Writing a “Baba Is You” solver

Julien Kirch

Version 0.1, 2019-05-11

Table of Contents

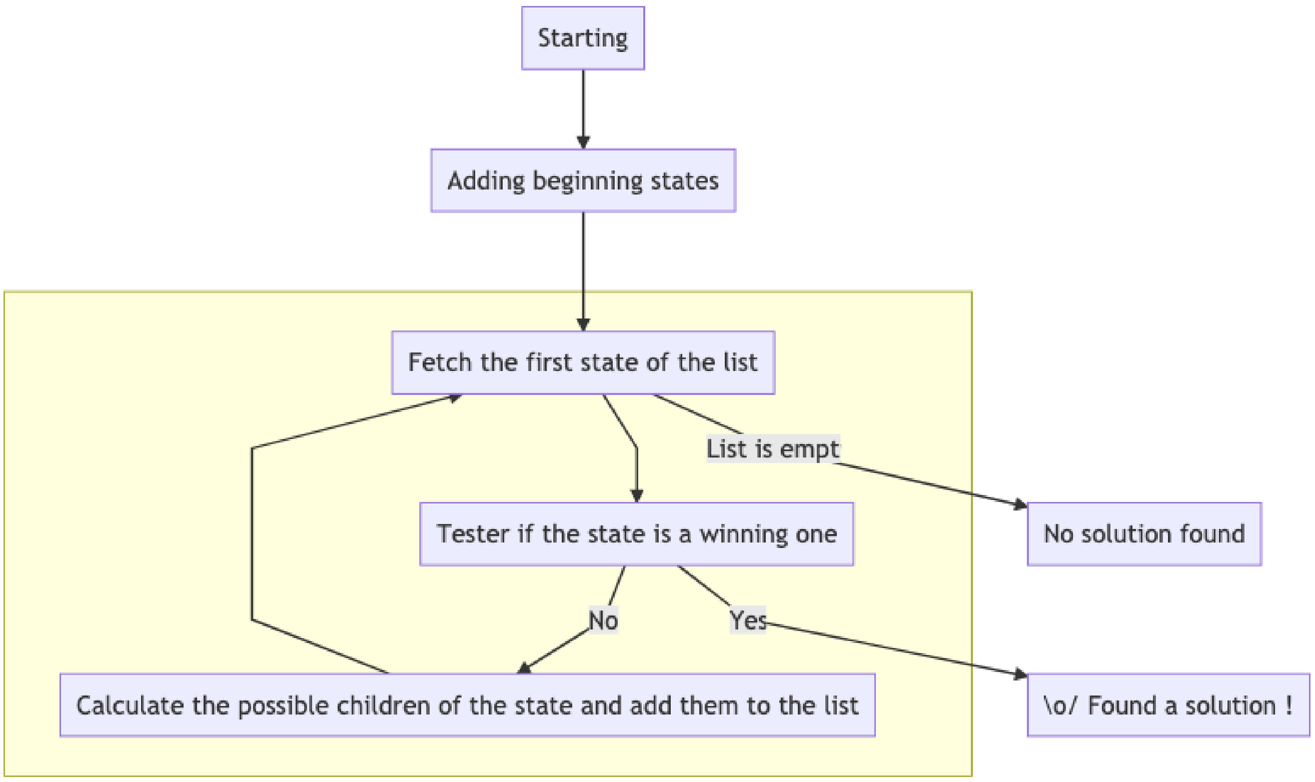
Part 1: reading levels

Baba Is You is a great puzzle game with a more-than-usual complex gameplay since manipulating the game rules is part of the game.

I think it's a good candidate to write a solver for, and to document how I'm doing it.

I'm writing while coding the solver and not as a post-mortem, so expect some bad design and implementation decisions along the way.

As I've explained in a previous article (in French), a puzzle solver rely on a loop used to iterate until a solution is found.



Starting
Adding beginning states
Fetch the first state of the list
List is empt
Tester if the state is a winning one
No solution found
No
Yes
Calculate the possible children of the state and add them to the list
\o/ Found a solution !

The first step is to be able to initialize the loop, which means reading the levels from disk.

Create the project

First thing is to bootstrap the project.

A solver has often complex data types and you need to do much refactoring so a language with static types is great, and a fast language is nice to have to make things faster, so my tool of choice for this case is Java.

So first create the project:

```
mvn archetype:generate \
-DgroupId=net.archiloque.babaisyousolver \
-DartifactId=babaisyousolver \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.4 \
-DinteractiveMode=false
```

Then add the required files (`LICENSE.txt` …) and push it to GitHub.

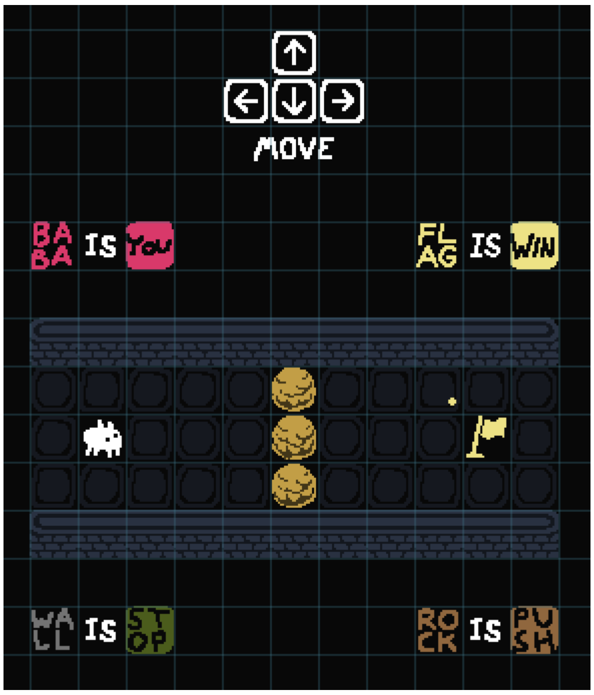
Read levels

For my other solvers, reading the levels relied on a fixed list of elements with an associated character.

For Baba, the number of possible tiles seems large from the levels I've played so I'm thinking of using another strategy:

1. Define all the possible tiles
2. For each level, declare which tiles match with each character, for the tiles to be used in the level

Levels use a square grid, and there is at most one item per tile in the starting position.



The first level

What I'd like the files to be:

levels/00/tiles.txt

```
b Baba
B Baba text
  empty
f flag
F flag text
I is text
P push text
r rock
R rock text
S stop text
w wall
W wall text
! win text
Y you text
```

levels/00/content.txt

```
BIY     FI!

wwwwwwwwwww
     r
 b   r   f
     r
wwwwwwwwwww

WIS     RIP
```

Note: I’d like to rely on emojis, like using □ to represents baba or □ for the flag, but the way macOS deal with emoji don’t make them monospace friendly, so it breaks the visual aspect of the level files and editing them becomes a pain.

I declare all the tiles used by the first level, as a `String` to be referenced by the level, and as an `int` to be referenced by the game:

Tiles.java

```
/**
 * All the tiles.
 * Should be declared in sorted order so the index
 * of the items in {@link Tiles#ALL_STRINGS}
 * matches the values of the int items.
 */
interface Tiles {

  String BABA_STRING = "Baba";
  String BABA_TEXT_STRING = "Baba text";
```

```
    String EMPTY_STRING = "empty";
    String FLAG_STRING = "flag";
    String FLAG_TEXT_STRING = "flag text";
    String IS_TEXT_STRING = "is text";
    String PUSH_TEXT_STRING = "push text";
    String ROCK_STRING = "rock";
    String ROCK_TEXT_STRING = "rock text";
    String STOP_TEXT_STRING = "stop text";
    String WALL_STRING = "wall";
    String WALL_TEXT_STRING = "wall text";
    String WIN_TEXT_STRING = "win text";
    String YOU_TEXT_STRING = "you text";

    String[] ALL_STRINGS = new String[]{
        BABA_STRING,
        BABA_TEXT_STRING,
        EMPTY_STRING,
        FLAG_STRING,
        FLAG_TEXT_STRING,
        IS_TEXT_STRING,
        PUSH_TEXT_STRING,
        ROCK_STRING,
        ROCK_TEXT_STRING,
        STOP_TEXT_STRING,
        WALL_STRING,
        WALL_TEXT_STRING,
        WIN_TEXT_STRING,
        YOU_TEXT_STRING,
    };

    int BABA = 0;
    int BABA_TEXT = BABA + 1;
    int EMPTY = BABA_TEXT + 1;
    int FLAG = EMPTY + 1;
    int FLAG_TEXT = FLAG + 1;
    int IS_TEXT = FLAG_TEXT + 1;
    int PUSH_TEXT = IS_TEXT + 1;
    int ROCK = PUSH_TEXT + 1;
    int ROCK_TEXT = ROCK + 1;
    int STOP_TEXT = ROCK_TEXT + 1;
    int WALL = STOP_TEXT + 1;
    int WALL_TEXT = WALL + 1;
    int WIN_TEXT = WALL_TEXT + 1;
    int YOU_TEXT = WIN_TEXT + 1;
}
```

I use `int`s instead of enum types for this kind of data for two reasons :

- they took less memory, and it makes a big difference when you have to store many levels;
- you can use the `int` as array index to retrieve data instead of using a `Hash`, which makes it much faster.

I have experienced both in my previous solvers.

You loose some type checking this way and makes code a bit less readable but it's an acceptable downside for me.

Perhaps later this file will become a pain to maintain so I'll end up putting the content in a config file and generate the Java code from it but for now I prefer to use a simpler way.

Drafting the API of how to read the level files:

LevelReader1.java

```
final class LevelReader {

  private LevelReader() {
  }

  private static final String TILES_FILES = "tiles.txt";
  private static final String CONTENT_FILES = "content.txt";

  /**
   * Read a level from a directory.
   */
  static @NotNull LevelReaderResult readLevel(
      @NotNull Path levelDirectory)
      throws IOException {
    return readContent(levelDirectory, readTiles(levelDirectory));
  }

  private final static Pattern TILES_REGEX =
      Pattern.compile("^(.{1}) (.+)$");

  /**
   * Read the tiles declaration
   * from the {@link LevelReader#TILES_FILES} file
   */
  static @NotNull Map<Character, Integer> readTiles(
      @NotNull Path levelDirectory) throws IOException {
    // @TODO probably add some code here
    return new HashMap<>();
  }

  /**
```

```
    * Read the file content
    * from the {@link LevelReader#CONTENT_FILES} file
    * relying the declared tiles
    */
   static @NotNull LevelReaderResult readContent(
       @NotNull Path levelDirectory,
       @NotNull Map<Character,
           Integer> tiles) throws IOException {
     // @TODO probably add some code here
     return null;
   }


   static final class LevelReaderResult {

     final int width;

     final int height;

     final @NotNull int[] content;

     LevelReaderResult(
         int width,
         int height,
         @NotNull int[] content) {
       this.width = width;
       this.height = height;
       this.content = content;
     }
   }
 }
```

Calling `LevelReader#readLevel` will return a `LevelReader.LevelReaderResult` containing the data in a ready to use format.

All the data will use arrays instead of arrays of arrays: it simplifies the copying, and makes everything faster.

Representing a position as a single integer can also make things simpler when you need to pass or return info since it use a single value instead of an object that need to be created each time.

To convert a line/column coordinate to a position used in a flat array and back you need to use this code:

```
int position = (indexLine * levelWidth) + indexColumn;

int indexLine = position / levelWidth;
int indexColumn = position % levelWidth;
```

I use JetBrains's annotations extensively to handle nullability in my code, when I code with Idea it's an acceptable substitute of have it build into the language.

Add some code

First the tests (just showing the names):

LevelReaderTest.java

```
public class LevelReaderTest {

  @Test
  public void readTilesOK(){}

  @Test
  public void readTilesFileNotFound(){}

  @Test
  public void readTilesTilesNotFound(){}

  @Test
  public void readTilesInvalidSyntax(){}

  @Test
  public void readContentOK(){}

  @Test
  public void readContentFileNotFound(){}

  @Test
  public void readContentInvalidLineLength(){}

  @Test
  public void readContentUnknownTile(){}
}
```

There's a bunch of tests for error cases because I hate to debug the code when there's an error with the level files so the error are as explicit as possible.

Then fill the content of the reader:

LevelReader.java

```
  /**
   * Read the tiles declaration
   * from the {@link LevelReader#TILES_FILES} file
   */
  static @NotNull Map<Character, Integer> readTiles(
      @NotNull Path levelDirectory
  ) throws IOException {
    Path elementsFile = getFile(levelDirectory, TILES_FILES);
    List<String> tilesContent = Files.readAllLines(elementsFile);

    Map<Character, Integer> result = new HashMap<>();
    for (String tileLine : tilesContent) {
      Matcher m = TILES_REGEX.matcher(tileLine);
      if (!m.find()) {
        throw new IllegalArgumentException(
            "Bad tile declaration [" + tileLine + "]");

      }
      Character character = m.group(1).charAt(0);
      String tile = m.group(2);
      int index = Arrays.binarySearch(Tiles.ALL_STRINGS, tile);
      if (index < 0) {
        throw new IllegalArgumentException(
            "Unknown tile [" + tile + "]");
      }
      result.put(character, index);
    }
    return result;
  }

  /**
   * Read the file content
   * from the {@link LevelReader#CONTENT_FILES} file
   * relying the declared tiles
   */
  static @NotNull LevelReaderResult readContent(
      @NotNull Path levelDirectory,
      @NotNull Map<Character, Integer> tiles
  ) throws IOException {
    Path elementsFile = getFile(levelDirectory, CONTENT_FILES);
    List<String> contentContent = Files.readAllLines(elementsFile);

    int height = contentContent.size();
    int width = contentContent.get(0).length();
    int[] content = new int[height * width];

    for (int lineIndex = 0; lineIndex < height; lineIndex++) {
      String contentLine = contentContent.get(lineIndex);
      if (contentLine.length() != width) {
        throw new IllegalArgumentException(
            "[" +
                contentLine +
```

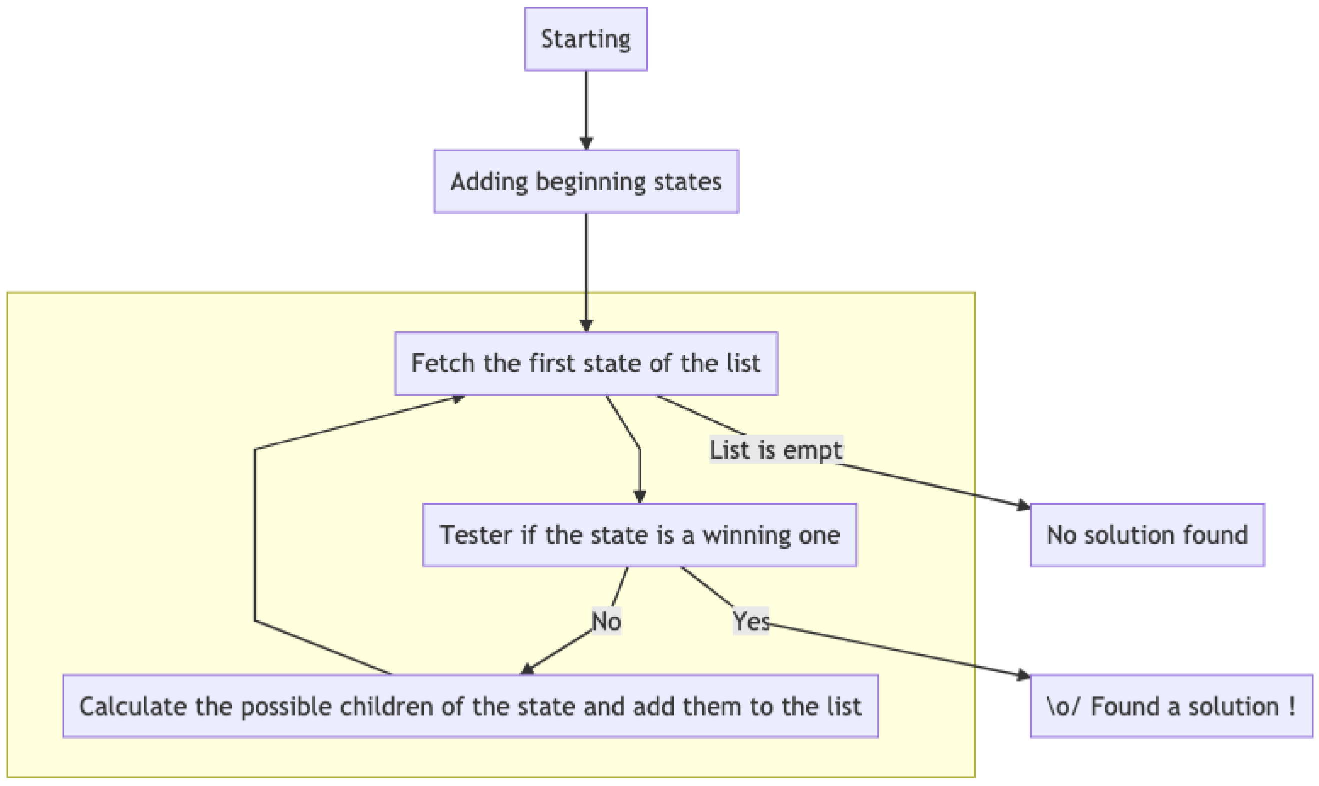
```java
                "] is not " +
                width +
                " characters long at line " +
                lineIndex +
                " of " +
                elementsFile.toAbsolutePath());
    }

    for (int columnIndex = 0; columnIndex < width; columnIndex++) {
      char c = contentLine.charAt(columnIndex);

      if (!tiles.containsKey(c)) {
        throw new IllegalArgumentException(
            "Unknown tile [" +
                c +
                "] at line " +
                lineIndex +
                " of " +
                elementsFile.toAbsolutePath());
      }
      int position = (lineIndex * width) + columnIndex;
      content[position] = tiles.get(c);
    }
  }
  return new LevelReaderResult(width, height, content);
}

private static @NotNull Path getFile(
    @NotNull Path directory,
    @NotNull String fileName
) throws FileNotFoundException {
  Path file = directory.resolve(fileName);
  if (!Files.exists(file)) {
    throw new FileNotFoundException(
        file.toAbsolutePath().toString());
  }
  return file;
}
```

Now the code reads a level from disk and the next part will start implementing the solving logic.

Part 2: plumbing

Following the first part about reading the levels, it's time to do some plumbing.

Before dealing with the solving logic, some plumbing is required to establish the game loop.


Starting
Adding beginning states
Fetch the first state of the list
List is empt
No solution found
Tester if the state is a winning one
No
Yes
Calculate the possible children of the state and add them to the list
\o/ Found a solution !

Most of the code shown here is ported from my previous efforts.

The level

First the `Level` class will contain data related to a game level, at first only the information found in the level read in the first part, then — by experience — I'll add other data that will make the calculations simpler and/or faster.

Level.java

```
class Level {

  final int width;

  final int height;

  final int size;

  final @NotNull int[] content;

  Level(
      int width,
      int height,
      @NotNull int[] content) {
    this.width = width;
    this.height = height;
    this.size = width * height;
    this.content = content;
  }

}
```

Then the `State` will represent a state of the game being solved. It will follow the same path as the `Level`: it starts as a small class with few attributes (the current status of the game and a pointer to the `Level`) and becomes larger as features are added, for example to store the dynamic rules.

State.java

```
class State {

  private final @NotNull Level level;

  /**
   * Probably not the right format, just drafting
   */
  private final @NotNull int[] content;

  State(
      @NotNull Level level,
      @NotNull int[] content) {
    this.level = level;
    this.content = content;
  }

  /**
   * Process the current state
   *
   * @return true if we found a solution
   */
  boolean processState() {
    // @TODO probably add some code here
    return false;
  }
}
```

The list of states

To manage the list of states several possibilities are available, like using a FIFO queue for depth-first search or a stack for breadth-first search.

The choice depends of the type of problem and I don’t have a criteria to choose: it something I tweak along.

My last projetc used a FIFO queue so let’s use one.

FiFoQueue.java

```
final class FiFoQueue<E> {

  private @Nullable FiFoQueue.QueueElement<E> currentElement;

  FiFoQueue() {
  }

  void add(@NotNull E newElement) {
    currentElement =
        new QueueElement<>(newElement, currentElement);
  }

  @Nullable E pop() {
    if (currentElement != null) {
      E element = currentElement.element;
      currentElement = currentElement.next;
      return element;
    } else {
      return null;
    }
  }

  boolean isEmpty() {
    return currentElement == null;
  }

  private final static class QueueElement<E> {

    private final @NotNull E element;

    private final @Nullable FiFoQueue.QueueElement<E> next;

    QueueElement(
        @NotNull E element,
        @Nullable FiFoQueue.QueueElement<E> next) {
      this.element = element;
      this.next = next;
    }
  }

}
```

I use a custom implementation since the code is short and it’s only doing what I need.

We can them add the code related to the state lists in the `Level`.

Level.java

```java
private final @NotNull FiFoQueue<State> states =
    new FiFoQueue<>();

void createInitStates() {
  addState(content);
}

void addState(@NotNull int[] content) {
  states.add(new State(this, content));
}

@Nullable State solve() {
  while (!states.isEmpty()) {
    State state = states.pop();
    if (state.processState()) {
      return state;
    }
  }
  return null;
}
```

Calling `State#processState` will add some new possible `State`s to the list, so the minimal version of the solver loop is now complete.

The entry point

The last part is the entry point of the game.

App.java

```java
/**
 * Entry point
 */
public class App {

  private static final SimpleDateFormat DATE_FORMAT =
      new SimpleDateFormat("yyyy-MM-dd kk:mm:ss.SSS");

  public static void main(String[] args)
      throws IOException {
    if (args.length == 0) {
      throw new IllegalArgumentException(
          "A level path should be specified");
    }
    processLevel(Path.of(args[0]));
  }

  private static void processLevel(
      @NotNull Path path
  ) throws IOException {
    print(path, "Reading level");
    LevelReader.LevelReaderResult levelReaderResult =
        LevelReader.readLevel(path);
    Level level = new Level(
        levelReaderResult.width,
        levelReaderResult.height,
        levelReaderResult.content);
    level.createInitStates();
    print(path, "Solving level");
    long startTime = System.nanoTime();
    State solution = level.solve();
    long stopTime = System.nanoTime();
    String endTime =
        LocalTime.MIN.plusNanos((stopTime - startTime)).
            toString();
    if (solution != null) {
      print(
          path,
          "Solved in " + endTime);
    } else {
      print(
          path,
          "Failed in " + endTime);
    }
  }

  private static void print(
      @NotNull Path path,
      @NotNull String message) {
    System.out.println(
        DATE_FORMAT.format(new Date()) +
            " " +
            path.toAbsolutePath() +
            " " +
            message);
  }

}
```

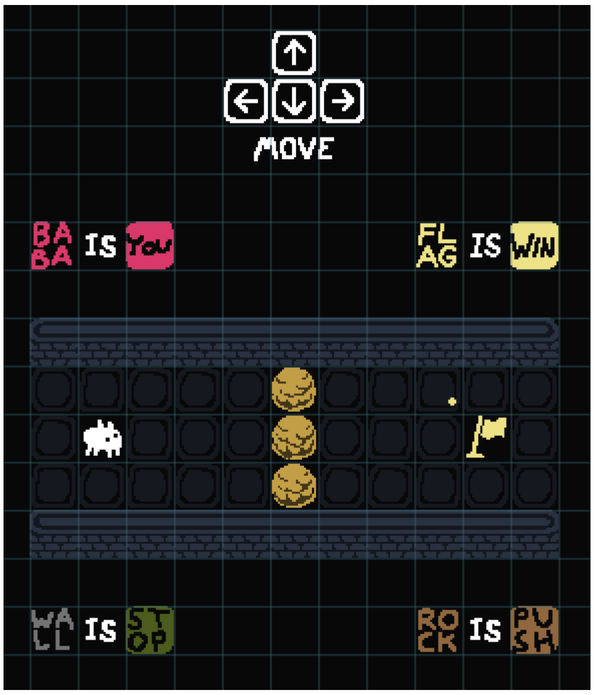
The workflow is now completed, and the program can be run to (kinda) solve the level:

```
java -jar target/babaisyousolver-1.0-SNAPSHOT.jar levels/00
2019-05-08 23:03:06.827 babaisyousolver/levels/00 Reading level
2019-05-08 23:03:06.877 babaisyousolver/levels/00 Solving level
2019-05-08 23:03:06.882 babaisyousolver/levels/00 Failed in 00:00:00.000008930
```

In the next part I'll start adding some logic in the code to make good use of all this plumbing.

Part 3: finding a solution to the first level

Following the second part, I'll deal with the logic to find a solution to the first level.



Here it is

The first level has a feature that will things much simpler: the rules can't be changed.

This means that fixed rules can be enough to solve this level, and that the changing rules features can be dealt with later.

First the algo

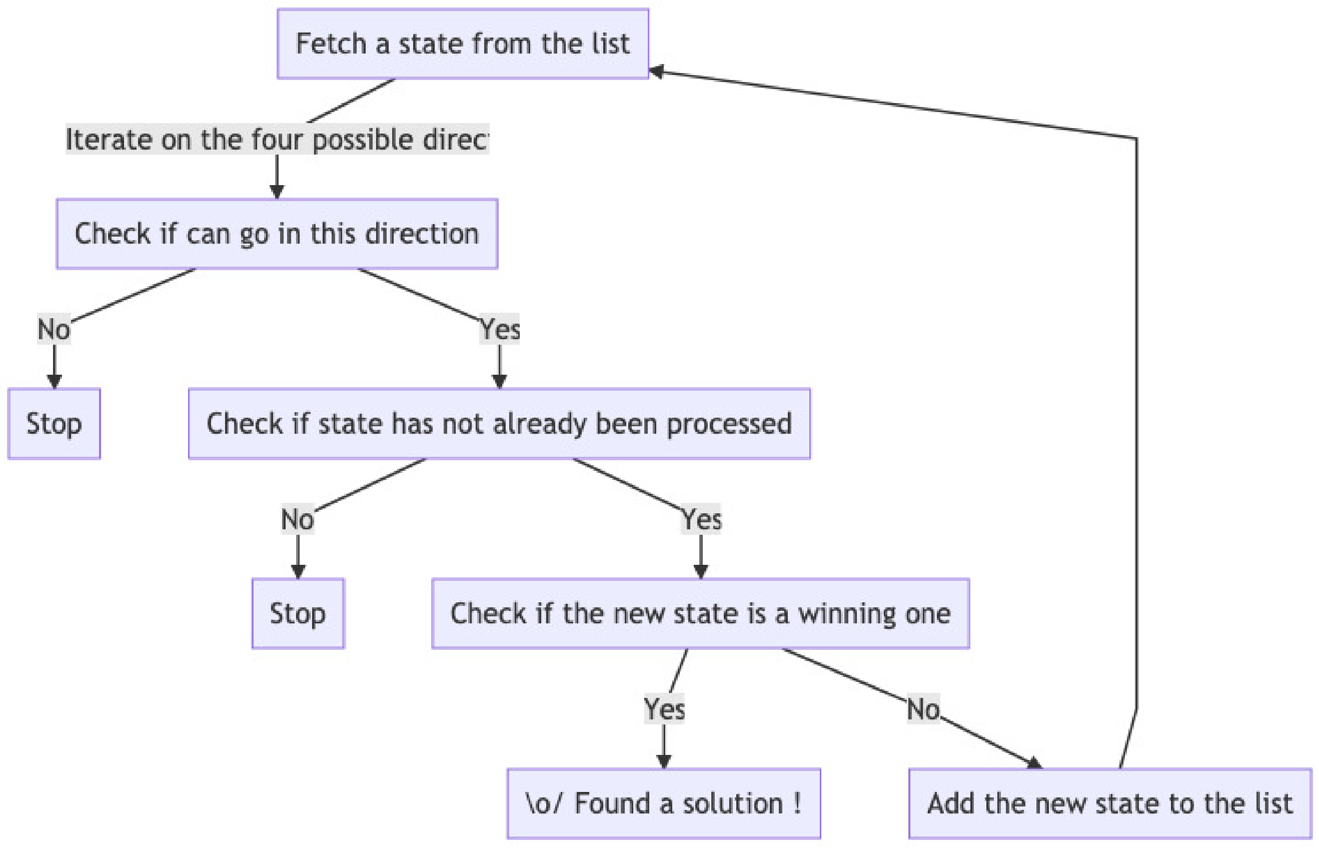
Each turn the solver considers a game state, from it there is *at most* four possible moves: going up, down, right or left.

At most because some direction can be blocked, things can kill you (not in the first level), or you can alter a rule and stop to control anything (not in the first level either).

If you can go in a direction, you may reach a winning state, in this case the the solver stops.

If you don't solve the level, you can add this new state to the list, and go on with the main loop.

There's a catch: Baba is not a "burning bridge" game where you can't move back. Before checking if a new state is interesting, it must be vetted against the previously met cases to avoid doing the same thing again and again.


Fetch a state from the list
Iterate on the four possible direc
Check if can go in this direction
No
Yes
Stop
Check if state has not already been processed
No
Yes
Stop
Check if the new state is a winning one
Yes
No
\o/ Found a solution !
Add the new state to the list

The inner algo

Then some code

First thing is to detect if a move is possible.

In the further levels of the game, “you” can be several items, but in the first level you only control Baba, so first let’s find Baba.

So being possible to go in a direction means only testing if Baba can go in this direction.

First I check if Baba can physically go in this direction from the level point of view.

It requires to be able to specify a direction.

Direction.java

```java
interface Direction {

  byte UP = 0;
  byte DOWN = 1;
  byte LEFT = 2;
  byte RIGHT = 3;

  char[] VISUAL = new char[]{
      '↑', '↓', '←', '→'
  };

}
```

Then the code:

State.java

```java
class State {

  private final @NotNull Level level;

  /**
   * Probably not the right format, just drafting
   */
  private final @NotNull int[] content;

  State(
      @NotNull Level level,
      @NotNull int[] content) {
    this.level = level;
    this.content = content;
  }

  /**
   * Process the current state
   *
   * @return true if we found a solution
   */
  boolean processState() {
    int babaPosition = findBaba();
    int babaLine = babaPosition / level.width;
    int babaColumn = babaPosition % level.width;

    // Up
    if (babaLine > 0) {
      if (tryToGo(babaPosition, Direction.UP)) {
        return true;
      }
    }

    // Down
    if (babaLine < (level.height - 1)) {
      if (tryToGo(babaPosition, Direction.DOWN)) {
        return true;
      }
    }

    // Left
    if (babaColumn > 0) {
      if (tryToGo(babaPosition, Direction.LEFT)) {
        return true;
      }
    }

    // Right
    if (babaColumn < (level.width - 1)) {
      if (tryToGo(babaPosition, Direction.RIGHT)) {
        return true;
      }
    }

    return false;
  }

  boolean tryToGo(int currentPosition, byte direction) {
    // @TODO probably add some code here
    return false;
  }

  /**
   * Find the index of the baba position.
   *
   * @return the position or -1 if not found
   */
  int findBaba() {
    for (int i = 0; i < content.length; i++) {
      if (content[i] == Tiles.BABA) {
        return i;
      }
    }
    return -1;
  }
```

And testing it:

StateAvailableMovementsTest.java

```java
class StateTryToGoTest {

  private static final class LevelToTestTryToGo extends Level {

    private final List<int[]> states = new ArrayList<>();

    LevelToTestTryToGo(
        int width,
        int height,
        @NotNull int[] content) {
      super(width, height, content);
    }

    @Override
    void addState(@NotNull int[] content) {
      states.add(content);
    }
  }

  /**
   * Cases are tested with a level of ?x1
   * Baba is in the first position and tries to go left
   */
  void checkMoveSimple(
      int[] content,
      boolean result,
      @NotNull int[][] possibleNextMoves) {
    LevelToTestTryToGo level = new LevelToTestTryToGo(
        content.length,
        1,
        content);
    State state = new State(level, content);
    assertEquals(
        result,
        state.tryToGo(0, Direction.RIGHT));
    assertEquals(possibleNextMoves.length, level.states.size());
    for (int i = 0; i < possibleNextMoves.length; i++) {
      assertArrayEquals(
          possibleNextMoves[i],
          level.states.get(i));
    }
  }

  @Test
  void testMoveEmpty() {
    checkMoveSimple(
        new int[]{Tiles.BABA, Tiles.EMPTY},
        false,
        new int[][]{new int[]{Tiles.EMPTY, Tiles.BABA}}
    );
  }

  @Test
  void testMoveWall() {
    checkMoveSimple(
        new int[]{Tiles.BABA, Tiles.WALL},
        false,
        new int[0][]
    );
  }

  @Test
  void testMoveFlag() {
    checkMoveSimple(
        new int[]{Tiles.BABA, Tiles.FLAG},
        true,
        new int[0][]
    );
  }
}
```

Then it’s time to deal with the content. The behavior depends of the content of the position Baba wants to move to. The code must do three things: test if the movement is valid, test if the new position is a winning one, and calculate the new board position.

It would be possible to write each step in separate pass but I find it easier to do it in one time because all three steps depends mostly of the kind of elements on the target tile.

Calculating a new board position is done by cloning the existing one and updating the values that need to be updated.

Pushing a rock is more complex case than the others, so I’ll ignore it first:

State.java

```java
boolean tryToGo(int currentPosition, byte direction) {
  int targetPosition = calculatePosition(currentPosition, direction);
  int targetPositionContent = content[targetPosition];

  int[] newContent;
  switch (targetPositionContent) {
    case Tiles.WALL:
      return false;
    case Tiles.EMPTY:
      newContent = content.clone();
      newContent[targetPosition] = Tiles.BABA;
      newContent[currentPosition] = Tiles.EMPTY;
      level.addState(newContent);
      return false;
    case Tiles.ROCK:
      // @TODO implements this
      return false;
    case Tiles.FLAG:
      return true;
    default:
      throw new IllegalArgumentException("" + targetPositionContent);
  }
}

private int calculatePosition(int position, byte direction) {
  switch (direction) {
    case Direction.UP:
      return position - level.width;
    case Direction.DOWN:
      return position + level.width;
    case Direction.LEFT:
      return position - 1;
    case Direction.RIGHT:
      return position + 1;
    default:
      throw new IllegalArgumentException("" + direction);
  }
}
```

And testing it:

StateTryToGoTest.java

```java
class StateTryToGoTest {
```

```java
private static final class LevelToTestTryToGo extends Level {

  private final List<int[]> states = new ArrayList<>();

  LevelToTestTryToGo(
      int width,
      int height,
      @NotNull int[] content) {
    super(width, height, content);
  }

  @Override
  void addState(@NotNull int[] content) {
    states.add(content);
  }
}

/**
 * Simple cases are tested with a level of 2x1
 * Baba is in the first position and tries to go left
 */
void checkMoveSimple(
    int[] content,
    boolean result,
    @NotNull int[][] possibleNextMoves) {
  LevelToTestTryToGo level = new LevelToTestTryToGo(
      2,
      1,
      content);
  State state = new State(level, content);
  assertEquals(
      result,
      state.tryToGo(0, Direction.RIGHT));
  assertEquals(possibleNextMoves.length, level.states.size());
  for (int i = 0; i < possibleNextMoves.length; i++) {
    assertArrayEquals(
        possibleNextMoves[i],
        level.states.get(i));
  }
}

@Test
void testMoveEmpty() {
  checkMoveSimple(
      new int[]{Tiles.BABA, Tiles.EMPTY},
      false,
      new int[][]{new int[]{Tiles.EMPTY, Tiles.BABA}}
    );
  }

  @Test
  void testMoveWall() {
    checkMoveSimple(
        new int[]{Tiles.BABA, Tiles.WALL},
        false,
        new int[0][]
    );
  }

  @Test
  void testMoveFlag() {
    checkMoveSimple(
        new int[]{Tiles.BABA, Tiles.FLAG},
        true,
        new int[0][]
    );
  }
}
```

A solution

The title of this part is “finding *a* solution”, because I’ll first write the code to find a valid solution, with a simplified algo that don’t cover all possible cases, then I’ll update it to make it compliant with the game rules.

I’ll do it this way because the update will add complexity to the behaviors, and I prefer to first reach a state where I solved something.

When moving a rock, the simplest case is to check if

- the rock is not against a border of the level
- there is nothing behind it

In this case Baba can go in this direction and push the rock.

State.java

```
case Tiles.ROCK:
  // did we reach the border of the level ?
  if(!canGoThere(targetPosition, direction)) {
    return false;
  }
  // the position behind  the rock
  int behindTheRockPosition = calculatePosition(targetPosition, direction);
  int behindTheRockPositionContent = content[behindTheRockPosition];
  // it it empty?
  if(behindTheRockPositionContent != Tiles.EMPTY) {
    return false;
  }
  // nice, we build the new content
  newContent = content.clone();
  newContent[targetPosition] = Tiles.BABA;
  newContent[currentPosition] = Tiles.EMPTY;
  newContent[behindTheRockPosition] = Tiles.ROCK;
  level.addState(newContent);
  return false;
```

StateTryToGoTest.java

```
@Test
void testMoveRock() {
  checkMoveSimple(
      new int[]{Tiles.BABA, Tiles.ROCK},
      false,
      new int[0][]
  );

  checkMoveSimple(
      new int[]{Tiles.BABA, Tiles.ROCK, Tiles.ROCK},
      false,
      new int[0][]
  );

  checkMoveSimple(
      new int[]{Tiles.BABA, Tiles.ROCK, Tiles.EMPTY},
      false,
      new int[][]{new int[]{Tiles.EMPTY, Tiles.BABA, Tiles.ROCK}}
  );
}
```

If the program now, it will run indefinitely. Not because of a bug but because of a missing feature: it lacks the ability to remember that a state has already been visited, thus doing the same thing again and again.

Java makes it a bit more complicated than it should do:

Level.java

```java
/**
 * This set will be able to handle duplication of {@link State}
 * The custom {@link Comparator} is required to avoid
 * only comparing the arrays' addresses
 */
private final Set<int[]> pastStates =
  new TreeSet<>(new Comparator<>() {
    @Override
    public int compare(int[] o1, int[] o2) {
      for (int i = 0; i < size; i++) {
        int cmp = o2[i] - o1[i];
        if (cmp != 0) {
          return cmp;
        }
      }
      return 0;
    }
  });

void addState(@NotNull int[] content) {
  if (!pastStates.contains(content)) {
    pastStates.add(content);
    states.add(new State(this, content));
  }
}
```

And … it works, the program can solve the level:

```
java -jar target/babaisyousolver-1.0-SNAPSHOT.jar levels/00
2019-05-08 23:03:06.827 babaisyousolver/levels/00 Reading level
2019-05-08 23:03:06.877 babaisyousolver/levels/00 Solving level
2019-05-08 23:03:06.882 babaisyousolver/levels/00 Solved in 00:00:00.001229643
```

\o/

In the next part we'll have a look on the solution.

Part 4: printing a solution

Following the third part, where I solved the first level, this short part will deal with printing the solution.

If the computer say it, it must be true!

I said I solved the first level, at least it's what the computer told me:

```
java -jar target/babaisyousolver-1.0-SNAPSHOT.jar levels/00
2019-05-08 23:03:06.827 babaisyousolver/levels/00 Reading level
2019-05-08 23:03:06.877 babaisyousolver/levels/00 Solving level
2019-05-08 23:03:06.882 babaisyousolver/levels/00 Solved in 00:00:00.001229643
```

But it would be nice if the code told me *how* it solved it, it would allow me to check if the solution looks reasonable, and it would allow me to use the solution to solve the level.

Which is the goal of the whole thing.

States gonna be stateful

To implement this, a `State` must store the movements that lead to it, which means each `State` must concatenate the movement it adds to a list of previous movements.

State.java

```java
final @NotNull byte[] previousMovements;

State(
    @NotNull Level level,
    @NotNull int[] content,
    @NotNull byte[] movements) {
  this.level = level;
  this.content = content;
  this.previousMovements = movements;
}

boolean tryToGo(
    int currentPosition,
    byte direction) {
  int targetPosition = calculatePosition(currentPosition, direction);
  int targetPositionContent = content[targetPosition];

  int[] newContent;
  switch (targetPositionContent) {
    case Tiles.WALL:
      return false;
    case Tiles.EMPTY:
      newContent = content.clone();
      newContent[targetPosition] = Tiles.BABA;
```

```
        newContent[currentPosition] = Tiles.EMPTY;
        level.addState(newContent, addMovement(direction));
        return false;
      case Tiles.ROCK:
        // did we reach the border of the level ?
        if (!canGoThere(targetPosition, direction)) {
          return false;
        }
        // the position behind  the rock
        int behindTheRockPosition = calculatePosition(targetPosition, direction);
        int behindTheRockPositionContent = content[behindTheRockPosition];
        // it it empty?
        if (behindTheRockPositionContent != Tiles.EMPTY) {
          return false;
        }
        // nice, we build the new content
        newContent = content.clone();
        newContent[targetPosition] = Tiles.BABA;
        newContent[currentPosition] = Tiles.EMPTY;
        newContent[behindTheRockPosition] = Tiles.ROCK;
        level.addState(newContent, addMovement(direction));
        return false;
      case Tiles.FLAG:
        return true;
      default:
        throw new IllegalArgumentException("" + targetPositionContent);
    }
  }

  /**
   * Add a new movement at the end of the array
   */
  private @NotNull byte[] addMovement(byte movement) {
    int previousLength = previousMovements.length;
    byte[] result = new byte[previousLength + 1];
    System.arraycopy(previousMovements, 0, result, 0, previousLength);
    result[previousLength] = movement;
    return result;
  }
```

Then when the solution is found, I write it in the same directory as the level, so any change can be identified with a source control system. The file is deleted before trying to solve the level so a solvinf failure is easy to detect.

If we don't pretty print it, the path would be something like `→→→→→→←←←` which is not easy to read.

App.java

```
  private static final String SOLUTION_FILES = "solution.txt";

  private static void processLevel(
      @NotNull Path path
  ) throws IOException {
    print(path, "Reading level");
    LevelReader.LevelReaderResult levelReaderResult =
        LevelReader.readLevel(path);
    Level level = new Level(
        levelReaderResult.width,
        levelReaderResult.height,
        levelReaderResult.content);
    level.createInitStates();
    Path solutionFile = path.resolve(SOLUTION_FILES);
    Files.deleteIfExists(solutionFile);
    print(path, "Solving level");
    long startTime = System.nanoTime();
    State solution = level.solve();
    long stopTime = System.nanoTime();
    String endTime =
        LocalTime.MIN.plusNanos((stopTime - startTime)).
            toString();
    if (solution != null) {
      print(
          path,
          "Solved in " + endTime);
      writeSolution(solution.previousMovements, solutionFile);
    } else {
      print(
          path,
          "Failed in " + endTime);
    }
  }

  private static void writeSolution(
      @NotNull byte[] solution,
      @NotNull Path solutionPath)
      throws IOException {
    List<String> steps = new ArrayList<>();
    byte currentMovement = -1;
    int numberOfMovesThisWay = 1;
    for (byte c : solution) {
      if (c != currentMovement) {
        if (currentMovement != -1) {
          steps.
              add("" +
                  numberOfMovesThisWay +
                  Direction.VISUAL[currentMovement]);
        }
        currentMovement = c;
```

```
        numberOfMovesThisWay = 1;
      } else {
        numberOfMovesThisWay += 1;
      }
    }
    // complete with last move
      steps.
          add("" +
              numberOfMovesThisWay +
              Direction.VISUAL[currentMovement]);

    String content = String.join(" ", steps);
    Files.write(solutionPath, content.getBytes());
  }
```

The result:

levels/00/solution.txt

```
6→ 3← 1↓ 5→
```

Wait, what?

Huh ? The last step is missing ?!

Which is normal, since the data used is the path to the last `State`, which lacks the last movement done inside the `State`.

The solution is to add the last movement to the past movements, and to return this value as the result, it requires a bunch of modifications but the migration is easy.

State.java

```
/**
 * Process the current state
 *
 * @return the path to the solution if found, null if not.
 */
@Nullable byte[] processState() {
    int babaPosition = findBaba();
    int babaLine = babaPosition / level.width;
    int babaColumn = babaPosition % level.width;

    byte[] result;
    // Up
    if (babaLine > 0) {
      result = tryToGo(babaPosition, Direction.UP);
      if (result != null) {
        return result;
        }
      }

      // same for other directions

      return null;
    }

    @Nullable byte[] tryToGo(
        int currentPosition,
        byte direction) {
      int targetPosition = calculatePosition(currentPosition, direction);
      int targetPositionContent = content[targetPosition];

      int[] newContent;
      switch (targetPositionContent) {
        case Tiles.WALL:
          return null;
        case Tiles.EMPTY:
          newContent = content.clone();
          newContent[targetPosition] = Tiles.BABA;
          newContent[currentPosition] = Tiles.EMPTY;
          level.addState(newContent, addMovement(direction));
          return null;
        case Tiles.ROCK:
          // did we reach the border of the level ?
          if (!canGoThere(targetPosition, direction)) {
            return null;
          }
          // the position behind  the rock
          int behindTheRockPosition = calculatePosition(targetPosition, direction);
          int behindTheRockPositionContent = content[behindTheRockPosition];
          // it it empty?
          if (behindTheRockPositionContent != Tiles.EMPTY) {
            return null;
          }
          // nice, we build the new content
          newContent = content.clone();
          newContent[targetPosition] = Tiles.BABA;
          newContent[currentPosition] = Tiles.EMPTY;
          newContent[behindTheRockPosition] = Tiles.ROCK;
          level.addState(newContent, addMovement(direction));
          return null;
        case Tiles.FLAG:
          return addMovement(direction);
        default:
          throw new IllegalArgumentException("" + targetPositionContent);
      }
    }
```

In `Level` the value is propagated.

Level.java

```java
@Nullable char[] solve() {
    char[] result;
    while (!states.isEmpty()) {
      State state = states.pop();
      result = state.processState();
      if (result != null) {
        return result;
      }
    }
    return null;
  }
```

And in `App`, this value is printed.

App.java

```java
byte[] solution = level.solve();
long stopTime = System.nanoTime();
String endTime =
  LocalTime.MIN.plusNanos((stopTime - startTime)).
      toString();
if (solution != null) {
  print(
    path,
    "Solved in " + endTime);
  writeSolution(solution, solutionFile);
```

Which prints the right solution

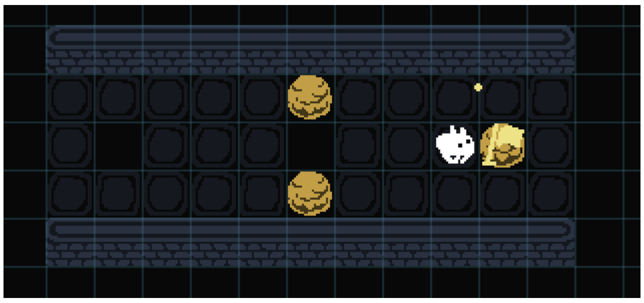
levels/00/solution.txt

```
6→ 3← 1↓ 5→ 1↑
```

Part 5: adding missing behaviors

Following the fourth part, where I coded what was missing to display a solution, I'll now improve the behavior of the solver to make it closer to the game.

Two elements on one place



This is it

Pushing several rocks