algowiki-project.org /en/Purdom's_algorithm

Purdom's algorithm - Algowiki

12-15 minutes

Алгоритм Пурдома	
Sequential algorithm	
Serial complexity	$O(\|E\|+\mu 2)$
Input data	$O(\|E\|+\|V\|)$
Output data	$O(\|V\|2)$
Parallel algorithm	
Parallel form height	N/A
Parallel form width	N/A

Primary author of this description: I.V.Afanasyev.

Contents

1 Properties and structure of the algorithm

1.1 General description of the algorithm

Purdom's algorithm[1] finds the transitive closure of a directed graph in time $O(|E|+\mu|V|)$, where $\mu\leq|E|$ is the number of strongly connected components of this graph. The algorithm can be modified to verify whether node pairs in the given set belong to the transitive closure.

1.2 Mathematical description of the algorithm

Purdom's algorithm is based on the following considerations:

1. If nodes v and w belong to the same strongly connected component of the graph G, then its transitive closure $G+$ contains the arcs (v,w) and (w,v).
2. If nodes x and y belong to the same strongly connected component of the graph G and the same is true of nodes z and t , then the arcs (x,z), (x,t), (y,z), and (y,t) simultaneously either belong or not belong to the transitive closure.

Thus, the search for the transitive closure of the graph G reduces to finding the transitive closure of the acyclic graph G obtained by merging each strongly connected component of G into a single node. The transitive closure of an acyclic graph is calculated with the use of the topological sort of its nodes.

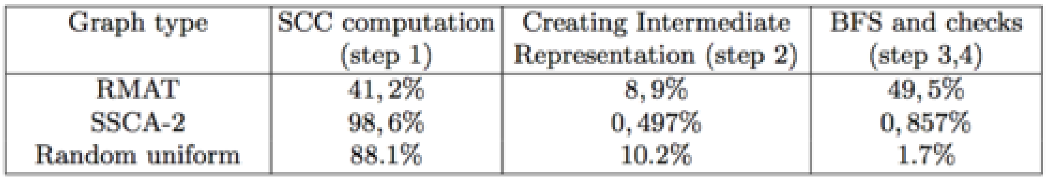
1.3 Computational kernel of the algorithm

The algorithm consists of three important computational stages. It is essential to assess the relative execution time of each stage. The stages are:

1. Search for the strongly connected components in the original graph.
2. Creating intermediate representation.

3. Breadth-first searches in the intermediate representation resulting in the final transitive closure.

1. Suppose that the algorithm should verify whether node pairs in the given set belong to the transitive closure. In this case, the relative weight of an individual computational stage may vary widely. It obviously depends on the number of input node pairs. Besides, it may depend on the type of the input graph because the structure and the number of strongly connected components affect strongly both the time for finding these components and the time for performing breadth-first searches. The table below demonstrates (for a certain implementation of Purdom's algorithm) the percentage of the total execution time spent on the individual stages. Various types of graph were examined. The number of nodes was 223, and the number of input node pairs was 10000.



Graph type	SCC computation (step 1)	Creating Intermediate Representation (step 2)	BFS and checks (step 3,4)
RMAT	41, 2%	8, 9%	49, 5%
SSCA-2	98, 6%	0, 497%	0, 857%
Random uniform	88.1%	10.2%	1.7%

Figure 1. Comparison of execution times of the individual stages for a certain implementation of Purdom's algorithm

It is important to notice that some algorithms for finding strongly connected components (for instance, the DSCS algorithm) are also based on breadth-first searches. Thus, breadth-first searches can be regarded as the computational kernel of the algorithm. The computational kernel of the breadth-first search is, in turn, the traversal of neighbor nodes for a singled-out node and the addition of not yet visited nodes to the set of "advanced" nodes.

2. Now, suppose that the full transitive closure is sought. Then the computationally most laborious stage are the breadth-first searches in the intermediate representation because they should be performed starting from all of its nodes. Their number can be significant if the original graph has a large number of strongly connected components.

1.4 Macro structure of the algorithm

There are four stages in the computation:

1. Find the strongly connected components of the original graph, replace each component by a single node, and remove the resulting loops.
2. Perform the topological sort of the acyclic graph G obtained at stage 1.
3. Calculate the transitive closure of G, moving from nodes with larger indices to those with smaller ones.
4. Reconstruct the transitive closure of the original graph from the transitive closure of G.

The last stage is not required if the transitive closure of G is regarded as a "packed" transitive closure of G.

1.5 Implementation scheme of the serial algorithm

Stage 1 (*calculation of strongly connected components*) can be implemented by using Tarjan's algorithm[2]. This algorithm finds strongly connected components in the course of the depth-first search.

Stage 2 (*topological sort*) can be implemented either by Kahn's algorithm [3] or by successively using the depth-first search[2] for a preorder numbering of nodes.

Stage 3 (*transitive closure* of G) is performed by the following algorithm:

```
Input data:
    acyclic graph $G$ with nodes $V$, indexed in the topological order, and arcs $E$.
Output data:
    collection of nodes $T(v)$ for each node $v \in V$; thus, the transitive closure of $G$ consists of arcs $(v, w)$, $v \in V$, $w \in T(v)$.

for each $v \in V$ do $T(v) := \{ v \}$
for $v \in V$ in reverse order:
    for each $(v, w) \in E$ do $T(v) := T(v) \cup T(w)$
```

Stage 4 (*transitive closure* of G) is performed by the following algorithm:

```
Input data
    graph G with nodes V;
    strongly connected components SCC(v) of G;
    graph [math]\tilde G[/math] with nodes [math]\tilde
V[/math];
    transitive closure [math]\tilde T[/math] of
[math]\tilde G[/math].
Output data:
    collection of nodes T(v) for each node v ∈ V; thus,
the transitive closure of G consists of arcs (v, w), v ∈
V, w ∈ T(v).

for each v ∈ V do T(v) := { v }
for each v ∈ [math]\tilde V[/math]:
    for each w ∈ [math]\tilde T(v)[/math]:
        T(v) := T(v) ∪ SCC(w)
    for each x ∈ SCC(v):
        T(x) := T(v)    /* assignment by reference */
```

Or one of the following functions can be used for constructing the transitive closure from the packed data $T(v)$:

```
Input data
    graph G with nodes V;
    strongly connected components SCC(v) графа G;
    mapping R(v) of the nodes of G on the nodes of
[math]\tilde G[/math]; graph [math]\tilde G[/math] with
nodes [math]\tilde V[/math];
    transitive closure [math]\tilde T[/math] of
[math]\tilde G[/math].

function transitive_closure(v):
    T := { v }
    for each w ∈ [math]\tilde T(R(v))[/math]:
        T := T ∪ SCC(w)
    return T
```

```
function is_in_transitive_closure(v, w):
    return R(w) ∈ [math]\tilde T(R(v))[/math]
```

1.6 Serial complexity of the algorithm

The first two stages are performed with the use of depth-first search and have the complexity $O(|E|)$.

The basic operation of the third stage is the merging of node lists. If these lists are stored in a sorted form, their merging is executed in linear time. The number of lists does not exceed $|V|$, and each list contains at most μ nodes, where μ is the number of strongly connected components. Then the overall complexity is $O(\mu2)$. An alternative format for storing node lists is a bit mask; the overall complexity remains the same.

The complexity of the fourth stage is $O(\mu|V|)$. Indeed, for each of the μ strongly connected components, one calculates its list of nodes in linear time, and each list contains at most $|V|$ nodes. Since the strongly connected components are disjoint, it is not necessary at this stage to store lists in a sorted form.

Thus, the overall complexity is $O(|E|+\mu2)$ if the explicit construction of the transitive closure is not required and $O(|E|+\mu|V|)$, otherwise.

1.7 Information graph

The information graph of Purdom's algorithm shown in figure 2 demonstrates connections between the basic parts of the algorithm. This version corresponds to the case where the algorithm should verify whether node pairs in the given set belong to the transitive closure.

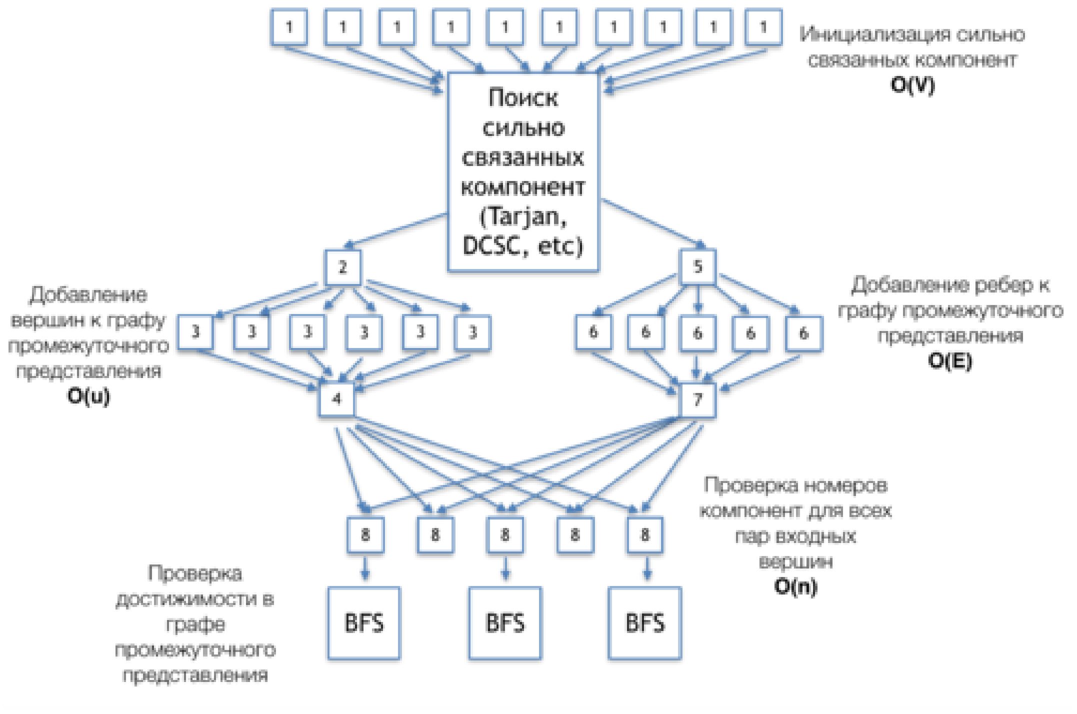

Инициализация сильно связанных компонент
O(V)
Поиск сильно связанных компонент (Tarjan, DCSC, etc)
Добавление вершин к графу промежуточного представления
O(u)
Добавление ребер к графу промежуточного представления
O(E)
Проверка номеров компонент для всех пар входных вершин
O(n)
Проверка достижимости в графе промежуточного представления
BFS

Figure 2. Information graph of Purdom's algorithm.

[1] - initialization of the array for storing the indices of strongly connected components.

[The search for strongly connected components (Tarjan, DCSC, etc)] is an independent block of this algorithm. It can be realized by any algorithm for finding strongly connected components: DCSC, Tarjan etc. The information graphs of these algorithms can be found in the corresponding sections.]

[2] - memory allocation for the nodes of the intermediate representation.

[3] - addition of nodes to the intermediate representation. Each node corresponds to a single strongly connected component.

[4] - calculation the number of nodes in the intermediate representation.

[5] - memory allocation for the arcs of the intermediate representation.

[6] - addition of arcs connecting different strongly connected components to the intermediate representation.

[7] - calculation the number of arcs in the intermediate representation.

[8] - test of the indices for each pair of strongly connected components.

[BFS] - breadth-first searches within the same strongly connected component.

1.8 Parallelization resource of the algorithm

In this section, we discuss the parallelization resource of Purdom's algorithm using its information graph shown in figure 2.

The initialization of strongly connected components [1] can be done in parallel and requires $|V|$ operations.

The parallelization resource of the search for strongly connected components can vary and depends on the choice of algorithm.

The addition of nodes and arcs to the intermediate representation [2]-[7] requires $|E|$ and $|u|$ operations (where u is the number of strongly connected components in the original graph), which can be executed in parallel. Moreover, nodes and arcs are added to the graph independently from each other; consequently, the addition of nodes can be performed in parallel with the addition of arcs.

Tests for the indices of strongly connected components [8] can also be done in parallel. They require $|n|$ operations (where n is the number of output node pairs), which can also be executed in parallel.

If both nodes of a pair belong to the same strongly connected component, then a breadth-first search [BFS] is performed. These searches for different pairs can also be done in parallel. The parallelization resource of the breadth-first search is described in the corresponding section.

The height and width of the parallel form depend on the structure of graph (that is, on the number and location of strongly connected components) because this structure affects the height and width of the parallel form of algorithms for finding strongly connected components.

1.9 Input and output data of the algorithm

The size of input and output data depends on the variant of the problem to be solved. Below, we consider two cases.

1. Search for the full transitive closure

Input data: graph $G(V,E)$ with $|V|$ nodes v_i and $|E|$ arcs $e_j=(v^{(1)}_j,v^{(2)}_j)$.

Size of the input data: $O(|V|+|E|)$.

Output data: indicate for each pair of graph nodes whether they belong to the transitive closure.

Size of the output data: $O(|V|^2)$.

2. Test for the membership in the transitive closure for n given node pairs (v_i,v_j)

Input data: graph $G(V,E)$ with $|V|$ nodes v_i and $|E|$ arcs $e_j=(v^{(1)}_j,v^{(2)}_j)$, n node pairs (v_i,v_j).

Size of the input data: $O(|V|+|E|+n)$.

Output data: indicate for each input pair of nodes whether they belong to the transitive closure.

Size of the output data: $O(n)$.

1.10 Properties of the algorithm

2 Software implementation of the algorithm

2.1 Implementation peculiarities of the serial algorithm

2.2 Possible methods and considerations for parallel implementation of the algorithm

2.3 Run results

2.4 Conclusions for different classes of computer architecture

3 References

1. ↑ Purdom, Paul, Jr. “A Transitive Closure Algorithm.” Bit 10, no. 1 (March 1970): 76–94. doi:10.1007/BF01940892.
2. ↑ Jump up to: 2.0 2.1 Tarjan, Robert. “Depth-First Search and Linear Graph Algorithms.” SIAM Journal on Computing 1, no. 2 (1972): 146–60.
3. ↑ Kahn, A B. “Topological Sorting of Large Networks.” Communications of the ACM 5, no. 11 (November 1962): 558–62. doi:10.1145/368996.369025.