

## Part 2: Integer Linear Programs and RWA

*Neal Charbonneau*

## Outline

This document discusses ILPs, their complexity, formulating RWA problems as ILPs, and lastly how to use CPLEX as a solver. This will only cover using CPLEX's interactive optimizer, not the libraries for various programming languages. We'll use MathProg again for CPLEX.

## Integer Linear Programming

An integer linear program is a linear program with the additional constraint that some or all of the variables must have integer values. In a pure ILP, all variables must be integers. A mixed ILP (MILP) has some variables that can have real values and some must be integers. Another common ILP is a 0-1 ILP, where all the variables are either 0 or 1.

Solving a linear program can be done in polynomial time, but solving an integer linear program is NP-complete, meaning it is unlikely a polynomial time algorithm exists to solve them. We'll take a look at how to go about solving ILPs and see why it is different than LPs.

### Solving ILPs

One tempting way to solve an ILP may be to remove the integer restrictions, then once a solution is found using a regular LP algorithm, round the variables up or down. This, unfortunately, does not work. After rounding it is unlikely the solution is the integer optimal solution and it may not even be a valid solution. Consider the simple ILP shown below.

$$\begin{aligned} & \mathbf{max} \quad 2x_1 + 3x_2 & (1) \\ & \mathbf{subject\ to} \\ & \quad .3x_1 + x_2 \leq 3 \\ & \quad - .3x_1 + x_2 \geq 0 \\ & \quad x_1, x_2 \geq 0 \text{ and integer} \end{aligned}$$

The graph corresponding to this ILP can be seen in Figure 1 (with the feasible region shaded in grey). Since this is an ILP, the only valid solutions are where  $x_1$  and  $x_2$  are integers, as shown by the small diamonds on the graph. If we remove the integrality constraint, then the optimal solution to the problem is at point (5, 1.5), as shown on the graph (on a vertex, as we would expect). We can't round this solution to get a solution to the ILP. If we round 1.5 up or down we get two solutions, (5,1) and (5,2), that do not lie

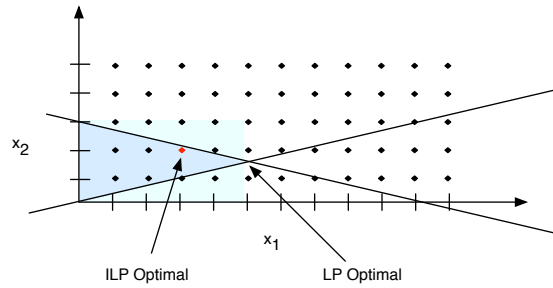


Figure 1: Feasible region for ILP (1). LP optimal solution vs. ILP optimal solution.

within the feasible region. The actual solution to the ILP is shown as a red diamond at (3,2). The optimal solution to the ILP can be arbitrarily far away from the optimal solution to the LP. None of the algorithms for solving LPs can help us solve ILPs directly.

A simple algorithm we could write to solve an ILP all combinations of the integer valued variables and select the combination with the best objective value. We can do this by building a search tree. Say we had an ILP with three variables,  $0 \leq x_1 \leq 3$  and  $0 \leq x_2, x_3 \leq 1$ . In order to search all the points, we can try all combinations of the variables by constructing the tree shown in Figure 2. The obvious problem with this approach is that the number of nodes to explore in the tree grows exponentially. It also may not be possible to bound all of the variables. For example, a 0-1 ILP with 200 variables would require searching  $\approx 1.6 * 10^{60}$  nodes. Two hundred variables may seem like a lot considering the examples we have seen before, but it is not uncommon to have many more than that.

Clearly, this solution technique is not practical except for very small problem instances. While relaxing an ILP (by relaxing we mean removing the integer constraints for the variables) to create an LP is not useful for solving the ILP directly, consider the following observations. First, relaxing an ILP and solving the resulting LP will provide an upper bound (for maximization problems) or lower bound (for minimization problems) for the

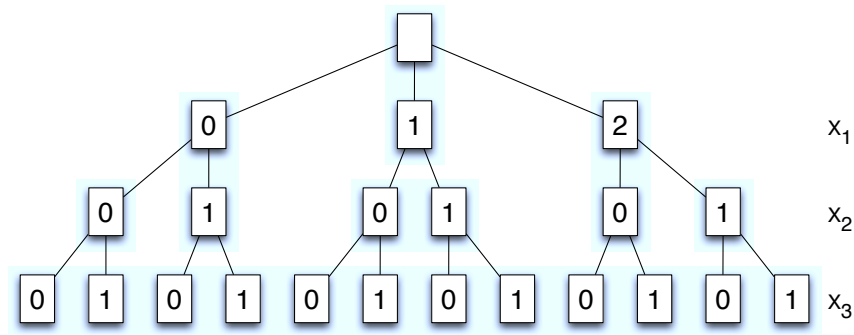


Figure 2: Enumerating all possible values of  $x_1$ ,  $x_2$ , and  $x_3$ .

objective function's value of the ILP. For example, the relaxation for ILP (1) results in a solution with value 14.5. It is not possible for the ILP to have a solution higher than this. The solution to the ILP is, in fact, 12. Next, consider a solution to the relaxed LP. If we have a non-integer variable, say  $x_i = r, r \in \mathbb{R}$ , then the actual value of  $x_i$  in the solution to the ILP is either  $\geq \lceil r \rceil$  or  $\leq \lfloor r \rfloor$ . We can combine these two observations to create a *branch-and-bound* algorithm to solve ILPs.

The branch-and-bound algorithm is similar to the enumeration approach we discussed earlier, but here we build the tree incrementally and we only follow branches that are promising, i.e. branches that may lead to the optimal solution. The branch-and-bound algorithm has three main concepts. The first is branching, where we select a node in the tree to investigate further. The next is bounding, where we obtain new upper/lower bound to the solution by exploring nodes. Lastly, there is pruning where we permanently discard a branch in the tree. The best way to understand how branch-and-bound works is through an example.

Consider the ILP (2) (from [1]). We start by solving the LP relaxation of ILP (2).

$$\begin{aligned}
 & \mathbf{max} \quad 2x_1 + 3x_2 & (2) \\
 & \mathbf{subject \ to} \\
 & \quad x_1 + 3x_2 \leq 8.25 \\
 & \quad 2.5x_1 + x_2 \leq 8.75 \\
 & \quad x_1, x_2 \geq 0 \text{ and integer}
 \end{aligned}$$

We are going to build a tree with the LP relaxation of the ILP as the root. Solving the LP relaxation results in a solution where  $x_1 = 2.796$  and  $x_2 = 1.826$ . These are not integer values, so this is not a solution to the ILP. We now know that in the integer optimal solution, either  $x_1 \geq 3$  or  $x_1 \leq 2$  since it cannot take a real-number value.

We are now going to branch from the root. We do this by selecting one of the variables, say  $x_1$ , and adding two constraints to get the two branches. For the first branch, we create a new ILP with an additional constraint of  $x_1 \leq 2$  and for the second branch we create another new ILP with the constraint  $x_1 \geq 3$ . We then choose one of the nodes and solve the LP relaxation. This is shown in Figure 3(a). In the figure we have the original ILP and the result from its LP relaxation. Then we added two new nodes to the tree by adding the constraints above. We chose to solve the left node's LP relaxation. We have not yet found a solution with only integer values, and we haven't explored all possible branches in the tree, so we must continue expanding the tree. To do this we branch the left node to get the tree in Figure 3(b).

The solution to the LP relaxation of this node results in a feasible solution (all integer values) with an objective value of 10. There is still more of the tree to search though, so we choose the sibling node next. The relaxation to this node is not feasible though, so we are done here as shown in Figure 3(c).

We move to the next node that is unsolved and solve the LP relaxation. This results in the values shown in Figure 3(d). The objective function of this relaxation is 9.75. This value is less than our current feasible solution (10.25), so we can prune this branch (no nodes in this branch can be better than our current feasible solution). At this point, we

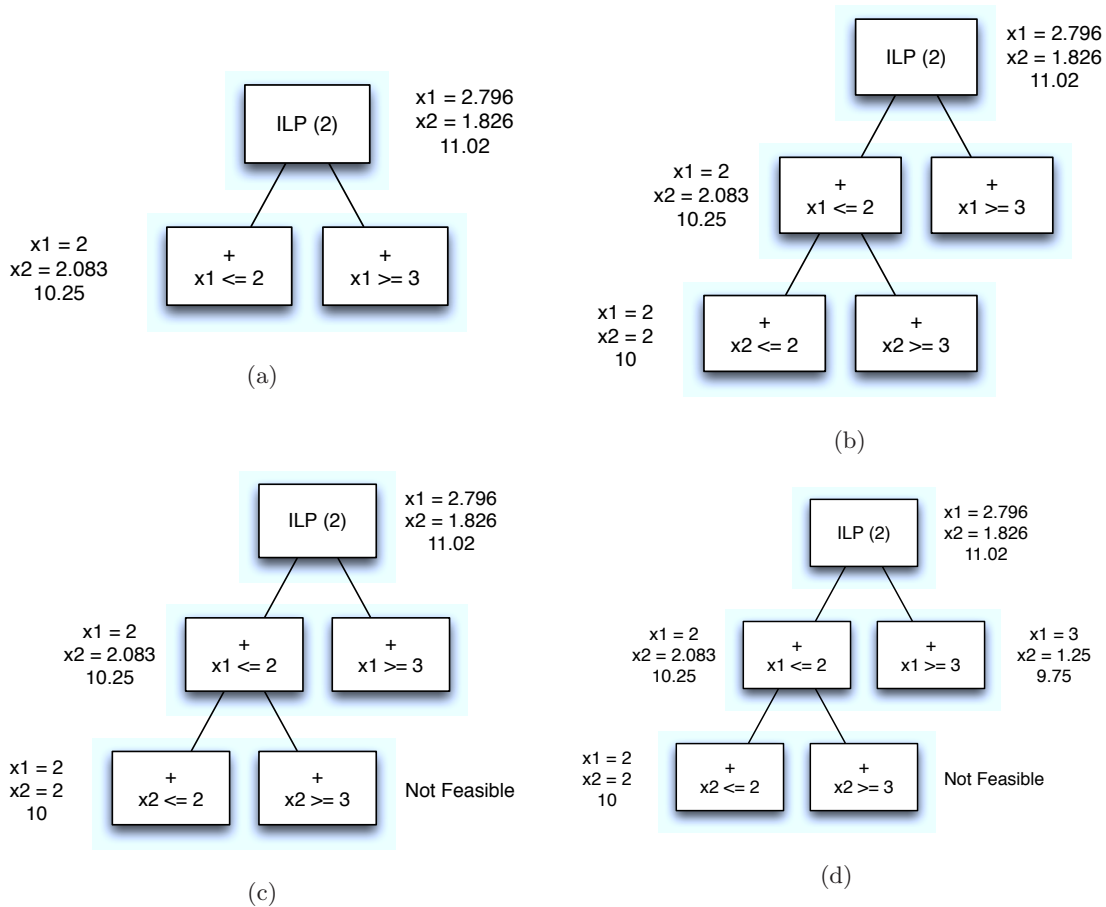


Figure 3: Sample run of a branch-and-bound algorithm on ILP (2). In (a) the original relaxation is solved and variable  $x_1$  is chosen to create two branches. We then solve the left branch. In (b), we branch on  $x_2$  and the left child results in an integer solution. The bound is now set to 10. In (c) the other branch is solved, but is not feasible so we do not continue with it. In (d) we still have one more branch to explore (the root's right child). Solving the LP relaxation results in a solution with objective value 9.75. This is lower than our bound (of 10), so we can prune this branch. There are no more branches to explore so we are done.

have no more nodes to branch from, so we are done. The optimal solution to ILP (2) is  $x_1 = x_2 = 2$ .

The basic branch-and-bound algorithm can be described as follows:

- 1: Solve LP relaxation on current branch
  - 1a: If the relaxation is less (for maximization problems) than our current bound (best feasible solution), terminate this branch.
  - 1b: Otherwise, if the solution is integer, update the bound if necessary.
- 2: Create two more branches. Select a variable that is not an integer and add two constraints.
- 3: Repeat until no branches remain.

The integer feasible solutions we discover create a bound we can use to prune branches, which allows us to skip large parts of the tree.

This is the simplest approach to solving ILPs. ILP solvers implement much more complicated algorithms and heuristics, but are typically based on the same branch-and-bound concept. The only way to find the optimal solution is to explore the entire enumeration tree. If we are able to prune many branches then finding the optimal solution may not take long. If we cannot, however, the runtime is exponential in the number of variables (which can be in the thousands). It is important that ILP solvers use heuristics and advance algorithms that allow them to prune the tree as much as possible in order to provide solutions to complex ILPs. In theory it is possible to solve any ILP with this technique, but it is often not practical to do so for large ILPs due to their complexity.

## Shortest Path Example

We previously looked at the problem of finding the shortest path in a network between two nodes by using constraints from the Bellman-Ford algorithm. In this section, we'll create an alternate formulation of the problem using *indicator variables*. Indicator variables are just variables that take the value of 0 or 1. In this case, we will use indicator values to determine if an edge  $e \in E$  belongs on the shortest path or not. The problem is the same. We are given a network,  $G = (V, E)$ , with a cost function  $c : E \rightarrow \mathbb{R}^+$ , a source  $s \in V$ , and a destination  $d \in V$ . We want to find the shortest path from  $s$  to  $d$ . Here, we will find the cost and the path, instead of just the cost like last time.

Let's introduce a binary variable  $x$ . We'll use this as our indicator variable. We want  $x_{i,j} = 1$  if the link  $(i, j)$  is used in the shortest path, and 0 otherwise. In this way,  $x$  is used to indicate which links are used on the path. We'll start with the objective function. We want to minimize the cost of the path since we are looking for the shortest path:

$$\text{minimize } \sum_i \sum_j x_{i,j} c(i, j)$$

This objective function will ensure that the path we find is the shortest path. Now we need to define the constraints that will ensure the path we find is a valid path. The constraints need to make sure that our indicator  $x$  works as we expect. Lets consider the source node. We know that there should be no edges of the path going *into* the source node. We also know there should be exactly one edge of the path leaving the source node. We can turn this into a constraint:

$$\sum_i x_{i,s} - \sum_j x_{s,j} = -1$$

The first summation represents the edges coming into the source node. We shouldn't have any so this should be zero. The second summation is the edges leaving the source node and we should have one.

We have a similar constraint for the destination node. There should be exactly one edge on the path entering the destination, and no edges leaving the destination:

$$\sum_i x_{i,d} - \sum_j x_{d,j} = 1$$

Now consider the intermediate nodes on the path. If there is an edge on the path entering node  $x$ , then node  $x$  should also have another edge on the path leaving it (otherwise it'd be the destination). This can be captured in the following constraint that ensures any intermediate node with an incoming edge on the path also has an outgoing edge:

$$\sum_i x_{i,k} - \sum_j x_{k,j} = 0 \quad \forall k \neq s, d$$

Because of the first two constraints, the last constraint will also ensure that intermediate nodes only have a single incoming and outgoing edge on a path (instead of multiple incoming/outgoing).

The best way to understand how this works is by running this ILP through a solver and looking at the output. The final formulation is shown below followed by a MathProg script to solve it. The network in the script is NSFnet. It may help by substituting a smaller network at first.

Given:

- $s$ : The source node.
- $d$ : The destination node.
- $c(i, j)$ : Cost function defining the cost of link  $(i, j)$ .

Find:

- $x_{i,j}$ : 1 iff link  $(i, j)$  is used on the shortest path from  $s$  to  $d$ .

$$\begin{aligned} & \textbf{minimize} && \sum_i \sum_j x_{i,j} c(i, j) \\ & \textbf{subject to:} && \sum_i x_{i,s} - \sum_j x_{s,j} = -1 \\ & && \sum_i x_{i,d} - \sum_j x_{d,j} = 1 \\ & && \sum_i x_{i,k} - \sum_j x_{k,j} = 0 \quad \forall k \neq s, d \\ & && x_{i,j} \in \{0, 1\} \end{aligned}$$

Listing 1: Shortest Paths ILP in MathProg

```

1 set NODES := {1..14};
2 param s := 1;
3 param d := 11;
4 set OTHERS := NODES diff {s,d};
5 param C {NODES,NODES};
6
7 var x {NODES, NODES} binary;
8
9 minimize cost: sum{i in NODES, j in NODES} x[i,j]*C[i,j];
10
11 s.t. sNode: (sum { i in NODES } x[i,s]) -
12           (sum { j in NODES } x[s,j]) = -1;
13 s.t. dNode: (sum { i in NODES } x[i, d]) -
14           (sum {j in NODES } x[d,j]) = 1;
15 s.t. flow{k in OTHERS}: (sum {i in NODES } x[i,k]) -
16           (sum{j in NODES} x[k, j]) = 0;
17 s.t. hasLink{i in NODES, j in NODES}: x[i,j]*C[i,j] <= 999;
18
19 solve;
20
21 printf "Cost of path: %f\n",
22       sum{i in NODES, j in NODES} x[i,j]*C[i,j];
23
24 printf{i in NODES, j in NODES} "link %d-%d is %s\n",
25       i, j, (if x[i,j] < 1 then "UNUSED" else "USED");
26
27 data;
28
29 param C: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 :=
30 1 0 110 160 1000 1000 1000 1000 280 1000 1000 1000 1000 1000
31 2 110 0 60 100 1000 1000 1000 1000 1000 1000 1000 1000 1000
32 3 160 60 0 1000 1000 200 1000 1000 1000 1000 1000 1000 1000
33 4 1000 100 1000 0 60 1000 1000 1000 1000 1000 240 1000 1000
34 5 1000 1000 1000 60 0 110 80 1000 1000 1000 1000 1000 1000
35 6 1000 1000 200 1000 110 0 1000 1000 1000 120 1000 1000 200
36 7 1000 1000 1000 1000 80 1000 0 70 1000 1000 1000 1000 1000
37 8 280 1000 1000 1000 1000 1000 70 0 70 1000 1000 1000 1000
38 9 1000 1000 1000 1000 1000 1000 1000 70 0 90 1000 50 1000
39 10 1000 1000 1000 1000 1000 120 1000 1000 90 0 1000 1000 1000
40 11 1000 1000 1000 240 1000 1000 1000 1000 1000 1000 0 80 1000
41 12 1000 1000 1000 1000 1000 1000 1000 1000 50 1000 80 0 30
42 13 1000 1000 1000 1000 1000 200 1000 1000 1000 1000 1000 30 0
43 14 1000 1000 1000 1000 1000 1000 1000 1000 50 1000 80 1000 30
44
45 end;

```

The MathProg script should be self-explanatory based on what we've discussed previously. Notice on line 4 there is a *diff* function that performs set difference. Using an ILP to solve the shortest path problem is not a good idea since there are much more efficient algorithms to do so. This example provides a good illustration of indicator variables that will be used in ILPs for RWA in the next section.

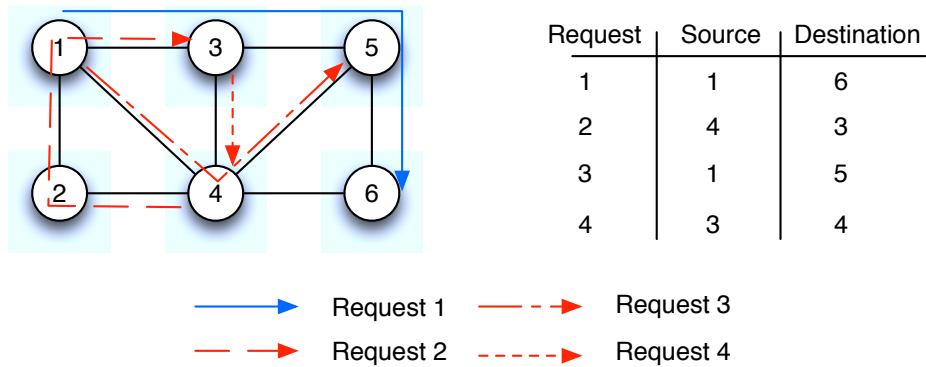


Figure 4: Example static unicast RWA. Two wavelengths are required since two requests share link (1-3).

## Static Unicast Routing and Wavelength Assignment

This section will go over the formulation of an ILP for unicast static routing and wavelength assignment (RWA) with the wavelength continuity constraint. We use the same concept of indicator variables as the previous section to tell us which unicast request uses which wavelength on which links. The goal of our ILP will be to minimize the number of wavelengths required. It will find routes and assign wavelengths to those routes for all requests in a static request set.

Let's define the problem formally and build the ILP incrementally. We are given a graph  $G = (V, E)$  representing a network. A unicast request can be defined as a two-tuple,  $R = (s, d)$ , where  $s \in V$  is the source node and  $d \in V$  is the destination. We will assume each request requires one wavelength. We are given a set of unicast requests,  $\Delta = \{R_1, R_2, \dots, R_n\}$ . We must assign a lightpath to each request while minimizing the number of wavelengths used. We assume that the wavelength continuity constraint applies, so each request can use only a single wavelength on all its links. We also assume the wavelength clash constraint applies, meaning on a given link, any particular wavelength can only be used once.

An example of static unicast RWA can be seen in Figure 4. Here there are four requests in the static set. The set is shown in the table on the right. We must find a route and assign a wavelength to each one and minimize the number of wavelengths required. In this example, there are two wavelengths required. Request 1 has its own wavelength, while the other three share a wavelength. We need two wavelengths because two requests share link (1,3) and because of the wavelength clash constraint, they cannot use the same wavelength.

Solving RWA is NP-complete so people typically write heuristics to solve the problem sub-optimally. We will now write an ILP to solve it optimally. Our objective is to minimize the number of wavelengths and we need to find which paths and which wavelengths each request uses. We'll use an indicator variable, similar to what we used for our shortest path example. Here, the indicator variable,  $y$ , will determine which wavelength and which links each request uses.  $y_{i,j}^{r,w}$  will be 1 if request  $r$  uses wavelength  $w$  on link  $(i, j)$ . To see how

this works, consider the example in Figure 4. For request 1, the indicator variable will be 1 for the following:  $y_{1,3}^{1,2}, y_{3,5}^{1,2}, y_{5,6}^{1,2}$ . This says that request 1, using wavelength 2, uses links (1,3), (3,5), (5, 6). The variable will be 0 for  $y_{4,6}^{1,2}$ , for example, since request 1 does not use link (4,6) on wavelength 2. We will use another indicator variable,  $C^{r,w}$ , which will be 1 iff request  $r$  uses wavelength  $w$ . Since we have the wavelength continuity constraint, each request will use exactly one wavelength. This variable will allow us to ensure that each request uses a single wavelength. In the example in Figure 4,  $C^{4,1}$  and  $C^{1,2}$  would both be set to 1 since request 4 uses wavelength 1 and request 1 uses wavelength 2. We need one final variable,  $maxWL$ , which is used to keep track of the highest wavelength index used in the network. In the example,  $maxWL$  would be equal to 2 since we used two wavelengths.

Those are the variables the ILP will solve for. We need to give the ILP our adjacency matrix, or the cost function, similar to our shortest path example. We also need to give it the source and destinations of each request in the static set. We'll use  $s_i$  and  $d_i$  to represent the source and destination node of request  $i$ . Lastly, we need to give it some maximum number of wavelengths,  $W$ , that it can use (it will use the minimum number possible).

We're now ready to start defining the objective function and constraints. The objective function is easy. We want to minimize the number of wavelengths used. We have a variable that represents this,  $maxWL$ , so our objective is simply: **minimize:**  $maxWL$ . Now we have the constraints. The constraints must force the variables we defined earlier to take on only valid values. Let's start with  $maxWL$ . We want this variable to represent the maximum wavelength in use in the network. To find this, we can introduce the following constraint:

$$maxWL \geq C^{r,w} * w \quad \forall r, w$$

This constraint will make sure  $maxWL$  is as big as the highest wavelength in use. Consider our previous example,  $C^{1,2}, C^{2,1}, C^{3,1}, C^{4,1}$  where all set to 1. Now this means that  $maxWL \geq 2$  since  $C^{1,2} * 2$  is the largest value for any  $C$  taking the value 1.

Next, lets write out the constraint for the wavelength clash constraint. This says that no link in the network can have the same wavelength used more than once. It can be written as:

$$\sum_r (y_{i,j}^{r,w} + y_{j,i}^{r,w}) \leq 1 \quad \forall i, j, w$$

We check this constraint for every individual link and every wavelength on that link. We make sure that *at most* 1 request is using that particular wavelength on that particular link. In the previous example, this will be check for link (1, 2) on wavelength 1 (and all other links/wavelengths). It will ensure that  $(y_{1,2}^{1,1} + y_{2,1}^{1,1}) + (y_{1,2}^{2,1} + y_{2,1}^{2,1}) + (y_{1,2}^{3,1} + y_{2,1}^{3,1}) + (y_{1,2}^{4,1} + y_{2,1}^{4,1}) \leq 1$ . This means that at most one of these variables can be set to 1, in other words only 1 request can use this link on wavelength 1, because if more than one request used it then the above equation could not be less than or equal to 1.

Now let's look at the constraints for the wavelength continuity constraint, which says that each request must use exactly one wavelength on all the links. There are two constraints in the ILP to handle this. We have to make sure both  $C$  and  $y$  have valid values with respect to the wavelength continuity constraint. The first constraint can be written as:

$$\sum_w C^{r,w} = 1 \quad \forall r$$

This says that each request must use *exactly* one wavelength. For example, take request 1 in the previous example. This will expand to  $C^{1,1} + C^{1,2} = 1$ , so only one of the variables can be set to one (in this case it was  $C^{1,2}$ ). We must also make sure that the variable  $y$ , which tells us the routes and wavelengths assigned for each request, uses the wavelength specified by  $C$ .

$$y_{i,j}^{r,w} + y_{j,i}^{r,w} \leq C^{r,w} \quad \forall r, w, i, j (j > i)$$

The previous constraint ensures that exactly one  $C^{r,w}$  is set to 1 for each request  $r$ . Now this constraint makes sure that each  $y_{i,j}^{r,w}$  uses the same wavelength. If  $C^{r,w}$  is 0, then the request cannot use any links on that wavelength because the sum cannot be greater than 0. Only when  $C^{r,w}$  is 1 for a particular wavelength can the request use that wavelength over any links. For example,  $y_{1,3}^{1,2} + y_{3,1}^{1,2} \leq C^{1,2}$  is valid because request 1 uses wavelength 2, so request 1 can use link (1,3) on wavelength 2 (assuming  $C^{1,2}$  is set to 1).

Those constraints take care of wavelength assignment. We now need constraints to take care of routing. We can use the same constraints as the shortest path example. First, we have the constraint that ensures no edges on the path enter the source node, we can only have edges leaving the source node on the path to the destination.

$$\sum_w \sum_i y_{i,s_i}^{r,w} - \sum_w \sum_j y_{s_i,j}^{r,w} = -1 \quad \forall r$$

The constraint is the same as the shortest path routing problem, except for the fact that we also sum over all the wavelengths as well since each request can only use a single wavelength. We also have multiple paths we are checking so we apply this constraint for all requests  $r$ .

Lastly, we have our other two constraints for routing, same as the shortest path problem

$$\sum_w \sum_i y_{i,d_i}^{r,w} - \sum_w \sum_j y_{d_i,j}^{r,w} = 1 \quad \forall r$$

$$\sum_w \sum_i y_{i,k}^{r,w} - \sum_w \sum_j y_{k,j}^{r,w} = 0 \quad \forall r, k (\neq i, j)$$

That completes the ILP for static unicast RWA. We have constraints that ensure the wavelength continuity and clash constraint are satisfied. We also have constraints to ensure the routing is valid. The number of variables for this ILP can grow to be very larger. Each  $y_{i,j}^{r,w}$  is considered a variable. You can think of this as a 4-D array. If we have 16 links, 4 requests, and a maximum of 4 wavelengths, then we have  $16 * 4 * 4 = 256$  variables (not including our other variables). Let's consider a more realistic example. Say we have

NSFnet, with 42 links. If we have 100 requests, we can use 100 as the maximum number of wavelengths required. This ILP would have 420,000 variables. Remember, the complexity of the ILP grows *exponentially* in the number of variables. This should give you some idea as to why ILPs are not practical for large networks.

The full ILP is shown below. Along with this PDF there are also files with the MathProg script that implements it with the network and requests shown in Figure 4. Note,  $W$ , in this ILP can be found using some heuristic. The smaller we can get  $W$  the faster the ILP will run because there will be fewer variables.

Given:

- $s_i$ : The source node of request  $i$ .
- $d_i$ : The destination node of request  $i$ .
- $W$ : The maximum number of wavelengths.

Find:

- $y_{i,j}^{r,w}$ : 1 iff wavelength  $w$  is used on link  $(i,j)$  for the lightpath of request  $r$ .
- $maxWL$ : The maximum number of wavelengths required for the RWA.
- $C^{r,w}$ : 1 iff request  $r$  uses wavelength  $w$ .

$$\begin{aligned}
& \mathbf{minimize} \quad maxWL \\
& \mathbf{subject\ to:} \quad maxWL \geq C^{r,w} * w \quad \forall r, w \\
& \quad \sum_r (y_{i,j}^{r,w} + y_{j,i}^{r,w}) \leq 1 \quad \forall i, j, w \\
& \quad \sum_w C^{r,w} = 1 \quad \forall r \\
& \quad y_{i,j}^{r,w} + y_{j,i}^{r,w} \leq C^{r,w} \quad \forall r, w, i, j (j > i) \\
& \quad \sum_w \sum_i y_{i,s_i}^{r,w} - \sum_w \sum_j y_{s_i,j}^{r,w} = -1 \quad \forall r \\
& \quad \sum_w \sum_i y_{i,d_i}^{r,w} - \sum_w \sum_j y_{d_i,j}^{r,w} = 1 \quad \forall r \\
& \quad \sum_w \sum_i y_{i,k}^{r,w} - \sum_w \sum_j y_{k,j}^{r,w} = 0 \quad \forall r, k (\neq i, j) \\
& \quad y_{i,j}^{r,w} \in \{0, 1\} \quad \forall i, j, r, w \\
& \quad maxWL \geq 1, \text{ integer} \\
& \quad C^{r,w} \in \{0, 1\} \quad \forall r, w
\end{aligned}$$

## Multicast Routing and Wavelength Assignment

In this section, we'll see an example of multicast RWA. A unicast request has a single source and single destination. A multicast request has a single source and a *set* of destinations. We can define a multicast request formally as  $R = (s, D)$  where  $s \in V$  and  $D \subseteq V - \{s\}$ . The static multicast RWA problem is similar to the unicast problem but now we need to find a light-tree instead of a lightpath for each request. The constraints dealing with the wavelength constraints are similar, but the routing constraints are different because we are building a tree instead of a simple path. The ILP is below, we'll look at the constraints one by one.

The parameters to the ILP are:

- $s_m$ : The source node of request  $m$ .
- $D_m$ : The destination set of request  $m$ .

The ILP will solve for the following variables:

- $y_{i,j}^{m,w}$ : 1 if wavelength  $w$  is used on link  $(i, j)$  for multicast request  $m$ , 0 otherwise.
- $maxWL$ : The largest wavelength index in use.
- $C^{m,w}$ : 1 if wavelength  $w$  is used by request  $m$ , 0 otherwise.
- $U_i^m$ : order node  $i$  was added to the tree for multicast request  $m$ . This prevents loops.

**minimize**  $maxWL$

$$\mathbf{subject\ to:} \quad maxWL \geq C^{m,w} * w \quad \forall m, w \quad (3)$$

$$\sum_m (y_{i,j}^{m,w} + y_{j,i}^{m,w}) \leq 1 \quad \forall i, j, w \quad (4)$$

$$\sum_w C^{m,w} = 1 \quad \forall m \quad (5)$$

$$y_{i,j}^{m,w} + y_{j,i}^{m,w} \leq C^{m,w} \quad \forall m, w, i, j (j > i) \quad (6)$$

$$\sum_i \sum_w y_{i,j}^{m,w} = 1 \quad \forall m, j \in D_m \quad (7)$$

$$\sum_j \sum_w y_{s_m,j}^{m,w} \geq 1 \quad \forall m \quad (8)$$

$$\sum_j \sum_w y_{j,s_m}^{m,w} = 0 \quad \forall m \quad (9)$$

$$\sum_j \sum_w y_{j,i}^{m,w} \leq 1 \quad \forall m, i \neq s_m \quad (10)$$

$$\sum_j y_{i,j}^{m,w} - |V| \sum_j y_{j,i}^{m,w} \leq 0 \quad \forall m, w, i \neq s_m \quad (11)$$

$$\sum_j y_{j,i}^{m,w} - \sum_j y_{i,j}^{m,w} \leq 0 \quad \forall m, w, i \neq D_m \quad (12)$$

$$U_i^m - U_j^m + |V| y_{i,j}^{m,w} \leq |V| - 1 \quad \forall m, w, i, j \quad (13)$$

The parameters are the same as the unicast case except now we have destination sets instead of single destinations. We have one additional variable,  $U$ , that will help us create the trees (we will see shortly). The objective function is the same. Constraints (3)-(6) are all in the unicast ILP. They deal with the wavelength assignment. Constraints (7)-(13) are different since they deal with routing of a tree not a path. Constraint (7) specifies that each destination in the destination set must have an incoming edge on the tree (each destination must be reached). Constraint (8) specifies that the source node must have at least one outgoing edge on the tree. In the unicast case, it must have *exactly* one, but here we are making a tree so it can have more than one. Constraint (9) specifies that the source cannot have any incoming edges on the tree (we have a similar constraint for the unicast case). Constraint (10) specifies that each node (other than the source) can have at most one edge coming into it (otherwise it would not be a tree). Constraint (11) specifies that a node can have outgoing edges on the tree only if it has an incoming edge. If a node  $i$  has no incoming edge, then the second summation will be 0 so the first summation (the outgoing edges) would also have to be 0. Constraint (12) specifies that any intermediate nodes in the tree (not including the destination) that have an incoming edge must have one or more outgoing edges. Constraint (13) prevents loops from forming. For example, if we did not have this constraint, we could have a part of the tree that goes from node 1 to node 2 to node 3 and back to node 1. There are no constraints yet to prevent this. With constraint (13), if there is an edge on the tree from  $i$  to  $j$ , then  $U_i^m < U_j^m$ . Now if we try to add an edge that goes from a node added later to a node added earlier (the node added later would have a higher  $U_i^m$  value), constraint (13) would be violated.

## Using CPLEX

Up to this point, we have used the GNU Linear Programming Kit to solve our LP/ILPs. This solver works well for LPs or small ILPs, but as the ILPs get larger, GLPK simply cannot handle them. The commercial solver CPLEX (by IBM) can solve large problems significantly faster than any of the open source solvers.

There are different ways to run CPLEX. They provide both an interactive mode, where you run the program from the command prompt, and a programming API where you can access the solver through various programming languages. There is also a program called AMPL that allows you to write LP/ILPs in a language very similar to MathProg (MathProg is actually a subset of AMPL) and then uses CPLEX to execute them for you. This section will assume that you do not have AMPL, but still want to use MathProg, which we have been learning, to use CPLEX. CPLEX does not support MathProg natively, so we cannot use it directly.

## Converting MathProg to MPS

In order to use MathProg with CPLEX (without having a license to AMPL), we need to convert our MathProg script to something CPLEX can read. This can be done using the GNU Linear Programming Kit (the same solver we have been using). We can simply issue a single command to get a format CPLEX can read:

```
$ > glpsol -m <MathProg model> --check --wfreemps <output>.mps
```

This command will read your MathProg model, verify it, then output it to the MPS format. This .mps file can then be used in CPLEX.

## Running CPLEX in Interactive Mode

To run CPLEX in interactive mode, just run the binary named *cplex*. This will start CPLEX up and give you a command prompt. Say that the .mps file created by GLPK is called *output.mps*. The first thing to do is tell CPLEX to read the file with the command: *read output.mps*. When CPLEX solves ILPs, it allows integer values to be a very small fraction off of actual integer values. If you need exact integer values instead of very close estimates, you can issue the command *set mip tolerances integrality 0*. Now to actually solve the problem, issue the command *optimize*. CPLEX will display the progress the solver makes. Once it is done you will see the objective function, runtime, and some other stats. You can look at the value of the variables here using the *display* command or write them to a file. To write only the non-zero variables, issue the command *set output writelevel 3*, followed by *write output.sol*. The file *output.sol* is an XML file that contains the value of all your variables. You can parse this to extract whatever information you need. You can exit the command prompt by issuing the *quit* command.

CPLEX has many options that allow you to configure how the solver works. I'll only mention a few here. The user manual has many more details.

- *Tuning*: CPLEX comes with a tuning tool that can help you solve your ILPs faster. The tuning tool will try running your ILP with a number of different settings and tell you which is the best.
- *Emphasis*: You can tell CPLEX whether it should focus on finding the optimal solution or focus on finding some feasible (though maybe not optimal) integer solution. The command is *set emphasis mip X*, where X can be 0-4. 0 means equal emphasis on optimality and feasibility, 1 is more emphasis on feasibility, 2 is more emphasis on optimality, 3 is exclusively search on optimal, and 4 is almost exclusive search for feasible solutions.
- *Restarting CPLEX*: You can give CPLEX an initial solution file that you obtained either from a previous unfinished run of CPLEX or created by some heuristic program for the same problem. The files are saved in the MST format (more information available from the user manual). Once your problem file (mps) is read in, you can read in the MST file and CPLEX will use this information for solving the problem (hopefully in less time with this initial solution).

## Example Files

Included with this document are several sample scripts. I've included all of the MathProg scripts for the simple LPs (from the first set of notes) and the shortest path ILP. I have

included a set of scripts for the unicast RWA ILP. Here is an explanation of each of the files in the *unicast* folder.

- *6node.txt*: This file represents the network as an adjacency matrix. The first line of the file is the number of nodes and the remaining is the adjacency matrix. Values of 0 or -1 indicate no link.
- *reqs.txt*: I have another program that generates request sets and writes them to a file. In this case, each request is a unicast request and takes up two lines, the first line is the source and next is the destination. These requests correspond to the requests from Figure 4.
- *template.mod*: This is the MathProg file for unicast RWA. It has a few placeholder text: NUM\_WL, NUM\_NODES, NUM\_REQ. These typically change for different runs so the script described next fills in variables for them.
- *makeILP.sh*: This BASH script will take the *template.mod* file and replace the placeholder text with actual values passed as arguments. The final MathProg script will be called ILP.mod.
- *gen\_data.py*: The data for the ILP is stored in a separate data file. This python script will generate that file. Given the network file (*6node.txt*) and the requests file (*reqs.txt*), it will generate ILP.dat, which contains the data for ILP.mod.

Typically, you want to compare the ILP results to the results from some heuristic you wrote. In this case, I have a program that implements the heuristic. It generates a request set, writes the set to a file, and solves the problem returning the number of wavelengths required. The request set and number of wavelengths are then used with *makeILP.sh* and *gen\_data.py* to create the MathProg files. Then *glpsol* is called to generate the MPS files for CPLEX.

## References

- [1] "Integer linear programming." [Online]. Available: <http://classweb.gmu.edu/aloerch/IP540.pdf>