Martin Broadhurst

UNBOUNDED KNAPSACK USING DYNAMIC PROGRAMMING IN C

FEBRUARY 19, 2017 | MARTIN

The unbounded knapsack problem is to try to fill a knapsack of a given capacity with items of given weight and value in such a way as to maximise the value of the knapsack. There is no limit on the number of instances of each item, hence the "unbounded" label. The decision problem of whether the knapsack can be filled to greater than or equal to a value is NP-complete. The maximisation problem can be solved in pseudo-polynomial time using dynamic programming.

The algorithm works by filling an array of *0, …, the capacity of the knapsack* from the bottom up with the value most valuable knapsack for each capacity. In order to work out at each capacity what the most profitable knapsack is, it considers each item in turn and finds out what the value of the knapsack would be if this was the last item added. This value is the value of the earlier knapsack that is lighter by the *weight* of the current item, plus the *value* of the current item. Since the knapsack values are calculated from lower to higher capacities, the value of this earlier, lighter knapsack can always be found without needing to recompute it. Once each of the items has been considered as the most recent addition, the one that gives rise to the most valuable knapsack is added, and this value is recorded in the current cell of the array. The algorithm then proceeds to the next cell of the array. When the algorithm reaches the top of the array, and fills that cell in, it has found the most valuable knapsack for the specified capacity.

There is an implicit link between each knapsack in the array and the previous lighter knapsack from which it was derived by adding an item. These links form multiple chains through the array. The chain that begins at the top cell is the one that traces the path of additions that have led to the most profitable knapsack. In order to realise this link, the array cells contain pointers to the earlier knapsack from which they were constructed, and these pointers are set as the algorithm progresses. Once the array is full, the values of the items can be retrieved by following this linked list of pointers.

```
1  struct knapsack {
2      unsigned int profit;
3      struct knapsack *prev;
4  };
5  typedef struct knapsack knapsack;
6
7  /* Find the minimum weight item with this profit */
```

```c
int min_weight_item(unsigned int profit, const unsigned int *weights,
        const unsigned int *profits, size_t len)
{
    int item = -1;
    unsigned int i;
    for (i = 0; i < len; i++) {
        if (profits[i] == profit) {
            if (item == -1 || weights[i] < weights[item]) {
                item = i;
            }
        }
    }
    return item;
}

unsigned int unbounded_knapsack(unsigned int capacity, unsigned int *weigl
        unsigned int *profits, unsigned int *counts, size_t len)
{
    knapsack *z = malloc((capacity + 1) * sizeof(knapsack));
    unsigned int c, i;
    unsigned int solution, profit;
    z[0].profit = 0;
    z[0].prev = NULL;
    knapsack *current;
    /* Fill in the array */
    for (c = 1; c <= capacity; c++) {
        z.profit = z.profit;
        z.prev = &(z);
        for (i = 0; i < len; i++) {
            if (weights[i] <= c) {
                /* prev is the best knapsack without adding this item */
                knapsack *prev = z + (c - weights[i]);
                if (prev->profit + profits[i] > z.profit) {
                    z.profit = prev->profit + profits[i];
                    z.prev = prev;
                }
            }
        }
    }
    /* Read back the best solution */
    for (profit = z[capacity].profit, current = z[capacity].prev;
            current != NULL;
            profit = current->profit, current = current->prev) {
        counts[min_weight_item(profit - current->profit, weights, profits

    }
    solution = z[capacity].profit;
    free(z);
    return solution;
}
```

An example program:

```c
static void print_knapsack(const unsigned int *counts, const unsigned int
{
```

```
    unsigned int i;
    for (i = 0; i < len; i++) {
        if (counts[i] > 0) {
            printf("%d x %d\n", counts[i], profits[i]);
        }
    }
}

int main(void)
{
    unsigned int weights[] = {4, 3, 5, 7, 11};
    unsigned int profits[] = {5, 3, 6, 2, 7};
    unsigned int counts[5] = {0};
    const size_t len = sizeof(weights) / sizeof(unsigned int);
    const unsigned int capacity = 17;
    printf("The maximum profit is %u\n",
           unbounded_knapsack(capacity, weights, profits, counts, len));
    print_knapsack(counts, profits, len);
    return 0;
}
```

Output:

```
The maximum profit is 21
3 x 5
1 x 6
```

ALGORITHMS C DYNAMIC PROGRAMMING KNAPSACK NP-COMPLETE UNBOUNDED KNAPSACK