# Fast Approximate Shortest Hyperpaths for Inferring Pathways in Cell Signaling Hypergraphs

**Spencer Krieger**[1]
Department of Computer Science, The University of Arizona, Tucson, AZ, USA

**John Kececioglu**
Department of Computer Science, The University of Arizona, Tucson, AZ, USA

## Abstract

Cell signaling pathways, which are a series of reactions that start at receptors and end at transcription factors, are basic to systems biology. Properly modeling the reactions in such pathways requires *directed hypergraphs*, where an edge is now directed between two sets of vertices. Inferring a pathway by the most parsimonious series of reactions then corresponds to finding a *shortest hyperpath* in a directed hypergraph, which is NP-complete. The state of the art for shortest hyperpaths in cell-signaling hypergraphs solves a mixed-integer linear program to find an optimal hyperpath that is restricted to be acyclic, and offers no efficiency guarantees.

We present for the first time a heuristic for general shortest hyperpaths that properly handles *cycles*, and is guaranteed to be *efficient*. Its accuracy is demonstrated through exhaustive experiments on all instances from the standard NCI-PID and Reactome pathway databases, which show the heuristic finds a hyperpath that *matches* the state-of-the-art mixed-integer linear program on over 99% of all instances that are acyclic. On instances where only cyclic hyperpaths exist, the heuristic *surpasses* the state-of-the-art, which finds no solution; on every such cyclic instance, enumerating all possible hyperpaths shows that the solution found by the heuristic is in fact *optimal*.

**2012 ACM Subject Classification** Applied computing → Bioinformatics; Applied computing → Systems biology; Theory of computation → Shortest paths; Mathematics of computing → Hypergraphs

**Keywords and phrases** Systems biology, cell signaling networks, reaction pathways, directed hypergraphs, shortest hyperpaths, efficient heuristics, hyperpath enumeration

**Digital Object Identifier** 10.4230/LIPIcs.WABI.2021.20

**Supplementary Material** Source code for the shortest hyperpath heuristic and hyperpath enumeration is available at `http://hyperpaths.cs.arizona.edu`.

**Funding** This research was supported by the US National Science Foundation through Grants CCF-1617192 and IIS-2041613 to John Kececioglu.

**Acknowledgements** We thank T. M. Murali for introducing us to the problem of shortest hyperpaths in cell-signaling hypergraphs, orienting us to the biology literature, and discussing the JUP/DSP example; Anna Ritz for discussing the NCI-PID and Reactome datasets, and providing the BioPax parser; and the anonymous referees for their helpful comments.

## 1 Introduction

Signaling pathways are cornerstones of molecular and cellular biology. They underly cellular communication, govern environmental response, and their perturbation has been implicated in the cause of many diseases. While signaling pathways have classically been modeled as ordinary graphs, using directed or undirected edges to link pairs of interacting molecules [31, 32], both Klamt, Haus and Theis [20] and Ritz, Tegge, Kim, Poirel and Murali [28] have shown that ordinary graphs cannot adequately represent cellular activity that involves the assembly and disassembly of protein complexes, and multiway reactions among such complexes.

[1] Corresponding author.

Directed *hypergraphs* are generalizations of ordinary graphs, where an edge (now called a hyperedge) is directed from one set of vertices (called its tail) to another set of vertices (called its head), and have been used to model many cellular processes [25, 16, 6, 14, 20, 24, 33, 27, 28]. In particular, a biochemical reaction that involves multiple *reactants* – all of which must be present for the reaction to proceed – and that results in multiple *products* – all of which are produced upon its completion – is correctly captured by a single *hyperedge*, directed from its set of reactants to its set of products. Despite hypergraphs affording more faithful models of reaction networks, the lack of practical hypergraph algorithms has hindered their potential for correctly representing and reasoning about molecular reactions.

Biologically, a typical cell-signaling pathway consists of membrane-bound receptors that bind to extracellular ligands, triggering intracellular cascades of reactions, culminating in the activation of transcriptional regulators and factors [2]. Computationally, treating receptors as sources, and transcription factors as targets, finding the most efficient way to synthesize a particular transcription factor from a set of receptors maps to the shortest hyperpath problem we consider here: Given a cell-signaling network whose reactants and reactions are modeled by the vertices and weighted hyperedges of a directed hypergraph, together with a set of sources and a target, find a hyperpath consisting of hyperedges from the sources to the target of minimum total weight. We briefly summarize prior work on related problems next.

## Related work

Hypergraphs have been studied in the algorithms community [18, 12, 4] and applied within systems biology to metabolic networks [8, 1, 3, 5] and cell-signaling networks [27, 26, 11, 30].

In the field of algorithms, Italiano and Nanni [18] first proved that finding a shortest source-sink hyperpath is NP-complete, even when hyperedges have a single head vertex. In a seminal paper that is the source for much of the subsequent work on hypergraphs, Gallo, Longo, Pallottino and Nguyen [12] explore special cases of hypergraphs, and define several versions of hyperpaths, including what they call a $B$-path (though see the correction of Nielsen and Pretolani [22]), which is essentially equivalent to our definition of hyperpath in Section 2. They show the vertices reachable from a source vertex in a hypergraph can be found in time linear in the total size of the tail and head sets of all hyperedges, give an efficient algorithm for a variant of shortest hyperpaths with a so-called additive cost function, and prove that finding a minimum cut in a hypergraph is NP-complete. Ausiello and Laura [4] survey results on hypergraphs whose hyperedges have singleton head sets, and note that a consequence of the NP-completeness reduction [18] for shortest hyperpaths from the set cover problem is that, unless $\mathrm{P} = \mathrm{NP}$, no approximation algorithm can exist for shortest hyperpaths on hypergraphs of $n$ vertices with approximation ratio $\big(1-o(1)\big)\ln n$.

In metabolic networks, Cottret, Milreu and Acuña et al. [8] examine the minimum precursor problem: given a hypergraph $G$, a set of sources $S$, and a set of targets $T$, find a source subset $P \subseteq S$ of minimum cardinality that has a hyperpath from $P$ to $T$. They show this problem is NP-complete, and give an algorithm that enumerates all minimal precursor sets whose hyperpath is acyclic. Acuña, Milreu and Cottret et al. [1] subsequently enumerate all minimal precursor sets allowing cycles. Andrade, Wannagat and Klein et al. [3] extend these algorithms to accommodate stoichiometry and conserve intermediate metabolites within the hyperpath. Carbonell, Fichera, Pandit and Faulon [5] give an efficient algorithm to find a source-sink hyperpath if one exists – irrespective of its length – and prove that finding any hyperpath that must contain a specified set of hyperedges is NP-complete. They also offer an approach to hyperpath enumeration that relies on solutions to this NP-complete problem, for which they employ a heuristic.

In cell-signaling networks, Ritz, Avent and Murali [27, 26] were the first to solve the shortest *acyclic* hyperpath problem by formulating it as a mixed-integer linear program (MILP) – the current state of the art for shortest hyperpaths – and showed that in practice optimal acyclic hyperpaths can be found even for large cell-signaling hypergraphs. Their formulation does not extend to hyperpaths with cycles, and requires exponential time in the worst-case (which may be unavoidable, as the acyclic problem remains NP-complete). Recently, Franzese, Groce, Murali and Ritz [11] defined a parameterized notion of connectivity that interpolates between hyperpath- and ordinary-path-connectivity, while Schwob, Zhan and Dempsey [30] modified the acyclic MILP of Ritz et al. [26] to include time-dependence among reactions.

## Our contributions

In contrast to prior work, we give a heuristic for shortest hyperpaths that handles cycles, is worst-case efficient, and finds hyperpaths that are demonstrably optimal or close to optimal in real cell-signaling hypergraphs. In particular, we make the following contributions.

- We present an *efficient heuristic* for shortest hyperpaths, that on a hypergraph of size $\ell$, which measures the total cardinality of all hyperedge tail and head sets, with $m$ hyperedges that are doubly-reachable from the source and sink vertices, and $k$ defined analogously to $\ell$ over these doubly-reachable hyperedges, runs in $O(\ell + m^2 k)$ time.
- We also give a practical algorithm for *hyperpath enumeration* that generates all possible hyperpaths, allowing us to tractably measure how close our heuristic is to the optimum.
- Our heuristic *matches the state-of-the-art* MILP for shortest acyclic hyperpaths on over 99% of all instances from two standard databases of cell-signaling pathways.
- Our heuristic *surpasses the state-of-the-art* on instances where every source-sink hyperpath is cyclic, and hence the MILP finds no solution. On all such cyclic instances, our hyperpath enumeration algorithm verified that the heuristic was in fact *optimal*.

To our knowledge, this heuristic is the *first in the literature* for shortest source-sink hyperpaths in general directed hypergraphs, where hyperedges have arbitrary tail and head sets, and the length of a hyperpath is the sum of the weights of its hyperedges.

## Plan of the paper

The next section defines the shortest source-sink hyperpath problem. Section 3 then presents our heuristic for shortest hyperpaths, and analyzes its time complexity. Section 4 compares the heuristic, through experiments on all source-sink instances from standard databases, to the state-of-the-art MILP for acyclic instances, or to the optimum of all enumerated hyperpaths for cyclic instances, and discusses biological examples of cyclic shortest hyperpaths. Section 5 concludes, and Appendix A gives our algorithm for generating all hyperpaths.

## 2 Shortest hyperpaths in directed hypergraphs

A directed hypergraph is a generalization of an ordinary directed graph, where an edge, instead of touching two vertices, now connects two subsets of vertices. Formally, a directed *hypergraph* is a pair $(V, E)$, where $V$ is a set of vertices, and $E$ is a set of directed *hyperedges*.[2] Each hyperedge $e \in E$ is an ordered pair $(X, Y)$, where both $X, Y \subseteq V$ are vertex subsets. Edge $e$ is directed from set $X$ to set $Y$. We call set $X$ the *tail* of $e$, and set $Y$ the *head*

[2] The literature sometimes uses the term *hyperarc* for an edge in a directed hypergraph, but we prefer the simpler term *hyperedge* (just as the term *edge* is used for both directed and undirected ordinary graphs).

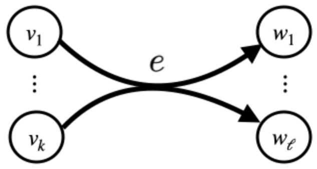

**Figure 1** A hyperedge $e$ with $\text{tail}(e) = \{v_1, \ldots, v_k\}$ and $\text{head}(e) = \{w_1, \ldots, w_\ell\}$. To use $e$ in a hyperpath $P$, every vertex $v_i \in \text{tail}(e)$ must have a preceding hyperedge $f$ on $P$ with $v_i \in \text{head}(f)$.

of $e$, and refer to these sets by the functions $\text{tail}(e) = X$ and $\text{head}(e) = Y$. We also refer to the *in-edges* of vertex $v$ by $\text{in}(v) := \{e \in E : v \in \text{head}(e)\}$, and the *out-edges* of $v$ by $\text{out}(v) := \{e \in E : v \in \text{tail}(e)\}$. Figure 1 shows a directed hyperedge.

In ordinary directed graphs, a path from a vertex $s$ to a vertex $t$ is a sequence of edges starting from $s$ and ending at $t$, where for consecutive edges $e$ and $f$ in the sequence, the preceding edge $e$ must enter the vertex that the following edge $f$ leaves. We say $t$ is reachable from $s$ when there is such a path from $s$ to $t$.

In generalizing these notions to directed hypergraphs, the conditions both for when a hyperedge can follow another in a hyperpath, and when a vertex is reachable from another, become more involved. A hyperpath is again a sequence of hyperedges, but now for hyperedge $f$ in a hyperpath, for every vertex $v \in \text{tail}(f)$, there must be some hyperedge $e$ that precedes $f$ in the hyperpath for which $v \in \text{head}(e)$. Reachability is captured by the following notion of superpath.

▶ **Definition 1** (Superpath). *In a directed hypergraph $(V, E)$, an $s,t$-superpath, for vertices $s, t \in V$, is an edge subset $F \subseteq E$ such that the hyperedges of $F$ can be ordered $e_1, e_2, \ldots, e_k$, where*

**(i)** $\text{tail}(e_1) = \{s\}$,

**(ii)** *for each $1 < i \le k$,*

$$\text{tail}(e_i) \subseteq \{s\} \cup \bigcup_{1 \le j < i} \text{head}(e_j),$$

**(iii)** *and $t \in \text{head}(e_k)$.*

*For an $s,t$-superpath, we call $s$ its source, $t$ its sink, and we say $t$ is reachable from $s$.*

We can now define hyperpaths in terms of superpaths. Recall that a set $S$ is *minimal* with respect to some property $X$ if $S$ satisfies $X$, but no proper subset of $S$ satisfies $X$.

▶ **Definition 2** (Hyperpath). *An $s,t$-hyperpath is a minimal $s,t$-superpath.*

In other words, a hyperpath $P$ is a superpath for which removing any edge $e \in P$ leaves a subset $P - \{e\}$ that is no longer a superpath. Essentially, hyperpaths eliminate unnecessary edges from superpaths. Figures 5 and 6 later show examples of hyperpaths.

We say a hyperpath $P$ contains a *cycle* if, for every ordering $e_1, \ldots, e_k$ of its hyperedges satisfying properties (i)–(iii) in the definition of superpath, $P$ contains some hyperedge $f$ with a vertex in $\text{head}(f)$ that also occurs in $\text{tail}(e)$ for an earlier hyperedge $e$ in the ordering. While in ordinary graphs a minimal $s,t$-path can never contain a cycle, in hypergraphs an $s,t$-hyperpath can in fact contain cycles, as shown in the examples discussed in Section 4.6.

We can now define the shortest hyperpath problem. For an edge weight function $\omega(e)$, we extend $\omega$ to edge subsets $F \subseteq E$ by $\omega(F) := \sum_{e \in F} \omega(e)$.

▶ **Definition 3** (Shortest Hyperpaths). *The Shortest Hyperpaths problem is the following. Given a directed hypergraph $(V, E)$, a positive edge weight function $\omega : E \to \mathcal{R}^+$, source $s \in V$ and sink $t \in V$, find an $s,t$-hyperpath $P \subseteq E$ of minimum total weight $\omega(P)$.*

Note that for positive edge weights, Shortest Hyperpaths is equivalent to finding an *$s,t$-superpath* of minimum total weight.

Shortest Hyperpaths with a single source and sink vertex also captures more general versions of the problem with *multiple sources* and *multiple sinks*, as follows. To find a hyperpath that starts from a set of sources $S \subseteq V$, simply add a new source vertex $s$ to the hypergraph together with a single hyperedge $(\{s\}, S)$ of zero weight, and equivalently find a hyperpath from the single source $s$. To find a hyperpath that reaches *all* vertices in a set of sinks $T \subseteq V$, add a new sink vertex $t$, a zero-weight hyperedge $(T, \{t\})$, and equivalently find a hyperpath to the single sink $t$. To find a hyperpath that reaches *some* vertex in a set of sinks $T \subseteq V$, add new sink vertex $t$, zero-weight hyperedges $(\{v\}, \{t\})$ from all $v \in T$, and again equivalently find a hyperpath to the single sink $t$. Thus versions of shortest hyperpaths with multiple sources and sinks can be reduced to the problem with a single source and sink.

Shortest Hyperpaths is NP-complete [18] (even for acyclic hypergraphs with singleton head sets), so we likely cannot efficiently compute shortest hyperpaths in the worst-case. The next section presents an efficient heuristic for shortest hyperpaths that is highly accurate at finding demonstrably optimal or near-optimal hyperpaths in real cell-signaling hypergraphs.

## 3 An efficient shortest hyperpath heuristic

We now give a fast heuristic for Shortest Hyperpaths that always finds an $s,t$-hyperpath if one exists. While the heuristic is not guaranteed to find a shortest $s,t$-hyperpath, our later experiments on real cell-signaling hypergraphs show it quickly finds a hyperpath that is optimal or remarkably close to optimal on the overwhelming majority of instances from exhaustive experiments over the two standard cell-signaling databases in the literature. We present detailed *pseudocode* for the heuristic at a level that can be directly implemented, as the heuristic is carefully designed and many of its component algorithms are surprisingly tricky to implement correctly. After describing the heuristic, we give a *time analysis* that shows it is always efficient.

The pseudocode accesses a hypergraph $G$ through fields $G$.vertices and $G$.edges. We access the tail-set and head-set of a hyperedge edge $e$ through fields $e$.head and $e$.tail. We access the set of in-edges and out-edges of a vertex $v$ through fields $v$.in and $v$.out. For a list $Q$ that is handled as a queue, operation $Q.\mathrm{Put}(x)$ appends item $x$ to the rear of $Q$, while operation $Q.\mathrm{Get}()$ removes and returns the item at the front of $Q$. For a min-heap $H$, operation $H.\mathrm{Insert}(x, k)$ inserts item $x$ with key $k$ into $H$, and returns a reference $p$ to the heap node containing this pair $(x, k)$ in $H$; operation $H.\mathrm{Extract}()$ removes and returns the item in $H$ with minimum key; and operation $H.\mathrm{Decrease}(p, k)$ takes a reference $p$ to a heap node in $H$ and decreases its key to $k$ if $k$ is smaller than its current key. All functions assume hypergraph $G$ is passed by reference.

The heuristic builds on fast algorithms for computing reachability in a hypergraph, and we discuss these algorithms first. In general, vertex $v$ is *forward reachable* from source $s$ in hypergraph $G$ if there is an $s,v$-hyperpath in $G$. A hyperedge $e$ is forward reachable from $s$ if all vertices $v \in \mathrm{tail}(e)$ are forward reachable from $s$. When $e$ is forward reachable from $s$, an *$s,e$-hyperpath* is a hyperpath from $s$ to tail$(e)$ followed by $e$. A vertex $v$ is *backward traceable* from sink $t$ if $v = t$, or recursively if $v \in \mathrm{tail}(e)$ for an edge $e$ where some $w \in \mathrm{head}(e)$ is backward traceable from $t$. A hyperedge $e$ is backward traceable from $t$ if some $v \in \mathrm{head}(e)$ is backward traceable from $t$.

Figure 2 gives pseudocode for the functions `ForwardReachable` and `BackwardTraceable`. Function `ForwardReachable` returns the set of all hyperedges that are forward reachable from source $s$, while function `BackwardTraceable` returns the set of hyperedges that are backward traceable from sink $t$. Function `ForwardReachable` uses Boolean vertex field $v$.reached, and integer edge field $e$.count, which it assumes have already been initialized to the values $v.\text{reached} = \mathsf{false}$ for all $v \in V$ and $e.\text{count} = \big|\text{tail}(e)\big|$ for all $e \in E$. Function `BackwardTraceable` also uses Boolean edge field $e$.marked, which it similarly assumes is initialized to false for all $e$. (This initialization will be done once for hypergraph $G$ in the shortest hyperpath heuristic, which allows these functions when called repeatedly to run in time bounded by just the size of the forward-reachable or backward-traceable subgraphs.) Function `ForwardReachable` uses field $e$.count to detect when all vertices in tail($e$) have been reached from $s$, and hence $e$ is now reached from $s$. Function `BackwardTraceable` performs a similar but simpler computation in reverse from sink $t$. The worst-case time for both these functions is linear in the size of the subgraph they explore, as analyzed in Section 3.1.

Figure 3 gives pseudocode for function `ShortestHyperpathHeuristic`, our heuristic. Like Dijkstra's algorithm for shortest paths in an ordinary graph (see [7, pp. 658–663]), this function maintains a heap $H$, but in contrast to Dijkstra's algorithm this is now a heap of hyperedges $e$ (rather than a heap of vertices), which are prioritized by keys that are the best known estimate of the length of a shortest $s, e$-hyperpath. We refer to this estimate as the current *path length* for $e$. The heuristic starts from the out-edges of source $s$, and in a reaching computation repeatedly extracts from heap $H$ the hyperedge $e$ with minimum key. When hyperedge $e$ is removed from $H$, the estimated path length for $e$ is evaluated, and stored in field $e$.length. To compute this length estimate, it must construct the best $s, e$-hyperpath it can find, and evaluate its total weight. Of course, computing an optimal $s, e$-hyperpath is NP-complete, so it uses a greedy heuristic to construct this path by function `RecoverShortPath`. This path-construction heuristic consists of two steps: (1) recovering an $s, e$-superpath by recursively backward-traversing hyperedges that enter tail($e$), and (2) finding a minimal subset of this superpath that is an $s, e$-hyperpath while attempting to minimize its weight.

Figure 4 gives pseudocode for function `RecoverShortHyperpath` that implements this greedy path-construction heuristic. For the first step, recovering the $s, e$-superpath $S$ is done by recursively backward-traversing what we call *in-edges*: those hyperedges whose head-sets intersect the tail-set of a given hyperedge. Function `ShortestHyperpathHeuristic` maintains for a hyperedge $e$ the field $e$.inedges, which stores the subset of the in-edges to $e$ whose length field has been determined.

For the second step, function `RecoverShortHyperpath` attempts to find the minimum weight subset of $S$ that is still a superpath by greedily considering hyperedges $f \in S$ for removal in decreasing order of $f$.length. This repeatedly tests whether tail($e$) is still reachable from $s$ on removing $f$ by calling Boolean function `IsReachable`. Pseudocode for `IsReachable` is not given, but it simply implements a version of function `ForwardReachable` that halts and returns `true` as soon as it adds $e$ to the set of hyperedges reachable from $s$, or returns false after collecting the entire reachable set without encountering $e$.

To summarize, the shortest hyperpath heuristic proceeds greedily like Dijkstra's algorithm, but with some important differences: it maintains a heap of hyperedges prioritized by estimated shortest path lengths to tail-sets, records a set of potential in-edges to a given hyperedge used for recovering a hyperpath to the hyperedge, and recovering such a hyperpath now involves another greedy heuristic to find a minimal superpath of small total weight.

```
function ForwardReachable ($s$, $G$) begin        • Find all edges forward-reachable from source $s$ in $G$

    Create queue $Q$                                 • Initialize a queue of reached vertices
    $Q$.Put($s$)
    $s$.reached := true
    ($F$, $R$) := ($\emptyset, \emptyset$)

    while not $Q$.Empty() do begin                   • Process the reached vertices $v$
        $v$ := $Q$.Get()
        $R$ $\cup$:= $\{v\}$

        for $e \in v$.out do begin                   • Detect which out-edges $e$ of $v$ are now reached
            $e$.count $-$:= 1
            if $e$.count $= 0$ then begin            • All vertices in tail($e$) have been reached, so $e$ is reached
                $F$ $\cup$:= $\{e\}$
                for $w \in e$.head do
                    if not $w$.reached then begin
                        $Q$.Put($w$)
                        $w$.reached := true
                    end
            end
        end
    end

    for $v \in R$ do begin                           • Restore fields, and return the reachable hyperedges
        $v$.reached := false
        for $e \in v$.out do $e$.count $+$:= 1
    end
    return $F$
end

function BackwardTraceable ($t$, $G$) begin       • Find all edges backward-traceable from sink $t$ in $G$

    Create queue $Q$                                 • Initialize a queue of reached vertices
    $Q$.Put($t$)
    $t$.reached := true
    ($F$, $B$) := ($\emptyset$, $\emptyset$)

    while not $Q$.Empty() do begin                   • Process the reached vertices $v$
        $v$ := $Q$.Get()
        $B$ $\cup$:= $\{v\}$

        for $e \in v$.in do                          • Collect the traceable hyperedges $e$
            if not $e$.marked then begin
                $F$ $\cup$:= $\{e\}$
                $e$.marked := true
                for $w \in e$.tail do
                    if not $w$.reached then begin
                        $Q$.Put($w$)
                        $w$.reached := true
                    end
            end
    end

    for $v \in B$ do $v$.reached := false             • Restore fields, and return the traceable hyperedges
    for $e \in F$ do $e$.marked := false
    return $F$
end
```

**Figure 2** Reachability computations. Function `ForwardReachable`, given source vertex $s$ in hypergraph $G$, returns all hyperedges $e$ for which tail($e$) is reachable by a hyperpath from $s$. Function `BackwardTraceable`, given sink vertex $t$ in $G$, returns all hyperedges $e$ for which some vertex $v \in$ head($e$) is backward-traceable from $t$. These functions assume fields $v$.reached, $e$.marked, and $e$.count have been initialized to false, false, and $|\text{tail}(e)|$, respectively, for all $v$ and $e$ in $G$.

```
function ShortestHyperpathHeuristic ($s$, $t$, $G$, $\omega$) begin        • Find a short $s,t$-hyperpath in $G$

    for $v \in G$.vertices do                                                • Initialize fields
        ($v$.reached, $v$.removed) := (false, false)
    for $e \in G$.edges do
        ($e$.count, $e$.marked, $e$.node, $e$.inedges) := ($|e$.tail$|$, false, nil, $\emptyset$)

    $D$ := ForwardReachable($s$, $G$)                                        • Restrict $G$ to doubly-reachable edges $D$
            $\cap$ BackwardTraceable($t$, $G$)
    Remove from $G$ all edges not in $D$

    Create min-heap $H$                                                      • Initialize edge heap $H$
    for $e \in s$.out with $e$.tail $= \{s\}$ do
        $e$.node := $H$.Insert($e$, $\omega(e)$)
    $s$.reached := true

    while not $H$.Empty() do begin                      • Process reached hyperedges by their path lengths
        $e$ := $H$.Extract()
        $e$.removed := true

        $P$ := RecoverShortHyperpath($s, e, G$)        • Recover a short hyperpath to $e$, and its path length
        $e$.length := $\omega(P)$

        $F$ := $\emptyset$                     • Collect for $e$ its out-edges $F$, and detect which are now reached
        for $v \in e$.head do begin
            for $f \in v$.out do begin
                if not $v$.reached and not $f$.marked then
                    $f$.count $-:=$ 1
                if not $f$.marked then begin
                    $F \cup:= \{f\}$
                    $f$.marked := true
                end
            end
            $v$.reached := true
        end
        for $f \in F$ do
            $f$.marked := false

        for $f \in F$ do begin                 • Update path lengths, in-edges, and add reached edges to $H$
            $f$.inedges $\cup:= \{e\}$
            if $f$.node $\neq$ nil and not $f$.removed then              • Update path length to an edge on $H$
                $H$.Decrease($f$.node, $\omega$(RecoverShortHyperpath($s$, $f$, $G$)))
            else if $f$.node $=$ nil and $f$.count $= 0$ then                    • Add reached edge to $H$
                $f$.node := $H$.Insert($f$, $\omega$(RecoverShortHyperpath($s$, $f$, $G$)))
        end
    end

    ($P^*$, $L^*$) := ($\emptyset, \infty$)                                 • Recover the best $s,t$-hyperpath $P^*$
    for $e \in t$.in do
        if $e$.node $\neq$ nil then begin
            $P$ := RecoverShortHyperpath($s$, $e$, $G$)
            if $\omega(P) < L^*$ then
                ($P^*$, $L^*$) := ($P$, $\omega(P)$)
        end

    Restore to $G$ all edges not in $D$                  • Unrestrict $G$, and return the best hyperpath
    return $P^*$
end
```

**Figure 3** Efficient heuristic for shortest source-sink hyperpaths. Given source $s$, sink $t$, and edge weights $\omega$, function `ShortestHyperpathHeuristic` finds an $s,t$-hyperpath in hypergraph $G$, attempting to minimize its length. If no $s,t$-hyperpath exists, the empty path is returned. For doubly-reachable hyperedges $e$, the heuristic maintains fields $e$.length (the total weight of the shortest hyperpath found to $e$), and $e$.inedges (the subset of edges $f$ with head($f$) touching tail($e$) where $f$.length is known), which are used in `RecoverShortHyperpath` to recover a short hyperpath to $e$.

```
function RecoverShortHyperpath ($s$, $e$, $G$) begin        • Recover a short $s, e$-hyperpath in $G$

    Create queue $Q$                                     • Initialize a queue with the in-edges entering $e$
    for $f \in e$.inedges do begin
        $Q$.Put($f$)
        $f$.marked := true
    end

    $S$ := $\{e\}$                                       • Collect $s, e$-superpath $S$, tracing backward from $e$
    while not $Q$.Empty() do begin
        $f$ := $Q$.Get()
        $S$ $\cup$:= $\{f\}$
        for $g \in f$.inedges do
            if not $g$.marked then begin
                $Q$.Put($g$)
                $g$.marked := true
            end
    end
    for $f \in S$ do
        $f$.marked := false

    Remove from $G$ all edges not in $S$                 • Greedily construct an $s, e$-hyperpath $P \subseteq S$
    $S$ $-$:= $\{e\}$
    $P$ := $\{e\}$
    for $f \in S$ in decreasing order of $f$.length do begin
        Remove $f$ from $G$
        if not IsReachable($s$, $e$, $G$) then begin
            Restore $f$ back to $G$
            $P$ $\cup$:= $\{f\}$
        end
    end

    Restore back to $G$ all edges removed                • Restore $G$, and return hyperpath $P$
    return $P$
end
```

**Figure 4** Recovering a short hyperpath from the source to a hyperedge. Given source vertex $s$ and hyperedge $e$, function `RecoverShortHyperpath` returns an $s, e$-hyperpath $P$ in hypergraph $G$, attempting to minimize its length. The edges of hyperpath $P$ are greedily selected from an $s, e$-superpath $S$ that is guaranteed to exist in $G$, where $S$ is recovered by tracing backward from $e$.

We note for this heuristic that the inapproximability of the shortest hyperpath problem [4], together with the polynomial time analysis of the following section, imply that unless P = NP, the heuristic cannot be a constant-factor approximation algorithm for shortest hyperpaths.

## 3.1 Time complexity

We now bound the time complexity of our shortest hyperpath heuristic. To obtain the overall running time of function `ShortestHyperpathHeuristic`, we analyze in turn its component functions `ForwardReachable`, `BackwardTraceable`, and `RecoverShortHyperpath`.

Our analysis is in terms of the following parameters measured on a hypergraph (or an induced subgraph). For a hypergraph $G$ with vertices $V$ and hyperedges $E$, we denote its number of vertices and edges by $n := |V|$ and $m := |E|$. We also use the *size* parameter

$$\ell \;:=\; \sum_{e \,\in\, E} \Big( \big|\mathrm{tail}(e)\big| \;+\; \big|\mathrm{head}(e)\big| \Big) \,,$$

and *degree* parameter

$$d \;:=\; \max_{v \,\in\, V} \Big\{ \, \big|\mathrm{in}(v)\big|, \; \big|\mathrm{out}(v)\big| \, \Big\} \,.$$

We assume all tail and head sets are nonempty, and every vertex is touched by a hyperedge, which implies $m + n = O(\ell)$. When we need to refer to these measures for a particular hypergraph $G$, such as on an induced subgraph, we explicitly subscript the parameters by the specific hypergraph, such as $n_G, \ldots, d_G$, where these parameters are then measured in terms of the vertices and edges of subscripted hypergraph $G$.

For the reachability computations `ForwardReachable` and `BackwardTraceable`, their running time can be expressed in an output-sensitive way, in terms of the size of the edge sets they return (or equivalently the size of the subgraph of $G$ they explore). For `ForwardReachable`, let $R \subseteq V$ be the set of vertices reachable from source $s$, and $F \subseteq E$ be the set of hyperedges reachable from $s$ that are returned. The total time for `ForwardReachable` is dominated by the time for its main while-loop, which takes time $\Theta\big(\sum_{v\in R}|\mathrm{out}(v)| \;+\; \sum_{e\in F}|\mathrm{head}(e)|\big)$, or equivalently, $\Theta\big(\sum_{e\in E}|\mathrm{tail}(e)\cap R| \;+\; \sum_{f\in F}|\mathrm{head}(f)|\big)$. In terms of input hypergraph $G$, this is $O(\ell_G)$. For `BackwardTraceable`, let $B \subseteq V$ be the set of vertices it reaches from sink $t$, and $F \subseteq E$ be the set of hyperedges traceable from $t$ that are returned. A similar analysis shows the time for `BackwardTraceable` is $\Theta\big(\sum_{e\in E}|\mathrm{head}(e)\cap B| \;+\; \sum_{f\in F}|\mathrm{tail}(f)|\big)$, which is again $O(\ell_G)$. So the time for both `ForwardReachable` and `BackwardTraceable` is $O(\ell_G)$, but can be bounded more tightly in terms of the subgraph they actually explore.

For function `RecoverShortHyperpath`, all its computations are performed on $G$ restricted to edge subset $D \subseteq E$, the doubly-reachable edges that are both forward-reachable from $s$ and backward-traceable from $t$. Let us call the doubly-reachable subgraph of $G$ induced by $D$, that `RecoverShortHyperpath` computes over, hypergraph $H$. The time to collect $s, e$-superpath $S$ by tracing back from $e$ is at most $O\big(\sum_{f\in S}\sum_{v\in\mathrm{tail}(f)}|\mathrm{in}(v)|\big)$, which is $O(d_H\,\ell_H)$. The time to greedily construct the $s, e$-hyperpath $P \subseteq S$, in terms of its cardinality $k = |S|$, is at most $O(m_H + k\log k + k\,\ell_H)$, which is $O(k\,\ell_H)$. Thus the total time for `RecoverShortHyperpath` is $O(d_H\,\ell_H) \,+\, O(k\,\ell_H)$, which is $O(\ell_H\,m_H)$.

For function `ShortestHyperpathHeuristic`, we break its time down as follows. The time for the initialization, collecting the doubly-reachable edges $D$ by calling `ForwardReachable` and `BackwardTraceable`, and restricting $G$ to its subgraph $H$ induced by $D$, is $O(\ell_G)$. The main while-loop executes for $m_H$ iterations, and spends $O(m_H\,\log\,m_H)$ time for all Extracts. The total time across all iterations to compute $s, e$-hyperpath $P$ for all extracted edges $e$ by calling `RecoverShortHyperpath` is $O(\ell_H\,m_H^2)$. The total time to collect the out-edges $F$ for extracted $e$ across all iterations is $O\big(\sum_{e\in D}\sum_{v\in\mathrm{head}(e)}|\mathrm{out}(v)|\big) = O(d_H\,\ell_H)$. The total time across all iterations for Decrease and Insert, which take $O(1)$ amortized time per edge in $F$ using a Fibonacci heap (see [7, pp. 510–522]), is also $O(d_H\,\ell_H)$. The time to recover the best $s, t$-hyperpath $P^*$ is $O(d_H\,\ell_H\,m_H)$. Adding up these bounds, the total time for `ShortestHyperpathHeuristic` is $O(\ell_G) + O(m_H\,\log\,m_H) + O(\ell_H\,m_H^2) + O(d_H\,\ell_H) + O(d_H\,\ell_H\,m_H)$, which is $O(\ell_G \,+\, \ell_H\,m_H^2)$.

Notice that the overall running time of the heuristic is dominated by the total time to recover short hyperpaths, which requires invoking `RecoverShortHyperpath` whenever the path length to a hyperedge is updated. This is necessary in hypergraphs, since in contrast to ordinary graphs the length of the hyperpath to a hyperedge can no longer be expressed as a simple function (like a sum or a minimum) of the lengths of the hyperpaths to its in-edges.

To summarize, the *time complexity* of the shortest $s, t$-hyperpath heuristic, in terms of the number of hyperedges $m$ and size parameter $\ell$, measured on both the input hypergraph $G$ and its restriction to the doubly-reachable subgraph $H$ of hyperedges that are simultaneously forward-reachable from source $s$ and backward-traceable from sink $t$, is

$$O\Big(\ell_G \;+\; \ell_H\,m_H^2\Big)\,.$$

As demonstrated in Section 4, for real biological instances the size of the doubly-reachable subgraph $H$ is significantly smaller than the full input hypergraph $G$, so designing the heuristic to compute mainly over $H$ yields a large speedup in practice.

The next section shows this heuristic is remarkably *close to optimal* on real cell-signaling hypergraphs. Given that no practical exact algorithm exists for general shortest hyperpaths, we determine the optimum by enumerating all $s, t$-hyperpaths, and taking the minimum of their lengths. Appendix A gives our algorithm for generating all source-sink hyperpaths.

## 4 Experimental results

We now present results from computational experiments on real pathway databases comparing our heuristic to the optimum, study the prevalence of cyclic instances of shortest hyperpaths, report actual running times, and discuss biological examples of cyclic hyperpaths.

### 4.1 Datasets

We evaluate the quality of our heuristic on four datasets built by combining different annotated signaling pathways from two pathway databases, NCI-PID and Reactome. NCI-PID [29] is a curated human-pathway database containing biochemical reactions for complex assembly, cellular transport, and transcriptional regulation. Reactome [19] also contains curated human signaling pathways, and is actively maintained with new reactions being continuously added. We constructed hypergraphs from three subsets of NCI-PID pathways used in Ritz et al. [28], named the `Small`, `Medium`, and `Large` datasets. The `Small` dataset is a small Wnt signaling pathway consisting of the union of two pathways: “degradation of $\beta$-catenin” and “canonical Wnt signaling”. The `Medium` dataset is a larger Wnt signaling pathway including four additional pathways: “noncanonical Wnt signaling”, “Wnt signaling network”, “regulation of nuclear $\beta$-catenin”, and “presenilin action in Notch and Wnt signaling”, which correspond to non-canonical branches of Wnt signaling. The `Large` dataset contains all NCI-PID pathways. Similarly, the `Reactome` dataset is the union of all Reactome pathways. The NCI-PID and Reactome pathways were downloaded in the BioPAX format [10] from Pathway Commons, and processed using a parser from Franzese et al. [11] built on PaxTools [9].

To construct the hypergraphs for each dataset, we mapped each entity (protein, small molecule, and so on) to a vertex in the hypergraph. We parsed each complex as a unique vertex, different from the entities in the complex. Multiple forms of the same protein map to different vertices denoting compartmentalization and post-translational modifications, such as phosphorylation and ubiquitination. We treated each variant as a distinct entity because many pathways show the transportation of a protein from one cellular compartment to another, or a protein being marked for degradation by ubiquitination, so the corresponding vertices need to be distinct to reflect these variants. Each reaction was mapped to a hyperedge, where the reactants and positive regulators comprise the hyperedge tail, and the head contains the products. All hyperedges were given unit weight, even though the heuristic handles weighted edges, as many NCI-PID reactions have no reaction rate.

Table 1 gives statistics on these four signaling hypergraphs. The hypergraphs are very sparse: there are fewer hyperedges than vertices in all four datasets. The hypergraphs from the `Large` and `Reactome` datasets contain respectively 40 and 433 self-loops, showing that many cyclic hyperpaths are likely to exist. However, a small number of these self-loops are unreachable, due to an otherwise unreachable vertex appearing in both the head and tail of the hyperedge. The sources and targets in the experiments are respectively vertices with no in-edges (or whose only in-edge is an unreachable self-loop), and vertices with no out-edges.

**Table 1** Dataset Summaries.

| | NCI-PID | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Dataset | Small | | Medium | | Large | | Reactome | |
| Vertices | | 56 | | 350 | | 9,009 | | 20,458 |
| Hyperedges | | 36 | | 228 | | 8,456 | | 11,802 |
| Pathways | | 2 | | 6 | | 213 | | 2,516 |
| Sources | | 19 | | 138 | | 3,200 | | 8,296 |
| Targets | | 10 | | 102 | | 2,636 | | 5,066 |
| Self-loops | | 1 | | 8 | | 40 | | 433 |
| Unreachable self-loops | | 1 | | 7 | | 14 | | 32 |
| | mean | max | mean | max | mean | max | mean | max |
| Tail size | 1.8 | 3 | 1.9 | 5 | 1.9 | 10 | 2.4 | 26 |
| Head size | 1.3 | 3 | 1.3 | 4 | 1.1 | 5 | 1.6 | 28 |
| Forward-reachable set | 35 | 35 | 192 | 192 | 6,169 | 6,169 | 4,645 | 4,645 |
| Backward-traceable set | 28 | 28 | 49 | 70 | 1,198 | 2,863 | 4,027 | 7,021 |
| Doubly-reachable set | 27 | 27 | 42 | 60 | 756 | 1,836 | 929 | 1,725 |
| In-degree | 0.8 | 5 | 0.8 | 15 | 1.0 | 323 | 0.9 | 1,056 |
| Out-degree | 1.1 | 4 | 1.2 | 24 | 1.7 | 326 | 1.4 | 1,167 |

On average, hyperedges from all four hypergraphs have small head and tail sets, and the average in- and out-degree of vertices is low, reflecting the sparseness of the hypergraphs.

## 4.2 Experimental setup

To prepare hypergraphs from each dataset for our experiments, we parsed the union of the pathways in the dataset. We then connected a supersource to all source vertices (vertices without in-edges) by a single zero-weight hyperedge. We also connected vertices whose only in-edge was a self-loop to the supersource, as otherwise the hyperedge was not traversable. For each target (a vertex without out-edges), we connected the target to the sink by a zero-weight ordinary directed graph edge, giving us a target instance. Note that the supersource is the same for each instance. These vertices are reasonable choices for sources and targets, as they are the molecules at which biologists stopped annotating a given pathway.

For each target instance, we trimmed the hypergraph to the doubly-reachable set: the set of hyperedges that were both forward-reachable from the source, and backward-traceable from the sink. Table 1 gives the average and maximum size of the forward-reachable, backward-traceable, and doubly-reachable sets over all instances for a given dataset, which dramatically decrease the size of the hypergraph. For each instance, we found a hyperpath from the supersource to the sink using our shortest hyperpath heuristic, and compared its length to the solution of the MILP of Ritz et al. [26] if the heuristic hyperpath was acyclic. For each instance with a cyclic heuristic hyperpath, we exhaustively enumerated all supersource-to-sink hyperpaths, and compared the heuristic hyperpath to the shortest hyperpath. Enumerating all hyperpaths takes many hours in practice, and is not feasible to compute for all target instances.

## 4.3 Abundance of cyclic hyperpaths

Cyclic shortest hyperpaths appear in all four datasets. To take one example, in the `Small` and `Medium` datasets, the shortest – and only – hyperpath from ubiquitinated $\beta$-catenin to APC is cyclic, so for this target instance the acyclic shortest hyperpath MILP returns no

**Table 2** Acyclic Instance Summaries.

| Dataset | NCI-PID Small | NCI-PID Medium | NCI-PID Large | Reactome |
|---|---|---|---|---|
| Target instances | 10 | 102 | 2,636 | 5,066 |
| Reachable instances | 10 | 90 | 2,220 | 2,432 |
| Acyclic instances | 9 | 89 | 2,182 | 2,210 |
| Heuristic optimal | 100% (9) | 100% (89) | 99% (2,159) | 100% (2,210) |

hyperpath. This particular source-target pair is specially selected to create a cyclic shortest hyperpath, as ubiquitinated $\beta$-catenin has an in-edge and APC has an out-edge, so they would not normally be considered sources and targets under our definition. Nevertheless, this demonstrates there are cyclic hyperpaths in the NCI-PID database, even in the union of just two pathways, that are missed by the state of the art when computing only acyclic shortest hyperpaths.

In the `Large` dataset, 38 instances have cyclic heuristic hyperpaths. Of these, 22 were cyclic because of a self-loop, and 16 were cyclic due to a non-trivial cycle. For all these instances, no acyclic hyperpath exists between the supersource and the sink. It is likely that even more cycles exist within the hypergraph from the `Large` dataset, as there were 8 self-loops that were not on any hyperpath found by the heuristic.

In the `Reactome` dataset, we found 22 targets with cyclic shortest hyperpaths, where only one of these targets was cyclic due to a self-loop. Reactome is much sparser than NCI-PID, and 432 of the 433 self-loops in Reactome are never used in a heuristic hyperpath.

The abundance of cyclic hyperpaths in the NCI-PID and Reactome datasets demonstrates the importance of a shortest hyperpath algorithm that properly handles cycles. We discuss two examples of these cyclic shortest hyperpaths later in Section 4.6.

## 4.4 Quality of the hyperpath heuristic

To determine the quality of our hyperpath heuristic, we compared the length of the heuristic hyperpath to the optimal shortest hyperpath. On the target instances with cyclic heuristic hyperpaths, there is no practical exact algorithm for finding the shortest hyperpath. Therefore, to find the optimum we generated all source-sink hyperpaths and kept the shortest one, using the enumeration algorithm presented in Appendix A. On target instances where the heuristic returned an acyclic hyperpath, we compared its length to the hyperpath returned by the MILP for shortest acyclic hyperpaths. It is possible that a shorter cyclic hyperpath exists, but enumerating all hyperpaths is not practical for all instances.

Table 2 summarizes the quality of the heuristic on acyclic instances. On the `Small`, `Medium`, and `Reactome` datasets, the heuristic hyperpath is optimal on all target instances, meaning the heuristic hyperpath and the shortest acyclic hyperpath from the MILP have the same length. On the `Large` dataset, the heuristic is optimal on over 99% of the instances, demonstrating the quality of our heuristic on these biological datasets. On the small fraction of instances where our heuristic was suboptimal, the maximum length difference between the heuristic hyperpath and the MILP hyperpath was 6, while the median difference was 1.

Table 3 summarizes the quality of the heuristic on instances where it returned a cyclic hyperpath. On all these cyclic instances, the acyclic MILP returned no hyperpath, so we could not compare the heuristic to an optimal shortest hyperpath other than by exhaustively enumerating all hyperpaths and picking the shortest. Each cyclic instance from the `Reactome` dataset had many enumerated hyperpaths, with an average of 25.6 hyperpaths enumerated

**Table 3** Cyclic Instance Summaries.

| | NCI-PID | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Dataset | Small | | Medium | | Large | | Reactome | |
| Target instances | | 10 | | 102 | | 2,636 | | 5,066 |
| Reachable instances | | 10 | | 90 | | 2,220 | | 2,432 |
| Cyclic instances | | 1 | | 1 | | 38 | | 22 |
| Heuristic optimal | | 100% | | 100% | | 100% | | 100% |
| Non-trivial cycles | | 1 | | 1 | | 22 | | 21 |
| | mean | max | mean | max | mean | max | mean | max |
| Number of paths* | 1 | 1 | 1 | 1 | 49.8 | 364 | 25.6 | 136 |
| Path length range† | 0 | 0 | 0 | 0 | 9.2 | 43 | 7.3 | 15 |

*Total number of hyperpaths for a cyclic target instance
†Difference between the length of the longest hyperpath and the heuristic hyperpath

per instance and a maximum of 136 hyperpaths. The hyperpaths also tended to have different lengths with a maximum difference between the heuristic hyperpath length and the longest enumerated hyperpath length of 15, with an average of 7.3. The cyclic instances from the `Large` dataset had even more alternate hyperpaths, with an average of 49.8 hyperpaths per instance, and a maximum of 364. These instances also had an even greater range of hyperpath lengths, with an average difference of 9.2, and one instance where the difference between the longest hyperpath and the shortest was 43 hyperedges. This demonstrates that the heuristic is discriminating between hyperpaths of different lengths and choosing the shortest hyperpath over worse hyperpaths, indicating the quality of the heuristic. For all cyclic target instances, the enumerated hyperpaths were cyclic, and many shared the same cycle, with most of their differences occurring in hyperedges outside this cycle.

## 4.5 Implementation and running time

The heuristic is implemented in Python 2.7.3, comprising around 500 lines of code. The parser used to convert the BioPAX format into hypergraphs is from [11]. For directed hypergraph representations and reachability calculations, we used Halp (`github.com/Murali-group/halp/`). All heuristic and enumeration source code is available at `hyperpaths.cs.arizona.edu`.

Experiments were run on a laptop with a 2.9 GHz Intel Core i5 CPU, and 16 GB of RAM. The running time of the hyperpath heuristic was 55 seconds on average for the instances from the `Large` and `Reactome` datasets, which have just under 1,000 hyperedges on average. Memory usage was low, taking less than 2 GB of memory for the heuristic.

## 4.6 Biological examples

We now discuss two instances with cyclic shortest hyperpaths from the `Large` and `Reactome` datasets. The hyperpath found by our heuristic for these two instances is optimal (as was the case for all instances where the heuristic found a cyclic path), and is drawn in Figures 5 and 6. We describe the hypergraph structure and constituent reactions for each instance.

### Assembly of the JUP/DSP complex

The first example captures the assembly of the JUP/DSP complex from the `Large` dataset. Figure 5 shows the shortest hyperpath returned by our heuristic with the JUP/DSP complex as the target. All vertices at the top of the figure are connected to the supersource.

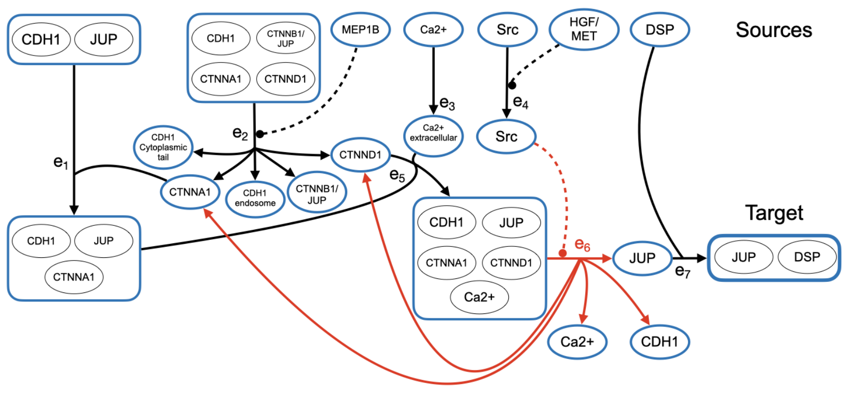

**Figure 5** The cyclic shortest hyperpath from the supersource to the JUP/DSP complex in the `Large` dataset. All vertices in the hyperpath connected to the supersource are shown at the top of the figure. The hyperedges in this hyperpath come from four different pathways, and show the different complexes JUP participates in until finally being free to bind with desmoplakin (DSP). Hyperedge $e_6$, shown in red, creates two separate cycles back to $\alpha$-catenin and $\delta$-catenin.

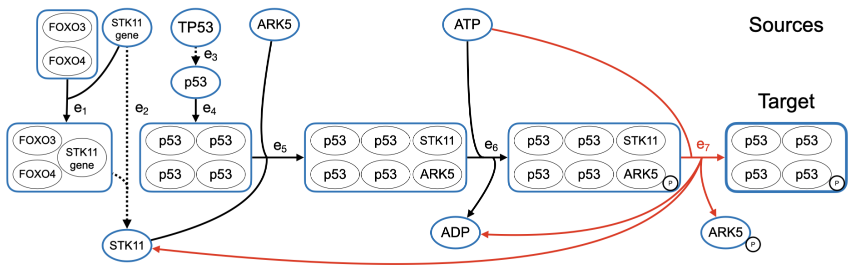

**Figure 6** The cyclic shortest hyperpath from the supersource to phosphorylated p53 in the `Reactome` dataset. All vertices in the hyperpath connected to the supersource are shown at the top of the figure. The hyperedges in this hyperpath show the transcription of STK11 and p53 (TP53) before NUAK1 (ARK5) participates in the phosphorylation of the p53 tetramer. Hyperedge $e_7$, shown in red, creates a cycle when the phosphorylation of p53 breaks up a complex, returning STK11 to its solitary state. Hyperedges $e_2$ and $e_3$ show transcription, and are drawn dotted.

This hyperpath includes seven hyperedges from four different NCI-PID pathways: "E-cadherin signaling in the nascent adherens junction" (hyperedges $e_1$ and $e_5$), "Posttranslational regulation of adherens junction stability and dissassembly" (hyperedges $e_2$, $e_6$ and $e_7$), "Signaling events mediated by PRL" (hyperedge $e_3$), and "Signaling events mediated by hepatocyte growth factor receptor (c-Met)" (hyperedge $e_4$). We briefly describe the key events in this hyperpath. Protein $\gamma$-catenin (also known as junction plakoglobin or JUP) is initially complexed with Cadherin 1 (CDH1) in the tail of hyperedge $e_1$. In hyperedge $e_2$, the metalloprotease meprin$\beta$ cleaves E-cadherin (CDH1), releasing it from its complex with $\alpha$-catenin (CTNNA1) and $\delta$-catenin (CTNND1) [17]. The CDH1/JUP complex adds $\alpha$-catenin (CTNNA1, in hyperedge $e_1$) and CTNND1 and $Ca^{2+}$ (in hyperedge $e_5$) to form a five-member complex. Hepatocyte growth factor (HGF) activates the proto-oncogene tyrosine-protein kinase Src (hyperedge $e_4$) [23]. Src regulates the breakup of this complex into its individual components [21] (hyperedge $e_6$), freeing JUP to bind with DSP and creating the two cycles in this hyperpath via CTNNA1 and CTNNB1. The hyperpath culminates in the formation of a complex between desmoplasmin (DSP) and JUP.

The hypergraph for this instance is large, with 6,168 forward-reachable hyperedges, 2,642 backward-traceable hyperedges, and 1,665 doubly-reachable hyperedges. There is no acyclic hyperpath from the supersource to JUP/DSP. When enumerating all $s, t$-hyperpaths for this instance, there were 16 alternate hyperpaths, and the longest hyperpath had 3 more hyperedges than the heuristic path, which was verified to be optimal.

### Phosphorylation of p53

The second example captures the phosphorylation of p53 by NUAK1 (ARK5) from the `Reactome` dataset. The heuristic hyperpath, which is optimal, is shown in Figure 6. All of the vertices at the top are connected to the supersource.

Hyperedge $e_1$ shows the complex formation of FOXO3 and FOXO4 with the STK11 gene, allowing for the transcription of the gene in hyperedge $e_2$. Hyperedges $e_3$ and $e_4$ deal with the transcription of protein p53 (TP53), and its formation into a homotetramer. The p53 tetramer then forms a complex with NUAK1 (ARK5) and STK11 in hyperedge $e_5$, allowing for the phosphorylation of NUAK1 via ATP in hyperedge $e_6$. Once NUAK1 is phosphorylated, it directly phosphorylates p53 [15], activating it and allowing it to assist in DNA damage repair. The final hyperedge $e_7$, shown in red, breaks apart the p53 tetramer/NUAK1/STK11 complex, resulting in a cycle of free STK11. This hyperpath features two transcriptional hyperedges $e_2$ and $e_3$, shown dotted.

This example from `Reactome` is slightly smaller than the example from the `Large` dataset, with only 4,645 forward-reachable edges, 7,021 backward-traceable edges, and 1,632 hyperedges in the doubly-reachable set. There was no acyclic hyperpath for this instance. In contrast to the first example, no alternate hyperpaths to the target exist in the hypergraph.

## 5 Conclusion

We have presented the first heuristic for Shortest Hyperpaths in general directed hypergraphs with positive edge weights, where the length of a hyperpath is the sum of the weights of its hyperedges. The heuristic handles cycles, is guaranteed to be efficient, and is highly accurate in practice. It matches the state-of-the-art mixed-integer linear program for shortest acyclic hyperpaths on over 99% of all instances from the NCI-PID and Reactome databases, and surpasses the state-of-the-art on all instances where no acyclic hyperpath exists. Moreover, exhaustive enumeration of all source-sink hyperpaths demonstrates that on every cyclic instance from these databases, the heuristic was provably optimal.

### Further research

Given that we can quickly find hyperpaths that are close to optimal on real cell-signaling hypergraphs, several research directions beckon. While the inapproximability of Shortest Hyperpaths [4] rules out a constant-factor approximation unless P = NP, is there an approximation algorithm whose *approximation ratio* on hypergraphs with $n$ vertices matches the theoretical lower bound of $\ln n$? More practically, given that in our experiments our heuristic was suboptimal only on acyclic instances, is there a fast method for *acyclic hyperpaths* that outperforms our heuristic? Since a user would like to know how close to optimal a computed hyperpath is for their particular input graph, is there an efficient heuristic that, as well as giving an upper bound on the optimum through its hyperpath, also outputs a *lower bound* on the length of the shortest hyperpath? Many intriguing research avenues are open.

## References

1 Vicente Acuña, Paulo Vieira Milreu, Ludovic Cottret, Alberto Marchetti-Spaccamela, Leen Stougie, and Marie-France Sagot. Algorithms and complexity of enumerating minimal precursor sets in genome-wide metabolic networks. *Bioinformatics*, 28(19):2474–2483, July 2012.

2 Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell.* Garland Science, New York, 2007.

3 Ricardo Andrade, Martin Wannagat, Cecilia C. Klein, Vicente Acuña, Alberto Marchetti-Spaccamela, Paulo V. Milreu, Leen Stougie, and Marie-France Sagot. Enumeration of minimal stoichiometric precursor sets in metabolic networks. *Algorithms for Molecular Biology*, 11(1):25, 2016.

4 Giorgio Ausiello and Luigi Laura. Directed hypergraphs: introduction and fundamental algorithms—a survey. *Theoretical Computer Science*, 658:293–306, 2017.

5 Pablo Carbonell, Davide Fichera, Shashi B. Pandit, and Jean-Loup Faulon. Enumerating metabolic pathways for the production of heterologous target chemicals in chassis organisms. *BMC Systems Biology*, 6(1):10, 2012.

6 Tobias S. Christensen, Ana P. Oliveira, and Jens Nielsen. Reconstruction and logical modeling of glucose repression signaling pathways in *Saccharomyces cerevisiae*. *BMC Systems Biology*, 3(1):7, 2009.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 3rd edition, 2009.

8 Ludovic Cottret, Paulo Vieira Milreu, Vicente Acuña, Alberto Marchetti-Spaccamela, Fábio Viduani Martinez, Marie-France Sagot, and Leen Stougie. Enumerating precursor sets of target metabolites in a metabolic network. In *Proceedings of the 8th Workshop on Algorithms in Bioinformatics*, pages 233–244, 2008.

9 Emek Demir, Özgün Babur, Igor Rodchenkov, Bülent Arman Aksoy, Ken I Fukuda, Benjamin Gross, Onur Selçuk Sümer, Gary D Bader, and Chris Sander. Using biological pathway data with Paxtools. *PLoS Computational Biology*, 9(9):e1003194, 2013.

10 Emek Demir, Michael P. Cary, and Suzanne Paley et al. The BioPAX community standard for pathway data sharing. *Nature Biotechnology*, 28(9):935–942, 2010.

11 Nicholas Franzese, Adam Groce, T.M. Murali, and Anna Ritz. Hypergraph-based connectivity measures for signaling pathway topologies. *PLoS Computational Biology*, 15(10):1–26, October 2019.

12 Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2–3):177–201, 1993.

13 H.W. Hamacher and M. Queyranne. $K$ best solutions to combinatorial optimization problems. *Annals of Operations Research*, 4:123–143, 1985.

14 Lenwood S. Heath and Allan A. Sioson. Semantics of multimodal network models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(2):271–280, 2009.

15 X Hou, J-E Liu, W Liu, C-Y Liu, Z-Y Liu, and Z-Y Sun. A new role of NUAK1: directly phosphorylating p53 and regulating cell proliferation. *Oncogene*, 30(26):2933–2942, June 2011.

16 Zhenjun Hu, Joe Mellor, Jie Wu, Minoru Kanehisa, Joshua M. Stuart, and Charles DeLisi. Towards zoomable multidimensional maps of the cell. *Nature Biotechnology*, 25(5):547–554, 2007.

17 Maya Huguenin, Elaine J. Müller, Sandra Trachsel-Rösmann, Beatrice Oneda, Daniel Ambort, Erwin E. Sterchi, and Daniel Lottaz. The metalloprotease meprinbeta processes E-cadherin and weakens intercellular adhesion. *PLoS One*, 3(5):e2153, May 2008.

18 Giuseppe F. Italiano and Umberto Nanni. Online maintenance of minimal directed hypergraphs. Technical report, Department of Computer Science, Columbia University, 1989.

19 G. Joshi-Tope, M. Gillespie, I. Vastrik, P. D'Eustachio, E. Schmidt, B. de Bono, B. Jassal, G.R. Gopinath, G.R. Wu, L. Matthews, S. Lewis, E. Birney, and L. Stein. Reactome: a knowledgebase of biological pathways. *Nucleic Acids Research*, 33:D428–432, 2005.

20 Steffen Klamt, Utz-Uwe Haus, and Fabian Theis. Hypergraphs and cellular networks. *PLoS Computational Biology*, 5(5):e1000385, 2009.

21 Susana Miravet, José Piedra, Julio Castaño, Imma Raurell, Clara Francì, Mireia Duñach, and Antonio García de Herreros. Tyrosine phosphorylation of plakoglobin causes contrary effects on its association with desmosomes and adherens junction components and modulates $\beta$-catenin-mediated transcription. *Molecular and Cellular Biology*, 23(20):7391–7402, 2003.

22 Lars Relund Nielsen and Daniele Pretolani. A remark on the definition of a $B$-hyperpath. Technical report, Department of Operations Research, University of Aarhus, 2001.

23 Felipe Palacios, Jogender S. Tushir, Yasuyuki Fujita, and Crislyn D'Souza-Schorey. Lysosomal targeting of E-cadherin: a unique mechanism for the down-regulation of cell-cell adhesion during epithelial to mesenchymal transitions. *Molecular and Cellular Biology*, 25(1):389–402, 2005.

24 Emad Ramadan, Sudhir Perincheri, and David Tuck. A hyper-graph approach for analyzing transcriptional networks in breast cancer. In *Proceedings of the 1st ACM Conference on Bioinformatics and Computational Biology*, pages 556–562, 2010.

25 Emad Ramadan, Arijit Tarafdar, and A. Pothen. A hypergraph model for the yeast protein complex network. In *Proceedings of the 18th Parallel and Distributed Processing Symposium*, pages 189–196, 2004.

26 Anna Ritz, Brendan Avent, and T.M. Murali. Pathway analysis with signaling hypergraphs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 14(5):1042–1055, 2017.

27 Anna Ritz and T.M. Murali. Pathway analysis with signaling hypergraphs. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 249–258, 2014.

28 Anna Ritz, Allison N. Tegge, Hyunju Kim, Christopher L. Poirel, and T.M. Murali. Signaling hypergraphs. *Trends in Biotechnology*, 32(7):356–362, 2014.

29 Carl F. Schaefer, Kira Anthony, Shiva Krupa, Jeffrey Buchoff, Matthew Day, Timo Hannay, and Kenneth H. Buetow. PID: the Pathway Interaction Database. *Nucleic Acids Research*, 37:D674–679, 2009.

30 Michael R. Schwob, Justin Zhan, and Aeren Dempsey. Modeling cell communication with time-dependent signaling hypergraphs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, to appear 2019.

31 Roded Sharan and Trey Ideker. Modeling cellular machinery through biological network comparison. *Nature Biotechnology*, 24(4):427–433, 2006.

32 Marc Vidal, Michael E. Cusick, and Albert-Lásló Barabási. Interactome networks and human disease. *Cell*, 144(6):986–998, 2011.

33 Wanding Zhou and Luay Nakhleh. Properties of metabolic graphs: biological organization or representation artifacts? *BMC Bioinformatics*, 12(1):132, 2011.

## A Appendix: Generating all source-sink hyperpaths

In this appendix, we give our algorithm for tractably generating all the hyperpaths in a hypergraph from a fixed source to a fixed sink.

The technique of inclusion and exclusion of Hamacher and Queyranne [13] provides a general method for generating all the solutions to any combinatorial optimization problem whose feasible solutions are subsets of a ground set – where in our context, hyperpaths are subsets of hyperedges from a hypergraph – but it relies on the ability to efficiently compute a feasible solution that is constrained to include a given *in-set* and exclude a given *out-set*. Interestingly, for hyperpaths, Carbonell et al. [5] have shown that just determining whether an $s,t$-hyperpath exists that contains a specified in-set of hyperedges (regardless of the length of the hyperpath) is already NP-complete. Consequently, we cannot generate all $s,t$-hyperpaths using the standard inclusion-exclusion technique, as we cannot tractably solve the resulting subproblem that has both in- and out-set constraints.

```
function AllHyperpaths ($s$, $t$, $G$) begin                    • Generate all $s,t$-hyperpaths in $G$

    Create queue $Q$                  • Initialize a queue of subproblems, and a set $\mathcal{A}$ of hyperpaths
    $Q$.Put$\big((\emptyset, \emptyset)\big)$
    $\mathcal{A}$ := $\emptyset$

    while not $Q$.Empty() do begin                            • Process all subproblems on the queue
        (Out, Keep) := $Q$.Get()

        $P$ := OneHyperpath($s$, $t$, Out, $G$)      • Find an $s,t$-hyperpath excluding edges in Out
        if $P \neq \emptyset$ and $P \notin \mathcal{A}$ then begin
            $\mathcal{A}$ $\cup$:= $\{P\}$                                        • Save the new hyperpath

            $K$ := Keep                                  • Add all child subproblems to the queue
            for $e \in P$ with $e \notin$ Keep do begin
                $Q$.Put$\big((\mathrm{Out} \cup \{e\}, K)\big)$              • Children cannot exclude edges in Keep ...
                $K$ $\cup$:= $\{e\}$                                • ... or edges excluded by prior siblings
            end
        end
    end

    return $\mathcal{A}$                                        • Return the set $\mathcal{A}$ of all hyperpaths
end
```

**Figure 7** Generating all source-sink hyperpaths. Function `AllHyperpaths`, given source vertex $s$, sink vertex $t$, and hypergraph $G$, returns the set of all $s,t$-hyperpaths in $G$. It calls a function `OneHyperpath` that returns an $s,t$-hyperpath not containing any hyperedge from a specified set Out, and which returns the empty path if no such hyperpath exists.

Instead, we generate all hyperpaths through a simple and practical algorithm that only involves out-sets, given in Figure 7. Function `AllHyperpaths` returns a list of all $s,t$-hyperpaths in hypergraph $G$, leveraging a function `OneHyperpath` that just has to return one $s,t$-hyperpath $P$ in $G$ that does not contain any hyperedges from set Out (so $P \cap \mathrm{Out} = \emptyset$), or determine that no such hyperpath exists. This constrained hyperpath problem with only out-sets is easy to solve: remove all hyperedges in set Out from $G$, collect all vertices $R$ and hyperedges $F$ reachable from $s$ in this reduced hypergraph, and if $t \in R$, then find any minimal subset $P \subseteq F$ in which $t$ is still reachable from $s$; otherwise if $t \notin R$, no such hyperpath exists. Function `OneHyperpath` can efficiently find such an $s,t$-hyperpath $P$ excluding set Out using repeated calls to `ForwardReachable` (given earlier in Figure 2).

Function `AllHyperpaths` uses a queue of subproblems. A subproblem is described by a pair (Out, Keep), which corresponds to finding an $s,t$-hyperpath excluding Out, where any subsequent subproblems that arise from this given subproblem must not exclude any hyperedges from set Keep (though their solutions are not required to actually use edges from Keep). The purpose of this set Keep is to ensure that all subproblems ever placed on the queue have distinct Out sets. (So any given subproblem described by an out-set is only ever solved once.) A subproblem that arises from a given one we call a *child* subproblem (as the entire collection of subproblems conceptually forms a tree that is explored breadth-first using the queue). Each child subproblem excludes one edge from the hyperpath found for its parent subproblem; in this way, the children will generate hyperpaths that are distinct from their parent hyperpath, if they have a solution. (Once a subproblem becomes infeasible due to its out-set eliminating any $s,t$-hyperpath as a solution, it also does not generate further subproblems.) Though the whole approach never repeatedly solves the same subproblem, in contrast to the inclusion-exclusion technique it can generate the same hyperpath from different subproblems, so we check whether hyperpath $P$ is distinct from those already found before adding it to the list $\mathcal{A}$ of all hyperpaths.

We bound the running time for `AllHyperpaths` as follows. Let $m$ be the number of hyperedges in hypergraph $G$, and $\ell$ be the size parameter for $G$ defined in Section 3.1. Solving a given subproblem by `OneHyperpath` involves at most $m$ calls to `ForwardReachable`, which takes $O(\ell\, m)$ time. Suppose `AllHyperpaths` terminates after solving $k$ subproblems from the queue. Its total time is then $O(k\, \ell\, m)$.

In the worst-case, this is $O(2^m\, \ell\, m)$ time, since the out-sets of subproblems are all distinct. In practice, though, typically $k \ll 2^m$, so the running time is much faster than the worst-case suggests. Function `AllHyperpaths` can tractably generate all source-sink hyperpaths for large hypergraphs, as shown in Section 4, since many of its subproblems quickly become infeasible for real cell-signaling networks.