Robust Recoverable Path Using Backup Nodes

Marjan van den Akker[1], Hans L. Bodlaender[1], Thomas C. van Dijk[2(✉)], Han Hoogeveen[1], and Erik van Ommeren[1]

[1] Universiteit Utrecht, Utrecht, The Netherlands
{J.M.vandenAkker,H.L.Bodlaender,J.A.Hoogeveen}@uu.nl
[2] Lehrstuhl Für Informatik I, Universität Würzburg, Würzburg, Germany
thomas.van.dijk@uni-wuerzburg.de

Abstract. We consider routing in networks in the presence of node failures. The focus is specifically on the single-node failure model, which captures the resilience of networks in a realistic fault setting. We introduce a model of recoverable routing, where we ask for an s-t-path that can be repaired easily and locally by assigning 'backup nodes:' when a node on the path fails, it is replaced by its backup node. We resolve the basic algorithmic and complexity questions for finding paths in this model, depending on the properties we require of the backup assignment. For some cases we provide polynomial-time algorithms, and for the others we prove $\mathcal{NP}$-completeness and provide exponential-time algorithms. Lastly, we consider a problem variant where the path is given and ask for a backup assignment.

1 Introduction

We consider a network modeled by a simple graph $G = (V, E)$ with a source node $s \in V$ and a destination node $t \in V$. We write paths using square brackets, like $[p_1, p_2, \ldots, p_k]$. In order to route a packet through the network, we are looking for a path P from s to t. This problem is complicated by considering node failures, for which we want to provide a certain level of robustness. Our robustness condition—that the path must remain valid in the presence of a single node failure—would, by itself, make us overly cautious. We therefore include a recovery model which allows for the easy recovery of a valid path in case a failure makes our initial solution invalid. In contrast to earlier studies of this fault model (e.g. [1,8,10]), we focus on the crucial complexity issues of the model. Our usage of preassigned backup nodes contrasts related work on recoverable paths, which includes subgraph selection [5], network design [4] and cost uncertainty [6] (as opposed to node failure). Particularly related to the current work are the online replacement paths of Adjiashvili et al. [2]; in contrast, our strategy with backups affects the path only locally. We model the network as a graph, disregarding any spatial structure that might be present in the network [3].

In our failure model we will concern ourselves with single-node failures: any single node $v \notin \{s, t\}$ can fail. Asking for a path P that is valid in any failure scenario is then uninteresting. (Only the path $[s, t]$ would be valid, and only

R.M. Freivalds et al. (Eds.): SOFSEM 2016, LNCS 9587, pp. 95–106, 2016.
DOI: 10.1007/978-3-662-49192-8_8

if $\{s,t\} \in E$). We therefore introduce a recovery procedure. In case a failure invalidates our initial path P, we want there to be an easy, local way to repair P. In this way the failure can be dealt with in an online fashion: just travel along P, and if the path turns out to be blocked by a node failure, take a local detour and then resume along on P as originally planned. We do this by preassigning backup nodes. Consider the energy usage implied by a certain path: the nodes involved expend energy to transmit the packet. Our recovery procedure changes the path only locally: the recovered path is mostly equal to the input path, even if the fault occurs early on the path. This means that the recovery procedure does not force unexpected energy expense onto many unrelated nodes.

A *path with backups* R assigns to each *main node* p_i a single *backup node* b_i. If p_i fails, then b_i will take its place. We say "p_i is backed up by b_i" and "b_i backs up p_i." We write a path with backups as $[\frac{p_1}{b_1}, \frac{p_2}{b_2}, \ldots, \frac{p_k}{b_k}]$. By $path(R)$ we denote the path formed by the main nodes of R : $path(\ [\frac{p_1}{b_1}, \frac{p_2}{b_2}, \ldots, \frac{p_k}{b_k}]\ ) = [p_1, p_2, \ldots, p_k]$.

Definition 1 (Recoverable path). *A path with backups* $R = [\frac{p_1}{b_1}, \frac{p_2}{b_2}, \ldots, \frac{p_k}{b_k}]$ *is called* recoverable *if and only if the following two properties are satisfied. First,* $path(R)$ *is a path in* G*. Secondly, the following recovery procedure succeeds for any node* $v \notin \{s,t\}$*. Take* $path(R)$*, but wherever* $p_i = v$*, use* b_i *instead: the resulting path* P *must be a path in* $G - v$*.*

Note that if a failing node v occurs in $path(R)$ more than once, then it may happen that the recovery procedure substitutes a different backup node for each occurrence: at this point, we do not require that $p_i = p_j$ implies $b_i = b_j$. A recoverable path R is called *simple* if and only if $path(R)$ is simple.

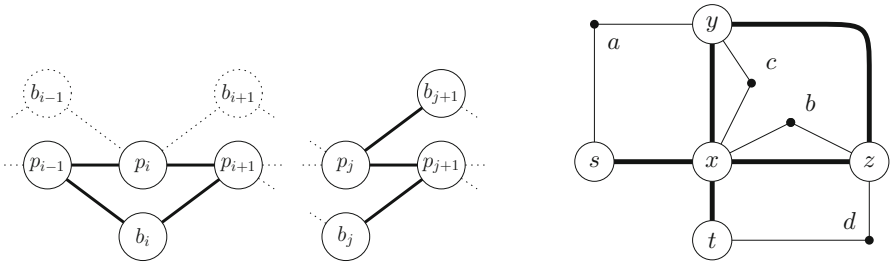
Definition 2 (Recoverable s-t-path). *A recoverable path* $R = [\frac{p_1}{b_1}, \frac{p_2}{b_2}, \ldots, \frac{p_k}{b_k}]$ *is a* recoverable s-t-path *if and only if* $p_1 = b_1 = s$ *and* $p_k = b_k = t$*.*

In this paper we consider the combinatorial problem of finding optimal recoverable s-t-paths. We look at several variations of this problem and give a polynomial-time algorithm or hardness result; the results are summarised at the end of this section. First we make some basic observations. See Fig. 1(a) for an illustration of the edge sets involved in the following two statements.

Lemma 1. $R = [\frac{p_1}{b_1}, \frac{p_2}{b_2}, \ldots, \frac{p_k}{b_k}]$ *is a recoverable* s-t*-path if and only if the following conditions all hold:*

(1) $p_1 = b_1 = s$,
(2) $p_k = b_k = t$,
(3) $p_i \notin \{s,t\} \implies p_i \neq b_i$,
(4) the following edges are in E*, for all* $1 < i < k$: $\{p_{i-1}, p_i\}, \{p_i, p_{i+1}\}, \{p_{i-1}, b_i\}$ *and* $\{b_i, p_{i+1}\}$.

Proof. The first two conditions come from the definition of recoverable s-t-path. The third condition is the observation that a node cannot backup itself (except s and t). Consider the edges mentioned in condition four. The first two edges



(a) Two ways to index the edges involved in a recoverable path; see Lemma 1 and Proposition 1

(b) In this graph $[s, \frac{x}{a}, \frac{y}{b}, \frac{z}{c}, \frac{x}{d}, t]$ is a recoverable s-t-path.

Fig. 1. Structures involved in recoverable paths

correspond to $path(R)$ being a path in G. The latter two edges correspond to a valid recovery path in case p_i fails. The graph G is simple, so for any i we have $p_i \neq p_{i+1}$. As only a single node can fail, no edge is required from b_i to b_{i+1}. If these edges are present in G, then R is a recoverable s-t-path. If R is a recoverable s-t-path, then these edges are present in G. □

Proposition 1. *The edge set in condition 4 of the preceding lemma is equivalently defined as follows. For all* $1 \leq i < k : \{p_i, p_{i+1}\}, \{p_i, b_{i+1}\}$ *and* $\{b_i, p_{i+1}\}$.

Note that a recoverable path has $b_{i-1} \neq p_i$, since equality would imply a self-loop on p_i. We will look at four variations of the RECOVERABLE PATH problem by considering two questions.

- Is a node allowed to back up multiple main nodes, or does $b_i = b_j$ imply $p_i = p_j$? That is, is the relation from main nodes to their backup injective? If we do not enforce this, some nodes may experience high load after recovery.
- Is a node allowed to be backed up by multiple nodes if it occurs as a main node multiple times, or does $p_i = p_j$ imply $b_i = b_j$? That is, is the relation from main nodes to their backup a function? If we enforce this, the recovery procedure is truly local in the sense that it does not need to know where on the path it is in order to perform its rerouting.

Let B be the relation consisting of all pairs (p_i, b_i) occurring on a recoverable s-t-path. By *injective backup*, we mean the property that B is injective; we similarly define *functional backup*. For both or neither property we say *one-to-one* and *many-to-many* respectively. Note that one-to-one backup is both functional and injective.

Lemma 2. *A graph G has a simple recoverable s-t-path if and only if it has a recoverable s-t-path R with functional backup (that is, $p_i = p_j$ implies $b_i = b_j$).*

Proof. A simple s-t-path is trivially functional since $p_i \neq p_j$ for all i and j. In the other direction, let s-t-path R have functional backup. If R is not simple, it can be made simple by shortcuts. Let $i < j$ and $p_i = p_j$. Since the backup

relation is function, we have $b_i = b_j$. Remove steps p_{i+1} through p_j from R: the result is still a recoverable path. Repeat until R is simple. □

It may seem reasonable to restrict our attention only to simple recoverable paths. However, there exist graphs that have a recoverable s-t-path but do not have a *simple* recoverable s-t-path: see Fig. 1(b) for an example. Note that the indicated path does not have functional backup.

Results. We give a polynomial-time algorithm for RECOVERABLE PATH with many-to-many backup, including some weighted versions (Sect. 2). The three other variants are $\mathcal{NP}$-complete. We give exponential-time algorithms for these hard variants (Sect. 3). Finally, we look at a related problem: a normal path is given and we ask whether backups can be assigned to make the path recoverable (Sect. 4). We show $\mathcal{NP}$-completeness for the injective case and give an exponential time algorithm. For the other three cases we provide polynomial-time algorithms.

2 Polynomial-Time Algorithm for Many-to-Many Backup

Here we present a polynomial-time algorithm for RECOVERABLE PATH with many-to-many backups. We also solve some weighted variations. All of these problems are solved in $\mathcal{O}(nm)$, where $n = |V|$ and $m = |E|$. This improves to $\mathcal{O}(d^2 n)$ time on graphs of bounded degree d.

We will find a recoverable s-t-path in G by finding a normal path in a suitably defined auxiliary graph G^A.

Definition 3 (Auxiliary graph G^A). *The auxiliary graph of $G = (V, E)$ is the undirected graph $G^A = (V^A, E^A)$, with $V^A = \{(v, w)_V \mid v, w \in V, v \neq w \vee v \in \{s, t\}\} \cup \{(v, w)_E \mid \{v, w\} \in E\}$, and $E^A = \{\{(v, w)_V, (v, x)_E\} \mid \{w, x\} \in E\} \cup \{\{(v, w)_E, (w, x)_V\} \mid \{v, x\} \in E\}$.*

Lemma 3. *There exists a recoverable s-t-path R in G if and only if there exists a normal $(s, s)_V$-$(t, t)_V$-path P in G^A.*

Proof. Interpret P's nodes alternatingly as (main node, backup node) pairs in G and as edges in E. Lemma 1 shows that this path in G^A is equivalent to a recoverable path in G.

(1) Starting from $(s, s)_V \in V^A$ guarantees $p_1 = b_1 = s$.
(2) Going to $(t, t)_V \in V^A$ guarantees $p_k = b_k = t$.
(3) By construction, $(v, v)_V \notin V^A$ unless $v \in \{s, t\}$. (Recall that G is simple.)
(4) By construction, an edge in E^A exists if and only if the edges required by Proposition 1 exist in E. □

Lemma 4. *The auxiliary graph G^A has $O(n^2)$ nodes, has $O(nm)$ edges, and can be constructed in $O(n^2 + nm)$ time.*

Proof. We build an adjacency matrix of G in $O(n^2)$ time. For each node v and edge $\{w, x\} \in E$, we check if $\{v, x\} \in E$, and if so, add the edge $\{(v, w)_V, (v, x)_E\}$ to G^A. For each node w and edge $\{v, x\} \in E$, we check if $\{v, w\} \in E$, and if so, add the edge $\{(v, w)_E, (w, x)_V\}$ to G^A. We can build these adjacency lists with radix sort.

Theorem 1. Recoverable path *with many-to-many backup can be solved in* $\mathcal{O}(nm)$ *time.*

Proof. Assume $s \neq t$. First remove all isolated nodes from G, so $n \leq 2m$. Build G^A and use depth first search to check if there is a path from $(s, s)_V$ to $(t, t)_V$. Correctness follows from Lemma 3; runtime from Lemma 4. □

In case G has bounded maximum degree, a linear-time algorithm exists.

Definition 4 (Type-2 auxiliary graph $G^{A'}$). *The type-2 auxiliary graph of* $G = (V, E)$ *is the directed graph* $G^{A'} = (V^{A'}, E^{A'})$, *with* $V^{A'} = \bigcup_{\{u,v\} \in E} \{(u, v), (v, u)\}$, *and* $E^{A'} = \{((v, w), (w, x)) \mid \exists y \in V : \{v, y\} \in E \wedge \{y, x\} \in E\}$.

Lemma 5. *There is a recoverable* s-t*-path in* G *if and only if* $s = t$ *or there are* $v, w \in V$ *such that there is a path from* $\{s, v\}$ *to* $\{w, t\}$ *in* $G^{A'}$.

Proof. Again, we check the properties required in Lemma 1; this time, the existence of an arc in $G^{A'}$ corresponds precisely to the existence of the required edges in G. □

Theorem 2. Recoverable path *with many-to-many backup can be solved in* $O(d^2 n)$ *time on graphs with maximum degree* d.

Proof. The type-2 auxiliary graph $G^{A'}$ has $O(dn)$ nodes, and $O(d^2 n)$ arcs. Construct it in $O(d^2 n)$ time. Use depth first search to find a path from (s, v) to (w, t) in $G^{A'}$, for some $v, w \in V$. □

We can extend this auxiliary-graph approach to handle several weighted versions of the problem. In a network context, this can be used to model, for example, delay times or energy costs. Consider a weight function $w : E \to \mathbb{Z}_{\geq 0}$. We first ask for a recoverable s-t-path R such that the weight of $path(R)$ is minimised and call this problem Recoverable shortest path.

Theorem 3. Recoverable shortest path *with many-to-many backup and integer weights can be solved in* $\mathcal{O}(nm)$ *time.*

Proof. We use the auxiliary graph G^A and introduce a weight function $E^A \to \mathbb{Z}_{\geq 0}$. A recoverable path R uses the edge $\{v, w\} \in E$ for $path(R)$ if and only if the corresponding path in G^A uses the node $(v, w)_E$ or $(w, v)_E$. We therefore want to weight the usage of node $(v, w)_E$ by $w(\{v, w\})$; we achieve this by assigning that weight to all of its incoming edges. With integer weights, we can use a standard linear-time algorithm [9] to find the minimum-weight (s, s)-(t, t)-path. Since G^A has $\mathcal{O}(n^2)$ nodes and $\mathcal{O}(nm)$ arcs, this gives an $\mathcal{O}(nm)$ time algorithm for Recoverable shortest path. □

The preceding version of the problem only considers the weight of the path in case nothing goes wrong. If a node failure impacts the path, we are faced with the recovery procedure, which in general will give a path of different weight. To take this into account, we look at the expected length of the recovered path: Expected shortest recoverable path. We will work with a probability distribution over which node fails, if any. The case of no failure is denoted by $\varnothing$. Let $f : \{\varnothing\} \cup V - \{s, t\} \to \mathbb{Q}$ be this probability mass function.

Theorem 4. Expected shortest recoverable path *with many-to-many backup can be solved in* $\mathcal{O}(nm)$ *time.*

Proof. We introduce a weight function $E^A \to \mathbb{Q}_{\geq 0}$. An auxiliary edge in E^A corresponds directly to certain edges E (see Definition 3). We can determine the expected weight these edges contribute when included in a recoverable path. Then by linearity of expectation the shortest path in G^A is the recoverable path in G with the lowest expected length. Runtime is again $\mathcal{O}(nm)$. □

3 Exponential-Time Algorithms

In contrast to the many-to-many backup case, Recoverable path is hard when we require that the backup relation is functional, injective or one-to-one. A proof based on reduction from 3-cnf-sat is omitted for space.

Theorem 5. Recoverable path *with injective backup is* $\mathcal{NP}$*-complete. The same holds for functional backup and one-to-one backup.*

In this section we provide dynamic-programming algorithms that work, with minor modification, for each of the three hard variants of Recoverable path. In the functional and injective cases, it runs in $\mathcal{O}(2^n \cdot n^3)$ time and $\mathcal{O}(2^n \cdot n^2)$ space. In the one-to-one case, it runs in $\mathcal{O}(4^n \cdot n^3)$ time and $\mathcal{O}(4^n \cdot n^2)$ space.

For notational convenience we define the concept of a *friend set*.

Definition 5 (Friend set $F(x, y, z)$). *Let x, y, z be nodes in G. The* friend set *of (x, y, z) is the set of nodes v that can backup node y on a recoverable path where y occurs between x and z. That is* $F(x, y, z) = \{\ v \in V \mid (x, v) \in E \wedge (v, z) \in E\ \wedge v \neq y\ \}$.

Consider functional backup. This means that every occurrence of a node p as main node on a recoverable path must be backed up by the same node. By Lemma 2 we know that, in fact, there exists a recoverable s-t-path with functional backup if and only if there exists a simple recoverable s-t-path, so once we have used a node as main node we can disregard ever using it again.

We solve the problem based on a recurrence relation for a boolean function p_{fun} in the parameters $y, z \in V$ and $S \subseteq V$. The value of $p_{\text{fun}}(S, y, z)$ is the following: does there exist a simple recoverable path with $p_1 = b_1 = s$, ending with main node y followed by z, and using, besides y and z, exactly the nodes in S as main nodes.

As a base case for our recurrence relation, we can observe that $p_{\text{fun}}(\varnothing, y, z)$ is true precisely if $y = s$ and $(y, z) \in E$: the path must start at s and the edge must exist. For non-empty S, we have that $p_{\text{fun}}(S, y, z)$ is true if and only if the edge (y, z) actually exists, and there exists a predecessor x for y such that

1. a valid backup exists for y, and
2. by recursion, there exists a recoverable path ending in x and y that further uses precisely the nodes in $S - \{x\}$ as main nodes.

This gives the following equation, with the base case that $p_{\text{fun}}(\varnothing, y, z)$ is true if and only if $y = s \wedge (y, z) \in E$.

$$p_{\text{fun}}(S, y, z) \;=\; (y, z) \in E \wedge \exists x \in S : \Big(\exists b \in F(x, y, z) \;:\; p_{\text{fun}}(S - \{x\}, x, y)\Big) \quad (1)$$

Note that checking for the existence of $b \in F(x, y, z)$ corresponds to checking for the edges required in condition 4 of Lemma 1.

Theorem 6. Recoverable path *with functional backup can be solved in* $\mathcal{O}(\,2^n \cdot n^3\,)$ *time and* $\mathcal{O}(\,2^n \cdot n^2\,)$ *space.*

Proof. Check, using dynamic programming, whether $p_{\text{fun}}(S, y, t)$ is true for any $S \subseteq V$ and $y \in V$. Because of the recurrence relation of p_{fun}, this is equivalent to the existence of a recoverable s-t-path: exactly the edges required by Lemma 1 are present. The parameter S ensures that the path is simple, which ensures functional backup by Lemma 2.

As for runtime and space, we start out by noting that the dynamic program has $\mathcal{O}(\,2^n \cdot n^2\,)$ states. These can be calculated in $\mathcal{O}(\,n\,)$ time each as follows. Checking $(y, z) \in E$ is a simple test. Then there are existential quantifiers over $x \in S$ and $b \in F(x, y, z)$, both of which might range over $\Theta(\,n\,)$ items. Note however that b is not used in the recurrence. We can therefore precompute $(\exists b \in F(x, y, z))$ for all combinations of nodes $x, y, z \in V$. Then this check runs in constant time by table lookup. □

Theorem 7. Recoverable path *with injective backup can be solved in* $\mathcal{O}(\,2^n \cdot n^3\,)$ *time and* $\mathcal{O}(\,2^n \cdot n^2\,)$ *space.*

Proof (sketch). The approach is similar to the functional case. The polynomial term in the runtime can be kept at n^3 using mutual recursion relations, alternately considering pairs of subsequent path nodes and pairs of a main node and its backup node (as in Definition 3). □

Lastly we consider the one-to-one case. The established machinery unfortunately leads to a runtime of $\Theta^*(\,4^n\,)$: it seems that we need to know two things for every node, namely if it is already used as main node and, independently, if it is already used as a backup node. This is because we allow a node to be both main node and (elsewhere) backup node on the same path; this is simply something the definitions permit. (If we were to disallow nodes being both main node and backup node on the same path, an $\mathcal{O}^*(\,2^n\,)$-time algorithm like the previous ones would be possible.)

Theorem 8. Recoverable path *with one-to-one backup can be solved in* $\mathcal{O}(\,4^n \cdot n^3\,)$ *time and* $\mathcal{O}(\,4^n \cdot n^2\,)$ *space.*

4 Backup Assignment

In this section we take a look at a problem related to finding a recoverable path. This time we are given an s-t-path P in G and the question is: does there exist a *recoverable* s-t-path R such that $path(R) = P$? We call this the Backup assignment problem.

We can again look at four variations based on what kind of backup relation we allow. We show that the injective variant of the problem is $\mathcal{NP}$-complete and give an exponential time algorithm. We give polynomial-time algorithms for the other three variants.

For the analysis of the problem, we again use *friend sets* (compare Definition 5). This time it is convenient to index them differently.

Definition 6 (Friend set $F(P, i)$). *Let $P = [p_1, \ldots, p_k]$ be a path in G and let p_{i-1}, p_i, p_{i+1} be consecutive nodes on P. The* friend set *of index i is the set of nodes v that can backup p_i. That is $F(P, i) = F(p_{i-1}, p_i, p_{i+1}) = \{\ v \in V \mid (p_{i-1}, v) \in E \wedge (v, p_{i+1}) \in E\ \wedge v \neq p_i\ \}$.*

4.1 Polynomial Cases

We now give polynomial-time algorithms for three problem variants. The algorithm for the many-to-many variant is the simplest.

Theorem 9. Backup assignment *with many-to-many backup can be solved in polynomial time.*

Proof. According to Lemma 1, the edges required for a node to be in a friend set are exactly those that are required to be a legal backup. Because the backup relation is allowed to be many-to-many, every node can be considered separately. Therefore, in a solution to Backup assignment with many-to-many backup, any node can be backed up by any node from its friend set and only by those. If the algorithm fails—because some $F(P, i)$ is empty—no valid backup assignment exists. This greedy assignment can clearly be done in polynomial time. □

The functional variant is not much more complicated.

Theorem 10. Backup assignment *with functional backup can be solved in polynomial time.*

Proof. Compared to the many-to-many case, we have the extra condition that every time a node p occurs on P it must be assigned the same backup. Therefore we can assign to it a certain backup node b only if that is valid for every occurrence of p. This leaves only the nodes that are in the intersection of friend sets of all occurrences of p. Among those, the choice can again be made arbitrarily. This too can clearly be done in polynomial time. □

We will solve the one-to-one variant of the problem using bipartite matching. As the name suggests, we will use the matching to assign main nodes to backup nodes. We will now construct a bipartite graph G_m that models the right constraints.

Note that a node may occur on a recoverable path both as a main node and as a backup node. We therefore construct two sets of nodes, which together form the node set of G_m.

- A set of nodes V_P representing the main nodes of the path, with a node for every distinct node occurring on P, except s and t.
- A set of nodes V_b representing potential backup nodes, with a node for every node in $V - \{s, t\}$.

We exclude s and t because in a recoverable s-t-path these are necessarily assigned to backup themselves: by definition $p_1 = b_1 = s$ and $p_k = b_k = t$. Then with one-to-one backup the nodes s and t are fully occupied and can be disregarded.

To obtain the edge set of G_m, we insert an edge between a node $p \in V_P$ and a node $b \in V_b$ if and only if b is a legal backup for p, that is, if and only if b is in the friend set of all occurrences of p. An example of this construction can be seen in Fig. 2.

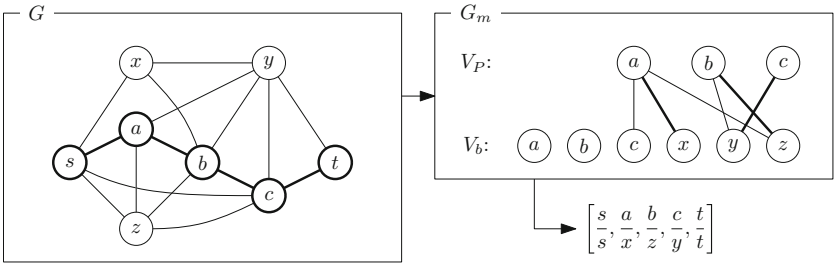


Fig. 2. Example of the matching graph G_m for one-to-one Backup assignment

Theorem 11. Backup assignment *with one-to-one backup can be solved in polynomial time.*

Proof. By construction, the graph $G_m[V_P]$ contains no edges and neither does $G_m[V_b]$. Then a matching in G_m has size at most $|V_P|$ and any edge in any matching involves exactly one node from V_P and one from V_b. We will interpret an edge in the matching as assigning a main node to a backup node.

By construction of the edge set, there exists a one-to-one backup assignment if and only if there exists a matching of size $|V_P|$ in G_m. The algorithm constructs G_m and finds a maximum-cardinality matching. If the matching has size $|V_P|$ then we have a valid backup assignment. If the maximum matching is smaller,

no valid backup assignment exists. The graph G_m can clearly be constructed in polynomial time and the matching can also be found in polynomial time (see for example [7]). □

4.2 Exponential-Time Algorithm

Now we turn to the remaining case of injective backup, which, as we mentioned already at the beginning of this section, is $\mathcal{NP}$-complete. An instance of Backup assignment consists of both a graph G and a prescribed path P. Note that in an $\mathcal{NP}$-completeness proof this path P will, in general, be nonsimple: an injective backup assignment for a simple path is necessarily one-to-one and can, by our preceding results, be found in polynomial time. The following theorem can be proved using reduction from 3-cnf-sat.

Theorem 12. Backup assignment *with injective backup is* $\mathcal{NP}$*-complete.*

Our exponential-time algorithm for Backup assignment with injective backup is based on dynamic programming. Note that since the backup relation is not required to be functional, we need to assign backups for the *occurrences* of nodes on P, not just one backup node for every distinct node on P.

We start off with an observation about the structure of injective backup assignments and some notation.

Lemma 6. *Consider an injective backup assignment and let* $p_i = p_j$ *be distinct occurrences of a single node. Suppose* $b_i \neq b_j$ *and* $b_i \in F(P, j)$*. Then changing* p_j*'s backup from* b_j *to* b_i *results in another valid injective backup assignment.*

Proof. The backup assignment itself is valid: $b_i \in F(P, j)$. There is also no problem with injectivity, since $p_i = p_j$. □

This shows that there is some freedom in injective backup assignments: if multiple nodes are used to back up the various occurrences of a single node v, these can be freely changed within the limitation of the above lemma. This is used to argue the correctness of some arbitrary choices the algorithm makes when picking backup nodes.

Definition 7 (Index set). *The index set* $\mathcal{I}(v)$ *of a node* v *is the set of indices where the node* v *occurs on the path* P*, that is,* $\mathcal{I}(v) = \{i \in \mathbb{N} \mid p_i = v\}$.

Definition 8 (Node multiplicity). *The multiplicity* $\mu(v)$ *of a node* v *is the number of times* v *occurs on the path* P*, that is,* $\mu(v) = |\mathcal{I}(v)|$*. Let* $\mu_{max} = \max\{\ \mu(v) \mid v \in V\ \}$.

Definition 9 ($\prec$, v_{min}, v_{max}, $pred(v)$). *Fix an arbitrary total order* $\prec$ *on* V*. Let* v_{min} *be the minimum node according to* $\prec$*, and* v_{max} *be the maximum. By* $pred(v)$ *we denote the predecessor of* v *according to* $\prec$.

We will now set up a function for use in the dynamic programming algorithm. The nodes of the graph are handled one by one, in some order; for each node v, we consider the occurrences of v in the order that they occur on P.

Definition 10. *The boolean function $a(v, O, B)$ is defined for arguments $v \in V, O \subseteq \mathcal{I}(v), B \subseteq V$. It is defined to be true if and only if there is a way to assign backups that is injective, where exactly the nodes in B are used as backup, and where exactly the following occurrences have been assigned a backup: all $p_i \prec v$, and all p_j for $j \in O$. (Thus leaving all other occurrences unassigned.)*

Then we can solve Backup assignment with injective backup as follows. Check whether $a(\ v_{max},\ \mathcal{I}(v_{max}),\ B\ )$ is true for any subset B: by definition this means assigning backups to all occurrences on P, using any set of backup nodes.

Theorem 13. Backup assignment *with injective backup can be solved in $\mathcal{O}^*(\ 2^{n+\mu_{max}}\ )$ time.*

Proof. Calculate $a(v, O, B)$ for all combinations of $v \in V$, $O \in \mathcal{I}(v)$ and $B \subseteq V$, using the following recurrence relation.

If some occurrence of v is already backed up (that is, $O \neq \varnothing$), we can recurse on which node $b \in B$ is its backup. In view of Lemma 6, we can then immediately use b to backup as many other occurrences of v as possible: since we are already deciding to use b as backup for *some* occurrence of v, it cannot be wrong to use it for more occurrences. There is still the choice of which occurrence in O to recurse on, but this choice can be made arbitrarily. Call this node $arb(O)$. Then

$$a(\ v,\ O,\ B\ )\ =\ \exists b \in\ B \cap F(\ P, arb(O)\ )\ :\ a(\ v,\ O'(b),\ B - \{b\}\ ) \qquad (2)$$

$$\textbf{where } O'(b)\ =\ \{\ i \in O \mid b \notin F(P, i)\ \}.$$

Here, O' is the set of occurrences that cannot be backed up using a particular choice of b.

The preceding case handled $O \neq \varnothing$. The case $O = \varnothing$ is quite simple, since directly from the definition of $a(\cdot)$ we have the following equality (for $v \neq v_{min}$).

$$a(v, \varnothing, B)\ =\ a(\ pred(v),\ \mathcal{I}(pred(v)),\ B\ ) \qquad (3)$$

This leaves setting the base case for our recursion. This is also easily accomplished from the definition. We let $a(v_{min}, \varnothing, \varnothing) =$ true: it is indeed possible to back up no occurrences using no backup nodes.

The algorithm then checks whether $a(v_{max}, \mathcal{I}(v_{max}), B)$ is true for any $B \subseteq V$. Correctness of the algorithm follows from correctness of the recurrence. Dynamic programming ensures that $a(\cdot)$ is only ever evaluated once for every value of the parameters; call these the dynamic programming states. Evaluating a single dynamic programming state can clearly be done in polynomial time. For the runtime up to polynomial factors, it then remains to bound the number of different dynamic programming states. The total number of states is

$$\sum_{v \in V} \left(2^{|\mathcal{I}(v)|} \cdot 2^{|V|}\right) \stackrel{def}{=} \sum_{v \in V} \left(2^{\mu(v)} \cdot 2^n\right) \leq \sum_{v \in V} \left(2^{\mu_{max}} \cdot 2^n\right) = n \cdot 2^{\mu_{max}} \cdot 2^n.$$

This gives total running time of $\mathcal{O}^*(\ 2^{n+\mu_{max}}\ )$. □

5 Conclusion

We have introduced a model of recoverable routing in the single-node failure model. As with other *robust recoverability* models, the motivation is as follows. Choosing a solution that is feasible for any failure scenario is overly cautious—in our case it would only allow paths of one hop. On the other hand, unrestricted replanning in case of a failure can be too costly in terms of computational power or the information available. We therefore plan a route that, in case of failure, can be fixed easily and locally. For this model, we have resolved the basic algorithmic and complexity questions.

We have presented several algorithms. For the polynomially-solvable case of Recoverable path we have given an $\mathcal{O}(\,nm\,)$-time algorithm. For the functional and injective cases, the runtime of $\mathcal{O}^*(\,2^n\,)$ that our algorithms achieve seems reasonable. When generalised to the one-to-one case, however, our algorithm runs in $\Theta^*(\,4^n\,)$ time. It seems to us there should be a better way to handle the one-to-one case.

References

1. Abbasi, A.A., Younis, M.F., Baroudi, U.A.: Recovering from a node failure in wireless sensor-actor networks with minimal topology changes. IEEE Trans. Veh. Technol. **62**(1), 256–271 (2013)
2. Adjiashvili, D., Oriolo, G., Senatore, M.: The online replacement path problem. In: Bodlaender, H.L., Italiano, G.F. (eds.) ESA 2013. LNCS, vol. 8125, pp. 1–12. Springer, Heidelberg (2013)
3. Álvarez-Miranda, E., Candia-Véjar, A., Carrizosa, E., Pérez-Galarce, F.: Vulnerability assessment of spatial networks: models and solutions. In: Fouilhoux, P., Gouveia, L.E.N., Mahjoub, A.R., Paschos, V.T. (eds.) ISCO 2014. LNCS, vol. 8596, pp. 433–444. Springer, Heidelberg (2014)
4. Álvarez-Miranda, E., Ljubić, I., Raghavan, S., Toth, P.: The recoverable robust two-level network design problem. INFORMS J. Comput. **27**(1), 1–19 (2015)
5. Büsing, C.: The exact subgraph recoverable robust shortest path problem. In: Ahuja, R.K., Möhring, R.H., Zaroliagis, C.D. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 231–248. Springer, Heidelberg (2009)
6. Büsing, C.: Recoverable robust shortest path problems. Networks **59**(1), 181–189 (2012)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
8. Nanda, A., Rath, A.K., Rout, S.K.: Node sensing & dynamic discovering routes for wireless sensor networks. Computing Research Repository (CoRR), abs/1004.1678 (2010)
9. Thorup, M.: Undirected single-source shortest paths with positive integer weights in linear time. J. ACM **46**(3), 362–394 (1999)
10. Wang, Y.-H., Chao, C.-F.: Dynamic backup routes routing protocol for mobile ad hoc networks. Inf. Sci. **176**(2), 161–185 (2006)