Report on Automata Network "Process Hitting" and Local Causality

Karolina Zoń

Université Paris Sud XI

1. Introduction

In Process Hitting, which is used to compute possible scenarios of actions in graphs, we base on Finite-State Machine model of computation. With several automata and informations about them we can predict if some specific state in any specific automaton may be reached. We can also find out if some event (that means sequence of actions in group of automata) can happen after other (what may be also named as if some sequences may be linked).

At the beginning the State Graph (SG) has to be built, to obtain the complete view on given model of automata. Each node of this graph is a set of current states of all automata, edges of this graph indicate possible transitions, assuming that only one automaton can change its state at one time. The graph is built on base of sequence of possible actions, that can occur in automata model, which is previously given. With this graph we can check, if with given initial states we can reach desired final states.

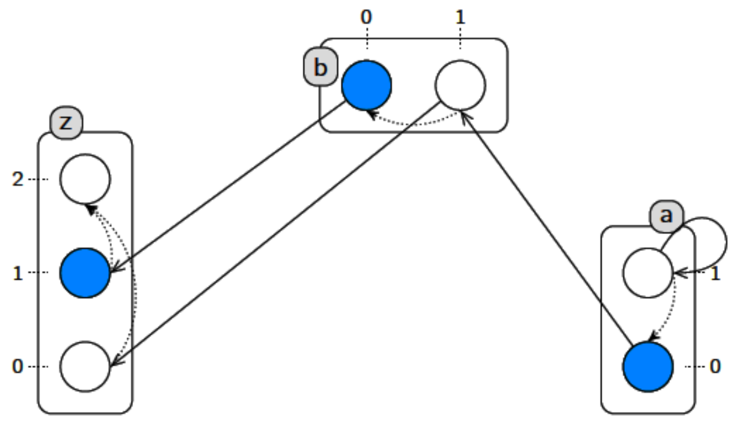
Notation:

a, b, z: automata

a_0, b_0, z_1: processes

{**a_0, b_0, z_1**}: set of automata; {**0, 0, 1**}: set of states;

$a_0 \rightarrow b_1 \urcorner b_0$: action (**$a_0$** makes **$b_1$** transit into **$b_0$**)



Possible scenario:

We want to reach $\mathbf{z_2}$. To do so, the automata **b** must be in state 0. It is, so it makes $\mathbf{z_1}$ transit into $\mathbf{z_2}$. If automata **b** would be in state 1, we would need first automata **a** to transit **b** from $\mathbf{b_1}$ to $\mathbf{b_0}$.

Another possible scenario:

We want to reach $\mathbf{z_0}$. But as it's visible on the graph, it's not possible to reach $\mathbf{z_0}$ from current configuration. That means that there is no solution for reaching $\mathbf{z_0}$.

Another way to analyze automata model is Local Causality, may be said “reversed state graph searching”. The objective of this method is to look for final states first and after finding them collect all ways to reach it. This kind of searching is significantly faster, as we don’t need to compute full graph, but just do it partially – only the ways of our interest are taken into account. Then with given initial states we can search through all ways to find out if there is a possibility to reach this particular final state.

2. Computing State Graph in Java

Algorithm:

Notation:

action: $\mathbf{a}_i \rightarrow \mathbf{b}_j \ulcorner \mathbf{b}_k$ where **a, b**: automata; **i, j, k**: states

edge: <**states**> → {<$\mathbf{states}_1$>, <$\mathbf{states}_2$>, … , <$\mathbf{states}_N$>}

```
List<states> stateList;
HashMap<states, List<states>> edges;

function addNewEdge(states x, states y) {
    edges[x] <- edges[x] + y
}

function constructGraph(actions) {
    for each states s in statesList
        for each action n in actions
            if (s[a] = n.i and s[b] = n.j) {
                newStates = s;
                newStates[b] <- k;
                if (statesList.contains(newStates) {
                    if (edges.containsKey(s)) {
                        if (!edges[s].contains(newStates) {
```

```
                    addNewEdge(s, newStates);
                }
            }
            else {
                edges.addKey(s);
                addNewEdge(s, newStates);
            }
        }
        else {
            statesList <- statesList + newStates;
            if (!edges.contains(s)) {
                edges.addKey(s);
            }
            addNewEdge(s, newStates);
        }
    }
}
```

The program that computes State Graph and search for a possibility to reach the final states takes as an argument sequence of possible actions, initial states and mentioned final states. Based on sequence of actions we can get information on how many automata we've got and how many states those automata contain. Then with all those informations we can start building the graph.

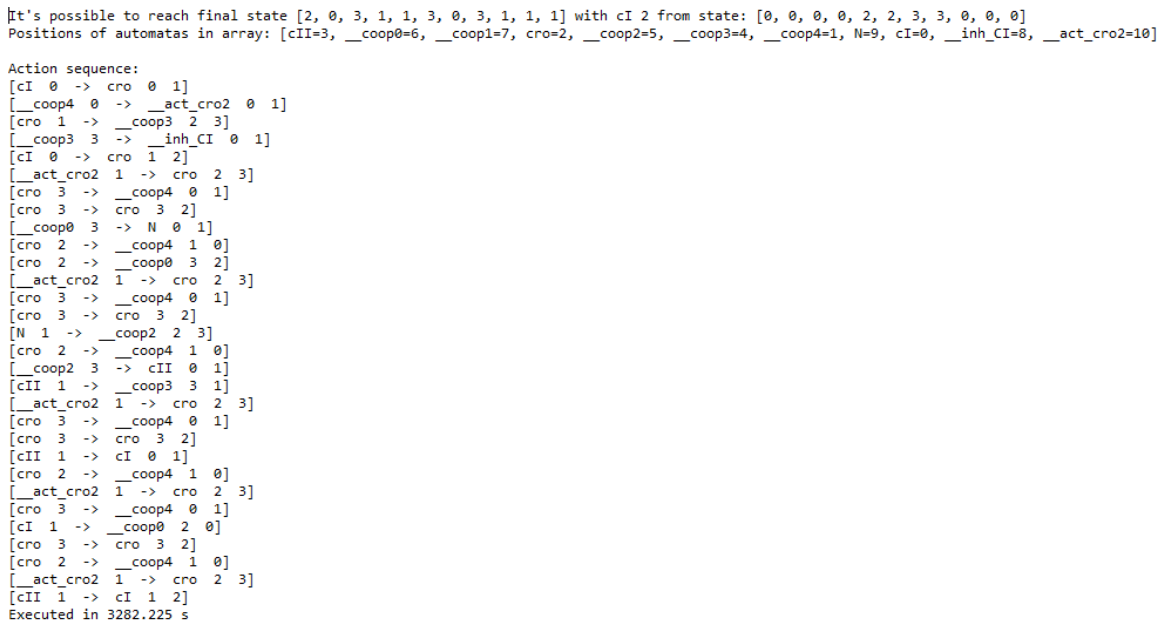
```
a 0 -> b 1 0
a 1 -> a 1 0
b 1 -> z 0 2
b 0 -> z 1 2
```

Example of possible actions list

First we put initial states as current states. With them we are searching through all possible actions to find the matching ones. So we can create transitions (graph edges) from them to another ones. We have a list of all states, that at the beginning contains only initial states, but during searching we are adding every new developed state (if it's not on the list yet), so after analyzing initial states, we can do the same for others. With others we have to always check, if given edge doesn't exist yet, to avoid redundancy. After completing the algorithm we have developed whole State Graph.

While having a graph, it's time to search for the final states we want, so it's necessary to use a particular algorithm, in this case it's depth-first search. The method for searching is getting the current state and it goes down through all until it will find the wanted final states.

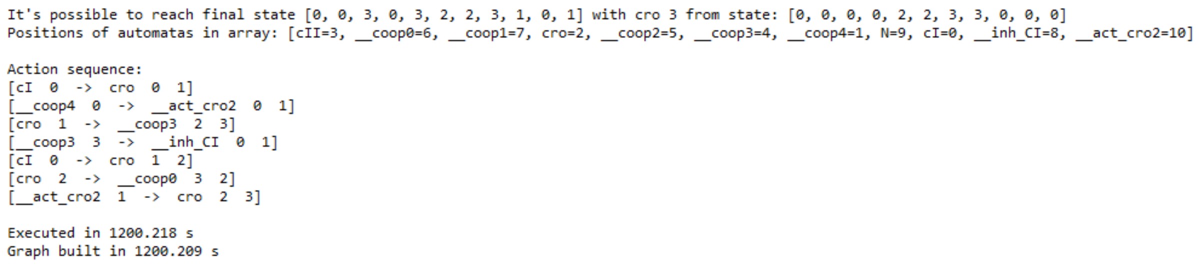
Final result depends on if we have found requested final states or not. If we have, we will get the information of which state it is (may be useful if wanted final states were just partial) and sequence of actions that have to happen to reach it. The following screen is the example of that situation.



```
It's possible to reach final state [2, 0, 3, 1, 1, 3, 0, 3, 1, 1, 1] with cI 2 from state: [0, 0, 0, 0, 2, 2, 3, 3, 0, 0, 0]
Positions of automatas in array: [cII=3, __coop0=6, __coop1=7, cro=2, __coop2=5, __coop3=4, __coop4=1, N=9, cI=0, __inh_CI=8, __act_cro2=10]

Action sequence:
[cI  0  ->  cro  0  1]
[__coop4  0  ->  __act_cro2  0  1]
[cro  1  ->  __coop3  2  3]
[__coop3  3  ->  __inh_CI  0  1]
[cI  0  ->  cro  1  2]
[__act_cro2  1  ->  cro  2  3]
[cro  3  ->  __coop4  0  1]
[cro  3  ->  cro  3  2]
[__coop0  3  ->  N  0  1]
[cro  2  ->  __coop4  1  0]
[cro  2  ->  __coop0  3  2]
[__act_cro2  1  ->  cro  2  3]
[cro  3  ->  __coop4  0  1]
[cro  3  ->  cro  3  2]
[N  1  ->  __coop2  2  3]
[cro  2  ->  __coop4  1  0]
[__coop2  3  ->  cII  0  1]
[cII  1  ->  __coop3  3  1]
[__act_cro2  1  ->  cro  2  3]
[cro  3  ->  __coop4  0  1]
[cro  3  ->  cro  3  2]
[cII  1  ->  cI  0  1]
[cro  2  ->  __coop4  1  0]
[__act_cro2  1  ->  cro  2  3]
[cro  3  ->  __coop4  0  1]
[cI  1  ->  __coop0  2  0]
[cro  3  ->  cro  3  2]
[cro  2  ->  __coop4  1  0]
[__act_cro2  1  ->  cro  2  3]
[cII  1  ->  cI  1  2]
Executed in 3282.225 s
```

Otherwise, we will get information, that it's not possible to reach given final states.

This solution gives the right answer, but it needs a lot of time to compute the graph. The example above is the solution to a graph built with 97 possible actions on 11 states. As it's displayed at the end, program needed almost an hour to compute a graph to this example and solve it. On another example below, we can see the total time of execution of program and the time of just building a graph. We can see that it's the vast majority of the time used.



```
It's possible to reach final state [0, 0, 3, 0, 3, 2, 2, 3, 1, 0, 1] with cro 3 from state: [0, 0, 0, 0, 2, 2, 3, 3, 0, 0, 0]
Positions of automatas in array: [cII=3, __coop0=6, __coop1=7, cro=2, __coop2=5, __coop3=4, __coop4=1, N=9, cI=0, __inh_CI=8, __act_cro2=10]

Action sequence:
[cI  0  ->  cro  0  1]
[__coop4  0  ->  __act_cro2  0  1]
[cro  1  ->  __coop3  2  3]
[__coop3  3  ->  __inh_CI  0  1]
[cI  0  ->  cro  1  2]
[cro  2  ->  __coop0  3  2]
[__act_cro2  1  ->  cro  2  3]

Executed in 1200.218 s
Graph built in 1200.209 s
```

If there are a lot of automata and states or/and a lot of possible actions, the graph becomes very large. Wide part of it may be computed for nothing, because then it will not be used as not interesting to the case. The solution for that problem is changing the method of Graph State to the Graph of Local Causality, featured in next section.

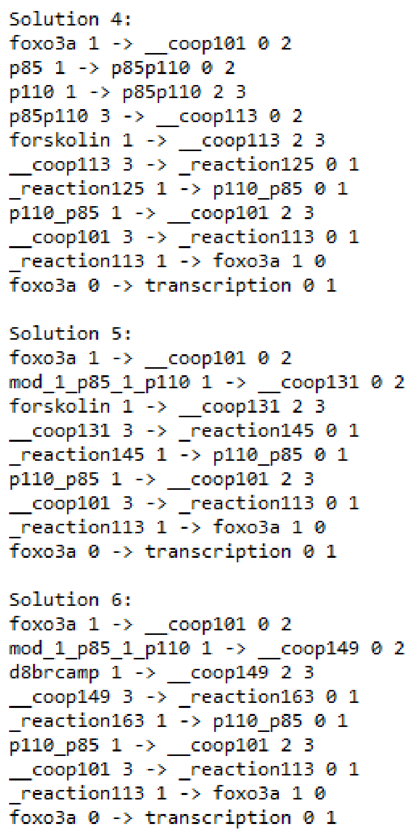
3. Algorithm on “simple Graph of Local Causality”

The point of Local Causality method, as mentioned in the introduction, is to find a way to final states starting from final states themselves. We look for states that are needed to reach the final states, and do it again for those states. We repeat it until we find all ways to reach wanted states. After that we can look through all ways to find ones that match given initial states. This part, the algorithm on simple Graph of Local Causality was the object of my interest.

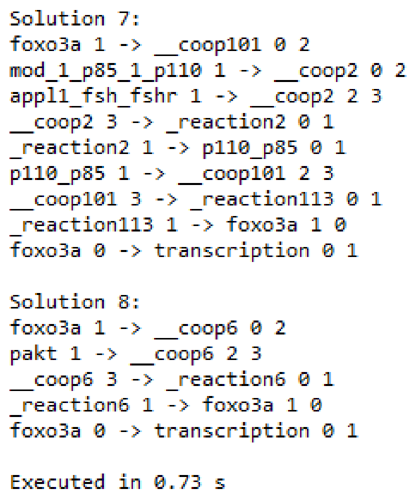
As input we get initial states of the graph, final state we search for, list of possible actions and this time additionally list of ways to reach wanted final state. This list must be analyzed firstly in order to obtain a Local causallity graph. For each final state, here called *process*, we have an *objective* specifying how the process is changed. If it's possible to change it, next we have a *solution*, that specifies which process (processes) cause the change of the state. Additionally there may be objective of the objective, so called *redirection*, that indicates another way to reach given state faster than with specified objective. Then the graph is analyzed to check if those given actions may be executed.

Single analysis returns the sequence of actions performed to obtain the specified state. As a graph may be of any size, it's possible that during execution of it, it will call itself recursively so a long sequence can be returned or multiply sequences, if there will be more solutions. Example of the result (reaching state *transcription 1* from *transcription 0*):

```
Solution 1:
foxo3a 1 -> __coop101 0 2
p85 1 -> p85p110 0 2
p110 1 -> p85p110 2 3
p85p110 3 -> __coop105 0 2
d8brcamp 1 -> __coop105 2 3
__coop105 3 -> _reaction117 0 1
_reaction117 1 -> p110_p85 0 1
p110_p85 1 -> __coop101 2 3
__coop101 3 -> _reaction113 0 1
_reaction113 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1

Solution 2:
foxo3a 1 -> __coop101 0 2
mod_1_p85_1_p110 1 -> __coop106 0 2
d8cpt2mecamp 1 -> __coop106 2 3
__coop106 3 -> _reaction118 0 1
_reaction118 1 -> p110_p85 0 1
p110_p85 1 -> __coop101 2 3
__coop101 3 -> _reaction113 0 1
_reaction113 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1

Solution 3:
foxo3a 1 -> __coop101 0 2
p85 1 -> p85p110 0 2
p110 1 -> p85p110 2 3
p85p110 3 -> __coop108 0 2
d8cpt2mecamp 1 -> __coop108 2 3
__coop108 3 -> _reaction120 0 1
_reaction120 1 -> p110_p85 0 1
p110_p85 1 -> __coop101 2 3
__coop101 3 -> _reaction113 0 1
_reaction113 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1
```



```
Solution 4:
foxo3a 1 -> __coop101 0 2
p85 1 -> p85p110 0 2
p110 1 -> p85p110 2 3
p85p110 3 -> __coop113 0 2
forskolin 1 -> __coop113 2 3
__coop113 3 -> _reaction125 0 1
_reaction125 1 -> p110_p85 0 1
p110_p85 1 -> __coop101 2 3
__coop101 3 -> _reaction113 0 1
_reaction113 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1

Solution 5:
foxo3a 1 -> __coop101 0 2
mod_1_p85_1_p110 1 -> __coop131 0 2
forskolin 1 -> __coop131 2 3
__coop131 3 -> _reaction145 0 1
_reaction145 1 -> p110_p85 0 1
p110_p85 1 -> __coop101 2 3
__coop101 3 -> _reaction113 0 1
_reaction113 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1

Solution 6:
foxo3a 1 -> __coop101 0 2
mod_1_p85_1_p110 1 -> __coop149 0 2
d8brcamp 1 -> __coop149 2 3
__coop149 3 -> _reaction163 0 1
_reaction163 1 -> p110_p85 0 1
p110_p85 1 -> __coop101 2 3
__coop101 3 -> _reaction113 0 1
_reaction113 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1
```



```
Solution 7:
foxo3a 1 -> __coop101 0 2
mod_1_p85_1_p110 1 -> __coop2 0 2
appl1_fsh_fshr 1 -> __coop2 2 3
__coop2 3 -> _reaction2 0 1
_reaction2 1 -> p110_p85 0 1
p110_p85 1 -> __coop101 2 3
__coop101 3 -> _reaction113 0 1
_reaction113 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1

Solution 8:
foxo3a 1 -> __coop6 0 2
pakt 1 -> __coop6 2 3
__coop6 3 -> _reaction6 0 1
_reaction6 1 -> foxo3a 1 0
foxo3a 0 -> transcription 0 1

Executed in 0.73 s
```

As we can see, using this method even big examples are being computed in a significantly shorter amount of time. In this example list of 2421 possible actions on 70 states have been used.

Function takes as input a state to reach, current states and optionally the objective, if we want to reach it with some concrete one (it happens, when there is a redirection of another objective, that means reaching a partial result for a redirection by another objective). At the beginning it checks if the process we are looking for is not a current one, so there would be no point of search. Then search through the graph of local causality if the process contains the objective, otherwise, no result is returned. If there is an objective the algorithm checks if it has an redirection, so it can apply the function for the state needed by redirection and then continue with the changed objective (possible redirection: objective is “*a* 0 → *a* 2”, where *a* is a whichever automaton and the redirection is “*a* 1 → *a* 2”, so we first we apply the same algorithm for obtaining the process *a* 1 from *a* 0, and then we are solving a problem for a redirection, *a* 2 from *a* 1):

Next step is searching for a solution/solutions for the given objective and for all of them (as they can be several) searching for processes to solve them. As processes may be many or they may change the state in more than one step (for example: automaton *a* changing automaton *b* from state 0 to state 2 by changing it firstly to 1 and then to 2) there must be applied the function of *ordering* that returns the object of a special class ProcessOrder in which there is a list of processes to apply in order of applying and list of actions performed. With this object given, this function will be used for all processes in every solution. It must be taken into account, that if there was a redirection before, we have already a list of some actions that had to be performed before, so the first step is to adding them (if they exist) to the final list of actions. If there are several processes, we apply the algorithm for each of them and then succesively complete the final list.

Notation:

process $\mathbf{a}_\mathbf{i}$ **:** automaton **a** with state **i**

objective $\mathbf{a}_\mathbf{i} \rightarrow \mathbf{b}_\mathbf{j} \ulcorner \mathbf{b}_\mathbf{k}$ **:** process $\mathbf{a}_\mathbf{i}$ changes state of automaton **b** from **j** to **k**

solution **s :** {$\mathbf{a}_\mathbf{i}$**,** $\mathbf{b}_\mathbf{j}$**,** …} to execute objective processes $\mathbf{a}_\mathbf{i}$**,** $\mathbf{b}_\mathbf{j}$**,** … are needed

Algorithm:

```
HashMap<String, List<String[]>> processes;
HashMap<String, List<Integer>> objectives;
HashMap<String, String[]> objectives_redirected;
HashMap<Integer, List<String[]>> solutions

Class Path {int[] currentStates, List<action> listOfActions};

path join(path p, path q) {
  path r;
       r.x = q.x
  r.l = p.l + q.l;
  return r;
}

set(Paths) solve(currentStates c, process p, objectiveForced) {
       if p.i != c[p.a] {
              if objectives.contains(p) {
                     if objectiveForced = null {
                            for all objectives o in objectives
```

```
                    if o.i = c[a]
                        objective ← o;
                        break;
            }
            else
                objective ← objectiveForced
                if objectives_redirected.contains(objective) {
                  objRedir ← objectives_redirected(objective);
                  objTemp ← objective[ai] + "→ " + objRedir[ai];
                  preResult ← solve(c, p, objTemp);
                  objective ← objRedir;
// preResult = {p1, p2, ...};
                } else {
                   preResult = {(x=c , l=[])};
            }
            if solutions.contains(objective) {
                for I in {0,..,#solutions-1} {
                    if processes.contains(s) {
                        result <- preResult;
                        processesOrderedList ← ordering(processes[solution]);
                        for each path p in preResult {
                            for each process q in processOrderedList {
                                resultP = solve(p.x, q, null);
                                for r in resultP
                                    result <- join(results, r);
                                    result.actions <- result.actions +
                                                    action_performed_by_this_process;
                                    update currentStates of result;
                            }
                        }
                    }
                    else continue; //this solution doesn't have any processes, error
                }
                return result;
            }
            return null; // objective doesn't have a solution, error
        }
        return null; //process doesn't have an objective, error
    }
    else return solution with empty list of actions;
}
```

The latest big improvement was changing the method of ordering processes, from bulding a whole state graph to find a way to wanted state using specified processes, to building just a graph of a particular (changed) automata, regardless of any partial changes (of other automata). As the number of computations is significantly decreased, the algorithm may be applied for big result and return the result in seconds instead of hours, as it was before).

4. Algorithm on "General Case of Local Causality"

Notation:

process $\mathbf{a_i}$ **:** automaton **a** with state **i**

objective $\mathbf{a_i \rightarrow b_j \ulcorner b_k}$ **:** process $\mathbf{a_i}$ changes state of automaton **b** from **j** to **k**

solution **s :** {$\mathbf{a_i}$**,** $\mathbf{b_j}$**,** …} to execute objective processes $\mathbf{a_i}$**,** $\mathbf{b_j}$**,** … are needed

Algorithm:

```
HashMap<solution, List<process>> processes;
HashMap<process, List<objective>> objectives;
HashMap<objective, List<objective>> objectives_redirected;
HashMap<objective, List<solution>> solutions
Stack<process> processesStack;
List<objective> redirections;
List<Paths> redirectionPaths;

Class Path {int[] currentStates, List<action> listOfActions};

function cycleDetect(process x) {
    for each process p in processesStack {
        if (x == p)
            return true;
    }
    return false;
}

function getObjective(process x, objectiveForced f, currentStates c) {
    if (f != null) {
        return f;
    }
    else if (objectives.containsKey(x)) {
        for each objective o in objectives[x] {
            if (o.i == c[a])
                return o;
        }
    }
    else
        return null;
}
```

```
function redirection(process x, path p, boolean redirectedBefore){
    for each redirection y in redirections
        if (x[a] == y[r][a] && x[i] == y[r][i]) {
            redirections <- redirections - y;
            processToFind <- {y[r][a], y[r][i]};
            processStack.push(processToFind);
            List<Paths> tmpResult = {};
            tmpResult <- solve(p.currentStates, processToFind, y[r], true);
            for each path r in tmpResult {
                if (!redirectedBefore)
                    redirectionPaths <- redirectionPaths + r;
            }
        }
}

function saveRedirection(objective x){
    if (objectives_redirected.containsKey(x))
        for each objective o in objectives_redirected[x]
            redirections <- redirections + {x, o};
}

function checkRedirections(process x) {
    List<Paths> paths = {};
    if (redirectionPaths != null) {
        for all paths p in redirectionPaths {
            lastAction <- p.listOfActions[lastElement];
            if (lastAction[b] == x[a] && lastAction[k] == x[i])
            paths <- paths + p;
            redirectionPaths <- redirectionPaths - p;
        }
    }
    return paths;
}

function solve(currentStates c, process p, objectiveForced f, boolean redirection) {
    if (c[p.a] != p.i) {
        if ((objective = getObjective(p, f, c)) != null) {
            saveRedirection();
            if (solutions.containsKey(objective)) {
                for each solutions s in solutions[objective] {
                    if (processes.containsKey(s)) {
                        processesOrderedList <- ordering(processes[s]);
                        List<Paths> listOfResult = {};
                        listOfResult <- listOfResult + Path(c, emptyList);
                        error <- false; cycle <- false;
                        for each process x in processesOrderedList {
```

```
                        if (cycleDetect(x) == true) {
                            cycle <- true;
                            break;
                        }
                        processStack.push(x);
                        for each result r in listOfResult {
                            List<Paths> tmpResult = {};
                            tmpResult <- solve(r.currentStates, x, null, redirection);
                            if (tmpResult == null) {
                                error <- true;
                                break;
                            }
                            tmpResult <- tmpResult + checkRedirections(x);
                            additionalActionsList <- actions performed by x;
                            for all result tr in tmpResult {
                                if (additionalActionsList[0].i != tr.currentStates[a]) {
                                    error <- true;
                                    break;
                                }
                                listOfResults <- listOfResults + tr;
                    listOfResults[tr].currentStates[additionalActionsList[lastElement][b]] =
                                additionalActionsList[lastElement][k];
                                listOfResults[tr].listOfActions <-
                                listOfResult[tr].listOfActions + additionalActionsList;
                                redirection(x, listOfResults[tr], redirection);
                            }
                        }
                    }
                    if (!cycle && !error)
                        result <- result + listOfResults;
                    else
                        cycle = error <- false;
                }
                else
                    continue;
                if (oneSolution == true)
                    break;
            }
        }
        else {
            processStack.pop();
            return null;
        }
    }
    else {
        processStack.pop();
        return null;
```

```
            }
        }
        else {
            processStack.pop();
            result <- result + Path(c, emptyList);
            return result;
        }
}
```

This algorithm provides a method to compute more complicated cases than the algorithm before. It's possible to find a way through the graph with cycles, multiple redirections or some “dead ends” (when it's coming to solve an objective with no solution, for example).

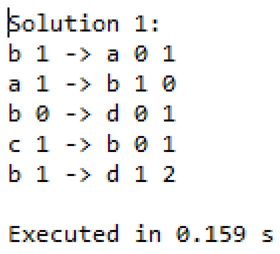
Cycles handling: To avoid getting into a cycle (what leads to the infinite loop in the program) in this algoritm all processes of ongoing analysis (during the recursion) are stored in the global stack. Stack is used before resolving subsequent process to check, if it's not already part of the path that is actually analyzed. In some cases this method may be unreliable, as at some point the same process may be solved in other way, here comes the substitute way of storing objectives instead of processes. Both may be used in the most of cases.

“Dead ends”: when the process without objective or objective without a solution is trying to be solved, the algorithm “catches” that and removing the ongoing path from prospective paths. The function returns “null” value for a case like this, then the previous recurse of function gets the error and stop processing the wrong path and pass on to the next solution, if there is one.

Redirections: First change of redirections treatment is that it's no longer solved just after detecting that the redirection exists for a particular objective. Anytime the objective is analyzed, function *saveRedirections* checks, if there is a redirection to the given objective. If there is, it is stored in the global redirections list *redirections*. The next step is checking, for each analyzed process (with an actual, solved path to reach it), if it is useful for any of stored redirections, by the *redirection* function. If the answer is positive, the redirected objective is being solved, out of ongoing processing and if it's solvable, the result (Path) is added to the global list of *redirectionPaths.* The point of this step is to avoid situation, when the redirection exists for objective different than the objective to the process requested by user, as in this situation returned Path will only lead to some process that was needed during the executing of a program. To solve this problem, there is a function *checkRedirections* whose job is to check if there is a path in *redirectionPaths* that leads to the process different than requested. This function is being applied after each “come back” from the recursion. If the process that it's led to is the same as the process just solved, function will join the path to the already existing paths to solve given process. After that the path is removed from the *redirectionPaths* as well as each treated redirection is removed from *redirectios* global list after solving, to avoid the cycle in redirections.

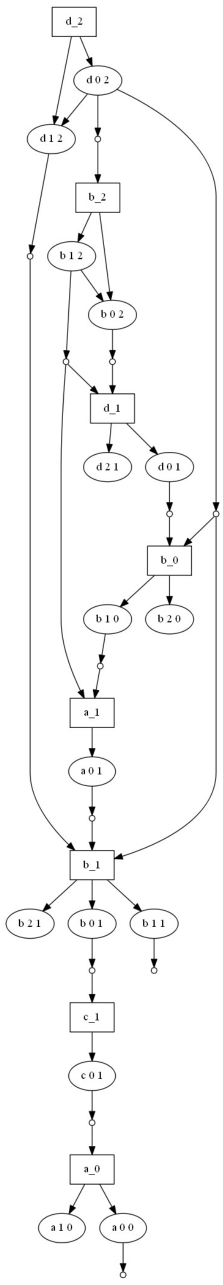
The time of executing remains approximated to the time of executing program of a simple case, but now many more situation may be solved.

Example of execution of the program:



```
Solution 1:
b 1 -> a 0 1
a 1 -> b 1 0
b 0 -> d 0 1
c 1 -> b 0 1
b 1 -> d 1 2

Executed in 0.159 s
```

This is the solution to the graph that can't be solved with the previous algorithm, because of its complexity:



5. Conclusion

The best way so far to find out existence of the way to reach defined states from given initial states is using the general case of Local Causality. There are still possible configuration, that this algorithm is not able to solve, but the improvement from the State Graph way is meaningly observable. Perhaphs, to solve even more complicated cases, entirely different algorithm will be necessary, but for that kind of examples this one is satisfying.