A GP 2 Implementation of the Bellman-Ford Algorithm

Graham Campbell 

September 2019

There are various equivalent formulations of the shortest path problem, such as finding a shortest path between two vertices or between any vertex and some fixed destination. It is equivalent to solve the so called *single-source shortest path problem*: Given some fixed source vertex, find the weight of the shortest path from this to each of the other vertices.

When weights are induced by embedding a connected graph in $\mathbb{R}^n$, for sufficiently large n, clearly the shortest path function is a metric, however it is often useful to consider weights that are not metrics or even quasimetrics (symmetry not required). The Bellman-Ford algorithm, first proposed by Shimbe in 1955 [1], solves the single-source shortest path problem in the most general case, in $O(|E| \cdot |V|)$ time, and has the lowest worst-case complexity of any algorithm known to solve this problem.

The algorithm works by *relaxing* edges, progressively decreasing an estimate of the shortest path from each vertex from the source. In this Chapter, we will show that the Bellman-Ford algorithm can be adapted so that it can be implemented in a rule-based graph language. Moreover, by using root nodes in GP 2 [2], we can achieve the same worst-case time complexity, given a graph of bounded degree.

Problem Specification

We start by specifying the single destination shortest path problem in a way that makes sense for GP 2. We shall represent the source node by colouring it red, and weight represented by integer labels on edges. We now define an input graph, and a shortest path:

Definition 1 (Input graph)**.** An *input graph* has no root nodes and is unmarked apart from exactly one node coloured red (the source node). Nodes may be arbitrarily labelled, but all edges must be integer labelled (with weights).

Definition 2 (Shortest path)**.** Given an input graph and a vertex v, the weight of the shortest path between the source and v, if it exists, is the smallest weight of any path between the source and v, where the weight of a path is defined to be the sum of the weights of its edges. If no between the source and v exists, then we define the weight of the shortest path to be ∞.

Proposition 3 (Shortest path characterisation)**.** A shortest path weight between the source and v exists iff there does not exist a path between the source and v containing a cycle of negative weight.

Clearly, we can satisfy this condition by requiring the graph to be acyclic or have only positive weights, however this may to too restrictive in practice, or simply inconvenient. The Bellman-Ford algorithm is correct on all weighted graphs, and is able to detect when a shortest path doesn't exist due to (countably) infinitely many paths between the source and v with infinitely descending weight. Note that by the definition of shortest path weight, even if there is no path, there is still a shortest path weight (defined to be infinite).

Definition 4 (Shortest paths specification)**.** The shortest paths specification is as follows.

- *Input*: An *input graph*.
- *Output*: Fail iff there does not exist a shortest path weight between the source and any other node. Otherwise, produce the input graph augmented with a green loop on each vertex labelled with the weight of its shortest path if it is finite, otherwise there is no additional looped edge.

GP 2 Implementation

The key to being able to being able to implement this specification efficiently in GP 2 is using DFS. The program (Figure 1) starts by visiting every node using a DFS, pushing each new node on to a stack as it goes. After the DFS, we initialize every reachable node with an infinite distance, and the source with distance zero. As we do this, we also take a copy of the stack. The reason for having two stacks of nodes is that we can now perform a doubly-nested iteration of the vertices, allowing us to execute the Bellman-Ford algorithm.

```
Main = init; DFS!; Setup; Relax; Check
DFS = forward!; try back else break
Setup = setup_0; setup_1; setup_2!; setup_3
Relax = (try outer_next then RelaxInner else break)!; outer_return_0!;
outer_return_1
RelaxInner = (RelaxEdge; try inner_next else break)!; inner_return!
RelaxEdge = {relax_outgoing_0, relax_outgoing_1}!; unmark_outgoing!
Check = (CheckEdge; try inner_next else break)!; CheckEdge;
inner_return_0!; inner_return_1
CheckEdge = if {relax_loop, relax_outgoing_0, relax_outgoing_1} then fail
```

Figure 1: The GP 2 program `bellman-ford`

The correctness of our implementation follows from the correctness of DFS, adding every node reachable from the source to a stack, and then the correctness of the original Bellman-Ford algorithm, as given in Theorem 24.4 of [3]. For our purposes, a *node stack* is a non-empty linked list with a single outgoing dashed edge leaving every node in the list, connecting to some node not in the linked list. We call the first node in the list the *head*.

Proposition 5 (Correctness of Initialisation)**.** The program `init; DFS!; Setup` is correct with respect to the following specification.

- *Input*: An *input graph*.
- *Output*: The input graph augmented with a blue *node stack* and a green *node stack*, such that:
  1. For both stacks, their head's outgoing dashed edge must connect to the red source node. Every node reachable from the red source in the input graph must be connected to both node stacks by a dashed edge, and only those nodes. Moreover, the head of both stacks must be a root node, and no other nodes may be roots.
  2. The red source node must have, added, a single green looped edge labelled with 0.
  3. Every node reachable from the red source node must now be unmarked, and must have, added, a single green unlabelled looped green edge.
  4. No extra nodes and edges can exist in the output, other than those just described. None of the labels in the input can be modified.

Proposition 6 (Correctness of Relax and Check)**.** The program `Relax; Check` is correct with respect to the following specification.

- *Input*: A graph satisfying the output of Proposition 5 given a valid input.
- *Output*: Ignoring the graph stacks and green loops in the input, the output must satisfy the shortest paths specification (Definition 4).

Theorem 7 (Correctness of `bellman-ford`)**.** The program `bellman-ford` fulfills the shortest paths specification.

Proof. By Propositions 5 and 6, we have the result, since the `bellman-ford` program is the sequential composition of the two sub-programs shown to be correct. □

Proposition 8 (Complexity of Initialisation)**.** The program `init; DFS!; Setup` terminates in $O(|V|)$ time.

Proposition 9 (Complexity of Relax)**.** The program `Relax` terminates in $O(|V|)$ time.

Proposition 10 (Complexity of Check)**.** The program `Check` terminates in $O(|V|^2)$ time.

For graphs of bounded degree, $|V|$ and $|E|$ only differ by a constant, thus our program has exactly the same time complexity as the standard implementation.

Theorem 11 (Complexity of `bellman-ford`)**.** Given an input graph of bounded degree, `bellman-ford` will terminate in quadratic time with respect to the number of nodes in the input graph.

Proof. By Propositions 8, 9, and 10, we have the result, since the `bellman-ford` program is the sequential composition of the three sub-programs, which have at most quadratic time complexity. □

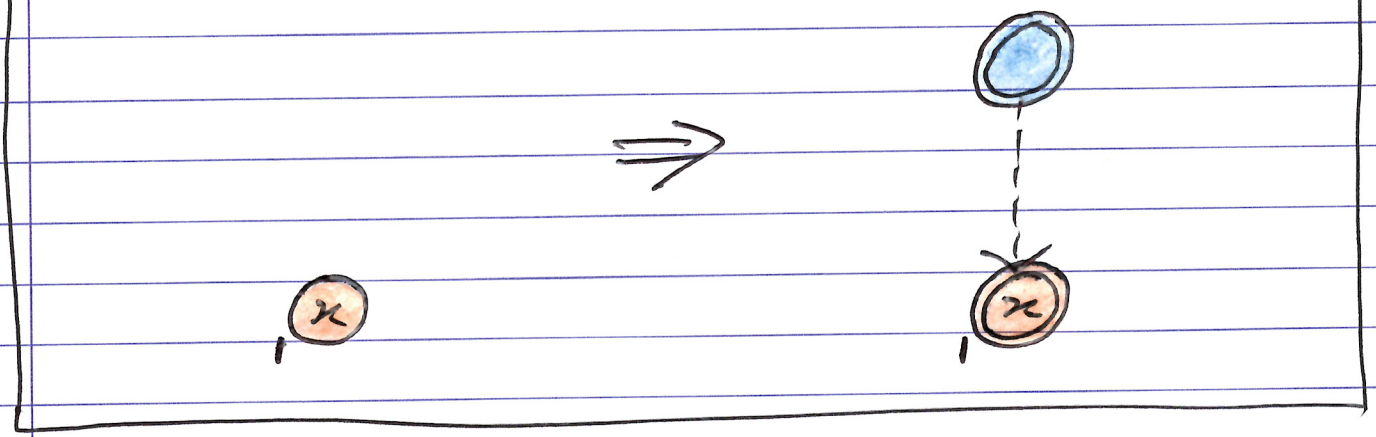
References

[1] A. Shimbe, "Structure in communication nets", in *Proc. Symposium on Information Networks (New York, 1954)*, Brooklyn microwave research institute. Polytechnic institute, 1955, pp. 199–203.

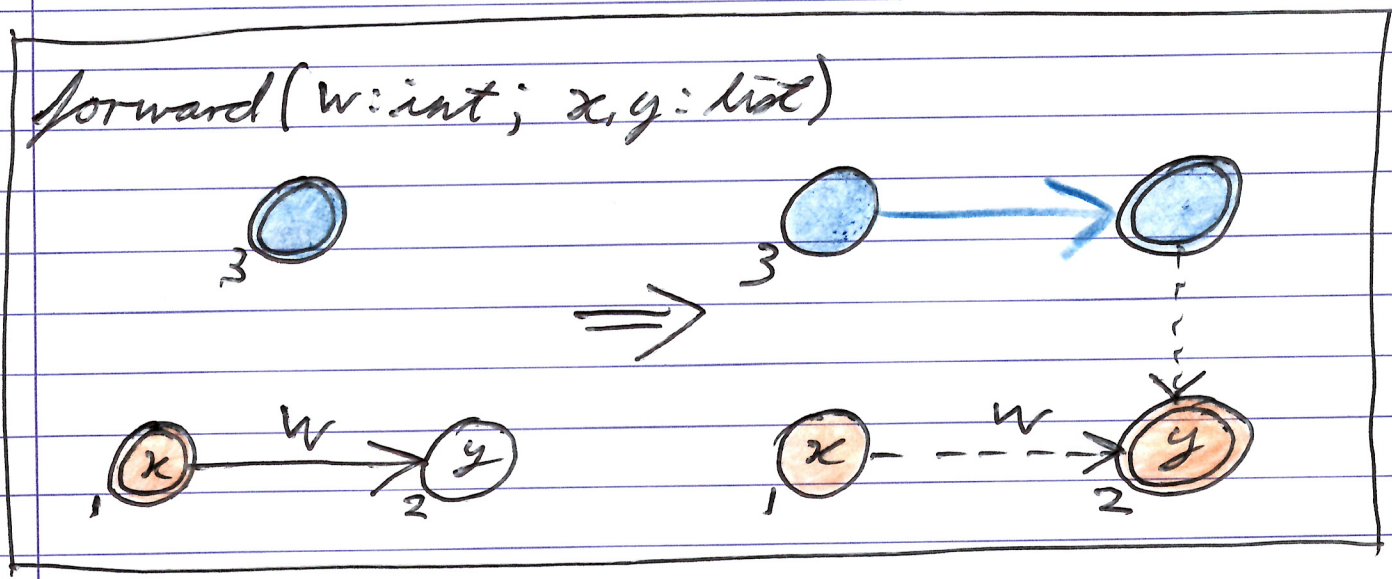
[2] C. Bak, "GP 2: Efficient implementation of a graph programming language", PhD thesis, Department of Computer Science, University of York, UK, 2015. [Online]. Available: `https://etheses.whiterose.ac.uk/12586/`.

[3] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

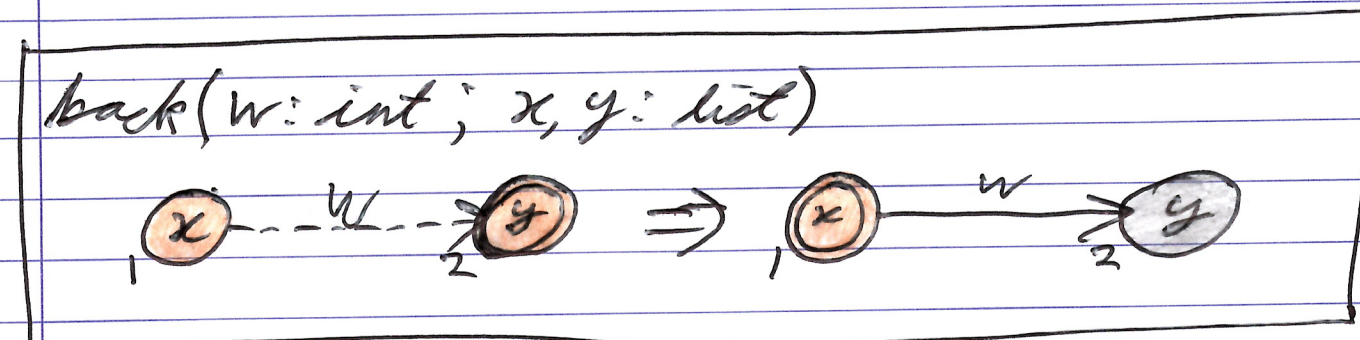
init(x: list)



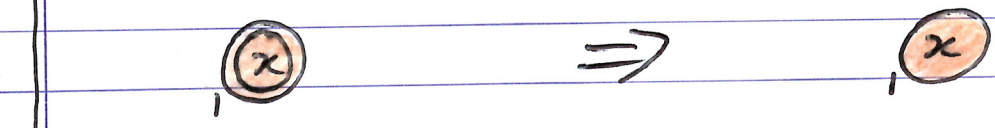
forward(w: int; x, y: list)



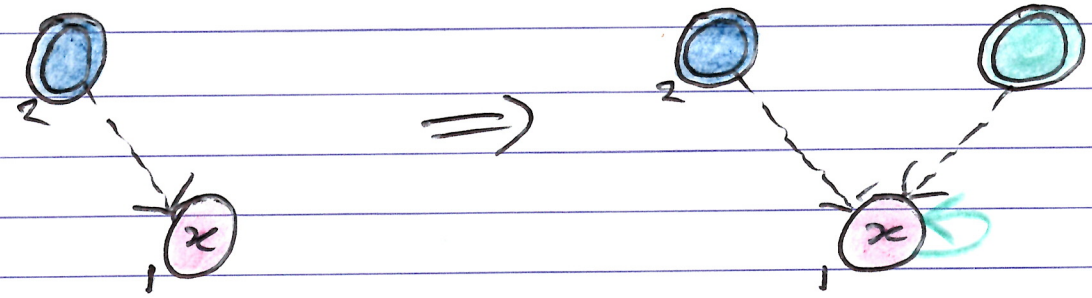
back(w: int; x, y: list)



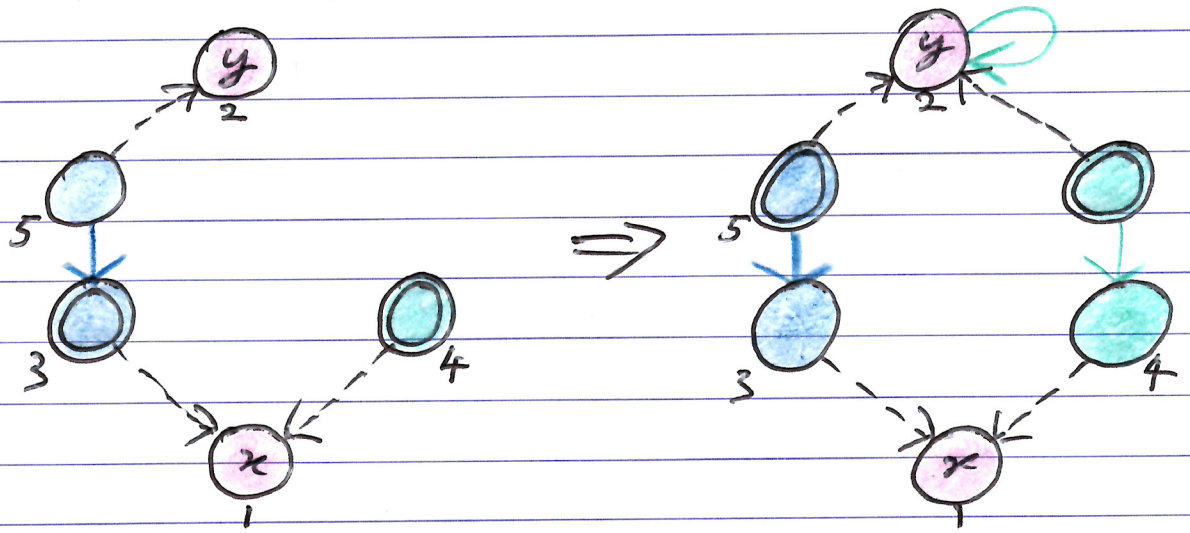
setup_0(x: list)



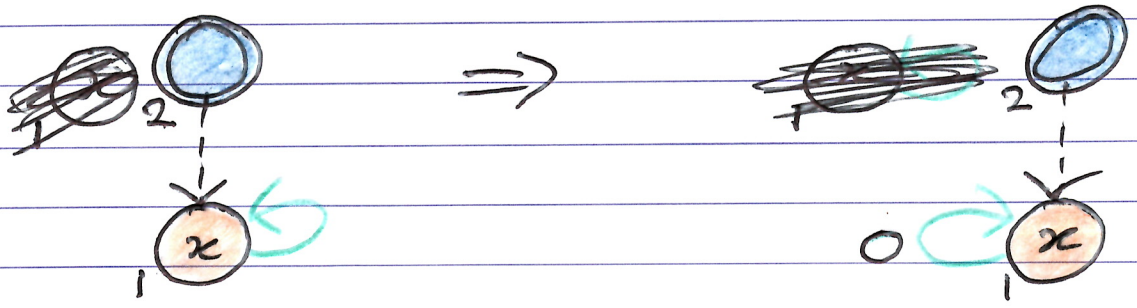
setup_1 (x : list)



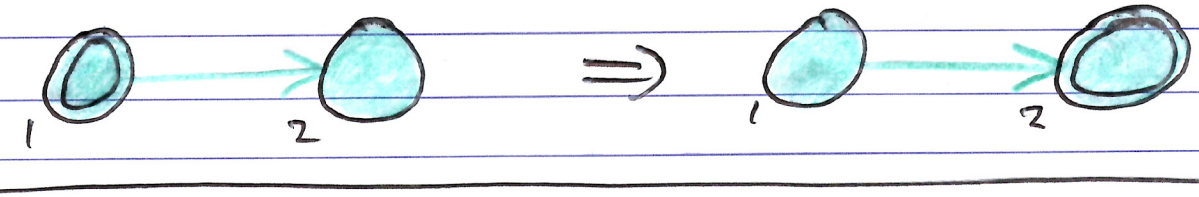
setup_2 (x, y : list)



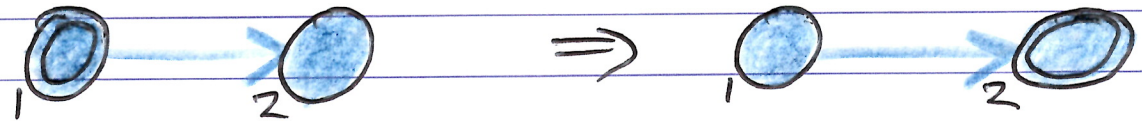
setup_3 (x : list)



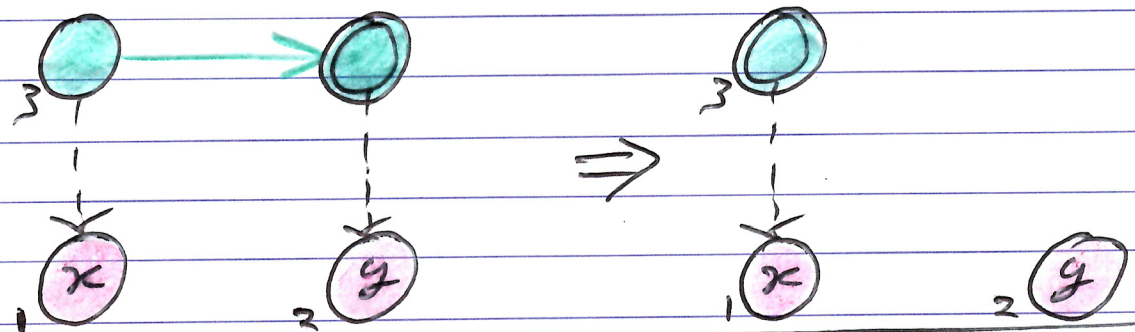
outer_next()



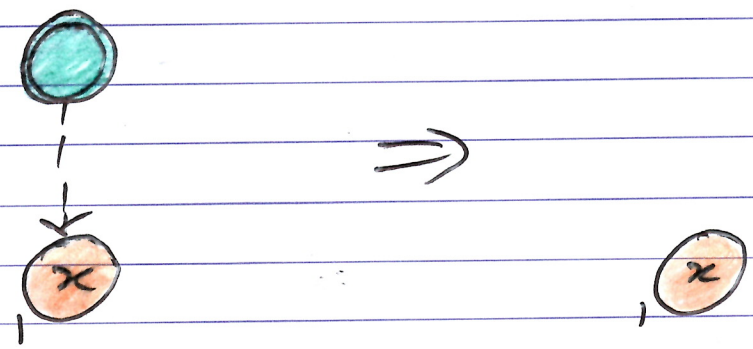
inner_next()



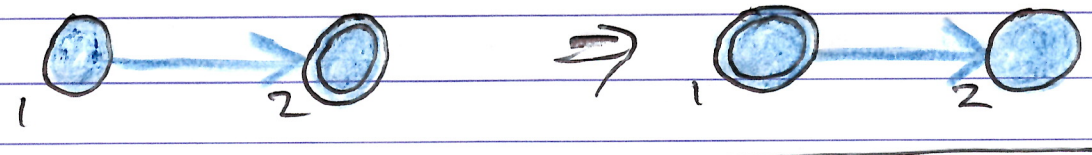
outer_return_0(x, y: list)



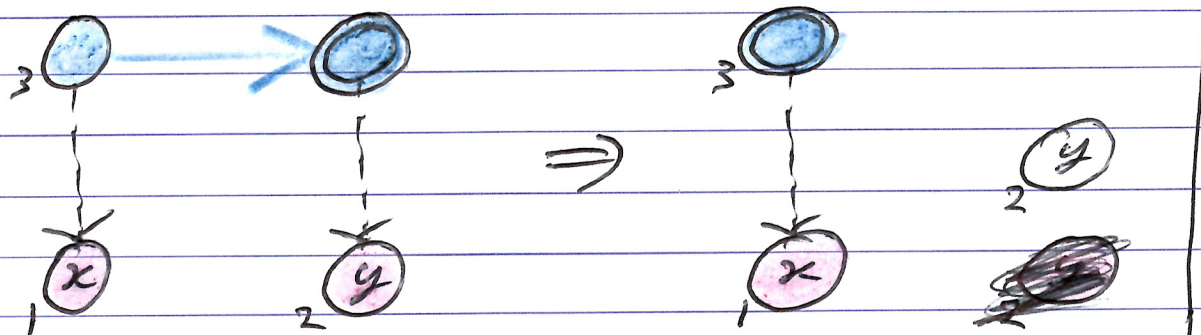
outer_return_1(x: list)



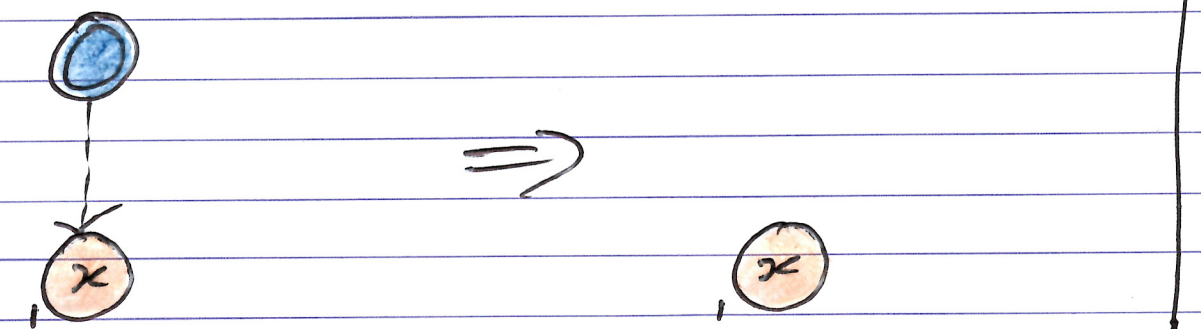
inner_return()



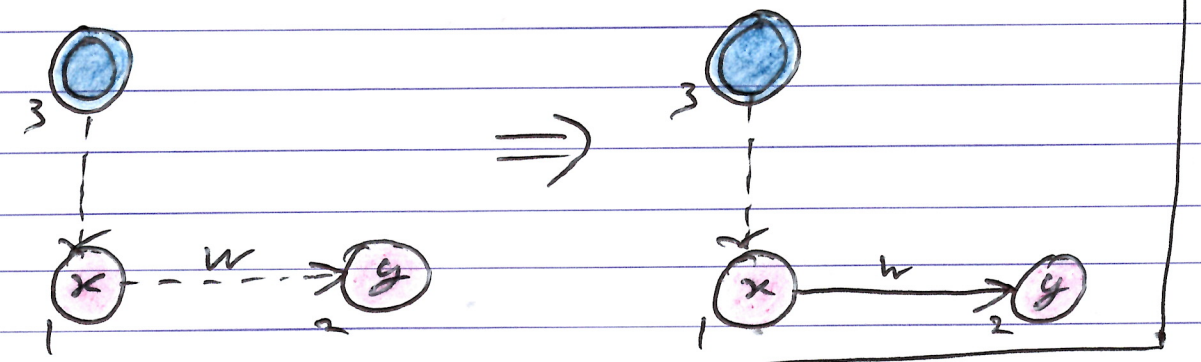
inner_return_0(x, y: list)



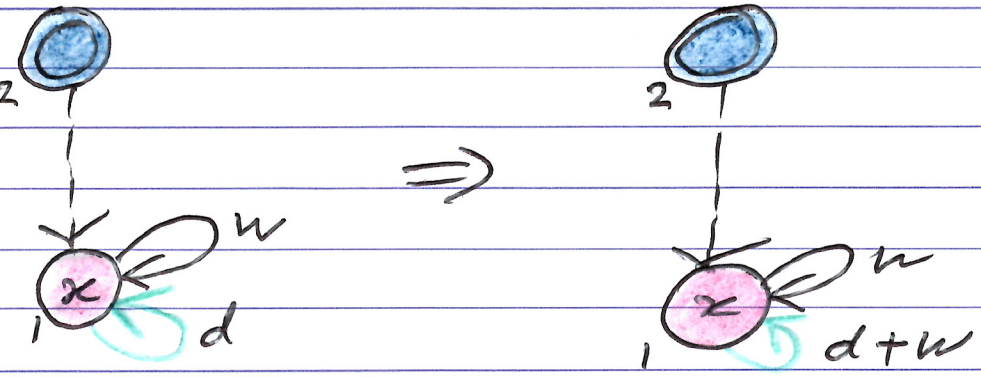
~~vector~~ inner_return_1(x: list)



unmark_outgoing(w: int; x, y: list)

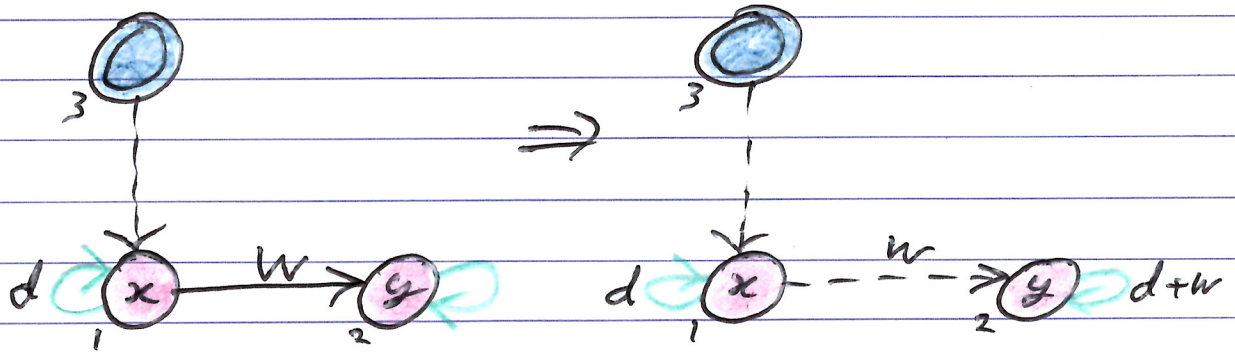


relax_loop(d,w: int; x: list)

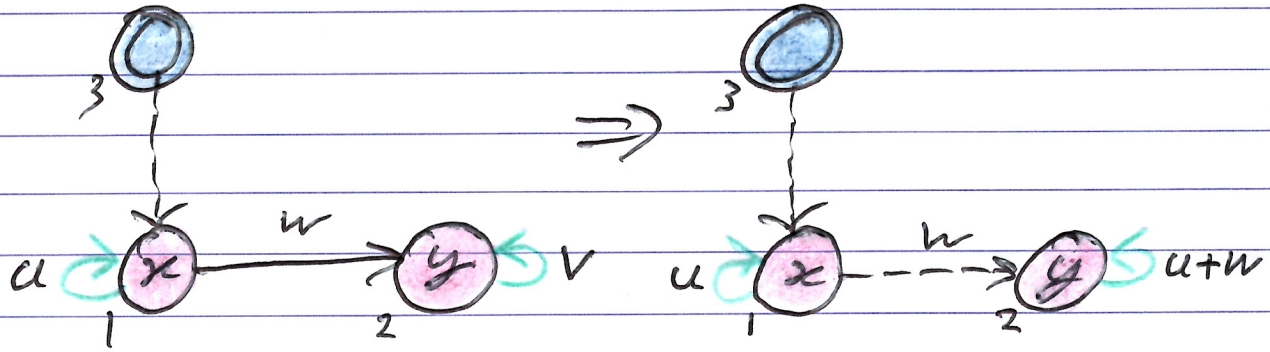


where $w < 0$

relax_category_0(d,w: int; x,y: list)



relax_category_1(u,v,w: int; x,y: list)



where $v > u+w$