Algorithms for Building LUT for STPG-T_c Problem document revision:0.0.1

I. Algorithms and source code

To facilitate understanding of the codebase, we have abstracted the core implementations into distinct algorithmic modules. Each algorithm's implementation spans multiple source files, and there is no direct one-to-one mapping between the conceptual algorithms and the actual code structure. For thorough comprehension, we strongly recommend examining the source code directly.

All Python (.py) files in this project are functional, self-contained modules. To execute specific functionalities: Run the corresponding file directly via CLI. Note this is a research prototype: No intelligent automation features Zero GUI support, Terminal-exclusive operation (Linux environments) and Mandatory runtime: Python $\geq$ 3.10. Users must manually initialize databases to verify lookup tables, though the initialization methods are implemented in the codebase. For solving complete STPG+T_c problems, users are responsible for providing valid runtime environments for SCIP-Jack or Gurobi solvers (either via binary files or licensed installations).

II. Source code structure

We provide the following simplified table to illustrate the code structure. For full functional details, readers must refer directly to the source files.

III. Algorithms and Key Data Structure for Building and Checking LUT

In Algorithm 1, $Prefix$ is given, we firstly get the last path score $V_{P_{(R)}}$ in $Prefix$ to initialize $Suffix$, then form the initial Smt_t with UpdSMT(), and evaluate $T_c(95\%)\big|Smt_t$ with EvaluateTC() (lines 1-4) . If $T_c(95\%)\big|Smt_t \geq T_S$, no valid $Suffix$, function return $SMT = None$ and exist (lines 5-6). Then loop in findSS() to process each $Suffix$ path (lines 10-28). Starting from the last suffix path ($l = N - 1$) (line 10), attempt to assign its path score to the infinite path score (line 12). Then form new Smt_t with $Prefix$ and $Suffix$ (line 13). If the resulting $T_c(95\%)\big|Smt_t \leq T_s$, retain this assignment and proceed to the previous suffix index ($l = l - 1$) (lines 14-16). Otherwise, perform a binary search on next path index $l-1$ between the current value V_{P_R} and the infinite path score to find the largest $V_{P_{(l-1)}}$ that maintains $T_c(95\%)|_{SMT_t} \leq T_s$ (lines 18-28). Then return Smt_t and exist (lines 29-30).

Building on the FPSS result, the SPFS algorithm iteratively explores adjacent boundary points by perturbing the suffix as shown in Algorithm 2:

Algorithm 1: FPSS Algorithm

1 **Function** `FPSS` ($Prefix$, T_s)
  Input: $Prefix$ ($\{V_{P_{(1)}}, \ldots, V_{P_{(R)}}\}$), T_s
  Output: Boundary point SMT (or None)
2   $Suffix \leftarrow \{V_{P_{(R)}}, \ldots, V_{P_{(R)}}\}$;
3   $Smt_t \leftarrow$ UpdSMT($Prefix, Suffix$) ;
4   $T_c \leftarrow$ EvaluateTC(Smt_t) ;
5   **if** $T_c > T_s$ **then**
6   **return** `None` ;
7   Smt_t=`findSS`($Prefix$,$Suffix$) ;
8   **return** Smt_t ;
9 **Function** `findSS` ($Prefix$,$Suffix$)
  Input: $Suffix$ (with inital path scores setting and checked), $Prefix$ + $Suffix$ to get SMT.
  Output: Boundary point SMT
10   $l \leftarrow N - 1$;
11   **while** $l > R$ **do**
12     $Suffix[l] \leftarrow$ INF ;
13     $Smt_t \leftarrow$ UpdSMT($Prefix, Suffix$) ;
14     $T_c \leftarrow$ EvaluateTC(Smt_t) ;
15   **if** $T_c \leq T_s$ **then**
16       $l \leftarrow l - 1$;
17   **else**
18       $a \leftarrow V_{P_{(R)}}$, $b \leftarrow$ INF ;
19   **while** $b > a$ **do**
20         $mid \leftarrow$ findMidScore((a, b) ;
21         $Suffix[l-1] \leftarrow mid$;
22         $Smt_t \leftarrow$ UpdSMT($Prefix, Suffix$) ;
23         $T_c \leftarrow$ EvaluateTC(Smt_t) ;
24   **if** $T_c \leq T_s$ **then**
25           $a \leftarrow mid$;   `// for upper`
26   **else**
27           $b \leftarrow mid$;   `// for lower`
28       $Suffix[l-1] \leftarrow a$;
29   **return** Smt_t ;
30   **return** Smt_t ;

The algorithm start with the FPSS-derived boundary point. Locate the $Suffix$ path index l with the highest $V_{P_{(i)}} < INF$ value (line 2). Then fix the first $l-2$ path scores and recompute the suffix path from index $l-1$ to N by initializing them with next path score of $V_{P_{(l-1)}}$ (lines 5-7). Form new Smt_t with $Prefix$ and $Suffix$, if the new configuration

TABLE I
SOURCE FILE DESCRIPTION

file or dir	*description*
SRC	source code dir, most algorithm or utilities are in this dir.
SRC/simusrc	source code dir of DES simulator for Raft, the code in this dir will be copied to working dirs to simulate.
ResultDataHis	commonly used data files dir. e.g. the databases or some key mappings.
ResultDataHis/testcases	the test cases dirs. All test cases modified from Steinerlib, and use .stp format.
ResultDataHis	commonly used data files dir. e.g. the databases or some key mappings.
document	documents of this project, like copyright declaring, readme file, and this document.
TestWorkingGDir	The working dir for simulations.
SRC/BorderPointDB.py	boundary point database related utility. Helper function to access boundary point LUT and build LUT.
SRC/pathsSeqGen.py	Path value definition utility and simulation history database utility.
SRC/simusrc/linkDelayMdl.py	Link delay distribution definition related utility.
SRC/simusrc/SGBDESIntevals.py	SGB related application simulation program.
SRC/sortlinklist.py	SMT and suffix tree data structures. And classes for FPSS/SPSS.
SRC/findTailsFHSTtoSHFL.py	utility for simulations and automate boundary points calculation.
SRC/collectData.py	utility for experiments data collection.
SRC/BPDATA.py	SMT validation checking utility.
SRC/BPDBCreate.py	search suffix tree building utility.
SRC/DataBaseTestResults.py	database operations utility.
SRC/BPDBCreate.py	SMT validation checking utility.
SRC/Dim2Order.py	Path score operations utility.
SRC/oneDimOrder.py	Path score testing utility.
SRC/postVerifiy.py	test cases test and verification utility.
SRC/STPGcommon.py	test case file operation utility.
SRC/sciprelatedRemoveAlg.py	SCIP-Jack modified solver for commit time constrained problems.
SRC/GurobiSTPGSolver.py	Gurobi solver for commit time constrained problems.
SRC/simuLoadBaseOnBD.py	boundary points calculation automation utility.

violates $T_c(95\%)\big|Smt_t \leq T_s$, decrement l and try next path score of $V_{P_{(l-1)}}$ (lines 8-12). Otherwise, using findSS() in Algorithm 1 to find boundary point Smt_t and record it in bplst (lines 13-14). Then extract new $Surffix$ from newly found boundary point, iterating over $l-1$ hops to discover additional boundary points (lines 15-16). Stop when l is outside the suffix range (line 17). This iterative process generates zero or more $Suffix$ configurations for each $Prefix$, enabling comprehensive boundary point enumeration.

While FPSS and SPFS enable systematic boundary point discovery, they require repeated simulations (each taking 5–6 minutes), which becomes computationally prohibitive at scale. To mitigate this, we exploit Theorem 3 and the discrete nature of boundary points distribution, and form a Stripe-Based Optimization in this subsection.

The LUT is built by systematically enumerating prefix configurations and their corresponding boundary point suffix groups, leveraging the FPSS/SPFS algorithms and stripe-based optimization. The high-level steps for constructing the complete LUT are as Algorithm 3:

1. Prefix Enumeration (line 3): EnumPrefixes() enumerate some key feasible prefix configurations within the effective region $\mathcal{P}_{\text{eff}}$. Since prefix is in $\mathcal{P}_{\text{eff}}$, the valid path score for each valid prefix is also in some region $[P_{min}^{valid}, P_{max}^{valid}]$. The key feasible prefixes can be some path scores in range $[P_{min}^{valid}, P_{max}^{valid}]$. We can denote $len(\text{bplst}) = C_{Enu}$ as the total number of path scores to enumerated in this step.

2. Suffix group generation (line 4): Leveraging the FPSS/SPFS algorithms to compute their boundary points suffix groups with function SimSuffixGroups(). The results merge back to bplst.

3. Stripe Discovery (line 5): Identify contiguous intervals of prefixes that share identical suffix group, defining each interval as a stripe $[\mathcal{P}\int^i_{\min}, \mathcal{P}\int^i_{\max}]$.

4. Boundary Labeling (line 6): In the $\mathcal{P}_{\text{eff}}$ database, mark all prefixes within identified stripe as labeled (flag = 1).

5. Unlabeled Region Processing (lines 7-13): For unlabeled regions in $\mathcal{P}_{\text{eff}}$ database:

(1) Estimate prefixes stripe using interpolation from adjacent labeled prefixes by getRegPrefix() (line 9).

(2) Validate the estimated prefixes stripe via simulation (lines 10-12) and refine stripe boundaries if inconsistencies arise (line 13).

6. Termination: The LUT is complete when all prefixes in $\mathcal{P}_{\text{eff}}$ database are labeled with valid stripes (line 8).

The computational bottleneck of LUT construction lies in the need for simulations to validate $T_c \leq T_s$, with each simulation requiring 5–6 minutes. Below, we define simulation count as the complexity metric. Let $V_{\min} \to V_{\max}$ contain ρ discrete values; the valid prefix space size is $\text{len}(\mathcal{P}_{\text{eff}}) = \binom{\rho+R-1}{R}$. Assume determining a suffix group for a prefix via FPSS/SPFS requires ζ_1 simulations on average, while prefixes

Algorithm 2: SPFS Algorithm

Function SPFS($Prefix$, $Suffix$, T_s)
  Input: $Prefix$, $Suffix$, T_s
  Output: List of boundary points
  $l \leftarrow$ last non-INF suffix index ;
  bplst =[];
  while $l > R$ **do**
    $nexV \leftarrow \texttt{next}(Suffix[l-1])$;
    for $j \in [l-1, N]$ **do**
      $Suffix[j] \leftarrow nexV$;
    $Smt_t \leftarrow$ UpdSMT$(Prefix, Suffix)$;
    $T_c \leftarrow$ EvaluateTC(Smt_t) ;
    if $T_c > T_s$ **then**
      $l \leftarrow l-1$;
      continue;
    Smt_t = findSS($Prefix$, $Suffix$);
    bplst.append(Smt_t);
    $Suffix = Smt_t[R:N]$;
    $l \leftarrow l-1$;
  return bplst;

Algorithm 3: LUT Build Algorithm

Function BuildLUT($\mathcal{P}_{eff}$, T_s)
  Input: Valid prefix region $\mathcal{P}_{\text{eff}}$, SLA threshold T_s
  Output: database with prefix-suffix mappings
  bpDB $\leftarrow$ CreateDB();
  bplst $\leftarrow$ EnumPrefixes($\mathcal{P}_{\text{eff}}$);
  bplst $\leftarrow$ SimSuffixGroups(bplst, T_s);
  stripeLst $\leftarrow$ DiscoverStripes(bplst);
  LabelPrefixes(bpDB, stripeLst);
  unlabRegs $\leftarrow$ GetUnlabRegs(bpDB);
  while unlabRegs *not empty* **do**
    bplst $\leftarrow$ getRegPrefix(unlabRegs);
    bplst $\leftarrow$ SimSuffixGroups(bplst, T_s);
    stripeLst $\leftarrow$ DiscoverStripes(bplst);
    LabelPrefixes(bpDB, stripeLst);
    unlabRegs $\leftarrow$ GetUnlabRegs(bpDB);
  return bpDB;

found by 'getRegPrefix' require ζ_2 simulations. Let η_i denote the number of initial points and η_u the number of new boundary points added during unlabeled region processing. The total simulation count is $\text{SimCount} = \zeta_1 \cdot \eta_i + \zeta_2 \cdot \eta_u$. By the stripe-based method, a few representative points replace full-region evaluation. Assuming an average stripe interval size ψ, the total evaluated points satisfy $\eta_i + \eta_u = 2 \cdot \text{len}(\mathcal{P}_{\text{eff}})/\psi$. Since $\text{len}(\mathcal{P}_{\text{eff}})$ grows combinatorially with R, the LUT complexity is exponential in R, $\text{SimCount} = \mathcal{O}(a^R)$ where a is a system-dependent constant. However, parallelization accelerates computation (1) Prefixes from EnumeratePrefixes() are processed in parallel. (2) Unlabeled regions are partitioned

Data Definiation 1: treeitm data structure

struct {
  int Value; //node value (as ID of node) ;
  int childLowV; //min child value of this node ;
  int childUpV; //max child value of this node ;
  dict siblMap; //store siblings of this node ;
  treeitm FirstChild; //first child of this node ;
  treeitm Parent; //parent of this node;
} *treeitm*;

and validated concurrently. Thus, for moderate R, offline LUT construction remains feasible.

To enable efficient online queries, we preprocess each suffix group into a lookup tree during LUT initialization.

Each suffix group is represented as a binary search tree where nodes correspond to paths ordered by V_{P_i}. Each leaf node stores metadata for constraint checking (e.g., delay bounds, tree relations). As show in Data Definiation 1.

Tree construction function described in Algorithm 4:

A root node for the tree is created (line 2). This root node typically has a default or placeholder value (e.g., 0). The algorithm iterates through each suffix (path) in the input list. For every suffix, it starts traversing from the root node of the tree. It checks if the curNode in the tree already has a child with a value matching the nodeid (line 7). This involves looking at the FirstChild of the curNode and then searching through its siblMap if the FirstChild doesn't match (lines 8-13). If a matching child node exists, the algorithm moves to this existing child node, and it becomes the curNode for the next nodeid in the path (line 22). Otherwise, a new tree node is created with the current nodeid. Its parent is set to the curNode (line 15). If the curNode did not have any children before, this new node becomes its FirstChild. If the curNode already had a FirstChild, the new node is added as a sibling to that FirstChild (and registered in the siblMap of the FirstChild). The algorithm then moves to this newly created node, and it becomes the curNode (lines 16-20). After all nodeids in a path have been processed, the algorithm moves to the next suffix in the input list and repeats (line 3). Once all suffixes have been processed, the function returns the root node of the fully constructed tails tree (line 23).

After we build the look up table, we know all the prefix and corresponding suffix groups. We use Algorithm 4 to build the search tree for each suffix group. Given an SMT candidate SMT_x, the online lookup table checking procedure is as Algorithm 5: In line 2, SplitSMT() split SMT_x into prefix (first R paths) and suffix (remaining $N-R$ paths). Then Query the LUT (bpDB) to find the stripe containing the prefix (line 3), retrieve the search tree (SchTr) associated with the stripe, the search tree was build by Algorithm 4 when pbDB as loaded in memory. Finally (line 4), Algorithm 6 verify the SMT_x satisfy $T_c(95\%) \le T_s$ by chkSuffixes().

In Algorithm 6, the comparison begins from a specific node in the tails tree, which should ideally be the root of the tree

Algorithm 4: Suffixes tree build algorithm

```
1  Function buildSearchTree(suffixes)
       Input  : suffixes: a list of ascending path scores
       Output: root of the tails tree been built
2      root ← treeitm (0,None) ;    // root node
3      foreach path in suffixes do
4          curNode ← root;
5          foreach nodeid in path do
6              foundChild ← None ;
7              if curNode.FirstChild is not None then
8                  curChild ← curNode.FirstChild ;
9                  while curChild is not None do
10                     if curChild.Value = nodeid then
11                         foundChild ← curChild;
12                         break ;
13                     curChild ←
                         curChild.siblMap[nodeid ] ;
14             if foundChild is None then
15                 newChild ← treeitm (nodeid,
                     curNode) ;
16                 if curNode.FirstChild is None then
17                     curNode.addfirstChild
                         (newChild, nodeid) ;
18                 else
19                     curNode.FirstChild.addSibling
                         (nodeid, newChild) ;
20                 curNode ← newChild;
21             else
22                 curNode ← foundChild;
23     return root ;
```

Algorithm 5: Online Constraint Checking Algorithm

```
1  Function OnlineCheck(SMT_x)
       Input: Candidate SMT SMT_x, precomputed bpDB
       Output: Feasibility status valid(0/-1)/invalid(1)
2      (Prefix, Suffix) ← SplitSMT(SMT_x);
3      SchTr ← getStripeSchTree(Prefix, bpDB);
4      Result ← SchTr.chkSuffix(Suffix);
5      return Result;
```

for a complete validation (lines 2-3). The algorithm processes each nodeid in the input suffix one by one (lines 5-25). For the current nodeid, it attempts to find a corresponding child node under the curNode in the tree (lines 6-7). It first checks if curNode's FirstChild matches the nodeid (lines 10-11). If not, it searches through the siblMap of the curNode's FirstChild with a value equal to nodeid (line 13). If a matching child is found: This matching child becomes the new curNode in the tree (line 15), and the algorithm proceeds

Algorithm 6: Use suffixes tree to check suffix validity.

```
1  Function chkSuffix(self,suffix)
       Input  : suffix: ascending path score list
       Output: check boundary result: 0 (math), 1
               (bigger), -1 (less)
2      if self.Parent is not None then
3          raise ValueError("Not called in root node") ;
4      curNode ← self;
5      foreach idx, nodeid in enum(suffix) do
6          if curNode.FirstChild is not None then
7              curChild ← curNode.FirstChild ;
8              matchedChild ← None ;
9              while curChild is not None do
10                 if curChild.Value = nodeid then
11                     matchedChild ← curChild;
12                     break ;
13                 curChild ←
                     curChild.siblMap.get(nodeid, None) ;
14             if matchedChild is not None then
15                 curNode ← matchedChild;
16             else
17                 if nodeid > curNode.childUpV then
18                     return 1 ;
19                 else
20                     return -1 ;
21         else
22             if nodeid > curNode.childUpV then
23                 return 1 ;
24             else
25                 return -1 ;
26     return 0
```

to the next nodeid in the input tail (line 26). If the nodeid falls within the range of $childLowV$ to $childUpV$ (lines 16-25), but no exact child was found, the function returns 0. This signifies that while there isn't a direct path continuation for this specific nodeid, the nodeid itself is considered within an acceptable range defined by the tree structure at that point.