OPL: An Environment for Posing and Solving Discrete Optimization Problems

Bruce MacLeod
Robert Moll
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 91-24
April 1991

Abstract

In this work we report on a general and extensible framework, called OPL, for quickly constructing reasonable solutions to a broad class of complex discrete optimization problems. Our approach rests on the observation that many such problems can be represented by linking together variants of well-understood primitive optimization problems. We exploit this representation by building libraries of solution methods for the primitive problems. These library methods are then suitably composed to build solutions for the original problem.

To demonstrate the efficacy of the approach we report here on several OPL applications, including a vehicle routing problem and a real-time computer task and resource scheduling problem.

1 Introduction and Background

In this paper we report on a notation and computing environment for representing and solving complex combinatorial optimization problems. The problems we consider arise in such diverse areas as scheduling, vehicle routing, layout, and warehousing.

Our notation and solution methodology is called OPL (for Optimization Programming Language). In OPL problems are represented by means of a graph structure called an HCG (Hierarchical Containment Graph). Nodes in a graph represent either primitive objects or organized collections of primitive objects. The ways in which primitive objects can be organized reflect the fundamental data structures that appear in many combinatorial optimization problems, e.g. bounded length lists, unordered sets, rings. An arc represents a function that associates objects in the source node with objects in the target node. This mapping represents a fundamental (and reasonably well-understood) optimization problem involving objects that appear in the source and target nodes associated with the arc.

OPL achieves its power by means of an abstraction mechanism, which allows a user to create a library of solution methods for the fundamental optimization problems that are associated with the arcs of an HCG representation. Such solution methods are generally approximation algorithms or local search routines. For example, a suitably constrained map from primitive objects to ring structures can model the traveling salesman problem; it can be optimized, for example, using library versions of a nearest neighbor heuristic – an approximation routine, and Lin and Kernighan's 2-opt – a local search heuristic. Many library routines have been defined as part of the OPL environment and their use involves simple calls to the appropriate library function.

The *language of policy programs* is the solution component of the OPL environment. Given an optimization problem and an associated HCG, a user creates solutions – that is, policy programs – which involve calls to library routines appropriate to the primitive optimization problems identified in the problem's HCG. OPL allows these library calls to be embedded in traditional programming language constructs (if, while, ...). An important feature of the language of policy programs is its extensibility. New fundamental data structures and new primitive optimization problems can be defined, and given a primitive optimization problem, new solution methods can be created and added to the solution library. An OPL prototype has been written in Common Lisp. It has been used to solve problems in vehicle routing, warehouse layout, real time scheduling, and check clearing operations in a bank [24, 25, 26].

Other advanced software systems have been proposed for solving problems in Operations Research. Software environments for mathematical programming include AMPL [7], GAMS [1], and Platform [27]. Software systems based on an intelligent search of the solution space have also been developed. These include the technique of Global Search [33] and the ALICE system [21]. The problem of developing languages for describing Operations Research problems has also recieved some attention. The structured modeling notation, described in [8, 11], supports the descriptions of a variety of computational and data processing issues that arise in Operations Research and Management Science. NETWORKS is a modeling system which is based on graph grammers [18]. It allows users to specify the characteristics of a problem instance and a problem class interactively. The OPL notation bears some similarity to the above two modeling systems. All three approaches use graphs to represent relationships that exist between principal entities that make up a particular OR problem. However, the focus of our work is on problem solving. For a more detailed review of advanced Operations Research software systems see [9].

This paper describes the OPL environment and the results of applying OPL to three different problems. In Section 2 we provide an informal description of the OPL notation and solution machinery. Section 3 uses the classroom scheduling problem to help describe the process of HCG construction. In Section 4 we consider the style and effectiveness of OPL policy programs by comparing our performance on the multiple depot vehicle routing problem with published results. Section 5 considers a real-time computer resource allocation and scheduling problem. General observations from this line of investigation are given in the concluding section.

2 Description of OPL and a Simple Example

2.1 OPL Representation Machinery

OPL's problem representation machinery is motivated by four goals:

- Problem representations should be natural;
- Representations should lead to problem solutions that can be constructed using software libraries;

- Natural representations should lead to high quality solutions; and
- The representation system should be extensible.

These goals are achieved using hierarchical constraint graphs, or HCG's, to structure the description of optimization problems.

In an HCG, nodes correspond to a set of objects of a particular fundamental type (or a tuple of such types). Such types can be primitive, indivisible objects, or container types, which are types that "hold" other objects. Containers can hold objects in different ways; the range of possibilities reflects the class of fundamental data structures that appear in a great many combinatorial optimization problems, e.g. bounded length lists, unordered sets, rings. An arc represents a function that associates objects in the source node with objects in the target node. We view such a map as representing a fundamental (and reasonably well-understood) optimization problem that determines the mapping between the data structures identified in the source and target nodes associated with the arc. As an example, Figure 2.1 indicates that objects of type X are to be organized in objects of type Y. An icon, in this case ⊞ indicates the structural organization of X objects in the Y objects.

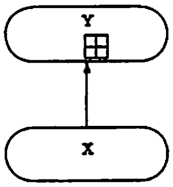

Figure 1: A Simple HCG

Often such a relationship can be thought of as asserting: " map each object of type X into a structured container of type Y". Indeed, we will frequently use "object-container" terminology when we refer to the elements of our generalized assignment mappings. If an object is a container, we assume that its internal positions have a unique numerical value. If p belongs to container structure Y, then position(p) gives this location.

The system's current existing container types are described below. Additional types are easy to add.

- *unordered sets*, ⊞ is a structure that is used for partition/clustering problems.

- *ring (bounded or unbounded)*, ∴ , describes an organization of objects into a ring data structure. These structures are used for TSP style routing problems.
- *slotted object*, ▥ is used for representing discrete schedules, as well as for representing other "slotted" situations that are common in optimization problems, e.g., parking trucks in loading bays, plugging computer boards into backplanes, or assigning goods to particular bins in a row in a warehouse.
- *continuous interval*, ▭ this structure is used primarily to represent the placement of objects (or tasks) in continuous time.

An HCG arc from X to Y represents a function from objects of type X to objects of type Y. The most important kind of map is called an **installation map**. Such maps place objects from X "inside" one of the structures in node Y. Suppose, for example, that in some optimization problem we wish to assign packages to trucks. We may represent this assignment with an installation map from X to Y, where X consists of primitive objects – the packages, and Y consists of unordered sets – the contents of each truck. Mapping p $\in$ X to one of the unordered sets of Y represents the placement of p in the corresponding truck. We allow the target node of an installment map to have one further attribute. The node may have either a fixed number of objects – trucks, for example – or we may allow the node to have a growing number of objects, which are constructed by a "generator" function associated with the node.

A second kind of map from node Z to node W, called an **attribute map**, associates objects of type Z with objects of type W in a 1-1 fashion. Thus if Z represents faculty members at a university as primitive objects, and W is a collection of slotted objects which are to be interpreted as faculty schedules, then an attribute map from Z to W in effect associates each faculty member with a unique schedule.

Finally we allow the following kind of map, called a **collapsing** map. Suppose node M consists of multiple objects of type S, and suppose node N consists of a single object of type S. Then the map c: M $\longrightarrow$ N is collapsing if, for m $\in$ O, O $\in$ M, position(m,O) = position(c(m),N). The primary role of collapsing maps, as we shall see in sections 3 and 5, is in scheduling problems.

2.1.1 Kernel Optimization Problems and Abstraction

An installation map from primitive objects in X to, say, unordered set objects of type Y, is, in a sense, an incomplete object: no objective function has been supplied, and no predicates have been identified that constrain the assignment. This situation is remedied by supplying an objective function, a primary constraint, and, when necessary, a collection of secondary or minor constraints. In our trucking example, for instance, a weight limit may constrain the assignment of packages to trucks – and weight might therefore be the primary constraint of the problem. An attribute such as refrigeration – does a package require refrigeration and is a particular truck refrigerated? – would then function as a minor constraint. A typical objective function for this example might be to minimize the number of trucks necessary to carry a particular load.

By attributing such objective function and constraint information to an installation map, a user is frequently constructing a map that represents a familiar and well-understood optimization problem. It is thus convenient to introduce an abstraction facility. That is, given an installation map
i: M $\longrightarrow$ N with objective function f and major constraint c, we write (define-map NAME (Source-type Dest-type f c)) to indicate that the map named NAME is an object with the indicated attributes. We ignore minor constraints in the naming process. Thus, in our trucking example, we might name the induced problem generalized-bin-packing (in the case where the goal is to minimize the number of trucks needed.)

Finally given an optimization problem that has been formulated as an abstraction as described above, we allow for the creation of problem solving methods that are appropriate for each type of problem created and which are bound to that named map. Thus, after creating generalized-bin-packing, such familiar bin-packing approximation algorithms as first-fit, next-fit, and best-fit can be defined and installed in a library of routines that are applicable to the generalized-bin-packing named map. We also allow local search algorithms. Thus Kernighan and Lin's 2-opt [22] algorithm for local search is bound to a ring-placement installation map.

Below we identify seven named maps between objects and container structures. In each case we supply an icon that denotes the organization of objects in the container.

- *Generalized Bin Pack (GBP)*, ⊞, seeks an assignment of objects to container "bins" that minimizes the number of containers needed. Containers have a capacity constraint and objects have a size with respect to the containers.

- *Capacitated Partition (CP)*, ⊞, seeks a feasible partition of objects into disjoint collections of unordered subsets. Containers have a scalar capacity constraint which limits their carrying capacity. Note that in this case the objective function is constant.

- *Generalized Graph Partition (GGP)*, ⊞, seeks a minimum cost partition of objects. Costs are incurred if two objects are not located in the same container. The cost of separating object i from object j is given by the ij^{th} entry in a cost matrix.

- *Traveling Salesman (TSP)*, ⁘, maps objects to one of several unbounded rings. Tour cost is calculated by adding the "distances" between adjacent objects in the tour. A distance matrix provides the distance between pair of objects. A tour of minimum cost is sought.

- *Discrete Scheduling (DS)*, ▥, seeks a non-overlapping placement of objects in slots in a slotted container structure. Objects (tasks) to be scheduled occupy a fixed number of consecutive slots, and containers have a fixed number of slots in which to place objects.

- *Interval Scheduling (IS)*, ◫, seeks to minimize the maximum time on any schedule. Each object (task) requires a fixed amount of processing time and schedules may have a maximum processing time.

- *Fixed Assignment (FA)*, implements an assignment that is predetermined. Each object has the name of the container and the position in that container that it must be placed in. There is no optimization or constraint data associated with this relation. Objects can be organized according to any of the primitive structures of OPL.

The goal, then, of OPL is to formulate an optimization problem using named installation maps. Once an HCG of this kind has been built, then an overall solution can be constructed by suitably composing the library methods that are bound to the named maps appearing in the HCG.

2.2 Policy Programs

The OPL programming primitive that provides access to library functions of named installation maps is called an *improvement policy.* An improvement policy consists of a sequence of approximation and/or local search library routines. Each element of the sequence is associated with an arc that is present in the HCG. An improvement policy seeks to place or rearrange a collection of objects in such a way that the new solution extends or improves the old solution. More concretely, an improvement policy consists of:

- a collection of objects, called *primary objects*, which the improvement policy acts upon.
- a *dispatch function* which, each time it is called, chooses the next primary object to be considered by the improvement sequence.
- a sequence $(t_1,...,t_k)$ of transformations called an *improvement sequence*. Each t_j is an approximation or local search library routine that is associated with one of the arcs (and thus, one of the named maps) in the HCG.
- a collection of HCG arcs called *bound* arcs, which identify those assignments in an existing partial solution that may not be altered by the improvement sequence.

An improvement policy uses the dispatch function to choose one primary object from the collection of primary objects. Transformations in the improvement sequence are applied to the primary object if the primary object is from the same class as the tail node of the transformation arc; otherwise, the transformation is applied to the appropriate relatives of the primary object. *Relatives* of object A are all objects which have been assigned to A (either directly or indirectly) or which A has been assigned to (again, either directly or indirectly). The transformation is applied to all relatives of the primary object which have the same class as the tail node of the arc. The transformation function can assign or modify the assignments of an object as long as assignments associated with bound arcs are not altered.

A transformation terminates when all allowable placements and rearrangements have been considered. Backtracking occurs when a transformation cannot locate a successful candidate. In this case, the OPL machinery backs up to the previous transformation and attempts an alternative assignment or rearrangement.

A *policy program* is an optimizing algorithm for a problem instance that has improvement policies as primitives, and includes, in addition, elementary programming constructs (if, do-while, for, ..). Data structuring functions (sorting, extracting, merging, ...) are also available.

2.3 An Introductory Example

We demonstrate the workings of OPL by illustrating its application to instances of vehicle routing, a classical problem in Operations Research.

In the simplest version of the Vehicle Routing Problem (VRP), vehicles deliver packages to a collection of geographically dispersed customers. Each vehicle has a maximum weight carrying capacity. We assume that vehicles are identical and that vehicle weight capacity is limited in the sense that no vehicle can carry a significant fraction of the packages. The goal of a VRP instance is to find a solution that respects all constraints and at the same time minimizes the total travel time of all the vehicles.

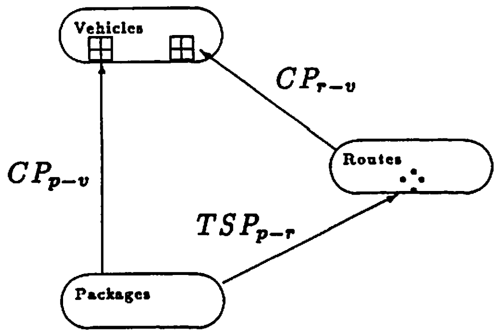

Figure 2: HCG for the VRP Problem

The HCG in Figure 2 represents the relationships that exist between the packages, the routes, and the vehicles that make up the VRP problem. The named installment map CP_{p-v} is a capacitated partition map; it partitions packages among vehicles. The named installment map TSP_{p-r} represents the delivery of packages and the relative position of packages in the ring structures that determine the tour paths of the vehicle routes. Finally, CP_{r-v} is a capacitated partition map. It assigns routes to trucks. CP_{r-v} is, in a sense, a trivial map once CP_{p-v} and TSP_{p-r} have been established, since a route can only be assigned to the vehicle that already holds those packages. This principle, which we call *the transitivity of containment*, plays a significant conceptual and computational role in our work. It says that composed maps that follow alternative paths between two nodes in an HCG must lead to consistent assignments. (In section 5 we will see an example of a situation in which this transitivity assumption is violated.)

Informally we solve VRP by 1) crudely clustering packages (i.e. delivery sites) and assigning each cluster to a unique vehicle; 2) routing each vehicle; and 3) optimizing each route.

We realize these steps as follows. First associate an angle with each package delivery site, using the dispatch depot of the vehicles as origin. Then sort the packages by angle and assign them to vehicles –a partition relationship– on a "next-fit" basis. That is, attempt to place a package in the current vehicle. If it won't fit, place it instead in the next unoccupied vehicle. This procedure crudely clusters packages with nearby destinations into the same vehicle. Next route each vehicle: examine each package in a vehicle, in turn, and insert it in that vehicle's route in a position closest to a package that has already been placed in the route. Finally, optimize routes by applying 2-opt local search to each route.

Our algorithm does a reasonable job of solving this simple version of VRP. Below we describe how to build a policy program that realizes this algorithm. We construct this policy program in four steps.

- **Step One**:

  Step one establishes the packages to vehicles assignment. First sort by angle, as described above. Next apply our first improvement policy, which we call IP_1. Its improvement sequence consists of a single library approximation routine, which solves the assignment problem embodied in CP_{pv} using the classical "next-fit" algorithm. In abbreviated form we write this policy as:

  $$IP_1(\text{packages,next-fit}(CP_{pv}))$$

  Thus, IP_1 is a bona-fide policy consisting of a transformational sequence with one entry (the call to the next-fit library routine). The collection of primary objects to which IP_1 is applied consists of all packages. The set of bound arcs associated with IP_1 is empty. The dispatch function of IP_1 is empty, so by default, the list order of primary objects applies.

- **Step Two:**

  Improvement policy IP_2 routes the packages associated with each vehicle. The algorithm considers each package in a vehicle and places that package in the best possible position in the route–the position which incurs the least additional travel distance. The improvement sequence is again one library approximation procedure, which we abbreviate as follows:

  $$\text{nearest-neighbor}(TSP_{pr})$$

The underlying OPL machinery will maintain consistency automatically in the following sense: packages in different vehicles cannot be assigned to the same route.

IP_2 will be called as many times as there are vehicles. Each time it is called, the packages belonging to the vehicle under consideration are designated as the primary objects. This process is captured in the FOR loop of the policy program presented below.

- **Step Three:**
  IP_3's singleton improvement sequence improves upon the existing partial configuration by performing local search on each of the routes using 2-opt the library routine local search. The packages to vehicles and packages to routes arcs are designated as bound. We write
  $$IP_3(\text{packages},\text{2-opt}(TSP_{pr}))$$
  to describe IP_3. A 2-opt interchange is accepted if the total travel distance in a vehicles decreases.

- **Step Four:**
  IP_4 completes the remaining assignment, CP_{rv}. Since the packages in a route have already been assigned to a vehicle, each route is bound to the vehicle that its packages are assigned to. In addition, it has been assumed that no travel time constraints are associated with this relation. Thus, the required assignment has been made implicitly, and we make the assignment explicit using the following policy:
  $$IP_4(\text{routes},\text{first-fit}(CP_{rv}))$$
  For each route, the underlying OPL machinery allows only the single feasible vehicle to be considered, and so each route is trivially assigned to the proper vehicle.

The preceding steps yield the following program.

```
packages = sort(packages, polar-coordinates)
$IP_1$(packages,next-fit($CP_{pv}$))
FOR each vehicle in vehicles
    begin
        packages = packages in vehicle
        $IP_2$(packages, nearest-neighbor($TSP_{pr}$))
        $IP_3$(packages,2-opt($TSP_{pr}$))
    end
$IP_4$(routes,first-fit($CP_{rv}$))
```

The program can now be applied to the problem instance.

The policy program could be augmented further to obtain additional improvement. For example, after the FOR loop has finished, packages could be moved or interchanged between routes (and vehicles) by local search library routines move-1 and swap-2 and then 2-opt local search could be applied again to individual routes. This strategy involves two local search procedures operating together in a single improvement policy. The ability to compose multiple local search and approximation routines in a single improvement policy adds considerable power to OPL, as we shall demonstrate in Section 4, where we present an extended OPL analysis of a generalized VRP problem.

This simple example illustrates the style in which OPL problems are representation and solved. In the next sections three rather different problems are considered. Our purpose is to show OPL's flexibility as a representation system as well as its effectiveness as a problem-solving idiom.

3 The Course Scheduling Problem

The course scheduling problem arises when an academic institution offers courses to students who have some flexibility in course selection and sequencing. The academic institution attempts to find a schedule of courses that minimizes the number of student conflicts and is feasible with respect to classroom usage and faculty schedules [4]. We use the course scheduling problem to illustrate the HCG specification process. In our simplified version of the problem we assume that the assignment of faculty to courses has already been completed. We ignore such issues as preassignments, infeasible course assignments and multiple sections of a course.

The construction of faculty schedules is considered first. We model this problem using three nodes: COURSES, FACULTY, and F-SCHEDULES (for faculty schedules). (See Figure 3a). Each faculty member is represented by a unique object in the FACULTY node. Similarly each course is represented by a unique object in the COURSES node. The arc from COURSES to FACULTY assigns each course to its predetermined instructor. Each element in the F-SCHEDULES node is a slotted object representing an initially blank timetable consisting of slots that represent the time periods that a course can be taught. For example, one of the faculty schedule time slots might represent (Tuesday, Thursday 9-10:30). Each instance of a course object includes data that gives the maximum number of students that can

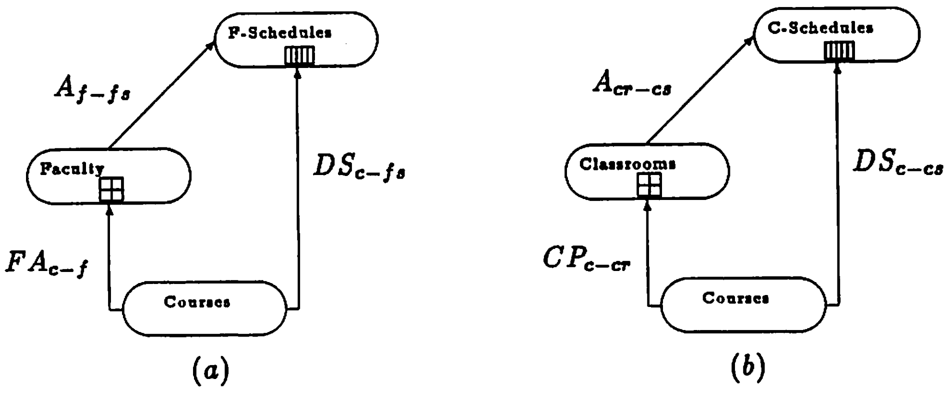

Figure 3: Pieces of Classroom Scheduling HCG

attend that course, and also a list of courses which, if scheduled at the same time, would introduce conflicts for some students.

The map FA_{c-f} is fixed – it reflects the predetermined course to faculty assignment. A_{f-fs} is an attribute map. It assigns to each faculty member a unique, initially blank schedule. The named installation map DS_{c-fs} requires that a course be assigned to a slot (time period) in the appropriate faculty schedule. The diagram in Figure 3a shows two paths from COURSES to F-SCHEDULES. By our principle of containment transitivity, we insist that these two mappings be consistent. In this case, consistency means that when a course is assigned to a schedule via path DS_{c-fs}, that schedule's associated faculty member must be the same person that the course is assigned to via the FA_{c-f} named map.

The second part of the course scheduling problem, classroom scheduling, is developed in similar manner (see Figure 3b). Three nodes are needed: COURSES, CLASSROOMS, and C-SCHEDULES (for classroom schedules). The COURSES node is described above. The CLASSROOM node contains objects that stand for individual classrooms. C-SCHEDULE objects are slotted objects (discrete schedules) that represent classroom schedules. Slots in the schedule represent the time periods when a course can be taught in a classroom. Classrooms objects carry data to indicate the capacity required in the classroom. As with the FACULTY and F-SCHEDULE nodes discussed above, each classroom is associated with its own C-SCHEDULE node, and hence A_{cr-cs} is an attribute map. The CP_{c-cr} is a capacitated partition map that assigns each course to a classroom that can hold it. The capacity limits are formed by including data about the maximum number of students that could fit into a classroom and the actual number of students in a course into the capacitated partition named map. The map DS_{c-cs} assigns a course to a time period in its associated classroom.

Now we wish to integrate these two scheduling problems. To accomplish this, we create a new "DISCRETE-TIME" node which contains a single slotted object that represents the time periods at which classes can be taught. The node serves to coordinate the faculty schedules with the classroom schedules (see Figure 4). The maps from F-SCHEDULES and C-SCHEDULES to DISCRETE-TIME are collapsing maps. Thus the position of each course in a faculty schedule (time at which the course is offered by faculty) is mapped to the same position in the discrete-time container structure. Similarily, the position of each course in a classroom schedule is mapped to the same position in the discrete-time container structure. By transitivity, classes – that is, objects of type COURSES – must be mapped to the same time slots in the DISCRETE-TIME node when viewed from the faculty schedule and the class schedule subgraphs. In this way, OPL can correctly model scheduling problems involving schedule coordination.

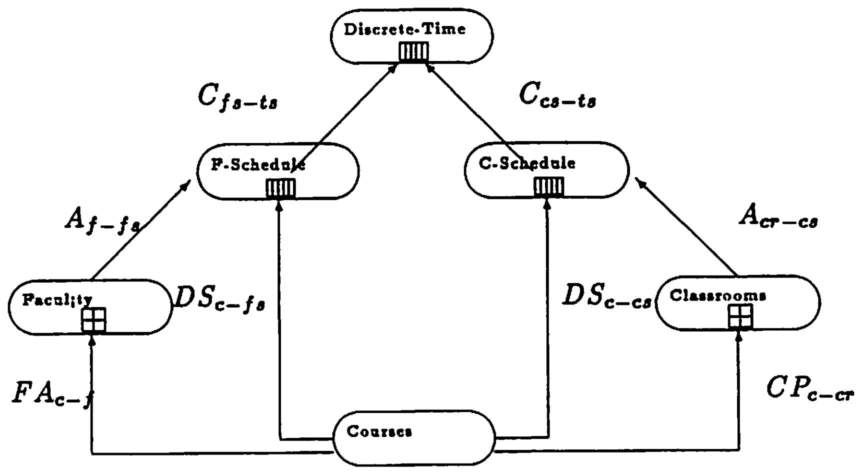

Figure 4: HCG for the Classroom Scheduling Problem

The objective function for this problem – the number of potential student scheduling conflicts – is determined by adding up the number of course conflicts that occur on each student's schedule, and then summing over all students. In the OPL environment this function is coded up by the user and then used to extend the list of named maps. In this case, a new named map is created by modifying the objective function call for the existing discrete-schedule named map. Approximation and local search functions that apply to discrete schedule named map can be inherited by the new map. Therefore, for this problem, local search functions that move or swap courses between classrooms are part of the environment and are referenced in the context of the new named installation map.

A constraint associated with the CP_{c-cr} map restricts course to classroom assignments in which the maximum number of students in the course is greater than the maximum capacity of a classroom.

A problem's HCG can suggest different problem solving strategies. For instance, suppose classroom space is tight. An OPL algorithm might work on classroom assignments (CP_{c-cr}), and classroom scheduling (the DS_{c-cs}) first. Local search routines could be applied to rearrange courses to make the best use of classroom space while taking into consideration student course conflicts. After developing a feasible assignment of courses to classrooms, the faculty schedules would be done. Any infeasibility in faculty schedules (two courses taught at the same time) could be dealt with by minor rearrangements of the classroom schedules or classroom assignments.

4 Benchmarking OPL algorithms: The Multiple Depot Vehicle Routing Problem

Our initial OPL benchmarks have involved the *Depot Vehicle Routing Problem*, or DVRP, which is a generalization of the simple VRP problem outlined in the introductory section. DVRP is like VRP, except that vehicles and packages are distributed among several depots, and deliveries are done from these sites. In this section we demonstrate that OPL is capable of producing high quality solutions when compared with previously studied approaches to DVRP. Not surprisingly, many algorithms developed using OPL are similar to previously developed algorithms. We consider this a strength of the notation, since OPL algorithms can be developed quickly and correctly through the extensive use of library routines.

A number of researchers have proposed algorithms for solving the DVRP that conform to the following outline. First, an assignment of customers (that is, package delivery sites) to depots is developed. Then the independent VRP problems at each depot are solved. Finally, in some cases, local search heuristics are applied to improve the solution.

The algorithm presented in [14] assigns customers to depots according to a heuristic which considers the distance to depots as well as the distance to nearest customer. The independent VRP problems are solved using a sweep algorithm [3], and vehicle tours are developed using the local search algorithms presented in [22, 23]. Tilman and Cain [34] develop an algorithm based on the savings method. Single customer tours are constructed initially with each tour bound to its the closest depot. The algorithm then merges tours if there is a subsequent decrease (a "savings") in route length. The assignment of customers to depots can change in the route merging procedure. Golden et al. [16] combine some of the characteristics of both of the above algorithms to produce a method which is applicable to large DVRP problems. Wren and Holliday [35] and Salhi and Rand [32] develop algorithms which repeatedly apply a collection of improvement procedures to a number of different starting solutions. More recent work on the DVRP has involved the simultaneous solution of both the multiple depot location problem and the multiple depot routing problem [30]. The algorithm for solving the DVRP portion of the problem works as follows: the savings method is applied to the problem to develop a collection of routes, after which two local search routines are applied.

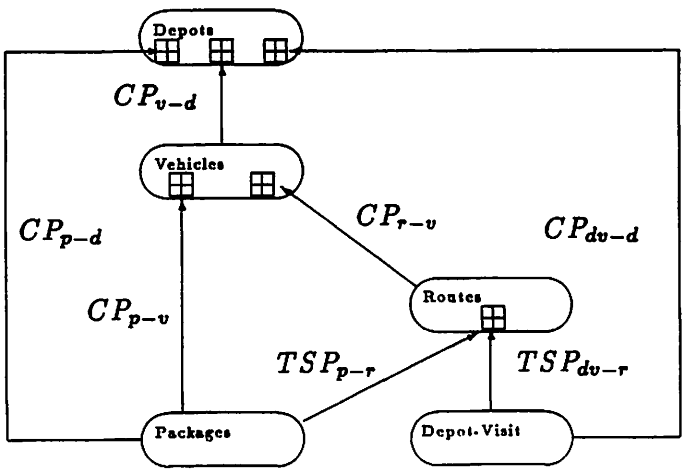

Figure 5: HCG for the DVRP Problem

An HCG for DVRP is given in Figure 5. The intuitive embedding of VRP in DVRP is reflected in the HCG in a natural way. Indeed, if CP_{p-d} is satisfied first, then what remains is a collection of separate VRP instances, and this is certainly a plausible initial solution strategy. Of course after these independent VRP instances have been solved, it may make sense to introduce further crosstalk between depots (using a "2-swap" local search routine) and then optimize any altered routes a second time with, say, 2-opt.

As reported in [26] and [24], we created about a dozen policy programs for DVRP and applied them to the DVRP data sets cited in [29, 30]. The best of these policy programs outperformed the results reported in those papers on all three reported data sets. Let us examine the workings of this "best" policy program informally.

The policy program is built in two phases, and proceeds as follows: first a feasible solution is built: packages (i.e. delivery sites for packages) are first assigned to their nearest depots, subject to depot capacity constraints. Next VRP routing and optimization is applied to each separate depot, along the lines described above. This concludes phase I of the algorithm.

Phase II is a pure local search stage. Starting with the final solution of phase I, packages are moved one at a time to more promising routes; packages in different routes exchange positions; packages in different routes exchange routes and are placed in the "best" position in the other package's original route; and 2-opt is applied. In general these exchanges take place across routes associated with different depots.

In summary, then, our DVRP solution for the datasets described in [29, 30] works by 1) Factoring the DVRP instance into separate VRP problems by partioning packages among depots; 2) solving each VRP instance separately; and 3) incrementally mixing these solutions repeatedly, each time consolidating results by reoptimizing each subproblem. We stress that this DVRP solution is built easily by appropriately composing OPL library functions.

As a further test of OPL problem solving capabilities, we considered two of the problems (problems 6,7) described and solved in [13]. Our problem solving machinery is compared to this particular work because it was one of the few sophisticated, "well tuned" methods for DVRP that also included datasets.

Our first attempt at solving these two problems involved the application of the "best" policy program developed for the [29, 30] datasets. The results were approximately 4% above the reported results in [13]. At this point, we had a choice. While the sophisticated heuristics described in [13] could be developed and incorporated as a library routine in the OPL environment, we instead tried a different approach: we used randomization to construct initial solutions to the [13] datasets.

In the context of OPL, randomized assignments can be constructed for any collection of arcs in an HCG. In the DVRP packages could be randomly distributed to depots, vehicles, or positions in routes. Vehicles could also be randomly distributed to depots. Our initial results indicate that for these two problems, packages should be assigned to the closest depot and then randomization of package to route and vehicle assignments is most effective. Therefore our second solution involves distribution of packages to closest depot and then random assignments of packages to vehicles and positions in routes. Once a random solution is constructed, the sequence of improvement routines described in step 3 of the "best" policy program is applied to the starting solution. The best among all random starting solutions was retained. Using this approach we developed a solution for dataset 6 in [13] that is 1% over the reported results. For dataset 7 in [13] we constructed a solution that is 0.5% below the reported results.

Extensions to the DVRP are described in [25]. In particular we describe our first solution to a vehicle routing problem in which vehicles are leased, and for which there are multiple depots, precedence relations on site visits, and time windows for package delivery. The objective of the problem is to find a minimal cost fleet that respects all problem constraints. This example highlights OPL's robustness as a modeling tool. Once the appropriate primitives were in place, the construction of an OPL problem representation and associated initial solution was straightforward.

5 Real Time Scheduling and Resource Allocation in OPL

For our final OPL example we describe a particularly complicated optimization problem from the domain of real-time scheduling and resource allocation. This problem illustrates the level of problem complexity that is within the scope of OPL.

5.1 Allocation and Scheduling of Tasks and Resources

The real-time computer resource allocation and scheduling problem, or CTRAS, involves the allocation and scheduling of tasks and resources in a distributed computing network. We formulate this problem as follows. We assume that a computer system is equipped with a set of identical processors, as well as a collection of non-identical resources. Tasks are to be executed by the processors. Each task requires a certain amount of processor time from one of the processors, and, in addition, each may require different resources to be available during task execution. The types of resources include files, data structures, and physical components of a distributed network. As with tasks, each resource is assigned to a unique processor. A task may access a resource in shared or exclusive mode. If task A requires resource R in exclusive access, then no other task can use this resource while A is executing. Shared mode access to a resource means that different tasks can use that resource at the same time.

Each task has an arrival time, a deadline, and a required computation time. Tasks must be scheduled for processing between their arrival and deadline times.

CTRAS has one additional feature that can affect the quality of solutions dramatically. If task A requires resource R, and A and R are assigned to different processors, then an execution time tax is charged to A to reflect the cost of network activity incurred when A accesses R. We assume a "higher" execution tax rate in the case where A accesses R in exclusive mode.

This problem arose in the context of a larger project which considers real-time processing in a distributed network. [1] The results of solving the allocation and scheduling of tasks given a fixed resource/processor assignment has been considered in [31]. Here we consider the situation in which resources can be assigned to any processor. Therefore the assignment of both tasks and resources to processors must be made.

Clearly a good allocation of tasks and resources to processors allows tasks to access resources locally, thereby avoiding the network traffic tax.

5.2 HCG Representation

We describe the construction of the HCG in stages. The full HCG is given in Figure 6. First, the problem of scheduling tasks on processors is considered and issues of resource allocation and scheduling are ignored. Three nodes and three named maps are sufficient: TASKS, PROCESSORS, and P-SCHEDULES and CP_{t-p}, IS_{t-ps}, A_{p-ps}. The CP_{t-p} map partitions tasks among processors. Partition capacity is equal to the maximum schedule

[1]We gratefully acknowledge the help of Professor Krithi Ramarithan who, as one of the principal investigators of the SPRING project, introduced us to this problem.

time of the processor and each task has a size equal to its required execution time. Detailed scheduling considerations are ignored. IS_{t-ps} maps tasks to processor schedules using the interval scheduling named map. After a task has completed an IS_{t-ps} assignment, it will occupy a continuous time segment in the processor schedule of length equal to the task's processing time. A_{p-ps} is an attribute map which associates each processor with a schedule of tasks on that processor.

Next, we extend the HCG to include resource allocation and scheduling. Associated with each task is a collection of resource requests. For example, task A may need to use resources 1,4, and 5. When task A executes on its processor, resources 1,4, and 5 must be available and are used by task A (either in exclusive or shared mode). To model this in an HCG setting, a RTP (resource-use/task pair) node is created. An RTP object is associated with each resource request of a task. The fixed assignment named map FA_{rtp-t} explicitly designates which task a particular RTP element is associated with.

RTP objects are also associated with a node called RESOURCES, which has one object instance for each resource in the system. In the above example, the three RTP objects associated with task A have fixed assignments to resources 1, 4, and 5 respectively. Each resource in the RESOURCES node has a schedule, and these schedules are modeled with the R-SCHEDULES node. RTP objects must be scheduled with respect to the resource schedule; an interval schedule map, IS_{rtp-rs}, designates the scheduling of contiguous blocks of time on resource schedules for each task's associated RTP objects. Finally, an attribute map A_{r-rs} makes the one-to-one correspondence between resources and resource schedules.

An RTP object must be scheduled during the same time interval that its component task executes on its processor. This coordination of assignments is achieved in a manner similar to the schedule coordination in the classroom scheduling example in Section 3. Two collapsing maps, one from processor-schedules (C_{ps-t}) and the other from resource-schedules (C_{rs-t}) force the consistent scheduling of resources and tasks. For example, if task A has been scheduled from time 20 to 50 on a processor schedule, then the C_{ps-t} collapsing map will cause task A to be assigned to the 20 to 50 time interval in the TIME node structure. And if task A requests resource 1, say in exclusive mode, then the C_{rs-t} collapsing map and the transitivity principle will guarantee that the RTP instance associated with task A's resource 1 use is scheduled during that same interval.

Finally, resources must be allocated to processors. The map CP_{r-p} designates this allocation. This allocation decision may affect a task's processing time. If a task needs a resource and this resource is located on a processor other than the task's processor, then the appropriate network tax is added to the total task processing time. Note that the resource allocation must be completed before the tasks and rtps are scheduled. The total task processing time is then determined as a function of the resource allocation.

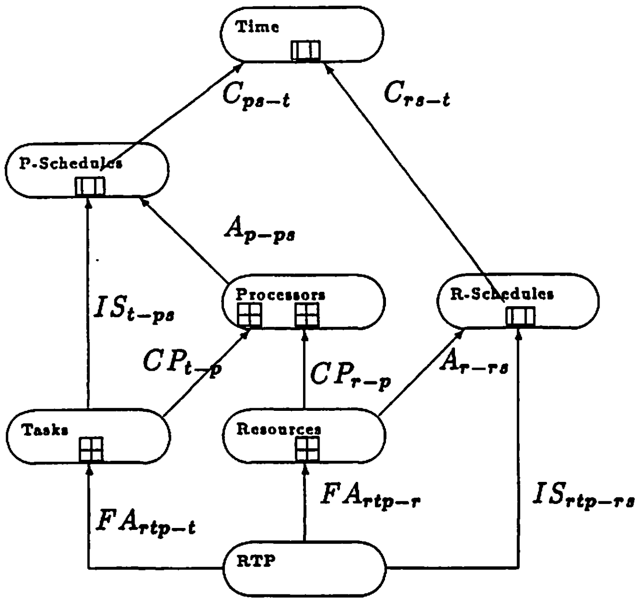

Figure 6: Combined Allocation and Scheduling of Tasks and Resources

In addition to the structural characteristics defined above, there are minor constraints associated with IS_{t-ps} which force task scheduling assignments to occur between the task arrival and deadline times.

We note one final peculiarity of HCG for CTRAS. Our principle of transitivity fails for paths between the RTP and PROCESSORS nodes, since if task A requires resource R, there is no guarantee that they will be assigned to the same processor. Thus it is necessary to disable transitivity checking at the PROCESSORS node.

5.3 OPL Algorithms

The most prominent complication in CTRAS is the following entanglement: the task to processor assignment is influenced by the resource to processor assignment and the resource to processor assignment is influenced by the task to processor assignment. We consider three different strategies for addressing this difficulty.

The first strategy is driven by task requirements: it completely assigns and schedules a task and the resources required by that task before going on to the next task. The second strategy develops the assignments of resources to processors first, after which the scheduling of tasks and resource requests are completed together. The last strategy develops the allocation and scheduling in a staged manner. The resource to processor assignment is developed first, then the task to processor allocation is made, and finally the task and resource request schedules are constructed. The three strategies are described in more detail below:

- **Integrated** : A task and the resources it requests are all allocated and scheduled jointly. The order of task placement is determined by a weighted average of task deadline and earliest starting time. The performance of the minimum weighted deadline and earliest starting time ordering is described in [31]. Given a partial assignment and a set of partial schedules, a task is assigned to the processor that requires the least resource copying time and is feasible with respect to the task schedule. If a task requires a resource that has not been assigned to a processor, then the resource is assigned to the processor chosen by the task.

- **Mixed** : The second strategy develops the assignments of resources to processors first, after which the scheduling of tasks and resource requests are completed together.

  In the first part of the mixed strategy, resources are randomly assigned to processors. Then assignments are modified by the local search routines **move** and **swap**. A resource will be moved or swapped with another resource if there is an overall improvement in the total *resource adjacency*. The resource adjacency of two resources is the fraction of total schedule time that two resources are requested by the same task. This gives a measure of how beneficial it would be to have two resources located on the same processor. Changes in resource assignments are also affected by the total time required by the tasks that use a resource. If the total time requested by tasks that use a processor's resources exceeds the total amount of processor time, then a surcharge is added to the local search metric. This completes the resource assignment stage.

After the assignment of resources to processors is made, each task and its resource requests are allocated and scheduled jointly by a single improvement policy. This improvement policy assigns tasks in order of minimum deadline and minimum earliest starting time to the processor that requires the least resource copying time.

- **Staged** : Resource assignment are developed first in the same style as the mixed approach. Then tasks are assigned in order of weighted minimum deadline and minimum earliest start time to the processor that requires the least resource copying time. After developing an initial allocation of all tasks to processors, local search routines **move** and **swap** are applied to reduce the total resource copying time by tasks. After all task to processor rearrangements are completed, tasks are scheduled on processors and resource requests are scheduled on resource schedules in a single improvement policy.

The number of tasks that can be feasibly scheduled by a problem solving strategy gives a measure of that strategy's performance. To distinguish the performance of different methods, extremely difficult test datasets were constructed and strategies were compared based on the fraction of tasks that could be successfully scheduled.

The dataset construction process was implemented with an OPL policy program that was modeled after the dataset construction algorithm given in [31]. First a set of resources are created and assigned randomly to processors. Next a task is generated with a random computation time. Then it is scheduled on the processor that allows the earliest starting time. Resource usage patterns are assigned to the task after its schedule on a processor has been set. These resources are chosen at random from the set of resources, such that no resource scheduling conflicts occur. Once resource usage patterns are determined for a task, its running time can be determined. We now report the task's arrival time as its starting time in the synthetic schedule, and its deadline as a function of completion time in that schedule. Clearly the dataset generation process develops very tight schedules.

We consider the amount of time needed to access resources across the network to be a parameter of the CTRAS problem. The resource access time

Resource Copy Tax	Integrated	Mixed	Staged
1%	85 %	85 %	90%
10%	83 %	78%	87%
25%	81%	77%	87 %
50%	81%	76%	77 %
100%	87 %	80%	76%
150%	91 %	82%	75%

Table 1: Results of Three Strategies for the CTRAS Problem

will of course vary according to the type of task and the type of resource. In order to compare the above strategies, we simplified this potentially non-uniform data characteristic to one value. This value is taken to be the percentage increase in task processing time if a resource must be accessed across the network. For instance, if a task requires exclusive access to a resource on a different processor in the distributed network and this parameter is equal to 10%, then the processing time of the task will increase by a total of 20% (exclusive access requires access to the network twice). In Table 1 the results of applying our three strategies to datasets constructed with different resource copy times are presented.

The three strategies represent a spectrum of decomposition approaches to problem solving. The results from this table can be interpreted with the level of decomposition in mind.

The problem decomposition strategy of the staged approach allows initial allocation decisions to be rearranged. When resource copy costs are low, this strategy is effective because the problem resembles a graph partitioning problem with multiple partitions. That is, each resource can be considered a node in a graph. The cost of separating two resources is proportional to the sum of the remote access costs incurred by tasks which require both resources. All other things being equal, two resources that are jointly accessed by a collection of tasks are better off being placed on the same processor. Local search performs well on the graph partitioning problem and the performance of local search in rearranging resources on processors can be seen in light of this comparision to graph partitioning.

But as a comparison with the mixed strategy indicates, local search on the resource to processor assignment is not always adequate. The assignment of tasks to processors can also be effectively rearranged when the resource copy costs are low. Local search on the task to processor named map (CP_{t-p}) allows the movement and interchange of tasks if an overall reduction in the resource copying costs occurs. When resource costs are low, these rearrangements reduce the total processing time and thereby improve the overall chances of producing good schedules.

When the resource copy costs are high, the scheduling aspects of the problem take over and the rearrangements of allocation decisions are no longer as effective. The rearrangements of the staged and mixed strategies cannot appropriately anticipate the scheduling problems which arise in later parts of the problem solving process. The integrated strategy does relatively better because both the allocation and scheduling of a task and its associated resource use are considered together.

This section has described the results of a rapid prototyping session for the CTRAS problem. When resource copying costs are low, a problem solving strategy which built a solution one "arc" at a time was the most effective. When resource copying costs were high, the results of this analysis indicate that an integrated solution, where all relations are satisfied together, was a more effective. OPL and the HCG specification machinery provided the appropriate structure to describe and evaluate these different problem solving strategies.

6 Conclusions

The OPL problem solving notation and associated programming environment has been designed to allow for the natural representation and solution to a broad class of discrete optimization problems. Problems are represented as graphs, in which nodes stand for the objects of the problem, and arcs represent recognizable primitive optimization problems. OPL allows solutions to be built quickly with the aid of extensible software libraries, which provide off the shelf solution machinery for the atomic "pieces" of a problem's OPL representation.

OPL is extensible. Primitive problems can be added to the existing list of named maps and solution methods for new and existing named maps can be added to the OPL libraries.

The constructive manner in which HCG's and policy programs are built makes problem integration a fundamental part of the OPL environment. Indeed, problem integration has been applied at some level to each of the problems described in this paper. An HCG specification of the classroom scheduling problem was constructed by 'gluing" together the two, potentially independent, problems of faculty scheduling and classroom scheduling. The multiple depot routing problem was an easy extension – both notationally and computationally – of the single depot routing problem. And the CTRAS problem gives an example of integrating both allocation and scheduling decisions for the two distinct groups of objects. The ability to piece together the description and solution of related subproblems is an invaluable technique for addressing complex optimization problems.

We expect OPL to evolve as experience with one class of problems adds new insights for problem representation and new entries in the environment's software libraries. Indeed, this evolutionary process allowed much of the CTRAS problem to be built from the "pieces" of the multiple depot vehicle routing problem, and we expect further leaps of this kind in the future.

References

[1] Bisschop, J. and A. Meeraus, "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment", *Math. Programming Studies* Vol 20, 1982, North-Holland, Amsterdam.

[2] Bodin, L.,Goldin, B., Assad, A., Ball, M., "Routing and Scheduling of Vehicles and Crews", *Computers and Operations Research*, Vol. 10, No. 2, pp. 63-211, 1983.

[3] Clarke, G., Wright, J., "Scheduling of Vehicles from a Central Depot to a Number of Delivery Points", *OR*, Vol. 12, 1964, pp. 568-581.

[4] de Werra, D. "An introduction to timetabling", *European Journal of Operations Research*, Vol 19, No 2, 1985.

[5] Dinkel, John J., John Mote and M.A. Venkataramanan, " An Efficient Decision Support System for Academic Course Scheduling", *Operations Research* Vol 37, No. 6, 1989.

[6] Dyer, James S., and John Mulvey, "An Integrated Optimization/Information System for Academic Departmental Planning", *Management Science* Vol 22, No. 12, August 1976.

[7] Fourer, R, D.M. Gay, and B.W. Kernighan, "AMPL: A Mathematical Programming Language," *Computing Science Technical Report* No. 133, 1987, AT&T Bell Laboratories, Murray Hill, NJ 07974

[8] Geoffrion, A., "An Introduction to Structured Modeling", *Management Science* 33:5, May 1987.

[9] Geoffrion, A., "Modeling Approaches and Systems Related to Structured Modeling", Working Paper 339, Western Management Science Institute, May 1987, University of California, Los Angeles.

[10] Geoffrion, A., "SML : A Model Definition Language for Structured Modeling" Working Paper 360, Western Management Science Institute, 1988, University of California, Los Angeles.

[11] Geoffrion, A., "The Formal Aspects of Structed Modeling", *Operations Research*, Vol 37, No. 1, 1989.

[12] Gheysens, F., Golden, B., Assad, A., "A Comparison of Techniques for Solving the Fleet Size and Mix Vehicle Routing Problem", *OR Spektrum*, Vol. 6, 1984, pp. 207-216.

[13] Gillet, B. and J. Johnson, (1974) "Sweep Algorithm for the Multiple terminal vehicle dispatch algorithm" 46th ORSA meeting San Juan, Puerto Rico.

[14] Gillet, B. and J. Johnson, (1976) "Multi-terminal vehicle dispatch algorithm" *Omega* 4 pp 711-718.

[15] Glassey, C. Roger, and Micheal Mizrach, "A Decision Support System for Assigning Classes to Rooms", *Interfaces* 16:5 1986.

[16] Golden, B., T. Magnati, and H. Nguyen "Implementing vehicle routing algorithms", *Networks* 7 pp 113-148, 1977 .

[17] Golden, B., Assad, A., Levy, L., Gheysens, F., "The Fleet Size and Mix Vehicle Routing Problem", College of Business and Management Technical Report 82-020, University of Maryland, 1982.

[18] Jones, Christopher, "An Introduction to Graph-Based Modeling Systems", ORSA Journal on Computing, Vol. 2, Spring 1990.

[19] Kernighan and Lin (1970), "An Effective Heuristic Procedure for Partitioning Graphs" *BSTJ* No. 2

[20] Langston, M. (1987) "A Study of Composite Heuristic Algorithms" *J. Opl. Res. Soc.* Vol 38, No. 6. pp 539-544.

[21] Lauriere, J.L., "Alice: A Language for Intelligent Combinatorial Exploration" *A.I. Journal*, 1978.

[22] Lin, S. "Computer Solutions of the TSP", *BSTJ*, No. 10, December, 1965, pp. 2245-2269

[23] Lin S., and Kernighan, B.W. "An effective Hueristic Procedure for the Traveling Salesman Problem" *Operations Research* 21 pp 498-516.

[24] MacLeod, B. "OPL: A Notation and Solution Methodology for Hierarchically Structured Optimization Problems", Ph.D. Thesis, University of Massachusetts, 1989.

[25] MacLeod,B. and Moll, R.N. "A Toolkit for Vehicle Routing", Proceedings of The IEEE Conference on Systems Integration, Morristown, N.J., April, 1990.

[26] Moll, R.N. and MacLeod, B. "Optimization Problems in a Hierarchical Setting", COINS Technical Report 88-87, October, 1988, University of Massachussets, Amherst, MA 01003

[27] Palmer, K., "A Model Management Framework for Mathematical Programming", 1984, Wiley, New York.

[28] Papadimitriou, C. and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[29] Perl, J. and M. Daskin "A Warehouse Location Routing Problem", *Transporation Research-B* Vol 19B, No. 5, pp. 381-396, 1985.

[30] Perl, J. "The Multi-Depot Routing Allocation Problem" *American Journal of Mathematical and Management Sciences* Vol 7, pp. 7-34, 1987.

[31] Ramamritham, Krithi, John A. Stankovic, and Perng-Fei Shiah, "O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems", *International Conference on Parallel Processing*, 1989.

[32] Salhi, S. and G. Rand "Improvements to Vehicle Routing Heuristics", *Jrnl. Opl. Res. Soc.* Vol 38 No. 3, pp 293-295, 1987.

[33] Smith, D. "Structure and Design of Global Search Algorithms" *Kestrel Institute Technical Report* July 1988. Palo Alto, California 94304. 1988.

[34] Tillman, F. and T. Cain "An Upper Bounding Algorithm for the Single and Multiple Terminal Delivery Problem", *Management Science* Vol 18, No 11, pp 664-682, 1972.

[35] Wren, A. and A. Holliday, "Computer Scheduling of Vehicles from One or More Depots to a Number of Delivery Points", *Operational Research Quarterly* Vol 23, No. 3, 1972.