Chapter 1

Introduction

1.1 A Brief History

nothing short of amazing. This hardware improvement made it possible to tackle larger

always a *global upper bound* on th eoptimal value. In the branch and bound algorithm we maintain a list of

Bounding Operation
Input: A subproblem S, described in terms of a "small" set of inequalities L^0 such that $S = fx^s : s \in F$ and $ax^s \cdot \bar{} \; 8 (a;\bar{}) \in L^0g$ and ®

Generic Branch and Cut Algorithm
Input: A data array specifying the problem instance.
Output: The global optimal solution

1.4.4 Branch, Cut and Price

Finally, when both variables and cutting planes are generated dynamically during LP-based branch and bound, the technique becomes known as *branch, cut and price* (BCP). In such a scheme, there is a pleasing symmetry between the treatment of cuts and variables. However, it is important to note that while branch, cut and price does combine ideas from both branch and cut and branch and price (which are very similar to each other anyway), combining the

number of global cuts and variables that need to be accounted for during the solution pro-

challenges inherent in BCP. In the remainder of this section, we will further discuss this distinction and the details of how it is implemented.

Variables and Cuts

Although their algorithmic roles are different, variables and cuts as objects are treated identically in COIN/BCP. We will describe the various types of variables.

to the variable. Using the schedule planning example, the compact representation may be the information which flight legs a particular plane is going to fly. From this information it's easy to derive when the plane is on the ground and hence it is easy to compute the coe□cients of the column for constraints that, say, specify that at a given time at a given airport only so many planes can be on the ground.

To summarize the advantages and disadvantages of the various variable types:

- core variable: always stay in the formulation which is both good (no bookkee5(fly)82(.)5e.lrequir

start information is either inherited from the parent or comes from earlier partial processing of the node itself (see Section 1.6.1). Along with the set of active objects, we must also store

- Compute an initial upper bound using heuristics.
- Perform problem preprocessing.
- Initialize the BCP algorithm by constructing the root node.
- Initialize output devices and act as a central repository for output.
-

The Cut Generator M41.dule

The *cut generator*

Search Chains and Diving

Once execution of the algorithm begins, the tree manager's primary job is to guide the

If no upper bounding subroutine is available, then a unique two-phase algorithm can also be invoked. In the two-phase method, the algorithm is first run to completion on the specified

protocol supporting dynamic spawning of processes and basic message-passing functions. All communication subroutines interface with COIN/BCP through a separate communications API. As mentioned above, currently PVM is the only message-passing protocol supported, but interfacing with another protocol is a straightforward exercise.

Chapter 2

Getting Started: Sample Compiling

Having familiarized yourself with the overall design of COIN/BCP

- `Osi`: open solver interface,
- `Dfo`: derivative free optimization,
- `COIN`: to get all modules,

If you are just starting with Bcp, get the module `Bcp-all`. It will automatically get the two sample applications (`Mkc` and `MaxCut`), as well as the other necessary modules (`Osi` and `Vol`). Note that the directory

– OSLDIR

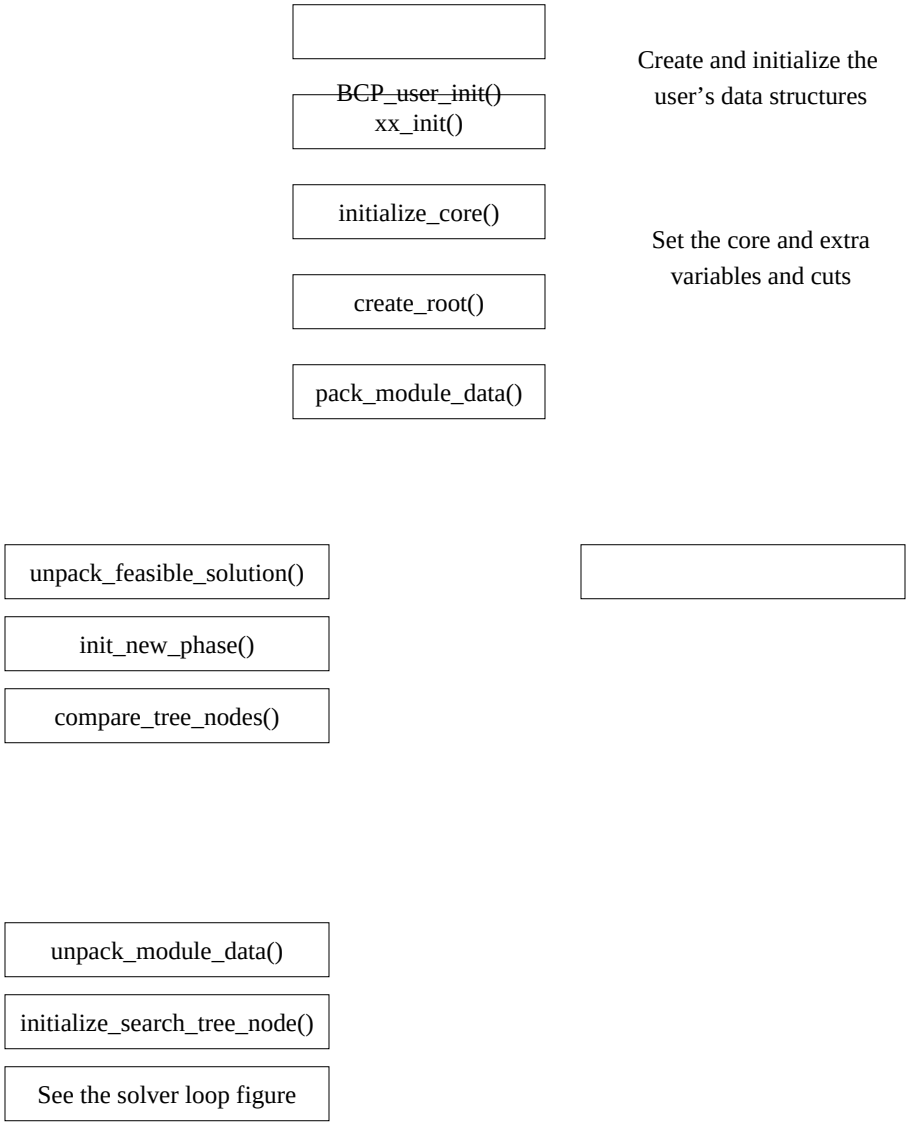
Chapter 3

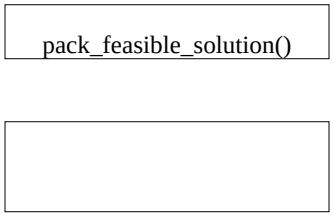
Developing Applications with COIN/BCP

3.2 Overview of the Class Hierarchy

We now briefly describe the class hierarchy from the user's point of view. Our aim here is not to describe the full class structure, but just those parts that the user needs to be familiar with in order to derive new user classes and override the appropriate methods.

could be defined in a separate base class and then, using multiple inheritance, derived


BCP_user_init()
xx_init()
Create and initialize the user's data structures
initialize_core()
create_root()
Set the core and extra variables and cuts
pack_module_data()
unpack_feasible_solution()
init_new_phase()
compare_tree_nodes()
unpack_module_data()
initialize_search_tree_node()
See the solver loop figure


pack_feasible_solution()

- (un)pack_cut_

pair, unpack_primal_solution() in the cut generator. There is no reason to override it if no cut generator processes are started.

2 pack_dual_solution()

3.4. DETAILS OF THE INTERFACE

jective values in the children and makes a decision based on those (the decision is

there is no need to pack algorithmic variables. They are only received with the primal solution.

- `pack_cut_algo()`: pack an algorithmic cut. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic cut in

3.5.2 Generating variables

Generally speaking, dynamic variable generation (often called column generation) is used less frequently than dynamic cut generation. If it is possible to e□ciently generate all variables explicitly in the root node and there is enough memory to store them, this is generally the best thing to do. This allows variables to be fixed by reduced cost and nodes to be fathomed without expensive pricing (see the last paragraph). However, sometimes this is either not possible or not e□cient because (1) there is not enough memory to store all of the variables in the matrix at once, (2) it is expensive to generate the variables, or (3) there is an e□cient method of pricing large subsets of variables at once. There may also be

Create the root node.

- Override `create_root()` in BCP_tm_user.

Modify the LP solver parameters.

- Override `modify_lp_parameters()` in BCP_lp_user

Define data structure to store and send feasible solutions.

- Derive a new solution class from BCP_solution.
- Override `(un)pack_feasible_solution()` in the classes BCP_lp_user BCP_tm_user (unpacking).

Define data structure to send LP solutions.

- Override `(un)pack_`*f*`primal,dual`*g*`s0lution()`

appropriate module. This method will return a pointer to the data structure for the appropriate module. Casual users are advised against modifying COIN/BCP's internal data structures directly.

3.7 Inter-process Communication

The implementation of COIN/BCP strives to shield the user from having to know anything

3.8.2 Debugging with PVM

If you wish to venture into debugging your distributed application, then you simply need to set the parameter `DebugXxProcesses`, where `Xx` is the name of the module you wish to debug, to the value "1" (representing true) in the parameter file. This will tell PVM to spawn the particular process or processes in question under a debugger. What PVM actually d actJ/F6n this caseactJ/F6s to launch the script `$PVM_ROOT/lib/debugger`. You will undoubtedly want to modify this script to launch your preferred debugger 6n the manner you deem fit.

Chapter 4

Sample Application: The MKC Problem

In this chapter we describe how the solver for the MKC problem were implemented. This

4.3 A formulation suitable for column generation

This new formulation has significantly more columns than the original formulation, on the other hand it results in a well studied problem, the set packing problem ([?]).

There are two types of constraints in this formulation. The first type corresponds to the slabs in the problem, the second type to the orders. The variables represent feasible production patterns, that is, variable u has a 1 in the row corresponding to the slab the production pattern is to be made of and 1's in the rows corresponding to the orders in the production pattern. Each variable isariable indicating production pattern is
chosen in the solution or not. Let us introduce the following notation:

- P is the set of feasible production patterns;
- P

eliminating the need to bother about cuts.

For the variables first we had to decide which ones are going to be core variables and which

4.4.3 Packing and unpacking

Chapter 5

Sample Application: The Maximum Cut Problem

idea during initial development since it makes debugging much easier. Because we are not using a separate cut generator, we do not need to consider the BCP_cg_user class either.

As with virtually any BCP implementation, we will need to consider the BCP_tm_user and BCP_lp_user classes. Also, because we will be dynamically generating algorithmic cuts, we will need to derive a new class to represent the cycle cuts (5.5) from the class BCP_cut_algo. Finally, we w wew ww wwanomBCP

- `create_root()`: To initialize the root node, we use some heuristics to generate an initial set of cycle cuts. However, as noted before, these are "extra" cuts and do not get put into the core. They may be removed later in the calculation.

- `display_feasible_solution()`

Bibliography

[1]