Chapter 2

Lattice Linear Predicate Detection

2.1 Introduction

In this chapter we discuss a general method called Lattice-Linear Predicate (LLP) algorithm that can be used to solve a wide variety of problems including the stable marriage problem, the shortest path problem, the assignment problem, the minimum spanning tree problem and the housing allocation problem.

Section 2.2 defines lattice-linear predicates formally. These predicates are defined on a distributive lattice. The problem is framed as searching for an element in the lattice that satisfies the predicate. Generally, we are interested in the least element in the lattice that satisfies the predicate. Section 2.3 gives the notation that we use for programs based on lattice-linear predicates. It specifies the lattice that we are working on, the starting element in the lattice, the top element of the lattice and the predicate *forbidden* which allows us to advance in the lattice. Section 2.4 lists some desirable properties of the LLP algorithm. The algorithm is naturally *nondeterministic*. There are multiple ways that the algorithm can advance in the lattice. The algorithm is *parallel*. It can advance on multiple components in parallel. The algorithm is *online*. It does not inspect any element higher than the current element of the lattice that is under consideration. Finally, the algorithm returns an answer that is optimal for all components.

2.2 Lattice-Linear Predicates

Let L be the lattice of all n-dimensional vectors of reals greater than or equal to zero vector and less than or equal to a given vector T where the order on the vectors is defined by the component-wise natural $\leq$. The minimum element of this lattice is the zero vector. The lattice is used to model the search space of the combinatorial optimization problem. For simplicity, we are considering the lattice of vectors of non-negative reals; later we show that our results are applicable to any distributive lattice. The combinatorial optimization problem is modeled as finding the minimum element in L that satisfies a boolean *predicate* B, where B models *feasible* (or acceptable solutions). We are interested in parallel algorithms to solve the combinatorial optimization problem with n processes. We will assume that the systems maintains as its state the current candidate vector $G \in L$ in the search lattice, where $G[i]$ is maintained at process i. We call G, the global state, and $G[i]$, the state of process i.

Finding an element in lattice that satisfies the given predicate B, is called the *predicate detection* problem. Finding the *minimum* element that satisfies B (whenever it exists) is the combinatorial optimization problem. We now define *lattice-linearity* which enables efficient computation of this minimum element. Lattice-linearity is first defined in [CG98] in the context of detecting global conditions in a distributed system where it is simply called linearity. We use the term *lattice-linearity* to avoid confusion with the standard usage of linearity.

A key concept in deriving an efficient predicate detection algorithm is that of a *forbidden* state. Given a predicate B, and a vector $G \in L$, a state $G[i]$ is *forbidden* (or equivalently, the index i is forbidden) if for any vector $H \in L$, where $G \leq H$, if $H[i]$ equals $G[i]$, then B is false for H. Formally,

Definition 2.1 (Forbidden State [CG98]) *Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\geq 0}$, and a predicate B, we define $forbidden(G, i, B) \equiv \forall H \in L : G \leq H : (G[i] = H[i]) \Rightarrow \neg B(H)$.*

We define a predicate B to be *lattice-linear* with respect to a lattice L if for any global state G, B is false in G implies that G contains a *forbidden state*. Formally,

Definition 2.2 (lattice-linear Predicate [CG98]) *A boolean predicate B is* lattice-linear *with respect to a lattice L iff $\forall G \in L : \neg B(G) \Rightarrow (\exists i : forbidden(G, i, B))$.*

We now give some examples of lattice-linear predicates.

1. **Job Scheduling Problem**: Our first example relates to scheduling of n jobs. Each job j requires time t_j for completion and has a set of prerequisite jobs, denoted by $pre(j)$, such that it can be started only after all its prerequisite jobs have been completed. Our goal is to find the minimum completion time for each job. We let our lattice L be the set of all possible completion times. A completion vector $G \in L$ is feasible iff $B_{jobs}(G)$ holds where $B_{jobs}(G) \equiv \forall j : (G[j] \geq t_j) \wedge (\forall i \in pre(j) : G[j] \geq G[i] + t_j)$. B_{jobs} is lattice-linear because if it is false, then there exists j such that either $G[j] < t_j$ or $\exists i \in pre(j) : G[j] < G[i] + t_j$. We claim that $forbidden(G, i, B_{jobs})$. Indeed, any vector $H \geq G$ cannot be feasible with $G[j]$ equal to $H[j]$. The minimum of all vectors that satisfy feasibility corresponds to the minimum completion time.

2. **Prefix Sum Problem**: Our second example relates to (exclusive) prefix sum of an array A with non-negative reals. We are required to output an array G such that $G[j]$ equals sum of all entries in A from 0 to $j-1$. We define G to be feasible iff B_{prefix} holds where $B_{prefix} \equiv (\forall j > 0) : (G[j] \geq G[j-1] + A[j-1])$. Again, it is easy to verify that B_{prefix} is lattice-linear. The minimum vector G that satisfies B_{prefix} corresponds to the exclusive prefix sum of the array A.

3. **Continuous Optimization Problem**: We are required to find minimum nonnegative x and y such that $B \equiv (x \geq y^2/4 + 5) \wedge (y \geq x - 4)$. We view this problem as finding minimum (x, y) pair such that B holds. It is easy to verify that B is lattice-linear. If the first conjunct is false, then x is forbidden. Unless x is increased the predicate cannot become true, even if other variables (y for this example) increase. If the second conjunct is false, then y is forbidden. In this case, $x = 6$ and $y = 2$ is the pointwise minimum solution. For some predicates, there may not be any solution. For example, when $B \equiv (x \geq 2y^2 + 5) \wedge (y \geq x - 4)$, there is no nonnegative (x, y) pair such that B holds (verify this!). The predicate B is still lattice-linear but by advancing along x and y we go beyond any bounded (x, y).

4. **A Non Lattice-Linear Predicate** As an example of a predicate that is not lattice-linear, consider the predicate $B \equiv \sum_j G[j] \geq 1$ defined on the space of two dimensional vectors. Consider the vector G equal to $(0,0)$. The vector G does not satisfy B. For B to be lattice-linear either the first index or the second index should be forbidden. However, none of the indices are forbidden in $(0,0)$. The index 0 is not forbidden because the vector $H = (0,1)$ is greater than G, has $H[0]$ equal to $G[0]$ but it still satisfies B. The index 1 is also not forbidden because $H = (1,0)$ is greater than G, has $H[1]$ equal to $G[1]$ but it satisfies B.

The following Lemma is useful in proving lattice-linearity of predicates.

Lemma 2.3 *Let B be any boolean predicate defined on a lattice L of vectors.*
(a) Let $f : L \rightarrow \mathbf{R}_{\geq 0}$ be any monotone function defined on the lattice L of vectors of $\mathbf{R}_{\geq 0}$. Consider the predicate $B \equiv G[i] \geq f(G)$ for some fixed i. Then, B is lattice-linear.
(b) Let L_B be the subset of the lattice L of the elements that satisfy B. Then, B is lattice-linear iff L_B is closed under meets.
(c) If B_1 and B_2 are lattice-linear then $B_1 \wedge B_2$ is also lattice-linear.

Proof: (a) Suppose B is false for G. This implies that $G[i] < f(G)$. Consider any vector $H \geq G$ such that $H[i]$ is equal to $G[i]$. Since $G[i] < f(G)$, we get that $H[i] < f(G)$. The monotonicity of f implies that $H[i] < f(H)$ which shows that $\neg B(H)$.
(b) This is shown in [CG98]. Assume that B is not lattice-linear. This implies that there exists a global state G such that $\neg B(G)$, and $\forall i : \exists H_i \geq G : (G[i] = H_i[i])$ and $B(H_i)$. Consider $Y = \cup_i \{H_i\}$. All elements of $Y \in L_B$. However, $inf\ Y$ which is in G is not an element of L_B. This implies that L_B is not closed under the infimum operation. Conversely, let $Y = \{H_1, H_2, \ldots, H_k\}$ be any subset of L_B such that its infimum G does not belong to L_B. Since G is the infimum of Y, for any i, there exists $j \in \{1 \ldots k\}$ such that $G[i] = H_j[i]$. Since $B(H_j)$ is true for all j, it follows that there exists a G for which lattice-linearity does not hold.

(c) Follows from the equivalence of meet-closed predicates with lattice-linearity and that meet-closed predicates are closed under conjunction. For a more direct proof, suppose that $\neg(B_1 \wedge B_2)$. This implies that one of the conjuncts is false and therefore from the lattice-linearity of that conjunct, a forbidden state exists.

■

For the job scheduling example, we can define B_j as $G[j] \geq max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$. Since $f_j(G) = max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$ is a monotone function, it follows from Lemma 2.3(a) that B_j is lattice-linear. The predicate $B_{jobs} \equiv \forall j : B_j$ is lattice-linear due to Lemma 2.3(c). Also note that the problem of finding the minimum vector that satisfies B_{jobs} is well-defined due to Lemma 2.3(b).

We now discuss detection of lattice-linear predicates which requires an additional assumption called the *efficient advancement property* [CG98] — there exists an efficient (polynomial time) algorithm to determine the forbidden state. This property holds for all the problems considered in this book. Once we determine j such that $forbidden(G, j, B)$, we also need to determine how to advance along index j. To that end, we extend the definition of forbidden as follows.

Definition 2.4 (α-forbidden) *Let B be any boolean predicate on the lattice L of all assignment vectors. For any G, j and positive real $\alpha > G[j]$, we define forbidden(G, j, B, α) iff*

$$\forall H \in L : H \geq G : (H[j] < \alpha) \Rightarrow \neg B(H).$$

Given any lattice-linear predicate B, suppose $\neg B(G)$. This means that G must be advanced on all indices j such that $forbidden(G, j, B)$. We use a function $\alpha(G, j, B)$ such that $forbidden(G, j, B, \alpha(G, j, B))$ holds whenever $forbidden(G, j, B)$ is true. With the notion of $\alpha(G, j, B)$, we have the algorithm LLP shown in Fig. LLP. The algorithm LLP has two inputs — the predicate B and the top element of the lattice T. It returns the least vector G which is less than or equal to T and satisfies B (if it exists). Whenever B is not true in the current vector G, the algorithm advances on all forbidden indices j in parallel. This simple parallel algorithm can be used to solve a large variety of combinatorial optimization problems by instantiating different $forbidden(G, j, B)$ and $\alpha(G, j, B)$.

Algorithm LLP: Algorithm LLP to find the minimum vector at most T that satisfies B

```
1  vector function getLeastFeasible(T: vector, B: predicate)
2  var G: vector of reals initially ∀i : G[i] = 0;
3  while ∃j : forbidden(G, j, B) do
4      for all j such that forbidden(G, j, B) in parallel:
5          if (α(G, j, B) > T[j]) then return null;
6          else G[j] := α(G, j, B);
7  endwhile;
8  return G; // the optimal solution
```

Theorem 2.5 *Suppose there exists a fixed constant $\delta > 0$ such that $\alpha(G, j, B) - G[j] \geq \delta$ whenever $forbidden(G, j, B)$. Then, the parallel algorithm LLP finds the least vector $G \leq T$ that satisfies B, if one exists.*

Proof: Since $G[j]$ increases by at least δ for at least one forbidden j in every iteration of the *while* loop, the algorithm terminates in at most $\sum_i \lceil T[i]/\delta \rceil$ number of steps.

We show that the algorithm maintains the invariant $(I1)$ that for all indices j, any vector V such that $V[j]$ is less than $G[j]$ cannot satisfy B. Formally, the invariant $(I1)$ is

$$\forall j : (\forall V \in L : (V[j] < G[j]) \Rightarrow \neg B(V)).$$

Initially, the invariant holds trivially because G is initialized to 0. Suppose $forbidden(G, j, B)$. Then, we increase $G[j]$ to $\alpha(G, j, B)$. We need to show that this change maintains the invariant. Pick any V such that $V[j] < \alpha(G, j, B)$. We now do a case analysis. If $V \geq G$, then $\neg B(V)$ holds from the definition of $\alpha(G, j, B)$. Otherwise, there exists some k such that $V[k] < G[k]$. In this case $\neg B(V)$ holds due to $(I1)$.

We can now show Theorem 2.5 using the invariant. First, suppose that the algorithm LLP terminates because $\alpha(G, j, B) > T[j]$. In this case, there is no feasible vector in L due to the invariant (because the predicate B is false for all values of $G[j]$). Now suppose that the algorithm terminates because there does not exist any j such that $forbidden(G, j, B)$. This implies that G satisfies B due to lattice-linearity of B. It is also the least vector that satisfies B due to the invariant $(I1)$.

■

For the job scheduling example, we get a parallel algorithm to find the minimum completion time by using $forbidden(G, j, B_{jobs}) \equiv (G[j] < t_j) \vee (\exists i \in pre(j) : G[j] < G[i] + t_j)$, and $\alpha(G, j, B_{jobs}) = \max\{t_j, \max\{G[i] + t_j | i \in pre(j)\}\}$.

For the prefix sum example, we get a parallel algorithm by using $forbidden(G, j, B_{prefix}) \equiv (G[j] < G[j-1] + A[j-1])$ and $\alpha(G, j, B_{prefix}) = G[j-1] + A[j-1]$ for all $j > 0$.

We now show, on account of Lemma 2.3(c), that if we have a parallel algorithm for a problem, then we also have one for the constrained version of that problem.

Lemma 2.6 *Let LLP be the parallel algorithm to find the least vector G that satisfies B_1 if one exists. Then, LLP can be adapted to find the least vector G that satisfies $B_1 \wedge B_2$ for any lattice-linear predicate B_2.*

Proof: The algorithm *LLP* can be used with the following changes:
$forbidden(G, j, B_1 \wedge B_2) \equiv forbidden(G, j, B_1) \vee forbidden(G, j, B_2)$, and
$\alpha(G, j, B_1 \wedge B_2) = \max\{\alpha(G, j, B_1), \alpha(G, j, B_2)\}$.

■

For example, suppose that we want the minimum completion time of jobs with the additional lattice-linear constraint that $B_2(G) \equiv (G[1] = G[2])$. B_2 is lattice-linear with $forbidden(G, 1, B_2) \equiv (G[1] < G[2])$ and $forbidden(G, 2, B_2) \equiv (G[2] < G[1])$. By applying, Lemma 2.6, we get a parallel algorithm for the constrained version.

Note that the straightforward application of *LLP* may not give the most time-efficient parallel algorithm. The efficiency of the algorithm may depend upon $\alpha(G, j, B)$ chosen for the predicate B. We will later show such optimizations for many problems such as the shortest path algorithm.

2.3 Notation

We first go over the notation used in description of our parallel algorithms. Fig. 2.1 shows parallel algorithms for the job-scheduling and the shortest path problems. We have a single variable G in all the examples shown in Fig. 2.1.

All other variables are derived directly or indirectly from G. G is an array of objects such that $G[j]$ is the state of thread j for a parallel program. There are three sections of the program.

The **init** section is used to initialize the state of the program. All the parts of the program are applicable to all values of j. For example, the *init* section of the job scheduling program in Fig. 2.1 specifies that $G[j]$ is initially $t[j]$. Every thread j would initialize $G[j]$.

The **always** section defines additional variables which are derived from G. The actual implementation of these variables are left to the system. They can be viewed as macros.

The third section gives the desirable predicate either by using the **forbidden** predicate or **ensure** predicate. The *forbidden* predicate has an associated *advance* clause that specifies how $G[j]$ must be advanced whenever the forbidden predicate is true. For many problems, it is more convenient to use the complement of the forbidden predicate. The *ensure* section specifies the desirable predicates of the form $(G[j] \geq expr)$ or $(G[j] \leq expr)$. The statement *ensure* $G[j] \geq expr$ simply means that whenever thread j finds $G[j]$ to be less than *expr*; it can advance $G[j]$ to *expr*. Since *expr* may refer to G, just by setting $G[j]$ equal to *expr*, there is no guarantee that $G[j]$ continues to be equal to *expr* — the value of *expr* may change because of changes in other components. We use *ensure* statement whenever *expr* is a monotonic function of G and therefore the predicate is lattice-linear.

P_j: Code for thread j
// common declaration for all the programs below
shared var G: array[$1..n$] of $0..maxint$;
job-scheduling:
    input: $t[j] : int$, $pre(j)$: list of $1..n$;
    init: $G[j] := t[j]$;
    forbidden: $G[j] < \max\{G[i] + t[j] \mid i \in pre(j)\}$;
    advance: $G[j] := \max\{G[i] + t[j] \mid i \in pre(j)\}$;

job-scheduling:
    input: $t[j] : int$, $pre(j)$: list of $1..n$;
    init: $G[j] := t[j]$;
    ensure: $G[j] \geq \max\{G[i] + t[j] \mid i \in pre(j)\}$;

shortest path from node s: Parallel Bellman-Ford
    input: $pre(j)$: list of $1..n$; $w[i, j]$: int for all $i \in pre(j)$
    init: if $(j = s)$ then $G[j] := 0$ else $G[j] :=$ maxint;
    ensure: $G[j] \leq \min\{G[i] + w[i, j] \mid i \in pre(j)\}$

Figure 2.1: LLP Parallel Program for (a) job scheduling problem using forbidden predicate (b) job scheduling problem using ensure clause and (c) the shortest path problems

2.4 Properties of the LLP Algorithm

The LLP algorithm has many useful properties. We list them here so that the reader can apply them to various problems studied in this book.

These properties are applicable to all the problems for which the LLP algorithm is used.

1. **Nondeterminism in Evaluation of Forbidden Predicate**: Given a global state G, there may be multiple indices j for which $G[j]$ is forbidden. The LLP algorithm is correct irrespective of the order in which these indices are updated. The efficiency of the algorithms may differ depending upon the order in which these indices are updated, but the correctness is independent of the order. For example, in the stable marriage problem, the final answer returned is independent of the order in which men propose. In the shortest path problem, the final answer returned is independent of the order in which the nodes update their estimates. It is important to note that the term *answer* is with respect to the LLP algorithm. For the shortest path problem, the lattice is with respect to the cost and not the actual paths. There may be multiple shortest paths to a vertex and the order of evaluation of forbidden indices may result in the paths that are returned be different. However, they would all have the same cost.

2. **Parallel Evaluation of Forbidden Predicate**: Suppose that G is shared among different threads such that thread j is responsible for evaluating $forbidden(G, j)$. While this thread is evaluating this predicate other threads may have advanced on other indices, i.e., thread j may have old information of $G[i]$ for $i \neq j$. However, this would still keep the algorithm correct. For example, in the stable marriage problem, men can propose to women in parallel. In the shortest path algorithm, multiple vertices can update the estimate of their lower bounds and their parents in parallel.

3. **No Lookahead Required for evaluation of Forbidden Predicate**: The LLP algorithm determines whether an index j is forbidden depending upon only the current global state G. This means that these algorithms are applicable in online settings where the future part of the lattice is revealed only when a forbidden index needs to advance. For example, in the stable marriage problem, when we are computing the man-optimal stable marriage, a man may not reveal his preference list. Only when he is rejected (his state is forbidden), he needs to advance on his choices and therefore reveal the next woman on his list. The same reasoning applies to the housing allocation problem. For some problems, such as the shortest path problem, the weights on the edges is required to evaluate forbidden predicate. Therefore, the graph must be known and static for the LLP algorithm.

4. **The Optimal Feasible Global State is Optimal for Individual Nodes**: Suppose that for the stable marriage problem, one of the men, say m_1 is interested in finding the stable marriage in which he has the topmost choice possible for him. This man does not care how other men are paired. If we take the solution derived from the LLP algorithm, then the woman m_1 is paired with would be identical to the woman m_1 is paired with in any stable marriage that is optimal from just the perspective of m_1. More generally, let G be the global state that is feasible and optimal with respect to index i, i.e., for all feasible H, $G[i] \leq H[i]$. Let G_{llp} be the global state computed by the LLP algorithm, then $G[i] = G_{llp}[i]$. This follows from the meet-closure property of feasible states. If $G[i] < G_{llp}[i]$, then the global state given by $G \sqcap G_{llp}$ is also feasible and strictly smaller than G_{llp} contradicting that G_{llp} is the least global state that satisfies the feasibility predicate.

2.5 Problems

1. Let G be a n-dimensional vector of positive numbers. Let $pred$ be a binary acyclic relation on $[n]$. Show that the predicate $B \equiv \forall (i,j) \in pred : G[j] \geq G[i] + 1$ is a lattice-linear predicate.

2. Let $(X, \leq)$ be a poset. A subset $Y \subseteq X$ is an order ideal if it satisfies

$$\forall u, v \in X : (v \in Y) \wedge (u \leq v) \Rightarrow (u \in Y)$$

   Show that the predicate $B(Y) \equiv$ "Y is an order ideal of $(X, \leq)$" is lattice-linear in the boolean lattice of all subsets of X.

3. Show that lattice-linearity is not closed under disjunction.

4. Show that lattice-linearity is not closed under negation.

2.6 Bibliographic Remarks

The lattice-linearity property is described in [CG98]. Its application to stable matching is first shown in [Gar17]. Its application to design of parallel algorithms is in [Gar20].