# ECE2810J — Data Structures and Algorithms

*Programming Assignment 3*

Instructor: Yutong Ban
— UM-SJTU-JI (Fall 2024)

**Notes**

- Submission: on JOJ
- Due at Dec 1st, 2024, by 23:59

## 1 Introduction

In this assignment, you are tasked with implementing a C++ solution to a simplified version of the "sokoban" game, where the objective is to push specific boxes to designated target positions on a grid. The challenge lies in devising an efficient strategy to navigate the grid, avoid obstacles, and ensure that all required boxes reach their target locations. Your program should be able to handle various configurations of boxes, walls, and goals, ensuring the correct boxes are pushed to their intended destinations. This project is C++ only, no other language is allowed.

## 2 Programming Assignment

### 2.1 Read Map

The map is designed as a grid where each cell is either a walkable path, a wall, or a point of interest (like the starting point or a box). The first line of the file defines the dimensions of the map in terms of columns and rows. The rest of the file specifies the content of each cell in the grid, using a series of symbols to represent different elements:

- `.` - **Path**: This represents a walkable path where the player can move freely. Boxes can also be pushed across this type of cell.
- `#` - **Wall**: This represents an impassable barrier. Neither the player nor the boxes can pass through these cells.
- `S` - **Starting Point**: This is where the player begins. Each map should contain exactly one `S` character.
- `B` - **Box's Original Position**: The initial position of a box.
- `T` - **Target Position for Boxes**: The destination cell for a box.
- `R` - **Box at Target Position**: This indicates that a box has been successfully pushed to its target position.

To ensure a valid and challenging map, the following constraints must be met:

(1) The outermost boundary of the map must be composed entirely of walls (`#`) to prevent the player or boxes from exiting the map area.

(2) Each map must contain exactly one starting point (`S`).

(3) For test cases of the first part on JOJ, the number of boxes (`B`) will be less or equal to 8 and the result of rows times columns should be less or equal to 800.

(4) For test cases released on Canvas, it can be very complicated.

The player can move in four cardinal directions:

- Up

- Down
- Left
- Right

The player can push a box if it is adjacent in one of these directions, and if the cell behind the box in that direction is a path (`.`). Boxes can only be pushed. In other words, they cannot be pulled. Only one box can be pushed at a time, and the player must move to the former box's position after pushing it.
Below is an example of a map configuration file. In this example, the map size is specified on the first line as `11 10`, indicating 10 rows and 11 columns. The outer boundary is composed of walls (`#`), while inner cells define paths, the starting point, box positions, and target positions.

```
11 10
###########
#..#..#...#
#..B....#.#
#..##.#...#
##.#...B###
##.#.....##
#SBBTT#..##
#..#.TT.###
####..#####
###########
```

## 2.2 Search routes

Help the player to find the shortest route to push all the boxes to the expected place. Please notice that it is possible that there are multiple boxes and multiple target positions. The player should push all the boxes to the target positions.

## 2.3 Show the route

The task requires the program to display the player's route on the map, detailing each movement step-by-step. Each step should be represented by a character indicating the direction moved:

- `U` for up
- `R` for right
- `L` for left
- `D` for down

For instance, given the map configuration shown, the route may look like:

`DRURRUURUURRDDDLLLDDRRRURULLLDLDRULLLUUULUURDRRURDDDDDLLLUUULURRRURDDDDULD`

This format will be used as the return value of the `solve` function provided in the template code.

**Bonus Requirements**

(1) Display the map, player, boxes, and other elements at each step using the same format as the input map. Store all the map state in a file, ensuring it is clear and easy to read.

(2) Use a graphical interface to visually represent each step, showing the map, player, boxes, and other elements in their updated positions. The output should be displayed as an animation, with each step shown for a brief period before transitioning to the next step.

### 2.4 Unsolvable Maps

In some cases, a map configuration may be unsolvable due to the positioning of walls or boxes, creating an impasse that prevents the player from reaching all targets. Such configurations are referred to as "invalid maps." This occurs when one or more boxes are placed in positions from which they cannot be moved to their designated target cells.
A map is considered invalid if:

- There is more than one starting point or no starting point.
- There are more boxes than target positions.
- A box is surrounded by walls or other boxes in a way that makes it immovable.
- Any box is placed in a corner or against a wall where it has no path to the target position.

The following example demonstrates a typical invalid map layout:

```
11 10
###########
#..#..#...#
#.#B....#.#
#..##.#...#
##.#....###
##.#.....##
#S..TT#..##
#..#.TT.###
####..#####
###########
```

In this map: The box (`B`) is located next to walls on two sides, which prevents it from being moved toward the target position (`P`).
When the program encounters such a configuration, it should recognize the situation as unsolvable and output a message as "No solution!" The program should then terminate without attempting to find a solution or display a route.

## 3 Input / Output

As mentioned above, the input map is stored in a file. To test your program, you can use the following command to read the map from a file:

```
./main < inputMap.txt
```

For submission on JOJ, the output (i.e. the return value of the function) should either be "No solution!" or the route to push all the boxes to the target positions.
If you have implemented the first bonus requirements, you should first display the steps in the format mentioned above, and then display the map at each step. We highly recommend you to store the map at all steps in a file. For example, if the output is stored in a file named `output.txt`, you can use the following command:

```
./main < inputMap.txt > output.txt
```

If you have implemented the second bonus requirements, you should display the map in a graphical interface.

# 4 Submission

(1) For part 1 and part 3 on JOJ, submit your source code in one file named `sokoban.hpp`. Your source code should include a function named `solve` and a function named `read_map`. The test cases in part 1 and part 3 are similar, but part 3 has stricter time and memory limitations.

(2) For part 2 on JOJ, put the detailed route output in the `answers` part in `sokoban.hpp`. The test cases will be released on canvas, which may take a long time for your program to run. Please notice that part 2 and bonus will be graded if and only if you can pass 70 percent of the test cases in part 1.

(3) Submit the source code for part 2 on Canvas along with a corresponding `Makefile` to compile the source code. (i.e. The code you used to get answer for part 2 on JOJ. ) A sample `Makefile` is provided on Canvas, which you may modify as needed. Your code submitted on Canvas should be able to compile successfully and output the detailed route with the corresponding input file.

(4) (Optional) If you have implemented the first bonus requirement, you also need to submit the output files on Canvas. The naming format should match the original file name. For example, if the input file is named `big_1.in`, the output file should be named `big_1_detail.out`. Failure to follow the correct naming format will result in an invalid submission. You only need to submit the output file for one test case.

(5) (Optional) If you have implemented the second bonus requirement. You may come to the TA's office hours to demonstrate your program. The due date will be the same. The TA will give you a score based on the demonstration.

## Basic Tips for Optimizing Your Program

### Deadlocks

- What is a freeze deadlock?
- List common freeze deadlocks.
- Analyze the pros and cons of adding more deadlock detections.

### Efficiency

- How much memory do you need to store a "state"?
- With 100 bits, how many states can you store?
- If visiting one state takes 1ns, how much time will it take to visit through all possible states represented by 100 bits?
- How much information do you need to backtrace a path?

### C++ Features

- Is `std::pair` fast or slow?
- How many temporary objects have you created in each iteration? Are they really necessary?

# 5 Implementation Requirements and Restrictions

## 5.1 Requirements

- You must make sure that your code compiles successfully on a Linux operating system with `g++` and the options `-std=c++1z -Wall -Wextra -Werror -pedantic -Wno-unused-result -Wconversion -Wvla`.
- You can use any header file defined in the `C++17` standard. You can use cppreference as a reference.

## 5.2 Memory Leak

You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call valgrind, which can tell whether you have any memory leaks. (You need to install valgrind first if your system does not have this program.) The command to check memory leak is:

`valgrind --leak-check=full <COMMAND>`

You should replace `<COMMAND>` with the actual command you use to issue the program under testing. For example, if you want to check whether running program

`./main < input.txt`

causes memory leak, then `<COMMAND>` should be "`./main < input.txt`". Thus, the command will be

`valgrind --leak-check=full ./main < input.txt`

## 5.3 Code Quality

We will use cpplint, cppcheck to check your code quality. If your code does not pass these checks, you will lose some points.
For cpplint, you can use the following command to check your code quality:

```
cpplint --linelength=120 --filter=-legal,-readability/casting,-whitespace,-runtime/printf,
-runtime/threadsafe_fn,-readability/todo,-build/include_subdir,-build/header_guard *.h *.cpp
```

For cppcheck, only things related to **error** will lead to deduction. You can use the following command to check your code quality:

```
cppcheck --enable=all *.cpp
```

# 6 Grading

## 6.1 Correctness

We will use the test cases on JOJ and Canvas to verify the correctness of your code. This section will have a total score of 100 points. Your score will be calculated as follows:

$$\min\left(100, \frac{\text{score in part 1} + \text{score in part 2} + \text{score in part 3}}{14}\right)$$

Please notice that the total points on JOJ will be 2150 with 1000 points for part 1, 400 points for part 2, and 750 points for part 3. You only need to get 1400 points on JOJ to get full score in this section.

### 6.2 Code Quality

Bad code quality will lead to a deduction of up to 20 points, which will be applied directly to your final score.

### 6.3 Bonus

For the first bonus requirement, you can get up to 10 points. For the second bonus requirement, you can get up to 30 points. The total score for the whole project will not exceed 130 points.

## 7 Late Policy

Same as the course policy.

## 8 Honor Code

Please **DO NOT** violate honor code.
Some behaviors that are considered as cheating:

- Reading another student's answer/code, including keeping a copy of another student's answer/code
- Copying another student's answer/code, in whole or in part
- Having someone else write part of your assignment
- Using test cases of another student
- Testing your code with another one's account
- Post your code to github
- Using any kind of AI tools to solve the problem

## 9 Acknowledgement

This programming assignment is designed based on ENGR1510J Lab 3 and ENGR1510J Project 3.