Exact Algorithm for Independent Set

2017-09-12, rev. b33ef67

Independent Set

Consider an undirected unweighted graph $G = (V, E)$ and set $n = |V|$. The Maximum Independent Set problem (MIS) is to find a subset S of the vertices of largest size such that no pair of different vertices in S are connected by an edge in G. The size of the largest independent set in G is denoted by $\alpha(G)$.

MIS is NP-hard, but we will nevertheless implement algorithms that compute $\alpha(G)$ for small instance graphs G and argue both empirically and theoretically about their running time.

Notation. For a subset $X \subseteq V$, we will by $G[X]$ denote the graph *induced* by X, the graph that remains if we keep only the vertices in X and the edges among them. For a vertex v, we denote by $N[v]$ its *closed neighborhood*, i.e. v and all its neighbors. We will slightly abuse set difference and union notation by using $-$ and $+$ instead. $X + Y - Z$ for three sets X, Y, Z are those elements that are in either X or Y but not in Z.

Algorithm R_0

Consider the following simple recursive algorithm working on an input graph $G = (V, E)$. (Call it Algorithm R_0):

1. If the input graph is empty, return 0.
2. If the input graph G has a vertex v without any neighbors, return $1 + R_0(G[V - v])$. See fig. 1.
3. Otherwise find a vertex u of maximum degree (if there are several any will do) and return

$$\max(1 + R_0(G[V - N[u]]), R_0(G[V - u])),$$

keeping in mind that that $N[u]$ includes u. See fig. 2.

To see why algorithm R_0 correctly computes $\alpha(G)$, note that if there is an isolated vertex v, a maximum size independent set will always contain it (Row 2). On the other hand, if there are no isolated vertices, we can pick a vertex u of maximum degree and branch on the two cases that u is in the maximum independent set (in which case we discard all vertices adjacent to u from G), or that u is not in the maximum independent set (Row 3).

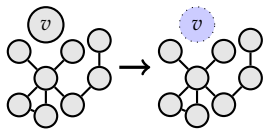


Figure 1: A graph with an isolated node v. We can safely say v is in the MIS, and continue without it.

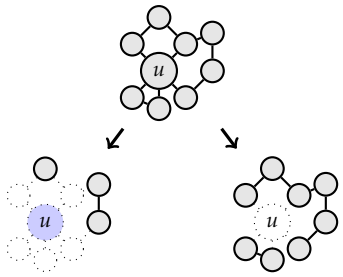


Figure 2: Vertex u has highest degree. Left branch: Assume u is in the MIS – remove u and its neighbors, since the neighbors can no longer be in the MIS. Right branch: Assume u not in the MIS – just remove u.

Inputs

The data directory contains eleven random input instances of increasing size. All input files are text files on the following format describing the undirected graph $G = (V, E)$: First comes a positive integer n giving the number of vertices in the graph. Next follow n rows each containing n 0/1-entries describing the so-called *adjacency matrix* of the graph. The jth integer at the ith row is a 1 if and only if $ij \in E$.

file	$\|V\|$	$\alpha(G)$
g30.in	30	14
g40.in	40	16
g50.in	50	19
g60.in	60	20
g70.in	70	22
g80.in	80	24
g90.in	89	26
g100.in	100	27
g110.in	110	29
g120.in	120	30
g130.in	130	?

Deliverables

1. Implement algorithm R_0 and run it on the instances provided in data/g30.txt, data/g40.txt, ..., data/g60.txt.

   Count the number of recursive calls of R_0 for each instance and plot the logarithm of that number vs the instance vertex size.

   What is the time complexity dependence on n?

   Use whatever programming language and libraries you want, but make sure that your code is not unnecessarily long.

   First implement the algorithm by making a new copy of the graph at each recursive call, just to make it as easy as possible to verify that the algorithm works as intended. Once you get this to work, you can pay attention to the problem of efficiently representing a graph through the recursion so that it is easy to remove (and add, see below) vertices to it by temporarily modifying the graph at hand.

   Attach a printout to the report or have it checked by your lab assistant.

2. Add the following line to algorithm R_0 after its first line checking for emptiness, and call the resulting algorithm R_1 (replacing the recursive calls to R_0 by R_1):

   If G has a vertex v of degree 1 return $1 + R_1(G[V - N[v]])$.

   Before you write any code – draw (by hand) pictures such as those above to illustrate what happens with your graph when you run algorithm R_1 on it. Use the pictures when you implement the algorithm, and when you argue the correctness of it, i.e., when you motivate why it always computes $\alpha(G)$. You are supposed to bring your pictures and show them to your lab assistant.

   After you have implemented algorithm R_1, run it on the instances data/g30.in, data/g40.in, ..., data/g100.in.

   Count the number of recursive calls of R1 for each instance and plot the logarithm of that number vs the instance vertex size.

What is the time complexity dependence on n?

3. Add the following line to algorithm R_1 after its first line checking for emptiness, and call the resulting algorithm R_2 (replacing the recursive calls to R_1 by R_2):

   If G has a vertex v with exactly two neighbors u, w, then if $uw \in E$ return $1 + R_2(G[V - N[v]])$, else add a new vertex z to the graph with edges to every neighbor of u and w, except v, and return $1 + R_2(G[V - N[v] + z])$.

   If G doesn't have a vertex v with exactly two neighbors, then we proceed just as in algorithm R_1.

   Draw pictures such as those above to illustrate algorithm R_2, and use the pictures both when you implement it, and when you argue the correctness of it, i.e., motivate why it always computes $\alpha(G)$.

   Run algorithm R_2 on the instances data/g30.in, data/g40.in, ..., data/g120.in.

   Count the number of recursive calls of R_2 for each instance and plot the logarithm of that number vs the instance vertex size.

   What is the time complexity dependence on n?

4. Fill out the report on the next page; you can just use the LaTeX code if you want.

Independent Set Lab Report

by Alice Cooper and Bob Marley[1]

[1] Complete the report by filling in your names and the parts marked [...]. Remove the sidenotes in your final hand-in.

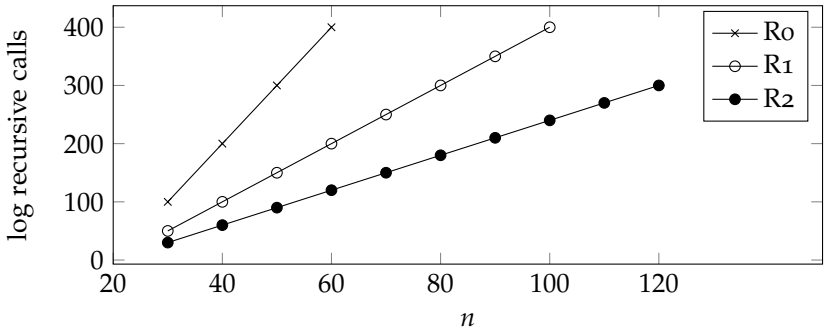
Correctness

Algorithm R1 correctly computes $\alpha(G)$ because [...].[2]
Algorithm R2 correctly computes $\alpha(G)$ because [...].

[2] Fill in the [...] with an argument why R1 indeed computes the size of the maximum independent set. Use as little mathematical notation as necessary. A "proof" is whatever convinces me.

Empirical Running time

Experiments. [3]



[3] Plot the logarithm of the number of recursive calls for the three algorithms R_0, R_1, and R_2, in a combined image.
Use whatever software you like to produce the image; the placeholder image on the left is constructed in the LaTeX source, and should give you a feel of what the result could look like (with completely different values on the y-axis.)

The running times of algorithm R_0, R_1, and R_2 appear to be [...], [...], and [...], respectively.[4].

[4] Replace the [...] by a function of n on the form $O(c^n)$. Use your measurements on the run time to estimate c in the three cases.

Theoretical Upper Bound

Denote be $T_i(n)$ the worst runtime of algorithm Ri on *any* graph on n vertices. Note that $T_i(n)$ is a non-decreasing function of n. For R_0 we can conclude that

$$\begin{aligned} T_0(n) &\leq \max(T_0(n-1), T_0(n-1) + T_0(n-1-d_{\max})) \\ &\leq T_0(n-1) + T_0(n-2) \end{aligned}$$

with $d_{\max}$ the degree of the vertex we branch on. The hard part is the one when there are no isolated vertices, in which case the vertex u we are branching on has at least one neighbor.

For R1 we have that[5]

$$T_1(n) = [\ldots]$$

[5] Derive similar recursive bounds on $T_1(n)$ and $T_2(n)$.

For R2 we have that

$$T_2(n) = [\ldots]$$

Worst Case Upper Bound The running times of algorithm R_0, R_1, and R_2 are in [...], [...], and [...], respectively.[6].

[6] Replace the [...] by a function of n on the form $O(c^n)$. Use the recursive bounds you've derived above. *Hint: A recurrence of the form $T(n) \leq \sum_{i=1}^{k} a_i T(n-i)$ is called a linear homogeneous recurrence relation with constant coefficients. To solve it, you can set $T(n) \leq c^n$ where c is the largest real root to the characteristic polynomial $x^k - \sum_{i=1}^{k} a_i x^{k-i}$.*

Optional

Add more "algorithmic intelligence" to the algorithm R_2 in order to tackle the instance in data/g130.in. Try to make it run in less than 10,000,000 recursive calls.

Suggestions for possible speed-ups:

- Is there some clever (branching) rule for vertices of degree 3?
- Can we use the information of the largest found independent set so far, to reduce further computation time?
- What if the graph gets disconnected?

Perspective

This lab establishes medium skills in recursive algorithms implementation, and simple means to analyze their running time.

The algorithms investigated here can hardly be called "advanced", but the main message is that no one knows how to do significantly better with any other means. Indeed, the strongest theoretical worst case run time bound for maximum independent set to date is obtained by a computerized calculation on a huge set of branching reduction rules, just like the ones we've looked at here [Robson. Algorithms for maximum independent sets. Journal of Algorithms, 7(3):425–440, 1986]. For a much cleaner analysis with just a few rules, obtaining a comparative bound, see [Fomin, Grandoni, and Kratsch. A measure & conquer approach for the analysis of exact algorithms, JACM 56 (5), article No. 25, 2009].