Mathworks Candidate Presentation

Nguyen Dinh Pham

November-2019

Agenda

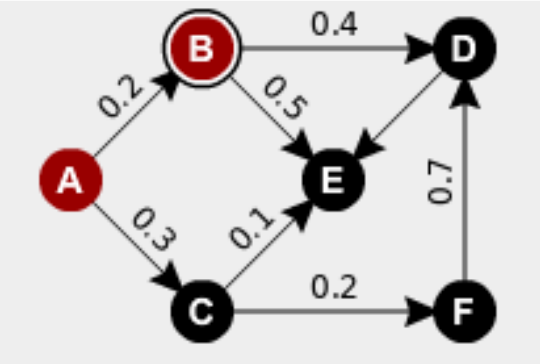
- Introduction
- C++ Project
- Another Project

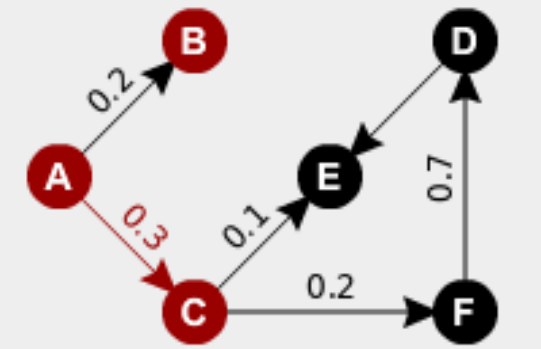
Introduction

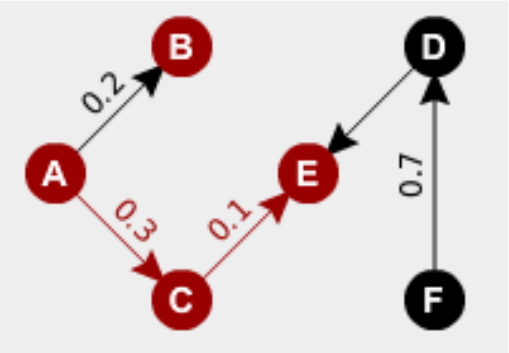
- Education: PhD in Computer Science (December 2019)
- Experience:
  - Engineer (onsite contractor), Nortel (Avaya), Ireland. Verification and performance testing, telecom-software integrated system.
  - E-commercial PHP SWE, Lazada (Alibaba), Vietnam.
  - Intern SWE, Splice Machine, USA. SQL database over Hadoop and Spark.
  - Teaching/Research Assistant.
- Interests: Concurrency, distributed computing, compiler, functional programming.
- Matlab experience: basic engineering computing (course work).

C++ Project: Graph Influence Problem

- Diffusion process over a graph. $G(V, E)$, edge $(u, v) = w$: some probability.
- S: seed set
- Activation of neighbors by the edge probability, until no more new nodes. Size of the final set: $I(S)$, the *Influence* of S.







Proposed Problem

- Influence Maximization with probabilistic guarantee.
  Motivation: Expectation is limited, for example, low variance is also desired.
  $M_\delta(I(S)) = \max\{a | \Pr[I(S) \geq a] \geq \delta\}$
  Optimization problem $\textsc{MaxProbInf}$: $|S| = k$, maximize $M_\delta(I(S))$.
- Multi-criteria Approximation with theoretical bound.
- Implementation: C++ with OpenMP.

Algorithm 1 k-Influence set by MultiCritMdelta

1: **function** MultiCritMdelta($G(V, E), k, \delta, n, N$)
2: $l \leftarrow 1$
3: $h \leftarrow n$
4: **for** $step \leftarrow [1 \ldots \log n]$ **do**
5: $S \leftarrow \emptyset$
6: $\lambda \leftarrow (l + h)/2$
7: $C \leftarrow \delta\lambda$, and $F(S) \leftarrow F_\lambda(S)$
8: **while** $F(S) < C$ **do**
9: Pick node j that minimizes $\frac{w_j}{F(S \cup \{j\}) - F(S)}$
10: $S \leftarrow S \cup \{j\}$
11: feasible if $|S| \leq k \ln(N\lambda)$: $l \leftarrow \lambda$
12: infeasible if $|S| > k \ln(N\lambda)$: $h \leftarrow \lambda$
return S

Design

- Data structures + functions, organized by namespaces.
- Simple interface, no requirement for API reuse $\Longrightarrow$ No need for OO.
- Smart pointers and modern C++ container operations.
- Simple parallel computing: Each thread performs sampling, accumulates the measurements, update the shared total accumulation.

```
namespace datatypes {
  using LInt = long long int;
  using Vertex = std::tuple<LInt, LInt>;
  using Edge = std::tuple<LInt, LInt, double>;
  struct GraphByEdges {
    std::unique_ptr<std::vector<Vertex>> vertexes;
    std::unique_ptr<std::vector<Edge>> edges;
    GraphByEdges(
      std::unique_ptr<std::vector<Vertex>> &&pV,
      std::unique_ptr<std::vector<Edge>> &&pE)
    : vertexes(std::move(pV)), edges(std::move(pE)) {}
    GraphByEdges (const GraphByEdges&) = delete;
    GraphByEdges& operator= (const GraphByEdges&) = delete;
  };
  struct NodeIndexedCover {
    LInt cc_id;
    LInt cc_size;
  };
}
```

```
namespace graph {
  std::unique_ptr<datatypes::GraphByEdges> read_edges(...);
  std::unique_ptr<std::vector<datatypes::Node>>
      get_graph_nodes(...);
  std::unique_ptr<datatypes::Graph> sample_graph(...,const
      std::unique_ptr<util::Dice> &dice);
  std::unique_ptr<std::vector<std::unique_ptr<datatypes::
      ConnectedComp>>> connected_component(
    std::unique_ptr<datatypes::Graph>& graph);
  std::unique_ptr<std::vector<datatypes::NodeIndexedCover>>
      get_cover(
    std::unique_ptr<datatypes::Graph>& graph);
}
```

```
auto node_measure =
  make_unique<vector<NodeMeasure>>(nodes->size());
#pragma omp parallel for
for (int i = 0; i < num_threads; i++) {
  auto node_indexed_measure =
    make_unique<vector<LInt>>(nodes->size(), 0);
  for (int j = 0; j < batch_size; j++) {
    auto g =
      graph::sample_graph(graph_edges, nodes, dices[i]);
    auto temp = graph::connected_component(g);
    unique_ptr<vector<NodeIndexedCover>> nics =
      graph::get_cover(g);
    _update_node_measure(
      node_indexed_measure, nics, kset_ids);
  }
  #pragma omp critical
  _gather_node_measure(node_indexed_measure, node_measure);
}
```

```
class Task {
public:
  CyclicBarrier* cb;
  std::mutex* update_lock;
  Task(CyclicBarrier* cb, std::mutex* update_lock)
    : cb(cb), update_lock(update_lock) {}
  void operator()(...) {...}
}

for (int j = 0; j < nthreads; j++) {
  Task task(cb, update_lock);
  threads.push_back(std::thread(task, ...));
}
for (auto& th : threads) { th.join(); }
```

```
class CyclicBarrier {
public:
  int const count;
  std::atomic<int> seats;
  std::atomic<int> gen;
  explicit CyclicBarrier(int count)
      : count(count), seats(count), gen(0) {}

  void wait() {
    int const cur = gen;
    if (--seats == 0) {
      seats = count; ++gen;
    } else {
      while (gen == cur) {
        std::this_thread::yield();
      }
    }
  }
};
```

Scala Project: Actor Model

- Actor: message-passing concurrent entities.
- Messages: Multiple writers, single consumer.
- Networking: Asynchronous TCP, (almost) transparent.
- Replace the low-level threads and event loops.
- Concurrency and distribution by actors and messages.

Why not Actor in C++

Requirement: Using only Standard Library.

- C++: No thread pools.
- C++: Simulate *pattern matching* is difficult (via metaprogramming).
- C++: Serialization and asynchronous networking are difficult.

Other difficulties:

- Type safety: difficult to enforce message types for actors.
- Shared messages life cycle (one message sent to many actors).

A nice implementation: CAF: `https://actor-framework.org/`

Thank you!

Q & A