ENGRI1101: Introduction to Operations Research

Instructor: Frans Schalekamp, Cornell

Notes by Joyce Shen, FA24

1 Traveling Salesman Problem

- **Intuition**: want to visit n cities and minimize total distance traveled
- **Input**: integer n specifying number of cities + costs $c(i,j)$ for each star/city $i = 1, 2, ...n$ and $j = 1, 2, ..., n$
- **Output**: $\pi(1), \pi(2), ..., \pi(n)$ where π is the permutations of numbers $1, ..., n$ where $\pi(i)$ is the i-th star/city to be visited
- **Goal**: find a solution π that minimizes the total travel time, i.e.,

$$\text{minimize} \sum_{i=1}^{n-1} [c(\pi(i, i+1))] + \pi(n-1, n)$$

We won't know a 'fast' algorithm (besides try everything aka brute force) that always gives an optimal solution for TSP – for an input of size n, we have $(n-1)!$ possible solutions (because we can start from any arbitrary city). Instead, have possible **heuristics** (a fast algorithm that is not necessarily optimal; good guess)

1.1 Heuristics

Random Neighbor

Algorithm: Start at some node. Randomly select one of the nodes which has not been visited to visit next until all nodes have been visited. This is a bad heuristic.

Nearest Neighbor

Algorithm: Start at some node. Visit the closest unvisited node next (if there are multiple closest nodes, choose one randomly) until all nodes have been visited. Return to the start.

Notes: Doesn't make 'smart' choices when some nodes are equidistant, doesn't return the same path every time as there is randomness (when choosing arbitrary nodes).

Nearest Insertion

Algorithm: Start with a "tour" on two of the nodes (e.g., the closest pair of nodes). Find the closest unvisited node to any node currently in tour. Insert the node into the tour at the best place (if there are multiple closest nodes, choose one to add randomly).

Notes: Slightly better than nearest neighbor.

Farthest Insertion

Algorithm: Start with a "tour" on two of the nodes (e.g., the closest pair of nodes). Find the node whose smallest distance to a node already in the tour is maximized. Insert the node into the tour at the best place (if there are multiple farthest nodes, choose one to add randomly).

Notes: On average (+ by observation from lab), this is the best heuristic so far.

OPT2

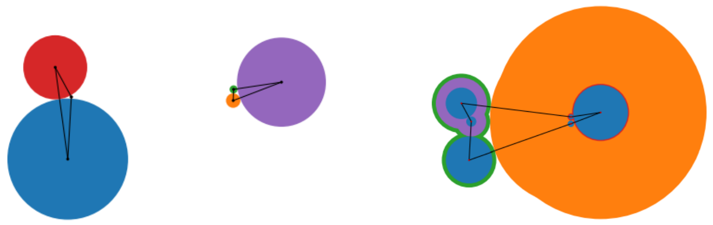
Besides heuristics to create tours, can also use heuristics to improve existing tours. For example if the tours produce paths the cross at any point, can 'uncross' them to improve the distance – OPT2-ing any of the above heuristics brings the path length very close to the optimal (but not quite).

Argument of Optimality

To argue that a solution is **optimal**, one approach is to look at **lower bounds**.

For example, a lower bound for the TSP could be $nc_{\min}$, as we know we need to visit all n cities and therefore transition n times, where $c_{\min}$ is the minimum distance between two cities.

Alternatively, we can use the idea of **discs**, where we create *nonoverlapping discs* around each node. Because they are nonoverlapping, we know any tour needs to enter and leave each node's disc, adding a distance of $2r$ to the total tour. On top of discs, we can add **moats** around the discs to account for empty space:



Although this is a better lower bound, this strategy of **discs and moats** does not always produce a lower bound equal to the length of the shortest tour.

2 Shortest Path Problem

- **Intuition**: want to find the fastest/cheapest/shortest route between two points
- **Input**: a directed graph $G = (V, A)$, a source node $s \in V$, and a nonnegative length $\ell(i, j)$ for each arc $(i, j) \in A$
- **Output**: a path from s to each node $i \in V$
- **Goal**: minimize the length of each path from s to $i \in V$

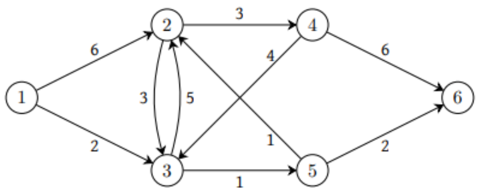
2.1 Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path for a directed graph with *nonnegative* weights, outputting a **shortest path tree**.

Algorithm 1 Dijkstra's Algorithm

```
$S \leftarrow \emptyset$
best$(i) \leftarrow +\infty$ and from$(i) \leftarrow$ undefined for all $i \in V$
settle the vertex closest to $s$
**while** there is still some $i \in V$ that hasn't been settled **do**
    settle the shortest reachable node $n$ from the currently settled vertex
        aka: (add $n$ to $S$)
    update best$(n)$ and from$(n)$ accordingly
**end while**
**return** $S$
```

Can be helpful to trace through Dijkstra's algorithm with a table to keep track of the best(i) and from(i) for each $i \in V$:



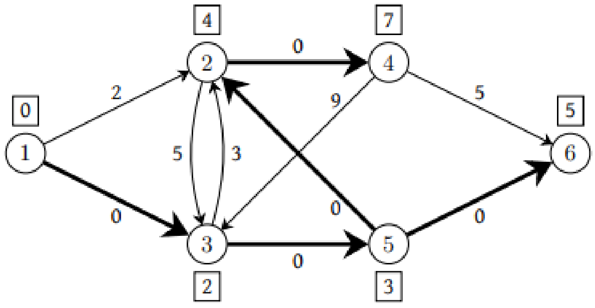
	1		2		3		4		5		6	
	best	from	best	from	best	from	best	from	best	from	best	from
1	0	–	∞	none	∞	none	∞	none	∞	none	∞	none
2			6	1	2	1	∞	none	∞	none	∞	none
3			6	1			∞	none	3	3	∞	none
4			4	5			∞	none			5	3
5							7	2			5	3
6							7	2				

Argument of Optimality

To verify the solution + that the solution is actually optimal, can modify each of the costs such that

$$\bar{\ell}(i,j) = \ell(i,j) + \text{best}(i) - \text{best}(j)$$

By doing so, for each arc (i,j) in the path given by Dijkstra, verify that the modified cost is 0. If so, then it is indeed a shortest path.
** Note that we can easily see this bc the new costs are still nonnegative – thus the lower bound of the cost is 0 – precisely the costs given by Dijkstra's.



Called the idea of **presents and penalties**, where a penalty is applied every time one *enters* a node ($+\text{best}(i)$) and a present is given every one *exits* a node ($-\text{best}(j)$).

3 Minimum Spanning Tree Problem

- **Intuition**: want to connect a network of nodes together in the cheapest way possible
- **Input**: undirected graph $G = (V, E)$ and a nonnegative cost $c(i, j)$ for each $\{i, j\} \in E$
- **Output**: a subset $T \subseteq E$ such that any two nodes in V are connected just through edges in T
- **Goal**: minimize the total cost of edges in T, i.e.,

$$\text{minimize} \sum_{\{i,j\} \in T} c(i, j)$$

Like the MST, there is an actual algorithm to find the optimal solution. In fact, there are multiple: Note that all these algorithms produce *an* optimal solution, but by no means is the optimal solution unique, meaning these algorithms can all return different optimal solutions. Also note that the optimal solution always has $n - 1$ edges.

3.1 Kruskal's Algorithm

Algorithm 2 Kruskal's Algorithm - $O(m \log n)$

```
Sort edges $e \in E$ from cheapest to most expensive
$T \leftarrow \emptyset$
for each edge $e \in E$ do
    if $e$ does not create a cycle in $T$ then
        Add $e$ to $T$
    end if
end for
return $T$
```

3.2 Prim's Algorithm

Algorithm 3 Prim's Algorithm - $O(m \log n)$

```
$T \leftarrow \emptyset$
take an arbitrary $r \in V$ as the root
while $(V, T)$ is not connected do
    add the cheapest edge out of $r$'s components to $T$
end while
return $T$
```

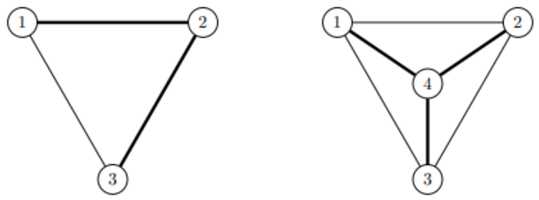
3.3 Reverse Delete

Algorithm 4 Reverse Delete - $O(m \log m)$

```
Sort edges $e \in E$ from most expensive to cheapest
$T \leftarrow E$
for each edge $e \in E$ do
    if removing $e$ does not disconnect $T$ then
        remove $e$ from $T$
    end if
end for
return $T$
```

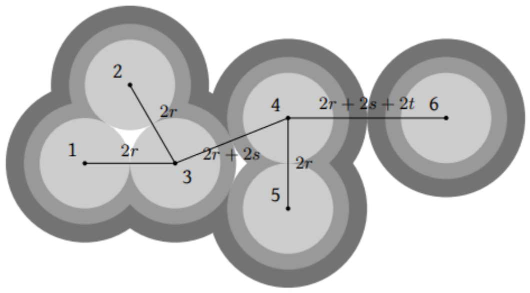
Model Flaws - Steiner Tree Problem

Note that our model is actually flawed in the real world. Sometimes the cheapest way to connect all the nodes we want isn't by using paths directly between the nodes, but by passing through a different 'optional' **Steiner node**. However, there is no known efficient algorithm to determine where these additional switches should be placed.



Argument of Optimality

The strategy we use here to determine a good lower bound + prove optimality is very similar to TSP, except we use discs of uniform size and iteratively add moats:



Intuitively, can note that the components that are merged are connected by the cheapest edge that is between the components – which is exactly the edge that Kruskal's algorithm would have added.

4 Maximum Flow Problem

- **Intuition**: want to maximize the flow through a series of pipes and a particular source and sink node
- **Input**: directed graph $G = (V, A)$, source node $s \in V$ and sink node $t \in V$, and nonnegative capacity $u(i, j)$ for each arc $(i, j) \in A$
- **Output**: a flow value $f(i, j)$ for each arc $(i, j) \in A$ such that
  1. $0 \leq f(i, j) \leq u(i, j)$ for each $(i, j) \in A$
  2. $\sum_{j:(j,i)\in A} f(j, i) = \sum_{j:(i,j)\in A} f(i, j)$ for each node $i \in V \setminus \{s, t\}$
     **(in = out except at source + sink)
- **Goal**: maximize the net flow into the sink node, i.e.,

$$\text{maximize} \sum_{j:(j,t)\in A} f(j, t) - \sum_{j:(t,j)\in A} f(t, j)$$

Recall that the constraints a flow must satisfy include:

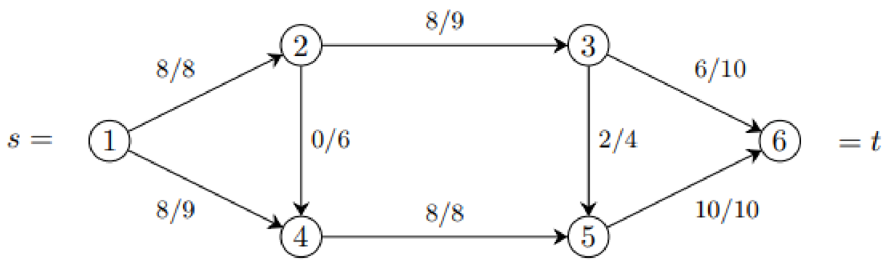
- **nonnegativity**: $f(i, j) \geq 0$ on each arc $(i, j) \in A$ (output req I)
- **capacity**: $f(i, j) \leq c(i, j)$ on each arc $(i, j) \in A$ (output req I)
- **conservation**: output req II

4.1 Ford-Fulkerson Algorithm

At every iteration, this algorithm constructs a new **feasible flow** with larger flow value from a given feasible flow. As we can always start with a feasible flow of 0, this ends up giving us an optimal solution.

Algorithm 5 Ford-Fulkerson

```
start with a feasible flow $f$ (set all $f(i, j) = 0$ for all $i, j \in A$)
construct the residual graph $G_f = (V, A_f)$ for the current flow
while there is a path from $s$ to $t$ in the residual graph do
    let $\delta$ be the minimum residual capacity of the arcs from $s \to t$
    update the flow;
    for each arc $(i, j)$ in the path from $s \to t$ do
        if $(i, j)$ is a foward arc then
            increase the flow of $f(i, j)$ by $\delta$
        else
            decrease the flow of $f(i, j)$ be $\delta$
        end if
    end for
end while
```



Recall that the residual graph of each iteration can be contructed as $G_f = (V, A_f)$ such that V = the same node set V as the input and A_f consists of

1. **forward arcs** (i, j) for arcs (i, j) in the input graph where we can *incease flow* (i.e., where the capacity $u(i, j) > f(i, j)$) with the residual capacity $= u(i, j) - f(i, j)$
2. **backwards arcs** (i, j) for arcs (j, i) in the input graph where we can *decrease flow* (i.e., the flow on the arc $f(i, j) > 0$) with the residual capacity $= f(j, i)$

Argument for Optimality – Upper Bounds

Upper bounds for the maximum flow problem can be created by **partitioning** the node set V into two disjoint sets S and T where the source node $s \in S$ and sink node $t \in T$, we can use the capacities on the arcs from nodes in S to nodes in T as an upper bound. The idea is that these flows must start with $s \in S$ and **leave** S at some point to get into T and eventually $t \in T$.

From the Ford-Fulkerson Algorithm, we can actually partition V by defining

- S = all nodes reachable from s in the final residual graph G_f
- T = $V \setminus S$

**note we can find all reachable nodes by using Djikstra's! (or bfs it doesn't really matter)

4.2 Min Cut Problem

We can define the Min Cut Problem as the problem to find the best upper bound on the value of any flow – aka, the best possible partition of V. We call such a partitioning (S, T) a **cut** and the value of the upper bound it induces the **capacity** of the cut.

- **Input**: a directed graph $G = (V, A)$, source node $s \in V$, sink node $t \in V$, and a nonnegative capacity $u(i, j)$ for each arc $(i, j) \in A$
- **Output**: a cut (S, T) (parition of V such that $s \in S$ and $t \in T$)
- **Goal**: minimize the capacity of the cut (S, T), i.e.,

$$\text{minimize} \sum_{(i,j)\in A, i\in S, j\in T} u(i, j)$$

We can see that this is actually the max flow problem but 'approached from the opposite way': we are reading

$$\text{value of } f \leq \text{capacity of } (S, T)$$

which is exactly the same as

$$\text{capacity of } (S, T) \geq \text{value of } f$$

Thus, the Max Flow and Min Cut problem are a **pair** of problems that address the fundamental problem. For any input to the max flow or min cut problem, **the value of a maximum flow and the capacity of a minimum cut are equal.**

4.3 Integrality Property

Note that if the input to a maximum flow problem has **integer capacities** for every $u(i, j)$, then there exists a maximum flow such that $f(i, j)$ is also an integer for every arc (i, j) – and the Ford Fulkerson algorithm will find such a flow.

**Note that this seems trivial, but is very useful when performing reductions. For example, the Student-Advisor problem can be reduced to a max flow problem because we are sure we end up with integer flows, which is the only solution that makes sense.

4.4 Modelling with the MFP

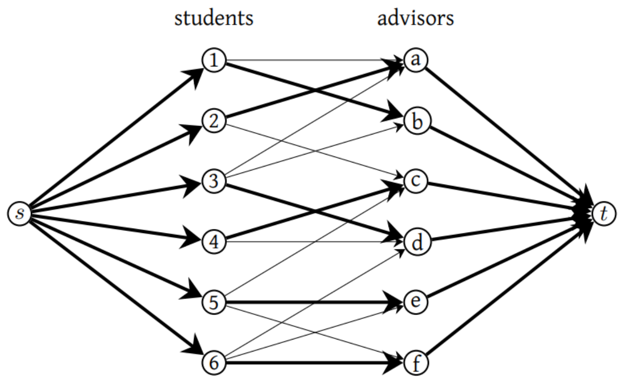
4.4.1 Bipartite Matching

Intuition: we want to match students to advisors; students have a list of potential advisors they'd like to work with, and each advisor should be matched with at most one student

Input: a list of students, advisors, and potential advisors for each student

Let's model the possible student-advisor pairings as a graph. Specifically, this graph is **bipartite** : the node set is able to be partitioned into two sets such that all edges span from one set to the other. In this case, we have student-nodes comprising one set and advisor-nodes comprising the other.

Now, we can add an s and t source and sink node. We'll add edges with capacity 1 from s to the student nodes to ensure each student is matched with only one advisor**, and edges with capacity 1 from the advisor nodes to t to ensure each advisor is matched with at most one student**. We claim that a max flow to this graph gives an assignment between students and advisors that matches as many pairs as possible.



We can make these claims due to the **integrality property of the Ford-Fulkerson algorithm.

4.4.2 Baseball Elimination

Intuition: if there are n teams competing and we are partially though the season, given the current statistics for the teams and the number of games yet to be played, is it possible that some teams are already eliminated?

Input: a summary of the remaining games $g(i, j)$ for the season between each pair of teams i and j for $i, j \in \{1, ..., n\}, i \neq j$ and the standings (number of wins) of the teams so far denoted $w(i)$

Let's focus on team n: we want to determine if team n has already been eliminated from the championship. Without loss of generality, we can assume that if team n has *not* already been eliminated, there is a scenario where n is champion and n wins all it's remaining games. We'll focus on this scenario.

Call W the total number of wins that team n has by the end of the season:

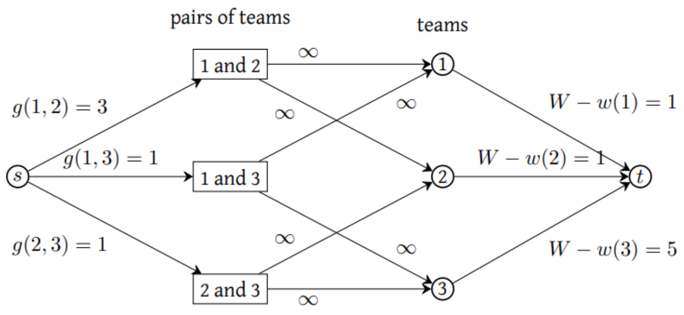
$$W = w(n) + \sum_{j \in \{1,\ldots,n\}, j \neq i} g(i, j)$$

In order for n to be champion, *none of the other teams can have more than W wins.*

We can model this as a max flow problem with a s and t source and sink with 2 layers of nodes:

1. the possible **pairings of teams** left to compete (excluding pairings w/ team n)
2. the **remaining teams** (also excluding team n)

We will set capacities from s to layer 1 equal to the number of games left to be played between two pairings ($g(i, j)$ for $i, j \in \{1, ..., n - 2\}, i \neq j$) and capacities from layer 2 to t equal to the number of games each team is allowed to win ($W - w(i)$ for each team $i \neq n$). We will set capacities of ∞ from layer 1 to 2.



In this case, we can think of the flow as 'wins' – if we achieve a max flow with value equal to the number of games remaining in the season, then we know there exists a way to 'allocate' these wins such that no team ends up with more than W wins. If we achieve a max flow smaller than this value, then we know not all the remaining games can be assigned winners without usurping team n.

- If we get a smaller max flow than the optimal, we can identify where exactly team n's hope for a champion dies by examining the **minimum cut**.

5 Transportation Problem

5.1 The General Transportation Problem

- **Intuition**: want to ship inventory from warehouses to stores such that the demand is met but the cost is minimized
- **Input**: m warehouses, n stores, supply s_i for the warehouses $i \in \{1, ..., m\}$, demand d_j for the stores $j \in \{1, ..., n\}$, and per-unit shipping costs c_{ij} between warehouse i and store j. This input can be written in the form of:

		j	1	2	3
		d_j	3	5	7
i	s_i	c_{ij}			
1	7		8	3	2
2	8		1	4	5
3	3		8	6	7

- **Output**: x_{ij} describing the number of units shipped from warehouse i to store j satisfying
  - $\sum_{j=1}^{n} x_{ij} \leq s_i$ for each $i = 1, 2, ...m$ ensuring that a warehouse doesn't ship out more than its supply
  - $\sum_{i=1}^{m} x_{ij} \geq d_j$ for each $j = 1, 2, ...n$ ensuring that each store's demand is met
- **Goal**: minimize the total shipping cost

$$\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

Note that the only potentially optimal solution is a solution where demands are met exactly – no excess. First, we will create a **balanced input** , where total demand is equal to total supply.

If the total demand is less than the total supply, we balance the inputs be adding a **'dummy store'** with demand

$$d_{n+1} = \sum_{i=1}^{m} s_i - \sum_{j=1}^{n} d_j$$

**This only works if the total supply is greater than the total demand, which is always going to be the case in this course.

Once we have a balanaced input, we can continue to modify the cost-assignment matrix until we have a 0 in at least every row and every column. At this point, we can go ahead and construct a maximum flow graph: as per usual, we'll have a start and end s and t with multiple 'layers' of nodes:

1. **warehouses**: the edges from s to the warehouses will have capacity of s_i
2. **stores**: the edges from the stores to t will have capacity d_j

The edges between the warehouses and the stores will have capacity ∞, but we will only draw edges between a warehouse and a store if the per unit shipping cost in the modified cost matrix is equal to 0.

More specifically, if there doesn't exist a 0 cost solution right off the bat, we can examine the minimum cut (S, T) from the residual graph and modify the costs by:

- Decrease c_{ij} for every warehouse $i \in S$ and every store $j = 1, 2, ..., n$
- To ensure we have non-negative shipping costs, now increase c_{ij} for every store $j \in S$ and warehouse $i = 1, 2, ..., m$

5.2 Assignment Problem

The **Assignment Problem** is a special case of the Transportation Problem

6 Linear Programming & the Simplex Method

7 Branch and Bound

Linear Programming problems have the Simplex Method, *Integer* Linear Programming problems have **Branch and Bound**: this method iteratively splits the feasible region into smaller sections if it's possible that the smaller section contains a better solution than what we have so far.

Recall any solution to the ILP is a feasible solution to the LP-relaxation. The (integer floored) optimal solution to the LP-relaxation is in turn an *upper bound* on the objective value of an optimal ILP.

- The idea is to first split the feasible region into two parts and solve the ILP for those parts separately. We know the feasible solution must be in one of those two sections, so solving the both and taking the best solution is equivalent to solving the original problem itself.

Consider:

$$\begin{aligned} \text{maximize: } & 4x + 5y \\ \text{subject to: } & x + 9y \leq 55 \\ & 9x + y \leq 48 \\ & 2x + 20y \leq 121 \\ & x, y \geq 0, \text{ integer} \end{aligned}$$

The optimal solution to the LP-relaxation is $(x^*, y^*) \approx (4.71348, 5.57865)$ with objective value $z^* \approx 46.74719$. Flooring that, we have an upper bound on the ILP solution of 46. From this objective value, we can determine how to split the feasible region: $x \geq 5$ and $x \leq 4$**

**this split ensures that we get *new information* when solving – as (x^*, y^*) is not in either.

$x \geq 5$ – Solving the LP-relaxation with this additional constraint gives

$$(x^*, y^*) = (3, 5), z^* = 35$$

Because we solved the LP-relaxation, this is an upper bound on the objective value in this region. This upper bound happens to be integer and thus feasible as an ILP – thus, we've found the optimal ILP solution for this region and don't need to subdivide further.

$x \leq 4$ – Solving the LP-relaxation with this additional constraint gives

$$(x^*, y^*) = (4, 5.65), z^* = 44.25$$

Because this is an LP-relaxation, we round down the objective value to get an upper bound of 44 in this region. At this point, we don't have the optimal ILP solution yet so we further split this region.

We use the second variable y and its fractional value 5.65 to subdivide:

$y \geq 6$: $(x^*, y^*) = (0.5, 6), z^* = 32$

> This objective value is lower than our previous max ILP solution of 35, so we can disregard this region (note we didn't have to solve an ILP; an LP was enough to reach this conclusion)

$y \leq 5$: $(x^*, y^*) = (4, 5), z^* = 41$

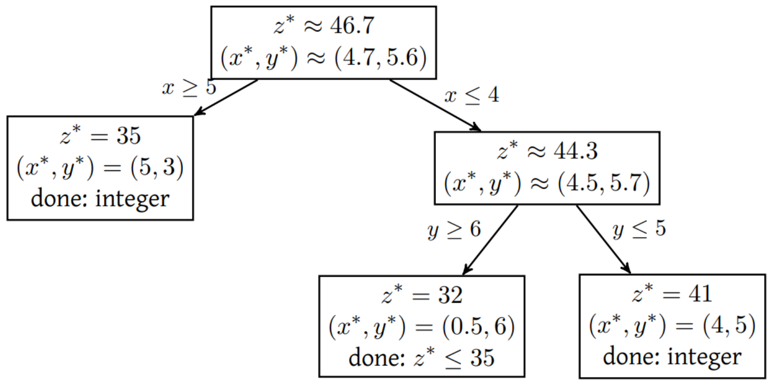
> This upper bound also happens to be integer, so we've found the optimal ILP solution for this region. This is highest objective value, and because we've implicitly considered all feasible integer solutions, we can conclude this is an optimal solution to the ILP.

Note it's possible that a subdivided region is actually **infeasible**, with no solutions to the LP-relaxation. In these cases, we can disregard that region.

Algorithm 6 Branch and Bound

```
$\mathcal{P} \leftarrow \{F\}$
Create $\alpha$ to store the best objective value so far
while $\mathcal{P}$ is not empty do
    Take a set $S$ from $\mathcal{P}$ and solve the corresponding LP with feasible region $S$
    if the LP is infeasible or objective value is less than $\alpha$ then
        We can disregard $S$
    else the objective value is larger than $\alpha$
        if The objective value is integer then
            $\alpha \leftarrow$ objective value
        else continue subdividing
            Add $S \cup \{x_i \leq \lfloor x_i \rfloor\}$ and $S \cup \{x_i \geq \lceil x_i \rceil\}$ to $\mathcal{P}$
        end if
    end if
end while
Return $\alpha$
```

Keep track of this process is through a **branch and bound tree**. For example:



8 Linear Programming Duality