Graph Builder

AI – Spring '23 – Gabor

Definitions

Almost all the problems that we examine in this course can be represented as or on a graph. This is true not only for the problems in this course, but also for interesting problems in the world at large – they can almost all be reduced to a graph. Therefore, it makes sense to establish some common graph terminology:

Graph: a graph, $G(V, E)$ is a collection of vertices, V, some of which are connected by edges, E.

Vertex (pl. vertices) is equivalent to **node**. The two terms are interchangeable. Examples of vertices are cities, people, numbers, words, letters.

Edge is equivalent to **link**. The two terms are interchangeable. An edge indicates that two vertices are related. The relation may be symmetric (ie. v_1 is related to v_2 implies v_2 is related to v_1 (eg. siblings or "sum of values is even"), or not (eg. v_1 is a parent of v_2 or individual lanes of streets between landmarks (streets in this sense are one-way)). If the edges of a graph are directed, the graph is often called a **digraph** (short for directed graph). Unless otherwise stated, the graphs in this class are **undirected**.

Unless otherwise stated, graphs have no **self-loop**s (ie. a vertex is never related to itself), and graphs do not have **multi-edges** (multiple edges between the same pair of vertices). The former may happen infrequently, the latter almost never. Typically, if an edge needs to be indicated explicitly, then it is shown as a tuple, (v_1, v_2) with both digraphs and undirected graphs, and it is understood that for an undirected graph, the tuple is not ordered.

Graphs are typically depicted as circles, dots, or small squares some of which are connected by line segments, arcs, or arbitrarily complicated curves. The connections correspond to the edges and their shape (ie. line segment or more complicated) is irrelevant. It only matters whether they are connected.

The **size** of a graph $G(V, E)$ is $|V|$.

Given a graph $G(V, E)$, The **neighbors** of $v \in V$, $N(v)$, are those vertices $w \in V$ such that $(v, w) \in E$. In other words, the neighbors of v are exactly those vertices that one can get to from v. The **degree** of a vertex, v, is simply the number of neighbors that it has: $|N(v)|$. Sometimes, we may be interested in those neighbors of vertex v that are within a specific subset W of V. That is, the neighbors of v restricted to W $\cong N_W(v) \cong N(v) \cap W$. We generalize even this to restricted neighborhoods so so that if $X, W \subset V$, then

$$N_W(X) = \left(\bigcup_{x \in X} N(x) \right) \cap W$$

In other words, the neighbors of X restricted to W is the union of the neighbors of each vertex in X intersected with W.

A **path** from $v_0 \in V$ to $v_n \in V$ is a sequence of vertices $v_0, v_1, v_2, \ldots v_n$ all in V such that $(v_i, v_{i+1}) \in E$ for $i \in \{0,1, \ldots n-1\}$. The path could also be given as a sequence of edges: $(v_0, v_1), (v_1, v_2), \ldots (v_{n-1}, v_n)$. The length of path is n regardless of the form used. In other words, the length of the path is the number of edges in the path.

If the starting vertex of a path is the same as the ending vertex of a path, then it is not called a path, but rather a **cycle**. The length of the cycle is equal to the number of edges, which is the same as the number of vertices.

Simple paths / cycles

Two vertices, $v, w \in V$ are **connected** if there is a path from v to w. In this case, the distance from v to w is the length of the shortest path from v to w. A graph is **connected** if there is a path from v to w for all $v, w \in V$. A vertex v is always connected to itself, and the distance is 0.

Connected components
Subgraphs
Cliques, Bipartite, etc.
Isomorphisms
Graph Labeling
Cartesian product
Examine how connected works with digraphs

Graphs can be more complicated than what has been described so far. In particular, it is common to affix values to either vertices or edges. For example, cities might have elevations, or average temperatures associated with them, or persons might have birthdays and addresses that are associated. Edges between cities could have distances associated with them and/or speed limits and/or time estimates of how long it takes to proceed along a given route. Certain edges or nodes could also have names associated with them. All of these could be considered properties of vertices and edges.

Representing Graphs

There are two standard ways to represent graphs in their simplest form. One of these is to have a matrix where at position (j, k) there is a 1 iff the edge from j to k exists and a 0 otherwise. It should be clear that the number of entries will be $|V|^2$. If $|V|$ is large then its square might be prohibitively large. All the graphs that we examine will be sparse. That means that $|E|/|V|$ does not grow too large (perhaps it stays under 10) as the graphs get larger. This, in turn, means that the vast majority of entries in the edge matrix are 0s, which can be very costly in the amount of memory that it uses.

Therefore, the following method is much more common. The edges could be represented by a list (or dictionary if the vertices are not numbered) of length $|V|$, where the entry at each index position (ie. for each vertex) v of the list is a list or set of the neighbors of v.

In the situation where vertices and edges may have properties, there are a few ways to go. One way is to have a separate data structure for the vertices where the entry for each vertex is a dictionary where the key/values are the property name and value for each relevant property. A similar separate data structure for the edges could also made.

An alternate method is to incorporate the properties directly into the vertex or edge data structure. For example, the vertex data structure could now house a list for each vertex where the entry at index position 0 is the neighbors and the entry at index position 1 is dictionary of property names/values for the vertex. While this allows one to have a single data structure, it makes it sufficiently messier that I usually opt for a vertexProperties data structure and an edges data structure.

One could do the same thing with edgeProperties, but because the edges for a given vertex can be construed as a set, one could replace this set with a dictionary, where the value for a given destination vertex is either the value for that edge, when there is only one property involved, or a dictionary of property name/value-s.

Eventual examples section

For most of the problems in this class the first or second version will be ample. However, there will be a few instances where we may get more complicated graphs, so the next lab explores this.

Graph Lab specification

AI – Spring 2024 – Gabor

This lab is about taking a specification for a graph and implementing that graph as a Python data structure. The language for the specification is sufficiently rich with enough subtlety that it makes sense to devote an entire lab to it.

There are three command line directives to parse, all of which serve to build up a graph. The first one, specifying the size of the graph and some basic structural information, will always be given, while the other two, specifying edges and edge and vertex properties) can happen repeatedly and in any order.

Your script will expose eight functions upon being imported:

`grfParse(lstArgs)`:	returns graph constructed from `lstArgs`
`grfSize(graph)`:	returns the number of vertices the graph has
`grfNbrs(graph, v)`:	returns a set or list of the neighbors (ie. ints) of vertex v
`grfGProps(graph)`:	returns the dictionary of properties for the graph, including 'width' and 'rwd'
`grfVProps(graph, v)`:	returns the dictionary of properties of vertex v
`grfEProps(graph, v1, v2)`:	returns the dictionary of properties of edge (v1, v2)
`grfStrEdges(graph)`:	returns a string representation of the graph edges
`grfStrProps(graph)`:	returns a string rep. of the graph edges and properties

`grfParse()` should return the complete graph data structure. If you have an adjacency list and a separate `vertexProperties`, then you could package them together as a tuple or list such as `(edges, vertexProperties)`. You could also package it up as `{'edges': edges, 'vertexProperties': vertexProperties}`. These representations are examples, and you may decide upon a different implementation in your own code. The grader will not directly examine your data structure, but it will pass your graph data structure to the seven functions above which take it as argument.

This is similar to the codingBat.com labs in that you are defining functions within your code that the grader will test when it does an import of your code. However, as with the Othello 4 and 6 labs, if your script is called from the command line, then it should construct the graph specified in the command line arguments and show the resulting graph printed out. Your code should not rely on global variable to ensure that it words both on the command line and when it is imported. Here is how you should structure your code:

```
import sys; args = sys.argv[1:]

def func1(…):
  -- func1 implemented here --

def func2(…):
  -- func2 implemented here --

…

def grfParse(lstArgs):
  for arg in lstArgs:
    if arg is a graphDirective: create the graph
    elif arg is a …: handle that type of directive and update graph
    ..
  return graph

def main():
  graph = grfParse(args)
  edgesStr = grfStrEdges(graph)
  propsStr = grfStrProps(graph)
  output edgesStr
  print(propsStr)

if __name__  == '__main__ ': main()

# Final comment goes here
```

Input directives – Graph directive

There are three input directives. The first of these is: `G[graphType]#[W#][R#]`

This directive specifies the size and nature (structure) of the graph. The required size of the graph is given by the first #. All other parts are optional. There are two types of graphs that you should be prepared to work with. A graphType of N specifies that there are no initial edges presumed. In this class, we are usually concerned with graphs that can be laid out on a rectangular grid, which is specified by a G (the default). A gridworld type of graph presumes the graph has a certain height and width, and it assumes that all vertices not at the perimeter have bidirectional edges to their four non-diagonal neighbors. The four corner vertices get two neighbors each, while the remaining perimeter vertices get three neighbors each. G is a compact way to specify many edges. All the graphs in this lab are digraphs.

For N type graphs, W is ignored, if provided. For gridworlds, the width of the graph defaults to the smallest int, at least the square root of the size of the graph, that evenly divides the size of the graph. This is the standard that has been used throughout the course. If `W` is provided, it overrides the default width. If `W` is 0 for a Gridworld graph, then it is actually an N type graph.

Several subsequent labs will work directly with the graph data structure built up in this lab, and one aspect of those labs is that certain vertices or edges will have rewards. Sometimes these rewards will be specified explicitly and other times, a default reward will be assumed. If `R#` is specified, it sets the default reward. If it is not set, the default for the default reward is 12. The property key is 'rwd'.

Vslcs

In the below, Vslcs (short for vertex slices) are a shorthand way of specifying a list of vertices – a list of possibly many integers. In particular, Vslcs is a comma separated list of slices (as strings without spaces), where the underlying number of elements is given by the size of the graph. The examples below assume that the number of elements involved (the size of the graph) is 12.

vSlcs	Expanded
-3	[9]
1::2	[1, 3, 5, 7, 9, 11]
-2::-3,3::4	[10, 7, 4, 1, 3, 7, 11]
1,-2,3	[1, 10, 3]
1:4,6:9,-2:	[1, 2, 3, 6, 7, 8, 10, 11]
1::3,5::3,-2:	[1, 4, 7, 10, 5, 8, 11, 10, 11]

Vertex Directive

The second directive takes the form:

`Vvslcs[R[#]TB]+`

The initial `V` says this is a Vertex directive. The `vslcs` specifies the set of vertices to which this applies. If a vertex appears more than once in the expanded `vslcs` list, it's the same as if it had appeared just once. The `R` (reward) attribute says to associate the specified numeric reward with the indicated vertices. If the `#` is missing, the reward is the default reward from the Graph directive.

The B stands for Block or Barrier. Suppose that W is the set of vertices that `vslcs` specifies and let X be $V - W$. Furthermore, let F be the set of edges that would have existed when the graph was first specified with the G directive – that is, the default set of edges according to the graph

type. Then this directive says to remove all currently existing edges between W and X, and to add in any default edges that hadn't been there: $F - (W \times X) - (X \times W)$.

Extra for experts: For most of the subsequent labs, if a vertex (or edge) has a reward, it will be considered terminal. However, there is a lab where this is not true. Therefore, a small portion of this lab (2%?) of this lab is for implementing the T directive.

1st directive	2nd directive	3rd directive	Net effect (trm => terminal, R => default rwd)
V2			Nothing
V2R			R, trm
V2T			Trm
V2RT			R, non-trm
V3R	V3R7		R7, trm
V3R	V3R7T		R7, non-trm
V3R	V3T		R, non-trm
V4T	V4RT	V4T	R, trm
V4T	V4T	V4R	R, trm
V4T	V4T	V4RT	R, non-trm

The T attribute says to toggle terminality. However, if a non-terminal vertex has no reward yet, then setting a reward (V#R9) sets the reward and makes the vertex terminal, while setting a reward with terminality (V#R9T) sets the reward and makes the vertex non-terminal. In all other cases the appearance of T toggles the already existing terminality.

Note that although it is silly, if there are multiple occurrences of R, T, or B within a given directive, only the last one matters. For example `V5BR7BTR13T` is the same as `V5R13BT`. That is, T (or B for that matter) toggle once within a directive, even if they appear multiple times.

Edge Directive

The third directive has two forms:

```
E[mngmnt]vslcs1[=~]vslcs2[R[#]T]*
E[mngmnt]vslcs1[NSEW]+[=~][R[#]T]*
```

In the first form, the edges are specified by doing a zip between the expanded lists from `vslcs1` and `vslcs2`. In the second form, the list from `vslcs1` are construed to be starting vertices, and `[NSWE]+` indicates directions that one could go in from the starting vertex, if that direction is supported. Ie. in the case of `G6`, `4N` is defined (ie. vertex 1), but `1N` is not defined (ignored, since one can't go north from the top part of the perimeter). For each indicated and existing direction, one gets an edge. For example, `G6 E[mngmnt]1::2NE=…` generates a list of edges (1,2), (3,0), (3,4), (5,2). If an edge appears in the final edge list more than once, it is the same as appearing once.

What to do with the produced edge list is specified by `mngmnt`:

!	Remove any extant edges
+	Add new with properties, ignore (skip) extant edges
*	Add if missing, apply properties to new and previously extant
~	Toggle (default)
@	Apply properties to extant edges

There are only two properties to worry about, R and T, and they work in the same way that they do with vertices. Note that when edge removal is in play, the R and T are irrelevant.

Now the [=~] specifies whether the edges are bidirectional (=) or directed (~). Bidirectional edges are not single edges that go in both directions. Instead, a reference to a bidirectional edge is actually a shortcut for two separate edges. In particular, they may have different rewards, or one may have a reward and the other not. For example, `G6 E1~2 E@1=2R7 E1=2R9 V2B E1~2R` starts off creating a graph of size 6, width 3. `E1~2` says to toggle the (directed) edge going from 1 to 2, meaning that it will no longer exist. `E@1=2R7` says to assign a reward of 7 to the existing (2,1) edge. `E1=2R9` says to toggle both potential edges between 1 and 2. Thus, (2,1) is removed and (1,2) is created with a reward of 9. `V2B` says to remove edges (1,2), (2,5), (5,2), since they exist, and to newly add in edge (2,1), but without reward, as when an edge is removed, all its properties are lost, and those properties are not recovered if the edge is subsequently reinstated in a following directive. The final `E1~2R` creates an edge (1,2) with a reward of 12.

grfStrEdges(graph)

In future labs, we will look for paths within a given graph, which will be output as a subset of the edges. This could get arbitrarily messy, but we will be working within a gridworld where typically most or all of the edges correspond to immediately adjacent cells. This means the available edges from most vertices are going to a cell that is immediately adjacent to the starting cell when laid out on a grid: North (up from), E (to the right of), W (to the left of), or S (below) the starting cell, or some combination thereof. In other words, there are 16 possible combinations of N, E, W, S that can come together for most cells.

Therefore, for each cell in the gridworld, you will indicate which standard directions are possible from that square. The possibilities are:

	J: NW	N: North ^: WNE	L: NE	
<: NWS	W: West	+: NEWS	E: East	>: NES
	7: SW	v: WSE S: South	r: SE	

along with - : EW, | : NS, and . : none

In future labs is a special symbol, *, which indicates that there is a (terminal) reward in the cell.

This means that it is possible to specify the vertex/edge structure of a vanilla gridworld graph with a number of characters that is equal to the size of the graph. In particular, for each vertex, print the letter that corresponds to its local neighbors. It may be, however, that some edges (neighbors) don't fall into this paradigm, namely jumps. For graphs which have jumps, which is all edges of an N graph, the jumps are indicated immediately after any grid struct spec (which for N graphs is the empty string). Their form is semicolon separated vslcs pairings:

$vslcs1_1[\sim=]vslcs1_2;vslcs2_1[\sim=]vslcs2_2;\ldots$

The grader does not check for a most efficient encoding, but the same edge should not occur twice, and the lengths of the expanded vslcs must be the same on both sides of a ~ or =.

Implementation Issues

It is very easy to get mired in this lab, not because it is conceptually difficult but as there are so many little things that one must do just right. It is vital to read the entire spec, then reread it, to understand what your data structure must support. It is also important to write well structured code. For example, you should have a function whose job it is to take vslcs as a string and convert it to the appropriate data struct of integers. It is also likely to be the case that you will redo certain subfunctions that you write as your understanding is amended based on grader feedback.

The grader tests your script in batches of 20, with .3 seconds per each test, which should be plenty of time. The entire test is terminated if any batch times out or the overall average drops to 50% or less. Per batch, the output will show the first test number in the batch (1, 21, 41, etc.) and then for each test it will show two letters and a possible number. The first letter indicates the test type that failed:

P: grfParse()	Z: grfSize()	N: grfNbrs()	V: grfVProps()
G: grfGProps	E: grfEProp()	S: grfStrEdges()	A: grfStrProps()
C: cmd line test			

The second letter indicates the type of failure:

M: function Missing
S: Script error
T: Timeout
V#: Value error – the # indicates the index of the first vertex where an error was encountered

For cmd line output there is also:

O: Output not found (ie. nothing was printed)
K: Keyword not foud (ie. missing Graph: at the start of a line)
E: Edges not found (ie. missing the string from grfEToStr() after the keyword)
J: Jump redundancy (ie. the same edge is encoded twice within the jumps section or a local neighbor is encoded as a jump).

It is possible that during the development of this lab you may find it expedient to alter data structures that may seem to have been suggested, and it is probably to your advantage to discover these types of issues earlier rather than later. To this end, do not try to implement everything before your first visit to the grader. Instead, it is strongly suggested that you implement a few functions and keep expanding. For example, it should be straightforward to implement an initial `grfParse(lstArgs)` by ignoring all but the first argument! The point of this is that you will have defined a function with admittedly limited utility, but you will see that the grader respects it. Of course, the next thing is to do is put in the `grfSize()` function, and then handle rewards on the vertices. Then there is a significant escalation in complexity when you handle the B directive with vertices (I suggest implementing the T directive only after handling edges, since its grade importance is quite low). There will be another step up as you move on to handle the various edge directives.

Development Strategy

The grader tests in the following order:
10 tests: Graph directive
60 tests: Vertex directive (without R and T options)
20 tests: E~vslcs=vslcs sans jumps
20 tests: E~vslcs~vslcs sans jumps
16 tests: E~vslcs[=~]vslcs with jumps
16 tests: E~vslcs[NSEW]+[=~]
14 tests: E[~!+]…
40 tests: rewards and all forms of E, but no T option
4 tests: all possible forms including with R and T options.

Therefore, for your initial development efforts, the first thing you should focus on exclusively is handling the Graph directive and to put in stub functions that return enough for the grader to be happy (see page 4 of this document). Here's an example:

`grfParse(['G8W4']):`	just has to return a graph data struct without toppling over
`grfSize(graph):`	8
`grfGProps(graph):`	`{'rwd':12, 'width':4}`
`grfVProps(graph, 1):`	{}
`grfNbrs(graph, 3):`	[2,7] or {2,7} or {2:some, 7:thing}
`grfEProps(graph, 4, 5):`	{}
`grfStrEdges(graph):`	'rvv7L^^J'
`grfStrProps(graph):`	`'rwd:12, width:4'`

Note that for the first 156 tests, `grfVProps()` and `grfEProps()` return the same thing and `grfSize()`, `grfGProps()`, and `grfStrProps()` vary minimally. There is currently no cmd line testing, but when you run your script, you should output `grfStrEdges(graph)` shown in 2D followed by `grfStrProps(graph).` Note that grfStrProps(graph) at this point is just comma separated list of key:val pairs.