Graph Pattern Matching Challenge

2015-12049 김상엽 / 2015-15636 김현빈

*** Environment : g++ (MinGW.org GCC-6.3.0-1) 6.3.0**

1. Compile and Execute

```
cd build
cmake ..
make
./main/program <data graph file> <query graph file> <candidate set file>
```

2. Functions

```
/**
 * @brief Deletes the i-th candidate from query vertex u's candidate set.
 *
 * @param u query vertex id.
 * @param i index in half-open interval [0, GetCandidateSetSize(u)).
 */
inline void CandidateSet::DeleteCandidate(Vertex u, size_t i)
```

```
/**
 * @brief Deletes whatever value @param qv is mapped to from entire candidate set.
 *
 * @param matched_to used to skip checking already-mapped vertices.
 * @param qv most recently mapped vertex.
 * @param del_temp used to store deleted value to later restore using ReverseDeletion.
 */
inline void CandidateSet::DeleteCandidate(int* matched_to, Vertex qv, std::vector<std::pair<Vertex, Vertex>> &del_temp)
```

```
/**
 * @brief Reverses the set of deletions currently at the top of our recent_deletions
 * stack. Pops stack after reversal.
 *
 * @param rd Pointer to stack which holds the deleted values.
```

```
 */
inline void CandidateSet::ReverseDeletion(std::stack<std::vector<std::pair<Vertex,
 Vertex>>> &rd)
```

```
 /**
 * @brief Prints out a match.
 */
  inline void print_match(const int* matched_to, const int num_query)
```

```
/**
 * @brief Deletes collisions created by matching a vertex.
 */
  inline void delete_collisions(Vertex qv, int* matched_to, const Graph &data, con
st Graph &query, CandidateSet &cs, std::stack<std::vector<std::pair<Vertex, Vertex
>>> &rd)
```

```
/**
 * @brief Calculates score metric used to determine backtracking order.
 *
 * @return float
 */
  inline float calc_score(Vertex v, int* matched_to, const Graph &query, Candidate
Set &cs)
```

```
/**
 * @brief Returns the next item in our backtracking order.
 *
 * @return std::pair<Vertex, float>
 */
  inline std::pair<Vertex, float> get_next(int* matched_to, const Graph &query, Ca
ndidateSet &cs, int num_query)
```

3. Pipeline

(1) Preprocessing (우선시되는 matching order)

- 가장 먼저 candidate set size가 1인 vertex들의 matching을 확정
- 본 프로그램은 matching이 발생할 때마다 `delete_collisions` 함수를 호출하여 해당 match와 충돌하는 candidate value (neighboring in query but does not neighbor in data if candidate is taken/candidate value is taken by match)를 삭제한다.

- Preprocessing에서는 가정하는 match 없이 확정적인 match만을 다루므로 여기서 발생하는 deletion들은 irreversible하다.

- 이 과정을 더 이상 변동이 없을 때까지 반복하여 초기에 match할 수 있는 값들은 모두 match하고 state space search를 시작한다.

```
  bool updated = true;
  while(updated){ // (3) Repeat process until there are no more vertices to match
using criteria (1)
    updated = false;
    for(int i = 0; i < num_query; i++) {
      if(matched_to[i] == NOT_MATCHED && cs.GetCandidateSize(i) == 1) {
        matched_to[i] = cs.GetCandidate(i, 0); // (1) First, match vertices that h
ave only one candidate
        unmatched--;
        updated = true;
        delete_collisions(i, matched_to, data, query, cs, recent_deletions); // (2
) Then find and delete collisions
      }
    }
  }
```

(2) Backtracking

```
  while(depth >= 0) {
    if(!going_down && try_loc[depth] == cs.GetCandidateSize(next[depth])) { // Pop
ped one call stack to find that we have exhausted all possibilities in this depth.
      cs.ReverseDeletion(recent_deletions); // Reverse most recent deletions.
      try_loc[depth] = 0; // Initialize this depth's try to 0.
      matched_to[next[depth-
-]] = NOT_MATCHED; // Initialize this depth's match to NOT_MATCHED.
      going_down = false; // Set direction to call stack pop.
      continue; // Go up one call stack.
    }
    std::pair<Vertex, float> next_(0, -2);
    if(going_down) { // Get next item in backtracking order only when we are movin
g deeper into the call stack.
      next_ = get_next(matched_to, query, cs, num_query); // Get next item.
      next[depth] = next_.first; // Set the vertex we are trying to match at this
depth.
    }
    if(next_.second == 0) { // Next item does not have any possibilities. The bran
ch we've taken in our state space search tree is wrong.
      cs.ReverseDeletion(recent_deletions);
      try_loc[depth] = 0;
      matched_to[next[depth--]] = NOT_MATCHED;
```

```
      going_down = false;
      continue;
    }
    Vertex cd = cs.GetCandidate(next[depth], try_loc[depth]); // Get the candidate
 we need to try next.
    matched_to[next[depth]] = cd; // Match vertex to candidate.
    delete_collisions(next[depth], matched_to, data, query, cs, recent_deletions);
 // Delete related collisions.
    going_down = true; // Set direction to next call.

    // Increment candidate search location so that when we come back to this depth
, we try the next value.
    // Increment depth by 1.
    try_loc[depth++]++;
    if(depth == unmatched) { // We have reached the end of our call stack. All ver
tices have been matched.
      print_match(matched_to, num_query); // Print out match.
      // check_match(matched_to, num_query, data, query);
      num_outputs++; // Increment num_output counter by 1.
      going_down = false;
      depth--;
      cs.ReverseDeletion(recent_deletions);
    }
    if(num_outputs == MAX_OUTPUTS) break; // If we have reached 100,000 outputs, e
nd loop and terminate program.
  }
```

backtracking 알고리즘

자세한 backtracking 작동원리는 주석에 충분히 설명되어 있으니 넘어가도록 하겠다.

- 알고리즘의 핵심은 내려가면서 collision 여부를 판단하는 것이 아닌, match마다 collision을 판별하여 candidate set에 직접적인 변화를 주고, 이를 다시 원상복귀 (atomicity가 보장되게) 할 수 있도록 deleted_element 들의 스택을 유지한다는 것이다.
- 이 때 collision 판별은 아직 match되지 않은 vertex 대상으로만 하여 불필요한 연산을 최소화해야 한다.
- 모든 collision을 cover하도록 알고리즘을 구성했다면, matching order 후반부에서 전반부와의 collision을 더 이상 고려할 필요가 없고, 뒤로 갈수록 candidate set size가 급격히 감소하므로 빠르게 답을 도출해낼 수 있다.

Next to-be-matched vertex를 뽑아오기 위한 score metric은 다음과 같은 코드로 구성되어 있다.

```
 /**
 * @brief Calculates score metric used to determine backtracking order.
 *
 * @return float
 */
```

```
  inline float calc_score(Vertex v, int* matched_to, const Graph &query, Candidate
Set &cs) {
    int selector = 1; // Selector value which determines which score metric we use
 to create a backtracking order.
    float nun = 0; // Number of unmatched neighbors

    for(size_t i = query.GetNeighborStartOffset(v); i < query.GetNeighborEndOffset
(v); i++) {
      if(matched_to[query.GetNeighbor(i)] == NOT_MATCHED) nun += 1;
    }

    size_t size = cs.GetCandidateSize(v);

    if(selector == 0) { // IF selector is 0, score = size of candidate set divided
 by number of unmatched neighbors.
      if(size == 0) return 0; // Needs to return 0 if there are no candidate sets
- used to quickly terminate branch.
      if(size == 1) return std::numeric_limits<float>::min();
      else if (nun == 0) return std::numeric_limits<float>::max();
      else return size / nun;
    } else if(selector == 1) { // IF selector is 1, score = size of candidate set.
      return size;
    } else { // IF selector is not 0 or 1, score = random float between 0 and 1 (i
nclusive).
      return size == 0 ? 0 : static_cast <float> (rand()) / static_cast <float> (R
AND_MAX); // Returns 0 if size == 0 for the same reason as above.
    }
  }
```

Score metric (minimize)

여기서 candidate set size는 collision이 제거되어 있기 때문에 꼭 확인해봐야 하는 "extendable" candidate set size를 의미한다.

보이듯이, 세 종류의 metric을 시도해봤는데, 첫 번째 metric의 number of unmatched candidates의 경우 아직 matching되지 않은 vertex 들에 미칠 수 있는 영향력의 척도로, collision을 최대화하여 빠르게 정답을 output하기 위해 활용하고자 했다. 이를 보완하는 목적으로 match-equal collision을 최대화하는 방향 (candidate vertex frequency를 활용하여) 또한 고려사항에 넣고자 하였으나, 구현 난이도에 비해 효과는 미미할 것으로 예상되었기 때문에 시도에서는 배제하였다. 두 번째 metric의 경우 가장 간단하게 candidate set size만을 활용하였고, 세 번째는 비교의 목적으로 랜덤한 score 값을 리턴해보았다.

1, 2 metric은 실질적으로 큰 차이가 없었으나 2번 metric이 iteration이 없어 계산이 조금 더 빠르기 때문에 2번 metric을 채택하였다. 3번 metric은 1, 2번에 비해 확연히 느린 속도를 보여주었다.

그럼에도 불구하고 yeast_s3과 yeast_s8은 엄청난 시간이 걸림을 관찰할 수 있었는데, 이는 잘못된 branch를 택했을 경우 너무 많은 state들이 실패하기만 하여 발생하는 문제일 것으로 예상된

다.

1, 2 metric을 번갈아가며 사용하는 metric도 시도해보았으나, 유의미한 차이는 관찰할 수 없었다.

4. Example Running

```
./main/program ../data/lcc_hprd.igraph ../query/lcc_hprd_n1.igraph ../candidate_set/lcc_hprd_n1.cs
```