SuperCC

User Guide

Version 2.0

August 31, 2021

Table of Contents

Introduction

SuperCC, also known as SuCC, is an emulator for the MS and Lynx rulesets of the game *Chip's Challenge.* This user guide assumes knowledge of the game. If anything is unclear, please consult the Wiki.

While the program originated with optimization in mind, users have also found use for it in playtesting custom levels and even playing the game on a turn-by-turn basis. Scores achieved in SuperCC are not valid leaderboard scores.

This user guide was written in order to document the numerous features of the program as well as provide instructions on how to use them.

Credits

SuperCC Creator

Markus O.

SuperCC Maintenance, Improvement, and Contribution

A Sickly Silver Moon, Bacorn, quiznos00, Andrew Gapic

QA, Testing, Bug Reports, Technical Help

Andrew E., chipster1059, IHNN, Indyindeed, jamesa7171, Jay Bee, Naemuti, pieguy, quiznos00, random 8, Sharpeye, stubbscroll, The Architect, vortex178, and several others that weren't able to make this list for a variety of reasons

Special Thanks

Chuck Sommerville for creating Chip's Challenge.

Microsoft for their well known port.

The Chip's Challenge community for all their support and usage, and for all the people who've had the teams back throughout development, without them this program would not be nearly as large as it currently is.

IHNN and Sharpeye for their large contributions (Extreme amounts of QA + logic handling and helping to beta test Lynx respectively).

User Guide

Bacorn with proofreading by A Sickly Silver Moon

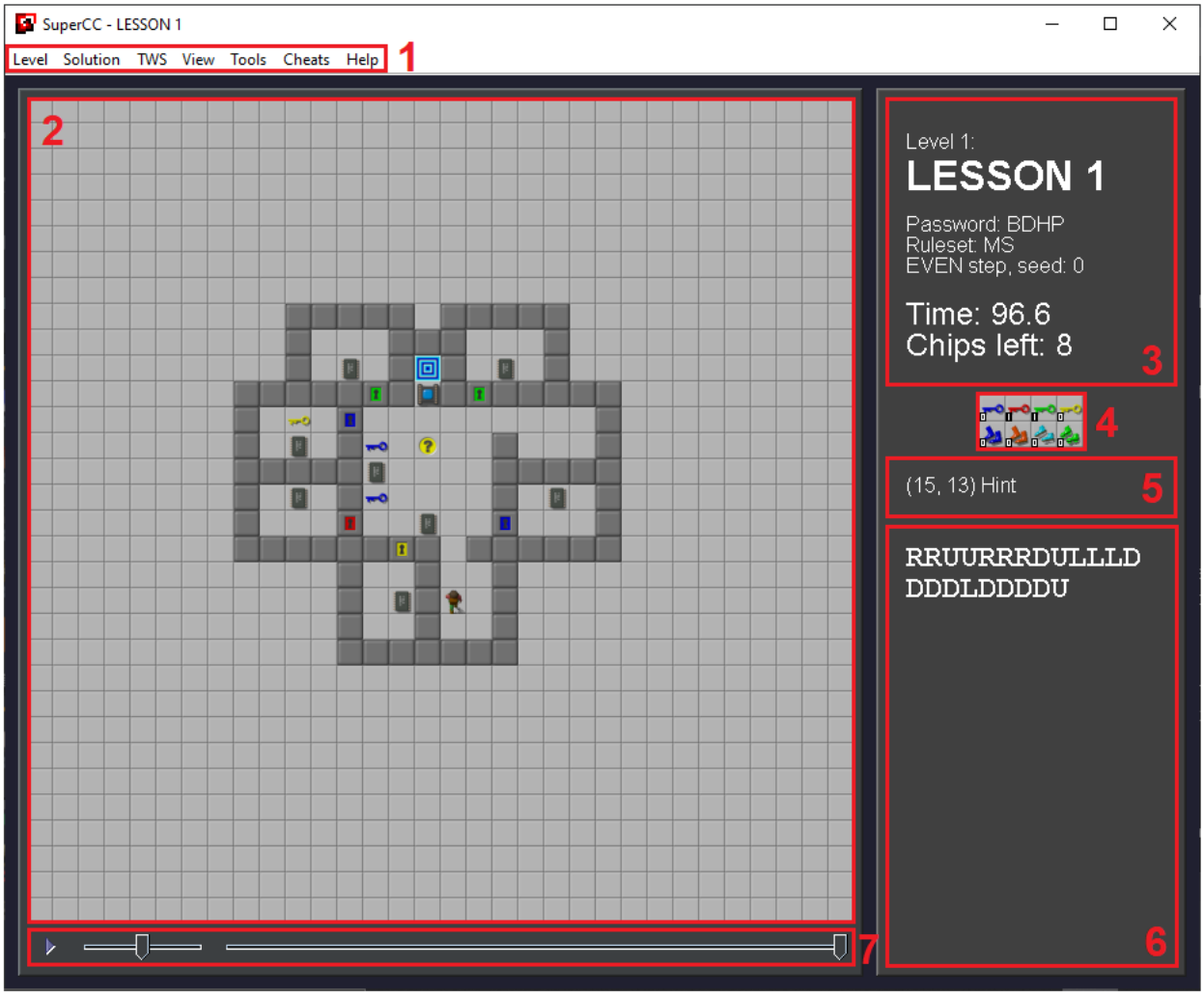
Chapter 1 – Getting Started

This chapter will cover how to get SuperCC up and running as well as how to play through the levels.

Download

In order to download the program, you should go to the download page and preferably choose the most recent version. The downloaded file will be a JAR file that requires Java to run. To get Java, go to this website and follow the instructions there. Make sure it's at least Java 16. To open the program, simply double click the file.

User Interface



SuperCC User Interface

1. **Menu**
   Most options and features are located here.
2. **Level Area**
   Shows the current state of the level and allows manipulation of specific tiles.
3. **Level Info Area**
   Shows the level number, title, password, ruleset, step, seed, time left, and chips left.
4. **Inventory**
   Shows how many of each item you currently have.
5. **Tile Information Area**
   Shows the coordinates and name of the tile the mouse is hovering over. It also displays the last completed action (like savestate operations).
6. **Move Display Area**
   A scrollable area that prints the entire sequence of moves.
7. **Playback Area**
   You can go move by move either manually or automatically with adjustable speed.

Opening a levelset

In order to open a levelset, from the menu choose **Level > Open** levelset or simply press the **Ctrl + Shift + O** shortcut. From the file selection, choose the levelset you want to load; by default it displays only DAT and CCL files. SuperCC will remember the last folder you loaded a levelset from and automatically open it the next time.

Navigating a levelset

Restart Level

In order to restart the current level to its initial state, from the menu choose **Level > Restart** or press the **Ctrl + R** shortcut. When you do this, the savestate rewinds to the beginning, so history is preserved.

Next/Previous Level

Going to the next level is done through the menu **Level > Next** or the shortcut **Ctrl + N**. Similarly, going to the previous level can be done through the menu **Level > Previous** or the shortcut **Ctrl + P**. When you go outside the range of the set this way, it wraps to the other end.

Go To

If you want to go to a specific level immediately, find the menu option **Level > Go to…** or use **Ctrl + G**. Then you type the number of the level and press the **OK** button or **Enter** on the keyboard.

Playing

After loading a level, you may begin playing it. By default, you can press the **Up**, **Left**, **Down**, and **Right** arrow keys to move in that direction, and press **Space** to wait half a move. You can also press **Backspace** to undo a move, and **Enter** to redo one. Once you make another move manually, all undone moves are erased. These controls are fully customizable by going to **Tools > Controls** in the menu.

You may also use mouse clicks to move the player. They only work in the normally visible viewport, that is in the typical 9×9 area around the player.

Playback

SuperCC Playback Interface

1. **Play/Pause Button**
   Allows starting and stopping playback.
2. **Speed Slider**
   Sets the speed of playback.
3. **Move Slider**
   Allows manual traversal of the move history.

Performing any move manually discards all future moves.

Chapter 2 – Configuration

SuperCC has many options to configure the user interface and controls to your liking. You can choose whether to show button connections or even more technical things like ordering in the slip list.

Controls

You can change control bindings by going to **Tools > Controls** in the menu. In order to change the control, click on the button next to the label and press the key you want to bind to that control. The settings are automatically saved. The movement tab contains all types of moves the player can make. The tools tab offers controls to travel through the move history.

A key bound to a control cannot be used to create savestates and macros.

View

The **View** menu offers many options for modifying the display. Below is an explanation of each option.

Tileset
: Manages which graphic spritesheet is used to display tiles in the level area. The setting is saved for each ruleset separately.

Tile Size
: Sets the size in pixels of tiles displayed. The custom option allows for arbitrary tile size between 1×1 and 256×256.

Game Window Size
: Sets the level area viewport size in tiles. 9x9 is the size normally visible in the game, while 32×32 is the full level size. You can also set your own viewport size between 1×1 and 32×32.

Show Monster List
: Toggles whether tiles display the monsters' position in the monster list.

Show Slip List
: Toggles whether tiles display the tiles' position in the slip list. This includes blocks and monsters.

Show Clone Connections
: Toggles whether the program always draws lines between red buttons and the clone machine they're connected to. Otherwise, it just shows the connection when hovering over the button.

Show Trap Connections
: Toggles whether the program always draws lines between brown buttons and the trap they're connected to. Otherwise, it just shows the connection when hovering over the button.

Show Move History

Toggles whether the program draws lines that show the path the player has taken through the level.

Switch Decimal Notation

Toggles the time left display between the .x and (-.x) notations.

Chapter 3 – Solutions, Savestates, and Macros

SuperCC has many options for managing and keeping track of solutions, savestates, and macros at any stage of level solving.

Solutions

A solution is a collection of data that recreates some state. It contains moves, seed, step, ruleset, initial slide, and encoding in a JSON format.

Saving a Solution

You can save a solution through **Solution > Save** or **Ctrl + S**. By default, the file will be saved in *<current directory>/succsave/<levelset name>/<level number>_<level title>-<ruleset>.json*. If such a file already exists, it will be overwritten. If you wish to save in a different location and/or with a different name, use **Solution > Save as** or **Ctrl + Shift + S**.

Opening a Solution

You can open a solution through **Solution > Open** or **Ctrl + O**.

Copying

Solutions can also be copy and pasted with raw text. To copy one, use **Solution > Copy solution** or simply **Ctrl + C**. This will generate the entire JSON and place it in your clipboard.

You can also copy the raw moves up to and including the current position into your clipboard with **Solution > Copy all previous moves**. Similarly, you can do the same with future moves with **Solution > Copy all future moves**. You can also get the raw moves of macros at **Solution > Copy macro moves** and choosing a macro.

Pasting

To paste a copied solution, use **Solution > Paste solution** or **Ctrl + V**. If the copied solution is a complete JSON, it will run equivalently to opening one — from the beginning of the level. However, if the solution isn't a proper JSON, it will try parsing the string as raw moves, meaning it only looks for the characters u, r, d, l, -, ↖, ↙, ↘, and ↗, both uppercase and lowercase, and characters representing mouse moves. It executes them on the spot similar to a macro.

Savestates

A savestate is a snapshot of a state of the level. They can be used to quickly go back to a specific point. When you change the level they are deleted.

Creating a Savestate

A savestate may be created with the combination **Shift + <any key>**. You cannot use keys that are bound to some control.

Loading a Savestate

To load a savestate saved under some key, just press it.

Saving Savestates to Disk

If you want to keep your savestates even after quitting SuperCC or changing a level, you can use **Solution > Save all states to disk** to do so. This will create a SER file in the place solutions are saved with the same name. **Note:** The current moves and state are not saved unless they were saved into a savestate.

Loading Savestates from Disk

To load all saved savestates for a level, simply go to **Solution > Load states from disk** and choose the appropriate file. You may then use the savestates.

Macros

Macros are recorded sequences of moves that can be executed at any point. They're essentially a copy-paste function for moves. They can be very useful if you want to modify an earlier point in the route without having to redo the rest.

Creating a Macro

To start recording moves, press **Ctrl + Shift + <number key>**. When you are done, repeat that combination to end recording.

Loading a Macro

To run your recorded moves, simply press **Ctrl + <number key>**.

Saving Macros to Disk

Macros are saved along with savestates. Therefore, they can be saved with **Solution > Save states to disk**.

Loading Macros from Disk

Similarly, use **Solution > Load states from disk** to load macros from disk.

Chapter 4 – TWS

SuperCC offers integration with TWS files both for reading and writing purposes.

All TWS functionality is offered in the **TWS** menu. The options are described below.

Write Solution to New TWS
: This creates a new TWS file that only contains the solution being saved for the current level.

Open TWS
: This loads the entire TWS for the levelset and allows you to load solutions from it.

Load Solution
: If a TWS is opened, it loads the solution saved within it.

Verify TWS
: If a TWS is opened, it goes through all level solutions and determines whether they were saved via Tile World, SuperCC, Melinda Router, or if there is no solution. The information is displayed in a popup window.

Chapter 5 – Miscellaneous

Level Settings

Settings related to the level itself can be found under the **Level** menu or accessed with the function keys **F1** – **F4**.

Change Stepping Value (F1)

Changes the initial step of the level. It cycles through all values for the ruleset. The MS ruleset has stepping values EVEN and ODD, Lynx has EVEN, EVEN + 1, EVEN + 2, EVEN + 3, ODD, ODD + 1, ODD + 2, and ODD + 3.

Set RNG Seed (F2)

Changes the initial seed of the level that affects all random elements. Valid values are between 0 and 2147483647 (2^{31}-1)

Change Ruleset (F3)

Flips between the MS and Lynx rulesets.

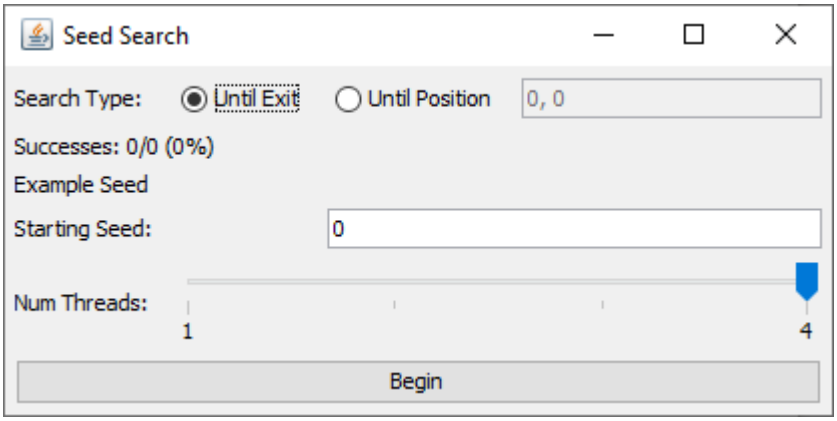
Change Initial RFF Direction (F4)

Cycles through the initial direction of RFFs: UP, RIGHT, DOWN, LEFT. This setting only applies to the Lynx ruleset.

Seed Search

A common problem in *Chip's Challenge* are random elements causing the same route to sometimes fail and sometimes succeed. The seed search tool will allow you to estimate the success rate of a given route and even find specific seeds that work. To access the tool, go to **Solution > Search for seeds** and select the JSON file of the route you want to test.

You may pause the search at any time and resume it later, or end it by closing the window.



Seed Search Window

Search Type

There are two options to search for seeds: **Until Exit**, which determines a seed successful if the solution ends on the player completing the level, or **Until Position**, which checks if the solution ends on the player being on the given coordinate.

Successes

Displays the proportion of seeds in which the solution is successful.

Example Seed

Shows the last found successful seed.

Starting Seed

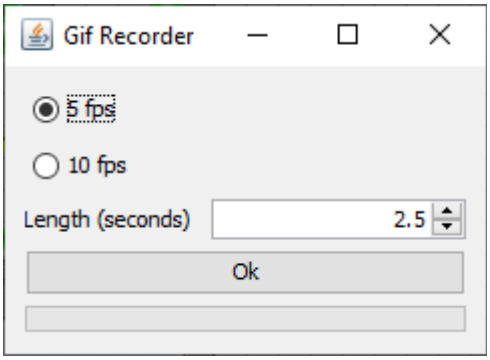
Allows you to set which seed the search will start from.

Num Threads

Allows you to set how many CPU threads will be used to search for seeds. Each thread will then search through its own range concurrently.

GIF Recorder

This tool allows you to output an animated GIF file of part of the playback. You can access it through **Tools > Record gif**.



GIF Recorder Window

There are only two settings for a GIF recording: should it play at 5 or 10 frames per second, and how many seconds it should play back. If the length is longer than the playback, it will only record for the duration of the playback.

The recording will start from the current state and progress through the move history until the end or the appropriate length is reached.

Command Line Usage

SuperCC can be run through a terminal with additional options. Below is the help for the usage.

```
usage: SuperCC.jar [-h] [LEVELSET [-lr N] [-s STEP] [-f DIR] [-m RULE] [TWS [--testtws]]]
-h        Display this help and exit.
-l        Load level number N.
-r        Load level with starting RNG seed N.
-s        Load level with a step parity of STEP.
-f        Load level with an initial random force floor direction of DIR.
-m        Load level with a ruleset of RULE.
LEVELSET  Open the levelset given by this path.
TWS       Set the TWS file to the one given by this path.
--testtws Perform a unit test on the given TWS file with the given levelset.

STEP must be one of: [EVEN, EVEN + 1, EVEN + 2, EVEN + 3, ODD, ODD + 1, ODD + 2, ODD + 3].
DIR must be one of:  [UP, LEFT, DOWN, RIGHT].
RULE must be one of: [MS, LYNX].

Each flag has an alternate form of: [-h/--help/-?], [-l/--level], [-r/--rng],
[-s/--step], [-f/--rff], [-m/--rules].
```

For example, to automatically open LESSON 8 in odd step in the lynx ruleset, one could type

```
java -jar SuperCC.jar data/CHIPS.DAT -l 8 -s ODD -m LYNX
```

Chapter 6 – Cheats

Cheats are a powerful routing feature that allow you to dynamically manipulate the state of the level.

Cheats are not saved in solutions, both JSON and TWS. They are however preserved in savestates.

Cheats Menu

Some cheats are located in the **Cheats** menu. Specifically ones that alter the general state of the level.

Change Inventory
: Opens a window that allows you to modify how many keys you have, what boots you have, and how many chips you still have to collect.

Change Timer
: Opens a window where you can change the time remaining. This affects blob and teeth movement.

Change Monster List Positions
: Opens a popup window where you can manually modify the monster list order.

Change Slip List Positions
: Opens a popup window where you can manually modify the slip list order.

Press Green Button
: Simulates a green button press.

Press Blue Button
: Simulates a blue button press. The tanks turn around immediately and start moving on the very next turn.

Right Click Menu

You can also modify single tiles of a level. Right clicking on a tile opens up a cheats menu for it. The options slightly differ depending on the type of tile.

Move Chip Here
: Instantly moves the player onto the chosen tile.

Remove Tile
: Removes the tile on the foreground layer, bringing forward whatever is in the background.

Insert Tile
: Inserts the chosen tile onto the foreground layer, pushing any non-floor tile already there to the background (MS) or simply overwriting it (Lynx). The hexadecimal values are as defined by the DAT format. Lynx doesn't allow adding creatures as inanimate tiles, including blocks.

Insert Creature

Inserts the chosen creature onto the current tile, facing the chosen direction, fully animated. Lynx considers blocks to be creatures and doesn't allow tanks to be still when it can move forward.

Change Creature's Direction

This option is only visible if the selected tile contains a creature. It allows you to instantly change its direction.

Animate Creature

This option only shows up on creatures that are not in the monster list. It adds the creature onto the end of the list.

Kill Creature

Removes the selected creature from the level, appropriately adjusting the monster order of other creatures.

Press Button

Shows up on buttons. Simulates a click of that button on the current tick.

Open Trap

Shows up on traps only in the MS ruleset. Opens the selected trap on the current tick.

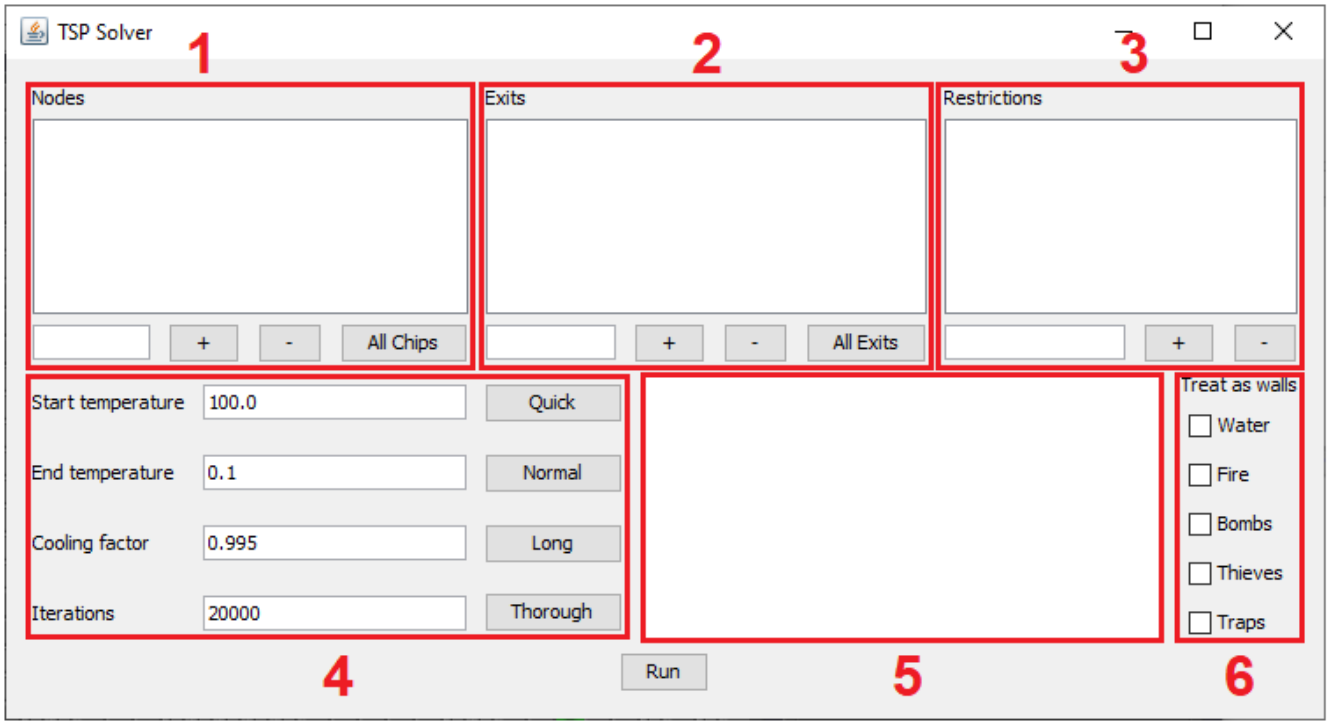
Close Trap

Shows up on traps only in the MS ruleset. Closes the selected trap on the current tick.

Chapter 7 – TSP Solver

TSP stands for Traveling Salesman Problem. It is the problem of finding the shortest path that visits all points once. Computationally, it is not a trivial problem and this tool uses heuristics to hopefully find a solution as close to optimal as possible. You may find this tool in **Tools > TSP Solver**.

User Interface



TSP Solver User Interface

1. **Nodes**
   This is the list of all positions the player has to visit. To add or remove a position, type in two numbers between 0 and 31 inclusive separated by a space and click the appropriate **+** or **–** button. The numbers represent the x and y coordinates respectively. Alternatively, you can press the **All Chips** button to automatically add all chips in the level to the list.
2. **Exits**
   This list contains the possible end points of the path. Only one of these can be chosen. The **All Exits** button automatically adds all exit tiles in the level to the list. All other user interface functionality is the same as in nodes.

3. **Restrictions**
   This list specifies which node needs to be visited before another. This requires the input of four numbers between 0 and 31 inclusive separated by spaces. They represent the x1 and y1 coordinates of the node that must be visited before the node at x2, y2. Be careful not to create a dependency loop, otherwise no solution will be found.
4. **Algorithm Settings**
   The solver uses the simulated annealing algorithm. In short, it generates solutions and randomly varies them. The lower the temperature, the more likely it is to reject worse solutions as it slowly homes in on the local optimum. The temperature drops fast at first, but slows down exponentially by the cooling as it approaches the end temperature. It takes the following four inputs:
   1. **Starting temperature** — the starting value of the temperature parameter, which decreases over time at each step.
   2. **Ending temperature** — the boundary value that ends the algorithm when temperature goes below it.
   3. **Cooling factor** — the value temperature is multiplied by at each step, it must by smaller than 1. Optimally, it should be as close to 1 as possible, although that can drastically increase running time.
   4. **Iterations** — how many different paths the algorithm will try at each step.

   You may use the presets **Quick**, **Normal**, **Long**, **Thorough** to automatically set the parameters.
5. **Information Display**
   This outputs the information about the progress of the algorithm.
6. **Treat As Walls**
   This setting will treat the selected tiles as walls when finding the shortest paths between nodes.

Running the Solver

When the algorithm is run, the level first restarts, so you may need to set up the level state as a separate level.

Next, a path-finding algorithm attempts to find the shortest distance between all nodes and exits. The path-finding algorithm ignores all monsters, as well as most tiles. The only thing it takes into account are: Chip, ice, force floors, teleports, and all acting walls except closed toggle doors.

Next, the simulated annealing algorithm is run on the generated distances. It finds progressively faster routes that go through all nodes and reaches an exit while obeying all restrictions.

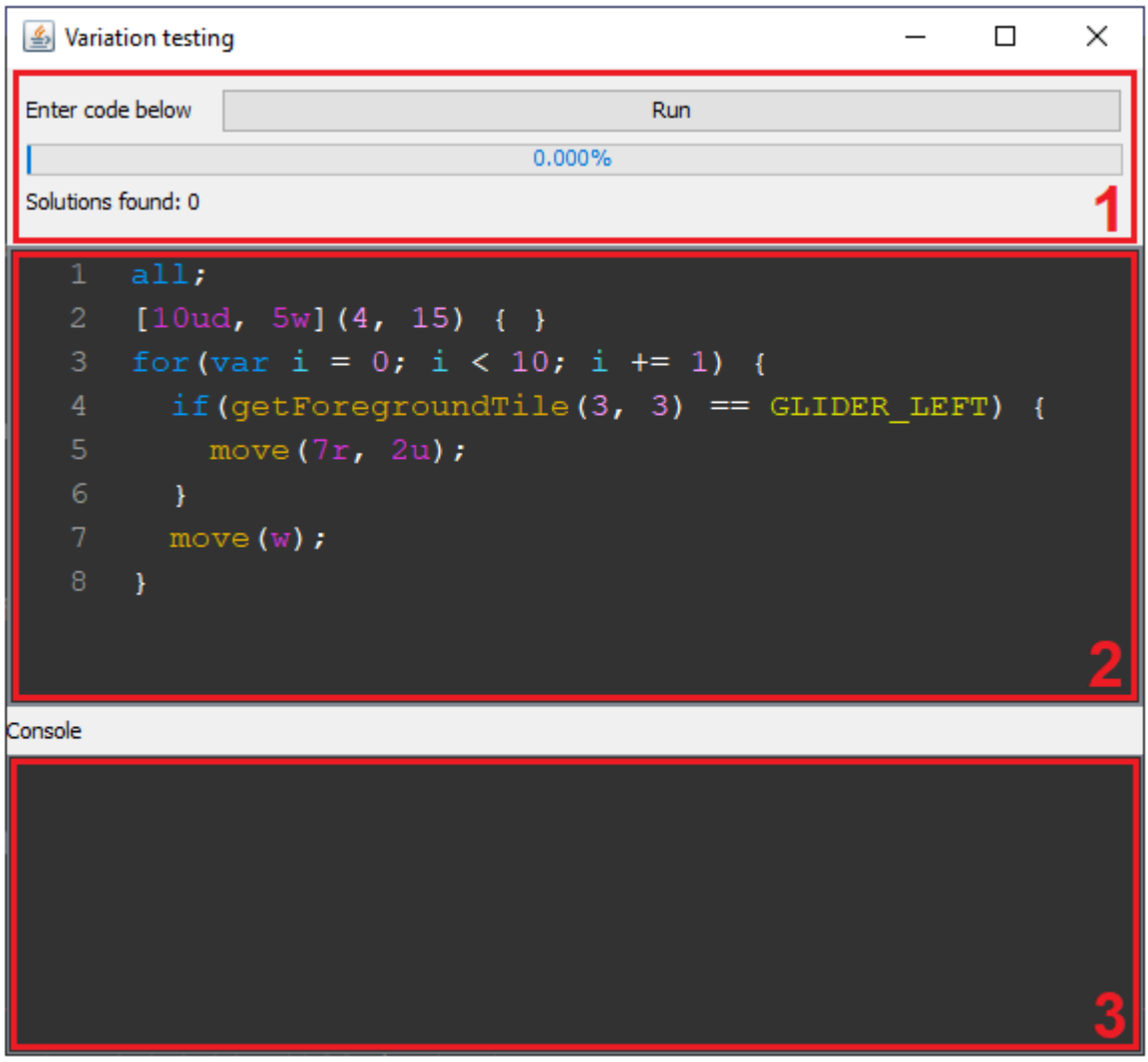
The result is not necessarily optimal due to the nature of the problem and the algorithm. It may require multiple runs to find the optimal route. The more nodes there are, the more possible routes there are, and that number increases very fast (factorial).

Chapter 8 – Variation Testing

Some levels may feature an area where the player cannot easily predict the outcome and must test every possibility in order to find the fastest route. This often includes lots of monster collisions. SuperCC's solution to this is **VariationScript**, a scripting language designed to automate variation testing.

VariationScript is similar to many C programming language family languages in its syntax. However, it does not belong to it. It abstracts away the generation of all permutations of moves as defined by the player and offers functions that give information about the state of the level at any point of the process.

Variation testing can be accessed through **Tools > Variation testing**.



Variation Testing Window

1. **Progress Area**
   This region displays the progress information while the script is running. There is a progress bar that shows the percentage of variations tested and the number of solutions found so far. Progress isn't necessarily linear, as large branches of the search space may be discarded, resulting in a jump in progress. To start the script press the **Run** button and to stop it, press it when it says **Stop**.
2. **Code Area**
   This is where the VariationScript code goes. Syntax highlighting is done automatically.
3. **Console**
   This displays all messages regarding the script. It displays all syntax and runtime errors encountered in the code. When running the script, it shows the upper bound of the number of variations to test. When the script ends, it shows how many of them have been tested, how many solutions were found, how long the script ran, and how many variations per second were tested on average. All print statements are also displayed here.

Basics

The language is divided into four basic units:

1. **Sequences**
   These represent a set of permutations to be generated in a specific order from some collection of moves. A valid script must contain at least one sequence.
2. **Statements**
   These are the building blocks of code, containing code to be executed.
3. **Expressions**
   These evaluate to some value.
4. **Literal Value**
   These represent a specific value of a certain type.

As mentioned above, **a valid script must contain at least one sequence**. All else is optional.

For the script to run successfully, it must have correct syntax and no runtime errors. Syntax errors are caught immediately on running the script and the console displays where the error is and why, although sometimes the mistake may not be immediately obvious. When multiple errors are present, it is best to take care of the first one before trying again, as a syntax error in one part of the program may cause more parsing errors in a later part as well.

Runtime errors happen when the script is already running and encounters something invalid that the syntax doesn't forbid. This may be, for instance, a mathematical operation performed on a variable that doesn't contain a number.

There may also be unknown errors or ones relating to the implementation of the language itself. Please report these errors with as much information as possible.

Comments are ignored by the script. A comment is anything after `//` until the end of the line.

Sequences

The most important unit of code. Without one, the script cannot run. Its purpose is to generate all permutations of some collection of moves in a particular, run them, and possibly execute some code during that.

Permutations are generated in a specific order that can be thought of as an alphabetical order but you can set the order yourself. By default, the order is `u`, `r`, `d`, `l`, `w`, `h`. This represents the moves up, right, down, left, wait, and half-wait respectively. This order means that up comes before right and so on. Shorter permutations come before longer ones.

The syntax of sequences is as follows:

```
[<forcedMoves>][<moves>](<range>) {
  order <moveOrder>;
  <control>: <statement>
}
```

- `<forcedMoves>` — a comma-separated collection of moves that each permutation must contain. The order of the moves doesn't matter, as the ordering is determined by the `order` statement.
- `<moves>` — a comma-separated collection of moves, just like `<forcedMoves>`, but this collection doesn't force all moves to appear in the permutation. This means that if the size of the collection is larger than the length of the permutation, the sequence will also generate all subsets of these moves in the same order as the permutations are generated in.
- `<range>` — the range of lengths of the permutations to be generated. This counts compound moves as a single move. The numbers must be given in the literal form. If no value is given, it assumes the size of `<forcedMoves>` and `<moves>`. If the range contains one number, that is the only length that will be generated. If the range is two comma-separated numbers, the generated permutations will have length between those values inclusive if possible, regardless of order.
- `order <moveOrder>` — an optional statement that allows you to set the ordering of generated permutations and subsets. `<moveOrder>` must contain each of the six types of moves exactly once in the order you want. An absence of this statement implies an `order urdlwh;`
- `<control>` — specifies what should happen at a specific point of running a variation. It is to be followed with a single statement (for multiple statements, use a block statement). A sequence can have any number of controls. They are as follows:
  - `start` — executed before a permutation is run
  - `beforeMove` — executed before a move of the permutation is made
  - `afterMove` — executed after a move of the permutation is made
  - `beforeStep` — executed before the player performs a move
  - `afterStep` — executed after the player performs a move
  - `end` — executed after a permutation is run

  Note the distinction between move and step. A move is the entire unit and is made of multiple steps. For example, `urr` is a single move consisting of three steps `u`, `r`, and `r`.

As an example:

```
[u, 2dd][3r, l](4, 6) {
  order dlruwh;
}
```

This will generate all permutations of lengths 4 to 6 that contain the moves `u`, `dd`, `dd`. Additionally, it can contain any of the moves `r`, `r`, `r`, `l` to fill out the length. The order was also set to `dlruwh`, which means that down moves are first in the "alphabetical" order.

Therefore, the first permutation will be `dddddlu` followed by `ddddul` then `ddlddu` and so on until it reaches `urrrdddd`.

Statements

These contain common control structures as well as code to be executed.

- `all <expression>;` — by default, VariationScript immediately stops running and returns the first solution it finds. To have it return more solutions, this statement must be present within the code, best at the beginning. The expression sets the maximum number of solutions the script will return. It is optional and by default it's set to 1000. If an expression is present, it must evaluate to a number.
- `if/else` — a conditional control structure. If the expression within the `if` statement is true, the following statement is executed. Otherwise, it tests the next `else if` branch or executes the statement after `else`. Whether an expression is true or false is explained in the next section. There can be any number of else if branches. To have it run multiple statements, a block statement is required. The syntax is:

  ```
  if(<expression>)
    <statement>
  else if(<expression>)
    <statement>
  else
    <statement>
  ```

- `for` — a looping control structure. It consists of three parts: the first one is an expression statement executed before the loop begins; the second one is an expression that evaluates to true or false, if true, the statement after the loop is executed, if false, the loop ends; the third one is an expression to be evaluated after a loop is complete and before the condition is checked. The most common loops set a counter variable in the first part, check if it's smaller than some value, and increase it by 1 in the third part. The syntax is:

  ```
  for(<expression statement>; <expression>; <expression>)
    <statement>
  ```

- Variable declaration — creates a variable and optionally initializes it with a value evaluated by an expression. An uninitialized variable has a value of `null` by default. Variable names must begin with a letter, _, @, #, or $ and must not be anything used by the language (literal values, functions, keywords, tiles, etc.) Use syntax highlighting to see if a name is valid. Variables are visible throughout the script and are not limited by any scope. They also do not carry over through permutations. The syntax is:

  ```
  var <variable name> = <expression>;
  ```

  ```
  var asd = 0;
  [u, r](){}
  asd += 1;
  print asd;
  ```

  The above code prints 1 twice, showing that each variation has its own state. There are no global variables.

- `return;` — returns the solution at the current point. If the solution limit is reached, the script ends. The limit is 1 by default, but can be changed with `all`. When the player reaches an exit, the solution is immediately returned.
- `continue;` — stops the current permutation and generates the next one.
- `terminate <expression>;` — this statement skips all permutations with a prefix of some length. If an expression is not given, the length is how many moves have been executed up to the current point. This means that there is something undesirable about starting with the current sequence of moves so it's skipped. The player dying is an immediate termination of the current prefix. If an expression that evaluates to a number is optionally given, it terminates the prefix of that length of the current permutation.
- `print <expression>;` — prints the value of the expression to the console.
- `{ <statements> }` — a block statement that can contain multiple statements. It's useful for having a control structure execute multiple statements.
- `<expression>;` — an expression statement. Mostly useful for a variable assignment.

Expressions

These evaluate to some value.

- Literal value
- Unary operation — takes an operator and an expression.
  `<operator> <expression>`

  VariationScript contains these unary operators:

  - `not`, `!` — flips the boolean value of the expression.

- Binary operation — takes two expressions with an operator between them.
  `<expression> <operator> <expression>`

  The operators are as follow:
  - `+`, `-`, `*`, `/` — respectively add, subtract, multiply, and divide the expressions.
  - `%` — calculates the remainder after dividing the first expression by the first.
  - `and`, `&&` — evaluates the boolean AND of both expressions.
  - `or`, `||` — evaluates the boolean OR of both expressions.
  - `=`, `+=`, `-=`, `*=`, `/=`, `%=` — assignment operators. The first expression needs to be a variable in this case. An operator before the equals sign signifies that it's assigning to the variable that operation between the variable and the second expression, e.g. `x += 3` is equivalent to `x = x + 3`. Returns the new value of the variable.
  - `<`, `<=`, `==`, `>=`, `>`, `!=` — comparison operators. They are respectfully less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to. The result is a boolean value of the comparison.
- Function call — VariationScript has many functions available that return some information regarding the script or the level. Some of them take arguments.
  - `previousMove()` — returns the last step of the last executed move.
  - `nextMove()` — return the first step of the next move to be executed.
  - `getMove(n)` — returns the first step of the `n`-th move in the current permutation.
  - `getOppositeMove(m)` — returns the opposite direction of the given move `m`. Waits do not have opposites and returns them unchanged.
  - `movesExecuted()` — returns the number of moves in the permutation that have been executed up to the current point. This does not include individual steps or moves done through move().
  - `moveCount(s)` — returns the number of times the step `s` appears in the current permutation.
  - `seqLength()` — returns the total number of moves in the permutation.
  - `getChipCount()` — returns the number of chips left to collect.
  - `getTimeLeft()` — returns the time left.
  - `getRedKeyCount()`, `getYellowKeyCount()`, `getGreenKeyCount()`, `getBlueKeyCount()` — returns the number of keys the player has of the respective key color.
  - `HasFlippers()`, `hasFireBoots()`, `hasSuctionBoots()`, `hasIceSkates()` — returns the boolean value of the player having the respective boot.
  - `getForegroundTile(x, y)`, `getBackgroundTile(x, y)` — returns respectively the tile in the foreground layer and background layer at position (`x`, `y`).
  - `getPlayerX()`, `getPlayerY()` — returns respectively the player's `x` and `y` coordinate.
  - `distanceTo(x, y)` — returns the Manhattan distance (distance in tiles) from the player to coordinate (`x`, `y`).
  - `move(<moves>)` — executes the comma-separated list of moves in the given order. These moves are not part of permutations and are not counted in other functions. They are completely independent.
- Variable — evaluates to its value.

Literal Values

These values are the most basic and fundamental data in a script. There are five types of literal values:

- Numbers — a series of digits optionally followed by a decimal point and another series of digits. They are implemented by 64-bit double-precision floating-point numbers which limits what numbers they can represent. E.g. `1`, `4.2`, `1638.435`
- Moves — an optional whole number followed by a string of letters `u`, `r`, `d`, `l`, `w`, `h`. The letters represent the sequence of moves and the number is how many times the entire sequence is repeated. E.g. `3ud` is equivalent to `ud, ud, ud`.
- Boolean values — can be either `true` or `false`. Conditions evaluate to one of these two values. The value `null` and the number `0` are evaluated as `false` in boolean contexts. All other numbers and moves and tiles are evaluated as `true`.
- `null` — represents the absence of value. It's often found in uninitialized variables and expressions that evaluate to it like the function call `move(...)`.
- Tiles — represents the value of a tile in a level. The literal names are as follow:

FLOOR
WALL
CHIP
WATER
FIRE
INVISIBLE_WALL
THIN_WALL_UP
THIN_WALL_LEFT
THIN_WALL_DOWN
THIN_WALL_RIGHT
BLOCK
DIRT
ICE
FF_DOWN
BLOCK_UP
BLOCK_LEFT
BLOCK_DOWN
BLOCK_RIGHT
FF_UP
FF_RIGHT
FF_LEFT
EXIT
DOOR_BLUE
DOOR_RED

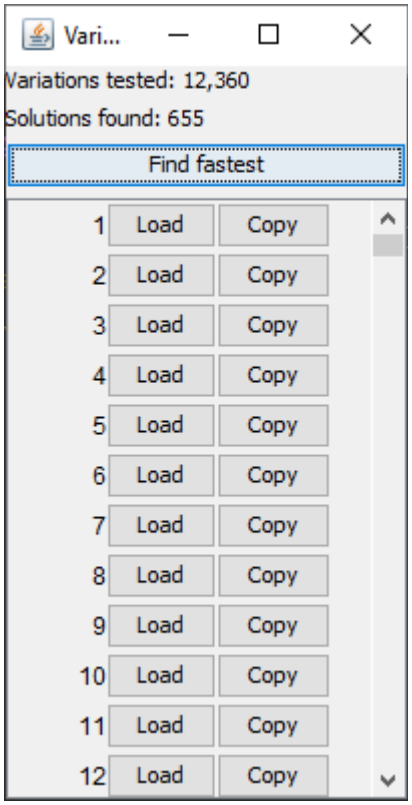
DOOR_GREEN
DOOR_YELLOW
ICE_SLIDE_SOUTHEAST
ICE_SLIDE_SOUTHWEST
ICE_SLIDE_NORTHWEST
ICE_SLIDE_NORTHEAST
BLUEWALL_FAKE
BLUEWALL_REAL
OVERLAY_BUFFER
THIEF
SOCKET
BUTTON_GREEN
BUTTON_RED
TOGGLE_CLOSED
TOGGLE_OPEN
BUTTON_BROWN
BUTTON_BLUE
TELEPORT
BOMB
TRAP
HIDDENWALL_TEMP
GRAVEL
POP_UP_WALL
HINT

THIN_WALL_DOWN_RIGHT
CLONE_MACHINE
FF_RANDOM
DROWNED_CHIP
BURNED_CHIP
BOMBED_CHIP
UNUSED_36
UNUSED_37
ICE_BLOCK
EXITED_CHIP
EXIT_EXTRA_1
EXIT_EXTRA_2
CHIP_SWIMMING_UP
CHIP_SWIMMING_LEFT
CHIP_SWIMMING_DOWN
CHIP_SWIMMING_RIGHT
BUG_UP
BUG_LEFT
BUG_DOWN
BUG_RIGHT
FIREBALL_UP
FIREBALL_LEFT
FIREBALL_DOWN
FIREBALL_RIGHT

BALL_UP
BALL_LEFT
BALL_DOWN
BALL_RIGHT
TANK_UP
TANK_LEFT
TANK_DOWN
TANK_RIGHT
GLIDER_UP
GLIDER_LEFT
GLIDER_DOWN
GLIDER_RIGHT
TEETH_UP
TEETH_LEFT
TEETH_DOWN
TEETH_RIGHT
WALKER_UP
WALKER_LEFT
WALKER_DOWN
WALKER_RIGHT
BLOB_UP
BLOB_LEFT
BLOB_DOWN
BLOB_RIGHT
PARAMECIUM_UP
PARAMECIUM_LEFT
PARAMECIUM_DOWN
PARAMECIUM_RIGHT
KEY_BLUE
KEY_RED
KEY_GREEN
KEY_YELLOW
BOOTS_WATER
BOOTS_FIRE
BOOTS_ICE
BOOTS_SLIDE
CHIP_UP
CHIP_LEFT
CHIP_DOWN
CHIP_RIGHT

Solutions

When a script is done running and at least one solution was returned, a popup window will appear with all the found solutions.

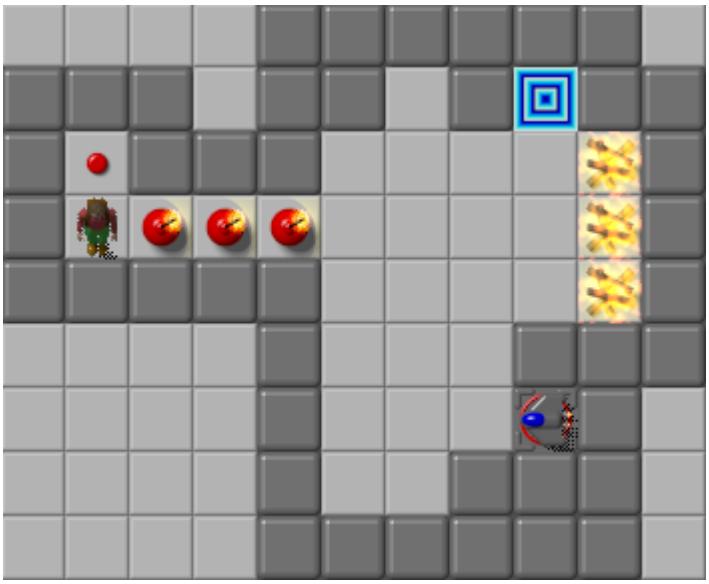


To load the entire solution into SuperCC, simply press **Load** next to the desired solution. **Copy** puts the JSON into your clipboard. The **Find fastest** button will load the solution that returned with the highest time left. In case of a tie, it chooses the first one found.

Examples

Now that the language's building blocks have been described, they can be put together to create powerful scripts that can try thousands of variations per second to find the fastest route.

Scenario 1



We see that we will have to clone some gliders to blow up the bombs so the player can reach the exit. This will require glider collisions to knock them into the bombs since their natural path leads them to fire, but not without some crossing over itself, allowing for the collisions. After cloning, we will have to wait until the bombs are cleared before we can go for the exit.

Therefore, an example script to help us find the fastest solution to this level may look like this:

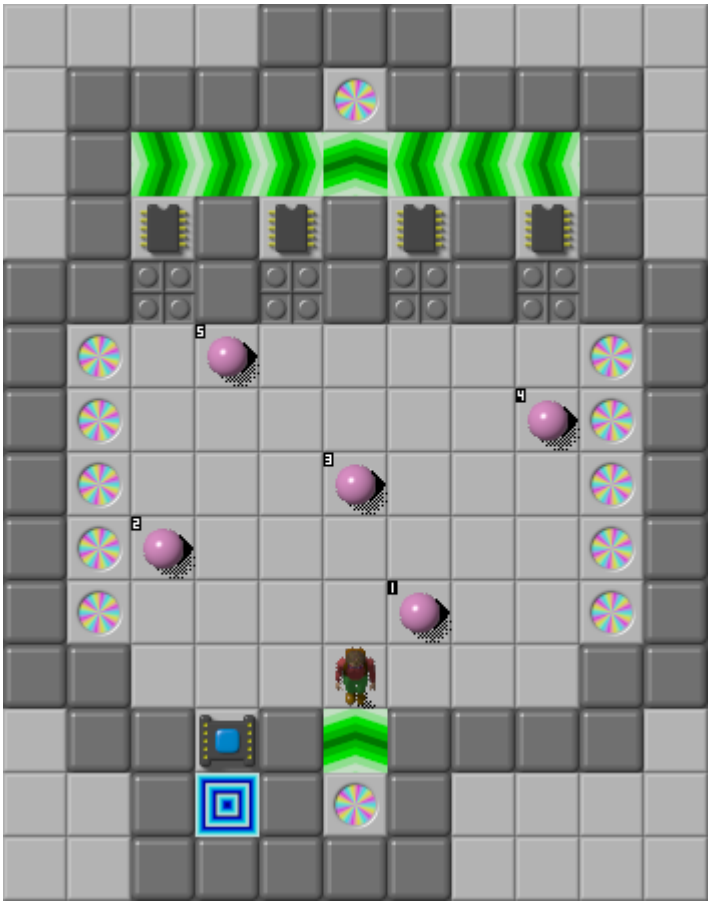
```
all;
[10ud, 5w](4, 15) { }

for(var i = 0; i < 10; i += 1) {
  if(getForegroundTile(3, 3) == GLIDER_LEFT) {
    move(7r, 2u);
  }
  move(w);
}
```

We obviously want all the solutions (the default limit of 1000 is enough here) since the first one might not necessarily be the fastest (due to waiting for the bombs to blow up). For the cloning part, we arbitrarily chose up to 10 button presses and up to 5 waits between them. We'll (also arbitrarily) try all permutations between 4 and 15 of these moves.

After the permutation is done, we wait up to 10 moves using a `for` loop. If the tile at (3, 3) has a glider going left, it means that it's about to blow up the last bomb and free the player. So we rush the player to the exit. If the player succeeds, the solution is automatically returned.

The program searches through 12,360 permutations and finds 655 successful solutions.

Scenario 2



For this example, let's ignore the obvious bust as it's not the point of this exercise.

In this level we need to cross the pink ball road four times to get the chips before exiting. We can notice some possible automation. Once the player is under the desired recessed wall, we can keep going up until the player goes through the teleport and steps onto tile (5, 11).

Notice that we require to make 20 moves up (4 times we need to go up 5 times to reach the last row before the recessed walls). We will also need to go left and right to position the player under one of the chips. That would be a total of 4 moves right and 4 moves left. However, maybe we can use a teleport to go faster. To use a teleport, we will need a total of 8 moves towards one side. So for symmetry, we add `8l` and `8r` to the pool. We can also count that we'll need exactly 29 moves in this solution.

The search space of `[20u][8l, 8r]() { … }` is huge, so we'll need to cut some of it out. First of all, a solution where the player goes `lr` or `rl` won't work, so we throw those out from the start. Another trick we can use is make sure that the player takes exactly 5 up moves to reach a chip, otherwise it means the player hit a wall between the chips. We discard these solutions.

We also see that any solution will be the same length, so we need only one. Therefore, we don't use the `all;` statement here.

With all these considerations, we can write up a script to solve this problem.

```
var upMoves = 0;
[20u][8l, 8r](29) {
  start: {
    var x = seqLength();
    for(var i = 0; i < x - 1; i += 1) {
      var m1 = getMove(i);
      var m2 = getMove(i + 1);
      if(m1 == getOppositeMove(m2)) terminate i + 1;
    }
  }

  afterMove: {
    if(getPlayerY() <= 5) {
      if(getForegroundTile(getPlayerX(), 3) == CHIP) {
        for(;getPlayerY() != 11;) {
          move(u);
        }
      }
    }

    if(previousMove() == u) {
      upMoves += 1;
    }

    if(upMoves == 5 and getChipsLeft() > 3) terminate;
    if(upMoves == 10 and getChipsLeft() > 2) terminate;
    if(upMoves == 15 and getChipsLeft() > 1) terminate;
  }
}
move(2l,2d);
```

There's a bit to unpack here.

First, in the `start` control, we go through all adjacent pairs of moves in the permutation. If they are opposites, we terminate the prefix up to where that pair appears. This throws out all solutions with `lr` and `rl` moves.

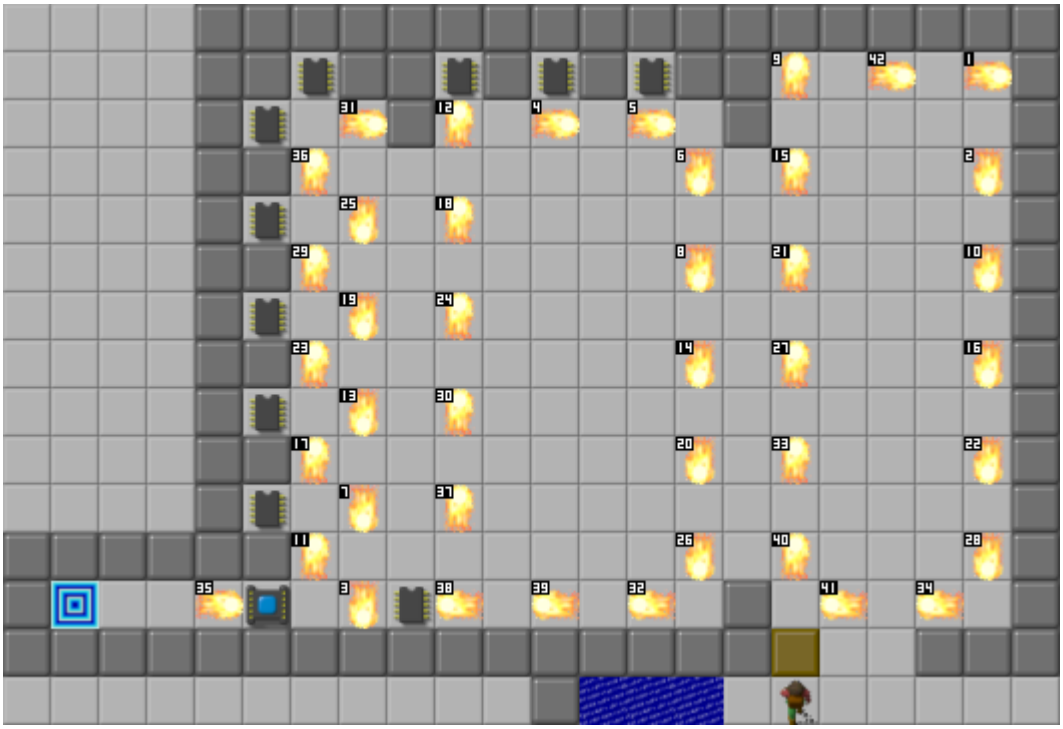
In the `afterMove` control, we first check if the player is on the row just under the recessed walls. If so, we check if there's a chip above it. Is so, we keep moving up until the player steps onto the force floor at (5, 11). We also check if we've just executed an up move. If so, we increment the counter of those moves and then check if we didn't oof against a wall. If we did, we terminate at this point.

When running the script, we see that is has a huge upper bound of variations (5,107,652,550). This is why we use these search space pruning techniques and just return the first solution, which is found after 942,327 variations. Upon loading it, we see it indeed does cleverly use the teleport to avoid waiting for a ball to pass.

Notice how the loop in the code `for(;getPlayerY() != 11;)` just uses the condition and ignores the other two parts. This is completely valid code and sometimes useful. Although, be careful not to cause infinite loops when using conditions other than the simple counter.

This script is far from perfect, but it demonstrates how we may approach finding a solution and pruning the search space. This is just one possible way of obtaining a specific type of solution.

Scenario 3



Here's a familiar level. The search space is too large to completely search. So in this example we will search through variations on the public bold route. For reference, the block is at (22, 20).

First we have to wait. Then we let up to two fireballs pass before pushing the block up. Then we let up to two more fireballs bounce off the block. We push the block up into the range (22, 10) – (22, 17). We push it over to column 20 and go under it. Then we do a combination of waits and up moves. If we push the block into the corner, we terminate the sequence of moves that lead to it. Finally, we just do what the public route does and get all the chips (but the last one) and return (if the player survived).

```
all;
move(w);
[2ww](0, 2) { }
move(u);

[2ww](0, 2) { }
move(2u);

[7u](0, 7) { }
move(rulldl);

[7u, 5w](1, 12) {
  afterMove: {
    if(getForegroundTile(20, 9) == BLOCK)
      terminate;
  }
}
move(ru, 3lluudd, llullrud, 4ddlr);
return;
```

The upper bound of this is not too high (216,072 variations). However, due to the large number of monsters in this level, the script will run noticeably slower.

The script ran through a total of 148,093 variations and found 44 solutions. Not all have a unique result and some even have an absurd one. A bit of manual labor will reveal that the bold route is contained in these results and it's the fastest route. Do not mistake this for proved optimality though, we just tried a specific type of solution.

The main points to observe here are that sometimes search spaces are very big and finding a way to automate variation testing is difficult. We may have to significantly narrow our search and try lots of things by hand via trial and error too see what works. It's also difficult to describe some things more specifically using just VariationScript.

Tips and Tricks

Below are some observations that might be useful in writing scripts.

- The number of permutations grows very fast. Even a relatively small number of moves can take a long time to test. Therefore you should try pruning as much of the search space as possible. The earlier you cut off a branch, the larger the cut.
- Custom move ordering can be a good optimization tool if successful variations are more likely to begin in a specific direction. e.g. if u and l are the most likely first 2 moves, you can use something like `order ulrdwh;` Note that this doesn't matter if you plan on running through all possible permutations.
- To speed up the search, you can run a script multiple times with different starting moves to explore only specific branches.
- Use forced moves when you know they must exist somewhere in the solution (e.g. if a goal tile is 10 tiles to the right of the player, `10r` is forced). The program will then only consider those permutations that contain all forced moves, reducing the upper bound of variations.
- Try to identify as many cases as you can where a sequence of moves won't work at all and terminate them at that point. E.g. if you want to avoid the player moving backwards you can write the following to remove branches that contain those moves:

```
start: {
  var x = seqLength();
  for(var i = 0; i < x - 1; i += 1) {
    var m1 = getMove(i);
    var m2 = getMove(i + 1);
    if(m1 == getOppositeMove(m2)) {
      terminate i + 1;
    }
  }
}
```

- Remember to write `all x;` somewhere in the code if you want to explore more variations instead of ending after finding the first solution.

- Only variable names are case-sensitive in VariationScript.
- The program only searches with the set seed, so it may be a good idea to set the correct one or remove random elements altogether.
- The speed of testing variations varies depending on the machine it's running on, and the level itself. Longer sequences and routes will take a longer time to compute. More monsters in a level also slows down testing. Therefore, it may be more efficient to test on a modified level that removes insignificant elements.
- Sequences with a lower bound of 0 are valid and means they're optional.