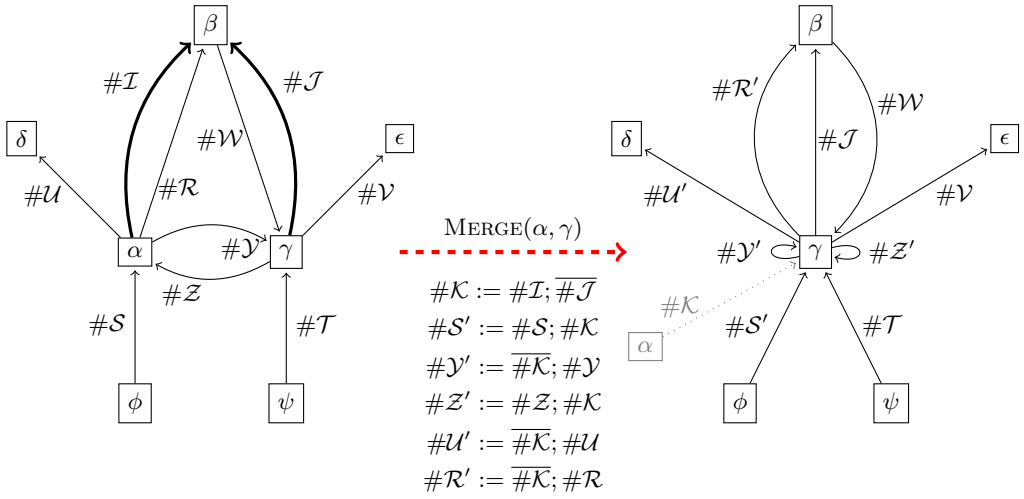


Figure 11 Illustration of the extended graph contraction process (showing only instruction sequence identifiers). The two thick edges have the same box attribute. It modifies the labels of all edges incident to α and registers one new instruction sequence identifier for each one.

When the rewriting process is over, the spanning edge form (rooted) spanning trees of the connected components. Therefore, when a pair of nodes are reachable by a path, this path can be assembled by following the spanning edges. Compared to Algorithms 1 and 2, this requires the following modifications:

- Not only the target, but also the origin of hot edges incident to an "absorbed" node must be modified. This requires each node to we aware not only of its incoming hot edges, but also of the outgoing ones.
- The instruction sequences on each edge must be maintained.
- When a wormhole $\alpha \xrightarrow[\#\mathcal{I}]{X} \beta \xleftarrow[\#\mathcal{J}]{X} \gamma$ is discovered, one of the two edges is removed from the graph, but its instruction sequence identifier must not be lost (otherwise we will not be able to form the spanning edge). Therefore, $(\alpha, X, \#\mathcal{I}, \beta, \#\mathcal{J}, \gamma)$ must be pushed onto the `pending` list.
- Spanning edges must be stored separately.

Fig. 13 illustrates the process. In the end, this produces the path #18, that expands as:

$$\#18 = \#16; \overline{\#14} = (A.4; \#15); (\overline{A.2}; \#11) = A.4; B.1; \overline{B.2}; D.3; C.5; \overline{C.3}; \overline{D.1}; \overline{A.2}; C.5; \overline{C.3}$$

This is indeed an exit path, but not one of optimal length (compare to the one given earlier in the text). Note that exponentially-long paths can be represented in polynomial space by the list of instruction sequence identifiers. We leave the problem of producing a minimal-length exit path open for future work.

9 Conclusion: a daunting open problem

Checking emptiness of the intersection of *two* context-free languages is undecidable (this enables the simulation of an arbitrary Turing machine). Therefore, it could very well be that trying to get out of *two* recursive mazes simultaneously might be difficult. This problem

can be thought of as follows: suppose you have an application to play recursive mazes[2]. In such a game, the mazes are orthogonal: paths are either horizontal or vertical. The player controls the location of a sprite using arrow keys, as is common in "top down" adventure games, such as *The Legend of Zelda*. Moving towards a wall does nothing: the sprite cannot cross the wall.

Now suppose that *two* instances of the application are launched, with two different mazes, and that *both* applications receive the same keyboard inputs (see Fig. 12). In the situation shown in the figure, going down does nothing; going up moves the sprite in the right maze; going left moves the sprite in the left maze; going right moves the sprite in both mazes. Given the geometric description of two mazes, the question is to decide if there is a (finite) sequence of keyboard inputs that exits both mazes.

It is not completely obvious that this problem is even decidable (we may get away with it because of the severe restrictions on the class of context-free languages associated to recursive mazes). In any case, this is an exciting perspective for future work.

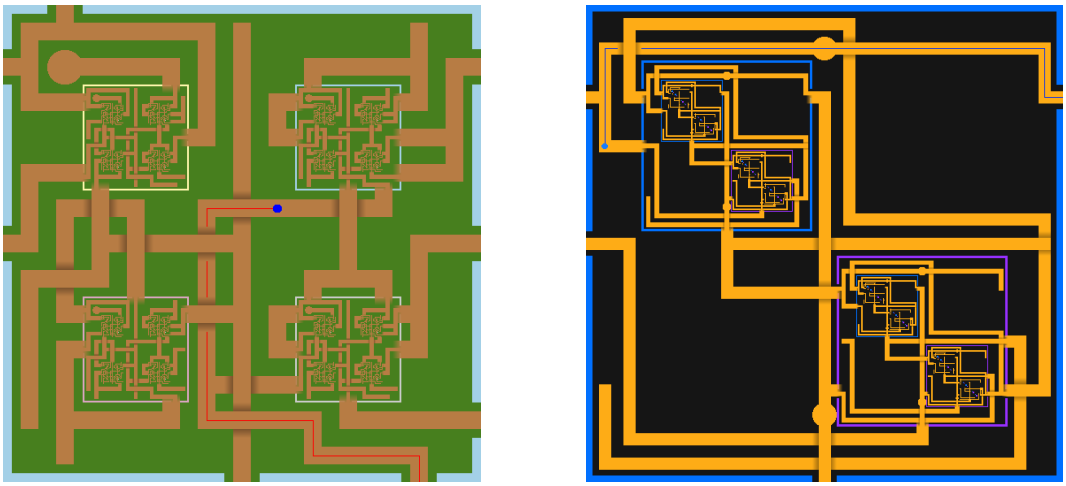
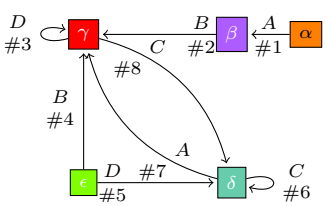
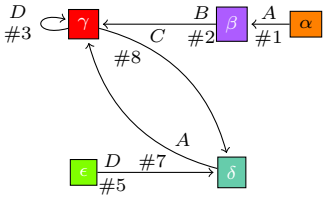


Figure 12 Solving two mazes at the same time – with a single keyboard. Is it decidable? The player is represented by a blue dot, and its current path is shown by a blue or red line.

[2] Such as the one we designed, available at `https://github.com/orelcosseron/Fractal-Maze-Python`.



$\#1 := A.4$
$\#2 := B.1$
$\#3 := D.1$
$\#4 := B.2$
$\#5 := D.3$
$\#6 := C.5$
$\#7 := A.2$
$\#8 := C.3$



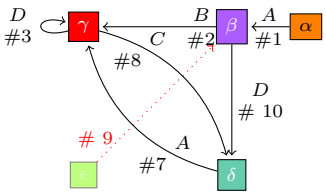
$\#1 := A.4$
$\#2 := B.1$
$\#3 := D.1$
$\#4 := B.2$
$\#5 := D.3$
$\#6 := C.5$
$\#7 := A.2$
$\#8 := C.3$

pending:

$\gamma \xrightarrow[\#8]{C} \delta \xleftarrow[\#6]{C} \delta$

$\beta \xrightarrow[\#2]{B} \gamma \xleftarrow[\#4]{B} \epsilon$

before the main loop



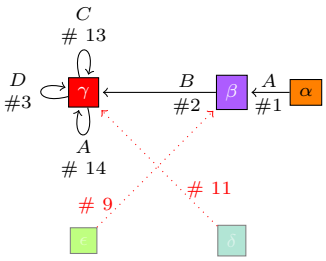
$\#1 := A.4$
$\#2 := B.1$
$\#3 := D.1$
$\#4 := B.2$
$\#5 := D.3$
$\#6 := C.5$
$\#7 := A.2$
$\#8 := C.3$

$\#9 := \#4; \overline{\#2}$
$\#10 := \overline{\#9}; \#5$

pending:

$\gamma \xrightarrow[\#8]{C} \delta \xleftarrow[\#6]{C} \delta$

$\beta \xrightarrow[\#2]{B} \gamma \xleftarrow[\#4]{B} \epsilon$

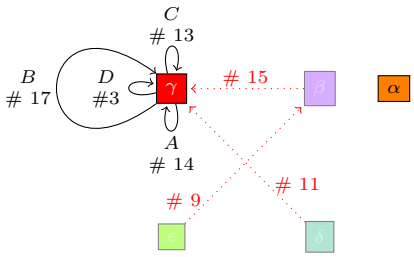


$\#1 := A.4$
$\#2 := B.1$
$\#3 := D.1$
$\#4 := B.2$
$\#5 := D.3$
$\#6 := C.5$
$\#7 := A.2$
$\#8 := C.3$

$\#9 := \#4; \overline{\#2}$
$\#10 := \overline{\#9}; \#5$
$\#11 := \#6; \overline{\#8}$
$\#12 := \#10; \#11$
$\#13 := \#8; \#11$
$\#14 := \overline{\#11}; \#7$

pending:

$\beta \xrightarrow[\#12]{D} \gamma \xleftarrow[\#3]{D} \gamma$

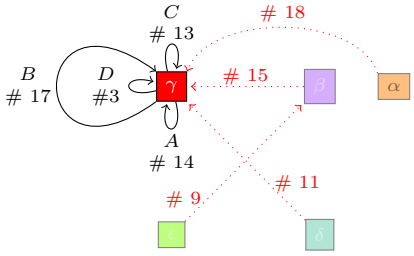


$\#1 := A.4$
$\#2 := B.1$
$\#3 := D.1$
$\#4 := B.2$
$\#5 := D.3$
$\#6 := C.5$
$\#7 := A.2$
$\#8 := C.3$

$\#9 := \#4; \overline{\#2}$
$\#10 := \overline{\#9}; \#5$
$\#11 := \#6; \overline{\#8}$
$\#12 := \#10; \#11$
$\#13 := \#8; \#11$
$\#14 := \overline{\#11}; \#7$
$\#15 := \#12; \overline{\#3}$
$\#16 := \#1; \#15$
$\#17 := \overline{\#15}; 2$

pending:

$\alpha \xrightarrow[\#16]{A} \gamma \xleftarrow[\#14]{A} \gamma$



$\#1 := A.4$
$\#2 := B.1$
$\#3 := D.1$
$\#4 := B.2$
$\#5 := D.3$
$\#6 := C.5$
$\#7 := A.2$
$\#8 := C.3$

$\#9 := \#4; \overline{\#2}$
$\#10 := \overline{\#9}; \#5$
$\#11 := \#6; \overline{\#8}$
$\#12 := \#10; \#11$
$\#13 := \#8; \#11$
$\#14 := \overline{\#11}; \#7$
$\#15 := \#12; \overline{\#3}$
$\#16 := \#1; \#15$
$\#17 := \overline{\#15}; 2$
$\#18 := \#16; \overline{\#14}$

pending:

Figure 13 Step-by-step illustration of the construction of an exit path. The plain edges are "hot". The dotted edges represent are the extra spanning edges).

References

1 Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. doi:10.1007/3-540-63141-0_10.

2 Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Lett. Program. Lang. Syst.*, 2(1–4):59–69, March 1993. doi:10.1145/176454.176484.

3 Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. In Faron Moller, editor, *Second International Workshop on Verification of Infinite State Systems, Infinity 1997, Bologna, Italy, July 11-12, 1997*, volume 9 of *Electronic Notes in Theoretical Computer Science*, pages 27–37. Elsevier, 1997. doi:10.1016/S1571-0661(05)80426-8.

4 John R Gilbert, Xiaoye S Li, Esmond G Ng, and Barry W Peyton. Computing row and column counts for sparse QR and LU factorization. *BIT Numerical Mathematics*, 41(4), December 2000. doi:10.1023/A:1021943902025.

5 John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, NY, 1971. doi:10.1016/B978-0-12-417750-5.50022-1.

6 Peter Gjøl Jensen, Stefan Schmid, Morten Konggaard Schou, and Jirí Srba. PDAAAL: A library for reachability analysis of weighted pushdown systems. In Ahmed Bouajjani, Lukás Holík, and Zhilin Wu, editors, *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings*, volume 13505 of *Lecture Notes in Computer Science*, pages 225–230. Springer, 2022. doi:10.1007/978-3-031-19992-9_14.

7 David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 21–30. ACM/SIAM, 1993. URL: http://dl.acm.org/citation.cfm?id=313559.313605.

8 Stefan Schwoon. *Model checking pushdown systems*. PhD thesis, Technical University Munich, Germany, 2002. URL: http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/schwoon.html.

9 John R. Stallings. Topology of finite graphs. *Inventiones mathematicae*, 71(3):551–565, 1983. doi:10.1007/BF02095993.

10 Benjamin Steinberg. Finite state automata: A geometric approach. *Transactions of the American Mathematical Society*, 353(9):3409–3464, 2001. URL: http://www.jstor.org/stable/2693797.

11 Benjamin Steinberg. Inverse automata and profinite topologies on a free group. *Journal of Pure and Applied Algebra*, 167(2):341–359, 2002. doi:6810.1016/S0022-4049(01)00030-5.

12 Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984. doi:10.1145/62.2160.

13 Mark J. P. Wolf. *101 Enigmatic Puzzles — Fractal Mazes, Quantum Chess, Anagram Sudoku, and More*. BookBaby, 2020.