Probabilistic Connectivity of Underwater Sensor Networks

Md Asadul Islam

University of Alberta

mdasadul@ualberta.ca

August 16, 2014

Overview

Why UWSNs?

UWSNs fuelled by many important underwater sensing applications and services such as

- **Scientific applications**: e.g., observing geological processes on the ocean floor, determining water characteristics, counting or imaging animal life
- **Industrial applications**: e.g., monitoring and control of commercial activities, determining routes for underwater cables, monitoring underwater equipment and pipelines for oil and mineral extraction, and monitoring commercial fisheries
- **Military and homeland security applications**: e.g., monitoring and securing port facilities
- **Humanitarian applications**: e.g., search and survey missions, disaster prevention tasks, identification of seabed hazards, locating dangerous rocks or shoals, and identifying possible mooring locations

Challenges of the underwater communication channel

Radio Communication

- suffer strong attenuation in salt water
- short distances (6-20 m) and low data rates (1 Kbps)
- require large antennas and high transmission power

Optical Communication

- strongly scattered and absorbed underwater
- limited to short distances (40 m)

Acoustic Communication

- suffers from attenuation, spreading, and noise
- very long delay because of low propagation speed.
- it is most practical method upto now

Challenges due to node mobility

Static Deployment

- nodes attached to underwater ground, anchored buoys, or docks
-

Semi-mobile Deployment

- nodes attached to a free floating buoy
- subject to small scale movement

Mobile Deployment

- composed of drifters with self/noself mobile capability
- are subject to large scale movement
- maintaining connectivity is important to perform localization, routing etc.

Kinematic Model

We note that this area is new to networking researchers where the obtained analytical results are rooted in the mathematically deep field of fluid dynamics.

- A particle pathline is a path followed by an individual particle in a flow
- A *stream* function denoted by ψ measures the volume flow rate per unit depth.
- Curves where ψ is constant are called *streamlines*

The stream function can be presented

$$\psi(x, y, t) = -\tanh\left[\frac{y - B(t)\sin(k(x - ct))}{\sqrt{1 + k^2 B^2(t)\cos^2(k(x - ct))}}\right] + cy \tag{1}$$

where $B(t) = A + \epsilon\cos(\omega t)$ and the x and y velocities are given by

$$\dot{x} = -\frac{\partial\psi}{\partial y}; \dot{y} = \frac{\partial\psi}{\partial x} \tag{2}$$

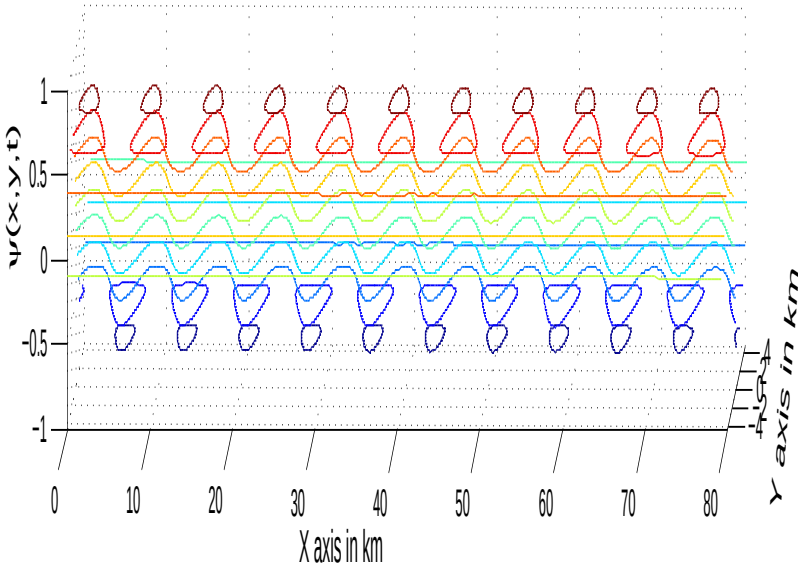


Figure 1: A 3D plot of 1

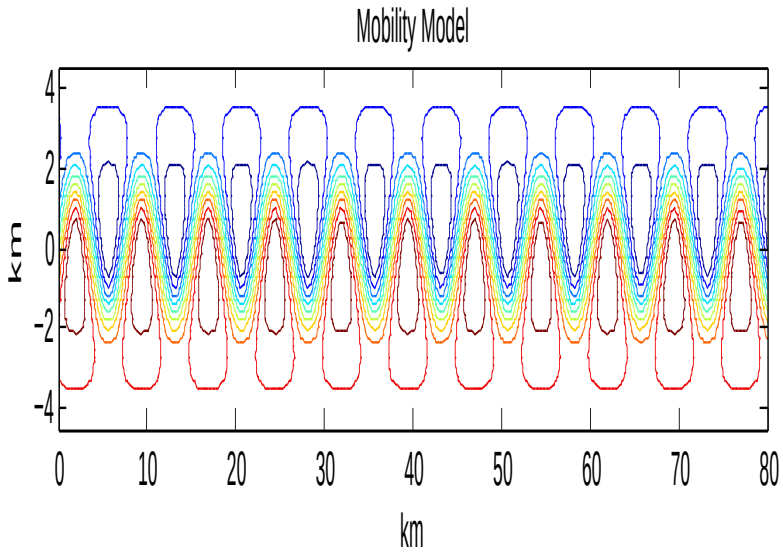


Figure 2: A plot of 1 at $t = 0$.

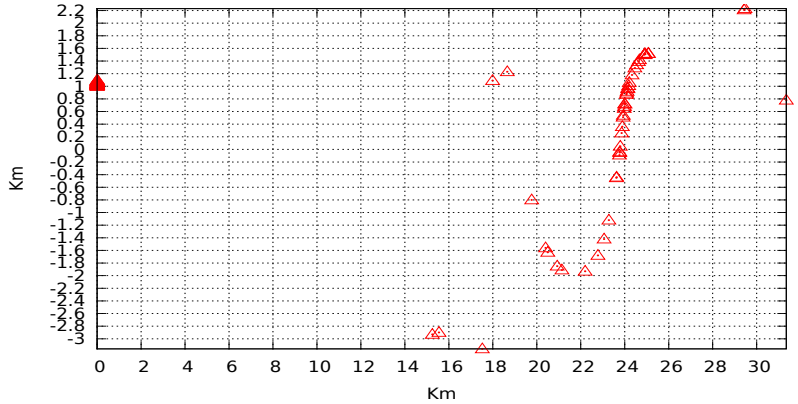


Figure 3: Start and end points of 50 nodes

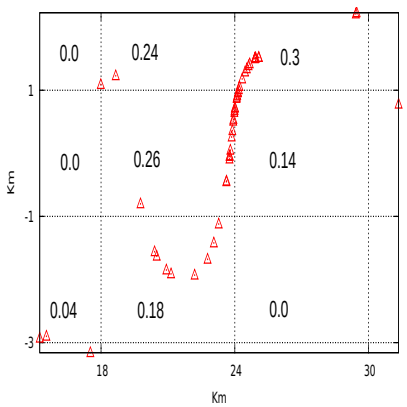


Figure 4: probabilistic distribution

Node Locality Sets

- $V = V_{sense} \cup V_{relay}$ the set of nodes in a given UWSN
  - V_{sense} denotes sensor nodes
  - V_{relay} denotes relay nodes.
  - V_{sense} has a distinguished *sink* node, denoted s,
- the geographic area considered rectangles of a superimposed grid layout.
- at time T, each node x can be in any one of a possible set of grid rectangles denoted $\mathrm{Loc}(x) = \{x[1], x[2], \ldots\}$.
- node x can be grid rectangle $x[i]$ with a certain probability $p_x(i)$.
- truncate some locality sets of low probability for convenience thus, $\sum_{x[i] \in \mathrm{Loc}(x)} p_x(i) \leq 1$, if $\mathrm{Loc}(x)$ is truncated.

Node Reachability

- node x can reach node y if the acoustic signal strength from x to y (and vice versa) exceeds a certain threshold value.
- if x and y can reach each other then $E_G(x, y) = 1$ otherwise $E_G(x, y) = 0$.
- we compute lower bounds on the likelihood that the network is totally, or partially, connected.
- we set $E_G(x[i], y[j]) = 1$ iff the two nodes x and y can reach each other if they are located anywhere in their respective rectangles $x[i]$ and $y[j]$.
- connectivity between x and y is ignored if they can reach each other at some (but not all) pairs of points in their respective rectangles.
- ignoring connectivity in such cases results in computing lower bounds on the network connectivity, as required.

Problem Definition

Definition (**the** *A-CONN* **problem**)

Given a probabilistic network G with no relay nodes, compute the probability $Conn(G)$ that the network is in a state where the sink node s can reach all sensor nodes. ■

Definition (**the** *AR-CONN* **problem**)

Given a probabilistic network G where V_{relay} is possibly non-empty, compute the probability $Conn(G)$ that the network is in a state where the sink node s can reach all sensor nodes. ■

Definition (**the** *S-CONN* **problem**)

Given a probabilistic network G with no relay nodes, and a required number of sensor nodes $n_{req} \leq |V_{sense}|$, compute the probability $Conn(G, n_{req})$ that the network is in a state where the sink node s can reach a subset of sensor nodes having at least n_{req} sensor nodes. ■

Problem Definition and Remarks

Definition (**the *SR-CONN* problem**)

Given a probabilistic network G where V_{relay} is possibly non-empty, and a required number of sensor nodes $n_{req} \leq |V_{sense}|$, compute the probability $Conn(G, n_{req})$ that the network is in a state where the sink node s can reach a subset of sensor nodes having at least n_{req} sensor nodes. ■

We note some of the above problems are special cases of other problems.

- A-CONN $\leq_p$ AR-CONN and S-CONN $\leq_p$ SR-CONN
- A-CONN $\leq_p$ S-CONN, since $n_{req} = |V_{sense}|$.
- Above problems share some basic aspects with the class of network reliability problems
- A probabilistic graphs arises when the given network is in some particular network states.
- A state S of V can be specified by $\{v_1[i_1], v_2[i_2], ..., v_n[i_n]\}$
- Assuming node locations are independent of each other, we have $Pr(S) = \prod_{v_\alpha \in V} p_{v_\alpha}[i_\alpha]$.

Example Probabilistic Network

Example

Figure 5 illustrates a probabilistic graph on 4 nodes where $V = \{s, a, b, c\}$. For the *A-CONN* problem, state $S_1 = \{s[2], a[2], b[2], c[2]\}$ is operating, and state $S_2 = \{s[1], a[1], b[1], c[2]\}$ is failed. ■

Given $G = (V, E_G, Loc, p)$ the *underlying graph* of G is a graph $\tilde{G}$ where

1. $V(\tilde{G}) = V$.
2. $E(\tilde{G})$ has an edge $e = (x, y)$ if for some positions $x[i]$ and $y[j]$ of nodes x and y, respectively, we have $E_G(x[i], y[j]) = 1$.

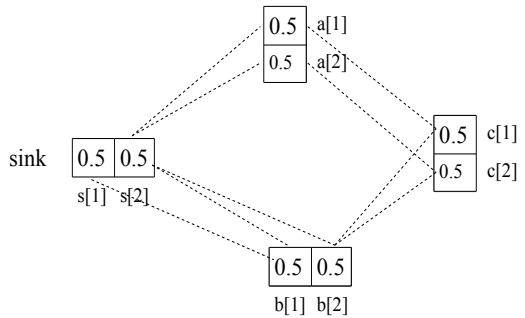


Figure 5: An example network

Example

The underlying graph of the network in figure 5 is the cycle (s, a, b, c). ■

k-Trees and Partial k-Trees

Definition

For a given integer $k \geq 1$, the class of k-trees is defined as follows

1. A k-clique is a k-tree.
2. If G_n is a k-tree on n nodes then the graph G_{n+1} obtained by adding a new node adjacent to every node in k-clique of G_n. ■

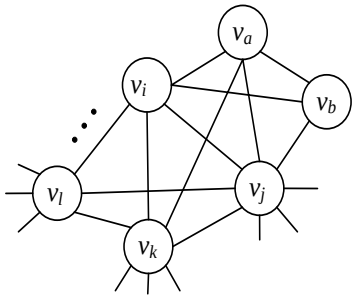


Figure 6: a fragment of a 3-tree

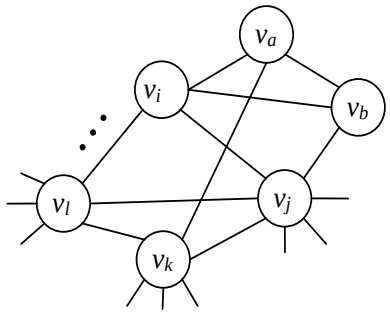


Figure 7: a fragment of partial 3-tree

A partial k-tree is a k-tree possibly missing some edges and a *k-perfect elimination sequence* (k-*PES*) of G is an ordering $(v_1, v_2, \ldots, v_r)$ of $v(G)$

Example

For the graph G in figure 7, $(v_a, v_b, v_i, v_j, v_k, v_l)$ is a 3-*PES*. ■

Graphs with Bounded Tree-width.

Definition

A *tree-decomposition* of a graph G is a family $(X_i : i \in I)$ of subsets of $V(G)$, together with a tree T with $V(T) = I$, with the following properties.

1. $\bigcup_{i \in I} X_i = V(G)$.
2. Every edge of G has both its ends in some X_i.
3. For $i, j, k \in I$, if j lies on the path of T from i to k then $X_i \cap X_k \subseteq X_j$. ■

The *width* of a tree-decomposition is $max(|X_i| - 1 : i \in I)$. The *tree-width* of G is the minimum $w \geq 0$ such that G has a tree-decomposition of width $\leq w$. ■

Tree Decomposition example

Example

Figure 8(a) illustrates a graph G having a tree-decomposition shown in figure 8(b). The width of the tree decomposition is 3. One may verify that 3 is actually the tree-width of G. ■

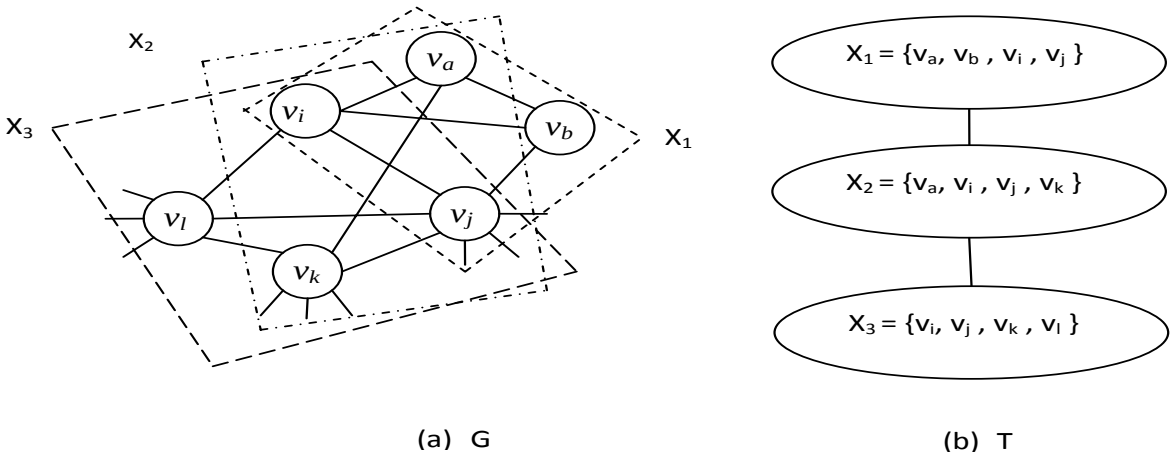


Figure 8: Tree decomposition

Dynamic Programing on Partial k-Trees

- Partial k-trees and graphs with bounded tree-width is rich with dynamic programming algorithms for solving many problems that are NP-complete in general.
- bodlander1988 and arnborg1991 formalizing graph and network properties as logical statements in a particular logic system.

A Steiner tree can be defined as follows.

Definition

Given a graph $G = (V, E)$ with positive integer edge costs, and a set of target nodes $V_{target} \subseteq V$, a Steiner tree, denoted $ST(G, V_{target})$, is a subtree $G' = (V', E')$ satisfying

1. $V_{target} \subseteq V' \subseteq V$,
2. the sum of edge costs in E' is minimum over all subtrees satisfying (1). ■

Solving Steiner tree problem on partial 2-tree

The algorithm devised in wald and colbourn 1983 works as follows. With each edge $\alpha = (x, y)$ of G, the algorithm associates six cost measures, which summarize the cost incurred so far of the subgraph S which has been reduced onto the edge (x, y)

1. $st(\alpha)$ is the minimum cost of a Steiner tree for S, in which x and y appear in the same tree.
2. $dt(\alpha)$ is the minimum cost of two disjoint trees for S including all targets, one tree involving x and the other y.
3. $yn(\alpha)$ is the minimum cost of a Steiner tree for S, which includes x but not y.
4. $ny(\alpha)$ is the minimum cost of a Steiner tree for S, which includes y but not x.
5. $nn(\alpha)$ is the minimum cost of a Steiner tree for S, which includes neither x nor y.
6. $none(\alpha)$ is the cost of omitting all vertices of S from a Steiner tree.

- An alternative approach to present the above algorithm is to store the six measures associated with edge $\alpha = (x, y)$ in a table, denoted $T_{x,y}$.
- The table provides key-value mappings.
- Roughly speaking, the keys replace the use of some of the names st, dt, yn, etc. with set notation.
- Not all names correspond to keys in this formulation.

Examples of names that correspond to keys are:
$st = \{x, y\}_1, dt = \{x\}_1\{y\}_1, yn = \{x\}_1\{y\}_0,$ and $ny = \{x\}_0\{y\}_1$.

Set Partitions

- Given a set X, a partition of X is a set $\{X_1, X_2, \ldots, X_r\}$, $1 \leq r \leq |X|$ such that
  1. $X = \bigcup_{i=1,2,\ldots,r} X_i$.
  2. The X_is are pairwise disjoint.
- In our algorithm, X is a set of nodes in a k-clique, and the partition of X are used as part of states of dynamic programs.
- The number of all possible partition of a set X on n elements are known as Bell numbers, denoted B_n.
- The first few Bell numbers are

$$B_0 = B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15,$$
$$B_5 = 52, B_6 = 203, B_7 = 877, B_8 = 4140, \ldots$$

- Bell numbers satisfy the following recurrence equation:

$$B_0 = 1, B_1 = 1 \text{ and } B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k.$$

Recognizing and Edge deletion problem on Partial k-tree

1. **Recognizing Partial k-Trees:** Given a graph G, and an integer $k \geq 1$, is G a partial k-tree?
   - The work of arnborg1987 shows $O(n^{k+2})$ algorithm for solving the problem.
   - If k is not fixed, then arnborg1987 shows that the problem is NP-complete.
   - For any fixed k, the work of bodlaender1993 improves on the above result by showing a linear time recognition algorithm.

   The above results produce also a k-*PES* if one exists.

2. **Edge Deletion Problem (EDP) to Obtain a Partial k-Tree:** Given a graph G, and an integer k, find the minimum number of edges whose deletion gives a partial k-tree.
   - For $k = 1$, the problem is simple.
   - For $k \geq 2$, the problem is NP-complete (From the work of elmallah1988 and the references therein)

The Random and Greedy Method.

The Random Method.

- **Case** $k = 1$**.** Choose any spanning tree.
- **Case** $k = 2$**.**
  - Choose a spanning tree $G' \subseteq G$.
  - Fix an ordering of the remaining edges not in G'.
  - For each edge e in the fixed order, test whether $G' + e$ is a partial 2-tree.
  - If yes, add e to G', and proceed to the next edge in the fixed order.
- **Case** $k = 3$**.**
  - Build a partial 2-tree $G' \subseteq G$.
  - Fix an ordering of the edges not in G'.
  - For each edge e, test whether $G' + e$ is a partial 3-tree.
  - If yes, add e to G', and proceed to the next edge in the fixed order.

The Greedy Method.

- **Case** $k = 1$**:** Choose a minimum spanning tree.
- **Case** $k = 2$ **and** 3**:** As in cases $k = 2$ and 3, respectively, of the random method, where we sort the remaining edges in a non-decreasing order of their costs to obtain the fixed order

Tree Network

- $type(x)$: $type(x) = 0$ and 1 if x is a relay node and a sensor node respectively.
- $n_{sense}(X)$: # of sensor nodes in a given subset of nodes $X \subseteq V$.
- $n_{relay}(X)$: # of relay nodes in a given subset of nodes $X \subseteq V$.
- $n(X) = n_{sense} + n_{relay}$.
- $n_{sense,min}(X)$: The minimum number of sensor nodes in a given subset $X \subseteq V$. So, $n_{sense,min}(X) = max(0, n_{req} - n_{sense}(\overline{X}))$

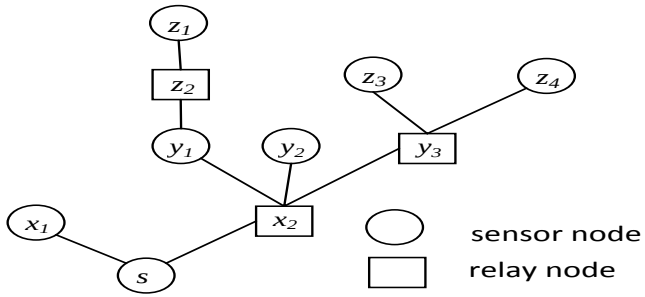


Figure 9: A tree network

Example

In figure 9, consider node x_2. $n(V_{x_2}) = 8$ where $n_{sense}(V_{x_2}) = 5$ and $n_{relay}(V_{x_2}) = 3$. Assuming $n_{req} = 5$ in an instance of the SR-CONN problem then $n_{sense,min}(V_{x_2}) = 3 = n_{req} - n_{sense}(\{s, x_1\}) = 5 - 2$. ■

Pseudo-code for function Conn

Function $\mathrm{Conn}(G, T, n_{req})$
Input: the *SR-CONN* problem where G has a tree topology T with no relay leaves
Output: $Conn(G, n_{req})$

1. **foreach** (node x and a valid location index i)
   set $R_x(i, type(x)) = 1$
2. **while** (T has at least 2 nodes)
   {
3. Let y be a non-sink leaf of T, and $x = parent(y)$
4. **foreach** (key $(i, count) \in R_y$) $R_y(i, count) *= p_y(i)$
5. set $R'_x = \phi$
6. **foreach** (pair of keys $(i_x, count_x) \in R_x$ and $(i_y, count_y) \in R_y$)
   {
7. $count = \min(n_{req}, count_x + count_y)$
8. **if** $(count < n_{sense,min}(\{x\} \cup V_y \cup_{z \in DCH(x)} V_z))$ **continue**
9. $R'_x(i_x, count) += R_y(i_y, count_y) \times R_x(i_x, count_x) \times E_G(x[i_x], y[i_y])$
   }
10. set $R_x = R'_x$; remove y from T
    }
11. return $\sum_{s[i] \in \mathrm{Loc}(s)} R_s(i, n_{req}) * p_s(i)$

Running time

Let n be the number of nodes in G, and ℓ_{max} be the maximum number of locations in the locality set of any node.

Theorem

Function Conn solves the $\mathrm{SR\text{-}CONN}$ problem in $O(n \cdot n_{req}^2 \cdot \ell_{max}^2)$ time

Proof. We note the following.

- Step 1: storing the tree T require $O(n)$ time.
- Step 2: the main loop performs $n-1$ iterations. Each of Steps 3, 5, and 10 can be done in constant time.
- Step 4: this loop requires $O(n_{req} \cdot \ell_{max})$ time.
- Step 6: this loop requires $O(n_{req}^2 \cdot \ell_{max}^2)$ iterations. Steps 7, 8, and 9 can be done in constant time.

Thus, the overall running time is $O(n \cdot n_{req}^2 \cdot \ell_{max}^2)$ time. ■

Theorem

Function Conn solves the $\mathrm{AR\text{-}CONN}$ problem in $O(n \cdot \ell_{max}^2)$ time

Cliques for A-CONN problem

- $K_{i,base}$: the k-clique to which node v_i is attached. For the 3-tree in figure 11, $K_{v_i,base}$ is the triangle (3-clique) on nodes $\{v_j, v_k, v_l\}$.
- $K_{v_i,1}, K_{v_i,2}, \ldots, K_{v_i,k}$: all possible k-cliques involving node v_i when this node becomes a k-leaf. For the 3-tree in the figure 11, we may set $K_{v_i,1}$ = the triangle (v_i, v_j, v_k), $K_{v_i,2}$ = the triangle (v_i, v_j, v_l), $K_{v_i,3}$ = the triangle (v_i, v_k, v_l).

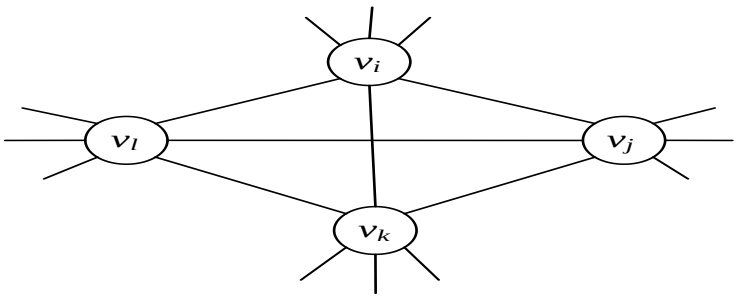


Figure 11: A fragment of a 3-tree

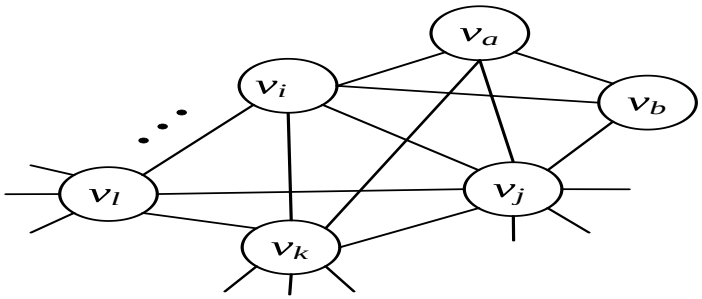


Figure 12: A 3-tree fragment

Example

In figure 12, if v_a and v_b have been deleted to make v_i a simplicial node, the information about the induced subgraph on nodes $(v_a, v_b, v_i, v_j, v_k)$ is summarized in table, say, $T_{v_i,1}$ associated with clique $K_{v_i,1} = (v_i, v_j, v_k)$. ■

State types for A-CONN problem

Definition (**state types of the** *A-CONN*)

Let $G_{v_i,\alpha}$ be a subgraph reduced onto the clique $K_{v_i,\alpha}$. Denote by $V_{v_i,\alpha}$ the set of nodes of the graph $G_{v_i,\alpha}$. Let $S = \{v_a[i_a] : v_a \in V_{v_i,\alpha}, i_a \in Loc(v_a)\}$ be a network state of $G_{v_i,\alpha}$. Then

$$A\text{-}type(S) = (V_1^{\mathrm{Loc}(V_1)}, V_2^{\mathrm{Loc}(V_2)}, \ldots, V_r^{\mathrm{Loc}(V_r)})$$

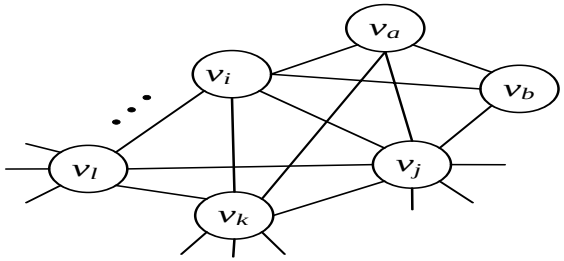


Figure 13: A 3-tree fragment

Example

In figure 12, denote by $G_{v_i,1}$ the graph induced on the set of nodes $\{v_a, v_b, v_i, v_j, v_k\}$. $G_{v_i,1}$ is reduced onto the clique $K_{v_i,1}$ = the triangle (v_i, v_j, v_k). Suppose that $S = \{v_a[1], v_b[2], v_i[1], v_j[2], v_k[3]\}$ is a possible network state of $G_{v_i,1}$. Moreover, suppose that S has 2 connected components : $\{v_a, v_b, v_i\}$ and $\{v_j, v_k\}$. Then state S is good for the A-CONN problem. State S induces the partition $(\{v_i\}, \{v_j, v_k\})$ on the triangle (v_i, v_j, v_k). Thus, $A\text{-}type(S) = (\{v_i\}^{(1)}, \{v_j, v_k\}^{(2,3)})$. ■

A sample Table

A table $T_{v_i,\alpha}$, where $\alpha = base, 1, 2, \ldots, k$ is key-value mapping where

- a key is a network state type of the graph $G_{v_i,\alpha}$
- each value is the probability.

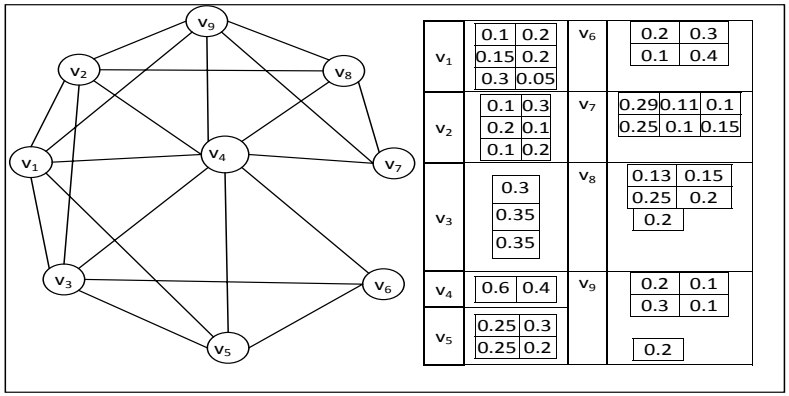


Figure 14: A UWSN modelled by a 3-tree

Example

Figure 14 illustrates a probabilistic network G on 9 nodes. The network has the topology of a 3-tree. The algorithm associates edge with each triangle a table. Each triangle can be partitioned in 5 different ways (since, the Bell number $B_3 = 5$). For example, the 5 partitions of triangle (v_1, v_2, v_3) are $\{v_1, v_2, v_3\}, \{v_1, v_2\}\{v_3\}, \{v_1, v_3\}\{v_2\}, \{v_2, v_3\}\{v_1\}$ and $\{v_1\}\{v_2\}\{v_3\}$. The number of possible keys that appear in the table of triangle (v_1, v_2, v_3) is $5 \times 6 \times 6 \times 3$ since $B_3 = 5, |Loc(v_1)| = 6$, $|Loc(v_2)| = 6$ and $|Loc(v_3)| = 3$. ■

The overall algorithm is organized around 3 functions :

- a top level main function (Algorithm 1),
- a middle level table merge function (Algorithm 2: *t_merge*), and
- a low level partition merge function (Algorithm 3: *p_merge*)

This function reduces the structure of the probabilistic network (and the underlying partial k-tree) G by iteratively processing and then deleting nodes according to the given *PES*, say $(v_1, v_2, \ldots, v_n)$ where $V_n = s$ (the sink node), until the network is reduced to the k-clique $(v_{n-k+1}, \ldots, v_n)$. Processing a node v_i entails merging the tables $\{K_{v_i,\alpha} : \alpha = base, 1, 2, \ldots, k\}$. This merging phase is done by merging a sequence of pairs of tables. The middle level table merge function is used for this purpose. After all iterations are done, the solution is computed from the table associated with the clique on nodes $(v_{n-k+1}, \ldots, v_n)$.

Function Main

Algorithm 1: Function Main(G, PES)

Input: An instance of the $A\text{-}CONN$ problem where G has a partial k-tree topology with a given PES=$(v_1, v_2, \ldots, v_n)$

Output: $Conn(G)$

Notation: $Temp$ is a temporary table.

1 **Initialization:** initialize a table T_H for each k-clique H of G.
T_H contains all possible state types on nodes of H.
2 **for** $(i = 1, 2, \ldots, |V| - k)$ **do**
3 $Temp = T_{v_i,1}$
4 **for** $(j = 2, 3, \ldots, k)$ **do**
5 $Temp = t\_merge(Temp, T_{v_i,j})$
end
6 $T_{v_i,base} = t\_merge(Temp, T_{v_i,base})$
7 **foreach** $(key \in T_{v_i,base})$ **do**
8 **if** (v_i *is a singleton part of* key) **then**
9 delete key from $T_{v_i,base}$
end
else
10 delete v_i and its associated position from key
end
end
end
11 **return** $Conn(G) = \sum$ values in table $T_{v_{n-k},base}$ corresponding to state types that have exactly one connected component

Function Table Merge

Algorithm 2: Function $t\_merge(T_1, T_2)$

Input: Two tables T_1 and T_2 that may share common nodes

Output: A merged table T_{out}

Initialization: Clear table T_{out}
set C = the set of common nodes between T_1 and T_2
foreach (*pair of state types* $key_1 \in T_1$ *and* $key_2 \in T_2$) **do**
    if (*any node in* C *lies in two different positions in* key_1 *and* key_2) **then**
        continue
    end
    set key_{out} = the state type obtained from node positions in key_1 and key_2, and the partition computed by $p\_merge(key_1, key_2)$
    set $p_{out} = T_1(key_1) \times T_2(key_2)$ adjusted to take the effect of common nodes in C into consideration
    if ($key_{out} \in T_{out}$) **then**
        update $T_{out}(k_{out})$ $+\!\!=p_{out}$
    end
    else
        set $T_{out}(key_{out}) = p_{out}$
    end
end
return T_{out}

Function Partition Merge

T_1 on (v_1, v_2, v_3)

.	.
$\{v_1, v_2\}^{(1,1)}\{v_3\}^{(2)}$	0.002
.	.
.	.
.	.

$\times$

T_2 on (v_1, v_3, v_4)

.	.
$\{v_1\}^{(1)}\{v_3, v_4\}^{(2,1)}$	0.008
.	.
.	.
.	.

$\Rightarrow$

$Temp$ on (v_1, v_2, v_3, v_4)

.	.
$\{v_1, v_2\}^{(1,1)}\{v_3, v_4\}^{(2,1)}$	0.0008
.	.
.	.
.	.

$T_{v_1,base}$ on (v_1, v_2, v_3, v_4)

.	.
$\{v_1, v_2\}^{(1,1)}\{v_3, v_4\}^{(2,1)}$	0.0008
.	.
.	.
.	.

$\Rightarrow$

$T_{v_1,base}$ on (v_2, v_3, v_4)

.	.
$\{v_2\}^{(1)}\{v_3, v_4\}^{(2,1)}$	0.0008
.	.
.	.
.	.

Function Partition Merge

$\boxed{\{v_1, v_2\}\{v_3\}} \times \boxed{\{v_1\}\{v_3, v_4\}} \Rightarrow \boxed{\{v_1, v_2\}\{v_3, v_4\}}$

Algorithm 3: function $p\_merge(P_1, P_2)$

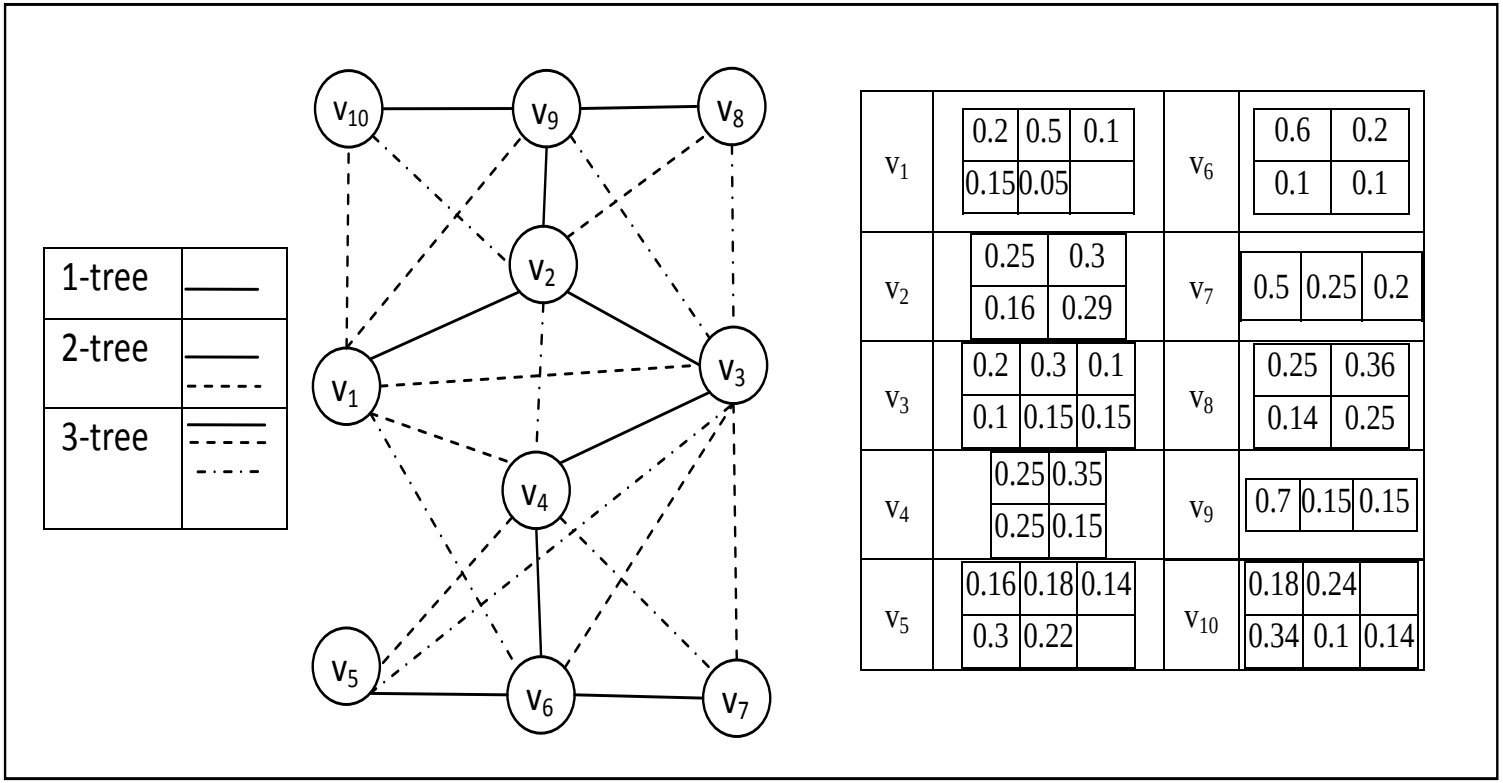
Input: Two partitions P_1 and P_2

Output: A partition P

Notation: s and t are two set iterators and their corresponding set are indicated by s^* and t^*.

```
1  foreach ( set s* in P2) do
2  |  P1.push_back(s*)
   end
3  for (s = P1.begin(); s ≠ P1.end(); ++s) do
4  |  for (t = s.next(); t ≠ P1.end(); ++t) do
5  |  |  if (s* ∩ t* ≠ ∅) then
6  |  |  |  P1.push_front(s* ∪ t*)
7  |  |  |  P1.delete(s*)
8  |  |  |  P1.delete(t*)
9  |  |  |  s = P1.begin()
10 |  |  |  break
   |  |  end
   |  end
   end
11 set P = P1
   return P
```

Test Networks



v_1	0.2	0.5	0.1	v_6	0.6 0.2 / 0.1 0.1
	0.15	0.05			
v_2	0.25	0.3		v_7	0.5 0.25 0.2
	0.16	0.29			
v_3	0.2	0.3	0.1	v_8	0.25 0.36 / 0.14 0.25
	0.1	0.15	0.15		
v_4	0.25	0.35		v_9	0.7 0.15 0.15
	0.25	0.15			
v_5	0.16	0.18	0.14	v_{10}	0.18 0.24 / 0.34 0.1 0.14
	0.3	0.22			

Figure 15: G_{10}

Test Networks(Cont.)

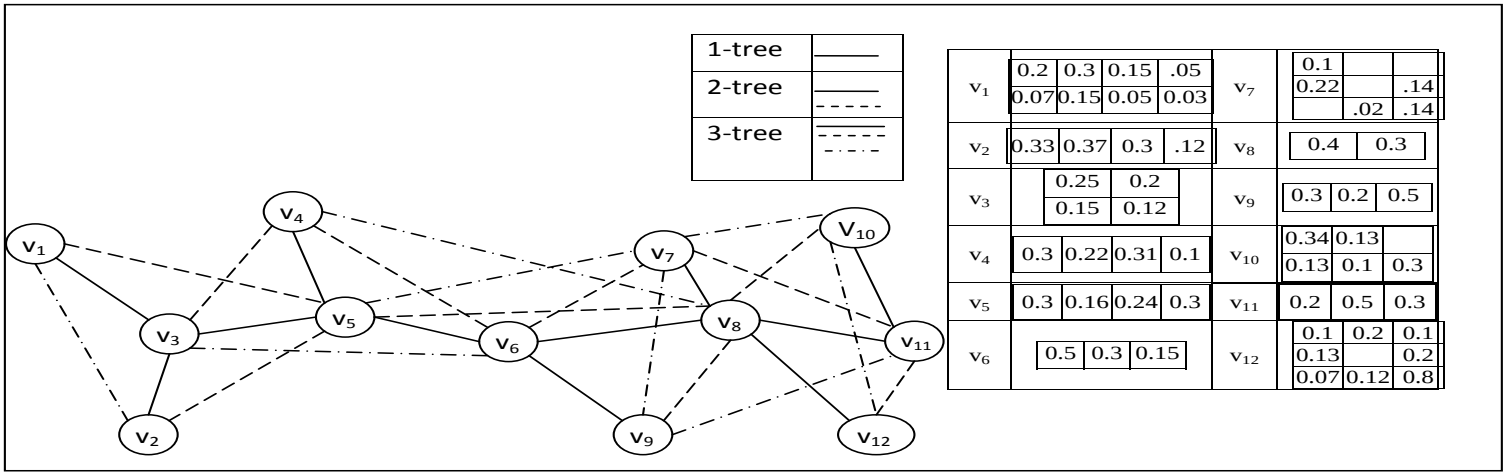


Figure 16: G_{12}

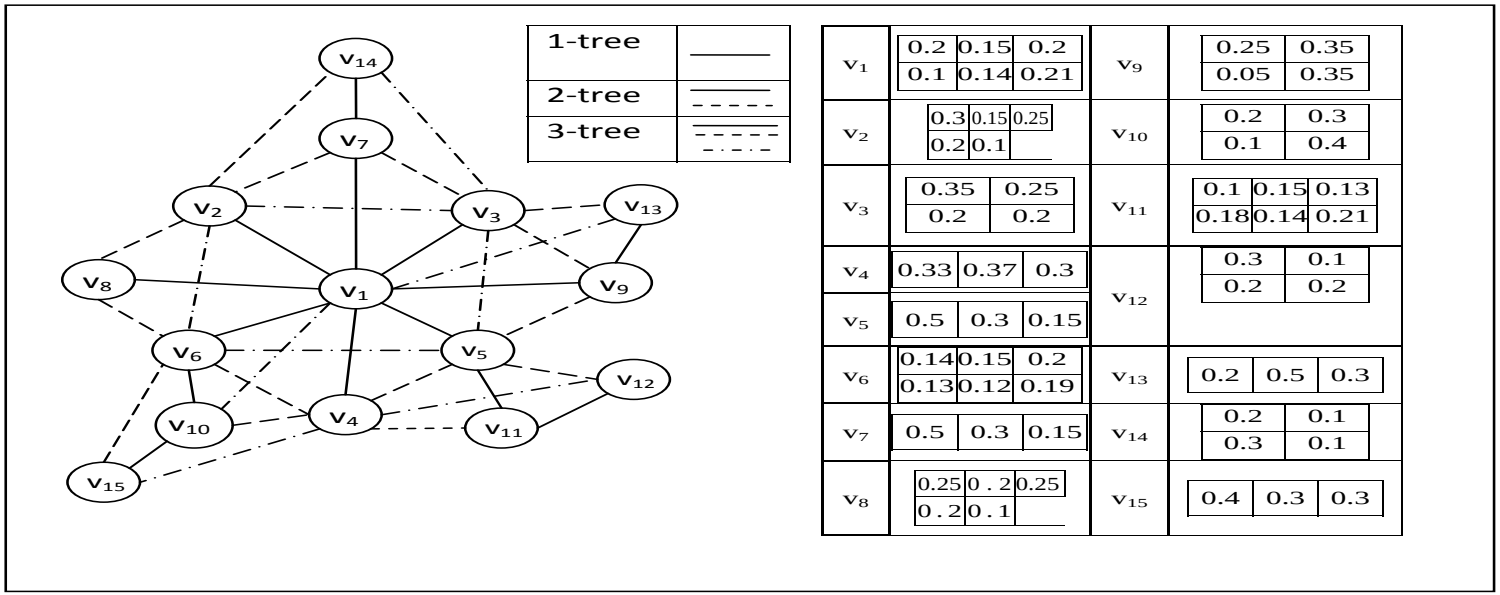


Figure 17: G_{15}

Running Time

k	Network G_{10}	Network G_{12}	Network G_{15}
1	90	130	200
2	1000	1380	1480
3	60000	875000	940000

Table 1: Running time in milliseconds

k	Network G_{10}	Network G_{12}	Network G_{15}
1	0.62	0.336	0.82
2	0.71	0.36	0.99
3	0.75	0.37	1

Table 2: Connectivity lower bounds using different partial k-trees

Effect of subgraph selection method

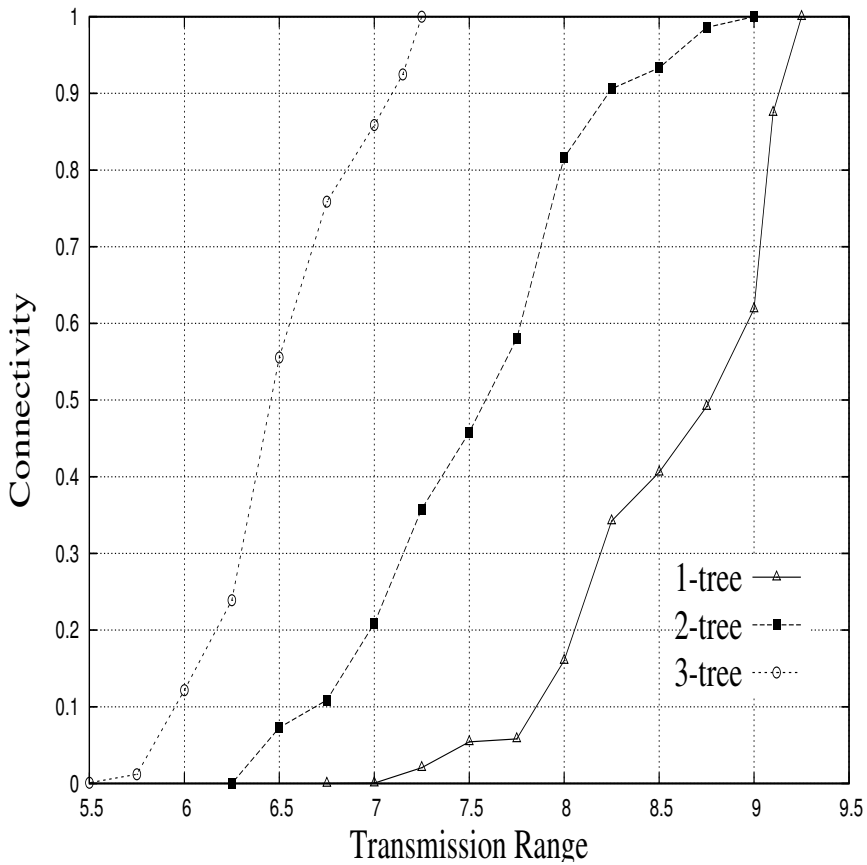


Figure 18: Connectivity versus transmission range with random subgraph selection

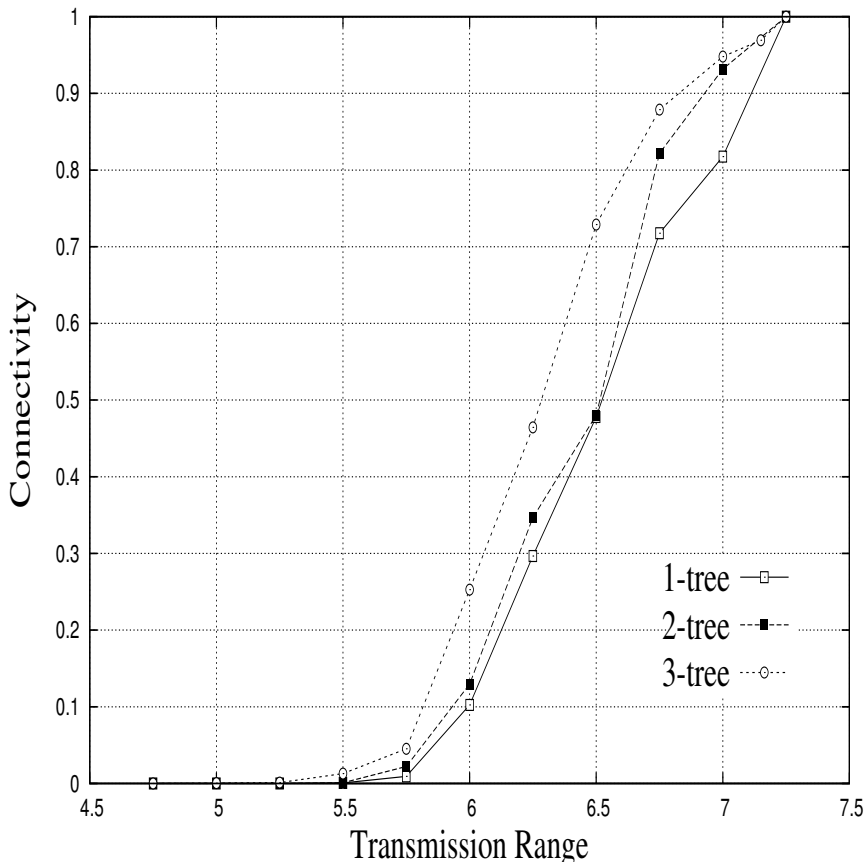


Figure 19: Connectivity versus transmission range with greedy subgraph selection