Description of sample models distributed with Ps-i

Vladimir Dergachev

March 11, 2003

Contents

1 Ps-i model components **1**
- 1.1 Agents and agentclasses . . . 2
  - 1.1.1 Attributes . . . 2
  - 1.1.2 Agentclasses . . . 3
- 1.2 Field . . . 3
- 1.3 Routines . . . 5
- 1.4 Rules . . . 6

2 Identity repertoire models **6**
- 2.1 *Cache* attribute evolution . . . 7
  - 2.1.1 Technical details . . . 9
- 2.2 Model characterization . . . 10

3 Identity repertoire models with border transformation rules **11**

4 Notation **13**

1 Ps-i model components

Ps-i is an environment for design and study of agent-based models.

Each model consists of description of different agent classes, a collection of agents called a *field*, an number of *routines* that compute various charac-

teristics and a number of *rules* that determine how the state of agents in the *field* changes from one simulation step to the next.

The user can also add a description of various quantities to compute and log during each simulation step (*statistics section*) and a description of visual presentation of the *field* of agents (*view section*).

In this paper we concentrate on describing how the model states evolves during simulation. Accordingly the description of *statistics* and *view* sections will be omitted.

1.1 Agents and agentclasses

An *agent* in Ps-i consists of an agentclass type which is represented as an integer number that can attain one of several symbolic values and a number of integer quantities called *attributes*.

1.1.1 Attributes

In the current implementation (Ps-i 3.0.5) each attribute is stored as a 64 bit value which contents are interpreted differently depending on how this quantity is used.

The possible interpretations are: boolean, integer, set and repertoire.

Boolean interpretation uses 0 to designate *false* condition and 1 to designate *true*. Also logical operations (*and, or*) will interpret any non-zero value as *true*.

Integer interpretation uses all 64 bits to make a signed integer.

Set interpretation views the attribute as a set of numbers from 0 to 63.

Repertoire interpretation is a set of numbers from 0 to repertoire size (which cannot be larger than 56) with one of elements marked as *activated* identity. Also each repertoire attribute is assigned a set of global values called *bias*.

All quantities computed by Ps-i are 64 bit and use integer arithmetic. This design decision was made in order to reduce implementation complexity and in anticipation of increased availability of 64-bit processors. The latter, unfortunately, has not been the case - even now (March 2003) both Intel and AMD consumer 64 bit processors have been years delayed and the integer performance of 64 bit workstations is much less than machines using conventional 32-bit x86 architecture.

On the positive side performance-oriented x86 computers typically employ processors that are able to process a lot more data than the memory interface can deliver. Due to the large number of agents employed in a typical Ps-i model and relatively small size of CPU caches the overhead of using 64 bit arithmetic is not as high as could be expected from the increase in instruction count. Also an additional speedup has been gained by using step-to-step optimizing JIT-compiler.

Other motivation for using 64-bit attributes was the ability to use fixed-point numbers (instead of floating point) for computation with adequate precision for most known tasks. This choice allows to bypass the problems of error control while summing many floating point numbers with possibly different orders of magnitude.

1.1.2 Agentclasses

It is probably best to think of an agentclass as a special integer type that is most closely related to *enumerated* type in C.

Unlike C, one can use an agentclass declaration to associate a particular set of attribute values that are used when making a "generic" agent of a particular agentclass. However, Ps-i does not restrict in any way the possible range of attribute values - whatever the agentclass of an agent may be.

Agentclass value is assigned based on the order the agentclasses appear in Ps-i model file. It is therefore recommended that the user always use symbolic constants to check the agentclass of an agent and never explicit integer numbers - this has the benefit of being resistant to insertion of new agentclasses.

Also agentclasses are used to structure evolution rules in a matrix based on which agentclass transforms into which. Besides simplifying construction of model specification this is used to optimize Ps-i model simulation by not evaluating conditions for rules whose source agentclass does not match anyway.

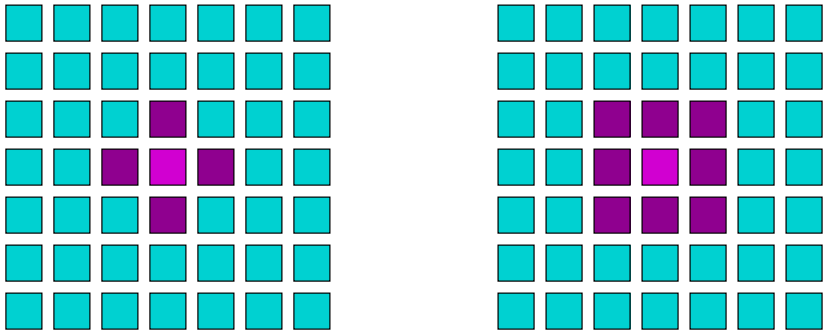
1.2 Field

While it is possible to specify several fields in Ps-i model specification file this feature has not been well developed as none of the models implemented so far required it. In particular, it is not possible at the moment for the agents of different fields to interact.

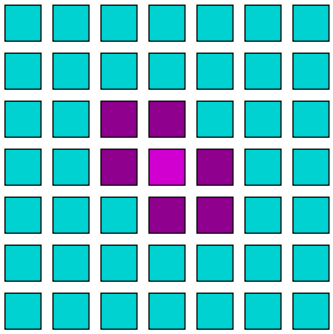
Ps-i field consists of a $N \times M$ matrix of square agents. Note, however that the arrangement of agents in the matrix (and the shape of agents displayed in field viewer) can be viewed as an artifact of using a fixed coordinate system, which is thus specified before any particular model has been written. The agents themselves are not related in any way.

In particular, only a few Ps-i *routines* operate on more than one agent and then the area of operation can be specified by the user. (We will refer to this area of operation as a *neighbourhood* of an agent). By varying the neighbourhood it is possible to realize different field geometries - even though they will still be viewed by Ps-i as a matrix of square agents.

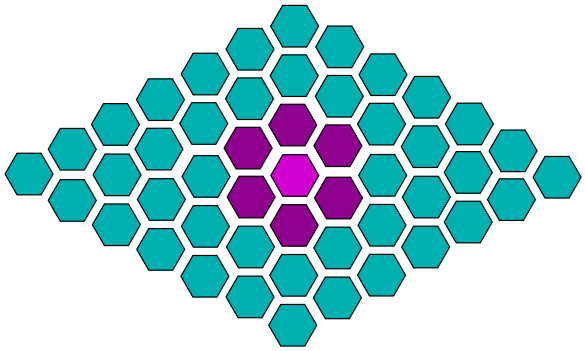
For example, the matrix of square agents is a good representation of field geometry when using the following neighbourhoods:



However, it is not in the following example:



A more accurate picture would be:



The shape of the neighbourhood can vary quite a bit and can even be randomized by use of hash routines.

1.3 Routines

Routines are functions that can be computed given the following data: Ps-i field, current time, model specification and a position of a particular agent on this field (called *current* agent). Evaluation of any routine does not affect Ps-i state. Thus any number of routines can be computed simultaneously without interference.

There are three major types of routines in Ps-i: *composite*, *built-in* and *prototype*.

Composite routines can be viewed as "calculator" expressions - they combine constants, attributes, time and results of computation of other routines. These expressions can be specified at runtime. By design (to prevent accidental lockup of Ps-i) it is not possible to use loops.

Built-in routines are implemented in C and allow access to Ps-i specification and computation of complex expressions. It is important to note that built-in routines are often faster than composite ones. Also the optimizing JIT-compiler recognizes some built-in routines (in particular *rectangle and circle*) which can result in significant reduction of computation time.

Prototype routines are typically used in situations where composite routine would have to use loops and built-in routine would be too restrictive. It is possible to view prototype routines as special-purpose loop constructs that guarantee finite execution (which could still take a long time) and which running time is easy to determine from specification.

Customized routines are particular instances of *prototype* routines.

Parameter and *inline* routines are specially marked *composite* routines - these marks are used by Ps-i user interface and do not affect simulation. (i.e. changing *parameter* to *composite* will result in the same simulation, but the corresponding parameter box will not appear in the "Model parameters" tab).

With the exception of built-in routine *rand* all routines return the same value if called repeatedly during the same simulation step. It is recommended that model specification files use hash routines instead of *rand* where randomness is required. *rand* is better suited to interactive use - for example in *effect* or *selection* tools.

1.4 Rules

During each step of simulation Ps-i computes a new field of agents from previous ones. In order to accomplish this Ps-i checks every agent for existence of a rule with matching source agentclass and condition that evaluates to true.

If such rule is found Ps-i assigns an agent with a target agentclass to the same position in the new field. The attributes of the new agent are set to the values of expressions specified in the rule description - or if the expression for a particular attribute is absent the attribute is simply copied.

If no rule has been found Ps-i copies the agent to the new field as is.

2 Identity repertoire models

Identity repertoire models consist of agents that have attribute *cache* of type *repertoire*. This attribute holds the set of *subscribed* identities of a particular agent and also an integer number number that specifies *activated* identity of this repertoire.

Also the following quantities are typically defined: *influence*, *sight radius*, *inactive* and *immutable*. Depending on a particular implementation they can be either attributes or routines.

A number of agentclasses are defined, in particular *basic*, *entrepreneur*, *border*, *fanatic* and *apathetic*. They all are usually seeded with random repertoires. *Entrepreneurs* commonly have larger sets of subscribed identities and larger influence. *Border* and *fanatic* agents have *immutable* set to *true*. *Border* and *apathetic* agents have *inactive* set to *true*.

Border agents are often used to alter the shape of the field (for example, to construct two weakly connected areas).

Also each agentclass has three quantities assigned to it: *level1*, *level2* and *level3*. These are typically declared as *parameter* routines so the user can easily change them during runtime.

There are several global quantities also declared as parameter routines: *permanent set*, *unobtainable set* and *self influence*.

Lastly, a single evolution cycle is partitioned into several time step and agents of different agentclass react at different steps. Typically entrepreneurs and innovators act first and all other agentclasses act last.

It is important to note that while sample models often have constant values assigned to parameter routines the user can use arbitrary expressions that could, in particular, depend on such dynamic parameters as position and agentclass of an agent. Therefore the distinction between per-agent, per-agentclass and global quantities is a matter of implementation and speed of particular model and can be blurred during runtime.

In a simple variant identity repertoire model the agentclass of an agent is always preserved during an evolution step, as well as agentclass specific quantities such as *influence*, *sight radius*, *immutable* and *inactive*. Thus to describe evolution of the model one describes how the repertoire attribute *cache* changes from step to step.

2.1 *Cache* attribute evolution

The *cache* attribute can be changed in one of three different ways during an evolution step: its *activated identity* can change to a different *subscribed* identity that is already present in the repertoire. Or one (non-*activated*) subscribed identity is replaced with identity not previously present in repertoire. Or the activated identity is replaced with identity that was not present in repertoire.

Which of the actions is taken (if any) is determined by the following algorithm.

First of all one finds the neighbourhood of the current agent that will be polled. This neighbourhood can be determined by the *sight radius* of the current agent and properties of agents with in. Typically agents with *inactive* set to *true* are excluded.

Next, one computes the count of activated identities in the neighbourhood. Each agent contributes the quantity equal to its influence. The central

agent of the neighbourhood (i.e. the agent for which the neighbourhood was computed) contributes *self influence* times its *influence*. If an identity was not activated on any of the agents its count is set to 0.

Each count is increased by *bias* value that corresponds to the same identity (or decreased if the value was negative).

$$count_i = \sum_{\substack{activated[A] = i \\ A \in neighbourhood}} influence[A] + bias_i$$

At this moment the routines that have *differential* in their name subtract from all counts the count of the activated identity of the current agent. This is an optional step that is present to increase computation speed for models that need this effect (it could be achieved by other means, but implementing it here improves both locality and instruction count).

$$differential\_count_i = \sum_{\substack{activated[A] = i \\ A \in neighbourhood}} influence[A] + bias_i - count_{activated[current]}$$

$$\text{if}(differential) \text{then } count_i := differential\_count_i$$

Now is the time to find out which identities could be affected by the operations.

The *discard candidate* is the subscribed identity in the repertoire with the smallest count that is not present in the *permanent set*.

$$discard\_candidate = \text{argmin}_{\substack{i \in repertoire \\ i \notin permanent\_set}} count_i$$

The *swapout candidate* is computed in the same way as discard candidate, but it is not allowed to be the current activated identity.

$$swapout\_candidate = \text{argmin}_{\substack{i \in repertoire \\ i \notin permanent\_set \\ i \neq activated}} count_i$$

The *rotate candidate* is the identity with the highest count among subscribed identities.

$$rotate\_candidate = \text{argmax}_{\,i \in repertoire}\ count_i$$

The *acquire candidate* is an identity that is not subscribed (and thus also not activated) that has the highest count and is not present in the *unobtainable set*.

$$rotate\_candidate = \text{argmax}_{\substack{i \notin repertoire \\ i \notin unobtainable\_set}} count_i$$

The repertoire is then transformed according to the following rules:

First of all if the count of the activated identity is at least as much as any other count the repertoire is left unchanged.

If *rotate candidate* has count at least as much as *level2* it is set to be the activated identity of the repertoire.

Otherwise, if both *acquire* and *discard* candidates have been found and the count of *acquire* candidate is at least *level3* the current activated identity is replaced by the *acquire candidate*.

Otherwise, if both *acquire* and *swapout* candidates are present and the count of *acquire* candidate is at least *level1* the *swapout* candidate in the repertoire (a subscribed identity) is replaced with the *acquire* candidate.

Otherwise the repertoire is left unchanged.

$$new\_repertoire = \begin{cases} count_{activated} \geq \max count_i & old\_repertoire \\ count_{rotate\_candidate} \geq level2 & \text{activate}(rotate\_candidate) \\ count_{acquire\_candidate} \geq level3 & \text{activate}(acquire\_candidate), \\ & \text{unsubscribe}(discard\_candidate) \\ count_{swapout\_candidate} \geq level3 & \text{subscribe}(acquire\_candidate), \\ & \text{unsubscribe}(swapout\_candidate) \\ otherwise & old\_repertoire \end{cases}$$

2.1.1 Technical details

The actual implementation of the above algorithm uses *prototype* routines that need to be customized for a particular model.

Parameters are specified somewhat differently than described above.

First of all the bias values are not subtracted. Rather the prototype routine is passed an expression (called *identity bonus*) that is evaluated for each identity and the result is added to the counts instead of the bias. The identity it is being evaluated for is passed as an argument. This has the advantage of allowing manipulation of bias values on per agent level and allowing implementation of other effects.

Secondly, the poll is computed as if *self influence* was set to 1 and no *self influence* parameter is passed. Rather, the *identity bonus* is used to make the adjustment (that is activated identity of the current agent receives an additional bonus of *self influence* minus 1 times *influence* of the current agent.

Thirdly, there are two different ways to specify neighbourhood polled.

Routines that have *region* in their name take an arbitrary expression and the neighbourhood is defined as all agents where this expression evaluates to true (i.e. non-zero). There is an optimization which recognizes *rectangle, toric rectange, circle* and *toric circle* builtin routines and attempts to estimate the neighbourhood by eliminating all agents for which it is sure to return false.

Routine that do not have *region* in their name use an additional parameter *sight radius* and restrict checking all agents to those within the distance of *sight radius* from the current agent. The optimization is not performed. These are best used with small *sight radius* values where brute force checking of expression that determines the neighbourhood is more efficient than symbolic estimation of its support (region where it is non-zero).

2.2 Model characterization

There are several quantities that are used to characterize identity repertoire models.

Tension is computed for each agent as the number of agents in its neighbourhood that have different activated identity. Note that the neighbourhood need not be the same as one used for repertoire evolution, however using the same neighbourhood for computing both *tension* and new *cache* attribute is a natural choice.

$$tension[A] = \sum_{\substack{B \in neighbourhood[A] \\ activated(B) != activated(A)}} 1$$

Active agent count is the number of agents with *inactive* equal to false.

$$active\_agents = \sum_{inactive(A)=false} 1$$

Total tension is the sum of *tension* values over all active agents (i.e with *inactive* equal false).

$$total\_tension = \sum_{inactive(A)=false} tension[A]$$

Activated and *subscribed* identity counts are the numbers of active agents that are activated on or subscribed to a particular identity.

$$activated_i = \sum_{\substack{inactive(A) = false \\ activated(A) = i}} 1$$

$$subscribed_i = \sum_{\substack{inactive(A) = false \\ i \in repertoire(A)}} 1$$

Total tension can also be computed separately for each set of agents activated on a particular identity.

$$tension_i = \sum_{\substack{inactive(A) = false \\ activated(A) = i}} tension[A]$$

3 Identity repertoire models with border transformation rules

This is a more complex models with additional rules that allow some of the agents to transform into border agents. The condition for the transformation is rather complex and makes uses of global statistics from the previous evolution step.

The implementation of this model is structured as following: the quantities that cannot be computed locally and are necessary to determine whether border transformation rule applies are passed as parameters of the models. The statistics necessary to determine these parameters is declared in the statistics section of the model. In the beginning of each evolution cycle a Tcl/Tk script computes the global parameter from the statistics thus completing the feedback loop.

Aside from statistics mentioned in the previous section (distribution of activated and subscribed identities and distribution of tension count) there is one more statistics needed for border transformation rules - *opposition count*. This is the count of active agents that do not have dominant identity *DI* in their repertoire.

DI is the identity with the largest number of active agents activated on it. It is computed from the distribution of activated identities. This necessitates reevaluation of *opposition count* during execution of Tcl/Tk feedback code.

$$DI = \text{argmax } activated_i$$

Other parameters are:

Herfindahl index *HI* is the sum of squares of proportions of agents activated on a particular identity from all agents. Since Ps-i only uses integer numbers this parameter is expressed in units of 1/10000.

$$HI = \frac{\sum activated_i^2}{(active\_agents)^2}$$

Average tension of the landscape *ATL* is the total tension of the landscape divided by the number of agents sampled to compute it. It is also expressed in units of 1/10000.

$$ATL = \frac{total\_tension}{active\_agents}$$

Average tension index *ATI* describes the average tension of agents activated on particular identity. This is actually a routine which takes one parameter (identity) and returns a value that is expressed in units of 1/10000.

$$ATI_i = \frac{tension_i}{activated_i}$$

Subordinate identity *SI* - this routine returns true for any identity that is not dominant and holds at least 10% of agents activated on it.

$$SI_i = \begin{cases} 1 & activated_i > 0.1 \cdot active\_agents \\ 0 & otherwise \end{cases}$$

Opposing identity *OI* returns true if the identity passed as a parameter has a property that fewer than 20% of agents activated on it are subscribed to the dominant identity *DI*.

$$OI_i = \begin{cases} 1 & opposition\_count_i > 0.8 \cdot activated_i \\ 0 & otherwise \end{cases}$$

We can now describe the condition for the border transformation rule: An agent can transform into border agent if its activated identity is both

subordinate and opposing and its tension level is at least 3. Typically only a random portion of these agents is allowed to transform, this is governed by parameter BC. Also a useful modification is to restrict agents to transform only within a specific region *bt_transform_area*.

$$eligible\_for\_bt[A] = SI_{activated(A)} \cap OI_{activated(A)} \cap (tension[A] \geq 3)$$

The quantities computed by Tcl/Tk feedback code are displayed as part of statistics (which is updated after the code has finished computing) for the convenience of the user. This requires very little cpu time.

4 Notation

The operators argmin and argmax return the value of the argument that minimizes (resp. maximizes) the expression they operate on. There is a question of what happens when two arguments result in equal values of the expression. Ps-i code resolves this by using the smallest argument.