Options	File (by build order)	Size
	Runtime.lib	22836
	Interface.lib	12812
D N V R	implement_mod.pas	954
D N V R	base_mod.pas	1272
D N V R	system_mod.pas	2266
D N V R	relations_mod.pas	5562
D N V R	action_mod.pas	9588
D N V R	plot_mod.pas	6302
D N V R	draw_mod.pas	4902
D N V R	transfer_mod.pas	4856
D N V R	simul.pas	9848
	Total Code Size	81198

```
unit implement_mod;

interface

 type
  string_t = string;

 type
  seed_t = longint;
  res_t = longint;
  trust_t = longint;

 function val (s: string_t): integer;

 function valr (s: string_t): real;

 function conv (i, f: integer): string_t;

 function convr (r: real; f, d: integer): string_t;

 function len (s: string_t): integer;

 function conc (s1, s2: string_t): string_t;

 function copystr (s: string_t; i, l: integer): string_t;

 procedure deletestr (var s: string_t; i, l: integer);

 function posstr (s, pattern: string_t): integer;

 procedure reset_file (var f: text; name: string_t);

 procedure rewrite_file (var f: text; name: string_t);


implementation

 function val (s: string_t): integer;
  var
   result: integer;
 begin
  readstring(s, result);
  val := result;
 end;

 function valr (s: string_t): real;
  var
   result: real;
 begin
  readstring(s, result);
  valr := result;
 end;

 function conv (i, f: integer): string_t;
 begin
  conv := stringof(i : f);
 end;
```

```
{$PUSH}
{$R-}
 function convr (r: real; f, d: integer): string_t;
 begin
  convr := stringof(r : f : d);
 end;
{$POP}

 function len (s: string_t): integer;
 begin
  len := length(s);
 end;

 function conc (s1, s2: string_t): string_t;
 begin
  conc := concat(s1, s2);
 end;

 function copystr (s: string_t; i, l: integer): string_t;
 begin
  copystr := copy(s, i, l);
 end;

 procedure deletestr (var s: string_t; i, l: integer);
 begin
  delete(s, i, l);
 end;

 function posstr (s, pattern: string_t): integer;
 begin
  posstr := pos(pattern, s);
 end;

 procedure reset_file (var f: text; name: string_t);
 begin
  reset(f, name);
 end;

 procedure rewrite_file (var f: text; name: string_t);
 begin
  rewrite(f, name);
 end;

end.
```

```
unit base_mod;

interface
 uses
  implement_mod;

 const
  max_clist = 10;

 type
  clist_t = record
     length: integer;
     entry: array[1..max_clist] of string_t;
    end;

 var
  seed, seed0: seed_t;    {random seed}

 function make_clist (s: string_t): clist_t;

 function bern (p: real): boolean;

 function random_integer (k: integer): integer;

 function normal (m, s: real): real;
  {Normal distribution generator adopted from Kreutzer using the rejection method.}
  {The two exponential distributions rnd1 and rnd2 are generated according to the}
  {inverse transform method. See Ross ch 5. The uniform distribution is taken from}
  {the funciton random given above.}


(*    ------------------------------------------------------------    *)

implementation
 function posi (s, substr: string_t; i: integer): integer;
  var
   result: integer;

 begin
  result := posstr(copystr(s, i, len(s) - i + 1), substr);
  if result = 0 then
   posi := 0
  else
   posi := result + i - 1;
 end;

 procedure trim (var s: string_t);
  var
   find: integer;
 begin
  repeat
   find := posstr(s, '  ');
   if find <> 0 then
    deletestr(s, find, 1);
  until find = 0;
 end;
```

```
function make_clist (s: string_t): clist_t;
 var
  i, p, p0: integer;
  clist: clist_t;
begin
 trim(s);
 for i := 1 to max_clist do
  clist.entry[i] := '';
 i := 0;
 clist.length := 0;
 if len(s) <> 0 then
  begin
   p := 1;
   p0 := 0;
   s := conc(s, ' ');
   while (p <= len(s)) and (p <> p0) do
    begin
     p0 := p;
     p := posi(s, ' ', p0);
     if p = 0 then
      p := p0
     else
      begin
       i := i + 1;
       clist.entry[i] := copystr(s, p0, p - p0);
       clist.length := i;
       p := p + 1;
      end;
    end;
  end;
 make_clist := clist;
end;



function random: real;
    {random number generating function, see Park and Miller, 1988}
 const
  a = 16807;
  m = 2147483647;
  q = 127773;
  r = 2836;
 var
  lo, hi, test: longint;
begin
 hi := seed div q;
 lo := seed mod q;
 test := a * lo - r * hi;
 if test > 0 then
  seed := test
 else
  seed := test + m;
 random := seed / m;
end;

function bern (p: real): boolean;
begin
```

```
    bern := random < p;
  end;

  function random_integer (k: integer): integer;
  begin
    random_integer := trunc(random * k) + 1;
  end;

  function normal (m, s: real): real;
    {Normal distribution generator adopted from Kreutzer using the rejection method.}
    {The two exponential distributions rnd1 and rnd2 are generated according to the}
    {inverse transform method. See Ross ch 5. The uniform distribution is taken from}
    {the funciton random given above.}

    var
      rnd1, rnd2: real;
  begin
    repeat{rejection loop}
      rnd1 := -ln(random); {generate the exponentials by inverse transformation}
      rnd2 := -ln(random);
    until (2 * rnd1) >= sqr(rnd2 - 1.0); {rejection test}
    if random < 0.5 then
      rnd2 := -rnd2;
    normal := s * rnd2 + m; {scale the acquired random variable}
  end;

end.
```

```
unit system_mod;

interface
 uses
  implement_mod, base_mod;

 const
  n_max = 400;
  buffer_rows = 41; {number of rows= nx*2+1}
  buffer_cols = 120;

 type
  id_t = 1..n_max;

  mode_t = (total, terr, pol, child);  {used in write_list}

  coord_t = integer;
  pos_t = record
     x, y: coord_t;
   end;

  act_t = (c, d);
  rel_t = (self, other);
  history_t = array[self..other] of act_t;
  culture_t = (prey, pred);

  list_type_t = (actor_list, child_list, neigh_list);

  id0_t = 0..n_max;

  list_t = ^entry_t;
  link_t = list_t;
  all_t = link_t;

  entry_t = record
     head: boolean;
     prev, next: list_t;

     id: id0_t;
     res: res_t;
     case boolean of
      false: (
         link: link_t;
         trust: trust_t;
         act: act_t;
         agent: id0_t;
         target: id0_t;
         history: history_t;
         dres: res_t;
         child: boolean;
         terr: boolean;
         pol: boolean;
         claim: boolean;
      );
      true: (
         threat: id0_t;
         members: list_t;
```

```
        n_members: integer;
        obligation: boolean;
      );
    end;

  actor_t = record
    pos: pos_t;
    res: res_t;
    culture: culture_t;
    parent: id0_t;
    children: integer;
    neighs: list_t;
    pol_neighs: integer;
    trust: trust_t;
    all, counter_all: all_t;
    threat: id0_t;
    reached: boolean;
  end;

(*  ------------------------------------------------------------  *)

 type
  system_t = array[id_t] of actor_t;
  locked_t = array[id_t] of boolean;

 var
  locked: locked_t;
  system: system_t;
  alliance_list: list_t;
  all_id: integer;
  n, nx, ny, sov_states, pred_states: integer;
  time: integer;

{params...}
  all_contribution: integer;

  event, alliances, first_system: boolean;


 procedure new_list (var result: list_t);

 procedure new_link (var link: link_t);

 function empty (list: list_t): boolean;

 function last (list: list_t): boolean;

 function add_link (list: list_t): link_t;

 function add_link_last (list: list_t): link_t;

 procedure delete_link (e: link_t);

 function list_length (list: list_t): integer;

 procedure write_list (list: list_t; mode: mode_t);
```

```
function find_link (list: list_t; id: id_t): list_t;

procedure delete_list (list: list_t);


function atom (actor: id_t): boolean;

function sov (actor: id_t): boolean;

function head (actor: id_t): boolean;

function subord (actor: id_t): boolean;

function siblings (i, j: id_t): boolean;

function compatriots (i, j: id_t): boolean;

function foreigners (i, j: id_t): boolean;

function adjacent (i, j: id_t): boolean;

function govt (i: id_t): id_t;

function aligned (i: id_t): boolean;


implementation
 procedure new_list (var result: list_t);
 begin
  new(result);
  with result^ do
   begin
    head := true;
    prev := nil;
    next := nil;
   end;
 end;


 procedure new_link (var link: link_t);
 begin
  new_list(link);
  link^.head := false;
 end;


 function empty (list: list_t): boolean;
 begin
  empty := list^.next = nil;
 end;

 function last (list: list_t): boolean;
 begin
  last := list^.next = nil;
 end;
```

```
function add_link (list: list_t): link_t;
 var
  new_entry:  list_t;
begin
 new(new_entry);
 with new_entry^ do
  begin
   head := false;
   prev := list;
   if empty(list) then
    next := nil
   else
    begin
     next := list^.next;
     next^.prev := new_entry;
    end;
   list^.next := new_entry;
   add_link := new_entry;
  end;
end;

function add_link_last (list: list_t): link_t;
 var
  new_entry, link: link_t;
begin
 link := list;
 if not empty(list) then
  repeat
   link := link^.next;
  until last(link);
 new(new_entry);
 new_entry^.head := false;
 link^.next := new_entry;
 new_entry^.prev := link;
 new_entry^.next := nil;
 add_link_last := new_entry;
end;


procedure delete_link (e: link_t);
begin
 with e^ do
  begin
   if last(e) then
    prev^.next := nil
   else
    begin
     prev^.next := next;
     next^.prev := prev;
    end;
  end;
 dispose(e);
end;

function list_length (list: list_t): integer;
 var
```

```
    l: link_t;
    result: integer;
  begin
   result := 0;
   l := list;
   if not empty(list) then
    repeat
     l := l^.next;
     result := result + 1;
    until last(l);
   list_length := result;
  end;

  procedure write_list (list: list_t; mode: mode_t);
   var
    p: list_t;
  begin
   p := list;
   if not empty(list) then
    begin
     repeat
      p := p^.next;
      if (mode = total) or ((mode = terr) and p^.terr) or ((mode = pol) and p^.pol) or ((mode = child) and p^.chilc
then
       write(p^.id : 3);
     until last(p);
    end;
  end;

  function find_link (list: list_t; id: id_t): list_t;
   var
    e: link_t;
    found: boolean;
  begin
   e := list;
   found := false;
   if not empty(list) then
    begin
     repeat
      e := e^.next;
      if e^.id = id then
       found := true;
     until found or last(e);
    end;
   if not found then
    e := nil;
   find_link := e;
  end;


  procedure delete_list (list: list_t);
  begin
   if not last(list) then
    delete_list(list^.next);
   dispose(list);
  end;
```

```
function atom (actor: id_t): boolean;
begin
 atom := (system[actor].children = 0);
end;

function sov (actor: id_t): boolean;
begin
 sov := (system[actor].parent = 0);
end;

function head (actor: id_t): boolean;
begin
 head := not atom(actor) and sov(actor);
end;

function subord (actor: id_t): boolean;
begin
 subord := (system[actor].parent <> 0);
end;


function siblings (i, j: id_t): boolean;
begin
 siblings := (system[i].parent <> 0) and (system[i].parent = system[j].parent);
end;

function compatriots (i, j: id_t): boolean;
begin
 compatriots := (i = j) or siblings(i, j) or (system[i].parent = j) or (system[j].parent = i);
end;

function foreigners (i, j: id_t): boolean;
begin
 foreigners := not compatriots(i, j);
end;

function adjacent (i, j: id_t): boolean;
{returns true if i is territorially adjacent to a member of j}
  var
    neigh: link_t;
    result: boolean;
 begin
  result := false;
  neigh := system[i].neighs;
  repeat
    neigh := neigh^.next;
    if neigh^.terr and compatriots(neigh^.id, j) then
      result := true;
  until last(neigh) or result;
  adjacent := result;
 end;

function govt (i: id_t): id_t;
begin
 if sov(i) then
```

```
      govt := i
    else
      govt := system[i].parent;
  end;

  function aligned (i: id_t): boolean;
  begin
    aligned := system[i].all <> nil;
  end;

end.
```

```
unit relations_mod;

interface
 uses
  implement_mod, base_mod, system_mod;

 var
  res_m, res_s: res_t;
  freq_pred: integer;

 procedure reset_link (i: id_t; link: link_t);

 function find_neigh (i, j: integer): link_t;

 procedure add_relation (i, j: id_t; var new_neigh_i: link_t);

 procedure set_terr (neigh: link_t; value: boolean);

 procedure reset_action (neigh: link_t);

 procedure set_pol (neigh: link_t; value: boolean);

 function counter_balance (i: id_t): id0_t;

 procedure delete_all (all: all_t);

 procedure add_member (all: all_t; i: id_t);

 procedure delete_member (all: all_t; m: link_t);

 procedure create_all (threat: id_t);

 procedure adjust_all (threat: id_t; all: all_t);

 procedure remove_member (i: id_t);

 procedure add_child (i, j: id_t);

 procedure remove_child (i, j: id_t);
{removes a child j from the neigh list of i by setting child bit to false}
{NOTE: entry still remains in neigh list}

 function within_range (x, y: coord_t): boolean;

 function index (x, y: coord_t): id_t;
{converts from x,y coordinates to index into system array}

 procedure new_system;

 procedure delete_system;


implementation

 procedure reset_link (i: id_t; link: link_t);
 begin
  with link^ do
```

```
    begin
      id := i;
      trust := 0;
      act := c;
      agent := 0;
      target := 0;
      history[self] := c;
      history[other] := c;
      res := 0;
      dres := 0;
      child := false;
      terr := false;
      pol := false;
      claim := false;
    end;
end;

function find_neigh (i, j: integer): link_t;
 var
   result: link_t;
begin
 find_neigh := find_link(system[i].neighs, j);
end;

procedure add_relation (i, j: id_t; var new_neigh_i: link_t);
 var
   new_neigh_j: link_t;
begin
 if i = j then
   begin
     new_neigh_i := nil;
     new_neigh_j := nil;
   end
 else
   begin
     new_neigh_i := find_neigh(i, j);
     if new_neigh_i = nil then
       begin
         new_neigh_i := add_link(system[i].neighs);
         new_neigh_j := add_link(system[j].neighs);
         reset_link(j, new_neigh_i);
         reset_link(i, new_neigh_j);
         new_neigh_i^.link := new_neigh_j;
         new_neigh_j^.link := new_neigh_i;
       end;
   end;
end;

procedure set_terr (neigh: link_t; value: boolean);
begin
 neigh^.terr := value;
 neigh^.link^.terr := value;
end;

procedure reset_action (neigh: link_t);
begin
 neigh^.act := c;
```

```
  neigh^.history[self] := c;
  neigh^.history[other] := c;
  neigh^.link^.act := c;
  neigh^.link^.history[self] := c;
  neigh^.link^.history[other] := c;
end;

procedure set_pol (neigh: link_t; value: boolean);
begin
  neigh^.pol := value;
  neigh^.link^.pol := value;
end;



function counter_balance (i: id_t): id0_t;
begin
  if system[i].all = nil then
    counter_balance := 0
  else
    counter_balance := system[i].all^.threat;
end;

procedure delete_all (all: all_t);
  var
    neigh: link_t;
begin
  if all <> nil then
    begin
      system[all^.threat].counter_all := nil;
      neigh := all^.members;
      if not empty(all^.members) then
        repeat
          neigh := neigh^.next;
          system[neigh^.id].all := nil;
        until last(neigh);
      delete_list(all^.members);
      delete_link(all);
    end;
end;

procedure add_member (all: all_t; i: id_t);
  var
    new_member: link_t;
begin
  if system[i].all <> nil then
    begin
      writeln('ERROR ADD ALL ', i);
    end;
  new_member := add_link(all^.members);
  new_member^.id := i;
  all^.res := all^.res + system[i].res;
  system[i].all := all;
  system[i].all^.n_members := system[i].all^.n_members + 1;
end;
```

```
procedure delete_member (all: all_t; m: link_t);
begin
 system[m^.id].all := nil;
 all^.res := all^.res - system[m^.id].res;
 delete_link(m);
 all^.n_members := all^.n_members - 1;
end;

procedure create_all (threat: id_t);
 var
  neigh, new_member: link_t;
  all: all_t;
  counter: id0_t;
begin
 all := add_link(alliance_list);
 all_id := all_id + 1;
 all^.id := all_id;
 new_list(all^.members);
 all^.threat := threat;
 all^.obligation := false;
 all^.res := 0;
 if system[threat].counter_all <> nil then
  begin
   writeln('ERROR ALL: ', threat); {###}
  end;
 system[threat].counter_all := all;
 neigh := system[threat].neighs;
 repeat
  neigh := neigh^.next;
  if neigh^.pol then
   begin
    counter := counter_balance(neigh^.id);
    if (system[neigh^.id].threat = threat) and not ((counter <> 0) and (counter <> threat)) then
     add_member(all, neigh^.id);
   end;
 until last(neigh);
end;

procedure adjust_all (threat: id_t; all: all_t);
 var
  neigh, prev_neigh, new_member: link_t;
begin
{First go through the alliance checking if members should be deleted}
 neigh := all^.members;
 repeat
  prev_neigh := neigh;
  neigh := neigh^.next;
  if (system[neigh^.id].threat <> threat) then {or not sov(neigh^.id)}
   begin
    delete_member(all, neigh);
    neigh := prev_neigh;
   end;
 until last(neigh);
{Then go through all the threatened neighbors checking if anyone should be added}
 neigh := system[threat].neighs;
 repeat
  neigh := neigh^.next;
```

```
      if neigh^.pol then
       if (system[neigh^.id].threat = threat) and (system[neigh^.id].all = nil) then
        add_member(all, neigh^.id);
     until last(neigh);
    end;

    procedure remove_member (i: id_t);
     var
      m: link_t;
      all: all_t;
    begin
     all := system[i].all;
     if all <> nil then
      begin
       m := find_link(all^.members, i);
       delete_member(all, m);
       if all^.n_members <= 1 then
        delete_all(all);
      end;
    end;

    procedure set_boundaries (j: id_t);
     var
      neigh: link_t;
      new_entry: link_t;
    begin
     neigh := system[j].neighs;
     repeat
      neigh := neigh^.next;
      set_pol(neigh, false);
      reset_action(neigh);
      if neigh^.terr then
       if siblings(j, neigh^.id) then
        begin
         set_pol(neigh, true);
        end
       else if foreigners(j, neigh^.id) then
        begin
         add_relation(system[j].parent, govt(neigh^.id), new_entry);
         set_pol(new_entry, true);
        end;
     until last(neigh);
    end;

    procedure add_child (i, j: id_t);
   {adds the child j to the neigh list of i if not already in}
     var
      new_child: link_t;
      dummy: integer;
    begin
     if i = j then
      begin
       dummy := 0;
       system[dummy].res := 0;
      end;
     locked[i] := true;
     locked[j] := true;
```

```
    sov_states := sov_states - 1;
    if system[j].culture = pred then
     pred_states := pred_states - 1;
    add_relation(i, j, new_child);
    new_child^.child := true;
    system[i].children := system[i].children + 1;
    system[j].parent := i;
    system[j].children := 0;
    system[i].res := system[i].res + system[j].res;
    system[j].res := 0;
    set_pol(new_child, false);
    reset_action(new_child);
    new_child^.act := c;
    new_child^.link^.act := c;
    set_boundaries(j);
    if alliances then
     begin
      if system[j].counter_all <> nil then
       delete_all(system[j].counter_all);
      if system[j].all <> nil then
       remove_member(j);
     end;
   end;

   procedure reset_boundaries (i, j: id_t);
    var
     neigh, new_entry: link_t;
     sov_neigh: id_t;

    procedure prepare_boundaries;
    begin
     neigh := system[j].neighs;
     repeat
      neigh := neigh^.next;
      set_pol(neigh, false);
     until last(neigh);
    end;

   begin
    prepare_boundaries;
    neigh := system[j].neighs;
    repeat
     neigh := neigh^.next;
     if neigh^.terr then
      begin
       if sov(neigh^.id) then
        begin
         set_pol(neigh, true);
         reset_action(neigh);
         sov_neigh := neigh^.id;
        end
       else
        begin
         sov_neigh := system[neigh^.id].parent;
         add_relation(j, sov_neigh, new_entry);
         set_pol(new_entry, true);
         reset_action(new_entry);
```

```
        end;
       if i <> sov_neigh then {add new foreign policy relation to head}
        begin
         add_relation(i, sov_neigh, new_entry);
         set_pol(new_entry, adjacent(neigh^.id, i));
         reset_action(new_entry);
        end;
      end;
   until last(neigh);
 end;

 procedure remove_child (i, j: id_t);
{removes a child j from the neigh list of i by setting child bit to false}
{NOTE: entry still remains in neigh list}
   var
    old_child: link_t;
    dres: res_t;
 begin
   locked[i] := true;
   locked[j] := true;
   sov_states := sov_states + 1;
   if system[j].culture = pred then
    pred_states := pred_states + 1;
   system[j].parent := 0;
   old_child := find_neigh(i, j);
   dres := system[i].res div (system[i].children + 1);
{reclaim resources}
   system[i].res := system[i].res - dres;
   system[j].res := dres;
   old_child^.child := false;
   system[i].children := system[i].children - 1;
   reset_action(old_child);
   reset_boundaries(i, j);
 end;



 function within_range (x, y: coord_t): boolean;
 begin
   within_range := (x >= 1) and (x <= nx) and (y >= 1) and (y <= ny);
 end;

{$PUSH}
{$R-}
 function index (x, y: coord_t): id_t;
{converts from x,y coordinates to index into system array}
 begin
   index := (x - 1) * nx + y;
 end;
{POP}

 procedure make_neigh (i, x, y: integer);
   var
    neigh: link_t;
    j: id_t;
 begin
   if within_range(x, y) then
```

```
      begin
       j := index(x, y);
       if i < j then
        begin
         add_relation(i, j, neigh);
         set_terr(neigh, true);
         set_pol(neigh, true);
         reset_action(neigh);
        end;
      end;
    end;

    procedure new_neighbors (var system: system_t);
     var
      x, y, i: integer;
    begin
     i := 0;
     for x := 1 to nx do
      for y := 1 to ny do
       begin
        i := i + 1;
        make_neigh(i, x + 1, y);
        make_neigh(i, x - 1, y);
        make_neigh(i, x, y + 1);
        make_neigh(i, x, y - 1);
       end;
    end;

    procedure assign_culture (freq_pred: integer);
     var
      i, j, n_pred, current_pred, select: id0_t;
      link: link_t;
      list: list_t;
    begin
     n_pred := trunc(freq_pred * n / 100);
     pred_states := n_pred;
     current_pred := 0;
     if n_pred = 0 then
  {nothing}
      else if n_pred = n then
       for i := 1 to n do
        system[i].culture := pred
      else
       begin
        new_list(list);
        for i := 1 to n do
         begin
          link := add_link(list);
          link^.id := i;
         end;
        repeat
         select := random_integer(n - current_pred);
         j := 0;
         link := list;
         repeat
          link := link^.next;
          j := j + 1;
```

```
      until last(link) or (j = select);
      system[link^.id].culture := pred;
      delete_link(link);
      current_pred := current_pred + 1;
     until current_pred = n_pred;
     delete_list(list);
    end;
  end;


  procedure new_system;
   var
    x, y, i: integer;
  begin
   seed0 := seed;
   sov_states := nx * ny;
   pred_states := 0;
   new_list(alliance_list);
   i := 0;
   time := 0;
   all_id := 0;
   for x := 1 to nx do
    begin
     for y := 1 to ny do
      begin
       i := i + 1;
       with system[i] do
        begin
         pos.x := x;
         pos.y := y;
         if res_s <> 0 then
          res := round(normal(res_m, res_s))
         else
          res := res_m;
         culture := prey;
         parent := 0;
         children := 0;
         threat := 0;
         all := nil;
         counter_all := nil;
         new_list(neighs);
        end;
      end;
    end;
   assign_culture(freq_pred);
   new_neighbors(system);
  end;

  procedure delete_system;
   var
    i: integer;
  begin
   for i := 1 to n do
    begin
     delete_list(system[i].neighs);
    end;
   delete_list(alliance_list);
```

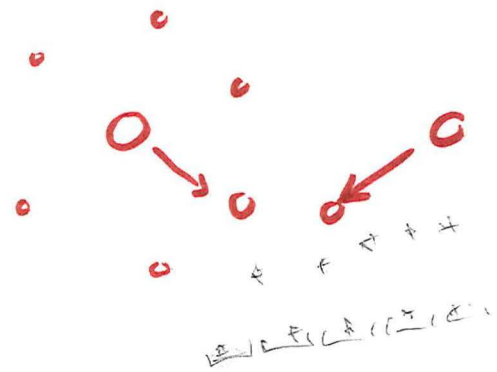
```
    end;

  end.
```

```
unit action_mod;
{July 12, 1993: dividing the decide function for each culture type}
{see action_mod_old.pas file}

interface
 uses
  implement_mod, base_mod, system_mod, relations_mod;

 const
  d_trust = 1000;
  offset_trust = 1000;

 var
  harvest_m, harvest_s, min_res: res_t;
  bell_pred: integer;
  superior_ratio, victory_ratio, grad, mobil: integer;
  peace_div, victory_cost, defeat_cost, war_cost: res_t;
  min_threat: res_t;
  pr_oblig: integer;
  forgive, provoke: integer;
  soc_neg, soc_pos: trust_t;

 procedure perceptions;

 procedure decisions;

 procedure interactions;

 procedure update_resources;

 procedure update_all_resources;

 procedure collapse (j: id_t);

 procedure conquer (i, j: id_t);

 procedure structural_change;

 procedure diplomacy;



(*  ------------------------------------------------------------  *)

implementation

 type
  random_buffer_t = array[id_t] of id0_t;

 function random_act (p: real): act_t;
 begin
  if bern(p) then
   random_act := d
  else
   random_act := c;
 end;
```

```
 procedure update_trust_OLD (i: id_t);
  var
   neigh: link_t;
 begin
  neigh := system[i].neighs;
  repeat
   neigh := neigh^.next;
   if neigh^.pol then
    if neigh^.res = 0 then
     neigh^.trust := 0
    else
     neigh^.trust := -(neigh^.link^.res div neigh^.res);
  until last(neigh);
 end;

 procedure update_threat (i: id_t);
  var
   neigh: link_t;
   trust, min_trust: longint;
   threat: id0_t;
   pol_neighs: integer;
 begin
  threat := 0;
  min_trust := 999999999;
  system[i].trust := 0;
  pol_neighs := 0;
  neigh := system[i].neighs;
  repeat
   neigh := neigh^.next;
   if neigh^.pol then
    begin
     pol_neighs := pol_neighs + 1;
     if alliances then
      if system[i].all <> nil then {need assistance?}
       if system[i].all^.threat = neigh^.id then
        if (neigh^.history[other] = d) then
         begin
          system[i].all^.obligation := true;
{writeln('CALL FOR HELP: ', i);}
         end;
     trust := neigh^.trust; {* res prop to res anyway}
     system[i].trust := system[i].trust + trust;
     if trust < min_trust then
      begin
       threat := neigh^.id;
       min_trust := trust;
      end;
    end;
  until last(neigh);
{update threat}
  system[i].pol_neighs := pol_neighs;
  if min_trust < min_threat then
   system[i].threat := threat
  else
   system[i].threat := 0;
```

```
  if (soc_neg <> 0) or (soc_pos <> 0) then
   if system[i].trust < soc_neg * pol_neighs then
    if system[i].culture = prey then
     begin
      system[i].culture := pred;
      pred_states := pred_states + 1;
     end
    else if system[i].trust > soc_pos * pol_neighs then
     if system[i].culture = pred then
      begin
       system[i].culture := prey;
       pred_states := pred_states - 1;
      end;
 end;

 procedure perceptions;
  var
   i: id_t;
 begin
  for i := 1 to n do
   if sov(i) then
    begin
{update_trust(i);}
 {use res_i and res_j instead}
     update_threat(i);
    end;
 end;

(* ------------------------------------------------------------------ *)

 function valid (i: integer): boolean;
 begin
  valid := (i > 0) and (i <= n);
 end;

 function select_agent (i, j: id_t): id0_t;
  var
   k: id0_t;
   buffer: random_buffer_t;
   agentp: link_t;
   result: id0_t;
 begin
  result := 0;
  if atom(i) then
   result := i
  else
   begin
    if adjacent(i, j) then
     begin
      k := 1;
      buffer[1] := i;
     end
    else
     k := 0;
    agentp := system[i].neighs;
    repeat
     agentp := agentp^.next;
```

```
{select agent adjacent to corp(j)}
      if agentp^.child then
       if adjacent(agentp^.id, j) then
        begin
         k := k + 1;
         buffer[k] := agentp^.id;
        end;
     until last(agentp);
     if k <> 0 then
      result := buffer[random_integer(k)];
   end;
  if not valid(result) then
   writeln('ERROR IN SELECT AGENT ', i : 4, j : 4, '=', result);
  select_agent := result;
end;
```



```
function select_target (agent, j: id_t): id0_t;
 var
  k: id0_t;
  buffer: random_buffer_t;
  targetp: link_t;
  result: id0_t;
begin
 if atom(j) then
  result := j
 else
  begin
   result := 0;
   k := 0;
   targetp := system[agent].neighs;
   repeat
    targetp := targetp^.next;
    if targetp^.terr then
     if compatriots(targetp^.id, j) then  {problem, compatriot called with illegal indices}
      begin
       k := k + 1;
       buffer[k] := targetp^.id;
      end;
   until last(targetp);
   if k <> 0 then
    result := buffer[random_integer(k)];
  end;
 select_target := result;
end;
```

```
function ext_res (i, threat: id_t): res_t;
begin
 if aligned(i) then
  if system[i].all^.threat = threat then
   ext_res := trunc((1.0 - all_contribution / 100.0) * system[i].res + all_contribution * system[i].all^.res / 1(
  else
   ext_res := system[i].res
 else
  ext_res := system[i].res;
end;
```

```
 function balance (i, j: id_t): real;
{returns true if superior, res_j is total res of j regardless of allocation}
{alliance resources should be taken into consideration}
  var
   own_res, other_res: res_t;
 begin
  if alliances then
   begin
    own_res := ext_res(i, j);
    other_res := ext_res(j, i);
   end
  else
   begin
    own_res := system[i].res;
    other_res := system[j].res;
   end;
  balance := own_res / other_res;
end;

function capita_res (i: id_t): res_t;
begin
 capita_res := system[i].res div (system[i].children + 1);
end;

function binary_act (a: act_t): res_t;
begin
 if a = d then
  binary_act := 1
 else
  binary_act := 0;
end;

function peace (neigh: link_t): boolean;
begin
 peace := (neigh^.history[self] = c) and (neigh^.history[other] = c);
end;

function distance (i, j: id_t): real;
 var
  dist: pos_t;
begin
 dist.x := (system[i].pos.x - system[j].pos.x);
 dist.y := (system[i].pos.y - system[j].pos.y);
 distance := sqrt(dist.x * dist.x + dist.y * dist.y);
end;

function battle_res (i, agent: id_t; res: res_t): res_t;
 var
  c: real;
begin
 if (grad = 100) or atom(i) then
  c := 1
 else
  begin
   c := exp(distance(i, agent) * ln(grad / 100.0));
  end;
```

```
    battle_res := round(c * res);
  end;

  procedure decide_prey (i: id_t; var attacks: integer);
    var
      neigh: link_t;
  begin
    attacks := 0;
    neigh := system[i].neighs;
    repeat
      neigh := neigh^.next;
      if neigh^.pol then
        begin
          neigh^.act := neigh^.history[other]; {TFT rule}
          if alliances then
            if system[i].all <> nil then
              if (system[i].all^.threat = neigh^.id) and system[i].all^.obligation then
                if bern(pr_oblig / 100.0) then
                  begin
                    neigh^.act := d; {assistance to alliance partners}
                    if peace(neigh) then
                      begin
                        neigh^.agent := select_agent(i, neigh^.id);
                        neigh^.target := select_target(neigh^.agent, neigh^.id);
                      end;
{WRITELN('OBLIG FULFILLED', i, neigh^.agent, neigh^.target);}
                    if (neigh^.history[self] = d) or (neigh^.history[other] = d) then
                      attacks := attacks + 1;
                  end;
        end;
    until last(neigh);
  end;


  procedure decide_pred (i: id_t; var attacks: integer);
    var
      neigh, weakest: link_t;
      rest: longint;
      max, ratio: real;
  begin
    weakest := nil;
    max := -999999999;
    attacks := 0;
    neigh := system[i].neighs;
    repeat
      neigh := neigh^.next;
      if neigh^.pol then
        begin
          neigh^.act := neigh^.history[other]; {TFT rule}
          if (system[i].culture = pred) and (neigh^.history[self] = d) then
            neigh^.act := d; {unforgiving and exploitative}
          if alliances then
            if system[i].all <> nil then
              if (system[i].all^.threat = neigh^.id) and system[i].all^.obligation then
                begin
                  neigh^.act := d; {assistance to alliance partners}
```

```
            if peace(neigh) then
              begin
                neigh^.agent := select_agent(i, neigh^.id);
                neigh^.target := select_target(neigh^.agent, neigh^.id);
              end;
{WRITELN('OBLIG FULFILLED', i, neigh^.agent, neigh^.target);}
          end;
        if (neigh^.history[self] = d) or (neigh^.history[other] = d) then
          attacks := attacks + 1;
        ratio := balance(i, neigh^.id);
{res := ext_res(neigh^.id, i) - battle_res(i, neigh^.id, system[i].res);}
        if ratio > max then
          begin
            weakest := neigh;
            max := ratio;
          end;
      end;
    until last(neigh);
{predators opportunistically make an unprovoked attack against weakest neigh}
    if (system[i].culture = pred) and (weakest <> nil) and (attacks = 0) then
      begin
        if bern(bell_pred / 100.0) and (max > superior_ratio / 100.0) then
          begin
            weakest^.agent := select_agent(i, weakest^.id);
            weakest^.target := select_target(weakest^.agent, weakest^.id);
            weakest^.act := d;
            attacks := 1;
          end;
      end;
  end;

  procedure allocate_res (i: id_t; attacks: integer);
    var
      neigh: link_t;
      share: real;
      available_res: res_t;
  begin
{if attacks > 0}
    available_res := system[i].res - (1 + system[i].children) * min_res;
    neigh := system[i].neighs;
    repeat
      neigh := neigh^.next;
      if (neigh^.pol) then {and (neigh^.act = d)}
        begin
          if capita_res(i) >= min_res then
            begin
              neigh^.res := round(((1 + (mobil / 100.0) * binary_act(neigh^.act)) * available_res) / (system[i].pol_neig
(mobil / 100.0) * attacks));
            end
{share := (offset_trust + d_trust - neigh^.trust) / (system[i].pol_neighs * (offset_trust + d_trust) - system[i].tr
{neigh^.res := trunc(share * available_res)}
{NOTE: the factor 2 in the denominator #REAL}
          else
            neigh^.res := 0;
        end
    until last(neigh);
  end;
```

Choose path once ∀ and base subsequent calculation

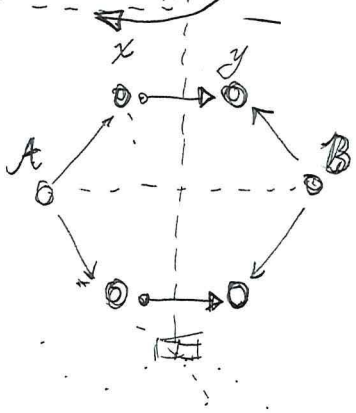
Problem:
- target/agent not chosen
- how calculate distance?

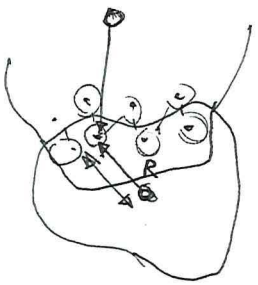
battle-field

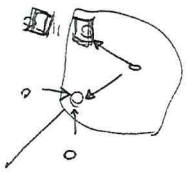
local balance:
victim := buffer[random_integer[k]]

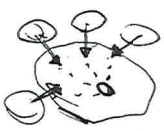
victim

add pull out









```
procedure decide (i: id_t);
 var
  attacks: integer;
begin
 if system[i].culture = prey then
  decide_prey(i, attacks)
 else
  decide_pred(i, attacks);
 allocate_res(i, attacks);
end;

procedure decisions;
 var
  i: id_t;
begin
 if sov_states > 1 then
  for i := 1 to n do
   if sov(i) then
    decide(i);
end;
```

Laws of Interaction

```
function attack (neigh_i, neigh_j: link_t): boolean;
begin
 attack := (neigh_i^.act = d) and (neigh_j^.act = d) and (neigh_i^.history[self] = c) and (neigh_j^.history[self] =
end;

function win_battle (i, j: id_t; res_i, res_j: res_t): boolean;
begin
 win_battle := res_i > res_j * (victory_ratio / 100.0);
end;

function in_same_alliance (i, j: id_t): boolean;
 var
  a1, a2: id0_t;
begin
 a1 := counter_balance(i);
 a2 := counter_balance(j);
 in_same_alliance := (a1 = a2) and (a1 <> 0);
end;

procedure update_trust (var trust: trust_t; sensitivity, delta: real);
begin
 trust := trunc((1.0 - sensitivity) * trust + sensitivity * delta);
 if trust > d_trust then
  trust := d_trust
 else if trust < -d_trust then
  trust := -d_trust;
end;
```

```
 procedure update_threatened (i, j: id_t);
{update threat levels of all units j threatened by i, provided that i is a predator}
  var
   neigh: link_t;
   sensitivity: real;
   obligation: boolean;
 begin
  obligation := false;
  if counter_balance(i) = j then
   if system[i].all^.obligation then
    obligation := true;
  if not obligation then
   begin
    neigh := system[i].neighs;
    repeat
     neigh := neigh^.next;
     if neigh^.pol then
      begin
       if neigh^.id = j then
        sensitivity := 1.0
       else
        sensitivity := provoke / 100.0;
       update_trust(neigh^.link^.trust, sensitivity, -d_trust);
      end;
    until last(neigh);
   end;
 end;


{$PUSH}
{$R-}
 procedure interaction (i, j: id_t; neigh_i, neigh_j: link_t);
  var
   ires, jres: res_t;

  procedure discount_res;
  begin
   ires := battle_res(i, neigh_i^.agent, neigh_i^.res);
   jres := battle_res(j, neigh_j^.agent, neigh_j^.res);
  end;
{$POP}

 begin
{need to avoid simultaneous attacks because agent/target can only be set}
{by one party}
  if attack(neigh_i, neigh_j) then
   if bern(0.5) then
    neigh_i^.act := c
   else
    neigh_j^.act := c;
{update history}
  neigh_i^.history[self] := neigh_i^.act;
  neigh_i^.history[other] := neigh_j^.act;
  neigh_j^.history[self] := neigh_j^.act;
  neigh_j^.history[other] := neigh_i^.act;

  if (neigh_i^.act = c) and (neigh_j^.act = c) then
```

```
      begin
        neigh_i^.dres := peace_div;
        neigh_j^.dres := peace_div;
        update_trust(neigh_i^.trust, forgive / 100.0, +d_trust);
        update_trust(neigh_j^.trust, forgive / 100.0, +d_trust);
      end
    else if (neigh_i^.act = c) and (neigh_j^.act = d) then
      begin
        neigh_i^.target := neigh_j^.agent;
        neigh_i^.agent := neigh_j^.target;
        discount_res;
        neigh_i^.dres := -defeat_cost * jres div 100;
        neigh_j^.dres := -victory_cost * ires div 100;
        update_threatened(j, i);
        if in_same_alliance(j, i) then
          begin
            remove_member(j);
          end;
      end
    else if (neigh_i^.act = d) and (neigh_j^.act = c) then
      begin
        neigh_j^.target := neigh_i^.agent;
        neigh_j^.agent := neigh_i^.target;
        discount_res;
        neigh_i^.dres := -victory_cost * jres div 100;
        neigh_j^.dres := -defeat_cost * ires div 100;
        update_threatened(i, j);
        if in_same_alliance(i, j) then
          begin
            remove_member(i);
          end;
      end
    else {d,d}
      begin
        discount_res;
        neigh_i^.dres := -war_cost * jres div 100;
        neigh_j^.dres := -war_cost * ires div 100;
        update_trust(neigh_i^.trust, provoke / 100.0, -d_trust);
        update_trust(neigh_j^.trust, provoke / 100.0, -d_trust);
      end;

  {check claims}
    neigh_i^.claim := false;
    neigh_j^.claim := false;
    if (neigh_i^.act = d) and win_battle(i, j, ires, jres) then
      neigh_i^.claim := true;
    if (neigh_j^.act = d) and win_battle(j, i, jres, ires) then
      neigh_j^.claim := true;
    event := neigh_i^.claim or neigh_j^.claim;
  end;


  procedure interactions;
{only confronts states; no internal interaction; update history}
    var
      i, j: id_t;
```

```
    neigh: link_t;
begin
 for i := 1 to n do { count up and down }
  if sov(i) then
   begin
    neigh := system[i].neighs;
    repeat
     neigh := neigh^.next;
     j := neigh^.id;
     if neigh^.pol then
      if i < j then
       interaction(i, j, neigh, neigh^.link);
    until last(neigh);
   end;
end;
```

Resources

```
function harvest (m, s: res_t): res_t;
begin
 if s = 0 then
  harvest := m
 else
  harvest := trunc(normal(m, s));
end;

procedure update_resources;
 var
  i: id_t;
  neigh: link_t;
  dres, tax: res_t;
begin
 for i := 1 to n do
  if sov(i) then
   begin
    dres := 0;
    tax := 0;
    neigh := system[i].neighs;
    repeat
     neigh := neigh^.next;
     if neigh^.pol then
      dres := dres + neigh^.dres
     else if neigh^.child then
      tax := tax + harvest(harvest_m, harvest_s);
    until last(neigh);
    system[i].res := system[i].res + harvest(harvest_m, harvest_s) + tax + dres;
    if system[i].res < 0 then
     system[i].res := 0;
   end;
end;

procedure update_all_resources;
 var
  all: all_t;
  member: link_t;
begin
 all := alliance_list;
 if not empty(alliance_list) then
  repeat
```

```
    all := all^.next;
    all^.res := 0;
    all^.obligation := false;
    member := all^.members;
    repeat
     member := member^.next;
     all^.res := all^.res + system[member^.id].res;
    until last(member);
  until last(all);
end;
```

Structural Change

```
procedure unlock;
 var
  i: id_t;
begin
 for i := 1 to n do
  locked[i] := false;
end;

procedure reset_reach (i: id_t);
 var
  child: link_t;
begin
 child := system[i].neighs;
 repeat
  child := child^.next;
  if child^.child then
   system[child^.id].reached := false;
 until last(child);
end;

procedure reach (i: id_t);
 var
  neigh: link_t;
begin
 neigh := system[i].neighs;
 repeat
  neigh := neigh^.next;
  if neigh^.terr and compatriots(i, neigh^.id) and not system[neigh^.id].reached then
   begin
    system[neigh^.id].reached := true;
    reach(neigh^.id);
   end;
 until last(neigh);
end;

procedure topology (i: id_t);
 var
  child: link_t;
begin
 reset_reach(i);
 reach(i);
 child := system[i].neighs;
 repeat
  child := child^.next;
  if child^.child and not system[child^.id].reached then
```

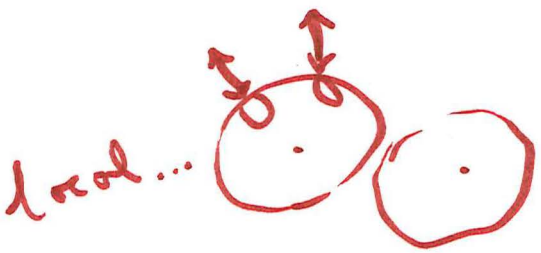
```
      remove_child(i, child^.id);
    until last(child);
  end;

  procedure collapse (j: id_t);
    var
      child_j: link_t;
  begin
    child_j := system[j].neighs;
    repeat
      child_j := child_j^.next;
      if child_j^.child then
        remove_child(j, child_j^.id);
    until last(child_j);
  end;

  procedure conquer (i, j: id_t);
    var
      invaded: id0_t;
  begin
{rewire hierarchy}
{assume target j is state}
    locked[i] := true;
    locked[j] := true;
    if subord(j) then
      begin {intrusion}
        invaded := system[j].parent;
        remove_child(invaded, j);
        topology(invaded);
      end
    else if not atom(j) then {collapse}
      collapse(j);
    add_child(i, j);
    if system[i].culture = pred then
      update_threatened(i, j);
  end;


  procedure structural_change;
    var
      i, state: id0_t;
      up: boolean;
      neigh: link_t;
  begin
{add spontaneous collapse}
    unlock;
    up := bern(0.5); {random direction of execution}
    for i := 1 to n do
      begin
        if up then
          state := i
        else
          state := n - i + 1;
        if sov(state) and not locked[state] then
          begin
            neigh := system[state].neighs;
            repeat
```

```
          neigh := neigh^.next;
          if neigh^.pol and neigh^.claim and not locked[neigh^.id] then
           begin
            neigh^.claim := false;
            if (neigh^.target >= 1) and (neigh^.target <= n) then {ERROR CHECK}
             conquer(state, neigh^.target)
            else
             writeln('ERROR CALLING CONQUER: ', state, neigh^.target);
           end;
         until last(neigh);
        end;
      end;
     for i := 1 to n do
      if sov(i) then
       if head(i) then
        if not locked[i] then
         if capita_res(i) < min_res then
          collapse(i);
end;
```

spontaneous collapse

res transf

```
procedure diplomacy;
 var
  threat, threatee, counter: id0_t;
  neigh: link_t;
  all: all_t;
begin
 for threat := 1 to n do
  if sov(threat) then
   begin
    threatee := 0;
    neigh := system[threat].neighs;
    repeat
     neigh := neigh^.next;
     counter := counter_balance(neigh^.id);
     if neigh^.pol then
      if (system[neigh^.id].threat = threat) and not ((counter <> 0) and (counter <> threat)) then
{can be added if threatened by threat and not engaged in other alliance}
       threatee := threatee + 1;
    until last(neigh);
    all := system[threat].counter_all;
    if all = nil then {no alliance exists}
     if threatee < 2 then
{do nothing}
      else
       create_all(threat)
    else {alliance already exists}
     if threatee < 2 then
      delete_all(all)
     else
      adjust_all(threat, all);
   end;
end;


end.
```



```
unit plot_mod;

interface
 uses
  implement_mod, base_mod, system_mod, relations_mod;

 type
  buffer_t = array[1..buffer_rows]  of string[buffer_cols];
  plot_mode_t = (always, never, count);
  plot_info_t = (no_info, id_info, res_info, all_info, threat_info, ext_info);

 var
  buffer: buffer_t;
  plot_mode: plot_mode_t;
  plot_info: plot_info_t;
  graphics: boolean;
  width: integer;

{procedure write_list (list: list_t; mode: mode_t);}

 procedure write_data (i: integer);

 procedure plot (width: integer);

 procedure plot_neighs (i: integer);

 procedure plot_all;

 procedure snapshot (width: integer; clist: clist_t);

 procedure display (clist: clist_t);

implementation


 procedure write_data (i: integer);
  var
   neigh: link_t;
 begin
  with system[i] do
   begin
    writeln('POS : ', pos.x : 4, pos.y : 4, ' PAR: ', parent : 4, 'CHILD: ', children : 3);
    writeln('THREAT:', threat : 4, ' RES:', res : 8, ' CULT:', culture);
    write('ALL:');
    if all = nil then
     write(' none')
    else
     write_list(all^.members,  total);
    writeln;
    neigh := system[i].neighs;
    writeln('ID TRUST RES DRES ACT AGENT TARG [HIST] CHILD TERR POL');
    repeat
     neigh := neigh^.next;
     with neigh^ do
      begin
       write(id : 4, ' ', trust : 5, '|', res : 8, dres : 6);
       write('|', act, agent : 4, target : 4, '[', history[self], history[other], ']');
```

```
            writeln(child, ' ', terr, ' ', pol, ' ');
          end;
        until last(neigh);
      end;
    end;

{$PUSH}
{$R-}
  procedure reset_buffer (width: integer);
    var
      x, y: integer;
  begin
    for x := 1 to buffer_rows do
      begin
        buffer[x] := '                                                                    ';
      end;
  end;
{$POP}

  function ext_res (i: id_t): res_t;
  begin
    if aligned(i) then
      ext_res := (1 - round(all_contribution)) * system[i].res + round(all_contribution) * system[i].all^.res
{NOTE all_contribution treated as toggle #REAL}
    else
      ext_res := system[i].res;
  end;


{$PUSH}
{$R-}
  procedure dashes (i, j, width: integer);
    var
      k: integer;
  begin
    for k := 0 to width - 1 do
      buffer[i, j + k] := '-';
  end;
{$POP}

{$PUSH}
{$R-}
  procedure blanks (i, j, width: integer);
    var
      k: integer;
  begin
    for k := 0 to width - 1 do
      buffer[i, j + k] := ' ';
  end;
{$POP}

{$PUSH}
{$R-}
  procedure write_buffer (i, j: integer; s: string_t);
    var
      k: integer;
  begin
```

```
    for k := 0 to len(s) - 1 do
      buffer[i, j + k] := s[k + 1];
  end;
{$POP}

{$PUSH}
{$R-}
 procedure plot_region (reg: id_t; width: integer);
  var
   i, j, x, y: integer;

  procedure plot_west;
  begin
   buffer[i, j - 1] := '|';
   buffer[i - 1, j - 1] := '+';
   buffer[i + 1, j - 1] := '+';
  end;

  procedure plot_east;
  begin
   buffer[i, j + width] := '|';
   buffer[i - 1, j + width] := '+';
   buffer[i + 1, j + width] := '+';
  end;

  procedure plot_north;
  begin
   dashes(i - 1, j, width);
   buffer[i - 1, j + width] := '+';
   buffer[i - 1, j - 1] := '+';
  end;

  procedure plot_south;
  begin
   dashes(i + 1, j, width);
   buffer[i + 1, j + width] := '+';
   buffer[i + 1, j - 1] := '+';
  end;

 begin
  x := system[reg].pos.x;
  y := system[reg].pos.y;
  i := 2 * x;
  j := 2 + (1 + width) * (y - 1);

  if sov(reg) then
   begin
    if system[reg].culture = pred then
     buffer[i, j] := 'x'
    else if width = 1 then
     buffer[i, j] := 'o'
    else
     buffer[i, j] := ' ';
    if plot_info <> no_info then
     if plot_info = id_info then
      write_buffer(i, j + 1, conv(system[reg].res, width - 1))
     else if plot_info = res_info then
```

```
          write_buffer(i, j + 1, conv(system[reg].res, width - 1))
        else if plot_info = threat_info then
          write_buffer(i, j + 1, conv(system[reg].threat, width - 1))
        else if plot_info = all_info then
          write_buffer(i, j + 1, conv(counter_balance(reg), width - 1))
        else if plot_info = ext_info then
          write_buffer(i, j + 1, conv(ext_res(reg), width - 1));
      end;

      if not within_range(x, y - 1) then
       plot_west
      else if foreigners(reg, index(x, y - 1)) then
       plot_west
      else
       buffer[i, j - 1] := ' ';

      if not within_range(x, y + 1) then
       plot_east
      else if foreigners(reg, index(x, y + 1)) then
       plot_east
      else
       buffer[i, j + width] := ' ';

      if not within_range(x - 1, y) then
       plot_north
      else if foreigners(reg, index(x - 1, y)) then
       plot_north
      else
       blanks(i - 1, j, width);

      if not within_range(x + 1, y) then
       plot_south
      else if foreigners(reg, index(x + 1, y)) then
       plot_south
      else
       blanks(i + 1, j, width);

    end;
  {$POP}

    procedure plot_corp (state: id_t; width: integer);
     var
      child: link_t;
    begin
     plot_region(state, width);
     child := system[state].neighs;
     repeat
      child := child^.next;
      if child^.child then
       plot_region(child^.id, width);
     until last(child);
    end;

    procedure plot_actor (state: id_t; width: integer);
    begin
     if atom(state) then
      plot_region(state, width)
```

```
    else
      plot_corp(state, width);
  end;

{$PUSH}
{$R-}
  procedure plot_battle (neigh_i, neigh_j: link_t; width: integer);
    var
      i, j, dx, dy: integer;
      agent_pos, target_pos: pos_t;
      ch: char;
      offense, defense: boolean;
  begin
    agent_pos := system[neigh_i^.agent].pos;
    target_pos := system[neigh_i^.target].pos;
    dx := (target_pos.x - agent_pos.x);
    dy := (target_pos.y - agent_pos.y);
    ch := '*';
    offense := (neigh_i^.act = d) and (neigh_j^.act = c);
    defense := (neigh_j^.act = d) and (neigh_i^.act = c);
    if offense or defense then
      if dy = 0 then
        begin
          if ((dx < 0) and offense) or ((dx > 0) and defense) then
            ch := '^'
          else
            ch := 'v';
        end
      else
        begin
          if ((dy < 0) and offense) or ((dy > 0) and defense) then
            ch := '<'
          else
            ch := '>';
        end;
    if (dx = 0) and (dy > 0) then
      dy := width;
    buffer[2 * agent_pos.x + dx, 2 + (1 + width) * (agent_pos.y - 1) + dy] := ch;
  end;
{$POP}

  procedure plot_interaction (i, width: integer);
    var
      neigh: link_t;
      agent: integer;
  begin
    neigh := system[i].neighs;
    repeat
      neigh := neigh^.next;
      if neigh^.pol and ((neigh^.act = d) or (neigh^.link^.act = d)) and (i < neigh^.id) then
        begin
          plot_battle(neigh, neigh^.link, width);
        end;
    until last(neigh);
  end;

  procedure plot_actors (width: integer);
```

```
    var
      i: integer;
    begin
     for i := 1 to n do
      if sov(i) then
       plot_actor(i, width);
     for i := 1 to n do
      if sov(i) then
       plot_interaction(i, width);
    end;

  {$PUSH}
  {$R-}
    procedure plot_buffer (width: integer);
     var
      x, y: integer;
    begin
     for x := 1 to nx * 2 + 1 do
      begin
       for y := 1 to ny * (1 + width) + 1 do
        write(buffer[x, y]);
       writeln;
      end;
    end;
  {$POP}

    procedure plot (width: integer);
    begin
     reset_buffer(width);
     plot_actors(width);
     plot_buffer(width);
    end;

    procedure plot_neighs (i: integer);
     var
      neigh: link_t;
      p: pos_t;
    begin
     reset_buffer(1);
     plot_actors(1);
     neigh := system[i].neighs;
     repeat
      neigh := neigh^.next;
      if neigh^.pol then
       begin
        p := system[neigh^.id].pos;
        buffer[2 * p.x, 2 * p.y] := 'P';
       end;
     until last(neigh);
     plot_buffer(1);
    end;

    procedure plot_all;
     var
      all: all_t;
      m: link_t;
    begin
```

```
  writeln('ID THREAT RES MEMBERS');
  all := alliance_list;
  if empty(alliance_list) then
   writeln('none')
  else
   repeat
    all := all^.next;
    write(all^.id : 4);
    write(all^.threat : 4);
    write(all^.res : 8);
    m := all^.members;
    if EMPTY(M) then
     WRITELN('ERROR ')
    else
     repeat
      m := m^.next;
      write(m^.id : 4);
     until last(m);
    writeln;
   until last(all);
end;

procedure snapshot (width: integer; clist: clist_t);
 var
  f: text;
  x, y, i: integer;
  ch: char;
begin
 rewrite(f, clist.entry[2]);
 writeln(f, nx, ny, width, time);
 reset_buffer(width);
 plot_actors(width);
 for x := 1 to nx * 2 + 1 do
  begin
   for y := 1 to ny * (1 + width) + 1 do
    begin
     ch := buffer[x, y];
     if buffer[x, y] = ' ' then
      begin
       i := index(x div 2, y div (width + 1) + 1);
       if system[govt(i)].culture = pred then
        ch := '.';
      end;
     write(f, ch);
    end;
   writeln(f);
  end;
 close(f);
end;

procedure display (clist: clist_t);
 var
  f: text;
  nx, width, i, j: integer;
begin
 reset(f, clist.entry[2]);
 readln(f, nx, ny, width, time);
```

```
    writeln('Snapshot at time ', time);
    for i := 1 to 2 * nx + 1 do
     begin
      readln(f, buffer[i]);
      for j := 1 to (width + 1) * ny + 1 do
       if buffer[i, j] = '.' then
        buffer[i, j] := ' ';
      writeln(buffer[i]);
     end;
    close(f);
  end;

end.
```

```
unit draw_mod;

interface
 uses
  implement_mod, base_mod, system_mod, relations_mod, plot_mod;

 var
  box: pos_t;
  halfbox: pos_t;
  gray_shades: boolean;


 procedure create_window;

 procedure reset_window;

 procedure save_draw (s: string_t);

 procedure draw (width: integer; init_borders: boolean);

 procedure draw_file (clist: clist_t);

 procedure text_rect (top, left, bottom, right: integer);


implementation

 procedure create_window;
  const
   size = 9;
  var
   style: integer;
   wrect: rect;
 begin
  wrect.left := 1;
  wrect.top := 40;
  wrect.right := 380;
  wrect.bottom := 380;
  setdrawingrect(wrect);
  showdrawing;
  TextSize(size);
  TextFace([condense]);
  getfnum('Courier', style);
  textfont(style);
 end;

 procedure reset_window;
 begin
  forecolor(whitecolor);
  paintrect(0, 0, 380, 380);
  forecolor(blackcolor);
 end;

 function conv_pos (pos: pos_t): pos_t;
  var
   result: pos_t;
 begin
```

```
    result.x := halfbox.x + (pos.x - 1) * box.x;
    result.y := halfbox.y * width + (pos.y - 1) * box.y * width;
    conv_pos := result;
  end;

  procedure draw_region (reg: id_t; width_ext: integer; init_borders: boolean);
    const
      width = 1;

    var
      screen, pos: pos_t;
      color: integer;

{$PUSH}
{$R-}
    procedure draw_borders (reg: id_t; pos: pos_t; dx, dy, blankcolor: integer);
      var
        neighbor: pos_t;

      procedure draw_border (pos: pos_t; dx, dy: integer; border: boolean; blankcolor: integer);
        var
          screen, start, finish: pos_t;
      begin
        if gray_shades then
          if border then
            forecolor(blackcolor)
          else
            forecolor(blankcolor)
        else if border then
          forecolor(blackcolor)
        else
          begin
            if blankcolor = redcolor then
              begin
                forecolor(blackcolor);
                penpat(gray);
              end
            else
              forecolor(whitecolor);
          end;
        screen := conv_pos(pos);
        start.x := -(1 - abs(dx)) + abs(dx) * dx;
        finish.x := (1 - abs(dx)) + abs(dx) * dx;
        start.y := -(1 - abs(dy)) + abs(dy) * dy;
        finish.y := (1 - abs(dy)) + abs(dy) * dy;
        moveto(screen.y + start.y * halfbox.y * width, screen.x + start.x * halfbox.x);
        lineto(screen.y + finish.y * halfbox.y * width, screen.x + finish.x * halfbox.x);
        penpat(black);
      end;

    begin
      neighbor.x := pos.x + dx;
      neighbor.y := pos.y + dy;
      if not within_range(neighbor.x, neighbor.y) then
        draw_border(pos, dx, dy, true, blankcolor)
      else
        draw_border(pos, dx, dy, foreigners(reg, index(neighbor.x, neighbor.y)), blankcolor);
```

```
  end;

  procedure fill_box (screen: pos_t; color: integer);
   var
    upper, lower: pos_t;
  begin
   if gray_shades then
    forecolor(color)
   else if color = redcolor then
    begin
     forecolor(blackcolor);
     penpat(gray);
    end
   else
    forecolor(whitecolor);
   paintrect(screen.x - halfbox.x + 1, screen.y - halfbox.y * width + 1, screen.x + halfbox.x, screen.y + halfbox
width);
   forecolor(blackcolor);
   penpat(black);
  end;

begin
 pos := system[reg].pos;
 screen := conv_pos(pos);
 if system[govt(reg)].culture = pred then
  color := redcolor
 else
  color := whitecolor;
 fill_box(screen, color);
 moveto(screen.y - halfbox.y * width + 5, screen.x + 4);
 if sov(reg) then
  begin
   if not gray_shades and (plot_info <> no_info) then
    begin
     forecolor(whitecolor);
     paintrect(screen.x - 5, screen.y - halfbox.y * width + 2, screen.x + 5, screen.y + halfbox.y * width - 2);
     forecolor(blackcolor);
    end;
   if plot_info = no_info then
    drawstring('*')
   else if plot_info = id_info then
    drawstring(conv(reg, width))
   else if plot_info = res_info then
    drawstring(conv(system[reg].res, width))
   else if plot_info = threat_info then
    drawstring(conv(system[reg].threat, width))
   else
    drawstring(conv(round(counter_balance(reg)), width));
  end;
 if init_borders or locked[reg] then
  begin
   draw_borders(reg, pos, -1, 0, color);
   draw_borders(reg, pos, 1, 0, color);
   draw_borders(reg, pos, 0, -1, color);
   draw_borders(reg, pos, 0, 1, color);
  end;
end;
```

```
{$POP}


procedure draw_corp (state: id_t; width: integer; init_borders: boolean);
 var
  child: link_t;
begin
 draw_region(state, width, init_borders);
 child := system[state].neighs;
 repeat
  child := child^.next;
  if child^.child then
   draw_region(child^.id, width, init_borders);
 until last(child);
end;

procedure draw_interaction (i: id_t; width: integer);
 var
  neigh: link_t;

 procedure plot_battle (neigh_i, neigh_j: link_t; width: integer);
  var
   dx, dy: integer;
   agent_pos, target_pos, agent_s, target_s, mid_s: pos_t;
 begin
  agent_pos := system[neigh_i^.agent].pos;
  target_pos := system[neigh_i^.target].pos;
  dx := (target_pos.x - agent_pos.x);
  dy := (target_pos.y - agent_pos.y);
  agent_s := conv_pos(agent_pos);
  target_s := conv_pos(target_pos);
  mid_s.x := (agent_s.x + target_s.x) div 2;
  mid_s.y := (agent_s.y + target_s.y) div 2;
  moveto(mid_s.y, mid_s.x);
  lineto(mid_s.y - 4 * dy, mid_s.x - 4 * dx);
 end;

begin
 neigh := system[i].neighs;
 repeat
  neigh := neigh^.next;
  if neigh^.pol and (neigh^.link^.act = d) then
   begin
    plot_battle(neigh, neigh^.link, width);
   end;
 until last(neigh);
end;

procedure draw_actor (state: id_t; width: integer; init_borders: boolean);
begin
 if atom(state) then
  draw_region(state, width, init_borders)
 else
  draw_corp(state, width, init_borders);
 draw_interaction(state, width);
end;
```

```
  procedure draw (width: integer; init_borders: boolean);
   var
    i: integer;
  begin
   for i := 1 to n do
    if sov(i) then
     if locked[i] or (width > 1) or init_borders then
      draw_actor(i, width, init_borders);
   for i := 1 to n do
    if sov(i) then
     draw_interaction(i, width);
   forecolor(whitecolor);
   paintrect(310, 0, 320, 100);
   forecolor(blackcolor);
   moveto(0, 320);
   if gray_shades then
    drawstring(concat('T=', conv(time, 5)));
  end;

  procedure draw_file_reg (x, y: integer; culture: culture_t; sover: boolean; width: integer);

   var
    screen, pos: pos_t;
    color: integer;

   function conv_pos (pos: pos_t): pos_t;
    var
     result: pos_t;
   begin
    result.x := halfbox.x + (pos.x - 1) * box.x;
    result.y := halfbox.y * width + (pos.y - 1) * box.y * width;
    conv_pos := result;
   end;

{$PUSH}
{$R-}
   procedure draw_borders (pos: pos_t; dx, dy, blankcolor: integer);
    var
     neighbor: pos_t;
     ch: char;
     border: boolean;
     extra: integer;

    procedure draw_border (pos: pos_t; dx, dy: integer; border: boolean; blankcolor: integer);
     var
      screen, start, finish: pos_t;
    begin
     if border then
      forecolor(blackcolor)
     else if gray_shades then
      forecolor(blankcolor)
     else
      begin
       forecolor(blackcolor);
       penpat(gray);
      end;
     screen := conv_pos(pos);
```

```
    start.x := -(1 - abs(dx)) + abs(dx) * dx;
    finish.x := (1 - abs(dx)) + abs(dx) * dx;
    start.y := -(1 - abs(dy)) + abs(dy) * dy;
    finish.y := (1 - abs(dy)) + abs(dy) * dy;
    moveto(screen.y + start.y * halfbox.y * width, screen.x + start.x * halfbox.x);
    lineto(screen.y + finish.y * halfbox.y * width, screen.x + finish.x * halfbox.x);
    penpat(black);
   end;

  begin
   neighbor.x := pos.x + dx;
   neighbor.y := pos.y + dy;
   if dy > 0 then
    extra := width - 1
   else
    extra := 0;
   ch := buffer[x * 2 + dx, y * (width + 1) + dy + extra];
   border := not ((ch = ' ') or (ch = '.'));
   draw_border(pos, dx, dy, border, blankcolor);
  end;

  procedure fill_box (screen: pos_t; color: integer);
   var
    upper, lower: pos_t;
  begin
   if gray_shades then
    forecolor(color)
   else if color = redcolor then
    begin
     forecolor(blackcolor);
     penpat(gray);
    end
   else
    forecolor(whitecolor);
   paintrect(screen.x - halfbox.x + 1, screen.y - halfbox.y * width + 1, screen.x + halfbox.x, screen.y + halfbox
width);
   forecolor(blackcolor);
   penpat(black);
  end;

 begin
  pos.x := x;
  pos.y := y;
  screen := conv_pos(pos);
  if culture = pred then
   color := redcolor
  else
   color := whitecolor;
  fill_box(screen, color);
  moveto(screen.y - halfbox.y * width + 4, screen.x + halfbox.x);
  if not gray_shades and (width <> 1) then
   begin
    forecolor(whitecolor);
    paintrect(screen.x + halfbox.x - 8, screen.y - halfbox.y * width + 2, screen.x + halfbox.x - 1, screen.y + ha
* width - 8);
    forecolor(blackcolor);
   end;
```

```
    if sover then
     if (width = 1) then
      drawstring('*');
    draw_borders(pos, -1, 0, color);
    draw_borders(pos, 1, 0, color);
    draw_borders(pos, 0, -1, color);
    draw_borders(pos, 0, 1, color);
   end;

   procedure draw_file (clist: clist_t);
  {only width=1 supported; need to print entries for other widths}

    var
     f: text;
     i, j, nx, ny, x, y, width, time: integer;
     culture: culture_t;
     ch: char;
     sover: boolean;
   begin
    reset(f, clist.entry[2]);
    readln(f, nx, ny, width, time);
    writeln('Snapshot at time ', time);
    for i := 1 to 2 * nx + 1 do
     begin
      readln(f, buffer[i]);
     end;
    close(f);
    reset_window;
    for x := 1 to nx do
     begin
      for y := 1 to ny do
       begin
        i := 2 * x;
        j := (1 + width) * (y - 1) + 2;
        ch := buffer[i, j];
        if (ch = '.') or (ch = 'x') then
         culture := pred
        else
         culture := prey;
        sover := (ch = 'o') or (ch = 'x');
        draw_file_reg(x, y, culture, sover, width);
       end;
     end;
   end;

   procedure save_draw (s: string_t);
   begin
    savedrawing(s);
   end;

   procedure text_rect (top, left, bottom, right: integer);
    var
     wrect: rect;
   begin
    wrect.top := top;
    wrect.left := left;
    wrect.bottom := bottom;
```

```
      wrect.right := right;
      settextrect(wrect);
    end;

  end.
```

```
unit transfer_mod;

interface
 uses
  implement_mod, base_mod, system_mod, relations_mod, action_mod, plot_mod, draw_mod;

 procedure transfer (clist: clist_t);

implementation

 procedure transfer (clist: clist_t);
  type
   transfer_mode_t = (write_mode, read_mode);
  var
   f: text;
   transf_mode: transfer_mode_t;

  procedure transf (var f: text; var i: integer);
  begin
   if transf_mode = write_mode then
    writeln(f, i)
   else
    readln(f, i);
  end;

  procedure transfr (var f: text; var r: real);
  begin
   if transf_mode = write_mode then
    writeln(f, r)
   else
    readln(f, r);
  end;

  procedure transseed (var f: text; var i: seed_t);
  begin
   if transf_mode = write_mode then
    writeln(f, i : 10)
   else
    readln(f, i);
  end;

  procedure transfres (var f: text; var i: res_t);
  begin
   if transf_mode = write_mode then
    writeln(f, i : 10)
   else
    readln(f, i);
  end;

  procedure transftrust (var f: text; var i: trust_t);
  begin
   if transf_mode = write_mode then
    writeln(f, i : 10)
   else
    readln(f, i);
  end;
```

```
procedure transfb (var f: text; var b: boolean);
 var
  ch: char;
begin
 if transf_mode = write_mode then
  if b = true then
   writeln(f, 'T')
  else
   writeln(f, 'F')
 else
  begin
   readln(f, ch);
   if ch = 'T' then
    b := true
   else
    b := false
  end;
end;

procedure transfc (var f: text; var c: culture_t);
 var
  ch: char;
begin
 if transf_mode = write_mode then
  if c = pred then
   writeln(f, '1')
  else
   writeln(f, '0')
 else
  begin
   readln(f, ch);
   if ch = '1' then
    c := pred
   else
    c := prey;
  end;
end;

procedure transfs (var f: text; var s: id0_t);
begin
 if transf_mode = write_mode then
  writeln(f, s)
 else
  readln(f, s);
end;

procedure transfa (var f: text; var a: act_t);
 var
  ch: char;
begin
 if transf_mode = write_mode then
  if a = c then
   writeln(f, 'c')
  else
   writeln(f, 'd')
 else
  begin
```

```
        readln(f, ch);
        if ch = 'c' then
          a := c
        else
          a := d;
       end;
    end;

    procedure transfln (var f: text);

    begin
     if transf_mode = write_mode then
      writeln(f)
     else
      readln(f);
    end;

    procedure transf_dummy (var f: text);
     var
      ch: char;
    begin
     if transf_mode = write_mode then
      writeln(f, '/')
     else
      readln(f, ch);
    end;

    procedure transf_globals;
     var
      version: integer;
    begin
     version := 1;
     transf(f, version);
     transseed(f, seed);
     transseed(f, seed0);
     transf(f, time);
     transf(f, n);
     transf(f, nx);
     transf(f, ny);
     transf(f, sov_states);
     ;
{transf_dummy(f)}
    transf(f, all_id);
{transf_dummy(f);}
    transfb(f, alliances);
{transfln(f);}

    transfres(f, res_m);
    transfres(f, res_s);
    transfres(f, harvest_m);
    transfres(f, harvest_s);
    transf(f, freq_pred);
    transf(f, bell_pred);
{transfln(f);}

    transf(f, superior_ratio);
    transf(f, victory_ratio);
```

```
      transfres(f, min_res);
      transf(f, grad);
      transf(f, all_contribution);
      transf(f, pr_oblig);

      transfres(f, peace_div);
      transfres(f, victory_cost);
      transfres(f, defeat_cost);
      transfres(f, war_cost);
      transf(f, mobil);
      transftrust(f, soc_neg);
      transftrust(f, soc_pos);
      transftrust(f, min_threat);

      transf(f, forgive);
      transf(f, provoke);
    end;

    procedure transf_state (i: integer);
     var
      state: integer;
    begin
     with system[i] do
      begin
       state := i;
       transf(f, state);
       transf(f, pos.x);
       transf(f, pos.y);
       transfres(f, res);
       transfs(f, parent);
       transf(f, children);
       transfs(f, threat);
{transf(f, pol_neighs);}
{transftrust(f, trust);}
{transf_dummy(f);}
       transfc(f, culture);
       transfb(f, reached);
{transfln(f);}
      end;
    end;

    procedure transf_link (neigh: link_t);
    begin
     with neigh^ do
      begin
       transfs(f, id);
       transftrust(f, trust);
       transfs(f, agent);
       transfs(f, target);
       transfres(f, res);
       transfres(f, dres);
{transf_dummy(f);}
       transfa(f, act);
       transfa(f, history[self]);
       transfa(f, history[other]);
       transfb(f, child);
       transfb(f, terr);
```

```
        transfb(f, pol);
        transfb(f, claim);
    {transfln(f);}
      end;
    end;


    procedure transf_all (all: all_t);
    begin
     with all^ do
      begin
       transfs(f, id);
       transfs(f, threat);
       transfres(f, res);
       transf(f, n_members);
       transfb(f, obligation);
    {transfln(f);}
      end;
    end;

    procedure save_system;
     var
      neigh: link_t;
      i: integer;
    begin
     for i := 1 to n do
      begin
       transf_state(i);
       neigh := system[i].neighs;
       repeat
        neigh := neigh^.next;
        with neigh^ do
         begin
          writeln(f, '>');
          transf_link(neigh);
         end;
       until last(neigh);
       writeln(f, '.');
      end;
    end;

    procedure restore_system;
     var
      i: integer;
      neigh: link_t;
      ch: char;
    begin
     for i := 1 to n do
      begin
       transf_state(i);
       new_list(system[i].neighs);
       system[i].all := nil;
       system[i].counter_all := nil;
       repeat
        readln(f, ch);
        if ch = '>' then
         begin
```

```
          neigh := add_link_last(system[i].neighs);
          transf_link(neigh);
        end;
      until ch = '.';
    end;
  end;

  procedure restore_links;
   var
    i: integer;
    neigh, neigh_j: link_t;
  begin
   for i := 1 to n do
    begin
     neigh := system[i].neighs;
     repeat
      neigh := neigh^.next;
      if (i < neigh^.id) then
       begin
        neigh_j := find_neigh(neigh^.id, i);
        neigh^.link := neigh_j;
        neigh_j^.link := neigh;
       end;
     until last(neigh);
    end;
  end;

  procedure save_all;
   var
    all: all_t;
    m: link_t;
  begin
   all := alliance_list;
   if empty(all) then
    writeln(f, '.')
   else
    begin
     repeat
      writeln(f, '>');
      all := all^.next;
      transf_all(all);

      m := all^.members;
      repeat
       m := m^.next;
       writeln(f, 'm');
       writeln(f, m^.id);
      until last(m);
      writeln(f, ',');
     until last(all);
     writeln(f, '.');
    end;
  end;

  procedure restore_all;
   var
    all: all_t;
```

```
      m: link_t;
      ch, ch2: char;
      id: integer;
    begin
     new_list(alliance_list);
     repeat
      readln(f, ch);
      if ch = '>' then
       begin
        all := add_link_last(alliance_list);
        transf_all(all);
        system[all^.threat].counter_all := all;
        new_list(all^.members);
        repeat
         readln(f, ch2);
         if ch2 = 'm' then
          begin
           m := add_link_last(all^.members);
           readln(f, id);
           m^.id := id;
           system[id].all := all;
          end;
        until ch2 = ',';
       end;
     until ch = '.';
    end;



  begin
   if clist.entry[1] = 'save' then
    begin
     transf_mode := write_mode;
     rewrite_file(f, clist.entry[2]);
     transf_globals;
     save_system;
     save_all;
    end
   else
    begin
     if graphics then
      reset_window;
     if not first_system then
      delete_system;
     transf_mode := read_mode;
     reset_file(f, clist.entry[2]);
     transf_globals;
     restore_system;
     restore_links;
     restore_all;
    end;

   close(f);
  end;

end.
```

```
program simulation (input, output);
{930705 implementing defensive alliances; esp. deterrent function through all_contribution}
{Ideas: threat is now based on attacks; this may be too late; better use res*culture}
{Next to be implemented: save and load}
{Other changes: fewer modules}
{Complete conv program}
{930706 save/restore continued}
{will try with linked list of alliances; DONE}
{NEW: resources based on integers only; DONE}
{930713}
{Plan:}
{1. Revise collapse conditions; DONE}
{2. Clean decide. Allocate_res?}
{3. Revise trust mechanism -- should depend on dR? or R? WORKING 930714}
{4. Include state log; NOT DONE}
{5. Revisions and tests of grad mechanism  DEBUGGED}
{6. Soc. mechanism. First version IMPLEMENTED; testing mix of all and soc a la Waltz/Wendt}
{ }
{STAT package will have to feature:}
{1. #states (sov,pred), size; 2. alliances; 3. state attr (min_res,max_res,av_res,hitlist)}
{930714 debugged new save/restore (more compact)}
{   also new proc snapshot , etc; still incomplete w / r to width }
{   also implemented pixel graphics for the printing and saving with the photo function}
{  MOST URGENT: revise grad again by including the right grad in relational res registers}
{930718 New modularization: implement_mod.p and draw_mod.p only implementation dependent modules}
{   also created new module transfer_mod.p}
{   - consistent use of generic types: res_t, trust_t}
{   - replication implemented}
{   - statistics}
{   - improved initiatiation of culture with fixed proportion of preds}
{FIX: resource_all in prop. to trust?}
{FIX: more opportunistically forgiving predators; no kamikaze}
{930721 thinking about trust}
{Fixing bugs in Draw about width}
{ADD: actions in draw}
{FIX: 1 2 3 4... bug}

 uses
  implement_mod, base_mod, system_mod, relations_mod, action_mod, plot_mod, draw_mod, transfer_mod;

 type
  stat_mode_t = (none, every10, every);

 var
  current_actor: id_t;
  max_col: integer;
  stat_mode: stat_mode_t;
  stat_file: text;
  stat_file_open, stat_record, plot_stat: boolean;

{stat vars}
  n_pred, dyad, war, n_pred_units, area_pred: integer;
  sum_trust, min_trust: trust_t;
  max_res, res, pred_res: res_t;
  hegemon: id0_t;

 procedure calculate_statistics;
```

```
  var
    neigh: link_t;
    i: id_t;
  begin
    n_pred := 0;
    dyad := 0;
    war := 0;
    n_pred_units := 0;
    area_pred := 0;
    min_trust := 999999;
    max_res := -999999;
    sum_trust := 0;
    res := 0;
    pred_res := 0;
    hegemon := 0;
    for i := 1 to n do
      begin
        if system[i].culture = pred then
          n_pred_units := n_pred_units + 1;
        if system[govt(i)].culture = pred then
          area_pred := area_pred + 1;
      end;
    for i := 1 to n do
      begin
        if sov(i) then
          begin
            res := res + system[i].res;
            if system[i].culture = pred then
              begin
                n_pred := n_pred + 1;
                pred_res := pred_res + system[i].res;
              end;
            if system[i].res > max_res then
              begin
                hegemon := i;
                max_res := system[i].res;
              end;
            neigh := system[i].neighs;
            repeat
              neigh := neigh^.next;
              if (neigh^.pol) and (i < neigh^.id) then
                begin
                  dyad := dyad + 1;
                  if (neigh^.act = d) or (neigh^.link^.act = d) then
                    war := war + 1;
                  sum_trust := sum_trust + round((neigh^.trust + neigh^.link^.trust) / 2);
                  if neigh^.trust < min_trust then
                    min_trust := neigh^.trust;
                  if neigh^.link^.trust < min_trust then
                    min_trust := neigh^.link^.trust;
                end;
            until last(neigh);
          end;
      end;
  end;
```

```
  procedure  display_statistics;
   var
    col, col_pred, col_sov: integer;
  begin
   col_pred := round(n_pred * max_col / n);
   col_sov := round(sov_states * max_col / n);
   write(time : 5);
   for col := 1 to col_pred - 1 do
    write('.');
   if (col_pred <> col_sov) and (col_pred <> 0) then
    write('+');
   for col := 1 to col_sov - 1 - col_pred do
    write('.');
   write('*');
   for col := col_sov to max_col do
    write(' ');
   writeln('I');
  end;

  procedure  record_statistics;
  begin
   writeln(stat_file, seed0, time : 7, sov_states, n_pred : 5, dyad : 5, war : 5, sum_trust, min_trust, res, max_
 n_pred_units : 5, area_pred : 5);
{Last two entries used to be: pred_res, hegemon}
  end;

{$PUSH}
{$R-}
  procedure  plot_statistics;
  begin
   writeln(time : 5, sov_states : 8, n_pred * 100.0 / sov_states : 4 : 0, '% pred;');
   writeln(war * 100.0 / dyad : 4 : 0, '% war;', sum_trust / dyad : 6 : 0, ' av.trust;', min_trust, ' min trust');
  end;
{$POP}

  procedure  record_file (clist: clist_t);
  begin
   if (clist.length = 1) then
    if stat_file_open then
     begin
      calculate_statistics;
      record_statistics;
     end
    else
     writeln('ERROR: No file')
   else if clist.entry[2] = 'end' then
    begin
     stat_record := false;
     stat_file_open := false;
     close(stat_file);
    end
   else
    begin
     if stat_file_open then
      close(stat_file)
     else
      begin
```

```
        stat_file_open := true;
        stat_record := true;
        rewrite(stat_file,  clist.entry[2]);
       end;
    end;
end;

procedure statistics;

begin
 calculate_statistics;
 if (stat_mode = every) or ((stat_mode = every10) and (time mod 10 = 0)) then
  display_statistics;
 if stat_record and stat_file_open then
  record_statistics;
 if plot_stat then
  plot_statistics;
end;

procedure simul (clist: clist_t);
 var
  time0, until_time, old_states, count_tenth: integer;
  tenth: real;
begin
 if clist.length = 1 then
  until_time := time + 1
 else
  until_time  :=  val(clist.entry[2]);
 time0 := time;
 tenth := (until_time - time) / 10.0;
 if tenth * n < 500.0 then
  count_tenth := 100
 else
  begin
   count_tenth := 1;
   if plot_mode = count then
    write(0 : 3);
  end;
 event := false;
 if graphics then
  draw(width,  true);
 while (sov_states <> 1) and (time < until_time) do
  begin
   time := time + 1;
   if (plot_mode = count) and (time >= count_tenth * tenth + time0) then
    begin
     write(count_tenth : 3);
     count_tenth := count_tenth + 1;
    end;
   old_states := sov_states;
   event := false;
   perceptions;
   decisions;
   interactions;
   update_resources;
   structural_change;
   if alliances then
```

```
      begin
       diplomacy;
       update_all_resources;
      end;
     if (plot_mode = always) then
      if graphics then
       if (soc_neg <> 0) or (soc_pos <> 0) then
        draw(width, true)
       else if plot_info = no_info then
        draw(width, false)
       else
        draw(width, true)
      else
       plot(width);
     if plot_stat or stat_record or (stat_mode = every) or ((stat_mode = every10) and (time mod 10 = 0)) the
      statistics;
   end;
  if (count_tenth < 100) and (plot_mode = count) then
   writeln(' DONE!');
  writeln('T= ', time : 5, ' Sov= ', sov_states : 4, ' ', pred_states / sov_states * 100 : 4 : 0, '% pred');
end;


procedure set_params (clist: clist_t);
begin
 if clist.entry[2] = 'res' then
  system[current_actor].res := val(clist.entry[3])
 else if clist.entry[2] = 'culture' then
  if clist.entry[3] = 'pred' then
   system[current_actor].culture := pred
  else
   system[current_actor].culture := prey
 else if clist.entry[2] = 'init_res' then
  begin
   res_m := val(clist.entry[3]);
   res_s := val(clist.entry[4]);
  end
 else if clist.entry[2] = 'box' then
  begin
   box.x := val(clist.entry[3]);
   box.y := val(clist.entry[4]);
   halfbox.x := box.x div 2;
   halfbox.y := box.y div 2;
  end
 else if clist.entry[2] = 'min_res' then
  min_res := val(clist.entry[3])
 else if clist.entry[2] = 'harvest' then
  begin
   harvest_m := val(clist.entry[3]);
   harvest_s := val(clist.entry[4]);
  end
 else if clist.entry[2] = 'seed' then
  seed := val(clist.entry[3])
 else if clist.entry[2] = 'freq_pred' then
  freq_pred := val(clist.entry[3])
 else if clist.entry[2] = 'bell_pred' then
  bell_pred := val(clist.entry[3])
```

```
else if clist.entry[2] = 'sup' then
 superior_ratio := val(clist.entry[3])
else if clist.entry[2] = 'all_contribution' then
 all_contribution := val(clist.entry[3])
else if clist.entry[2] = 'victory' then
 victory_ratio := val(clist.entry[3])
else if clist.entry[2] = 'mobil' then
 mobil := val(clist.entry[3])
else if clist.entry[2] = 'grad' then
 grad := val(clist.entry[3])
else if clist.entry[2] = 'min_threat' then
 min_threat := val(clist.entry[3])
else if clist.entry[2] = 'soc-' then
 soc_neg := val(clist.entry[3])
else if clist.entry[2] = 'soc+' then
 soc_pos := val(clist.entry[3])
else if clist.entry[2] = 'forgive' then
 forgive := val(clist.entry[3])
else if clist.entry[2] = 'provoke' then
 provoke := val(clist.entry[3])
else if clist.entry[2] = 'oblig' then
 pr_oblig := val(clist.entry[3])
else if clist.entry[2] = 'width' then
 width := val(clist.entry[3])
else if clist.entry[2] = 'max_col' then
 max_col := val(clist.entry[3])
else if clist.entry[2] = 'stat' then
 if clist.entry[3] = 'every10' then
  stat_mode := every10
 else if clist.entry[3] = 'every' then
  stat_mode := every
 else
  stat_mode := none
else if clist.entry[2] = 'graphics' then
 if clist.entry[3] = 'true' then
  graphics := true
 else
  graphics := false
else if clist.entry[2] = 'gray' then
 if clist.entry[3] = 'true' then
  gray_shades := true
 else
  gray_shades := false
else if clist.entry[2] = 'alliances' then
 if clist.entry[3] = 'true' then
  alliances := true
 else
  alliances := false
else if clist.entry[2] = 'record' then
 if clist.entry[3] = 'true' then
  stat_record := true
 else
  stat_record := false
else if clist.entry[2] = 'plot' then
 begin
  if clist.entry[3] = 'always' then
   plot_mode := always
```

```
      else if clist.entry[3] = 'never' then
       plot_mode := never
      else if clist.entry[3] = 'count' then
       plot_mode := count
      else if clist.entry[3] = 'id' then
       plot_info := id_info
      else if clist.entry[3] = 'res' then
       plot_info := res_info
      else if clist.entry[3] = 'threat' then
       plot_info := threat_info
      else if clist.entry[3] = 'all' then
       plot_info := all_info
      else if clist.entry[3] = 'ext' then
       plot_info := ext_info
      else if clist.entry[3] = 'no_info' then
       plot_info := no_info
      else
       writeln('ERROR');
    end
   else if clist.entry[2] = 'interact' then
    begin
     peace_div := val(clist.entry[3]);
     victory_cost := val(clist.entry[4]);
     defeat_cost := val(clist.entry[5]);
     war_cost := val(clist.entry[6]);
    end
   else if clist.entry[2] = 'actor' then
    current_actor := val(clist.entry[3])
end;


procedure init_system;

begin
 stat_record := false;
 stat_file_open := false;
 plot_stat := false;
 box.x := 20;
 box.y := 20;
 halfbox.x := box.x div 2;
 halfbox.y := box.y div 2;
 current_actor := 1;
 seed := 1;
 seed0 := 1;
 freq_pred := 20;
 bell_pred := 100;
 superior_ratio := 300;
 victory_ratio := 300;
 all_contribution := 100;
 pr_oblig := 100;
 mobil := 0;
 grad := 100;
 min_threat := 0;
 soc_neg := 0;
 soc_pos := 0;
 res_m := 50;
 res_s := 20;
```

```
  min_res := 10;
  harvest_m := 0;
  harvest_s := 0;
  peace_div := 0;
  victory_cost := 0;
  defeat_cost := 5;
  war_cost := 5;
  provoke := 50;
  forgive := 1;
  width := 1;
  plot_mode := always;
  plot_info := no_info;
  n := 0;
  alliances := false;
  graphics := true;
  stat_mode := none;
  max_col := 25;
  gray_shades := true;
  create_window;
  first_system := true;
  text_rect(40, 385, 400, 640);
end;

procedure display_params;
begin
  writeln('INIT:');
  writeln('Res:', res_m : 8, res_s : 8, ' Harvest:', harvest_m : 8, harvest_s : 8);
  writeln('freq_pred:', freq_pred : 4);
  writeln('COMBAT:');
  writeln('Interaction: Peace, Victory, Defeat, War:', peace_div : 4, victory_cost : 4, defeat_cost : 4, war_cost
  writeln('Victory: min_res:', min_res : 8, ' victory_ratio:', victory_ratio : 4);
  writeln('Predator: bell_pred :', bell_pred : 4, ' sup:', superior_ratio : 4);
  writeln('Mobil: ', mobil : 4, ' Grad: ', grad : 4);
  writeln('PERCEPTION:');
  writeln('Forgive:', forgive : 4, ' Provoke:', provoke : 4, ' Min threat:', min_threat, ' Soc:', soc_neg : 6, '/', soc
: 6);
  writeln('ALLIANCES:');
  writeln('Alliances:', alliances, ' Pr_oblig:', pr_oblig : 4);
  writeln('DISPLAY:');
  writeln('width: ', width : 4, ' plot:', plot_mode, ' actor:', current_actor : 4);
end;


procedure init (clist: clist_t);
begin
  if graphics then
    reset_window;
  new_list(alliance_list);
  if first_system then
    first_system := false
  else
    delete_system;
  nx := val(clist.entry[2]);
  ny := val(clist.entry[3]);
  n := nx * ny;
  new_system;
end;
```

```
procedure replicate (clist: clist_t);
 var
  repl, n_repl, start: integer;
begin
 n_repl := val(clist.entry[3]);
 if clist.length = 4 then
  start := val(clist.entry[4])
 else
  start := 1;
 for repl := start to start + n_repl - 1 do
  begin
   seed := repl;
   write('REPL: ', seed : 6, '  ');
   init(make_clist(conc(conc('init ', conv(nx, 10)), conc(' ', conv(ny, 10)))));
   simul(clist);
   if stat_file_open then
    begin
     calculate_statistics;
     record_statistics;
    end
  end;
end;
procedure help;
begin
 writeln('Parameters: set ...');
 writeln('Init: res <m> <s>; harvest <m> <s>; freq_pred <%>');
 writeln('Decisions: bell_pred <%>; sup <ratio>; victory <ratio>');
 writeln('Mobilization: mobil <times>; Resource gradient: grad <k-exp>');
 writeln('Interaction: interact <peace> <victory> <defeat> <war>');
 writeln('Display: width <i>; plot <never/always/event/res/threat/all>; graphics <true/false>');
end;

procedure load (clist: clist_t);
forward;

procedure cli (var f_in: text; loading: boolean);
 var
  s: string_t;
  clist: clist_t;
begin
 repeat
  write(time : 5, '>');
  readln(f_in, s);
  if loading then
   writeln(s);
  clist := make_clist(s);
  if clist.entry[1] = 'init' then
   init(clist)
  else if clist.entry[1] = 'plot' then
   if clist.length = 1 then
    plot(width)
   else
    display(clist)
  else if clist.entry[1] = 'snapshot' then
   snapshot(width, clist)
  else if clist.entry[1] = 'draw' then
```

```
    if clist.length = 1 then
     draw(width, true)
    else
     draw_file(clist)
   else if clist.entry[1] = 'neigh' then
    plot_neighs(val(clist.entry[2]))
   else if clist.entry[1] = 'conq' then
    conquer(val(clist.entry[2]), val(clist.entry[3]))
   else if clist.entry[1] = 'set' then
    set_params(clist)
   else if clist.entry[1] = 'simul' then
    if clist.length = 2 then
     simul(clist)
    else
     replicate(clist)
   else if clist.entry[1] = 'load' then
    begin
     load(clist);
     clist.entry[1] := 'void';
    end
   else if clist.entry[1] = 'help' then
    help
   else if clist.entry[1] = 'data' then
    write_data(val(clist.entry[2]))
   else if clist.entry[1] = 'all' then
    plot_all
   else if clist.entry[1] = 'params' then
    display_params
   else if clist.entry[1] = 'reset' then
    reset_window
   else if clist.entry[1] = 'photo' then
    save_draw(clist.entry[2])
   else if (clist.entry[1] = 'save') or (clist.entry[1] = 'restore') then
    transfer(clist)
   else if (clist.entry[1] = 'clear') then
    reset_window
   else if (clist.entry[1] = 'record') then
    record_file(clist)
   else if (clist.entry[1] = 'stat') then
    begin
     calculate_statistics;
     plot_statistics;
    end
   else if clist.entry[1] = 'exit' then
    writeln('EXIT')
   else if clist.length = 0 then
    begin
     clist.entry[2] := conv(time + 1, 10);
     simul(clist);
    end
   else
    writeln('ERROR');
  until clist.entry[1] = 'exit';
end;

procedure load (clist: clist_t);
 var
```