Heuristic Search

- It assumes an evaluation function that measures the estimated distance to a goal ($h(n)$) can be defined
- This function is used in order to guide the search, at each step the more promising state or action is chosen
- To find a solution is not always guaranteed (if there is one)
- The optimal solution is not always guaranteed (the solution with the lower cost)
- There are several algorithms:
  - Branch and Bound, Best First Search
  - A*, IDA*
  - Local search (Hill climbing, Simulated annealing, Genetic algorithms)

Branch and Bound

Branch and Bound

- Generalizes BFS, DFS
- For each state generated the cost of the path from the initial state is stored. The algorithm keeps memory of the global minimum cost.
- If the cost of the path of a state is greater that the minimum cost it is not expanded
- If the cost of each action in the problem is the same for all, the behaviour of the algorithm is a level wise search (breadth first search)

Greedy Best First

Algorithm: Greedy Best First

```
St_open.add(initial state)
current ← St_open.first()
while not is_goal?(Current) and not St_open.empty?() do
    Est_open.delete_first()
    Est_closed.add(Current)
    Successors ← generate_successors (Current)
    Successors ← treat_duplicated (Successors, St_closed, St_open)
    St_open.add(Successors)
    Current ← St_open.first()
end
```

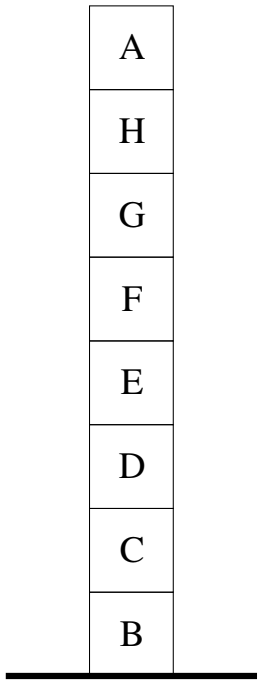
- The structure for the open nodes is a priority queue
- The priority is the value that gives the estimation function (cost of the remaining path to the goal)
- Each iteration the algorithm chooses the state that is the nearest to the goal (first node from the queue), this implies that the optimal solution is not guaranteed

Importance of the estimator

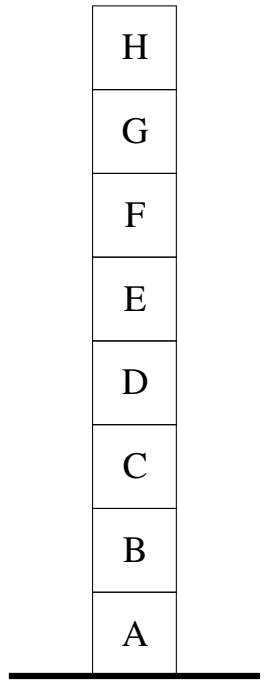
Actions:
- put a block on the table
- stack a block over another block

Heuristic 1:
- Add 1 for each block that is over the correct block
- Substract 1 if the block is not over the correct block

Heuristic 2:
- If the structure under a block is correct, add 1 for each block in the structure
- If the structure under a block is not correct, substract 1 for each block in the structure

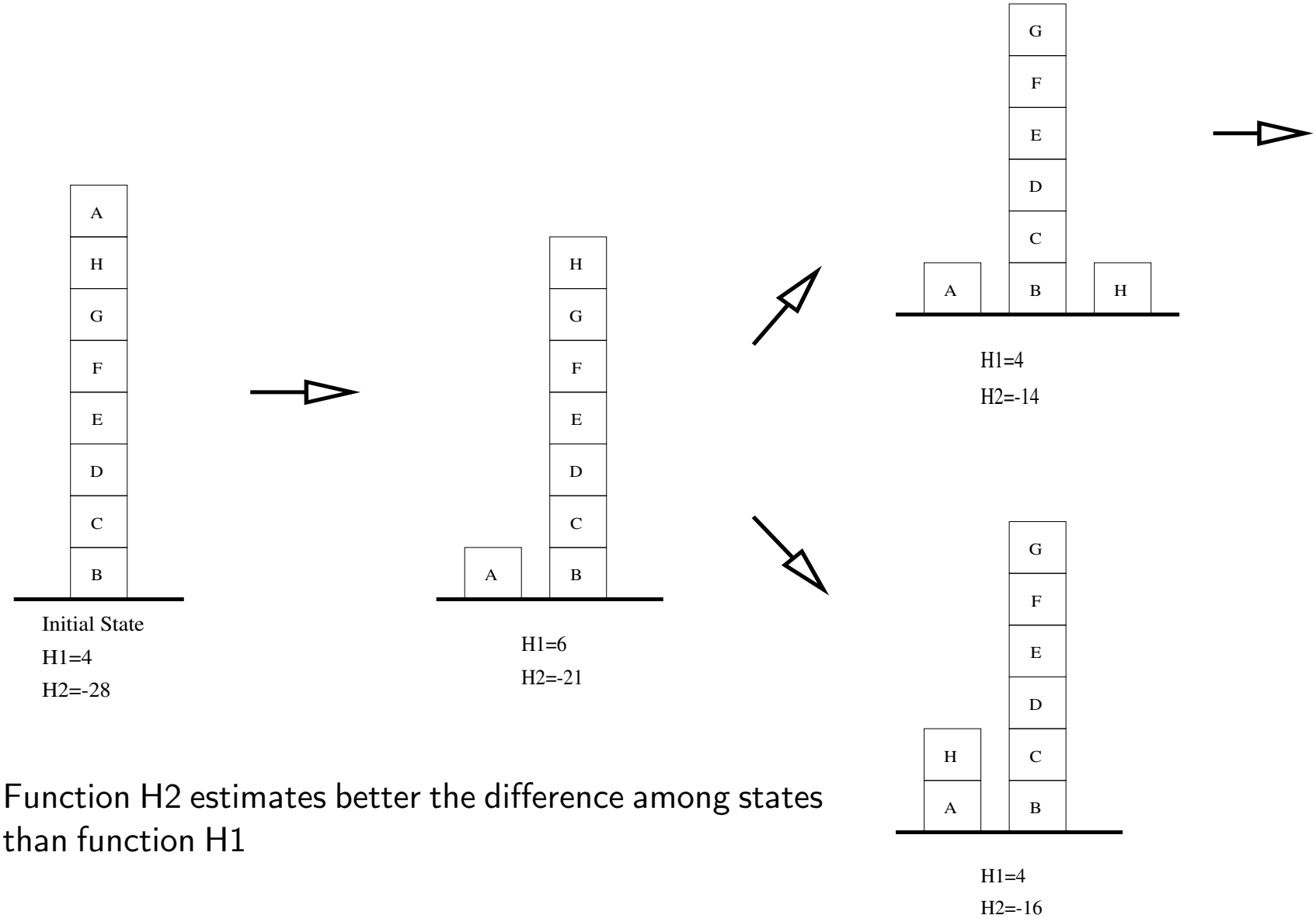


Initial State
H1=4
H2=−28



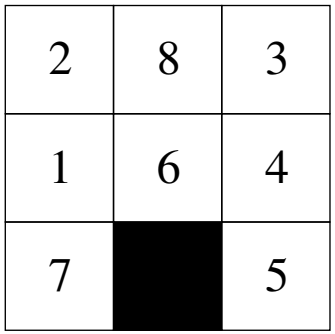
Goal State
H1=8
H2= 28 (= 7+6+5+4+3+2+1)

Importance of the estimator

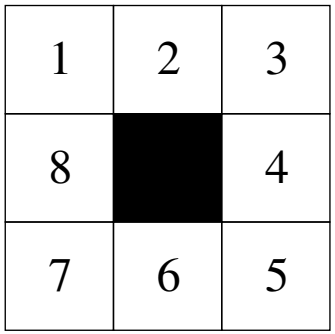


Function H2 estimates better the difference among states than function H1

Heuristics



Initial State

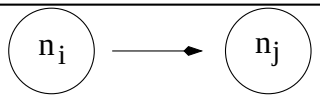


Goal State

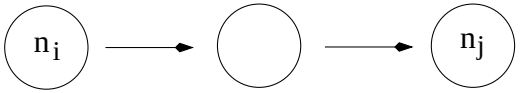
Possible heuristics (estimation of the number of steps of the solution)

- $h(n) = w(n) = \#$ tiles in wrong position
- $h(n) = p(n) =$ sum of distances of tiles to final position
- $h(n) = p(n) + 3 \cdot s(n)$
  where $s(n)$ is computed using all the positions except the central one, if the tile next to a tile in a position is not its consecutive one, adds 2 to the value, if the central position is not blank, adds 1 to the value

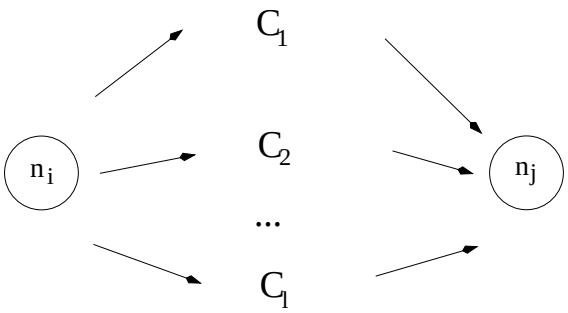
Costs



Cost of an edge $c(n_i, n_j) > 0$



Cost of a path $C = \sum_{x=i}^{j-1} c(n_x, n_{x+1})$



Cost of the minimal path $K(n_i, n_j) = \min_{k=1}^{l} C_k$

If n_j is a goal node $h^*(n_i) = K(n_i, n_j)$

If n_i is an initial node $g^*(n_j) = K(n_i, n_j)$

If there are many goal nodes $T = \{t_1, \ldots, t_l\}$ $h^*(n_i) = \min_{k=1}^{l} K(n_i, t_k)$

If there are many initial nodes $S = \{s_1, \ldots, s_l\}$ $g^*(n_j) = \min_{k=1}^{l} K(s_k, n_j)$

A* Search

The estimation function has two components:

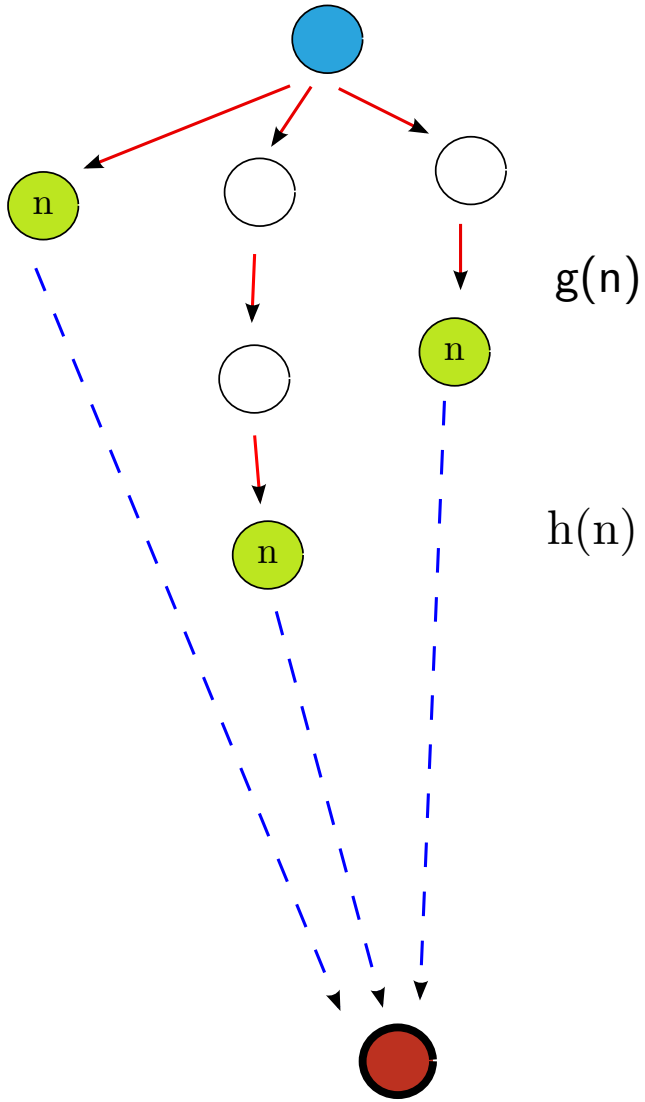
1. Cost from the initial node to the current node
2. Cost (estimated) from the current node to the goal node

$$f(n) = g(n) + h(n)$$

- f is the estimated value of the total cost of the path that contains n
- h (heuristic) is the estimated value of the cost from the node n to the goal node
- g is the actual cost from the shortest known path from the initial state to the node n

As a convention the preference is to expand the node with the lower f, ties are break choosing the lower h.

A* Search



$$f(n)=g(n)+h(n)$$

A* Search

The characteristics of this function allow to change the behaviour of the algorithm

- If $\forall n \;\; h(n) = 0$, the search is controlled by g (we will have a *Branch & Bound search*)
- If $\forall n \;\; h(n) = 0$ and the cost of all actions is 1 we will have *Breadth-first search*. If the cost is 0, we will have *random search*
- Being h an estimate of the real cost h^*, the nearest h to h^* the algorithm will behave more as a Depth-first search. If $h = h^*$ then A* converges directly to the goal

It can be proven that if $\forall n \;\; h(n) \leq h^*(n)$, then A* will always find the optimal path to the goal node (if there is one).

A* Algorithm

Algorithm: A*

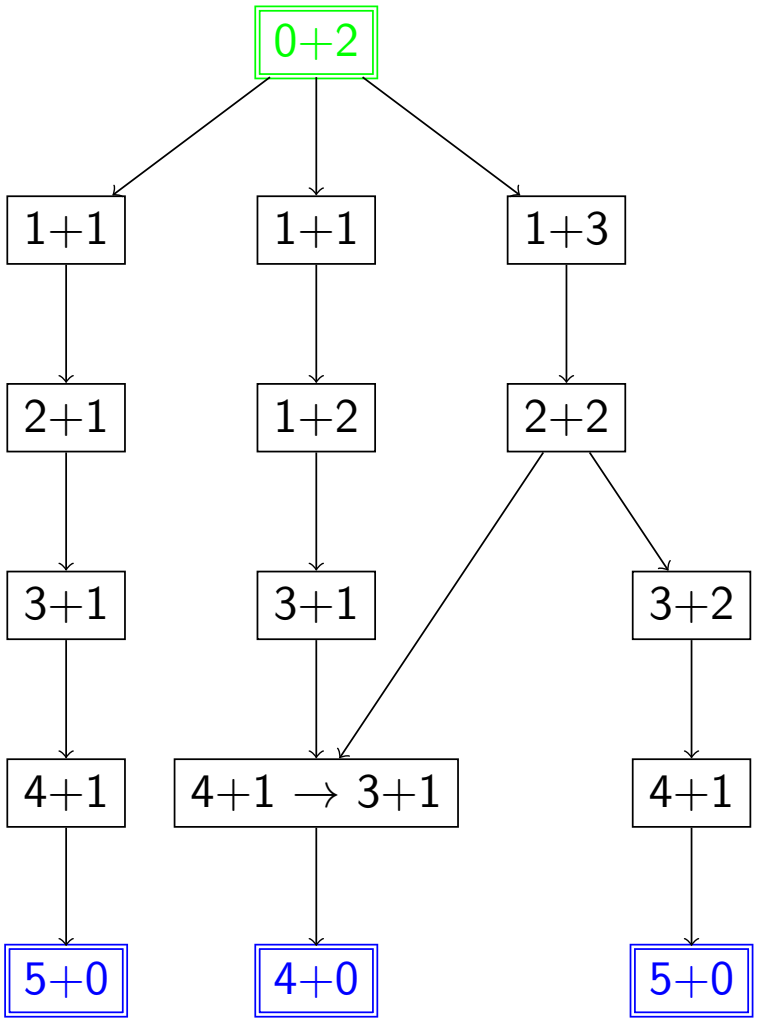
```
St_open.add(initial state)
Current ← St_open.first()
while not is_goal?(Current) and not St_open.empty?() do
    St_open.delete_first()
    St_closed.add(Current)
    Successors ← generate_successors (Current)
    Successors ← treat_duplicated (Successors, St_closed, St_open)
    St_open.add(Successors)
    Current ← St_open.first()
end
```

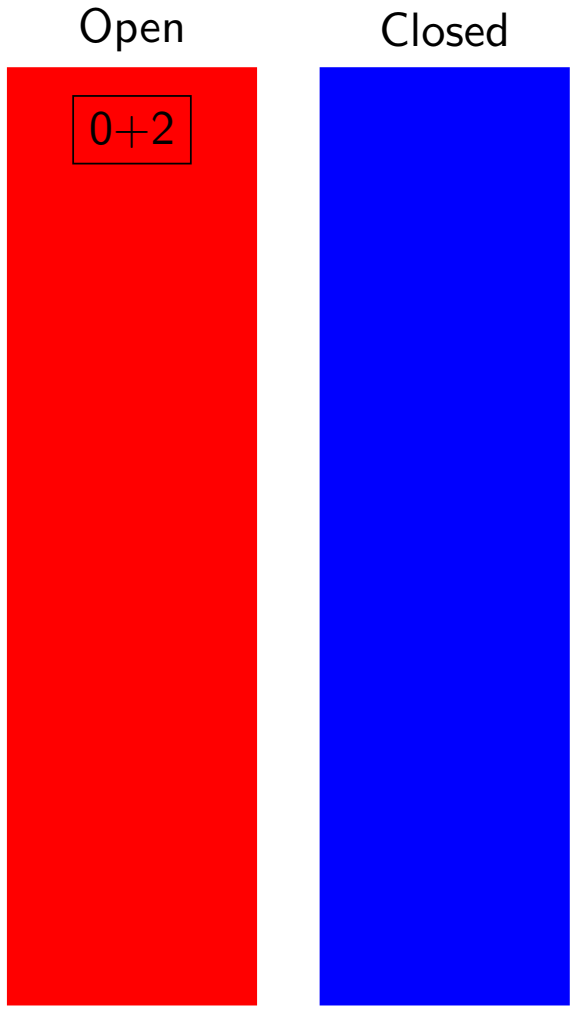
- The structure for the open nodes is a priority queue
- The priority is the value of the estimation function ($f(n) = g(n) + h(n)$)
- Each iteration the node with the shortest path is chosen (the first one from the queue)
- It is the same algorithm than Best First search!

Treatment of duplicated nodes

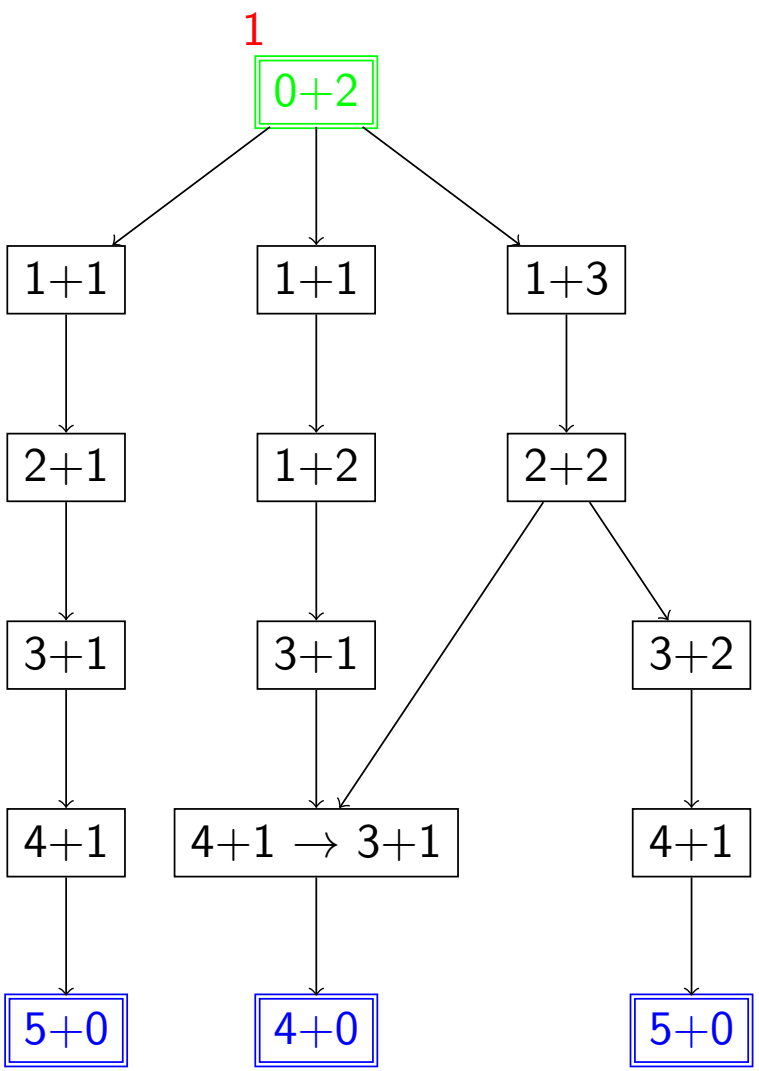
- If the duplicate node is in the structure for open nodes
  - If it has less cost, the old one is substituted by the current one, this means that the position of the node in the structure could change
  - If the cost is greater or equal the node can be discarded
- If the duplicate node is in the structure for closed nodes
  - If it has less cost, it is reopen and it is inserted in the structure for open nodes with the new cost Attention! The successors are not changed, if it is necessary they will be reopen in the future
  - If the cost is greater or equal the node can be discarded

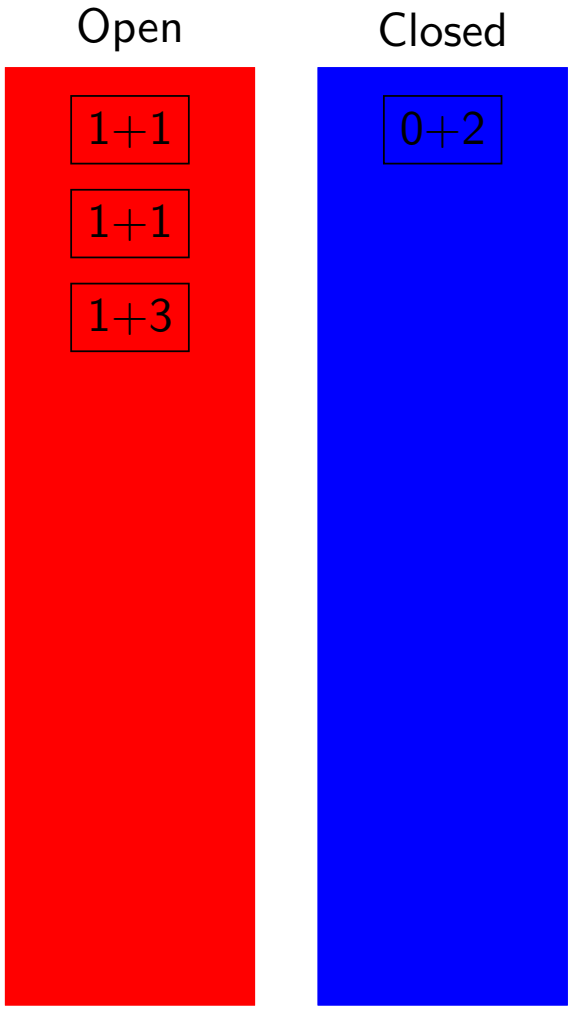
A* Example



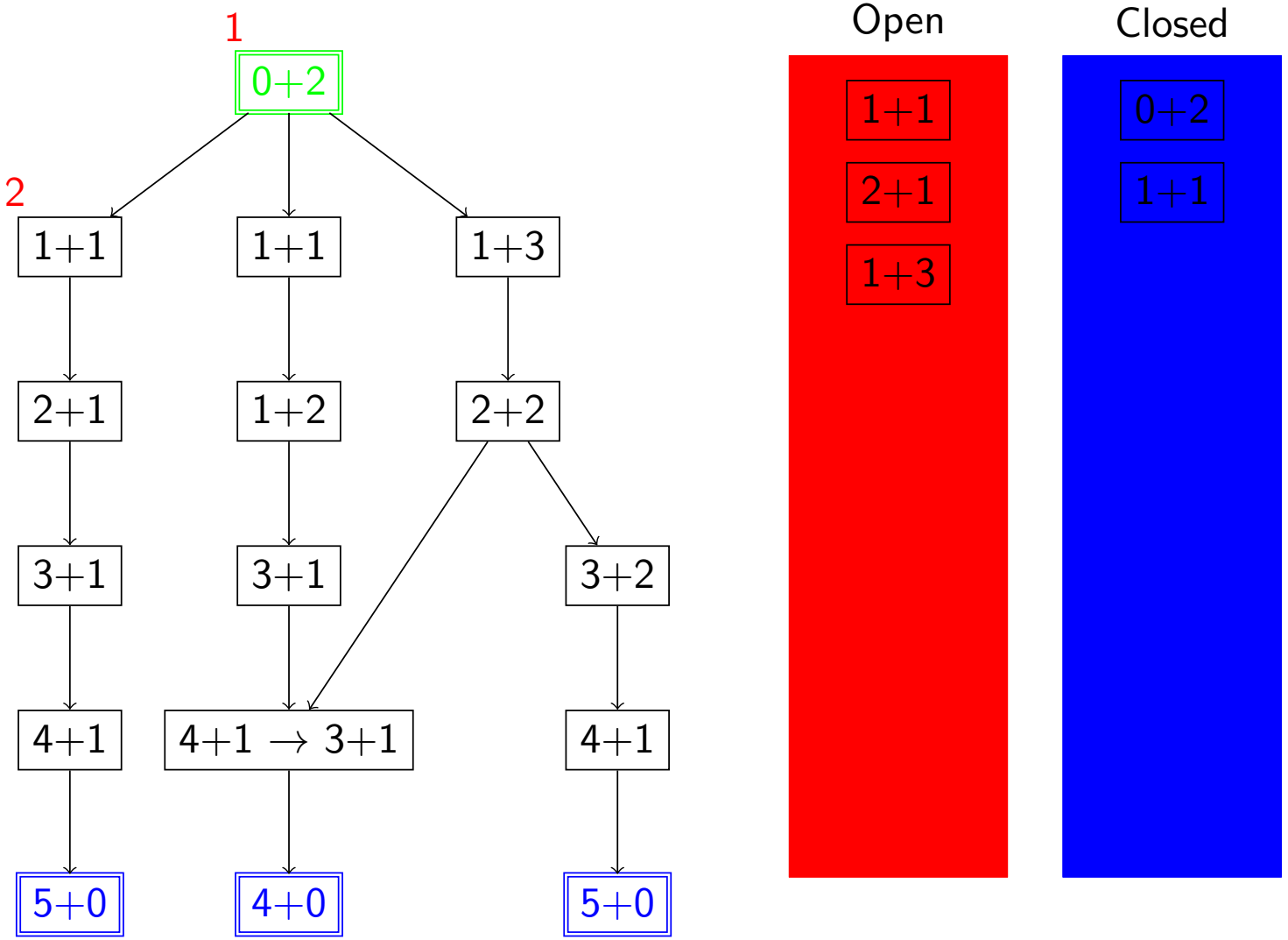


A* Example

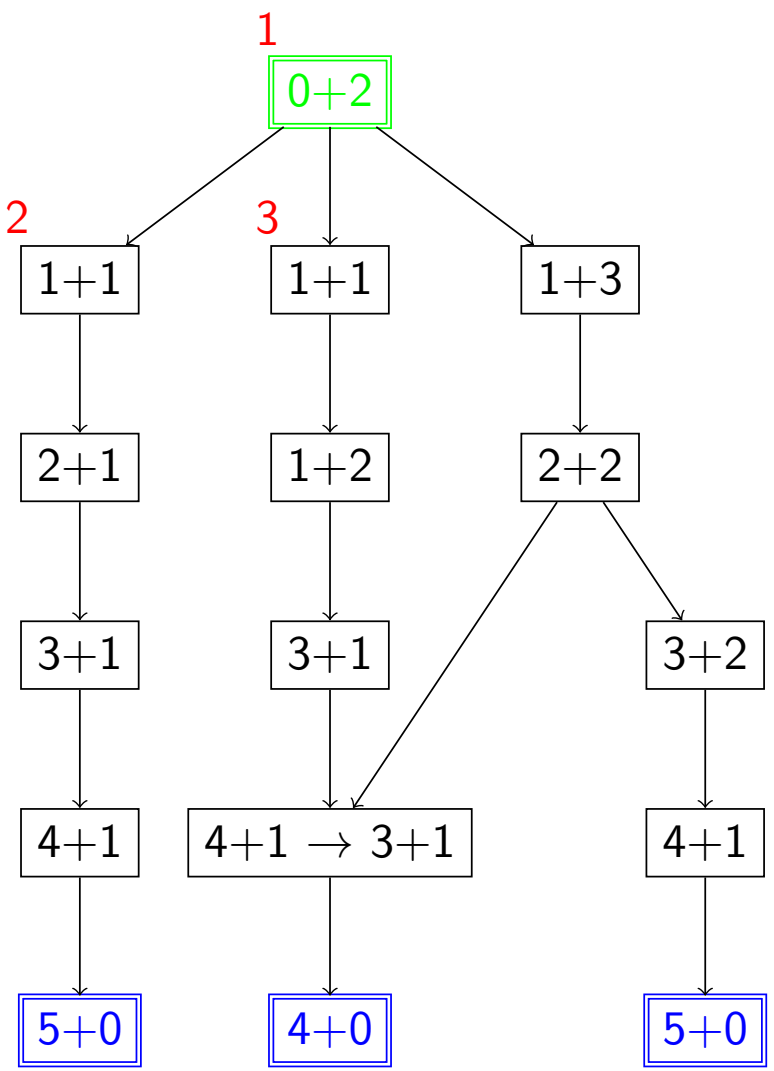

1
0+2
1+1
1+1
1+3
2+1
1+2
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0

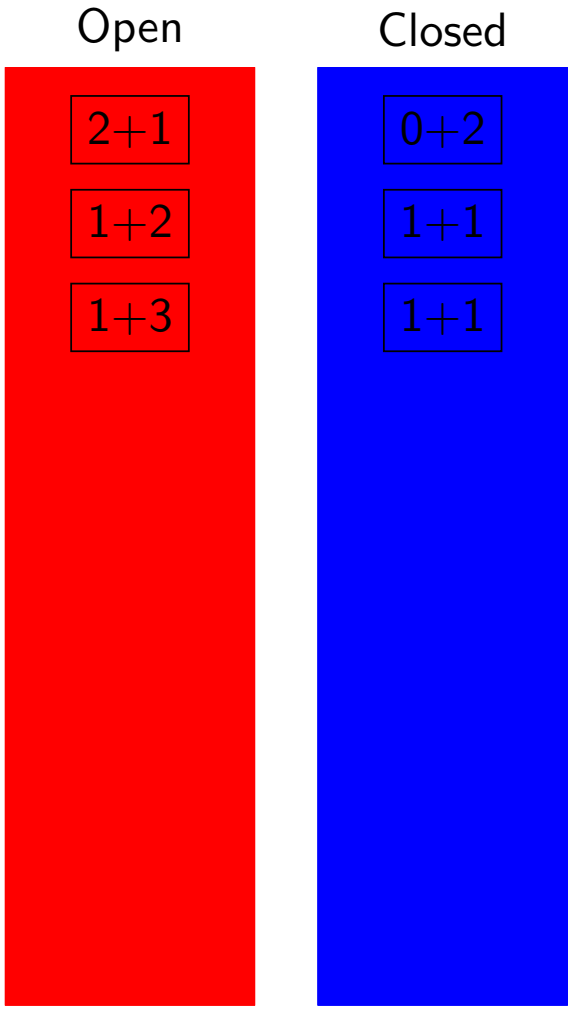

Open
Closed
1+1
1+1
1+3
0+2

A* Example

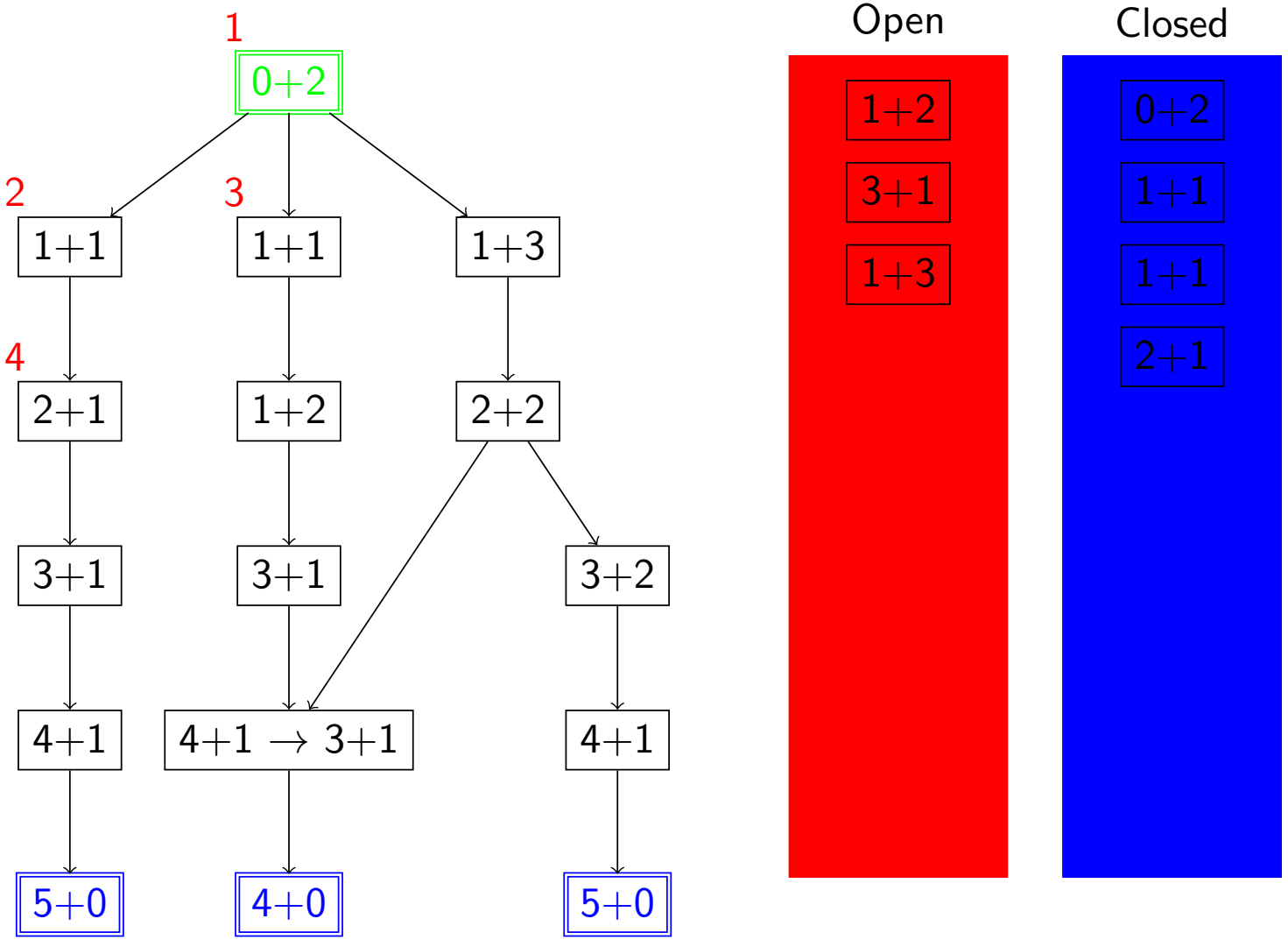

1
0+2
2
1+1
1+1
1+3
2+1
1+2
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0
Open
1+1
2+1
1+3
Closed
0+2
1+1

A* Example

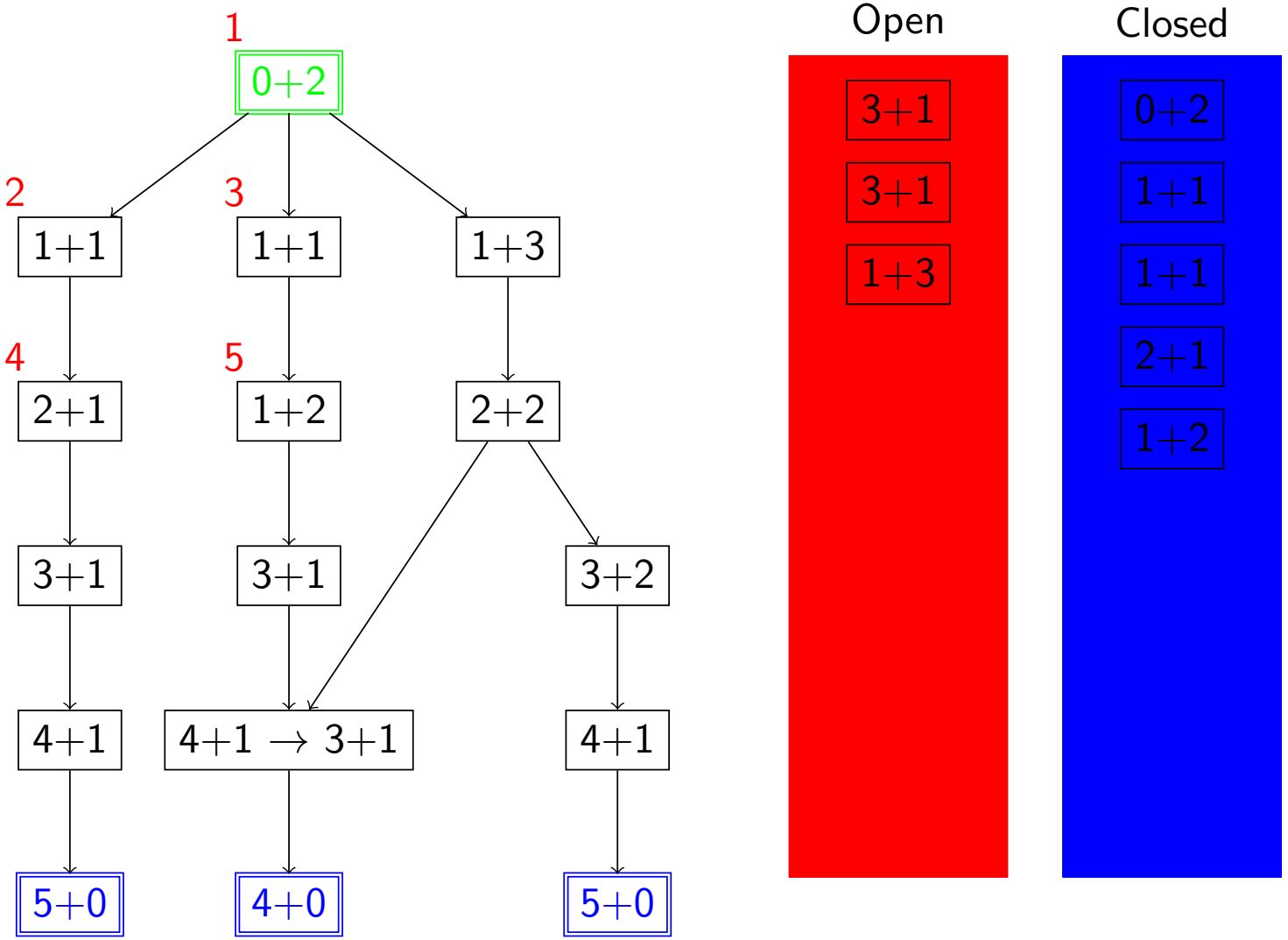

1
0+2
2
3
1+1
1+1
1+3
2+1
1+2
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0


Open
Closed
2+1
1+2
1+3
0+2
1+1
1+1

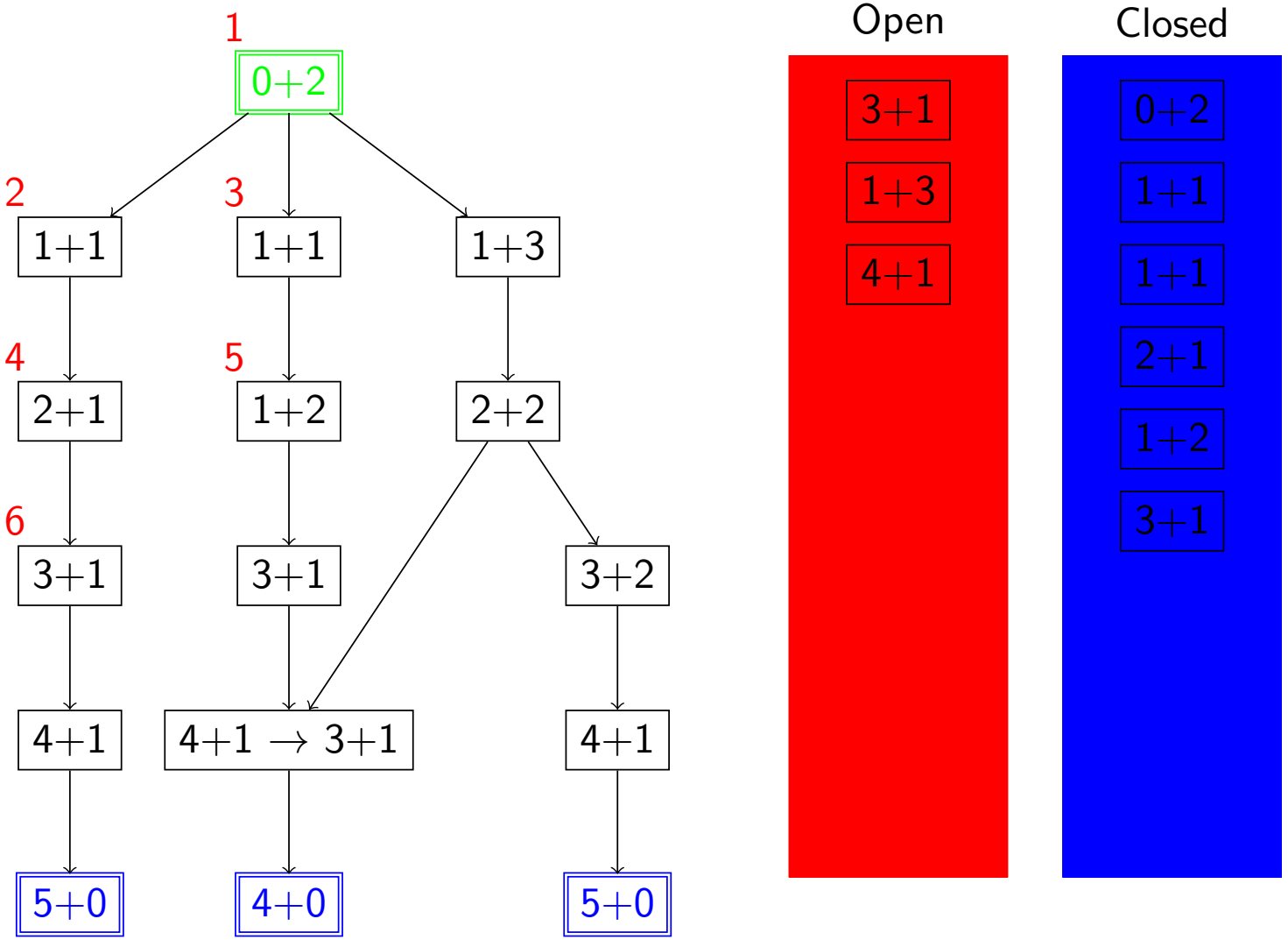
A* Example


1
0+2
2
1+1
3
1+1
1+3
4
2+1
1+2
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0
Open
1+2
3+1
1+3
Closed
0+2
1+1
1+1
2+1

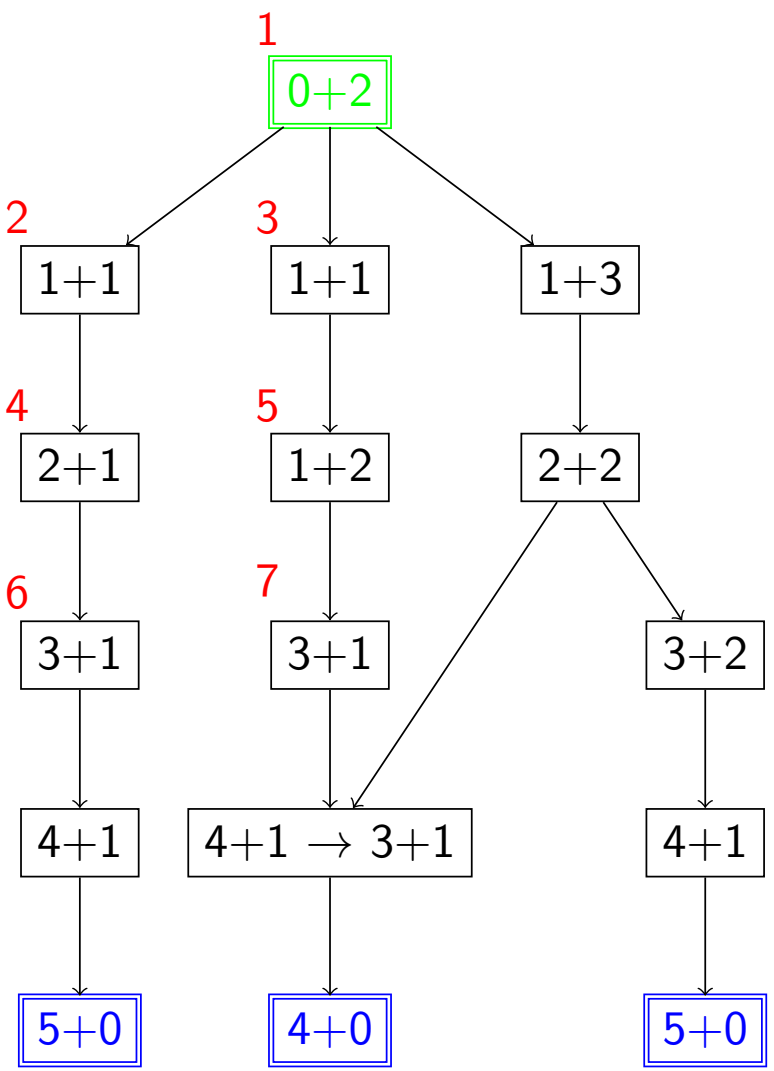
A* Example

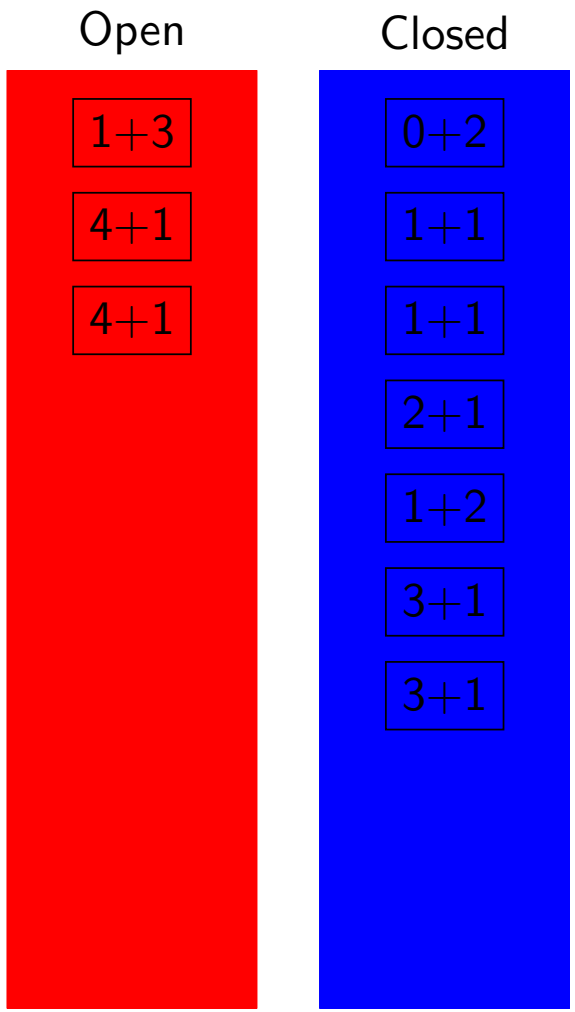

1
0+2
2
1+1
3
1+1
1+3
4
2+1
5
1+2
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0
Open
3+1
3+1
1+3
Closed
0+2
1+1
1+1
2+1
1+2

A* Example

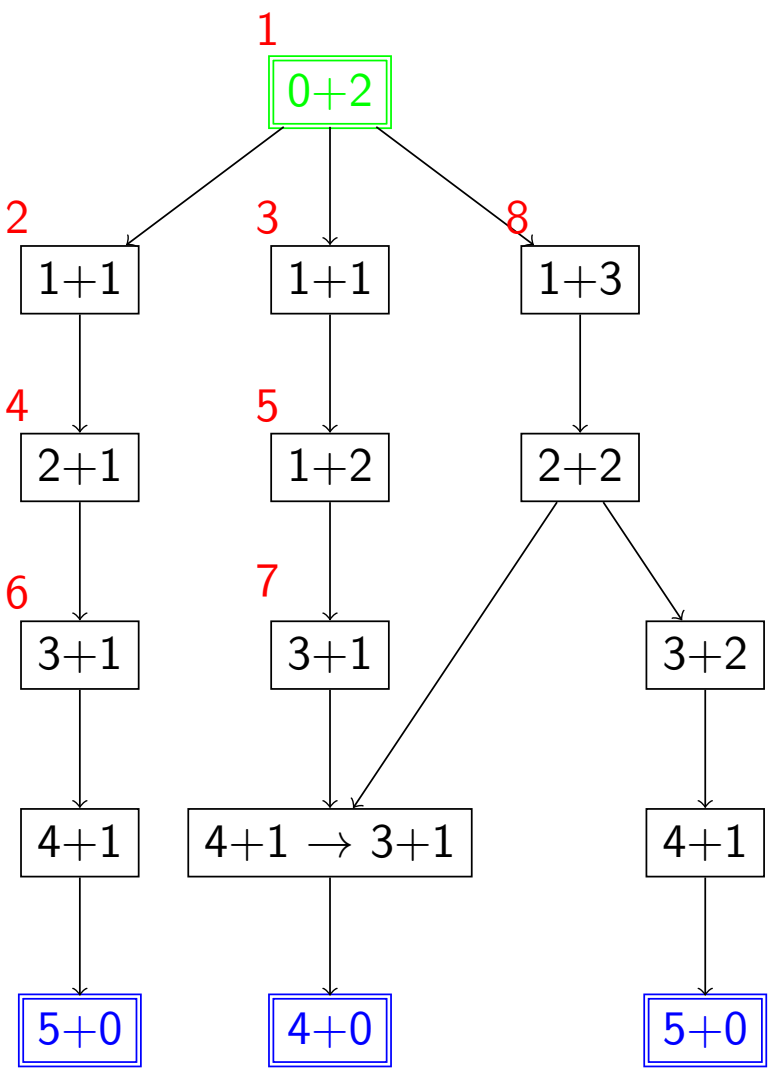

1
0+2
2
1+1
3
1+1
1+3
4
2+1
5
1+2
2+2
6
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0
Open
3+1
1+3
4+1
Closed
0+2
1+1
1+1
2+1
1+2
3+1

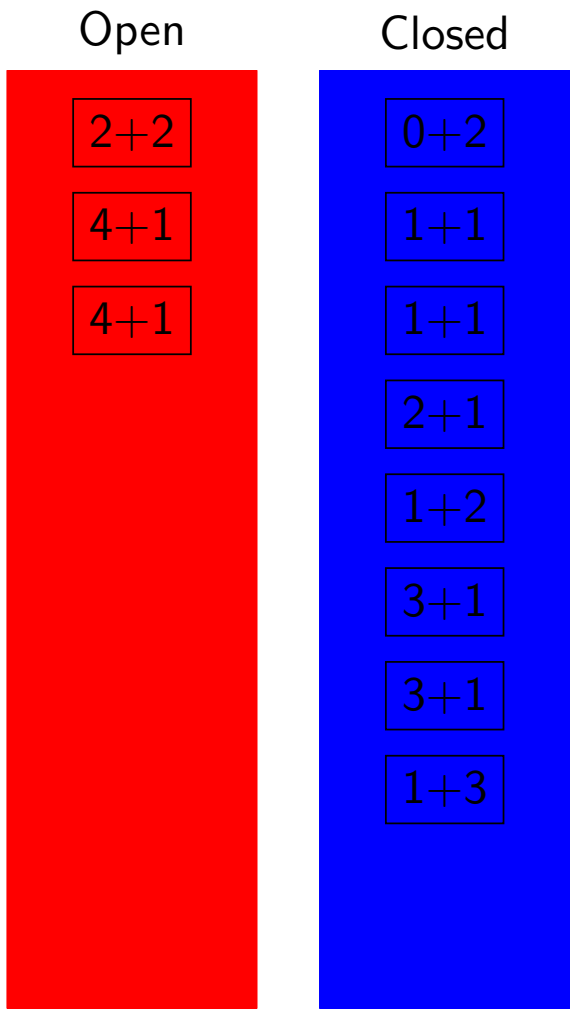
A* Example


1
0+2
2
1+1
3
1+1
1+3
4
2+1
5
1+2
2+2
6
3+1
7
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0

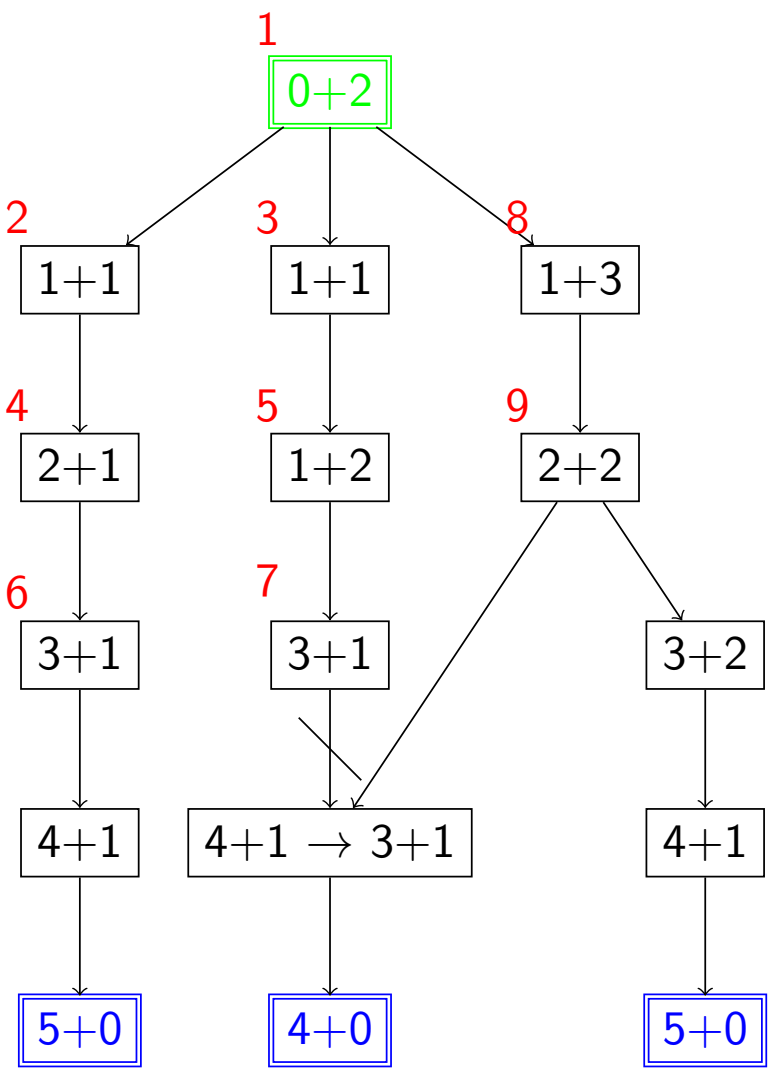

Open
1+3
4+1
4+1
Closed
0+2
1+1
1+1
2+1
1+2
3+1
3+1

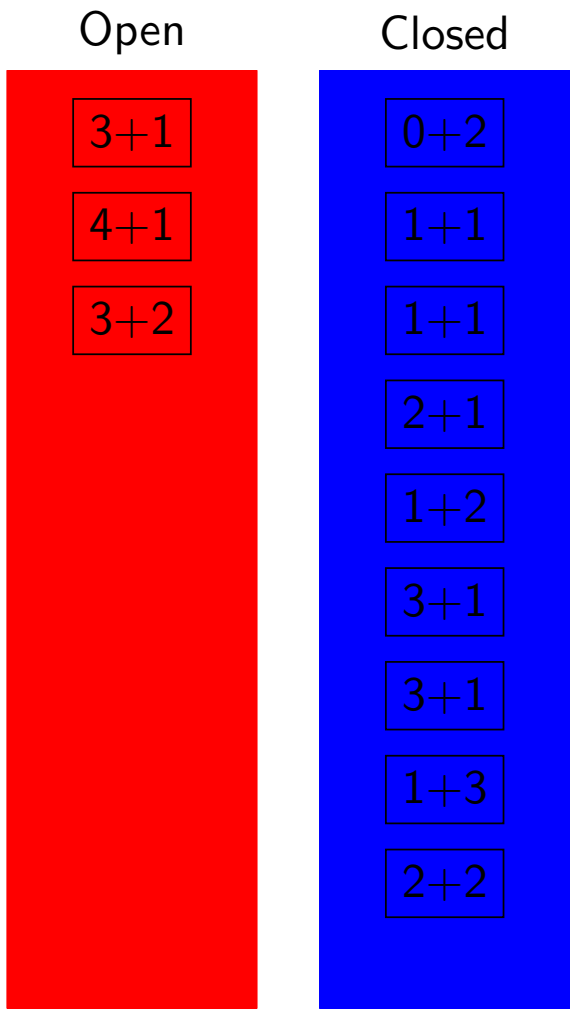
A* Example


1
0+2
2
1+1
3
1+1
8
1+3
4
2+1
5
1+2
2+2
6
3+1
7
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0

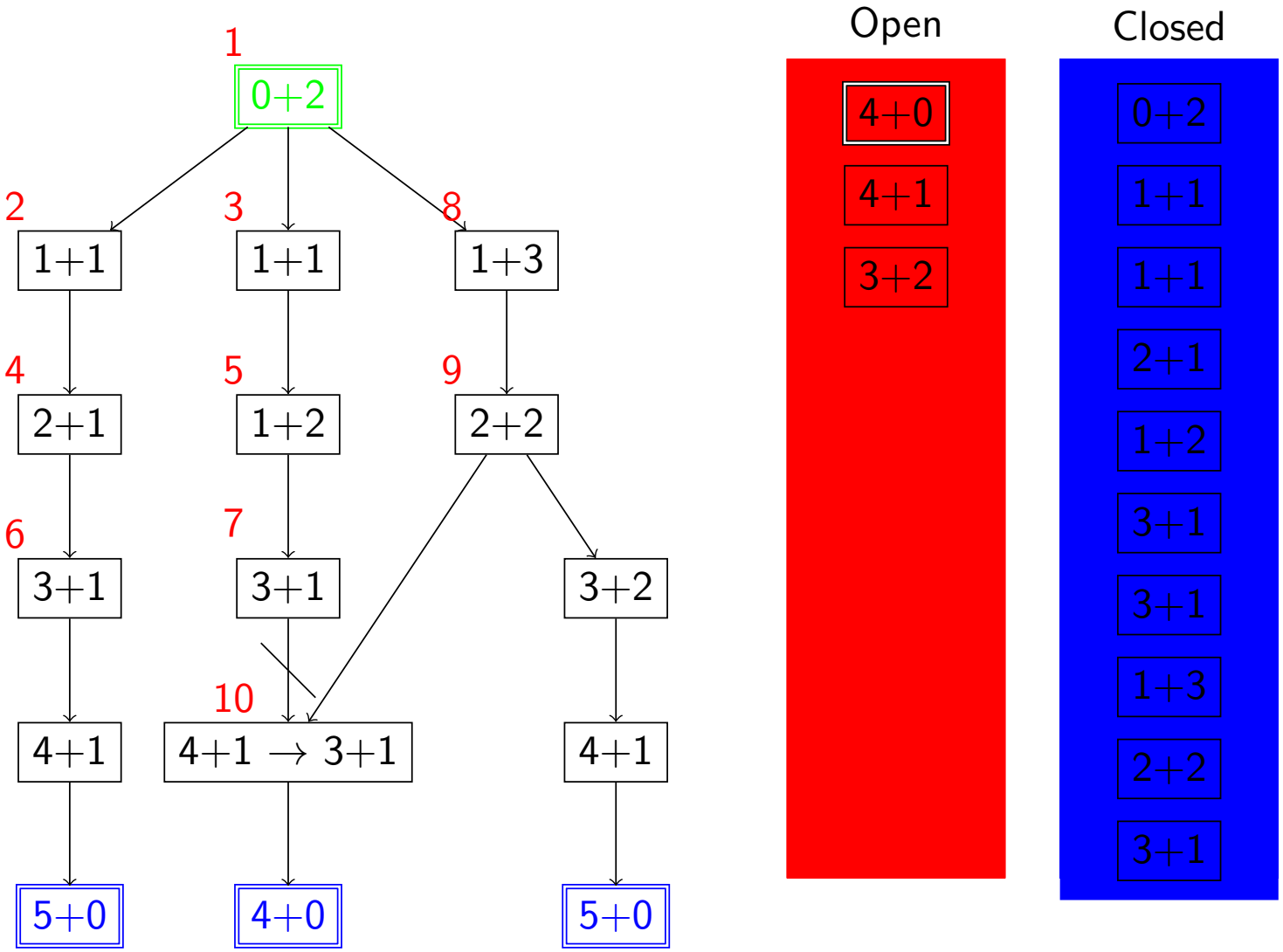

Open
2+2
4+1
4+1
Closed
0+2
1+1
1+1
2+1
1+2
3+1
3+1
1+3

A* Example

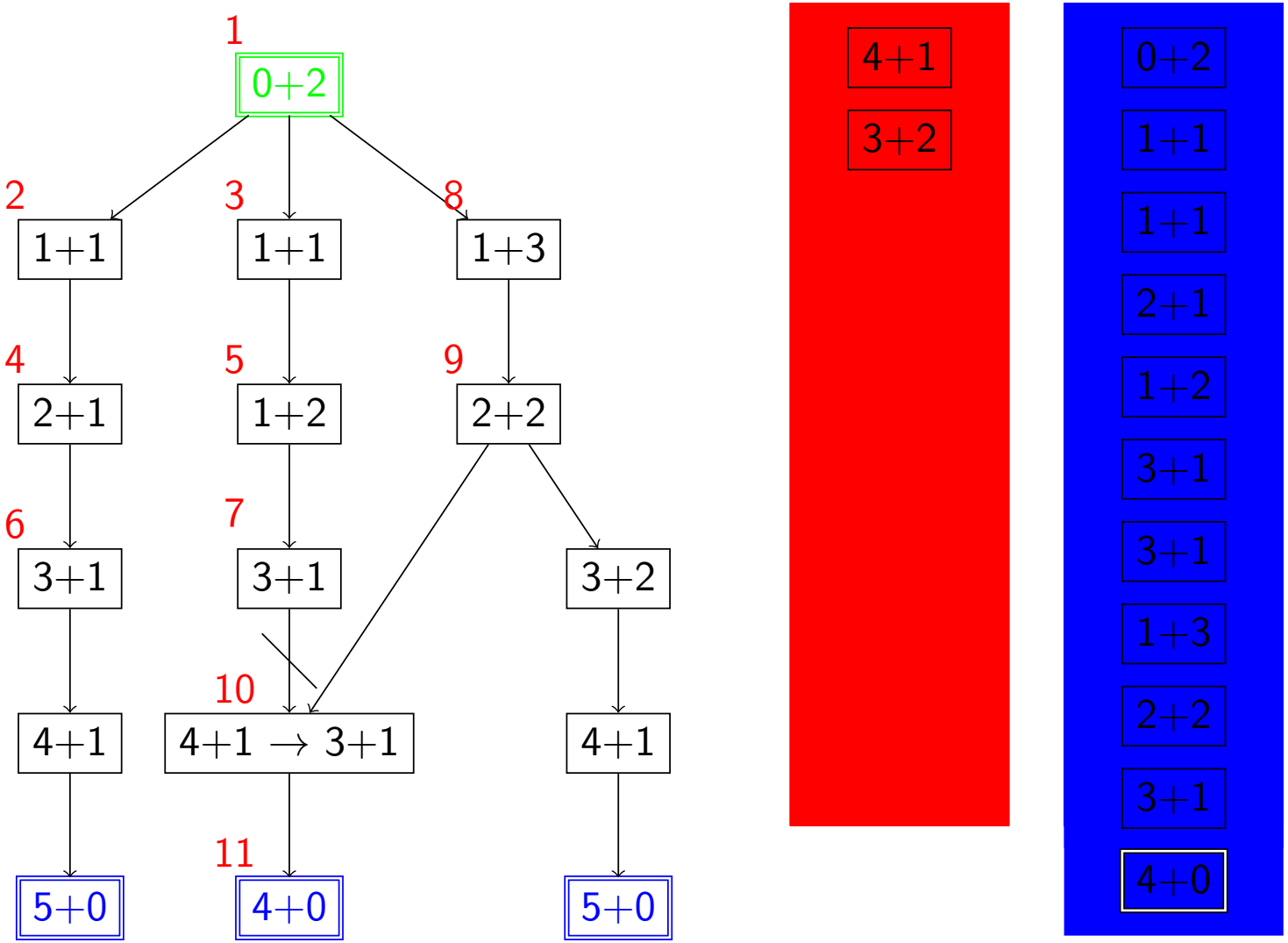

1
0+2
2
1+1
3
1+1
8
1+3
4
2+1
5
1+2
9
2+2
6
3+1
7
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0


Open
3+1
4+1
3+2
Closed
0+2
1+1
1+1
2+1
1+2
3+1
3+1
1+3
2+2

A* Example


1
0+2
2
1+1
3
1+1
8
1+3
4
2+1
5
1+2
9
2+2
6
3+1
7
3+1
3+2
10
4+1 → 3+1
4+1
4+1
5+0
4+0
5+0
Open
4+0
4+1
3+2
Closed
0+2
1+1
1+1
2+1
1+2
3+1
3+1
1+3
2+2
3+1

A* Example

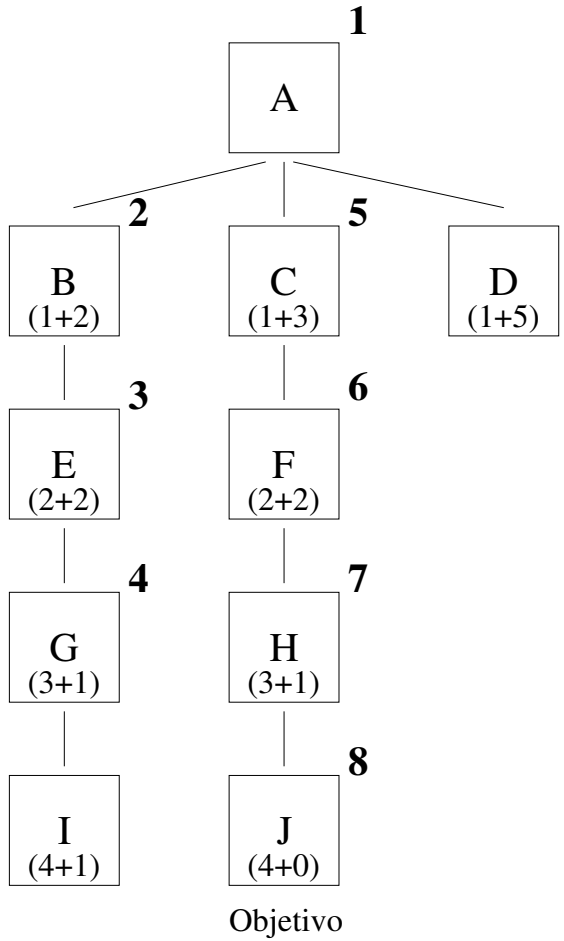

1
0+2
2
1+1
3
1+1
8
1+3
4
2+1
5
1+2
9
2+2
6
3+1
7
3+1
3+2
10
4+1 → 3+1
4+1
4+1
11
5+0
4+0
5+0
Open
4+1
3+2
Closed
0+2
1+1
1+1
2+1
1+2
3+1
3+1
1+3
2+2
3+1
4+0

Admissibility

- The A* algorithm will find the optimal solution depending on the heuristic
- If the heuristic is admissible then **optimality** is guaranteed
- An heuristic is admissible if:

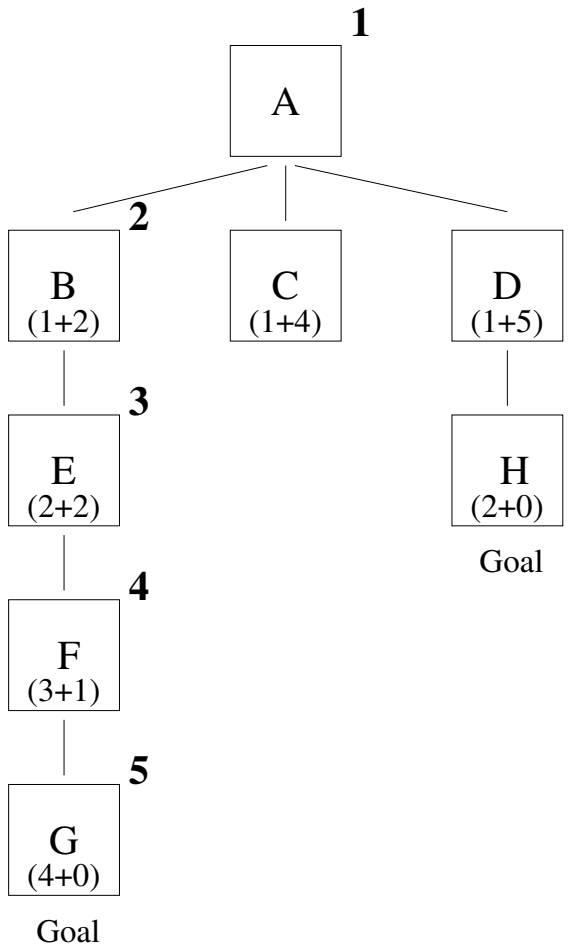
$$\forall n \;\; 0 \leq h(n) \leq h^*(n)$$

- This means that $h(n)$ must be an optimistic estimator of the actual cost $h^*(n)$



h underestimates h^*

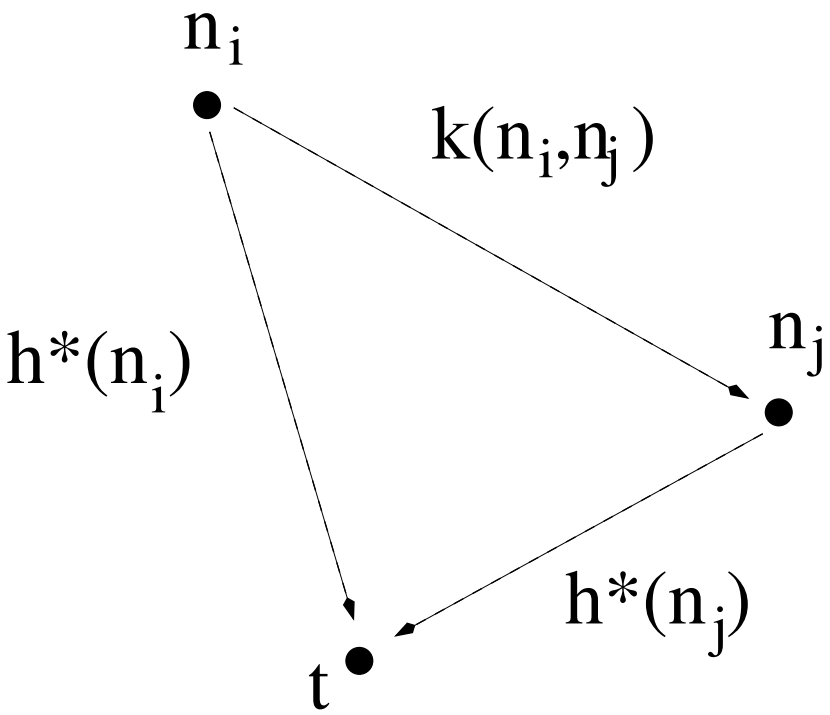
$h(B) = 2$, lower that the real cost, we waste some time but other paths will be explored (C)



h overestimates h^*

if it was a shortest path from D, we will never explore it

Consistency (Monotonicity)



- $h^*(n_i)$ = minimal cost from n_j to t
- $h^*(n_j)$ = minimal cost from n_i to t
- $h^*(n)$ holds the triangular inequality

$$h^*(n_i) \leq k(n_i, n_j) + h^*(n_j)$$

- Consistency demands that $h(n)$ has the same behaviour than $h^*(n)$

$$h(n_i) - h(n_j) \leq k(n_i, n_j)$$

- Consistency of $h(n) \Longrightarrow h(n)$ is an uniform estimate of $h^*(n)$

Utility of consistency

- $h(n)$ consistency $\Longrightarrow$ $g(n) = k(s, n)$ (shortest path to n)
- If $g(n) = k(s, n)$, given that $h(n)$ is always the same value $\Longrightarrow$ $f(n)$ is minimal
- In this case it is not necessary to treat closed duplicate nodes because all expanded nodes will not be re expanded (we have reached the nodes from the shortest path)

More informed

Given a problem there are as many A* instantiations to solve it as heuristic we can define.

More informed algorithm

For A_1^* and A_2^* with admissible heuristics, if

$$\forall n \neq goal \;\; 0 \leq h_2(n) < h_1(n) \leq h^*(n)$$

then A_1^* is more informed than A_2^*, the heuristic $h_1(n)$ is also more informed than $h_2(n)$

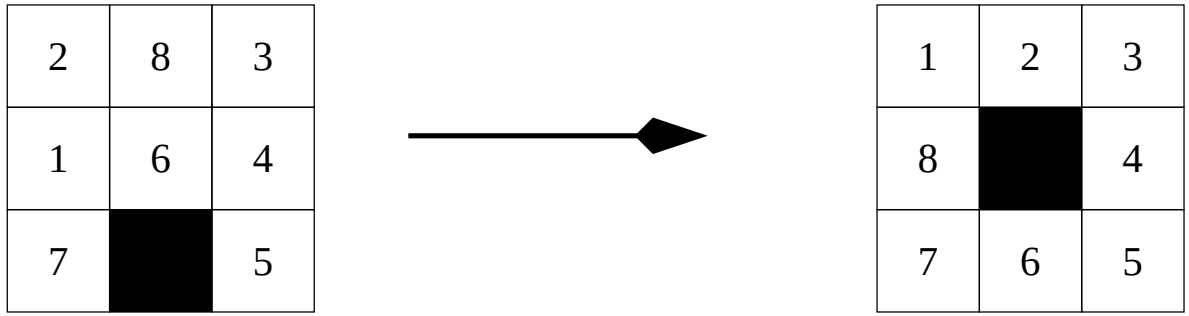
- If A_1^* is more informed than A_2^* then if node n is expanded by $A_1^* \Longrightarrow$ n is expanded by A_2^* (but not the other way)
- This means that A_1^* expands less nodes than A_2^*

More informed algorithm

- Does this mean that we have to choose the heuristic that gives the more informed algorithm?
- Trade off between:
  - Temporal cost to compute h
    - $h_1(n)$ will need more time than $h_2(n)$ to be computed
  - Number of reexpansions
- Non admissible heuristics
  - It could be interesting to use non admissible heuristics to speed up the search
  - A^*_ϵ algorithms (ϵ-admissibility)

More informed algorithms - 8 puzzle

Eight puzzle



h_0

$h_0(n) = 0$ Equivalent to breadth-first search, h_0 admissible, lots of expanded nodes

h_1

$h_1(n) = \#misplaced\ tiles$ h_1 admissible, A_1^* more informed than A_0^*

More informed algorithms - 8 puzzle

h_2

$h_2(n) = \sum_{i \in [1,8]} d_i$ $\quad d_i \equiv$ distances from the position of tile i to its final position

h_2 admissible, Statistically can be proven that A_2^* is more informed than A_1^*

h_3

$h_3(n) = \sum_{i \in [1,8]} d_i + 3 \cdot S(n)$; *with* $S(n) = \sum_{i \in [1,8]} s_i$

$$s_i = \begin{cases} 0 \text{ if tile } i \text{ is not in the center and correct successor} \\ 1 \text{ if tile } i \text{ is in the center} \\ 2 \text{ if tile } i \text{ is not in the center and incorrect successor} \end{cases}$$

h_3 is not admissible, so it is not more informed than h_2^*, but is faster

Memory bound optimal search

- A* algorithm solve problems that need the optimal solution
- Spatial and temporal cost is in the average case better that blind search algorithms given the adequate heuristic
- Unfortunately there are problems that can not be solved using A* algorithm because of the size of their search space
- Besides the number of nodes that A* needs to store grows exponentially if:

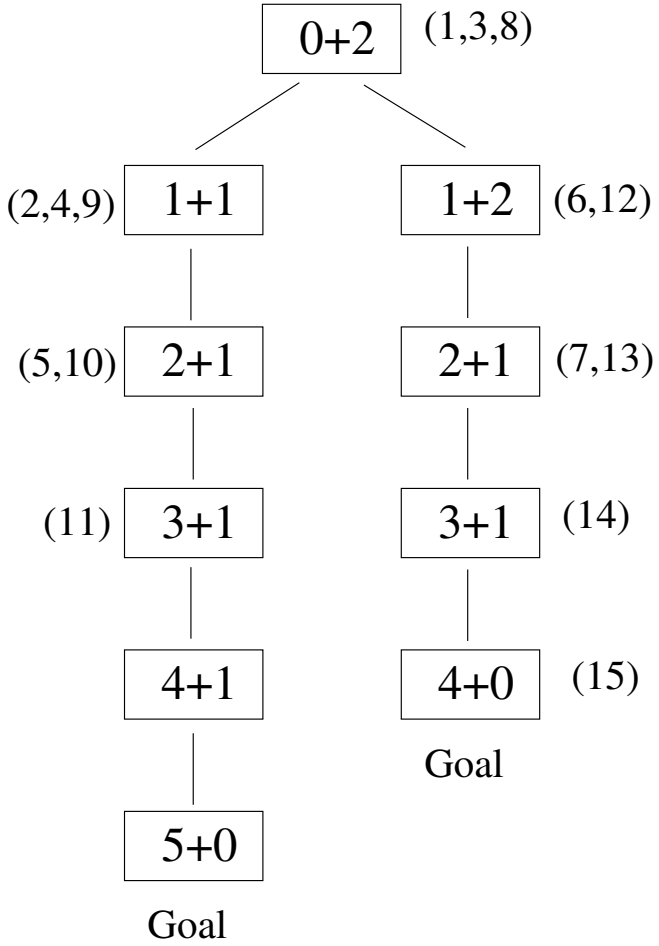
$$|h(n) - h^*(n)| \geq \log(h^*(n))$$

- There are algorithms that can find the optimal solution with limited memory:
  - IDA*
  - Recursive Best First
  - Memory Bound A* (MA*)

IDA* algorithm

- Similar to ID (repeated depth limited first search increasing the depth of search)
- ID uses as a limit the levels to the root
- IDA* uses a bound over the value of the heuristic function f

Attention! Each iteration the algorithm performs a DFS, the depth is the value of the heuristic f
We start with an initial maximum depth = $f(\textit{initial state})$



IDA* Algorithm

Algorithm: IDA* (limit: integer)

```
depth ← f(initial state)
Current ← initial state
while not is_goal?(Current) and depth<limit do
    St_open.reset()
    St_open.add(initial state)
    Current ← St_open.first()
    while not is_goal?(Current) and not St_open.empty?() do
        St_open.delete_first()
        St_closed.add(Current)
        Successors ← generate_successors (Current, depth)
        Successors ← treat_duplicated (Successors, St_closed, St_open)
        St_open.add(Successors)
        Current ← St_open.first()
    depth ← depth+1
```

- The function `generate_successors` only generate nodes with f less or equal than the limit of the current iteration
- The structure for the open nodes is a stack (Depth-first search)
- As in all the depth-first algorithms duplicated treatment eliminates the space gain

Other memory bound search algorithms

- IDA* reexpansions have an elevated temporal cost
- There are algorithms that usually need to re expand less nodes
- The main idea is to discard the less promising nodes but to store enough information to know when to re expand
- Examples:
  - Recursive Best first
  - Memory Bound A* (MA*)

Recursive Best First

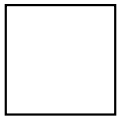
- It is a recursive implementation of the Best First algorithm with linear spatial cost $O(bd)$
- Discards a partial path when its cost is worst than the best alternative
- The cost of the discarded path is stored in its ancestor substituting its own cost
- The path will be re expanded in the future if its cost is the best alternative

Recursive Best First - Algorithm

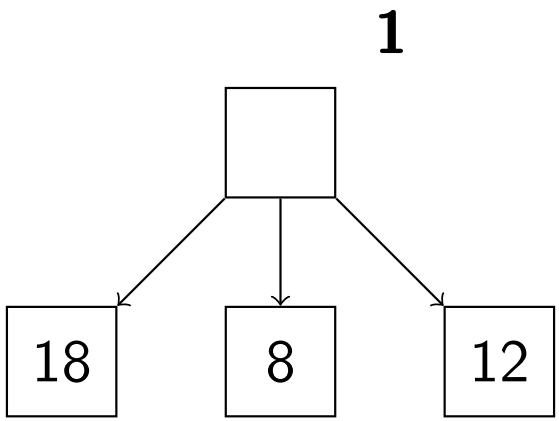
Procedure: Recursive-BFS (node, alternative_c, new_cost, solution)

```
if is_solution?(node) then
    Solution.add(node)
else
    Successors ← generate_successors (node)
    if Successors.empty?() then
        new_cost ← +∞; solution.reset()
    else
        end← false
        while not end do
            best ← Successors.best_node()
            if best.cost() > alternative_c then
                end ← true; solution.reset(); new_cost ← best.cost()
            else
                Second ← Successors.second_best()
                recursive-BFS(best,min(alternative_c, Second.cost()), new_cost, solution)
                if solution.empty?() then
                    best.cost(new_cost)
                else
                    solution.add(best); end ← true
```

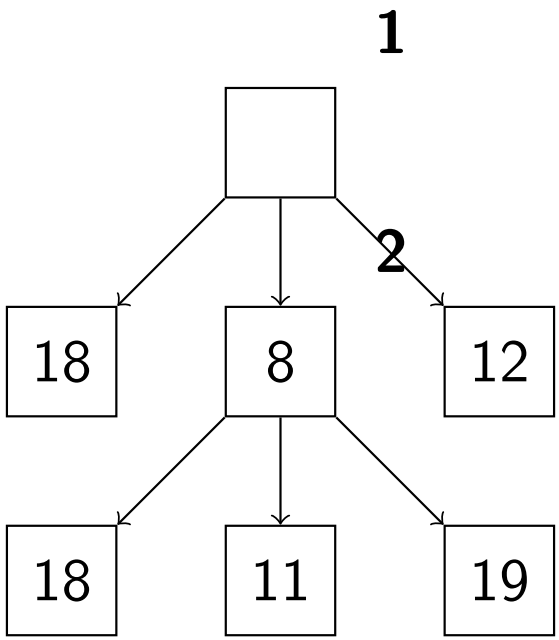
Recursive Best First - Example



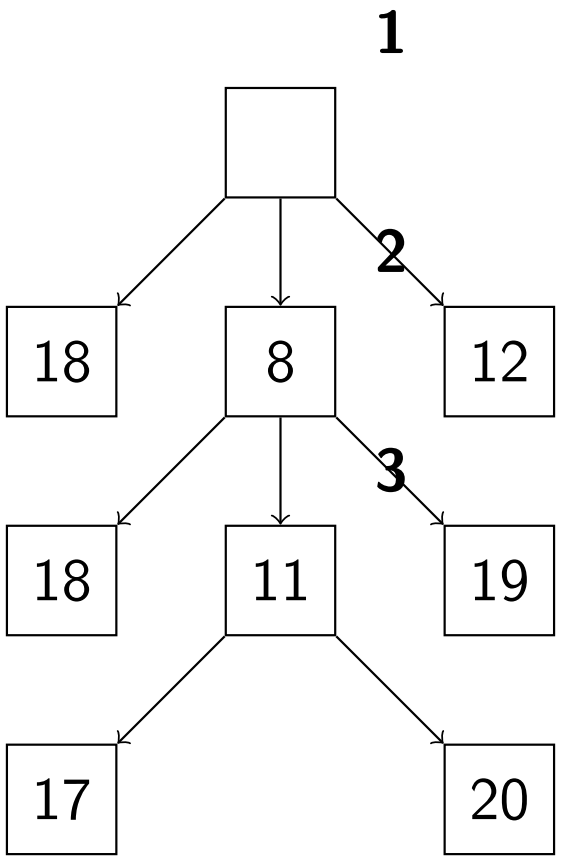
Recursive Best First - Example



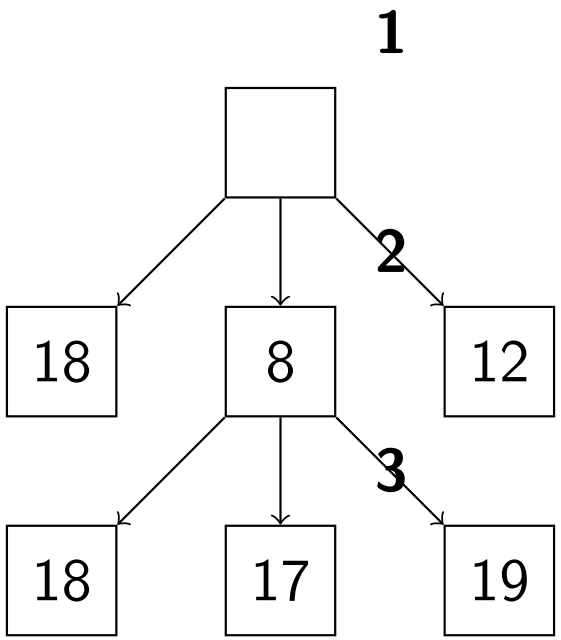
Recursive Best First - Example



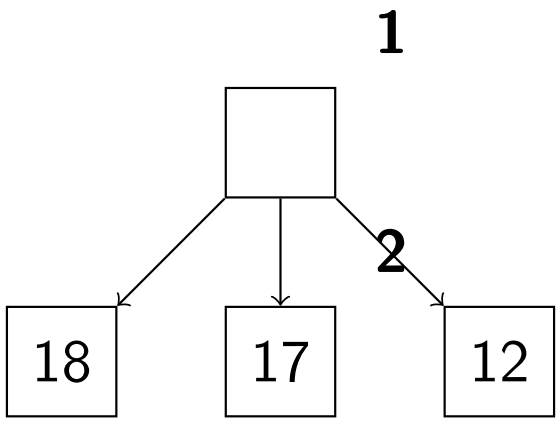
Recursive Best First - Example



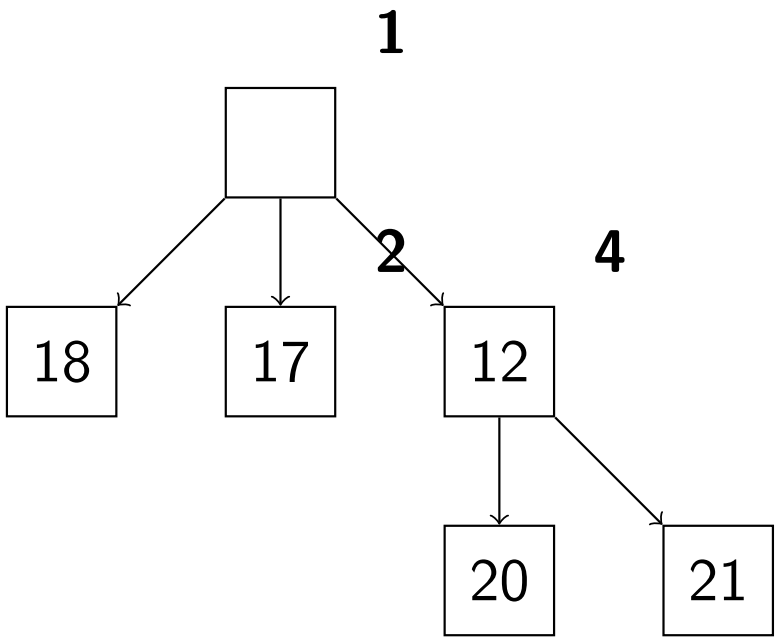
Recursive Best First - Example



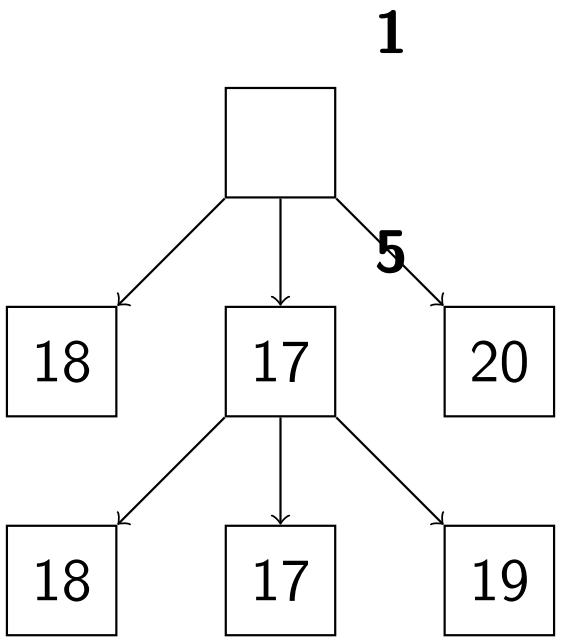
Recursive Best First - Example



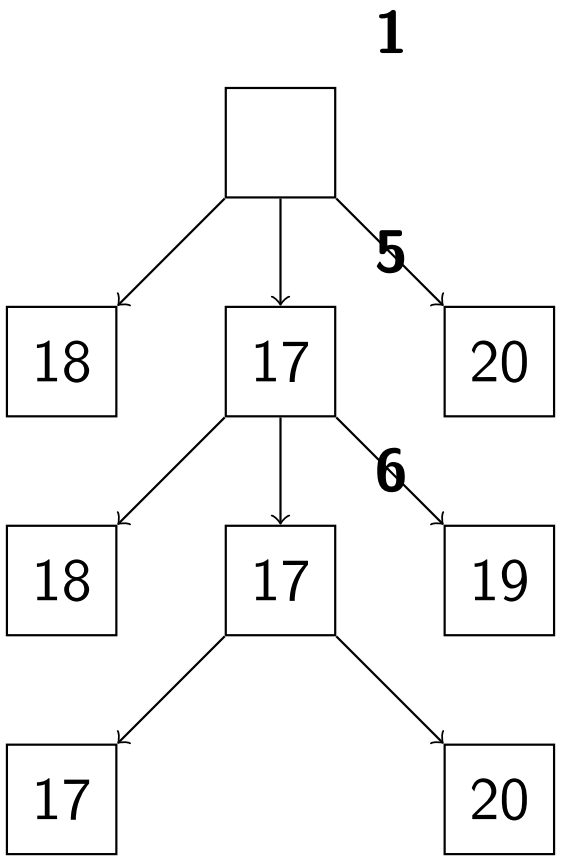
Recursive Best First - Example



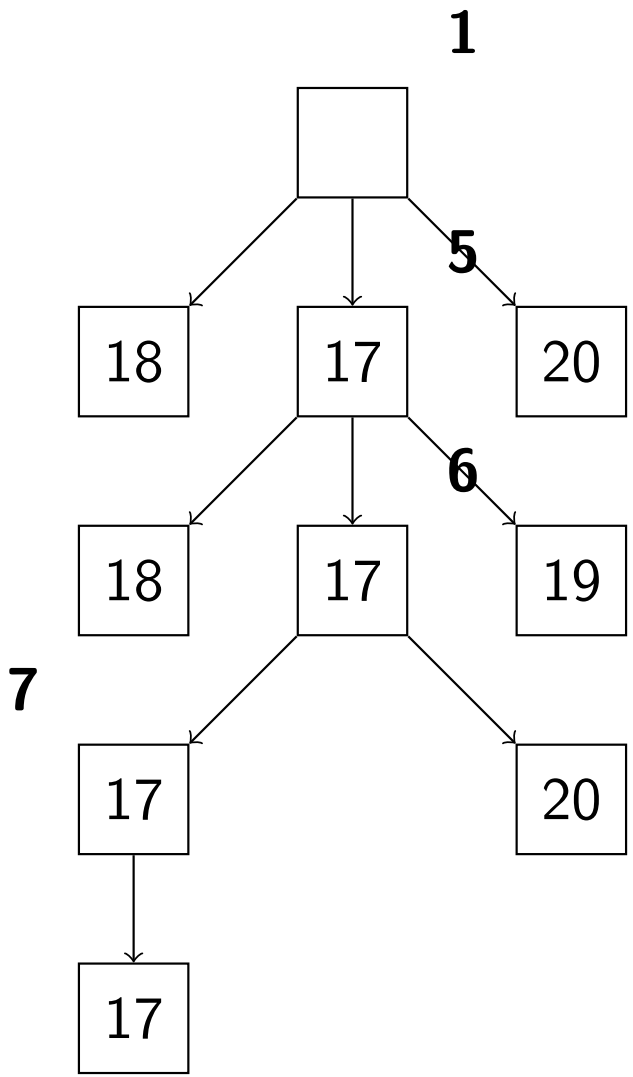
Recursive Best First - Example



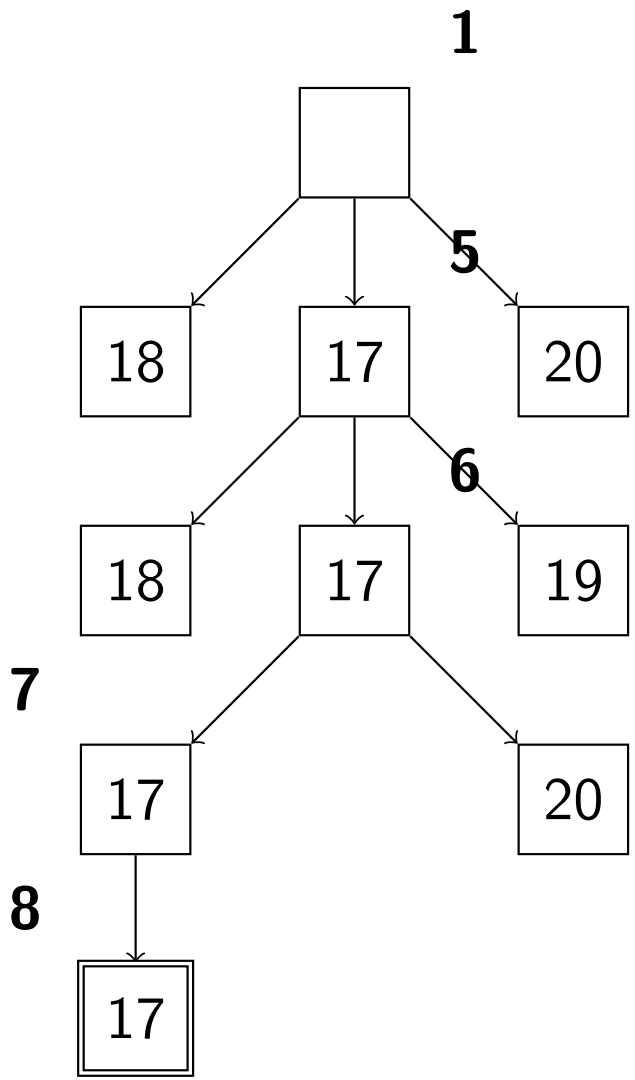
Recursive Best First - Example



Recursive Best First - Example



Recursive Best First - Example



Recursive Best First

- This algorithm usually has less re expansions than IDA*
- Memory limitation (linear space) yields lots of re expansions in certain problems
- Because of not treating duplicate nodes, path loops can elevate the temporal cost
- Solution: Relax the memory restrictions

Memory Bound A* (MA*)

- Sets a memory limit (number of nodes that can be stored, more than $O(bd)$)
- An A* search is performed storing nodes while memory is available
- When the memory is exhausted, nodes from the less promising paths are discarded storing their cost in the ascendants
- Nodes are re expanded if the forgotten nodes are better that all the current paths
- The algorithm is complete (finds a solution) if the path to the solution is shorter than the total memory