Assignment 0 — 2012

Engi-6892. Theodore S. Norvell

Solution, 2012

Parts (a) and (b) are an exercise in carefully explaining algorithms. Use pseudocode and clear English as appropriate. You can look in the text book for examples of pseudo code or at a document on my notation that I will post. You will by no means be marked on the efficiency of your algorithms, however you should make an effort to make your answer to part (b) more efficient than your answer to part (a).

(a) Design an algorithm to solve the following problem. Describe the algorithm as clearly and carefully as you can.

Problem: Unrestricted Jigsaw
Input: A set of n^2 square pieces p_0, p_1, $\cdots$, p_{n^2-1}. Each piece is coloured with four colours: p_i.north, p_i.west, p_i.south, p_i.east.
Output: An arrangement of the pieces into an n by n grid so that:

- if two pieces are horizontally adjacent, the one to the left has a east colour equal to the west colour of the one on the right; and
- if two pieces are vertically adjacent, the one above has a south colour equal to the north colour of the one below;

If no such arrangement is possible, a message to that effect.

An example input would be the following set of pieces.[0]

Solution

The following solution uses backtracking.

Call the procedure search (see Fig. 0) passing in the set of all n^2 pieces, a grid variable that hold a 2 D array of pieces, and a boolean variable. On returning from the procedure, if the boolean variable is true, then the grid variable holds a solution. Otherwise there is no solution.

A piece p fits at position (i, j) of the *grid* iff

- either $i = 0$ or $grid(i-1, j)$.south $= p$.north and
- either $j = 0$ or $grid(i, j-1)$.east $= p$.west

[0] For no extra credit you can try putting these pieces into a 3 by 3 grid. I'll post a solution.

```
proc search( var remainingPeices : Set[Piece], var grid : {0,..n} × {0,..n} → Piece, var success :
  Bool )
  if remainingPeices = ∅ then success := true
  else
      val i := (n² − |remainingPeices|) div n
      val j := (n² − |remainingPeices|) mod n
      for p ← remainingPeices do
          if p fits at position (i, j) of the grid then
              grid(i, j) := p
              remainingPeices := remainingPeices − {p}
              search(remainingPeices, grid, success )
              remainingPeices := remainingPeices ∪ {p}
              if success then return end if
          end if
      end for
      success := false
  end if
end proc
```

Figure 0: Procedure search

(b) Design an algorithm to solve the following problem. Describe the algorithm as clearly and carefully as you can.

Problem: Restricted Jigsaw.

Input: A set of n^2 square pieces p_0, p_1, $\cdots$, p_{n^2-1}. Each piece is coloured with four colours: p_i.north, p_i.west, p_i.south, p_i.east.

Precondition: You may assume that each colour is used at most twice as an east or west colour and at most twice as a north or south colour.[1] Furthermore there are n colours that appear only once as a north colour and never as a south colour, n colours that appear only once as a south colour and never as a north colour; n colours that appear once as an east colour and never as a west colour, and n colours that appear once as a west colour and never as an east colour.[2]

[1] Using set notation we can express the restriction by saying that for each colour c, $|E_c \cup W_c| \leq 2$ and $|N_c \cup S_c|$, where

$$\begin{aligned} E_c &= \{i \in \{0,..n^2\} \mid c = p_i.\text{east}\} \\ W_c &= \{i \in \{0,..n^2\} \mid c = p_i.\text{west}\} \\ N_c &= \{i \in \{0,..n^2\} \mid c = p_i.\text{north}\} \\ S_c &= \{i \in \{0,..n^2\} \mid c = p_i.\text{south}\}. \end{aligned}$$

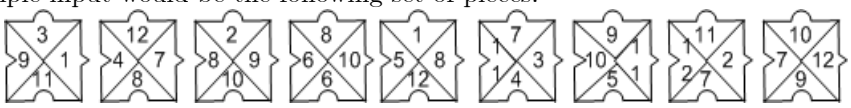
[2] Let's call these colours, respectively, north edge colours, south edge colours, etc. This condition makes it easy

Output: Either an arrangement of the pieces into an n by n grid so that:

- if two pieces are horizontally adjacent, the one to the left has a east colour equal to the west colour of the one on the right; and
- if two pieces are vertically adjacent, the one above has a south colour equal to the north colour of the one below;

or, if no such arrangement is possible, a message to that effect.

An example input would be the following set of pieces.



(Note. Clearly your algorithm for part (a) will also work for part (b). Try to design an algorithm for part (b) that takes advantage of the restriction on the input.)

Solution

From the preceding we can see that the colours can be classified as follows.

- There are exactly n colours that appear as north colours but not south colours. Call these north edge colour.
- Likewise for the other directions, so we have exactly n south edge colours, exactly n east edge colours, and exactly n west edge colour.
- Any piece with a north edge colour on its north side can only go on the north edge (i.e. row 0 of the grid).
- Likewise for the other colours.
- For a piece to fit in the north-west corner, it must have a north edge colour on its north side and a west edge colour on its west side. Conversely a piece like that can only go in the north-west corner (i.e. at $(0, 0)$). So unless there is exactly one such piece, the puzzle can not be solved. Likewise for the other corners.

We can execute the following algorithm

0. Start with an empty $n \times n$ grid.
1. Determine which colours are north-edge colours. Likewise for west-edge colours.
2. North-west corner
   (a) If there no piece that has a north-edge colour on its north and a west-edge colour on its west, there is no solution, quit. Otherwise, place such a piece at location $grid(0, 0)$
3. Western edge
   (a) For i from $[1, ..n - 1]$

to identify edge and corner pieces. The north west corner (if there is one) will have a north edge colour as its north colour and a west edge colour as its west colour, and so on.

i. Find a piece with a north colour equal to $grid(i-1,0).south$. If there is none there is no solution, stop.

ii. If there is one, it must be unique. Place it at $grid(i,0)$.

4. The rest

   (a) For i from $[0,..n]$

      i. For j from $[1,..n]$

         A. Find a piece that has its west colour equal to $grid(i,j-1).east$. If there is none, there is no solution, stop.

         B. Otherwise, let p be the piece found, check that either $i = 0$ or $i > 0$ and $grid(i-1,j).south = p.north$. If not, there is no solution, stop.

         C. Place p at $grid(i,j)$.

If the algorithm runs to the end, there is a solution and it is in *grid*.

Briefly answer each of the following questions as best you can. Keep in mind that my point in asking you the following questions is to get you thinking about the kinds of issues that this course is about. At this point I have *no* expectation that you will come up with "correct" answers, although that would be a happy outcome. My expectation is that you will give these questions some careful thought and write something brief and cogent.

(c) We need quick algorithms. But how can we determine whether an algorithm is quick or not without implementing it and running it on all possible inputs?[3]

SolutionI see two approaches to this problem

- **Reason about it.** Often the time taken by an algorithm does not depend on the exact input, but only on its size. In this case we can work out how many times each operation will be executed as a function of the input size. Multiplying each by the time required (on a given computer) and adding we can get an accurate time for the algorithm as a function of input size. If the time taken depends on more than just the input size, we can focus on the worst case or on typical cases.

- **Test it.** Another approach is to implement the algorithm and test it on sample inputs that are somehow typical. From the results of these tests, we may be able to extrapolate to larger inputs than we have tried.

Both of these approaches are limited to a given computer (or finite set of computers). Luckily there are no magic instructions: anything that can be done in a fixed amount of time one computer will take a fixed amount of time on any other computer. For example the CDC-6600 (at the request of the NSA) could compute the population count (number of 1 bits) of a 60 bit word with one instruction. But on a machine that did not have a population count instruction, this operation can be emulated in a fixed number of instructions. Thus information gathered using one general purpose computer is generally applicable to other computers.

[3] And note that implementing these algorithms and running them on all possible inputs is hardly feasible, given that there are an infinite number of possible inputs.

In order to abstract away from the particulars of the implementation and the computer it is run on, we can look at what kind of a function the algorithm's worst-case time function is. I.e. how the time grows as the input size increases: linearly, quadratically, cubically, exponentially, and so on.

In the second part of the course, we will look at algorithm complexity and in particular how the complexity class of an algorithm's time function can be found.

(d) Assuming each "basic operation"[4] takes 1ns, *estimate* how long each of your algorithms could take (in the worst case) for an input of 100 pieces. Is there something profoundly different about the two algorithms?

Solution

For problem (a): The structure of the algorithm is essentially loops within loops within loops. See Fig. 1.The depth of the loops is 100 (for 100 pieces). The loops iterate for up to 100, 99, 98, times. But there are a couple of issues. First are the if commands; these mean that the loops may not execute. Second is the fact that once success is set to true, all the loops stop. Ok, but, we only need to consider the worst case and the worst case for this algorithm happens when there are 99 all-blue pieces and one all-red pieces which is the last one to be considered. So the number of times the algorithm tries to fit the red pieces into the last slot is at least $99 \times 98 \times \cdots \times 1$. So there are at least 99! operations being done. Ignoring all the other operations, this will take around $99! = 9.332\,6 \times 10^{155}$ns, or

$$\frac{9.332\,6 \times 10^{155}}{10^9 \times 60 \times 60 \times 24 \times 365.25 \times 1000} = 2.957\,3 \times 10^{136}$$

millennia. (Of course this is an underestimate, as I ignored lots of options.)

[For interest sake, let's consider the expected time for a more realistic puzzle where say 4 colours are roughly evenly distributed what happens. Well the chance that a piece fits is 1 for the first piece, about 1/4 for other north and west edge pieces and about 1/16 for most pieces. Then we might be looking more at some multiple of

$$\begin{aligned}
&100 \times \frac{99}{4} \times \frac{98}{4} \times \frac{97}{4} \times \frac{96}{4} \times \frac{95}{4} \times \frac{94}{4} \times \frac{93}{4} \times \frac{92}{4} \times \frac{91}{4} \\
&\times \frac{90}{4} \times \frac{89}{16} \times \frac{88}{16} \times \frac{87}{16} \times \frac{86}{16} \times \frac{85}{16} \times \frac{84}{16} \times \frac{83}{16} \times \frac{82}{16} \times \frac{81}{16} \times \\
&\cdots \\
&\times \frac{10}{4} \times \frac{9}{16} \times \frac{8}{16} \times \frac{7}{16} \times \frac{6}{16} \times \frac{5}{16} \times \frac{4}{16} \times \frac{3}{16} \times \frac{2}{16} \times \frac{1}{16}
\end{aligned}$$

operations. Which gives (some multiple of) 7.87×10^{22} times the age of the universe. For 10 colours, with a roughly random distribution, the typical spot can only be filled by one piece. But there are still 100 choices for the north-west corner and a 1/10 chance that any given piece will fit on the north or west edges. Just considering these two edges gives a number that is (some multiple of) several millennia.

$$100 \times \frac{99!}{80! \times 10^{19}} = 1.304 \times 10^{20}$$

[4] As part of your answer, you may want to clarify which operations you consider basic operations.

]

How about problem (b). Most of the work will be in step 4. Well here we have a triply nested loops. If the innermost loop goes through every piece, then it checks 90 pieces the first time (in the worst case), 89 the second and so on. This makes 4095 checks. If each check takes 10 basic operations[5], that's 41 μs. There will be other operations, but none will take nearly as much time as this inner loop.

(e) The programs have to be correct. How can we determine whether an algorithm is correct for all possible inputs without implementing it and testing it on all possible inputs?[6]

As with trying to determine the time taken we can take two approaches

- **Reason about it.** When we devise an algorithm or write a computer program we usually have some reason for writing down the steps we do. Experience is important, but usually we are solving a problem that is not exactly like any problem we've solved before. Computer engineers do not just memorize a bunch of algorithms in school and find themselves at a loss when confronted by a problem they haven't seen before. Rather they apply a combination of creativity, experience, and reason to find a solution. If we apply reason, we should be able to arrive at a correct solution. And yet bugs still occur. Thus it seems that applying reason is not enough —or should not be enough— to convince others or even ourselves that we haven't made a mistake. If we can record our reasoning, then we can check it and others —including computers— can check it too.

- **Test it.** As the question itself points out, testing has its limitations. We can typically not test every possible input, and when the system is nondeterministic —as happens routinely in concurrent systems— even testing every possible input may not be sufficient to show that a program is correct. Despite these caveats, testing is extremely important. We should test software on typical cases, extreme cases, and very large inputs. As one student suggested last year, randomly generated inputs can be very good for detecting errors. These have the advantage of being easy to generate. They have the disadvantage that, for such inputs, it can be difficult to determine what the correct output is. But, short of exhaustive testing of a deterministic system, testing is of limited use in ensuring correctness. As Edger Dijkstra put it

  Program testing can be used to show the presence of bugs, but never to show their absence!

  Of course showing the presence of bugs is a valuable service. But if I'm going to put my life in the hands of a piece of software, I'd like to have their absence shown.

The first part of the course looks at how we can record our reasoning and check it.

(f) Do your algorithms embody patterns that we might be able to apply to other problems?

[5] By basic operation I mean the sort of thing that can be done with one instruction in a typical computer, e.g., one fetch from memory, one store to memory, one comparison, one branch and so on.

[6] Again this strategy is not possible in principle, owing to the infinite number of possible inputs. Furthermore, even if we give a finite selection of inputs, there is also the compounding problem of having to determine whether the algorithm has output the correct answer in each case.

The first algorithm is a backtracking search. As the time analysis shows, this is a pattern that can easily take a lot of time. However, for small instances, it can be an effective way to solve problems. In some cases, such as the jigsaw puzzle, we actually don't know of any methods that are significantly better.

The second algorithm is an example of an iterative algorithm that builds a solution by building onto a solution for a smaller problem. That is we find a solution for filling in the first k squares, by first finding a solution for the first $k-1$ squares. Such a solution is sometimes called a greedy solution. When a greedy solution can be found, it is obviously better than a backtracking solution.

```
for p ← remainingPeices do
        if p fits at position (i, j) of the grid then
       grid(i, j) := p
         remainingPeices := remainingPeices − {p}
        for p ← remainingPeices do
        if p fits at position (i, j) of the grid then
             grid(i, j) := p
             remainingPeices := remainingPeices − {p}
             ⋮
                  for p ← remainingPeices do
                       if p fits at position (i, j) of the grid then
                                grid(i, j) := p
                                remainingPeices := remainingPeices − {p}
                                success := true
                                remainingPeices := remainingPeices ∪ {p}
                                if success then goto 99 end if
                           end if
                  end for
                  success := false
                  99:
             ⋮
             remainingPeices := remainingPeices ∪ {p}
             if success then goto 1 end if
        end if
   end for
        success := false
   1:
        remainingPeices := remainingPeices ∪ {p}
       if success then goto 0 end if
         end if
  end for
  success := false
0:
```

Figure 1: The structure of the search