Theory of Computer Games (Fall 2020)
Homework 1

NTU CSIE

Due: 14:20 (UTC+8), November 5, 2020

Outline

Original Game - Sokoban

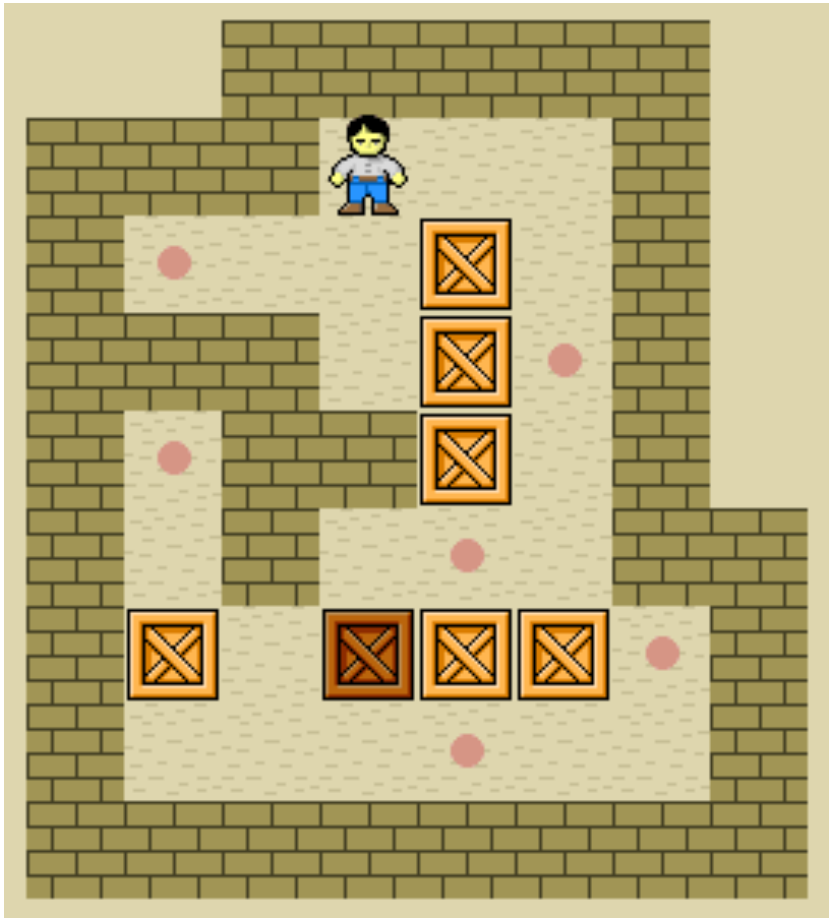
Elements

- **Box**: an object that can be pushed into target
- **Target**: a floor square marked as storage location
- **Wall**: a stationary object that can't be moved

Rules

1. Player acts as a warehouse keeper located at an empty floor square.
2. Player may move UP/DOWN/RIGHT/LEFT to an adjacent empty square.
3. Player may push a box by walking up to it and pushing it to an adjacent empty square beyond.
4. The objective is to place all boxes at storage locations.

Sokoban - Illustration



Source: en.wikipedia.org/wiki/Sokoban

Figure: Sokoban illustration

Sokoban Variant - Sokoboru

SokoBoru

SokoBoru is a variant of Sokoban with additional element: BALL.

- **Objective**: place all balls inside boxes.

Elements

- **Box**: an object that can be pushed
- **Ball**: a **sliding object** that can be pushed
- **Wall**: a stationary object that can't be moved

Sokoboru - Rules

Rules

1. A box can be pushed to an adjacent square.
2. A ball is a **sliding object** that can be pushed and **slide** until it hits another object.
3. If a ball is pushed towards an empty box, then it will be **trapped** inside the box (can't be pulled out).
4. If a ball is pushed towards a non-empty box/wall/another ball, then it will stop at the adjacent square of the solid object.
5. There are equal number of balls and boxes.
6. Both empty and non-empty boxes are movable.

Sokoboru - Penalty

Penalty

- walk $= 1$
- pushing **ball** or **empty box** $= 1$
- pushing **non-empty box** $= 2$

Storyline

Role

- **Virus Buster**: A secret agent who has been specially trained to detect and capture viruses quickly.

Elements

- **Virus**: An extremely contagious virus that need to be contained immediately.
- **Isolation Box**: An isolation box to contain the virus.

Mission

Protect humanity by containing all viruses inside isolation boxes.

Play Sokoboru Yourself

- Under directory `hw1`, type the command
  `$ make`
  to build the execution file, `sokoboru`
- Type
  `$ ./sokoboru -i inputfile [-o outputfile] [-s n]`
  to start the game from stage *n* in puzzle file `inputfile` and record the solution in file `outputfile`
- To play with tiny puzzle, execute
  `$ ./sokoboru -i testdata/tiny.in`

Outline

Requirements

HW Requirements

1. Implement an optimal Sokoboru solver.
2. Design a Sokoboru puzzle.
3. Analyze the performance of different search algorithms.

Part I: Sokoboru Solver

Basic Requirements

1. Write a program to read puzzles from standard input and write solutions to standard output
2. Thread limit: 2 threads
3. Time limit: 60 seconds (for each puzzle file)
4. Memory limit: 4GB

Puzzles

We provide 3 puzzle files under directory `testdata`, namely:

- `tiny.in` + hidden `tiny.in`
- `small.in` + hidden `small.in`
- `medium.in` + hidden `medium.in`
- hidden `large.in`

Input Format

Puzzle File

- Each puzzle file contains 10 test cases .
- The first line of each test case contains 2 positive integers, n and m, representing the height and width respectively.
  - $1 \leq n, m \leq 15$
  - $nm \leq 50$
  - $1 \leq ball, box \leq 15$
- The following n lines describe the initial board. Each line is a string composed of #, @, O, $, *, - of length m.

Input Format (Cont'd)

Legend

- #: Wall
- @: Player
- O: Ball
- $: Box
- *: Ball inside box
- -: Empty square/floor

Output Format

Solution File

- There are 2 output lines for each test case.
- The first line is a non-negative integer k which represents the penalty.
- The second line is a string composed of $\wedge$, $\vee$, $<$, $>$.
  - $\wedge$: UP
  - $\vee$: DOWN
  - $<$: LEFT
  - $>$: RIGHT
- There should be no infeasible action in your solver's output.

Verifier

Verifier

You can verify whether an input and/or output file is valid by executing:

- `$ ./verifier -i inputfile`
  check if `inputfile` is a valid puzzle file.
- `$ ./verifier -o outputfile`
  check if `outputfile` is a valid solution file.
- `$ ./verifier -i inputfile -o outputfile`
  check if `inputfile` and `outputfile` are valid, then check if `outputfile` solves `inputfile`

under `hw1` directory.

Part II: Design Your Own Puzzle

Basic Requirements

1. Provide one valid Sokoboru puzzle and its solution named `[student_id].in` and `[student_id].out` respectively.
2. The puzzle and solution should be validated by verifier.
3. Analyze the complexity of the puzzle in your report.

Part III: Algorithm Analysis

Report Structure

Your report should include but not limited to:

1. Implementation
   1. How to compile and run your code in linux. (If TA has difficulty compiling your code, you will be required to demonstrate the process.)
   2. What algorithm and heuristic you implemented.
2. Experiments
   1. Comparison between different search algorithms (execution time, memory, number of nodes, etc.)
3. Discussion
   1. The complexity of Sokoboru puzzle.
   2. The complexity of different search algorithms.
   3. The complexity of the puzzle you designed.

Outline

Submission

- Directory hierarchy:
  - student_id // e.g. r08922166 (lowercase)
    - Makefile // make your code
    - src // a folder contains all your codes
    - student_id.in // your puzzle
    - student_id.out // your solution
    - report.pdf // your report
- Compress your folder into a zip file and submit to https://www.csie.ntu.edu.tw/~tcg/2020/hw1.php.
- Due to server limitation, the file size is restricted to 2 MB.
- If your program has a pattern database greater than 2 MB, you can simply upload the code that generates the pattern database. The database should be generated within 30 minutes.

Grading Policy

1. Sokoboru solver (8 points + 2 bonus)
   - Solve `tiny.in` within 60 seconds (1 point)
   - Solve `small.in` within 60 seconds (2 points)
   - Solve `medium.in` within 60 seconds (2 points + 1 point if you solve it within 1 second)
   - Solve hidden `large.in` within 60 seconds (3 points + 1 point if you solve it within 1 second)
   - If your solver fails to solve a puzzle file (every stage) correctly within the time limit, you won't get any point.
   - Suppose your solver produces a K-penalty solution for a single test case, and the optimal penalty is K_0, you'll get $0.1 + 0.1\lfloor \frac{K_0}{K} \rfloor$ point. (10 test cases per puzzle file)

Grading Policy (Cont'd)

2. Puzzle creation (2 points)
   - Your puzzle and solution files should pass `verifier` to get the 2 points.
   - If your puzzle is considered complex enough, you'll get an extra bonus.
3. Report (5 points)
   - Your score will be evaluated with TA's HNN (human neural network) model.