Assignment #3
3 Jugs Puzzle

Pei-yih Ting

http://www.mathsisfun.com/games/jugs-puzzle.html

http://www.cut-the-knot.org/ctk/Water.shtml

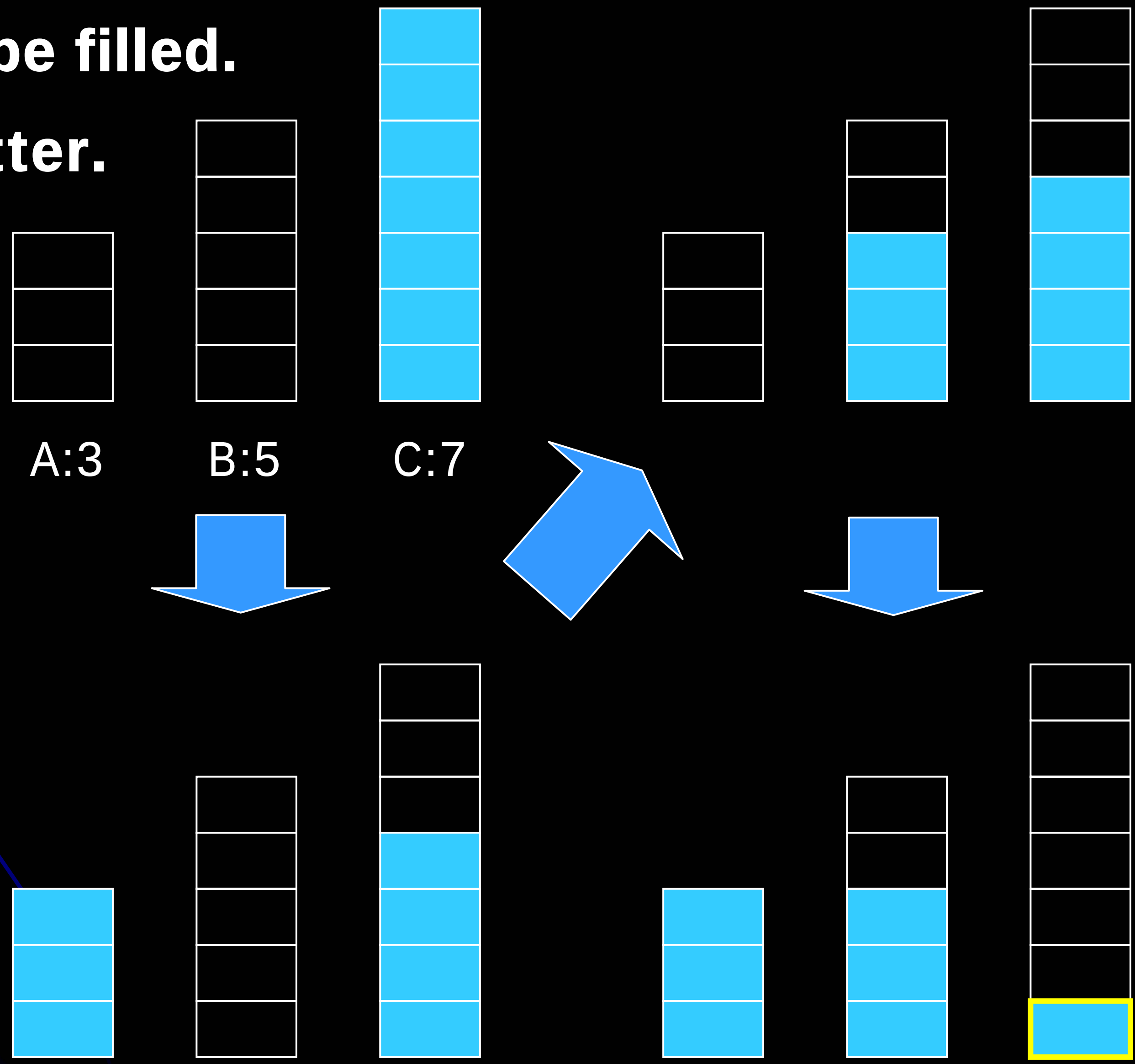
Problem

- Siméon Denis Poisson
  - *Two friends who have an eight-quart jug of water wish to share it evenly. They also have two empty jars, one holding five quarts, the other three. How can they each measure exactly four quarts of water?*
- Another story
  - *Three men robbed a gentleman of a vase, containing 24 ounces of balsam. Whilst running away they met a glass seller, of whom they purchased three vessels. On reaching a place of safety they wished to divide the booty, but found that their vessels could hold 5, 11, and 13 ounces respectively. How could they divide the balsam into equal portions?*

A Simple Example

Let the third jar be filled.

Our target is 1 litter.



0: 0 0 7
1: 3 0 4
2: 0 3 4
3: 3 3 **1**

Configurations & Decisions

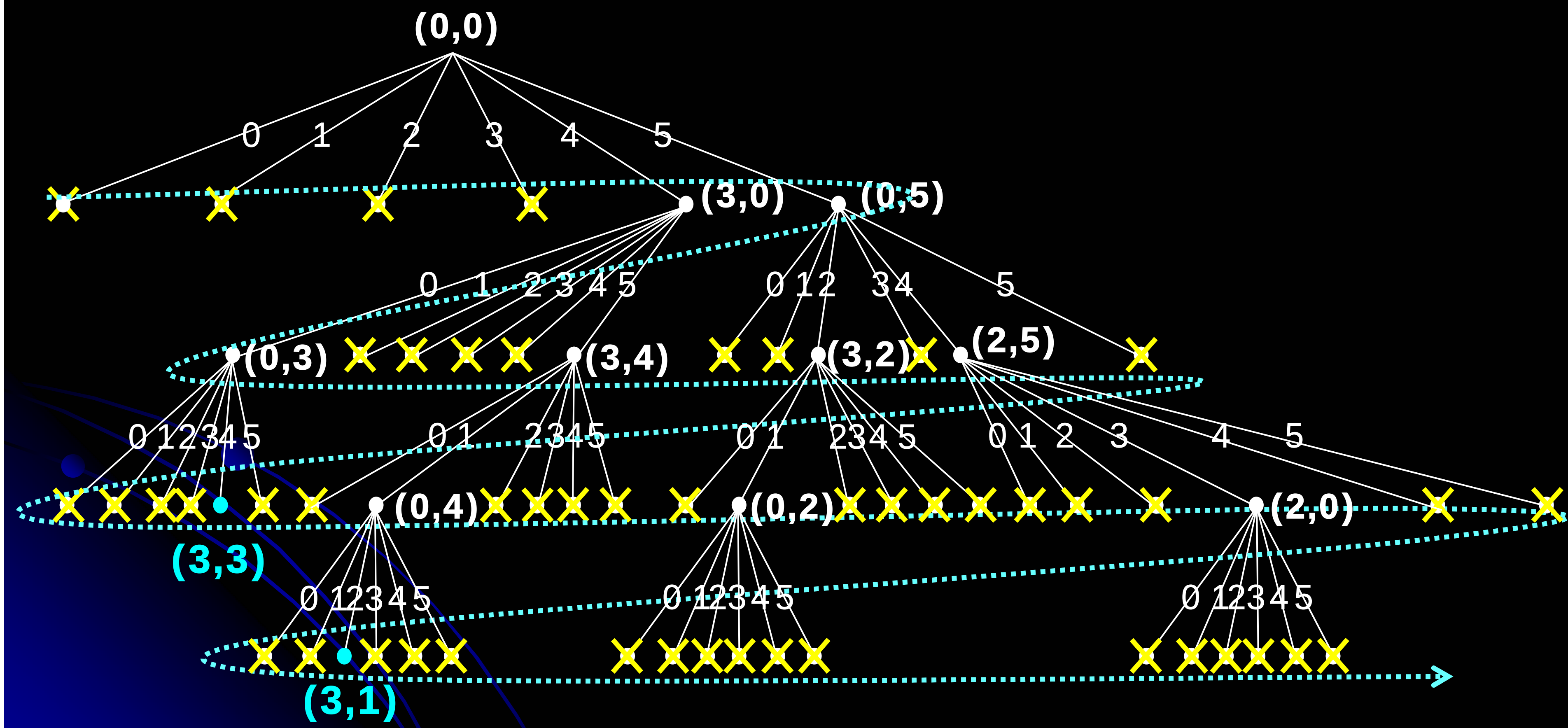
- You can represent the configuration of the puzzle at each instant as a 3-tuple, e.g. (3, 0, 4), or simply as a pair (3, 0)
- At each instant, the player has the following **six** possible decisions to choose:

  A ==> B B ==> A C ==> A
  A ==> C B ==> C C ==> B

- The pouring of water at each step stops when either 1. **the target jar is full** or
  2. **the source jar is empty**
  because the jars do not have any mark on them.

Breadth-First Search

Exhaustive search over all possible decisions

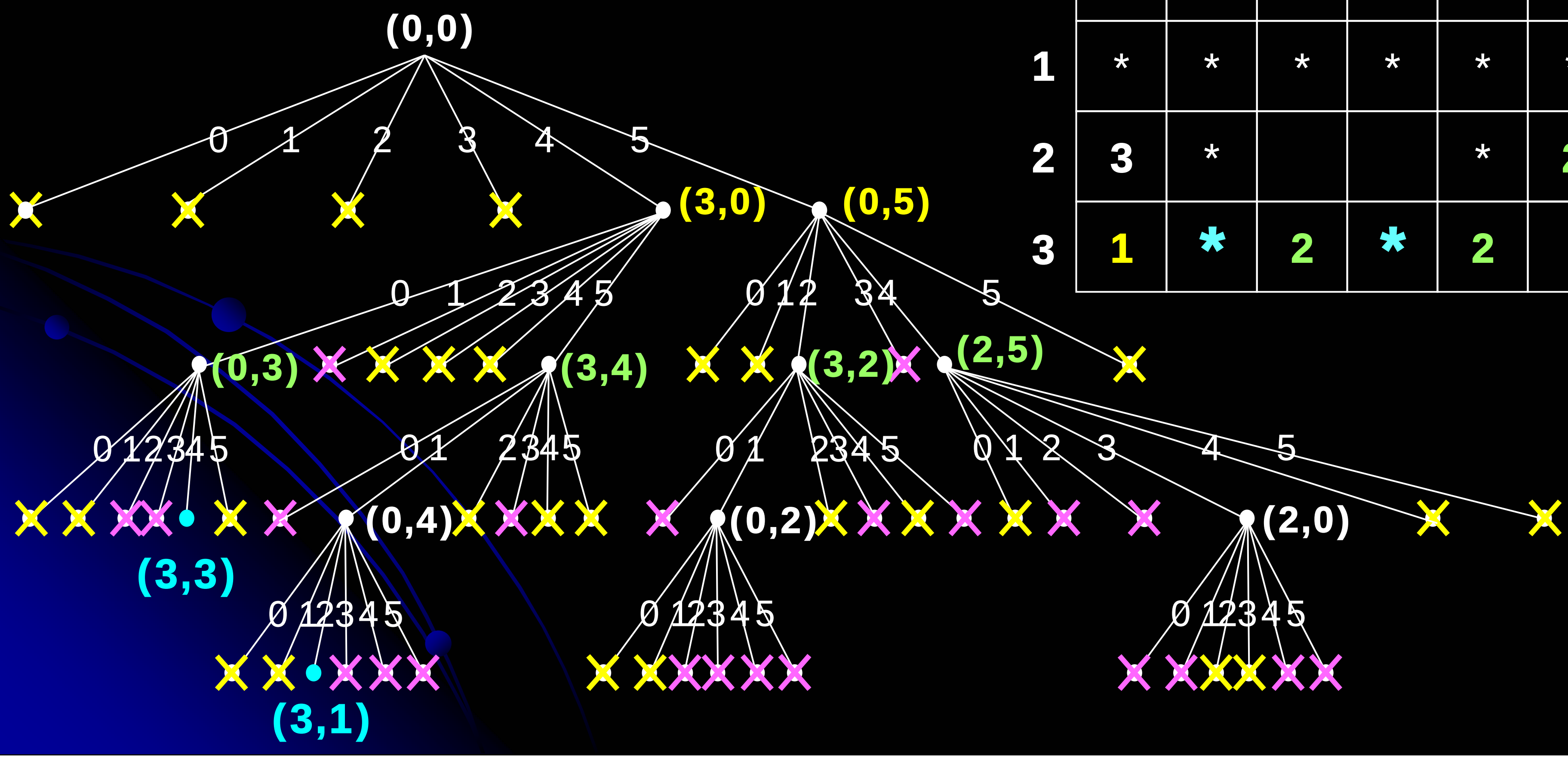


0: A==>B
1: A==>C
2: B==>A
3: B==>C
4: C==>A
5: C==>B

BFS Implementation

- Mark all possible goal configurations
- Keep "# of steps from the start configuration" in each cell
- Skip visited configurations

	0	1	2	3	4	5
0	0	*	3	2	3	1
1	*	*	*	*	*	*
2	3	*			*	2
3	1	*	2	*	2	



BFS Implementation (cont'd)

	0	1	2	3	4	5
0	0	*				
1	*	*	*	*	*	*
2		*			*	
3		*		*		

	0	1	2	3	4	5
0	0	*				1
1	*	*	*	*	*	*
2		*			*	
3	1	*		*		

	0	1	2	3	4	5
0	0	*		2		1
1	*	*	*	*	*	*
2		*			*	2
3	1	*	2	*	2	

	0	1	2	3	4	5
0	0	*		2		1
1	*	*	*	*	*	*
2		*			*	2
3	1	*	2	*	2	

	0	1	2	3	4	5
0	0	*	3	2	3	1
1	*	*	*	*	*	*
2	3	*			*	2
3	1	*	2	*	2	

	0	1	2	3	4	5
0	0	*	3	2	3	1
1	*	*	*	*	*	*
2	3	*			*	2
3	1	*	2	*	2	

- Starting from 0, label direct followers as 1
- Find all 1's, label direct followers as 2
- Find all 2's, label direct followers as 3
- … until no more direct followers

Inefficient for jugs with large capacity

BFS Implementation (cont'd)

- Instead of finding the next configuration globally in the array, let's chain all configurations scheduled to be considered when we search with the BFS algorithm. You will find the following configurations sequentially: (0,0), (3,0),(0,5), (0,3),(3,4),(3,2),(2,5), **(3,3),** (0,4),(0,2),(2,0), **(3,1)** as you consider the six possible decisions.
- Let's extend our array to keep this sequence. (formally this is a variation of a ***queue*** data structure.)

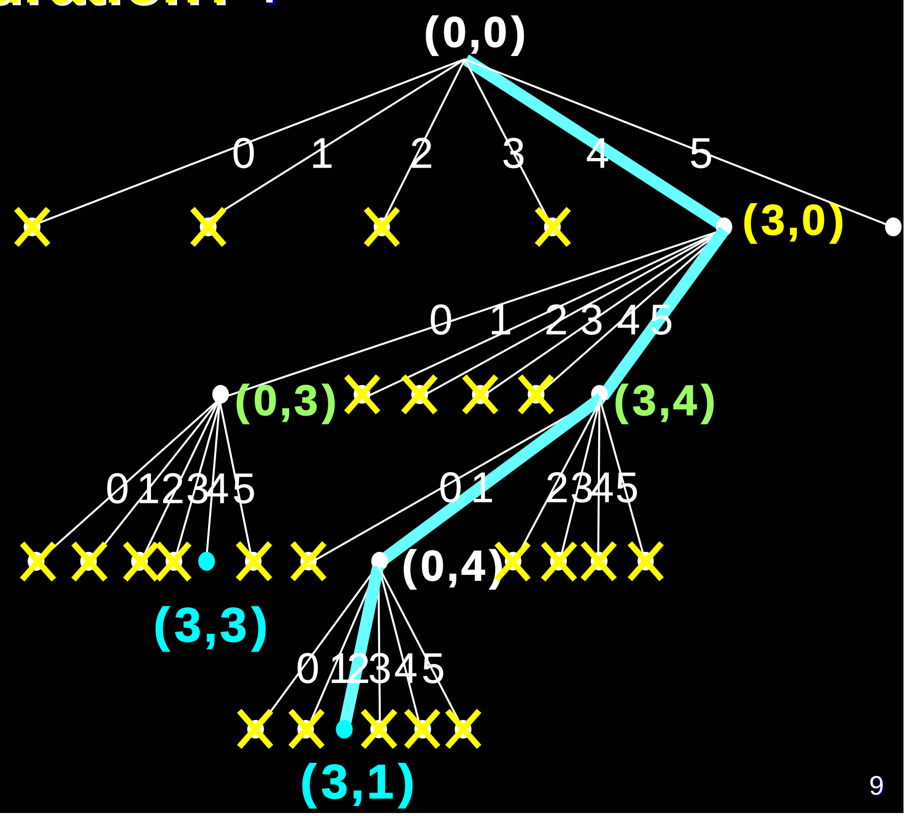
	0	**1**	**2**	**3**	**4**	**5**
0	**0 / (3,0)**	*	**3 / (2,0)**	**2 / (3,4)**	**3 / (0,2)**	**1 / (0,3)**
1	*	*	*	*	*	*
2	**3 / (3,1)**	*			*	**2 / (3,3)**
3	**1 / (0,5)**	*** / (-,-)**	**2 / (2,5)**	*** / (0,4)**	**2 / (3,2)**	

- The next configuration can now be explored following the link.
- You can save some memory by encoding (r,s) as r*6+s

BFS Implementation (cont'd)

- The algorithm can stop the first time it finds a goal configuration.

- The final thing to work on is "**how to print the steps once we find a target configuration?**".
- As the algorithm proceeds from the start configuration to the goal configuration. It has to keep track of the parent configuration with a **backward link** for each node. e.g. (3,1) ➔ (0,4) ➔ (3,4) ➔ (3,0) ➔ (0,0)



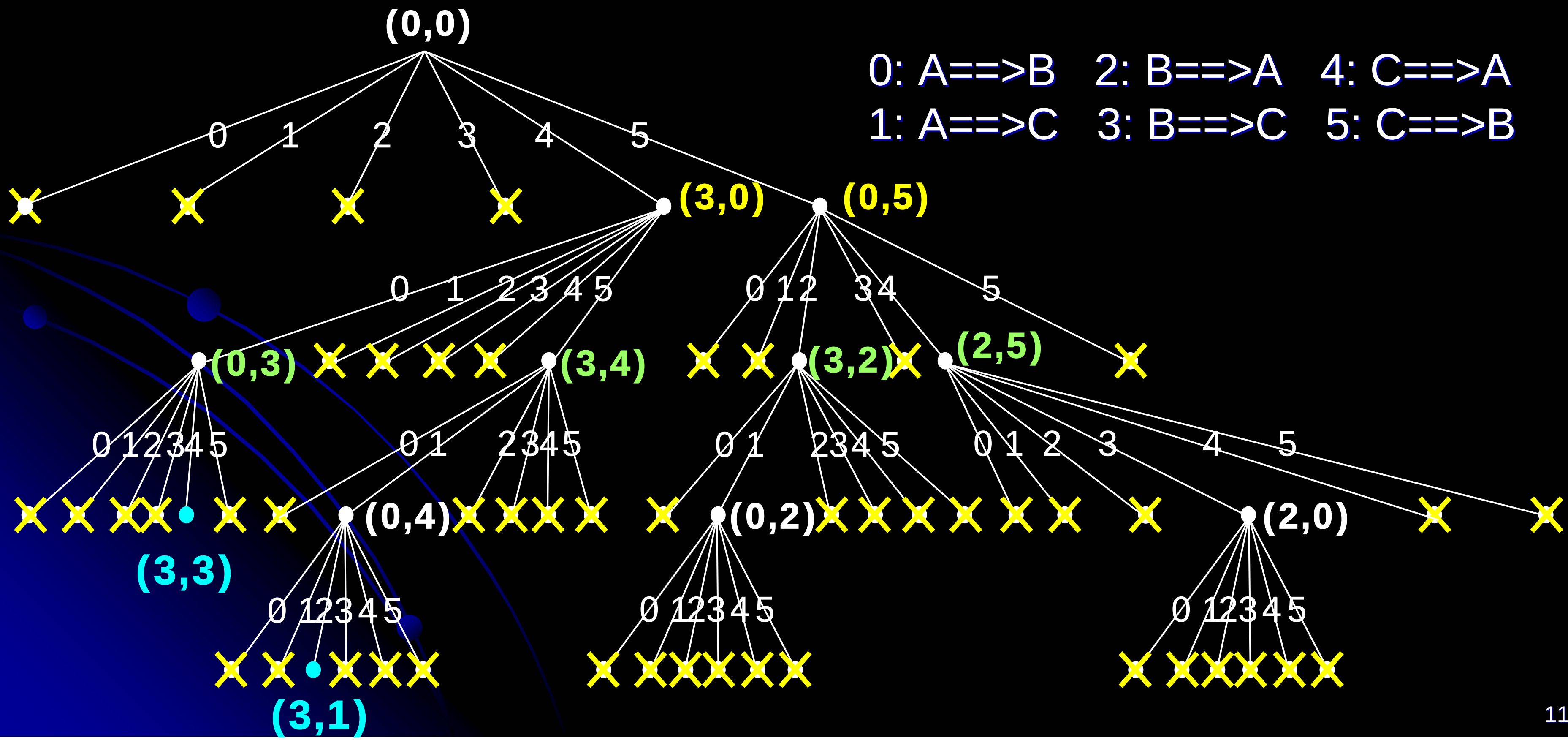
BFS Implementation (cont'd)

- Each configuration has a **unique** parent configuration.
- Extend further our two-dim array implementation to keep the extra parent information as we visit each configuration the first time.
- e.g. (3,1) → (0,4) → (3,4) → (3,0) → (0,0)

	0	1	2	3	4	5
0	0 / (3,0)	*	3 / (2,0) (3,2)	2 / (3,4) (3,0)	3 / (0,2) (3,4)	1 / (0,3) (0,0)
1	*	*	*	*	*	*
2	3 / (3,1) (2,5)	*			*	2 / (3,3) (0,5)
3	1 / (0,5) (0,0)	* / (-,-) (0,4)	2 / (2,5) (0,5)	* / (0,4) (0,3)	2 / (3,2) (3,0)	

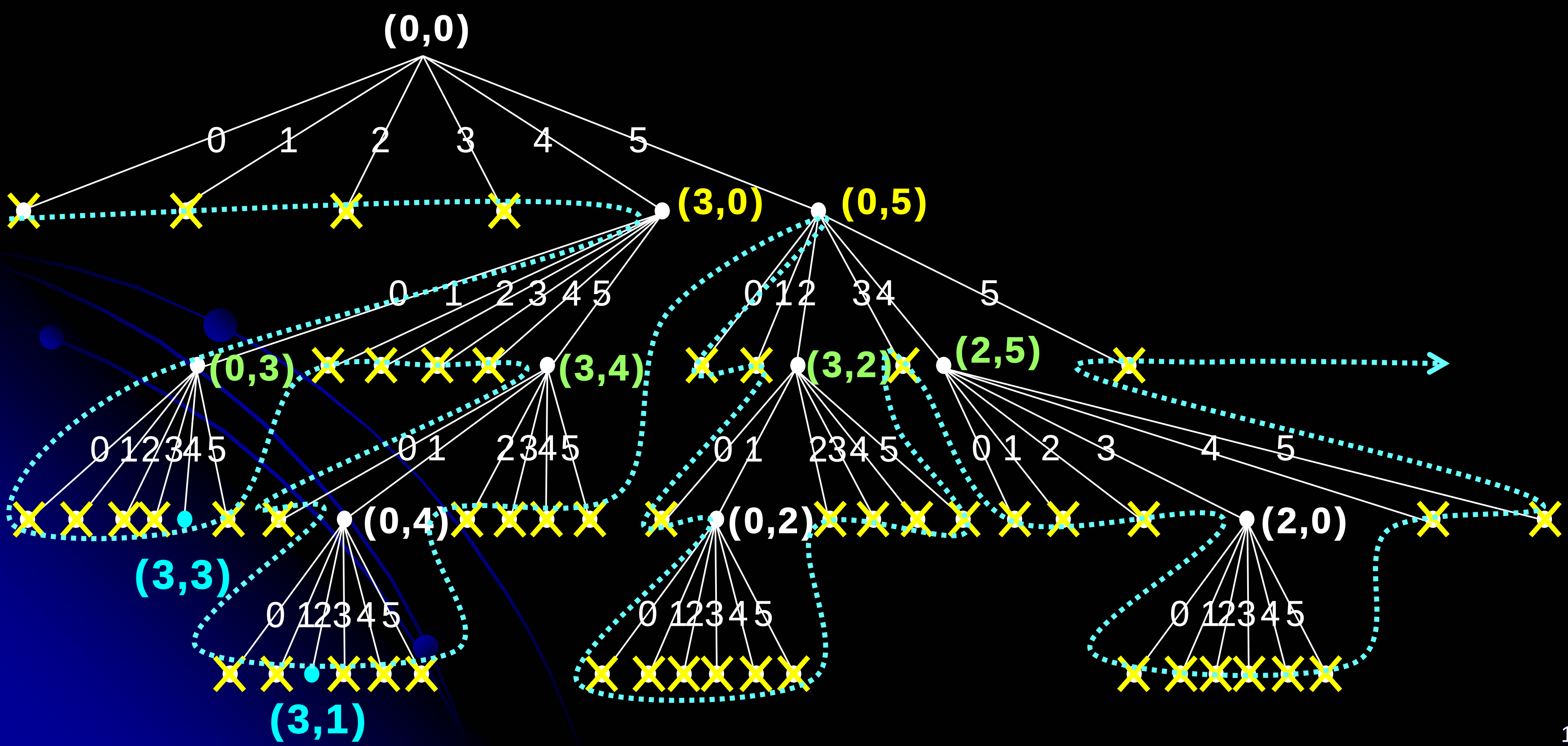
- You can save some memory by encoding (r,s) as r*6+s

	0	1	2	3	4	5
0	0 / (3,0)	*	3 / (2,0) (3,2)	2 / (3,4) (3,0)	3 / (0,2) (3,4)	1 / (0,3) (0,0)
1	*	*	*	*	*	*
2	3 / (3,1) (2,5)	*			*	2 / (3,3) (0,5)
3	1 / (0,5) (0,0)	* (0,4)	2 / (2,5) (0,5)	* / (0,4) (0,3)	2 / (3,2) (3,0)	



Depth-First Search

- Again, an exhaustive search over all possible decisions
- Visit all nodes in a different order from BFS: go as deepest as possible until no descendent exists, then the siblings.



DFS Implementation

- Because each configuration is visited at most once (some configurations might not be visited), we can estimate the **upper bound of the depth** of the search tree.
- For our previous example, **depth <= (3+1)*(5+1) - #goals**.
- You might get a more accurate estimate by considering the "barycentric coordinates" described in
  http://www.cut-the-knot.org/ctk/Water.shtml
- **Iterative** implementation:

  remembering the current decisions in an **array** (see next slide)

  1. Generate all possible **decisions** (next slide)
  2. Calculate the corresponding **configuration** of jugs and verify if **a.** it is valid, **b.** it has been visited, **c.** it is one of the goals
- **Recursive** implementation:

  remembering the remaining decisions in the **system stack**

Generating All Possible Decisions with 2-layer for loop

```
void next(int decisions[], int depth) {
    int i;
    for (i=depth-1; i>0; i--)
        if (decisions[i]<5)
            { decisions[i]++; return; }
        else
            decisions[i] = 0;
    decisions[0]++;
}

int a=3, b=5, depth=(a+1)*(b+1)-11;
int decisions[MAX_DEPTH];
for (i=0; i<depth; i++) decisions[i] = 0;
for (; decisions[0]<6; next(decisions, depth))
    printArray(decisions, depth);
```

decisions

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	0	2
0	0	0	0	0	3
0	0	0	0	0	4
0	0	0	0	0	5
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	0	1	2
0	0	0	0	1	3
0	0	0	0	1	4
0	0	0	0	1	5
0	0	0	0	2	0
0	0	0	0	2	1
0	0	0	0	2	2
0	0	0	0	2	3

…