Enumerating all connected subgraphs

Günter Rote

September 30, 2025

Redelmeier [7] described an algorithm for enumerating polyominoes, i.e., connected subsets of squares in the square lattice.

More generally, this algorithm works for enumerating connected sets containing a specified root vertex s_0 in an arbitrary undirected graph G. We are given a threshold k and we wanto to enumerate all connected graphs up to size k, or only those of size exactly k.

1 Redelmeier's algorithm

The algorithm maintains a connected subset $S \subseteq V$, drawn black in Figure 1. The *neigbors* N are the vertices in $V \setminus S$ that are adjacent to some vertex of S. The remaining vertices $V \setminus (S \cup N)$ are the *unseen* vertices. They are not shown at all in Figure 1. The neigbors N are partitioned into the *excluded* set X (gray) and the *untried set* U (white).

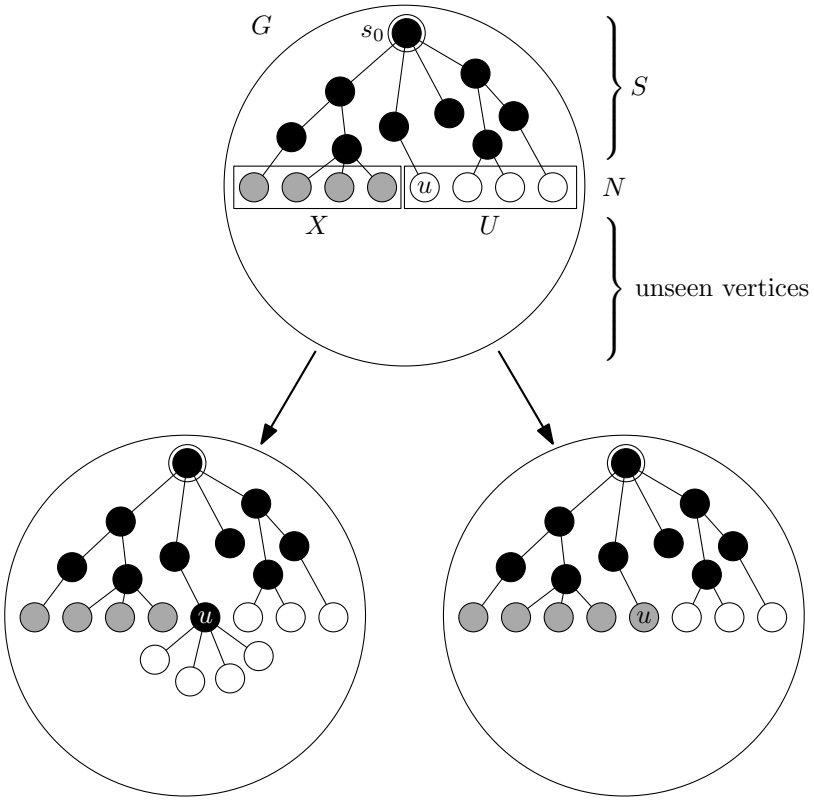

Figure 1: Schematic drawing of the sets S, X, U, and branching according to the vertex u.

From this configuration, the algorithm will generate the set $\mathcal{C}_k(S, U)$ of all connected subsets (up to the size limit k) that

1. contain S,
2. do not contain any vertex of X.

If $|S| = n$, or if the untried set U is empty, there is nothing else to do. Otherwise, the algorithm picks an arbitrary untried vertex $u \in U$ and partitions $\mathcal{C}_k(S, U)$ into two classes: those solutions that contain u and those that don't.

- In the first branch, u is moved from U to S, and in addition, all *new neighbors* of u (those that were in the unseen set) are added to the untried set U.
- In the second branch, u is simply moved from U to X.

Both branches are treated recursively.

Figure 1 shows a spanning tree T of $S \cup N$: Whenever a new neigbor v of u is added to U, we make v a child of u. The resulting tree plays no role for the algorithm, but it helps to understand the process. The graph G itself will have more edges, but they are not shown.

The general algorithm leaves the choice of $u \in U$ open. Redelmeier proposed to organize U as a stack and take u from the top of the stack. He points out that, with an implementation of the stack as a linked list, this facilitates the management of the set U for recursive invocations: Instead of copying U, it can be saved and restored in constant time.

However, this is also true when we organize U as a queue. In that case, we can get constant-time operations even when impementing the queue in a fixed-size array. (See the prototype implementation in PYTHON in Appendix A.)

2 Enumerating spanning trees

We have seen that Redelmeier's algorithm implicitly creates a spanning tree of S.

A connected set S has in general many spanning trees, but there are ways to associate a unique spanning tree to S. One can take any spanning-tree algorithm, such as depth-first search (DFS) or breadth-first search (BFS). The resulting spanning tree will in general depend on the order in which the incident edges of a vertex are visited. Therefore, we will assume that the incident edges of each vertex are given in a fixed order, in an adjacency list. [1] This defines a unique spanning tree $T_{\mathcal{A}}(S)$ for each connected set S, where $\mathcal{A}$ refers to some spanning-tree algorithm, possibly with specific implementation details insofar as they affect the outcome.

This leads to the following approach for generating all connected sets S: We enumerate all spanning trees $T_{\mathcal{A}}(S)$.

Depending on $\mathcal{A}$, this leads to different algorithms. Details to be figured out.

It turns out that Redelmeier's algorithm corresponds to two algorithms that are obtained in this way, depending on the organization of the untried set U.

Organizing U as a stack corresponds to the algorithm "QUICKSEARCH", and organizing U as a queue corresponds to BFS.

DFS leads to a different algorithm, which has not been investigated.

QUICKSEARCH refers to the variation of BFS where the vertices to be explored are organized as a stack instead of a queue. Knuth [4, Algorithm Q][2] termed this algorithm *Quick digraph search.*[3]

Maybe the BFS variant with a queue has advantages when it comes to shortcutting the final levels of Redelmeier's algorithm? One might want look at some typical configurations (S, U) that are generated with some fixed size $|S|$ (in the plane). (But higher dimensions might be different.)

The DFS algorithm has not been explored at all.

Theorem 1. *For $\mathcal{A} =$ BFS and $\mathcal{A} =$ QUICKSEARCH, when $\mathcal{A}$ is applied to the induced subgraph $G[S]$ with root s_0, it produces the same tree as Redelmeier's algorithm, with the appropriate discipline of organizing U.*

Proof. Algorithm $\mathcal{A}$ maintains a set R of vertices that need to be explored, either as a stack or as a queue.

Consider some connected set $S_0 \subseteq V$ with $|S_0| \leq n$.

We consider two processes:

[1] We may allow the visiting order to depend on the incoming edge, e.g. in a plane graph

[2] see `https://www-cs-faculty.stanford.edu/~knuth/sgb.html`

[3] This is a popular trick question for students of graph algorithms: BFS involves a queue, whereas DFS implicitly involves a stack, through recursion. Therefore, doesn't the substition of a queue by a stack in BFS lead to DFS?

- Following the branches of Redelmeier's algorithm that lead to S_0.
- Algorithm $\mathcal{A}$, applied to the subgraph $G[S_0]$. We have to assume that the adjacency lists in $G[S_0]$ are obtained from those of G by simply omitting the neighbors that are not in S_0, without perverting the order of the remaining elements.

We claim that we can coordinate to two processes so that the following invariants are maintained:

- The list R of Algorithm $\mathcal{A}$ is a sublist of the list U of Redelmeier's algorithm; more precisely, $R = U \cap S_0$. The common elements appear in the same order.
- The constructed spanning subtree of Algorithm $\mathcal{A}$ coincides with the spanning tree T of Redelmeier's algorithm restricted to the vertices of S_0.

Both algorithms remove a vertex from the top of a stack or from the beginning of a queue. The difference is that the respective list (stack or queue) R in Algorithm $\mathcal{A}$ contains only a subset of the vertices of the corresponding list U in Redelmeier's algorithm (namely, the subset $U \cap S_0$).

Suppose Redelmeier's algorithm picks a vertex $u \in U$.

If $u \notin S_0$, Redelmeier's algorithm follows the branch where u is not included in S. The vertex u is simply removed from U, while Algorithm $\mathcal{A}$ does nothing. The invariants are maintained.

If $u \in S_0$, the new neighbors of u are added to U, and edges to the parent node u are added to T. Algorithm $\mathcal{A}$ does the same thing, but only for those neighbors that are in S_0. The elements are inserted in the same order at the same end of the list (top of the stack or end of the queue). The invariants are maintained. □

With the help of this connection between Redelmeier's algorithm and spanning-tree algorithms, we might be able to answer the following question:

> Given S and U, does the configuration (S, U) arise in Redelmeier's algorithm?

Or in a better formulation:

> Given S, in which configuration (S, U) is S generated in Redelmeier's algorithm?

3 Other work

3.1 Komusiewicz and Sorge

Komusiewicz and Sorge [6, Section 3.1] give an algorithm for connected subgraph enumeration that is somewhat different.

It can be cast into the framework of Redelmeier's algorithm in the following way:

1. The sets $X \sqcup U = N$ are not explicitly represented. Instead, the algorithm maintains a subset $W \subseteq S$, and then $X := N(W) \setminus S$, and $U := N \setminus X$. (The set S is called P in [6].)
2. The algorithm selects the vertex v of lowest index in $S \setminus W$, according to some numbering of the vertices, and adds u to W. It will now determine the fate of the vertices in the set $Q := N(v) \setminus (X \cup S)$. (Note that these are not the "new neighbors" of Redelmeier's algorithm, but some subset of U.)
3. The algorithm generates $2^{|Q|}$ branches by distributing Q to X and S in all possible ways. (Formally, this is done by adding to S each of the $2^{|Q|}$ subsets M of Q. The complementary subset $Q \setminus M$ is thereby implicitly added to X.)

References

[1] Khaled Elbassioni. A polynomial delay algorithm for generating connected induced subgraphs of a given cardinality. *Journal of Graph Algorithms and Applications*, 19(1):273–280, Jan. 2015. doi:10.7155/jgaa.00357.

[2] Shant Karakashian, Berthe Y. Choueiry, and Stephen G. Hartke. An algorithm for generating all connected subgraphs with k vertices of a graph. Technical Report UNL-CSE-2013-0005, Department of Computer Science and Engineering, University of Nebraska–Lincoln, 2013. URL: https://consystlab.unl.edu/Documents/StudentReports/TR-UNL-CSE-2013-0005.pdf.

[3] Zahra Razaghi Moghadam Kashani, Hayedeh Ahrabian, Elahe Elahi, Abbas Nowzari-Dalini, Elnaz Saberi Ansari, Sahar Asadi, Shahin Mohammadi, Falk Schreiber, and Ali Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC Bioinformatics*, 10:Article number 318, 2009. doi:10.1186/1471-2105-10-318.

[4] Donald E. Knuth. *Combinatorial Algorithms, Part 4*, volume 4D of *The Art of Computer Programming*. Addison-Wesley, 2027+. In preparation. Draft of Section 7.4.1.2, "Depth-first search" at https://cs.stanford.edu/~knuth/fasc12a.ps.gz, version January 1, 2025.

[5] Christian Komusiewicz and Frank Sommer. Enumerating connected induced subgraphs: Improved delay and experimental comparison. *Discrete Applied Mathematics*, 303:262–282, 2021. Combined Special Issue: 1) 17th Cologne–Twente Workshop on Graphs and Combinatorial Optimization (CTW 2019); Guest edited by Johann Hurink, Bodo Manthey 2) WEPA 2018 (Second Workshop on Enumeration Problems and Applications); Guest edited by Takeaki Uno, Andrea Marino. doi:10.1016/j.dam.2020.04.036.

[6] Christian Komusiewicz and Manuel Sorge. An algorithmic framework for fixed-cardinality optimization in sparse graphs applied to dense subgraph problems. *Discrete Applied Mathematics*, 193:145–161, 2015. doi:10.1016/j.dam.2015.04.029.

[7] D. Hugh Redelmeier. Counting polyominoes: Yet another attack. *Discrete Mathematics*, 36(2):191–203, 1981. doi:10.1016/0012-365X(81)90237-5.

[8] Shanshan Wang, Chenglong Xiao, and Emmanuel Casseau. Algorithms with improved delay for enumerating connected induced subgraphs of a large cardinality. *Information Processing Letters*, 183:106425, 2024. doi:10.1016/j.ipl.2023.106425.

A Redelmeier's algorithm with a queue

```
"""
Redelmeier's algorithm with the untried set organized as a queue.
Prototype implementation.
The queue is implemented as an array.
"""
from collections import defaultdict

nmax = 10
PRINT_SOLUTIONS = False

queuelength = nmax * 3 + 10 # polyomino plus neighbors plus safety buffer
Q = [0] * queuelength
occupied_or_adjacent = defaultdict(bool)
count = defaultdict(int)

for x in range(nmax):
    occupied_or_adjacent[x,-1] = occupied_or_adjacent[-x,0] = True
    # lower border and starting cell

polyomino = defaultdict(str)

def construct(stackbegin, stackend, n):
    """Current polyomino has n cells.
    UNTRIED points are stored in Q[stackbegin] ... Q[stackend-1]."""
    count[n] += 1
```

```
    if PRINT_SOLUTIONS:
        polyomino[0,0]="S" # mark the start position
        print_grid(polyomino, text = f" {n=}, number {count[n]}")
        print()
    if n>=nmax:
        return
    for i in range(stackbegin, stackend):
        #print(f"{n=} {i=} {stackbegin}:{stackend} {Q[stackbegin:stackend]}",)
        #print_grid(occupied_or_adjacent)
        x,y = Q[i]
        # include the cell (x,y):
        polyomino[x,y] = "X" # needed only for printing
        #occupied_or_adjacent[x,y] = "S" # helpful for debugging
        new_neighbors = [nbr for nbr in ((x-1,y),(x,y-1),(x+1,y),(x,y+1))
                         if not occupied_or_adjacent[nbr]]
        for k,nbr in enumerate(new_neighbors):
            occupied_or_adjacent[nbr]=True
            Q[stackend+k] = nbr

        # recursive call:
        construct(i+1, stackend+len(new_neighbors), n+1)

        for nbr in new_neighbors:
            occupied_or_adjacent[nbr]=False # undo the mark; nbr becomes "unseen"
        polyomino[x,y] = " " # needed only for printing
        #occupied_or_adjacent[x,y] = True # reset the debugging marker

def print_grid(GRID, text=""):
    # print the pattern represented in the dictionary GRID
    xmin = min(x for x,y in GRID.keys())
    xmax = max(x for x,y in GRID.keys())
    ymin = min(y for x,y in GRID.keys())
    ymax = max(y for x,y in GRID.keys())
    pattern = [["." for _ in range(1+xmax-xmin)] for _ in range(1+ymax-ymin)]
    for (x,y),letter in GRID.items():
        if letter is False:
            letter = " "
        elif letter is True:
            letter = "X"
        pattern[ymax-y][x-xmin]=letter
    print("\n".join("".join(l) for l in pattern)+text + f",  {xmin}<=x<={xmax}, {ymin}<=y<={ymax}")

Q[0] = (0,0) # The starting square (0,0) is put on the queue
construct(0, 1, 0) # start the enumeration
for i,val in sorted(count.items()):
    print(f"{i:2} {val:6}") # results

```