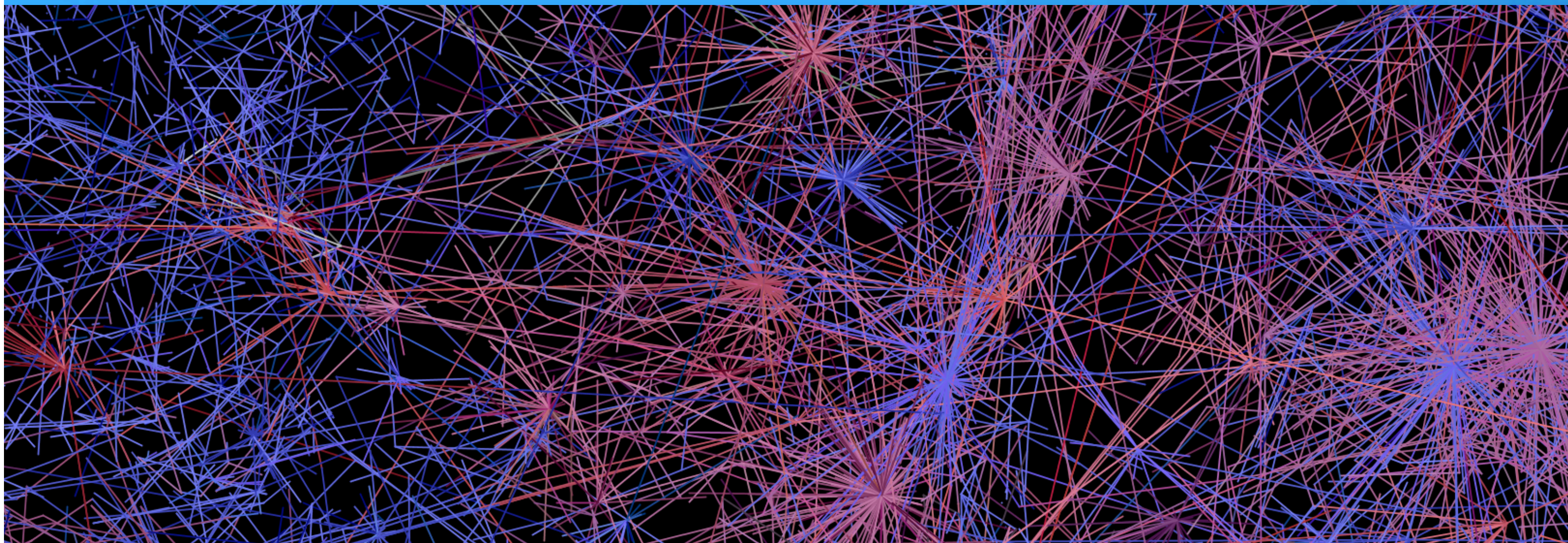


Load Balancing for Traffic in Networks

(with Dóra Erdös)

Feiyu Shi

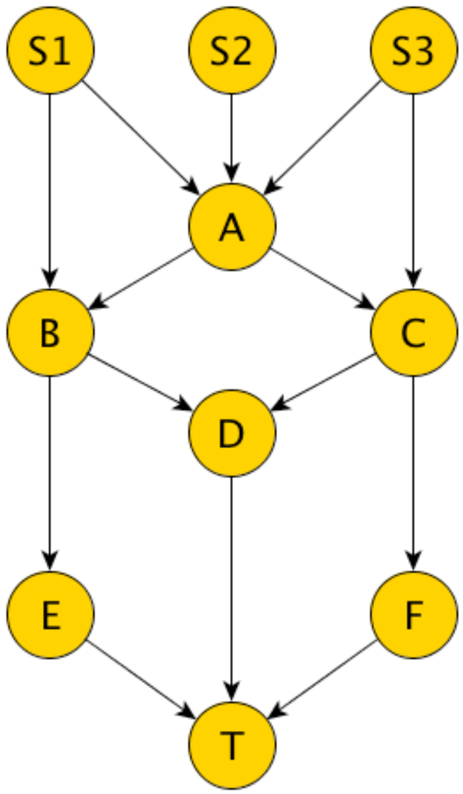
4/29/2014

Recap

- Problem:
- Given a set of k nodes, partition it into **two** groups such that the difference of total number of shortest paths covered by each group is minimal.
- Steps: node centrality -> group centrality -> load balancing
- ~~Original setting: in Undirected Graph~~
- Current setting: in Directed Acyclic Graph (DAG)

Example Graph

- Given a group of nodes: A, B, C, D, E, F, divide this group into two groups, such that the difference of the group centralities of these two groups are minimized.



Node Centrality

- Prefix: $PREFIX(v) = \#PATHS(S,v) = \sum_{s \in S} \#PATHS(s,v)$
- Suffix: $SUFFIX(v) = \#PATHS(v,T) = \sum_{t \in T} \#PATHS(v,t)$
- Impact: $I(v) = PREFIX(v) \times SUFFIX(v)$
- From general paths to shortest paths: check if parents of v are on a shortest path. If d(s,v) – d(s,x) = 1, then x is in Pi'(v). Only includes parents from Pi'(v) in the computation of prefix and plist (suffix).

Compute Prefix

- Observation: every path from source s to a node v has to go through one of v's parents.
- Recurrence relation:
- Base: $PREFIX(s) = 1$
- Induction: $PREFIX(v) = \sum_{x \in Pi(v)} PREFIX(x)$
- Order: topological order determines ancestors of v.
- Thus, $PREFIX(v) = \#PATHS(S,v) = \sum_{x \in Pi(v)} \#PATHS(S,x)$

Compute Suffix

- $Plist_v$: contains for every ancestor y of v the number of paths that go from y to v.
- Another recurrence relation:
- Base: $PLIST_y[y] = 1$
- Induction: $PLIST_v[y] = \sum_{x \in Pi(v)} PLIST_x[y]$
- Finally: $SUFFIX(v) = \sum_{t \in T} PLIST_t(v)$

Group Centrality

- Suppose current set A's group centrality is C(A). After we add another node v to set A, we get the new group centrality C'(A).
- Define the conditional impact: $I_A(v) = C'(A) - C(A) \leq I(v)$.
- If A = empty set, $I_{\varnothing}(v) = I(v)$
- It's the number of shortest paths that were not covered by A.
- Set Prefix(a) to zero;
- Set $Plist_v(a)$ to zero.

Load Balancing: Problem

- Let $G(V,E)$ be a DAG, where |V| = n. Given a group of nodes, $V_0 \subseteq V$ where $|V_0|$ = k. Find an assignment of two groups V1 and V2, where $V_1 \cap V_2 = \varnothing$ and $V_1 \cup V_2 = V_0$, such that the difference of group centralities of V1 and V2 is minimized.
- Recall: brute-force algorithm and greedy algorithm

Baseline: Brute-force Algorithm

- Enumerate all possible assignments: 2^k combinations.
- The evaluate the group centralities of each possibility.
- Find the solution having the minimal group centrality difference.
- Slowest, but can find all the best assignments.

Implementation of Brute-force Alg.

- First, find the power set of the set of k nodes.
- For example for set of {A, B, C}, its power set is{ ∅, {A}, {B}, {C}, {A, B}, {A, C}, {B, C}, {A, B, C}}.
- Second, compute the group centrality of each set in the power set.
- Third, compute the difference of one set and its complement set. Find the minimum of the differences.
- Finally, find all assignment of sets that equals the min diff.

Greedy Algorithm

- Inspired by the concept of conditional impact.
- If we move node v from G1 to G2, then the new centralities of two groups $C1' = C1 - I_{G1}(v)$, $C2' = C2 + I_{G2}(v)$.
- Then new difference: $C1' - C2' = (C1 - C2) - (I_{G1}(v) + I_{G2}(v))$.
- Heuristic: choosing the node whose new difference is 'absolutely' closest to the previous difference.

Implementation of Greedy Algorithm

- Input: G(V,E), Vo
- Output: V1 or V2
- 1. V1 = Vo, V2 = $\varnothing$
- 2. Compute diff = $C_{sp}(V1) - C_{sp}(V2)$.
- 3. Compute diff' = $C'_{sp}(V1) - C'_{sp}(V2)$. Find v_i whose |diff'| is minimal (there may not be only one v_i, but randomly choose only one).
- 4. Remove the node from V1 and add it to V2.
- if |diff'| <= |diff| and V1 is not empty then
  - 5. recursively solve the problem with new V1, V2.
- else
  - 6. return old V1 or V2.
- end if.

Improvement: Greedy Search Alg.

- Based on greedy algorithm.
- Can find more than one solution if there exists many solutions. But it's not guaranteed that it can find all the solutions. (Remember brute-force alg. can find all the solutions.) The order to pick nodes matters.
- New: Backtracking search. Try all possibilities. Then choose the best combination.
- New: Pruning. Maintaining a list 'visited'.

Implementation of Greedy Search Alg.

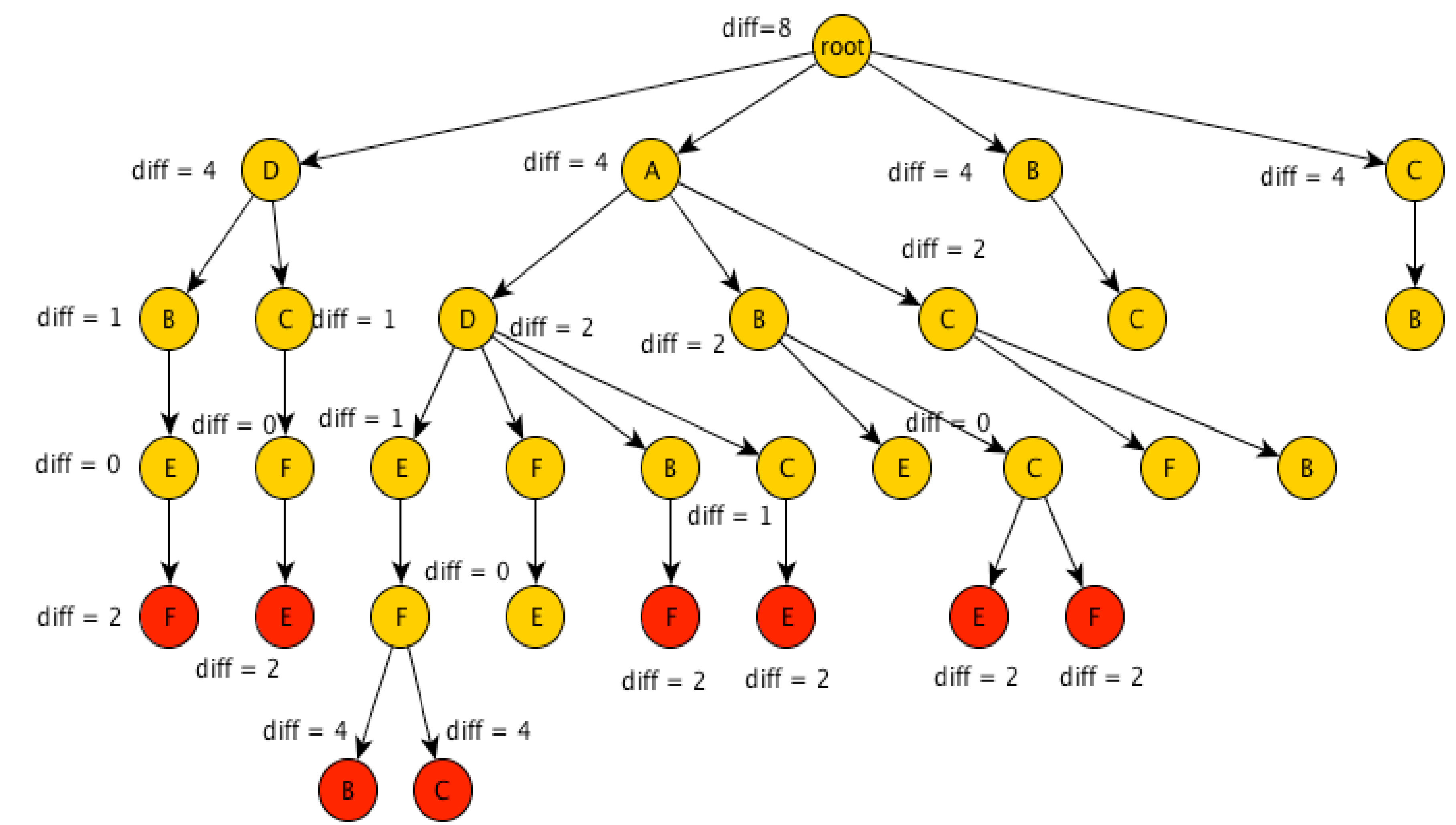
- Input: G(V,E), Vo
- Output: A list of V1 or V2
- 1. V1 = Vo, V2 = $\varnothing$
- 2. Compute diff = C_{sp}(V1) – C_{sp}(V2).
- 3. Compute diff' = C'_{sp}(V1) – C'_{sp}(V2). Find all v_i whose |diff'| is minimal ......

- Return a list of assignments by choosing the best ones with min |diff|.

Greedy Search Alg. (Cont.)

- for each v_i do
  - 4. Remove v_i from V1 and add it to V2.
  - If neither new V1 nor new V2 is in the list 'visited' then
    - 5. Compute the new diff'.
    - If |diff'| < |diff| and V1 is not empty then
      - If diff' = 0 then add new V1, V2 to list 'assign' and 'visited'. end if
      - 6. recursively solve the problem with new V1, V2.
    - else
      - 7. add old V1or V2 to list 'assign' and 'visited'.
    - end if
  - end if
  - 8. Return a list of assignments from 'assign' with min difference
- end for

Search Graph of Greedy Search Alg.



Full Search Algorithm

- Similar to greedy search algorithm, but need to compute group centralities for each node and relax the constraint:
  - Instead of using greedy heuristic, we loop over all possible nodes.
- Give all possible solutions, the same as brute-force method.
- Spends more time (even more than brute-force).

Time Complexity Analysis

- Recall : the time complexity for computing group centrality is $O(k\,|E|\,\Delta)$. Node centrality is $O(|E|\,\Delta)$.
- Brute-force: $O(2^k\, k|E|\Delta)$.
- Greedy: k level. 1 node / level. So $O(k^2|E|\Delta)$.
- Full search: worst case $k!(1+1+1/2!+1/3!+\ldots+1/(k-1)!) \approx k!\, e$. So $O(k!\, k|E|\Delta)$.
- Greedy search: somewhere between $O(k^2|E|\Delta)$ and $O(k!|E|\Delta)$, and usually $< O(2^k\, k|E|\Delta)$ (from test result).

Experiments

Dataset Preparation, Validation, Time complexity

Dataset (Thanks to Dóra)

- Synthetic DAG dataset:
- Three parameters: n, d, l. n is the number of nodes in the graph, d is a density parameter and l is a limit.
- Node are labeled as integers. For every node i to node j, we try to generate an edge with probability d if $i < j \le i+l$. $d*l$ = expected degree of every node. $|E| \approx d*l*n$.
- We generated
  - Dataset1: 10 graphs with varying d from 0.1 to 1.0 (n=200, l=10).
  - Dataset2: 10 graphs with varying n from 100 to 1000 (not used).
  - Data: 1 DAG, n = 50, d = 0.4, l = 5.
- We compare performance of three algorithms on these datasets.

Dataset (cont.)

- Dataset1: Choose first 10 nodes as sources, last 10 nodes as destinations.
- DAG 50_0.4_5: first 5 nodes as sources, last 5 nodes as destinations.
- Choice of group:
  - Dataset1: 5 nodes. For graph with n nodes, choose 0.4n, 0.45n, 0.5n, 0.55n, 0.6n.
  - DAG 50_0.4_5: 10 groups with number of nodes from 4 to 13. Smaller group $\subseteq$ larger group.

Tests

- Two aspects: validation and time complexity.
  - Validation: test if the algorithm returns (all) the correct answer.
  - Time complexity: measure the running time for each alg. in ms.
- Two set of experiments:
  - On dataset1: focus on validation
  - On DAG 50_0.4_5: focus on time complexity (k).

Dataset1: Validation

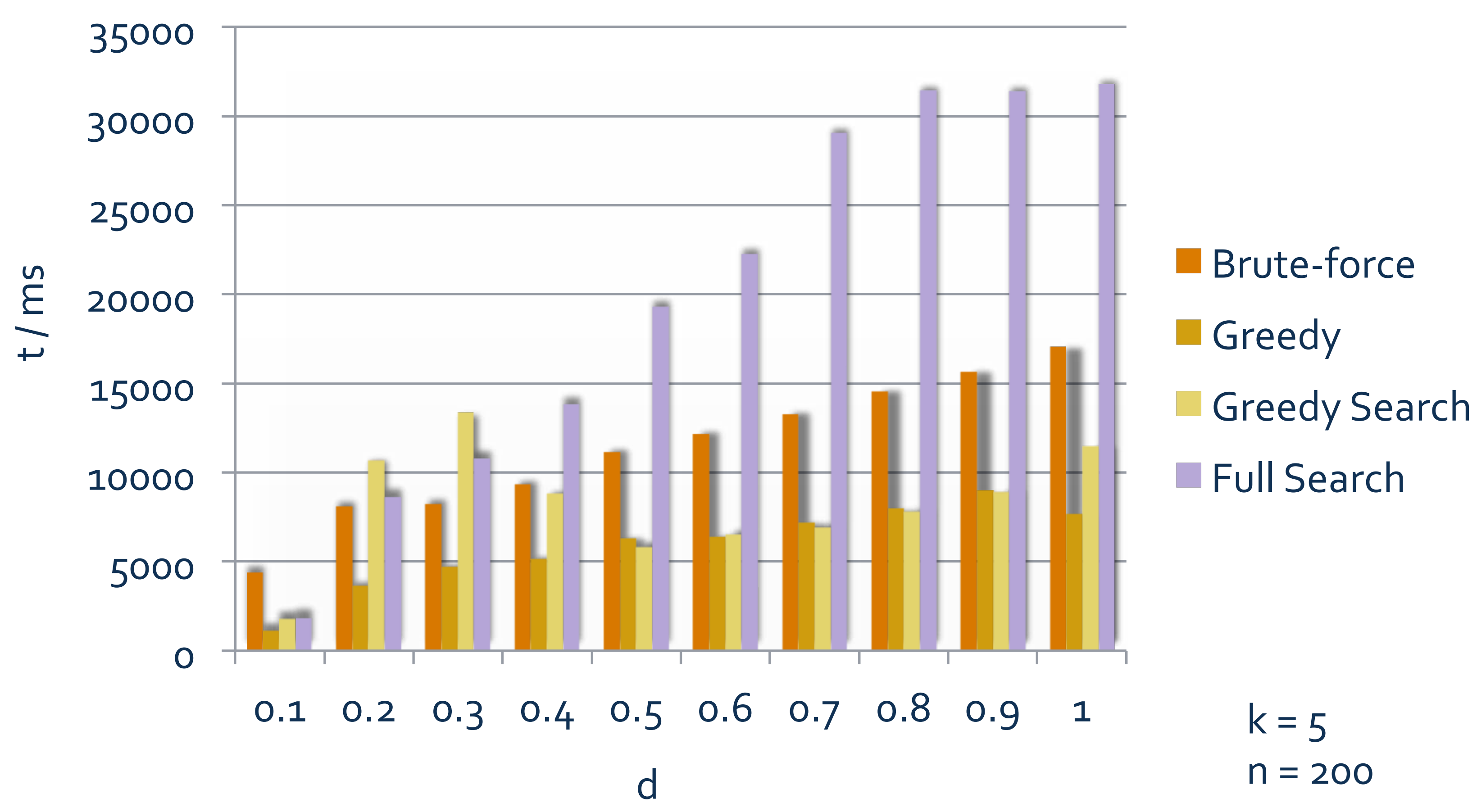
Test	1	2	3	4	5	6	7	8	9	10
num.	16	16	16	8	4	2	1	1	1	1
G	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
GS #	16	16	16	7	3	1	1	1	1	1
GS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
R %	100	100	100	87.5	75	50	100	100	100	100
FS #	16	16	16	8	4	2	1	1	1	1
FS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
R %	100	100	100	100	100	100	100	100	100	100

1. num.: total number of solutions（from baseline method).
2. G: greedy validation.
3. GS #: number of solutions from greedy search method.
4. GS: greedy search validation.
5. R %: retrieval rate of greedy search method; GS # / num.

Dataset1: Validation (cont.)

- Special cases:
  - 1. 0 vs. 0 (test 1)
  - 2. largest vs. 0 (test 2, 3)
- Sample test results:
  - 227568 vs. 19836
  - 32148083 vs. 43532838

Dataset1: Time Complexity

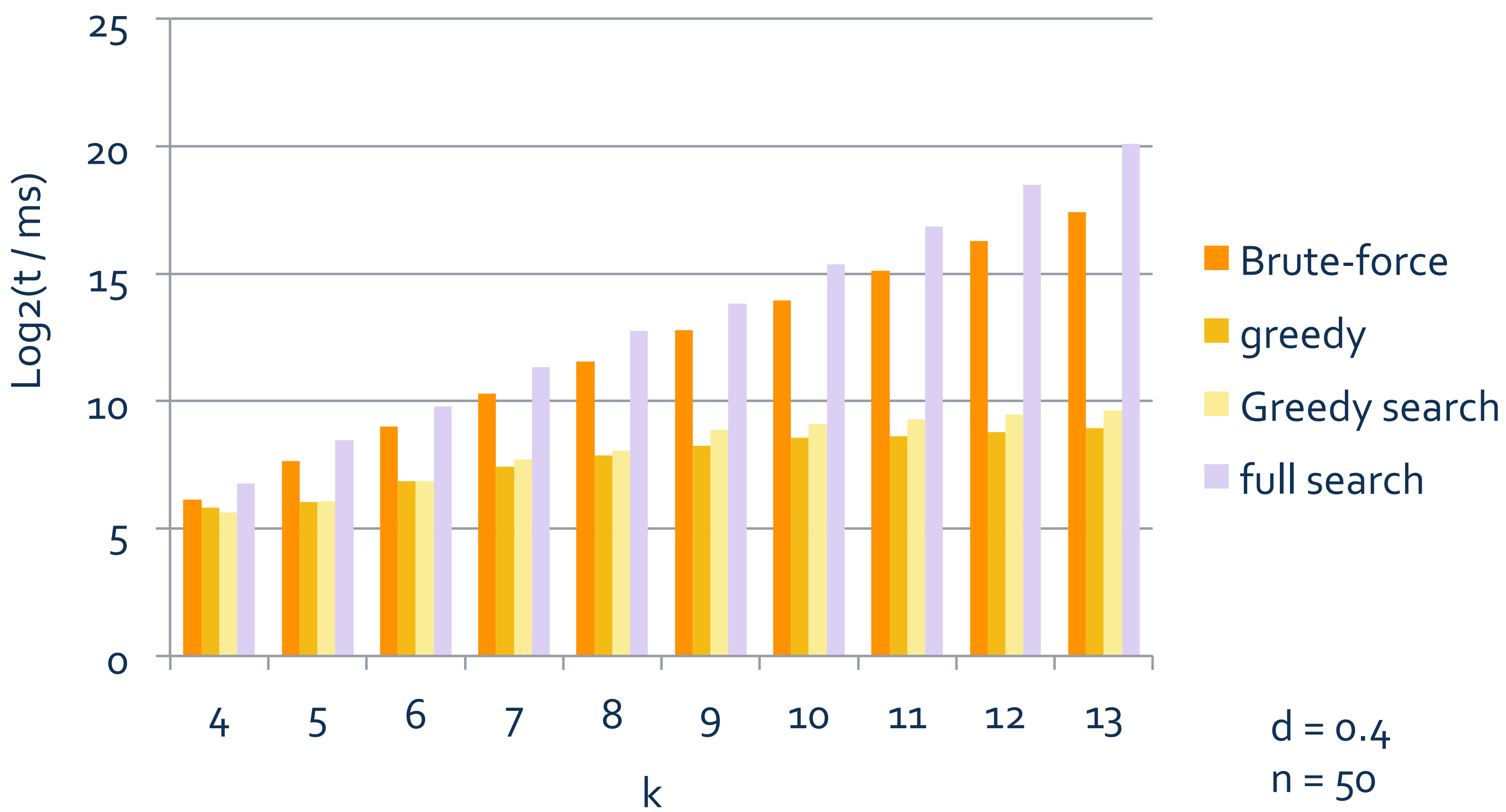


DAG 50_0.4_5: Validation

k	4	5	6	7	8	9	10	11	12	13
num.	1	1	2	4	4	8	8	8	8	8
G	Y	Y	Y	Y	Y	Y	N	Y	Y	N
GS #	1	1	1	3	3	7	0	7	7	0
GS	Y	Y	Y	Y	Y	Y	N	Y	Y	N
R %	100	100	50	75	75	87.5	0	87.5	87.5	0
FS #	1	1	2	4	4	8	8	8	8	8
FS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
R %	100	100	100	100	100	100	100	100	100	100

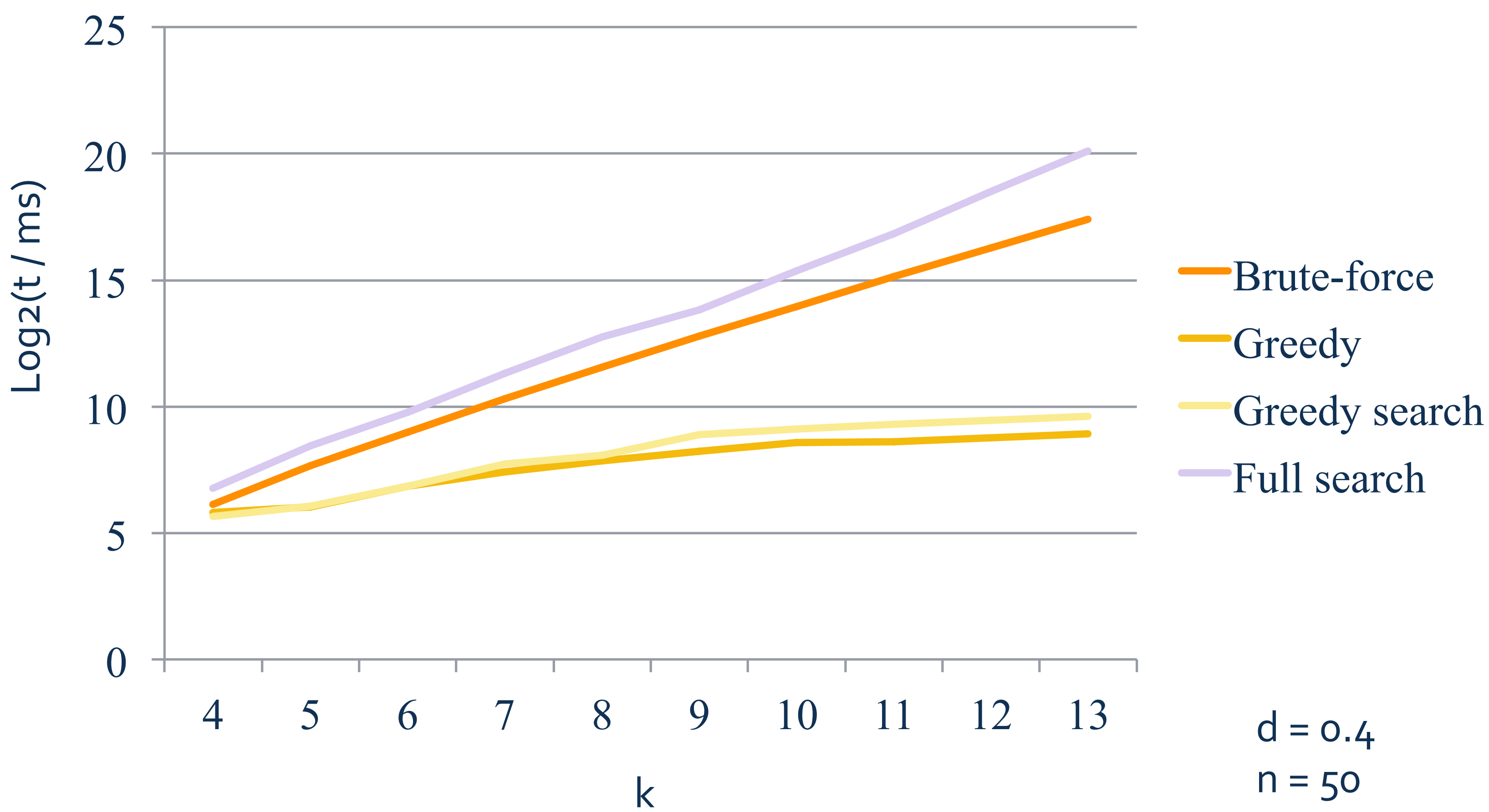
Note:
Impacts from greedy-like alg (k = 10): 751 vs. 733. Baseline: 711 vs. 700.

DAG 50_0.4_5: Time Complexity



d = 0.4
n = 50

DAG 50_0.4_5: Time Complexity



d = 0.4
n = 50

Conclusions

- Greedy algorithm is good enough (90% chance) sometimes.
- Greedy search algorithm works as good as greedy algorithm. But it might be suitable when there is a need for more than one possible solutions.
- Brute-force and full search algorithms are able to get all possible solutions.
- Future study may be applying these algorithms to undirected graphs.

Questions?