Ad Hoc Networks 23 (2014) 145–162

Contents lists available at ScienceDirect

Ad Hoc Networks

journal homepage: www.elsevier.com/locate/adhoc

A fault-tolerant relay placement algorithm for ensuring k vertex-disjoint shortest paths in wireless sensor networks

Lanny Sitanayah [a,*], Kenneth N. Brown [b], Cormac J. Sreenan [a]

[a] Mobile & Internet Systems Laboratory, School of Computer Science and IT, University College Cork, Ireland
[b] Insight Centre for Data Analytics, School of Computer Science and IT, University College Cork, Ireland

ARTICLE INFO

Article history:
Received 10 September 2013
Received in revised form 5 June 2014
Accepted 16 July 2014
Available online 24 July 2014

Keywords:
Wireless sensor networks
Network deployment planning
Relay placement
Vertex-disjoint paths

ABSTRACT

Wireless sensor networks (WSNs) are prone to failures. To be robust to failures, the network topology should provide alternative routes to the sinks so when failures occur the routing protocol can still offer reliable delivery. Our contribution is a solution that enables fault-tolerant WSN deployment planning by judicious use of a minimum number of additional relays. A WSN is robust if at least one route with an acceptable length to a sink is available for each sensor node after the failure of any $k-1$ nodes. In this paper, we define the problem for increasing WSN reliability by deploying a number of additional relays to ensure that each sensor node in the initial design has k length-bounded vertex-disjoint shortest paths to the sinks. To identify the maximum k such that each node has k vertex-disjoint shortest paths, we propose Counting-Paths and its dynamic programming variant. Then, we introduce GRASP-ARP, a centralised offline algorithm that uses Counting-Paths to minimise the number of deployed relays. Empirically, it deploys 35% fewer relays with reasonable runtime compared to the closest approach. Using network simulation, we show that GRASP-ARP's designs offer a substantial improvement over the original topologies, maintaining connectivity for twice as many surviving nodes after 10% of the original nodes have failed.

1. Introduction

In wireless sensor networks (WSNs), sensor nodes transmit their data wirelessly over a multi-hop network to sink nodes, where data is either processed or transmitted on through a high-speed connection. These networks are subject to failures: the wireless devices are often unreliable, they have limited battery life, transmissions may be blocked by changes in the environment, and the devices may be damaged, e.g. by weather, wildlife or human intervention. For dealing with failures, techniques for reliable routing in WSNs such as [1–4] have been proposed and are well-understood, but to be effective these depend on a physical network topology that ensures alternative routes to the sink are in fact available. This requires sensor network deployment to be planned with an objective of ensuring some measure of robustness in the topology, so that when failures occur routing protocols can continue to offer reliable delivery. Our contribution is a solution that enables fault-tolerant WSN deployment planning by judicious use of additional relay nodes.

One key objective in the topology design of a WSN is to ensure some measure of robustness. In particular, one standard criterion is to ensure routes to the sink are available for all remaining sensor nodes after the failure of up to $k-1$ nodes. This can be achieved by ensuring that every node in the initial design has k vertex-disjoint shortest paths to the sinks: i.e. at least k paths that share no

* Corresponding author.
E-mail addresses: ls3@cs.ucc.ie (L. Sitanayah), k.brown@cs.ucc.ie (K.N. Brown), cjs@cs.ucc.ie (C.J. Sreenan).

http://dx.doi.org/10.1016/j.adhoc.2014.07.003
1570-8705/

intermediate nodes. Vertex-disjoint paths are also required to provide multi-path routing capability for some protocols [5]. Since WSNs often have data latency requirements, there may be a limit to the path length from sensor to sink. To ensure that sensors have sufficient paths of the right length, it may be necessary to add a number of additional relays, which do not sense, but only forward data from other nodes. Installing additional relay nodes comes at a cost that includes not just the hardware purchase but more significantly the installation and ongoing maintenance, thus motivating solutions that minimise the number of additional relays.

The novel problem we address in this paper is that of finding a minimal set of relays which ensures k length-bounded vertex-disjoint shortest paths to a sink for each sensor node. We define the single-tiered, constrained partial fault-tolerant relay placement problem for k vertex-disjoint shortest paths with length constraints for WSNs with data sinks, which we call the additional relay placement (ARP) problem. We present two centralised algorithms to be run during the initial topology planning, i.e. prior to network deployment and operation, to select the fewest locations to deploy relay nodes to improve the reliability of the network:

1. *Counting-Paths* is a heuristic algorithm that identifies for each node x, the maximum k such that x has k vertex-disjoint shortest paths. Counting-Paths looks for k vertex-disjoint shortest paths, where the sum of the lengths is minimal, and tries to minimise the spread between the lengths. It is formulated as minimise $\sum_{i=1}^{k} l_i + (l_{\max} - l_{\min})$. k denotes the number of disjoint paths, l is the length of a path, $l_{\min}$ is the shortest length, and $l_{\max}$ is the longest length of the disjoint paths. We also propose its dynamic programming variant.
2. *Greedy randomised adaptive search procedure for additional relay placement* (GRASP-ARP) is a local search algorithm based on GRASP [6] that uses Counting-Paths to minimise the number of relays that need to be deployed.

We assume that we are given a pre-planned WSN with a connected finite set of sensors and one or more sinks. We make no assumptions on the geographical or physical properties of the area in which the WSN is to be deployed, but we assume a limited set of possible locations for relays, and a connectivity graph, showing the set of feasible links between all nodes.

The rest of the paper is organised as follows. We define the problem in Section 2, survey the related work in Section 3, present the proposed Counting-Paths algorithm in Section 4 and its evaluation in Section 5. We show that Counting-Paths runs faster than the closest comparator approaches and is able to identify the maximum k. In addition, its dynamic programming variant improves on the runtime. We detail GRASP-ARP in Section 6. We demonstrate empirically in Section 7 that it finds solutions requiring 35% fewer additional relay nodes for small values of k compared to the closest approach from the literature. We also show that GRASP-ARP scales better, finding solutions in reasonable time for problems with hundreds of nodes. We then evaluate the resulting topologies for robustness in Section 8, by simulating network operation while nodes are failing. We show that after only a small number of failures, the new topologies are significantly more robust than the original topologies without relays, maintaining connectivity for up to twice as many sensor nodes. Finally, Section 9 concludes the paper. Parts of this work were first presented in [7].

2. Background and problem statement

A WSN can be modelled as a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. Each edge connects two vertices that are within transmission range of each other,[1] and the two vertices are said to be *adjacent*. A *path* of length t between two vertices v and w is a sequence of vertices $v = v_0, v_1, \ldots, v_t = w$, such that v_i and v_{i+1} are adjacent for each i. A path from a vertex v to a set of vertices W is simply a path from v to any vertex $w \in W$. Two paths P and Q from v to w are *vertex-disjoint* if they have no nodes in common except for v and w. Two vertices are *connected* if there is a path between them. A graph is connected if every pair of vertices is connected. A *cutset* is a set $C \subset V$ such that $(V - C, E\downarrow_{V-C})$ is disconnected (where $E\downarrow_X$ means a set of edges restricted to those connecting only vertices in X). A graph is *k-connected* if it has no cutset of size less than k. If a graph is k-connected, every vertex has k vertex-disjoint paths to any other vertex.

The k length-bounded vertex-disjoint shortest paths problem is described as follows: given a graph $G = (V, E)$ and a pair $(s, t) \in V, s \neq t$, find k vertex-disjoint shortest paths connecting the pair (s, t) with bounded-length l, if they exist.

3. Related work

The problem of placing relay nodes for increased reliability has long been acknowledged as a significant problem, and we summarise the existing algorithms for WSNs in Table 1. A *single-tiered* network has a flat architecture, where all nodes can forward packets from other nodes. A *two-tiered* network is a clustered network, where sensor nodes transmit their own data directly to a cluster head. In *connectivity* problems, the aim is to ensure the network is connected (or $k = 1$), while in *survivable* problems, the aim is to ensure k-connectivity for $k > 1$. In *unconstrained* problems, relay nodes can be placed anywhere, while in *constrained* problems, they are limited to a set of candidate locations. *Partial fault-tolerance* requires k-connectivity only between every sensor node, while *full fault-tolerance* requires k-connectivity between both sensor and relay nodes. We choose to focus on single-tiered networks as this is most common in the research literature and for published WSN deployments. Furthermore we assume the constrained approach for possible relay locations, which we believe is more reasonable for real-world deployments.

[1] For simplicity we assume bi-directional links, but this could be easily relaxed by specifying a more complex connectivity graph.

Table 1
Relay placement algorithms.

Algorithms	k	Routing	Deployment locations	Fault-tolerance
Bredin et al. [8]	$\geqslant 1$	1-Tiered	Unconstrained	Full
Pu et al. [9]	$\geqslant 1$	1-Tiered	Unconstrained	Partial
Han et al. [10]	$\geqslant 1$	1-Tiered	Unconstrained	Full/partial
Ahlberg et al. [11]	$\geqslant 1$	1-Tiered	Unconstrained	Partial
Zhang et al. [12]	2	1/2-Tiered	Unconstrained	Partial
Misra et al. [13]	1, 2	1-Tiered	Constrained	Partial
Hao et al. [14]	2	2-Tiered	Constrained	Partial
Tang et al. [15]	1, 2	2-Tiered	Unconstrained	Partial
Liu et al. [16]	1, 2	2-Tiered	Unconstrained	Full
Kashyap et al. [17]	$\geqslant 2$	2-Tiered	Un/constrained	Partial
This paper	$\geqslant 2$	1-Tiered	Constrained	Partial

Finally we assume partial fault-tolerance, reflecting the fact that relays are deployed for connectivity only and do not have a sensing role.

Bredin et al. [8] develop k-connectivity-repair by finding a minimum-weight vertex k-connected subgraph from a weighted complete graph, adding edges in increasing weight and for each edge, deploying k relays every transmission range distance and $k-1$ relays at endpoints of the edge. Partial k-connectivity-repair by Pu et al. [9] is similar to k-connectivity-repair [8], but only places one relay every transmission range distance and none at endpoints. Connectivity-first [10] finds a minimum k-connected spanning graph from a weighted complete graph by adding edges that have the highest contribution to connectivity and the least weight. Redundant router placement [11] uses the Ford–Fulkerson method to count the number of paths from sensor to sink. If the number of available paths is not sufficient, the algorithm adds paths by placing relays on straight lines between sensors and the sink, starting with the furthest sensor from the sink. Zhang et al. [12] study the single-tiered and two-tiered fault-tolerant relay placement problem. The single-tiered one constructs a complete graph, finds a 2-connected spanning subgraph and steinerises the edges. The steinerisation process calculates edges' weight by dividing the Euclidean distance of any two vertices by the relay's transmission radius. For each edge, a number of relays is deployed along the straight line. The two-tiered approach finds the fewest relays as cluster heads, connects them using the Steiner minimum tree and duplicates each relay found. Misra et al. [13] propose connected and survivable relay node placement. Both connected and survivable assign a weight to each edge equal to the number of candidate relays the edge is incident with. Connected relay node placement computes a low weight connected subgraph, while survivable relay node placement computes a low weight 2-connected subgraph. Relays are then deployed at the candidate locations that appear in the subgraph.

2-connected relay node double cover [14] selects a relay that can cover as many sensors, which are not covered by two relays, as possible. Then, it selects some relays that ensure two disjoint paths for the previously selected relay. Connected relay node single cover and 2-connected relay node double cover [15] divide the region into cells, find possible positions to deploy relays, find a solution to cover $(k=1)$ or double cover $(k=2)$ sensors in each cell using exhaustive search, then add extra relays if needed. Liu et al. [16] develop minimum relay-node placement for 1 and 2-connectivity. Minimum relay-node placement for 1-connectivity finds the fewest relays that can cover all sensors and connects them using the Steiner minimum tree, while minimum relay-node placement for 2-connectivity adds three relays in the transmission range's circle of each relay found in minimum relay-node placement for 1-connectivity. Kashyap et al. [17] propose k-vertex connectivity, where from a complete graph of cluster heads, it assigns edge weights, finds a minimum cost vertex k-connected spanning subgraph, and deploys relays along the subgraph's edges.

All of the algorithms cited find k-connectivity for WSNs without sinks. Unlike our algorithm, the published algorithms do not place any constraints on the path lengths. The closest problem definition to ours is Misra et al.'s [13], but it can only establish up to 2-connected networks. Other works with similar objectives to ours include Bredin et al. [8], Pu et al. [9], Han et al. [10] and Ahlberg et al. [11], although they are for unconstrained deployment locations. Bredin et al.'s [8] considers full fault-tolerant relay node placement. This was then modified by Pu et al. [9] for partial fault-tolerance after noting that there is no need to ensure multiple paths for the relays. The simulation results in Han et al.'s work [10] show that Bredin et al.'s [8] is more efficient for partial fault-tolerance, while Han et al.'s [10] is more efficient for full fault-tolerance in terms of the number of additional relays. Therefore, among all existing algorithms, we infer that the most relevant one to our work is Pu et al.'s [9], except that it assumes unconstrained relay locations and has no limit on path length. The unconstrained problem can be easily converted into the constrained problem by introducing a set of candidate locations.

Offline algorithms to discover k vertex-disjoint shortest paths in existing networks are well studied in the literature. Torrieri [5] calculates a set of vertex-disjoint paths in polynomial time. Bhandari [18] proposes k runs of a modified Dijkstra algorithm, each of which requires $O(|V|^2)$ time, to find k vertex-disjoint shortest paths between a source and a sink. There is a close relation between k-connectivity and maximum flow problems. Maximum flow algorithms, such as Ford–Fulkerson [19], are used to find edge-disjoint paths [20], and thus need to be extended with a vertex-splitting technique as used by Bhandari [18]. Edge-disjoint paths share no edges, but are less robust than vertex-disjoint paths, and so we restrict our discussion to the latter. Since we are only interested in vertex-disjoint shortest paths, we will use the term disjoint paths throughout this paper, unless we need to differentiate from edge-disjoint paths.

The greedy randomized adaptive search procedure (GRASP) [21,6,22] is a metaheuristic which captures good features of pure greedy algorithms and random construction procedures. It is an iterative process. In each iteration, it consists of two phases: the construction phase and the local search phase. The construction phase builds a feasible solution as a good starting solution for the local search phase. The probabilistic component of a GRASP is

characterised by randomly choosing one of the best possible candidates, instead of the overall best one. Since the solution produced by the construction phase is not necessarily the local optimum, the local search phase is utilised to improve it. A local search algorithm works in an iterative fashion by replacing the current solution by a better one from the neighbourhood of the current solution. It terminates when no better solution is found. Martins et al. [23] use GRASP to solve the Steiner tree problem in graphs (SPG). SPG is similar to the relay placement problem, in that it must select from a set of candidate nodes in order to connect a number of designated terminals, although its aim is to find a minimal spanning tree rather than a forest with vertex-disjoint paths.

4. Counting-Paths

We first describe Counting-Paths, which is a variant of Bhandari's algorithm [18] with boosting over split vertices to spread the path lengths. Since Counting-Paths is heuristic, it does not always guarantee finding the smallest sum of lengths. Counting-Paths utilises the Ford–Fulkerson [19] maximum flow algorithm to find the disjoint paths. In each of its iterations, Counting-Paths finds the shortest path from a source node to a sink using the breadth first search technique. Without graph modification, Ford–Fulkerson can only discover edge-disjoint paths [20] because if the capacity of all edges is one unit, Ford–Fulkerson's paths will not share a common edge, but may share common vertices. Therefore, before finding the second shortest path, we modify the original graph by using the vertex-splitting technique as is used by Bhandari [18]. Vertex-splitting along the paths that have been discovered can exclude all possible paths that intersect them. To count the number of disjoint paths for all nodes, we propose a dynamic programming variant of Counting-Paths, where we start counting from sensor nodes closer to the sink. This scheme speeds up the counting process for the entire network.

We will discuss the problem of finding disjoint paths by firstly presenting the basic Counting-Paths algorithm to solve the single source – single sink problem. In this problem, we check whether or not a node has k disjoint paths to a sink. Then, we will present the dynamic programming variant of Counting-Paths to solve the multiple sources – single sink problem. After that, we will discuss the variations of the algorithm for cases with multiple sinks.

4.1. Single source – single sink problem

In *finding k disjoint paths for the single source – single sink* problem, given a graph $G = (V, E)$, we check if a source $s \in V$ has k disjoint paths to a destination $t \in V, t \neq s$, by finding the k disjoint paths from s to t, where we try to minimise $\sum_{i=1}^{k} l_i + (l_{\max} - l_{\min})$. k denotes the number of disjoint paths, l is the length of a path, $l_{\min}$ is the shortest length, and $l_{\max}$ is the longest length of the disjoint paths. If $k = \infty$, we find all possible disjoint paths from s to t.

Counting-Paths uses the Ford–Fulkerson method, which is iterative. It starts by giving an initial flow of value zero. Then at each iteration, the flow value is increased by finding an augmenting path from the source to the sink along which we can send more flow. A path P has a cost attribute, denoted as $cost(P)$. The cost of pushing a flow along an edge is defined as one unit of cost to send one unit of flow from a vertex to one of its adjacent vertices. A *path cost* is the total amount of cost to push each flow along each edge on a path. The cost is subtracted with a flow if the direction of the path is opposite to the direction of the flow. Given a flow network and a flow, the residual network consists of edges that can admit more flow. Formally, if we have a flow network G with a source and a sink, the residual network G_{res} is the network with residual capacity $capacity_{\text{res}}(v, w) = capacity(v, w) - flow(v, w)$.

A flow network is a directed graph, where each edge has a capacity. In our scenario for k disjoint paths, the WSN topology is an undirected graph and the total capacity of each edge is one. Therefore, we need a slight modification of Ford–Fulkerson to work with our specific network requirements. We also utilise vertex-splitting [19] as is used in Bhandari's algorithm [18] to exclude all possible paths that intersect the previously discovered paths. Because we use the vertex-splitting technique, we modify breadth first search to boost over split vertices to account for the extra path lengths. This modification will be explained later in the algorithm.

Algorithm 1. Counting-Paths

Input: G, s, t, k
Output: $P_i,\ \forall i = 1, \ldots c, k$
1 **for** $i \leftarrow 1$ to k **do**
2 **if** $i > 1$ **then**
3 Split vertices on the shortest paths except s and t;
4 Modify the residual network G_{res};
5 Replace external edges connected to the vertices on the shortest paths except s and t;
6 **end**
7 **if** there exists a path P_i from s to t in G_{res} **then**
8 Push flow along P_i towards t;
9 **end**
10 **if** $i > 1$ **then**
11 Remove overlapping edges;
12 **end**
13 **end**
14 **return** $P_i,\ \forall i = 1, \ldots c, k$;

We present the basic Counting-Paths algorithm in Algorithm 1 to solve the k disjoint paths for the single source – single sink problem. Counting-Paths is a combination of Ford–Fulkerson with breadth first search and the vertex-splitting technique. It takes as input a graph G, a source s, a destination t, and the number of disjoint paths sought k. The details are given below and an example to illustrate the steps when we explain the algorithm is shown in Fig. 1.

Suppose we have an input network as depicted in Fig. 1(a) and want to find two disjoint paths from the source s to the sink t. An undirected edge (v, w) in the residual network shows that a directed edge may exist

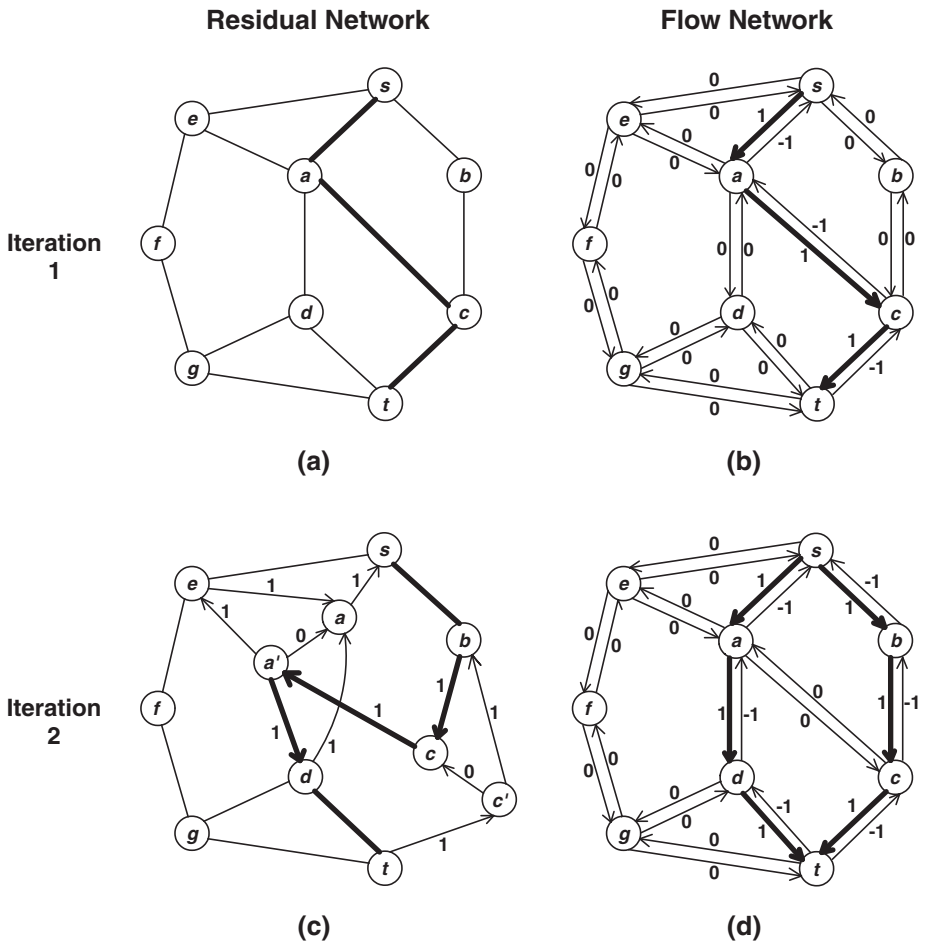

Fig. 1. Two successive iterations of Counting-Paths for $k = 2$. (a) and (c) are the residual network G_{res} of each iteration with a bold augmenting path P from the source s to the sink t. (b) and (d) show the new *flow*.

either from v to w or from w to v with the total capacity of one. For example, the first augmenting path is $P_1 = \{s, a, c, t\}$ as shown in Fig. 1(a). The flow is pushed from s to t along P_1 as shown in Fig. 1(b). We follow the pseudo-code in Algorithm 1 to find the second disjoint path as described below.

4.1.1. Split vertices

This step explains line 3 in Algorithm 1. Each vertex on the shortest paths in the residual network G_{res}, except the source s and the sink t, is split into two vertices, namely the *original vertex* and the *primed vertex*. The two vertices are joined by a directed edge of zero capacity and directed from the primed vertex to the original vertex. This is illustrated in Fig. 1(c). Vertices a and c are split into vertices a and a', c and c', respectively. We draw directed edges of zero capacity from a' to a and from c' to c. Details for other edges will be given in the following steps.

4.1.2. Modify residual network

This step explains line 4 in Algorithm 1. Recall that the residual network G_{res} is the network with residual capacity and the total capacity of each edge in our scenario is one. Therefore, for each edge (v, w) on the shortest paths, we have two cases:

If v is not the source vertex:

$$capacity_{res}(v', w) = capacity(v, w) - flow(v, w)$$
$$capacity_{res}(w, v') = capacity(v, w) - capacity_{res}(v', w)$$

If v is the source vertex:

$$capacity_{res}(v, w) = capacity(v, w) - flow(v, w)$$
$$capacity_{res}(w, v) = capacity(v, w) - capacity_{res}(v, w)$$

In our example, $capacity_{res}(s, a)$, $capacity_{res}(a', c)$ and $capacity_{res}(c', t)$ in the residual network in Fig. 1(c) are zero. However, for the clarity of the drawing purposes, the directed edges with zero capacity are not shown in the figure, except from the primed vertices to the original vertices. Moreover, $capacity_{res}(a, s), capacity_{res}(c, a')$ and $capacity_{res}(t, c')$ are all one.

4.1.3. Replace external edges

This step explains line 5 in Algorithm 1. We replace external edges connected to the vertices on the shortest paths with two oppositely directed edges of the same

capacity, and connected to the two split-vertices. External directed edges terminate on the original vertices, while they originate from the primed vertices. In the residual network in Fig. 1(c), we replace external edges connecting to vertices a and c, i.e. $(e,a),(d,a)$ and (b,c). Then, we draw directed edges of capacity one to the original vertices, i.e. from e to a, d to a, and b to c. We also draw the opposite directed edges from the primed vertices, i.e. from a' to e, a' to d, and c' to b. Note that other edges in the residual network, which are neither on the shortest path nor incident to the vertices on the shortest path, are left unmodified.

4.1.4. Find an augmenting path

This step explains how we find the shortest path in line 7 of the algorithm. In each iteration, we find an augmenting path from s to t that has the lowest path cost using breadth first search. Recall that the path cost, denoted as $cost(P)$, is the total amount of cost to push each flow along each edge on the path in the residual network G_{res}. The cost of the path is one for each edge which has no flow in it or -1 if we go against the flow. We add a little modification to breadth first search by giving advantage moves to the vertices on the previously discovered shortest paths, i.e. the split vertices. It means, when we discover a split vertex, we do not put it in the breadth first search's queue but examine it directly. This modification is aimed to tackle longer paths that are caused by overlapping edges. In our example in Fig. 1, there are two possible augmenting paths in the second iteration. They are $P_2 = \{s,b,c,a',d,t\}$ and $P_3 = \{s,e,f,g,t\}$. $cost(P_2) = 3$ because (c,a') has an opposite flow direction in Fig. 1(b), while $cost(P_3) = 4$. Therefore, we take P_2 as the next augmenting path because it has the lowest path cost as shown in bold edges in Fig. 1(c).

4.1.5. Push flow

This steps explains line 8 in Algorithm 1. If an augmenting path P exists, we merge the primed vertices with their original vertices and push the flow along P from s to t. Thus, for each edge (v,w) on P, we have:

$$flow(v,w) \leftarrow flow(v,w) + 1$$
$$flow(w,v) \leftarrow -flow(v,w)$$

Fig. 1(d) shows the new flow after we push the flow along $P_2 = \{s,b,c,a,d,t\}$. Note that $flow(a,c)$ and $flow(c,a)$ are now zero.

4.1.6. Remove overlapping edges

This steps explains line 11 in Algorithm 1. We remove the overlapping edges of the discovered paths to obtain the disjoint paths. This can be done by crossing over the two paths. If we have two paths, say $P_1 = \{v_1, v_2, v_3, v_4\}$ and $P_2 = \{v_5, v_3, v_2, v_6\}$, the common edge is (v_2, v_3) or (v_3, v_2). When we cross over the two paths, the results are $P_1 = \{v_1, v_2, v_6\}$ and $P_2 = \{v_5, v_3, v_4\}$. We have $P_1 = \{s,a,c,t\}$ in Fig. 1(a) and $P_2 = \{s,b,c,a,d,t\}$ in Fig. 1(c). These two paths share a common edge, i.e. (a,c) or (c,a). After removing the overlapping edge, the results as shown by the flow in Fig. 1(d) are $P_1 = \{s,a,d,t\}$ and $P_2 = \{s,b,c,t\}$. The length of both paths is three.

4.1.7. Analysis of Counting-Paths

Complexity of Counting-Paths: Counting-Paths, which uses Ford–Fulkerson with breadth first search, has lower time complexity than the algorithms proposed by Torrieri [5] and Bhandari [18]. Breadth first search has $O(|E|)$ time, which is slightly better than Bhandari's and Torrieri's algorithms that are based on Dijkstra $O(|V|^2)$ time. The time complexity of the Ford–Fulkerson algorithm is $O(|E|f)$, where f is the maximum flow in the graph. When we want to find k disjoint paths, the time complexity becomes $O(|E|k)$.

Correctness of Counting-Paths: We prove the correctness of Counting-Paths by comparing it to the Ford–Fulkerson algorithm. We first show that both vertex-splitting and external edge replacement in the residual graph do not change the problem for breadth first search. A vertex v on the previously discovered shortest path is split into two vertices v and v'. When v is split, the zero-length directed edge from v' to v enables breadth first search to include all possibilities of augmenting paths passing through v. Note that we call an edge that is not on the discovered shortest path but incident to v as an external edge. If the degree of v is two, v has no external edges because the two neighbours of v must also be on the discovered shortest path. If the degree of $v > 2$, v may be adjacent to one or more vertices that are not on the shortest path. Suppose there is a vertex w that is not on the shortest path and adjacent to v. In order for breadth first search to include all possibilities of augmenting paths from v to w and from w to v, the external edge is replaced with two directed edges from v' to w and from w to v, respectively. Since the vertex-splitting and the external edge replacement do not change things for the breadth first search part, if there are currently $k-1$ paths in the collection of the disjoint paths, breadth first search will find the last remaining augmenting path because breadth first search is complete. This means that if there is a solution, breadth first search will find it regardless of the kind of graph.

Secondly, we show that in each of its iterations, Counting-Paths finds disjoint paths. The Counting-Paths algorithm allows the edges of the new discovered shortest path to overlap with the previously found shortest paths. If there are some overlapping edges, Counting-Paths merges and reconstructs the paths by removing the overlapping edges that results in disjoint paths with no common edges and vertices, except the source and the destination. Since at the end of each iteration it produces disjoint paths, it stops when the number of disjoint paths is m. When Counting-Paths terminates, m is the maximum set of disjoint paths and there are no augmenting paths from the source to the destination remaining.

4.2. Multiple sources – single sink problem

The dynamic programming implementation of Counting-Paths is motivated by the fact that multi-hop WSNs are often characterised by many-to-one (convergecast) traffic patterns. If we must execute Counting-Paths for each node in the network, the overall time complexity increases to $O(|V||E|k)$. However, if we do not need to know the routing paths during the deployment process, it is not necessary for us to discover the actual paths, but only the

paths to neighbours that have k disjoint paths. This local information is used by nodes to forward their data to the nearest neighbours and the neighbours will decide where to forward them further.

In *finding k disjoint paths for the multiple sources – single sink* problem, given a graph $G = (V, E)$, we check if a source $s \in V$ has k disjoint paths to a destination $t \in V, t \neq s$, by finding the k disjoint paths, if they exist, from s to t or from s to any vertex $v \in V$ that has k disjoint paths. Below, we prove the result that justifies our dynamic programming approach.

Lemma 1. *Let v be a vertex which has vertex-disjoint paths to a subset W of k vertices none of which have a cutset of size $< k$. Then v has no cutset of size $< k$.*

Proof. Suppose v does have a cutset, C, of size $< k$. A set of size $< k$ can break at most $k - 1$ of the paths from v to W. Let $w \in W$ be any of the vertices whose paths from v are not broken by C, and so v is still connected to w. But w must be connected to the destination t, since w has no cutset of size $< k$. Therefore v is still connected to t. Therefore C is not a cutset for v. Contradiction. □

As a corollary, if a vertex v has vertex-disjoint paths to k vertices, each of which has k vertex-disjoint paths to the sink, then v must also have k vertex-disjoint paths to the sink.

Algorithm 2. Counting-Paths-DP

```
Input: G, S, t, k
Output: P_{i,j}, ∀i = 1,...c,|S|, ∀j = 1,...c,k
1  T ← {t}
2  for i ← 1 to |S| do
3    for j ← 1 to k do
4      if j > 1 then
5        Split vertices on the shortest paths except
         s_i ∈ S and r ∈ T;
6        Modify the residual network G_res;
7        Replace external edges connected to the
         vertices on the shortest paths except s_i ∈ S
         and r ∈ T;
8      end
9      if there exists a path P_{i,j} from s_i ∈ S to r ∈ T in
       G_res then
10       Push flow along P_{i,j} towards r;
11     end
12     if j > 1 then
13       Remove overlapping edges;
14     end
15   end
16   if s_i ∈ S has k disjoint paths then
17     T ← T ∪ {s_i};
18   end
19 end
20 return P_i, ∀i = 1,...c,k;
```

In our dynamic programming approach, we start by finding the k disjoint paths from vertices closer to the sink. For each vertex, if we can find disjoint paths to k vertices that have k disjoint paths, we do not need to find the k disjoint paths to the sink and we can proceed to the next vertex. The algorithm for the dynamic programming variant is given in Algorithm 2. It takes as input a graph G, a set S of source vertices, a destination t, and the number of disjoint paths sought k. T represents a collection of destination vertices, which are the sink and the vertices which have k disjoint paths to the sink. Note that line 3–15 are similar to Algorithm 1, but the shortest path may terminate at any vertices in T.

In the multiple sources – single sink problem, we vary the heuristic techniques to pick which vertex is examined first:

1. Order vertices by distance from sink, breaking ties by smallest ID.
2. Order vertices by distance from sink, breaking ties by highest degree then smallest ID.
3. Order vertices by distance from sink, breaking ties by most processed neighbours then smallest ID.
4. Dynamically order vertices by most processed neighbours, breaking ties by distance from sink then smallest ID.

We will evaluate these four heuristic techniques in the performance of the dynamic programming variant of Counting-Paths later in Section 5.2.

4.3. Single source – multiple sinks and multiple sources – multiple sinks problems

Two other variations of our problems are the single source – multiple sinks and multiple sources – multiple sinks problems. In these multiple sink cases, a well-known approach is to add a *supersink* as an imaginary vertex that has connection to the original sinks. By doing this, we reduce the problem of single source – multiple sinks to the problem of single source – single sink, while the problem of multiple sources – multiple sinks is simplified to the problem of multiple sources – single sink.

When there are many sinks, we have two cases, based on where the disjoint paths must terminate: *different-sinks* and *any-sinks*. The different-sinks problem is where the k disjoint paths must terminate at k different sinks to guarantee reliability of the network. The any-sinks problem is the case where the k disjoint paths may terminate at any sinks. In the different-sinks problem, for each connection from an original sink t to the supersink t', we set $capacity(t, t') = 1$, so the edge can be used at most once. However, for the any-sinks problem, we set $capacity(t, t') = k$, so the paths can traverse some original sinks more than once, but at most k times before reaching the supersink.

5. Evaluation of Counting-Paths

In this section, we implement and evaluate all algorithms' performance in C++. Later in Section 8, we use a network simulator to evaluate network operations. All experiments are carried out on a 2.40 GHz Intel Core2

Duo CPU with 4 GB of RAM. Our simulation results are based on the mean value of 20 different randomly generated network deployments, enough to achieve a 95% confidence in the standard error interval, which are shown as error bars in the results. We do not show error bars in line graphs and graphs with logarithmic scale to improve readability of the graphs. Our network consists of up to 100 nodes deployed within randomly perturbed grids of a two-dimensional area, where a node is placed in a unit grid square of 8 m $\times$ 8 m and the coordinates are perturbed. We generate 5×5, 7×7 and 10×10 grid squares to deploy 25, 49 and 100 nodes, respectively. All nodes use the same transmission range, i.e. 10 m, which is realistic for 0 dBm transmission power in indoor environments [24].

We compared the performance of the basic and the dynamic programming variant of Counting-Paths to the Modified Dijkstra algorithm by Bhandari [18] and two algorithms proposed by Torrieri [5], namely Fast Pathfinding and Maximum Paths. We followed the three algorithms detailed in [18,5], implemented them and then verified the results by using the examples in the papers.

Bhandari's algorithm [18] requires two runs of a modified Dijkstra algorithm to find two disjoint paths between a source and a sink. Dijkstra's algorithm is slightly modified to handle negative directed edges. The main idea is to exclude all possible paths between the source and the sink that intersect with the first shortest path found during the search for the second shortest path. Exclusion of such path is achieved by vertex-splitting along the first shortest path found. Bhandari's algorithm begins by finding the first shortest path for a pair of vertices under consideration using the modified Dijkstra algorithm. The graph is then modified by:

1. replacing edges on the shortest path by negative directed edges toward the source,
2. splitting vertices on the shortest path, joining them by zero weighted directed edges toward the source, and
3. replacing edges connected to vertices on the shortest path by two oppositely directed edges of the original weight.

After that, the modified Dijkstra algorithm is run again on the modified graph. The original graph is then restored and the overlapping edges of the two paths found are removed to obtain a pair of disjoint paths. For the k disjoint paths problem, where $k > 2$, the algorithm is performed iteratively to obtain more disjoint paths in a given network graph, provided such paths exist.

Torrieri's algorithms [5] calculate a set of short disjoint paths, which do not exceed the longest acceptable path length, between a source and a sink. In each iteration, the algorithms select the shortest path, remove the intermediate vertices in the path from further use by zeroing the rows and the columns of the intermediate vertices in the adjacency matrix and then select the next shortest path using only the remaining vertices. Fast Pathfinding is the simplest approximate algorithm, which executes only the first step in the construction of the optimal set. In this algorithm, if two or more remaining paths of length l are the shortest, one of them is chosen arbitrarily. Maximum Paths executes the first two steps in the construction of the optimal set. That is, if two or more remaining paths of length l are the shortest and they exclude the fewest other paths of length l, then one of the remaining paths is chosen arbitrarily.

In the simulation, we compared the efficiency and the accuracy of Counting-Paths against the three algorithms in terms of the number of table lookups, runtime, storage capacity and the average number of disjoint paths found per node.

5.1. Single source – single sink problem

In each topology, the location of the sink is fixed at the top-left corner of the network and the location of the source is at the bottom-right corner, so as to maximise the distance between them. By the simulation of single source – single sink, we evaluate how many disjoint paths each algorithm can find, so we set $k = \infty$.

Fig. 2 shows that Counting-Paths and Modified Dijkstra are more efficient than Fast Pathfinding and Maximum Paths in terms of the total numbers of table lookups. The numbers of table lookups for the two algorithms by Torrieri increase significantly when the number of nodes increases, because they try to find all possible combinations of paths. The results also show that Counting-Paths has fewer table lookups compared to Modified Dijkstra, because in the worst case, breadth first search has lower complexity, i.e. $O(|E|)$ time, than Dijkstra's $O(|V|^2)$ time [25]. These results correspond to the runtime reported in Table 2.

We compare the storage capacities in Fig. 3. Counting-Paths uses more storage than Modified Dijkstra because it has to maintain the residual network and the flow network for the Ford–Fulkerson algorithm, as well as a queue for breadth first search. Maximum Paths uses more storage than Fast Pathfinding because it stores all possible combinations of paths in each iteration, whereas Fast Pathfinding only stores sets of nodes to construct the paths.

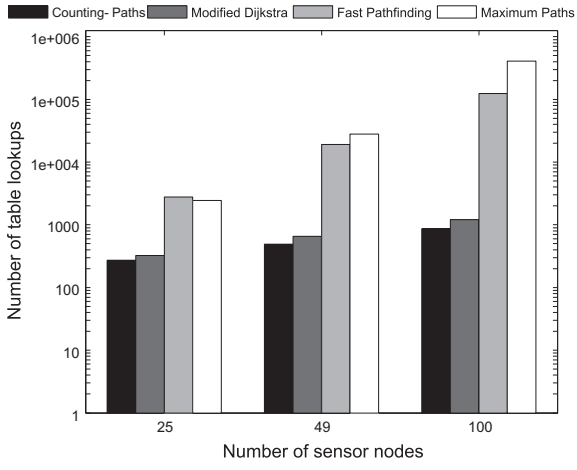

Fig. 2. Number of table lookups versus number of sensor nodes for single source – single sink.

Table 2
Disjoint paths algorithms' runtime for single source – single sink.

Algorithms	Runtime (s)		
	25-node	49-node	100-node
Counting-Paths	0.000313	0.001156	0.004609
Modified Dijkstra	0.000337	0.001212	0.004976
Fast pathfinding	0.001140	0.002938	0.013624
Maximum paths	8.001600	11.829600	16.473550

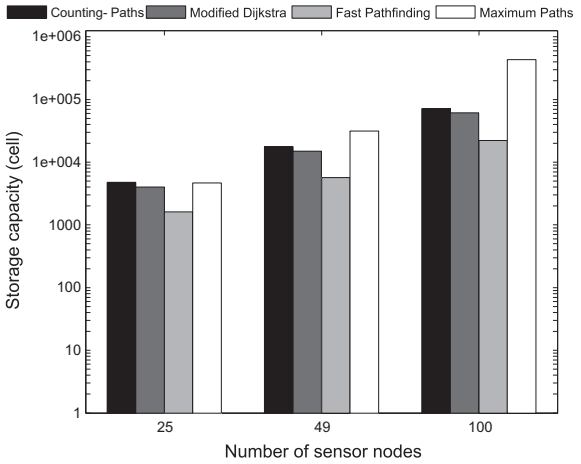

Fig. 3. Storage capacity versus number of sensor nodes for single source – single sink.

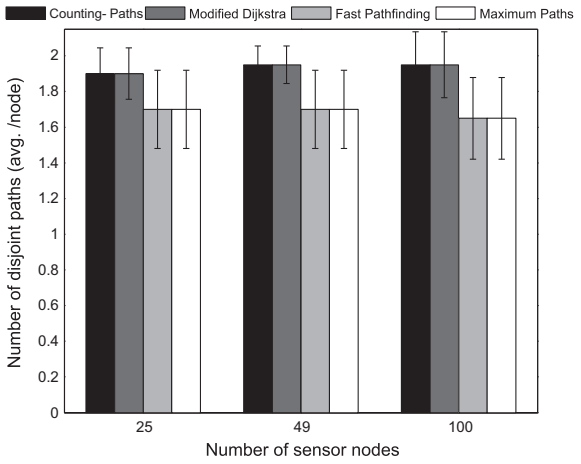

Fig. 4. Number of disjoint paths versus number of sensor nodes for single source – single sink.

Fig. 4 shows that Counting-Paths and Modified Dijkstra discover more disjoint paths than Fast Pathfinding and Maximum Paths. This happens because the first two algorithms allow overlapping edges, which will then be removed, and paths reconstruction. However, in Fast Pathfinding and Maximum Paths, once a path is selected, the intermediate nodes are removed from further search.

The relationship between the average number of disjoint paths found and the path length in 100-node networks is presented in Fig. 5. We compare Counting-Paths and Modified Dijkstra's results, especially for path lengths 14–16. There is high variation due to different topologies although there is a tendency for Modified Dijkstra to discover more shorter paths and more longer paths, while Counting-Paths tries to minimise the length difference between the shortest path and other alternative paths.

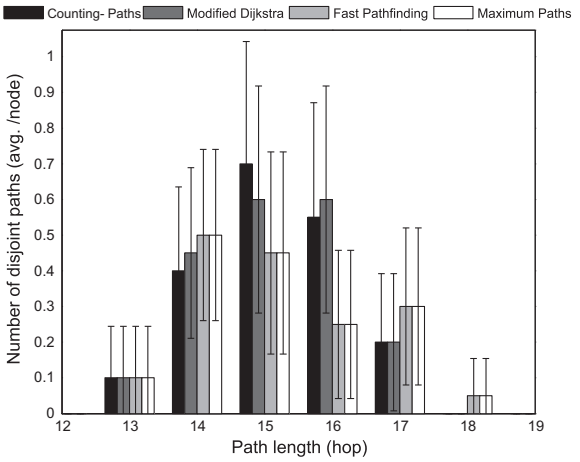

Fig. 5. Number of disjoint paths versus path length for single source – single sink.

5.2. Multiple sources – single sink problem

Our main aim of this section is to evaluate the dynamic programming (DP) variant of Counting-Paths by varying heuristic techniques to select which node is examined first as detailed in Section 4.2 and to compare them to the basic Counting-Paths. Since Counting-Paths is marginally faster and finds better results compared to Modified Dijkstra, as shown in the evaluation of single source – single sink, we do not evaluate Modified Dijkstra in this section. For the multiple sources – single sink problem, the location of the sink is still at the top-left corner of the network, while all sensor nodes are the source nodes. We want to find

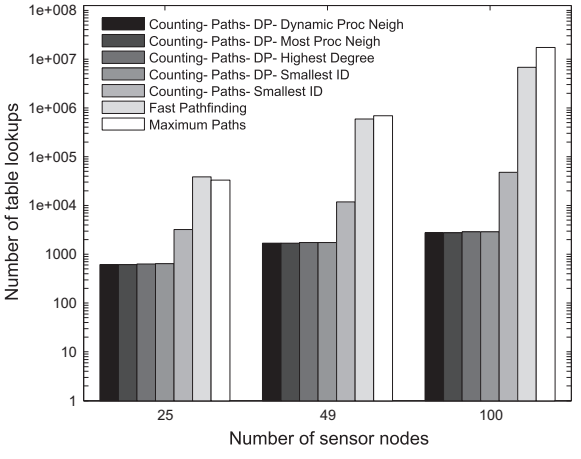

Fig. 6. Number of table lookups versus number of sensor nodes for multiple sources – single sink.

Table 3
Disjoint paths algorithms' runtime for multiple sources – single sink.

Algorithms	Runtime (s)		
	25-node	49-node	100-node
Counting-Paths-DP-Dynamic Most Proc Neigh	0.000515	0.003626	0.027218
Counting-Paths-DP-Most Proc Neigh	0.000476	0.003299	0.023007
Counting-Paths-DP-Highest Degree	0.000588	0.003531	0.025343
Counting-Paths-DP-Smallest ID	0.000563	0.003485	0.025336
Counting-Paths-Smallest ID	0.001258	0.008171	0.062727
Fast pathfinding	0.015026	0.070181	0.577030
Maximum paths	125.522800	373.737500	1010.415550

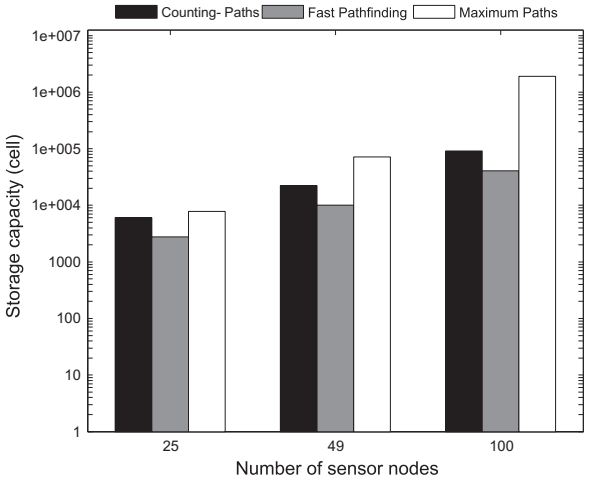

Fig. 7. Storage capacity versus number of sensor nodes for multiple sources – single sink.

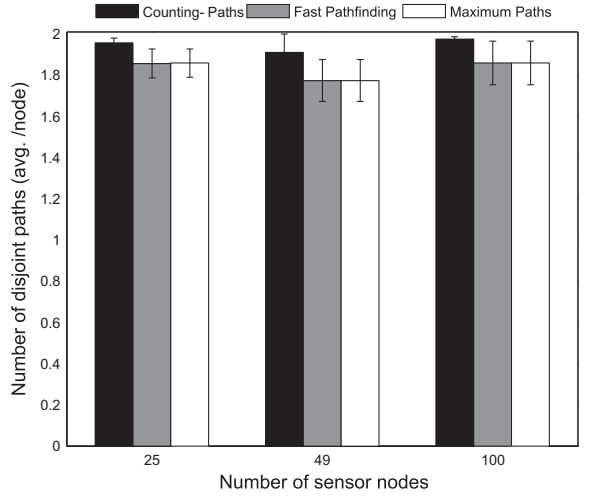

Fig. 8. Number of disjoint paths versus number of sensor nodes for multiple sources – single sink.

whether all nodes in the network have two disjoint paths, so we set $k = 2$.

Fig. 6 and Table 3 show the number of table lookups and the algorithms' runtime, respectively. Without dynamic programming, the number of table lookups is not influenced by which node is selected first, so we only use the ordering of smallest ID heuristic for the basic Counting-Paths. Dynamic programming reduces the runtime by at least 50% over the basic Counting-Paths. Out of the individual heuristic to select which node is examined first, the heuristic by most processed neighbours is the most effective one giving 10% improvement over the next best heuristic.

We show the storage capacity and the average number of disjoint paths found per node in Figs. 7 and 8, respectively. We only present the results for Counting-Paths using one bar for each group because the different variants record the same value. We observe that these two figures have similar trends with the single source – single sink cases in Figs. 3 and 4.

6. GRASP-ARP

GRASP-ARP is a local search algorithm to deploy additional relays for ensuring the existence of k disjoint paths in WSNs with sinks. Each iteration consists of construction phase and local search phase. The construction phase randomly selects relays from a set of candidate locations to guarantee the existence of k disjoint paths from every sensor node to the sink(s). The local search phase tries to minimise the number of selected relays by replacing them with a better set from the neighbourhood. GRASP-ARP requires repeated counting of the number of disjoint paths, for which we use either Counting-Paths or its dynamic programming variant. With the basic Counting-Paths, GRASP-ARP ensures a length constraint by rejecting solutions in the local search phase that do not meet the constraint. On the other hand, it runs faster with the dynamic programming variant.

We consider WSNs where the vertices are partitioned into sensors (T), relays (R) and sinks (S), so in the graph representation $V = T \cup R \cup S$. We define a WSN to be (k,l)-*sink-connected* if and only if for every vertex $v \in T$, there are k disjoint paths from v to S of length $\leqslant l$.

We can now define the *additional relay placement problem*: given a graph $G = (T \cup A \cup S, E)$, where A is a set of candidate locations to deploy relays, find a minimal subset $R \subseteq A$ such that $H = (T \cup R \cup S, E\downarrow_{T \cup R \cup S})$ is (k,l)-sink-connected.

For our algorithm, we introduce some secondary definitions. k_v is the number of length-bounded disjoint paths a sensor v has. $X \subseteq T$ is the set of sensors with $k_x < k, \forall x \in X$. $Y \subseteq T$ is the set of sensors with $k_y \geqslant k, \forall y \in Y$, and such that y is on a path of a sensor $x \in X$ to a sink or a vertex with at least k disjoint paths. $Z \subseteq T$ is a set of sensors with $k_z \geqslant k, \forall z \in Z$, such that z does not appear on a path of a sensor $x \in X$ to a sink or a vertex with k disjoint paths. To determine the sets X, Y, and Z, we use Counting-Paths to find k disjoint paths from all sensors to sinks. Then, for each sensor, we count how many disjoint paths satisfy the length restriction $l_{\max}$.

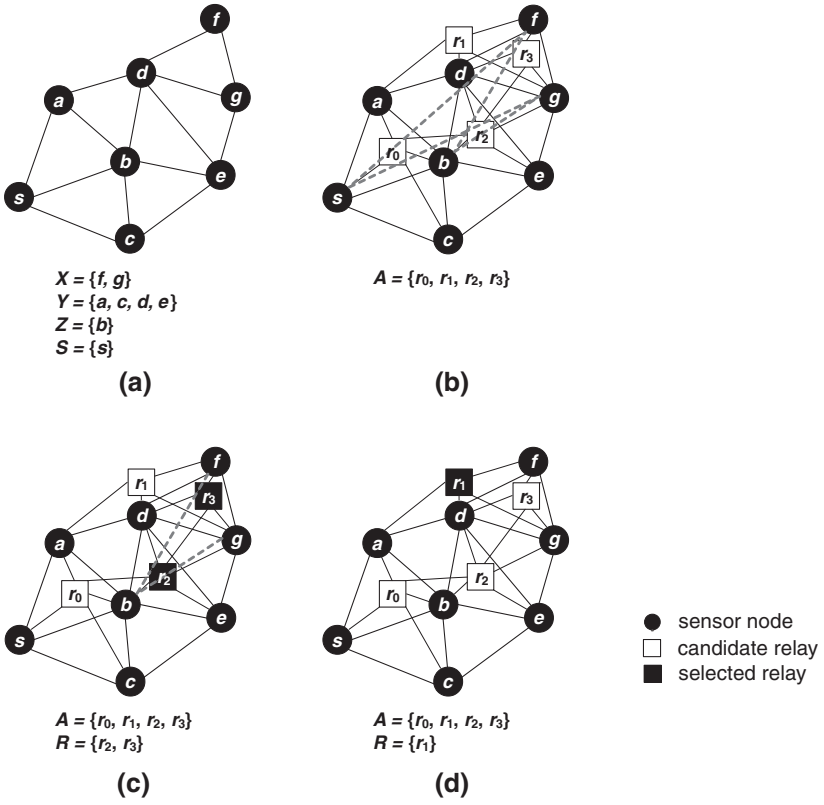

Fig. 9. An example of GRASP-ARP execution for $k = 3$ and $l = 4$. (a) is a network of sensors $T = \{a, b, c, d, e, f, g\}$ and a sink $S = \{s\}$, (b) shows a set of candidate locations to deploy relays $A = \{r_0, r_1, r_2, r_3\}$, (c) is a solution produced by the construction phase where a set of relay $R = \{r_2, r_3\}$ is selected, and (d) is a solution produced by the local search phase where $R = \{r_1\}$.

An example to illustrate the details of GRASP-ARP is shown in Fig. 9. Suppose we have a network as depicted in Fig. 9(a) and want to find 3 disjoint paths from every sensor in the set of sensors $T = \{a, b, c, d, e, f, g\}$ to the sink $S = \{s\}$, where the maximum path length $l = 4$. After running Counting-Paths, we identify $X = \{f, g\}$ because f and g only have 2 disjoint paths, $Y = \{a, c, d, e\}$ because they appear in the disjoint paths of f and g, and $Z = \{b\}$ because b does not appear in the disjoint paths of f or g.

6.1. Construction phase

The first step in any GRASP algorithm is to construct an initial solution. Given an original graph $G = (T \cup A \cup S, E)$, where $T = X \cup Y \cup Z$, firstly we find a graph $G' = (V', E')$, where $V' = X \cup Z \cup S$ and $E' = \{(v, w) \mid v \in X, w \in Z \cup S, \exists srp(v, w)\}$. $srp(v, w)$ denotes the shortest relay path from v to w in the original graph $G = (T \cup A \cup S, E)$ in terms of cost c, where all intermediate vertices in the path are candidate relays. The edge's cost $c'(v, w) = srp(v, w)$ is associated with each edge $(v, w) \in E'$. The edge's cost $c'(v, w)$ is calculated as the number of candidate relays in the shortest relay path $srp(v, w)$. In Fig. 9(b), given the set of candidate locations to deploy relays $A = \{r_0, r_1, r_2, r_3\}$, we show edges in E' by using dashed lines and their costs are $c'(f, b) = 2, c'(f, s) = 3, c'(g, b) = 1, c'(g, s) = 2$.

After that, we find a minimum forest F of G' such that for each $v \in X$, we select $k - k_v$ least cost edges to $w \in Z \cup S$. Then, we replace the edges in F with the edges in the shortest relay paths in the original graph G and save this set of paths P. We add randomisation to the initial solution by building a restricted candidate list with all edges $(v, w) \in E'$ such that $c'(v, w) \leqslant c'_{min} + \alpha(c'_{max} - c'_{min})$, where $0 \leqslant \alpha \leqslant 1$. c'_{min} and c'_{max} denote the least and the largest costs among all unselected edges, respectively. Then, $k - k_v$ edges are selected at random from the list. As shown in Fig. 9(c), $(f, b) \in E'$ is selected for f and $(g, b) \in E'$ is selected for g. Therefore, the solution produced by the construction phase is the relay set $R = \{r_2, r_3\}$.

6.2. Node-based local search

The next stage in a GRASP algorithm is to explore the neighbourhood of the initial solution, looking for lower cost solutions. Let R be the set of relays in the current forest F and $R_{used}(P)$ denotes the number of relays used in the current set of paths P. We explore the neighbourhood of the current solution by either adding a new relay $r \in A \setminus R$ into R, or by eliminating a relay $t \in R$ from R. In each iteration of the local search, the evaluation of elimination moves is performed only if there are no improving insertion moves. In our example, the solution produced by the local search phase is the relay set $R = \{r_1\}$ as shown in Fig. 9(d).

Algorithm 3. GRASP-ARP

Input: G, S, A, X, Z, k, α, *max_iterations*
Output: R^*, P^*
1 *best_value* $\leftarrow \infty$;
2 Compute $G' = (V', E')$ and $c'(v, w), \forall (v, w) \in E'$
3 **for** $i \leftarrow 1$ to *max_iterations* **do** /* Construction phase */
4 Find F of $G' = (V', E')$ with R and P as the result;
5 **repeat**
6 **repeat**
7 *insertion* $\leftarrow$ **false**; *elimination* $\leftarrow$ **false**;
8 *best_set* $\leftarrow R$; *best_number* $\leftarrow R_{used}(P)$; /*Insertion moves */
9 **foreach** $r \in A \backslash R$ **do**
10 Counting-Paths $\forall x \in X$ from F^{+r}, result in P_{new};
11 **if** $R_{used}(P_{new}) <$ *best_number* **then**
12 *best_set* $\leftarrow R \cup \{r\}$; *best_number* $\leftarrow R_{used}(P_{new})$;
13 **end**
14 **end**
15 **If** *best_number* $< R_{used}(P)$ **then**
16 $R \leftarrow R \cup \{r\}; F \leftarrow F^{+r}; P \leftarrow P_{new}; R_{used}(P) \leftarrow R_{used}(P_{new})$;
17 *insertion* $\leftarrow$ **true**;
18 **end**
19 **until** *insertion* = ***false***;
20 *best_set* $\leftarrow R$; *best_number* $\leftarrow R_{used}(P)$; /* Elimination moves */
21 **for each** $t \in R$ **do**
22 Counting-Paths $\forall x \in X$ from F^{-t}, result in P_{new};
23 **if** $R_{used}(P_{new}) <$ *best_number* **then**
24 *best_set* $\leftarrow R \backslash \{t\}$; *best_number* $\leftarrow R_{used}(P_{new})$;
25 **end**
26 **end**
27 **if** *best_number* $< R_{used}(P)$ **then**
28 $R \leftarrow R \setminus \{t\}; F \leftarrow F^{-t}; P \leftarrow P_{new}; R_{used}(P) \leftarrow R_{used}(P_{new})$;
29 *elimination* $\leftarrow$ **true**
30 **end**
31 **untill** *elimination* = ***false***;
32 **if** $R_{used}(P) <$ *best_value* **then** /* Best solution update */
33 $R^* \leftarrow R; P^* \leftarrow P$; *best_value* $\leftarrow R_{used}(P)$;
34 **end**
35 **end**
36 **return** R^*, P^*;

6.3. Algorithm description

The pseudocode for GRASP-ARP is given in Algorithm 3. It takes as input the original graph $G = (V, E)$, the set S of sinks, the set A of candidate relays, the set X of sensors with $k_x < k$, the set Z of sensors with $k_z \geqslant k$ but do not appear on any discovered paths, the number of disjoint paths sought k, the restricted candidate list parameter α ($0 \leqslant \alpha \leqslant 1$), and the number of iterations (*max_iterations*). Graph $G' = (V', E')$ is computed in line 2. The procedure is repeated *max_iterations* times. In each iteration, a greedy randomised solution for a minimum forest F of G' is constructed in line 4. Let R be the set of relays in the current forest F, P be the set of paths, and $R_{used}(P)$ be the number of relays used in P.

The local search starts with the initialisation in line 8. The loop from line 9 to 14 searches for the best insertion move. In line 10, we count the number of disjoint paths from each sensor $x \in X$, defined by the insertion of vertex r into the current set of relays. Let P_{new} be its new set of paths. In line 11, we check if this new solution P_{new} improves the current best solution. Solution updates are made in line 12. When all insertion moves have been evaluated, we check in line 15 if an improving solution has been found and update the solution in line 16. Then, the local search continues. If no improving solution is found from the insertion moves, the elimination moves from line 21 to 26 are evaluated. These moves are similar to the insertion moves, except they are defined by the elimination of node t from the current set of relay nodes. If, at the end of the local search,

we found a better solution compared to the best solution, updates are made in line 33. The best set R^* of relays and the best set P^* of paths are returned in line 36.

7. Evaluation of GRASP-ARP

The sensor nodes are deployed in randomly perturbed grids, where a sensor node is placed in a unit grid square of 8 m × 8 m and the coordinates are perturbed. In order to get sparse networks (average degree 2–3), we generate more grid points than the number of nodes. For example, we use 6 × 6, 8 × 8 and 11 × 11 grid squares to randomly deploy 25, 49 and 100 nodes, respectively. Candidate relays are also distributed in a grid area, where a candidate occupies a unit grid square of 6 m × 6 m. For 25-node, 49-node and 100-node topologies, we use 49, 100 and 196 candidate relays, respectively. Both sensor and relay nodes use the same transmission range, i.e. 10 m. The maximum path length (l_{max}) is set to 10 for 25-node, 15 for 49-node and 20 for 100-node networks.

We compared the performance of GRASP-ARP to partial k-connectivity-repair (K-CONN-REPAIR for short) proposed by Pu et al. [9]. We followed the algorithm detailed in [9]. It firstly computes a weighted complete graph, where the weight of an edge is one less than the Euclidean distance between a pair of sensors. The edge's weight represents the number of additional relays required to connect two sensors by a straight path. After that, this algorithm finds an approximate minimum-weight vertex k-connected subgraph by repeatedly adding edges in increasing order of weight until the subgraph is k-connected. If the subgraph is already k-connected, it repeatedly attempts to remove edges in decreasing order of weight, but putting the edge back if it is important for k-connectivity. Finally, it places one relay along each edge every one unit distance.

We also made two necessary modifications for this algorithm to work in constrained deployment locations, where relays can only be placed at candidate locations.

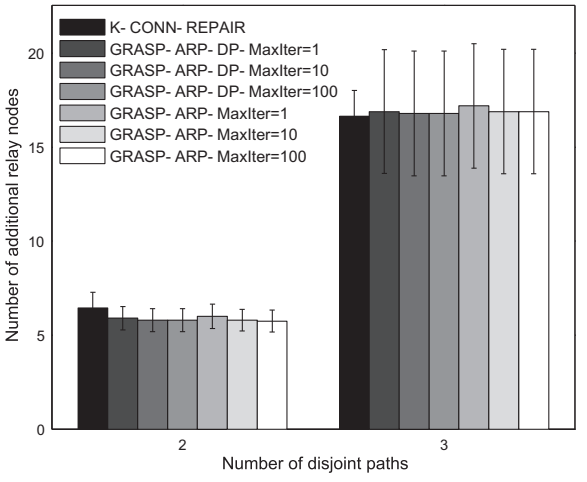

Fig. 10. Number of additional relay nodes needed versus number of disjoint paths required for multiple sources – single corner sink in 25-node networks.

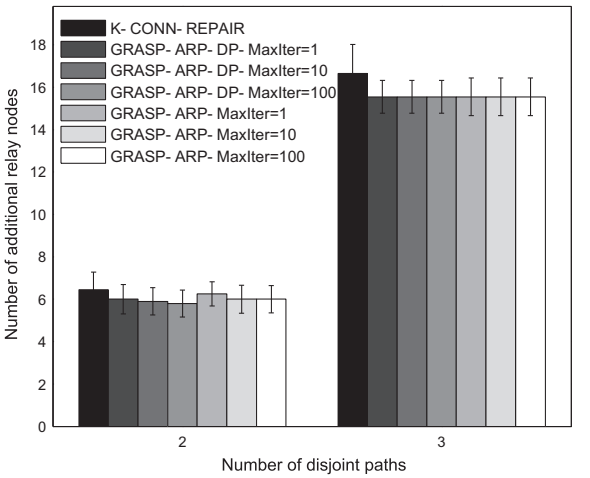

Fig. 11. Number of additional relay nodes needed versus number of disjoint paths required for multiple sources – single centre sink in 25-node networks.

The original K-CONN-REPAIR deploys relays along a straight line between two sensor nodes. Therefore, our first modification is to place relay nodes in candidate locations along the shortest relay path between two sensor nodes. When all relay nodes are deployed, we add our second modification by trying to remove relays one by one by still preserving the node k-connectivity. The connectivity is checked using a maximum network-flow-based checking algorithm [26] as is used in [27].

In the simulation, we compared the effectiveness and efficiency of GRASP-ARP against K-CONN-REPAIR in terms of the number of additional relays needed and the runtime.

7.1. Multiple sources – single sink problem

We choose the sink at the top-left corner or in the centre of the network, while all sensor nodes are the source nodes. In this simulation, we want to create networks with

Table 4
Additional relay placement algorithms' runtime for multiple sources – single sink in 25-node networks.

Algorithms	Runtime (s)	
	$k = 2$	$k = 3$
K-CONN-REPAIR	6.2891	7.4930
Sink at the corner		
GRASP-ARP-DP-MaxIter = 1	5.6796	11.0656
GRASP-ARP-DP-MaxIter = 10	24.3469	36.8148
GRASP-ARP-DP-MaxIter = 100	185.0354	291.5102
GRASP-ARP-MaxIter = 1	9.7672	19.2712
GRASP-ARP-MaxIter = 10	39.4860	59.1335
GRASP-ARP-MaxIter = 100	331.5993	459.0313
Sink at the centre		
GRASP-ARP-DP-MaxIter = 1	3.4163	8.5166
GRASP-ARP-DP-MaxIter = 10	9.7031	20.8140
GRASP-ARP-DP-MaxIter = 100	73.1475	144.7258
GRASP-ARP-MaxIter = 1	5.6696	14.2375
GRASP-ARP-MaxIter = 10	16.7616	34.3545
GRASP-ARP-MaxIter = 100	129.4321	239.1384

2-connectivity and 3-connectivity, so we use $k = 2$ and $k = 3$. We simulate GRASP-ARP using the dynamic programming (DP) variant and the basic Counting-Paths algorithm. For the dynamic programming variant, we use the dynamic ordering of most processed neighbours as the heuristic technique to pick which sensor node is examined first, because it has been shown as one of the best heuristics in the evaluation of Counting-Paths section. For the basic Counting-Paths algorithm, we use the ordering by smallest ID to select nodes, because the performance of the algorithm is not influenced by which node is selected first.

Fig. 10 shows the number of additional relays needed for $k = 2$ and $k = 3$ for the case where the sink location is at the top-left corner of 25-node networks, while Fig. 11 shows the results for the case where the sink is in the centre of the networks. GRASP-ARP finds nearly the same number of additional relays compared to *K*-CONN-REPAIR when the position of the sink is in the corner of the network and $k = 3$. However, it needs fewer relays for $k = 2$ because *K*-CONN-REPAIR deploys excessive relays for *k*-connectivity for each pair of nodes. When the position of the sink is in the centre of the network, GRASP-ARP outperforms *K*-CONN-REPAIR for both $k = 2$ and $k = 3$ because the sink has higher connectivity in that position. The results also show why more than one iteration is needed for the local search. For the case $k = 2$ and $k = 3$ in Fig. 10, and $k = 2$ in Fig. 11, *max_iteration* = 1 shows poorer results compared to *max_iteration* = 100. However, in these three cases, *max_iteration* = 10 is sufficient.

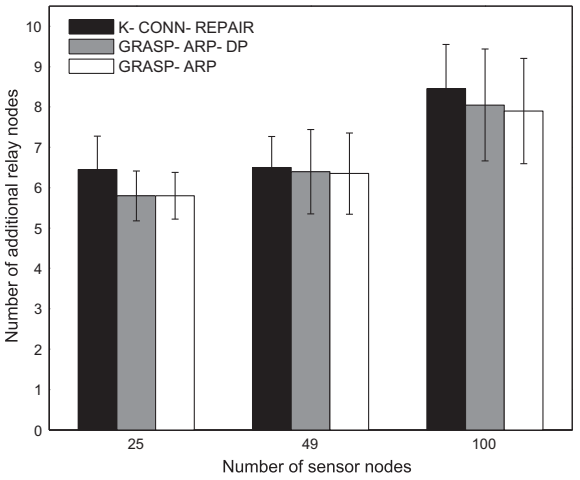

Fig. 12. Number of additional relay nodes needed versus number of sensor nodes for multiple sources – single corner sink.

Table 5
Additional relay placement algorithms' runtime for multiple sources – single sink.

Algorithms	Runtime (s)		
	25-node	49-node	100-node
K-CONN-REPAIR	6.2891	254.7343	10003.8000
GRASP-ARP-DP	24.3469	421.2829	7897.9650
GRASP-ARP	39.4860	619.3118	13964.9400

Table 4 shows the algorithms' runtime, where for small networks, GRASP-ARP takes longer compared to *K*-CONN-REPAIR especially for $k = 3$. This happens because GRASP-ARP repeatedly executes the Counting-Paths algorithm during the local search phase to find disjoint paths. GRASP-ARP with the dynamic programming variant of Counting-Paths (GRASP-ARP-DP) is faster than with the basic Counting-Paths (GRASP-ARP) because the dynamic programming has lower complexity.

We extend our simulation to larger networks up to 100 nodes. Fig. 12 depicts the number of additional relay nodes needed for 25, 49 and 100-node networks. In this simulation, we use $k = 2$ and set the sink position at the top-left corner of the network. We use *max_iteration* = 10 for GRASP-ARP because this number of iteration produces similar results to 100 iterations as shown in Figs. 10 and 11. In all sizes of networks, GRASP-ARP outperforms *K*-CONN-REPAIR with fewer relay nodes. The variations in

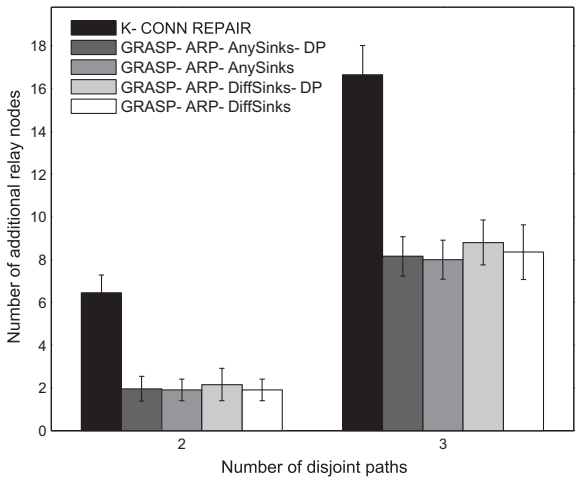

Fig. 13. Number of additional relay nodes needed versus number of disjoint paths required for multiple sources – multiple sinks in 25-node networks.

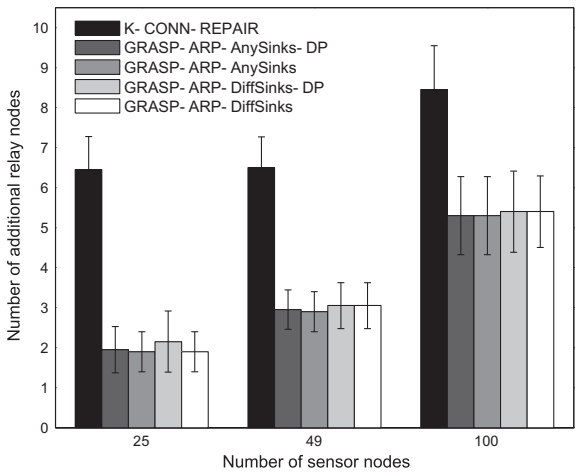

Fig. 14. Number of additional relay nodes needed versus number of sensor nodes for multiple sources – multiple sinks.

Table 6
Additional relay placement algorithms' runtime for multiple sources – multiple sinks.

Algorithms	Runtime (s)					
	25-node		49-node		100-node	
	$k=2$	$k=3$	$k=2$	$k=3$	$k=2$	$k=3$
K-CONN-REPAIR	6.2891	7.4930	254.7343	269.1470	10003.8000	11151.0500
GRASP-ARP-AnySinks-DP	1.7844	10.9984	37.7884	1127.0680	517.3695	18417.0190
GRASP-ARP-AnySinks	2.5180	16.5088	55.3843	1755.4586	735.7915	28789.0951
GRASP-ARP-DiffSinks-DP	1.6351	11.0267	31.3648	1061.5627	426.6710	19943.2187
GRASP-ARP-DiffSinks	2.3828	18.8875	51.4437	2370.4857	822.0718	49819.2537

the graph are caused by different topologies. The algorithms' runtime is presented in Table 5, where we show that GRASP-ARP-DP is faster for bigger problems, i.e. 100-node networks, and so it appears to scale better.

7.2. Multiple sources – multiple sinks problem

We use the same simulation settings as in the multiple sources – single sink cases. However, in this simulation, we have four sinks deployed at the top-left, top-right, bottom-left and bottom-right corners of the network, while all sensor nodes are the source nodes. We simulate GRASP-ARP using the dynamic programming variant and the basic Counting-Paths algorithm. Recall that in the multiple sink problem, there are two cases where the disjoint paths terminate at: *different-sinks* and *any-sinks*. The different-sinks problem is where the paths must terminate at k different sinks, while the any-sinks problem is the case where the paths may terminate at any sinks.

Fig. 13 shows the number of relay nodes required for $k=2$ and $k=3$ in 25-node networks. GRASP-ARP results shown here are the simulation results with *max_iteration* = 10. The results show that GRASP-ARP in the multiple sources – multiple sinks scenario outperforms *K*-CONN-REPAIR with at least 50% fewer additional relays. This happens because GRASP-ARP only finds k disjoint paths to the dedicated sinks, either different sinks or any sinks. On the other hand, *K*-CONN-REPAIR must provide k-connectivity for an entire network. The results also show that the different-sinks case requires more relays than the any-sinks case because disjoint paths must be established to different sinks.

Fig. 14 shows the simulation results when we extend the simulations to larger networks. We use $k=2$ and *max_iteration* = 10 for GRASP-ARP. In all network's sizes, GRASP-ARP outperforms *K*-CONN-REPAIR with fewer additional relay nodes. For 100-node networks, GRASP-ARP deploys 35% fewer relays. The algorithms' runtime is presented in Table 6.

8. Network performance evaluation

We evaluate the robustness of the topologies in the network simulator ns-2 [28] by killing nodes at random. Ns-2 is a discrete-event network simulator widely used for WSN and other network simulations. We take the resulting topologies generated by GRASP-ARP and deploy sensor nodes, relay nodes and sinks according to the deployment plans. We compare GRASP-ARP's designs to the original topologies, i.e. topologies without relays, in terms of network connectivity. Network connectivity is the percentage of live source nodes that are still connected to the sink through multi-hop communication. Our parameters used in ns-2 simulation are based on Tmote sky hardware [29] for 10-m transmission range.

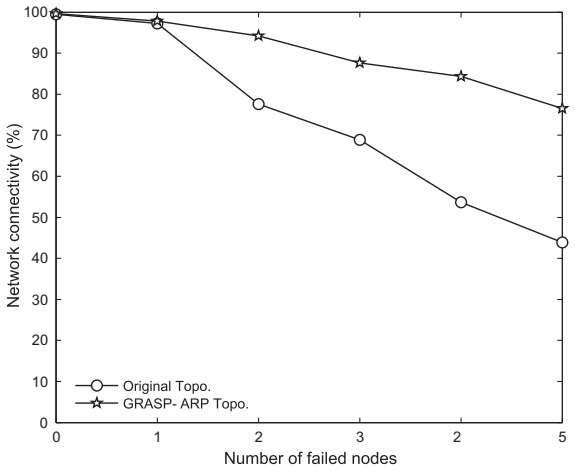

Fig. 15. Network connectivity with ER-MAC where a node dies every 1000 s for the case of multiple sources – single sink where $k=2$.

In each experiment, we simulate a multi-hop data gathering using ER-MAC [30,31] with its forward-to-parent routing mechanism. We choose ER-MAC because of its ability to adapt to traffic and topology changes. In the simulation, all sensor nodes are the source nodes that generate packets with a fixed traffic load, i.e. 0.1 packets/node/s. They also forward other nodes' packets toward the sink. Relay nodes do not generate packets, but only forward them, and are used from the start of the simulation. We do not assume that the relay nodes are more robust than the sensor nodes, so they too may fail during the simulation period. During the simulation, we increase the number of dead nodes gradually by killing one node, either a sensor node or a relay node, in each time step. The simulation results presented are based on the average of five topologies that are simulated five times each.

8.1. Network topologies with one sink

We simulate the original topologies, which are 100-node networks, and their resulting topologies generated by GRASP-ARP with the dynamic programming variant of

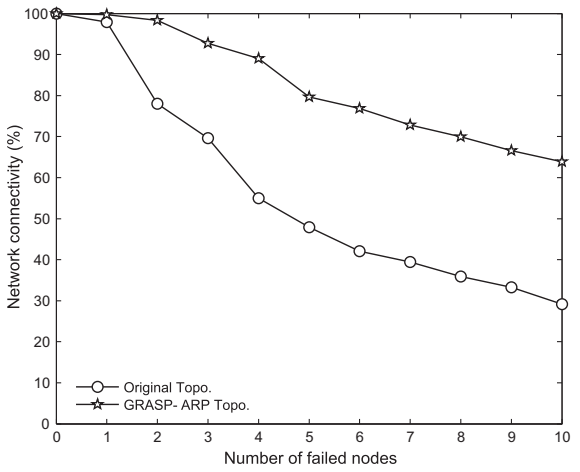

Fig. 16. Network connectivity with ER-MAC where a node dies every 1000 s for the case of multiple sources – single sink where $k = 3$.

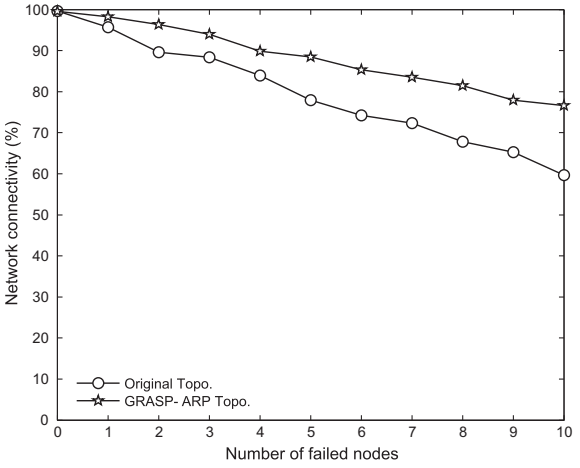

Fig. 17. Network connectivity with ER-MAC where a node dies every 250 s for the case of multiple sources – four sinks where $k = 2$.

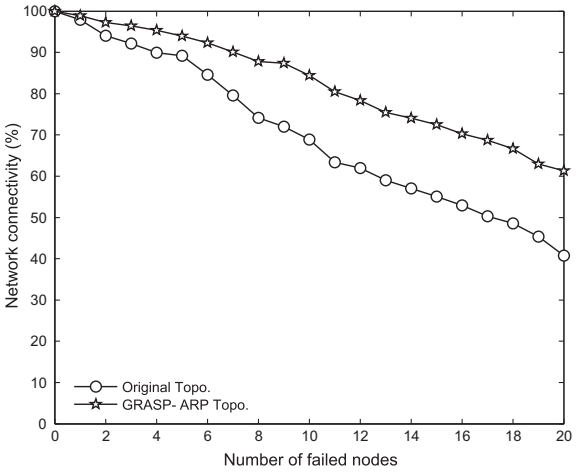

Fig. 18. Network connectivity with ER-MAC where a node dies every 250 s for the case of multiple sources – four sinks where $k = 3$.

Counting-Paths for $k = 2$. In all topologies, the sink is located at the top-left corner of the networks. In each experiment, we simulate a data gathering for 6000 s and kill one node every 1000 s, start from the 1000th s. Fig. 15 shows the ns-2 simulation results, where one failure causes both the original and GRASP-ARP topologies to lose around 2% of network connectivity. As the number of failures increases, the GRASP-ARP topologies improve over the original topology, maintaining 77% connectivity after 5 failures, while the original topology has dropped below 45%.

We then simulate a different set of 100-node topologies, and compare the topologies after failures for a connectivity value of $k = 3$. The results are shown in Fig. 16. The trend is similar to that for $k = 2$, with the GRASP-ARP topologies maintaining connectivity of 80% after 5 failures, while the original topology is again below 50%. The performance gap widens as we increase the failures, with GRASP-ARP maintaining 62% connectivity after 10 failures while the original topology has dropped to 30%.

8.2. Network topologies with four sinks

In the simulations with four sinks, we compare the original topologies, which are 100-node networks, and their resulting topologies for $k = 2$ generated by GRASP-ARP with the dynamic programming variant of Counting-Paths for the any-sinks cases. In all simulated topologies, we fix the locations to place the four sinks at the top-left, top-right, bottom-left, and bottom-right corners of the networks. In this experiment, we simulate data gathering for 3000 s only, because the period of one data gathering cycle in networks with many sinks is shorter than in the single sink problem. We increase the number of dead nodes by killing one node every 250 s. Fig. 17 shows that when the networks have multiple sinks, the topologies of GRASP-ARP achieve 20% higher network connectivity than the original topologies after the failure of 11 nodes. From this simulation set, we can infer that having many deployed sinks increases the robustness and scalability of the networks. We show this in the experiment by higher network connectivity compared to the single sink scenario.

Finally we extend the ns-2 simulations using a different set of 100-node topologies, and consider a connectivity value of $k = 3$. The results are shown in Fig. 18. After 20 failures, the connectivity of the original topology has dropped to 41%, while the GRASP-ARP topologies maintain connectivity of 61%.

9. Conclusion

Ensuring that WSNs are robust to failures requires that the physical network topology will offer alternative routes to the sink. This requires sensor network deployment to be planned with an objective of ensuring some measure of robustness in the topology, so that when failures do occur routing protocols can continue to offer reliable delivery. Our contribution is a solution that enables fault-tolerant WSN deployment planning by judicious use of additional relay nodes. We define the problem for increasing WSN

reliability by deploying a number of additional relays to ensure that each sensor node in the initial design has k disjoint paths with a length constraint to the sinks. We present two offline algorithms to be run during the initial topology planning to solve this problem. Counting-Paths counts the number of disjoint paths from each sensor node to the sinks and finds the k disjoint paths. GRASP-ARP is a local search algorithm to deploy a minimum number of additional relays at the possible candidate locations. We also adapt a version of the closest approach from the literature for comparison. Our simulation results show that our solution requires fewer relay nodes for larger problems than the competitor, and that different variants of our algorithm are significantly faster, allowing us to tackle larger problems. We also evaluate the robustness of the designs against node failures in simulation, where we demonstrate that the GRASP-ARP's topologies can provide robust delivery.

Acknowledgment

This research is funded by the Irish Higher Education Authority PRTLI-IV research program through the NEMBES project and by the Science Foundation Ireland through the CTVR project (SFI CSET 10/CE/I1853).

References

[1] A. Boukerche, R.W.N. Pazzi, R.B. Araujo, Fault-tolerant wireless sensor network routing protocols for the supervision of context-aware physical environments, J. Paral. Distrib. Comput. 66 (4) (2006) 586–599.
[2] O. Chipara, Z. He, G. Xing, Q. Chen, X. Wang, C. Lu, J. Stankovic, T. Abdelzaher, Real-time power-aware routing in wireless sensor networks, in: Proceedings of the 14th IEEE Workshop Quality of Service (IWQoS'06), 2006, pp. 83–92.
[3] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, P. Levis, Collection tree protocol, in: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09), 2009.
[4] Y. Zeng, C.J. Sreenan, L. Sitanayah, N. Xiong, J.H. Park, G. Zheng, An emergency-adaptive routing scheme for wireless sensor networks for building fire hazard monitoring, Sensors 11 (3) (2011) 2899–2919.
[5] D. Torrieri, Algorithms for finding an optimal set of short disjoint paths in a communication network, IEEE Trans. Commun. 40 (11) (1992) 1698–1702.
[6] T.A. Feo, M.G.C. Resende, Greedy randomized adaptive search procedures, J. Glob. Optim. 6 (1995) 109–133.
[7] L. Sitanayah, K.N. Brown, C.J. Sreenan, Fault-tolerant relay deployment for k node-disjoint paths in wireless sensor networks, in: Proceedings of the 4th International Conference on IFIP Wireless Days (WD'11), 2011, pp. 1–6.
[8] J.L. Bredin, E.D. Demaine, M. Hajiaghayi, D. Rus, Deploying sensor networks with guaranteed capacity and fault tolerance, in: Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'05), 2005, pp. 309–319.
[9] J. Pu, Z. Xiong, X. Lu, Fault-tolerant deployment with k-connectivity and partial k-connectivity in sensor networks, Wirel. Commun. Mob. Comput. 9 (7) (2008) 909–919.
[10] X. Han, X. Cao, E.L. Lloyd, C.C. Shen, Fault-tolerant relay node placement in heterogeneous wireless sensor networks, IEEE Trans. Mob. Comput. 9 (5) (2010) 643–656.
[11] M. Ahlberg, V. Vlassov, T. Yasui, Router Placement in Wireless Sensor Network, Tech. Rep. KTH/ICT/ECS, Royal Institute of Technology (KTH), Stockholm, Sweden, 2006.
[12] W. Zhang, G. Xue, S. Misra, Fault-tolerant relay node placement in wireless sensor networks: problems and algorithms, in: Proceedings of the 26th Annual IEEE Conference on Computer Communications (INFOCOM'07), 2007, pp. 1649–1657.
[13] S. Misra, S.D. Hong, G. Xue, J. Tang, Constrained relay node placement in wireless sensor networks to meet connectivity and survivability requirements, in: Proceedings of the 27th Annual IEEE Conference on Computer Communications (INFOCOM'08), 2008, pp. 281–285.
[14] B. Hao, J. Tang, G. Xue, Fault-tolerant relay node placement in wireless sensor networks: formulation and approximation, in: Proceedings of the Workshop High Performance Switching and Routing (HPSR'04), 2004, pp. 246–250.
[15] J. Tang, B. Hao, A. Sen, Relay node placement in large scale wireless sensor networks, Comp. Commun. 29 (4) (2006) 490–501.
[16] H. Liu, P. Wan, X. Jia, On optimal placement of relay nodes for reliable connectivity in wireless sensor networks, Combin. Optim. 11 (2) (2006) 249–260.
[17] A. Kashyap, S. Khuller, M. Shayman, Relay placement for fault tolerance in wireless networks in higher dimensions, Comput. Geom.: Theory Appl. 44 (4) (2011) 206–215.
[18] R. Bhandari, Optimal physical diversity algorithms and survivable networks, in: Proceedings of the 2nd IEEE Symposium on Computers and Communications (ISCC'97), 1997, pp. 433–441.
[19] L.R. Ford, D.R. Fulkerson, Flows in Networks, Princeton University Press, 1962.
[20] J. Kleinberg, E. Tardos, Algorithm Design, Addison-Wesley, 2011.
[21] T.A. Feo, M.G.C. Resende, A probabilistic heuristic for a computationally difficult set covering problem, Operat. Res. Lett. 8 (1989) 67–71.
[22] M.G.C. Resende, C.C. Ribeiro, Greedy randomized adaptive search procedures, in: F. Glover, G. Kochenberger (Eds.), State of the Art Handbook in Metaheuristics, Kluwer Academic Publishers, 2002, pp. 219–249.
[23] S.L. Martins, P.M. Pardalos, M.G.C. Resende, C.C. Ribeiro, Greedy randomized adaptive search procedures for the Steiner Problem in graphs, in: P.M. Pardalos, S. Rajasekaran, J. Rolim (Eds.), Randomization Methods in Algorithmic Design, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, vol. 43, American Mathematical Society, 1999, pp. 133–145.
[24] S. Wenming, H. Chuanhe, S. Mingkai, C. Yong, C. Zhe, Indoor localization scheme in wireless sensor networks using spatial information, in: Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM'06), 2006, pp. 1–5.
[25] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., The MIT Press, 2001.
[26] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice Hall, 1982.
[27] R. Ravi, D.P. Williamson, An approximation algorithm for minimum-cost vertex-connectivity problems, Algorithmica 18 (1) (1997) 21–43.
[28] The Network Simulator - ns-2 <http://www.isi.edu/nsnam/ns/> (30.04.10).
[29] Tmote Sky Datasheet <http://www.eecs.harvard.edu/konrad/projects/shimmer/references/tmote-sky-datasheet.pdf> (30.04.10).
[30] L. Sitanayah, C.J. Sreenan, K.N. Brown, ER-MAC: a hybrid MAC protocol for emergency response wireless sensor networks, in: Proceedings of the 4th International Conference on Sensor Technologies and Applications (SENSORCOMM'10), 2010, pp. 244–249.
[31] L. Sitanayah, C.J. Sreenan, K.N. Brown, A hybrid MAC protocol for emergency response wireless sensor networks, Ad Hoc Netw. 20 (2014) 77–95.

Lanny Sitanayah is a postdoctoral researcher in the Department of Computer Science at University College Cork, Ireland. She has a Ph.D. degree in Computer Science from University College Cork in 2013 and an M.Sc. degree in Computer Science from the University of Western Australia in 2009. Her research interests are in design, optimisation and test of real-time, adaptive and distributed systems; including wireless, mobile, ad hoc and sensor networks.

Kenneth N. Brown joined UCC Computer Science Department as a senior lecturer in 2003, where he is the Deputy Director of 4C: Cork Constraint Computation Centre. Prior to that he was a lecturer at the University of Aberdeen, a Research Fellow at Carnegie Mellon University, and a Research Associate at the University of Bristol. His research interests are in the application of AI, optimisation and distributed reasoning, with a particular focus on wireless networks.

Cormac J. Sreenan received the Ph.D. degree in Computer Science from Cambridge University. He is a professor of Computer Science at University College Cork (UCC) in Ireland. Prior to joining UCC in 1999, he was on the Research Staff at AT&T Labs Research, Florham Park, New Jersey, and at Bell Labs, Murray Hill, New Jersey. He is currently on the editorial boards of the IEEE Transactions on Mobile Computing, the ACM Transactions on Sensor Networks, and the ACM/Springer Multimedia Systems Journal. He is a fellow of the British Computer Society and a member of the IEEE and the ACM.