CSE 421 HW 2

Yunxin Gan

October 2024

1 Function ordering

2 Many degrees of Kevin Bacon

Shortest path = a path where the number of hops required to reach Kevin Bacon is less than or equal to hops required for all other paths.
Pseudocode:
Global initialization: mark all vertices "unvisited"

Algorithm 1 Pseudocode

Result: Return all shortest paths to t from s

```
BFS(s)
 mark s "visited"; R ← {s}; layer L_0 ← {s}; i ← 0
 if s = t then
 |  return 1
end
paths ← {s : 1}
 while L_i not empty do
 |  L_{i+1} ← ∅
 |   found ← false
 |   for each u ∈ L_i do
 |   |  for each edge (u, v) do
 |   |   |  if v is "unvisited" then
 |   |   |   |  mark v "visited"
 |   |   |   |   paths[v] = 0
 |   |   |   |   Add v to set R and to layer L_i + 1
 |   |   |  end
 |   |   |  if v ∈ L_i + 1 then
 |   |   |   |  paths[v] ← paths[v] + paths[u]
 |   |   |  end
 |   |   |  if v = t then
 |   |   |   |  found ← true
 |   |   |  end
 |   |  end
 |   |  mark u "fully-explored"
 |  end
 |  if found then
 |   |  break
 |  end
 |  i ← i + 1
end
return paths[t]
```

Proof of termination:
Each node is added to R and marked "visited" at most once. Since BFS ensure that nodes are explored in layers, that means each node is visited at most once.
Since each node is visited at most one time, then the while loop terminates when L_i becomes empty, or when we found the closest layer with t so we will break the loop.

Proof of correctness:
There are two claims that need to be proven for this algorithm to be correct:
The algorithm correctly computes the shortest path distances.
The algorithm counts all the shortest paths from s to t.

Proving it correctly computes the shortest path distance from s to all reachable nodes, including

t
Base case: initially the algorithm places s in layer L_0, and obviously the distance is zero, so the claim holds true.
Inductive hypothesis: assume that for all nodes $u \in L_i, dist(s, u) = i$

Assume we have a node $v \in L_{i+1}$, it is added to the layer when edge $u \in L_i$ found it, so that means that there is a path (s, v) whose length is $dist(s, u) + dist(u, v)$, and since we can only find v when $dist(u, v) = 1$, and $dist(s, u) = i$ by inductive hypothesis, $dist(s, v) = dist(s, u) + dist(u, v) = i + 1$
Thus, any node added to L_{i+1} is exactly $i + 1$ edges away from s. Hence, by induction, the algorithm correctly computes the shortest path distances from s to all reachable nodes.

Step 2: Next, we prove that the algorithm correctly counts the number of distinct shortest paths from s to t.
Base case (i=0):
Initially, the algorithm sets $paths[s] = 1$ because there is exactly one way to get from s to itself (the empty path). No other node has been visited yet, so $paths[u] = 0$ for all $u \neq s$
Inductive Hypothesis:
Assume that for all nodes $u \in L_i$, $paths[u]$ correctly counts the number of distinct shortest paths from s to u.

Assume we have a node $v \in L_{i+1}$, we know it is added to L_{i+1} when it is first visited as a neighbor to some $u \in L_i$. According to the algorithm, when v is first visited, the number of paths to v is initialized as $paths[v] = 0$. After that, every u that connects to v will add it's path count to v:

$$paths[v] \leftarrow paths[v] + paths[u]$$

Since all these paths are unique (they end on different nodes), it is okay to add them to the sum without considering overlap. Therefore, all distinct shortest paths to v that pass through a node in L_i is correctly counted. For nodes that are in a higher layer than L_i that connects to v, the path that connects to v cannot be the shortest path, and if there is one on a lower layer than the L_i shouldn't even have had it added because it would never be unvisited.

Thus, by induction, the value paths[t] at the end of the algorithm is the number of distinct shortest paths from s to t.

Conclusion: The algorithm is correct. It computes the shortest path distances using BFS and counts all distinct shortest paths from s to t by accumulating the path counts from nodes in previous layers.

Time complexity: Each node is visited once, and each edge is processed at most once, so the running time is linear with respect to the number of nodes and edges. Therefore, the time complexity is O(n+m).

3 What Kind of Bear are You?

a)

Define an edge as a same/different relationship between two photos.
Algorithm:

Algorithm 2 Pseudocode

Result: Return **Inconsistent**, **Underspecified**, or **Exact answer**

```
groups ← 0
for s in photos do
    if state(s) ≠ visited then
        type ← {s : 0}
        mark s "visited"; R ← {s}; layer L_0 ← {s}; i ← 0
        groups ← groups + 1
        while L_i not empty do
            L_{i+1} ← ∅
            for each u ∈ L_i do
                for each edge (u, v) do
                    if v is "unvisited" then
                        mark v "visited"
                        Add v to set R and to layer L_i + 1
                        if (u, v) =same species then
                            type[v] = type[u]
                        end
                        else
                            type[v] = 1 − type[u]
                        end
                    end
                    else
                        if ((u, v) = same ∧type[u] ≠ type[v] ∨ (u, v) = different ∧type[u] = type[v]) then
                            return "inconsistent"
                        end
                    end
                end
            end
            i ← i + 1
        end
    end
end
if groups = 1 then
    return "Exact answer"
end
else
    return "underspecified"
end
```

b)

Proof:

Termination:
The outer loop iterates over each p a finite number of times, and only initiates a search for a photo "node" which is not fully explored. Because there is a finite number of photos, it terminates.

The inner loop performs a BFS over each unvisited node s, and visits all nodes reachable from s via "same" or "different" edges. This is again bounded by the number of photos which is finite since it only explores each photo one time, and after all the photos are visited the search algorithm ends. Therefore, the inside loop will also terminate. For each u, the algorithm looks at all it's edges, and since that number is finite, it will alos terminate.
Therefore, the algorithm will terminate.

Correctness:
We model the problem as an undirected graph:
Vertices: Each photo is represented by a vertex.
Edges: Each edge represents a constraint between two photos:
A "same species" constraint forms an edge that ensures both vertices (photos) should belong to the same species group.
A "different species" constraint forms an edge that ensures the two vertices (photos) belong to different species groups.
First, proving correctness of graph construction:
The BFS ensures that all vertices in the same connected component are visited and labeled accordingly:
If two vertices are connected by a "same species" edge, they are assigned the same label (type[u] = type[v]).
If two vertices are connected by a "different species" edge, they are assigned opposite labels (type[u] = 1 - type[v]).
Thus, the BFS guarantees that all vertices connected by "same species" edges receive the same label, and all vertices connected by "different species" edges receive different labels.

Next, prove consistency check is correct:
For photos that were visited, we check for contradictions, and since initial labeling was correct, if a contradiction was found that means the pictures are truly inconsistent. If the pictures look are "same", but they are assigned different types, that is an inconsistency, and if the pictures look different but they are assigned the same type, then that is also an inconsistency, and just one inconsistency means the alogrithm should return "inconsistent" because it is now impossible to satisfy the constraints.

Next prove the final classification is correct:
After The main algorithm is finished, we check how many individual clusters there are. The only way to have an exact answer is if there is just one grouping. If there is only one connected component (i.e., all vertices are part of the same group and fully explored), the species assignments are complete and consistent.
However, when we have multiple disconnected components, we don't know whether the "0" in one segment's classification is the same as the "0" in the other grouping, so it is impossible to classify them into two distinct groups with full comfidence.

Why the Algorithm Handles All Cases Correctly:
Proving Detection of Inconsistency
Suppose the dataset is inconsistent. We need to show that our algorithm will correctly identify this inconsistency. By definition, inconsistency occurs when one or more images are assigned conflicting classifications, making it impossible to classify everything consistently.
To detect this, our algorithm visits each edge and ensures that no two edges assign contradictory classifications to the same image. Using the depth-first search (DFS) properties, we systematically explore each node and edge, checking for such conflicts. Since DFS guarantees that every node and edge in the connected component is examined, it is impossible for an inconsistency to go unnoticed. If any conflicting classification exists, it will be detected by the algorithm.
Proving Detection of Under-Specification
Next, consider the case where the data is under-specified. This means that there is not enough

information to classify every image, yet the data is consistent. Under-specification will occur only if there is an image that cannot be connected back to Image 1, since in this case, there would be no basis for determining that image's classification.
Since our algorithm's DFS starts at Image 1 and explores all reachable nodes, if any images remain unvisited by the end of the search, we know that these images are not connected to Image 1. This would indicate that the dataset is under-specified. Upon finding such unvisited nodes, the algorithm marks the data as potentially under-specified, provided no inconsistencies have been detected.

Proving Correctness in Exact Data

Finally, consider the scenario where the data is exact, meaning it is neither under-specified nor inconsistent. In this case, the graph is complete, with every node connected and no conflicting edges. We now prove that our algorithm will label each image correctly.

Base Case:

We begin by ensuring that Image 1 is labeled correctly. The intended label for Image 1 is A, and the algorithm indeed assigns this label at the start.

Inductive Hypothesis (IH):

Suppose that for an arbitrary k, our algorithm correctly labels the first k images. We now show that the $k+1$-th image will also be labeled correctly.

Inductive Step:

The algorithm assigns labels to nodes based on previously labeled nodes. Suppose, for the sake of contradiction, that the $k+1$-th image is labeled incorrectly. This would imply that an image already labeled has incorrectly assigned a label to the $k+1$-th image. However, by the inductive hypothesis, we know that all previous images were labeled correctly. Therefore, the $k+1$-th image must also be labeled correctly, as it receives its label from a correctly classified image.

Conclusion

Since the data is neither inconsistent nor under-specified, the algorithm will find no conflicts, and every image will have sufficient information to be labeled correctly. Given that k is arbitrary, we conclude that for an exact dataset, our algorithm will classify all images correctly.

c)

Runtime: BFS: p vertices and $s+d$ edges and we go over each at most once since after that it will be marked visited, so $O(p+s+d)$

4 A Package Deal

a)

There are two vertices for each package, so for package a, the vertex where a is included is a and where it is not included is $a\neg$.

At least one (a, b):
$a\neg \rightarrow b$
$b\neg \rightarrow a$
This is because if a is not installed b must be installed, and if b is not installed a must be, as at least one of them must be installed.

Dependent (a, b):
$a \rightarrow b$
$b\neg \rightarrow a\neg$
This is because if a is installed then b must be installed because a depends on b

Incompatible (a, b):
$a \rightarrow b\neg$
$b \rightarrow a\neg$
This is because they cannot both be installed so if one is installed the other must not be installed

b)

A directed path from x to y means that the inclusion or exclusion of the package associated with vertex x influences the inclusion or exclusion of the package associated with vertex y.
$a \rightarrow b$: a being included means b is as well.
$\neg a \rightarrow b$: a not being included means b is.
$a \rightarrow \neg b$: a being included means b is not.
$\neg a \rightarrow \neg b$: a not being included means b is also not.

c)

True.
Proof:
In an SCC, there is a path between any pair of vertices u and v, and vice versa. This bidirectional path structure ensures that any constraints between vertices can propagate throughout the component.
In a sense, if one of the nodes were included in the configuration, the fact every other node is reachable of an SCC would guarantee that the rest of the SCC has to be included, and so the only way for one of the nodes not to be included is if the entire graph is not included. Therefore, fixing one vertex fixes the rest and thus the SCC must be consistent.
Suppose that wasn't the case, and that there was a node u in the SCC that was not included while a different node v was. v would have a directed path to u, so v cannot be included while u is not.

d)

Algorithm 3 Pseudocode

```
Result: Return Impossible or Configuration
for every unvisited Node u do
    Mark u visited
     Run Tarjan's from u and mark connecting nodes visited
     for each package p ∈ SCC do
         if contains both include and exclude of p then
             Return Impossible.
         end
     end
    OD ← All nodes that have out degree zero.
     used ← []
     while used not filled up do
         SCC ← first in OD for each node p ∈ SCC do
             if p = "include package" then
                 used[p] ← included
             end
             else
                 used[p] ← excluded
             end
         end
         for each parent p of SCC do
             p.out ← p.out − 1
              if p.out = 0 then
                 add p to OD
             end
         end
    end
end
return used
```

e

Representation of Constraints
First, every type of constraint is correctly represented in the graph, where edges between nodes precisely capture the problem's requirements. For example, "at least one" constraints, dependencies, and incompatibilities are all reflected by directed edges between nodes in a way that enforces the relationships between packages.

Correctness of SCC Evaluation
Tarjan's algorithm accurately identifies all Strongly Connected Components (SCCs) in the graph. Since an SCC is defined as a group of nodes where every node can reach every other node, the states of the nodes in the SCC must be either all true or all false. This ensures internal consistency within the SCC. If there were contradictory states within an SCC (e.g., both a node a and its negation $\neg a$), then the problem would be unsolvable because you would have to both include and exclude the same package—an impossibility.

Proof That the Graph Becomes a DAG
After identifying the SCCs, we collapse each SCC into a single node, creating a **meta graph**. This meta graph is guaranteed to be acyclic (a Directed Acyclic Graph, or DAG) because if there were any cycles in the original graph, they would have been captured as SCCs. By collapsing these cycles into single nodes, the resulting meta graph contains no cycles.

Since the graph is now a finite DAG, we can apply basic properties of DAGs: there must be at least one node with no outgoing edges (a "sink" node), because if every node had an outgoing

edge, that would imply the existence of a cycle, which contradicts the definition of a DAG.

Processing SCCs

Consider an SCC with no outgoing edges in the meta graph. This SCC has no dependencies outside of itself, meaning that the states of its nodes can be determined independently of other SCCs. The packages in this SCC can all be consistently set to either true (included) or false (excluded) without risking any external contradictions. Internal consistency has already been guaranteed through the SCC evaluation.

Symmetry and Dependencies Between SCCs

For each constraint in the graph, if there is an edge from u to v, there is a corresponding edge from $\neg v$ to $\neg u$ (due to the nature of the "at least one" and dependency constraints). This creates a symmetry in the formation of SCCs. Each SCC has a counterpart SCC with the opposite nodes, and the direction of the edges between them is reversed. For example, a cycle $a \to b \to c \to a$ becomes $\neg c \to \neg b \to \neg a \to \neg c$.

Since the meta graph is acyclic, only one side of this symmetrical relationship can depend on the other, meaning one SCC can influence another, but not vice versa. This ensures that we can start by selecting the side of the graph that is not dependent on the other and assign states to the packages consistently. This property holds true even if the meta graph forms a forest (i.e., multiple disconnected DAGs), because if there were any inconsistencies between SCCs, they would have already been detected within the SCCs themselves due to the existence of cycles.

Handling Symmetry in the Algorithm

Even though the graph is symmetrical, the algorithm does not arbitrarily choose one side of the graph. Instead, it processes the SCCs in reverse topological order. If it encounters an SCC on the dependent side first, it cannot fully process that side right away. When it encounters a node (call it v) that connects to the opposite side of the symmetry, it must switch to processing the other side.

The reverse topological order ensures that the highest-ranked node on one side (according to the reverse topological sort) corresponds to the lowest-ranked node on the other side, due to the symmetry of the graph. Therefore, when the algorithm switches to the opposite side, it processes nodes that have not yet been visited on that side. This guarantees that the algorithm can resolve the dependencies between the two sides without creating inconsistencies, as it continues alternating between sides until all packages have been examined.

Conclusion

Thus, as long as there are no internal inconsistencies within any SCC, the algorithm accurately finds a solution that satisfies all constraints. By processing the SCCs in a consistent order, the algorithm ensures that no external inconsistencies arise between SCCs, and the configuration of packages is valid.

f

Time complexity is $O(p + n)$, with p being the relationships and n being the packages.
This is because we add $2p$ edges and in traversal we search each package and edge once at most.
Tarjan's algorithm is also $O(n + p)$