Hybrid Heuristic for the Maximum-Weight Independent Set Problem

László Hárs*

November 30, 1989

Abstract. We propose an approximate algorithm for finding the maximum-weight independent set of a given graph of n nodes with associated weights. It combines three different heuristics, and its running time is $O(n^2)$ in the practice.

We also propose a set of random graphs for testing and comparing the maximum-weight independent set algorithms. This graphs are easy to identically generate on any computer capable performing only 32-bit integer arithmetic.

Key words. Maximum-weight independent set problem, Random search, Local search.

1. Introduction

The maximum-weight independent set problem (MxWISP) is to find one of the subsets of the n nodes of a given graph (with associated weights to the nodes), in which subset no two nodes are adjacent and the sum of the weights of the nodes is the possible maximum. It is known (see [9]) that this problem, even if all the weights are all equal to 1 is NP-complete. It means that there is no hope to find an exact algorithm to solve it in time less than a polynomyal of n. The best known exact algorithm (see [10]) uses $O(2^{n/3})$ time, what is too long if n is larger than a few dozens.

In [11] a set of approximate algorithms is published for the case of unity weights, running in time $O(n^k)$ with all constants $k > 1$. However, if n is really large (more than a few hundreds), we can not usually afford a running time larger than $O(n^2)$. Therefore we modified the above algorithm (case $k = 2$) to handle weighted nodes. It gives quite good (large weight) independent set, but these sets often can be easily improved. This is why we followed it by an add-exchange heuristic step.

The algorithm is practically a local search heuristic and so it gets stuck at a local optimum. To increase the chance to get to the global maximum, or at least to find a larger local one, we restart the algorithm a few times. To insure that the new set will be different from all the previous ones we permanently delete from the graph a node

* University Bonn, Institute for Operations Research and
Eötvös Loránd University, Budapest, Department of Computer Science.

of the last found independent set, and then restart the algorithm. The deleted node is the one which is the least likely to appear in the global maximum of the weighted independent sets: its and its non-neighbors' weight give the least sum.

Theoretically it would be better to delete a minimum set of nodes before each restart of the algorithm, which set has at least one node in common with the previously found independent sets. It requires more running time, more complex algorithm and in the case of our test problems and some large real life problems no improvement was found in the quality of the solution.

2. Greedy deletion heuristic

We repeatedly delete the "worst" node v from the graph, until it becomes independent. The worst node v, in our case, is the one which has the smallest sum of weights of itself and of the nodes non-adjacent to v.

Remark: the other way around, to add the "best" node to the independent set (starting from a single node set) until the set remains independent, is not that good. If we ever make a poor choice it degrades the solution significantly. In the long sequence of additions the probability to choose a bad node is high. On the other hand, the deletion procedure used in our algorithm is not so sensitive. Deleting a good node usually leaves some almost as good ones in the graph.

However, the addition heuristic is useful as a second stage of an algorithm, when we can add only a few nodes to the independent set. This idea is extended in the next section.

3. k-opt heuristic

(We adopt the notation of a traveling salesman heuristic which shows some similarity to our algorithm. See [3] or [4].)

If an independent set S of nodes is already given (it may be a single node at start) we try to improve it. Choose a set A of k independent nodes not in S. Let B be the the set of nodes in S adjacent to at least one node in A. If the total weight of A is larger then that of B, then exchange their role in S ($S \leftarrow S - B + A$, i.e., delete the nodes of B from S and add the nodes of A).

Repeat the above attempt to improve for all the independent k-sets A of nodes not in S, until an improvement is found. If no improvement is found the k-opt procedure terminates, otherwise we restart it with the new independent set S.

There is no easy way known to generate all the independent k-sets for arbitrary k and usually there are to many of them if $k > 2$. Therefore we apply only the case of $k = 1$, when all the nodes outside of S are tried to add to S or to replace its neighbors in S.

The case of $k = 2$ requires much more time, but results a better solution. For each given problem we have to decide whether this extra time is allowable.

3.1. Simulated annealing

In the k-opt algorithm the order in which the k-subsets are generated can be arbitrary. At termination we reach a local maximum. To increase the chance to get to a better one, we may restart the algorithm several times with different order of the subset generation. This will be discussed for the whole algorithm in the next section. Another possibility is the use of the simulated annealing technique. (See [5].)

In our case it means the following: If a k-subset A gives an improvement, we perform the corresponding exchange of nodes. Otherwise we generate a random number in the range of $[0, 1]$, and perform the exchange only if the random number is larger then a certain limit p. This p is actually a function of two things: the decrease in the cost of the independent set if the exchange with set A would be done, and the number of iterations the algorithm has made. (Larger weight decreases should be made more unlikely to be accepted than smaller ones, and with the number of iterations the probability to perform a weight decreasing exchange should decrease.)

With suitable definition of this function p this algorithm can provide very good results, especially when the graph has much symmetry, but we do not know an algorithmic way to define a good one. Each problem may require a different definition. This is why we have not applied the simulated annealing in our final version of the general algorithm.

4. Restart

The second step (the k-opt heuristic) can give only a limited amount of improvement, since the first step (the greedy deletion) is already quite good. Therefore it is not wise to spend much time to improve only the second step.

We restart several times the whole algorithm. We need some insurance to avoid that the same solution will be repeated. We applied the simplest solution: before each restart of the algorithm we permanently delete one node of the last found independent set from the graph. If the graph has really many nodes (more than some hundreds) we can afford only some dozens of restarts. These are too few to drastically affect the structure of the graph, so more complex solutions, like deleting a minimal set of nodes which intersect all the previously found independent sets usually do not result in larger weight independent sets.

The permanently deleted node is the one which has the smallest weight free set in the graph, i.e., the weight of it and of the nodes not adjacent to it is the smallest. It is likely a member of smallest weight independent sets than the other nodes.

5. Running time

6. Test problems

We need a set of problems which can be identically generated on any computer, enabling us to compare algorithms. We use a simple random number generator which gives the same sequence of (pseudo) random integers on all the computers capable to perform 32-bit integer arithmetic operations.

6.1. Random number generator

The random number generator must use only integer arithmetic, since the floating point operations may give different results depending on the precision of the floating point arithmetic unit, the rounding rules applied, etc. We have chosen the simple congruency method from [8]: $X_{n+1} = (1 + 1664525 \times X_n) \bmod 2^{32}$. It gives quite a good sequence according to many statistical tests. (The use of two's complement representation of the integers does not make a difference. See [8].)

Avoiding the floating point operations makes a bit complex to transform this sequence to numbers of a given interval $[1..m]$. For simplicity, we confine ourselves to use only intervals with $m < 2^{15}$. The following Pascal procedure does the job. (Note that the division and multiplication with 2^{16} are used only to access the upper and lower half of the 32-bit machine word. In practice they can be replaced by machine dependent simple statements. Optimizing compilers, such as IBM's VS-PASCAL may do this automatically. We assume that the overflow of the integer multiplication is ignored at the first statement of the procedure and the least significant digits are given as the result. With some compilers one may need turn off overflow checking or simulate the 32-bit multiplication with multiplying the upper and lower half words and combine the results.)

```
Function Rand( Mx: Integer): Integer;                          {Mx < 32768}
Const    TwoTo16 = 65536;
Var      Half1, Half2: Integer;
Begin
  RndNmb := 1 + 1664525 * RndNmb;                {Global 32 bit integer}
  Half1 := RndNmb div TwoTo16;
  If Half1 < 0 Then Half1 := TwoTo16 + Half1;
  Half2 := RndNmb mod TwoTo16;
  Rand := 1 + ( Half1 * Mx + (Half2 * Mx) div TwoTo16 ) div TwoTo16;
End;
```

If it is used on computers with larger precision, the first executable statement should be replaced by

```
  RndNmb := (1 + 1664525 * RndNmb) mod TwoTo32;
```

with constant `TwoTo32` $= 2^{32}$.

6.2. Random test problems

We generate graphs of n nodes with known maximum weight independent set of cardinality m, because then we know how good solutions are given by our algorithms. 4 parameters are supplied by the user: $(n, m, e, seed)$, where e extra edges are added

to the n-node m-class graph, generated by the random number generator `Rand` starting with first random number *seed*.

1) Partition the set $\{v_1, \ldots, v_n\}$ of the nodes into m random classes, by choosing m random separation points from the possible $n-1$ ones. ($k_i \leftarrow \texttt{Rand}(n-1), i = 1, \ldots, m$. If the value of k_i has been already selected, we drop it and keep generating random numbers until an unused is found.)
2) Draw all the edges between the nodes of the same classes: $(v_{k_i+1}, \ldots, v_{k_{i+1}}), i = 0, \ldots, m; k_0 = 0; k_{m+1} = n$.
3) Assign random weights from the interval $[1, n]$ to the nodes.
4) Select the maximum weight node in each class of more than two nodes. Connect all pairs of these nodes by an edge.
5) Mark the nodes of the maximum weight independent set, save its total weight (see below).
6) Attempt to draw e more random edges. If the edge to be drawn already exists or both endpoints are marked, we forget about this edge. (If e is too large, this way we avoid falling into infinite cycle.)
7) Put the nodes in random order (generating a random permutation). This step effectively hides the simple structure of our graphs.

Lemma. *At step 5) the maximum weight independent set of nodes is given by the following: Take the nodes of all the single element classes. For all larger classes calculate the difference of the largest and the second largest weight. From the class where this difference is the biggest, take the largest weight node, from all other classes, take the second largest weight node.*

Proof. Step 2) ensures that at most one node from a class can participate in any independent set. Step 4) allows only one of the heaviest nodes of the classes (of at least 2 nodes) to appear in any independent set.

Construct a set of all the nodes of the single element classes and of the second largest weight nodes of the other classes. If we exchange in a class the second heaviest node to the heaviest one, we increase the total weight. The largest increase was requested by the Lemma.

Look the graph as it was after step 4). Any independent set can be transformed to the one described by the Lemma by adding nodes (if it has no node from a class), exchanging a node to the second heaviest one (if it has a node of smaller weight than the second largest weight) or exchange the heaviest node with the second heaviest one in exactly two classes (if not the largest difference class gives its heaviest node to the independent set). All these operation increase the weight of the independent set, and the final set is independent even after step 7), we are done. ■

7. Experiments

References

[1] Y. Rossier, M. Troyon, Th. M. Liebling "Probabilistic exchange algorithms and Euclidean traveling salesman problems," *OR Spectrum* **8** (1986), pp. 151–164.

[2] R. E. Tarjan "Data structures and network algorithms," *CBMS-NSF Regional Conference Series in Applied Mathematics* **44** (1983).

[3] B. L. Golden and W. R. Stewart "Empirical analysis of heuristics," in The Travelling Salesman Problem *edited by* E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan and D. B. Shmoys *John Wiley & Sons, Chichester*, **1986**, pp. 207–249.

[4] C. H. Papadimitriou, K. Steiglitz "Combinatorial Optimization: Algorithms and Complexity," *Prentice-Hall* **1982**.

[5] S. Kirkpatrick, C. D. Gelatt, Jr, M. P. Vecchi "Optimization by simulated annealing," *Science* **220, No. 4598** (1983), pp. 671–680.

[6] A. V. Aho, J. E. Hopcroft and J. D. Ulman "The design and analysis of computer algorithms," *Addison-Wesley, Reading, MA* **1974**.

[7] — "Reversible segment list," *(To appear)* **1989**.

[8] D. E. Knuth "The art of computer programing. Volume 2 (Seminumerical algorithms," *Addison-Wesley, Reading, MA* **1981**.

[9] M. Garey, D . Johnson "Computers and intractability: A guide to the theory of NP-completeness," *W. H. Freeman & Co., San Francisco* **1979**.

[10] R. E. Tarjan, A. E. Trojanowski "Finding a maximum independent set," *SIAM J. Comput.* **Vol. 6. No. 3.** (1977), pp. 537–546.

[11] M. Carter "A practical algorithm for finding the largest clique in a graph," *Dept. of Industrial Engineering, University of Toronto. Working Paper* **#84-08** (1984).