Branch and Bound

ALGORITHMICS

Vicente García Díaz
garciavicente@uniovi.es


Contents

BRANCH AND BOUND

Basic concepts

Branch and Bound

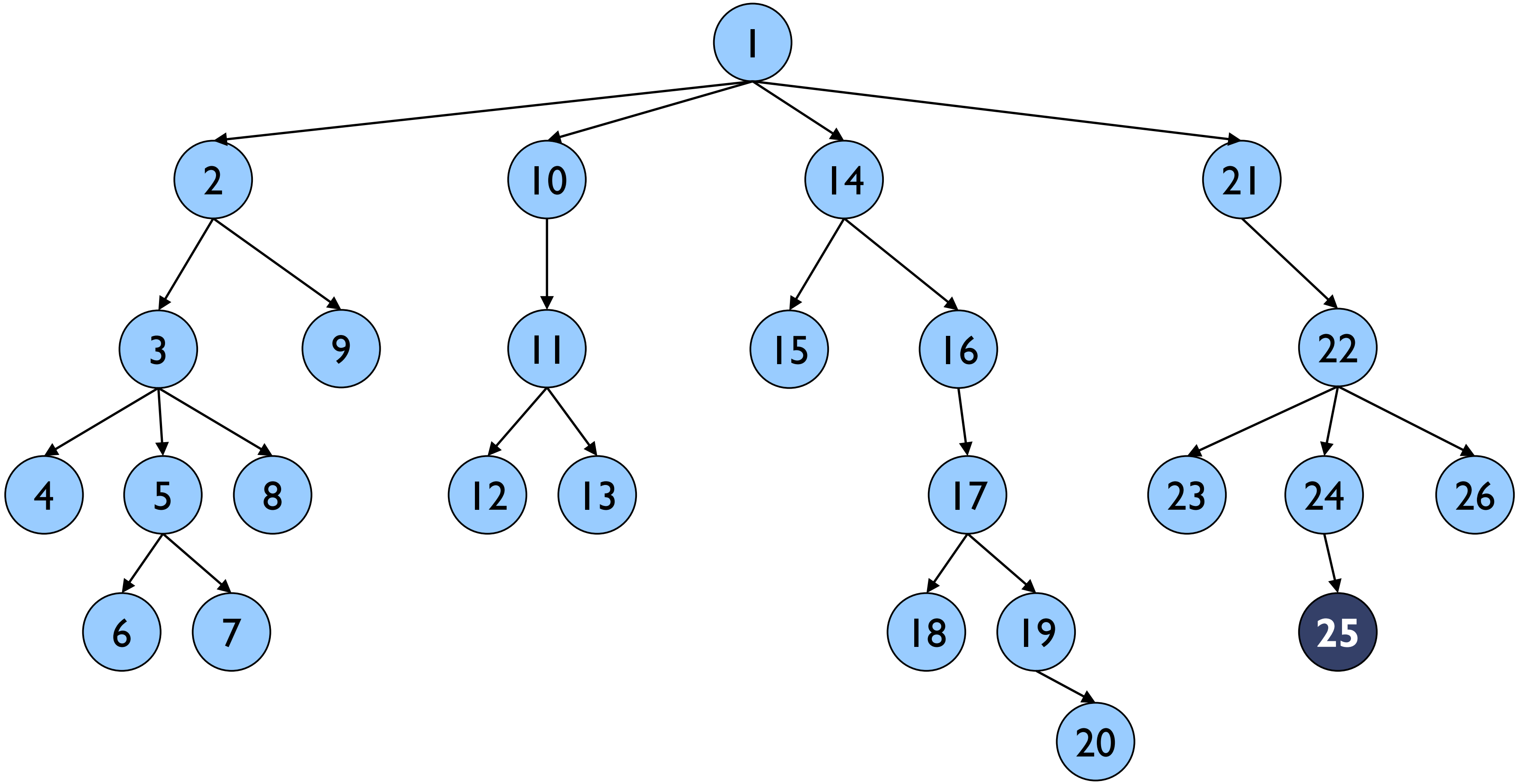
- This technique attempts to explore an implicit tree just like **backtracking**
- **Nodes** represent the state of the problem
- **Edges** represent valid changes
- To solve the problem, we must find a node or a path in the associated graph

Breadth-first search

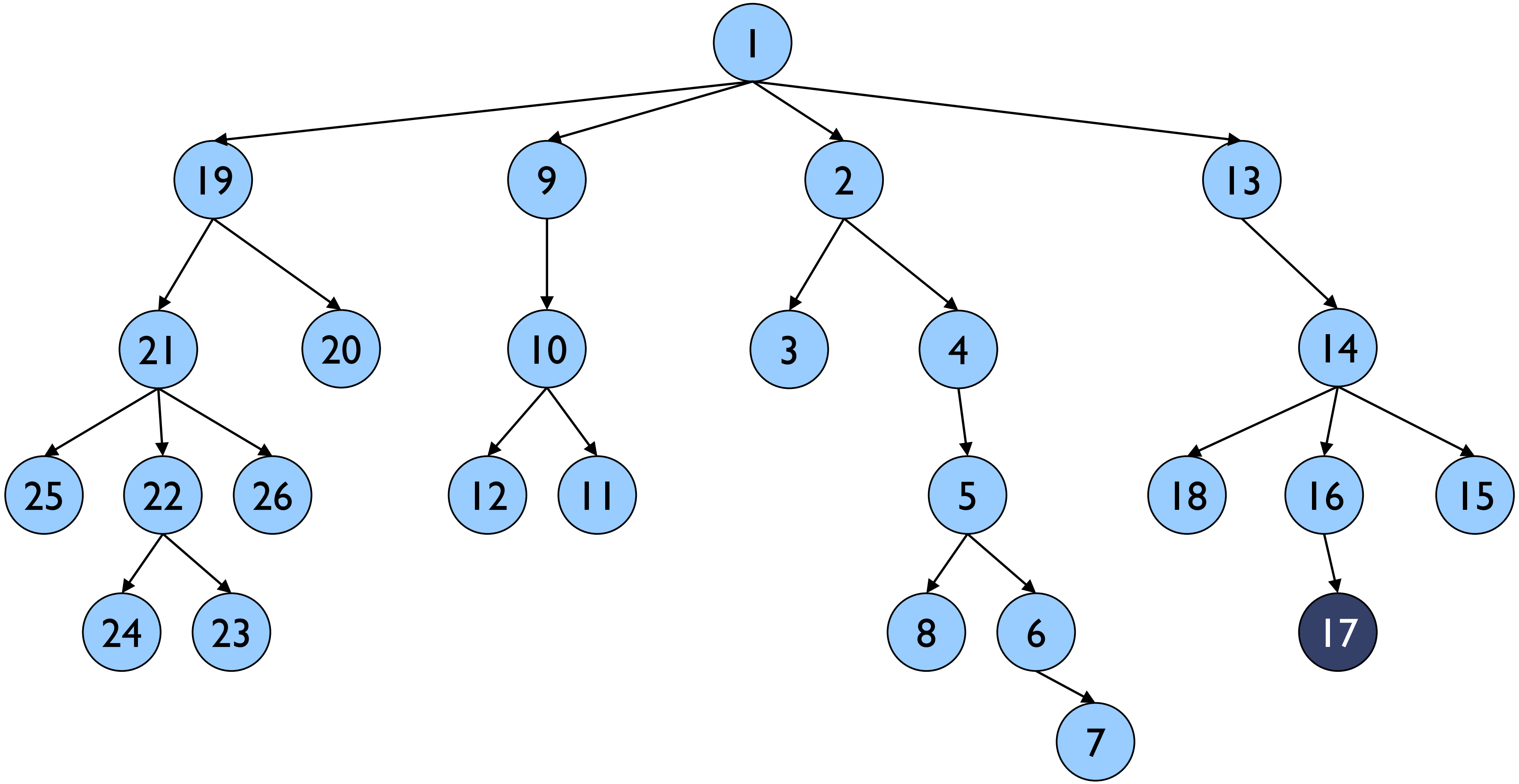
- With **backtracking**, the tree of states is developed in **depth-first search**

- With **branch and bound**, all children nodes are generated before moving to the next state → It performs a **breadth-first search**

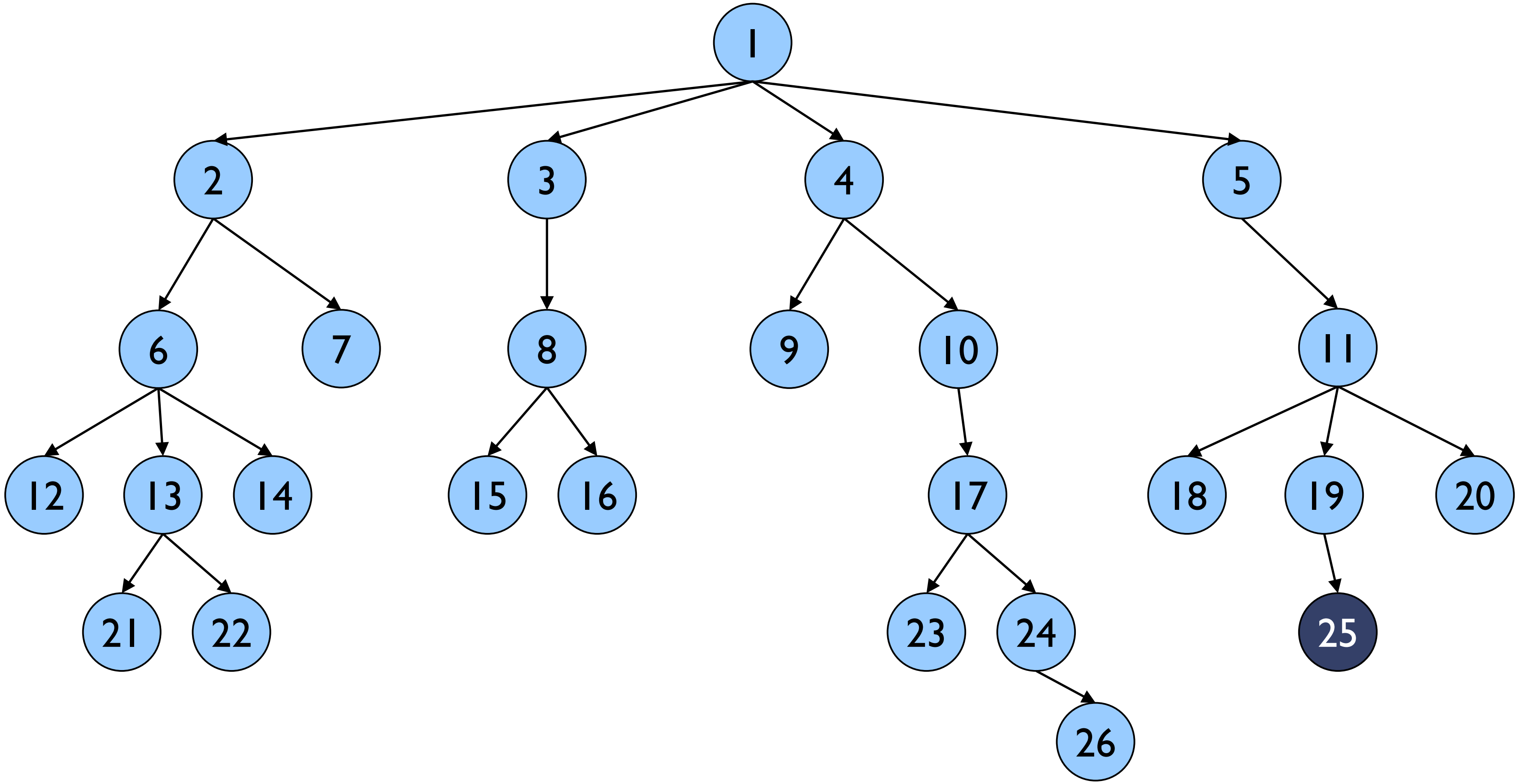
Depth-first search (backtracking)



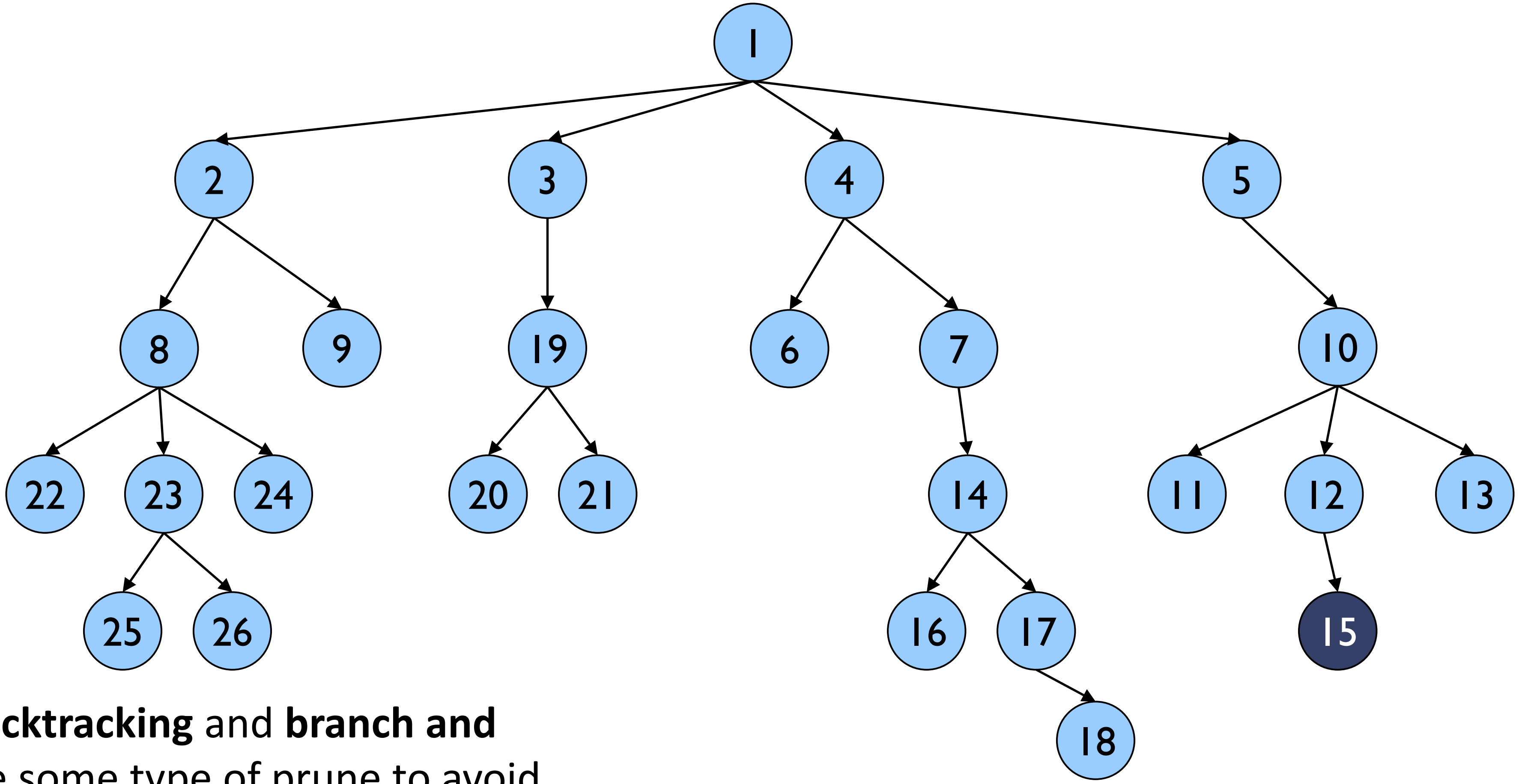
Depth-first search (backtracking with some domain knowledge)



Breadth-first search (branch and bound)



Breadth-first search (branch and bound with heuristic)



Both of them **backtracking** and **branch and bound** could use some type of prune to avoid developing bad known nodes

Advantages of breadth-first search

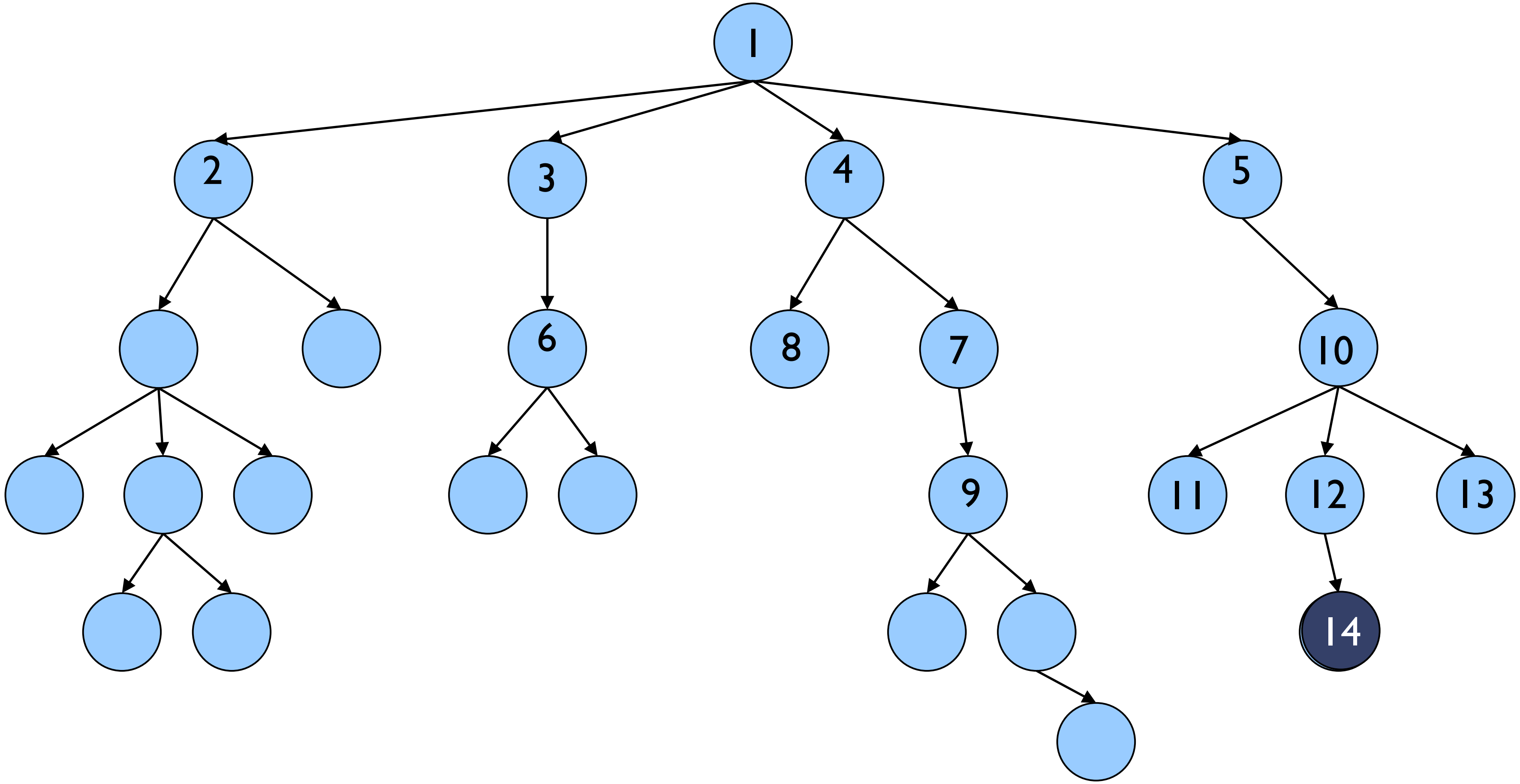
- If we only search for the first solution, and there are solution states near the root

- With backtracking (depth-first) it is possible to enter a branch with many nodes that do not reach any solution or even that has no end (infinite)
  - This, with breadth-first, never happens

Branching Functions

- The branching algorithms consist in developing nodes in the order indicated by a branching heuristic function
- In every moment, we take as the node to be developed, the best that can be at any level of the tree
- The branching algorithms make sense when the idea is to get the first solution and exit
- If you must develop the whole tree backtracking can be better

Branching Functions (II)



Branching data structure

- The general scheme is the same as seen for exploration with breadth-first search, changing the FIFO queue by a **priority queue**, to store the "active" states to be developed

- Priority queue (heap)

- The heap, by structure (is an array), is limited to a maximum size that can only have up to a maximum pending to be developed live states

Consequences of the fixed size of the heap

- We can only store a limited number of states

- We are doing an uncontrolled pruning

- We could not store some states with a bad heuristic
- We may not find the solution even if it exists

Pruning or bounding

- We could add a **pruning heuristic function** to any of the already seen algorithms (depth-first, breadth-first, …)

- The pruning function prevents the development of certain states that do nothing to find the solution to the problem:
  - Do not lead to any solution, or
  - Do not lead to better solutions, or
  - Are repeated developments

Pruning heuristic

- We assume that there is a **boolean** function that tells whether the children of a state can be pruned or not

```
if (node.getHeuristicValue() < pruneLimit)
   ds.insert(node);
else
   //do not store in the priority queue
```

- We can use this with depth-first or breadth-first algorithms

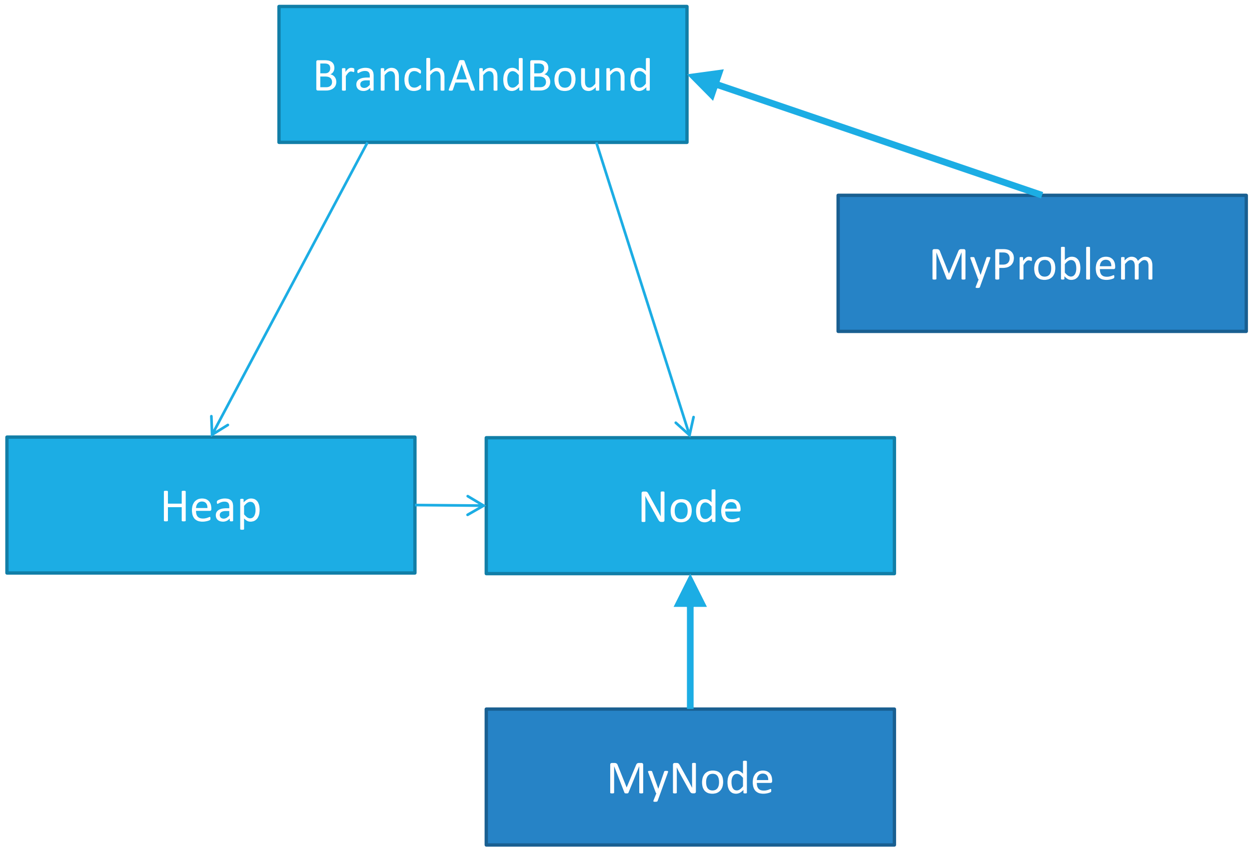
General scheme

```
protected Heap ds; //nodes to be explored (not used nodes)
protected Node bestNode; //to save the final node of the best solution
protected Node rootNode; //initial node

public void branchAndBound(Node rootNode) {
      ds.insert(rootNode); //first node to be explored
      pruneLimit = rootNode.initialValuePruneLimit();
      while (!ds.empty() && ds.estimateBest() < pruneLimit) {
               Node node = ds.extractBestNode();

               ArrayList<Node> children = node.expand();
               for (Node child : children)
                        if (child.isSolution()) {
                                 int cost = child.getHeuristicValue();
                                 if (cost < pruneLimit) {
                                          pruneLimit = cost;
                                          bestNode = child;
                                 }
                        }
                        else
                                 if (child.getHeuristicValue() < pruneLimit) {
                                          ds.insert(child);
                                 }
      } //while
}
```

Relationship among classes



BRANCH AND BOUND

Examples of use

The problem of assigning tasks to agents

- We must assign `j` tasks to `i` agents, so that each agent performs only a task
- We have a matrix of costs, to know the cost of performing a task `j` by an agent `i`
- Goal: Minimizing the sum of the costs for executing the `n` tasks

j (tasks)

i (agents)

	1	**2**	**…**	**n**
a	11	12	…	40
b	14	15	…	22
…				
n	17	14	…	28

Branching heuristic

- **Calculation:**
  - The sum of the already assigned until that state, plus the best case (minimum of each column) of what remains to be assigned
  - At each level we determine the assignation of another agent
- **Example:**
  - State a→1, it carries a cost of **11**+14+13+22=60
- **Use of the heuristic:**
  - Explore the state of the tree that presents a lower heuristic value

	1	2	3	4
a	11	~~12~~	~~18~~	~~40~~
b	~~14~~	15	13	22
c	~~11~~	17	19	23
d	~~17~~	14	20	28

Branching heuristic (II)

- **Calculation:**
  - The sum of the already assigned until that state, plus the best case (minimum of each column) of what remains to be assigned
  - At each level we determine the assignation of another agent
- **Example:**
  - State a→2 y b→1, it carries a cost of **12**+**14**+19+23=68
- **Use of the heuristic:**
  - Explore the state of the tree that presents a lower heuristic value

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Initial value prune limit

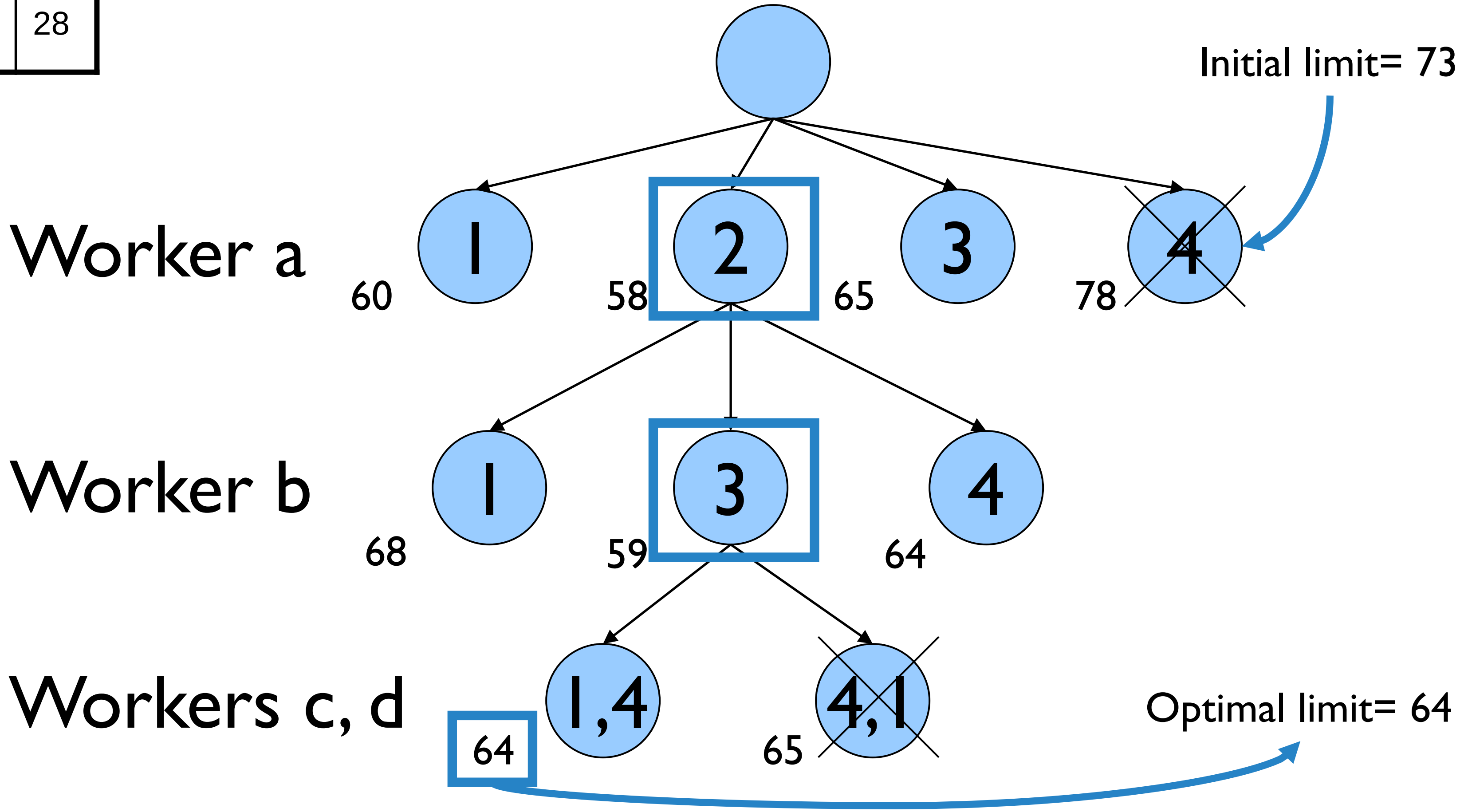
- **Calculation:**
  - Initially, the limit of pruning is the smallest of the sums of the two diagonals of the matrix of costs
- **Use:**
  - We prune every state that has a calculated value for the branching heuristic >= the limit of pruning
- **Change:**
  - The limit changes when we find a better solution

	1	2	3	4
a	**11**	12	18	**40**
b	14	**15**	**13**	22
c	11	**17**	**19**	23
d	**17**	14	20	**28**

- Initial value limit: $\boxed{\min(73,87)=73}$

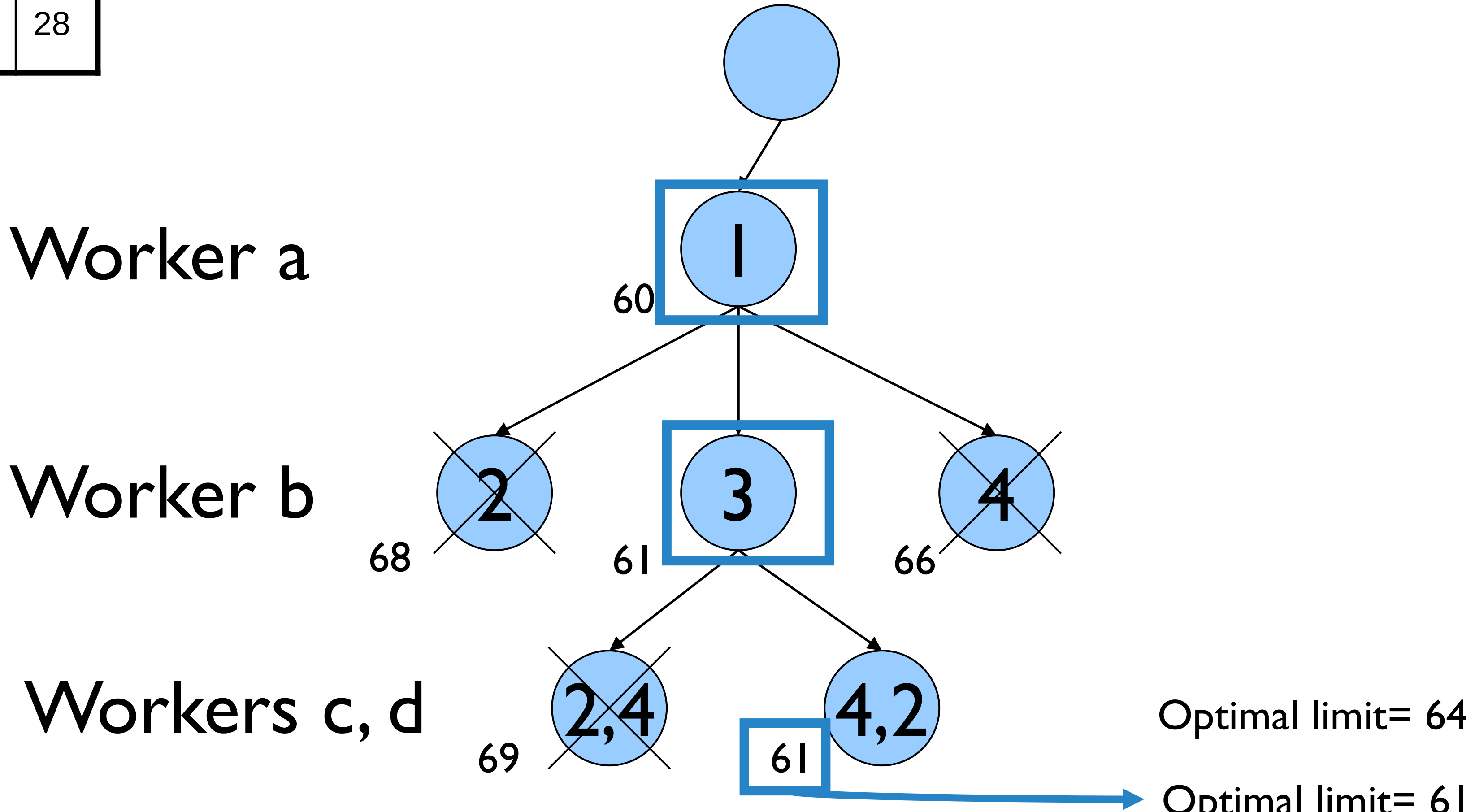
	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Solution



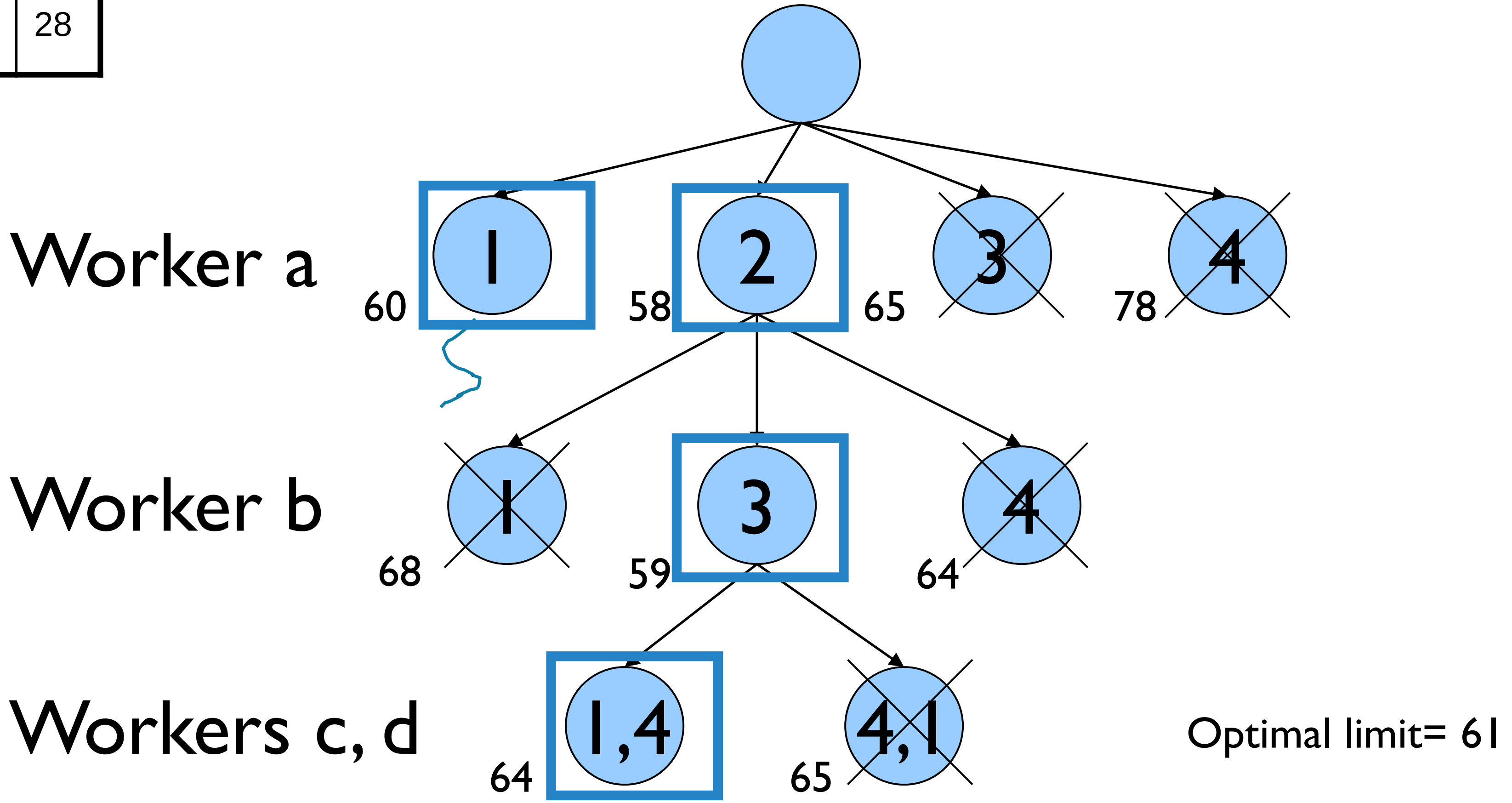
	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Solution (II)



	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Solution (III)

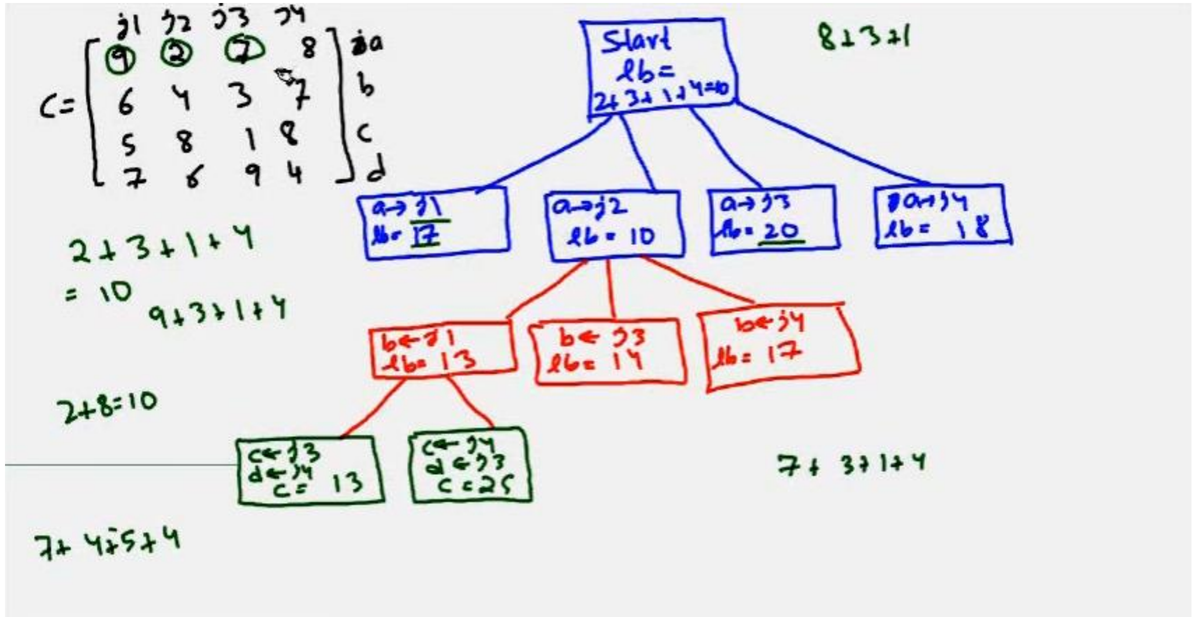


Solution (IV)

- Optimal solution:
  - (a, 1), (b, 3), (c, 4), (d, 2) → Total cost = 61

- We only have developed 4 final states (6 if we count the two 2 diagonals to initialize the pruning threshold), instead of the 4!=24 that the backtracking development implies

- This algorithm is much more efficient in finding optimal solutions



More at:

- Assignment Problem using Branch and Bound https://www.youtube.com/watch?v=BV2MIZna6PI

The problem of the puzzle

- This problem is developed on a board with 16 (n^2) positions, where there are 15 (n^2-1) pieces placed and there is an empty slot

- Pieces cannot be lifted from the board, so their movement is only possible through slides on it

14	2	3	12
9		4	8
13	10	11	7
5	1	15	6

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal

- The objective of the game is, from an initial state (disordered pieces), get a goal state, through a series of legal moves

- Valid movements are those in which one piece, adjacent to a free position, is moved to that position

- You can see these allowed movements as displacements from the empty cell to one of its four neighbors

Strategy

- The full state space is composed of 16! states
  - It would be computationally **unacceptable** the cost to create and explore the tree

- In fact, only half of these states are reachable from a given initial state, but still the number of states is still very large

- We **must** be able to determine from a certain state, if it can reach a goal state or not

Branching heuristic

- Example heuristics to develop states:
  - Smallest number of pieces placed on a wrong place

  - Smallest Manhattan distance of all the pieces regarding their final position
    - $\sum_{i=1}^{16} space(i)$, where `space(i)` is the number of movements (distance) needed to put the piece `i` in the position `i`

  - …

Pruning (bounding) heuristic

- Numbering boxes from 1 to 16, we define:
  - **`position(i)`**: position in the initial state of the piece number `i`
    - `position(16)` would be the position on the board of the empty piece
  - **`smaller(i)`**: is the number of pieces such that `j<i` and `position(j)>position(i)`

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

- The target state is reachable from a state if and only if
  - `sum(smaller(i))+x` is even
    - Where `x` is 1 If the empty cell is in one of the shaded cells and 0 if not

$$\sum_{i=1}^{16} smaller(i) + x$$

Example

position(1)=1
position(2)=5
position(3)=2
position(4)=3
position(5)=7
position(6)=10
position(7)=9
position(8)=13
position(9)=14
position(10)=15
position(11)=11
position(12)=8
position(13)=16
position(14)=12
position(15)=4
position(16)=6

smaller(1)=0	smaller(9)=0
smaller(2)=0	smaller(10)=0
smaller(3)=1	smaller(11)=3
smaller(4)=1	smaller(12)=6
smaller(5)=0	smaller(13)=0
smaller(6)=0	smaller(14)=4
smaller(7)=1	smaller(15)=11
smaller(8)=0	smaller(16)=10

$$\sum smaller(i) + x = 37 \quad (\mathrm{x} = 0)$$

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

- Therefore, the state **does not allow** to achieve the objective state

Example (II)

position(1)=1
position(2)=2
position(3)=3
position(4)=4
position(5)=5
position(6)=6
position(7)=7
position(8)=8
position(9)=9
position(10)=10
position(11)=12
position(12)=16
position(13)=13
position(14)=14
position(15)=15
position(16)=11

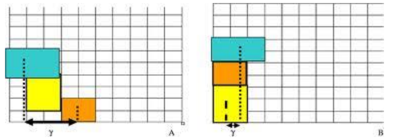
smaller(1)=0	smaller(9)=0
smaller(2)=0	smaller(10)=0
smaller(3)=0	smaller(11)=0
smaller(4)=0	smaller(12)=0
smaller(5)=0	smaller(13)=1
smaller(6)=0	smaller(14)=1
smaller(7)=0	smaller(15)=1
smaller(8)=0	smaller(16)=5

$$\sum smaller(i) + x = 8 \quad (\mathrm{x} = 0)$$

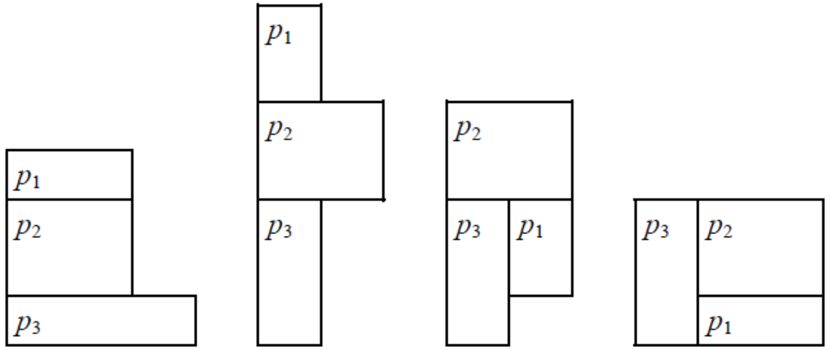
1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

- ➢ Therefore, the state **allows** to achieve the objective state

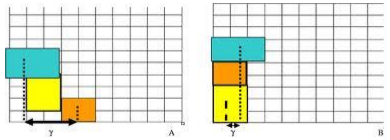
Optimal placement of rectangles



- Assuming we have n rectangular flat pieces $p_1, p_2, ..., p_n$, each with an area (a_i, b_i) where $(1 \le i \le n)$
- We want to fit them into a rectangular flat board
- The problem consists in finding an arrangement of the n pieces so that the board we need to hold them has the minimum area
  - E.g., pieces are `p1 = (1,2)`, `p2 = (2,2)` and `p3 = (1,3)`
  - The area of the covering boxes in each case is `12 (4x3)`, `14 (7x2)`, `10 (5x2)`, `9 (3x3)`



Approach

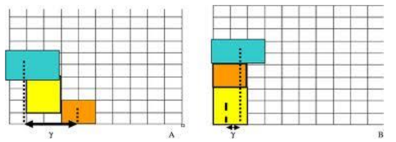


- In each step k we need to explore a piece that has not yet been placed

- For each of the pieces we have several options, since it can be placed around the pieces we have already placed both horizontally and vertically

- The simplest way to represent the solution is a matrix of integer numbers, where 0 indicates an open position and a value $k>0$ indicates that this position is occupied by the piece p_k

Branching and bounding



- Every node will **generate a child** for each possible position of the next piece not yet included in the board
- It is necessary to find the valid positions for the pieces
  - A valid position does not overlap with other pieces
  - A valid position is adjacent to other previously placed pieces
- We **prune** those nodes whose area so far exceeds that achieved by a previous solution
  - Remember we are looking for the minimum area
- Since we put a piece in each step, we have a **solution** when the level of the node is n
- The problem always has a solution because we assume that the board is large enough to accommodate all the pieces

The problem of the n queens

- The aim is to place `n` queens on a chessboard without any queen can capture another queen

- A queen can move any number of cells horizontally, vertically or diagonally

- Problem already solved with the backtracking technique
  - Now we will generate only those states in which queens do not eat other queens

Branching heuristic

- Develop the state for which we will have more queens placed
- In case of a tie, we put a queen on a cell whose longest diagonal length is the lowest one
  - `diagonal(i,j)`, length of the longest diagonal passing through the cell `(i,j)`
  - The table at right shows the values of the function in each cell

5	4	3	4	5
4	5	4	5	4
3	4	5	4	3
4	5	4	5	4
5	4	3	4	5

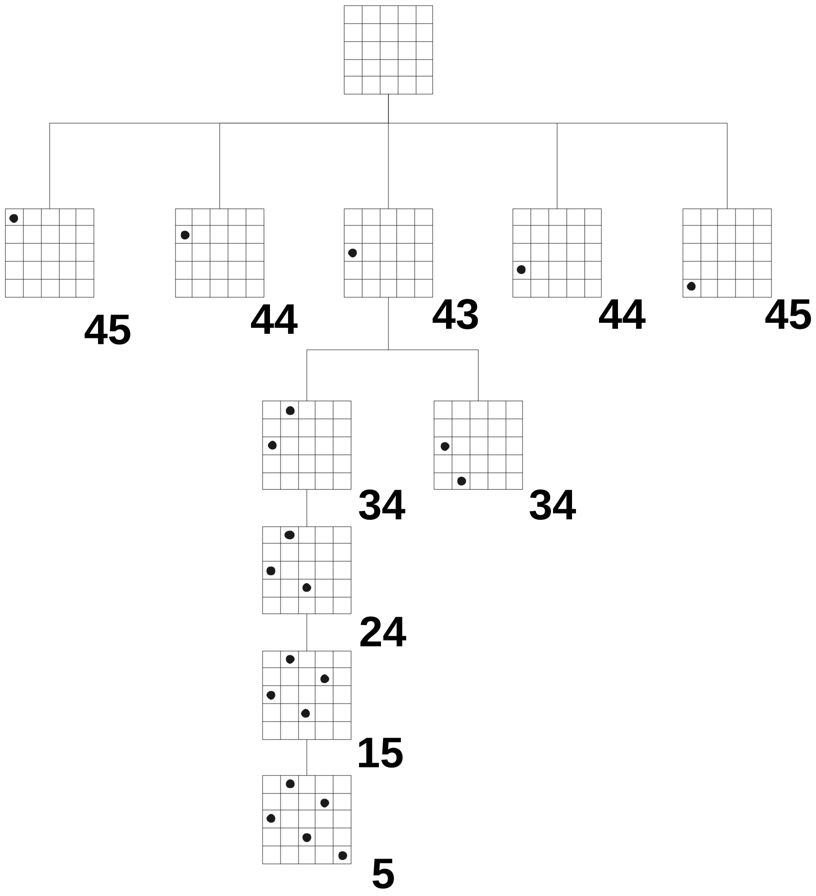
Assigning numerical values

- We seek a function → the lower the value, the better the state to be developed
  - State with more placed queens:
    - *n – nº placed queens*
  - In case of a tie, queen in a diagonal cell *diagonal(i,j)* which value is the lowest

- To merge the terms:
  - (*n – nº placed queens*) * 10 + *diagonal(i,j)*

General idea

- The "living" states are the nodes generated by the heuristic described and stored in the priority queue



Join at vevox.app

Or

Search Vevox in the app store

ID: 120-620-870

GET IT ON Google Play

Download on the App Store

Available as a Web App

POLL OPEN

If we only search for the first solution, and there are solution states near the root, the best choice is usually Backtracking

1. True
2. False

30

POLL OPEN

With Backtracking (depth-first) it is possible to enter a branch with many nodes that do not reach any solution or even that has no end (infinite)

1 True

2 False

30

POLL OPEN

If you must develop the whole tree, Backtracking is usually better than Branch and bound

1. True
2. False

30

POLL OPEN

Which data structure would be most appropriate for Branch and bound (without branching function) using Java as the programming language?

1. Stack
2. PriorityQueue
3. List
4. TreeSet
5. HashMap
6. ArrayList
7. Queue

30

POLL OPEN

Working with Backtracking, the tree of states is developed in

1. Depth-first search
2. Breadth-first search

30

POLL OPEN

With which technique we could use some kind of pruning to avoid developing bad known nodes?

1. Backtracking

2. Branch and bound

3. Both of them

30

POLL OPEN

A branching algorithm makes more sense when the idea is to get the first solution and exit

✓ 1. True

2. False

30

POLL OPEN

The pruning function prevents the development of certain states that do nothing to find the solution to the problem when they:

1. Are repeated developments
2. Do not lead to better solutions
3. Do not lead to any solution
4. All of them

30

POLL OPEN

Working with Branch and bound, the tree of states is developed in

1. Depth-first search

✓ 2. Breadth-first search

30

POLL OPEN

Which data structure would be most appropriate for Branch and bound (with branching function) using Java as the programming language?

1 List

2 Queue

✓ 3 PriorityQueue

4 ArrayList

5 TreeSet

6 Stack

7 HashMap

30

Bibliography