Dynamic programming

CS 2860: Algorithms and Complexity I

Magnus Wahlström, McCrea 113

Magnus.Wahlstrom@rhul.ac.uk

December 5, 2014

Dynamic Programming, review

- Some problems (e.g., Fibonacci numbers) have an easy recursive algorithm which is very slow
- Sometimes, the reason is that the program repeats computations – e.g., fib(10) computes fib(5) over and over and over
- Options for speed-up:
  - Memoization: Remember all computed subproblems to avoid recomputation in a big table
  - Dynamic programming: Replace "recursive scaffolding" to work directly on the table

Fibonacci: Recursive and iterative

Number of computation steps with/without speedup:

n	1	2	3	4	5	6	7	8	9	10	...
Recursive	1	2	3	5	8	13	21	34	55	89	
Iterative	1	2	3	4	5	6	7	8	9	10	

Recursive grows as $1.61\ldots^n$ (exponential) vs iterative $O(n)$

Dynamic programming features

- For both memoization and dynamic programming, time is roughly proportional to number of different subproblems
- If these are few, then an efficient solution is possible (at the cost of more memory)
- Alert: For some problems, first must invent good recursive procedure (which produces only few different subproblems)

Knapsack

Knapsack problem def.

Given n items $[I_1, I_2, \ldots, I_n]$, each with a size `I.size` (integer!) and a value `I.value`, and a total capacity c, find the most valuable way to pack items of total size at most c.

- Example: Items with sizes $(8, 5, 3, 2)$, values $(9, 6, 4, 3)$, capacity 10: Can pack $8 + 2$ for 12 points or $5 + 3 + 2$ for 13 points
- Will see:
  1. Recursive solution scheme
  2. A Dynamic Programming speedup for it

1. Recursive procedure

- Situation: Want to solve knapsack for n items, capacity c, recursively
- Idea:
  1. Pick an item (say, item n)
  2. Look for solutions that contain item n
  3. Look for solutions that do not contain item n
  4. Return the best of the two

Recursive procedure (more detail)

- Solving knapsack for items $[I_1, \ldots, I_n]$, total capacity c
- Solutions that contain item I_n:
  - Solve knapsack for items $[I_1, \ldots, I_{n-1}]$, total capacity $c - I_n.\texttt{size}$
  - Add I_n to the solution
- Solutions that do not contain item I_n:
  - Solve knapsack for items $[I_1, \ldots, I_{n-1}]$, total capacity c
  - Use this solution without adding anything
- Base case: $n = 0$ items (empty solution)

Recursive exponential-time Knapsack

```
Set<Item> best_knapsack(Item[] items, int n, int capacity) {
  if (n <= 0)
    (return empty Set<Item>);
  int size = items[n-1].size;
  if (size > capacity)
    return best_knapsack(items, n-1, capacity);
  else {
    Set<Item> sol1 = best_knapsack(items, n-1, capacity);
    Set<Item> sol2 = best_knapsack(items, n-1, capacity-size);
    sol2.add(items[n-1]);
    (return the best solution of sol1 and sol2);
  }
}
```

Branching algorithms

- This scheme (pick a decision, try both options recursively, return the best one) is called branching
- No better bound than $O(2^n)$ in general
- But with some tricks (pruning: remove "hopeless branches"), can work quite well for simpler (realistic) instances
- Example: SAT solvers: Very efficient solvers for otherwise difficult problems (e.g., hardware verification)

2. Identifying subproblems

- Common traits of all our subproblems:
  - Capacity is between 0 and original
  - Item list is $[I_1, I_2, \ldots, I_i]$: First i items
- In fact, this is all we need to know (solution independence)
- BestKnapsack(n, cap) = Best of:
  1. BestKnapsack(`n-1, cap`),
  2. BestKnapsack(`n-1, cap - size`$_n$) + Item$_n$.
- Only $n*$`cap` different subproblems solved!
- Memoization:
  - Keep a cache that stores and remembers the answer for BestKnapsack(x,y) for all calls (x,y) made to the function

3. Dynamic programming

- We identified `n*capacity` subproblems;
  if we solve these problems, we will find a solution
- With memoization, would in principle be done:
  - Construct `n*capacity` table of solutions
  - Run the recursive algorithm
  - For every recursive call (x,y), if it is in the table already:
    - Return the solution from the table
  - ...otherwise:
    - Keep running the algorithm
    - Get a solution S for (x,y)
    - Put table[x,y]=S, return S

DP table

- Table of size `n_items*capacity`, one entry per subproblem
- Entry table[i,c] should contain best solution using items $[I_1, I_2, \ldots, I_i]$ and max capacity c
- Fill in iteratively, bottom-up
- Use our recursive scheme to find how to do this...
  1. table[0,c]=0 for every c from 0 to max_capacity
  2. table[i,c]=table[i-1,c] if item i does not fit in capacity c
  3. table[i,c]=best(table[i-1,c],
     table[i-1,c-size(i)]+(item i)), otherwise
- Works if we fill in all values table[i-1,c2] before table[i,c]

Table fill-in example (solution values only)

Assume: Items of sizes $[2, 3, 8, 5]$, of values $[3, 4, 9, 6]$, capacity 10

Item set	0	1	2	3	4	5	6	7	8	9	10
$[I_1]$	0	0	3	3	3	3	3	3	3	3	3
$[I_1, I_2]$	0	0	3	4	4	7	7	7	7	7	7
$[I_1, I_2, I_3]$	0	0	3	4	4	7	7	7	9	9	12
$[I_1, I_2, I_3, I_4]$	0	0	3	4	4	7	7	9	10	10	13

So original problem ($i = 4$, $c = 10$) has best solution value 13.

Problem 2: Longest Common Subsequence

- Situation: Have two strings, want to find the longest subsequence that occurs in both (not substring!)
- Subsequence: May occur "with gaps"
  - Technically: S is subsequence of T: Can delete characters from T to get S
- Example: "abc" occurs in "fabric" and "tablecloth"
- Applications: DNA comparison, "diff" file comparisons
  - Related to edit distance (strings) or some notion of mutation distance (DNA)

Longest Common Subsequence: Recursion

- Ex: Find LCS of "fabric" and "tablecloth"
- Recursive calls: 'f' $\neq$ 't', so we need to search:
  1. Find LCS of "abric" and "tablecloth"
  2. Find LCS of "fabric" and "ablecloth"
- Ex 2: Find LCS of "abric" and "ablecloth":
  - Answer is 'a'+(LCS of "bric" and "blecloth")
- Convince yourself: There is no "danger" in always using the first letter if it is the same in both words

Longest Common Subsequence: Subproblems

- We have a recursive scheme ("remove" first character of one of the strings, unless the first characters match)
- Resulting subproblems:
    - LCS("fabric", "tablecloth")
    - LCS("abric,", "tablecloth")
    - LCS("fabric", "ablecloth")
    - LCS("bric", "tablecloth")
    - LCS("bric", "ablecloth")
    - ...
- All problems `LCS(String1(i...n1), String2(j...n2))`

Longest Common Subsequence: Composing

- Have subproblems: LCS(String1(i ... n1), String2(j ... n2))
- Answer stored in table[i,j]
- Have recursive scheme:
  - Base case: LCS("",S)="" (empty string), so table[n1, j]= "" for every value of j
  - If first characters match, set table[i,j]=(first character)+table[i+1,j+1]
  - Otherwise set table[i,j]=best(table[i+1,j], table[i,j+1])
- Dependency: Fill in all of row i+1 before row i, and entry (i,j+1) before entry (i,j)

Dynamic Programming, summary

- Powerful method to solve problem (recursively or iteratively) by composing the final solution out of smaller solutions
- Need "composing" strategy (recursive scheme) which produces few different subproblems
- Can get tricky to find the "right" composing strategy! (See [CLRS], Chapter 15 for examples)
- Dynamic Programming also heavily used in string problems, graph problems