ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Lớp: CS112.Q11.KHTN

Môn học: Phân Tích và Thiết Kế Thuật Toán

BTVN nhóm 8: Phương pháp thiết kế thuật toán gần đúng

Nhóm 4:
Lê Văn Thức
Bảo Quý Định Tân

Giảng viên:
Nguyễn Thanh Sơn

1 Lời giải: Lắp đặt cảm biến

1.1 Subtask 1: $1 \leq m, n \leq 20$

Ở subtask này, hướng giải của ta là duyệt trâu tất cả các cấu hình lựa chọn 2^m để quyết định sẽ chọn cảm biến nào để sử dụng. Khi chọn ra một tập các cảm biến sẽ sử dụng, ta sẽ kiểm tra coi tập đó có thỏa mản (phủ được toàn bộ $n \times n$ ô) hay không? Nếu có thì ta cập nhật đáp án nếu tổng chi phí của tập này tối ưu hơn những gì ta đã tìm được.

Lưu ý, để việc kiểm tra coi tập cảm biến được chọn có thỏa mãn hay không nhanh, ta có thể sử dụng **prefix sum 2D** để tăng tốc việc kiểm tra. Để kiểm tra, ta cần đánh dấu coi từng ô đã có cảm biến nào bao lên chưa nên ta gọi mảng $prefSum[x][y]$ = số lượng cảm biến sẽ được **bao thêm** lên các ô (i, j) với $x \leq i \leq n, y \leq j \leq n$. Cụ thể:

```
prefSum[max(1, x - r)][max(1, y - r)]++;
prefSum[max(1, x - r)][min(n + 1, y + r + 1)]--;
prefSum[min(n + 1, x + r + 1)][max(1, y - r)]--;
prefSum[min(n + 1, x + r + 1)][min(n + 1, y + r + 1)]++;
```

Sau đó để kiểm tra một ô ở vị trí (x, y) nào đó bất kỳ ta cần lấy tổng $prefSum[i][j]$ $(1 \leq i \leq x, 1 \leq j \leq y)$, ta có thể tận dụng luôn mảng có sẵn này để tính và kiểm tra:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        prefSum[i][j] += prefSum[i][j - 1] + prefSum[i - 1][j] - prefSum[i
    - 1][j - 1];
        if (!prefSum[i][j]) return false;
    }
}
```

1.1.1 Phân tích độ phức tạp thời gian:

- 1. Xác định thao tác cơ bản: phép tính và kiểm tra xem một ô có được cảm biến bao lên hay không?
- 2. Số lần thao tác cơ bản được thực hiện: Ta cần phải duyệt qua $n \times n$ ô với mỗi cấu hình chọn các cảm biến. Mà ta lại duyệt qua tổng cộng 2^m cấu hình khác nhau. Nên tổng số lần là $n^2 \times 2^m$.
- 3. Thiết lập hàm thời gian $T(n, m) = 2^m \times n^2$.
- 4. Lấy Big-O: $O(2^m \times n^2)$.

1.1.2 Phân tích độ phức tạp không gian:

- 1. Xác định các vùng nhớ được sử dụng: Ta đa số chỉ sài các biến bình thường, vector một chiều với kích thước theo m và lớn nhất là mảng hai chiều kích thước $n \times n$.
- 2. Tính tổng bộ nhớ: $O(1) + O(m) + O(n^2)$
- 3. Thiết lập hàm không gian $S(n, m) = O(1) + O(m) + O(n^2)$.
- 4. Lấy Big-O: $O(n^2)$.

1.1.3 Code c++:

```cpp
pair<int, vector<int>> trau() {
    int ans = inf;
    vector<int> ansList;

    for (int mask = 0; mask < (1 << m); mask++) {
        int sum = 0;
        vector<int> currList;
        for (int i = 1; i <= m; i++) {
            if (mask >> (i - 1) & 1) {
                sum += a[i].c;
                currList.pb(i);
            }
        }

        if (sum >= ans) continue;

        for (int i : currList) {
            int x = a[i].x;
            int y = a[i].y;
            prefSum[max(1, x - r)][max(1, y - r)]++;
            prefSum[max(1, x - r)][min(n + 1, y + r + 1)]--;
            prefSum[min(n + 1, x + r + 1)][max(1, y - r)]--;
            prefSum[min(n + 1, x + r + 1)][min(n + 1, y + r + 1)]++;
        }

        auto check = [&]() -> bool {
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                    prefSum[i][j] += prefSum[i][j - 1] + prefSum[i - 1][j]
   - prefSum[i - 1][j - 1];
                    if (!prefSum[i][j]) return false;
                }
            }
            return true;
        };

        if (check()) {
            ans = sum;
            ansList = currList;
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                prefSum[i][j] = 0;
            }
        }
    }

    return {ans, ansList};
}
```

1.1.4 Code python:

```python
def trau(self) -> Tuple[int, List[int]]:
    n = self.n
```

```
    m = self.m
    r = self.r
    diff = [[0] * (n + 3) for _ in range(n + 3)]
    grid = [[0] * (n + 3) for _ in range(n + 3)]
    best_cost = INF
    best_choice: List[int] = []
    for mask in range(1 << m):
        cost = 0
        choice = []
        for i in range(1, m + 1):
            if mask >> (i - 1) & 1:
                cost += self.sensors[i][2]
                choice.append(i)
        if cost >= best_cost:
            continue
        for idx in choice:
            x, y, _ = self.sensors[idx]
            x1 = max(1, x - r)
            x2 = min(n, x + r)
            y1 = max(1, y - r)
            y2 = min(n, y + r)
            diff[x1][y1] += 1
            diff[x1][y2 + 1] -= 1
            diff[x2 + 1][y1] -= 1
            diff[x2 + 1][y2 + 1] += 1
        ok = True
        for i in range(1, n + 1):
            running = 0
            for j in range(1, n + 1):
                running += diff[i][j]
                grid[i][j] = grid[i - 1][j] + running
                if grid[i][j] == 0:
                    ok = False
                    break
            if not ok:
                break
        if ok:
            best_cost = cost
            best_choice = choice[:]
        for i in range(1, n + 2):
            for j in range(1, n + 2):
                diff[i][j] = 0
                grid[i][j] = 0
    return best_cost, best_choice
```

1.2 Subtask 2: $1 \leq n \leq 50, 1 \leq m \leq 100$

Theo mình nghĩ và tìm hiểu thì bài toán này là NP-hard nên ta có thể tiếp cận theo cách tham lam/giải gần đúng.

Cách mà mình làm là ưu tiên chọn những cảm biến mà **"lời"** nhất. **"Lời"** ở đây mình định nghĩa theo tỷ lệ $\frac{\text{số lượng ô có thể bao thêm được (không tính những ô đã được bao)}}{\text{chi phí cảm biến}}$.

Tức những ô nào nếu lấy, càng bao thêm được càng nhiều **ô mới** thì càng **"lời"**, nhưng ta còn phải xem sét đến chi phí nữa, chưa chắc càng bao nhiều lại càng tốt.

```
int numUncover = n * n;
```

```
while (numUncover) {
    int numCoverOfCurrBestRatio = 0;
    int bestRatioPos = -1;

    for (auto &i : unUsedList) {
        int numCover = 0;
        for (auto &[x, y] : coverSet[i]) {
            numCover += (!prefSum[x][y]);
        }
        if (!(~bestRatioPos) || 1ll * a[bestRatioPos].c * numCover > 1ll *
   a[i].c * numCoverOfCurrBestRatio) {
            numCoverOfCurrBestRatio = numCover;
            bestRatioPos = i;
        }
    }

    if (!(~bestRatioPos) || numCoverOfCurrBestRatio == 0) {
        resetPref();
        return {inf, {}};
    }

    for (auto &[x, y] : coverSet[bestRatioPos]) {
        prefSum[x][y]++;
    }
    numUncover -= numCoverOfCurrBestRatio;
    ans += a[bestRatioPos].c;
    ansSet.insert(bestRatioPos);
    unUsedList.erase(bestRatioPos);
}
```

Và để hiệu quả thêm, mình còn thêm một heuristic là sẽ random bỏ ra theo tỷ lệ $\frac{1}{3}$ để chọn trước một vài cảm biến để loại ra luôn, không bao giờ chọn. Sau đó ta sẽ có một danh sách ứng cử viên những cảm biến có thể chọn và thực hiện thuật toán tham lam trên.

```
set<int> unUsedList;

for (int i = 1; i <= m; i++) {
    if (trickLord) {
        if (Rand(0, 2)) {
            unUsedList.insert(i);
        }
    }
    else {
        unUsedList.insert(i);
    }
}
```

Và vì **random** nên ta sẽ thực hiện việc này nhiều lần với hy vọng sẽ tìm được nghiệm tối ưu. Mỗi lần lại ngẫu nhiên loại trừ một vài cảm biến ứng cử viên, sau đó lại thực hiện tham lam để chọn các cảm biến để bao hết các ô.

```
for (int t = 1; t <= 333; t++) {
    auto tmp = bestRatio(true);
    if (tmp.fi < finalAns.fi) {
        finalAns = tmp;
    }
}
```

Và để đảm bảo hơn nữa, sau khi chọn xong một vài cảm biến có thể bao hết các ô. Mình lại thử trong từng cảm biến đã được chọn, nếu bỏ cảm biến này đi mà n^2 ô của ta vẫn được bao hết thì ta có thể an toàn bỏ cảm biến này đi để giảm chi phí.

```
auto tryDropGroup = [&](const vector<int>& group) -> bool {
    vector<pii> touched;
    for (int id : group) {
        for (auto &[x, y] : coverSet[id]) {
            if (dropDelta[x][y] == 0) {
                touched.pb({x, y});
            }
            dropDelta[x][y]++;
        }
    }
    bool ok = true;
    for (auto &[x, y] : touched) {
        if (prefSum[x][y] - dropDelta[x][y] <= 0) {
            ok = false;
            break;
        }
    }
    if (ok) {
        for (auto &[x, y] : touched) {
            prefSum[x][y] -= dropDelta[x][y];
        }
    }
    for (auto &[x, y] : touched) {
        dropDelta[x][y] = 0;
    }
    return ok;
};

bool improved = true;
while (improved) {
    improved = false;
    for (auto it = ansSet.begin(); it != ansSet.end(); ) {
        int id = *it;
        if (tryDropGroup(vector<int>{id})) {
            ans -= a[id].c;
            it = ansSet.erase(it);
            improved = true;
        } else {
            ++it;
        }
    }
    if (improved) continue;
    break;
}
```

1.2.1 Code c++ đầy đủ:

```
pair<int, vector<int>> bestRatio(bool trickLord = false) {
    auto resetPref = [&]() {
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                prefSum[i][j] = 0;
            }
```

```
        }
    };
    resetPref();

    int ans = 0;
    set<int> ansSet;

    set<int> unUsedList;

    for (int i = 1; i <= m; i++) {
        if (trickLord) {
            if (Rand(0, 2)) {
                unUsedList.insert(i);
            }
        }
        else {
            unUsedList.insert(i);
        }
    }


    int numUncover = n * n;
    while (numUncover) {
        int numCoverOfCurrBestRatio = 0;
        int bestRatioPos = -1;

        for (auto &i : unUsedList) {
            int numCover = 0;
            for (auto &[x, y] : coverSet[i]) {
                numCover += (!prefSum[x][y]);
            }
            if (!(~bestRatioPos) || 1ll * a[bestRatioPos].c * numCover > 1ll * a[i].c * numCoverOfCurrBestRatio) {
                numCoverOfCurrBestRatio = numCover;
                bestRatioPos = i;
            }
        }

        if (!(~bestRatioPos) || numCoverOfCurrBestRatio == 0) {
            resetPref();
            return {inf, {}};
        }

        for (auto &[x, y] : coverSet[bestRatioPos]) {
            prefSum[x][y]++;
        }
        numUncover -= numCoverOfCurrBestRatio;
        ans += a[bestRatioPos].c;
        ansSet.insert(bestRatioPos);
        unUsedList.erase(bestRatioPos);
    }

    int dropDelta[MAX_N][MAX_N];

    auto tryDropGroup = [&](const vector<int>& group) -> bool {
        vector<pii> touched;
        for (int id : group) {
            for (auto &[x, y] : coverSet[id]) {
```

```
                if (dropDelta[x][y] == 0) {
                    touched.pb({x, y});
                }
                dropDelta[x][y]++;
            }
        }
        bool ok = true;
        for (auto &[x, y] : touched) {
            if (prefSum[x][y] - dropDelta[x][y] <= 0) {
                ok = false;
                break;
            }
        }
        if (ok) {
            for (auto &[x, y] : touched) {
                prefSum[x][y] -= dropDelta[x][y];
            }
        }
        for (auto &[x, y] : touched) {
            dropDelta[x][y] = 0;
        }
        return ok;
    };

    bool improved = true;
    while (improved) {
        improved = false;
        for (auto it = ansSet.begin(); it != ansSet.end(); ) {
            int id = *it;
            if (tryDropGroup(vector<int>{id})) {
                ans -= a[id].c;
                it = ansSet.erase(it);
                improved = true;
            } else {
                ++it;
            }
        }
        if (improved) continue;
        break;
    }

    vector<int> ansList(all(ansSet));
    resetPref();
    return {ans, ansList};
}

for (int t = 1; t <= 333; t++) {
    auto tmp = bestRatio(true);
    if (tmp.fi < finalAns.fi) {
        finalAns = tmp;
    }
}
```

1.2.2 Phân tích độ phức tạp thời gian:

- 1. Xác định thao tác cơ bản: phép tính và kiểm tra để chọn ra cảm biến có tỷ lệ tốt. Hoặc kiểm tra coi một cảm biến được bỏ sau khi chọn thì có hợp lệ hay không.

- 2. Số lần thao tác cơ bản được thực hiện: Mỗi lần ta lại phải duyệt qua các cảm biến $O(m)$, với mỗi cảm biến lại phải duyệt qua những ô nó có thể bao để tính theo công thức tỷ lệ như trên $O(n^2)$. Hoặc duyệt qua các cảm biến $O(m)$ để thử bỏ từng cảm biến coi có được không sau khi đã chọn tập các cảm biến hợp lệ, mỗi lần kiểm tra ta lại phải duyệt qua $O(n^2)$ ô để tính toán và kiểm tra thử bỏ cảm biến hiện tại có được không.
- 3. Thiết lập hàm thời gian $T(n, m) = m \times n^2 \times 2$.
- 4. Lấy Big-O: $O(m \times n^2)$.

Và để xác định ta sẽ thực hiện lại việc tham lam ngẫu nhiên bao nhiêu lần thì có thể xem qua giới hạn thời gian và tốc độ của từng ngôn ngữ để xác định, như **python** thì mình để là 80 lần thì tốc độ chạy ổn và đúng hết mọi test, **c++** có thể chạy nhanh hơn nhiều nên mình để lập nhiều lần hơn 333 lần cho chắc ăn và vẫn đúng hết mọi test.

1.2.3 Phân tích độ phức tạp không gian:

- 1. Xác định các vùng nhớ được sử dụng: Ta đa số chỉ sài các biến bình thường, vector một chiều với kích thước theo m và lớn nhất là mảng hai chiều kích thước $n \times n$.
- 2. Tính tổng bộ nhớ: $O(1) + O(m) + O(n^2)$
- 3. Thiết lập hàm không gian $S(n, m) = O(1) + O(m) + O(n^2)$.
- 4. Lấy Big-O: $O(n^2)$.

1.2.4 Code python:

```python
def best_ratio(self, trick_lord: bool = False) -> Tuple[int, List[int]]:
    self.reset_pref()
    n = self.n
    pref = self.pref
    ans = 0
    ans_set = set()

    unused = set()
    for i in range(1, self.m + 1):
        if trick_lord:
            if self.rng.randint(0, 2):
                unused.add(i)
        else:
            unused.add(i)

    num_uncovered = n * n
    while num_uncovered > 0:
        best_id = -1
        best_gain = 0
        for idx in unused:
            gain = 0
            for x, y in self.cover_sets[idx]:
                if pref[x][y] == 0:
                    gain += 1
```

```
            if best_id == -1 or self.sensors[best_id][2] * gain > self.
    sensors[idx][2] * best_gain:
                best_id = idx
                best_gain = gain
        if best_id == -1 or best_gain == 0:
            self.reset_pref()
            return INF, []
        for x, y in self.cover_sets[best_id]:
            pref[x][y] += 1
        num_uncovered -= best_gain
        ans += self.sensors[best_id][2]
        ans_set.add(best_id)
        unused.discard(best_id)

    drop_delta = [[0] * (n + 2) for _ in range(n + 2)]

    def try_drop(group: List[int]) -> bool:
        touched = []
        for idx in group:
            for x, y in self.cover_sets[idx]:
                if drop_delta[x][y] == 0:
                    touched.append((x, y))
                drop_delta[x][y] += 1
        ok = True
        for x, y in touched:
            if pref[x][y] - drop_delta[x][y] <= 0:
                ok = False
                break
        if ok:
            for x, y in touched:
                pref[x][y] -= drop_delta[x][y]
        for x, y in touched:
            drop_delta[x][y] = 0
        return ok

    improved = True
    while improved:
        improved = False
        for idx in sorted(ans_set):
            if try_drop([idx]):
                ans -= self.sensors[idx][2]
                ans_set.remove(idx)
                improved = True
            if improved:
                break

    result = sorted(ans_set)
    self.reset_pref()
    return ans, result
```