LibRCPS Manual

Robert Lemmen <robertle@semistable.com>

License

librcps version 0.2, February 2008

Copyright © 2004 – 2008 Robert Lemmen <robertle@semistable.com>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Contents

1 The RCPS Problem **4**

1.1 Basics 4

1.2 The Classical Model 4

1.3 The Modes Extension 6

1.4 The Alternatives Extension 8

1.5 Nonrenewable Resources 8

2 LibRCPS **8**

2.1 Usage 8

3 C API Reference **9**

1 The RCPS Problem

1.1 Basics

No doubt, project management is a hot topic. Just googling for it will turn up thousands of methods, companies, software packages and consultants that deal with it. The reason for this is simple: work does not really scale well. A hundred people can not do a hundred times the work that a single person can do, but much, much less. This is due to the overhead of organizing the work, people stepping on each other's toes and the fact that some things just can not be sped up. Some tasks are much more friendly to parallelizing than others, and unfortunately the ones that are not are getting more and more important.

Project management has a lot of aspects, one of them is creating an optimal schedule or work plan: finding out when to do what. Failing to create such a schedule in any non-trivial project will most certainly lead to delays, contract violations and a lot of wasted money. Unfortunately creating such a schedule is a very difficult problem, NP-hard to be exact. This means that there is no way to find a perfect solution in polynomial time complexity, or even simpler: you will not find the perfect solution for any reasonably complex case.

This my sound frustrating, but it actually isn't that bad: you don't need to find *the* perfect solution, it usually is sufficient to find *a* good solution. Over time, a number of heuristics have been developed that find good solutions for the project scheduling problem, e.g. the critical path method. Unfortunately it can be proven that any simple or "local" heuristic creates suboptimal solutions in many cases and will even create really terrible solutions in some cases. The bottom line: if you are using local heuristics for project scheduling, you are doing something wrong.

Fortunately there is a class of generic methods to deal with optimization problem like this: meta-heuristsics. The two most common are genetic algorithms and ant systems (which have a lot in common). LibRCPS uses genetic algorithms to find solutions to project scheduling problems under limited resources. In the following sections we will explain what and how can be done with this library and when you will want to use it.

Please note that this library uses a *model* of your project that can only reflect a given set of properties, and can only schedule projects that fit into this category. Most notably it will do project scheduling under limited resources, but not (yet) resource levelling. Should your project have characteristics that make it impossible to use the model LibRCPS uses, please talk to us as we would be glad to extend the possibilities of this library where feasible.

1.2 The Classical Model

The calssical or academic definition of the basic RCPS problem is that your project consists of a number of smaller jobs. We will call the number of jobs J and refer to the individual job as j_i with i ranging from 1 to J. The set of jobs that make up the project is then $\mathcal{J} = \{j_1, \ldots, j_J\}$. The duration of a job j is denoted by p_j. We use abstract terms of duration and time here. This could easily mean "working days" or something similar and needs to be mapped to real-world time with

an appropriate function. We also use discrete multiples of one period and do not preempt jobs once they are started.

Additionally we have precedence relations between jobs in the set. So we have a set $\mathcal{P}_j$ for every job j indicating that every job $i \in \mathcal{P}_j$ has to be finished before j can be started. These precedence relations can easily be expressed as a directed graph over the jobs that has to be acyclic.

We also have a set of K resources $\mathcal{K} = \{k_1, \dots, k_K\}$ with a capacity of r_k each. As stated before this capacity is renewed after the job that used it up is stopped. Each job j requires r_{kj} units of the resource k while it is active.

We use two additional activities j_0 and j_{J+1} with $p_0 = p_{J+1} = 0$ and $r_{k0} = r_{kJ+1} = 0$ for all $k \in \mathcal{K}$. j_0 is in $\mathcal{P}_j$ for every j, and all j_i for $i \in 0 \dots J$ are in $\mathcal{P}_{J+1}$. We call these jobs "dummy" jobs as they are not real jobs of the project, don't take up time and don't use up resources. They are only used to simplify things: since no job can be started before j_0 we have a defined start job, and since every job has to be finished before j_{J+1} we always have the same finishing job. These dummy jobs are used in the discussion of the classical model, but our library does not require them.

For the problem above we are looking for a solution, that is a start time s_i for each job j_i.

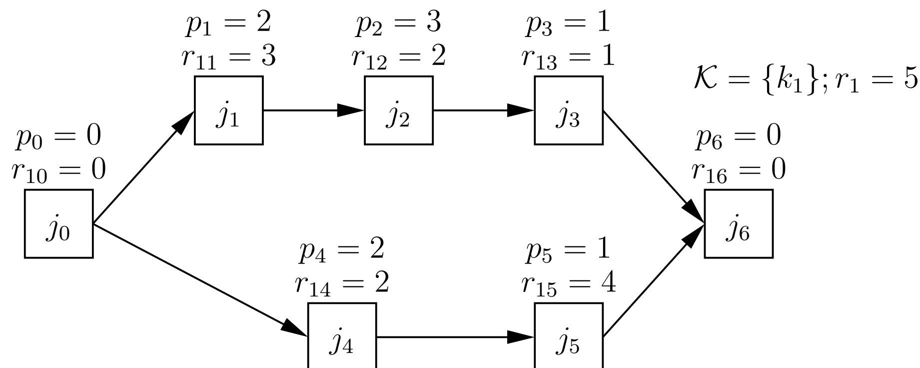

Figure 1: Example project

Let's assume a very simple project as an example: we only have a single resource with a capacity of 5 units and 5 jobs with the durations 2, 3, 1, 2, and 1 as well as resource requests for a single resource with amounts of 3, 2, 1, 2, and 4 respectively. (See figure 1) The algorithms described in this work now have the goal to produce a project schedule as seen in figure 2, where each job now has a start time, no resources are booked in excess of their capacity and the whole project finishes as early as possible. The left part of figure 2 is a simple Gantt Chart and gives the start time and duration for every job. The right part represents the usage of the resource by the jobs.

This basic model may look simple at first glance, but it can be proven that finding optimal solutions is NP-hard and, unlike some other problems where people perform better than current software solutions, can not be solved by real persons with desirable results.

In the next sections we will explain a couple of extensions that make the problem more complex, but also more applicable to real-world situations.

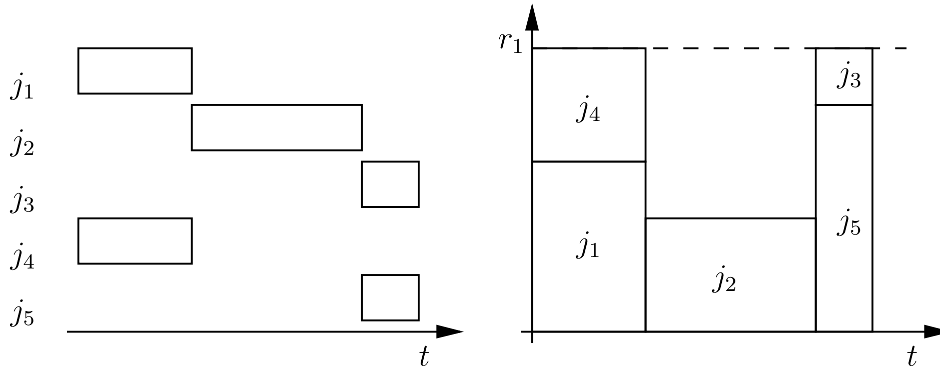

Figure 2: Example project schedule

1.3 The Modes Extension

The whole art of project management boils down to making the right decisions about how to do a certain project. The algorithm described up to now can help to define a schedule, but it cannot help you with other decisions, like how to do a single job. Unfortunately these decisions influence the scheduling process and therefore cannot be made independently. A simple example: You want to move a pile of gravel on a construction site and can choose whether you do it with one caterpillar in two days, or with two caterpillars in one day. The second option would of course hinder the second caterpillar from doing something other during that period. In some cases, for example if you have nothing else to do for the second caterpillar, you would want to use both. In some other cases, for example if it does not really matter how long you need to move the gravel, you would want to use only one. In order to incorporate this kind of decisions into the project scheduling problem an extension called "multiple modes", has been proposed and discussed. LibRCPS fully supports project scheduling with mutiple modes.

In the previous section we defined the basic project scheduling problem. For the multiple mode extension we remove both the resource usage and the make-span from the jobs and add a set of modes the job can be processed in. These modes have a make-span and a set of resource requests each. So we now have a set of J jobs $\mathcal{J} = \{j_1, \ldots, j_J\}$ each job j having a set of M_j associated modes that we denote by $\mathcal{M}_j = \{m_1, \ldots, m_{M_j}\}$ each. Each of these modes m_i has a duration of p_i and requires r_{ki} units of the resource k while it is active. Now every job is scheduled in one of its modes.

This allows us to define multiple alternative ways to do a single job. Taking the example above we would now have two modes for the job that moves the gravel. One mode would require two caterpillars as resources and would take a duration of only one day, the other mode would only require one caterpillar as a resource but take two days of duration. By integrating this decision into the genetic algorithm used to solve the project scheduling problem we have an extended, more complex, problem. But we are also searching a bigger space for optimal solutions that was inaccessible before and we are therefore likely to find better solutions.

A solution for this extended problem would consist of a start time for each of the jobs that obeys the

precedence relations as in the classical model and additionally for each job j_i a number n_i denoting which of the M_i modes is used.

Taking the example from the previous section and extending it to allow multiple modes, we could modify the jobs j_2 and j_5 so that they have two possible modes each. Additionally to the original mode we could add the possibility to do j_2 over a duration of 2 time units instead of 3 at an increased resource allocation of 4 instead of 2 units. For j_5 a mode that allows us to decrease the resource usage from 4 to 3 by increasing the job make-span from 1 to 2 could be added. Figure 3 illustrates this, each box representing a possible mode now. In figure 4 you can see that the original mode of j_5 is used and the new one for j_2 leaving us with a decreased total project make-span.

A possible solution in this example could be a permutation of the jobs like (j_4, j_1, j_2, j_3, j_5) and the selected modes of the jobs would be $n_1 = 1; n_2 = 2; n_3 = 1; n_4 = 1; n_5 = 1$. Please note that for the jobs n_1, n_3, and n_4 the first mode must always be selected, as they only have one.

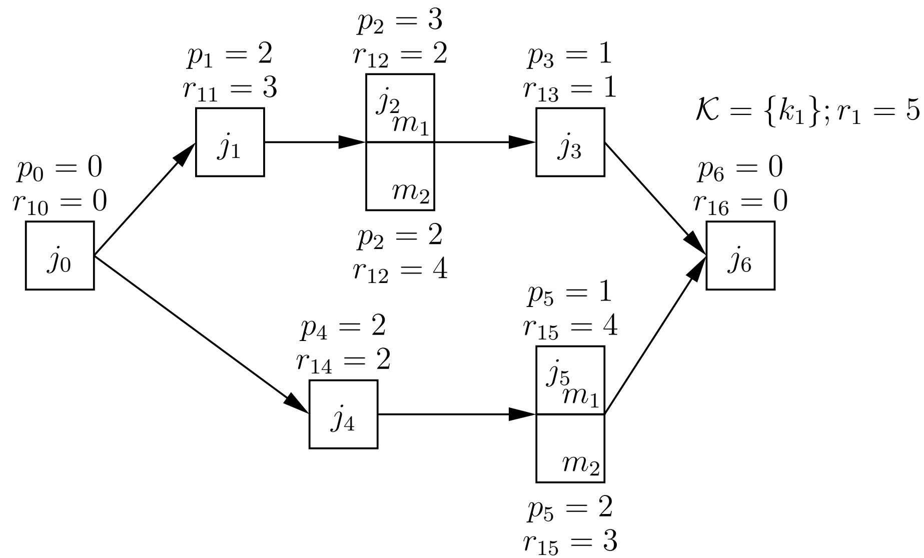

Figure 3: Example multi-mode project

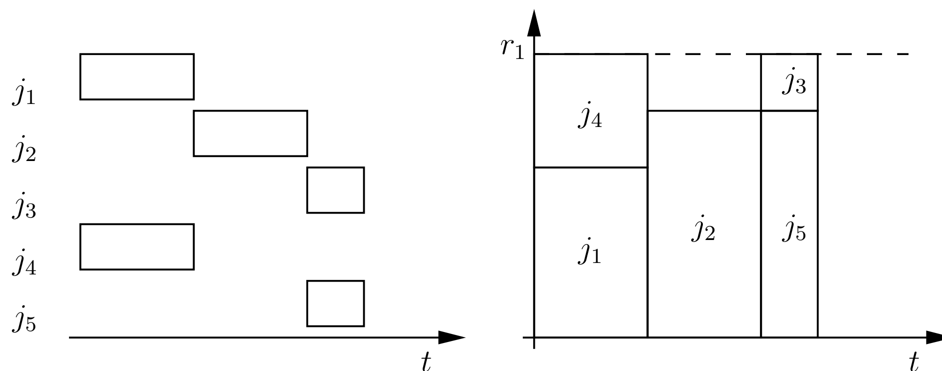

Figure 4: Example multi-mode project schedule

1.4 The Alternatives Extension

The resource alternatives extension is unique to LibRCPS and was the subject of the diploma thesis of the author. Basically we allow a job to declare alternative resource reuqirements. In a simple example you could have a job that either requires a worker A *or* worker B.

The motivation behind this is that in the classical model, you can have a number of different resources with a capacity each, but the individual instances of these resources are treated the same. Resource alternatives allows to differentiate them. As an example, you could have a couple of workers, of which some are skilled to use welders, some to drive a caterpillar and all to do basic works. If you have a job that requires two workers for basic works, one welder and one caterpillar driver, you will not be able to model this with the classical model. Using the modes extension, you will be able to model it but you will need about $N_w \cdot N_c \cdot N_g$ modes where N_w is the number of workers that can weld, N_c the number of workers that can drive a caterpillar, and N_g is the number of generic workers. This can quickly get out of hand ,especially as this number multiplies with the number from other resource requests. And to make things worse, the performance of the metaheuristic solver will degrade if the number of modes on a job differ extremly. Unfortunately this kind of modelling problem gets more and more common as the projects are getting more complicated and both personel and material are getting more specialized.

1.5 Nonrenewable Resources

2 LibRCPS

2.1 Usage

It is important to understand the basic usage pattern for LibRCPS, which is the same for any language. Using the library is done in a number of distinct steps:

1. Setting up the problem structure, resources, jobs, modes and alternatives
2. Declaring relationships between jobs
3. Setting up the solver
4. Running the Solver
5. Retrieving the results

While it is theoretically possible to do this in a different order in some cases (e.g. you could create the solver first), it is impossible in other cases (e.g. you need all jobs before you can start to declare precedence relationships between them) and confusing in others. It is therefore advisable to stick to this ordering. In the API reference for your language, you will find the methods used to do these tasks, which should be easy to use once you understood the basic model.

Please keep in mind that your task is to create a *model* of your project with the means provided by LibRCPS, and do not need to exactly replicate your problem. This means that you will sometimes need to apply simple transformations to get your job done. Some examples:

In some cases it might be desirable to declare a precedence between two tasks that also has a time component, e.g. "task B can not be started earlier than 2 days after task A". LibRCPS provides no means to do so, you will have to use an additional task between task A and task B that has a duration of 2 days and no resource allocations.

In other cases you will need to declare a dependency of type "not at the same time" which is the same as two precedence relations *ORed* together, which is not supported by LibRCPS in this manner. Use a dummy resource with an availability of 1 that is allocated by both tasks instead.

While this might look cumbersome at first glance, we actually consider this a feature: it keeps the interface to the library comparably clean, which would otherwise be clutteres with many redundant options.

3 C API Reference

In this Section we will explain how to interface a program written in the C language with LibRCPS. All the remarks in section 2.1 still apply.

First of all you will need to include the LibRCPS header file and make sure that you compiler finds it (which depends on the operating system and compiler used and is not discussed here):

```
#include <librcps.h>
```

This header file defines some values and declares a couple of opaque structures and methods to work on and with them. Opaque in this context means that you, as a user of the library, do not know how the structures look internally and should not bother. We will discuss defined values and structures as we need them for the methods.

```
char *rcps_version()
```

This method returns a string containing the version number of the library, which is usefull if you want to report it to the user or for checking that the library is actually the version that you require. The string is static and must not be freed.

```
struct rcps_problem* rcps_problem_new()
void rcps_problem_free(struct rcps_problem *p)
```

These methods are used to create and free a structure to hold the problem description, which we often refer to as a *model*. Creating this structure is normally among the first steps you do, destroying it among the last ones. As all the structures are opaque, you cannot free them yourself but must use the supplied method, which will also free every structure (job, resource, mode, alternative...) associated with it.

```
struct rcps_resource* rcps_resource_new()
```

```
void rcps_resource_free(struct rcps_resource *r)
```

Likewise, these methods are used to create and destroy structures that represent resources. As resources associated with a problem structure are freed automatically when the problem structure is freedm you will hardly ever need to call the second method yourself.

```
char* rcps_resource_getname(struct rcps_resource *r)
void rcps_resource_setname(struct rcps_resource *r, char *name)
int rcps_resource_gettype(struct rcps_resource *r)
void rcps_resource_settype(struct rcps_resource *r, int type)
int rcps_resource_getavail(struct rcps_resource *r)
void rcps_resource_setavail(struct rcps_resource *r, int avail)
```

Once you have created a resource structure, you can use these methods to query and manipulate the data in it. The name is used to identify a resource, but not used in our algorithm. You should still try to name resources uniquely to make things easier for yourself. The "availability" is the amount of the resource, and the type can either be `RCPS_RESOURCE_RENEWABLE` or `RCPS_RESOURCE_NONRENEWABLE` with the former being the default. Nonrenewable resources are not returned after the job is done, which can be used to model some situations, but is not fully supported yet. Do not use resource types in this version of the library!

```
void rcps_resource_add(struct rcps_problem *p, struct rcps_resource *r)
int rcps_resource_count(struct rcps_problem *p)
struct rcps_resource* rcps_resource_get(struct rcps_problem *p, int r)
struct rcps_resource* rcps_resource_getbyname(struct rcps_problem *p,
char *name)
struct rcps_resource* rcps_resource_remove(struct rcps_problem *p, int r)
```

These Metheods are used to associate a resource structure, once you set all it's fields, with a problem representation and to query the resources of a problem. the `rcps_resource_count()` method returns the number of resource definitions which can be used together with `rcps_resource_get()` or `rcps_resource_remove`. The latter function returns the resource structrure removed from the problem description, which you need to free yourself.

```
struct rcps_job* rcps_job_new()
void rcps_job_free(struct rcps_job *j)
```

Like the other methods of this type, these are used to create or free a structure representing a job or task. Please note that you do no need to free jobs if they are associated with a problem strcuture.

```
char* rcps_job_getname(struct rcps_job *j)
void rcps_job_setname(struct rcps_job *j, char *name)
```

Get and set the name of a job. The name is not used in the algorithm, and only helps you to identify the job.

```
void rcps_job_add(struct rcps_problem *p, struct rcps_job *j)
int rcps_job_count(struct rcps_problem *p)
struct rcps_job* rcps_job_get(struct rcps_problem *p, int j)
```

```
struct rcps_job* rcps_job_getbyname(struct rcps_problem *p, char *name)
struct rcps_job* rcps_job_remove(struct rcps_problem *p, int j)
```

As before, associate, remove and get the jobs in a problem.

```
void rcps_job_successor_add(struct rcps_job *j, struct rcps_job *s)
int rcps_job_successor_count(struct rcps_job *j)
struct rcps_job* rcps_job_successor_get(struct rcps_job *j, int s)
struct rcps_job* rcps_job_successor_remove(struct rcps_job *j, int s)
```

Add, get and remove precedence relations for a job. Please note that it is advisable to first create all the jobs, and then build the precedence graph with these methods. s denotes the successor of j.

```
int rcps_job_getstart_res(struct rcps_job *j)
int rcps_job_getmode_res(struct rcps_job *j)
```

Once the problem has been solved, you can use these methods to get the start time and selected mode for a job.

```
struct rcps_mode* rcps_mode_new()
void rcps_mode_free(struct rcps_mode *m)
```

Create and free modes for a job. Please note that each job *must* have at least one mode in our model.

```
int rcps_mode_getduration(struct rcps_mode *m)
void rcps_mode_setduration(struct rcps_mode *m, int d)
```

Get and set the duration of a mode. If your project does not really use modes, create one mode per job and set the duration there.

```
void rcps_mode_add(struct rcps_job *j, struct rcps_mode *m)
int rcps_mode_count(struct rcps_job *j)
struct rcps_mode* rcps_mode_get(struct rcps_job *j, int m)
struct rcps_mode* rcps_mode_remove(struct rcps_job *j, int m)
```

Manage modes on a job.

```
struct rcps_request* rcps_request_new()
void rcps_request_free(struct rcps_request *r)
```

Create or free resource allocations.

```
void rcps_request_add(struct rcps_mode *m, struct rcps_request *r)
int rcps_request_count(struct rcps_mode *m)
struct rcps_request* rcps_request_get(struct rcps_mode *m, int r)
struct rcps_request* rcps_request_remove(struct rcps_mode *m, int r)
```

Manage resource allocations.

```
int rcps_request_getalternative_res(struct rcps_request *r)
```

After the problem has been solved, get the selected resource alternative for this request.

```
struct rcps_alternative* rcps_alternative_new()
void rcps_alternative_free(struct rcps_alternative *a)
```

Create or free a resource alternative.

```
int rcps_alternative_getamount(struct rcps_alternative *a)
void rcps_alternative_setamount(struct rcps_alternative *a, int m)
struct rcps_resource* rcps_alternative_getresource(struct rcps_alternative *a)
void rcps_alternative_setresource(struct rcps_alternative *a,
struct rcps_resource *r)
```

Get and set the resource and the amount of the resource allocation in a given alternative.

```
void rcps_alternative_add(struct rcps_request *r, struct rcps_alternative *a)
int rcps_alternative_count(struct rcps_request *r)
struct rcps_alternative* rcps_alternative_get(struct rcps_request *r, int a)
struct rcps_alternative* rcps_alternative_remove(struct rcps_request *r,
int a)
```

Manage resource alternatives. Please note that in out model each request consists of alternatives, if you do not really use resource alternatives create exactly one alternative per request.

```
int rcps_check(struct rcps_problem *p)
```

This should be called before running the solver and is used to check the project for frequent problems. The method returns `RCPS_CHECK_OK` if no problems were found. In this version no real checks are done, but you should still run the method and check for the result. In future version there will be other return values for specific problems.

```
struct rcps_solver* rcps_solver_new()
void rcps_solver_free(struct rcps_solver *s)
```

Create or free a solver structure. Once your problem is created and checked, create a solver, run it and free it afterwards (and then get the results and free the problem).

```
int rcps_solver_getparam(struct rcps_solver *s, int param);
void rcps_solver_setparam(struct rcps_solver *s, int param, int value);
```

With these functions you can query and set parameters that affect the performance of the solver algorithm. In general you should not use these unless you know exactly what you are doing. The param argument can either be `SOLVER_PARAM_POPSIZE`, the population size, or one of `SOLVER_PARAM_MUTSCHED`, `SOLVER_PARAM_MUTMODE` and `SOLVER_PARAM_MUTALT` which are the mutation probabilities in the three chromosomes for the job schedule, the modes selection and the alternatives selection. These three are given in 1/10000s.

```
void rcps_solver_solve(struct rcps_solver *s, struct rcps_problem *p)
```

Run the solver on the problem.

```
int rcps_solver_getwarnings(struct rcps_solver *s);
```

In some cases, especially with nonrenewable resources, it is possible that the solver will not find a valid solution. In these cases, this will return 1 after the solver has been run. Every application should run this after running the solver and emit a warning if the result is != 0.

```
int rcps_solver_getreps(struct rcps_solver *s);
```

After the solver has been run, you can use this function to query the number of generational steps that were needed to find taht solution. This is not necessary for a normal application, but might help assessing the performance of the current parameter set.