CSE373: Data Structures and Algorithms

Algorithm Design Paradigms

Steve Tanimoto
Autumn 2016

This lecture material represents the work of multiple instructors at the University of Washington. Thank you to all who have contributed!

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Consider a large set of possible solutions, storing solutions to subproblems to avoid repeated computation
  - Fibonnaci with "memoizing", string alignment, all-pairs minimum-cost paths
- Backtracking
  - A clever form of exhaustive search

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Consider a large set of possible solutions, storing solutions to subproblems to avoid repeated computation
  - Fibonnaci with "memoizing", string alignment, all-pairs minimum-cost paths
- Backtracking
  - A clever form of exhaustive search

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- **Dynamic Programming**
  - **Consider a large set of possible solutions, storing solutions to subproblems to avoid repeated computation**
  - **Fibonnaci with "memoizing", string alignment, all-pairs minimum-cost paths**
- Backtracking
  - A clever form of exhaustive search

Dynamic Programming: Idea

- Divide a bigger problem into many smaller subproblems
- If the number of subproblems grows exponentially, a recursive solution may have an exponential running time ☹
- Dynamic programming to the rescue! ☺
- Often an individual subproblem may occur many times!
  - Store the results of subproblems in a table and re-use them instead of recomputing them
  - Technique called memoization

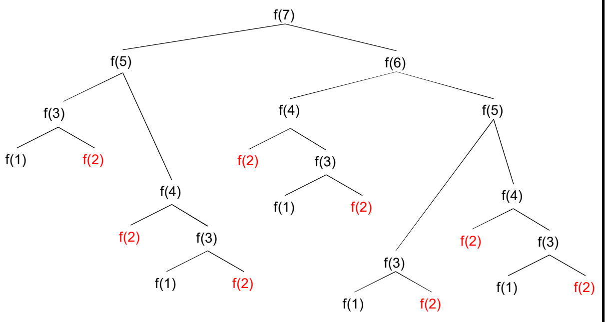
Fibonacci Sequence: Recursive

- The fibonacci sequence is a very famous number sequence
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- The next number is found by adding up the two numbers before it.
- Recursive solution:

```
fib(int n) {
  if (n == 1 || n == 2) {
    return 1
  }
  return fib(n – 2) + fib(n – 1)
}
```

- Exponential running time!
  - A lot of repeated computation

Repeated computation



Fibonacci Sequence: memoized

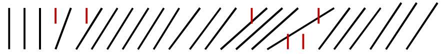
```
fib(int n):
  results = Map()  # Empty mapping container.
  results.put(1, 1)
  results.put(2, 1)
  return fibHelper(n, results)

fibHelper(int n, Map results):
  if (!results.contains(n)):
    results.put(n, fibHelper(n-2)+fibHelper(n-1))
  return results.get(n)
```

Now each call of `fib(x)` only gets computed once for each x!

Another Application of Dynamic Programming: The String Alignment Problem

- Given 2 strings, find a best alignment of them.

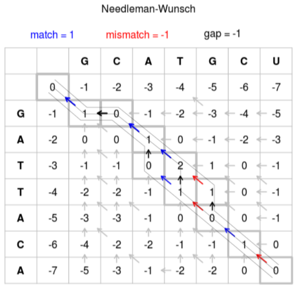
s = THEESE SEAMS TOO BEE STRENG



t = THIS SEEMS TO BE A STRING

- aligned to a character: | 1 if matches, -1 if different.
- aligned to a gap: | -1 for gap (on either top or bottom).
- score = sum of the individual alignment scores.

Construct a Scoring Matrix



Needleman-Wunsch

match = 1 mismatch = -1 gap = -1

		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

courtesy of https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm

Building the Matrix (using D.P.)

- Initialize the matrix by giving the top row and left column, as shown.
- Loop through the remaining cells, always working in a "corner" where the entries to the left and above are already defined.
- Compute the new value as the max of three possible cases:
  - match character on the top to the gap: take the score from the left and above and add gap cost (-1)
  - match character on the bottom (left in the matrix) to the gap: take the score from above and add gap cost (-1)
  - match character on the top to character on the bottom (left in the matrix): take the score from above-left (diagonally adjacent), and add the character match score (1 if characters are the same, -1 if they are different).
- At each cell, indicate where the value came from (point to one of the three cells, depending on how the max turned out.)

Backtracing to Get the Solution (D.P.)

- Start at the lower-right corner of the matrix.
- Follow the arrows (the markers that indicate where each cell's value came from).
- Reverse the resulting path to get an indication of the best alignment (and/or the longest common subsequence of the two strings).

- Time requirement: $\Theta(m \cdot n)$, where m and n are the lengths of the input strings.
- This is much better than a brute force algorithm that computes all possible alignments and then finds the one with the highest score. That would take time in $\Omega(2^{min(m,n)})$, which is at least exponential in the length of the shorter string.

Sample Applications of String Alignment

- Error correction in search queries.
- DNA sequence analysis (compare patient's DNA segment to a well-studied gene variation.
- 3D (depth) image from a stereo pair of images. (Each row of pixels from a left-eye image must be aligned with a row of pixels from a right-eye image before depth disparity values can be computed.)
- Computer analysis of musical themes and variations.
- Speech recognition at the phoneme-to-word level.

Comments

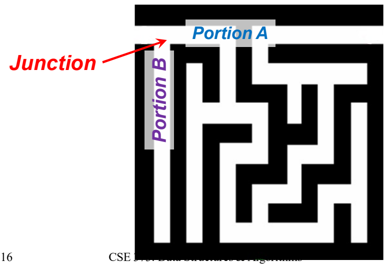
- Dynamic programming relies on working “from the bottom up” and saving the results of solving simpler problems
  - These solutions to simpler problems are then used to compute the solution to more complex problems
- Dynamic programming solutions can often be quite complex and tricky
- Dynamic programming is used for optimization problems, especially ones that would otherwise take exponential time
  - Only problems that satisfy the principle of optimality are suitable for dynamic programming solutions
  - i.e. the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems
- Since exponential time is unacceptable for all but the smallest problems, dynamic programming is sometimes essential

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Consider a large set of possible solutions, storing solutions to subproblems to avoid repeated computation
  - Fibonnaci with "memoizing", string alignment, all-pairs minimum-cost paths
- **Backtracking**
  - **A clever form of exhaustive search**

Backtracking: Idea

- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- A standard example of backtracking would be going through a maze.
  - At some point, you might have two options of which direction to go:



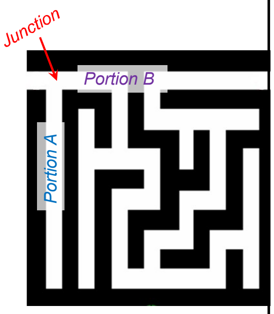
Backtracking

One strategy would be to try going through Portion A of the maze.

If you get stuck before you find your way out, then you *"backtrack"* to the junction.

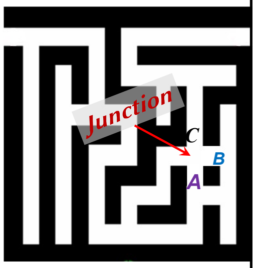
At this point in time you know that Portion A will *NOT* lead you out of the maze,

so you then start searching in Portion B

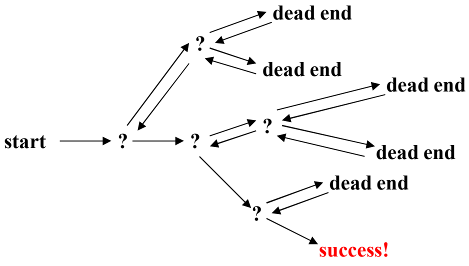


Backtracking

- Clearly, at a single junction you could have even more than 2 choices.
- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, *"backtrack"* to the junction and try the next choice.
- If you try all choices and never found a way out, then there IS no solution to the maze.



Backtracking (animation)



dead end
?
dead end
dead end
start
?
?
?
dead end
dead end
?
success!

Backtracking

- Dealing with the maze:
  - From your start point, you will iterate through each possible starting move.
  - From there, you recursively move forward.
  - If you ever get stuck, the recursion takes you back to where you were, and you try the next possible move.
- Make sure you don't try too many possibilities,
  - Mark which locations in the maze have been visited already so that no location in the maze gets visited twice.
  - (If a place has already been visited, there is no point in trying to reach the end of the maze from there again.

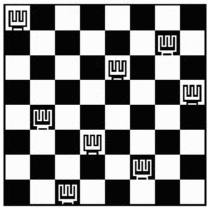
Backtracking

The neat thing about coding up backtracking is that it can be done recursively, without having to do all the bookkeeping at once.

- Instead, the stack of recursive calls does most of the bookkeeping
- (i.e., keeps track of which locations we've tried so far.)

Backtracking: The 8 queens problem

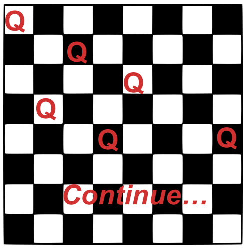
- Find an arrangement of **8** queens on a single chess board such that no two queens are attacking one another.
- In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).
  - Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.



Backtracking

The backtracking strategy is as follows:

1) Place a queen on the first available square in row 1.
2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
3) Continue in this fashion until either:
   a) You have solved the problem, or
   b) You get stuck.
   When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.



Animated Example:
http://www.hbmeyer.de/backtrack/achtdamen/eight.htm#up

Backtracking – 8 queens Analysis

- Another possible brute-force algorithm is generate all possible permutations of the numbers 1 through 8 (there are 8! = 40,320),
  - Use the elements of each permutation as possible positions in which to place a queen on each row.
  - Reject those boards with diagonal attacking positions.
- The backtracking algorithm does a bit better
  - constructs the search tree by considering one row of the board at a time, eliminating most non-solution board positions at a very early stage in their construction.
  - because it rejects row and diagonal attacks even on incomplete boards, it examines only 15,720 possible queen placements.
- 15,720 is still a lot of possibilities to consider
  - Sometimes we have no other choice but to do the best we can ☺

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Consider a large set of possible solutions, storing solutions to subproblems to avoid repeated computation
  - Fibonnaci with "memoizing", string alignment, all-pairs minimum-cost paths
- Backtracking
  - A clever form of exhaustive search