1 算法分析

可以划归为二分完美匹配问题:

- 将棋盘黑白染色后，黑色格子和白色格子分别作为点的集合.
- 相邻的格子间有边，作为二分集合间的边
- 覆盖了一黑一白的骨牌，则为选中了的边
- 完全覆盖，即为完全匹配

使用匈牙利算法，时间复杂度为$O(V*E)=O(N^2)$

2 伪代码

```
convert grids to graph
initialize all edges
for all unvisted nodes in graph
  breath-first-search along cross-flow
    when another unvisted node visited then
      extend cross-flow and stop BFS
      increase count
check if count equals the number of nodes pair
return isFullMatched
```

3 代码

```
// graph.h is omitted
int breath_first_search(Graph &graph, int source) {
  auto here = [source](int vertex) { return (vertex + source) % 2 == 0; };
  if (graph[source].covered()) {
    return 0; // failed
  }
  constexpr auto black = color_t::black;
  constexpr auto white = color_t::white;
  for (auto &vertex : graph) {
    vertex.color = white;
    vertex.discover_time = inf;
    vertex.parent = -2;
  }
  queue<int> qq;
  graph[source].color = black;
  graph[source].discover_time = 0;
  graph[source].parent = -1;
  qq.push(source);
  while (qq.size()) {
    auto &vertex = graph[qq.front()];
    qq.pop();
    if (!here(vertex.from)) {
      // choose matched directly
      auto &next_v = graph[vertex.mate];
      if (next_v.color == black)
        continue;
      next_v.color = black;
      next_v.discover_time = vertex.discover_time + 1;
      next_v.parent = vertex.from;
      qq.push(vertex.mate);
```

```
      continue;
    } else {
      FOR_EDGE(edge, vertex) {
        if (edge->to == vertex.mate)
          continue;
        auto &next_v = graph[edge->to];
        if (next_v.color == white) {
          next_v.color = black;
          next_v.discover_time = vertex.discover_time + 1;
          next_v.parent = vertex.from;
          if (!next_v.covered()) {
            // get it;
            for (int iter = edge->to; iter != -1;) {
              int pair = graph[iter].parent;
              int next = graph[pair].parent;
              graph[iter].mate = pair;
              graph[pair].mate = iter;
              iter = next;
            }
            return 1;
          }
          // or continue
          qq.push(edge->to);
        }
      }
    }
  }
  return 0;
}

bool tiled_cover(const vector<char> &mat, int row_num, int col_num) {
  auto index = [gap = (col_num | 1)](int i, int j) {
    return i * gap + j;
  };

  auto value = [&mat, col_num, row_num](int i, int j) {
    if (i >= row_num || j >= col_num) {
      return '\0';
    } else {
      return mat[i * col_num + j];
    }
  };
  int count[2] = {};
  col_num |= 1;
  for (int i = 0; i < row_num; ++i) {
    for (int j = 0; j < col_num; ++j) {
      count[(i + j) & 1] += !!value(i, j);
    }
  }
  if (count[0] != count[1]) {
    return false;
  }
  Graph graph(row_num * col_num);
  for (int i = 0; i < row_num; ++i) {
    for (int j = 0; j < col_num; ++j) {
      if (value(i, j) && value(i, j + 1)) {
        add_edge(graph, index(i, j), index(i, j + 1));
        add_edge(graph, index(i, j + 1), index(i, j));
```

```
      }
      if (value(i, j) && value(i + 1, j)) {
        add_edge(graph, index(i, j), index(i + 1, j));
        add_edge(graph, index(i + 1, j), index(i, j));
      }
    }
  }
  int sum = 0;
  int old_sum = -1;
  while (sum != old_sum) {
    old_sum = sum;

    for (int v = 0; v < graph.size(); ++v) {
      if (v == 7) {
        int a = 1 + 1;
      }
      sum += breath_first_search(graph, v);
    }
  }
  return sum == count[0];
}
```

4 测试样例

使用Google Test进行单元测试, 请查看test.cpp

```
TEST(tiled_cover, main) {
  string mat_str[] = {
      "0110", //
      "0111", //
      "0110", //
      "1110", //
  };
  vector<char> mat;
  for (auto &str : mat_str) {
    for (char s : str) {
      mat.push_back(s - '0');
    }
  }
  EXPECT_EQ(tiled_cover(mat, 4, mat_str[0].size()), true);
}

TEST(tiled_cover, test2) {
  vector<string> mat_str = {
      "010", //
      "111", //
      "010", //
      "010", //
      "111", //
      "010", //
  };
  vector<char> mat;
  for (auto &str : mat_str) {
    for (char s : str) {
      mat.push_back(s - '0');
    }
  }
  EXPECT_EQ(tiled_cover(mat, mat_str.size(), mat_str[0].size()), false);
```

```
}

TEST(tiled_cover, test3) {
  vector<string> mat_str = {
      "110101111", //
      "111111111", //
      "110101111", //
      "110101111", //
      "111111111", //
      "110101111", //
  };
  vector<char> mat;
  for (auto &str : mat_str) {
    for (char s : str) {
      mat.push_back(s - '0');
    }
  }
  EXPECT_EQ(tiled_cover(mat, mat_str.size(), mat_str[0].size()), true);
}

TEST(tiled_cover, test4) {
  vector<string> mat_str = {
      "1100101111", //
      "1101111111", //
      "1100101111", //
      "1100101111", //
      "1101111111", //
      "1100101111", //
  };
  vector<char> mat;
  for (auto &str : mat_str) {
    for (char s : str) {
      mat.push_back(s - '0');
    }
  }
  EXPECT_EQ(false, tiled_cover(mat, mat_str.size(), mat_str[0].size()));
}
```