1 Final charge of 3-vertices

The initial charge of a 3-vertex v is -1. Let f_1, f_2 and f_3 be the faces incident with v, with $|f_1| \leq |f_2| \leq |f_3|$. If $|f_1| = 3$, then $|f_2|, |f_3| \geq 5$, and v receives 1 from f_1. If $|f_1| = 4$, then either $|f_2| = 4$, $|f_3| = \Delta^\star$, and both f_1 and f_2 send $1/2$ to v; or $|f_2| \geq 5$ and f_1 sends 1 to v. In all these cases, v does not send any charge, and thus its final charge is 0.

Hence, suppose that $|f_1| \geq 5$. Then, v receives charge by $\texttt{iso}_\ell$ and $E_{\ell,i}$ rules, and sends charge by `through_heavy` rules. For all the possible combinations of lengths of the incident faces, the resulting charge is non-negative:

```
verify_deg3 :: Bool
verify_deg3 = all (λl → charge_deg3 l ≥ 0) lens
  where
    lens = [(f1, f2, f3) | f1 ← [5 .. Δ*],
                           f2 ← [f1 .. Δ*],
                           f3 ← [f2 .. Δ*],
                           cdeg [f1, f2, f3] ≥ Δ* + 2]

charge_deg3 (f1, f2, f3) = -1 + ruleE f1 (f2 `min` f3)
                              + ruleE f2 (f3 `min` f1)
                              + ruleE f3 (f1 `min` f2)
                              - through_heavy (f1 `min` f2)
                              - through_heavy (f2 `min` f3)
                              - through_heavy (f3 `min` f1)
```

2 Final charge of 4-vertices

The initial charge of a 4-vertex v is 0. If v is incident with at least one (≤ 4)-face, then its charge is not affected by any of the rules (it is not affected by $\texttt{iso}_\ell$ and $E_{\ell,i}$ rules, as the sink of v is the incident (≤ 4)-face). Otherwise, v receives charge by $\texttt{iso}_\ell$ and $E_{\ell,i}$ rules, and sends charge by `through_heavy` rules. Again, it is easy to verify that the resulting charge is non-negative:

```
verify_deg4 :: Bool
verify_deg4 = all (λl → charge_deg4 l ≥ 0) lens
  where
    lens = [(f1, f2, f3, f4) | f1 ← [5 .. Δ*],
```

```
                        f2 ← [f1 .. Δ*],
                        f3 ← [f1 .. Δ*],
                        f4 ← [f2 .. Δ*],
                        cdeg [f1, f2, f3, f4] ≥ Δ* + 2]

charge_deg4 (f1, f2, f3, f4) = ruleE f1 (f2 `min` f4)
                                 + ruleE f2 (f3 `min` f1)
                                 + ruleE f3 (f4 `min` f2)
                                 + ruleE f4 (f1 `min` f3)
                                 - through_heavy (f1 `min` f2)
                                 - through_heavy (f2 `min` f3)
                                 - through_heavy (f3 `min` f4)
                                 - through_heavy (f4 `min` f1)
```

3 Final charge of 5-vertices

The initial charge of a 5-vertex v is 1. The final charge is affected by the rules specific to 5-vertices, as well as $\texttt{iso}_\ell$ and $E_{\ell,i}$ rules from the faces that do not see any (≤ 4)-face at v, and by `through_heavy` rules. We verify that the resulting charge is non-negative:

```
verify_deg5 :: Bool
verify_deg5 = all (λx → charge_deg5 x ≥ 0) lens_deg5

to_maybe_isolated prev len next
  | prev < 5 || len < 5 || next < 5 = 0
  | otherwise = ruleE len (prev `min` next)

charge_deg5 (3, f2, 3, f4, f5) =
   1 - five_to_two_triangles (f2 ≤ 7 && f4 ≤ 7)
     - five_to_two_triangles (f2 ≤ 7 && f5 ≤ 7)
charge_deg5 (3, f2, f3, f4, f5) =
   1 - five_to_single_triangle n4 - (toInteger n4 % 1) * big_to_four
     + to_maybe_isolated f2 f3 f4
     + to_maybe_isolated f3 f4 f5
     - th * through_heavy Δ*
 where
   n4 = length $ filter (== 4) [f3, f4]
   th34 = boolTo01 (f3 ≥ 11 && f4 ≥ 11)
```

```
   th23 = boolTo01 (f2 ≥ 5 && f3 ≥ 5)
   th45 = boolTo01 (f4 ≥ 5 && f5 ≥ 5)
   th = th23 + th34 + th45
charge_deg5 (f1, f2, f3, f4, f5) =
   1 - (toInteger n4 % 1) * big_to_four
     + to_maybe_isolated f1 f2 f3
     + to_maybe_isolated f2 f3 f4
     + to_maybe_isolated f3 f4 f5
     + to_maybe_isolated f4 f5 f1
     + to_maybe_isolated f5 f1 f2
     - (toInteger th % 1) * through_heavy Δ*
 where
   flist = [f1, f2, f3, f4, f5]
   n4 = length $ filter (== 4) flist
   th = length $ filter (≥ 11)
               $ zipWith min flist (tail flist ++ [head flist])

lens_deg5 =
  do
    f1 ← [3 .. Δ*]
    let min_nxt = if f1 == 3 then 5 else f1
    f2 ← [min_nxt .. Δ*]
    f5 ← [min_nxt .. Δ*]
    f3 ← [f1 .. Δ*]
    f4 ← [f3 .. Δ*]
    guard (f3 + f4 ≥ 8)
    guard (cdeg [f1, f2, f3, f4, f5] ≥ Δ* + 2)
    return (f1, f2, f3, f4, f5)
```

4 Final charge of 6-vertices

The initial charge of a 6-vertex v is 2. The final charge is affected by the rules specific to 6-vertices, as well as $\texttt{iso}_\ell$ and $E_{\ell,i}$ rules from the faces that do not see any (≤ 4)-face at v, and by `through_heavy` rules. We verify that the resulting charge is non-negative:

```
verify_deg6 :: Bool
verify_deg6 = all (λx → charge_deg6 x ≥ 0) lens_deg6
```

```
charge_deg6 (f1, f2, f3, f4, f5, f6) =
   2 - (toInteger n4 % 1) * big_to_four
     + to_maybe_isolated f1 f2 f3
     + to_maybe_isolated f2 f3 f4
     + to_maybe_isolated f3 f4 f5
     + to_maybe_isolated f4 f5 f6
     + to_maybe_isolated f5 f6 f1
     + to_maybe_isolated f6 f1 f2
     - to_triangles
     - (toInteger th % 1) * through_heavy Δ*
 where
   flist = [f1, f2, f3, f4, f5, f6]
   to_triangles = case (f1, f3, f4, f5) of
                    (3,3,_,3) → six_to_three_triangles f2 f6
                                + six_to_three_triangles f2 f4
                                + six_to_three_triangles f4 f6
                    (3,_,3,_) → six_to_two_opp_triangles
                    (3,_,_,_) → six_to_le2_adj_triangles
                    _ → 0
   tht = case (f1, f3, f4, f5) of
           (3,3,_,3) → 0
           (3,_,3,_) → 2 * (boolTo01 (tht_eligible f2 f3)
                       + boolTo01 (tht_eligible f5 f6))
           (3,3,_,_) → boolTo01 (tht_eligible f4 f5)
                       + boolTo01 (tht_eligible f5 f6)
           (3,_,_,_) → boolTo01 (tht_eligible f2 f3)
                       + boolTo01 (tht_eligible f5 f6)
           _ → 0
   n4 = length $ filter (== 4) flist
   thb = length $ filter (≥ 11)
                $ zipWith min flist (tail flist ++ [head flist])
   th = thb + tht

tht_eligible f1 f2 = 5 ≤ f1 && 5 ≤ f2 && (f1 `min` f2) ≤ 10

lens_deg6 =
  do
    f1 ← [3 .. Δ*]
    let min_nxt = if f1 == 3 then 5 else f1
    f2 ← [min_nxt .. Δ*]
```

```
f6 ← [min_nxt .. Δ*]
f3 ← [f1 .. Δ*]
f5 ← [f3 .. Δ*]
f4 ← [f1 .. Δ*]
guard (f3 + f4 ≥ 8)
guard (f4 + f5 ≥ 8)
guard (cdeg [f1, f2, f3, f4, f5, f6] ≥ Δ* + 2)
return (f1, f2, f3, f4, f5, f6)
```

5 Final charge of (≥ 7)-vertices

Let v be a vertex of degree $d \geq 7$. Let $f_1, \ldots, f_d$ be the faces incident with v in order. For $1 \leq i \leq d$, let c_i consist of the following contributions to the final charge of v:

- If $|f_i| = 3$,
  - the amount of charge sent from v to f_i according to the rules for heavy vertices, and
  - the amount of charge sent from v due to f_i by through_heavy rules to the edges e of faces neighboring with f_i such that $5 \leq m(e) \leq 10$.
- If $|f_i| = 4$, the amount of charge v sends to f_i according to the rules for heavy vertices.
- If $|f_i| \geq 5$,
  - the charge received from f_i according to the $\texttt{iso}_\ell$ and $E_{\ell,i}$ rules, and
  - half the charge sent to incident edges e with $m(e) \geq 11$ using through_heavy rules.

Note that c_i depends only on $|f_{i-2}|, \ldots, |f_{i+2}|$ (with indices modulo d), and that the total change of charge of v is $\sum_{i=1}^{d} c_i$. Let $q_i = \frac{1}{2}c_{i-1} + c_i + \frac{1}{2}c_{i+1}$, with indices modulo d. By the case analysis below, $q_i \leq 1$, and thus $\sum_{i=1}^{d} c_i = \frac{1}{2}\sum_{i=1}^{d} q_i \leq \frac{d}{2}$. Hence, if $d \geq 8$, then the final charge of v is at least $d - 4 - \frac{d}{2} \geq 0$.

Suppose that $d = 7$. Call an index i *needy* if either $|f_{i-1}| = |f_{i+1}| = 3$, or $|f_{i-2}| = |f_i| = |f_{i+2}| = 3$, and *tame* otherwise. If v is incident with at most two triangles, then there are at least 4 tame indices i such that $|f_i| > 3$. In the case analysis below, we get a better estimate for q_i of such indices. These better estimates imply that $\sum_{i=1}^{7} q_i \leq 6$, and thus the amount of charge send by v is at most 3 and the final charge of v is non-negative. If v is incident with three triangles, then it sends `6_to_tri_3_light` to one of them, `6_to_tri_le2_adj` to the other two, and at most twice $\texttt{through\_heavy}_{\Delta^\star}$ to the edge not incident with any triangle. The sum of these charges is at most 3, and thus the final charge of v is non-negative.

```
verify_big :: Bool
verify_big = maxRelevantQ ≤ 1
             && 3 ∗ maxRelevantQ + 4 ∗ maxTameLQ ≤ 6
             && six_to_three_triangles 5 5
                + 2 ∗ six_to_le2_adj_triangles
                + 2 ∗ through_heavy Δ⋆ ≤ 3

maxRelevantQ = maximum $ map computeQ relevantAngles
maxTameLQ = maximum $ map computeQ tameOnLonger

type Angle = Array Int (Maybe Int)

computeQ :: Angle→Rational
computeQ st = computeC (-1) st / 2
              + computeC 0 st
              + computeC 1 st / 2

computeC :: Int → Angle → Rational
computeC at st =
  let rst = array (-2,2) [(i, fromJust (st ! (at + i))) | i ← [-2 .. 2]]
   in case rst ! 0 of
        3 → triangleContr (rst ! (-2)) (rst ! 2)
            + thContr (rst ! (-1)) (rst ! (-2))
            + thContr (rst ! 1) (rst ! 2)
        4 → big_to_four
        _ → -to_maybe_isolated (rst ! (-1)) (rst ! 0) (rst ! 1)
            + normalTh (rst ! 0) (rst ! 1) / 2
            + normalTh (rst ! 0) (rst ! (-1)) / 2
  where
```

```
    triangleContr 3 3 = six_to_three_triangles 5 5
    triangleContr x y
      | x ≤ 4 || y ≤ 4 = six_to_le2_adj_triangles
    triangleContr _ _ = six_to_two_opp_triangles
    thContr p1 p2
      | p2 ≤ 4 = 0
      | p1 ≥ 11 && p2 ≥ 11 = 0
      | otherwise = through_heavy Δ*
    normalTh p1 p2 = if p1 ≥ 11 && p2 ≥ 11
                       then through_heavy Δ*
                       else 0

relevantAngles :: [Angle]
relevantAngles =
  do
    let gener = array (-4, 4) [(i, Nothing) | i ←[-4 .. 4]]
    mid ← relevantForC 0 gener
    l ← relevantForC (-1) mid
    relevantForC 1 l

tameAngles :: [Angle]
tameAngles = filter isTame relevantAngles
  where
    isTame an
        | fj 0 == 3 = fj (-2) /= 3 || fj 2 /= 3
        | otherwise = fj (-1) /= 3 || fj 1 /= 3
      where
        fj i = fromJust (an ! i)
tameOnLonger = filter (λan → an ! 0 /= Just 3) tameAngles

relevantForC :: Int→Angle→[Angle]
relevantForC at st =
  do
    st' ← extend at st
    case fromJust (st' ! at) of
      3 → do
            st1 ← extend (at - 1) st'
            st2 ← extend (at - 2) st1
            st3 ← extend (at + 1) st2
            extend (at + 2) st3
```

```
      4 → return st'
      _ → do
            st1 ← extend (at - 1) st'
            extend (at + 1) st1
  where
    extend i ast =
      do
        iVal ← possibleChoices (ast ! i) (ast ! (i-1)) (ast ! (i+1))
        return $ ast // [(i, Just iVal)]
    possibleChoices (Just x) _ _ = [x]
    possibleChoices _ a b = [lbn a `max` lbn b .. Δ*]
    lbn (Just 3) = 5
    lbn (Just 4) = 4
    lbn _ = 3
```

6 Final charge of triangles

Let $Q = v_1v_2v_3$ be a triangle. For $i = 1, 2, 3$, let f_i denote the face incident with v_iv_{i+1} distinct from Q. Let N_i denote the list of lengths of faces at v_i in order, excluding f_i, Q, and f_{i-1}. Firstly, we distinguish all possible lengths of faces f_1, f_2, and f_3, and for the first and the last elements of N_i, we decide whether they have length 3, 4, or larger. We call this information the *shape* of Q. This gives us enough information to determine the amount of charge Q loses to incident 3-vertices, as well as the amount of charge received from the incident faces by the basic rules, the `short_to_lightA` rules, the `face_to_lightA` rules, the `light_C_extra` rules, the rule `four2`, rules `⋆_CC_to_5_extra` and `⋆_CC_to_11_extra`, and the rule `10_to_13_A_extra`.

```
data NListElt = NLThree | NLFour | NLLarge deriving (Eq, Show)
type NList = [NListElt]
nLists :: [NList]
nLists = [[]] ++ [[a] | a ← pos] ++ [[a,b] | a ← pos, b ← pos]
  where
    pos = [NLThree, NLFour, NLLarge]
type TriangleShape = (Int, NList, Int, NList, Int, NList)

triangleShapes :: [TriangleShape]
triangleShapes =
  do
```

```
    l1 ← [5 .. Δ*]
    v1 ← nLists
    l2 ← [l1 .. Δ*]
    guard (estCdeg l1 v1 l2 ≥ Δ* + 2)
    v2 ← nLists
    l3 ← [l2 .. Δ*]
    guard (estCdeg l2 v2 l3 ≥ Δ* + 2)
    v3 ← nLists
    guard (estCdeg l3 v3 l1 ≥ Δ* + 2)
    let sh = (l1, v1, l2, v2, l3, v3)
    guard (not $ reduTRIANGLE0 sh)
    guard (not $ reduTRIANGLE1 sh)
    guard (not $ reduTRIANGLE2 sh)
    return sh
  where
   estCdeg l1 [] l2 = cdeg [3, l1, l2]
   estCdeg l1 [NLThree] l2 = cdeg [3, l1, 3, l2]
   estCdeg l1 [NLFour] l2 = cdeg [3, l1, 4, l2]
   estCdeg l1 [NLLarge] l2 = Δ* + 2
   estCdeg l1 [_,_] l2 = Δ* + 2
   reduTRIANGLE0 (l1, v1, l2, v2, l3, v3)
     | bg v1 || bg v2 || bg v3 = False
     | null v1 && cdeg [3, l1, l2] ≤ Δ* + 2 = True
     | null v2 && cdeg [3, l2, l3] ≤ Δ* + 2 = True
     | null v3 && cdeg [3, l3, l1] ≤ Δ* + 2 = True
     | otherwise = False
   reduTRIANGLE1 ts = reduTRIANGLE1s ts
                      || reduTRIANGLE1s rs
                      || reduTRIANGLE1s (rotShape rs)
     where
       rs = rotShape ts
   reduTRIANGLE1s (l1, v1, l2, v2, l3, v3)
     | not (null v2) || not (null v3) || v1 /= [NLFour] || t ≥ 2 = False
     | otherwise = cdeg [3, l1, 4, l2] ≤ 2 * Δ* + 3 - t - l3
     where
       t = cdeg [3, l1, l3] + cdeg [3, l2, l3] - 2 * Δ* - 4
   reduTRIANGLE2 ts = reduTRIANGLE2s ts
                      || reduTRIANGLE2s rs
                      || reduTRIANGLE2s (rotShape rs)
     where
```

```
      rs = rotShape ts
  reduTRIANGLE2s (l1, v1, l2, v2, l3, v3)
    | bg v3 = False
    | null v1 && v2 == [NLFour]
      && cdeg [3,l1,l2] == Δ* + 2
      && cdeg [3,l2,4,l3] ≤ Δ* + 3 = True
    | null v2 && v1 == [NLFour]
      && cdeg [3,l2,l3] == Δ* + 2
      && cdeg [3,l1,4,l2] ≤ Δ* + 3 = True
    | otherwise = False
  bg [] = False
  bg [NLThree] = False
  bg _ = True

rotShape (l1, v1, l2, v2, l3, v3) = (l2, v2, l3, v3, l1, v1)

shapeCharge :: TriangleShape → Rational
shapeCharge ts = -1 + shapeChargeOne ts
                    + shapeChargeOne rts
                    + shapeChargeOne rrts
  where
    rts = rotShape ts
    rrts = rotShape rts

shapeChargeOne (l1, v1, l2, v2, l3, v3) =
    -d3 + basic + sla + fla + lca + f2 - cct + ten
  where
    d3 = if null v1 then 1 else 0
    isA = null v1 && null v2 && null v3
    isB = null v1 && null v2 && v3 == [NLThree]
    isC = not isA && not isB
    etp = aet (reverse v1)
    etn = aet v2
    aet [] = Deg3
    aet (NLThree:_) = DG4_Short
    aet (NLFour:_) = DG4_Short
    aet _ = DG4_Long
    basic = if isA then ruleA l2
            else if isB then ruleB l2
            else ruleC etp l2 etn
```

```
sla = if isA && 7 ≤ l2 && l2 ≤ 8
         then short_to_lightA l2 l1 l3
         else 0
fla = if isA && 9 ≤ l2 && l2 ≤ 11
         then flaVal
         else 0
flaVal
  | (l1 `min` l3) == Δ* + 6 - l2
    && (l1 `max` l3) ≤ Δ* + 8 - l2
       = let tl = l1 + l2 == Δ* + 6
             tr = l3 + l2 == Δ* + 6
          in face_to_lightA l2 (tl && tr)
  | otherwise = 0
lca = if isC && 6 ≤ l2 && l2 ≤ 7 then lcaVal else 0
lcaVal = (if null v1 then light_C_extra l2 l1 else 0)
         + (if null v2 then light_C_extra l2 l3 else 0)
f2 = if v1 == [NLFour] && l1 ≥ Δ* - 1 && l2 ≥ Δ* - 1
        then four2
        else 0
cct = if l2 == 5 || l2 == 11
        then cctVal v1 v2 + cctVal (reverse v2) (reverse v1)
        else 0
cctVal [] (NLThree:_) = star_CC_extra l2
cctVal _ _ = 0
ten = if isA && l2 == 10 then tenVal else 0
tenVal = (boolTo01 (l1 ≤ 13) + boolTo01 (l3 ≤ 13)) * ten_to_13_A_extra
```

In addition to the shape, we need more information about the face lengths to determine the amount of charge received by E rules through vertices of degree 4, sent or received by `through_heavy` rules, and using the rule `11_to_opp_66tri_extra`. Also, more information is needed to determine the amount of charge received from the incident (≥ 5)-vertices. In all the cases, the charge is due to a specific vertex v incident with Q, and its amount only depends on the lengths of the faces at v (in addition to the shape of the face). Hence, we can independently determine the worst case at each vertex of Q.

```
triangle_worst_case :: TriangleShape→Rational
triangle_worst_case ts = shapeCharge ts
                       + triangleWCV ts
                       + triangleWCV rts
                       + triangleWCV rrts
```

```
  where
    rts = rotShape ts
    rrts = rotShape rts

triangleWCV :: TriangleShape→Rational
triangleWCV ts@(l1, v1, l2, v2, l3, v3)
  | null v1 || v1 == [NLThree] || v1 == [NLFour] = 0
  | v1 == [NLLarge] = minimum $ triangleWCV4ch ts
  | otherwise = minimum $ triangleWCV5ch ts ++ triangleWCV6ch ts

triangleWCV4ch :: TriangleShape→[Rational]
triangleWCV4ch (l1, v1, l2, v2, l3, v3) =
  do
    lf ← [5 .. Δ*]
    guard (cdeg [3, l1, lf, l2] ≥ Δ* + 2)
    guard (not $ reduTRIANGLE1 lf)
    return $ isol lf - th lf + opp lf
  where
    isol lf = ruleE lf (l1 `min` l2)
    th lf = through_heavy (l1 `min` lf) + through_heavy (l2 `min` lf)
    opp lf = if lf == 11 && l1 ≤ 6 && l2 ≤ 6
                then eleven_to_opp_66tri_extra
                else 0
    reduTRIANGLE1 lf
      | not (null v2) || not (null v3) || t ≥ 2 = False
      | otherwise = cdeg [3, l1, lf, l2] ≤ 2 * Δ* + 3 - t - l3
      where
        t = cdeg [3, l1, l3] + cdeg [3, l2, l3] - 2 * Δ* - 4

triangleWCV5ch :: TriangleShape→[Rational]
triangleWCV5ch ts@(l1, v1@[n1,n2], l2, v2, l3, v3)
  | not compat = []
  | n2 == NLThree = triangleWCV533ch (l2, rv1, l1, rv3, l3, rv2)
  | n1 == NLThree = triangleWCV533ch ts
  | otherwise = return (ac l1 n1 + ac l2 n2 + bigv)
 where
  rv1 = reverse v1
  rv2 = reverse v2
  rv3 = reverse v3
  compat
```

```
    | n1 == NLThree && n2 /= NLLarge = False
    | n2 == NLThree && n1 /= NLLarge = False
    | n1 == NLFour && n2 == NLFour
      && cdeg [3, l1, 4, 4, l2] < Δ* + 2 = False
    | otherwise = True
  ac l NLFour = 0
  ac l NLLarge = if l ≤ 10 then through_heavy Δ* else 0
  bigv = five_to_single_triangle (boolTo01 (n1 == NLFour)
                                  + boolTo01 (n2 == NLFour))

triangleWCV533ch :: TriangleShape→[Rational]
triangleWCV533ch (l1, v1, l2, v2, l3, v3) =
  do
    lf ← [5 .. Δ*]
    guard (cdeg [3, l1, 3, lf, l2] ≥ Δ* + 2)
    guard (not $ reduTRIANGLE1 lf)
    return $ th lf + bigv
 where
  reduTRIANGLE1 lf
    | not (null v2) || not (null v3) || t ≥ 2 = False
    | otherwise = cdeg [3, l1, 3, lf, l2] ≤ 2 * Δ* + 3 - t - l3
    where
      t = cdeg [3, l1, l3] + cdeg [3, l2, l3] - 2 * Δ* - 4
  th lf
    | lf ≥ 11 && l2 ≥ 11 = -(through_heavy (lf `min` l1)) / 2
    | l2 ≥ 11 = -through_heavy l1
    | otherwise = through_heavy lf
  bigv = five_to_two_triangles (l1 ≤ 7 && l2 ≤ 7)

triangleWCV6ch :: TriangleShape→[Rational]
triangleWCV6ch (l1, [n1,n2], l2, v2, l3, v3) =
  return (ac l1 n1 + ac l2 n2 + bigv)
 where
  ac l NLLarge = if l ≤ 10 then through_heavy Δ* else 0
  ac l _ = 0
  bigv
    | n1 == NLThree && n2 == NLThree = six_to_three_triangles l1 l2
    | n1 == NLLarge && n2 == NLLarge = six_to_two_opp_triangles
    | otherwise = six_to_le2_adj_triangles
```

```
verify_triangle :: Bool
verify_triangle = all (λts → triangle_worst_case ts ≥ 0) triangleShapes
```

7 Final charge of 4-faces

The *shape* of a 4-face is defined analogously to the shape of a triangle. The shape gives enough information to determine the charge sent to incident 3-vertices, received from the incident faces by the basic rules and the `light_D_extra` rules, the charge received from incident (≥ 5)-faces, the charge sent by the rule `four2`, and the charge sent or received by the rule `four2`. We can also upper bound the charge sent by the `through_heavy` rules, and lower bound the charge received by $\texttt{iso}_\ell$ and $E_{\ell,i}$ rules.

```
type FFShape = (Int, NList, Int, NList, Int, NList, Int, NList)

ffShapes :: [FFShape]
ffShapes =
  do
    l1 ← [4 .. Δ*]
    v1 ← nLists
    l2 ← [l1 .. Δ*]
    guard (validNbr l1 v1 l2)
    guard (estCdeg l1 v1 l2 ≥ Δ* + 2)
    v2 ← nLists
    l3 ← [l1 .. Δ*]
    guard (validNbr l2 v2 l3)
    guard (estCdeg l2 v2 l3 ≥ Δ* + 2)
    v3 ← nLists
    l4 ← [l2 .. Δ*]
    guard (validNbr l2 v3 l4)
    guard (estCdeg l3 v3 l4 ≥ Δ* + 2)
    v4 ← nLists
    guard (validNbr l4 v4 l1)
    guard (estCdeg l4 v4 l1 ≥ Δ* + 2)
    let sh = (l1, v1, l2, v2, l3, v3, l4, v4)
    guard (not $ reduFOUR0 sh)
    guard (not $ reduFOUR1 sh)
    guard (not $ reduFOUR2 sh)
    return sh
```

```
 where
  estCdeg l1 [] l2 = cdeg [4, l1, l2]
  estCdeg l1 [NLThree] l2 = cdeg [4, l1, 3, l2]
  estCdeg l1 [NLFour] l2 = cdeg [4, l1, 4, l2]
  estCdeg l1 [NLLarge] l2 = Δ* + 2
  estCdeg l1 [_,_] l2 = Δ* + 2
  validNbr _ [] _ = True
  validNbr l1 v l2 = validOne l1 (head v) && validOne l2 (last v)
  validOne 4 NLThree = False
  validOne _ _ = True
  reduFOUR0 (l1, v1, l2, v2, l3, v3, l4, v4)
    | not (null v1 && null v2 && null v3 && null v4) = False
    | otherwise = l1 + l3 ≤ 9 || l2 + l4 ≤ 9
  reduFOUR1 (l1, v1, l2, v2, l3, v3, l4, v4)
    | not (null v1 && null v2 && null v3 && null v4) = False
    | l1 == 5 && l3 == 5 = l2 ≤ Δ* - 1 || l4 ≤ Δ* - 1
    | l2 == 5 && l4 == 5 = l1 ≤ Δ* - 1 || l3 ≤ Δ* - 1
    | otherwise = False
  reduFOUR2 (l1, v1, l2, v2, l3, v3, l4, v4)
    | not (null v1 && null v2 && null v3 && null v4) = False
    | l1 == 6 && l3 == 6 = l2 ≤ Δ* - 1
                           && l4 ≤ Δ* - 1
                           && (l2 `min` l4 ≤ Δ* - 2)
    | l2 == 6 && l4 == 6 = l1 ≤ Δ* - 1
                           && l3 ≤ Δ* - 1
                           && (l1 `min` l3 ≤ Δ* - 2)
    | otherwise = False

rotFFShape (l1, v1, l2, v2, l3, v3, l4, v4) = (l2, v2, l3, v3, l4, v4, l1, v1)

ffShapeCharge :: FFShape → Rational
ffShapeCharge ts = ffChargeOne ts
                 + ffChargeOne r1
                 + ffChargeOne r2
                 + ffChargeOne r3
  where
    r1 = rotFFShape ts
    r2 = rotFFShape r1
    r3 = rotFFShape r2
```

```
ffChargeOne (l1, v1, l2, v2, l3, v3, l4, v4) =
    -d3 + ge5 + basic + eis + lda - f2 + f1rec - f1send - th - ft5
  where
    d3 | null v1 = if l1 == 4 || l2 == 4 then 1 % 2 else 1
       | otherwise = 0
    ge5 = if length v1 == 2 then big_to_four else 0
    isG = null v1 && null v2 && null v3 && null v4 && l4 == 4
    isD = not isG
    etp = aet (reverse v1) l1
    etn = aet v2 l3
    aet [] l
      | l <= 4 = DG4_Short
      | otherwise = Deg3
    aet (NLThree:_) _ = DG4_Short
    aet (NLFour:_) _ = DG4_Short
    aet _ _ = DG4_Long
    basic
      | l2 == 4 = 0
      | otherwise = if isG then ruleG l2 else ruleD etp l2 etn
    eis
      | v1 == [NLLarge] && l1 > 4 && l2 > 4 = ruleE lbnd (l1 `min` l2)
      | otherwise = 0
      where
        lbndUA = Δ* + 6 - l1 - l2
        lbnd = if lbndUA < 5 then 5 else lbndUA
    lda = if isD && l2 == 6 then ldaVal else 0
    ldaVal = (if null v1 then light_D_extra l2 l1 else 0)
             + (if null v2 then light_D_extra l2 l3 else 0)
    f2 = if v1 == [NLThree]
           && l1 >= Δ* - 1
           && l2 >= Δ* - 1
         then four2 else 0
    f1rec
      | v1 == [NLLarge] && v2 == [NLLarge]
        && l1 == 4 && l2 == 4 && l3 == 4 = four1
      | otherwise = 0
    f1send
      | v1 == [NLFour] && v2 == [NLFour] && l2 == 4 = four1
      | otherwise = 0
    th
```

```
      | null v1 = through_heavy (l1 `min` l2)
      | v1 == [NLLarge] && l1 > 4 && l2 > 4 =
               through_heavy l1 + through_heavy l2
      | v1 == [NLLarge] = (through_heavy l1 + through_heavy l2) / 2
      | otherwise = 0
    ft5
      | null v3 && null v4 && l4 ≤ 5 = 0
      | null v1 && null v2 && l2 == 5 = four_to_five
      | otherwise = 0

verify_four_face :: Bool
verify_four_face = all (λfs → ffShapeCharge fs ≥ 0) ffShapes
```

8 Final charge of ℓ-faces for $5 \leq \ell \leq 13$

Firstly, we enumerate all possible numbers of various types of edges incident with the face—weak faces, i.e., A-triangles, B-triangles or columns; and small faces (other triangles and 4-faces) distinguished by the number of their vertices that are in another small face. Note that a weak face is not consecutive to another weak or small face. This information (which we call a *rough inventory*) of the face is sufficient to determine the final charge of ($\geq$12)-faces.

```
data FaceType = FTWeak | FTSmall0 | FTSmall1 | FTSmall2
                deriving (Eq, Ord, Ix, Show)
type RoughInventory = Array FaceType Int

inventories :: Int → [RoughInventory]
inventories len =
  return (array (FTWeak,FTSmall2)
                [(FTWeak,0),(FTSmall0,0),
                 (FTSmall1,0),(FTSmall2,len)])
  `mplus`
  do
    wk ← [0 .. (len `div` 2)]
    -- reducible configuration TRIANGLE_0
    guard $ not (len == 5 && wk > 0)
    -- reducible configuration SIX_0
    guard $ not (len == 6 && wk > 1)
    small0 ← [0 .. len]
```

```
    guard $ 2 * small0 + 2 * wk ≤ len
    small1 ← [0, 2 .. len]
    let occ = 2 * small0 + 2 * wk + (3 * small1 `div` 2)
    guard $ occ ≤ len
    small2 ← [0 .. len - occ]
    guard $ not (small1 == 0 && small2 > 0)
    return (array (FTWeak,FTSmall2)
                  [(FTWeak,wk),(FTSmall0,small0),
                  (FTSmall1,small1),(FTSmall2,small2)])

riIsolated :: Int → RoughInventory → Int
riIsolated len inv = len - 2 * nsin - ncha - (inv ! FTSmall2)
  where
    nsin = inv ! FTSmall0 + inv ! FTWeak
    ncha = 3 * inv ! FTSmall1 `div` 2

chargeGe12 :: Int → RoughInventory → Rational
chargeGe12 len inv = (toInteger len % 1) - 4 - sent
  where
    niso = riIsolated len inv
    nxBig = 2 * inv ! FTSmall0 + inv ! FTSmall1
    nxSmall = inv ! FTSmall1 + 2 * inv ! FTSmall2
    sent = toInteger (inv ! FTWeak) % 1 * weak len
           + toInteger nxSmall % 1 * small len True
           + toInteger nxBig % 1 * small len False
           + toInteger niso % 1 * iso len
```

On ($\leq$11)-faces, we need to know a more detailed inventory information, distinguishing the weak faces into A-triangles (also specifying whether the vertex opposite to the face is adjacent to another vertex of degree three incident with a triangle, which is necessary to resolve the through_heavy contributions), B-triangles or columns, etc. This is no longer sufficient to determine the exact final charge of the face, but it gives us a lower bound, which enables us to filter out most of the options.

```
data DetFace = TriaA Bool | TriaB | Column
             | TriaC AdjEdgeType AdjEdgeType
             | Four AdjEdgeType AdjEdgeType
             | Iso | Long Int
             deriving (Eq, Ord, Show)
type Inventory = M.Map DetFace Int
```

```
ftBasic len (TriaA True) = ruleA len
ftBasic len (TriaA False) = ruleA len - through_heavy (Δ* + 6 - len)
ftBasic len TriaB = ruleB len
ftBasic len Column = ruleG len
ftBasic len (TriaC a1 a2) = ruleC a1 len a2
ftBasic len (Four a1 a2) = ruleD a1 len a2
ftBasic len Iso = ruleE len 5

part :: Int → Int → [[Int]]
part 1 x = return [x]
part n x =
  do
    f ← [0 .. x]
    r ← part (n - 1) (x - f)
    return (f:r)

clarify :: Int → RoughInventory→[Inventory]
clarify len ri =
  do
    [at,af,b,g] ← part 4 (ri ! FTWeak)
    [c33, c3l, cll, d33, d3l, dll] ← part 6 (ri ! FTSmall0)
    [cs3, csl, ds3, dsl] ← part 4 (ri ! FTSmall1)
    [css, dss] ← part 2 (ri ! FTSmall2)
    return $ M.fromList [(TriaA True, at), (TriaA False, af),
                         (TriaB, b), (Column, g),
                         (TriaC Deg3 Deg3, c33),
                         (TriaC Deg3 DG4_Long, c3l),
                         (TriaC DG4_Long DG4_Long, cll),
                         (Four Deg3 Deg3, d33),
                         (Four Deg3 DG4_Long, d3l),
                         (Four DG4_Long DG4_Long, dll),
                         (TriaC Deg3 DG4_Short, cs3),
                         (TriaC DG4_Long DG4_Short, csl),
                         (Four Deg3 DG4_Short, ds3),
                         (Four DG4_Long DG4_Short, dsl),
                         (TriaC DG4_Short DG4_Short, css),
                         (Four DG4_Short DG4_Short, dss),
                         (Iso, riIsolated len ri)]
```

```
lbInvCharge :: Int → Inventory → Rational
lbInvCharge len inv =
    toInteger len % 1 - 4 - basic - sla - fla - lca - lda - elop - tta
  where
    basic = M.foldrWithKey basicAcc 0 inv
    basicAcc _ 0 acc = acc
    basicAcc fc k acc = acc + (toInteger k % 1) * ftBasic len fc
    sla
     | len < 7 || len > 8 = 0
     | otherwise = toInteger (inv M.! TriaA True + inv M.! TriaA False) % 1
                   * maxSla len
    fla
     | len < 9 || len > 11 = 0
     | otherwise = toInteger (inv M.! TriaA True + inv M.! TriaA False) % 1
                   * maxFla len
    cends = 2 * inv M.! TriaC Deg3 Deg3
                + inv M.! TriaC Deg3 DG4_Long
                + inv M.! TriaC Deg3 DG4_Short
    dends = 2 * inv M.! Four Deg3 Deg3
                + inv M.! Four Deg3 DG4_Long
                + inv M.! Four Deg3 DG4_Short
    lca
     | len < 6 || len > 7 = 0
     | otherwise = toInteger cends % 1 * maxLca len
    lda
     | len /= 6 = 0
     | otherwise = toInteger dends % 1 * maxLda len
    maxLca len = maximum [light_C_extra len x
                          | x ← [Δ*+5-len .. Δ*]]
    maxLda len = maximum [light_D_extra len x
                          | x ← [Δ*+4-len .. Δ*]]
    maxSla len = maximum [short_to_lightA len x y
                          | x ← [Δ*+6-len .. Δ*],
                            y ← [Δ*+6-len .. Δ*]]
    maxFla len = face_to_lightA len True `max` face_to_lightA len False
    elop
     | len /= 11 = 0
     | otherwise = toInteger (inv M.! Iso) % 1 * eleven_to_opp_66tri_extra
    tta
     | len /= 10 = 0
```

```
    | otherwise = toInteger (inv M.! TriaA True + inv M.! TriaA False) % 1
                  * 2 * ten_to_13_A_extra

relevantInventories :: Int→[Inventory]
relevantInventories len = filter (λinv → lbInvCharge len inv < 0) cis
  where
    ris = inventories len
    cis = concatMap (clarify len) ris

-- further functions do not consider the special case of all incident faces
-- being small; verify that all such cases were proven to be discharged
-- in relevantInventories
verifyAllSmall :: Int → Bool
verifyAllSmall len = all (λinv → lbInvCharge len inv ≥ 0) invs
  where
   ri = array (FTWeak,FTSmall2)
              [(FTWeak,0),(FTSmall0,0),(FTSmall1,0),(FTSmall2,len)]
   invs = clarify len ri
```

Next, we list all possible cyclic orders of the items in the inventories, and assign labels to the neighboring faces.

```
data LBSize = Exact Int | AtLeast Int deriving (Eq,Show)
data Neighborhood = Nbhd {nbElts :: Int,
                          nbItems :: [DetFace],
                          nbFaces :: Array Int LBSize}

instance Show Neighborhood where
  showsPrec _  Nbhd{nbElts = n, nbItems = it} = shows (take n it)

allOrders :: Inventory → [Neighborhood]
allOrders inv = allOrdersExt inv 0 []

allOrdersExt inv nf acc
  | emptyInv inv = let ca = acc ++ ca
                   in [Nbhd {nbElts = length acc, nbItems = ca,
                             nbFaces = array (1,nf)
                                             [(i, AtLeast 5)
                                              | i ← [1..nf]]}]
  | otherwise =
      do
```

```
        (bl, inv') ← getBlock inv
        allOrdersExt inv' (nf + 1) (Long (nf + 1) : bl ++ acc)

emptyInv inv = all (== 0) $ M.elems inv

getBlock :: Inventory → [([DetFace], Inventory)]
getBlock inv =
 do
   (o, nin) ← takeOne (TriaA True) inv `mplus`
              takeOne (TriaA False) inv `mplus`
              takeOne TriaB inv `mplus`
              takeOne Column inv `mplus`
              takeOne (TriaC Deg3 Deg3) inv `mplus`
              takeOne (TriaC Deg3 DG4_Long) inv `mplus`
              takeOne (TriaC DG4_Long DG4_Long) inv `mplus`
              takeOne (Four Deg3 Deg3) inv `mplus`
              takeOne (Four Deg3 DG4_Long) inv `mplus`
              takeOne (Four DG4_Long DG4_Long) inv `mplus`
              takeOne Iso inv
   return ([o], nin)
 `mplus`
   longBlocks inv

takeOne x inv =
   do
     guard (inv M.! x > 0)
     return (x, M.adjust (subtract 1) x inv)

longBlocks :: Inventory → [([DetFace], Inventory)]
longBlocks inv =
  do
    (s, inv') ← starter False inv
    (e, inv'') ← starter True inv'
    (m, ir) ← mids inv''
    return (s : m ++ [e], ir)
  where
    starter rev ain =
      do
        (s,nin) ← takeOne (TriaC Deg3 DG4_Short) ain `mplus`
                  takeOne (TriaC DG4_Long DG4_Short) ain `mplus`
```

```
                    takeOne (Four Deg3 DG4_Short) ain `mplus`
                    takeOne (Four DG4_Long DG4_Short) ain
          return (if rev then revDF s else s, nin)
    revDF (TriaC x y) = TriaC y x
    revDF (Four x y) = Four y x
    mids ain =
      return ([], ain) `mplus`
      do
        (m, nin) ← takeOne (TriaC DG4_Short DG4_Short) ain `mplus`
                    takeOne (Four DG4_Short DG4_Short) ain
        (md, fin) ← mids nin
        return (m : md, fin)

rotations act 0 = [act]
rotations act rem = act : rotations rst (rem - length fp - 1)
  where
    (fp, rst) = break isLong $ tail act
isLong (Long _) = True
isLong _ = False
reverseDF n act = revic
  where
    revi = map revSingle $ reverse $ take n $ tail act
    revic = revi ++ revic

revSingle (TriaC x y) = TriaC y x
revSingle (Four x y) = Four y x
revSingle x = x

rotRevs :: Neighborhood → [[DetFace]]
rotRevs Nbhd{nbElts = n, nbItems = it} = rots ++ revs
  where
    rots = rotations it n
    revs = map (reverseDF n) rots

smallestAmongSymmetric :: Neighborhood → Bool
smallestAmongSymmetric x@Nbhd{nbElts = n, nbItems = it} = all geq (rotRevs x)
  where
    geq act = geq' n act it
    geq' 0 _ _ = True
    geq' k (h1:t1) (h2:t2)
```

```
      | isLong h1 && isLong h2 = geq' (k - 1) t1 t2
      | h1 == h2 = geq' (k - 1) t1 t2
      | otherwise = h1 > h2

orderReducible :: Int → Neighborhood → Bool
orderReducible len nb@Nbhd{nbElts = n, nbItems = it} = not $ any (redu len) rots
  where
    rots = rotRevs nb
    redu 6 x = reduSIX1 x || reduSIX2 x || reduSIX3 x
    redu 7 x = reduSEVEN0 x
    redu _ _ = False
    reduSIX1 (Long _ : Column : Long _ : TriaC Deg3 _ : _) = True
    reduSIX1 (Long _ : Column : Long _ : Four Deg3 _ : _) = True
    reduSIX1 _ = False
    reduSIX2 (Long _
              : TriaA True
              : Long _
              : TriaC Deg3 Deg3 : _) = True
    reduSIX2 _ = False
    reduSIX3 (Long _
              : Four Deg3 Deg3
              : Long _
              : TriaA True
              : Long _
              : TriaC Deg3 _ : _) = True
    reduSIX3 (Long _
              : Four Deg3 Deg3
              : Long _
              : TriaA True
              : Long _
              : Four Deg3 _ : _) = True
    reduSIX3 _ = False
    reduSEVEN0 (Long _ : TriaA _
                : Long _ : TriaA True
                : Long _ : TriaA _ : _) = True
    reduSEVEN0 (Long _ : TriaA _
                : Long _ : TriaA True
                : Long _ : TriaB : _) = True
    reduSEVEN0 (Long _ : TriaB
                : Long _ : TriaA True
```

```
                : Long _ : TriaB : _) = True
    reduSEVEN0 _ = False

orders :: Int → [Neighborhood]
orders len = filter (orderReducible len) nsym
  where
    ris = relevantInventories len
    os = concatMap allOrders ris
    nsym = filter smallestAmongSymmetric os
```

Next, we establish lower bounds on the lengths of the neighboring faces, based on the reducible configurations, and filter out again those that we know to have non-negative charge based on this information. This suffices to discharge faces of lengths 6 and 11.

```
d3te (TriaA _) = True
d3te TriaB = True
d3te (TriaC _ Deg3) = True
d3te _ = False

d4te (TriaA _) = True
d4te TriaB = True
d4te (TriaC _ Deg3) = True
d4te (Four _ Deg3) = True
d4te _ = False

d3be (TriaA _) = True
d3be TriaB = True
d3be (TriaC Deg3 Deg3) = True
d3be _ = False

lowerBoundLengths :: Int → Neighborhood → Neighborhood
lowerBoundLengths len nb@Nbhd{nbFaces = fc} = nb{nbFaces = nfc}
  where
    rr = rotRevs nb
    nfc = foldr lbCurrent fc rr
    lbCurrent cur = lbTRIANGLE0 cur
                  ∘ lbSEVEN2 cur
                  ∘ lbSEVEN3 cur
                  ∘ lbEIGHT0 cur
                  ∘ lbEIGHT5 cur
```

```
                ∘ lbGEN2 cur
lbTRIANGLE0 (Long a : TriaA _ : _) = lbnd a (Δ* + 6 - len)
lbTRIANGLE0 (Long a : TriaB : _) = lbnd a (Δ* + 6 - len)
lbTRIANGLE0 (Long a : Column : _) = lbnd a Δ*
lbTRIANGLE0 (Long a : TriaC Deg3 _ : _) = lbnd a (Δ* + 5 - len)
lbTRIANGLE0 (Long a : Four Deg3 _ : _) = lbnd a (Δ* + 4 - len)
lbTRIANGLE0 _ = id
lbSEVEN2 _ | len /= 7 = id
lbSEVEN2 (Long a : TriaA True : Long _ : TriaA True : _) = lbnd a Δ*
lbSEVEN2 _ = id
lbSEVEN3 (Long _ : x : Long _ : TriaA True : Long a : y : _)
           | d3be x && d3te (revSingle y) = lbnd a Δ*
lbSEVEN3 _ = id
lbEIGHT0 _ | len /= 8 = id
lbEIGHT0 (Long _ : TriaA _
          : Long a : TriaA True
          : Long _ : TriaA _ : _) = lbnd a Δ*
lbEIGHT0 (Long _ : TriaA _
          : Long a : TriaA True
          : Long _ : TriaB : _) = lbnd a Δ*
lbEIGHT0 (Long _ : TriaB
          : Long a : TriaA True
          : Long _ : TriaA _ : _) = lbnd a Δ*
lbEIGHT0 (Long _ : TriaB
          : Long a : TriaA True
          : Long _ : TriaB : _) = lbnd a Δ*
lbEIGHT0 _ = id
lbEIGHT5 _ | len /= 8 = id
lbEIGHT5 (Long _ : x : Long _
          : TriaA True : Long _
          : TriaA True : Long a : y : _)
      | d3be x && d3te (revSingle y) = lbnd a (Δ* - 1)
lbEIGHT5 _ = id
lbGEN2 _ | len < 7 = id
lbGEN2 (Long _ : TriaA True
        : Long a : TriaA True : _) = lbnd a (Δ* + 7 - len)
lbGEN2 _ = id
lbnd a x afc | x == Δ* = afc // [(a, Exact Δ*)]
lbnd a x afc = case afc ! a of
                 Exact v | v ≥ x → afc
```

```
                    AtLeast v | v ≥ x → afc
                    AtLeast _ → afc // [(a, AtLeast x)]

lbNeighborhoodCharge :: Int → Neighborhood → Rational
lbNeighborhoodCharge len Nbhd{nbElts = n, nbItems = it, nbFaces = fc} =
toInteger len % 1 - 4 - sent
  where
    rots = take n $ tails it
    sent = sum $ map (seqCharge len fc) rots

geThan (Exact a) = a
geThan (AtLeast a) = a

seqCharge len fc (_ : Long a : TriaA isT : Long b : _ ) =
    ruleA len
    - (if isT then 0 else thCharge (fc!a) (fc!b))
    + extraA len (fc!a) (fc!b)
    + ten13extra len (fc!a)
    + ten13extra len (fc!b)
seqCharge len fc (_ : _ : TriaB : _) = ruleB len
seqCharge len fc (_ : _ : Column : _) = ruleG len
seqCharge len fc (_ : Long a : TriaC Deg3 Deg3 : Long b : _) =
    ruleC Deg3 len Deg3
    + extraC len (fc!a)
    + extraC len (fc!b)
seqCharge len fc (_ : Long a : TriaC Deg3 rt : nx : _)
    | rt /= Deg3 = ruleC Deg3 len rt + extraC len (fc!a) - ccExtra len nx
seqCharge len fc (_ : pv : TriaC lt Deg3 : Long b : _)
    | lt /= Deg3 = ruleC lt len Deg3 + extraC len (fc!b) - ccExtra len pv
seqCharge len fc (_ : _ : TriaC lt rt : _)
    | lt /= Deg3 && rt /= Deg3 = ruleC lt len rt
seqCharge len fc (x : Long a : Four Deg3 Deg3 : Long b : y : _) =
    ruleD Deg3 len Deg3
    + extraD len (fc!a) + extraD len (fc!b)
    - fourFive len (fc!a) (fc!b) (d4te x || d4te (revSingle y))
seqCharge len fc (_ : Long a : Four Deg3 rt : _)
    | rt /= Deg3 = ruleD Deg3 len rt + extraD len (fc!a)
seqCharge len fc (_ : _ : Four lt Deg3 : Long b : _)
    | lt /= Deg3 = ruleD lt len Deg3 + extraD len (fc!b)
seqCharge len fc (_ : _ : Four lt rt : _)
```

```
    | lt /= Deg3 && rt /= Deg3 = ruleD lt len rt
seqCharge len fc (_ : Long a : Iso : Long b : _) =
    ruleE len (geThan (fc!a) `min` geThan (fc!b))
    + elevenOpp len (fc!a) (fc!b)
seqCharge len fc (_ : _ : Long _ : _) = 0

-- by FOUR_1
fourFive 5 x y _ | x == Exact (Δ* - 1)
                 || y == Exact (Δ* - 1) = four_to_five
-- by FOUR_2
fourFive 5 _ _ True = four_to_five
fourFive _ _ _ _ = 0

ccExtra 5 (TriaC _ _) = star_CC_extra 5
ccExtra 11 (TriaC _ _) = star_CC_extra 11
ccExtra _  _ = 0

thCharge lf rf = through_heavy (geThan lf `min` geThan rf)

extraA len lf rf
  | 7 ≤ len && len ≤ 8 = short_to_lightA len (geThan lf) (geThan rf)
  | 9 ≤ len && len ≤ 11 = if mbL || mbR
                          then face_to_lightA len (mbL && mbR)
                          else 0
  | otherwise = 0
 where
  mbL = geThan lf ≤ Δ* + 6 - len
        && geThan rf ≤ Δ* + 8 - len
  mbR = geThan rf ≤ Δ* + 6 - len
        && geThan lf ≤ Δ* + 8 - len

extraC len lb
  | 6 ≤ len && len ≤ 7 = light_C_extra len (geThan lb)
  | otherwise = 0

extraD len lb
  | len == 6 = light_D_extra len (geThan lb)
  | otherwise = 0

elevenOpp 11 lf rf
```

```
  | geThan lf ≥ 7 || geThan rf ≥ 7 = 0
  | otherwise = eleven_to_opp_66tri_extra
elevenOpp _ _ _ = 0

ten13extra 10 lf
  | geThan lf > 13 = 0
  | otherwise = ten_to_13_A_extra
ten13extra _ _ = 0

lenOrders :: Int → [Neighborhood]
lenOrders len = filter (λnb → lbNeighborhoodCharge len nb < 0) los
  where
    os = orders len
    los = map (lowerBoundLengths len) os
```

Finally, for the remaining cases, we enumerate all possible lengths of "interesting" faces that affect the charge sent or received by the face, i.e., next to A-triangles, degree three vertices of C-triangles at (≤ 7)-faces, and degree three vertices of non-column 4-faces at 5-faces, and verify that for all the possible cases, the final charge is non-negative.

```
interestingFaces :: Int → Neighborhood → S.Set Int
interestingFaces len Nbhd{nbElts = n, nbItems = it} = ifs
  where
    rots = take n $ tails it
    ifs = foldr markInt S.empty rots
    markInt (Long a : TriaA _ : Long b : _) aif =
        S.insert a $ S.insert b aif
    markInt (Long a : TriaC Deg3 Deg3 : Long b : _) aif
      | len ≤ 7 = S.insert a $ S.insert b aif
      | otherwise = aif
    markInt (Long a : TriaC Deg3 _ : _) aif
      | len ≤ 7 = S.insert a aif
      | otherwise = aif
    markInt (_ : TriaC _ Deg3 : Long a : _) aif
      | len ≤ 7 = S.insert a aif
      | otherwise = aif
    markInt (Long a : Four Deg3 Deg3 : Long b : _) aif
      | len == 5 = S.insert a $ S.insert b aif
      | otherwise = aif
    markInt (Long a : Four Deg3 _ : _) aif
```

```
      | len == 5 = S.insert a aif
      | otherwise = aif
    markInt (_ : Four _ Deg3 : Long a : _) aif
      | len == 5 = S.insert a aif
      | otherwise = aif
    markInt _ aif = aif

decideFaceLengths :: Int → Neighborhood → [Neighborhood]
decideFaceLengths len nb@Nbhd{nbFaces = fc} =
    map (λf → nb{nbFaces = f}) fcs
  where
    intf = S.toList $ interestingFaces len nb
    fcs = decide fc intf
    decide afc [] = return afc
    decide afc (h:t) =
       do
         ch ← choices (afc ! h)
         let nfc = afc // [(h, Exact ch)]
         decide nfc t
      where
        choices (Exact v) = [v]
        choices (AtLeast v) = [v .. Δ*]

exactReducible :: Int → Neighborhood → Bool
exactReducible len Nbhd{nbElts = n, nbItems = it, nbFaces = fc} = any exred rots
  where
    rots = take n $ tails it
    exred ap
      | len == 7 = reduSEVEN1 ap
      | len == 8 = reduEIGHT1 ap || reduEIGHT2a ap
                   || reduEIGHT2b ap || reduEIGHT3 ap
                   || reduEIGHT4a ap || reduEIGHT4b ap || reduEIGHT6 ap
      | len == 9 = reduGEN0 ap || reduGEN1 ap
      | len == 10 = reduTEN ap || reduGEN0 ap || reduGEN1 ap
      | otherwise = False
    reduSEVEN1 (x : Long a : TriaA True : Long b : y : _) =
          d3te x && fc ! a == Exact (Δ* - 1)
          && fc ! b == Exact (Δ* - 1)
          && d3te (revSingle y)
    reduSEVEN1 _ = False
```

```
reduEIGHT1 (x : Long a : TriaA True : Long b : y : _) =
    d3te x && gls && d3te (revSingle y)
  where
    gls = (fc ! a == Exact (Δ* - 2) && fc ! b /= Exact Δ*)
          || (fc ! b == Exact (Δ* - 2) && fc ! a /= Exact Δ*)
reduEIGHT1 _ = False
reduEIGHT2a (x : Long a : TriaA True : Long b : TriaA _ : Long c : _) =
    d3te x && gls
  where
    gls = fc ! a /= Exact Δ*
          && fc ! b /= Exact Δ*
          && fc ! c /= Exact Δ*
reduEIGHT2a _ = False
reduEIGHT2b (Long a : TriaA _ : Long b : TriaA True : Long c : y : _) =
    d3te (revSingle y) && gls
  where
    gls = fc ! a /= Exact Δ*
          && fc ! b /= Exact Δ*
          && fc ! c /= Exact Δ*
reduEIGHT2b _ = False
reduEIGHT3 (Long a : TriaA True : Long b : TriaA True : Long c : _) =
    gls
  where
    gls = fc ! b /= Exact Δ*
          && (fc ! a == Exact (Δ* - 2)
              || fc ! b == Exact (Δ* - 2))
reduEIGHT3 _ = False
reduEIGHT4a (x : Long a : TriaA True : Long b : y : Long c : _) =
    d3te x && d3be y && gls
  where
    gls = (fc ! a == Exact (Δ* - 2)
           || fc ! b == Exact (Δ* - 2))
          && fc ! c /= Exact Δ*
reduEIGHT4a _ = False
reduEIGHT4b (Long c : y : Long b : TriaA True : Long a : x : _) =
    d3te (revSingle x) && d3be y && gls
  where
    gls = (fc ! a == Exact (Δ* - 2)
           || fc ! b == Exact (Δ* - 2))
          && fc ! c /= Exact Δ*
```

```
    reduEIGHT4b _ = False
    reduEIGHT6 (x : Long a : TriaA True : Long _ : TriaA True : Long b : y : _) =
        d3te x && d3te (revSingle y) && gls
      where
        gls = (fc ! a == Exact (Δ* - 2) && fc ! b /= Exact Δ*)
              || (fc ! b == Exact (Δ* - 2) && fc ! a /= Exact Δ*)
    reduEIGHT6 _ = False
    reduTEN (x : Long a : TriaA True : Long b : TriaA True : Long c : y : _) =
        d3te x && d3te (revSingle y) && gls
      where
        gls = fc ! b == Exact (Δ* - 3)
              && not (geThan (fc ! a) ≥ (Δ* - 1)
                      && geThan (fc ! c) ≥ (Δ* - 1))
    reduTEN _ = False
    reduGEN0 (x : Long a : TriaA True : Long b : y : _) =
        d4te x && d4te (revSingle y) && gls
      where
        gls = (fc ! a == Exact (Δ* + 6 - len)
               || fc ! b == Exact (Δ* + 6 - len))
              && not (geThan (fc ! a) ≥ Δ* + 9 - len
                      || geThan (fc ! b) ≥ Δ* + 9 - len)
    reduGEN0 _ = False
    reduGEN1 (x : Long a : TriaA True : Long b : y : _) =
        (d3te x || d3te (revSingle y)) && gls
      where
        gls = fc ! a == Exact (Δ* + 6 - len)
              && fc ! b == Exact (Δ* + 6 - len)
    reduGEN1 _ = False

exLenOrders :: Int → [Neighborhood]
exLenOrders len = filter (λnb → lbNeighborhoodCharge len nb < 0) rlos
  where
    os = lenOrders len
    los = concatMap (decideFaceLengths len) os
    rlos = filter (not ∘ exactReducible len) los
```

Let us now verify all medium-length faces are discharged

```
verify_medium len
  | len ≥ 12 = all (λri → chargeGe12 len ri ≥ 0) ris
  | len == 6 || len == 11 = null (lenOrders len)
```

```
  | otherwise = null (exLenOrders len)
  where
    ris = inventories len

verify_medium_faces :: Bool
verify_medium_faces = all verifyAllSmall [5..11]
                      && all verify_medium [5..13]
```

9 Final charge of (≥ 14)-faces

Let f be an ℓ-face with $\ell \geq 14$. Let us assign the charge sent by f to its vertices: If v_1v_2 is an edge of f shared with an A-triangle, B-triangle, and column, $\texttt{weak}_\ell/2 = 1 - \frac{4}{\ell}$ is assigned to v_1 and to v_2. If v_1 is an isolated vertex, it is assigned $\texttt{iso}_\ell = 1 - \frac{4}{\ell}$.

If $v_1v_2v_3$ is a path in the boundary of f and v_1v_2 is in a C-triangle or non-column 4-face, then $\texttt{small}_{\ell,a(v_1)}$ is assigned to v_1 and $\texttt{small}_{\ell,a(v_2)}$ is assigned to v_2. If v_2v_3 is also in a triangle or a 4-face Q (which is clearly not an A-triangle, B-triangle or a column), then $a(v_2) = 1$ and we additionally assign $\texttt{small}_{\ell,1}$ to v_2 for Q, and thus v_2 is assigned $2\texttt{small}_{\ell,1} = 1 - \frac{4}{\ell}$ in total. Otherwise, $a(v_2) = 0$ and v_2 is assigned $\texttt{small}_{\ell,0} = 1 - \frac{4}{\ell}$.

Therefore, the charge sent by f is at most $\ell\left(1 - \frac{4}{\ell}\right) = \ell - 4$, and thus the final charge of f is non-negative.

10 Final charge of edges (`through_heavy` rules)

Suppose that $e = uv$ is an edge that sends charge by `through_heavy` rules. If e is an edge with $m(e) \geq 11$, then u is not a vertex of degree three contained in a triangle, e sends $\texttt{through\_heavy}_{m(e)}$ across v, and receives either $\texttt{through\_heavy}_{m(e)}$ or $\texttt{through\_heavy}_{\Delta^s tar} \geq \texttt{through\_heavy}_{m(e)}$ from u or the incident triangles or 4-faces.

If $m(e) \leq 10$, then e sends charge only if there is a triangle $Q = uxy$ such that ux and e are incident with the same face f at u. If either $\deg(u) \geq 6$, or $\deg(u) = 5$ and u is not contained in two triangles, then e sends $\texttt{through\_heavy}_{\Delta^\star}$ to Q, and receives $\texttt{through\_heavy}_{\Delta^\star}$ from u. If $\deg(u) = 5$ and u is contained in another triangle $Q' \neq Q$, then e only sends $\texttt{through\_heavy}_{|f|}$ to Q' when $|f| \geq 11$, and in that case e receives $\texttt{through\_heavy}_{|f|}$ from Q. In all the cases, the final charge of e is non-negative.

11 Charge values

The functions defined here describe the amounts of charge sent by particular rules.

```
weak :: Int → Rational
weak 12 = 4 % 3
weak 13 = 14023 % 10080
weak len | len > 13 = 2 % 1 - 8 % toInteger len

-- small len nt, where nt = True if the next edge
-- on the face is in a ≤ 4-face.
small :: Int → Bool → Rational
small 12 True = 1 % 3
small 12 False = 2 % 3
small 13 True = 6137 % 20160
small 13 False = 14023 % 20160
small len True | len > 13 = 1 % 2 - 2 % toInteger len
small len False | len > 13 = 1 % 1 - 4 % toInteger len

iso :: Int → Rational
iso 12 = 23827 % 36960
iso 13 = 1097 % 1680
iso len | len > 13 = 1 % 1 - 4 % toInteger len

ruleA :: Int → Rational
ruleA 6 = 1 % 1
ruleA 7 = 1 % 1
ruleA 8 = 1 % 1
ruleA 9 = 17383 % 15120
ruleA 10 = 8983 % 7560
ruleA 11 = 4 % 3
ruleA len | len > 11 = weak len

ruleB :: Int → Rational
ruleB 6 = 3 % 4
ruleB 7 = 14 % 15
ruleB 8 = 14 % 15
ruleB 9 = 17383 % 15120
ruleB 10 = 8983 % 7560
ruleB 11 = 4 % 3
```

```
ruleB len | len > 11 = weak len

ruleG :: Int → Rational
ruleG 6 = 3 % 4
ruleG 7 = 9 % 10
ruleG 8 = 82 % 105
ruleG 9 = 2743 % 2520
ruleG 10 = 16217 % 15120
ruleG 11 = 4 % 3
ruleG len | len > 11 = weak len

-- type of the edge following after a triangle or 4-face
-- Deg3: adjacent through a 3-vertex (and thus the face has
--       length ≥ 5 if len < Δ*, since cyclic degree of
--       the vertex is at least Δ* + 2)
-- DG4_Short: adjacent through a ≥4-vertex and length ≤ 4
-- DG4_Long: adjacent through a ≥4-vertex and length ≥ 5
data AdjEdgeType = Deg3 | DG4_Short | DG4_Long deriving (Eq, Ord, Show)

ruleC :: AdjEdgeType → Int → AdjEdgeType → Rational
ruleC DG4_Long 5 DG4_Long = -11507 % 36960
ruleC DG4_Long 5 DG4_Short = -7 % 40
ruleC DG4_Long 5 Deg3 = 349 % 840
ruleC DG4_Short 5 DG4_Short = -1 % 7
ruleC DG4_Short 5 Deg3 = 13 % 30
ruleC Deg3 5 Deg3 = 53 % 120

ruleC DG4_Long 6 DG4_Long = -10 % 33
ruleC DG4_Long 6 DG4_Short = 1 % 336
ruleC DG4_Long 6 Deg3 = 1 % 2
ruleC DG4_Short 6 DG4_Short = 0 % 1
ruleC DG4_Short 6 Deg3 = 1 % 2
ruleC Deg3 6 Deg3 = 97 % 160

ruleC DG4_Long 7 DG4_Long = 2 % 55
ruleC DG4_Long 7 DG4_Short = 1 % 336
ruleC DG4_Long 7 Deg3 = 211 % 336
ruleC DG4_Short 7 DG4_Short = 0 % 1
ruleC DG4_Short 7 Deg3 = 1 % 2
ruleC Deg3 7 Deg3 = 13 % 15
```

```
ruleC x len y | 5 ≤ len && len ≤ 7 = ruleC y len x
ruleC p len n | len > 7 = ruleC_single len p + ruleC_single len n

ruleC_single :: Int → AdjEdgeType → Rational
ruleC_single 8 DG4_Long = 583 % 1680
ruleC_single 8 DG4_Short = 193 % 840
ruleC_single 8 Deg3 = 7 % 15

ruleC_single 9 DG4_Long = 7223 % 30240
ruleC_single 9 DG4_Short = 1517 % 7560
ruleC_single 9 Deg3 = 17383 % 30240

ruleC_single 10 DG4_Long = 83 % 378
ruleC_single 10 DG4_Short = 47851 % 166320
ruleC_single 10 Deg3 = 8983 % 15120

ruleC_single 11 DG4_Long = 17 % 33
ruleC_single 11 DG4_Short = 7 % 22
ruleC_single 11 Deg3 = 3 % 5

ruleC_single len adj | len > 11 = small len (adj == DG4_Short)

ruleD :: AdjEdgeType → Int → AdjEdgeType → Rational
ruleD DG4_Long 5 DG4_Long = 1 % 4
ruleD DG4_Long 5 DG4_Short = 0 % 1
ruleD DG4_Long 5 Deg3 = 349 % 840
ruleD DG4_Short 5 DG4_Short = 1 % 15
ruleD DG4_Short 5 Deg3 = 1 % 8
ruleD Deg3 5 Deg3 = 4 % 7

ruleD DG4_Long 6 DG4_Long = 1 % 4
ruleD DG4_Long 6 DG4_Short = 0 % 1
ruleD DG4_Long 6 Deg3 = 3 % 8
ruleD DG4_Short 6 DG4_Short = -13 % 60
ruleD DG4_Short 6 Deg3 = 1 % 8
ruleD Deg3 6 Deg3 = 67 % 120

ruleD DG4_Long 7 DG4_Long = 1 % 4
ruleD DG4_Long 7 DG4_Short = 0 % 1
```

```
ruleD DG4_Long 7 Deg3 = 3 % 8
ruleD DG4_Short 7 DG4_Short = 3 % 7
ruleD DG4_Short 7 Deg3 = 3281 % 20160
ruleD Deg3 7 Deg3 = 13 % 15

ruleD x len y | 5 ≤ len && len ≤ 7 = ruleD y len x
ruleD p len n | len > 7 = ruleD_single len p + ruleD_single len n

ruleD_single :: Int → AdjEdgeType → Rational
ruleD_single 8 DG4_Long = 41 % 105
ruleD_single 8 DG4_Short = 1 % 4
ruleD_single 8 Deg3 = 1 % 3

ruleD_single 9 DG4_Long = 1009 % 2160
ruleD_single 9 DG4_Short = 5 % 18
ruleD_single 9 Deg3 = 5017 % 10080

ruleD_single 10 DG4_Long = 20743 % 40320
ruleD_single 10 DG4_Short = 0 % 1
ruleD_single 10 Deg3 = 4615 % 8064

ruleD_single 11 DG4_Long = 13 % 22
ruleD_single 11 DG4_Short = 26 % 165
ruleD_single 11 Deg3 = 13 % 22

ruleD_single len adj | len > 11 = small len (adj == DG4_Short)

-- mLen is the minimum of lengths of the other two faces
-- that the current face meets at the vertex
ruleE :: Int → Int → Rational
ruleE len@5 mLen = if mLen < 15 then 1 % 7 else -61 % 240
ruleE len@6 mLen = if mLen < 14 then 49 % 240 else -1 % 15
ruleE len@7 mLen = if mLen < 13 then 79 % 240 else -1 % 15
ruleE len@8 mLen = if mLen < 13 then 41 % 105 else 9 % 28
ruleE len@9 mLen = if mLen < 12 then 7 % 15 else 1 % 3
ruleE len@10 mLen = if mLen < 12 then 7 % 15 else 1 % 3
ruleE len@11 mLen = if mLen < 11 then 7 % 15 else 1 % 3
ruleE len _ | len > 11 = iso len

big_to_four :: Rational
```

```
big_to_four = 1 % 4

-- depends on the number of incident 4-faces
five_to_single_triangle :: Int → Rational
five_to_single_triangle 0 = 767 % 1680
five_to_single_triangle 1 = 737 % 1680
five_to_single_triangle 2 = 37 % 120

-- True if both faces incident with the triangle have length at most 7
five_to_two_triangles :: Bool → Rational
five_to_two_triangles True = 83 % 140
five_to_two_triangles False = 57 % 140

six_to_le2_adj_triangles :: Rational
six_to_le2_adj_triangles = 63 % 80

six_to_two_opp_triangles :: Rational
six_to_two_opp_triangles = 767 % 1680

six_to_three_triangles :: Int → Int → Rational
six_to_three_triangles 6 6 = 8 % 15
six_to_three_triangles x y | x + y ≤ 12 = 113 % 120
six_to_three_triangles _ _ = 881 % 1680

short_to_lightA :: Int → Int → Int → Rational
short_to_lightA len x y | x > y = short_to_lightA len y x
short_to_lightA len x y =
  case (len, Δ⋆ - x, Δ⋆ - y) of
      (7, 1, 1) → 1 % 15
      (7, 1, 0) → 1 % 30
      (7, 0, 0) → 0 % 1
      (8, 2, 2) → 1 % 7
      (8, 2, 1) → 1 % 7
      (8, 2, 0) → 3 % 28
      (8, 1, 1) → 1 % 15
      (8, 1, 0) → 1 % 30
      (8, 0, 0) → 0 % 1

-- True if both adjacent faces have length Δ⋆ + 6 - len
face_to_lightA :: Int → Bool → Rational
```

```
face_to_lightA 9 False = 185 % 6048
face_to_lightA 9 True = 3257 % 30240
face_to_lightA 10 False = 583 % 10080
face_to_lightA 10 True = 583 % 5040
face_to_lightA 11 False = 0 % 1
face_to_lightA 11 True = 0 % 1

light_C_extra :: Int → Int → Rational
light_C_extra len x =
  case (len, Δ* - x) of
    (6, 1) → 0 % 1
    (6, 0) → 0 % 1
    (7, 2) → 1 % 30
    (7, 1) → 1 % 30
    (7, 0) → 0 % 1

light_D_extra :: Int → Int → Rational
light_D_extra len x =
  case (len, Δ* - x) of
     (6, 2) → 1 % 30
     (6, 1) → 1 % 30
     (6, 0) → 0 % 1

through_heavy :: Int → Rational
through_heavy len = if len ≥ 12 then 17 % 80 else 0 % 1

four_to_five :: Rational
four_to_five = 109 % 840

four1 :: Rational
four1 = 1 % 2

four2 :: Rational
four2 = 1 % 2

star_CC_extra :: Int → Rational
star_CC_extra 5 = 37 % 240
star_CC_extra 11 = 14 % 165

eleven_to_opp_66tri_extra :: Rational
```

```
eleven_to_opp_66tri_extra = 28 % 165

ten_to_13_A_extra :: Rational
ten_to_13_A_extra = 89 % 6048
```