1. The high-level idea of the reduction is this: the first row of tiles should be forced to be the start configuration of Turing Machine $M$ on input $w$ (padded with blanks). Each successive row of a tiling should be forced to be a TM configuration that legally follows from the one encoded in the previous row. In this way, if $M$ does not halt on $w$, then there are tilings of all possible sizes, and hence we have a “yes” instance of TILE. On the other hand, if $M$ halts on $w$ after $t$ steps, then there are tilings only up to size $t$, and hence we have a “no” instance of TILE.

In order to enforce these ideas using only horizontal and vertical compatibility relations, we will label each tile not just with a symbol in a configuration, but also with the left and right neighbors of that symbol. We will then exploit the observation that triplets of symbols in a configuration $C_i$ exactly determine the corresponding triplets of symbols allowed in a legal next configuration $C_{i+1}$.

Here are the details. We first modify $M$ so that it never tries to move off the left end of the tape (it can start by placing a marker and copying the input 1 step to the right to implement this). Also we ensure that once $M$ reaches $q_{accept}$ or $q_{reject}$ there are no transitions available to leave those states (so once $M$ halts there are no further legal configurations according to the transition function). It is also easier if we modify our format for configurations so that we have available new symbols which are all pairs $(a,q)$, where $a$ is from the tape alphabet, and $q$ is a state of $M$. A configuration that we formerly wrote $u_1u_2...u_mqv_1v_2...v_n$ is now written $u_1u_2...u_m(v_1,q)v_2...v_n$.

Now we describe the tiles. First we have a set of tiles that produce the first row containing the start configuration of $M$ on input $w = w_1w_2...w_n$. These tiles look like this when assembled properly:

$$[(w_1,q_0),w_2,w_3][w_2,w_3,w_4][w_3,w_4,w_5]...[w_{n-2},w_{n-1},w_n][w_{n-1},w_n,().[w_n,.,.]][.,.,.]...[.,.,.]$$

We include these $n+1$ tiles in our tile set and name the first one $t_1$, to ensure that it occurs in position $(1,1)$. Our horizontal compatibility relation will contain all pairs of tiles $((a,b,c),(d,e,f))$ with $b=d$ and $c=e$. Note that these $n+1$ tiles are distinct tile-types from the ones described next.

We now include one tile for every triple of symbols $(a,b,c)$ in which at most one of the three symbols is a state symbol $q$. We will add vertical compatibility pairs $((a,b,c),(d,e,f))$ if

- $(a,b,c) = (d,e,f)$ and none of $a,b,c,e$ contain a state symbol, and $d = a$ or $d = (a,q)$, and $f = c$ or $f = (c,q)$, or
2. The key here is that $H$ is fixed. Suppose it has $k$ vertices. Given an input $G$, we can simply exhaustively try all potential subgraph isomorphisms. Specifically, we try every subset of $k$ vertices of $G$, and for each subset we try every one of the $k!$ possible ways of mapping it to the vertices of $H$. We can easily check, whether $H$ equals the subgraph induced by a given subset of $G$’s vertices (permuted according to a given mapping) – in fact this can be done in time polynomial in $k$.

Thus the overall running time of the algorithm is

$$\binom{|G|}{|H|} \cdot k! \cdot k^{O(1)}$$

Note that $\binom{n}{k} \leq n^k$, and $k! \leq k^k$, so we have an overall running time of $|G|^{O(k)}$. Since $k$ is fixed, this is polynomial in the size of the input, $G$. 

- $a = (x, q)$ and $\delta(q, x) = (r, d, L)$ and $e = b$ and $f = c$, or
- $a = (x, q)$ and $\delta(q, x) = (r, d, R)$ and $e = (b, r)$ and $f = c$, or
- $b = (x, q)$ and $\delta(q, x) = (r, e, L)$ and $d = (a, r)$ and $f = c$, or
- $b = (x, q)$ and $\delta(q, x) = (r, e, R)$ and $d = a$ and $f = (c, r)$, or
- $c = (x, q)$ and $\delta(q, x) = (r, f, L)$ and $d = a$ and $e = (b, r)$, or
- $c = (x, q)$ and $\delta(q, x) = (r, f, R)$ and $d = a$ and $f = c$
3. Following the hint, we will build up a table $T$ which has a TRUE/FALSE entry for each $0 \leq B' \leq B$ telling us whether some multiset of the $x_i$ sum to exactly $B'$. Specifically, let $T$ be an array with $B + 1$ entries. Initialize $T[i]$ to be FALSE for all $i$. For $B' = 0, 1, 2, \ldots B$ do:

- if some $x_i = B'$ then $T[B'] = \text{TRUE}$.
- else if for some $x_i$ we have $T[B' - x_i] = \text{TRUE}$, then set $T[B'] = \text{TRUE}$ (otherwise leave $T[B']$ set to FALSE).

At the end, we accept iff $T[B]$ is TRUE.

In each step we need to check all of the $x_i$, so the running time for one iteration of the loop is $O(n)$. The overall running time is $O(Bn)$ which is polynomial in the size of the input (here we use crucially that $B$ is presented in unary).

We should also argue that this algorithm is correct. We claim that after the $B'$-th iteration, all entries of $T$ up to and including $B'$ are correct (i.e. they are TRUE if there is a multiset of the $x_i$ summing to that value, and FALSE otherwise). The base case (when $B' = 0$) is clearly satisfied. Now assume $T$ is correct up to and including $B' - 1$. There is a non-empty multiset summing to exactly $B'$ iff for some $i$, there is a non-empty multiset summing to $B' - x_i$. By induction $T[B' - x_i]$ is correct, and so we correctly fill in $T[B']$ in the $B'$-th iteration of the main loop.