Informatics 2 – Introduction to Algorithms and Data Structures

Coursework 2 - Algorithms for Independent Set

This document is the specification for Coursework 2 of Inf2-IADS. It is being released Tuesday 14th March (week 8), and your submissions will be due at NOON on Thursday 30th March 2023 (Thursday of week 10). This is a summative coursework, and your grade for this coursework will contribute 15% of your overall mark for Inf2-IADS.

Your work (written arguments and also code) should be developed and written on an individual basis. It is fine to discuss the task with classmates, and to help each other understand what need to be done. However you SHOULD NOT COPY CODE OR WORKED SOLUTIONS from classmates, or from online websites, or anywhere else, and you should not give or take key details for the implementation. If you discover literature which gives you ideas for (the final part of) the coursework, please make sure you credit those sources in your report. Please also check out the "Own Work" declaration.

The Independent Sets problem

In this coursework we are going to explore some algorithms for the *Independent Set* problem. We have already seen the decision version of this problem in lectures, together with a proof of NP-completeness for the basic unweighted version of the problem[1] . For this reason, we know that we should not hope to find a polynomial-time algorithm to exactly solve the problem for all input graphs.

Basic Definitions

In the unweighted version of this problem, we are given an undirected graph $G = (V, E)$ on $n = |V|$ nodes. The nodes of the graph represent some kind of agents with intrinsic value to us, and our aim is to find a subset $I \subseteq V$ of vertices of maximum cardinality, subject to some edge constraints; the edges in E encode these constraints, with edge (u, v) meaning we cannot have both u and v in the set I, and a candidate independent set I is required to satisfy $(\neg u \in I) \vee (\neg v \in I)$ for every $(u, v) \in E$.
In the *weighted* version of the independent sets problem, we also get a weighting $W : V \to \mathbb{N}$ as input, along with the graph $G = (V, E)$.
The optimisation versions of these problems are as follows:

Independent Set (IS): Given a undirected graph $G = (V, E)$, return a set $I \subset V$ such that $|\{u, v\} \cap I| \leq 1$ for every $(u, v) \in E$, and such that $|I|$ is maximized subject to these constraints.

Weighted Independent Set (WIS): Given a undirected graph $G = (V, E)$, together with a weighting $W : V \to \mathbb{N}$ on the nodes of this graph, return a set $I \subset V$ such that $|\{u, v\} \cap I| \leq 1$ for every $(u, v) \in E$, and such that $\sum_{u \in I} W(u)$ is maximized subject to these constraints.

Our Approach

As mentioned, we don't expect to be able to achieve a polynomial-time algorithm to return maximum independent sets for general graphs in all cases. However, we can make progress on IS and WIS in the following two ways:

[1] Of course, the NP-completeness proof was be set-up in terms of the decision version of the problem, where we query whether there was a Independent set of value $\geq k$. However, if we were able to compute the maximum possible IS in polynomial-time, just checking that answer against k would also answer the decision question.

- We can consider simple *heuristic* algorithms, these being methods which might not give rigorous performance guarantees for the worst-case, but which may do fairly-well on many instances of the problem. We can analyse whether these run in polynomial-time, analyse when they do (or do not) work, and code them up. We will work with variants of the Greedy algorithm.
- We can consider *subclasses* of undirected graphs, which may have polynomial-time algorithms to exactly solve for a maximum independent set. In this direction, we will focus on *trees*.

The specific tasks you are required to take care of are given in the following sections.
You will be asked to submit 2 files for this coursework (submitted through Learn):

- A .pdf file containing the worked-solutions for Part A and Part D, named `IADS_cwk2_solutions.pdf`. This may be prepared either by scanning-in (legible) handwritten solutions, or alternatively solutions prepared in LaTeX or another word processing package.
- Your completion of the `independent.py` file, containing your Part B and Part C implementations.

Your implementations should be in Python, and since we will mark the coursework in `python3.8.10` on DICE, you should check that your code runs successfully with this installation. To create a "level playing field", we ask that you do not use `numpy` or other data-structures/graph libraries in your work.

1 Part A [40 marks]: Theoretical Analysis of Greedy

In this section, you are asked to analyse some aspects of the Greedy Algorithm (for independent sets) on graphs and bipartite graphs.
We will define two variants of the *Greedy algorithm* for Independent set, as shown below. In this presentation we assume that the vertices of the graph are named $0, \ldots, n-1$, compatible with Python array indexing.
The first algorithm GreedyIS is a simple algorithm for the *unweighted problem*. It builds an Independent Set of maximal size (but not necessarily maximum) by adding one vertex at a time. However, when it chooses the "next independent vertex" *argmin*, this choice is done with respect to the *residual* graph after previous Independent Vertices (and their neighbours) have been removed. This requires the modification of the Adjacency list structure (removing the entirety of *argmin*'s list, but also the same for all of *argmins*'s neighbours (since these cannot be ever added to the independent set in the future). This is what is intended in the line 10, and will be seen in the worked example below the algorithms statement.

Algorithm GreedyIS(G=(V, E))

1. $n \leftarrow |V|$
2. Adj is an adjacency list structure for E; IS and deg are initialised to all-0s
3. **for** $u \leftarrow 0$ **to** $n-1$ **do**
4. $\quad deg[u] = len(Adj[u])$ // (set the initial deg values for all nodes)
5. **while** (some vertices remain in the graph) **do**
6. $\quad u \leftarrow \arg\min\{deg[v]; v \in V, IS[v] == 0\}$ // (node with min *residual* degree
7. $\quad IS[u] \leftarrow 1$ // gets added to the IS)
8. $\quad$ **for** $(w \in Nbd(u))$ **do**
9. $\quad\quad IS[w] \leftarrow -1$ // (mark u's neighbours as disallowed)
10. $\quad\quad$ Update Adj, deg to reflect deletion of $\{u\} \cup Nbd(u)$
11. **for** $u \leftarrow 0$ **to** $n-1$
12. $\quad IS[u] \leftarrow \max\{0, IS[u]\}$ // (tidy: delete marks of disallowed vertices)
13. **return** IS

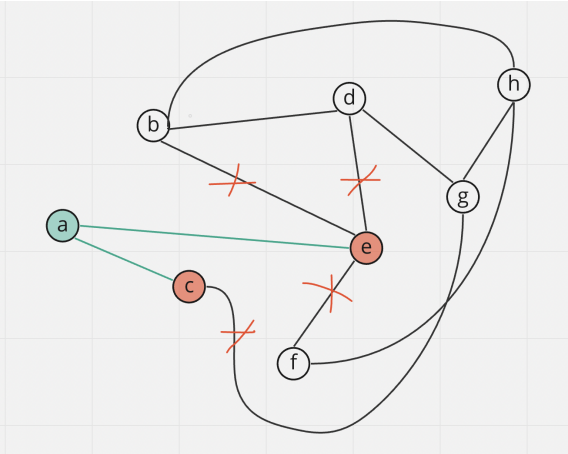
Algorithm

The algorithm above is for the unweighted version of Independent Set.
If we are considering WIS, where the input *also* has a list of weights $W(0), \ldots, W(n-1)$ for the vertices, then the value of an Independent Set is influenced by the weights (as specified in the definition of WIS on page 1. In this case, instead of defining $u \leftarrow \arg\min\{deg[v]; v \in V, IS[v] == 0\}$ to select the next vertex, we would instead choose one of the following: [2]

$$(a) \quad u \leftarrow \arg\max\left\{\frac{W(u)}{deg[u]} : u \in V, IS[u] \text{ is } 0\right\}$$
$$(b) \quad u \leftarrow \arg\max\{W(u) : u \in V, IS[u] \text{ is } 0\}$$

Although the use of arg max in these selections may seem to be in opposition to the spirit of GreedyIS for the unweighted case, note that in the (a) option for weighted, we have $1/deg[u]$ in the maximization, and this corresponds to smaller $deg[u]$ values (again, we remark that we will evaluate $1/deg[u]$ as ∞ when $deg(u)$ is 0). In particular, if we had initialised $W(u) \leftarrow 1$ for all u in the unweighted case, then option (a) is exactly the same criterion as used in GreedyIS.
The pseudocode for GreedyIS can be adapted to incorporate weights by updating the selection criterion to (a), while making sure to initialise weights to 1 for the unweighted case. We can also write a variant GreedyIS(b) of the method which will use the selection criterion (b) for WIS, with the remainder of the pseudocode unchanged.
We end with one important clarification: it will often be the case that some vertices in the graph have the same degree. If this situation arises in the choice of the min-degree vertex, ties should be broken in *lexicographic order*.

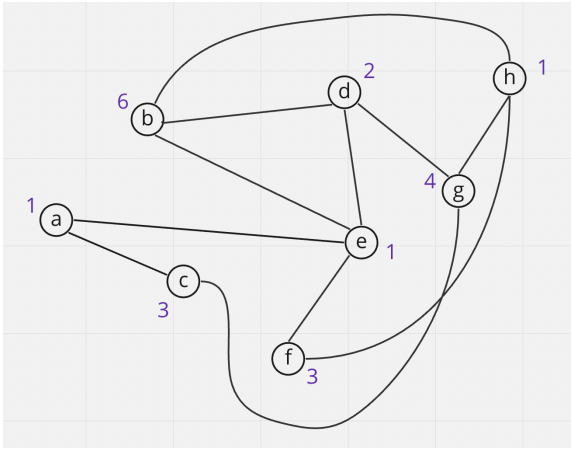
A(i) Consider the unweighted graph below being executed on by GreedyIS. The diagram shows the first node that will be added to the Independent Set, shaded in Green. The diagram also indicates the node's neighbours in red (to reflect the fact they cannot be added to the IS) and also shows the adjacent edges of these neighbours as "crossed out" (in line 10 of the pseudocode we see the detail that *Adj* and *deg* need to be updated to reflect the deletion of *argmin* and its neighbours).



Complete the execution of GreedyIS on this graph, drawing the leftover subgraph of interest after each iteration of the **while**. (**5 marks**)

[2] For option (a), we will allow the abuse of arithmetic to interpret $\frac{W(u)}{deg(u)}$ as ∞ when $deg(u)$ is 0, to ensure that these isolated vertices are always prioritised.

A(ii) Consider the weighted graph below, where the purple values are the weights of the nodes. Execute the weighted GreedyIS (in other words, using criterion (a)) on this graph, drawing the leftover subgraph of interest after each iteration of the **while** loop. (**5 marks**)



A(iii) Consider the same weighted graph as in 2., and execute the weighted GreedyIS(b) on this graph, drawing the leftover subgraph of interest after each iteration of the **while** loop. (**5 marks**)

A(iv) Assuming that the graph $G = (V, E)$ is represented in Adjacency List format, justify in detail the fact GreedyIS can be implemented in $O(n^2 + m)$ worst-case running time, where $n = |V|, m = |E|$. (**10 marks**)[3]

note: this will require you to take real care in how the adjustment of *Adj* is done in line 10. The key is to only update/delete what is really necessary for the Algorithm, rather than being concerned with an accurate representation of the residual graph.

A(v) Consider an unweighted *bipartite* graph $G = (L \cup R, E)$, with $L = \{u_1, \ldots, u_{\lceil n/3 \rceil}\} \cup \{w_1, w_2, w_3, w_4\}$, and $R = \{v_1, \ldots, v_n\}$. The edge set $E = E_1 \cup E_2$ consists of the following edges:

$$\begin{aligned} E_1 &= \{(u_i, v_i) : i = 1, \ldots, \lceil n/3 \rceil\} \cup \\ &\quad \{(u_i, v_{i+\lceil n/3 \rceil}) : i = 1, \ldots, \lceil n/3 \rceil\} \cup \\ &\quad \{(u_i, v_{i+2\lceil n/3 \rceil}) : i = 1, \ldots, \lceil n/3 \rceil\} \\ E_2 &= \{(w_i, v_j) : i = 1, \ldots, 4, j = 1, \ldots, n\} \end{aligned}$$

Essentially, the E_2 edges form a complete bipartite graph between the 4 special vertices $\{w_1, w_2, w_3, w_4\}$ and the set R, while the edge set E_1 is a subgraph where each of the vertices $\{u_1, \ldots, u_{\lceil n/3 \rceil}\}$ have 3 adjacent edges and all of the vertices in R have 1 adjacent edge.

Work out the independent set I which will be constructed by GreedyIS, justifying your answer with respect to the degrees of the different vertices as the algorithm proceeds. Show that this set will be approximately 1/3 of the size of the true *maximum* independent set for this graph? (**10 marks**)

note: You are welcome to assume n is a multiple of 3.

A(vi) How would you alter the graph construction of (v) to get progressively worse "approximation factors" for the result returned by GreedyIS? (**5 marks**)

[3] In fact, it is possible to implement the algorithm in worst-case $\Theta(n \lg(n) + m)$ time, but that requires careful design and use of some finicky auxiliary data arrays/structures, hence we are not asking for this. Worth thinking about for your own interest.

2 Part B [20 marks]: Working with Graphs in Python

In this Part of the coursework we complete setting-up/evaluation tasks for reading-in files and building the corresponding graphs. `independent.py` has the class `Graph` (as well as a small `AdjNode` class), which includes two declarations of the `__init__` method.
We will assume that the vertex set V of an input graph $V = \{0, \ldots, n-1\}$, to be compatible with Python indexing. This means that the graphs you input should conform to this assumption, with all the nodes mentioned in the file being between 0 and `num` inclusive.
The first `__init__(self, num)` in the file is a basic method for initialising the graph in advance of calling "Graph generation" methods - this sparse `__init__(self, num)` does *not* need to be edited by you, and its purpose is to perform initialisation in a case where the following step will be to subsequently generate a random graph of the appropriate size. However you should examine this basic method to see the structure of the properties of `Graph`, these being `n`, `weights`, `graph` (which is the adjacency list itself) and `degs`.
The second `__init__` method is intended for the case when the graph is input as a file (called `filename`) and you are asked to implement this method:

```
def __init__(self,num,filename,weighted):
```

We will allow for two kinds of input files (weighted and unweighted), and your implementation will need to identify and handle these two cases appropriately (to assign the value `num` to `self.n`, to build the Adjacency Lists into `graph`, as well as counting the degrees of the nodes in `degs`, and storing the node weights into `weights`).

- We may have files representing *unweighted* graphs. In this case the file should simply be a list of lines each containing two natural numbers which are the endpoints of an edge in the graph.

  `2 4`

  This line would specify an edge between nodes 2 and 4 to be added into the Adjacency list `graph` (4 being added into `graph[2]`'s list, and 2 also being added into `graph[4]`'s list), and there would also be an appropriate update to the `degs` entries for these nodes.

  This case will be identified by a successful test of `not weighted` against that input parameter, but the `weights` list should be initialised to "all 1s" for this unweighted case.

  See `graph1U` for an example of the format.

- In the *weighted* case, the file will start with the list of weights for each of the nodes, with each node being treated by an individual line. The format of these lines looks identical to the edges, however the entry `u w` should instead be interpreted as `W(u) <- w`. Note that although the pattern looks identical, the weights are not restricted to be $<$ `num -1`.

  We will assume that the weights are given in the initial `num` lines in the file, with the `num+1`th line being a separating line to give a clear visual separation before the edges are listed.

  See `graph1W` for an example of the format.

We ask you to implement `__init__` so that we initialise the following variables as described.

- `self.n` - this should be initialised to `num`
- `self.graph` - this is to be an Adjacency list data structure, with a cell/list for every `u = 0` ... `self.n-1`. `self.graph[u]` will be initialised to `None` at the outset, but once `u` gets some adjacent edges, it will contain the head of the list of adjacent nodes to `u` in the graph.

  The given method `add_edge` will be invaluable in getting the edges added into `graph`.

- `self.weights` - this is a list of length `self.n` which should be assigned the weights of the input graph (or the values 1, for the unweighted case).
- `self.degs` - this is a list of length `self.n` which should be assigned the degrees of the nodes of the input graph.

The above are the *only* class variables which should be assigned.

notes:

- You may assume that the nodes of the graphs are all natural numbers `0, ..., n-1` and that each line describing an edge has 2 such numbers separated by whitespace. However, I ask that you don't make assumptions about the number of digits, or the amount of whitespace between them.

  Note that in the weighted case, the file starts with lines for the weights. These have a similar format to the edges except that the second number on each line can be bigger than `n-1`. Also, there is an extra"separation line" (to be ignored) before the file has the "edge lines".

- The majority of the 20 marks are going for correctness, but we will some tests on running time (for some big input files) too. We ask you not to be careless in the time for setting-up the data structures.

- You may welcome to add checks to make sure that certain assumptions (node labels are all at most `num-1`, and/or no duplicate edges in the input file. However, we will not test your code against non-conforming inputs.

testing: If you successfully implement this `__init__`method for the unweighted case, you can use the given `print_out_graph(self)` method to check that you have a correct implementation. The output for (unweighted) input graph `graph1U` should be:

```
>>> g = independent.Graph(7,"graph1U",False)
>>> g.print_out_graph()
Vertex 0(weight 1): -> 4 -> 2

Vertex 1(weight 1): -> 4 -> 6 -> 3 -> 2

Vertex 2(weight 1): -> 3 -> 5 -> 4 -> 1 -> 0

Vertex 3(weight 1): -> 6 -> 5 -> 2 -> 1

Vertex 4(weight 1): -> 1 -> 2 -> 0

Vertex 5(weight 1): -> 6 -> 3 -> 2

Vertex 6(weight 1): -> 5 -> 3 -> 1
```

The output for weighted input graph `graph1W` should be:

```
>>> g =  independent.Graph(7,"graph1W",True)
>>> g.print_out_graph()
Vertex 0(weight 2): -> 4 -> 2

Vertex 1(weight 3): -> 4 -> 6 -> 3 -> 2

Vertex 2(weight 4): -> 3 -> 5 -> 4 -> 1 -> 0

Vertex 3(weight 1): -> 6 -> 5 -> 2 -> 1

Vertex 4(weight 8): -> 1 -> 2 -> 0

Vertex 5(weight 6): -> 6 -> 3 -> 2

Vertex 6(weight 2): -> 5 -> 3 -> 1
```

3 Part C [20 marks] - implementing the GreedyIS Heuristics

3.1 GreedyIS (15 marks)

In Part A, we spent a considerable amount of time exploring the Greedy methods for IS and WIS. There are a couple of variants of these, but the main one is the weighted generalisation of the displayed pseudocode GreedyIS - the generalisation being to have a weights list and to use condition (a) to select the "next node" for IS. We do not need to check whether our graph is weighted or not in working with the generalisation, as we will have initialised the `weights` list to "all 1s" in the unweighted case, thus removing the distinction.
Your first assignment is to implement the `GreedyIS(self)` method for weighted (criterion (a)) graphs, taking into account the details given in Part A. Apart from the logical structure of that algorithm, there are some programming/Python issues that you will need to take care of:

- We would ask that you do not destroy the `self.graph` adjacency list structure during the iteration of the Greedy loop. For that reason, one of the first steps of your implementation should be to create a `deepcopy` of `self.graph` to use as a "scratch" representation.
- Certain operations on an Adjacency list data structures can be expensive, and this is particularly the case for edge deletion (as we may need to explore through a list). Please think carefully about what changes are *essential* for the correct implementation of the algorithm, and only do the necessary.

Below is the correct output for operation of `GreedyIS()` on the input graph `graph1U`:

```
>>> import independent
>>> g = independent.Graph(7,"graph1U",False)
>>> IS = g.GreedyIS()
>>> IS
[1, 1, 0, 0, 0, 1, 0]
```

Below is the correct output for operation of `GreedyIS()` on the input graph `graph1U`:

```
>>> import independent
>>> g = independent.Graph(7,"graph1W",True)
>>> IS = g.GreedyIS()
>>> IS
[0, 0, 0, 0, 1, 1, 0]
```

3.2 GreedyIS(b) (5 marks)

We already discussed the variant of "Greedy" for the weighted case, where we use criterion (b) to choose the "next node" for IS. This is a minor variation on the previous method, and much of the code may be reused. You are asked to implement the criterion(b) variant within `graph.py`:

```
def GreedyIS_b(self):
```

Test your new function, and in particular, consider an evaluation against the performance of GreedyIS. As it happens, this method returns an identical answer to GreedyIS for file `graph1W`, but there will be many other weighted files where the answer is different.

```
>>> import independent
>>> g = independent.Graph(7,"graph1W",True)
>>> IS = g.GreedyIS_b()
>>> IS
[0, 0, 0, 0, 1, 1, 0]
```

4 Part D [20 marks] - Independent sets on trees

Suppose we consider the special case of IS, WIS where the input graph is a *tree* $T = (V, E)$ with an identified root $r \in V$ (there are no other limitations on structure). This means that for every $v \in V(T)$, the subtree rooted at v is well-defined - we will refer to this subtree as T_v.
In this section, you are asked to develop a polynomial-time algorithm for solving WIS (and including IS, when weights are all 1).
For every $v \in V$, we define the following:

- $\kappa_{v,0}$ to be the total weight of the maximum-weight Independent set of subtree T_v which does not include v itself.
- $\kappa_{v,1}$ to be the total weight of the maximum-weight Independent set of subtree T_v with v belonging to the independent set.

D(i) Develop a *pair of recurrences* which express the value of $\kappa_{v,0}$ (and similarly of $\kappa_{v,1}$) in terms of the κ values of the child nodes of v. Include the "base case" whsn v is a leaf, and then describe how we can exploit the recurrences to design a polynomial-time algorithm to solve WIS for a tree. (**10 marks**)

D(ii) In the earlier parts of this coursework, we developed algorithms for inputting general graphs into an Adjacency list Data Structure. We won't know whether some of these graphs were actually trees or not. Can you propose an algorithm/method to check `self.graph` to determine whether it is a tree? Discuss likely running-time, and how we would then convert the graph to a tree-like data structure in low polynomial-time. (**5 marks**)

D(iii) In Part B we discussed the Greedy method for constructing Independent sets of a general (weighted or unweighted) graph, and saw that it does not always return an optimal result for general graphs.

Consider the criterion (a) method that was implemented as GreedyIS. Will this method result in a maximum independent set when the input graph is an unweighted tree? Justify your answer. (**5 marks**)

5 Submission

You should submit a completed `independent.py` file, plus a solutions file named `IADS_cwk2_solutions.pdf` with a labelled Part B and Part D sections.
Please ensure that your code runs successfully in the `python3.8.10` installation on DICE before you submit.
You should submit your coursework via Learn. Go to the "Assessment" page from which you obtained this instruction document, and click on the link bearing the title of this coursework. This will allow you to upload your files (or as many of these as are relevant for you):

`independent.py` `IADS_cwk2_solutions.pdf`

We ask that you do not submit any other files than these.
You may re-submit as many times as you wish up to the deadline **(NOON (BST) on Thurs 30 March)**. Your most recent submission before the deadline will be the one that is marked.

Mary Cryan, March 2023