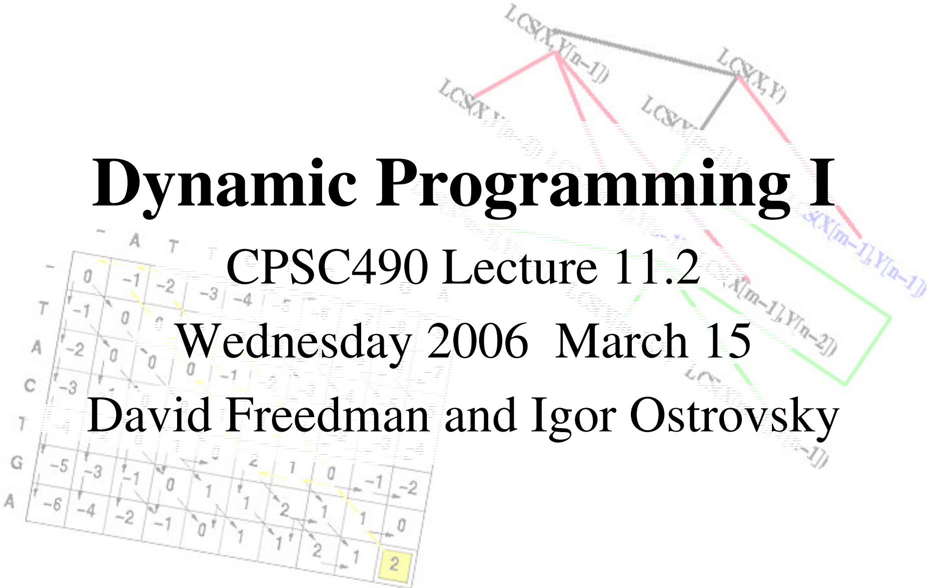


Dynamic Programming I

CPSC490 Lecture 11.2

Wednesday 2006 March 15

David Freedman and Igor Ostrovsky

Etymology of Dynamic Programming

- Richard Bellman, the of Bellman-Ford algorithm fame, invented dynamic programming in 1953.[1]
- Bellman was dependent on government funding at a time when the American Secretary of Defence was "hostile to mathematical research," and so he came up with the name as "something not even a congressman could object to." [2]
- *Dynamic* was intended as a synonym for *pejorative* (to make worse). [3]
- *Programming* was intended to be the mathematical concept of programming, as in *linear programming*[4] (ask a mathematician, I don't know).

[1,4] Tan, Gang. "Chapter 6: Dynamic Programming." *Boston College, CS383*. Fall 2005. Available: http://www.cs.bc.edu/~gtan/teaching/cs383f5/slides/cs383_06dynamic-programming.pdf

[2,3] Wikipedia contributors. "Dynamic Programming." *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/w/index.php?title=Dynamic_programming

What Are Some DP Problems?

- DP problems we've already seen in this course are:
  - Bellman-Ford shortest path algorithm
  - Dijkstra's shortest path algorithm
- Classic DP problems we'll be looking at today are:
  - longest increasing subsequence problem
  - backpacker problem / knapsack problem

Longest Increasing Subsequence Problem

- Given a set of numbers, pick out, in order of appearance, as elements as possible such that each chosen element picked is larger than the previously picked element.
- For example, given the set:
  { 14, **1**, 17, **2**, 16, 17, **3**, 15, **4**, 1, **5**, 18, 13, **6**, **7**, 19, **8**, 12, 1, **9**, **10**, 8 }

  the long increasing subset is:
  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
- How do we solve this problem? Forget about it for now, we'll get to it later.
- How is this a DP problem? Before we answer that, we must first answer...

What is Dynamic Programming?

- Dynamic programming **is not** an algorithm
- Dynamic programming **is not** a style of programming.
- Dynamic programming **is** the set of problems which have both **overlapping subproblems** and **optimal substructure**.

What Is An Optimal Substructure?

- A problem with an optimal substructure is one which can be expressed entirely in terms of **overlapping subproblems**.
- ...but what's an overlapping subproblem?

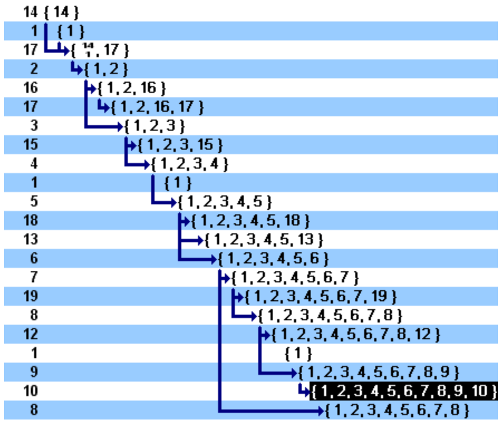
Overlapping Subproblems

- Overlapping subproblems are simple problems which have to be solved repeatedly to solve the overall problem.
- In the longest increasing subsequence problem, we can break it down into:
  - What is the longest increasing subsequence that ends at each element?
  - What is the longest of any of those subsequences?

Longest Increasing Subsequence Implementation

Q: What is the longest increasing subsequence that ends at each element?

A: If this is the smallest item we've seen so far, then it's the only item in the set. Otherwise, find the previous item with the longest-longest subset which is smaller than the element, and set the longest subset of this element to the union of the element and that subset.



Longest Increasing Subsequence Code


```
static int longestIncreasingSubsequence(int[] p_asuperSequence) {
    int[] abest = new int[p_asuperSequence.length];
    int nret = 0;

    for (int i = 0; i < p_asuperSequence.length; i++) {
        for (int j = 0; j < i; j++)
            if (p_asuperSequence[i] > p_asuperSequence[j])
                abest[i] = Math.max(abest[i], abest[j] + 1);
        nret = Math.max(nret, abest[i]);
    } //next i
    return nret + 1;
} //end method
```


Think About...

- Why is a shortest path problem DP?
- What are the subproblems of the Bellman-Ford algorithm?
- What are the subproblems of Dijkstra's algorithm?

Backpacker/Knapsack Problem

This is one of the classic DP problems. Here's the general problem:

There is a guy with a knapsack of capacity *c* who can only carry items which fit into that knapsack. (*c* may be a weight, or a volume, it doesn't really matter).

The guy has available to him a set of items, each of which has a value, and wants to load up his knapsack with the most valuable set of items which he can carry, and he needs to do it quickly.

I've often imagined the guy is a shoplifter, because who else might have these requirements?

Subproblems of the Backpacker Problem

- There are two subproblems within the backpacker problem:
  - Examine subsets of the items: if the items available are { i_0, i_1 .. i_n }, then, for every integer k from $0..n$, consider only the subset of items i_0 .. i_k.
  - Much less intuitively, examine lesser maximum capacities: if the maximum capacity is c, then for every integer w from $0..c$, find the maximum value of of a subset of the subset i_0 .. i_k whose weight is $< w$.

Backpacker Implementation

Items:

item A	1kg	$100
item B	3kg	$200
item C	5kg	$301
item D	7kg	$400
item E	9kg	$500

Max Weight: 10kg

How to calculate each cell:

for every cell [*x*, *y*]
  if *x* = 0 or *y* = 0, then
    cell[x, y] := `0kg $0 { }`
  otherwise
    let *m* := the item for column *y*
    let *w* := the weight of *m*
    if *w* is more than the row's maximum weight
      cell[*x*, *y*] := *cell*[*x* - 1, *y*]
    otherwise
      cell[*x*, *y*] := the pricier of *cell*[*x* - 1, *y*] and (*cell*[*x* - 1, *y* - *w*] + *m*)

	no items	item A	items A/B	items A/B/C	items A/B/C/D	items A/B/C/D/E
kg <= 0	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 1	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 2	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 3	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 4	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 5	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 6	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 7	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 8	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 9	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }
kg <= 10	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }	??kg $?? { ??? }

Backpacker Implementation

Items:

item A	1kg	$100
item B	3kg	$200
item C	5kg	$301
item D	7kg	$400
item E	9kg	$500

Max Weight: 10kg

How to calculate each cell:

for every cell [*x*, *y*]
  if *x* = 0 or *y* = 0, then
    cell[x, y] := `0kg $0 { }`
  otherwise
    let *m* := the item for column *y*
    let *w* := the weight of *m*
    if *w* is more than the row's maximum weight
      cell[*x*, *y*] := *cell*[*x* - 1, *y*]
    otherwise
      cell[*x*, *y*] := the pricier of *cell*[*x* - 1, *y*] and (*cell*[*x* - 1, *y* - *w*] + *m*)

	no items	item A	items A/B	items A/B/C	items A/B/C/D	items A/B/C/D/E
kg <= 0	0kg $0 {}	0kg $0 {}	0kg $0 {}	0kg $0 {}	0kg $0 {}	0kg $0 {}
kg <= 1	0kg $0 {}	1kg $100 { A }	1kg $100 { A }	1kg $100 { A }	1kg $100 { A }	1kg $100 { A }
kg <= 2	0kg $0 {}	1kg $100 { A }	1kg $100 { A }	1kg $100 { A }	1kg $100 { A }	1kg $100 { A }
kg <= 3	0kg $0 {}	1kg $100 { A }	3kg $200 { B }	3kg $200 { B }	3kg $200 { B }	3kg $200 { B }
kg <= 4	0kg $0 {}	1kg $100 { A }	4kg $300 { A, B }	4kg $300 { A, B }	4kg $300 { A, B }	4kg $300 { A, B }
kg <= 5	0kg $0 {}	1kg $100 { A }	4kg $300 { A, B }	5kg $301 { C }	5kg $301 { C }	5kg $301 { C }
kg <= 6	0kg $0 {}	1kg $100 { A }	4kg $300 { A, B }	6kg $401 { A, C }	6kg $401 { A, C }	6kg $401 { A, C }
kg <= 7	0kg $0 {}	1kg $100 { A }	4kg $300 { A, B }	6kg $401 { A, C }	6kg $401 { A, C }	6kg $401 { A, C }
kg <= 8	0kg $0 {}	1kg $100 { A }	4kg $300 { A, B }	8kg $501 { B, C }	8kg $501 { B, C }	8kg $501 { B, C }
kg <= 9	0kg $0 {}	1kg $100 { A }	4kg $300 { A, B }	9kg $601 { A, B, C }	9kg $601 { A, B, C }	9kg $601 { A, B, C }
kg <= 10	0kg $0 {}	1kg $100 { A }	4kg $300 { A, B }	9kg $601 { A, B, C }	9kg $601 { A, B, C }	9kg $601 { A, B, C }

Backpacker Questions

- **Do we really need to keep track of the weights, values, and items?**
- **Do we need a huge 2-dimensional array?**
- **What's the running time of this algorithm?**

Backpacker Questions

- **Do we really need to keep track of the weights, values, and items?**
  - Absolutely not. At a minimum, we need the cost, although we *may* want the weight or item list depending on whether we were asked to list the items, state the value, or state the weight.
- **Do we need a huge 2-dimensional array?**
  - You could use a full 2d matrix like we just saw (most examples on the web do), but as we only look at two columns at at time, it's possible to implement this using two arrays of whose length is the maximum weight (next slide)
- **What's the running time of this algorithm?**
  - O(items * weight)

Backpacker Code


```
static int backpackerMaxValue(int p_nmaxWeight, int[] p_aweight, int[] p_acost) {
    int ncurr = 0, nprev = 1; //column indices. invar: ncurr ∈ { 0, 1 }, nprev == 1 – ncurr
    int nweight; //row index, also max weight for the row
    int[][] abestValue = new int[2][p_nmaxWeight + 1]; //optimal value for each row
                                                       //for the current (ncurr) and
                                                       //previous (nprev) columns.
    //for every item
    for (int nitem = 0; nitem < p_acost.length; nitem++) {

        //ignore items heavier than the knapsack
        if (p_aweight[nitem] > p_nmaxWeight)
            continue;

        //swap ncurr and nprev
        ncurr = (nprev = ncurr) ^ 1;

        //for all rows where the item is heavier than the row’s maximum weight
        for (nweight = 1; nweight < p_aweight[nitem]; nweight++)
            abestValue[ncurr][nweight] = abestValue[nprev][nweight];

        //for all rows where the item is at most than the row’s maximum weight
        for (; nweight <= p_nmaxWeight; nweight++)
            abestValue[ncurr][nweight] = Math.max(
                abestValue[nprev][nweight - p_aweight[nitem]] + p_acost[nitem],
                abestValue[nprev][nweight]
            );
    } //next nitem

    return abestValue[ncurr][p_nmaxWeight];
} //end method
```


Questions?

References

Bellman, Richard. *Eye of the Hurricane: An Autobiography*. World Scientific Pub Co Inc, 1984. ISBN: 9971966018.

Frank and Igor. “Dynamic Programming.” *University of British Columbia, CS490*. Available: Mike and Dustin.

Michie, Donald. “Memo Functions and Machine Learning.” *Nature*, 218:19-22. Macmillan Publishers, 1968.

Rain Man. Dir. Barry Levinson. Perfs. Dustin Hoffman, Tom Cruise. Film. United Artists, 1988.

Tan, Gang. “Chapter 6: Dynamic Programming.” *Boston College, CS383*. Fall 2005. Available: http://www.cs.bc.edu/~gtan/teaching/cs383f5/slides/cs383_06dynamic-programming.pdf

Wikipedia contributors. “Dynamic programming” *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Dynamic_programming

Wikipedia contributors. “Hamiltonian path” *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Hamiltonian_path

Wikipedia contributors. “Knapsack problem” *Wikipedia, The Free Encyclopedia*. February 2005. Available: http://en.wikipedia.org/wiki/Knapsack_problem

Wikipedia contributors. “Longest increasing subsequence problem” *Wikipedia, The Free Encyclopedia*. January 2005. Available: http://en.wikipedia.org/wiki/Longest_increasing_subsequence_problem

Wikipedia contributors. “Memoization” *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Memoization

Wikipedia contributors. “Optimal substructure” *Wikipedia, The Free Encyclopedia*. January 2005. Available: http://en.wikipedia.org/wiki/Optimal_substructure

Wikipedia contributors. “Overlapping subproblem” *Wikipedia, The Free Encyclopedia*. February 2005. Available: http://en.wikipedia.org/wiki/Overlapping_subproblem

Wikipedia contributors. “Traveling salesman problem” *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Traveling_salesman_problem