BIOINFORMATICS ORIGINAL PAPER
Vol. 22 no. 6 2006, pages 685–691
doi:10.1093/bioinformatics/btk035

Genome analysis

A greedier approach for finding tag SNPs

Chia-Jung Chang[1], Yao-Ting Huang[1] and Kun-Mao Chao[1,2,*]
[1]Department of Computer Science and Information Engineering and [2]Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan

Received on November 4, 2005; revised on December 10, 2005; accepted on December 30, 2005
Advance Access publication January 10, 2006
Associate Editor: Martin Bishop

ABSTRACT

Motivation: Recent studies have shown that a small subset of Single Nucleotide Polymorphisms (SNPs) (called tag SNPs) is sufficient to capture the haplotype patterns in a high linkage disequilibrium region. To find the minimum set of tag SNPs, exact algorithms for finding the optimal solution could take exponential time. On the other hand, approximation algorithms are more efficient but may fail to find the optimal solution.

Results: We propose a hybrid method that combines the ideas of the branch-and-bound method and the greedy algorithm. This method explores larger solution space to obtain a better solution than a traditional greedy algorithm. It also allows the user to adjust the efficiency of the program and quality of solutions. This algorithm has been implemented and tested on a variety of simulated and biological data. The experimental results indicate that our program can find better solutions than previous methods. This approach is quite general since it can be used to adapt other greedy algorithms to solve their corresponding problems.

Availability: The program is available upon request.

Contact: kmchao@csie.ntu.edu.tw

1 INTRODUCTION

The genetic variations in DNA sequences have a major impact on genetic diseases and phenotypic differences. Among various genetic variations, the single nucleotide polymorphism (SNP) is the most frequent form which has fundamental importance for disease association and drug design. A SNP is a genetic variation when a single nucleotide (i.e. A, C, G, or T) in the DNA sequence is altered and kept through heredity thereafter. A set of linked SNPs on one chromosome is called a haplotype. Millions of SNPs have been identified and these data are now publicly available (Helmuth, 2001; Hinds *et al*., 2005; Altshuler *et al*., 2005).

In recent years, the patterns of linkage disequilibrium (LD) observed in the human population reveal a block-like structure (Bafna *et al*., 2003; Daly *et al*., 2001; Patil *et al*., 2001; Hinds *et al*., 2005; Zhang *et al*., 2004a). The entire chromosome can be partitioned into high LD regions interspersed by low LD regions. The high LD regions are usually called 'haplotype blocks' and the low LD ones are referred to as 'recombination hotspots.' Within a haplotype block, there is little or no recombination that occurs and the SNPs are highly correlated. Consequently, a small subset of SNPs (called tag SNPs or haplotype tagging SNPs) is sufficient to capture the haplotype pattern of the block. Using tag SNPs for association studies can greatly reduce the genotyping cost since it does not require genotyping all SNPs.

A number of methods have been proposed to find tag SNPs. These methods are mainly based on the following three models. The methods based on the first model try to identify a minimun set of LD bins such that SNPs within a bin are in high LD with each other (e.g. $r^2 \geq 0.8$) (Carlson *et al*., 2004). The second model assumes that the haplotype blocks have been delimited in advance, and these methods find a minimum set of SNPs which is able to distinguish each pair of haplotypes in a block (Patil *et al*., 2001; Zhang *et al*., 2002). The methods based on the third model usually assume that the number of tag SNPs is specified as an input parameter, and they identify tag SNPs which can reconstruct the haplotype of an unknown sample with high accuracy (Halperin *et al*., 2005; He *et al*., 2005).

We would like to note that methods based on the third model aim to find a set of SNPs which can predict the haplotype of an unknown sample with high accuracy. On the other hand, LD-based and block-based methods both focus on minimizing the number of tag SNPs for whole genome association studies (Crawfod and Nickerson, 2005). The LD-based methods identify tag SNPs that can represent other SNPs which are distant apart. The tag SNPs found by block-based methods are mainly used to represent SNPs in a continuous region (Hinds *et al*., 2005). However, the tag SNPs found by the LD-based methods may fail to distinguish all haplotypes in an LD bin[1]. But the tag SNPs found by block-based methods can distinguish all haplotypes in a block.

This paper studies the block-based model. In a large-scale study of human Chromosome 21, Patil *et al*. (2001) developed a greedy algorithm to partition the haplotypes into 4135 blocks with 4563 tag SNPs. Zhang *et al*. (2002, 2003, 2004a) used a dynamic programming approach to reduce the numbers of blocks and tag SNPs to 2575 and 3562, respectively. To avoid the influence of missing data, Huang *et al*. (2005) showed that there exists a set of SNPs, called robust tag SNPs, which is able to tolerate a fixed number of missing data. The problem of finding robust tag SNPs is the generalized version of the problem of finding tag SNPs. Both problems are known to be NP-hard (Garey and Johnson, 1979; Zhang *et al*., 2002; Huang *et al*., 2005).

The brute force method or the branch-and-bound method can find the optimal solution of these problems but may take infeasible time on large datasets (Huang *et al*., 2005; Zhao *et al*., 2005). On the

*To whom correspondence should be addressed.

[1]In practice, the tag SNPs found by LD-based methods may fail to represent other SNPs since the LD between two SNPs are usually set to a relative high but not to a perfect threshold (e.g. r^2 is 0.8 instead of 1.0).

other hand, approximation algorithms such as the greedy algorithm are fast on all datasets but may fail to find the optimal solution (Huang *et al*., 2005; Zhao *et al*., 2005; Zhang *et al*., 2004b). Here we briefly summarize the main ideas of the branch-and-bound and the greedy methods (Cormen *et al*., 2001).

- The concept of the branch-and-bound algorithm is to divide the feasible region of a problem to create smaller subproblems, cut out impossible ones and recursively solve the subproblems to get an optimal solution.
- The concept of the greedy method is to make a choice that appears to be the best at each step, and repeat this process until a feasible solution is found.

In this paper, we propose a hybrid method of the above two algorithms to solve the problem of finding robust tag SNPs. Our method adopts the idea of the branch-and-bound method to partition the original problem into a number of subproblems. Each subproblem is created by including or excluding some choices made by the greedy algorithm. The subproblems are then solved by a greedy method. By our method, we can explore larger solution space to obtain a better solution than a traditional greedy algorithm within a reasonable period of time. The trade-off between the efficiency and solutions of the algorithm can be explicitly adjusted by users.

We have implemented the algorithm and tested it on a variety of simulated and biological data. The experimental results indicate that this algorithm is quite efficient and the solutions are better than those of previous methods. Furthermore, our algorithm only spends few seconds to find optimal solutions in many cases. In other cases, our solutions can be very close to the optimal solutions without sacrificing the efficiency too much.

2 METHODS

We are given a haplotype block containing n SNPs and p haplotype patterns. Each SNP can distinguish some haplotype patterns from the others [see Fig. 1a and b]. To tolerate m missing SNPs, the problem of finding robust tag SNPs asks for a minimum set of SNPs such that each pair of haplotype patterns is distinguished by at least $m + 1$ SNPs (Huang *et al*., 2005). The problem of finding minimum tag SNPs is a special case of finding robust tag SNPs when setting $m = 0$.

The relation between the SNPs and the haplotype patterns can be represented as an n-sized set of sets D and a $\binom{p}{2}$-sized set E (Fig. 1c). Each element of E represents a pair of haplotype patterns. Each $D_i \in D$ is a subset of E and stands for pairs of patterns distinguished by the i-th SNP. For example, $D_1 = \{E_1, E_2, E_5, E_6\}$ in Figure 1c. We say that E_j is covered by D_i if $E_j \in D_i$. The problem of finding robust tag SNPs is finding a minimum set $C \subseteq D$ where every element of E must be covered by C at least $m + 1$ times. Here is the formal definition.

Given a set of sets D, a set $E = \bigcup_{D_i \in D} D_i$ and an integer m, choose a $C \subseteq D$ with minimum size that

$$\forall E_j \in E, \quad \sum_{D_i \in C} x_j \geq m + 1,$$

where

$$x_j = \begin{Bmatrix} 1 \mid E_j \in D_i \\ 0 \mid E_j \in D_i \end{Bmatrix}.$$

One previous approach to solve this problem is a greedy algorithm (Huang *et al*., 2005). It always selects a SNP which distinguishes most pairs of haplotypes at each step. Formally speaking, the greedy algorithm adds $D_k \in D$ to C at each step where $|D_k| = \max_{Di \in D} |D_i|$. In the following section, we introduce a novel concept of Greedy-Partition-Tree (GPT) for improving the previous greedy algorithm.

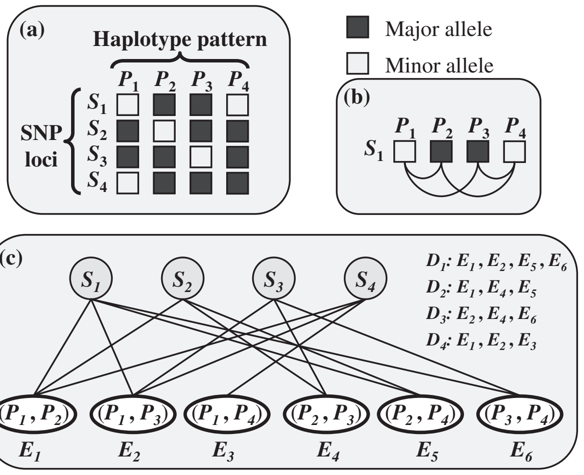

Fig. 1. (**a**) Four haplotype patterns with four SNPs. There are $\binom{4}{2}$ pairs of haplotype patterns which are (P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_3) and (P_2, P_4), (P_3, P_4). (**b**) Each SNP can distinguish some haplotype patterns from the others. For example, the SNP S_1 can distinguish between P_1 and P_2, P_1 and P_3, P_2 and P_4, and P_3 and P_4. (**c**) The relation between the SNPs and the haplotype patterns.

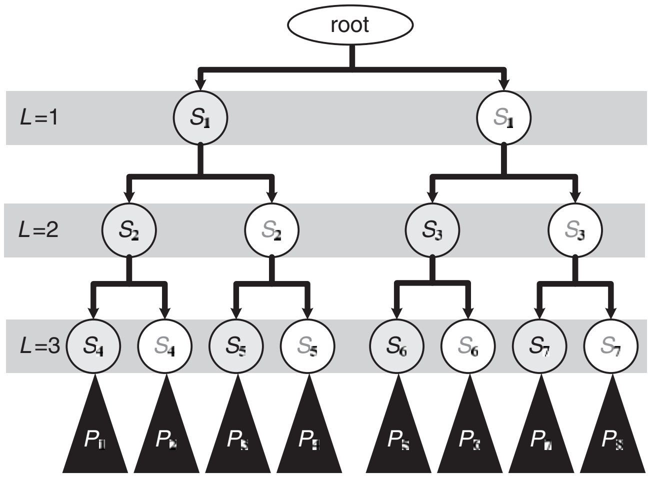
root
L=1
L=2
L=3
S_1 S_1
S_2 S_2 S_3 S_3
S_4 S_4 S_5 S_5 S_6 S_6 S_7 S_7
P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8

Fig. 2. A GPT with $L = 3$. Each gray node represents inclusion of a SNP while its sibling (white) represents exclusion of the same SNP. The black triangles represent subproblems partitioned by the GPT. The solution of each subproblem can be used to infer a feasible solution of the original problem.

2.1 Greedy-Partition-Tree

A GPT is a complete binary tree whose height is specified by a parameter L. Figure 2 illustrates an example of a GPT with $L = 3$. The GPT is constructed using the greedy algorithm described above. In the GPT, each internal node except the root stands for the inclusion or exclusion of a SNP selected by this greedy algorithm. Each path of length l from the root to an internal node represents a set of inclusion and exclusion of l SNPs. By including or excluding some SNPs, the GPT partitions the original problem into small ones. In the following, we describe the detailed steps for constructing a GPT. Note that initially the GPT has only one root node, which stands for the original problem.

- Step 1. For each leaf of the GPT, run the greedy algorithm to find a SNP S_i according to the constraint based on the path to the root. For example, when the height of the GPT is 2 in Figure 2, for the gray S_2 node we use the greedy algorithm to find a SNP (i.e. S_4) while S_1 and S_2 must be selected as tag SNPs. For the white S_2 node we find a SNP (i.e. S_5) according to the constraint where S_1 must and S_2 must not be selected as tag SNPs.
- Step 2. Branch two child nodes from each leaf. One child node represents the selection of S_i and the other means the de-selection of it. For the above example, we find SNP S_4 and branch two child nodes (the gray S_4 node and the white S_4 node) from the gray S_2 node. Both the child nodes inherit D and E from their parent except D_i is deleted. For the child node which represents the selection of S_i, some E_js are covered by D_i and E_j is deleted if E_j is covered $m + 1$ times.
- Step 3. If the height of the GPT is still less than L, go to Step 1.

After constructing the GPT, each leaf stands for a subproblem partitioned by the GPT. For example, P_1 in Figure 2 represents a subproblem where S_1, S_2 and S_4 have already been selected as part of the tag SNPs. A solution of P_1 plus S_1, S_2 and S_4 is a feasible solution of the original problem. On the other hand, a solution of P_2 plus S_1 and S_2 (without S_4) is a feasible solution of the original problem. We run the greedy algorithm to solve each subproblem P_i and obtain a set of feasible solutions of the original problem. The best solution among them is chosen as the output. The following is the pseudo code of our algorithm. The inputs E, D, m are as defined in the beginning of this section and are parameters of the adopted greedy algorithm. The parameter L is used to define the height of the GPT.

```
Algorithm: GPT
  Input: $E$, $D$, $m$, $L$
  Output: A subset of $D$ which can cover $E$ for $m + 1$ times
  Build a tree $T$ with only one node root
  for $i = 1$ to $L$ do
    for each leaf at level $i - 1$ of $T$ do
      1. Run the greedy algorithm to find a SNP $s$ according to the
         constraints on the path from leaf to root
      2. Branch two nodes from leaf
        2.1 The left child is a constraint that SNP $s$ must be
            selected as a tag SNP
        2.2 The right child is a constraint that SNP $s$ mustn't be
            selected as a tag SNP
  end for
      /* $T$ becomes a tree of height $i$ */
  end for
  for each leaf of $T$ do
  Run the greedy algorithm to find a feasible solution according
  to the constraints on the path to root
  end for
  Compare all feasible solutions and return the best one
```

The solution of the GPT is at least as good as that of a greedy algorithm because the solution of the lef most subproblem (i.e. P_1 in Fig. 2) plus the SNPs along the path (i.e. S_1, S_2 and S_4) to the root is exactly a greedy solution. The greedy algorithm is able to find a solution within a factor of $\ln((m + 1)|E|)$ of the optimal solution (Huang *et al*., 2005). Hence, the solution found by GPT is also guaranteed to have the same approximation ratio.

The GPT provides the flexibility for the user to balance the efficiency of the program and the quality of the solution by adjusting L. When $L = 0$, the algorithm is exactly the same as a greedy algorithm. By increasing L, the solution of GPT can be improved but the running time may also increase. When $L = |S|$, GPT becomes a branch-and-bound algorithm and the solution found is the optimal solution. From our empirical studies, the solution of GPT is significantly improved by increasing L and the optimal solution can even be found in many datasets with small L.

2.2 Improvement of efficiency

To improve the efficiency of our algorithm, two considerations are undertaken. One is to accelerate the greedy algorithm and the other is to prune unnecessary branches. The running time of the greedy algorithm can be reduced by adopting a heap structure (Cormen *et al*., 2001). The heap structure is used to speed up the step of selecting a SNP from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$. The greedy algorithm runs in time $\mathcal{O}(|E|n \ \log n.)$ and the complexity of GPT is thus $\mathcal{O}(2^L.|E|n \log n)$.

Just as the branch-and-bound algorithm cuts out impossible subproblems, we can also prune unnecessary branches from the GPT to skip some impossible solution space. To prune unnecessary branches, we compute the lower bound of a subproblem by modifying the approach of Zhao *et al*. (2005). The lower bound is compared with the current best solution. The subproblem will not be solved further if its lower bound is not better than the current best solution.

2.3 Improvement of solutions

When building a GPT, there could be many redundant SNPs which distinguish the same pairs of haplotype patterns. If two nodes in distinct subtrees of the GPT include the same type of redundant SNPs, they may lead to the same subproblem. For example, if $m = 0$ and S_1 and S_3 in Figure 2 distinguish the same pairs of haplotype patterns, S_2 is equal to S_6 because we use the same greedy algorithm under the same constraint. Therefore, the solution found in P_1 plus S_4 must be equal to the solution found in P_5. It takes place easily because redundant SNPs have the same priority to be selected by the greedy algorithm. Therefore, before constructing the GPT, we group SNPs that distinguish the same pairs of haplotypes. In each group of SNPs, the number of redundant SNPs selected as robust tag SNPs is at most $m + 1$. The extra SNPs could be deleted from each group without affecting the optimal solution. The deletion of these SNPs not only increases the efficiency, but also increases the opportunity to find a better solution.

3 EXPERIMENTAL RESULTS

We have implemented the GPT in C and run the program on a PC with a 2.4 GHz CPU and 256 MB memory. The data we tested include a variety of simulated and biological data. We compared the results of our program with those of a greedy algorithm, an LP-relaxation algorithm and a brute force algorithm for searching the optimal solution proposed by Huang *et al*. (2005). These algorithms are referred to as ‘greedy,’ ‘LP’ and ‘OPT’ respectively in the following.

3.1 Experimental results on simulated data

The first set of simulated data is generated by randomly assigning the major or minor alleles to each SNP on a haplotype. This simulation considers the bottleneck situation where all SNPs reach complete linkage equilibrium. We generate 100 datasets and each of them contains 10 haplotypes with 40 SNPs. The results of ‘greedy,’ ‘LP’ and ‘OPT’ are compared with our GPT of $L = 10$. Figure 3 plots the average numbers of robust tag SNPs found by each algorithm for tolerating different number of missing SNPs.

The optimal solutions for $m > 1$ cannot be found by OPT within a reasonable period of time and are not shown in this figure. In this experiment, the GPT only takes 6 seconds to find the optimal solutions when $m = 1$. As m increases, GPT significantly outperforms all other algorithms and the solutions can be found in seconds.

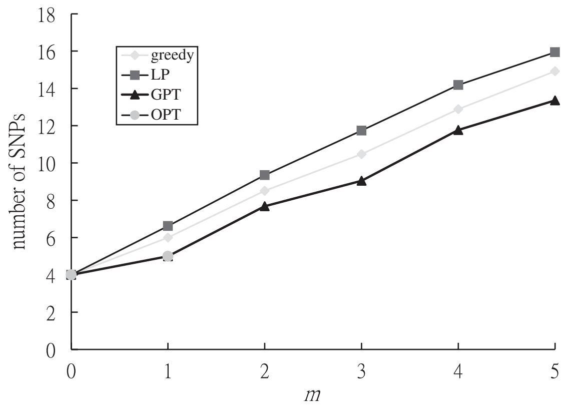

Fig. 3. Experimental results on random data. The *x*-axis stands for the number of missing SNPs to be tolerated (i.e. *m*). The *y*-axis stands for the average numbers of robust tag SNPs found by different algorithms, 'greedy,' 'LP,' 'GPT' and 'OPT' over 100 datasets.

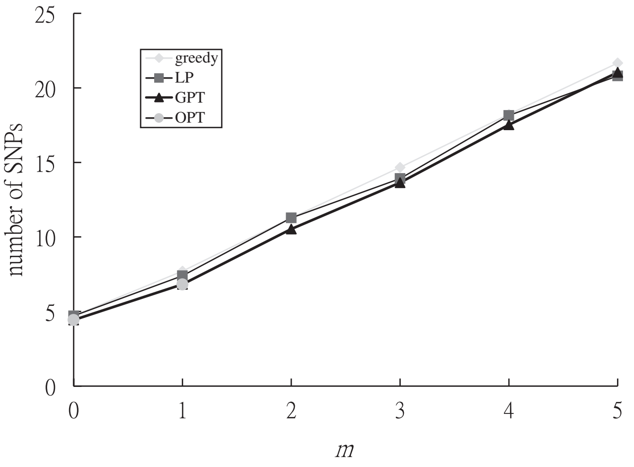

Fig. 4. Experimental results on Hudson's data. The *x*-axis stands for the number of missing SNPs to be tolerated (i.e. *m*). The *y*-axis stands for the average numbers of robust tag SNPs found by different algorithms, 'greedy,' 'LP,' 'GPT,' and 'OPT' over 100 datasets.

The second set of simulated data is generated by Hudson's program (Hudson, 2002) which can simulate a set of haplotypes under the assumption of neutral evolution and uniformly distributed recombination rate using the coalescent model. We use Hudson's program to generate 100 datasets and each of them contains 10 haplotypes with 40 SNPs. The results of this experiment are shown in Figure 4. The parameter *L* of the GPT is also set to 10 in this experiment.

The optimal solutions for $m > 1$ again cannot be found within a reasonable period of time. The experimental result indicates that GPT is the only algorithm which finds the optimal solution when $m = 1$. When *m* increases, the GPT still outperforms other algorithms. It only takes several seconds for GPT to output a solution. The numbers of robust tag SNPs found by each algorithm are more than those of randomly generated data. It is because the coalescent haplotypes generated by Hudson's program are similar to each other. A SNP can distinguish fewer haplotypes on average. Thus, we need more SNPs to construct a feasible solution. In this experiment, the numbers of robust tag SNPs found by all algorithms are not too much different. It is because that most SNPs in these datasets distinguish similar haplotypes. The space of improvement is relatively small since the amount of distinct SNPs that can be chosen by GPT is insufficient.

Table 1. Comparison of the numbers of tag SNPs produced by four algorithms on Patil's data

Algorithm	$m = 0$	$m = 1$	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$	$m = 7$
Greedy	1461	1992	2035	1878	1801	1567	1473	1329
LP-relaxation	1508	1916	2059	1830	1799	1538	1475	1310
GPT	1446	1913	2011	1826	1779	1535	1462	1310
OPT	1446	1913	2011	X	X	X	X	X

m is the number of missing data tolerated. The parameter *L* of GPT is set to 12 in this experiment. The *X* means that the solutions cannot be found by OPT in feasible time.

3.2 Experimental results on biological data

We first test these algorithms on public haplotype data from Patil *et al*. (2001). Patil's data include 20 haplotypes of 24 047 SNPs spanning over ~32.4 MB on human Chromosome 21, which are partitioned into 4135 haplotype blocks. Each block contains 2–7 haplotypes and each haplotype has 1–114 SNPs. The blocks with only two haplotypes are not included in our experiments because any SNP could be the tag SNP and the optimal solution can be easily found by any algorithm. Therefore, there are 612 haplotype blocks tested in our experiments. We apply the GPT of $L=12$ and other algorithms on these haplotype blocks. The numbers of total robust tag SNPs found by these algorithms are listed in Table 1.

The total number of robust tag SNPs decreases as *m* goes large because many blocks do not have enough SNPs for tolerating large missing data. The optimal solutions for $m > 2$ cannot be found within a reasonable period of time and are not listed in the table. In this experiment, the GPT is the only algorithm that finds the optimal solutions for $m \leq 2$. The solutions of the GPT are still better than the other two algorithms as *m* goes large. However, the outperformance of the GPT is not obvious because of numerous small blocks in Patil's data. The optimal solutions of these short blocks can be easily found by all algorithms. As a result, the GPT fails to find better solutions in these short blocks and the improvement is not significant.

We then test these programs on the whole-genome haplotype data from Hinds *et al*. (2005). Hinds *et al*. (2005) genotyped 1 586 383 SNPs in 71 Americans of European, African and Asian ancestry and inferred haplotypes from these diploid genotype data. The inferred haplotypes were partitioned into blocks separately for each of the three population samples. Here we choose the sample of African-American as our experimental target. There are 22 chromosomes plus the X and Y chromosomes partitioned into 235 771 haplotype blocks and we apply the GPT of $L = 3$ on the haplotype blocks of each chromosome. We list the number of tag SNPs (i.e. $m = 0$) of all chromosome found by different algorithms in Table 2. We can see that the the solutions found by GPT are better than those of greedy and LP-relaxation algorithms. In fact, GPT finds optimal solutions of all chromosomes when *L* is only set to 3. In addition, the GPT takes only 15 s to find solutions of all chromosomes.

Table 2. Comparison of the numbers of tag SNPs produced by four algorithms on Hinds's data when $m = 0$

Algorithm	chr1	chr2	chr3	chr4	chr5	chr6	chr7	chr8	chr9	chr10	chr11	chr12
Greedy	47 346	53 364	38 478	41 446	39 893	35 065	32 387	36 254	23 615	28 678	28 181	26 244
LP-relaxation	47 269	53 255	38 414	41 388	39 821	35 019	32 309	36 192	23 586	28 607	28 149	26 196
GPT	47 193	53 165	38 338	41 310	39 753	34 953	32 254	36 146	23 549	28 570	28 098	26 149
OPT	47 193	53 165	38 338	41 310	39 753	34 953	32 254	36 146	23 549	28 570	28 098	26 149
Algorithm	**chr13**	**chr14**	**chr15**	**chr16**	**chr17**	**chr18**	**chr19**	**chr20**	**chr21**	**chr22**	**chrX**	**chrY**
Greedy	25 174	21 592	19 000	17 937	14 438	20 021	7654	13 973	10 421	8503	14 119	151
LP-relaxation	25 130	21 553	18 951	17 892	14 410	19 992	7649	13 958	10 393	8486	14 079	151
GPT	25 083	21 527	18 917	17 869	14 396	19 963	7637	13 934	10 383	8473	14 056	151
OPT	25 083	21 527	18 917	17 869	14 396	19 963	7637	13 934	10 383	8473	14 056	151

The parameter L of GPT is 3. The GPT finds the optimal solutions for all chromosomes.

In all experiments, the GPT can nearly find the optimal solutions when $m \leq 2$ and only takes a few seconds to finish execution. In comparison with the greedy algorithm, the GPT spends more time to explore larger solution space and thus can find better solutions than the traditional greedy algorithm.

4 DISCUSSION

4.1 Trade-off between efficiency and solutions of GPT

In this subsection, we discuss the trade-off between efficiency and solutions of GPT. By increasing L, the solutions of GPT can be improved but the efficiency may be sacrificed. The efficiency of GPT is measured by the elapsed time to run datasets with different L. The solution improved by different L is measured by improved ratio, which is defined as $(I_0 - I_k)/I_0 * 100\%$ where I_k is the number of robust tag SNPs found by GPT with $L = k$.

Figure 5a plots the improved ratio with respect to L for the experiment on 100 randomly generated datasets (i.e. the datasets used in Fig. 3). As $L = 10$, all the curves start to converge, because the solutions found by the GPT are quite close to the optimal solution. For example, the improved ratio of the curve with $m = 1$ stops growing at 15.68% because the optimal solution has been found. This phenomenon indicates that setting $L = 10$ can gain the most improvement in solutions for the GPT. The solution of GPT is not significantly improved as $L > 10$.

We then plot the efficiency deterioration of increasing L on Figure 5b. The x-axis stands for L and the y-axis stands for the total elapsed time of GPT running on these 100 datasets. It can be observed that when L increases by 1, the elapsed time is ~1.6–1.8 times of the original running time. Furthermore, the GPT only takes <50 s to run these datasets with different m when $L \leq 10$. As a result, the parameter L of GPT is best set to around 10 to obtain the best improvement in solutions, which still keeps the running time within a reasonable period of time.

Figure 5c plots the improved ratio of GPT on the 100 datasets generated by Hudson's program, and Figure 5d plots the corresponding elapsed time of GPT with different L on these datasets. As L fixes, the improved ratio of GPT on Hudson's data is less than that on random data. In Figure 5c, most curves converges around $L = 12$. Although the GPT has already found the optimal solution, its improved ratio is only 12.5%, which is less than that of previous experiment. The reason is that the space for improvement is not much on Hudson's datasets (see Section 3.1). On the other hand, we find that the GPT is more efficient when running on Hudson's datasets than on random datasets. For example, it only takes <30 s to run the datasets with different m when $L = 10$. It is because the running time of GPT is sensitive to the number of haplotypes distinguished by each SNP (see Section 2.2). The SNPs generated by Hudson's program can distinguish only a few haplotypes. Consequently, the GPT is quite efficient in this experiment. This implies that we can set larger L for GPT to obtain better improvement of solutions on Hudson's datasets.

Figure 5e depicts the improved ratio of GPT on Patil's data and Figure 5f plots the corresponding elapsed time for L ranging from 0 to 14. In Figure 5e, all curves start converging around $L = 8$. The improved ratio for $m = 1$ and $L = 14$ is only 5.53%, which is much less than the above two experiments. As mentioned in Section 3.2, there are many short blocks in Patil's data and all algorithms nearly find the optimal solution in short blocks. The improvement of GPT is amortized among numerous short blocks and thus the improved ratio is smaller. However, in Figure 5f, we observe that the elapsed time of GPT for every curve is <1 min, even when $L = 14$. This owes to the fact that the common haplotypes in Patil's haplotype blocks are rare. Most blocks contain less than five common haplotypes. Since the time complexity of GPT is relative to the number of haplotypes in the block, it is quite fast when running on Patil's datasets. This phenomenon suggests that we can set larger L for GPT on Patil's data without sacrificing the efficiency too much.

In the experiment of Hinds's data, the GPT finds the optimal solutions for all chromosomes when $L = 3$. Table 3 lists the total number of tag SNPs found by GPT on all chromosomes for different L. The optimal solutions can be found with small L because most blocks contain <10 SNPs. However, owing to the huge amount of blocks across all chromosomes, the GPT spends most of the time on I/O instead of real computation. As a result, the elapsed time of GPT is completely dominated by the I/O time instead of computation time. Therefore, when L increases from 0 to 3, the elapsed time of GPT does not reveal the exponential growth as expected.

In all experiments, the improved ratio of the GPT decreases as m becomes large. It is because a larger amount of SNPs are required to tolerate more missing data. However, in each haplotype block,

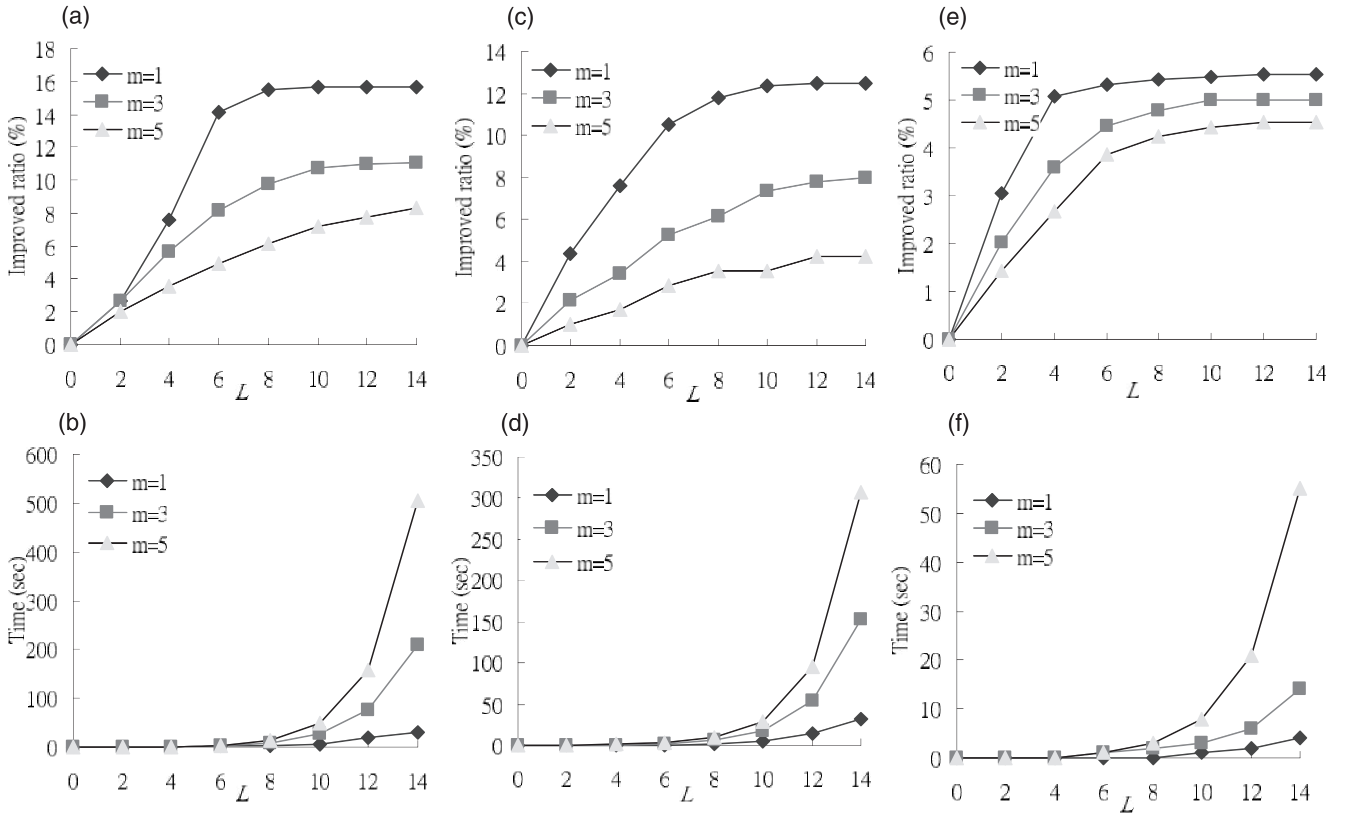

Fig. 5. Experimental results for different L values. (**a**) Improved ratio on random data. (**b**) Elapsed time on random data. (**c**) Improved ratio on Hudson's data. (**d**) Elapsed time on Hudson's data. (**e**) Improved ratio on Patil's data. (**f**) Elapsed time on Patil's data. In (a), (c) and (e), the x-axis stands for the parameter L. The y axis stands for the improved ratio of solutions of related L. Let the number of SNPs found by GPT with $L = 0$ be I_0 and the number of SNPs found by GPT with $L = k$ be I_k. The improved ratio is computed by $(I_0 - I_k)/I_0 * 100\%$. In (b), (d) and (f), the x-axis stands for L and the y-axis stands for the elapsed time needed to run 100 datasets.

Table 3. Experimental results for different L values on Hinds's data

	$L = 0$	$L = 1$	$L = 2$	$L = 3$
Number of tag SNPs	603 934	601 885	601 868	601 867
Elapsed time (s)	12	12	13	15

The program spends most time on I/O because there are a huge amount of blocks which consist of only a few SNPs.

the number of SNPs is fixed. When m becomes large, any feasible solution has to use more SNPs in the block, which implies that there are fewer SNPs that remain. Usually, the remaining unselected SNPs can only distinguish fewer haplotypes than the selected ones. Thus, the GPT has less space for improvement over any feasible solution.

On the other hand, the elapsed time also increases as m becomes large. One intuitive reason is that the GPT has to find more SNPs to tolerate more missing data. In addition, the running time of GPT is also proportional to the frequency of the adjustment of the internal heap (see Section 2.2). As m becomes larger, the frequency of the adjustment of the heap also increases. This is also a common phenomenon that happened to each algorithm. For example, the OPT program fails to output the optimal solution for $m > 2$ in most experiments. A feasible set of robust tag SNPs for tolerating missing data is usually more difficult to be found as m increases.

Since the GPT provides the flexibility to adjust L to balance efficiency and solutions, choosing a proper L is another important issue. From our empirical study, the proper choice of parameter L for GPT heavily depends on the types of datasets. In the experiment of randomly generated data, if the running time of GPT is up to 1 min, the L can be best set to 10 and we can gain significant improvement of solutions. On the other hand, for the datasets on Hudson's and Patil's data, the GPT is quite efficient in these experiments. Therefore, we can set larger L (e.g. $L = 18$) for GPT to gain sufficient improvement of solutions. As for the haplotype blocks with only a few SNPs, the GPT is able to find the optimal solution with small L (e.g. L is only set to 3 in Hinds' datasets).

4.2 Comparison with MLR-tagging

This subsection compares GPT with a method based on the third model. The method we choose is called MLR-Tagging which selects a set of SNPs to predict the haplotype of an unknown sample with high accuracy. The MLR-Tagging program is based on multivariate least square prediction and is superior to its previous version which adopts the linear reduction method (He *et al*., 2005). The program was downloaded from http://alla.cs.gsu.edu/~software/tagging/tagging.html. The MLR-tagging program requires that the number of tag SNPs is specified as an input parameter and the output is a set of tag SNPs with the specified size. We set the parameter '1' and run the program iteratively by increasing the parameter by one at a time until the selected tag SNPs can distinguish all haplotype patterns. The final parameter is treated as the number of tag SNPs produced by MLR-Tagging when $m = 0$.

We test GPT and MLR-Tagging on simulated datasets used in Section 3. In the experiments on the 100 random datasets, the total numbers of tag SNPs found by MLR-Tagging and GPT are 579 and 400, respectively. In the experiments on the 100 Hudson's datasets, the MLR-Tagging requires 762 tag SNPs and the GPT only needs 443 tag SNPs. However, it should be noted that the objectives of these two methods are quite different. MLR-Tagging aims to find tag SNPs for predicting unknown haplotypes and to minimize the prediction error. On the other hand, GPT focuses on finding a minimum set of tag SNPs which can distinguish all haplotype patterns. Therefore, this comparison is not completely fair. If the number of tag SNPs or the genotyping cost is the primary concern, GPT is a more suitable approach than MLR-Tagging.

5 CONCLUSION

In this paper, we propose a hybrid method called GPT to solve the problem of finding robust tag SNPs by combining the ideas of the branch-and-bound method and the greedy algorithm. The original problem is partitioned into a fixed number of subproblems, and each subproblem is then efficiently solved by a greedy algorithm. The GPT explores larger solution space than a traditional greedy algorithm and thus can obtain a better solution. In addition, the GPT offers the flexibility for the user to adjust the running time to approach the optimal solution as close as possible. The experimental results indicate that the GPT outperforms several existing methods and can find solutions quite close to the optimal solutions in feasible time. The GPT can also benefit from the parallel computation because the partitioned subproblems can be solved independently. In fact, the GPT is a general idea that can be used to adapt different greedy algorithms to solve their corresponding problems. The solutions found by GPT is guaranteed to be at least as good as the original greedy algorithms for the problem.

ACKNOWLEDGEMENTS

We thank the reviewers for their helpful comments. C.-J.C., Y.-T.H. and K.-M.C. were supported in part by NSC grants 93-2213-E-002-029 and 94-2213-E-002-091 from the National Science Council, Taiwan.

Conflict of Interest: none declared.

REFERENCES

Altshuler,D. *et al*. (2005) A haplotype map of the human genome. *Nature*, **437**, 1299–1320.

Bafna,V., Halldórsson,B.V., Schwartz,R., Clark,A.G. and Istrail,S. (2003) Haplotypes and Informative SNP Selection Algorithms: Don't Block Out Information. In *Proceedings of the Research in Computational Molecular Biology (RECOMB)* Berlin, Germany, pp. 19–27.

Carlson,C.S. *et al*. (2004) Selecting a maximally informative set of single-nucleotide polymorphisms for association analyses using linkage disequilibrium. *Am. J Hum. Genet*., **74**, 106–120.

Cormen,T.H., Leiserson,C.E., Rivest,R.L. and Stein,C. (2001) *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Crawfod,D.C. and Nickerson,D.A. (2005) Definition and clinical importance of haplotypes. *Annu. Rev. Med*., **56**, 303–320.

Daly,M.J. *et al*. (2001) High-resolution haplotype structure in the human genome. *Nat. Genet*., **29**, 229–232.

Garey,M.R. and Johnson,D.S. (1979) *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, NY.

Halperin,E. *et al*. (2005) Tag SNP selection in genotype data for maximizing SNP prediction accuracy. *Bioinformatics*., **21** (Suppl. 1), i195–i203.

He,J. *et al*. (2005) Linear reduction methods for tag SNP selection. *Int. J. Bioinformatics Res. Appl*., **1**, 249–260.

Helmuth,L. (2001) Genome research: map of the human genome 3.0. *Science*, **293**, 583–585.

Hinds,D.A. *et al*. (2005) Whole-genome patterns of common DNA variation in three human populations. *Science*, **307**, 1072–1079.

Huang,Y.T. *et al*. (2005) Selecting additional tag SNPs for tolerating missing data in genotyping. *BMC Bioinformatics*, **6**, 263.

Hudson,R.R. (2002) Generating samples under a Wright–Fisher neutral model of genetic variation. *Bioinformatics*, **18**, 337–338.

Patil,N. *et al*. (2001) Blocks of limited haplotype diversity revealed by high-resolution scanning of human chromosome 21. *Science*, **294**, 1719–1723.

Zhang,K. *et al*. (2002) A dynamic programming algorithm for haplotype block partitioning. *Proc. Natl Acad. Sci. USA*, **99**, 7335–7339.

Zhang,K. *et al*. (2003) Haplotype block partition with limited resources and applications to human chromosome 21 haplotype data. *Am. J. Hum. Genet*., **73**, 63–73.

Zhang,K. *et al*. (2004a) Haplotype block partition and tag SNP selection using genotype data and their applications to association studies. *Genome Res*., **14**, 908–916.

Zhang,P. *et al*. (2004b) A double classification tree search algorithm for index SNP selection. *BMC Bioinformatics*, **5**, 89.

Zhao,W. *et al*. (2005) Efficient RNAi-based gene family knockdown via set cover optimization. *Artif. Intell. Med*., **35**, 61–73.