```
; Mechanically Proving Approximation Bounds on Lattice-Linear Parallel
; Algorithms
;
; Abdullah Rasheed (EID: ahr875)
;
; 1. Introduction
;
; Lattice-Linear Predicates (LLPs) describe a class of predicates
; defined  over lattices of vectors for which the problem of finding a
; minimal satisfying vector has a simple parallel/distributed algorithm.
; LLPs have been shown to be effective at parallelizing a variety of
; well-known optimization problems improving upon existing solutions. Garg's
; algorithm for detecting LLPs runs in time proportional to the height of
; the lattice, offering a large cut-down on the solution space. For
; lattices with very large height, however, it is simply infeasible to
; parallelize any solution. Many such problems are NP-hard, and thus if P !=
; NP, then there does not exist any algorithm for them in NC.
;
;
; To combat the hardness of such problems, a common remedy is to
; resort to approximations. While LLPs describe optimization problems,
; it is hard to encode approximation algorithms as LLPs because of
; their optimality. In this project, we discover a method for
; exploiting the lattice via abstraction to implicitly achieve
; approximations with LLPs. Furthermore, we represent this method in ACL2 to
; mechanically construct robust proofs of correctness.
;
; 2. Background
;
; A lattice is a partially-ordered set where each pair of elements has a
; well-defined least upper bound (join) and greatest lower bound (meet).
; The lattice is said to be distributive if join distributes over meet
; (equivalent to meet distributes over join).
;
; A Galois connection between two lattices A and B is a pair of functions
; alpha : A -> B and gamma : B -> A such that gamma(alpha(a)) >= a and
; alpha(gamma(b)) <= b. A Galois connection is called a Galois insertion
; (GI) if alpha(gamma(b)) = b. That is, B is exactly representable in A. We
; will assume only GIs throughout this project since is well-known that a
; Galois connection may be reduced to a GI. It is also known that alpha
; uniquely determines gamma, and vice versa. Finally, it is known that for
; order-preserving f : A -> A (alpha o f o gamma) <= g pointwise for all g :
; B -> B such that alpha o f <= g o alpha. These properties allow us to
; simplify the project, and we do not verify them in ACL2.
;
; d is a quasi-metric on A if for all x,y,z in A, d(x,y) = 0 iff x = y,
; d(x,y) > 0 iff x > y, and d(x,y) + d(y,z) >= d(x,z). For the purposes of
; this problem, a specific measure and metric is important, but that is
; unimportant for this project.
;
```

```
; Let L be a distributive lattice of n-dimensional vectors and B be a
; predicate over L. Then, we define forbidden(G,i,B) := forall H in L such
; that G <= H, G[i] = H[i] ==> not B(H). This states that component i is
; forbidden in vector G in L if, no matter how G is progressed, fixing i
; keeps the predicate false. We then say that B is lattice-linear if for any
; G that makes B false, G has a forbidden component. For the purposes of
; minimality, we also need to know how much to progress a forbidden
; component so as to make it no longer forbidden while not jumping over the
; optimal solution. Let delta(G,i,B) be the greatest value such that for all
; H in L such that H >= G, H[i] < delta(G,i,B) ==> not B(H) whenever
; forbidden(G,i,B). That is, it is the least value we need to bring
; component i up to in order to make it no longer forbidden. Advancing a
; vector G with a forbidden component i is done by setting the ith component
; to delta(G,i,B).
;
; Distributivity in the lattice is only necessary for certain performance
; improvements, so for the purposes of this project we do not worry
; ourselves with it. We later note the potential for future work with
; distributive lattices. For the purposes of this paper, and as is typically
; and classically presented in LLPs, we will consider vectors ordered
; component-wise throughout this project and report.
;
; 3. A Method for Approximating LLP Problems
;
; Suppose we have an LLP B over a lattice of n-dimensional vectors L with a
; quasi-metric d defined over it. Let L' be a lattice of m-dimensional
; vectors with (alpha, gamma) a GI between L and L'. We say that f : L -> L,
; f' : L' -> L' are epsilon-complete (e.c.) w.r.t the GI if  d(f(G), (gamma
; o f' o alpha)(G)) <= epsilon for all G in L.
;
; The basic idea is as follows: with L' being a lattice with smaller height
; than L, progress through the smaller lattice, checking whether we have
; stepped over a feasible solution in L (by simply checking B(f(G)) where G
; is the last position we checked). If f(G) is feasible, then it must be
; optimal (proven in ACL2, described later), so d(f(G), (gamma o f' o
; alpha)(G)) = d(OPT, (gamma o f' o alpha)(G)) <= epsilon where OPT is the
; optimal solution in L. We can abuse the GI property to make this more
; efficient (also proven in ACL2) since repeated application of (gamma o f'
; o alpha) gives  (gamma o f' o alpha o gamma o f' o alpha) = (gamma o f' o
; alpha). Many applications of this will cancel out many instances of (alpha
; o gamma) by the GI property, thereby saving many unnecessary computations.
;
; Counter-intuitively, the epsilon distance does not accumulate because we
; "fix" a comparison point in L at each step. This is the crux and novelty
; of the project, proven in ACL2.
;
; 4. ACL2 Representation and Results
;
; In this section, we will use "components" and "states" interchangebly.
;
```

```
; We represent LLPs abstractly so as to apply these proofs to any concrete
; or realized instance by utilizing the "encapsulate" primitive from ACL2.
; This approach provides the definitional properties of LLPs without
; exposing the definition of the abstract functions and predicates. The
; encapsulate includes properties for the necessary theory / realizable
; functions and constants.
;
; In LLPs, lattices are typically over vectors of reals ordered
; component-wise,  however for the sake of simplicity in this project, we
; consider vectors of naturals ordered similarly. This comes in the form of
; the definitions vecp and lattice-leq, both of which have a small number of
; associated theorems.
;
; From these definitions and the basic properties provided by encapsulate,
; we prove a number of important lemmas. The most important theorems proven
; are:
;    1. bounded-by-opt:
;           This proves that advancing a vector below OPT keeps it
;           below OPT.
;    2. gi-makes-stepping-efficient:
;           This proves that GI's enable an optimization on
;           the approximation algorithm.
;    3. constant-approx-dist:
;           This proves that the algorithm gives a constant approximation
;           factor (namely, epsilon) GIVEN that it only takes a finite
;           number of steps to reach OPT in the standard LLP algorithm.
;    4. lattice-leq-component & advance-is-higher-lattice:
;           These prove that advancing a vector with the aforementioned
;           advance rule moves it strictly up the lattice. This directly
;           implies (with bounded-by-opt) that it only takes a finite
;           number of steps to reach OPT in the standard LLP algorithm
;           (thus supporting constant-approx-dist).
;
; 5. Conclusion
;
; In this project, presented a novel technique for representing
; approximation algorithms with LLPs, modeled this technique in ACL2, and
; utilized the ACL2 theorem prover to prove an approximation bound on this
; technique. Using an autmated theorem prover like ACL2 not only gave strong
; assurance of the technique's validity, but also offered valuable insights
; into automated deduction and the technique's underlying recursive
; structure. This project can be further extended to verify very important
; LLP optimizations utilizing lattice distributivity and Birkhoff's theorem.
; It would also be a good step to apply these ACL2 theorems to verify an
; application of this technique.


(defun indexp (x y) (and (natp x) (natp y) (>= x 1) (<= x y)))

(defun vecp (G n)
```

```
  (if (zp n)
      (null G)
      (and (consp G)
           (natp (car G))
           (<= 0 (car G))
           (vecp (cdr G) (- n 1)))))

(defthm vecp-length-strict
  (implies (and (natp n) (vecp G n))
           (equal (len G) n)))

(defthm vecp-is-true-listp
  (implies (and (natp n) (vecp x n))
           (true-listp x)))

(defun ith (G i)
  (if (or (atom G) (<= i 1))
      (car G)
      (ith (cdr G) (- i 1))))

(defthm vecp-nat-and-nonneg
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n))
           (natp (ith G i)))
    :rule-classes ((:type-prescription)))

(defun set-ith (G i v)
  (if (atom G)
      nil
      (if (equal i 1)
          (cons v (cdr G))
          (cons (car G)
                (set-ith (cdr G) (- i 1) v)))))

(defthm set-ith-sets
  (implies (and (natp n) (indexp i n) (vecp G n))
           (equal (ith (set-ith G i v) i) v)))

(defthm set-ith-conserves
  (implies (and (natp n) (indexp i n) (indexp j n)
                (not (equal i j)) (vecp G n))
           (equal (ith (set-ith G i v) j) (ith G j))))

(defthm set-ith-type-preservation
  (implies (and (natp n) (indexp i n) (vecp G n) (natp v) (>= v 0))
           (vecp (set-ith G i v) n)))

(defun lattice-leq (x y)
  (if (consp x)
      (and (<= (car x) (car y)) (lattice-leq (cdr x) (cdr y)))
      T))
```

```
(defthm lattice-leq-recurs-car
  (implies (and (natp n) (>= n 1) (vecp x n) (vecp y n) (lattice-leq x y))
           (<= (car x) (car y))))

(defthm lattice-leq-recurs-cdr
  (implies (and (natp n) (>= n 1) (vecp x n) (vecp y n) (lattice-leq x y))
           (lattice-leq (cdr x) (cdr y))))

(defthm lattice-leq-reflexive
  (implies (and (natp n) (vecp x n) (vecp y n))
           (lattice-leq x x)))

(defthm lattice-leq-antisymmetric
  (implies (and (natp n) (>= n 1) (vecp x n) (vecp y n)
                (lattice-leq x y) (not (equal x y)))
           (not (lattice-leq y x)))
    :hints (("Goal" :use (vecp-is-true-listp))))

(defthm lattice-leq-transitive
  (implies (and (natp n) (vecp x n) (vecp y n)
                (lattice-leq x y) (lattice-leq y z))
           (lattice-leq x z)))

(encapsulate
    (
        ((forbidden * *) => *)
        ((feasible *) => *)
        ((feasible-abs *) => *)
        ((alpha *) => *)
        ((gamma *) => *)
        ((advance * *) => *)
        ((advance-abs *) => *)
        ((dist * *) => *)
        ((ep) => *)
        ((delta * *) => *)
        ((get-forbidden-i * *) => *)
        ((get-forbidden *) => *)
        ((opt *) => *)
        ((stepmin) => *)
    )

    (local (defun feasible (x) (if x t t)))
    (local (defun feasible-abs (x) (if x t t)))
    (local (defun alpha (x) x))
    (local (defun gamma (x) x))
    (local (defun forbidden (G i) (if (and i G) nil nil)))
    (local (defun advance (G i) (if i G G)))
    (local (defun advance-abs (G) G))
    (local (defun dist (x y) (if (and x y) 0 0)))
```

```
(local (defun ep () 1))
(local (defun delta (G i) (+ 1 (ith G i))))
(local (defun get-forbidden-i (G i)
         (if (or (zp i) (atom G))
             nil
             (if (forbidden G i)
                 i
                 (get-forbidden-i G (- i 1))))))
(local (defun get-forbidden (G) (get-forbidden-i G (len G))))
(local (defun opt (n) (if (zp n) nil (cons 0 (opt (- n 1))))))
(local (defun stepmin () 1))

(defthm alpha-is-vecp
  (implies (and (natp n) (>= n 1) (vecp G n))
           (vecp (alpha G) n)))

(defthm gamma-is-vecp
  (implies (and (natp n) (>= n 1) (vecp G n))
           (vecp (gamma G) n)))

(defthm advance-abs-is-vecp
  (implies (and (natp n) (>= n 1) (vecp G n))
           (vecp (advance-abs G) n)))

(defthm opt-is-vecp (implies (natp n) (vecp (opt n) n)))

(defthm stepmin-is-natp (natp (stepmin)))

(defthm ep-is-natp (natp (ep)))

(defthm get-forbidden-sem-zero
  (implies (or (zp i) (atom G))
           (equal (get-forbidden G) nil)))

(defthm get-forbidden-sem-recurs
  (implies (and (natp n) (> n 0) (vecp G n) (get-forbidden G))
           (equal (get-forbidden G) (+ 1 (get-forbidden (cdr G))))))

(defthm get-forbidden-i-is-indexp
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n)
                (get-forbidden-i G i))
           (indexp (get-forbidden-i G i) n)))

(defthm get-forbidden-i-is-forbidden
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n)
                (get-forbidden-i G i))
           (forbidden G (get-forbidden-i G i))))

(defthm get-forbidden-is-indexp
  (implies (and (natp n) (>= n 1) (vecp G n) (get-forbidden G))
```

```
           (indexp (get-forbidden G) n)))

(defthm get-forbidden-is-forbidden
  (implies (and (natp n) (>= n 1) (vecp G n) (get-forbidden G))
           (forbidden G (get-forbidden G))))

(defthm get-forbidden-gives-nil
  (implies (and (natp n) (>= n 1) (vecp G n)
                (not (get-forbidden G)) (indexp i n))
           (not (forbidden G i))))

(defthm get-forbidden-nil-is-feasible
  (implies (and (natp n) (>= n 1) (vecp G n) (not (get-forbidden G)))
           (feasible G)))

(defthm advance-goes-up
  (implies (and (natp n) (>= n 1) (indexp i n) (vecp G n))
           (lattice-leq G (advance G i))))

(defthm advance-makes-unforbidden
  (implies (and (natp n) (>= n 1) (indexp i n) (vecp G n)
                (forbidden G i))
           (not (forbidden (advance G i) i))))

(defthm advance-progresses-by-delta
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n)
                (forbidden G i))
           (equal (advance G i) (set-ith G i (delta G i)))))

(defthm ag-galois-connection
  (implies (and (natp n) (>= n 1) (vecp G n) (vecp H n))
           (lattice-leq G (gamma (alpha G)))))

(defthm ag-galois-insertion
  (implies (and (natp n) (>= n 1) (vecp G n))
           (equal (alpha (gamma G)) G)))

(defthm llp-abs-soundness
  (implies (and (natp n) (vecp G n) (feasible-abs G))
           (feasible (gamma G))))

(defthm forbidden-def
  (implies (and (natp n) (>= n 1) (indexp i n) (vecp G n) (vecp H n)
                (forbidden G i) (lattice-leq G H)
                (equal (ith G i) (ith H i)))
           (not (feasible H))))

(defthm llp-def
  (implies (and (natp n) (>= n 1) (indexp i n) (vecp G n) (feasible G))
           (not (forbidden G i))))
```

```
(defthm ep-completeness
  (implies (and (natp n) (>= n 1) (indexp i n) (vecp G n)
                (forbidden G i))
           (<= (dist (advance G i)
                     (gamma (advance-abs (alpha G))))
               (ep))))

(defthm dist-triangle-inequality
  (implies (and (natp n) (>= n 1) (vecp G n) (vecp H n) (vecp A n))
           (<= (dist G A) (+ (dist G H) (dist H A)))))

(defthm dist-zp-is-eq
  (implies (and (natp n) (>= n 1) (vecp G n))
           (equal (dist G G) 0)))

(defthm delta-forbidden
  (implies (and (natp n) (>= n 1) (indexp i n) (vecp G n) (vecp H n)
                (forbidden G i) (lattice-leq G H)
                (< (ith H i) (delta G i)))
           (not (feasible H))))

(defthm no-forbidden-is-feasible
  (implies (and (natp n) (>= n 1) (vecp G n) (not (get-forbidden G)))
           (feasible G)))

(defthm opt-is-feasible
  (implies (and (natp n) (>= n 1))
           (feasible (opt n))))

(defthm opt-is-optimal
  (implies (and (natp n) (>= n 1) (vecp G n) (lattice-leq G (opt n))
                (not (equal G (opt n))))
           (not (feasible G))))

(defthm opt-is-optimal-contrapositive
  (implies (and (natp n) (>= n 1) (vecp G n) (feasible G))
           (lattice-leq (opt n) G)))

(defthm delta-type
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n))
           (natp (delta G i)))
    :hints (("Goal" :use (vecp-nat-and-nonneg))))

(defthm delta-bound
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n)
                (forbidden G i))
           (>= (- (delta G i) (ith G i)) (stepmin))))

(defthm increase-stepmin
```

```
      (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n)
                    (forbidden G i))
                (>= (delta G i) (+ (ith G i) (stepmin)))))

    (defthm delta-is-higher
      (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n)
                    (forbidden G i))
                (> (delta G i) (ith G i))))

    (defthm stepmin-pos (> (stepmin) 0))
)

(defthm delta-is-bounded
  (implies (and (natp n) (>= n 1) (indexp i n) (forbidden G i) (vecp G n)
                (vecp H n) (feasible H) (lattice-leq G H))
           (<= (delta G i) (ith H i)))
    :hints (("Goal" :use (delta-forbidden))))

(defthm monotone-update
  (implies (and (natp n) (>= n 1) (indexp i n) (vecp G n) (vecp H n)
                (lattice-leq G H) (<= v (ith H i)))
           (lattice-leq (set-ith G i v) H)))

(defthm detect-llp-step-stays-below
  (implies (and (natp n) (>= n 1) (vecp G n) (vecp M n) (indexp i n)
                (forbidden G i) (lattice-leq G M) (feasible M))
           (lattice-leq (advance G i) M))
    :hints (("Goal" :use (advance-progresses-by-delta
                          delta-is-bounded
                          monotone-update))))

(defthm advance-is-higher
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (forbidden G i))
           (> (ith (advance G i) i) (ith G i))))

(defthm this-is-so-simple
  (implies (and (natp n) (>= n 1) (vecp G n) (vecp H n)
                (> (ith H i) (ith G i)))
           (not (equal H G))))

(defthm this-is-even-simpler
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n)
                (forbidden G i)) (>= (delta G i) 0))
    :hints (("Goal" :use delta-type)))

(defthm advance-is-vecp
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (forbidden G i))
           (vecp (advance G i) n))
    :hints (("Goal" :use (advance-progresses-by-delta
                          delta-type this-is-even-simpler
```

```
                           set-ith-type-preservation))))

(defthm advance-is-higher-lattice
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (forbidden G i))
           (not (equal (advance G i) G)))
    :hints (("Goal" :use (advance-is-higher
                          advance-is-vecp
                          (:instance this-is-so-simple
                                     (H (advance G i)))))))

(defthm advance-conserves-other
  (implies (and (natp n) (>= n 1) (vecp G n)
                (indexp i n) (indexp j n)
                (forbidden G i) (not (equal i j)))
           (equal (ith (advance G i) j) (ith G j)))
  :hints (("Goal" :use (advance-progresses-by-delta set-ith-conserves))))

(defthm equal-implies-<=
  (implies (equal x y)
           (<= x y))
    :rule-classes (:rewrite))

(defthm <-implies-<=
  (implies (< x y)
           (<= x y))
    :rule-classes (:rewrite))

(defthm lattice-leq-component
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (indexp j n)
                (forbidden G i))
           (<= (ith G j) (ith (advance G i) j)))
  :hints (("Goal"
           :cases ((equal i j)
                   (not (equal i j)))
           :use ( advance-progresses-by-delta
                  set-ith-sets
                  advance-conserves-other
                  advance-is-higher ))))

(defthm no-forbidden-below-opt-is-opt
  (implies (and (natp n) (>= n 1) (vecp G n) (not (get-forbidden G))
                (lattice-leq G (opt n)))
           (equal (opt n) G)))

(defthm advance-keeps-other-states-fixed
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp j n) (indexp i n)
                (not (equal i j)) (forbidden G i))
           (equal (ith (advance G i) j) (ith G j)))
    :hints (("Goal" :use (advance-progresses-by-delta set-ith-conserves))))
```

```
(defthm forbidden-is-infeasible
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (forbidden G i))
           (not (feasible G))))

(defthm bounded-by-opt
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (forbidden G i)
                (lattice-leq G (opt n)))
           (lattice-leq (advance G i) (opt n)))
  :hints (("Goal" :use (opt-is-feasible opt-is-vecp
                        (:instance detect-llp-step-stays-below
                                   (M (opt n))))
                  :in-theory (disable advance-progresses-by-delta))))

(defthm apply-gi
  (implies (and (natp n) (>= n 1) (vecp G n))
           (equal (alpha (gamma (advance-abs (alpha G))))
                  (advance-abs (alpha G))))
  :hints (("Goal" :use ((:instance ag-galois-insertion
                                    (G (advance-abs (alpha G))))))))

(defun abstract-step (G k)
  (if (zp k)
      G
      (abstract-step (advance-abs G) (- k 1))))

(defun transit-step (G k)
  (if (zp k)
      G
      (transit-step (gamma (advance-abs (alpha G))) (- k 1))))

(defthm gi-makes-stepping-efficient
  (implies (and (natp n) (>= n 1) (natp k) (>= k 1) (vecp G n))
           (equal (gamma (abstract-step (alpha G) k)) (transit-step G k)))
  :hints (("Goal" :use ((:instance ag-galois-insertion
                                    (G (advance-abs (alpha G))))))))

(defthm step-to-opt
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (forbidden G i)
                (lattice-leq G (opt n))
                (lattice-leq (opt n) (advance G i)))
           (equal (opt n) (advance G i))))

(defthm constant-approx-dist
  (implies (and (natp n) (>= n 1) (vecp G n) (indexp i n) (forbidden G i)
                (lattice-leq G (opt n))
                (lattice-leq (opt n) (advance G i)))
           (<= (dist (opt n) (gamma (advance-abs (alpha G))))
               (ep)))
  :hints (("Goal" :use (step-to-opt ep-completeness))))
```