Tiling the Plane

Task G, 2004/2005

Tadeusz Sznuk

1 Overview

The solution is based on properties of points A,B,C,D,E,F mentioned in task specification. These properties are proved in section 3. They lead to simple $O(n^3)$ algorithm described in the same section. Section 2 contains various definitions used in section 3. The last section presents an algorithm for checking cyclic equivalence of words, which is used as a part of the the main algorithm (described in section 3).

2 Definitions

Distance Distance between points on the polygon boundary, measured by counterclockwise traversal of the boundary (starting at the first point). Distance between X and Y is denoted by $|XY|$.

Opposite point Opposite point of X is a point $\overline{X}$ such that $|X\overline{X}| = |\overline{X}X|$.

Solution Set $S = \{A, B, C, D, E, F\}$ is called a solution if points A,B,C,D,E,F have the properties mentioned in the task specification.

Solution point Point in a solution. When we say that some points are solution points we mean that they belong to the *same* solution - eg. "if X is a solution point then $\overline{X}$ is also a solution point" means that if $X \in S$ for solution S, then $\overline{X} \in S$. $next(X)$ means next solution point on the polygon boundary ($next(A) = B$), $prev(X)$ means previous point.

String representation We can represent part of the polygon boundary as a string containing letters `N`,`E`,`W`,`S` describing its counterclockwise traversal (using unit steps). We denote such representation of polygon boundary from X to Y by $str(X, Y)$. $compressedStr(X, Y)$ is a sequence of pairs $\langle direction, length\rangle$ equivalent to $str(X, Y)$.

String operations $rev(S)$ is a reverse of string S. $S^{(i)}$ is string S after a circular shift by i positions to the left. $neg(S)$ means string S with every direction changed to the opposite (`N` $\rightarrow$ `S`, `E` $\rightarrow$ `W`, ...). If S is a seqeunce of direction-length pairs, directions inside pairs are changed.

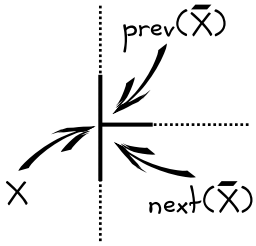
Cyclic equivalence $V_1 \cong V_2$ iff $\exists i : V_1^{(i)} = V_2$.

Evil points Solution point X is called *evil* iff neither X nor $\overline{X}$ is a vertex.

3 Solution

Lemma 1 (♠). *If solution point X is not a vertex then points $prev(\overline{X})$ and $next(\overline{X})$ are vertices.*

"Proof". Look at the tiling. □



Lemma 2 (♢). *If X is evil then all other solution points except $\overline{X}$ are vertices.*

Proof. Apply ♠ to X and $\overline{X}$ (which is not a vertex because X is evil). □

Lemma 3 (♣). *Points X and Y are consecutive points of a solution iff $str(X,Y) = neg(rev(str(\overline{X},\overline{Y})))$ and $str(Y,\overline{X}) \cong neg(rev(str(\overline{Y},X)))$*

Proof. If the solution $\{X,Y,Z,\overline{X},\overline{Y},\overline{Z}\}$ exists, then $str(Y,Z) = neg(rev(str(\overline{Y},\overline{Z})))$ and $str(Z,\overline{X}) = neg(rev(str(\overline{Z},X)))$. Therefore $str(Y,\overline{X}) = str(Y,Z)str(Z\overline{X}) = neg(rev(str(\overline{Y},\overline{Z})))neg(rev(str(\overline{Z},X))) = neg(rev(str(\overline{Z},X)str(\overline{Y},\overline{Z}))) = neg(rev(str(\overline{Y},X)))^{(|Z\overline{X})}$. □

Corollary. *We can find 4 solution points by checking all pairs of vertices X,Y together with their opposite points $\overline{X}$ and $\overline{Y}$ (♢). To verify that chosen points are part of a solution we can use ♣.*

Direct application of ♣ requires $str(Y,\overline{X})$ to be computed. This string may be very long. Because of this we would like to run the cyclic equivalence check on words in their compressed form (eg. `6 N 3 E` instead of `NNNNNNEEE`). It is easy to see that such a transformation does not change the outcome of our check as long as first letter of the string is different from the last one. We can ensure this property by shifting the string to the left until it is satisfied. This modification can be performed on the compressed string.

Corollary. *Conditions of ♣ can be checked without computing uncompressed representations of boundary parts.*

There are n^2 pairs of vertices (Note: we allow the same vertex to be selected twice to be able to find square tilings). Compressed representations of boundary parts can be computed in $O(n)$. Cyclic equivalence can be checked in $O(n)$, either by using any linear pattern matching algorithm ($s \cong t$ iff s is a substring of tt) or procedure from section 4. Therefore running time of algorithm presented below is $O(n^3)$.

Solution

```
procedure fix (s)
   if s contains more than one pair
      let ⟨d, l⟩ be the last direction-length pair in s.
      if first direction in s is d
         remove last pair from s.
         add l to the length of first pair in s
end

for each ⟨x, y⟩ ∈ {u|u ∈ V or ū ∈ V}
   if compressed_str(x, y) = neg(rev (compressed_str(x̄, ȳ)))
      s1 = fix (compressed_str (y, x̄));
      s2 = fix (neg (rev (compressed_str (ȳ, x))));
      if length(s1) = length(s2)
         if s1 ≅ s2 return true;

return false;
```

4 Cyclic Equivalence of Words

This section describes an algorithm which solves cyclic equivalence problem in linear time and constant memory, without using pattern matching. The algorithm is very simple and easy to implement. Java source for this algorithm is presented below.

Cyclic Equivalence of Words

```
static boolean cyclicEquiv (int[] v1, int[] v2) {
   int i, j, k, n;

   n = v1.length;
   if (n ≠ v2.length) return false;
   i = 0; j = 0; k = 0;
   while ((i < n) && (j < n)) {
      while (v1[(i + k) % n] == v2[(j + k) % n]) {
         k++;
         if (k == n) return true;
      }
      if (v1[i + k] > v2[j + k]){
         i = i + k + 1;
      } else {
         j = j + k + 1;
      }
      k = 0;
   }
   return false;
}
```

Lemma 4. *Function* `cyclicEquiv(`v_1`, `v_2`)` *returns* **true** *iff* $v_1 \cong v_2$.

Proof. Let D_1 be a set of all integers i such that $v_1^{(i)}$ (v_1 shifted by i positions) is greater (in lexicographical order) than v_2 shifted by j positions for some j ($i \in D_1$ iff $\exists j : v_1^{(i)} \gg v_2^{(j)}$). Let D_2 be a set of all integers j such that $v_2^{(j)}$ is greater than v_1 shifted by i positions for some i.

If $v_1 \cong v_2$ then $D_1 \neq n$ and $D_2 \neq n$, because there are integers p, q such that $v_1^{(p)} = v_2^{(q)}$) - $p \notin D_1$ and $q \notin D2$.

Before and after each iteration of the outer loop of `cyclicEquiv` the following invariant holds: $i \subseteq D_1$ and $j \subseteq D_2$. After the last iteration we have either $D_1 = n$ or $D_2 = n$ (so the result is `false`. The `return` statement in the inner loop is executed only when $v_1^{(i)} = v_2^{(j)}$. □

Lemma 5. *Function* `cyclicEquiv(v1, v2)` *works in time* $O(length(v_1))$

Proof. Every iteration of the outer loop increases $i + j$ by $1 + k$ where k is equal to number of iterations of the inner loop. Since $i + j < 2n$, the algorithm does at most $2n$ steps. □