1 Introduction

A table of frequently used symbols:

Symbol	Representation
g	the input graph
v	the set of nodes in g
e	the set of edges in g
n	a node in v
s	the start node in v
t	the target node in v
f	the number of atoms to conserve (flow to move)
G	the constructed metagraph
V	the set of metanodes in G
E	the set of metaedges in G
N	a metanode in V of the form $\{n_i : x_i\}$ where $n_i \in v$ and x_i is the number of atoms at n_i

Each $N \in V$ is composed of i many nodes from v. A metanode takes the form $\{n_i : x_i\}$ where $n_i \in v$ and x_i is the number of atoms at n_i. The $\sum_0^i x_i = k$ at any given N ensuring that all atoms are accounted for at each metanode.

The terms *state* and *metanode* are used interchangeably. Each metanode is simply a representation of where flow is distributed. The state $\{s : f\}$ indicates that there is f amount of flow at node s. Similarly, $\{a : 3, b : 2\}$ indicates that there are 3 atoms at node a and 2 atoms at node b.

The goal is to move all f atoms (flow) from node s to node t. In other words, we seek a path from metanode $\{s : f\}$ to metanode $\{t : f\}$ in the metagraph.

2 Overview

Algorithm 1: Metagraph Generation

```
Input: g, s, t, f
Output: G
S = {s : f} /* Create the start state */
V ← V ∪ S /* Add start state to the metagraph */
stack.push(S);
while |stack| ≠ 0 do
    current ← stack.pop();
    partials ← initialize empty list;
    for each inner node, n, in current do
        innerPartials ← list of all unique ways to move the flow residing at n;
        partials.append(innerPartials);
    nbrs ← all complete states, where each complete state is formed by merging together one partial
      state from each list in partials;
    for nbr ∈ nbrs do
        if nbr is valid and nbr ∉ deadset then
            V ← V ∪ nbr;
            E ← E ∪ (current, nbr);
            nbrCount ← nbrCount + 1;
            stack.push(nbr);
return G;
```

3 Algorithms

Algorithm 2: ConstructMetaGraph

Input: $g = (v, e)$: the input graph, $\{s, t\} \in v$: the start and target compounds, k: flow/number of atoms to conserve

Output: $G = (V, E)$: the meta-graph

$MG \leftarrow (MV = \emptyset, ME = \emptyset)$ `/* initialize new metagraph */`
$stack \leftarrow$ initialize empty stack;
$start \leftarrow \{s : k\}$ `/* Create a metanode with all` k `atoms at the start compound */`
$V \leftarrow V \cup start$ `/* Add the start state to the set of metanodes */`
$stack.push(start)$ `/* Add the start state to the stack to find its neighbors */`
$G \leftarrow$ **PopulateMetaGraph**$(G, stack, target)$ `/* Find neighboring metanodes to build` G `*/`
return G

Algorithm 3: PopulateMetaGraph

Input: $G = (V, E)$: the metagraph, *stack*: stack of metanodes that need to be explored, *target*: the metanode state with all k atoms at node t

Output: $G = (V, E)$: metagraph with any newly found metanodes added in

while $|stack| > 0$ **do**
    $current \leftarrow stack.pop()$ `/* Pop off a metanode to explore */`
    if $current = target$ **then**
        continue `/* If we've reached the target, no need to find nbrs */`
    else
        $nbrCount \leftarrow$ **IterativeFindMetaNbrs**$(current)$;
        if $nbrCount = 0$ ***and*** $current \neq target$ **then**
            $G \leftarrow$ **Prune**$(current)$ `/* This metanode is a terminus, so remove it from the metagraph */`
return G

Algorithm 4: IterativeFindMetaNbrs

Input: *current*: the metanode to find nbrs for, $G = (V, E)$: the metagraph so far
Output: $G = (V, E)$: metagraph with any newly found metanodes added in

```
partialStates ← initialize new list;
for n ∈ N do
    partialStates.append(RecursiveNbrSearch(n, flow(n), nbrs(n)));
metanbrs ← generateCompleteStates(partialStates);
for nbr ∈ metanbrs do
    if isValid(nbr) and nbr ∉ deadset then
        V ← V ∪ nbr;
        E ← E ∪ (current, nbr);
        nbrCount ← nbrCount + 1;
        stack.push(nbr);
return G
```

Algorithm 5: RecursiveNbrSearch

Input: n: the node we're trying to move flow away from, *flow*: the amount of flow residing at n that needs to be moved, *nbrs*: the unexplored neighbors of n
Output: *partialStates*: list of partial states (each state represents a unique move of the flow at n to its nbrs)

```
states ← initialize new list;
if |nbrs| ¿ 1 then
    nbr ← nbrs[0];
    for i = 0; i ≤ min(flow, capacity(parent, nbr)); i + +) do
        partialState ← initialize mapping of nodes (string) to flow (number);
        if i ≠ 0 then
            partialState.put(nbr, i);
        remaining ← recursiveNbrSearch(n, flow-i, nbrs-nbr)
        for state ∈ remaining do
            states.add(mergePartialStates(parialState, state))
if |nbrs| = 0 and flow ¿ 0 and current = target then
    /* If there is flow remaining, but the current node is the target node, the flow
       can remain stationary                                                     */
    state.put(current, flow); states.add(state);
else
    /* If there are no more nbrs and flow remains, no more valid states can be added,
       so return                                                                 */
return states
```