令和 5 年度

修士学位論文

Erlang における最適な監視ツリーの自動生成法

Automatic Generation of an Optimal Supervision Tree in Erlang

1265101　　佐々木 勝一

指導教員　　高田 喜朗

2024 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

Erlang における最適な監視ツリーの自動生成法

佐々木 勝一

Erlang とは並行指向の関数型プログラミング言語であり，OS スレッドより軽い軽量プロセスを標準で備えている．Erlang の大きな特徴として，想定外のエラーが発生した軽量プロセスはそのまま終了させて再起動する，という考え方のエラーハンドリングが主流な点が挙げられる．Erlang は，この考え方を実現する仕組みとして監視ツリーを提供している．監視ツリーの設定により，いくつかの制限はあるが，プロセス終了時に再起動されるプロセスらを制御できる．監視ツリーを作成する際の要求として，再起動中のプロセスはプログラムを実行できないから，一度に再起動されるプロセスの数を最小限にしたい．一方で，依存するプロセスが終了したプロセスは処理の継続が困難と判断できるから，終了したプロセスに依存する全てのプロセスは再起動したい．監視ツリーでは，仕様由来の制限により，どうしても無関係なプロセスを再起動せざるを得ない場合があり，これらの要求を満たす監視ツリーの作成は容易ではない．通常はこれらの要求を満たす監視ツリーをプログラマが注意深く作成するが，プロセスの数が増えてくると困難になる．

この問題解決のため，Neykova らは，プロセス間の通信の向きや順序を表現可能な型システムを利用し，従来の監視ツリーとは異なる仕組みで最小限のプロセスのみ再起動する手法を提案した．しかし，この手法はプロセスの定義方法に強い規約を設けており，従来の Erlang システムへの適用は難しい．

本研究では，最適な監視ツリーを自動生成するために解くべき問題を定式化し，その問題を解くアルゴリズムを提案する．解くべき問題—最適監視ツリー問題—は，出力する監視ツリーの「コスト」を最小化する最適化問題として定義した．コストとは，各プロセスが終了したときに再起動されるプロセスの数の合計である．また，監視ツリーは制約として，終了

したプロセスに依存する全てのプロセスが再起動されなければならない．最適監視ツリー問題を解くアルゴリズムの主要なアイディアは，vertex splitter と定義される「良い」性質を満たす，ある極小の頂点集合と，最適な監視ツリーは構造が一致する，という観察による．この観察に基づき，提案アルゴリズムは全ての極小の vertex splitter に対する監視ツリーを求め，その中から最もコストが小さい監視ツリーを出力する．アルゴリズムの最適性の証明もまた，ある極小の vertex splitter とコストが最小の監視ツリーは構造が一致することを示す形で実現した．この性質を直接利用してアルゴリズムの最適性を示す場合，全ての極小の vertex splitter の集合を求める必要があるが，これを多項式時間で解くアルゴリズムはまだ発見されていない．しかし，最適監視ツリー問題を解くためには，各極小 vertex splitter S ではなく vertex splitter の制約の一部を満たす空でない S の部分集合（の任意の一つ）を求めれば十分であることを証明し，これは多項式時間で解けることを示した．

入力のグラフの頂点数を V とすると，提案アルゴリズムは，病的なケースに対しては $\mathcal{O}\left(V^2 2^V\right)$ の計算時間がかかることがわかっている．しかし，現実的な多くの場合のグラフに対しては，$V = 70$ の場合でも，100 回の試行における実行時間の中央値は 1.3415 秒であった．2024 年 1 月 19 日時点で GitHub に公開されている 999 個の Erlang システムを調査したところ，本研究が再起動をサポートするプロセスの数が 70 個以下のシステムは約 99.6% だったため，提案アルゴリズムは十分高速に動作すると考える．

キーワード　最適化問題，Erlang，監視ツリー，耐故障性

Abstract

Automatic Generation of an Optimal Supervision Tree in Erlang

SASAKI, Shoichi

Erlang is a concurrency-oriented functional programming language that comes standard with lightweight processes that are lighter than OS threads. One of Erlang's main features is its error handling approach, where processes that encounter unexpected errors are just terminated and restarted. Erlang provides a supervision tree as a mechanism to realize this concept. The tree can control the processes that are restarted when a process terminates, with some restrictions. The requirement when creating a supervision tree is to minimize the number of processes that are restarted at one time, since processes that are restarting cannot execute programs. On the other hand, if a process that depends on a process terminates, it can be judged that the process cannot continue processing, so all processes that depend on the terminated process should be restarted. In a supervision tree, there are cases where, due to specification-derived limitations, it is inevitable to restart unrelated processes, and it is not easy to create a tree that satisfies these requirements. Normally, programmers carefully create a tree that satisfies these requirements, but this becomes difficult as the number of processes increases.

To solve this problem, Neykova et al. proposed a method to restart only a minimum number of processes by using a type system that can express the direction and order of communication between processes, a mechanism different from the conventional supervision tree. However, this method has strong conventions on how to define processes,

making it difficult to apply to conventional Erlang systems.

In this paper, we formulate the problem to be solved in order to automatically generate an optimal supervision tree, and propose an algorithm to solve the problem. The problem to be solved —Optimal Supervision Tree Problem— was defined as an optimization problem to minimize the *cost* of the output tree. The cost is the sum of the number of processes that are restarted when each process terminates. As a constraint, the supervision tree must restart all the processes that depend on the terminated processes. The main idea of the algorithm for solving the problem is based on the observation that the optimal supervision tree matches a structure with a some local minimum vertex set that satisfy the *good* property defined as a vertex splitter. Based on this observation, the proposed algorithm finds the supervision trees for all local minimum vertex splitters and outputs the tree with the lowest cost. Proof of the algorithm's optimality is also achieved by showing the same idea. Directly exploiting this property to show the optimality requires finding the vertex splitters, and no algorithm has yet been discovered to solve this in polynomial time. However, we proved that to solve this problem, it is sufficient to find not each local minimum vertex splitter S but a nonempty subset of S (any one of them) that satisfies the some constraints of vertex splitter, and showed that it can be solved in polynomial time.

Let V be the number of vertices in the input graph, the proposed algorithm takes $\mathcal{O}\left(V^2 2^V\right)$ computation time for corner cases. However, for many realistic cases, the median execution time over 100 trials was 1.3415 seconds, even for $V = 70$. A research of 999 Erlang systems available on GitHub as of January 19, 2024 shows that about 99.6 percent of the systems have less than 70 processes that are supported to restart by the proposed method, so we believe that the proposed algorithm is fast enough.

key words Optimization problem, Erlang, Supervision trees, Fault tolerance

目次

図目次

表目次

第 1 章

はじめに

1.1 動機

Erlang [1] とは並行指向の関数型プログラミング言語であり，OS スレッドより軽い軽量プロセスを標準で備えている．Erlang では，複数の軽量プロセスが Erlang VM 上で並行・並列に動作し，軽量プロセス間のデータ送受信による並行プログラミングが可能である．Erlang の大きな特徴として，想定外のエラーが発生した軽量プロセスはそのまま終了させて再起動する，という考え方のエラーハンドリングが主流な点が挙げられる．Erlang は，この考え方を実現する仕組みとして監視ツリーを提供している．監視ツリーの設定により，いくつかの制限はあるが，プロセス終了時に再起動されるプロセスらを制御できる．

監視ツリーを作成する際の要求として，再起動中のプロセスはプログラムを実行できないから，一度に再起動されるプロセスの数を最小限にしたい．一方で，依存するプロセスが終了したプロセスは処理の継続が困難と判断できるから，終了したプロセスに依存する全てのプロセスは再起動したい．監視ツリーでは，仕様由来の制限により，どうしても無関係なプロセスを再起動せざるを得ない場合があり，これらの要求を満たす監視ツリーの作成は容易ではない．通常はこれらの要求を満たす監視ツリーをプログラマが注意深く作成するが，プロセスの数が増えてくると困難になる．プロセスの全体像を把握することの困難さは [2] でも指摘されている．

この問題を解決するため，Neykova ら [3] は，プロセス間の通信の向きや順序を表現可能な型システムを利用し，従来の監視ツリーとは異なる仕組みで最小限のプロセスのみ再起動する手法を提案した．しかし，この手法はプロセスの定義方法に強い規約を設けており，

従来の Erlang システムへの適用は難しい．具体的には，プロセスを実装するために専用のコールバック関数を実装する必要や，プロセス間の通信には Erlang のプリミティブとは異なる専用の関数を用いる必要がある．そこで，従来の Erlang システムへの適用が容易な，プロセス数の増加に耐えうる再起動法を確立したい．

1.2 アプローチ

本研究では，従来の Erlang システムに適用可能なプロセスの再起動法の実現を目指し，最適な監視ツリーの自動生成法を提案する．提案手法の手順は大きく次の 2 段階に分かれる:

1. ユーザの Erlang コードから軽量プロセス間の依存関係を抽出する．
2. 抽出した依存関係から最適な監視ツリーを自動生成する．

ここで，プロセス間の依存関係とは，[3] と同様にプロセスの通信の向きと定義する．例えば，プロセス p_1 がプロセス p_2 にメッセージを送信している場合,「p_2 は p_1 に依存している」となる．また，最適な監視ツリーとは，再起動されるプロセス数が最小の監視ツリーと定義する．

手順 2 では，定式化された問題—最適監視ツリー問題—を解いて最適な監視ツリーを生成する．最適監視ツリー問題は，再起動されるプロセス数を最小化する最適化問題として定義する．本研究では，この問題を解くアルゴリズムを提案し，その出力の正当性と最適性を証明する．提案アルゴリズムの最悪時間計算量は指数的であるが，現実的な多くの場合に対しては十分高速に動作することを実験により示す．

なお，本研究では提案手法の手順 2 に着目しており，手順 1 に関しては以下の制約がある．まず，サポートするプロセスは，gen_server という特殊なプロセスのみである．Erlang には，Java の継承に似る，特定の振る舞いを行う軽量プロセスの実装を支援する機構があり，gen_server はその一種である．それらのプロセスはいくつかのコールバック関数を定義して実装する．また，gen_server 間の通信方法として，`gen_server:call/2,3`,

gen_server:cast/2 [*1] のみサポートしている．任意の gen_server g_1, g_2 と任意の Erlang 項 t に対して，g_1 が例えば gen_server:cast(g_2, t) を実行することは,「g_1 が g_2 にメッセージ t を送信する」という意味になる．最後に，プロセス間の依存関係の抽出は，単純なルールベースに基づいている．具体的には，gen_server を実装するモジュールの各コールバック関数のスコープのトップレベルにある，gen_server:call/2,3, gen_server:cast/2 の関数呼び出しのみを抽出している．また，それぞれの関数呼び出しの第一引数には，gen_server の名前を表すリテラルが与えられると仮定している．

1.3 貢献

本研究の主な貢献は次の通りである:

1. 最適監視ツリー問題の定式化（3 章）
2. 最適監視ツリー問題を解くアルゴリズムの開発・証明（4, 5 章）
3. 現実的な多くの場合において，提案アルゴリズムが十分高速に動作することを示すための実験的評価（6 章）

7 章では関連研究を述べ，8 章では結論を述べる．

[*1] Erlang では，モジュール m で定義されたアリティ n の関数 f を m:f/n と書く．

第 2 章

Erlang

Erlang における並行プログラミングは，軽量プロセス間のメッセージパッシングにより実現される．軽量プロセスは OS スレッドより軽いため，高頻度なプロセスの作成・削除にも耐えうる．Erlang の大きな特徴である「想定外のエラーが発生した軽量プロセスはそのまま終了させて再起動する」という考え方のエラーハンドリングは，Erlang が標準で提供するリンクという機構を用いて実現される．プロセスの終了は，リンクがつながっている全てのプロセスに通知される．この通知を受けたプロセスが終了したプロセスを再び作成することで，再起動によるエラーハンドリングが実現される．

Erlang には，Java の継承に似る，特定の振る舞いを行う軽量プロセスの実装を支援するビヘイビアという機構がある．ビヘイビアの一種が gen_server とスーパーバイザである．特にスーパーバイザは，先述したプロセスの監視・再起動を行うプロセスの実装を支援する．

本章では，並行プログラミングとエラーハンドリングに焦点を当てて Erlang を解説する．

2.1 並行プログラミング

メッセージパッシングによる並行プログラミングのため，Erlang は次の標準 API を提供する:

プロセスの作成 `spawn(Fun)` は，`Fun` 関数を実行する新しいプロセスを作成する．戻り値はプロセスの ID (PID) である．

データの送信 `Pid ! Term` は，プロセス `Pid` に対して Erlang 項 `Term` を送信する．戻り値は `Term` である．

```
-module(ping_pong).

-export([main/0]).

main() ->
    PingPid = self(),
    PongPid = spawn(fun() -> pong(PingPid) end),
    ping(10, PongPid).

%% Ping process
ping(0, Pid) -> Pid ! finished;
ping(N, Pid) ->
    Pid ! ping,
    receive pong -> ping(N - 1, Pid) end.

%% Pong process
pong(Pid) ->
    receive
        ping ->
            Pid ! pong,
            pong(Pid);
        finished -> ok
    end.
```

図 2.1: Ping-Pong プログラム

データの受信 `receive Pattern1 -> Exp1; Pattern2 -> Exp2; ...; PatternN -> ExpN end` は，受信したデータとパターンがマッチする節の式を評価する．戻り値はその評価結果である．

図 2.1 は，上記の API を用いて実装された Ping-Pong プログラムである．なお，`self/0` は，その関数を実行しているプロセスの PID を返す関数である．上記のプログラムの `ping_pong:main/0` を実行すると，まず Ping プロセスは Pong プロセスを作成し，2 つの

プロセスが 10 回 `ping` と `pong` を交互に送信しあった後，Ping プロセスが Pong プロセスに `finished` を送信して終了する．

2.2　再起動によるエラーハンドリング

Erlang は，プロセスの終了を他のプロセスに通知するリンクという機構を提供する．プロセスの終了は，そのプロセスにリンクが繋がっている全てのプロセスに通知される．リンクの作成・削除に関する標準 API は次の通りである:

リンクの作成 `link(Pid)` は，この関数を実行しているプロセスと `Pid` のプロセス間にリンクを作成する．戻り値は `true` である．

リンクの削除 `unlink(Pid)` は，この関数を実行しているプロセスと `Pid` のプロセス間のリンクを削除する．戻り値は `true` である．

図 2.2 は，上記の API を用いて実装された，素朴な再起動によるエラーハンドリングを行うプログラムである．なお，`process_flag(trap_exit, true)` は，プロセスの終了通知を受信するための設定である．この設定をしない場合，通知を受け取ったプロセスはそのまま終了する．上記のプログラムの `trivial_error_handling:main()` を実行すると，メッセージを受け取ると相手にフィボナッチ数を返信するサーバ（プロセス）を作る．その後，そのサーバに valid なリクエストを送信するが，次に invalid なリクエストを送信してサーバがエラーで終了する．最後に，終了したサーバを再び作成し，エラーが発生する前の状態に戻ったことを確認する．

2.3　gen_server

gen_server とはビヘイビアの一種であり，クライアント・サーバモデルにおけるサーバの機能を有する軽量プロセスの実装を支援する．プログラマは特定のコールバック関数を定義して，サーバの振る舞いをする軽量プロセスを実装できる．以下は gen_server が実装を要

```
-module(trivial_error_handling).

-export([main/0]).

main() ->
    process_flag(trap_exit, true), % Trap a notification
    FibServerPid = spawn(fun() -> fib_server() end),
    link(FibServerPid),

    FibServerPid ! {self(), 10},
    receive Response -> io:format("~p~n", [Response]) end, % Output 55

    FibServerPid ! {self(), -10}, % Fibonacci server crashes
    NewFibServerPid =
        % Spawn a fibonacci server again
        receive {'EXIT', FibServerPid, _} -> spawn(fun() -> fib_server()
            end) end,
    link(NewFibServerPid),

    NewFibServerPid ! {self(), 10},
    receive Response -> io:format("~p~n", [Response]) end. % Output 55

%% Fibonacci server
fib_server() ->
    receive
        {Pid, N} ->
            Pid ! fib(N),
            fib_server()
    end.

fib(0) -> 0;
fib(1) -> 1;
fib(N) when N >= 2 -> fib(N - 1) + fib(N - 2).
```

図 2.2: 素朴な再起動によるエラーハンドリング

求するコールバック関数である:

初期化 `Module:init/1` はプロセスを初期化する．この関数の実装は必須である．

同期通信 `Module:handle_call/3` は受信したリクエストに対して即座にレスポンスを返す．この関数の実装は必須である．

非同期通信 `Module:handle_cast/2` は受信したリクエストを処理する．レスポンスを返すとは限らない．この関数の実装は必須である．

想定外リクエストの処理 `Module:handle_info/2` は想定外のリクエストを処理する．この関数の実装はオプショナルである．

状態の公開 `Module:format_status/1,2` は公開するプロセスの状態を決定する．この関数の実装はオプショナルである．

状態のマイグレーション `Module:code_change/3` はプロセスの状態をマイグレーションする．この関数の実装はオプショナルである．

処理の継続 `Module:handle_continue/2` は中断された処理を継続する．この関数の実装はオプショナルである．

終了処理 `Module:terminate/2` はプロセス終了前にすべき処理を行う．この関数の実装はオプショナルである．

図 2.3 は，gen_server を用いて実装されたフィボナッチサーバである．サーバの状態はフィボナッチ数のメモであり，`Module:handle_call/3` では計算したフィボナッチ数を即座に返し，`Module:handle_cast/2` では裏でフィボナッチ数を計算している．

2.4　スーパーバイザ

スーパーバイザとはビヘイビアの一種であり，軽量プロセスを開始・停止・監視するプロセスを実装できる．スーパーバイザは gen_server とリンクを用いて実装されている．スーパーバイザは再起動によるエラーハンドリングを実現するための根幹であり，一般的にはスーパーバイザを用いて監視ツリーが作られる．監視ツリーにおいて，gen_server は葉，

```
-module(fib_server).

-behavior(gen_server).

-export([init/1, handle_call/3, handle_cast/2]).

%% Callback functions
init(_Args) -> {ok, #{}}.

handle_call(N, _From, State) ->
    {Ans, NewState} = fib(N, State),
    {reply, Ans, NewState}.

handle_cast(N, State) ->
    {_, NewState} = fib(N, State),
    {noreply, NewState}.

%% Private functions
fib(N, Memo) when N >= 0 ->
    case maps:find(N, Memo) of
        {ok, Ans} -> {Ans, Memo};
        error ->
            {Ans, NewMemo} =
                case N of
                    0 -> {0, Memo};
                    1 -> {1, Memo};
                    _ ->
                        {X, Memo2} = fib(N - 1, Memo),
                        {Y, Memo3} = fib(N - 2, Memo2),
                        {X + Y, Memo3}
                end,
            {Ans, maps:put(N, Ans, NewMemo)}
    end.
```

図 2.3: gen_server を用いたフィボナッチサーバの実装

スーパーバイザは内部節点に相当する．なお，後述する simple_one_for_one という再起動戦略のスーパーバイザを除いて，スーパーバイザの子プロセス間には順序が決められている．子プロセスはその順序で開始され，逆順に停止される．子プロセスを開始・停止する操作は同期的に実行される．

各スーパーバイザに設定される再起動戦略によって，再起動されるプロセスが決定される．Erlang が提供する再起動戦略は次の 4 つである:

- one_for_one
- one_for_all
- rest_for_one
- simple_one_for_one

one_for_one は終了した子プロセスのみを再起動する．one_for_all は終了した子プロセスだけではなく，他の全ての子プロセスもまた再起動する．rest_for_one は終了した子プロセスと「残り」の子プロセスを再起動する.「残り」の子プロセスとは，終了した子プロセスより順序が後のプロセスを指す．simple_one_for_one は終了した子プロセスのみを再起動する点は one_for_one と同じであるが，子プロセスが動的に追加・削除される点で異なる．

なお，スーパーバイザが再起動されるときは，そのスーパーバイザの子プロセスが再帰的に再起動される．また，スーパーバイザには再起動回数の上限や子プロセスの終了処理にタイムアウト時間を設定できる．

以下はスーパーバイザが実装を要求するコールバック関数である:

初期化 Module:init/1 はプロセスを初期化する．戻り値は，スーパーバイザの再起動戦略や監視する子プロセスの設定を表す Erlang 項である．この関数の実装は必須である．

図 2.4 は，2.3 節で述べたフィボナッチサーバを監視するスーパーバイザである．このスーパーバイザは，one_for_one の再起動戦略でフィボナッチサーバのみ監視する．Erlang の

```
-module(fib_server_supervisor).

-behavior(supervisor).

-export([init/1]).

init(_Args) ->
    SupFlags = #{strategy => one_for_one},
    ChildSpecs =
        [#{id => fib_server,
           start => {gen_server, start_link, [{local, fib_server},
               fib_server, [], []]},
           type => worker}],
    {ok, {SupFlags, ChildSpecs}}.
```

図 2.4: フィボナッチサーバを監視するスーパーバイザの実装

REPL による，このスーパーバイザの実行例を図 2.5 に示す.

```
1> {ok, Pid} = supervisor:start_link(fib_server_supervisor, []).
2> unlink(Pid). % フィボナッチサーバの終了通知がトップレベルまで届かないようにする
3> gen_server:call(fib_server, 10).
55
4> gen_server:call(fib_server, -10). % フィボナッチサーバがエラーで終了する
=ERROR REPORT====
...(omitted)...
5> gen_server:call(fib_server, 10). % フィボナッチサーバが再起動されている
55
```

図 2.5: スーパーバイザによるプロセスの再起動

第 3 章

問題の定式化

本章では,「最適」な監視ツリーを生成するために解くべき問題を定式化する．また，その定式化が妥当と主張するための予備実験を行う.

1 章で議論した通り，問題を解いて得たいのは，次の 2 つの制約を満たす監視ツリーである:

1. ある gen_server が再起動されるとき，その gen_server に依存する gen_server もまた再起動される.
2. 同時に再起動される gen_server の数が最小.

問題の定式化では，入力が gen_server の依存関係であり，出力が上記の制約を満たす監視ツリーとなるように定義する．なお，本研究では，スーパーバイザの再起動戦略のうち `simple_one_for_one` はサポートしない [*1].

3.1 定式化

$\mathbb{N} = \{1, 2, \ldots\}$ は自然数の集合とし，有限集合 A の要素数を $|A|$ と書く．また，任意の有向グラフ $G(V, E)$ と任意の頂点 $v, u \in V$ について，v から u に到達可能であることを

[*1] 2.4 節で述べた通り，`simple_one_for_one` のスーパーバイザの子プロセスは動的に追加・削除される．これらの操作を行う関数 `supervisor:start_child/2`, `supervisor:delete_child/2` は，第一引数にスーパーバイザの名前を要求する．よって，`simple_one_for_one` のスーパーバイザを用いる場合は，提案手法に入力されるユーザのコードにてその名前を指定する必要があるが，入力前はそのスーパーバイザが存在しないため不可能である．なんらかの規約を設けて名前を特定する方針も考えられるが，本研究では簡単のために `simple_one_for_one` をサポートしない選択をとった.

$v \rightsquigarrow_G u$ と書く．さらに，gen_server とスーパーバイザを合わせてプロセスと呼ぶ．

有向グラフ $G_d(V_g, E_g)$ を依存関係グラフと呼ぶ．V_g は gen_server の有限集合であり，$E_g \subseteq V_g \times V_g$ は gen_server 間の依存関係である．$(g_1, g_2) \in E_g$ は「g_1 は g_2 に依存する」ことを表す．

依存関係グラフ $G_d(V_g, E_g)$ に対し，後述の制約を満たす 6 字組 $T(V_c, V_s, r, \pi, ord, stg)$ を部分監視ツリーと呼ぶ．$V_c \subseteq V_g$ は葉の gen_server の有限集合，V_s はスーパーバイザの有限集合，$r \in V_s$ は根，$\pi : (V_c \cup V_s \setminus \{r\}) \to V_s$ は親，$ord : (V_c \cup V_s \setminus \{r\}) \to \mathbb{N}$ は兄弟間の順序，$stg : V_s \to \{\texttt{ofo}, \texttt{ofa}, \texttt{rfo}\}$ はスーパーバイザの再起動戦略を表す．なお，$V_c = V_g$ である部分監視ツリーは，単に監視ツリーと呼ぶ．部分監視ツリーは次の制約を満たす:

- $V_c \cap V_s = \emptyset$
- $(V_c \cup V_s, r, \pi)$ は根付き木
- 任意の $s \in V_s$ と $v, u \in \mathit{child}(T, s)$ に対して，$v \neq u$ ならば $ord(v) \neq ord(u)$
- 任意の $v, u \in V_c$ について，$v \rightsquigarrow_{G_d} u$ ならば $v \in \mathit{restart}(T, u)$

ただし，$\mathit{child}(T, v)$ は v の子プロセスの集合を表し，$\mathit{child}(T, v) = \{u \mid \pi(u) = v\}$ で定義される．また，$\mathit{restart}(T, v)$ は v の終了時に再起動されるプロセスの集合を表す．ここで，$\mathit{sibling}(T, v) = \mathit{child}(T, \pi(v))$，$\mathit{follow}(T, v) = \{u \in \mathit{sibling}(T, v) \mid ord(v) < ord(u)\}$ とすると，$\mathit{restart}(T, v \in V_c \cup V_s)$ は次の制約を満たす最小の集合である:

- $v \in V_c$ ならば $v \in \mathit{restart}(T, v)$
- $v \in V_s$ ならば $\bigcup_{u \in \mathit{child}(T,v)} \mathit{restart}(T, u) \subseteq \mathit{restart}(T, v)$
- $stg(\pi(v)) = \texttt{ofa}$ ならば $\bigcup_{u \in \mathit{sibling}(T,v)} \mathit{restart}(T, u) \subseteq \mathit{restart}(T, v)$
- $stg(\pi(v)) = \texttt{rfo}$ ならば $\bigcup_{u \in \mathit{follow}(T,v)} \mathit{restart}(T, u) \subseteq \mathit{restart}(T, v)$

部分監視ツリーのコストは，$\mathit{cost}(T) = \sum_{v \in V_c} |\mathit{restart}(T, v)|$ と定義される．

以上の定義を用いて，問題は以下の入力と出力で定義される:

入力 依存関係グラフ (V_g, E_g)

出力 コストが最小の 6 字組 $(V_g, V_s, r, \pi, ord, stg)$

3.2　スーパーバイザの再起動にかかるオーバーヘッドの調査

3.1 節で定式化した監視ツリーのコストには，スーパーバイザの数が含まれない．本節では，スーパーバイザの再起動にかかるオーバーヘッドが極端に小さいことを実験的に示し，定式化の妥当性を示す．

実験では，複数の gen_server を監視する 1 つのスーパーバイザの再起動時間を計測する．再起動時間の定義は後述する．実験の手順は次の通りである:

1. 実験に使用する gen_server とスーパーバイザの Erlang コードを自動生成
2. スーパーバイザの再起動時間を計測

各試行につき，gen_server の数を $100, 200, \ldots, 1000$ と変動させる．また，手順 2 では，計測する再起動時間のばらつきを抑えるために 50 回分の平均を取る．

スーパーバイザの再起動時間は，そのスーパーバイザの終了時間と，そのスーパーバイザの起動時間の差と定義する．終了時間とは，`gen_server:stop/1` によって，そのスーパーバイザを終了する直前の時間とする．また，スーパーバイザの起動時間は，そのスーパーバイザの後ろのスーパーバイザの `Module:init/1` が実行される時間とする．起動時間の定義がやや直感に反する理由は，Erlang の技術的問題 [*2] により，スーパーバイザの起動時間の計測は容易ではないからである．この技術的問題に対応するため，実際のスーパーバイザの起動時間より大きい時間を起動時間としている．本実験の目的は「スーパーバイザの再起動にかかるオーバーヘッドは極端に小さい」と示すことだから，実際の再起動時間より大きい再起動時間の計測でも十分である．

[*2] gen_server の場合，`gen_server:start/3` や `gen_server:start_link/3` の第 3 引数に `statistics` を含めて起動された gen_server は，`sys:statistics/2` を用いて起動時間を計測できる．スーパーバイザは gen_server を用いて実装されているため同様の計測方法が考えられるが，`statistics` を含めて起動されないため実現できない．Erlang の実装に手を加える形で，スーパーバイザが `statistics` を含めて起動されるようにする対応も考えられるが，その場合はその変更が実験結果に与える影響の調査も必要になる．

図 3.1: スーパーバイザの再起動時間

実験環境は以下の通りである:

CPU Intel(R) Xeon(R) E-2378 CPU @ 2.60GHz 64bits 8C/16T

DRAM UDIMM 32GB * 2(64 GiB)

OS Ubuntu 22.04.3 LTS

Erlang/OTP 26

Rebar3（ビルドツール） 3.18.0

実験の結果を図 3.1 に示す．図の横軸はグラフの頂点数，すなわち gen_server の数を表し，縦軸はスーパーバイザの再起動時間を表す．この実験の結果より，スーパーバイザの再起動時間は，子の gen_server の数が 1000 の場合，平均 5 ミリ秒であることがわかった．提案手法を用いて生成されるスーパーバイザが行う初期化処理は，リテラルを用いたスーパーバイザの戻り値の作成だけであり，これは定数時間で終了する．よって，監視ツリーの再起動時間で支配的なのは，ユーザが定義する gen_server の初期化・終了処理にかかる時間となる．以上より，スーパーバイザの再起動にかかるオーバーヘッドは極端に小さいと考える．

第 4 章

最適監視ツリー問題を解くアルゴリズム

本章では，3 章で定式化した問題を解くアルゴリズムについて議論する．まず初めに，準備として，アルゴリズムの定義やその性質の議論に利用する定義を述べる．次に，最適監視ツリー問題を解くアルゴリズムを定義し，そのアルゴリズムが正当性と最適性を満たすことを証明する．最後に，アルゴリズムの計算量を解析する．

4.1 準備

4.1.1 標準形

アルゴリズムの性質を議論するにあたって，特定の構造を持った監視ツリーのみについて議論できると便利である．そこで，任意の監視ツリーを表現可能な「標準形」の監視ツリーを定義する．標準形の監視ツリーとは，以下の制約を満たす監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ である:

1. 任意の $s \in V_s$ に対して，$stg(s) = \texttt{ofa}$ ならば $|child(T, s) \cap V_s| \leq 1$
2. 任意の $s \in V_s$ に対して，$stg(s) = \texttt{rfo}$ ならば，$|child(T, s) \cap V_s| = 0$ または ($|child(T, s) \cap V_s| = 1$ かつ ($\{s'\} = child(T, s) \cap V_s$ なる s' に対して，$follow(T, s') = \emptyset$))
3. 任意の $s \in V_s$ に対して，$stg(s) = \texttt{ofo}$ ならば，任意の $s' \in child(T, s) \cap V_s$ に対して，

$stg(s') \neq \texttt{ofo}$

4. 任意の $s \in V_s$ に対して，$stg(s) \in \{\texttt{ofo}, \texttt{ofa}\}$ ならば $|child(T, s)| \geq 2$
5. 任意の $s \in V_s$ に対して，$stg(s) = \texttt{rfo}$ ならば $|child(T, s)| \geq 1$

アルゴリズム 6 で定義する関数 `normalize` は，任意の監視ツリーを標準形の監視ツリーに変換する．アルゴリズム 1–5（それぞれ関数 `normalize1`, `normalize2`, `normalize3`, `normalize4`, `normalize5` を定義する）は，アルゴリズム 6 のヘルパー関数である．ここで，$make\_name()$ は重複しないランダムな名前を返す関数である．

これより，上記のアルゴリズムの正当性を証明する．以降では，任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ の各要素を $T.V_c$, $T.V_s$, $T.r$, $T.\pi$, $T.ord$, $T.stg$ と表記する場合がある．任意のグラフ $G(V, E)$ に対しても同様である．また，$subtree(T, s \in V_s)$ は T の部分監視ツリーを表し，次の制約を満たす $T'(V'_c, V'_s, r', \pi', ord', stg')$ と定義する:

- $V'_c = restart(T, s)$
- $s \in V'_s$
- 任意の $s' \in V'_s$ に対して，$child(T, s') \cap V_s \subset V'_s$
- $r' = s$
- 任意の $v \in V'_c \cup V'_s \setminus \{r'\}$ に対して，$\pi'(v) = \pi(v)$ かつ $ord'(v) = ord(v)$
- 任意の $v \in V'_s$ に対して，$stg'(v) = stg(v)$

さらに，任意の有限列 s $(= e_1, e_2, \ldots, e_n)$ に対して，$s[i]$ は e_i を表し，$|s|$ は s の長さ n を表す．列の要素を集合のように取り出す表記 $e \in s$ は，$e \in \{s[i] \mid i \in \mathbb{N}, 1 \leq i \leq |s|\}$ を意味する．

補題 4.1.1. 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$T' = \texttt{normalize1}(T)$ とすると，$V_c = T'.V_c$，かつ，T' は制約 *1* を満たし，かつ，(任意の $v \in V_c$ に対して，$restart(T, v) = restart(T', v)$) .

アルゴリズム 1 制約 1 を満たすよう変換する関数

1: **function** `normalize1`$(T(V_c, V_s, r, \pi, ord, stg))$
2: **for all** $s \in V_s$ **do**
3: **if** s が制約 1 を満たさない **then**
4: $s' \leftarrow make\_name()$
5: $V_s \leftarrow V_s \cup \{s'\}$
6: **for all** $c \in child(T, s) \cap V_s$ **do**
7: $\pi(c) \leftarrow s$
8: $ord(s') \leftarrow (\max_{c \in child(T,s)} ord(c)) + 1$
9: $\pi(s') \leftarrow s$
10: $stg(s') \leftarrow$ `ofo`
11: **return** T

アルゴリズム 2 制約 2 を満たすよう変換する関数

```
1: function normalize2(T(V_c, V_s, r, π, ord, stg))
2:     for all s ∈ V_s do
3:         if s が制約 2 を満たさない then
4:             c_s ← ord の昇順で並ぶ {c ∈ child(T, s) ∩ V_s | follow(T, c) ≠ ∅} の要素の列
5:             for i ← 1 to |c_s| do
6:                 c ← c_s[i]
7:                 s' ← make_name()
8:                 V_s ← V_s ∪ {s'}
9:                 for all c' ∈ follow(T, c) do
10:                    π(c') ← s'
11:                ord(s') ← ord(c) + 1
12:                stg(s') ← rfo
13:
14:                s'' ← make_name()
15:                V_s ← V_s ∪ {s''}
16:                π(s'') ← π(c)
17:                π(c) ← s''
18:                π(s') ← s''
19:                ord(s'') ← ord(c)
20:                stg(s'') ← ofo
21:    return T
```

アルゴリズム 3 制約 3 を満たすよう変換する関数

1: **function** `normalize3`$(T(V_c, V_s, r, \pi, ord, stg))$
2: **for all** $s \in V_s$ **do**
3: **while** s が制約 3 を満たさない **do**
4: **for all** $s' \in \{s' \mid s' \in child(T, s) \cap V_s, stg(s') = \texttt{ofo}\}$ **do**
5: **for all** $c \in child(T, s')$ **do**
6: $ord(c) \leftarrow (\max_{c' \in child(T,s)} ord(c')) + 1$
7: $\pi(c) \leftarrow s$
8: $V_s \leftarrow V_s \setminus \{s'\}$
9: **return** T

アルゴリズム 4 制約 4 を満たすよう変換する関数

1: **function** `normalize4`$(T(V_c, V_s, r, \pi, ord, stg))$
2: **for all** $s \in V_s$ **do**
3: **if** s が制約 4 を満たさない **then**
4: $stg(s) \leftarrow \texttt{rfo}$
5: **return** T

アルゴリズム 5 制約 5 を満たすよう変換する関数

1: **function** `normalize5`$(T(V_c, V_s, r, \pi, ord, stg))$
2: **for all** $s \in V_s$ **do**
3: **if** s が制約 5 を満たさない **then**
4: $V_s \leftarrow V_s \setminus \{s\}$
5: **return** T

アルゴリズム 6 任意の監視ツリーを標準形の監視ツリーに変換する関数

1: **function** `normalize`$(T(V_c, V_s, r, \pi, ord, stg))$

2: $T \leftarrow$ `normalize1`(T)

3: $T \leftarrow$ `normalize2`(T)

4: $T \leftarrow$ `normalize3`(T)

5: $T \leftarrow$ `normalize4`(T)

6: $T \leftarrow$ `normalize5`(T)

7: **return** T

証明. **まず，$V_c = T'.V_c$ であることを示す．**

アルゴリズムの中で V_c は再代入されないため成り立つ．

次に，T' は制約 1 を満たし，かつ，(任意の $v \in V_c$ に対して，$restart(T, v) = restart(T', v)$) であることを示す．

ループの各繰り返しに対して，ループ内の最初の文が評価される前の T を T_b とし，最後の文が評価された後の T を T_a とする．任意の $s' \in T_b.V_s$ に対して，T_b と s' が制約を満たすならば，そのような s' の子や子の順序，再起動戦略は変わらないから，T_a と s' もまた制約を満たす．同様の理由で，再起動されるプロセスの集合も等しい．

ここで，ループ変数を s とする．T_b と s に対して制約が成り立つ場合，$T_b = T_a$ だから T_a と s は制約を満たす．再起動されるプロセスの集合も等しい．そうでない場合，ループ内で新しく作られるスーパーバイザを x とすると，アルゴリズムの定義より，$child(T_a, s) \cap T_a.V_s = \{x\}$ かつ $T_a.stg(s) = \texttt{ofa}$ かつ $T_a.stg(x) = \texttt{ofo}$ だから，T_a と s，および x は制約を満たす．また，$child(T_b, s) \cap T_b.V_c = child(T_a, s) \cap T_a.V_c$ かつ $child(T_b, s) \cap T_b.V_s = child(T_a, x)$ だから，再起動されるプロセスの集合も等しい．

以上の処理が，制約を満たさない全ての $s \in V_s$ に対して実行され，一度制約を満たした s はその後常に制約を満たすため，T' は制約を満たし，再起動されるプロセスの集合も等しい．

□

補題 4.1.2. 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$T' = \texttt{normalize2}(T)$ とすると，T が制約 *1* を満たすならば，$V_c = T'.V_c$，かつ，T' は制約 *1, 2* を満たし，かつ，(任意の $v \in V_c$ に対して，$restart(T, v) = restart(T', v)$) .

証明. **まず，$V_c = T'.V_c$ であることを示す．**

アルゴリズムの中で V_c は再代入されないため成り立つ．

次に，T' は制約 1, 2 を満たし，かつ，(任意の $v \in V_c$ に対して，$restart(T, v) = restart(T', v)$) であることを示す．

外ループの各繰り返しに対して，ループ内の最初の文が評価される前の T を T_b とし，最後の文が評価された後の T を T_a とする．任意の $s' \in T_b.V_s$ に対して，T_b と s' が制約を満たすならば，そのような s' の子や子の順序，再起動戦略は変わらないから，T_a と s' もまた制約を満たす．同様の理由で，再起動されるプロセスの集合も等しい．

ここで，外ループ変数を s とする．T_b と s に対して制約が成り立つ場合，$T_b = T_a$ だから T_a と s は制約を満たす．再起動されるプロセスの集合も等しい．そうでない場合，内ループの各繰り返しに対して，ループ内の最初の文が評価される前の T を T'_b とし，最後の文が評価された後の T を T'_a とする．ここで，内ループ変数を添字とする c_s の要素を c とする．また，内ループの中で最初に作られるスーパーバイザを s' とし，次に作られるスーパーバイザを s'' とする．アルゴリズムの定義より，T'_b, T'_a は次の性質を満たす:

- $T'_a.stg(s') = \texttt{rfo}$
- $follow(T'_b, c) = child(T'_a, s')$
- 任意の $v \in child(T'_a, s')$ に対して，$T'_b.ord(v) = T'_a.ord(v)$
- $T'_a.stg(s'') = \texttt{ofo}$
- $child(T'_a, s'') = \{c, s'\}$
- $child(T'_a, T'_a.\pi(s'')) = child(T'_b) \cup \{s''\} \setminus (\{c\} \cup follow(T'_b, c))$
- 任意の $v \in child(T'_a, T'_a.\pi(s''))$ に対して，$T'_a.ord(v) \leq T'_a.ord(s'')$

c の定義より，$child(T'_a, s) \setminus \{s''\} \subseteq T'_a.V_c$ かつ $follow(T'_a, s'') = \emptyset$ だから，T'_a と $T'_a.\pi(s'')$ は制約を満たす．また，$T'_a.stg(s'') = \texttt{ofo}$ だから，T'_a と s'' もまた制約を満たす．さらに，c_s の末尾の c に対して，c の定義より，T'_a と s' は制約を満たす．

以上の処理が，制約を満たさない全ての $c \in c_s$ に対して実行され，外ループに関する議論と同様に，一度制約を満たした $s \in T'_a.V_s$ はその後常に制約を満たすため，T_a と s は制約を満たし，再起動されるプロセスの集合も等しい．外ループに対しても同様にして，T' は制約を満たし，再起動されるプロセスの集合も等しい．

□

補題 4.1.3. 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$T' = \texttt{normalize3}(T)$ とすると，T が制約 *1, 2* を満たすならば，$V_c = T'.V_c$，かつ，T' は制約 *1–3* を満たし，かつ,（任意の $v \in V_c$ に対して，$restart(T, v) = restart(T', v)$）.

証明. まず，$V_c = T'.V_c$ であることを示す.

アルゴリズムの中で V_c は再代入されないため成り立つ.

次に，T' は制約 1–3 を満たし，かつ,（任意の $v \in V_c$ に対して，$restart(T, v) = restart(T', v)$）であることを示す.

最も外側のループの各繰り返しに対して，ループ内の最初の文が評価される前の T を T_b とし，最後の文が評価された後の T を T_a とする．任意の $s' \in T_b.V_s$ に対して，T_b と s' が制約を満たすならば，そのような s' の子や子の順序，再起動戦略は変わらないから，T_a と s' もまた制約を満たす．同様の理由で，再起動されるプロセスの集合も等しい.

ここで，最も外側のループ変数を s とする．T_b と s に対して制約が成り立つ場合，$T_b = T_a$ だから T_a と s は制約を満たす．再起動されるプロセスの集合も等しい．そうでない場合，次に深いループの各繰り返しに対して，最後の文が評価された後の T を T'_a とする．s が制約を満たすまで，再起動戦略が `ofo` の s の子供は削除され続けるから，$subtree(T'_a, s)$ は制約を満たす．削除されるスーパーバイザの子や子の順序，再起動戦略は変わらないから，再起動されるプロセスの集合も等しい.

以上の処理が，制約を満たさない全ての $s \in V_s$ に対して実行され，外ループに関する議論と同様に，一度制約を満たした s はその後常に制約を満たすため，T_a と s は制約を満たし，再起動されるプロセスの集合も等しい．外ループに対しても同様にして，T' は制約を満たし，再起動されるプロセスの集合も等しい.

□

補題 4.1.4. 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$T' = \texttt{normalize4}(T)$ とすると，T が制約 *1–3* を満たすならば，$V_c = T'.V_c$，かつ，T' は制約 *1–4* を満たし，か

つ, (任意の $v \in V_c$ に対して, $restart(T, v) = restart(T', v)$) .

証明. **まず,** $V_c = T'.V_c$ **であることを示す.**

アルゴリズムの中で V_c は再代入されないため成り立つ.

次に, T' **は制約 1–4 を満たし, かつ,(任意の** $v \in V_c$ **に対して,** $restart(T, v) = restart(T', v)$**)**
であることを示す.

外ループの各繰り返しに対して, ループ内の最初の文が評価される前の T を T_b とし, 最後の文が評価された後の T を T_a とする. 任意の $s' \in T_b.V_s$ に対して, T_b と s' が制約を満たすならば, そのような s' の子や子の順序, 再起動戦略は変わらないから, T_a と s' もまた制約を満たす. 同様の理由で, 再起動されるプロセスの集合も等しい.
ここで, 外ループ変数を s とする. T_b と s に対して制約が成り立つ場合, $T_b = T_a$ だから T_a と s は制約を満たす. 再起動されるプロセスの集合も等しい. そうでない場合, アルゴリズムの定義より, $T_a.stg(s) = \texttt{rfo}$ かつ $child(T_a, s) = child(T_b, s)$ かつ $|child(T_a, s)| \leq 1$ である. よって, T_a と s は直ちに制約 1, 3, 4 を満たす. 制約 2 もまた, $|child(T_a, s)| \leq 1$ だから成り立つ. 同様に再起動されるプロセスの集合も等しい.
以上の処理が, 制約を満たさない全ての $s \in V_s$ に対して実行され, 一度制約を満たした s はその後常に制約を満たすため, T' は制約を満たし, 再起動されるプロセスの集合も等しい.

□

補題 4.1.5. 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して, $T' = \texttt{normalize5}(T)$ とすると, T が制約 *1–4* を満たすならば, $V_c = T'.V_c$, かつ, T' は制約 *1–5* を満たし, かつ, (任意の $v \in V_c$ に対して, $restart(T, v) = restart(T', v)$) .

証明. **まず,** $V_c = T'.V_c$ **であることを示す.**

アルゴリズムの中で V_c は再代入されないため成り立つ.

次に, T' **は制約 1–5 を満たし, かつ,(任意の** $v \in V_c$ **に対して,** $restart(T, v) = restart(T', v)$**)**
であることを示す.

外ループの各繰り返しに対して，ループ内の最初の文が評価される前の T を T_b とし，最後の文が評価された後の T を T_a とする．任意の $s' \in T_b.V_s$ に対して，T_b と s' が制約を満たすならば，そのような s' の子や子の順序，再起動戦略は変わらないから，T_a と s' もまた制約を満たす．同様の理由で，再起動されるプロセスの集合も等しい．

ここで，外ループ変数を s とする．T_b と s に対して制約が成り立つ場合，$T_b = T_a$ だから T_a と s は制約を満たす．再起動されるプロセスの集合も等しい．そうでない場合，アルゴリズムの定義より，$T_a.V_s \cup \{s\} = T_b.V_s$ である．すなわち，制約を満たさない s は T_a に存在しない．また，T_b と s は制約 5 に違反しているから，$|child(T_b, s)| = 0$ であり，再起動されるプロセスの集合も等しい．

以上の処理が，制約を満たさない全ての $s \in V_s$ に対して実行され，一度制約を満たした s はその後常に制約を満たすため，T' は制約を満たし，再起動されるプロセスの集合も等しい．

□

定理 4.1.6 (標準形)**.** 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$T' = \texttt{normalize}(T)$ とすると，T' は標準形の監視ツリー，かつ，$V_c = T'.V_c$ かつ（任意の $v \in V_c$ に対して，$restart(T, v) = restart(T', v)$）.

証明. `normalize1` 呼び出しの戻り値を T_1 とすると，補題 4.1.1 より，T_1 は制約 1 を満たし，かつ，$T.V_c = T_1.V_c$ かつ（任意の $v \in V_c$ に対して，$restart(T, v) = restart(T_1, v)$）を満たす．

`normalize2` 呼び出しの戻り値を T_2 とすると，補題 4.1.2 より，T_2 は制約 1, 2 を満たし，かつ，$T_1.V_c = T_2.V_c$ かつ（任意の $v \in V_c$ に対して，$restart(T_1, v) = restart(T_2, v)$）を満たす．

同様にして，`normalize3`, `normalize4`, `normalize5` 呼び出しの各戻り値に補題 4.1.3, 4.1.4, 4.1.5 を順に適用して，所望の結論が成り立つ． □

以降の議論において，特に断らない限り，監視ツリーは標準形の監視ツリーとする．また，監視アカシア [*1] とは，スーパーバイザの再起動戦略が `rfo` と `ofa` のみの標準形の監視ツリーとする．

4.1.2 補助的な表記の定義

今後の議論で補助的に用いる表記の定義を次に列挙する:

- 任意のグラフ $G(V, E)$ に対して:
  - 入次数: $\mathit{in\_degree}((V, E), v) = |\{u \mid (u, v) \in E\}|$
  - 出次数: $\mathit{out\_degree}((V, E), v) = |\{u \mid (v, u) \in E\}|$
  - 入口頂点: $\mathit{entrance}(G) = \{v \in V \mid \mathit{in\_degree}(G, v) = 0\}$
  - 出口頂点: $\mathit{exit}(G) = \{v \in V \mid \mathit{out\_degree}(G, v) = 0\}$
  - $\mathit{reachable}(U, (V, E)) = \{v \in V \mid u \in U, u \rightsquigarrow_G v\}$
  - $\mathit{reaching}(U, (V, E)) = \{v \in V \mid u \in U, v \rightsquigarrow_G u\}$
- 任意の族 $\mathbb{S}$ に対して:
  - $\mathit{flatten}(\mathbb{S}) = \{v \mid S \in \mathbb{S}, v \in S\}$
- 任意の監視ツリー $T(V_c, V_s, r, \pi, \mathit{ord}, \mathit{stg})$ に対して:
  - 祖先: $\mathit{ancestor}(T, v) = \{s \in V_s \mid v \in \mathit{restart}(T, s)\}$
  - 部分アカシア: $\mathit{subacacia}(T) = T'(V_c', V_s', r', \pi', \mathit{ord}', \mathit{stg}')$．ただし，
    - $V_c' = V_c \setminus \bigcup_{\{s \in V_s \mid \mathit{stg}(s) = \texttt{ofo}\}} \mathit{restart}(T, s)$
    - $V_s' = \{s \in V_s \mid v \in V_c', \pi(v) = s\}$
    - $r' = r$
    - 任意の $v \in V_c' \cup V_s' \setminus \{r'\}$ に対して，$\pi'(v) = \pi(v)$ かつ $\mathit{ord}'(v) = \mathit{ord}(v)$
    - 任意の $s \in V_s'$ に対して，$\mathit{stg}'(s) = \mathit{stg}(s)$

[*1] この名称は，ゲーム「Minecraft」に登場するアカシアの木は，その木の葉が左右どちらか一方に偏る傾向にあることに由来する．

- 任意の有限列 s $(= e_1, e_2, \ldots, e_n)$ に対して:
  - $i..j$ は以下の制約を満たす有限列 s':
    - $s'[1] = i$
    - 任意の整数 $1 < k \leq j - i + 1$ に対して，$s'[k] = s'[k-1] + 1$
  - $s[i..j]$ は以下の制約を満たす有限列 s':
    - 任意の整数 $1 \leq k \leq j - i + 1$ に対して，$s'[k] = s[i + k - 1]$
- $supervise\_with\_ofo(\mathbb{T}) = T(V_c, V_s, r, \pi, ord, stg)$ は，任意の監視ツリーの族 $\mathbb{T}$ に対して，次の制約を満たす最小の監視ツリー:
  - $V_c = \mathit{flatten}(\{t.V_c \mid t \in \mathbb{T}\})$
  - $V_s = \mathit{flatten}(\{t.V_s \mid t \in \mathbb{T}\}) \cup \{r\}$
  - 任意の $t \in \mathbb{T}$ に対して，$r = \pi(t.r)$
  - 任意の $t \in \mathbb{T}$ と $v \in t.V_c \cup t.V_s$ に対して，$\pi(v) = t.\pi(v)$ かつ $ord(v) = t.ord(v)$
  - 任意の $t \in \mathbb{T}$ と $v \in t.V_s$ に対して，$stg(v) = t.stg(v)$
  - $stg(r) = \texttt{ofo}$
- $merge(T'(V_c', V_s', r', \pi', ord', stg'), T''(V_c'', V_s'', r'', \pi'', ord'', stg'')) = T(V_c, V_s, r, \pi, ord, stg)$ は，次の制約を満たす最小の監視ツリー:
  - $V_c = V_c' \cup V_c''$
  - $V_s = V_s' \cup V_s''$
  - $r = r'$
  - ある $s \in V_s'$ が存在して，$child(T', s) \cap V_s' = \emptyset$ ならば，$\pi(r'') = s$ かつ $ord(r'') = \max(\{ord'(v) \mid v \in child(T', s)\}) + 1$
  - 任意の $v \in V_c' \cup V_s' \setminus \{r'\}$ に対して，$\pi(v) = \pi'(v)$ かつ $ord(v) = ord'(v)$
  - 任意の $v \in V_c'' \cup V_s'' \setminus \{r''\}$ に対して，$\pi(v) = \pi''(v)$ かつ $ord(v) = ord''(v)$
  - 任意の $v \in V_s'$ に対して，$stg(v) = stg'(v)$
  - 任意の $v \in V_s''$ に対して，$stg(v) = stg''(v)$

4.2 アルゴリズム

提案アルゴリズムでは，vertex splitter と呼ばれる「良い」性質を持った頂点集合を求めることが重要である．連結な有向非巡回グラフ $G(V, E)$ に対する vertex splitter $S \subseteq V$ は，次の 2 つの制約を満たす頂点集合と定義する:

1. 任意の $s \in S$ と $v \in V$ に対して，$s \rightsquigarrow_G v$ ならば $v \in S$
2. $S \neq V$ ならば $|\mathit{components}(\mathit{subgraph}(G, V \setminus S))| \geq 2$

ただし，任意の有向グラフ $G(V, E)$ に対して，$\mathit{components}(G)$ は辺の向きを無視したときの連結成分の集合を表し，$\mathit{subgraph}(G, U \subseteq V)$ は頂点集合が U である G の誘導部分グラフを表す．

提案アルゴリズムは，前処理として，依存関係 DAG と呼ぶ，入力の依存関係グラフ G_d の強連結成分グラフ $G_D(V, E, w)$ を求める．ただし，$w\colon V \to \mathbb{N}, w(v) = |v|$ である．提案アルゴリズムは，G_D を入力として，1) まず，G_D の全ての極小の vertex splitter の集合 $\mathbb{S}$ を求め，各 $S \in \mathbb{S}$ に対して，2) $T.V_c = \mathit{flatten}(S)$ かつ $\mathit{group\_no\_change}(\mathit{subgraph}(G_D, S), T)$ を満たすような監視アカシア T を求め，3) $\mathit{subgraph}(G_D, V \setminus S)$ の各連結成分に対し再帰して複数の監視ツリーを得て，4) その全ての監視ツリーを子とする，`ofo` のスーパーバイザが根の監視ツリー T' を作り，5) T'' を $\mathit{merge}(T, T')$ とし，6) 各 S に対する T'' の中で最もコストが小さいものを出力とする．ただし，$\mathit{group\_no\_change}(G_D(V, E, w), T)$ は，「任意の $v \in V$ に対して，ある $s \in T.V_s$ が存在して，$\mathit{child}(T, s) \cap T.V_c = v$」を表す述語である．このアルゴリズムを図 4.1 に図示する．なお，上記の手順に加えて，提案アルゴリズムは G_D の頂点集合をキーとするメモ化を行う．

手順 1 において，依存関係 DAG G_D の全ての極小の vertex splitter を求める関数 `take_all_local_minimum_vertex_splitters`(G_D) は，$G_D.V$ のべき集合から極小の vertex splitter のみフィルターする関数と定義する．手順 2 において，極小の vertex splitter の列 s に対する監視アカシアを求める関数 `transform_into_acacia`(s) は，s がトポロジカルソートの逆順で並んでいると仮定して，アルゴリズム 7 で定義される．

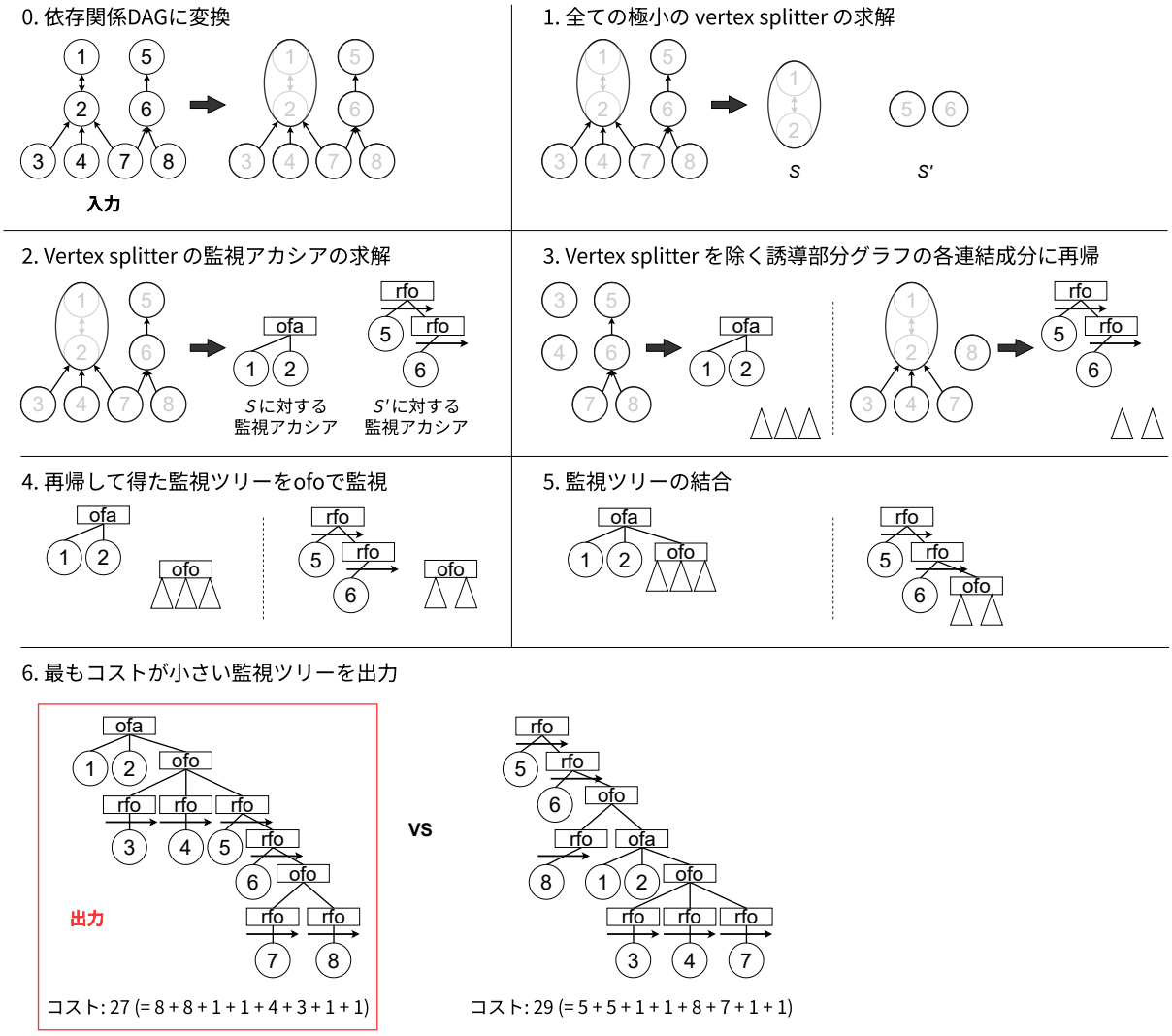


図 4.1: 提案アルゴリズムの概要

アルゴリズム 7 Vertex splitter の列を監視アカシアに変換する関数

```
1:  function transform_into_acacia(s)
2:      V_c ← s[1]
3:      r ← make_name()
4:      V_s ← {r}
5:      i ← 1
6:      for all v ∈ s[1] do
7:          π(v) ← r
8:          ord(v) ← i
9:          i ← i + 1
10:     if |s[1]| = 1 then
11:         stg(r) ← rfo
12:     else
13:         stg(r) ← ofa
14:     T ← (V_c, V_s, r, π, ord, stg)
15:     if |s| = 1 then
16:         return T
17:     else
18:         return merge(T, transform_into_acacia(s[2..|s|]))
```

アルゴリズム 8 連結な依存関係 DAG に対する最適な監視ツリーを求める関数

function $\texttt{transform}'(G_D(V, E, w), M)$
  if V をキーとする監視ツリー t が M に存在している **then**
    return t
  $\mathbb{T} \leftarrow \emptyset$
  for all $S \in \texttt{take\_all\_local\_minimum\_vertex\_splitters}(G_D)$ **do**
    $s \leftarrow S$ を G_D におけるトポロジカルソートの逆順で並べた列
    $T \leftarrow \texttt{transform\_into\_acacia}(s)$
    $\mathbb{T}' \leftarrow \emptyset$
    for all G'_D は $\mathit{subgraph}(G_D, V \setminus S)$ の各連結成分 **do**
      $\mathbb{T}' \leftarrow \mathbb{T}' \cup \texttt{transform}'(G'_D, M)$
    $T' \leftarrow \mathit{supervise\_with\_ofo}(\mathbb{T})$
    $\mathbb{T} \leftarrow \mathbb{T} \cup \mathit{merge}(T, T')$
  $T \leftarrow \mathbb{T}$ の中で最もコストが小さい監視ツリー
  V をキーとする T で M を更新
  return T

アルゴリズム 9 依存関係 DAG に対する最適な監視ツリーを求める関数

function $\texttt{transform}(G_D(V, E, w))$
  $M \leftarrow$ メモの初期状態
  if $|\mathit{components}(G_D)| = 1$ **then**
    return $\texttt{transform}'(G_D, M)$
  else
    $\mathbb{T} \leftarrow \emptyset$
    for all G'_D は G_D の各連結成分 **do**
      $\mathbb{T} \leftarrow \mathbb{T} \cup \texttt{transform}'(G'_D, M)$
    return $\mathit{supervise\_with\_ofo}(\mathbb{T})$

アルゴリズム 10 依存関係グラフに対する最適な監視ツリーを求める関数

```
1: function solve(G_d)
2:     G_D ← G_d に対する依存関係 DAG
3:     return transform(G_D)
```

以上の関数を用いて，連結な依存関係 DAG G_D とメモ M に対する最適な監視ツリーを求める関数 $\texttt{transform}'(G_D, M)$ は，アルゴリズム 8 で定義される．依存関係 DAG G_D に対する最適な監視ツリーを求める関数 $\texttt{transform}(G_D)$ はアルゴリズム 9 で定義され，依存関係グラフ G_d に対する最適な監視ツリーを求める関数 $\texttt{solve}(G_d)$ はアルゴリズム 10 で定義される．

4.3　正当性の証明

補題 4.3.1. 任意の連結な依存関係 *DAG* $G_D(V, E, w)$ と *vertex splitter* の列 s に対して，s が G_D におけるトポロジカルソートの逆順でソートされているならば，transform_into_acacia の出力は監視アカシアである．

証明. s の長さに関する帰納法で示す．ここで，A は次の制約を満たす監視ツリーとする:

- $A.V_c = \mathit{child}(A, A.r) = s[1]$
- もし $|s[1]| = 1$ ならば，$A.\mathit{stg}(A.r) = \texttt{rfo}$．そうでなければ，$A.\mathit{stg}(A.r) = \texttt{ofa}$

定義より，A は監視アカシアである．

$|s| = 1$ の場合

transform_into_acacia は A を出力するから，直ちに所望の結論を得る．

そうでない場合

G_D と $s[2..|s|]$ に transform_into_acacia を適用した結果を A' とする．A' に帰納法の仮定を適用して，A' は監視アカシアである．ここで，$A''(V_c, V_s, r, \pi, \mathit{ord}, \mathit{stg})$ は $\mathit{merge}(A', A'')$ とする．s は G_D におけるトポロジカルソートの逆順でソートされているから，任意の $v \in s[2..|s|]$ に対して，$s[1] \not\rightsquigarrow_{G_D} v$ である．また，A'' の定義より，任意の $v \in V_s$ に対して，$\mathit{stg}(v) \neq \texttt{ofo}$ である．よって，A'' は監視ツリーの制約を満たし，また監視アカシアである．transform_into_acacia は A'' を出力するから，所望の結論を満たす．

□

補題 4.3.2. 任意の連結な依存関係 *DAG* $G_D(V, E, w)$ に対して，`transform′` 関数の出力は監視ツリーである．

証明. メモされた監視ツリーが存在する場合

その監視ツリーを出力するから，直ちに所望の結論を満たす．

そうでない場合

S はある極小の vertex splitter とし，$V \setminus S$ の要素数に関する帰納法で示す．

$|V \setminus S| = 0$ の場合

全ての極小の vertex splitter の集合は $\{V\}$ であり，$subgraph(G_D, V \setminus S)$ は空グラフである．よって，`transform′` 関数の出力は，G_D と G_D におけるトポロジカルソートの逆順でソートした S の列に `transform_into_acacia` 関数を適用した結果である．補題 4.3.1 より，その結果は監視アカシアだから，所望の結論を満たす．

そうでない場合

A は，G_D と G_D におけるトポロジカルソートの逆順でソートした S の列に `transform_into_acacia` 関数を適用した結果とする．また，$\mathbb{T}$ は，$subgraph(G_D, V \setminus S)$ の各連結成分に対して `transform′` 関数を適用した結果とする．任意の $t \in \mathbb{T}$ に対して帰納法の仮定を適用して，t は監視ツリーである．さらに，T は $merge(A, supervise\_with\_ofo(\mathbb{T}))$ とする．

Vertex splitter の定義より，$reaching(S, G_D) \setminus S = V \setminus S$ だから，T は監視ツリーの制約を満たす．`transform′` 関数は，全ての極小の vertex splitter に対する T の中で，最もコストが小さい監視ツリーを出力するから，所望の結論を満たす．

□

補題 4.3.3. 任意の依存関係 *DAG* $G_D(V, E, w)$ に対して，`transform` 関数の出力は監視ツリーである．

証明. G_D に含まれる連結成分の数で場合分けして考える.

$|components(G_D)| = 1$ の場合

`transform` 関数は G_D に `transform′` 関数を適用した結果を直ちに出力する．その出力は補題 4.3.2 より監視ツリーだから，所望の結論を満たす.

そうでない場合

$\mathbb{T}$ は，G_D の各連結成分に対して `transform′` 関数を適用した結果とする．任意の $t \in \mathbb{T}$ に対して補題 4.3.2 を適用して，t は監視ツリーである．`transform` 関数は $supervise\_with\_ofo(\mathbb{T})$ を出力するから，所望の結論を得る.

□

定理 4.3.4. 任意の依存関係グラフ $G_d(V_g, E_g)$ に対して，`solve` 関数の出力は監視ツリーである.

証明. G_d に対する依存関係 DAG を $G_D(V, E, w)$ とする．`solve` 関数は G_D に `transform` 関数を適用した結果を出力するため，補題 4.3.3 より所望の結論を得る．□

4.4　最適性の証明

ここで，任意の依存関係 DAG $G_D(V, E, w)$ に対して，次の補助的な表記を定義する:

- 任意の $u \in \mathit{flatten}(V)$ に対して，$pack(G_D, u) = \{v \mid v \in V, u \in v\}$
- 任意の $U \subseteq \mathit{flatten}(V)$ に対して，$mappack(G_D, U) = \{pack(G_D, u) \mid u \in U\}$

補題 4.4.1. 任意の依存関係 DAG $G_D(V, E, w)$ と監視アカシア A, A' に対して，$group\_no\_change(G_D, A)$ ならば $cost(A) \leq cost(A')$.

証明. $A = (V_c, V_s, r, \pi, ord, stg)$ としたとき，V_c の分割の列 s は，次の制約を満たすとする:

- $\{v \mid v \in s\} = \{\{u\} \mid u \in V_c, stg(\pi(u)) = \texttt{rfo}\} \cup \{sibling(A,u) \cap V_c \mid u \in V_c, stg(\pi(u)) = \texttt{ofa}\}$
- 任意の $v, v' \in V_c$ に対して，$\pi(v) = \pi(v')$ かつ $stg(\pi(v)) = stg(\pi(v')) = \texttt{rfo}$ かつ $ord(v) < ord(v')$ ならば，ある $i, j \in 1..|s|$ が存在して，$i < j$ ならば，$s[i] = \{v\}$ かつ $s[j] = \{v'\}$
- 任意の $v, v' \in V_c$ と $\{u\} = sibling(A,v) \cap V_s$ なる u に対して，$follow(A,v) = \{u\}$ かつ $follow(A,v') \cup \{v'\} = child(A,u)$ ならば，ある $i \in 1..(|s|-1)$ が存在して，$s[i] = pack'(v)$ かつ $s[i+1] = pack'(v')$．ただし:
  - $pack'(v) = \begin{cases} \{v\} & stg(\pi(v)) = \texttt{rfo} \\ sibling(A,v) \cap V_c & \text{otherwise} \end{cases}$

同様にして，$A'.V_c$ の分割の列 s' を定義する．s, s' を用いて，$cost(A), cost(A')$ は次のように表せられる:

- $cost(A) = \sum_{i \in 1..|s|} \sum_{j \in i..|s|} |s[i]||s[j]|$
- $cost(A') = \sum_{i \in 1..|s'|} \sum_{j \in i..|s'|} |s'[i]||s'[j]|$

ここで，$group\_no\_change(G_D, A)$ だから，ある $1..|s|$ の分割 $\mathbb{I}$ が存在して，「$\{\bigcup_{i \in I} s[i] \mid I \in \mathbb{I}\} = \{s'[i] \mid i \in 1..|s'|\}$」である．以上より，

$$
\begin{aligned}
cost(A) &\leq cost(A') \\
\sum_{i \in 1..|s|} \sum_{j \in i..|s|} |s[i]||s[j]| &\leq \sum_{i \in 1..|s'|} \sum_{j \in i..|s'|} |s'[i]||s'[j]| \\
\sum_{\{i,j\} \in \{\{i,j\} \mid i,j \in 1..|s|\}} |s[i]||s[j]| &\leq \sum_{\{I,I'\} \in \{\{I,I'\} \mid I,I' \in \mathbb{I}\}} |\bigcup_{i \in I} s[i]||\bigcup_{j \in I'} s[j]| \\
0 &\leq \sum_{\{i,j\} \in \{\{i,j\} \mid I \in \mathbb{I}, i,j \in I, i \neq j\}} |s[i]||s[j]|
\end{aligned}
$$

が成り立つ．なぜなら，もし $group\_no\_change(G_D, A')$ の場合，任意の $I \in \mathbb{I}$ に対して $|I| = 1$ だから，$0 = \sum_{\{i,j\} \in \{\{i,j\} \mid I \in \mathbb{I}, i,j \in I, i \neq j\}} |s[i]||s[j]|$ である．また，そうでない場合，ある $I \in \mathbb{I}$ が存在して $|I| \geq 2$ だから，$0 < \sum_{\{i,j\} \in \{\{i,j\} \mid I \in \mathbb{I}, i,j \in I, i \neq j\}} |s[i]||s[j]|$ である．

□

補題 4.4.2. 任意の連結な依存関係 *DAG* $G_D(V, E, w)$ に対して，$|entrance(G_D)| = 1$ のとき，かつそのときに限り，任意の監視ツリー $T(flatten(V), V_s, r, \pi, ord, stg)$ は監視アカシア.

証明. **十分性（$\Longrightarrow$）**

T は監視アカシアでない標準形の監視ツリーと仮定すると，ある $s \in V_s$ が存在して，$stg(s) = \texttt{ofo}$ かつ $|child(T, s)| \geq 2$ である．よって，任意の $v \in entrance(G_D)$ に対して，ある $c \in child(T, s)$ が存在して，$v \in restart(T, c)$ かつ (任意の $c' \in child(T, s)$ に対して，$c' \neq c$ ならば $v \notin restart(T, c')$) が成り立つ．一方，任意の $u \in V$ に対して， $v \rightsquigarrow_G u$ だから，監視ツリーの制約を満たすためには $v \in restart(T, u)$ である必要があり，矛盾.

必要性（$\Longleftarrow$）

もし $|entrance(G_D)| > 1$ ならば，T は「ある $s \in V_s$ が存在して，$stg(s) = \texttt{ofo}$ かつ $entrance(G_D) = \{restart(T, c) \mid c \in child(T, s)\}$」を満たす場合がある．しかし，$T$ は監視アカシアだから ofo のスーパーバイザを含まず，矛盾.

□

補題 4.4.3. 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，T のコストが最小ならば，任意の $s \in V_s$ に対して，$stg(s) = \texttt{ofo}$ ならば，任意の $c \in child(T, s) \cap V_s$ に対して，$subtree(T, c)$ のコストは最小.

証明. A は $subacacia(T)$ とすると，T のコストは

$$cost(T) = cost(A) + |A.V_c||V_c \setminus A.V_c| + (|child(T, s) \cap V_c| + \sum_{c \in child(T,s) \cap V_s} cost(subtree(T, c)))$$

と表せられる．よって，もし T のコストが最小でなければ，ある $c \in child(T, r) \cap V_s$ と監視ツリー T' が存在して，T'' を $subtree(T, c)$ に束縛して，$T'.V_c = T''.V_c$ かつ $cost(T') < cost(T'')$ が成り立つ．そして，T において T'' を T' に差し替えることで，T のコストより小さい監視ツリーが作れる．しかし，T のコストは最小だから矛盾. □

補題 4.4.4. 任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$stg(r) = \texttt{ofo}$ ならば,（任意の $s \in child(T, r) \cap V_s$ に対して，$subtree(T, s)$ のコストは最小）のとき，かつそのときに限り，T のコストは最小.

証明. 十分性（$\Longrightarrow$）

T のコストは,

$$cost(T) = |child(T, r) \cap V_c| + \sum_{s \in child(T,r) \cap V_s} cost(subtree(T, s))$$

と表せられる．よって，もし T のコストが最小でないならば，ある $s \in child(T, r) \cap V_s$ と監視ツリー T' が存在して，T'' を $subtree(T, s)$ に束縛して，$T'.V_c = T''.V_c$ かつ $cost(T') < cost(T'')$ が成り立つ．しかし，T'' のコストは最小だから矛盾.

必要性（$\Longleftarrow$）

補題 4.4.3 より直ちに成り立つ.

□

補題 4.4.5. 任意の連結な依存関係 *DAG* $G_D(V, E, w)$ と *Vertex splitter* $S \subseteq V$ に対して，$V = S$ のとき，かつそのときに限り，$|entrance(G_D)| = 1$.

証明. 十分性（$\Longrightarrow$）

もし $|entrance(G_D)| > 1$ ならば，少なくとも $V \setminus entrance(G_D)$ は Vertex splitter である．しかし，これは前提の $V = S$ に矛盾する.

必要性（$\Longleftarrow$）

$entrance(G_D) = \{v\}$ なる頂点 v が S に含まれているかどうかで場合分けして考える.

$v \in S$ の場合

$reachable(\{v\}, G_D) = V$ だから，直ちに $V = S$ である.

$v \notin S$ の場合

$V \neq S$ だから，$G'_D(V', E', w')(= subgraph(G_D, V \setminus S))$ は以下の性質を満たす:

- ある V' の分割 $\mathbb{U}$ が存在して，$|\mathbb{U}| \geq 2$ かつ（任意の $U, U' \in \mathbb{U}$ と $u \in U$ と $u' \in U'$ に対して，$U \neq U'$ ならば $u \not\rightsquigarrow_{G'_D} u'$）

しかし，任意の $U, U \in \mathbb{U}$ と $u \in U$ と $u' \in U'$ に対して，$v \rightsquigarrow_{G'_D} u$ かつ $v \rightsquigarrow_{G'_D} u'$ だから，G'_D は連結であり，矛盾.

□

補題 4.4.6. 任意の連結な依存関係 *DAG* $G_D(V, E, w)$ に対して，$V_c = \mathit{flatten}(V)$ とすると，任意の監視ツリー $T(V_c, V_s, r, \pi, \mathit{ord}, \mathit{stg})$ に対して，T のコストが最小ならば，ある *Vertex Splitter* $S \subseteq V$ と $s \in V_s$ が存在して，$G'_D = \mathit{subgraph}(G_D, V \setminus S)$ とし，$\mathbb{U} = \mathit{components}(G'_D)$ とすると，$\mathit{stg}(s) = \texttt{ofo}$ かつ（任意の $s' \in \mathit{ancestor}(T, s)$ に対して，$\mathit{stg}(s') \neq \texttt{ofo}$）ならば，($\mathit{flatten}(S) = \{v \mid v \in V_c, s' \in \mathit{ancestor}(T, v), \mathit{stg}(s') \neq \texttt{ofo}\}$) かつ（$\{\mathit{flatten}(U) \mid U \in \mathbb{U}\} = \{\mathit{restart}(T, c) \mid c \in \mathit{child}(T, s)\}$）かつ（ある $v \in \mathit{exit}(G_D)$ が存在して，S は v を含む極小の *Vertex splitter*）.

証明. $V \setminus S$ の要素数に関する帰納法で示す.

$|V \setminus S| = 0$ の場合

補題 4.4.5 と補題 4.4.2 より，T は監視アカシアだから前提を満たす s は存在せず，直ちに成り立つ.

そうでない場合　まず，$\mathit{flatten}(S) = \{v \mid v \in V_c, s' \in \mathit{ancestor}(T, v), \mathit{stg}(s') \neq \texttt{ofo}\}$ であることを示す.

S' は $\{\mathit{pack}(G_D, v) \mid v \in V_c, s' \in \mathit{ancestor}(T, v), \mathit{stg}(s') \neq \texttt{ofo}\}$ とする．ここで，監視ツリーの定義より，任意の $U, U' \in \{\mathit{restart}(T, c) \mid c \in \mathit{child}(T, s)\}$ と $u \in U$ と $u' \in U'$ に対して，$U \neq U'$ ならば，$\mathit{pack}(G_D, u) \not\rightsquigarrow_{G_D} \mathit{pack}(G_D, u')$ が成り立つ．よって，S' は Vertex splitter の性質 2 を満たす．もし S' が Vertex splitter の性質 1 を満たさない場合，ある $v \in V_c$ と $u \in S'$ が存在して，$\mathit{pack}(G_D, v) \notin S'$ かつ $u \rightsquigarrow_{G_D} \mathit{pack}(G_D, v)$ が成り立つ．しかし，S' の定義より，$S' \not\supseteq \mathit{restart}(T, v)$ だから，監視ツリーの制約に違反し，矛盾.

次に，$\{\mathit{flatten}(U) \mid U \in \mathbb{U}\} = \{\mathit{restart}(T, c) \mid c \in \mathit{child}(T, s)\}$ であることを示す.

所望の結論を示すためには，「U」と同じ性質を「$\mathit{restart}(T, c)$」が持つことを示せば

よい．まず，$stg(s) = \texttt{ofo}$ だから，「任意の $c, c' \in child(T, s)$ と $v \in restart(T, c)$ と $v' \in restart(T, c')$ に対して，$c \neq c'$ ならば，$pack(G'_D, v) \not\rightsquigarrow_{G'_D} pack(G'_D, v')$」である．

次に，「任意の $c \in child(T, s)$ と $v, v' \in restart(T, c)$ に対して，$pack(G'_D, v) \rightsquigarrow_{G'_D} pack(G'_D, v')$ または $pack(G'_D, v') \rightsquigarrow_{G'_D} pack(G'_D, v)$」が成り立たないと仮定すると，ある $\mathbb{U}$ の分割 $\mathbf{U}$ が存在して，「$\{\bigcup_{U \in \mathbb{U}'} U \mid \mathbb{U}' \in \mathbf{U}\} = \{restart(T, c) \mid c \in child(T, s)\}$」が成り立つ．ここで，各変数を次のように束縛する:

- $\mathbb{T}$ は $\{subtree(T, c) \mid c \in child(T, s)\}$
- $\mathbb{T}'$ は $\{(flatten(U), V_s, r, \pi, ord, stg) \mid U \in \mathbb{U}, \exists V_s, r, \pi, ord, stg\}$

補題 4.4.3 より，任意の $t \in \mathbb{T}$ に対して t のコストは最小であり，補題 4.4.4 より，$subtree(T, s)$ のコストもまた最小である．以降では，$cost(supervise\_with\_ofo(\mathbb{T}')) < cost(subtree(T, s))$ を示し，矛盾が起こることを確認する．任意の $t \in \mathbb{T}$ を束縛し，$G''_D(V'', E'')$ を $subgraph(G'_D, mappack(G'_D, t.V_c))$ とする．もし t が監視アカシアならば，補題 4.4.2 と補題 4.4.5 より，V'' は極小の Vertex splitter である．そうでなければ，G''_D と t に帰納法の仮定を適用して，「$mappack(G''_D, subacacia(t).V_c)$ は極小の Vertex splitter」と「$\{flatten(U) \mid U \in \mathbb{U}''\} = \{restart(t, c) \mid c \in child(t, s'')\}$」を得る．ただし，$\mathbb{U}''$ は $components(subgraph(G''_D, V'' \setminus mappack(G''_D, subacacia(t).V_c)))$ であり，$s'' \in t.V_s$ は $t.stg(s'') = \texttt{ofo}$ かつ任意の $v \in ancestor(t, s'')$ に対して $t.stg(v) \neq \texttt{ofo}$ である．任意の $t' \in \mathbb{T}'$ に対しても，同様の操作で帰納法の仮定を適用できる．ここで，$\mathbb{T}'$ の定義より，ある $t' \in \mathbb{T}'$ が存在して，$subacacia(t).V_c = subacacia(t').V_c$ である．また，ある $\mathbb{T}'' \subset \mathbb{T}'$ と $s'' \in t'.V_s$ が存在して，$t'.stg(s'') = \texttt{ofo}$ かつ任意の $v \in ancestor(t', s'')$ に対して $t'.stg(v) \neq \texttt{ofo}$ ならば，$\{subtree(t, c) \mid c \in child(t, s'')\} = (\mathbb{T}'' \setminus \{t'\}) \cup subtree(t', s'')$ である．

よって，t のコストは

$$cost(t) = \sum_{t'' \in \mathbb{T}''} cost(t'') + \sum_{u \in mappack(G''_D, subacacia(t).V_c)} \sum_{u' \in U'} w(u)w(u')$$

$$(\text{ただし，} U' = \{u' \in mappack(G''_D, t.V_c) \mid u' \not\rightsquigarrow_{subgraph(G''_D, mappack(G''_D, t.V_c))} u\})$$

と表される．したがって，

$$0 < cost(subtree(T, s)) - cost(supervise\_with\_ofo(\mathbb{T}'))$$

$$0 < \sum_{t \in \mathbb{T}} cost(t) - \sum_{t' \in \mathbb{T}'} cost(t')$$

$$0 < \sum_{t \in \mathbb{T}} cost(t) - \sum_{t' \in \mathbb{T}'} cost(t')$$

$$0 < \sum_{t \in \mathbb{T}} \left(\sum_{u \in mappack(G''_D, subacacia(t).V_c)} \sum_{u' \in U'} w(u)w(u') \right)$$

$$(\text{ただし，} U' = \{u' \in mappack(G''_D, t.V_c) \mid u' \not\rightsquigarrow_{subgraph(G''_D, mappack(G''_D, t.V_c))} u\})$$

である．$|\mathbb{T}| \geq 2$ かつ（ある $t \in \mathbb{T}$ と $\mathbb{U}' \in \mathbf{U}$ が存在して，$t.V_c = \bigcup_{U \in \mathbb{U}'} U$ かつ（任意の $U \in \mathbb{U}'$ に対して，$|U| \geq 1$））だから，上記の不等式は成り立つ．

最後に，ある $v \in exit(G_D)$ が存在して，S は v を含む極小の Vertex splitter であることを示す．

もし S が極小でないならば，ある $x \in S$ が存在して，$S \setminus \{x\}$ もまた Vertex splitter である．ここで，準備として，次のように各変数を束縛する:

- S' は $S \setminus \{x\}$
- T' は「$flatten(S') = \{v \mid v \in T'.V_c, s' \in ancestor(T', v), T'.stg(s') \neq \texttt{ofo}\}$」を満たす，ある監視ツリー
- $s' \in T'.V_s$ は「（任意の $s'' \in ancestor(T', s')$ に対して，$T'.stg(s'') \neq \texttt{ofo}$）かつ $T'.stg(s') = \texttt{ofo}$」を満たす
- $\mathbb{U}'$ は「$\{flatten(U') \mid U' \in \mathbb{U}'\} = \{restart(T', c) \mid c \in child(T', s')\}$」を満たす $V \setminus S'$ の分割

- A は $subacacia(T)$
- A' は $subacacia(T')$
- $\mathbb{J} \subset \mathbb{U}$ と $U' \in \mathbb{U}'$ は「$(\bigcup_{U \in \mathbb{J}} U) \cup \{x\} = U'$」を満たす族と頂点集合
- $\mathbb{T}$ は $\{subtree(T, c) \mid c \in child(T, s), U \in \mathbb{J}, restart(T, c) = U\}$

以降では，$cost(T') < cost(T)$ を示し，矛盾が起こることを確認する．T と T' のコストはそれぞれ「$cost(T) = cost(A) + \sum_{u \in S} \sum_{u' \in V \setminus S} w(u)w(u') + \sum_{c \in child(T,s)} cost(subtree(T, c))$」,「$cost(T') = cost(A') + \sum_{u \in S'} \sum_{u' \in V \setminus S'} w(u)w(u') + \sum_{c' \in child(T',s')} cost(subtree(T', c'))$」と表せられる．よって，

$$cost(A) - cost(A') = w(x) \sum_{u \in S} w(u)$$

$$\begin{aligned}
&\sum_{u \in S} \sum_{u' \in V \setminus S} w(u)w(u') - \sum_{u \in S'} \sum_{u' \in V \setminus S'} w(u)w(u') \\
&= \sum_{u \in S' \cup \{x\}} \sum_{u' \in V \setminus (S' \cup \{x\})} w(u)w(u') - \sum_{u \in S'} \sum_{u' \in V \setminus S'} w(u)w(u') \\
&= \left(\sum_{u \in S'} \sum_{u' \in V \setminus S'} w(u)w(u') - w(x) \sum_{u \in S'} w(u) + w(x) \sum_{u' \in V \setminus S'} w(u') - w(x)^2 \right) - \\
&\quad \sum_{u \in S'} \sum_{u' \in V \setminus S'} w(u)w(u') \\
&= -w(x) \sum_{u \in S'} w(u) + w(x) \sum_{u' \in V \setminus S'} w(u') - w(x)^2
\end{aligned}$$

$$\begin{aligned}
&\sum_{c \in child(T,s)} cost(subtree(T, c)) - \sum_{c' \in child(T',s')} cost(subtree(T', c')) \\
&= \sum_{t \in \mathbb{T}} cost(t) - cost(subtree(T', c')) \qquad (\text{ただし，} restart(T', c' \in child(T', s')) = U') \\
&= \sum_{t \in \mathbb{T}} cost(t) - \left(w(x)^2 + w(x) \sum_{u' \in U' \setminus \{x\}} w(u') + \sum_{t \in \mathbb{T}} cost(t) \right) \\
&\qquad (subgraph(G'_D, U') \text{ と } subtree(T', c') \text{ と Vertex splitter } \{x\} \text{ に帰納法の仮定を適用して}) \\
&= -w(x) \sum_{u' \in U'} w(u')
\end{aligned}$$

となる．以上より，

$$
\begin{aligned}
&cost(T') < cost(T) \\
&\quad 0 < \left(w(x)\sum_{u\in S} w(u)\right) + \left(-w(x)\sum_{u\in S'} w(u) + w(x)\sum_{u'\in V\setminus S'} w(u') - w(x)^2\right) + \\
&\qquad \left(-w(x)\sum_{u'\in U'} w(u')\right) \\
&\quad 0 < \sum_{u\in S} w(u) - \sum_{u\in S'} w(u) + \sum_{u'\in V\setminus S'} w(u') - w(x) - \sum_{u'\in U'} w(u') \qquad (w(x) > 0 \text{ より}) \\
&\quad 0 < w(x) + \sum_{u'\in V\setminus S'} w(u') - w(x) - \sum_{u'\in U'} w(u') \\
&\quad 0 < \sum_{u'\in V\setminus S'} w(u') - \sum_{u'\in U'} w(u')
\end{aligned}
$$

である．$U' \subset V \setminus S'$ だから，この不等式は成り立つ．

□

補題 4.4.7. 任意の連結な依存関係 DAG $G_D(V, E, w)$ に対して，`transform'` が出力する監視ツリー T はコストが最小．

証明. G_D に含まれる入口頂点の数で場合分けして考える．

$|entrance(G_D)| = 1$ の場合

補題 4.4.2 より T は監視アカシアであり，$transform'$ 関数の定義より $group\_no\_change(G_D, T)$ だから，補題 4.4.1 より T のコストは最小である．

そうでない場合

$\mathbb{S}$ を `take_all_local_minimum_vertex_splitters`(G_D) に束縛する．コストが最小の監視ツリーは，ある $S \in \mathbb{S}$ に対して，補題 4.4.6 を満たす．$transform'$ 関数は，全ての $S \in \mathbb{S}$ に対して，補題 4.4.6 を満たす監視ツリーを求め，その中から最もコストが小さい監視ツリー T を出力する．したがって，T のコストは最小である．

□

補題 4.4.8. 任意の依存関係 *DAG* $G_D(V, E, w)$ に対して，*transform* が出力する監視ツリー T はコストが最小.

証明. 任意の $G'_D \in \{subgraph(G_D, U) \mid U \in components(G_D)\}$ に対して，G'_D に補題 4.4.7 を適用して，コストが最小の監視ツリー T' を得る．ここで，全ての T' を含む族を $\mathbb{T}$ とする．$\mathbb{T} = \{T\}$ ならば，直ちに T のコストは最小である．そうでない場合，$T = supervise\_with\_ofo(\mathbb{T})$ だから，補題 4.4.4 より T のコストは最小である． □

定理 4.4.9. 任意の依存関係グラフ $G_d(V_g, E_g)$ に対して，`solve` が出力する監視ツリー T はコストが最小.

証明. G_d の依存関係 DAG を $G_D(V, E, w)$ とする．G_D に補題 4.4.8 を適用して，直ちに所望の結論を得る． □

4.5　計算量の解析

簡単のため，`solve` 関数に入力されるグラフは連結な DAG と仮定して解析する．DAG でない場合は依存関係 DAG に変換すると頂点数が少なくなり，連結でない場合は各連結な部分グラフの頂点数が少なくなるから，この仮定を置いた解析は最も悲観的な場合の解析となる.

`transform′` 関数に入力される依存関係 DAG の頂点数を V とする．`take_all_local_minimum_vertex_splitters` 関数は，全ての頂点集合の部分集合から極小の vertex splitter をフィルターするから，計算量は $\mathcal{O}(2^V)$ である．`transform_into_acacia` 関数は，入力された列を先頭から順に監視ツリーへと変換し，列の各要素の監視ツリーへの変換は定数時間で計算可能だから，関数全体は線形時間で停止する．*merge* 関数の計算量もまた線形時間である．`transform′` 関数ではメモ化を用いており，メモのキーはグラフの頂点集合だから，メモのキーの最大数は 2^V である．`transform_into_acacia` 関数と *merge* 関数はどちらも線形時間で停止し，`take_all_local_minimum_vertex_splitters` 関数の計算量

は $\mathcal{O}(2^V)$ だから，`transform'` 関数の計算量は $\mathcal{O}(2^{2V})$ である．仮定より，`solve` 関数と `transform` 関数の計算量も同様である．

第 5 章

アルゴリズムの改善

4 章で述べたアルゴリズムでは全ての極小の vertex splitter の集合を求めるが，これを多項式時間で解くアルゴリズムはまだ発見されていない．しかし，最適監視ツリー問題を解く場合は，各極小 vertex splitter S ではなく vertex splitter の制約 1 を満たす空でない S の部分集合（の任意の一つ）を求めれば十分である．本章では，上記のアイディアを用いるアルゴリズムを説明し，その正当性・最適性の証明，計算量の解析を行う．

5.1 アルゴリズム

連結な依存関係 DAG G_D に対して，各極小 vertex splitter S ではなく vertex splitter の制約 1 を満たす空でない S の部分集合（の任意の一つ）を求める関数 `take_all_local_minimum_vertex_splitters`$'(G_D)$ は，アルゴリズム 11 で定義される．

アルゴリズム 11 各極小 vertex splitter S ではなく vertex splitter の制約 1 を満たす空でない の部分集合（の任意の一つ）を求める関数

1: **function** `take_all_local_minimum_vertex_splitters`$'(G_D(V, E, w))$

2: **return** $\left\{ \bigcap_{u \in \{u \mid u \in entrance(G_D), u \rightsquigarrow_G v\}} reachable(\{u\}, G_D) \mid v \in exit(G_D) \right\}$

連結な依存関係 DAG G_D に対する最適な監視ツリーを求める関数 `transform`$''(G_D)$ は，`take_all_local_minimum_vertex_splitters` 関数の代わりに `take_all_local_minimum_vertex_splitters` 関数を用いる `transform`$'$ 関数である．また，`transform` 関数は `transform`$'$ 関数の代わりに `transform`$''$ 関数を用いるものとする．

5.2 正当性の証明

補題 5.2.1. 任意の連結な依存関係 DAG $G_D(V, E, w)$ に対して，$\mathbb{S}$ を全ての極小の *vertex splitter* の集合，$\mathbb{S}'$ を $\texttt{take\_all\_local\_minimum\_vertex\_splitters}'(G_D)$ とすると，$\mathbb{S} = \{S \cup V' \mid S \in \mathbb{S}', V' \subset V\}$.

証明. Vertex splitter の定義より，任意の極小の vertex splitter S に対して，ある $E \subseteq \mathit{entrance}(G_D)$ が存在して，$\cap_{e \in E} \mathit{reachable}(\{e\}, G_D)$ である．よって，任意の極小の vertex splitter S に対して，ある $e \in \mathit{exit}(G_D)$ が存在して，E を $\{e' \in \mathit{entrance}(G_D) \mid e' \rightsquigarrow_{G_D} e\}$ とすると，$\cap_{e \in E} \mathit{reachable}(\{e\}, G_D) \subseteq S$ である．したがって，$\texttt{take\_all\_local\_minimum\_vertex\_splitters}'(G_D)$ の定義より，所望の結論は成り立つ. □

補題 5.2.2. 任意の連結な依存関係 DAG $G_D(V, E, w)$ に対して，任意の $S \in \texttt{take\_all\_local\_minimum\_vertex\_splitters}'(G_D)$ と $v \in S$ に対して，$v \neq \emptyset$.

証明. アルゴリズムは，出口頂点に到達可能な入口頂点から到達可能な頂点集合の積集合を取るから，必ずその出口頂点を含む．したがって，所望の結論は成り立つ. □

補題 5.2.3. 任意の連結な依存関係 DAG $G_D(V, E, w)$ に対して，$|\mathit{entrance}(G_D)| = 1$ ならば，$\texttt{take\_all\_local\_minimum\_vertex\_splitters}'(G_D) = \{V\}$.

証明. アルゴリズムは，出口頂点に到達可能な入口頂点から到達可能な頂点集合の積集合を取るから，入口頂点が一つの場合の結果は V となる．したがって，所望の結論は成り立つ. □

補題 5.2.4. 任意の連結な依存関係 DAG $G_D(V, E, w)$ に対して，$\texttt{transform}''$ 関数の出力は監視ツリーである.

証明. **メモされた監視ツリーが存在する場合**

その監視ツリーを出力するから，直ちに所望の結論を満たす.

そうでない場合

$\mathbb{S} = \texttt{take\_all\_local\_minimum\_vertex\_splitters}'(G_D)$ とする．$\mathbb{S}$ は補題 5.2.1, 5.2.2, 5.2.3 を満たす．$S \in \mathbb{S}$ を束縛し，$V \setminus S$ の要素数に関する帰納法で示す．

$|V \setminus S| = 0$ の場合

$\mathbb{S} = \{V\}$ であり，$subgraph(G_D, V \setminus S)$ は空グラフである．よって，$\texttt{transform}''$ 関数の出力は，G_D と G_D におけるトポロジカルソートの逆順でソートした S の列に `transform_into_acacia` 関数を適用した結果である．補題 4.3.1 より，その結果は監視アカシアだから，所望の結論を満たす．

そうでない場合

A は，G_D と G_D におけるトポロジカルソートの逆順でソートした S の列に `transform_into_acacia` 関数を適用した結果とする．$S \neq \emptyset$ だからこの適用は可能であり，補題 4.3.1 より A は監視アカシアである．また，$\mathbb{T}$ は，$subgraph(G_D, V \setminus S)$ の各連結成分に対して $\texttt{transform}''$ 関数を適用した結果とする．$S \neq \emptyset$ だから，任意の $t \in \mathbb{T}$ に対して帰納法の仮定を適用して，t は監視ツリーである．さらに，T は $merge(A, supervise\_with\_ofo(\mathbb{T}))$ とする．

Vertex splitter の定義より，$reaching(S, G_D) \setminus S = V \setminus S$ だから，T は監視ツリーの制約を満たす．$\texttt{transform}''$ 関数は，全ての極小の vertex splitter に対する T の中で，最もコストが小さい監視ツリーを出力するから，所望の結論を満たす．

□

5.3 最適性の証明

ここで，任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$subtree'(T, v \in V_c) = T'(V_c', V_s', r', \pi', ord', stg')$ は，次の制約を満たす最小の監視ツリーと定義する:

- $V_c' = restart(T, v)$
- $V_s' = \{\pi(u) \mid u \in V_c'\}$

- $r' = \pi(v)$
- 任意の $v \in V_c' \cup V_s' \setminus \{r'\}$ に対して，$\pi'(v) = \pi(v)$ かつ $ord'(v) = ord(v)$
- 任意の $v \in V_s'$ に対して，$stg'(v) = stg(v)$

補題 5.3.1. 任意の連結な依存関係 $DAG\ G_D(V, E, w)$ に対して，$V_c = \mathit{flatten}(V)$ とすると，任意の監視ツリー $T(V_c, V_s, r, \pi, ord, stg)$ に対して，$A = \mathit{subacacia}(T)$ とすると，任意の $v \in A.V_c$ に対して，$T' = \mathit{subtree}'(T, v)$ とすると，T のコストが最小ならば T' のコストは最小.

証明. T' のコストが最小ではないと仮定すると，T' よりコストが小さい監視ツリーで T における T' を差し替えることで，T よりコストが小さい監視ツリーが作れる．しかし，T のコストは最小なので矛盾. □

補題 5.3.2. 任意の連結な依存関係 $DAG\ G_D(V, E, w)$ に対して，$V_c = \mathit{flatten}(V)$ とすると，任意の監視ツリーの集合 $\mathbb{T}$ に対して,（任意の $T \in \mathbb{T}$ に対して，$T.V_c = V_c$ かつ T のコストは最小）ならば，G_D における任意の極小の *vertex splitter* の部分集合 X に対して，$X \neq \emptyset$ かつ X は *vertex splitter* の制約 *1* を満たし，かつ（ある $T \in \mathbb{T}$ が存在して，$X \subset \mathit{extract\_vertex\_splitter}(T, G_D)$）ならば，$G_D'(V', E', w') = \mathit{subgraph}(G_D, V \setminus X)$ とし，$V_c' = \mathit{flatten}(V')$ とすると，任意の監視ツリーの集合 $\mathbb{T}'$ に対して,（任意の $T' \in \mathbb{T}'$ に対して，$T'.V_c = V_c'$ かつ T' のコストは最小）ならば，任意の $T' \in \mathbb{T}'$ に対して，ある $T \in \mathbb{T}$ が存在して，$\mathit{extract\_vertex\_splitter}(T', G_D') \cup X = \mathit{extract\_vertex\_splitter}(T, G_D)$.

証明. A'' は $A''.V_c = \mathit{flatten}(X)$ を満たす，$\mathit{subgraph}(G_D, X)$ においてコストが最小の監視アカシアとする．また，任意の $T' \in \mathbb{T}'$ に対して，T'' は $\mathit{merge}(A'', T')$ とする．ここで，ある $t \in \mathbb{T}$ と $t' \in \mathbb{T}'$ が存在して，補題 5.3.1 より $\mathit{merge}(A'', t') = t$ である．よって，$\mathit{cost}(t') = \mathit{cost}(T')$ かつ $t'.V_c = T'.V_c$ だから，T'' のコストは最小である．さらに，補題 4.4.6 より，$\mathit{extract\_vertex\_splitter}(T'', G_D)$ は極小の vertex splitter S であり，$\mathit{extract\_vertex\_splitter}(T', G_D')$ は極小の vertex splitter S' である．T'' の定義より $S = S' \cup X$ を満たし，また，$T''.V_c = V_c$ かつ T'' のコストは最小だから $T'' \in \mathbb{T}$ であ

る．したがって，$extract\_vertex\_splitter(T', G'_D) \cup X = extract\_vertex\_splitter(T, G_D)$ が成り立つ． □

補題 5.3.3. 任意の連結な依存関係 $DAG\ G_D(V, E, w)$ に対して，$transform''$ が出力する監視ツリー T はコストが最小．

証明. $\mathbb{S} =$ `take_all_local_minimum_vertex_splitters`$'(G_D)$ とする．$\mathbb{S}$ は補題 5.2.1, 5.2.2, 5.2.3 を満たす．G_D に含まれる入口頂点の数で場合分けして考える．

$|entrance(G_D)| = 1$ の場合

補題 5.2.3 より $\mathbb{S} = \{V\}$ だから，T は監視アカシアである．$transform''$ 関数の定義より $group\_no\_change(G_D, T)$ だから，補題 4.4.1 より T のコストは最小である．

そうでない場合

コストが最小の監視ツリーは，ある $S \in \mathbb{S}$ に対して，S が極小の vertex splitter の場合は補題 4.4.6 を満たし，極小の vertex splitter の部分集合の場合は補題 5.3.2 を満たす．$transform''$ 関数は，全ての $S \in \mathbb{S}$ に対して，上記の補題を満たす監視ツリーを求め，その中から最もコストが小さい監視ツリー T を出力する．したがって，T のコストは最小である．

□

5.4 計算量の解析

4.5 節と同様に，簡単のため，`solve` 関数に入力されるグラフは連結な DAG と仮定して解析する．また，入力グラフの頂点数を V とする．

`take_all_local_minimum_vertex_splitters`$'$ 関数は，各出口頂点 e に対して，e に到達可能な入口頂点から到達可能な頂点集合の積集合を取る．よって，荒く見積もっても計算量は $\mathcal{O}(V^2)$ である．したがって，`transform`$'$ 関数の計算量は，$\mathcal{O}(V^2 2^V)$ である．

第 6 章

評価

5 章で改善したアルゴリズムは，解析的には計算に指数時間かかる．実際，図 6.1 のような病的なケースが入力されたとき，提案アルゴリズムは指数時間かかる．しかし，取りうるメモのキーは，空でない vertex splitter の性質 1 を満たす頂点集合を，グラフの頂点集合から削除して得られる頂点集合に限られるから，メモのキーの集合が頂点集合のべき集合と等しい場合は多くはないと予想される．そこで，本章では，提案アルゴリズムは病的なケースに対しては指数時間かかったとしても，現実的な多くの場合のグラフに対しては十分高速で動作することを示すための実験を行う．

6.1 アルゴリズムの実行時間

実験では，ランダムに生成したグラフを提案アルゴリズムに入力して実行時間を計測する．なお，そのグラフは頂点数と辺数を変動させ，頂点数は $10, 20, \ldots, 70$ と変動し，辺数は頂点数を 1, 2, 3 倍した値である．また，各変数の組み合わせに対して 100 回試行した．

実験結果を図 6.2 に示す．なお，実験環境は 3.2 節で述べた実験と同じである．図の横軸はグラフの頂点数，すなわち gen_server の数を表し，縦軸はアルゴリズムの実行時間を表

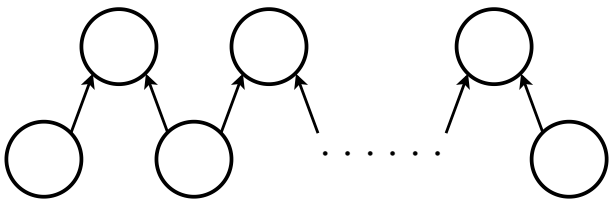
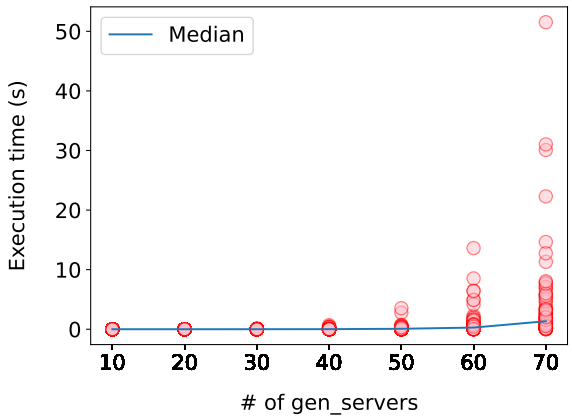
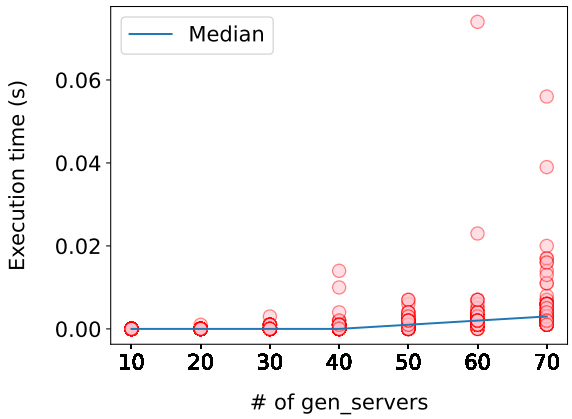


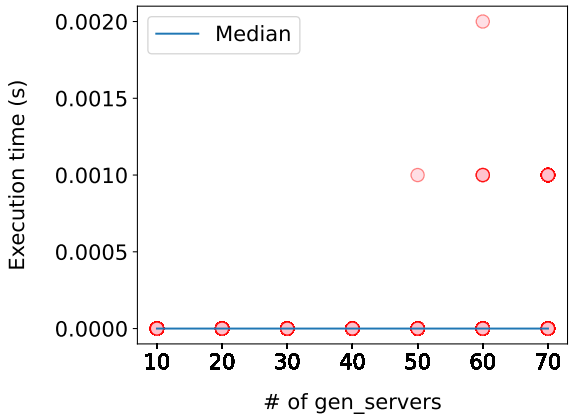
図 6.1: 病的なケース



(a) 辺数が頂点数と等しい場合



(b) 辺数が頂点数の 2 倍の場合



(c) 辺数が頂点数の 3 倍の場合

図 6.2: 提案アルゴリズムの実行時間

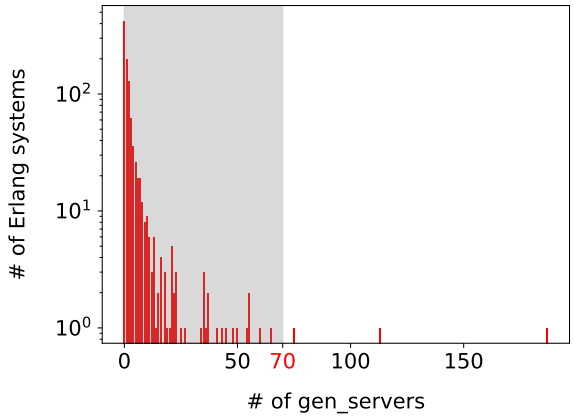


図 6.3: 一般的な Erlang システムに含まれる gen_server の数

す．この実験より，$V = 70$ の場合でも，実行時間の中央値は 1.3415 秒であった．なお，辺数が多いほど実行時間が短くなるのは，G_d の強連結成分が増えて G_D の頂点数が少なくなり，実行時間はより短くなるからであると考える．

ここで，一般的な Erlang システムに含まれる gen_server の数が 70 以下であれば，提案アルゴリズムは現実的な多くの場合において十分高速に動作すると考えられる．そこで，2024 年 1 月 19 日時点で GitHub に公開されている，スターが多い順の 999 個の Erlang システムを対象に，1 つのシステムに含まれる gen_server の数を調査した．なお，Erlang システムとは使用率が最も高い言語が Erlang であるリポジトリを指し，gen_server とは「`-behaviour(gen_server).`」が記述されているファイルとした．また，この条件では Erlang の言語処理系を管理するリポジトリ [*1] も該当するが，これは Erlang システムとは異なるため調査対象から除外した [*2]．

上記の実験結果を図 6.3 に示す．図の横軸は gen_server の数を表し，縦軸は Erlang システムの数を表す．この結果より，gen_server の数が 70 個以下のシステムは約 99.6%であることがわかった．

[*1] `https://github.com/erlang/otp`

[*2] 調査プログラムでは，条件を満たすリポジトリの検索に GitHub CLI を用いた．調査対象の数が 999 個である理由は，GitHub CLI で一度に検索できるリポジトリ数の上限が 1000 であり，ここから Erlang の言語処理系のリポジトリを除外しているからである．

以上の実験より，提案アルゴリズムは現実的な多くの場合において十分高速に動作すると考える．

第 7 章

関連研究

Neykova ら [3] は，プロセス間の通信の向きや順序を表現可能な型システム Multiparty session types を利用し，従来の監視ツリーとは異なる仕組みで最小限のプロセスのみ再起動する手法を提案した．本研究において，プロセス間の依存関係をプロセス間の通信の向きと定義している点は，この研究内容を踏襲している．異なる点としては，1.1 節で述べた通り，従来の Erlang システムへの適用の難易度である．先行研究では，プロセスを実装するために専用のコールバック関数を実装する必要や，プロセス間の通信には Erlang のプリミティブとは異なる専用の関数を用いる必要があり，従来の Erlang システムへの適用が難しい．一方で，本研究の提案手法は，プロセス間の依存関係の抽出を今後改良することで，従来の Erlang システムにそのまま適用できると期待される．また，提案手法はプロセスの再起動法として従来の監視ツリーを採用しているため，システム運用上の利点があると考える．提案手法は先行研究と異なりランタイムでは動かず，あくまで監視ツリーを生成するだけである．そのため，もし提案手法の実装が保守されなくなったとしても，Erlang 本体が保守されている限りユーザのシステム自体に影響を及ばさない．再起動法はシステムの根幹に関わるため，言語の標準機能やサポートが強力なライブラリ以外に依存したくないと考える開発者は多いのではないかと考えている．

Nyström ら [2] は，監視ツリーが「ロバスト」であることを保証する手法を提案した．この手法におけるロバスト性とは，軽量プロセス間にリンクが存在することや，スーパーバイザの終了時間の上限より再起動時間の上限が短い，といった性質である．本研究との違いとしては，この研究はユーザがロバストな監視ツリーを作成することを意図している点である．監視ツリーがロバスト性を満たすために重要な，スーパーバイザの終了時間・再起動時

間の上限は，そのスーパーバイザの子プロセスの終了時間・再起動時間の上限から計算可能である．また，スーパーバイザを自動で作成する場合は，自動生成されたスーパーバイザがプロセス作成と同時にリンクを繋げ，その後プロセスが終了するまでリンクを切らないことで，スーパーバイザとプロセス間のリンクの存在を保証できる．よって，提案手法で生成する監視ツリーがロバスト性を持たせることは，難しくはないと想定される．

本研究は，Erlang から強い影響を受けている，Scala のツールキット Akka [*1] や Go のフレームワーク Ergo [*2] で作成されたシステムに応用できる可能性がある．これらのツールキット・フレームワークでは，Erlang と同様に監視ツリーを用いてエラーハンドリングを行う．スーパーバイザに再起動戦略が存在する点も同様である．細かな仕様の差異は存在するが，本研究で提案する問題の定式化・アルゴリズムを適用できる可能性は高いと思われる．

[*1] https://github.com/akka/akka

[*2] https://github.com/ergo-services/ergo

第 8 章

おわりに

本研究では，従来の Erlang システムに適用可能なプロセスの再起動法の実現を目指し，最適な監視ツリーの自動生成法を提案した．最適な監視ツリーを生成するために解くべき問題を定式化し，その問題を解くアルゴリズムを開発した．提案アルゴリズムは計算に最悪指数時間かかるが，現実的な多くの場合のグラフに対しては，グラフの頂点数が 70 の場合でも，100 回の実験の試行における実行時間の中央値は 1.3415 秒であった．2024 年 1 月 19 日時点で GitHub に公開されている 999 個の Erlang システムを調査したところ，gen_server の数が 70 個以下のシステムは約 99.6% だった．

8.1 今後の課題

主に次の 4 つである:

1. 最適監視ツリー問題の計算複雑さの解明
2. 実際の Erlang システムに提案手法を適用した場合の実行時間の計測
3. プロセス間の依存関係の抽出方法の改良
4. より実用的な監視ツリーの出力を目指した問題の定式化

提案アルゴリズムは計算に最悪指数時間かかるが，最適監視ツリー問題が NP 困難かどうかはわかっていない．しかし，最適監視ツリー問題は，NP 困難な既知の最適化問題と似ている部分があるため，NP 困難であると予想している．例えば，最適監視ツリー問題はスケジューリング問題との関連性が観察できる．監視アカシアを考えると，ツリーの根から内部

接点の子に向かって順にそれぞれの子の集合を並べた列は，あるトポロジカルソートの逆順で並ぶ依存関係 DAG の頂点集合の列と等しい．このように，最適な監視ツリーはある頂点集合の列と似ているという意味で，最適監視ツリー問題はスケジューリング問題と少し似ている．多項式時間で解けるスケジューリング問題は存在するものの，ジョブ間に先行制約が入ると NP 困難になることが多々ある [4]．また，なんらかの意味で最適な，グラフの頂点集合の列を求める問題がいくつか知られているが [5]，その多くは NP 困難である．そのため，最適監視ツリー問題もまた NP 困難であると予想している．

6.1 節では，提案アルゴリズムが現実的なグラフに対して十分高速に動作する理由として，「実際のほとんどの Erlang システムに含まれる gen_server の数は，実験の大半のケースで高速に動作した頂点数以下だから」とした．しかし，より厳密にこれを主張するためには，提案手法を実際の Erlang システムに適用した場合でも十分高速に動作する必要がある．現在はプロセス間の依存関係の抽出方法が大きく簡略化されているため，この計測を行うためには，999 個の Erlang システムのプロセス間の依存関係を手作業で抽出する必要がある．

上記の課題を解決するため，プロセス間の依存関係の抽出方法を改良する方法がある．まず考えられる手法が，[2] で提案されている，プログラムの評価結果を近似する方法である．この近似は，プロセス構造がわかる程度には保守的すぎない．[2] では，この手法を用いてスーパーバイザをはじめとするプロセスを抽出し，監視ツリーがロバスト性を満たすか検査している．この手法を応用して，プロセス間の通信を抽出できると思われる．他に考えられる手法が，[3] では明示する必要のあった Multiparty session types (MPSTs) の型を推論する方法である．これを達成するためには，大きく分けて次の要素が必要であると考える:

- Erlang の静的な型検査
- MPSTs の型推論

Erlang の静的な型検査は盛んに研究されている [6, 7, 8, 9]．特に，[9] は Erlang における MPSTs の実装である．しかし，この実装もまた，[3] と同様にして，専用のコールバック関数の実装や専用のデータ送受信 API の使用を要求する．MPSTs の型推論は，[10] で研究

成果が報告されている．ここで，MPSTs では通信方式としてチャンネル方式とアクター方式の 2 つがあり，チャンネル方式ではチャンネルを通してデータが送受信されるのに対し，アクター方式ではプロセスに直接データが送信される．[10] が用いている方式はチャンネル方式であるが，[9] はアクター方式だから，直接の適用は容易ではないと推測される．また，Erlang の静的解析手法で，型推論に必要な情報が十分に求められるかどうかも未知数である．

最後に，今回の定式化における最適な監視ツリーは複数存在する場合があるが，その中からより実用的な監視ツリーを出力できるよう問題の定式化を改善できる可能性がある．例えば，今回の定式化では全ての gen_server を同一に扱ったが，壊れやすさや再起動時間を考慮できると，より実用的な監視ツリーが出力できると思われる．また，提案ツールが出力する監視ツリーは，OR マッパにより自動生成されたスキーマ定義のように，ユーザが手を加えて利用することも意図している．そのため，親子関係の距離など，ユーザの意図を考慮した監視ツリーを出力できると利便性の向上が期待できる．

謝辞

本研究を行うにあたり，粘り強いご指導をいただいた高知工科大学の高田喜朗教授に深く感謝いたします．また，副査を引き受けていただいた高知工科大学の松崎公紀教授と原田崇司講師に心より御礼申し上げます．研究室メンバをはじめ，苦楽を共にしてきた全ての方に感謝いたします．最後に，のびのびと勉学に励まさせていただき，かつ，生活を支えていただいた両親に感謝を申し上げます．

参考文献

[1] Joe Armstrong. A history of erlang. In *the third ACM SIGPLAN conference on History of programming languages*, pp. 6-1–6-26, 2007.

[2] Jan Henry Nyström. Automatic assessment of failure recovery in erlang applications. In *the 8th ACM SIGPLAN workshop on ERLANG*, pp. 23–32, 2009.

[3] Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *the 26th International Conference on Compiler Construction*, pp. 98–108, 2017.

[4] 木瀬洋, 関口恭毅. スケジューリング理論の基礎と応用-ii: ジョブショップ問題とその計算複雑さ. システム/制御/情報, Vol. 44, No. 10, pp. 601–608, 2000.

[5] Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, Vol. 34, No. 3, pp. 313–356, 2002.

[6] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pp. 167–178, 2006.

[7] Giuseppe Castagna, Guillaume Duboc, and José Valim. The design principles of the elixir type system. *arXiv preprint arXiv:2306.06391*, 2023.

[8] Gerard Tabone and Adrian Francalanza. Static checking of concurrent programs in elixir using session types. Technical report, Technical Report, University of Malta, Msida, Malta. Available at `https://gerardtabone.com/ElixirST/archive/UoM%20-%20technical%20report%202022.pdf`, 2022.

[9] Simon Fowler. An erlang implementation of multiparty session actors. *arXiv preprint arXiv:1608.03321*, 2016.

[10] Keigo Imai, Julien Lange, and Rumyana Neykova. Kmclib: Automated inference

and verification of session types from ocaml programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 379–386. Springer, 2022.