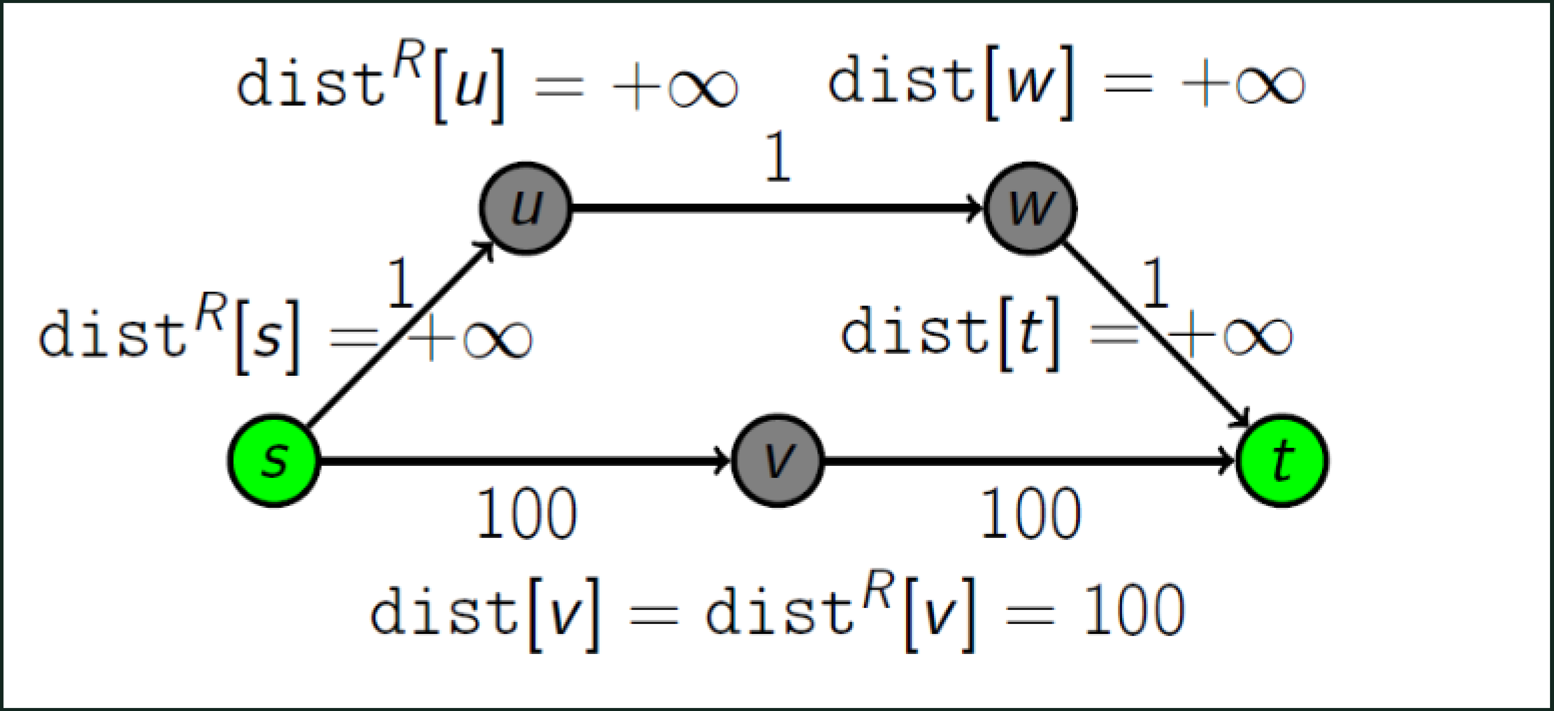
1. The Bidirectional Dijkstra algorithm from the lectures stops processing nodes as soon as it finds some node v that is ***processed*** (that is, extracted from the priority queue, and all the outgoing edges from it are relaxed) both by forward and backward search. Will the algorithm still work correctly if it stops processing nodes as soon as it finds some node v that is ***discovered*** (that is, put into the priority queue after some relaxation of the incoming edge) by both forward and backward searches?

1 / 1 point

[✓ No] Yes

Correct

Correct! In the graph below, after s is ***processed*** in the forward search and t is processed in the backward search (processed nodes are marked with green), nodes u and v are ***discovered*** in the forward search, and nodes v and w are ***discovered*** in the backward search (discovered nodes are marked with gray). Thus, node v is ***discovered*** both by forward and backward searches. However, for node v, we get $dist[v] + dist^R[v] = 100 + 100 = 200 > 3 = d(s,t)$, and for all other nodes either $dist$ or $dist^R$ is infinite, so the distance between s and t would be computed incorrectly.


$\mathrm{dist}^R[u] = +\infty$
$\mathrm{dist}[w] = +\infty$
1
u
w
1
$\mathrm{dist}^R[s] = +\infty$
1
$\mathrm{dist}[t] = +\infty$
s
v
t
100
100
$\mathrm{dist}[v] = \mathrm{dist}^R[v] = 100$

2. Suppose we've launched a ***correct*** Bidirectional Dijkstra algorithm from s and t, and it has stopped processing nodes after it had found a node v ***processed*** by both forward and backward searches. Is it necessarily true that for any shortest path P between s and t, any node u of this path is ***processed*** by either forward or backward search or both?

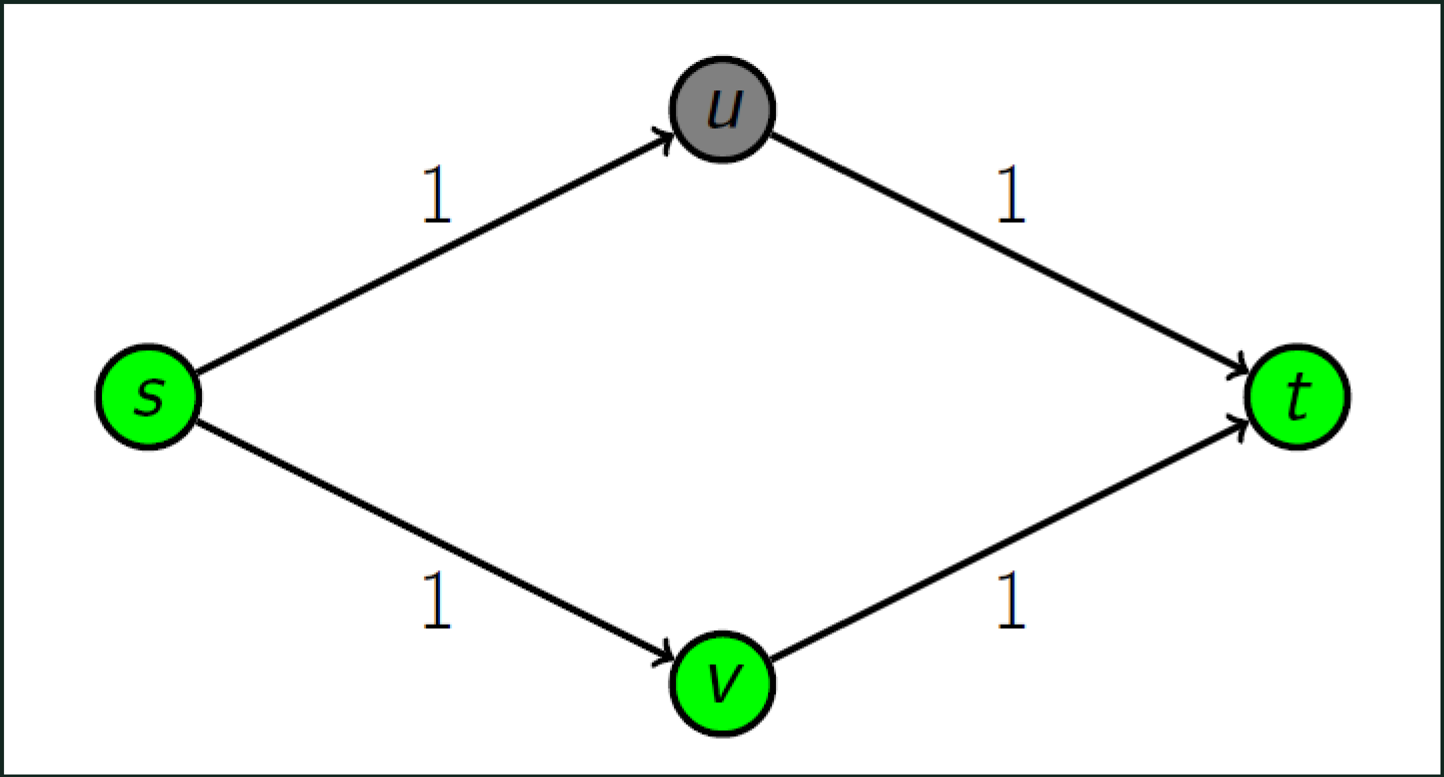
1 / 1 point

[✓ No]

Yes

Correct

Correct! In the graph below, after node s is processed in the forward search and node t is processed in the backward search (processed nodes are marked with green), nodes u and v are discovered in both searches, and they are both at distance 1 from both s and t. It could be so that v is processed before u both in the forward search and in the backward search (the order is undefined between u and v, because they are at the same distance, so this would depend on the implementation). In this case, the Bidirectional Dijkstra will stop processing nodes after v is processed by both forward and backward searches, node u won't be processed at all by any of the two searches, while there is a shortest path $s \to u \to t$ from s to t passing through u.



3. If given a weighted graph we substitute Dijkstra's algorithm by Breadth-First Search for forward and backward searches of the Bidirectional Search, and do everything else the same way, will this combination find correct distances?

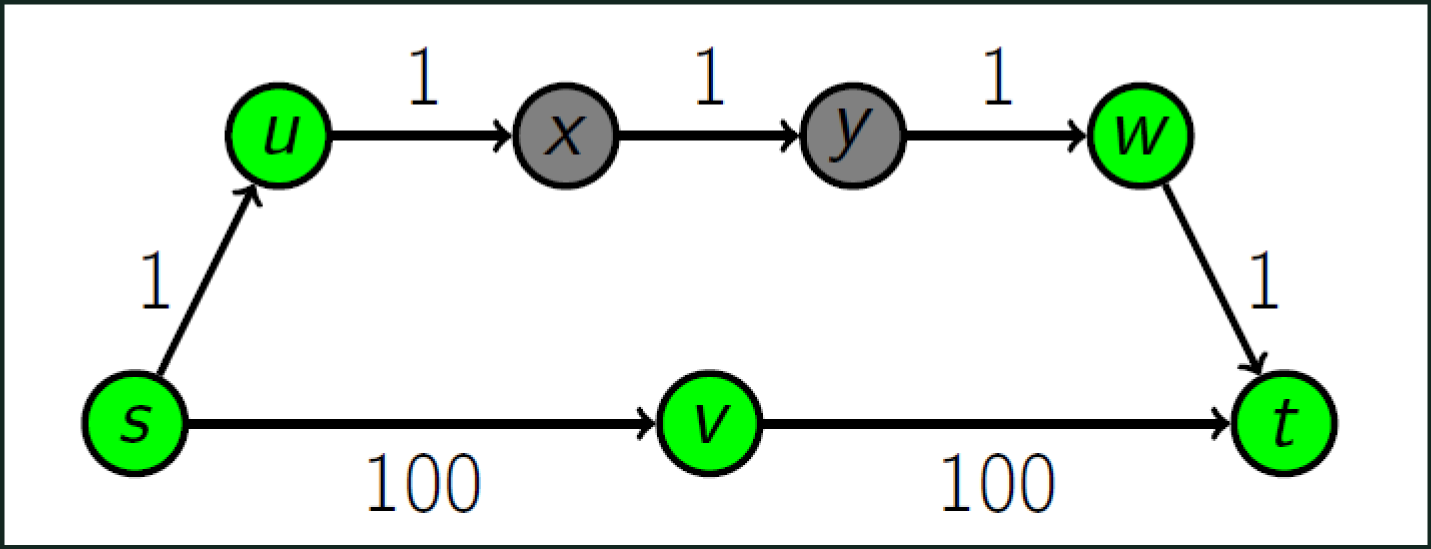
1 / 1 point

○ Yes

[✓ No]

Correct

Correct! Breadth-First Search finds and processes nodes in the order of increasing number of edges from the source node, and this is not the same as the order of increasing distance from the source node. For the graph below, after both forward and backward Breadth-First Searches process all nodes reachable via at most 1 edge from s or t, node v will be processed by both forward and backward searches. However, the path $s \to v \to t$ has length $100 + 100 = 200$, while the shortest path from s to t is $s \to u \to x \to y \to w \to t$ and has length $1 + 1 + 1 + 1 + 1 = 5$, but it won't be found, because at this moment x is not even discovered in the backward search, and y is not even discovered in the forward search.



4. In the Bidirectional Dijkstra algorithm from the lectures, we alternated steps of the forward and backward search. We want to optimize our algorithm, and instead alternate taking several steps of the forward search with taking several steps of the backward search, such that we switch from one search to another as soon as the last processed node in the current search is at strictly greater distance from the source of the current search than the last processed node from the alternative source in the alternative search. Will this algorithm work correctly?

1 / 1 point

○ No

[✓ Yes]

Correct

Correct! Our proof of correctness didn't use the way we alternate between the two searches. The only condition it used was that we stop when there is a node processed by both searches. So, changing the way we alternate between searches doesn't influence the correctness of the algorithm.

5. We want to implement an algorithm to solve the famous 15 puzzle for us. This puzzle can be thought of as a problem of finding the shortest path between the current configuration and the ideal configuration of the tiles when all the tiles are put in order as required. There is an edge from a configuration c_1 to another configuration c_2 if and only if there is a single move in position c_1 such that after making this move we get configuration c_2. The length of each edge is 1. Then the shortest path between the current configuration and the ideal one corresponds to the shortest sequence of moves solving the puzzle. As all the edges have length 1, we could try solving this puzzle with a Breadth-First Search algorithm. However, it turns out that there are so many different configurations of the 15 puzzle that this Breadth-First Search will require a lot of time to find solutions for some of the configurations.

1 / 1 point

The running time can be significantly improved by using A* algorithm. To do this, we need to find a good potential function which would be a lower bound on the distance from the current configuration to the ideal one. Which of the following heuristics would be a feasible potential function (select all that apply)?

- [✓] $\pi(c)$ = number of tiles in c which are in the wrong positions (different from their final positions).

Correct
Indeed, with each move at most one tile of the configuration changes its position, so at most one tile which was misplaced is now in the correct position, so this is indeed a lower bound on the number of moves from the current position to the ideal one.

- [ ] $\pi(c)$ = sum of the numbers written on the tiles
- [ ] $\pi(c)$ = row number of the free tile + column number of the free tile
- [✓] $\pi(c)$ = sum of the taxicab distances (or manhattan distances) from the current tile position to the position of this tile in the ideal configuration, over all the tiles.

Correct
Indeed, with each move at most one tile of the configuration changes its position, and the taxicab distance to the ideal position of this tile changes by at most one, so this is indeed a lower bound on the number of moves from the current position to the ideal one.

6. Consider the A* algorithm with the best possible potential $\pi(v) = d(v, t)$. Is it true that A* algorithm given such potential will only discover the edges of some shortest path from the source node to the target node?

1 / 1 point

Yes [✓ No]

Correct
With such potential, any edge on the shortest path from the source node to the target node will have length 0, and all the edges not lying on any shortest path will have bigger length. However, there can be several shortest paths between the source node and the target node, and in this case A* algorithm could discover not just edges of some shortest path, but also some of the edges of the other shortest paths.

7. If the forward potential for the bidirectional A* algorithm is the best possible ($\pi_f(v) = d(v, t)$), and the backward potential is just constant 0 ($\pi_r(v) = 0$), will this combination of forward and backward potentials work?

1 / 1 point

[✓ No] Yes

Correct
We need the edge lengths to be the same in the forward search and the backward search, and for this we need $\pi_f(u) + \pi_r(u)$ to be the same constant for all nodes u if the graph is connected. However, the values of π_f are different for different nodes if there are edges of non-zero length, while the values of π_r are the same for all nodes, so their sum cannot be constant.

8. If we store everything optimally, is it possible that the amount of memory needed to store the graph preprocessed by Contraction Hierarchies is smaller than the graph without preprocessing used by a regular Dijkstra's algorithm?

1 / 1 point

[✓ Yes, it is possible.]

No, it is not possible, because we add edges to the graph in the preprocessing phase, so the graph only gets bigger.

✓ Correct
We don't need to store some of the edges in the preprocessed graph: if there is an edge (u, v), such that $rank(u) > rank(v)$, it won't be used in the query when processing node u, so we can just remove it from the list of edges outgoing from u. Although shortcuts are added during preprocessing, this means that up to $\frac{1}{2}$ of the initial edges could be removed after preprocessing. In practice, it sometimes happens that for good implementations of Contraction Hierarchies they even save memory as compared to the regular Dijkstra's algorithm.

9. Can we stop the bidirectional search used in the query phase of Contraction Hierarchies as soon as some node v is processed both by forward and backward search?

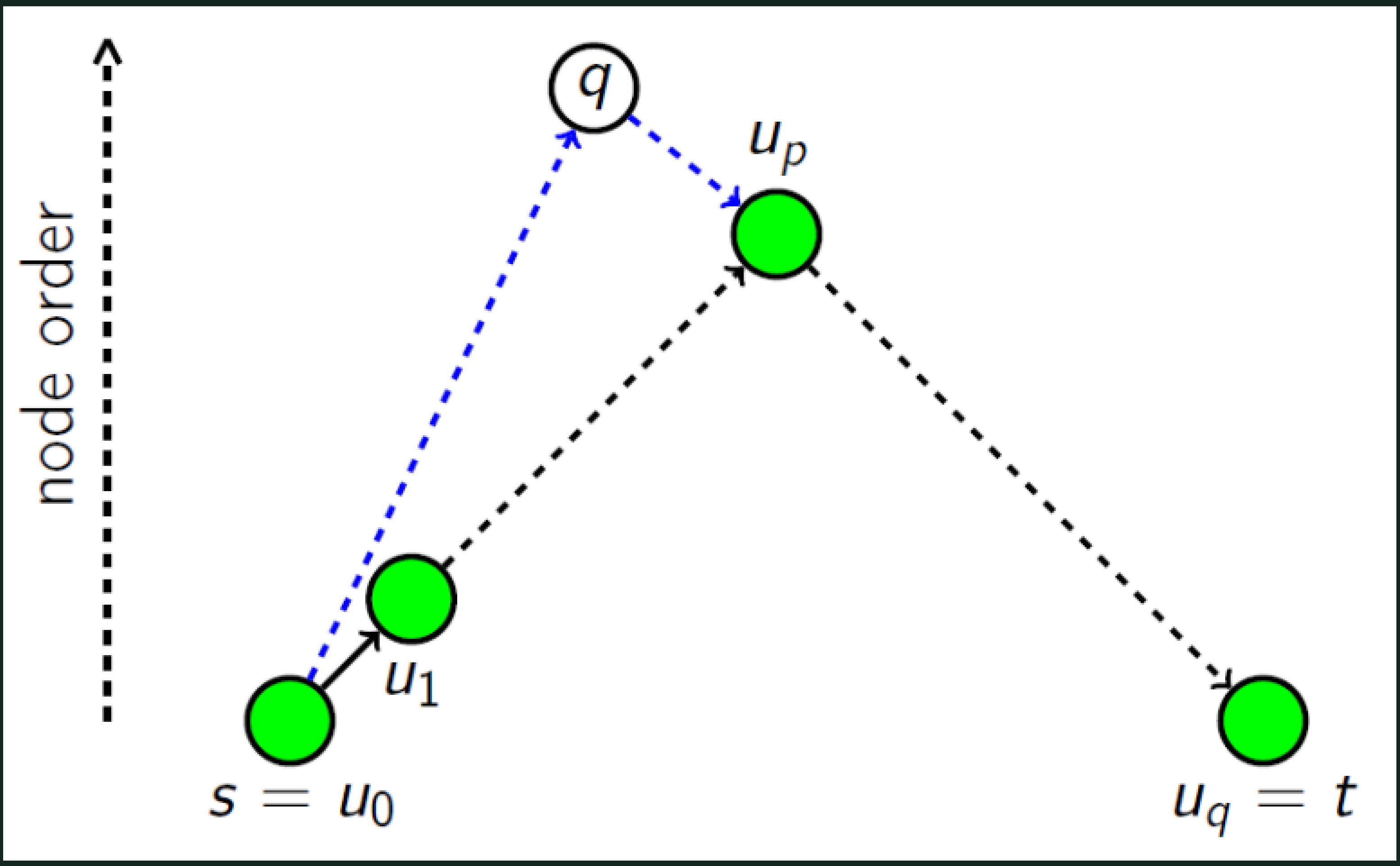
1 / 1 point

Yes

[✓ No]

✓ Correct
We restrict our forward and backward search to consider only increasing and decreasing paths respectively. The algorithm discussed in the lectures considers all the combinations of increasing and decreasing paths such that increasing and decreasing parts of those paths are shorter than the best currently found path between the source and the target. If we, however, stop as soon as some node is processed twice, it could turn out, for the example below, that there is a shorter path (marked as blue) going through up to node q, then down to u_p, then down to t as in the path found by the algorithm. The path from s to u_p through q could be shorter than the increasing path from s to u_p, because it is not increasing, so we didn't even consider it in the forward Dijkstra's algorithm when determining the distance from s to u_p.

Note that this also gives us the following idea for query optimization. When a node u is extracted from the priority queue in the forward search, we should look through all its predecessors. If there is some predecessor q such that $dist[q] + \ell(q, u) < dist[u]$, then we don't need to process the node u now: it will lead to a suboptimal path anyway. We can assign $dist[u] \leftarrow dist[q] + \ell(q, u) + 1$ (so that we still have a chance to find an increasing path from s to u of length exactly $dist[q] + \ell(q, u)$ and then relax the edges outgoing from u) and continue to the next iteration without processing u.



10. During the preprocessing phase, we need to know the following for each node:

1/1 point

1. Lists of incoming and outgoing edges.
2. Rank of the node $r(u)$ - its position in the order of contraction, to be used in the query phase.
3. Number of shortcuts added $s(u)$ in case u is contracted.
4. Edge difference $ed(u)$.
5. Number of contracted neighbors $cn(u)$.
6. Shortcut cover $sc(u)$ - the number of neighbors of u for which we will add at least one shortcut after contracting u.
7. Level $L(u)$ of the node u - an upper bound on the number of edges in any shortest path from any node s to u in the augmented graph.
8. Node importance $I(u) = ed(u) + cn(u) + sc(u) + L(u)$.

We want to reduce the amount of memory needed for preprocessing, so we want to store only some of these variables for each node, and compute the others on the fly while checking which shortcuts should be added if node u is contracted. Which of the following can be done efficiently?

● Store only the lists of incoming and outgoing edges, the rank $r(u)$ of the node (store some special value instead of the rank if the node is not contracted yet) and the level $L(u)$ of the node (store some special value instead of the level if the node is not contracted yet), and compute everything else based on it.

○ Store only the lists of incoming and outgoing edges and the rank $r(u)$ of the node (store some special value instead of the rank if the node is not contracted yet), and compute everything else based on it.

○ Store only the lists of incoming and outgoing edges and the number of shortcuts added in case u is contracted, and compute everything else based on it.

✓ **Correct**

Indeed, this is sufficient. The number of shortcuts $s(u)$ can be computed on the fly while checking which shortcuts need to be added if the node u is contracted. The edge difference $ed(u)$ is computed using the number of shortcuts added $s(u)$ and the sizes of the lists of incoming and outgoing edges. The number of contracted neighbors $cn(u)$ can be computed on the fly by looking through all the neighbors of the node and counting those for which the rank is already defined (it means that these nodes have been contracted already). The shortcut cover $sc(u)$ can be computed in the process of adding the shortcuts, if we temporarily store for each neighbor of the node whether there was a shortcut added to or from it, or not (these flags can be immediately forgotten after processing the current node). The level of the node being contracted $L(u)$ can be computed as the maximum of the levels of its contracted neighbors (the level is already defined for them) plus one. The node importance $I(u) = ed(u) + cn(u) + sc(u) + L(u)$ is based on 4 other values which we already either store or can compute on the fly.