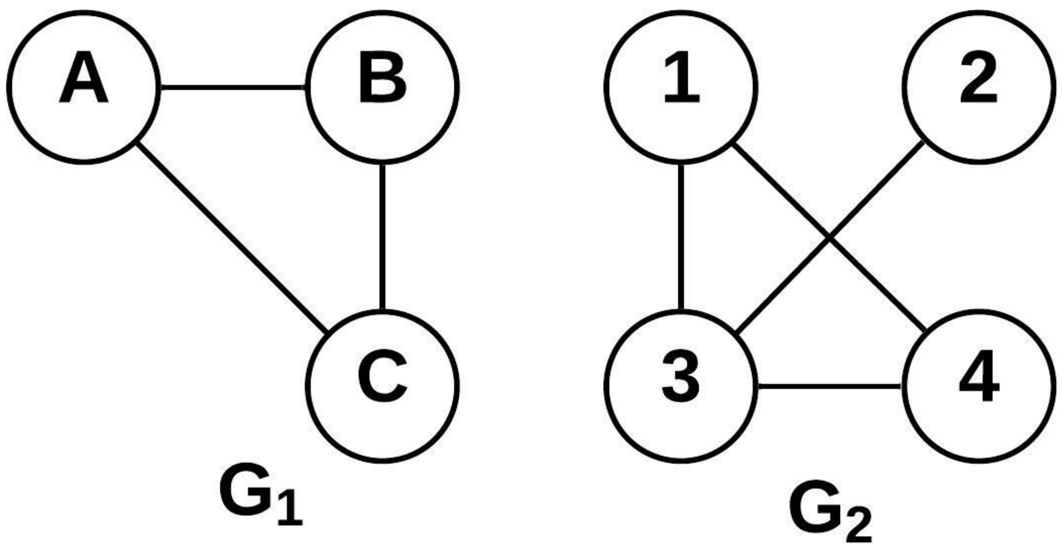
A PARALLEL ALGORITHM FOR SUBGRAPH ISOMORPHISM

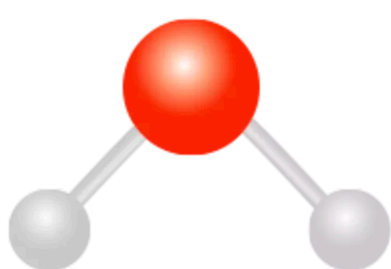
Rohan Yadav and Umut A. Acar

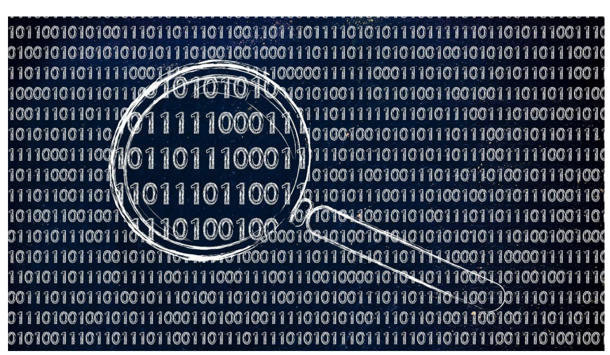
Carnegie Mellon University

SPAA 2019

SUBGRAPH ISOMORPHISM BACKGROUND



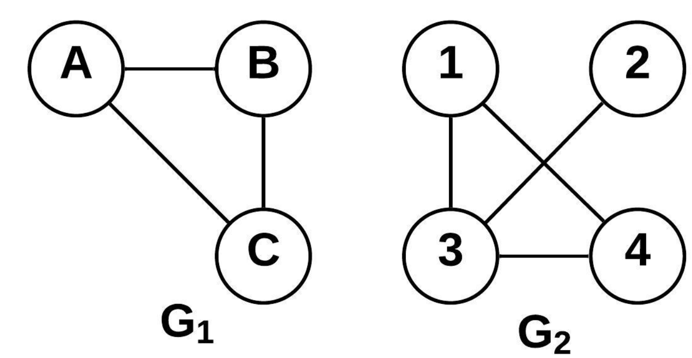




- Is G_1 a subgraph of G_2?
- Involves finding a mapping f of vertices from G_1 to G_2 that preserves edge relationships
  - For each edge (u, v) in G_1, $(f(u), f(v))$ must be in G_2
- Subgraph isomorphism is NP-Complete
- Has applications in
  - pattern recognition
  - biochemical applications
  - graph databases
  - …

STRATEGY: SEARCH ALGORITHMS

- Explore a state space to find the isomorphism
- A **match** is a tuple of vertices mapping a vertex in G_1 to a vertex in G_2
- A **matching** is a partial isomorphism between G_1 and G_2, represented by a set of matches
- Matchings grow and shrink by adding and removing matches
- A matching is **consistent** if it preserves edge relationships
- **Algorithm** —



Start with an empty matching M

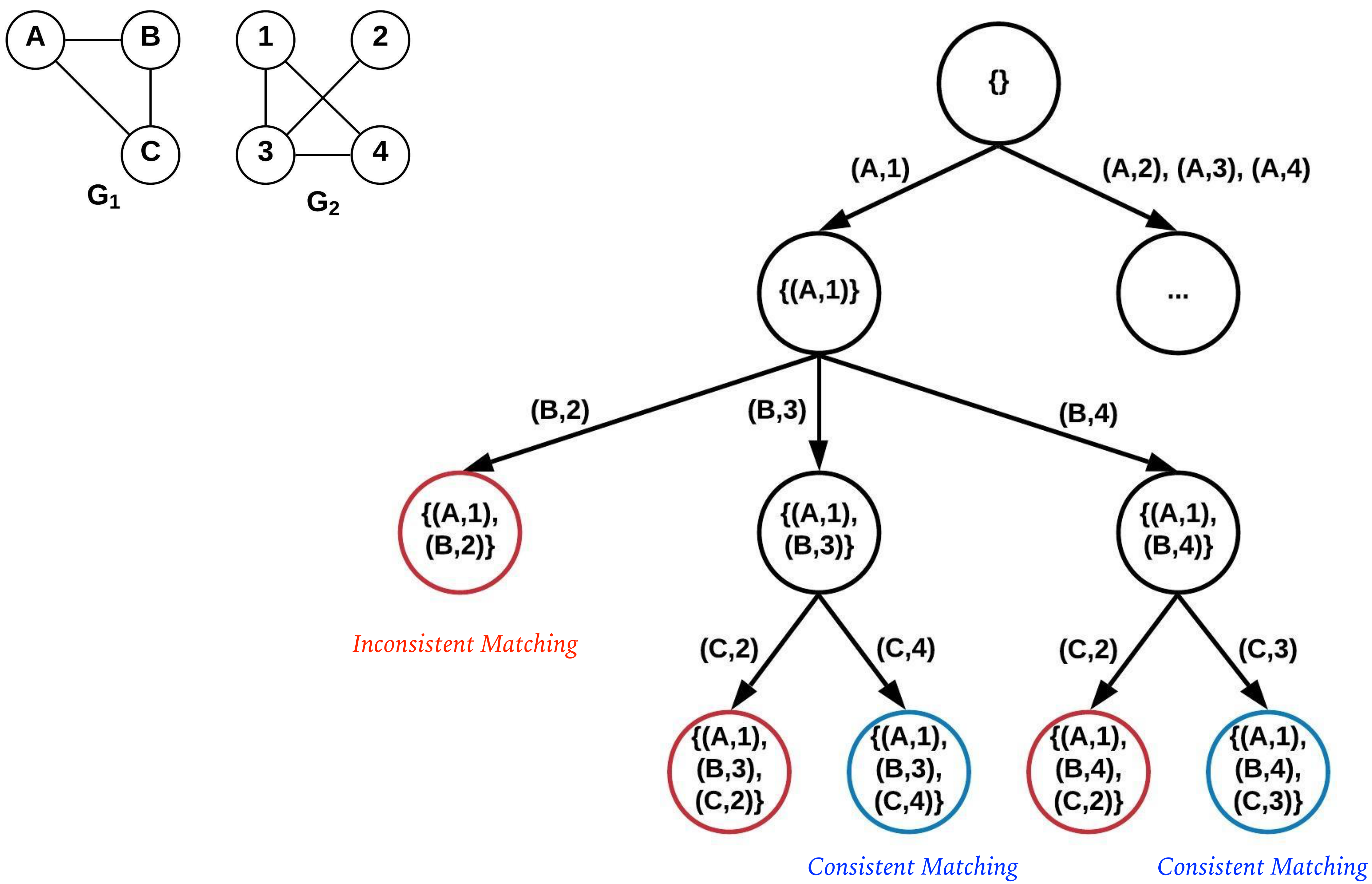
Loop:

add a new match (u, v) to M such that M is not visited

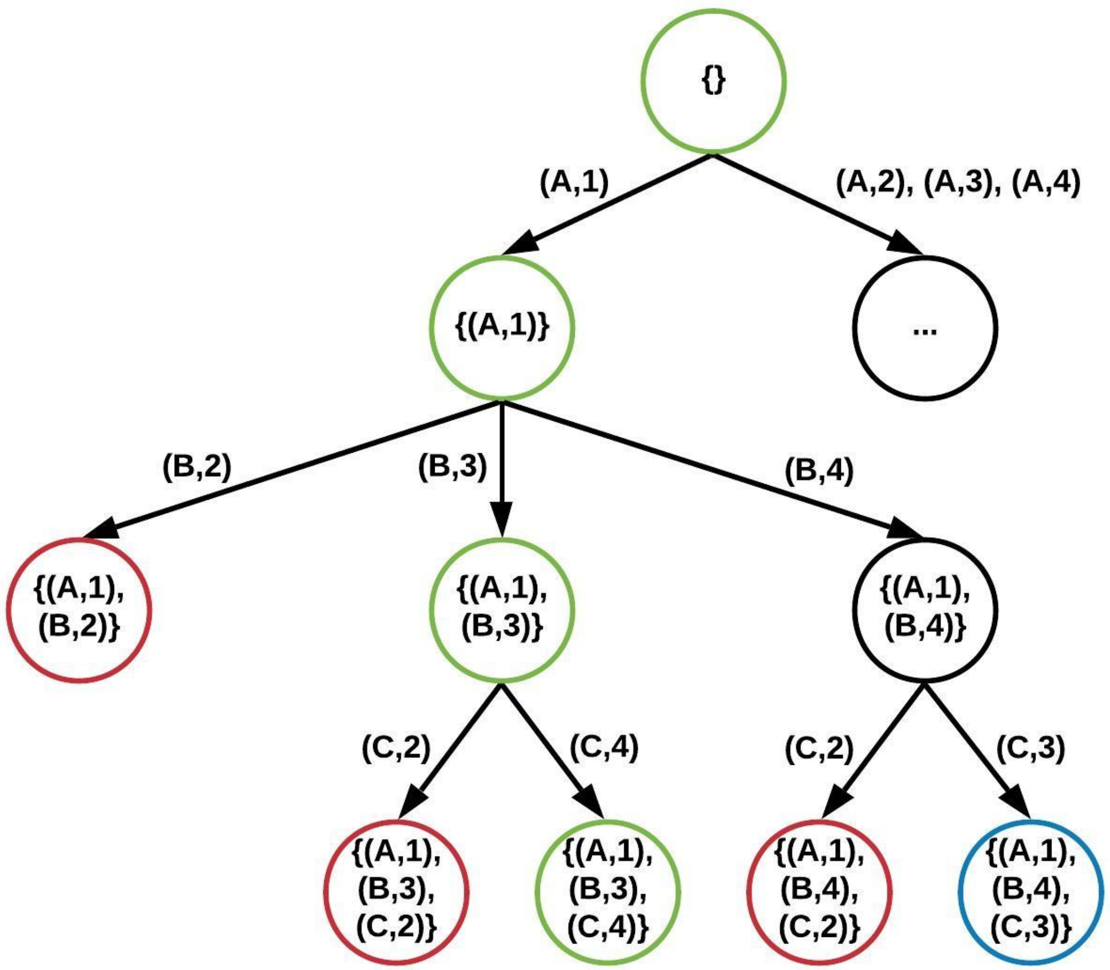
visit M if M is consistent

if M is isomorphism then done

SEARCH TREE STRUCTURE



BASIC TREE SEARCH ALGORITHM




```
def search M =
    for each child edge (u, v) of M:
        if M + (u, v) is consistent:
            add (u, v) to M
            if search M:
                return true
            remove (u, v) from M
    return false
```

- Key idea: efficient backtracking
- Core of known algorithms like VF2, RI [Cordella ’04, Bonnici ‘13]

CHALLENGES WITH PARALLELIZATION

```
def search M =
    parallel for each child edge
    (u, v) of M:
        if M + (u, v) is consistent:
            add (u, v) to M
            if search M:
                return true
            remove (u, v) from M
    return false
```

- Idea: try all children in parallel
- Performs poorly — why?
- Requires persistent matching structure
- Highly irregular branches
  - With pruning, search tree is irregular
  - Work is not predictable resulting in fine grained tasks
- Prior work has identified these issues

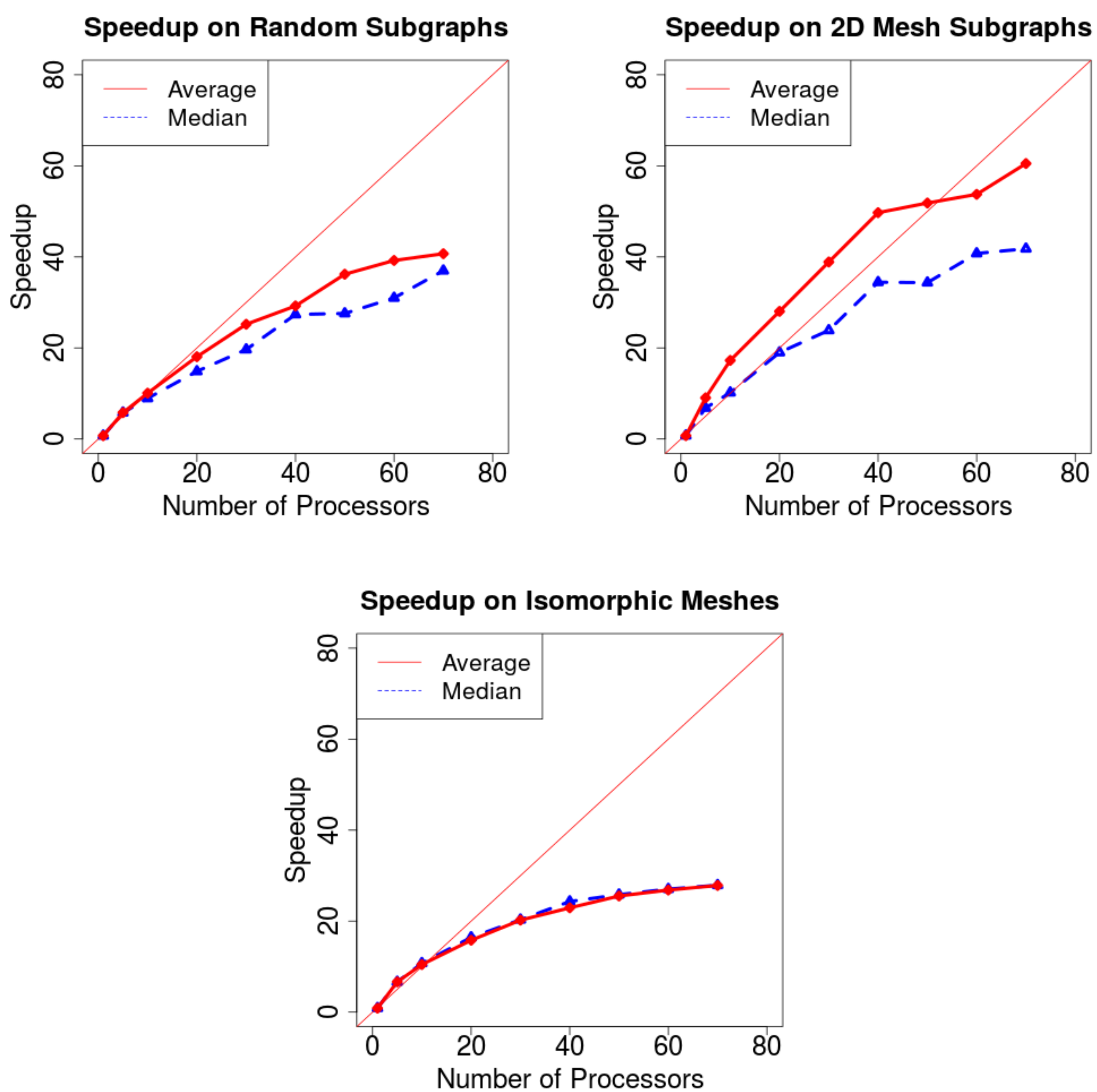
LAZY PARALLELISM

- Goal: maintain efficiency of sequential algorithm
- Create parallelism lazily on-demand
  - p workers
  - Each worker runs (almost) the sequential algorithm
  - But generates parallelism by splitting its work when requested

ALGORITHMS FOR LAZY SPLITTING

- Frontier data structure for representing work of each worker
  - Push and pop edges from frontier to explore
  - Split frontier when requested to share work
- Backtracking from failed searches
  - Frontier structure represents a search path
  - Split operation returns contiguous search paths
  - Frontier invariants make it possible to efficiently implement backtracking
- Scheduling for irregular and unpredictable parallelism
  - Amortization technique: workers share work only after performing enough work to pay for sharing [Acar et al. ’15]

PRELIMINARY RESULTS



Subset of preliminary results

- Based on VF2 implementation
- Tested on the database designed by the creators of the VF2 algorithm
- Overhead over VF2 algorithm is between 15%-40%
- Established bounds on single core work efficiency and *p*-processor space overhead

FUTURE WORK

- Extend the core ideas of our algorithm to more recent algorithms like VF3 [Vincenzo '16]
- Perform a larger experimental evaluation with more data sets and different metrics (aggregate speedup, etc.)
- Explore theory and analysis of lazy splitting style algorithms, prove guarantees about the span of our algorithm