On a Variant of the Change-Making Problem

Adam N. Letchford* Licong Cheng†

To appear in *Operations Research Letters*

Abstract

The *change-making problem* (CMP), introduced in 1970, is a classic problem in combinatorial optimisation. It was proven to be NP-hard in 1975, but it can be solved in pseudo-polynomial time by dynamic programming. In 1999, Heipcke presented a variant of the CMP which, at first glance, looks harder than the standard version. We show that, in fact, her variant can be solved in polynomial time.

Keywords: change-making problem; knapsack problems; combinatorial optimisation

1 Introduction

The *Change-Making Problem* (CMP) is a classic NP-hard combinatorial optimisation problem, first studied in [8]. We are given a positive integer n, positive integer *coin values* $v_1, \ldots, v_n$, and a positive integer *target value* t. The task is to determine a collection of coins, of minimum cardinality, whose total value is equal to the target value. For example, if $n = 3$, $v = (5, 4, 1)$ and $t = 13$, the optimal solution uses 3 coins in total (one coin of value 5 and two of value 4).

The CMP can be viewed as a special case of the "equality-constrained knapsack problem". We refer the reader to the books [10, 14] and the recent surveys [3, 4] for more on knapsack problems.

The CMP was shown to be NP-hard in [12]. On the other hand, the CMP is only NP-hard in the "weak" sense, since it can be solved in pseudo-polynomial time. (Indeed, it can be solved in $O(nt)$ time using a standard dynamic programming approach [17].) In practice, the CMP can often be solved quickly [14], although some instances with large target and coin values can be challenging [1].

*Department of Management Science, Lancaster University, Lancaster LA1 4YX, UK. E-mail: a.n.letchford@lancaster.ac.uk

†Former masters student in the Department of Management Science, Lancaster University, UK. E-mail: martinlicongcheng@gmail.com

In 1999, Heipcke [9] considered a variant of the CMP in which one must find a minimum-cardinality collection of coins such that it is possible to pay *any* positive integer amount up to the target t using only coins from the collection. In this paper, we will call this variant the "Heipcke" CMP, or H-CMP for short.

To make the meaning of the H-CMP clear, consider again the case in which $n = 3$, $v = (5, 4, 1)$ and $t = 13$. One can obtain a feasible solution of the H-CMP using 6 coins, by using one coin of value 5, two of value 4 and three of value 1. Indeed:

- For $i = 1, 2, 3$, we can pay an amount of i by using i coins of value 1.
- For $i = 4, 5, 6, 7$, we can pay an amount of i by using one coin of value 4 and $i - 4$ coins of value 1.
- For $i = 8, 9, 10, 11$, we can pay an amount of i by using two coins of value 4 and $i - 8$ coins of value 1.
- We can pay an amount of 12 by using the coin of value 5, one coin of value 4, and 3 coins of value 1.
- We can pay an amount of 13 by using the coin of value 5 and both coins of value 4.

One can check (by brute-force enumeration) that no solution using fewer than 6 coins exists for the given H-CMP instance.

Note that a feasible solution to an H-CMP instance exists if and only if $v_k = 1$ for some k. Indeed, if this condition holds, then we obtain a trivial feasible solution by selecting t coins of value 1. If the condition does not hold, then we cannot pay an amount of 1 no matter what collection of coins is chosen.

Heipcke developed two exact algorithms for the H-CMP, one based on constraint programming and the other based on integer programming. The computational results that she obtained with those algorithms suggested that the H-CMP may be significantly harder to solve than the standard CMP. Moreover, there is no obvious dynamic programming algorithm for the H-CMP.

In this paper we show that, despite appearances, the H-CMP can be solved in polynomial time. In fact, we show that it can be solved in only $O(n \log n)$ time.

The paper has a very simple structure. Section 2 is a literature review and Section 3 presents our algorithm and a proof of correctness.

2 Literature Review

This section contains a brief literature review. Subsection 2.1 concerns the standard CMP and Subsection 2.2 concerns the H-CMP.

2.1 The standard change-making problem

Wright [17] formulated the standard CMP as an *Integer Linear Program* (ILP). For $i = 1, \ldots, n$, the general-integer variable x_i represents the number of coins of type i used. We then have:

$$\begin{array}{rc} \min & \sum_{i=1}^{n} x_i \\ \text{s.t.} & \sum_{i=1}^{n} v_i x_i = t \\ & x \in \mathbb{Z}_+^n. \end{array}$$

Several exact algorithms have been proposed for the CMP. Chang & Gill [8] devised an algorithm based on recursion, which however is very slow in both theory and practice. Wright [17] pointed out that the CMP can be solved in $O(nt)$ time via dynamic programming. Martello & Toth [13] derived some useful lower bounds for the CMP, and showed how to embed them in an effective branch-and-bound algorithm. They presented an improved version of their algorithm in [14].

More recently, Chan & He [6] found an enhanced dynamic programming algorithm for the CMP, which runs in $O(n + t \log t \log \log t)$ time. A couple of years later [7], the same authors found a version of their algorithm which runs in $O(n + \bar{v} \log^3 \bar{v})$ time, where $\bar{v}$ is the maximum of the values v_i.

Chang & Gill [8] also pointed out that there is a natural "greedy" heuristic for the CMP: use as many as possible of the coins with the highest value, then as many as possible of the coins with the second-highest value, and so on. Several researchers have examined conditions on the values v_i such that, regardless of the target t, the greedy heuristic always finds an optimal solution to the CMP (e.g., [2, 5, 11, 15, 16]). We do not go into details, for brevity.

2.2 Heipcke's variant of the CMP

As mentioned in the introduction, Heipcke [9] applied both constraint programming and integer programming to the H-CMP. For clarity, we recall her ILP formulation. The variables x_i are defined just as for the standard CMP. Then, for $i = 1, \ldots, n$ and $s = 1, \ldots, t$, the general-integer variable y_{is} denotes the number of coins of type i that would be used to give exactly s in change. We then have:

$$\begin{array}{rcl} \min & \sum_{i=1}^{n} x_i & \\ \text{s.t.} & \sum_{i=1}^{n} v_i y_{is} = s & (s = 1, \ldots, t) \\ & y_{is} \le x_i & (i = 1, \ldots, n;\ s = 1, \ldots, t) \\ & x \in \mathbb{Z}_+^n, y \in \mathbb{Z}_+^{nt}. & \end{array}$$

The interpretation of the constraints is straightforward.

Note that this ILP has $n(t+1)$ variables and constraints, and is therefore of pseudo-polynomial size rather than polynomial. Moreover, in any feasible solution, $\Omega(t)$ variables will take a positive value. Thus, even if someone presented us with a feasible solution to the ILP, checking that it is indeed feasible would take $\Omega(t)$ time.

3 A Polynomial Time Algorithm

In this section, we present a polynomial-time algorithm for the H-CMP. For notational convenience, we assume throughout that $n \geq 2$. (This is without loss of generality, since instances with $n = 1$ are trivial.) We also assume, again without loss of generality, that $v_k \leq t$ for all t. Moreover, for reasons which will become clear, we let v_{n+1} denote $t + 1$.

The first step in our algorithm is to sort the coin values in increasing order. Once this is done, we can assume without loss of generality that $1 = v_1 < v_2 < \cdots < v_{n+1}$. Note that this sorting can be done in $O(n \log n)$ time using any of several well-known sorting algorithms.

Now observe that, if we wanted to pay the amount $v_k - 1$ for some $k \in \{2, \ldots, n+1\}$, then we would have to do it using coins of value less than v_k. This immediately suggests the following approach to the H-CMP. Start with all x variables set to 0. Then, for $k = 1, \ldots, n-1$, increase x_k to a value large enough to ensure that we have enough coins to pay any amount up to $v_{k+1} - 1$.

The details are given in Algorithm 1. Throughout the algorithm, N denotes the total number of coins selected so far, and V denotes their total value. More precisely, in the main loop, for any given value of k, N is updated to take the value $\sum_{i=1}^{k} x_i$, and V is updated to take the value $\sum_{i=1}^{k} v_i x_i$.

It is easy to verify that the algorithm runs in $O(n)$ time. Before proving that it constructs an optimal solution, we run the algorithm on two examples. (The first is taken from Heipcke's paper.)

Example 1: Let $n = 6$, $v = (1, 2, 5, 10, 20, 50)$ and $t = 99$. Heipcke showed that there are four optimal solutions, each using eight coins. The main loop of our algorithm proceeds as follows:

- $k = 1$: we set x_1 to $\lceil (2 - 1 - 0)/1 \rceil = 1$. N and V increase to 1.
- $k = 2$: we set x_2 to $\lceil (5 - 1 - 1)/2 \rceil = 2$. N increases to 3 and V increases to 5.
- $k = 3$: we set x_3 to $\lceil (10 - 1 - 5)/5 \rceil = 1$. N increases to 4 and V increases to 10.
- $k = 4$: we set x_4 to $\lceil (20 - 1 - 10)/10 \rceil = 1$. N increases to 5 and V increases to 20.

Algorithm 1: Solving the H-CMP

input : Integer $n \geq 2$; sorted coin values $v_1, \ldots, v_n$; positive integer target t

`// Initialisation`

for $k = 1, \ldots, n$ **do**

  Set x_k to 0;

end

Set N and V to 0;

Set v_{n+1} to $t + 1$;

`// Main loop`

for $k = 1, \ldots, n$ **do**

  if $v_{k+1} - 1 > V$ **then**

    Set x_k to $\left\lceil \frac{v_{k+1}-1-V}{v_k} \right\rceil$;

    Increase N by x_k and increase V by $v_k x_k$;

  end

end

output: Optimal solution vector x and total number of coins N

- $k = 5$: we set x_5 to $\lceil (50 - 1 - 20)/20 \rceil = 2$. N increases to 7 and V increases to 60.
- $k = 6$: we set x_6 to $\lceil (99 - 60)/50 \rceil = 1$. N increases to 8 and V increases to 110.

The resulting solution is $x = (1, 2, 1, 1, 2, 1)$, using 8 coins as desired. $\square$

Example 2: Let $n = 5$, $v = (1, 4, 7, 8, 10)$ and $t = 15$. The main loop of our algorithm proceeds as follows:

- $k = 1$: we set x_1 to $\lceil (4 - 1 - 0)/1 \rceil = 3$. N and V increase to 3.
- $k = 2$: we set x_2 to $\lceil (7 - 1 - 3)/4 \rceil = 1$. N increases to 4 and V increases to 7.
- $k = 3$: we set x_3 to $\lceil (8 - 1 - 7)/7 \rceil = 0$. N and V remain unchanged at 4 and 7, respectively.
- $k = 4$: we set x_4 to $\lceil (10 - 1 - 7)/8 \rceil = 1$. N increases to 5 and V increases to 15.
- $k = 5$: we set x_8 to $\lceil (15 - 15)/10 \rceil = 0$. N and V remain unchanged at 5 and 15, respectively.

The resulting solution is $x = (3, 1, 0, 1, 0)$, using 5 coins. One can check by brute-force enumeration that no solution with fewer than 5 coins exists. $\square$

We now prove correctness of the algorithm.

Theorem 1 *The vector x constructed by Algorithm 1 represents an optimal solution to the given H-CMP instance.*

Proof. In this proof, we view the algorithm as starting with an empty collection of coins, and then iteratively adding coins to the collection. That is, our final collection of coins contains x_k coins of value v_k for all k. We also let N_k and V_k denote the value of N and V, respectively, at the end of iteration k.

Now we prove that the vector x constructed by the algorithm represents a *feasible* H-CMP solution. When $k = 1$, we add $v_2 - 1$ coins of value 1 to our collection. Using some or all of those coins, we can pay any amount between 1 and $v_2 - 1$. Now we use induction. Suppose that we are at the start of iteration k for some $2 \leq k \leq n$, and assume that we can pay any amount between 1 and V_{k-1} using only the coins in our current collection, where $V_{k-1} \geq v_k - 1$. In iteration k, we add to our collection enough coins of value v_k to bring the total value up to $V_k \geq v_{k+1} - 1$. Now suppose that we wanted to use coins in our current collection to pay some amount y between 1 and V_k. We could use $\alpha = \lceil (y - V_{k-1})/v_k \rceil$ coins of value v_k, and then use coins of lower value to make up the remaining amount, if any. To see this, note that (i) $\alpha \leq x_k$ by construction, and (ii) $\alpha v_k \geq y - V_{k-1}$, which implies that the remaining amount to be paid, $y - \alpha v_k$, is no larger than V_{k-1}.

To complete the proof, we need to show that the vector x represents an *optimal* H-CMP solution. A first observation is that we need to have at least $v_2 - 1$ coins of value 1 in our collection, so that we are able to pay any amount between 1 and $v_2 - 1$. We add precisely this number of coins of value 1 to our collection in iteration 1 of the main loop. We also set N_1 to $v_2 - 1$. So, at the end of iteration 1, N_1 is the minimum possible value that x_1 can take in a feasible H-CMP solution, and x_1 has been set to this minimum value. Now we use induction a second time. Suppose that we are at the start of iteration k for some $2 \leq k \leq n$, and (i) we have already shown that all feasible x vectors satisfy $x_1 + \cdots + x_{k-1} \geq N_{k-1}$, and (ii) we have already set $x_1, \ldots, x_{k-1}$ to values that achieve this bound. We now need to ensure that there are enough coins of value $v_1, \ldots, v_k$ to pay any amount between 1 and $v_{k+1} - 1$. To minimise $x_1 + \cdots + x_k$, subject to the constraint $x_1 + \cdots x_{k-1} \geq N_{k-1}$, it suffices to set x_k to $\lceil ((v_{k+1} - 1) - V_{k-1})/v_k \rceil$. This is exactly what we do in iteration k of the main loop. $\square$

References

[1] K. Aardal & A.K. Lenstra (2004) Hard equality constrained integer knapsacks. *Math. Oper. Res.*, 29, 724–738.

[2] A. Adamaszek & M. Adamaszek (2010) Combinatorics of the change-making problem. *Eur. J. Combin.*, 31, 47–63.

[3] V. Cacchiani, M. Iori, A. Locatelli & S. Martello (2022) Knapsack problems–An overview of recent advances, Part I: Single knapsack problems. *Comput. Oper. Res.*, 143, article 105692.

[4] V. Cacchiani, M. Iori, A. Locatelli & S. Martello (2022) Knapsack problems–An overview of recent advances, Part II: Multiple, multidimensional, and quadratic knapsack problems. *Comput. Oper. Res.*, 143, article 105693.

[5] X. Cai (2009) Canonical coin systems for change-making problems. In J.-S. Pan, J. Liu & A. Abraham (eds) *Proc. HIS '09*, pp. 499-504. Los Alamitos, California: IEEE.

[6] T.M. Chan & Q. He (2020) On the change-making problem. In M. Farach-Colton & I.L. Gørtz (eds) *Proc. SOSA '20*, pp. 38–42. Philadelphia: SIAM.

[7] T.M. Chan & Q. He (2022) More on change-making and related problems. *J. Comput. Sys. Sci.*, 124, 159–169.

[8] S.K. Chang & A. Gill (1970) Algorithmic solution of the change-making problem. *J. ACM*, 17, 113–122.

[9] S. Heipcke (1999) Comparing constraint programming and mathematical programming approaches to discrete optimisation——the change problem. *J. Oper. Res. Soc.*, 50, 581–595.

[10] H. Kellerer, U. Pferschy & D. Pisinger (2004) *Knapsack Problems.* Berlin: Springer.

[11] D. Kozen & S. Zaks (1994) Optimal bounds for the change-making problem. *Theor. Comput. Sci.*, 123, 377–388.

[12] G.S. Lueker (1975) Two NP-complete problems in nonnegative integer programming. *Report No. 178*, Computer Science Laboratory, Princeton University, New Jersey.

[13] S. Martello & P. Toth (1980) Optimal and canonical solutions of the change making problem. *Eur. J. Oper. Res.*, 4, 322–329.

[14] S. Martello & P. Toth (1990) *Knapsack Problems: Algorithms and Computer Implementations.* Chichester, UK: Wiley.

[15] D. Pearson (2005) A polynomial-time algorithm for the change-making problem. *Oper. Res. Lett.*, 33, 231–234.

[16] Y. Suzuki & R. Miyashiro (2023) Characterization of canonical systems with six types of coins for the change-making problem. *Theor. Comput. Sci.*, 955, article 113822.

[17] J.W. Wright (1975) The change-making problem. *J. ACM*, 22, 125–128.