Solution — London

1 The problem in a nutshell

We are given a **newspaper** which we can cut apart, each piece then having a single letter on the front and a single letter on the back. With these pieces we want to know whether we can recreate a given **note**.

2 Modeling

The newspaper has two sides, each with h rows and w columns consisting of *capital English letters A to Z* (no spaces), which gives $h \cdot w$ pieces to write the note with. For each of these pieces we can choose to have it front or back facing up. We are asked if we can choose a subset of n of them to produce a given note of that length.

This is a clear resource assignment problem. We can model this in different ways but essentially the question is, can we match each letter of the note to a different piece of the newspaper which has that letter on the back or on the front. Each piece of the newspaper can only be used on one side which is why a direct greedy approach will not solve the complete problem.

The right toolbox. Which tools that we know might be applicable to the problem setting?

Greedy. An obvious greedy strategy is to match the pieces of the newspaper with positions of the note as soon as the letters fit. This strategy can have two forms, either we go through the positions in the note and match the first letter in the newspaper that matches, or we go through the pieces of newspaper and try to match them to places in the note where the letters occur. Both of these approaches are slow and will produce the wrong result. Only in special cases, such as the first test set as described later, this approach will succeed—but not in general!

As a graph problem. We need to assign resources (pieces of the newspaper) to positions of the note. So this we can model as simply matching pieces to note positions—a problem which can be modeled as a bipartite graph. Having it represented as a bipartite graph we can then apply both maximum matching and/or flow algorithms to solve the problem: we can create the note if and only if the matching covers all positions of the note or the flow saturates all edges to the sink. How this graphs is built in detail we explain in the respective solutions using matchings or flows.

We make a quick check but see that there is nothing unusual about the network or the relations so we should model a directed graph with `adjacency_list` .

3 Algorithm design

Having modeled the graph as a resource distribution problem we can now think about how to design the algorithm. The problem comes with essentially five different test sets with increasing difficulty.

Subtask specifications

1. For the first group of test sets, worth 10 points, you may assume that all letters on the back side of the newspaper are identical and that $h, w \leqslant 30$.
2. For the second group of test sets, worth 10 points, you may assume that only two different letters appear on the back side of the newspaper and that $h, w \leqslant 30$.
3. For the third group of test sets, worth 20 points, you may assume $h, w \leqslant 30$.
4. For the fourth group of test sets, worth 20 points, you may assume $h, w \leqslant 10^2$.
5. For the fifth and sixth group of test sets, worth 20 points each, there are no additional assumptions.

We will present three approaches using the tools from above which should be worth 10, 40 and 100 points respectively.

A greedy approach (10 points). Before we go into the BGL solutions, it is always worth considering applying a greedy algorithm. By the greedy alrogithm we mean in this case go through all pieces of the newspaper in any order and immediatly assign them to one of the positions of the note. Greedy works in the first testcase because all the letters on the backside are the same so we greedily assign the front side of the pieces.

For the greedy algorithm we need to check two things, are there enough letters of every kind on the newspaper, ignoring the letter on the back, and are there enough letters in total (so that we can use enough backside letters). For this a simple counting for each occurence of letters in the note and on the newspaper should suffice, which can be done in time $\mathcal{O}(h \cdot w)$.

A greedy strategy only works for the first two test sets, for the second test set it can be done by checking whether either of the letters on the backside is a bottleneck for the note and we will not explain that in detail.

```
1  void testcases(int T) {
2      // Read the input
3      int h, w; std::cin >> h >> w; int n_piece = h*w;
4      std::string note = "";  std::cin >> note;
5      std::vector<std::string> news_line(2*h);
6      for (int i = 0; i < 2* h; ++i)
7          std::cin >> news_line[i];
8      // For each letter A-Z (26 of them) check how
9      // often they occur in the note
10     std::vector<int> oc_letters(26, 0);
11     for (int i = 0; i < note.length(); ++i)
12         ++oc_letters[note[i] - 'A'];
13     // For each time a letter occurs on the newspaper,
14     // tick off an occurence in the note
15     for (int i = 0; i < n_piece; ++i)
16         --oc_letters[news_line[i/w][i%w] - 'A'];
```

```
17      // The letter on the back side should not matter
18      oc_letters[news_line[h][0] - 'A'] = 0;
19      // You need at least as many letters as in the note
20      if(h*w < note.length()){
21          std::cout << "No" << "\n";
22          return;
23      }
24      // If for any of the letters there are still some occurences
25      // not yet ticked off we do not have enough of them on the newspaper
26      for (int i = 0; i < 26; ++i)
27          if(oc_letters[i] > 0) {
28              std::cout << "No" << "\n";
29              return;
30          }
31      std::cout << "Yes" << "\n";
32      return;
33  }
```

Bipartite Matching (40 points). We apply the Edmund maximum matching algorithm to find a solution

Graph modeling Viewing this problem as a graph problem we need to figure out how to choose vertices and edges. We want a bipartite graph $G = (R \cup D, E)$ with resources (R, pieces of newspaper) and demand vertices (D, letters of the note). Make sure n is not larger than $h \cdot w$, the output becomes trivial otherwise. There are $h \cdot w$ many newspaper letter pairs and at most that many letters in the note, so the graph will have at most $2 \cdot h \cdot w$ vertices. We add an edge from vertices in R to vertices in D if the letter corresponding to the vertex in D matches one of the two letters in the letter pair of the vertex in R.

The way the matching solution works is for every position in the note we try to find a piece of newspaper which has the proper letter, without using any of the pieces more than once. Building the bipartite graph is done in time $\mathcal{O}(h \cdot w \cdot n)$. The graph may have many edges, can be a complete bipartite graph. Our maximum matching algorithm will have runtime up to $\mathcal{O}((h \cdot w)^2)$ which is not realy feasible for $h, w \approx 100$ so it will not pass the test set four.

The matching edges give a direct assignment of which piece of newspaper to put at which position in the note and this mapping is the proof that the note can be composed if and only if there is a matching of size n.

An example solution using this approach is found at the end this document.

Using flows (100 points). Until now we did not even use flows. It is worth mentioning that we can easily get the 40 points solution also with flows, by just modeling the matching problem as a flow. Take the graph from above, directed from left (R, pieces of newspaper) to right (D, letters of the note), add source with edges to R and target with all edges from D. All capacities set to one and then check if the maximum flow has a value of n. This is equivalent to the bipartite matching algorithm we discuss in the BGL tutorial 3.

The following observations are important. The position in the note is not as important as the actual letter at that position, of which there are only 26 possibilities, and this should make the

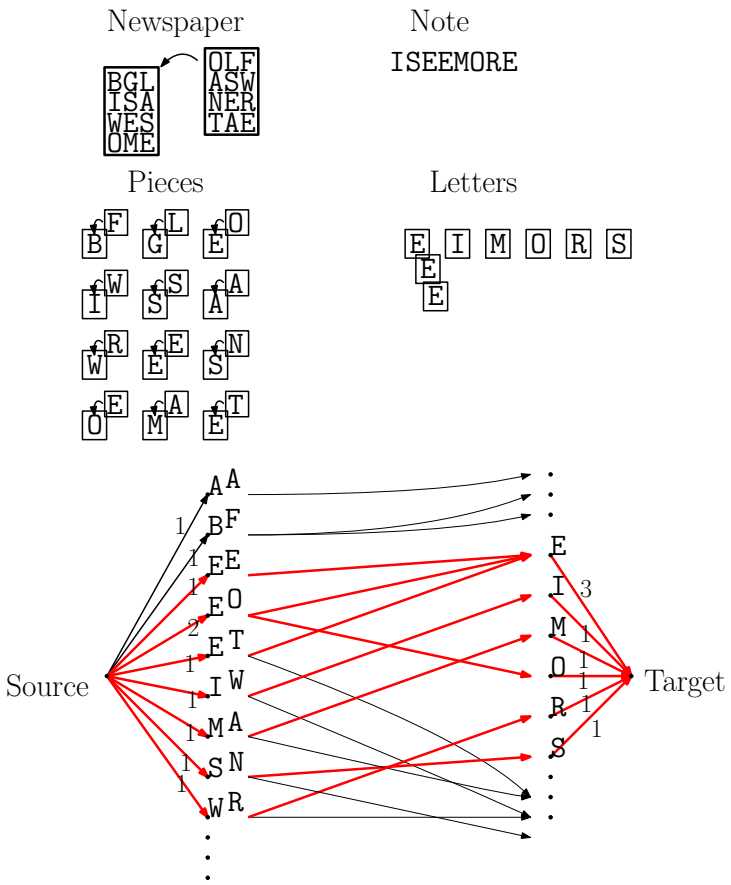

Figure 1: Modeling as a flow

graph much smaller. Similarly where in the newspaper a letter pair is found is unimportant, we are only interested in the letter values and of those there are also only 26^2 many ordered letter pairs possible, which is much smaller than $h \cdot w \leqslant 10^6$ and $n \leqslant 10^6$.

Graph modeling Viewing this problem as a graph problem we need to figure out how to choose vertices and edges. We want a bipartite graph $G = (R \cup D, E)$ with resources (R, pieces of newspaper) and demand vertices (D, letters of the note). Additionally we add a source and a sink. Make sure n is not larger than $h \cdot w$, the output becomes trivial otherwise. R is the set of all ordered letter pairs, which is of size 26^2 and D is the set of all letters, which is 26. We add edges from the source to each vertex in R with capacity equal to the number of occurences of that letter pair in the newspaper. We add edges from D to the sink with capacity equal to the number of occurences of that letter in the note. Add up to two edges from every vertex in R to D with capacity infinity (`INT_MAX` for example) to the corresponding letters of the representing letter pair.

Flow versus solution Now we claim an integer flow in this network corresponds to an assigning of the newspaper pieces to some, possibly all, of the letters of the note. Although we do not need to output how we would piece together the note, it is worth mentioning that the matching or flow algorithm directly give us an assignment of pieces to positions in the note. For matching

this should be obvious, we have a vertex for every piece of the newspaper and for every position in the note, so if a piece is matched with a position then that is the piece that we should use. For the flow we group together pieces with the same front and back and positions with the same letter. But this is not a problem as we can simply greedily fill up with as many pieces as flow through a vertex into any of the positions with the corresponding letters.

This also helps as a check to make sure we really have a correct solution, there is a flow if and only if there is a solution for the entire note.

Note If you only replaced the positions in the note by the 26 letters (vertices from D) but not the pieces you already get a significant speedup and will manage to get the first four test sets and be awarded with 60 points.

Graph on 28 nodes (100 points) As an alternative solution we can even further reduce the size of our graph because we can store the letter pairs as edges instead of vertices.

Graph modeling Build a graph on 26 vertices, one for every letter from A-Z, and add a source and a sink. Add all possible edges to form a complete graph. We set capacities as follows. For each edge from the source to a letter, e.g. 'A', set the capacity to the number of times 'A' appears on the front page. Set capacities of the edges from every letter to the sink with a capacity equal to the number of occurences of that letter in the note. Between two letters, e.g. from A to C, set a capacity equal to the number of times we have a letter pair with A on the front and C on the back of the newspaper.

Run a max flow algorithm on the graph and you can reconstruct the note if and only if the flow is equal to the length of the note.

Any flow in the graph corresponds to an assignment of the letters to the note in the following way. Every edge between the letters corresponds to a letter pair, which can occur multiple times on the newspaper, start of the edge corresponds to front page letter of the newspaper, end of the edge corresponds to the back. For every unit of flow on edges between the letters this means we take that many letter pairs of the corresponding type and flip them to the back side. Then we see if we can fill all the positions of the note with all the front and backside flips now fixed. The number of available A's is the number of frontside A's, minus the number of those we flipped, plus the ones which where flipped such that the backside now shows an A. Same for every other letter. For every vertex in the graph this corresonds to exactly in-capacity minus out-flow to other letters plus in-flow from other letters. If this is greater or equal to the capacity of the edge to the sink, it will be possible to create a flow with value equal to the length of the note.

4 Implementation

The implementation is standard if you use the provided sample code from the BGL Flow Tutorial. Be sure to respect that the backside of the newspaper is reversed. As in the sample code adding edges in the flow network can make your code less readable so it's helpful to create an Edge Adder class for this. Using strings is like using a vector of char.

Complete solution Implementation of a solution could look like this.

```cpp
#include <iostream>
#include <vector>
#include <string>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef boost::adjacency_list_traits<boost::vecS, boost::vecS, boost::directedS> traits;
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS, boost::no_property,
    boost::property<boost::edge_capacity_t, long,
      boost::property<boost::edge_residual_capacity_t, long,
        boost::property<boost::edge_reverse_t, traits::edge_descriptor> > > >   graph;
// Interior Property Maps
typedef boost::property_map<graph, boost::edge_capacity_t>::type       edge_cap_map;
typedef boost::property_map<graph, boost::edge_residual_capacity_t>::type  res_cap_map;
typedef boost::property_map<graph, boost::edge_reverse_t>::type     rev_edge_map;
typedef boost::graph_traits<graph>::edge_descriptor         edge;

// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class edge_adder {
    graph &G;
    edge_cap_map    &capacitymap;
    rev_edge_map    &revedgemap;

public:
    // to initialize the Object
    edge_adder(graph & G, edge_cap_map &capacitymap, rev_edge_map &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}
    // to use the Function (add an edge)
    void add_edge(int from, int to, long capacity) {
        edge e, rev_e;
        bool success;
        boost::tie(e, success) = boost::add_edge(from, to, G);
        boost::tie(rev_e, success) = boost::add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[rev_e] = 0; // reverse edge has no capacity!
        revedgemap[e] = rev_e;
        revedgemap[rev_e] = e;
    }
};

// Functions
// =========
// Function for an individual testcase
void testcases(int T) {
    int h, w;
    std::cin >> h >> w;
    int n_piece = 26*26;
    std::string note; std::cin >> note; int n = note.length();
    std::vector<std::string> line(2*h);
    for (int i = 0; i < 2* h; ++i)
        std::cin >> line[i];
```

```
    // Count occurences of letters in the note
    std::vector<int> letters(26, 0);
    for (int i = 0; i < n; ++i)
        ++letters[note[i]-'A'];
    // Count occurences of every pair of letters in the newspaper
    std::vector<int> tuplepieces(26*26, 0);
    for (int i = 0; i < h; ++i)
        for (int j = 0; j < w; ++j)
            ++tuplepieces[(line[i][j]-'A')*26 + (line[i+h][w-j-1]-'A')];
    // Create graph and maps for the flow
    graph G(n_piece+26+2);
    edge_cap_map capacitymap = boost::get(boost::edge_capacity, G);
    rev_edge_map revedgemap = boost::get(boost::edge_reverse, G);
    res_cap_map rescapacitymap = boost::get(boost::edge_residual_capacity, G);
    edge_adder adder(G, capacitymap, revedgemap);
    int source = n_piece + 26;
    int target = n_piece + 27;
    // Add edges
    // =========
    // An edge from the source to every possible piece with the proper capacity
    for (int i = 0; i < 26*26; ++i)
            adder.add_edge(source, i, tuplepieces[i]);
    // An edge from every letter to the target with the proper capacity
    for (int j = 0; j < 26; ++j)
        adder.add_edge(n_piece+j, target, letters[j]);
    // Edges between pieces and letters, capacity chosen high enough
    for (int i = 0; i < n_piece; ++i){
              adder.add_edge(i, n_piece+i/26, n);
              adder.add_edge(i, n_piece+i%26, n);
    }
    // Calculate the flow
    int flow = boost::push_relabel_max_flow(G, source, target);
    // If flow is n we can find a piece for every letter in the note
    if (flow == n)
        std::cout << "Yes" << "\n";
    else
        std::cout << "No" << "\n";
    return;
}

// Main function to loop over the testcases
int main() {
    std::ios_base::sync_with_stdio(false);
    int T; std::cin >> T;
    while(T--) testcases(T);
}
```

Matching As mentioned to get 40 points we could use a maximum matching algorithm such as follows.

```
#include <iostream>
#include <vector>
#include <string>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
// Namespaces
using namespace std;
```

```cpp

// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS >  graph;

// Functions
// =========
// Function for an individual testcase
void testcases(int T) {
        // Read the input
        int h, w;
        std::cin >> h >> w;
        int n_piece = h*w;
        std::string note;
        std::cin >> note;
        int n = note.length();
        std::vector<std::string> line(2*h);
        for (int i = 0; i < 2* h; ++i)
                std::cin >> line[i];
        // Make sure n is reasonable
        if(n > n_piece) {
                std::cout << "No" << "\n";
                return;
        }
        // Create the graph
        graph G(n_piece+n);
        // Add all edges between the pieces and the note positions
        char f, b;
        for (int i = 0; i < n_piece; ++i){
                f = line[i/w][i%w] ;
                b = line[i/w+h][w-1-(i%w)] ;
                for (int j = 0; j < n; ++j)
                        if (f == note[j] || b == note[j])
                                boost::add_edge(i, n_piece + j, G);
        }
        // Define exterior property map. This map maps vertices to vertices.
        std::vector<int> mate_map(n_piece + n);
        boost::edmonds_maximum_cardinality_matching(G,
                boost::make_iterator_property_map(mate_map.begin(),
                        boost::get(boost::vertex_index, G)));
        // Retrieve the size of the matching
        int matching_size = boost::matching_size(G,
                boost::make_iterator_property_map(mate_map.begin(),
                        boost::get(boost::vertex_index, G)));
        // Check if we could match every position of the note
        if (matching_size == n)
                std::cout << "Yes" << "\n";
        else
                std::cout << "No" << "\n";
        return;
}
// Main function to loop over the testcases
int main() {
        std::ios_base::sync_with_stdio(false);
        int T;  std::cin >> T;
        while(T--) testcases(T);
        return 0;
}
```