Manual Behaviour Algorithms

The three manual behaviours (and the force addition functions) are presented below as pseudo-code algorithms. It should be noted that 'send packet' and 'request packet' calls the networking transmission functions of the agent and 'move using forces' calls the locomotion functions of the agent.

Algorithm 1: Manual Behaviour 1: Basic relay

```
Input: neighbours: swarm agents and network-nodes
       within communication range
source: last known source network-node
sink: last know sink network-node
obstacle: relative location of closest obstacle

neighbours.sort(closeness);
for n ∈ neighbours do
    if n in direction of sink then
        n_sink← n;
        break;
for n ∈ neighbours do
    if n in direction of source then
        n_source← n;
        break;
if packet count < buffer*0.5 and n_source is source
 then
    request packet from n;
    end;
else if packet count > 0 and n_source not Null then
    send packet to n_source;
    end;
else
    force ← [0, 0];
    force ← add_attraction_force(force,n_source,1);
    force ← add_attraction_force(force,n_sink,1);
    if packet count == 0 then
        force ← add_attraction_force(force,source,5);
    else
        packet_neighbours ← [ n ∈ neighbours, if
          n.packet_count > 0];
        for n ∈ packet_neighbours do
            force ← ad-
              d_attraction_force(force,n,5/|packet_neighbours|);
    if norm(obstacle) < 2.0 then
        force ← add orbit(force, obstacle, True, 2);
    move using force;
    end;
```

Algorithm 2: Manual Behaviour 2: Obstacle avoiding

```
Input: neighbours: swarm agents and network-nodes
       within communication range
source: last known source network-node
sink: last know sink network-node
obstacle: relative location of closest obstacle to agent

neighbours.sort(closeness);
for n ∈ neighbours do
    if n in direction of sink then
        n_sink← n;
        break;
for n ∈ neighbours do
    if n in direction of source then
        n_source← n;
        break;
if packet count < buffer*0.5 and n_source is source
 then
    request packet from n;
    end;
else if packet count > 0 and n_source not Null
and n_source.signal_quality > 0.05 then
    /* signal_quality between -1:1, -1 =
       no connection, 1 = no power lost
       */
    send packet to n_source;
    end;
else
    force ← [0,0];
    force ← add_attraction_force(force,n_source,1);
    force ← add_attraction_force(force,n_sink,1);
    if norm(obstacle) < 5.0 then
        force ← add_repulsion_force(force, obstacle,
          2);
    if norm(obstacle)< 10.0 then
        /* orbit_direction is global
           variable                         */
        if orbit_direction == Null then
            orbit_direction←
              random(Clock,Anti_Clock);
        if (orbit_direction == Clock and
         !moved_in_last_step) or (orbit_direction ==
         Anti_Clock and moved_in_last_step) then
            force ← add orbit(force, obstacle, False,
              2);
            orbit_direction ← Anti_Clock;
        else
            force ← add orbit(force, obstacle, True,
              2);
            orbit_direction ← Clock;
    else
        orbit_direction ← Null;
    move using force;
    end;
```

Algorithm 3: Manual Behaviour 3: Jammer avoiding

```
Input: neighbours: swarm agents and network-nodes
       within communication range
source: last known source network-node
sink: last know sink network-node
obstacle: relative location of closest obstacle to agent
Output: Updated_force: tuple, magnitude and angle

neighbours.sort(closeness);
for n ∈ neighbours do
    if n in direction of sink then
        n_sink← n;
        break;
for n ∈ neighbours do
    if n in direction of source then
        n_source← n;
        break;
if packet count < buffer*0.5 and n_source is source
 then
    request packet from n;
    end;
else if packet count > 0 and n_source not Null then
    send packet to n_source;
    end;
else
    force ← [0,0];
    force ← add_attraction_force(force,n_source,1);
    force ← add_attraction_force(force,n_sink,1);
    if packet count == 0 then
        force ← add_attraction_force(force,source,5);
    else
        packet_neighbours ← [ n in neighbours, if n.
         packet_count>0]; for n ∈
         packet_neighbours do
            force ← ad-
             d_attraction_force(force,n,5/|packet_neighbours|);
    if  norm(obstacle) < 2.0 then
        force ← add orbit(force, obstacle, True, 2);
    for n ∈ neighbours do
        /* signal_quality between -1:1, -1
           = no connection, 1 = no
           signal-power lose              */
        if n is Jammer
        and n.signal_quality > -0.075 then
            force ← add_repulsion_force(force, n, 5);
            force ← add_orbit(force, n, 1);
    move using force;
    end;
```

Algorithm 4: add_attraction_force

```
Input: Old_force: existing force to add to
Target: object to create virtual force toward
Multiplier: Force magnitude multiplier
Output: Updated_force : tuple, magnitude and angle

if norm(target.relative_pos) < 1.0 then
    /* Too close, dont move towards     */
    return [0,0];
add_mag ← norm(target.relative_pos) * Multiplier;
add_ang ← atan2(target.relative_pos.y,
  target.relative_pos.x);
return add_force(add_mag, add_ang, Old_force);
```

Algorithm 5: add_orbit_force

```
Input: Old_force: existing force to add to
Target: object to create virtual force toward
Clockwise: obit in clockwise direction
Multiplier: Force magnitude multiplier
Output: Updated_force: tuple, magnitude and angle

add_mag ← norm(target.relative_pos) * Multiplier;
add_ang ← atan2(target.relative_pos.y,
  target.relative_pos.x);
if Clockwise then
    add_ang-=pi*0.5;
else
    add_ang+=pi*0.5;
return add_foce(add_mag, add_ang, Old_force);
```

Algorithm 6: add_repulsion_force

```
Input: Old_force: existing force to add to
Target: object to create virtual force toward
Multiplier: Force magnitude multiplier
Output: Updated_force: tuple, magnitude and angle

add_mag ← norm(target.relative_pos) * Multiplier;
add_ang ← atan2(target.relative_pos.y,
  target.relative_pos.x) + Pi;
return add_foce(add_mag, add_ang, Old_force);
```

```
Algorithm 7: add_force
Input: add_mag: magnitude of new force
add_ang: angle of new force
Old_force: tuple, magnitude and angle of existing
force
Output: Updated_force: tuple, magnitude and angle

old_mag ← Old_force[0];
old_ang ← Old_force[1];
new_mag ← sqrt( old_mag*old_mag
+ new_mag*new_mag
+ 2 * old_mag * new_mag * cos(new_ang - old_ang);
new_ang ← old_ang + atan2(new_mag * sin(new_ang
  - old_ang),
old_mag + new_mag * cos(new_ang - old_ang));
return [new_mag, new_ang];
```

Evolved behaviour Grammar

For evolving new behaviours for the swarm, the Grammar of Table 2 is utilised for rule-generation. Table 1 defines the variables and abbreviations used in this grammar. The process of decoding this grammar into rules is discussed in-depth in *Grammatical evolution : evolutionary automatic programming in an arbitrary language* by O'Neill, 2003.

Table 1: Grammar variable definitions

distance()	distance between target agent and self
move ()	call agent locomotion with direction and target
wait()	agent does nothing for the time-step
d_σ	constant. Estimated networking range. Set to 5 in this study
r	constant. Fixed distance. Set to 10 in this study
∃ network-node	Are any of the agent's neighbours a network node?
pac	number of packets held
away	move away from target
toward	move toward target
orbit	move perpendicular to target
cn	closest neighbour
source	last known network-node with packets to send
sink	last known network-node which packets were addressed to
srn	closest neighbour in direction of source
skn	closest neighbour in direction of sink
v	closest known jammer
o	closest known obstacle

Table 2: Grammar SLCS

Name	**Decoding Options**
$< Condition >$	$< distance >$; $< packet >$; $< neighbour\ type >$; $< Condition >$ & $< Condition >$
$< Action >$	$< transmission >$; $< move >$ wait()
$< distance >$	$(< distance\ type >) < op > (< distance\ type >)$
$< distance\ type >$	$distance(< target >)$; $< dist\ const. >$
$< op >$	$>$; $==$
$< dist\ const. >$	d_σ; r
$< packet >$	pac $= 0$; pac > 0
$< neighbour\ type >$	∃ network-node; ∄ network-node;
$< transmission >$	collect packet from source; send packet to $< tran.\ taret >$
$< move >$	move $(< direction >< target >)$
$< direction >$	away; toward orbit
$< target >$	*cn*; *source*; *sink* *srn*; *skn*; v; o'
$< tran.\ target >$	*cn*; *source*; *sink* *srn*; *skn*