Data Structures and Algorithms

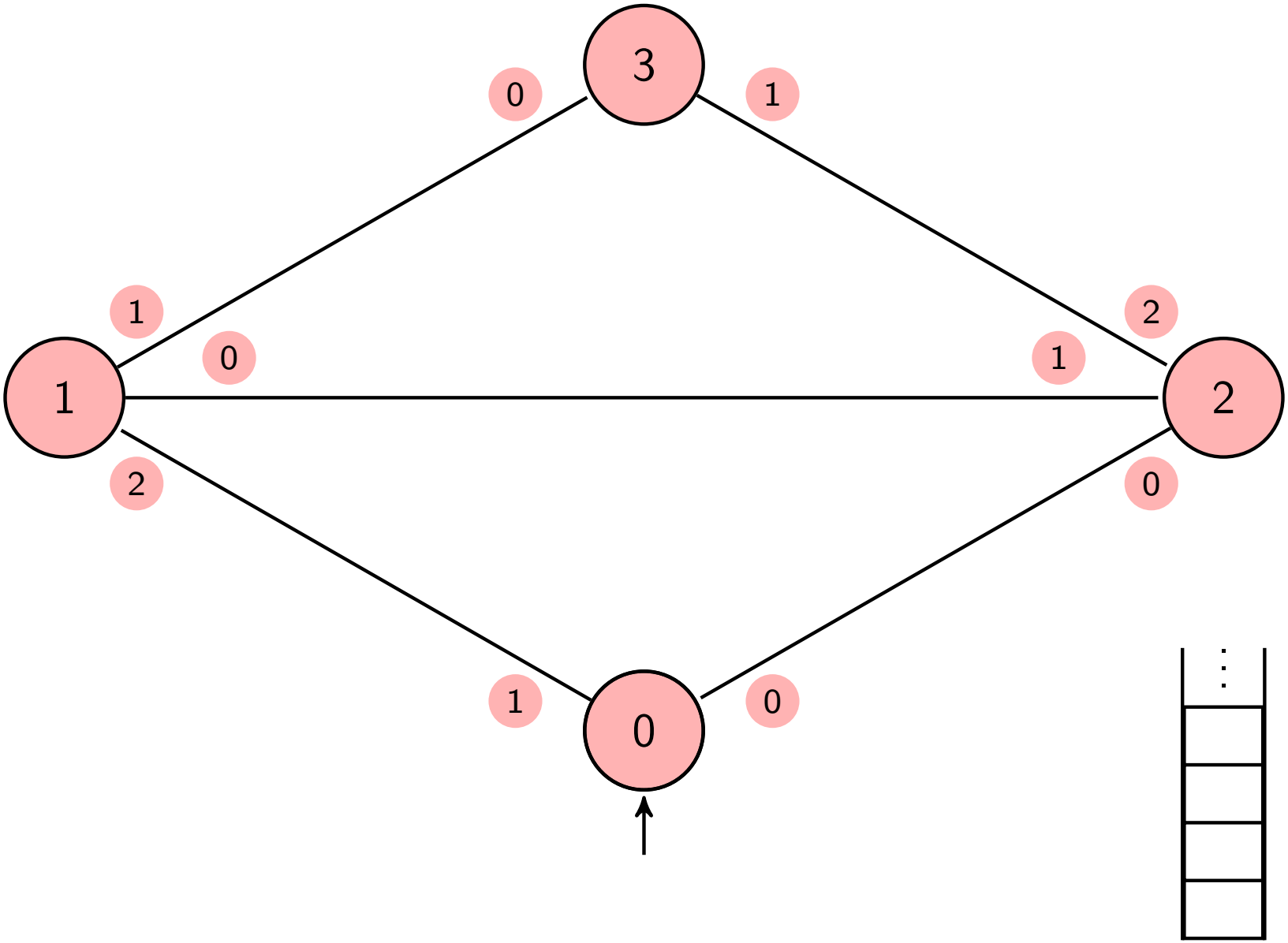
Assignment 2

Alan P. Sexton

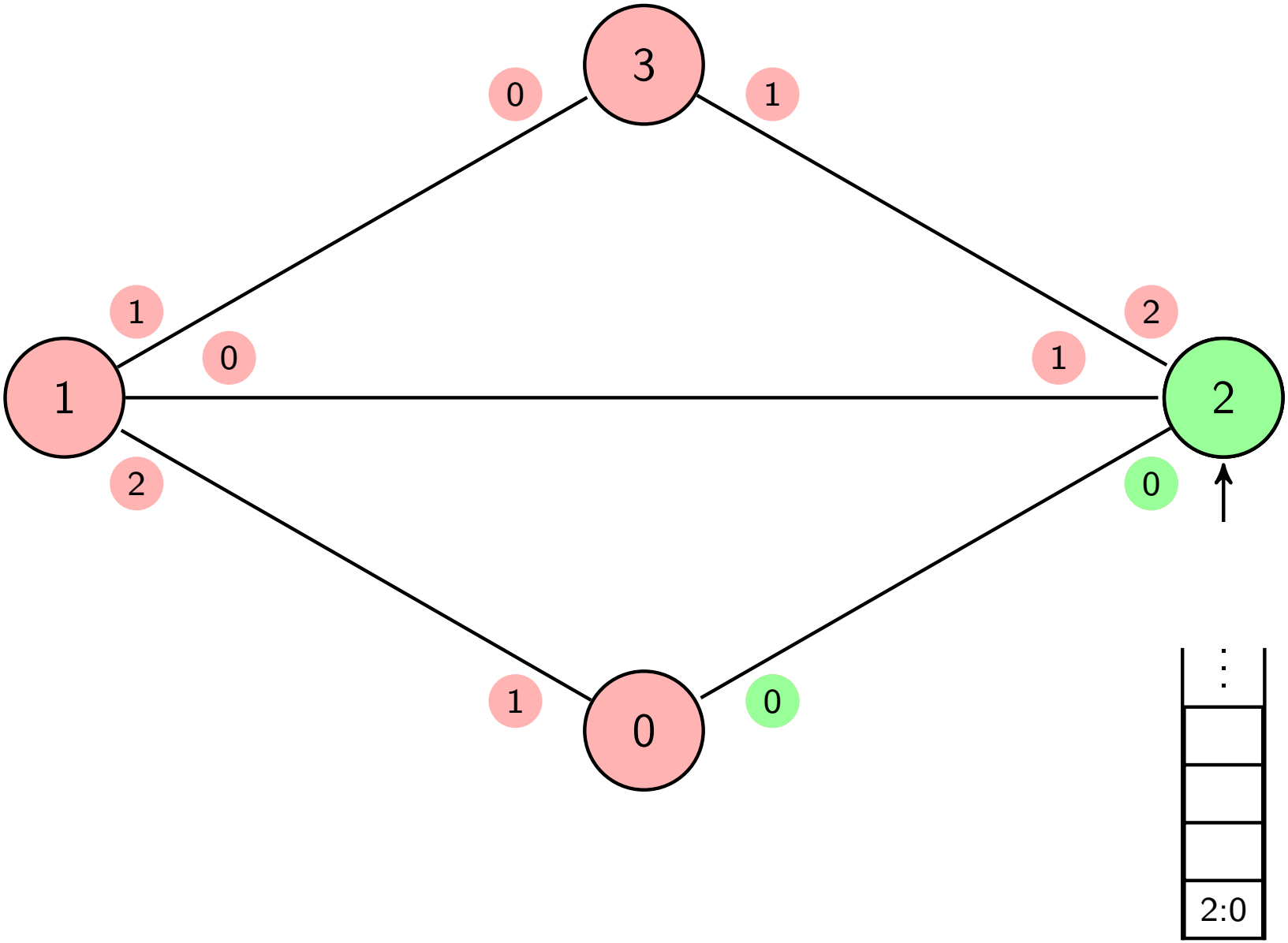
University of Birmingham

Spring 2020

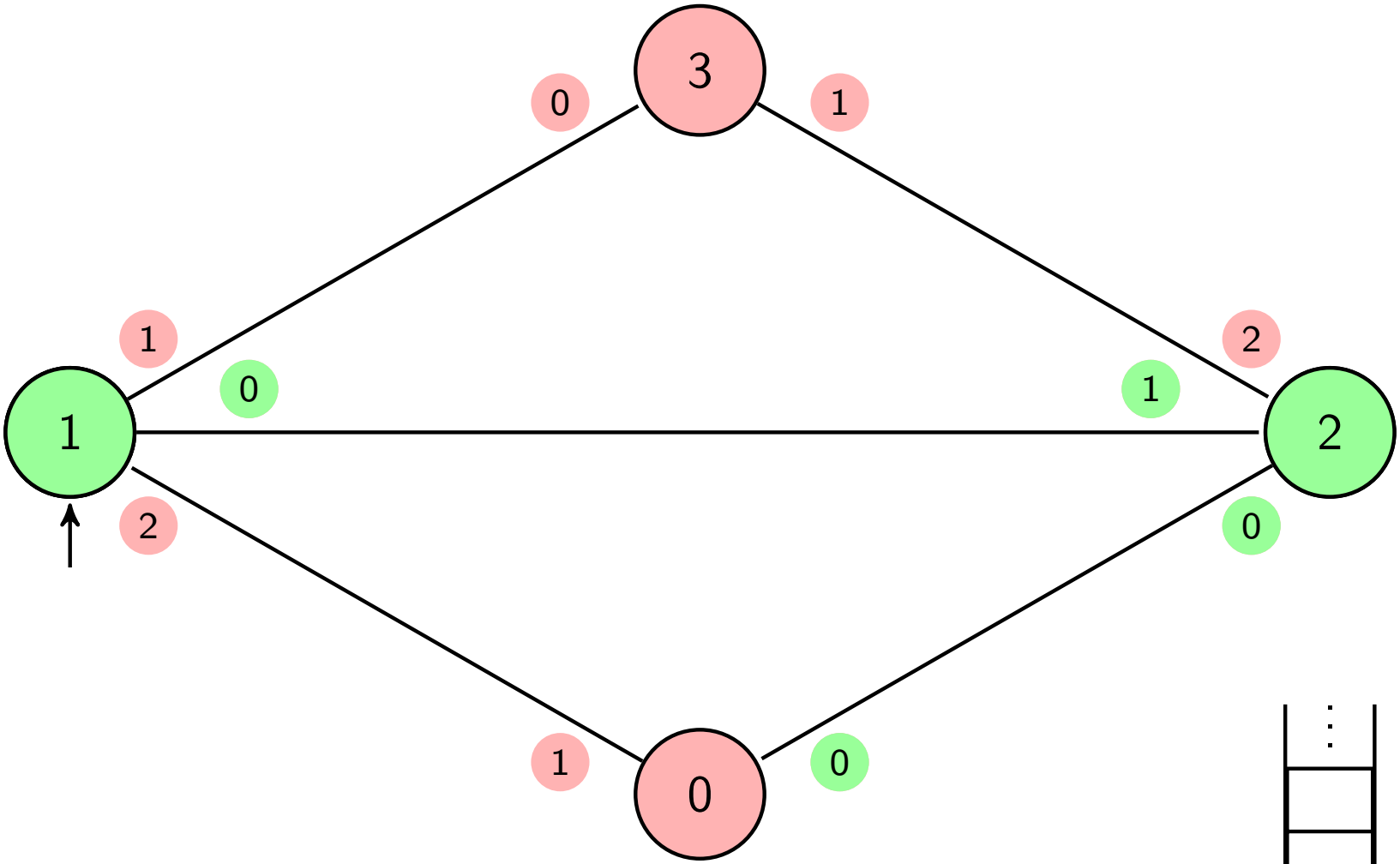
Example Search Execution



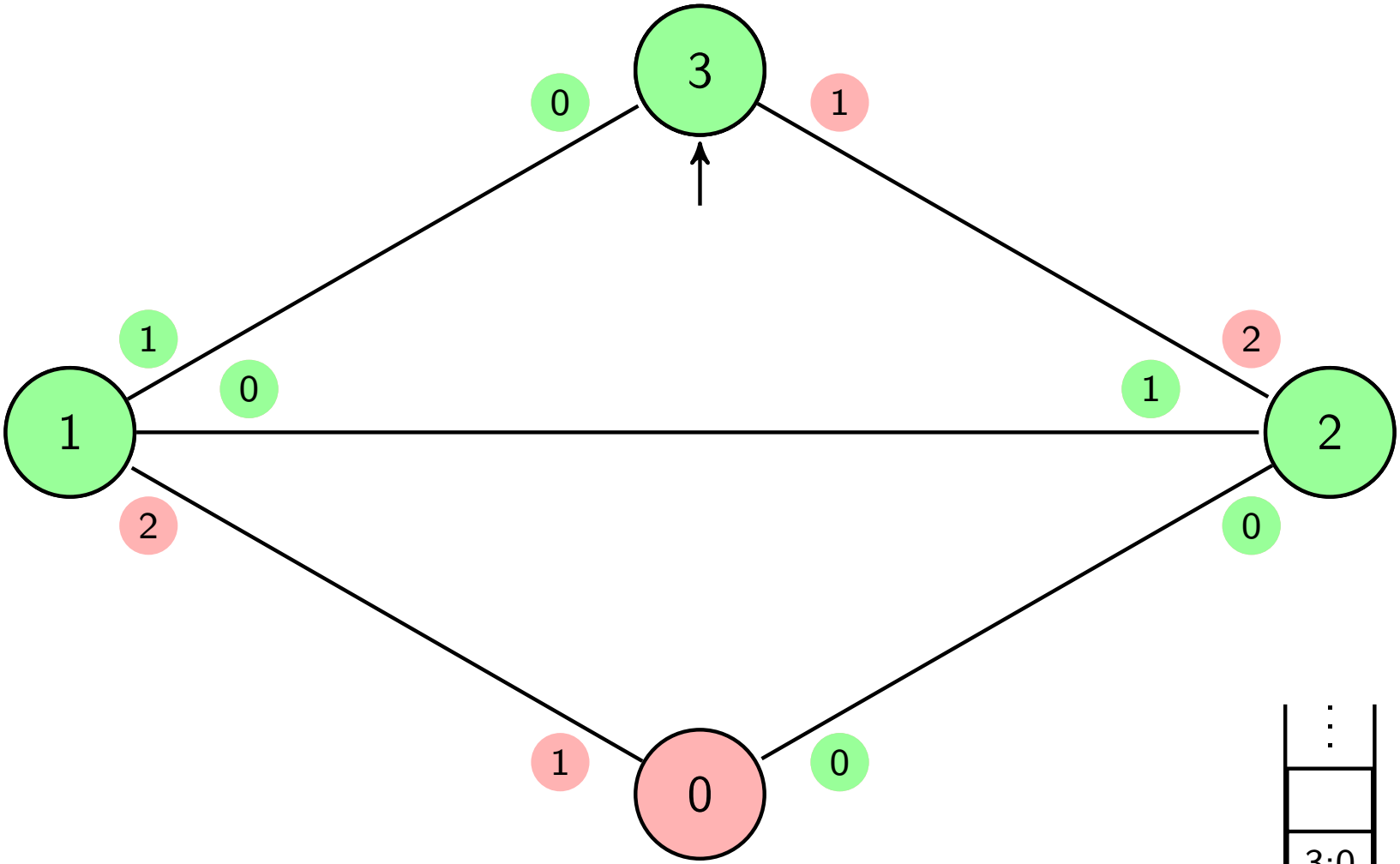
Example Search Execution



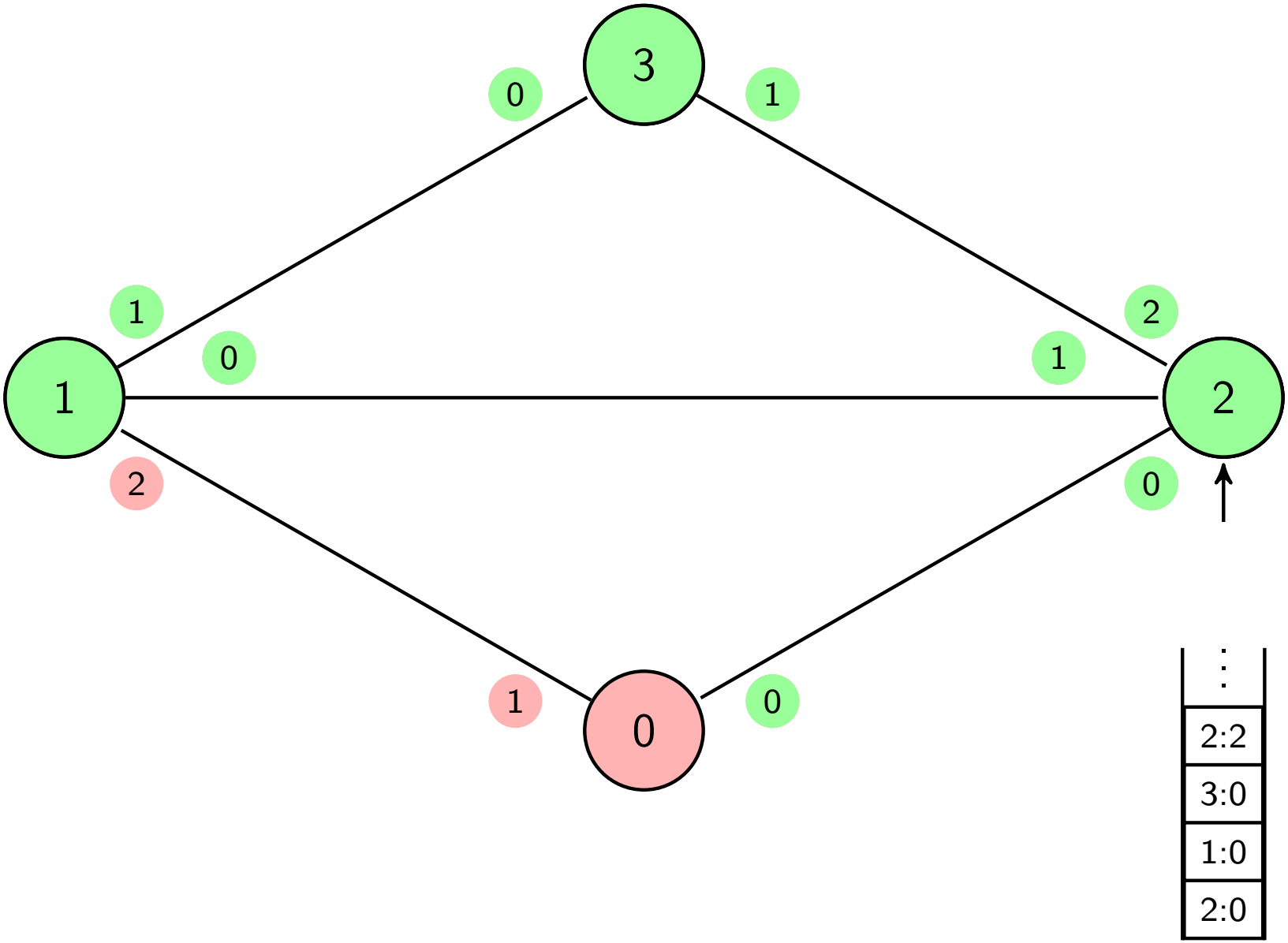
Example Search Execution



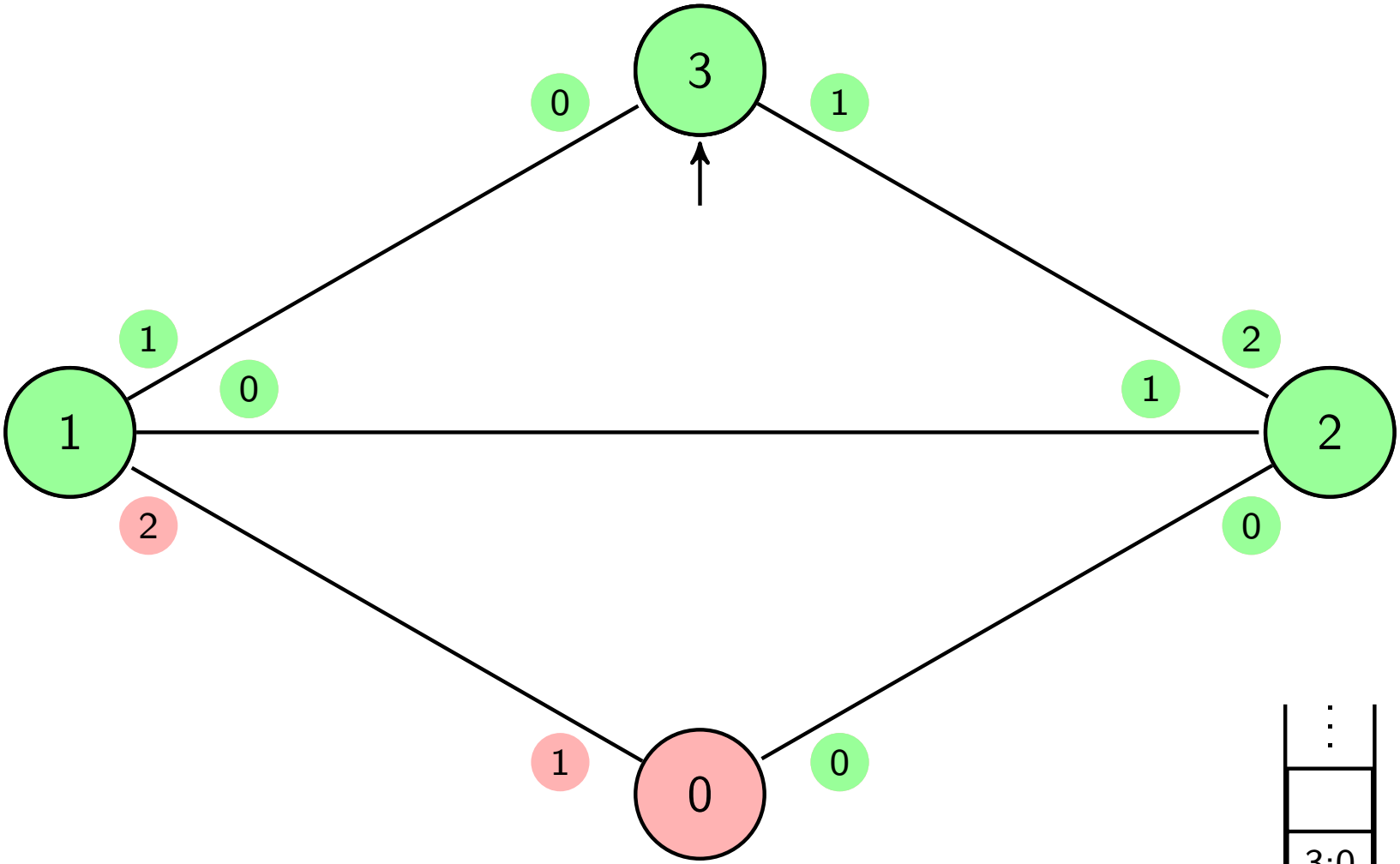
Example Search Execution



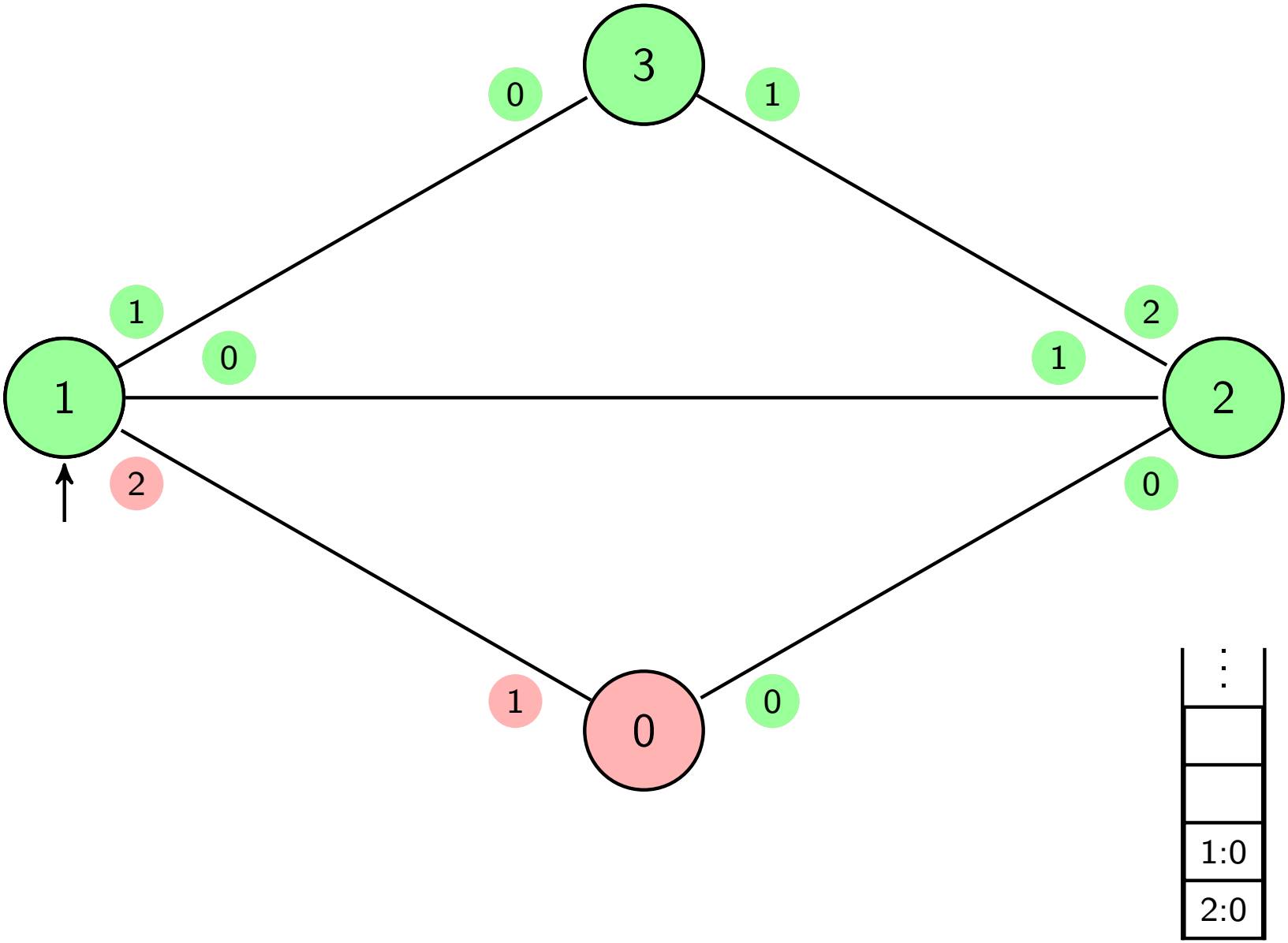
Example Search Execution



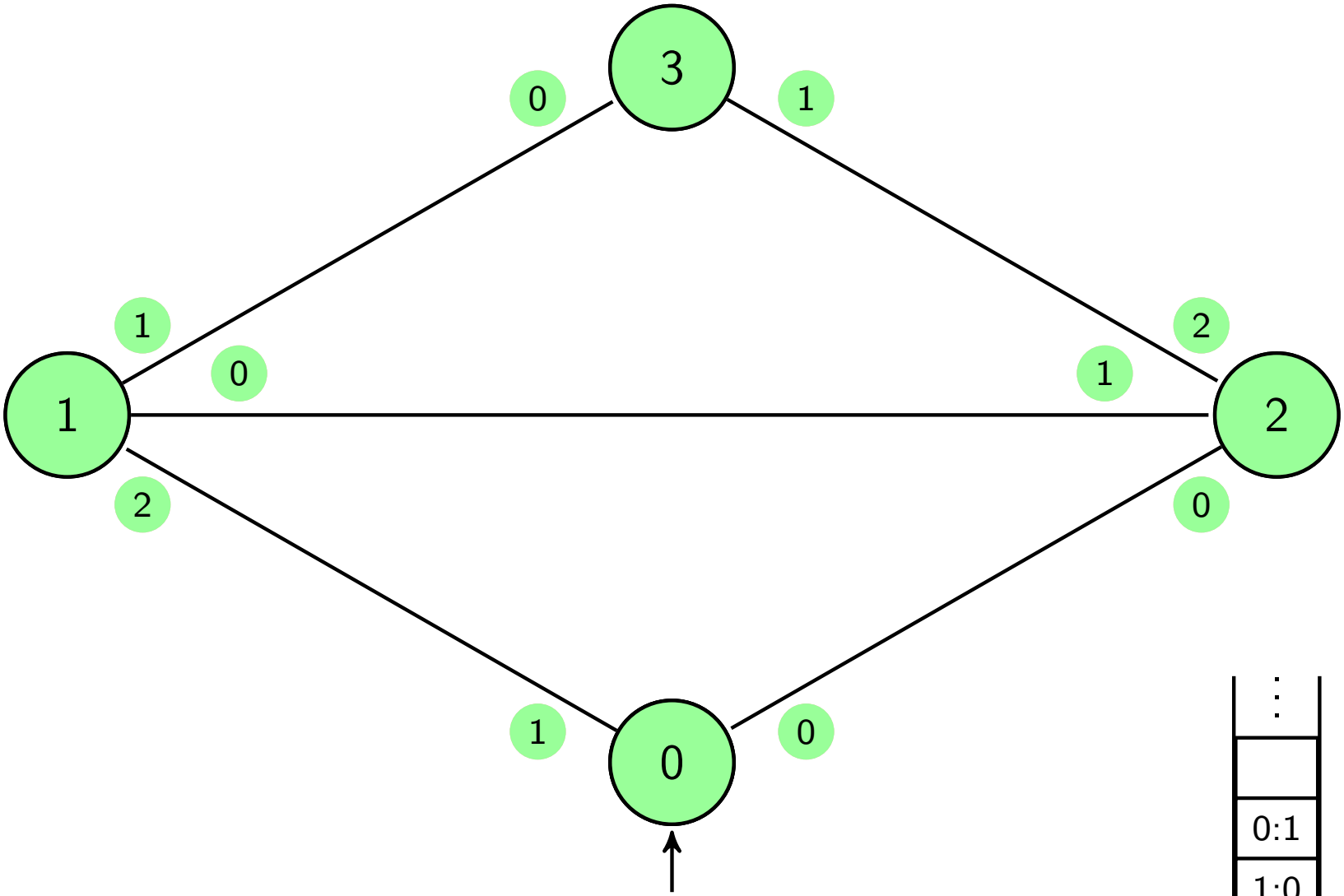
Example Search Execution



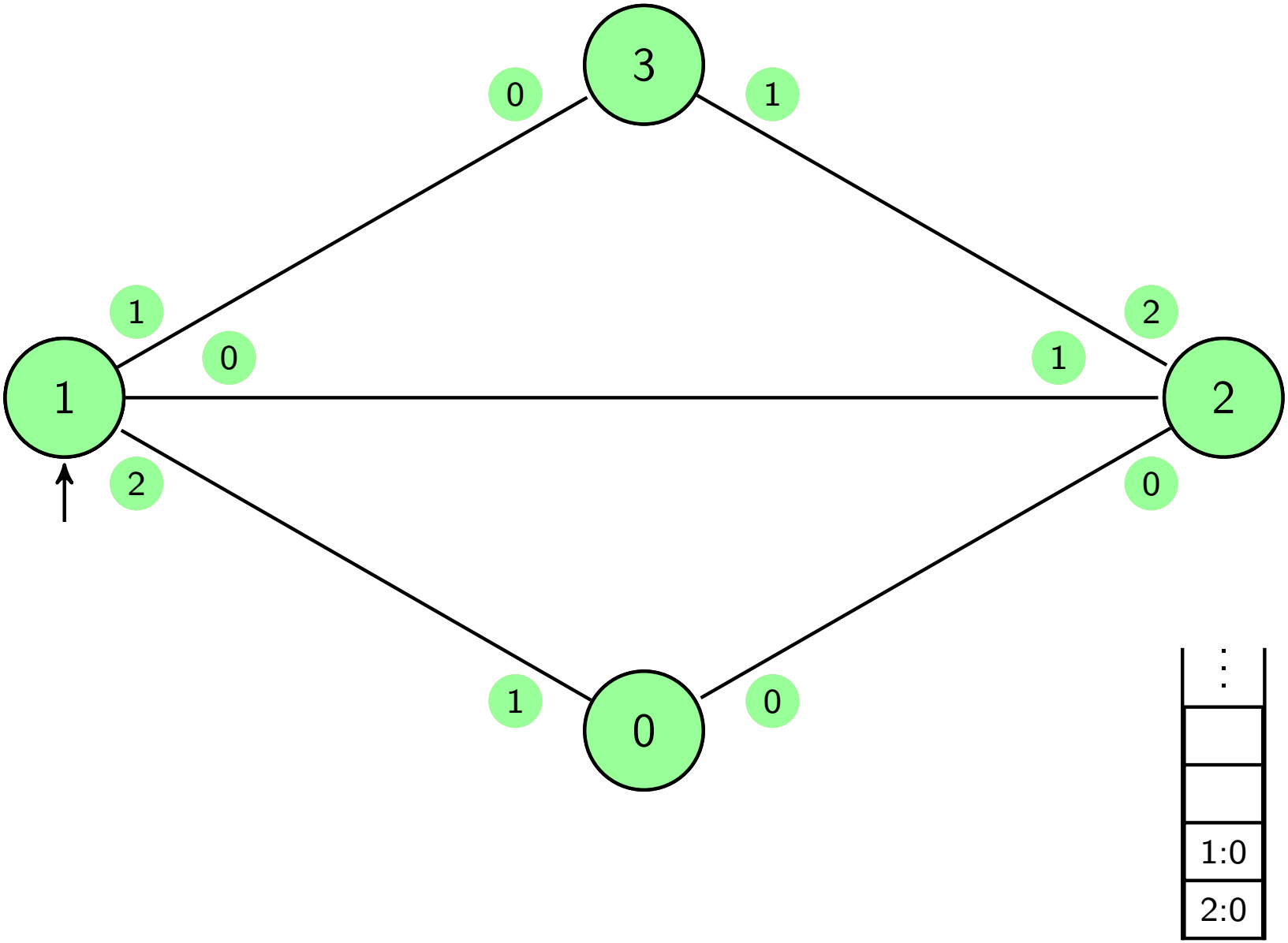
Example Search Execution



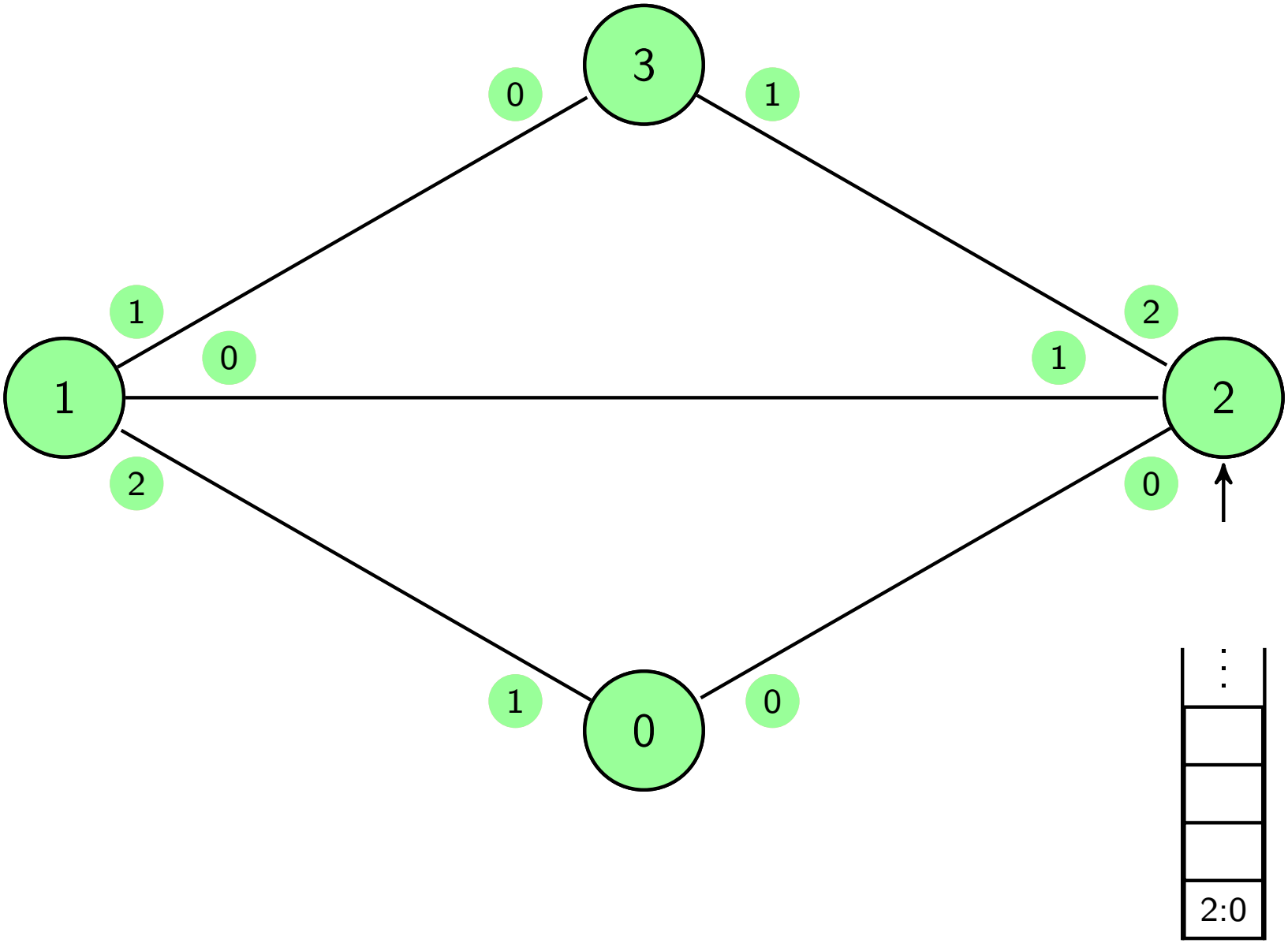
Example Search Execution



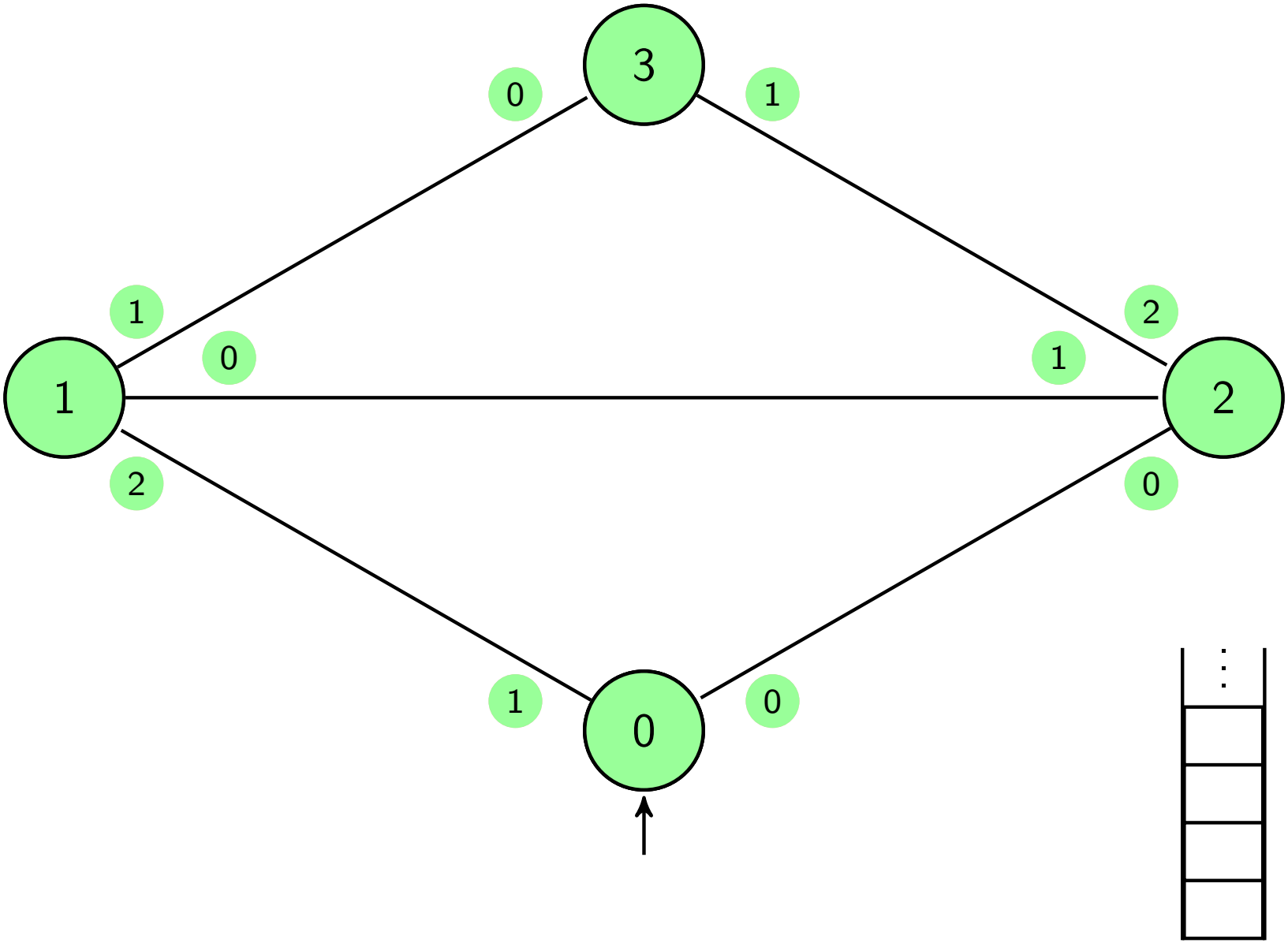
Example Search Execution



Example Search Execution



Example Search Execution



Choosing Data Structures

When programming in Java and a mechanism for holding, searching within, and manipulating collections of objects of some kind is needed, beginner programmers often reach for an `ArrayList<>`

- Very easy to understand
- Easy to manipulate `add(index, element)`, `get(index)`, `remove(index)`, `contains(object)`, etc.

However, while `ArrayList<>` is good at many things, there are other collection classes that are much better for some tasks

- Better means: enables shorter, simpler, more efficient code

When you find yourself reaching for `ArrayList<>`, **FIRST** consider whether your problem would be better dealt with by a:

- `List<>`, `Deque<>`, `Set<>`, `Map<>`, or one of their variants

Choosing Data Structures

When choosing a data structure, first identify the operations that your problem requires of the collection of objects you are working with

- This does not mean *"can I implement what I need using* `ArrayList<>`*?"* (the answer is usually yes, with sometimes a lot of effort), but rather *"what operations need to work in the end, and do I have to implement them or does the data structure provide them for free?"*
- A further question to bear in mind is: *"How efficient will the operations provided be? In particular, will they be efficient enough for my purposes?"*
- With these questions answered, you are in a position to select a data structure, while also paying attention to the complexity costs of the operations required using your selection.

Data Structures for Assignment 02

- The stack for handling back-tracking is already provided:
  - `Deque<Portal> visitStack`
- A data structure (`visited`) to keep track of which portals you have already traversed is also needed
- Use it when in a chamber to choose an unused portal to traverse, or, if there is none, to decide to back-track
- You could use an `ArrayList<>` for this, but you have to do more programming, and more complicated programming, to make it work than is necessary
- To choose an appropriate data structure, you need to think about what operations are needed. On this data structure we need to be able to efficiently and easily:
  - add a new portal that we have just visited to the structure
  - check if a portal is one that we have already visited, i.e. check if it is in the structure.

Data Structure for Visited Portals

- To use an `ArrayList<>` for this:
  - adding a portal would be easy, but you could end up adding the same portal multiple times: so you have to write code to check if it is there before you add it
  - checking if a portal is there already means iterating through the `ArrayList<>`, not hard to program, but inefficient and just adds more unnecessary code
- Intead, we use a `Set<>` data structure: in our case we will use a `HashSet<>`, which implements the `Set<>` interface:
  - Add a portal with the `add(portal)` method: no need to write code to check if the portal is already there
  - Check if the portal is in the structure with the `contains(portal)` method: this is very efficient

For every data structure you use, read through the API and make sure you are familiar with the different methods available to them, paying particular attention to their return values

Don't forget to look at the utility static methods in the `Collections` and `Arrays` classes