Some notes on complete search techniques

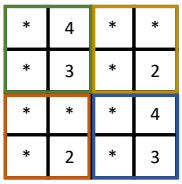
November 3, 2021

The contents of this document are quite tedious and if you don't have experience with these kinds of problems it might seem hard at first. We will go through these kinds of problems also in the revision sessions.

1 A solved example

In this section, we will see two ways to approach a search problem.
Task: Write a program that solves a 4×4 sudoku puzzle.
Definition: A sudoku puzzle is solved if all the following conditions are satisfied:

1. Each column contains each digit 1 to 4 exactly once.
2. Each row contains each digit 1 to 4 exactly once.
3. Each of the following four squares contains each digit $1-4$ exactly once.



Modelling: We have to decide how to represent the state of a game. A reasonable representation for the board is as a nested list of cells. Each cell can either contain a number $1-4$ or be empty, so we define a custom type. (There are other valid ways for doing this)

```
type cell = Val of int | Empty;;
type board = cell list list;;
```

Helper functions for the board: We will now define some helper functions for the board, such as checking if the board configuration is correct and setting a cell of the board. This will be used later on. (Don't spend too much time on these. There are several other ways that you could have implemented these and the important parts are below.)

```
let rec take i = function
  | [] -> []
  | x::xs ->
     if i > 0 then x :: take (i - 1) xs
     else [];;

let rec drop i = function
  | [] -> []
  | x::xs ->
     if i > 0 then drop (i-1) xs
     else x::xs;;

let rec nth n l =
  match n, l with
  | _, [] -> failwith "Empty list"
  | 0, x::_ -> x
  | _, _::xs -> nth (n-1) xs;;
```

```
type cell = Val of int | Empty;;
type board = cell list list;;

(* Checks if the givne list contains at most once every item from 1 to 4. *)
let rec is_perm_aux = function
  [], _ -> true
| Empty::xs, _ -> is_perm_aux (xs, Empty)
| v::xs, p -> (p <> v) && is_perm_aux (xs, v);;
let is_perm x = is_perm_aux (List.sort compare x, Empty);;

let check_rows board = List.for_all is_perm board;;

let get_cols board = List.map (fun x -> List.map (nth x) board) [0;1;2;3];;
let check_cols board = List.for_all is_perm (get_cols board);;

let get board i j = nth i (nth j board);;

(* Returns the square with a left top-most corner at (i, j) *)
let get_square board i j = List.map (fun (dx, dy) -> get board (i+dx) (j+dy))
    [(0,0);(1,0);(0,1);(1,1)];;
(* Returns all possible squares. *)
let get_squares board = List.map (fun (x, y) -> get_square board x y)
    [(0,0);(2,0);(0,2);(2,2)];;

let check_squares board = List.for_all is_perm (get_squares board);;

(* Returns if the board is valid. *)
let is_valid board = check_rows board && check_cols board && check_squares board;;

(* Counts the number of empty cells in the board *)
let count_empty board = List.length (List.filter (fun x -> x = Empty) (List.flatten
    board));;
```

Approach 1

We begin with the first approach for these kinds of problems. Notice that given an assignment for the empty cells we can use the helper function `is_valid` to check if this assignment solves the problem. *How many possible assignments are there?* Each empty cell can take 4 possible values, so there are $4^{\#\text{empty cells}}$. So, we just generate all possible assignments, substitute the empty cells in the board and check if the board is in a valid configuration. In more detail,

1. Write a function that applies the assignment on the board configuration.

```
(* Replaces as many Empty as there are. *)
let rec replace_in_row rr xx = match xx, rr with
| [], _ -> []
| Empty::xs, r::rem -> (Val r)::replace_in_row rem xs
| x::xs,_ -> x::replace_in_row rr xs;;

(* Undoes List.flatten, creating the 4x4 board. *)
let rec unflatten = function
 [] -> []
|xs -> let pref = take 4 xs and suf = drop 4 xs in
     pref::unflatten suf;;

let rec replace_in_board board assignments = unflatten (replace_in_row
    assignments (List.flatten board));;
```

2. Write a function that checks if the new board is valid (this was done in the previous section).

3. Generate all possible assignments.

```
let rec generate_all_seqs len =
   if len = 0 then [[]]
```

```
  else let smaller_seqs = generate_all_seqs (len - 1) in
     List.flatten (List.map (fun x -> List.map (fun y -> x::y)
         smaller_seqs) [1; 2; 3; 4]);;
```

4. Apply all assignments to the boards and search for one that is valid.

```
let rec exists p = function
 [] -> None
| (x::xs) -> if p x then Some x else exists p xs;;


let find_solution board =
  match exists (fun assigment -> is_valid (replace_in_board board
      assigment))
       (generate_all_seqs (count_empty board))
  with
    Some x -> Some (unflatten (replace_in_row x (List.flatten board)))
  | _ -> None;;
```

5. Check that it produces a correct solution on two examples:

```
find_solution [
 [Empty; Empty; Val 2; Val 3];
 [Val 3; Empty; Empty; Val 4];
 [Val 2; Empty; Val 3; Empty];
 [Val 1; Empty; Empty; Empty]] =
Some [
[Val 4; Val 1; Val 2; Val 3];
[Val 3; Val 2; Val 1; Val 4];
[Val 2; Val 4; Val 3; Val 1];
[Val 1; Val 3; Val 4; Val 2]];;

find_solution [
 [Empty; Val 4; Empty; Empty];
 [Empty; Val 3; Empty; Val 2];
 [Empty; Empty; Empty; Val 4];
 [Empty; Val 2; Empty; Val 3]] =
Some [
 [Val 2; Val 4; Val 3; Val 1];
 [Val 1; Val 3; Val 4; Val 2];
 [Val 3; Val 1; Val 2; Val 4];
 [Val 4; Val 2; Val 1; Val 3]];;
```

Approach 2

The previous approach is wasteful for a few reasons:

1. It first generates all possible assignments and then tries them. This means that even if say the second assignment solved the sudoku, it would still have to examine all of them.
2. It uses a lot of memory.
3. It tries a lot of assignments which are rejected for the same reason. For example, in the example configuration it will try all possible assignments with 4 in the first empty cell, even if none of these could work.

So the second approach works by incrementally trying to fill in the values in the board. If there is no violation for the current assignment of values, then we incrementally proceed with the assignment of the rest of the values. If there is a violation, then we try the next candidate value for that position. In more detail,

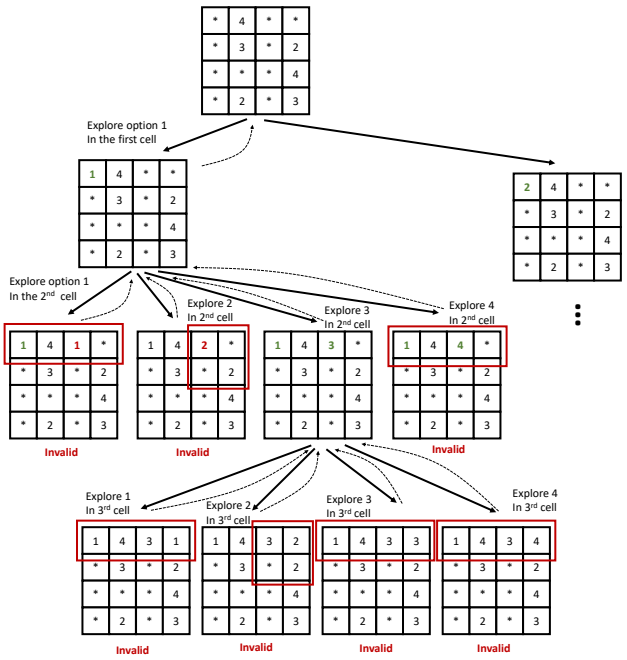
1. Change the `replace` function, so that it fills in the first empty cell.

```
(* Replaces the first occurrence of an empty. *)
let rec replace_first_in_row v xx = match xx with
| [] -> []
| Empty::xs -> (Val v)::xs
| x::xs-> x::replace_first_in_row v xs;;

let replace_first_in_board board v = unflatten (replace_first_in_row v
    (List.flatten board));;
```

2. Write a function that takes a board and tries all four possible values one by one until a valid assignment is found. It then recurses on that. If the recursive call returns an invalid board, then it backtracks and changes the value of the current cell. Note that the nested function `explore_options` tries all four possible options.

```
let rec explore board cnt =
  if cnt = 0 then Some board
  else let rec explore_options cur =
     if cur = 5 then None
     else let new_board = replace_first_in_board board cur in
        if is_valid new_board then
           let ans = explore new_board (cnt-1) in
           if ans = None then explore_options (cur+1)
           else ans
        else explore_options (cur+1)
     in explore_options 1;;

let find_solution board = explore board (count_empty board);;
```

Part of the backtracking can be seen in the following diagram



Exercise 1 [Extending the sudoku solver]

(a) Extend the second approach sudoku solver for 9×9 boards.

(b) Extend the first approach sudoku solver for 9×9 board. How do the running times of the two compare?

(c) Describe how you would make the check in the second approach more efficient.

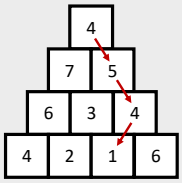
(d) How could you make the first approach more efficient using lazy lists?

(e) A friend of yours wants to make your sudoku solver (in the second approach) slow. Explain how

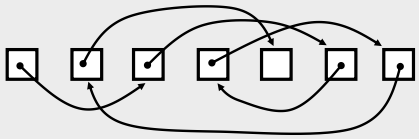
you would modify your code to make it harder for her/him.

For each of the following exercises, it suffices to implement at most one. For the rest, just explain how you would solve each using the first and the second approaches.

Exercise 2 [Sum paths in a triangle] You are given a triangle of positive integers represented as a nested list of integers and an integer k. Write an OCaml function that checks if there is a path from the top of the triangle to the bottom (can only move from an integer to one of the two directly below), so that the entries sum to k. For example, given the triangle `[[4];[7;5];[6;3;4];[4;2;1;6]]` and $k = 15$, the correct path is shown in red.



Exercise 3 [Hopping cover] There are n lily pads in a row. A frog ponders whether starting from the first lily pad it is possible to perform a sequence of jumps given lengths, so as to visit each lily pad only once. For example, given the lengths `[2,3,2,3,5,3]` it is possible to visit each lily pad only once, using the following sequence:



Write an OCaml function that takes a list of n integers and has to determine if with this sequence of jumps.

Exercise 4 [Knight cover] A knight in chess has at most 6 possible positions from a given cell on a checkboard. You are given a starting cell on a 5x5 board and you want to find the sequence of moves so that the knight visits every cell exactly once.

Exercise 5 [Jugs (Optional) (+++)]

(a) You are given three jugs with capacities 5, 8 and 12 litres. Can you measure exactly 6 litres if you are only allowed to a) completely fill in a jug, b) completely empty a jug into another jug and c) move liquid from one to another until the destination has no more capacity?

(b) How does this problem differ from the ones above?

(c) Write an OCaml function that takes three jug capacities and a target value and outputs a sequence of jug states (liquid in each jug).

Further Reading 1 Chapter 10 from Problems on Algorithms.

2 Combinatorial two-player games

More details will be given in the revision class.
Relevant past papers:

- **[2020P1Q2]**
- **[2018P1Q2]**
- **[2017P1Q1]**

- **[2011P1Q1]**
- **[2008P1Q6]**

Further Reading 2 Pages 1 to 10 from Ferguson's book.

Exercise 6 [Noughts and Crosses (Optional) (+++)] Attempt **[1997P1Q5]**. [*Do not spend too much time, think about the high level steps.*]