#4430--

Shortest Path in a Multi-Modal Transportation Network: Agent Simulation in a Product of Two State-Transition Networks

Andrew U. Frank,
frank@geoinfo.tuwien.ac.at
TU Wien, Department of Geoinformation and Cartography,
Gusshausstrasse 27-29/E127,
A-1040 Vienna, Austria

Abstract: Location Based Services that assist travelers in wayfinding are a prime application for expert system techniques. The use of public transportation leads nearly always to a combination of different services from different transportation companies (multi-modal transportation). Information systems must combine data for the different services and produce advice to navigate in space and to obtain the right tickets, reservations, etc. This information can be seen as the combination of two (or more) state-transition diagrams: one for the spatial navigation and one for the business (ticketing, validation, reservation) rules.
A (categorical) product combines two state-transition diagrams. The implementation is immediate using an intuitionistic logic reasoner built into the programming language, which infers typing for second order, polymorphic functions and allows their safe execution. The shortest path algorithm in this combined network produces sound advice and reminds the user to acquire tickets and plans the necessary navigation to ticket vending machines, etc.
The analysis shows how to specify the connections between the two graphs optimally. The approach combines typical expert system technologies like inference engines with object-oriented programing; recent advances increase the level of reasoning possible during compilation. The use of a high-level programming language with substantial inference power facilitates the formalization of domain knowledge and is a viable alternative to the classical expert system architecture.

```
"(Playing Chopin's etudes) taught me a lifelong lesson: that phenomena
perceived to be magical are always the outcome of complex patterns of
nonmagical activities taking place at a level below perception. In other
words, the magic behind magic is pattern."
Douglas Hofstadter, SCIENTIFIC AMERICAN, April, 1982, p. 17.
```

1 Improve the Quality of Mobility with Advice to Users of Public Transportation

It is widely accepted that individual car transportation is not a viable solution to increase and improve mobility in European cities, but to entice people to use public transportation is notoriously difficult. The standard argument is that public transportation is not available when and where it is required, which is certainly true for some mobility demands, but investigations show that many potential passengers do not use public transportation even when it is available. The simple reason: potential users do not know when and where public transportation is available and how they could use it; they lack information about schedule and the—often complex—tariff and ticketing rules. Location based services provide the information when and where it is necessary.

In most cases, the information is available and can be found on the web; most public transportation companies publish their schedules and line maps with stops. Public transportation combines in all but the

most simple cases several services from different providers. I considered recently a trip from Vienna to Tangier and a combination of a direct flight to Malaga, a bus to Algeciras, and the ferry to Tangier seemed more attractive than the unreliable multi-stop flight to Tangier. I realized quickly that combining the different schedules and rules was nearly impossible. The schedules and line information for the different transportation companies involved in a trip are not coordinated and the effort to collect the information and patch the trip together is a complicated task. An intelligent method to combine the available data and produce the information necessary for the user for the whole trip—door to door—and including ticketing rules, reservation system, etc. is a perfect application for artificial intelligence.

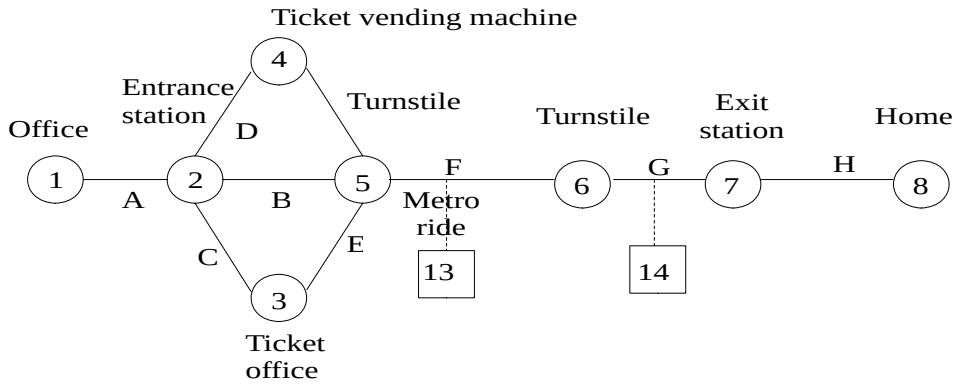

Figure 1: Simplified navigation graph; the circles are states (e.g., bifurcation points) and the edges are actions (locomotions); the boxes connect the actions with states in the business diagram in Figure 2

The scientific goal of Pontikakis (2006) was to understand how to combine spatial navigation (Figure 1) with the "navigation" of the state-transition diagram of the public transportation *business rules* (Figure 2). An agent simulation demonstrated the typical problems; e.g., how to plan locomotion such that travelers stop at convenient places to buy tickets they need later. Travel agents, railway information staff are the experts that know how to paste together the various data pieces found in published schedules. Some expert system exists for such applications, for example the widely used web pages of European national trains services, which use a European knowledge base of scheduled transportation. The knowledge base includes hints about distances and time necessary to change between stations, etc.—lacking are mostly the reservation and ticketing rules.

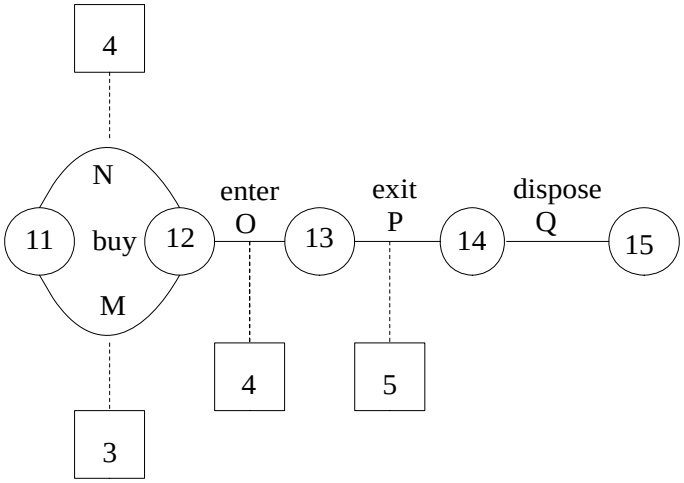

Figure 2: Simplified business graph, which describes actions necessary for a legal ride on Vienna's metro; the boxes indicate the places (from Figure 1) where actions can be carried out.

To construct an intelligent advisor for multi-modal, public transportation requires data about schedules, ticketing, connections between services, etc. and a method to combine these data for use by a shortest path algorithm. The contribution here focuses on an abstract description of the method to combine navigation and business graph; and how it is accessed by the shortest path algorithm. Descriptions of the form in which the data must be provided, or specifications for routines to translate existing data to this form, follow from this high level analysis. The analysis shows how the rules connecting the two graphs, shown as square boxes in figure 1 and 2, are used. The algorithm produces from two graphs plus connecting rules a single navigable graph that can be processed by a standard shortest path program.

2 Expert Systems for Geographic Information

An information system to produce advice for potential users of public transportation is an expert system, which, in its classical architecture, consists of a knowledge base containing formalized expert knowledge and a computerized inference engine to derive conclusions from the stored knowledge. The famous PROSPECTOR (McCammon 1984) expert system was built according to this paradigm to imitate human experts prospecting for minerals; commercially successful were systems to diagnose or configure technical systems. These expert systems used logic reasoning based on forward or backward chaining, e.g., Prolog (Clocksin et al. 1981). It was expected that expert system shells would simplify programming to achieve more "intelligent" programs than were common at the time.

In 1984 we extended our network database Panda (Frank 1984) with a Prolog reasoner to experiment with expert systems in GIS, but found the limitations of logic-based programming too restrictive for most

GIS problems (Frank et al. 1986). The subset of logic used in Prolog (and similar systems) interprets failure to prove as negation, which is a variant of the closed world assumption: Reiter (1984) everything of the world is known and what is not known is assumed to be false (not unknown), this logic is not an acceptable simplification of reasoning for a GIS.

In the 1980s object-oriented programming emerged from the Abstract Data Type research (Goguen 1978) (with Simula an early precursor (Nygaard et al. 1978)). Programming in a computationally complete language avoids the “failure as negation” simplification and allows to separates *false* from *unknown*. Expert knowledge is captured as a lattice of objects with operations (e.g., multiple inheritance of C++ (Stroustrup 1991)). Functional programming languages provide a mathematically rigorous framework (Backus 1978) to focus on the structure of the domain and not so much on the inference. From HOPE (Bailey 1990) via CAML we moved to HUGS (Hugs 2008) and use today Haskell (Peyton Jones 1999) with the customary “Glasgow” extensions.

Experience shows that a second order, polymorphic, lazy, functional language is very well suited to capture expert knowledge: overloaded operations capture the differences in operations, which are selected depending on the type to which the operation is applied. Modeling cognitive agents, which decide on actions based on their simulated perception of their simulated environment, is straight forward (Frank 2000; Raubal 2001; Krek 2002).

3 Navigation Systems

Computerized car navigation systems are probably the most popular application of geographic information systems today. The task is to find the shortest path in a (street) network. Navigation systems are built around one of the first thoroughly analyzed algorithms described by computer pioneer Edsger Dijkstra (1959). Dijkstra's shortest path, or the faster A*, algorithm (Hart et al. 1968) are at the core of the programs running in millions of car navigation systems and produce navigation advice for drivers all over the world.

4 Generalization: Shortest Path in State Transition Diagram

Navigation in a street-network by an agent (human or simulated) consists of a series of actions that each leads to a new state; when a bifurcation point is reached, a decision about the next action is taken, following some rule incorporated in the agent. This schema of *perception—decision—action* is applicable to other problems, where a sequence of steps is necessary to achieve a goal. In using a public transportation system, a passenger follows the graph in Figure 2, which describes the business rules; realistic rules may contain more bifurcations, when users have options, e.g., buying different types of tickets. An optimal sequence of actions can be found with the shortest path algorithm.

5 Combining Two State Transition Diagrams as Graph Product

A manual integration of the navigation and the business sets of rules is possible to produce a single state transition diagram, but is very time consuming when the graphs are large. Figure 3 gives two extremely simple graphs that are combined.

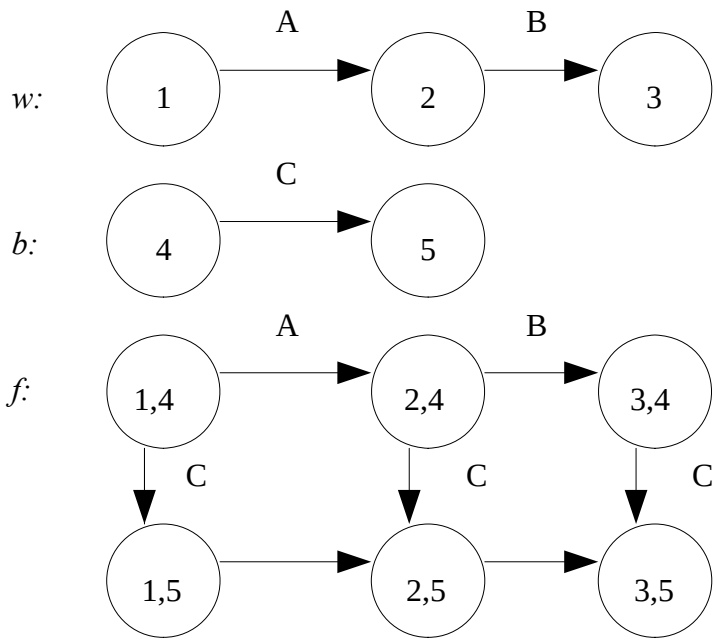

Figure 3: Two simple examples for graph w and b and their combination f.

The number of nodes of the combined graph is maximally the product of the number of nodes in each graph; the complexity of the combination grows rapidly and an automated construction is warranted.

Mathematical category theory gives a general solution that allows the integration of two (or more) state transition diagrams formally. Given two navigation problems with state spaces W (e.g., wayfinding, Figure 1) and B (e.g., business, Figure 2), and locomotion and business actions V and T respective, with state-transition diagrams described by two (partial) functions w and b. These functions are used in the shortest path algorithm to determine for a given node which next nodes are reachable.

$$w: W \times V \rightarrow W$$
$$b: B \times T \rightarrow B,$$

where w is the function encoding the city map with streets and public transportation lines, and b encodes the "business rules" about tickets, how they are obtained, validated, etc. Needed is a combined state-transition diagram with states S and actions P described as function $f: S \times P \rightarrow S$.

The question is how to construct f from w and b? It is possible to express each graph as a table, but I could not find an elegant approach to combining these two tables; it is certainly possible to write a program , but I did not see how to achieve the compact 5 lines of a second order functional language. In mathematical

category theory (Asperti et al. 1991; Mac Lane 1998) the combined state S is written as a product (pair) of the wayfinding and the business states W and B, and the combined actions P as the sum (union) of the locomotion V and business actions T. The combined state is the pair of the individual navigation (states 1, 2, 3 in Figure 3) and business (states 4, 5 in Figure 3) states, the actions are either a navigation (actions A, B in Figure 3) or a business action (action C in Figure 3).

$$S = W \times B$$
$$P = V + T$$

The desired function

$$f: S \times P \rightarrow S$$

becomes

$$f: (W \times B) \times (V + T) \rightarrow (W \times B).$$

To construct f, we need two isomorphisms h and k, which change the pairing, such that

$$h: (W \times B) \times V \rightarrow (W \times V) \times B$$
$$k: (W \times B) \times T \rightarrow W \times (B \times T).$$

From the given functions w and b we construct functions w' and b'

$$w': (W \times V) \times B \rightarrow W \times B$$
$$w' = w \times id \cdot h$$
$$b': (W \times B) \times T \rightarrow W \times B$$
$$b' = id \times b \cdot k$$

and a function j,

$$j: (W \times B) \times (V + T) \rightarrow (W \times B) \times V) + (W \times B) \times T).$$

The function j is an isomorphism in the distributive category of sets and functions (Cockett 1992), which we operate in. Combined this gives for $f = [w' , b'] \cdot j = [(w \times id) \cdot h, (id \times b) \cdot k] \cdot j$. Two or more navigation problems in real space and "business logic" space can be directly combined with this formula and we produce a state transition function that can be used for shortest path and similar algorithms.

The second order functions used here are:

$id: A \rightarrow A$ denotes the function that does nothing: $id\ a = a$;

$(f \times g)$ is the function, which applies f to the first part of a pair and g to the second part of the pair: $(f \times g)\ (a,b) = (f\ a,\ g\ b)$; and

$[f,g]$ means the application of f or g to the part of a sum type: $[f,g]\ (a+b) = f\ (a)\ + g\ (b)$.

A modern, high-level, functional language realizes very closely such expressions. The implementation of the above described solution for combining the two state-transition diagrams translates directly to an executable computer program (*cross* stands for × and *either* for *[]*) the core of which are 5 lines, defining 5 functions:

```
h ((w,b),v) = ((w,v), b)
k ((w,b),t) = (w, (b,t))
w' = cross (w, id) . h
b' = cross (id, b) . k
f = either w' b' . l
```

6. Combination Rules

The representation of the problem of merging two state-transition diagrams must be extended by *a*, the set of rules, which connects the two diagrams: for example the restriction that tickets can only be purchased at the ticket office, or that the turnstile cannot be passed without having a ticket, etc. Figure 1 says that one can only ride the metro (action E in Figure 1) if one is in possession of a ticket (state 13 in Figure 2). The analysis identifies how to express such combination rules and how to extend the approach to include them. The situation is specified in two lists, which contain pairs that give states of one graph and the transition in the other graph it allows, e.g., (13, E). From this list a function to filter possible actions is constructed, following a similar approach as above and using the same "helper" functions *h*, *k*, and *j*. Such rules are a restriction on the function *f* constructed and represented as a Boolean function *cond: (W x B) x (V + T) -> Bool*. The code is:

```
cond = either cw' cb' .l
cw' ((w1,b1), v) = hasNavAffordance (b1,v)
cb' ((w1, b1), t) = hasBusinessAffordance (w1,t)
```

where *hasNavAffordance* and *hasBusinessAffordance* are table lookups.

7. Cost of Actions

The cost of actions are labels on the edges and added in the definition of the graph. Graphs are often represented as list of tuples (*current state, action, next state, cost*); for example (1, A, 2, 4.3) says: from state 1 action A brings us to state 2 with cost 4.3. The functions *w* and *b* are then coded as look-up in these lists and the cost is retrieved. The selection of the path 1-2-3-5 or 1-2-4-5, i.e., buying the ticket at the office or at the vending machine depends only on the relative cost of the actions. In a functional language, cost values can be functions depending on other influences; if the cost for acquiring a ticket at the ticket office in terms of waiting time is high because a long line of people already waiting is present, the algorithm will select the longer walk to the ticket vending machine.

8. Shortest Path

An advice to travelers must plan ahead and have the agent add a detour to acquire a token before the necessity is discovered when standing in front of the turnstile and the agent has to backtrack to a place passed before without acquiring the token. A path 1, 2, 5, 3, 5, 6, 7, 8 (in Figure 1 and 2) is not desirable, but how to identify at point 2 the need for a ticket, which becomes only noticeable to the traveler at point 4? The optimal path in the combined (product) graph is, of course, 1, 2, 3, 5, 6, 7, and 8—produced by Dijkstra's shortest path algorithm automatically! Advice for travelers is produced by an algorithm to find the shortest (optimal) path through the combined graph. An optimal path algorithm requires functions (*[V]* means here a list of *V*)

a: given a state find possible actions: *a : W -> [V]*

w: get next state for given state and action: *w : (W,V) -> W*

c: get cost for action given a state: *c : (W, V) -> C*

The functions *a* and *w* can be derived from *f*. With these functions in place, a shortest path (i.e., with minimal sum of costs) is computed by Dijkstra's shortest path algorithm.

The approach shown here can be used to produce a description of the combined graph automatically, observing the restrictions of the combinations. The size of such a description is the product of the size of the two graphs, reduced by the combination rules. A shortest path algorithm can be using the three functions *a*, *w*, and *c* passed as "call back" functions. The performance advantage is that these functions search one or the other (or both) of the separate graph descriptions; these are much smaller than the combined description.

9. Conclusions

Artificial intelligence is here included in the powerful type checker, which makes programming at this high level of abstraction possible. The development from early Prolog and similar reasoners to the type reasoning system, which is behind the type inference in the code shown is useful to write reliable and general programs quickly (note: not all constructs have programmer written types—most are inferred!). The type system is an extension of the Hindley-Milner (1978) type system based on an intuitionistic (Martin-Löf) type theory (Nordström 1990). These theories were first influential in AI and lead to construction of automatic proof systems, but are now built into the programming languages.

The practical goal to construct portable navigation aid devices for pedestrians using public transportation systems requires the integration of navigation in space with "navigation in business logic". Our approach to use "artificial intelligence" in the form of a very powerful type theory and reasoning system included in the programming language and write the application domain (expert) knowledge in an object-oriented style in a functional programming language was useful to clarify notions related to spatial behavior, in particular wayfinding, based on cognitive agent simulation (Frank 2000; Raubal 2001; Krek 2002), but also in other investigations of spatial cognition (Twaroch 2007). Kuhn has used the same type theory to construct ontologies, which are not only *is_a* taxonomies but define taxa through applicable affordances, represented as operations (functions) in the class (Kuhn 2007).

The use of a mathematically sound theory (category theory) has produced a method to combine relatively simple pieces and to achieve "intelligent" behavior; confirming Hofstadter's motto. The example of using artificial intelligence to build an "expert" system giving advice to travelers does not follow the classical definition of a spatial expert system as knowledge base plus inference engines as I expressed in a series of articles in the mid-1980ies (Frank et al. 1987), but the inference engine is part of the programming language we use. I consider this change in architecture significant and generalizable.

Acknowledgments

I am grateful for the dissertation work of Alenka Krek, Martin Raubal, and Elissavet Pontikakis, which investigated each one of the topics recorded. Till Mossakowski produced the crucial hint to apply distributive categories. I thank the reviewers for constructive and helpful comments.

References

Asperti, A. and G. Longo (1991). Categories, Types and Structures - An Introduction to Category Theory for the Working Computer Scientist. Cambridge, Mass., The MIT Press.

Backus, J. (1978). "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." CACM 21: 613-641.

Bailey, R. (1990). Functional Programming with HOPE. New York, London, Toronto, Sydney, Tokyo, Singapore, Ellis Horwood.

Cockett, J. R. B. and R. A. G. Seely (1992). Weakly Distributive Categories. Applications of Categories in Computer Science: Proc. LMS Symp. on the Applications of Category Theory in Computer Science, Durham, Cambridge University Press.

Clocksin, W. F. and C. S. Mellish (1981). Programming in Prolog. Berlin, Springer-Verlag.

Dijkstra, E. W. (1959). "A Note on Two Problems in Connection with Graphs." Numerische Mathematik(1): 269-271.

Frank, A. U. (1984). Extending a network database with Prolog. Proceedings First International Workshop on Expert Database Systems, Kiawah Islands, SC, Oct. 25.

Frank, A. U., V. Robinson and H. A. Karimi (1987). "Expert Systems for Geographic Information Systems in Resource Management." AI Applications in Natural Resource Management 1.

Frank, A. U. (2000). Communication with Maps: A Formalized Model. Spatial Cognition II (Int. Workshop on Maps and Diagrammatical Representations of the Environment, Hamburg, August 1999). C. Freksa, W. Brauer, C. Habel and K. F. Wender. Berlin Heidelberg, Springer-Verlag. 1849: 80-99.

Goguen, J. A. (1978). Abstract Errors for Abstract Data Types. IFIP Working Conference on Formal Description of Programming Concepts. E. J. Neuhold, North-Holland: 491 - 526.

Hart, P. E., N. J. Nilsons and B. Raphael (1968). "A formal basis for the heuristic determination of minimum cost path." IEEE SSC 4: 100 - 107.

Hugs. (2008). "Hugs 1.4." from http://haskell.systemsz.cs.yale.edu/hugs/.

Krek, A. (2002). An Agent-Based Model for Quantifying the Economic Value of Geographic Information. Vienna, Technical University Vienna. PhD Thesis.

Kuhn, W. (2007). An Image-Schematic Account of Spatial Categories. 8th International Conference, COSIT 2007, Melbourne, Australia, Springer.

Lämmel, R. and E. Meijer (2005). Mappings Make Data Processing Go' round, Redmond, USA, Microsoft Corp.

Mac Lane, S. (1998). Categories for the Working Mathematician. New York, Berlin, Springer.

McCammon, R. B. (1984). User Guide to the USGS Prospector/KAS Expert System. US Geological Survey: 149.

Milner, R. (1978). "A Theory of Type Polymorphism in Programming." Journal of Computer and System Sciences 17: 348-375.

Nordström, B., K. Petersson and J. M. Smith (1990). Programming in Martin-Löf's Type Theory, Oxford University Press.

Nygaard, K. and O.-J. Dahl (1978). "The Development of the SIMULA Language." ACM SIGPLAN Notices 13(8): 245-272.

Peyton Jones, S., J. Hughes and L. Augustsson. (1999). "Haskell 98: A Non-Strict, Purely Functional Language." from http://www.haskell.org/onlinereport/.

Pontikakis, E. (2006). Wayfinding in GIS: Formalization of Basic Needs of a Passenger When Using Public Transportation. Vienna, Technical University Vienna. Doctor of Technical Science.

Raubal, M. (2001). Agent-Based Simulation of Human Wayfinding -A Model for Unfamiliar Buildings. Vienna, Technical University Vienna. PhD.

Reiter, R. (1984). Towards a Logical Reconstruction of Relational Database Theory. On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages. M. L. Brodie, M. Mylopolous and L. Schmidt. New York, Springer Verlag: 191-233.

Stroustrup, B. (1991). The C++ Programming Language. Reading, Mass., Addison-Wesley.

Twaroch, F. (2007). Sandbox Geography How to Structure Space in Formal Models. Wien, Technische Universität Wien. PhD.