Comparison of Well-Structured & Ill-Structured Task Environments and Problem Spaces

Vinod Goel
Institute of Cognitive Science
University of California, Berkeley
goel@cogsci.berkeley.edu

Abstract

Many of our results in the problem-solving literature are from puzzle-game domains. Intuitively, most of us feel that there are differences between puzzle problems and open-ended, real-world problems. There has been some attempt to capture these differences in the vocabulary of "ill-structured" and "well-structured" problems. However, there seem to be no empirical studies directed at this distinction. This paper examines and compares the task environments and problem spaces of a prototypical well-structured problem (cryptarithmetic) with the task environments and problem spaces of a class of prototypical ill-structured problems (design problems). Results indicate substantive differences, both in the task environments and the problem spaces.

Introduction

At the core of any theory of cognition, there will need to be a robust model of reasoning and problem solving. Over the years we have made considerable progress in developing such models (Chandrasekaran, 1983; Duncker, 1945; Ernst & Newell, 1969; Fikes & Nilsson, 1971; Greeno, 1978b; Kleinmuntz, 1966; Newell, 1980; Newell & Simon, 1972; Sacerdoti, 1980; Simon, 1978; Simon, 1983). However, much of this work has been done in the domain of puzzle-type problems, such as cryptarithmetic and the tower of hanoi.

While the investigation of puzzle domains has resulted in very important and significant results, most people share the intuition that there are important differences between solving a cryptarithmetic puzzle and, say, writing a novel or designing a bridge. It is not *a priori* clear that the results from the former will generalize to the latter domains. In fact, there are reasons to believe the contrary.

Reitman (1964), in a seminal paper, argued for a classification of problem types based on the distribution of information in the problem vector. We generally try to capture this distinction in the vocabulary of "well-structured" and "ill-structured" problems. Puzzle games are said to be well structured because the start states, goal states, evaluation functions, and transformation functions are well specified. For example, in cryptarithmetic, the start state is completely specified, as is the goal state. The transformation function, which is also specified, is restricted to two operations: replace a letter with a digit between 0 and 9, and add. Tasks such as writing a novel and designing a bridge are considered ill defined because the start state is incompletely specified, the goal state is specified to an even lesser extent, and the transformation function is completely unspecified.

The distinction is not, however, universally accepted (Simon, 1973), and there seem to be no empirical studies directed at it. In this paper I would like to argue that there is a substantive difference between ill-structured and well-structured problems. I would like to point out that there are a number of crucial differences in the task environments of ill-structured and well-structured problems, and present data indicating corresponding differences in the structure of problem spaces. This paper is a brief summary of work presented in full elsewhere (Goel, 1991; Goel & Pirolli, in press).

The general strategy is to examine and compare prototypical cases of well-structured problems and prototypical cases of ill-structured problems. Cryptarithmetic will be used as an example of a well-structured category, while various forms of design problem solving will be used as examples of the ill-structured category. It may seem odd to restrict the discussion in this fashion, but the strategy has a number of advantages. The design and puzzle game distinction is finer grained, and thus internally more homogeneous. This internal homogeneity will sharpen and highlight any differences across the two categories.

Comparison of Task Environments

There are a number of differences in the structure of the task environments of ill-structured and well-structured problems, in addition to the differences in the distribution of information in the problem vector noted by Reitman (1964). Some which are specific to

design and cryptarithmetic task environments are briefly discussed below.

One very important — but little-noted — difference has to do with the nature of the constraints in the two cases. In cryptarithmetic, as in all puzzles and games, the constraints are logical or constitutive of the task. That is, if one violates a constraint or rule, one is simply not playing that game. For example, if we are playing chess, and I move my rook diagonally across the board, I am simply not playing chess.

However, the constraints we encounter in most nongame situations are of a very different character. Some of these constraints are nomological; many of them are social, economic, cultural, etc. I will encompass the latter category under the predicate "intentional". While there is much to be said about this category, what is important for our purposes is that these constraints are not definitional or constitutive of the task. On the contrary, they are negotiable. For example, if you go to an architect and ask him to build you a new house, and he convinces you to renovate your existing house instead, or to live in a tree in the local park, it seems odd to say that he is not playing the game of design.

Nomological constraints are constraints dictated by natural law. So, for example, if a beam is to support a downward thrust of x psi, it must exert an upward thrust of equal or greater amount. These constraints, while never negotiable, are also not definitional or constitutive of the task. They, in fact, vastly underdetermine design solutions.

Another difference between design and cryptarithmetic problems is one of size and complexity. Cryptarithmetic problems take on the order of minutes to hours to complete. Design problems typically take on the order of days to months to complete.

There are also differences with respect to the lines of decomposition and the interconnectivity of parts. In both cases, the problems decompose into smaller problems. However, in cryptarithmetic, the lines of decomposition are determined by the logical structure of the problem. (So, for example, each row is treated as a component or module.) In design, on the other hand, lines of decomposition are determined by the physical structure of the world, practice within the community, and personal preference.

In terms of the interconnectivity of parts, one finds logical interconnections in cryptarithmetic (i.e., there is always the possibility that any row will sum to greater than 9 and affect the next row). Thus the subject has no choice or selectivity in attending to interconnections. Interconnections in design problems are contingent. This gives the designer considerable latitude in determining which ones to attend and which ones to ignore.

It is also the case that in design problems, as in most nongame situations, there are no right or wrong answers, though there are certainly better and worse answers (Rittel & Webber, 1974). In cryptarithmetic, as in most puzzle games, there are right and wrong answers, and clear ways of recognizing when they have been reached.

In design, as in many real-world tasks, there are consequential costs associated with errors. Resources and lives are often at stake. In cryptarithmetic, as in most games, errors may cause some embarrassment to the subject, but that is about the extent of the "damage."

Lastly, in design problems, as in many real-world situations, there is no immediate feedback from the world. Hence, it must be simulated, or self-generated. This requires considerable resource allocation for modeling and performance predicting. In cryptarithmetic there is genuine feedback after every operator application. It is, however, local feedback, and the final solution needs to satisfy global constraints.

This list is meant to be neither unique nor exhaustive. It is meant to indicate that there are a number of *substantive differences* in the task environments of at least some well-structured problems (cryptarithmetic) and some ill-structured problems (design problems). Given the logic of information processing theory, such differences should have psychological consequences at the level of the problem space. The balance of the paper describes a study which explores and articulates some of the differences in design and cryptarithmetic problem spaces.

Methodology and Database

The results presented here are based on single subject protocol studies (Ericsson & Simon, 1984). A total of sixteen protocols, twelve from design situations, and four from puzzle-game situations, were examined and compared. The design protocols were gathered from expert designers from the disciplines of architecture, mechanical engineering, and instructional design. The four puzzle protocols were from the domains of cryptarithmetic and the Moore-Anderson Tasks.[1] They were extracted from Newell and Simon (1972). The methods of collection and analysis of the data are described below. The results of the analysis of three of the design protocols — one from each discipline — and two cryptarithmetic protocols, are presented below.

Design Protocols

Subjects, Tasks, and Procedure: As noted above, the design protocols were collected from professional designers from the disciplines of

[1]The Moore-Anderson Task is a string transformation task isomorphic to theorem proving in the propositional calculus.

architecture, mechanical engineering, and instructional design. The architectural task called for the design of a self-help automated post office for the UC-Berkeley campus. The mechanical engineering task required the design of an "automated postal teller machine" for the above post office. The instructional design task was unrelated. It involved the design of a self-contained instructional package to teach lay people a reasonably complicated computational environment.

The procedures for collecting the protocols were the same in each case. Each subject was given a one-page design brief, and any related documents, and asked to specify a solution to the problem, to the degree of specificity allowed by time and resource constraints. They were allowed to use any external drawing aids/tools that they desired. All chose to use paper, pencil and/or pen.

The durations of the sessions varied from two to three hours. The experimenter was present to answer any questions relating to the experiment, and otherwise assume the role of the client. Subjects were encouraged to ask clarification questions as the need arose. The experimenter answered all questions, but at no time initiated the conversation.

Subjects were asked to "talk aloud" as they solved the problem. They were cautioned against trying to explain what they were doing. Rather, they were asked to vocalize whatever was "passing through their minds" at that time. Most of the subjects did not have much difficulty in doing this. Where subjects did lapse into periods of silence, the experimenter prompted them by asking "what are you thinking now?".

The sessions were videotaped. The tapes, along with the written and drawn material, constituted the data.

Coding Scheme: The protocols were transcribed, cross-referenced with the written material, and coded. The coding involved breaking the protocols into individual statements representing single "thoughts" or ideas. Content cues, syntactic cues, and pauses were used to effect this individuation. This resulted in very fine-grained units with a mean duration of eight seconds and a mean length of fifteen words. Each statement was coded for the operator applied (e.g. add, delete, justify, etc.), the content to which the operator was applied, the mode of output (verbal or written), and the source of knowledge (design brief, experimenter, self, inference).

These statements were then aggregated into modules and submodules, which are episodes organized around artifact components. For example, for the architect subjects, the modules were components like site, building, and services. The site submodules were components like circulation, landscaping, and site illumination. The building submodules included such things as doors, roof, and mail storage. The modules were then further aggregated into design-phase levels. The design-phase level coded for several things, the most important being design development phases such as problem structuring, preliminary design, refinement, and detail design. These categories were further coded for the aspect of design development attended to (e.g. people, purposes, behavior, function, and structure).

This resulted in a reasonably complex three-level, hierarchical scheme, similar in spirit, but not detail, to the one employed by Ullman, Dietterich, and Stauffer (1988). It is fully detailed elsewhere (Goel, 1991; Goel & Pirolli, in press).

Cryptarithmetic Protocols

Subjects, Tasks, and Procedures: The two cryptarithmetic protocols were gathered from published sources (Newell & Simon, 1972, Appendices 6.1, 7.1), recoded, and compared with the design protocols. These particular ones were chosen on the basis of their duration. The subjects for these studies were undergraduate students. The procedure of collecting the protocols was similar in relevant aspects to the one described above.

The task for both subjects (NS6.1 and NS7.1) was the following problem:

```
   DONALD          D=5
 + GERALD
 ---------
   ROBERT
```

Each letter stands for a digit. The digits encoded as DONALD and GERALD add up to the digits encoded as ROBERT. The task is to transform the letters into the appropriate digits. The clue given is that D=5.

Coding Scheme: The protocols were re-coded with a modified subset of the scheme devised for the design protocols. Three changes were required. First, it was found that while one could differentiate between problem structuring and problem solving, it was not possible to further differentiate problem solving into preliminary, refinement, and detail phases. Second, the aspect of design development category was not applicable. Third, information about mode of output was not available.

Comparison of Problem Spaces

This section discusses some of the characteristics of design problem spaces and notes how they differ from cryptarithmetic problem spaces. The results are presented in more detail elsewhere (Goel, 1991; Goel & Pirolli, in press).

Stopping Rules and Evaluation Functions: The stopping rules and evaluation functions in design problem spaces are determined by the designer rather than the structure of the problem. The decisions are based on personal preference and experience, professional standards and practice, and client expectations. The personalized nature of the stopping rules and evaluation functions can be explained by appealing to three factors in the task environment. First, there is not enough information in the problem statement to make these decisions. Second, there are no right and wrong terminating states. Third, there are few, if any, logical constraints.

In cryptarithmetic, the stopping rule is explicitly supplied and evaluation functions, at least locally, are determined by the structure of the problem. The issue of personal preference just does not enter the picture.

Memory Retrieval & Inferences: On a related front, a very small percentage of statements in design protocols (1.3%) is generated by overt deductive inferences. Most seem to be the result of memory retrieval and modification and/or nondemonstrative inference. The cryptarithmetic problem spaces had a much higher percentage of statements generated by demonstrative inference (41%).

This large difference in deductive reasoning is what one would expect, given the structures of the task environments. Deductive systems require logical constraints. As already noted, the constraints on cryptarithmetic are logical, while the constraints on nongame tasks, like design, are nonlogical.

Direction of Transformation Function: In well-structured domains, the transformation function maps the start state onto the goal state. In the design problem spaces, it was noted that the subjects would stop and turn things around. That is, they would try to manipulate the problem constraints and client expectations so as to change the start state to one which better fits their knowledge, experience, and expertise. One might call this phenomenon "reversing the direction of the transformation function," because the subject has prior knowledge of some goal state and is trying to transform the problem parameters to fit that goal state.

Again, the reason that this can occur is that the problem is incompletely specified, and the constraints are nonlogical, therefore manipulable. It cannot, and does not, occur in cryptarithmetic because of the logical nature of the constraints. Any attempt by the subject to change the parameters would be viewed as an inability or lack of desire to participate in the assigned task.

Solution Decomposition: Another interesting difference across the two problem spaces has to do with solution decomposition. There are two interesting findings. First, design problems are decomposed into many more modules than cryptarithmetic modules, and second, the density of interconnections between modules is higher in the cryptarithmetic case than the design case.

For example, subject S-A, working on the architectural task of designing a post office, decomposed the solution into 34 modules corresponding to structural and functional components such as roof, door, location of equipment, flow of traffic, etc. Given 34 modules, 1,122 interconnections are logically possible. The subject actually made only 7.4% (83.03) of these connections.

In cryptarithmetic, on the other hand, the problems were decomposed into 6 modules (corresponding to the six columns). But while the actual number of modules were fewer, the density of interconnections between modules was considerably greater. Subject NS6.1, for example, made 20% (6) of the logically possible connections.

The denser interconnectivity of the cryptarithmetic modules is what one might expect, given that they are intended to be multiple constraint satisfaction problems, and all the constraints are logical (so must be attended to). This is perhaps why such problems can have so few components and still be challenging. The reason design problems can have so many components and still be tractable is that the interconnections are contingent rather than logical. This gives the designer considerable flexibility in determining which one to attend to and which ones to ignore.

Development of Solution: Yet another interesting difference has to do with the incremental development of solutions in design problem spaces. One of the most robust findings in the literature of design problem solving is that, as design solutions are generated, they are retained, massaged and nurtured to completion (Kant, 1985; Ullman et al., 1988). They are not easily discarded.

A number of aspects of the design task environment favour such a strategy. First, there is the obvious fact that the problems are large, and given the sequential nature of human information processing, cannot be completed in a single cycle. Second, since there are few logical constraints to be violated, and no right or wrong answers, there is little reason to give up on a partial solution to start again from scratch. Third, incremental development is compatible with the "least-commitment" control strategy used by designers (see below).

In contrast, traversal of cryptarithmetic problem spaces have an all-or-nothing character about them. Most paths searched are wrong and independent of the correct path(s). Thus there is no sense of building up to a solution. Once a path is searched, and it turns out not to be on the solution path, the subject is no better off than before the search began. He must start again on another path.

Control Structure: There are also a number of interesting differences with respect to control strategies in the two cases. The design subjects used a control

strategy, not unlike the "least-commitment" control strategy identified by Stefik (1981). The basic feature of this strategy is that, when working on a particular module, it does not require the designer to complete that module before beginning another. Instead, one has the option of putting any module on "hold" to attend to other related (or even unrelated) modules, and returning to the first at a later time. This embedding can go several levels deep.

The control structure of the design subjects is naturally analyzed into three hierarchical levels: movement from module to module, movement from submodule to submodule, and movement internal to submodules. The first two levels are task-specific; that is, the modules and submodules vary from task to task. The third level, however, generalized across all three design tasks. The control structure within any level is repetitive, cyclical, and flexible. One effect of this repetition and reiteration is that most modules and submodules are considered in more than one context.

The cryptarithmetic strategy was interestingly different in some respects. While one could trace a cyclical, repetitive control structure, as in the design case, most of the problem solving occurred internal to modules/episodes. There was little carryover from previous visits to a module/episode. In fact, Newell and Simon (1972), in their original analysis of these protocols, claimed that in returning to a former state, the subject is in fact returning to a previous knowledge state with respect to the problem. If the subject goes down the wrong path and returns to the previous state, all that he knows is that the path just explored does not lead to the goal state. He does not have an enriched understanding of the state he is returning to. The complete control structures of a design and cryptarithmetic subject are traced out in Goel (1991).

Making & Propagating Commitments: While the least-commitment control strategy allows design subjects to keep options open, the solution must ultimately be brought to closure. This requires that one make and propagate commitments through the problem space. In the cryptarithmetic protocols, while commitments are certainly made, they are propagated only until a local evaluation function accepts or rejects them.

The last aspects of design and cryptarithmetic problem spaces that I would like to discuss have to do with the phases of solution development.

The development of a design solution has several distinct phases. Four of these phases are: problem structuring, preliminary design, refinement, and detailing. These phases differ with respect to the type of information dealt with, the degree of commitment to generated ideas, the level of detail attended to, and the number and types of transformations engaged in.

Problem structuring is the process of retrieving information from long-term memory and external memory and using it to construct the problem space; i.e., to specify start states, goal states, operators, and evaluation functions. Problem structuring relies heavily on the client and design brief as a source of information, considers information at a high level of abstraction, makes few commitments to decisions, and involves a high percentage of add and propose operators.

Preliminary design is a classical case of creative, ill-structured problem solving. It is a phase where alternatives are generated and explored. Alternative solutions are not, however, fully developed when generated. They emerge through incremental transformations of a few kernel ideas. These kernel ideas are images, fragments of solutions, etc. to *other* problems which the designer has encountered at some point in his life experience. Since these "solutions" are solutions to other problems which are being mapped onto the current problem, they are, not surprisingly, always out of context or in some way inappropriate and need to be modified to constitute solutions to the present problem.

This generation and exploration of alternatives is facilitated by the abstract nature of information being considered, a low degree of commitment to generated ideas, the coarseness of detail, and a large number of lateral transformations. A lateral transformation is one in which movement is from one idea to a slightly different idea, rather than a more detailed version of the same idea. These transformations are necessary for the widening of the problem space and the exploration and development of kernel ideas.

The *refinement* and *detailing phases* are more constrained and structured (though still very different from puzzle games). They are phases where commitments are made to a particular solution and propagated through the problem space. They are characterized by the concrete nature of information being considered, a high degree of commitment to generated ideas, attention to detail, and a large number of vertical transformations. A vertical transformation is one in which movement is from one idea to a more detailed version of the same idea. It results in a deepening of the problem space.

While these phases of design development may seem trivially obvious, they are rendered interesting by the fact that cryptarithmetic problem spaces cannot be individuated into similar phases. As already noted, in such game problems one gets more of the same activity. Either one is on a path which will abruptly lead to the solution, or one is not. There is no sense in which one builds up to a solution.

Conclusion

I have presented arguments and data to suggest that there are interesting differences in the task environments of (at least some) well-structured and (at

least some) ill-structured problems, and that these lead to some nontrivial differences in ill-structured and well-structured problem spaces.

The reader may have noted, however, that the comparison of problem spaces is not carried out at the level of states and operators, which is the level at which problem spaces are generally defined. It is conducted at a much more abstract level. The fact of the matter is, if one compares the problem spaces at the level of states and operators, it is difficult to differentiate the two problem spaces. It is only when one abstracts away from the low-level details — the sequence of states and operators — that the differences emerge. However, the fact that they emerge at this more abstract level does not make them any less real or interesting. On the contrary, generalizations at this level may serve to fill the theoretical gap that some argue exists in information processing theory between implementations of specific problem spaces and the general notion of an information processing system (Chandrasekaran, 1983; Goel & Pirolli, 1989; Greeno, 1978a).

Acknowledgements

The author is indebted to Peter Pirolli, Susan Newman, and Mimi Recker for helpful discussions and comments. This work has been supported by a Gale Fellowship, a Canada Mortgage and Housing Corporation Fellowship, a research internship at System Sciences Lab at Xerox PARC and CSLI at Stanford University, and an Office of Naval Research Cognitive Science Program grant (# N00014-88-K-0233 to Peter Pirolli).

References

Chandrasekaran, B. (1983). Towards a Taxonomy of Problem Solving Types. *AI Magazine, winter/spring*, 9-17.

Duncker, K. (1945). *On Problem Solving*. Westport, Connecticut: Greenwood Press.

Ericsson, K. A., & Simon, H. A. (1984). *Protocol Analysis: Verbal Reports as Data*. Cambridge, Massachusetts: The MIT Press.

Ernst, G. W., & Newell, A. (1969). *GPS: A Case Study in Generality and Problem Solving*. N.Y.: Academic Press.

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence, 2*, 189-208.

Goel, V. (1991) *Sketches of Thought: A Study of the Role of Sketching in Design Problem Solving and its Implications for the Computational Theory of Mind*. Ph.D. Dissertation, University of California, Berkeley.

Goel, V., & Pirolli, P. (1989). Motivating the Notion of Generic Design within Information Processing Theory: The Design Problem Space. *AI Magazine, 10* (1), 18-36.

Goel, V., & Pirolli, P. (in press). The Structure of Design Problem Spaces. *Cognitive Science*.

Greeno, J. G. (1978a). Natures of Problem-Solving Abilities. In W. K. Estes (Eds.), *Handbook of Learning and Cognitive Processes, Volume 5: Human Information Processing*. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Greeno, J. G. (1978b). A Study of Problem Solving. In R. Glaser (Eds.), *Advances in Instructional Psychology, Vol. 1*. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Kant, E. (1985). Understanding and Automating Algorithm Design. *IEEE Transactions on Software Engineering, 11*, 1361-1374.

Kleinmuntz, B. (Ed.). (1966). *Problem Solving: Research, Method, and Theory*. NY: John Wiley.

Newell, A. (1980). Reasoning, Problem Solving, and Decision Processes: The Problem Space as a Fundamental Category. In R. S. Nickerson (Eds.), *Attention and Performance VIII*. Hillsdale, N.J.: Lawrence Erlbaum.

Newell, A., & Simon, H. A. (1972). *Human Problem Solving*. Englewood Cliffs, N.J.: Prentice-Hall.

Reitman, W. R. (1964). Heuristic Decision Procedures, Open Constraints, and the Structure of Ill-Defined Problems. In M. W. Shelly & G. L. Bryan (Eds.), *Human Judgements and Optimality*. N.Y.: John Wiley and Sons.

Rittel, H. W. J., & Webber, M. M. (1974). Dilemmas in a General Theory of Planning. *DMG-DRS Journal, 8* (1), 31-39.

Sacerdoti, E. D. (1980). Problem Solving Tactics. *AI Magazine, 2* (1), 7-15.

Simon, H. A. (1973). The Structure of Ill-Structured Problems. *Artificial Intelligence, 4*, 181-201.

Simon, H. A. (1978). Information-Processing Theory of Human Problem Solving. In W. K. Estes (Eds.), *Handbook of Learning and Cognitive Processes, Vol.V.* (pp. 271-295). Hillsdale, N.J.: Lawrence Erlbaum Associates.

Simon, H. A. (1983). Search and Reasoning in Problem Solving. *Artificial Intelligence, 21*, 7-29.

Stefik, M. (1981). Planning and Meta-Planning (Molgen: Part 2). *Artificial Intelligence, 16*, 141-170.

Ullman, D. G.,Dietterich, T. G., & Stauffer, L. A. (1988). *A Model of the Mechanical Design Process Based on Empirical Data* (Tech. Report No. DPRG-88-1). Dept. of M.E., Oregon State University.