Expected to build an AI algorithm to solve Sokoban.

Game Rules

As explained in the wikipedia entry, The game is played on a board of squares, where each square is a floor or a wall. Some floor squares contain boxes, and some floor squares are marked as storage locations.

The player is confined to the board and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can move a box by walking up to it and pushing it to the square beyond. Boxes cannot be pulled, and they cannot be pushed to squares with walls or other boxes.

The number of boxes equals the number of storage locations.

The puzzle is solved when all boxes are placed at storage locations.

The Algorithm

A configuration of the Sokoban game is specified by the location of walls, boxes, storage areas and player. A configuration is called a state. The Sokoban Graph $G=\langle V,E\rangle$ is implicitly defined. The vertex set V is defined as all the possible configurations (states), and the edges E connecting two vertexes are defined by the legal movements (right, left, up, down). All edges have a weight of 1.

Your task is to find the path traversing the Sokoban Graph from the initial state (vertex) leading to a state (vertex) where all the boxes are located on a storage area. The best path is the shortest path. A path is a sequence of movements. You are going to use Dijkstra to find the shortest path first, along with some game-specific optimizations to speed up your algorithm.

When the AI solver is called (Algorithm 1), it should explore all possible paths (sequence of move actions) following a Dijktsra strategy, until a path solving the game is found. Note that we use transposition tables to avoid duplicate states in the search. If a state was already expanded (popped from the priority queue), we will not include it again in the priority queue (line 23 in Algorithm 1). We will also ignore states where the player doesn't move as a result of moving towards an adjacent wall, or a box which cannot move (line 18). We will finally avoid states where boxes are located in a corner (line 18, see file

utils.h). The algorithm should return the best solution found. This path will then be executed by the game engine if the option play_solution was used as an argument.

Algorithm 1 AI Sokoban Algorithm

```
1: procedure FindSolution(start, showSolution)
2:     n ← CreateInitNode(start)
3:     PushPriorityQueue(n)
4:     exploredTable ← create empty array
5:     hashTable ← initialize Hash Table for duplicate detection
6:     while priorityQueue ≠ empty do
7:         n ← PriorityQueue.pop()
8:         exploredNodes ← exploredNodes + 1
9:         exploredTable ← record n in array
10:        if WinningCondition(n) then                          ▷ Found a solution
11:            solution ← saveSolution(n)
12:            SolutionSize ← n.depth
13:            break
14:        end if
15:        for each move action a ∈ {Left, Right, Up, Down} do
16:            playerMoved ← applyAction(n, newNode, a)          ▷ Create Child newNode
17:            generatedNodes ← generatedNodes + 1
18:            if playerMoved is false or SimpleCornerDeadlock(newNode) then
19:                free(newNode)
20:                continue
21:            end if
22:            flatMap ← flattenMap(newNode)                     ▷ converts the map into a 1D map
23:            if flatMap is a duplicate then                    ▷ Check duplicates in HashTable
24:                duplicatedNodes ← duplicatedNodes + 1
25:                free(newNode)
26:                continue
27:            end if
28:            hashTable ← InsertHashTable(flatMap)
29:            PriorityQueue.push(newNode)
30:        end for
31:    end while
32: end procedure
```

You might have multiple paths leading to a solution. Your algorithm should consider the possible actions in the following order: left, right, up or down.

Make sure you manage the memory well. When you finish running the algorithm, you have to free all the nodes from the memory, otherwise you will have memory leaks. You will notice that the algorithm can run out of memory fairly fast after expanding millions nodes.

The applyAction creates a new node,

- that points to the parent,

- updates the state with the action chosen,

- updates the depth of the node,

- updates the priority (used by the priority queue) of the node to be the negative node's depth d (if the node is the dth step of the path, then its priority is -d). This ensures the expansion of the shortest paths first, as the priority queue provided is a max heap,

- updates the action used to create the node.

Check the file utils.h, hash_table.h, priority_queue.h where you'll find many of the functions in the algorithm already implemented. Other useful functions are located directly in the file ai.c , which is the only file you need to edit to write your algorithm inside the function findSolution . Look for the comment FILL IN THE GRAPH ALGORITHM . All the files are in the folder src/ai/.

Deliverables

Deliverable 1 - Dijkstra Solver source code

The source code for your solver must be written in C. Obviously, your source code is expected to compile and execute flawlessly using the following makefile command: make generating an executable called sokoban. Remember to compile using the optimization flag gcc -O3 for doing your experiments, it will run twice as quickly as compiling with the debugging flag gcc -g (see Makefile, and change the CC variable accordingly). Please submit your makefile with gcc -g option, as our scripts need this flag for testing. Your program must not be compiled under any flags that prevents it from working under gdb or valgrind.

Your implementation should work well over the first 3 layouts, but it will not be expected to find a solution to any layout, as it may exceed the available RAM in your computer before finding a solution. Feel free to explore maps given in the folder maps_suites. They are taken from the official benchmarks of sokoban. All you have to do is to copy and paste a single map into a new file, and then call it with your solver.

Deadlock Detection (optimizations)

The simplest way to improve the code is by improving the deadlock (http://sokobano.de/wiki/index.php?title=Deadlocks) detection implemented (SimpleCornerDeadlock) in Algorithm 1-line 18. You can find the C implementation inside utils.c:235. Implement at least 1 deadlock detection in the link above.

If you do any optimizations & changes to the Data Structures & deadlock detection improvement, please make sure to explain concisely what it is that you implemented, why you chose that optimization, how it affects the performance (show number of expanded nodes before and after the optimization for a set of test maps), and where the code of the optimizations is located. This explanation should be included in the file located at the root of the basecode: README.md. Please make sure that your solver still returns the optimal solution.

Maps & Solution formats

Puzzle File Format

We adapted the sokoban file reader to support the following specification, but we don't support comments specified by %.

#	Wall
	Space
.	Goal
@	Sokoban
$	Box
+	Sokoban on Goal
*	Box on Goal
%	Comment Line

Solution File Format

solutions returned by the solver follow this format

l	Move Left
r	Move Right
u	Move Up
d	Move Down
L	Push Left
R	Push Right
U	Push Up
D	Push Down

You can load your map and solution into the JS Visualiser.

JS Visualiser

For the visualiser (https://henrykautz.com/sokoban/Sokoban.html) to work, puzzles must be rectangular. If your map is not rectangular, just filled it in with walls instead of empty spaces.

The Code Base

You are given a base code. You can compile the code and play with the keyboard (arrows). You are going to have to program your solver in the file ai.c. Look at the file main.c (main function) to know which function is called to call the AI algorithm.

You are given the structure of a node, the state, a max-heap (priority queue) and a hashtable implementation to check for duplicate states efficiently (line 23 in the Algorithm 1). Look into the utils.* files to know about the functions you can call to apply an action to update a game state. All relevant files are located in the folder src/ai/.

You are free to change any file, but make sure the command line options remain the same.

Input

You can play the game with the keyboard by executing

./sokoban <map>

where map points to the file containing the sokoban problem to solve.

In order to execute your AI solver use the following command:

./sokoban -s <map> play_solution

Where -s calls your algorithm. play_solution is optional, if typed in as an argument, the program will play the solution found by your algorithm once it finishes. All the options can be found if you use option -h:

$./sokoban -h

USAGE

./sokoban <-s> map <play_solution>

DESCRIPTION

Arguments within <> are optional

-s calls the AI solver

play_solution animates the solution found by the AI solver

for example:

./sokoban -s test_maps/test_map2 play_solution

Will run the 2nd map expanding and will play the solution found.

Output

Your solver will print into an solution.txt file the following information:

Solution

Number of expanded nodes.

Number of generated nodes.

Number of duplicated nodes.

Solutions length

Number of nodes expanded per second.

Total search time, in seconds.

For example, the output of your solver ./sokoban -s test_maps/test_map2 could be:

SOLUTION:

rrrrrrdrdLLLLLLLLullluRRRRRRRururRRRRRRRRRR

STATS:

Expanded nodes: 978745

Generated nodes: 3914976

Duplicated nodes: 2288345

Solution Length: 43

Expanded/seconds: 244506

Time (seconds): 4.002942

Expanded/Second is computed by dividing the total number of expanded nodes by the time it took to solve the game. A node is expanded if it was popped out from the priority queue, and a node is generated if it was created using the applyAction function. This code is already provided.