Heuristic Search

- Best First Search
  - A*
  - IDA*
  - Beam Search
- Generate and Test
- Local Searches
  - Hill Climbing
    - Simple Hill Climbing
    - Steepest Ascend Hill Climbing
    - Stochastic Hill Climbing
  - Simulated Annealing
- Local beam search
- Genetic Algorithms

State spaces

$<S,P,I,G,W>$	- state space
S	- set of states
$P= \{(x,y) \mid x,y \in S\}$	- set of production rules
$I \in S$	- initial state
$G \subseteq S$	- set of goal states
$W: P \rightarrow \mathbf{R}^+$	- weight function

Best first Search

BestFirstSearch(**state space** Σ=<S,P,I,G,W>,f)

Open $\leftarrow$ {<f(I),I>}

Closed $\leftarrow \emptyset$

while Open $\neq \emptyset$ **do**

<f_x,x> $\leftarrow$ *ExtractMin*(Open)

if *Goal*(x, Σ) **then return** x

Insert(x,Closed)

for y $\in$ *Child*(x ,Σ) **do**

if y$\notin$Closed **then**

Insert(<f(y),y>,Open)

return *fail*

Open is implemented as a *priority queue*

Best First Search

- **Example:** The Eight-puzzle
- The task is to transform the initial puzzle

2	8	3
1	6	4
7		5

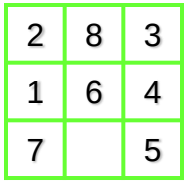
- into the goal state

1	2	3
8		4
7	6	5

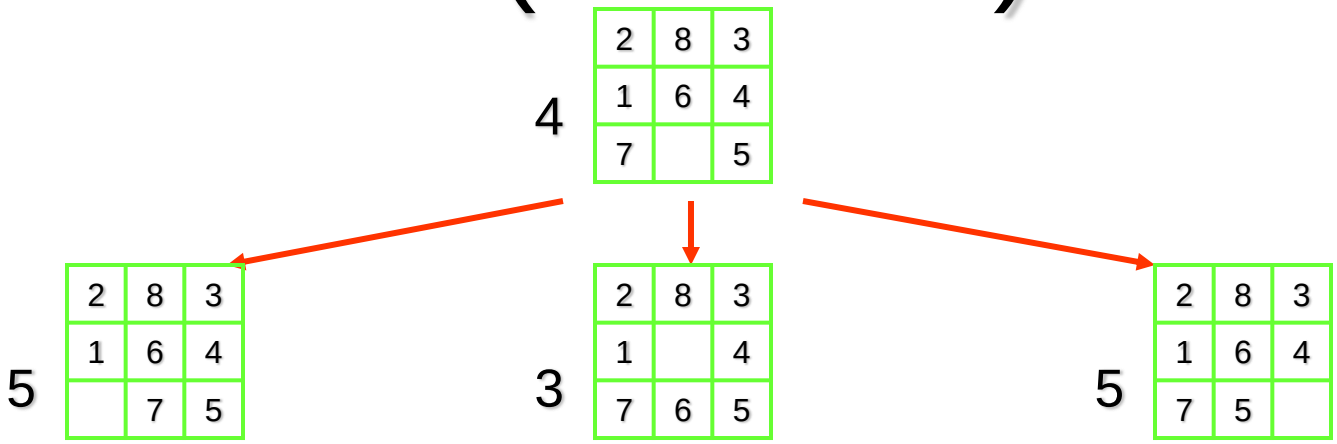
- by shifting numbers up, down, left or right.

Best-First (Heuristic) Search

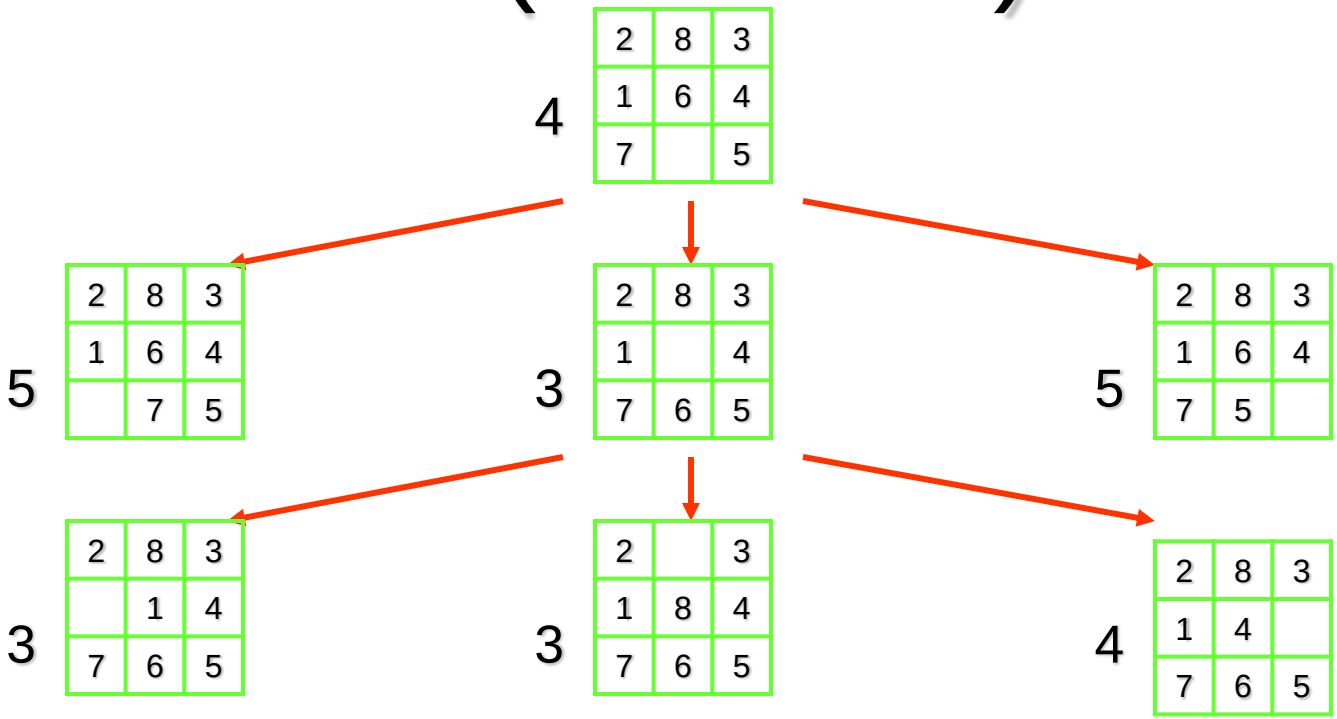
4



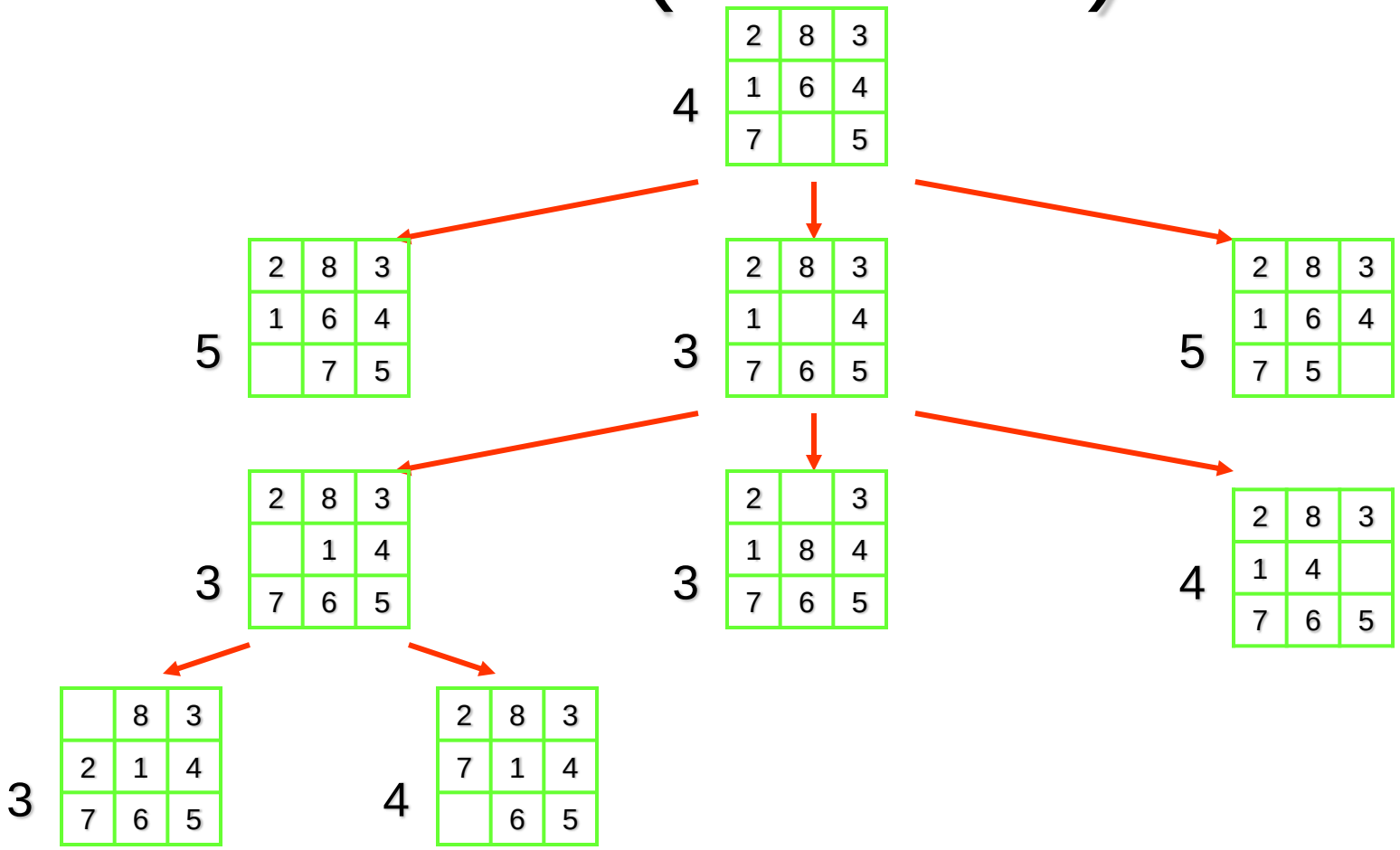
Best-First (Heuristic) Search



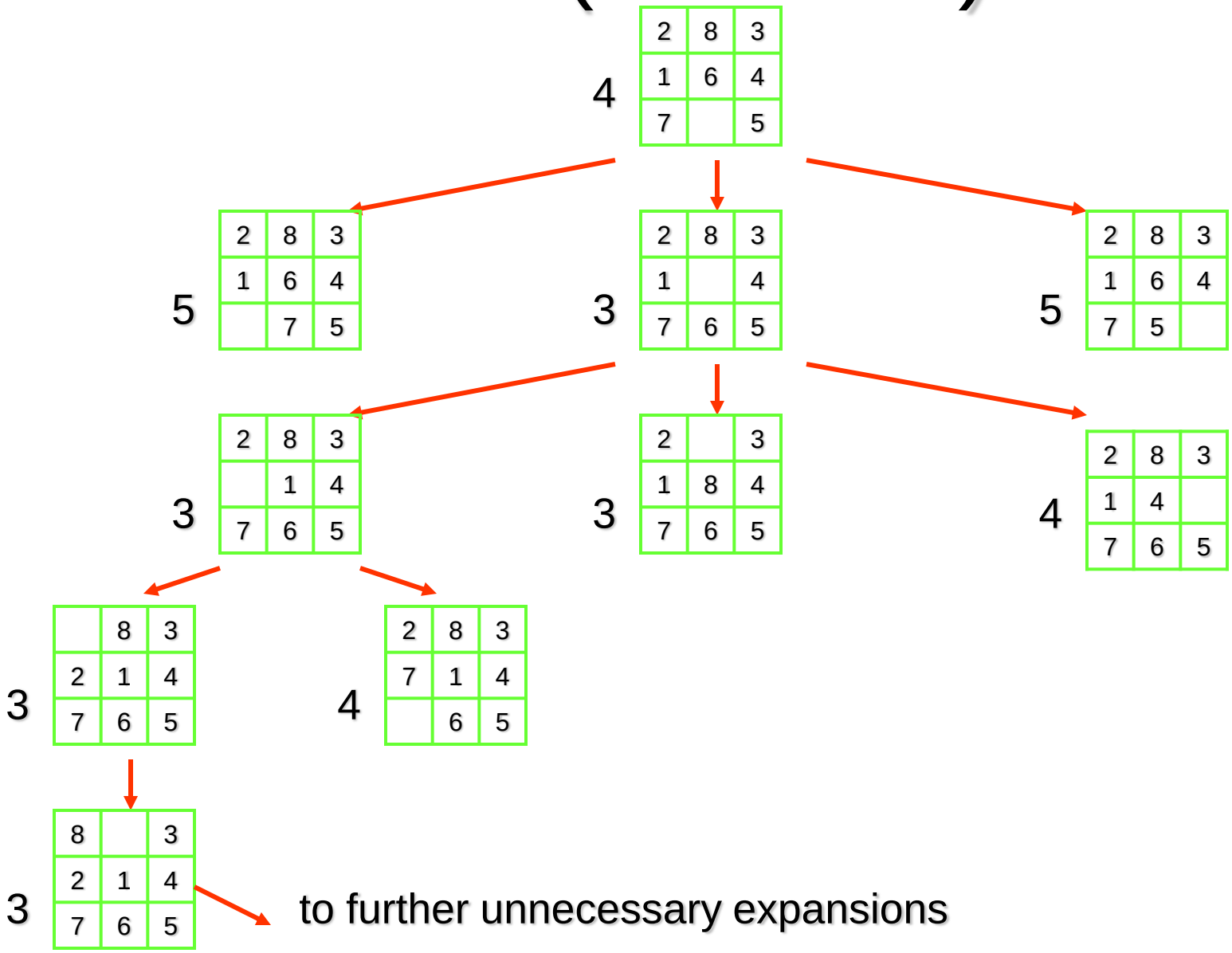
Best-First (Heuristic) Search



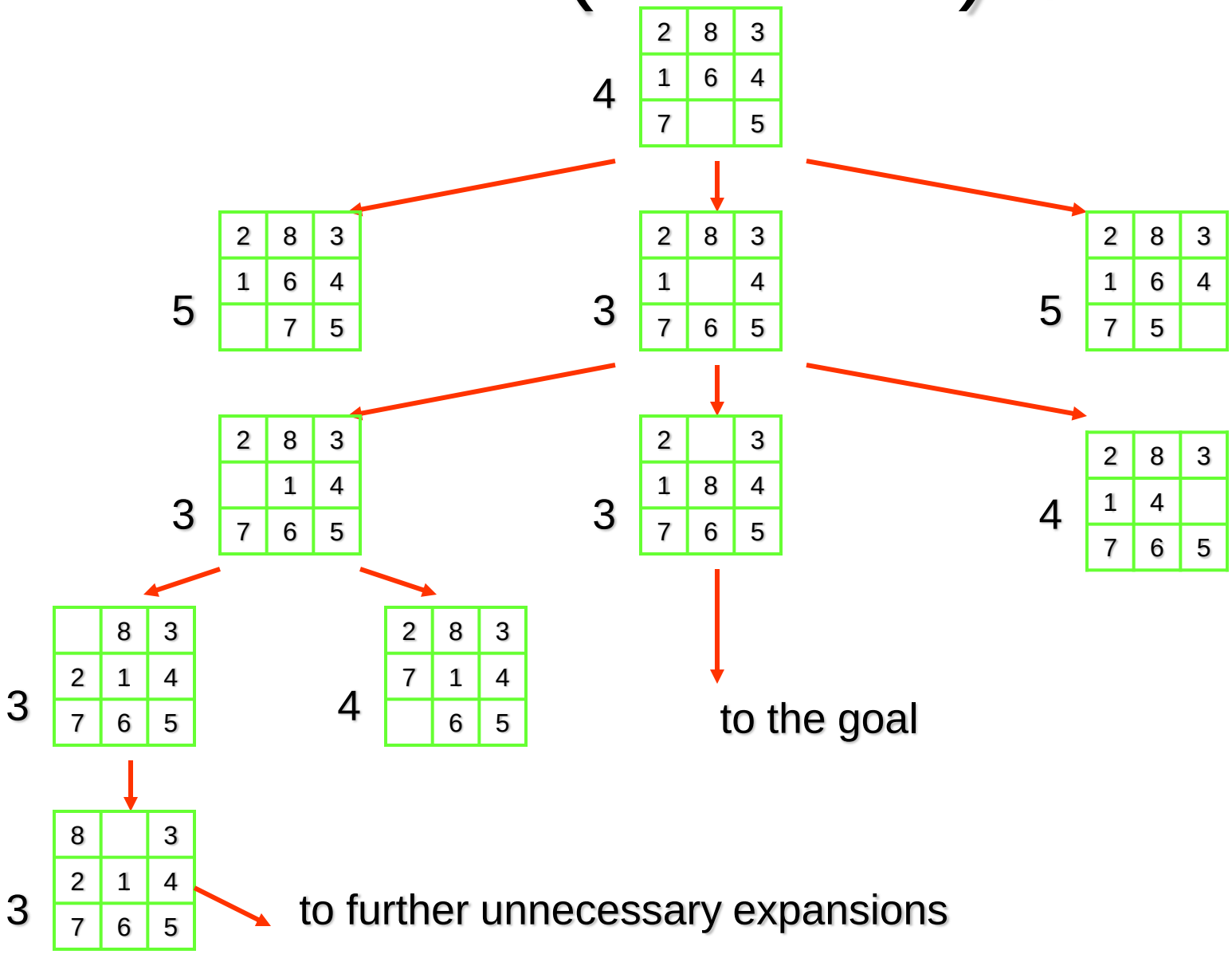
Best-First (Heuristic) Search



Best-First (Heuristic) Search



Best-First (Heuristic) Search



Best First Search - A*

<S,P,I,G,W> - state space

h*(x) - a minimum path weight from x to the goal state

h(x) - heuristic estimate of h*(x)

g(x) - a minimum path weight from I to x

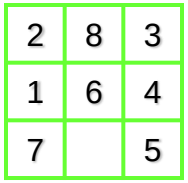
f(x) = g(x) + h(x)

Search strategies - A*

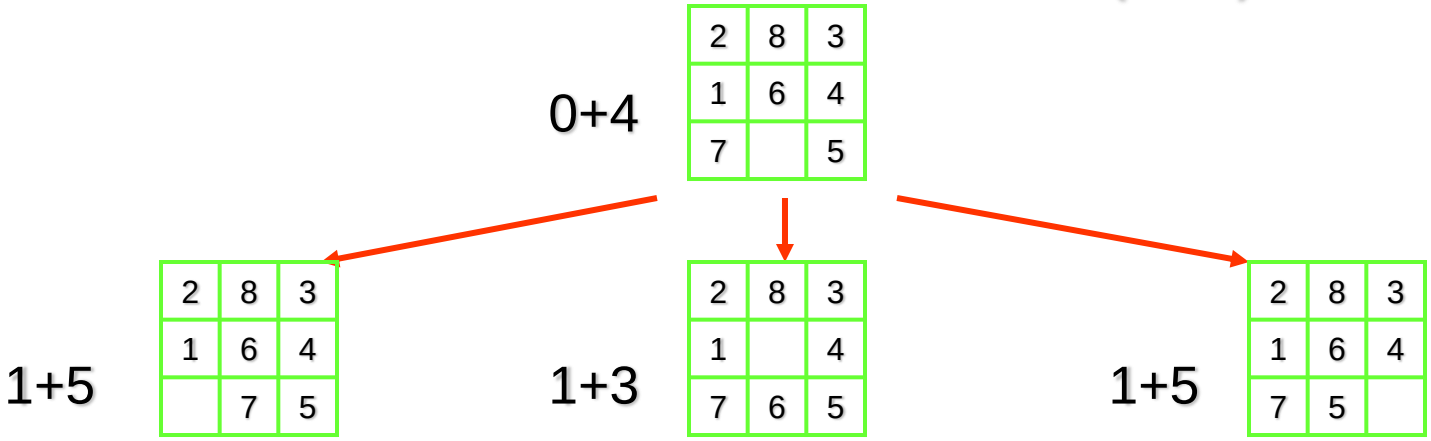
*A*Search*(**state space** Σ=<S,P,I,G,W>,*h*)
    Open $\leftarrow$ {<*h*(I),0,I>}
    Closed $\leftarrow \emptyset$
    while Open $\neq \emptyset$ **do**
        <f_x,g_x,x> $\leftarrow$ *ExtractMin*(Open) [minimum for f_x]
        if *Goal*(x, Σ) **then return** x
        Insert(<f_x,g_x,x>,Closed)
        for y $\in$ *Child*(x ,Σ) **do**
            $g_y = g_x + W(x,y)$
            $f_y = g_y + h(y)$
            if there is no <f,g,y> $\notin$ Closed with f$\leq f_y$ **then**
                Insert(< f_y,g_y,y>,Open) [replace existing <f,g,y>, if present]
    return *fail*

Best-First Search (A*)

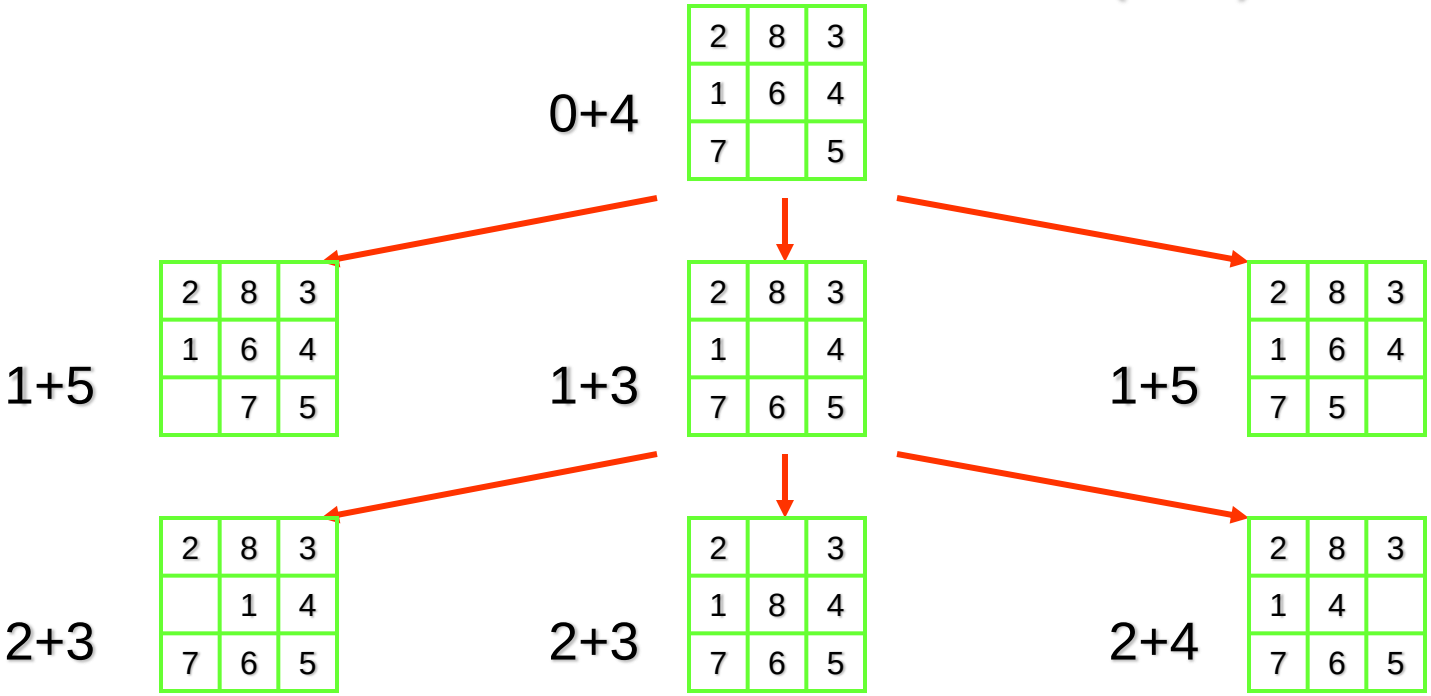
0+4



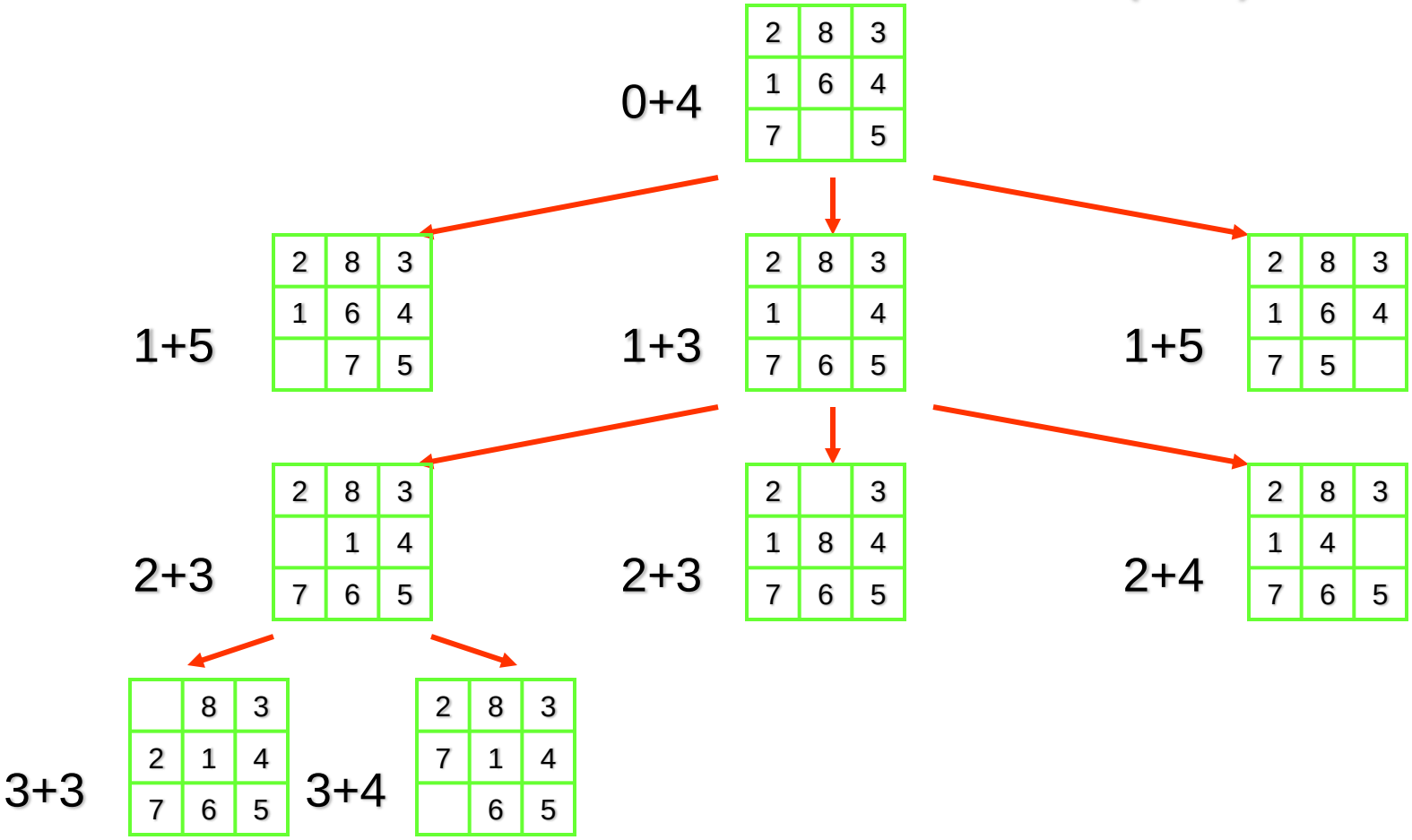
Best-First Search (A*)



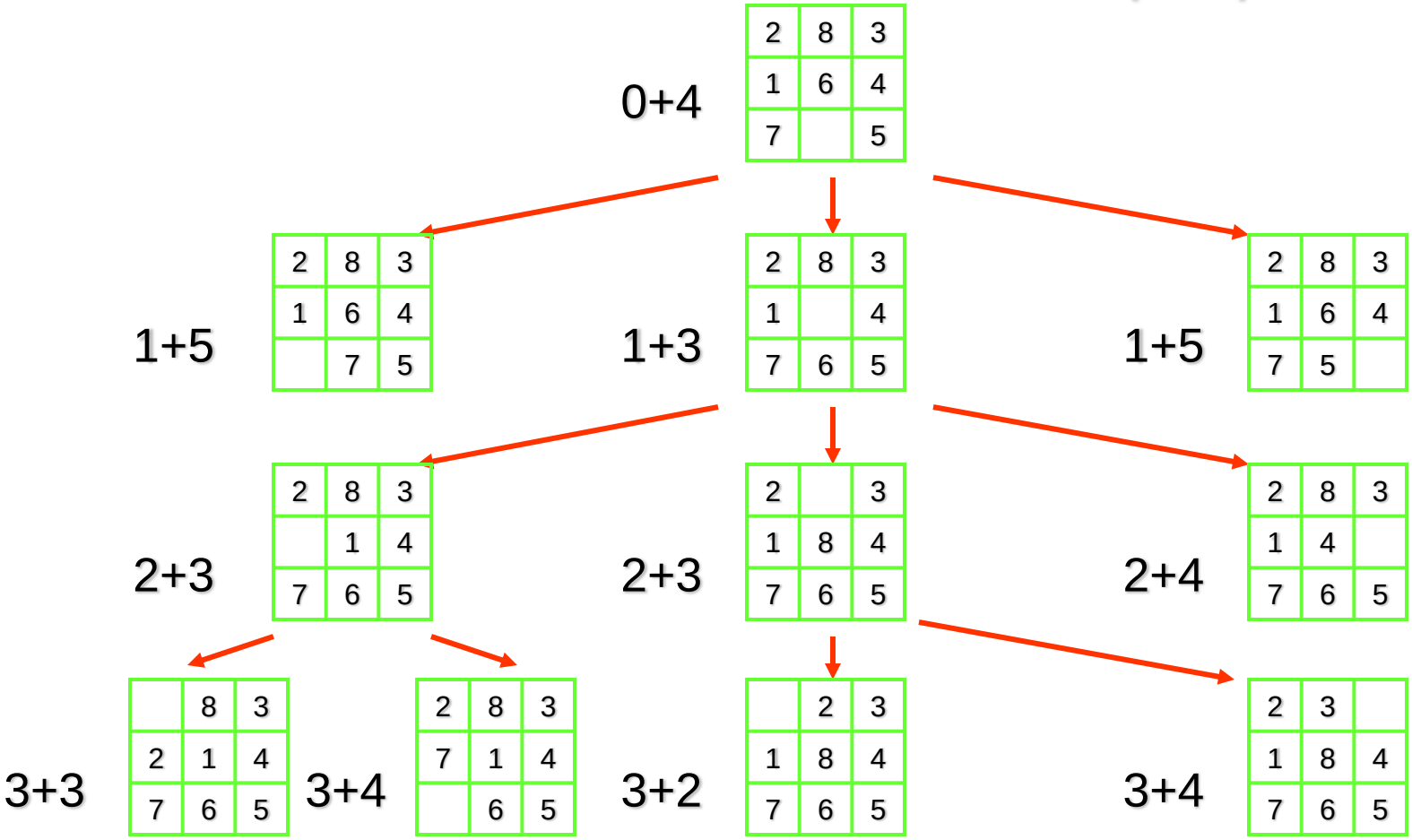
Best-First Search (A*)



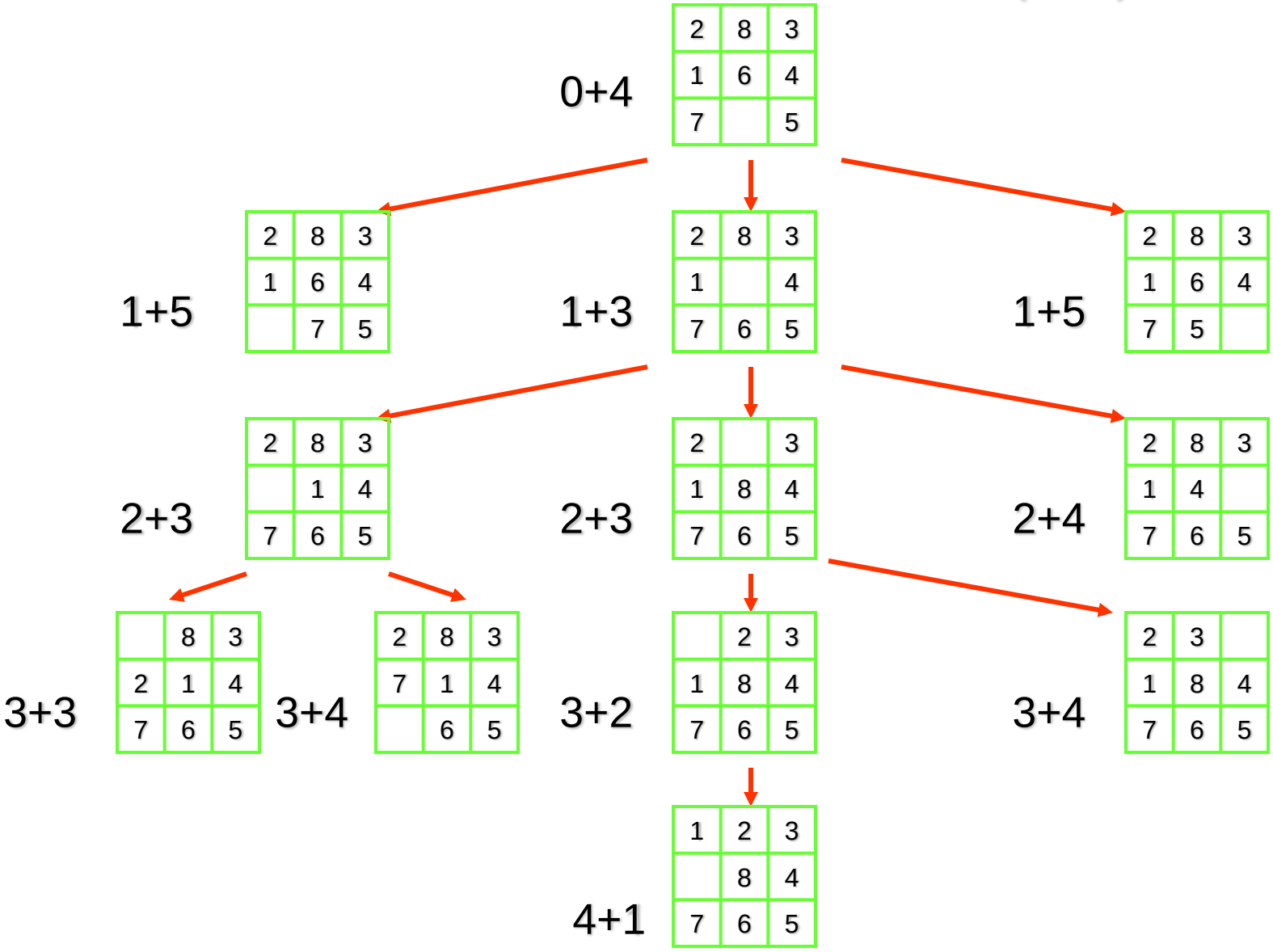
Best-First Search (A*)



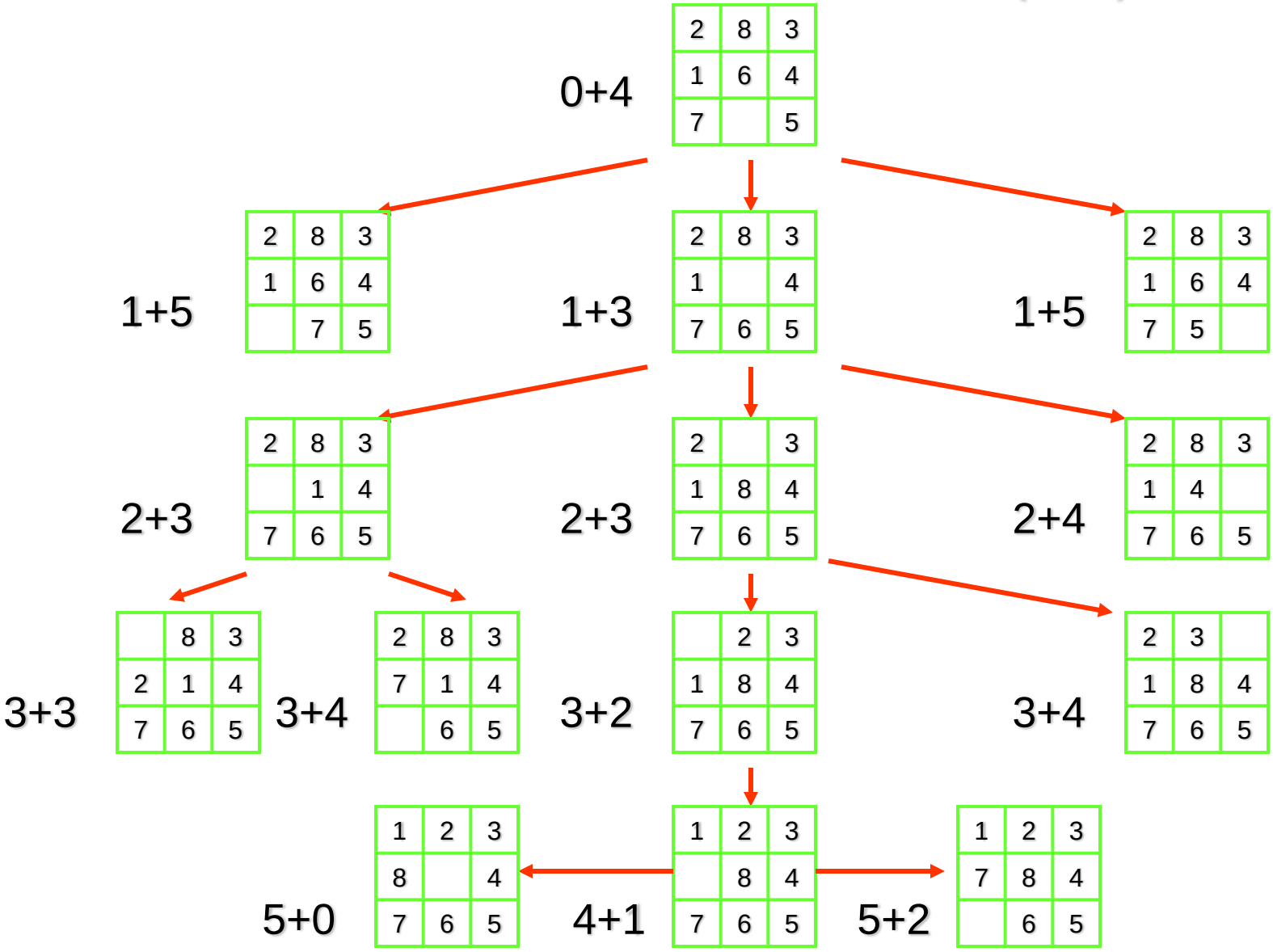
Best-First Search (A*)



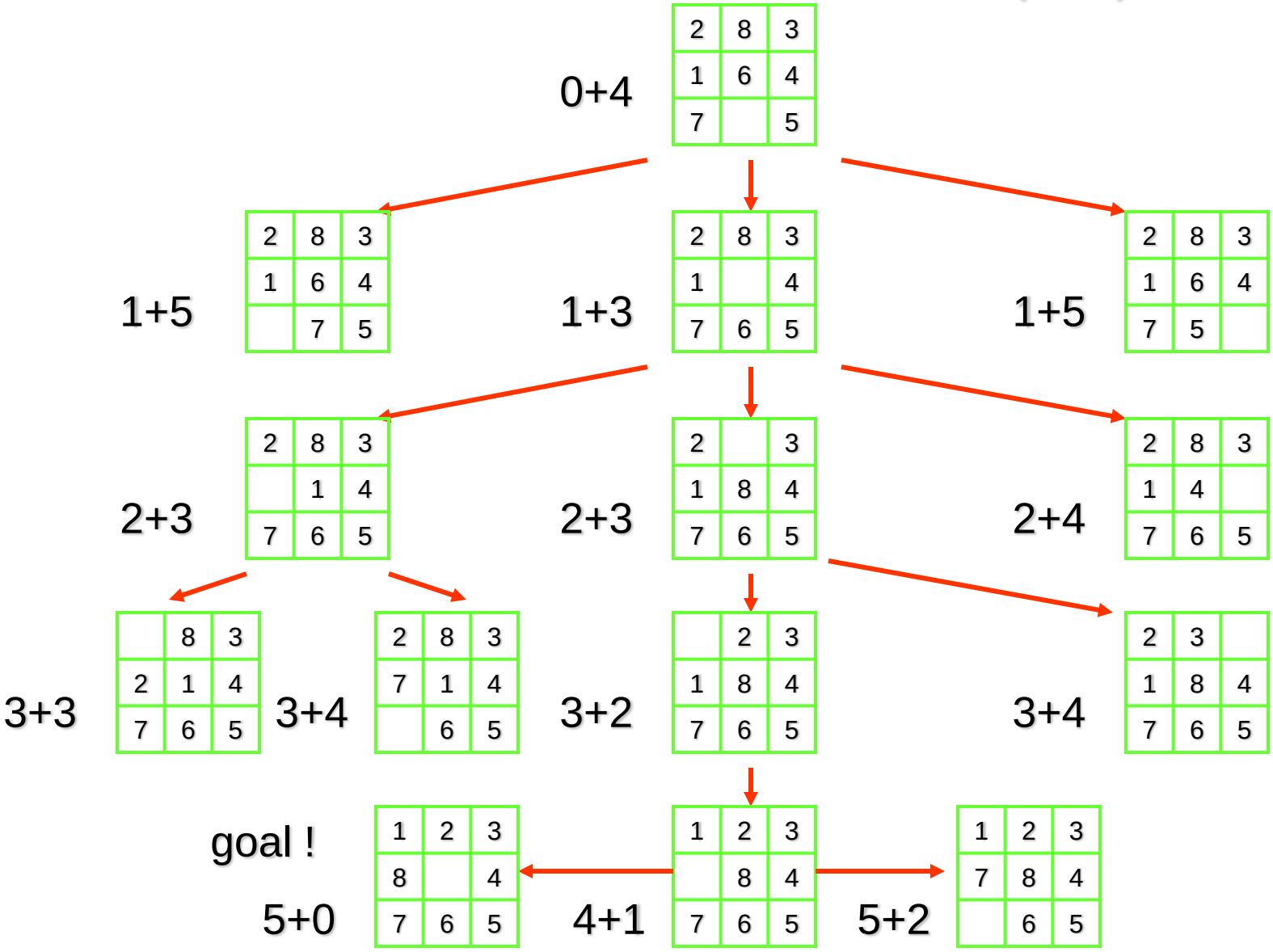
Best-First Search (A*)



Best-First Search (A*)



Best-First Search (A*)



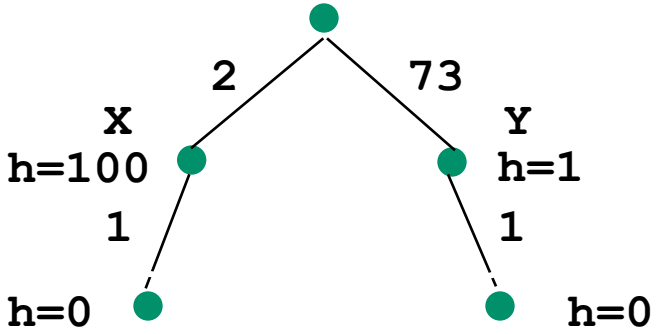
Admissible search

Definition

An algorithm is **admissible** if it is guaranteed to return an optimal solution (with minimal possible path weight from the start state to a goal state) whenever a solution exists.

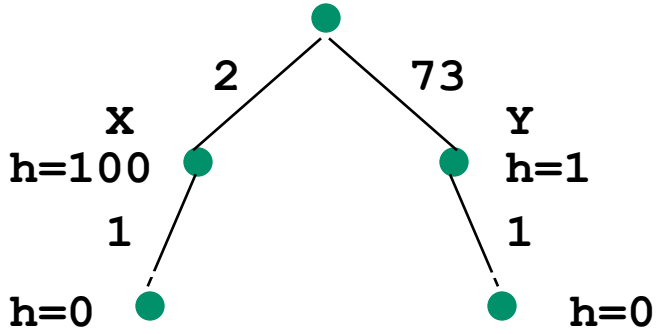
Admissibility

- What must be true about h for A* to find optimal path?



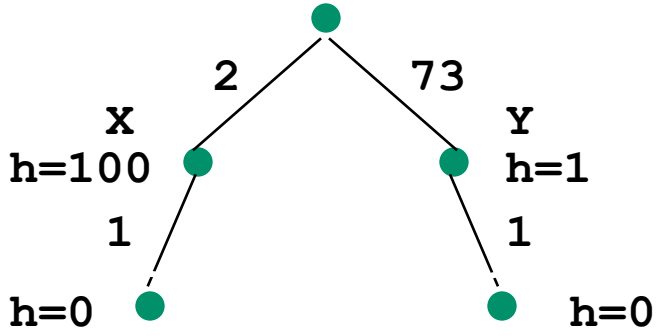
Admissibility

- What must be true about h for A* to find optimal path?
- A* finds optimal path if h is admissible; h is admissible when it never overestimates.



Admissibility

- What must be true about h for A* to find optimal path?
- A* finds optimal path if h is admissible; h is admissible when it never overestimates.
- In this example, h is not admissible.



g(X)+h(X) = 102

g(Y)+h(Y) = 74

Optimal path is not found!

Admissibility of a Heuristic Function

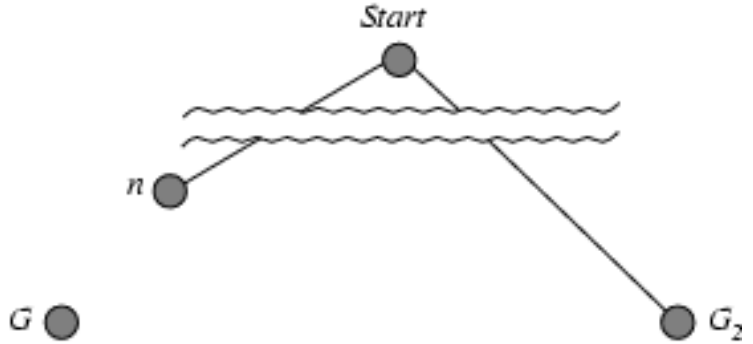
Definition

A heuristic function *h* is said to be **admissible** if

$0 \leq h(n) \leq h^*(n)$ for all $n \in S$.

Optimality of A* (proof)

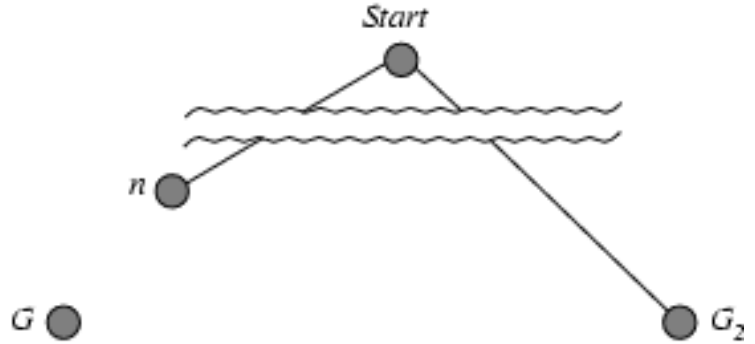
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $f(G) = g(G)$ since $h(G) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G_2) > f(G)$ from above

Optimality of A* (proof)

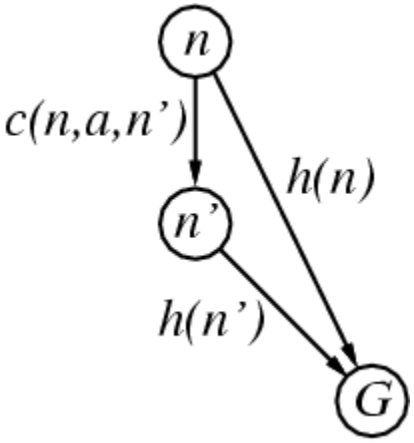
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.



- $f(G_2) > f(G)$ from above
- $h(n) \leq h^*(n)$ since h is admissible
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$, Hence $f(G_2) > f(n)$,
- and A^* will never select G_2 for expansion

Consistent heuristics

- A heuristic is consistent if for every node *n*, every successor *n′* of *n* generated by any action *a*,
- *h(n) ≤ c(n,a,n′) + h(n′)*



If *h* is consistent, we have

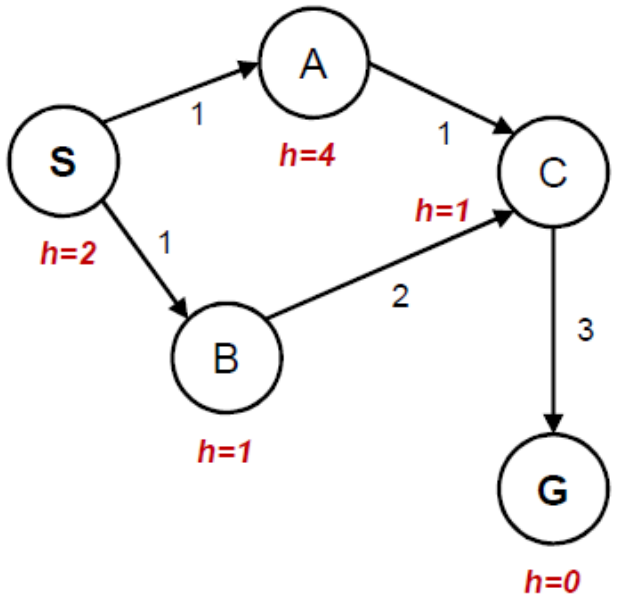
f(n') = g(n') + h(n')
= g(n) + c(n,a,n') + h(n')
≥ g(n) + h(n)
≥ f(n)

- i.e., *f(n)* is non-decreasing along any path.

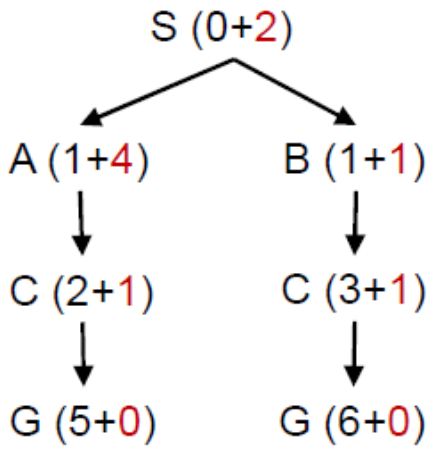
Consistent heuristics

A* Graph Search Gone Wrong?

State space graph



Search tree



Extreme Cases

- If we set h=0, then A* is Breadth First search; h=0 is admissible heuristic (when all costs are non-negative).
- If g=0 for all nodes, A* is similar to Steepest Ascend Hill Climbing
- If both g and h =0, it can be as dfs or bfs depending on the structure of Open

How to chose a heuristic?

- Original problem **P** / Relaxed problem **P'**

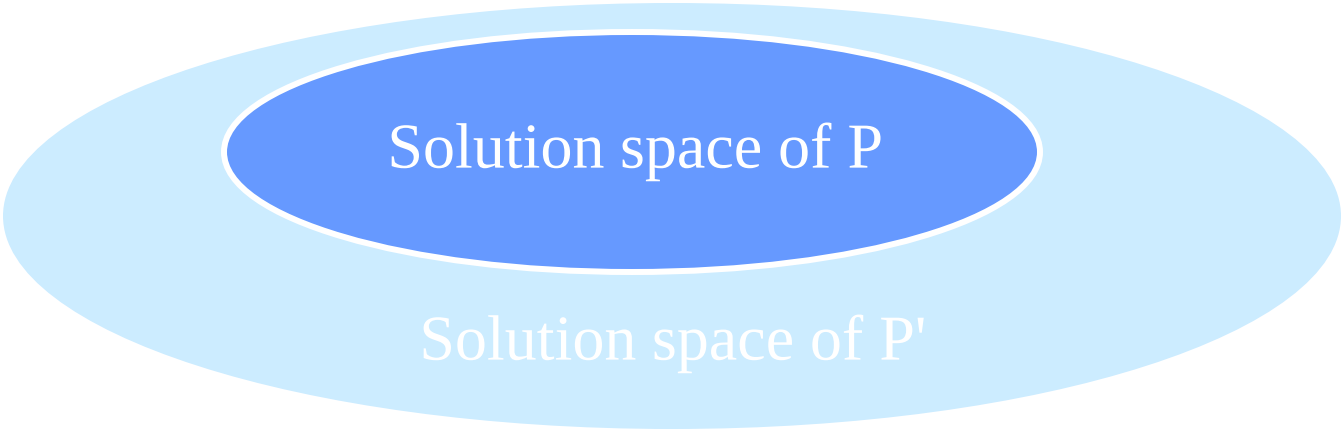
  A set of constraints — removing one or more constraints

  P is complex — **P'** becomes simpler

- Use cost of a best solution path from n in **P'** as h(n) for **P**
- **Admissibility:**

  h^* h

  cost of best solution in **P** >= cost of best solution in **P'**

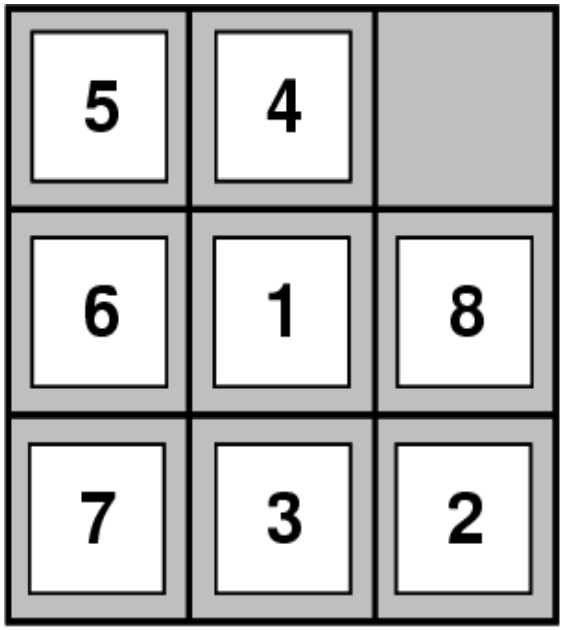


How to chose a heuristic - 8-puzzle

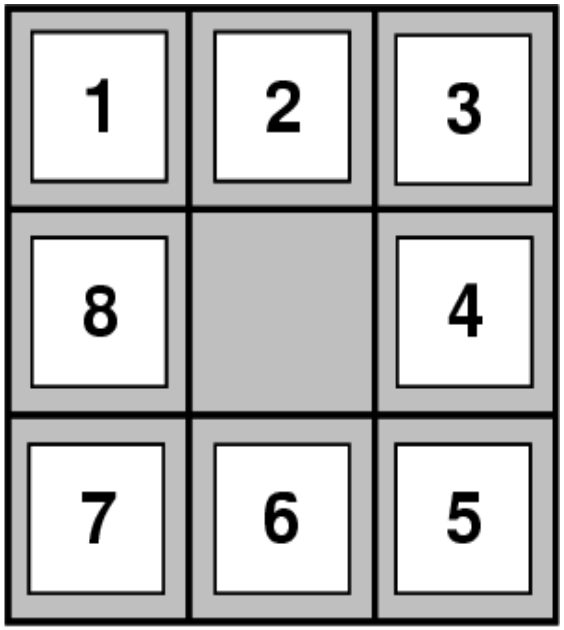
- Example: 8-puzzle
  - Constraints: to move from cell A to cell B
    cond1: there is a tile on A
    cond2: cell B is empty
    cond3: A and B are adjacent (horizontally or vertically)
  - Removing cond2:
    h2 (sum of Manhattan distances of all misplaced tiles)
  - Removing both cond2 and cond3:
    h1 (# of misplaced tiles)

  Here $h2 \geq h1$

How to chose a heuristic - 8-puzzle



Start State



Goal State

h1(start) = 7
h2(start) = 18

Performance comparison

	Search Cost (nodes)			Effective Branching Factor		
d	IDS	A* (h_1)	A* (h_2)	IDS	A* (h_1)	A* (h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	-	539	113	-	1.44	1.23
16	-	1301	211	-	1.45	1.25
18	-	3056	363	-	1.46	1.26
20	-	7276	676	-	1.47	1.27
22	-	18094	1219	-	1.48	1.28
24	-	39135	1641	-	1.48	1.26

Data are averaged over 100 instances of 8-puzzle, for varous solution lengths. Note: there are still better heuristics for the 8-puzzle.

Comparing heuristic functions

- Bad estimates of the remaining distance can cause extra work!
- Given two algorithms A_1 and A_2 with admissible heuristics h_1 and $h_2 < h^*(n)$,
- if $h_1(n) < h_2(n)$ for all non-goal nodes n, then A_1 expands at least as many nodes as A_2
- h_2 is said to be more **informed** than h_1 or, A_2 **dominates** A_1

Weighted A* (WA*)

$f_w(n) = (1–w)g(n) + w\,h(n)$

w = 0 - Breadth First
w = 1/2 - A*
w = 1 - Best First

IDA*: Iterative deepening A*

- To reduce the memory requirements at the expense of some additional computation time, combine uninformed iterative deepening search with A*
- Use an *f-cost* limit instead of a depth limit

Iterative Deepening A*

- In the first iteration, we determine a **"f-cost limit"** $f'(n_0) = g'(n_0) + h'(n_0) = h'(n_0)$, where n_0 is the start node.
- We expand nodes using the **depth-first algorithm** and backtrack whenever f'(n) for an expanded node n exceeds the cut-off value.
- If this search does not succeed, determine the **lowest f'-value** among the nodes that were visited but not expanded.
- Use this f'-value as the **new limit value** and do another depth-first search.
- Repeat this procedure until a goal node is found.

IDA* Algorithm - Top level

function IDA*(*problem*) **returns** *solution*

root := Make-Node(Initial-State[*problem*])

f-limit := f-Cost(*root*)

loop do

solution, *f-limit* := DFS-Contour*(root*, *f-limit)*

if *solution* is not null, **then return** *solution*

if *f-limit = infinity*, **then return** *failure*

end

IDA* contour expansion

function DFS-Countour(*node*,*f-limit*) **returns**

solution sequence and new *f-limit*

if f-Cost[*node*] > *f-limit* **then return** *(null***,** f-Cost[*node*] **)**

if Goal-Test[*problem*](State[*node*]) **then return** *(node,f-limit)*

for each node *s* **in** Successor(*node*) **do**

solution, *new-f :=* DFS-Contour*(s, f-limit)*

if *solution* is not null, **then return** *(solution, f-limit)*

next-f := Min(next-f, new-f); **end**

return *(null, next-f)*

IDA*: Iteration 1, Limit = 4

0+4

2	8	3
1	6	4
7		5

IDA*: Iteration 1, Limit = 4

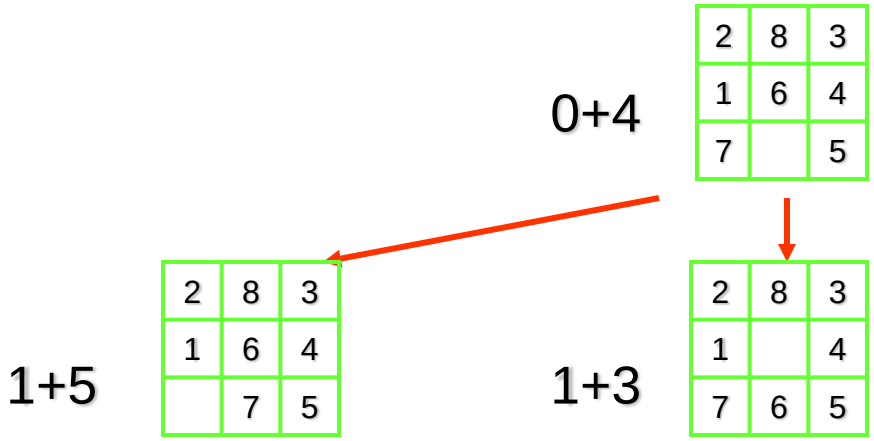
0+4

2	8	3
1	6	4
7		5

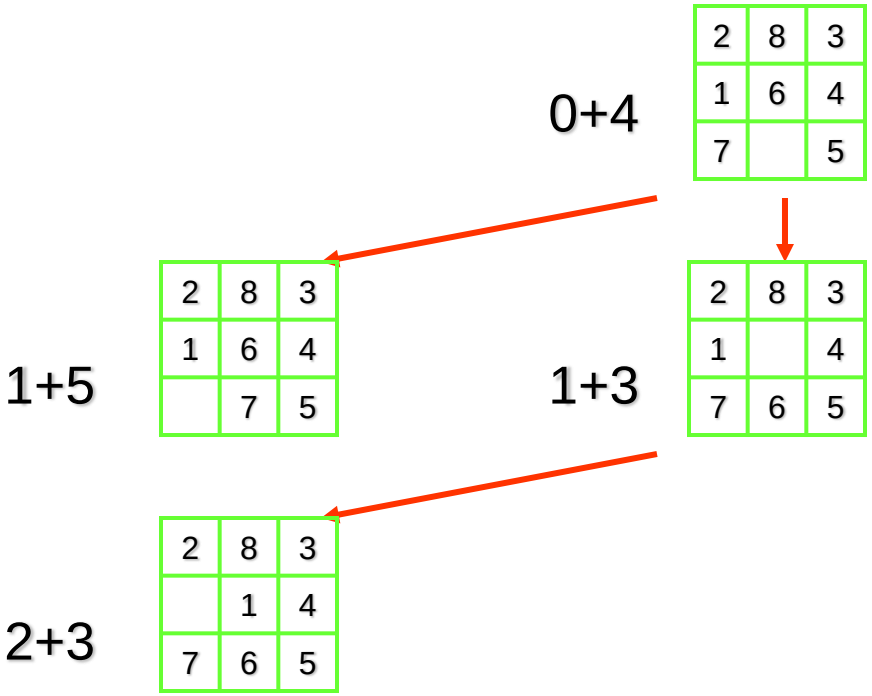
1+5

2	8	3
1	6	4
	7	5

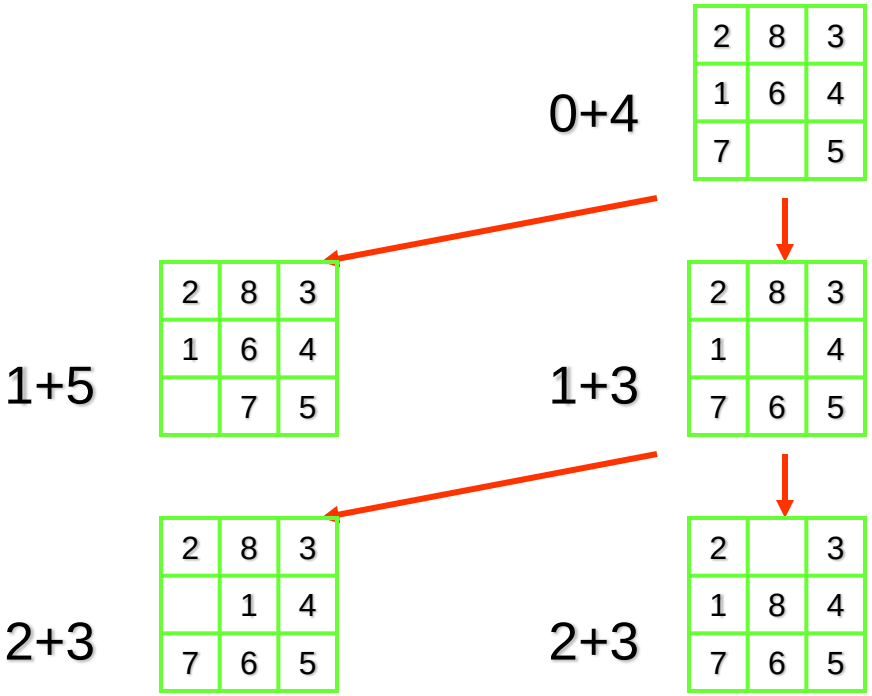
IDA*: Iteration 1, Limit = 4



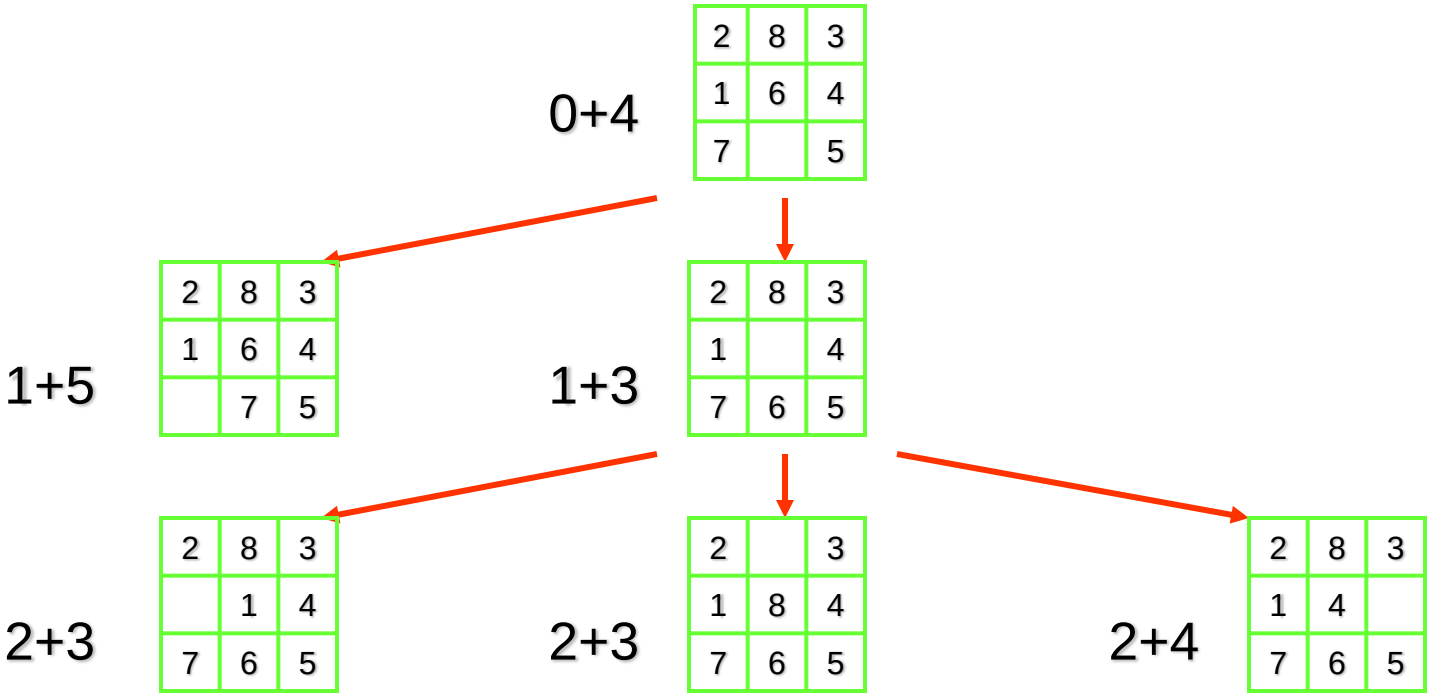
IDA*: Iteration 1, Limit = 4



IDA*: Iteration 1, Limit = 4



IDA*: Iteration 1, Limit = 4



IDA*: Iteration 2, Limit = 5

0+4

2	8	3
1	6	4
7		5

IDA*: Iteration 2, Limit = 5

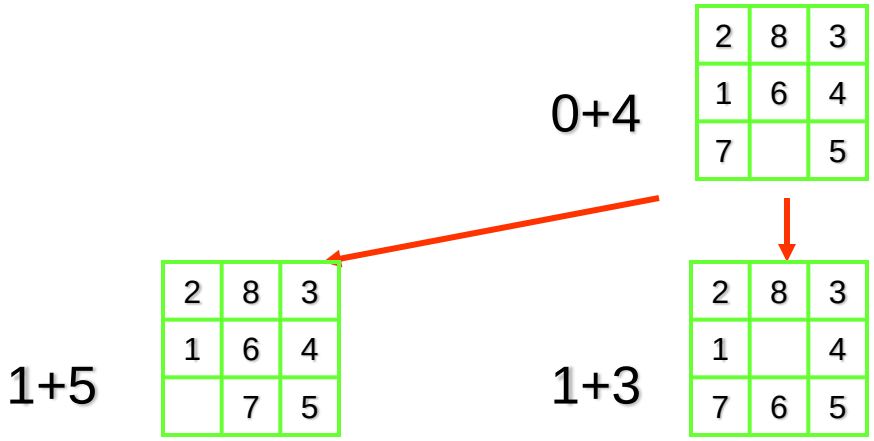
0+4

2	8	3
1	6	4
7		5

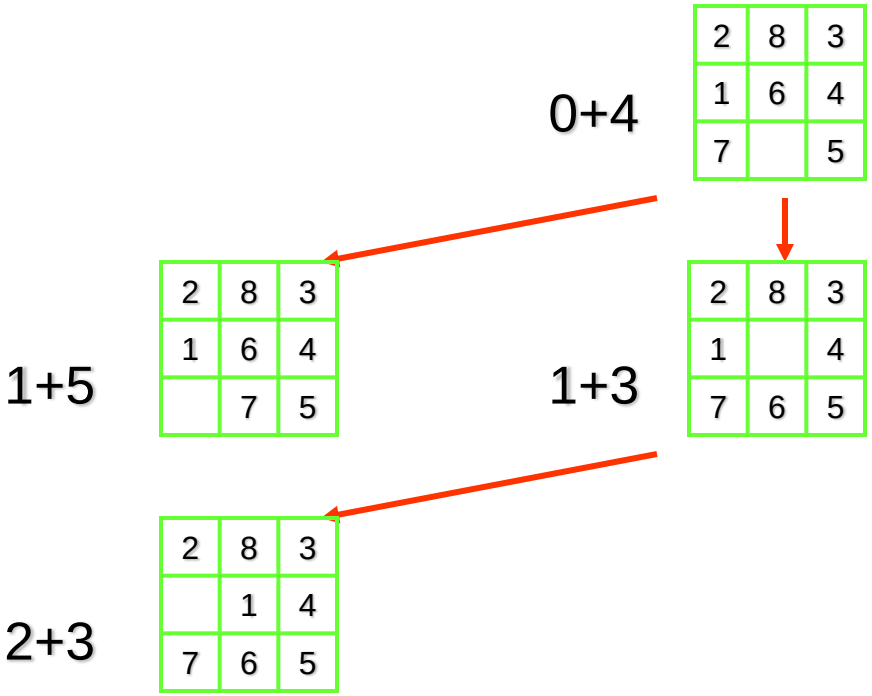
1+5

2	8	3
1	6	4
	7	5

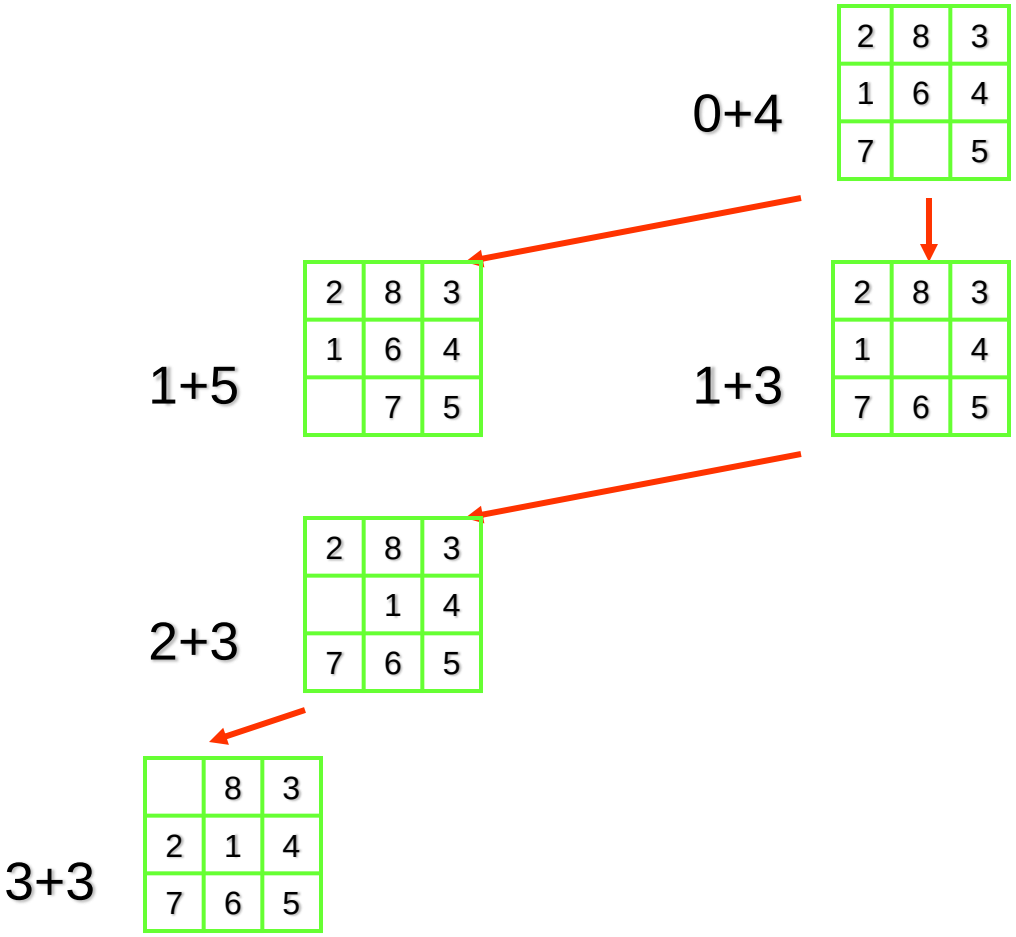
IDA*: Iteration 2, Limit = 5



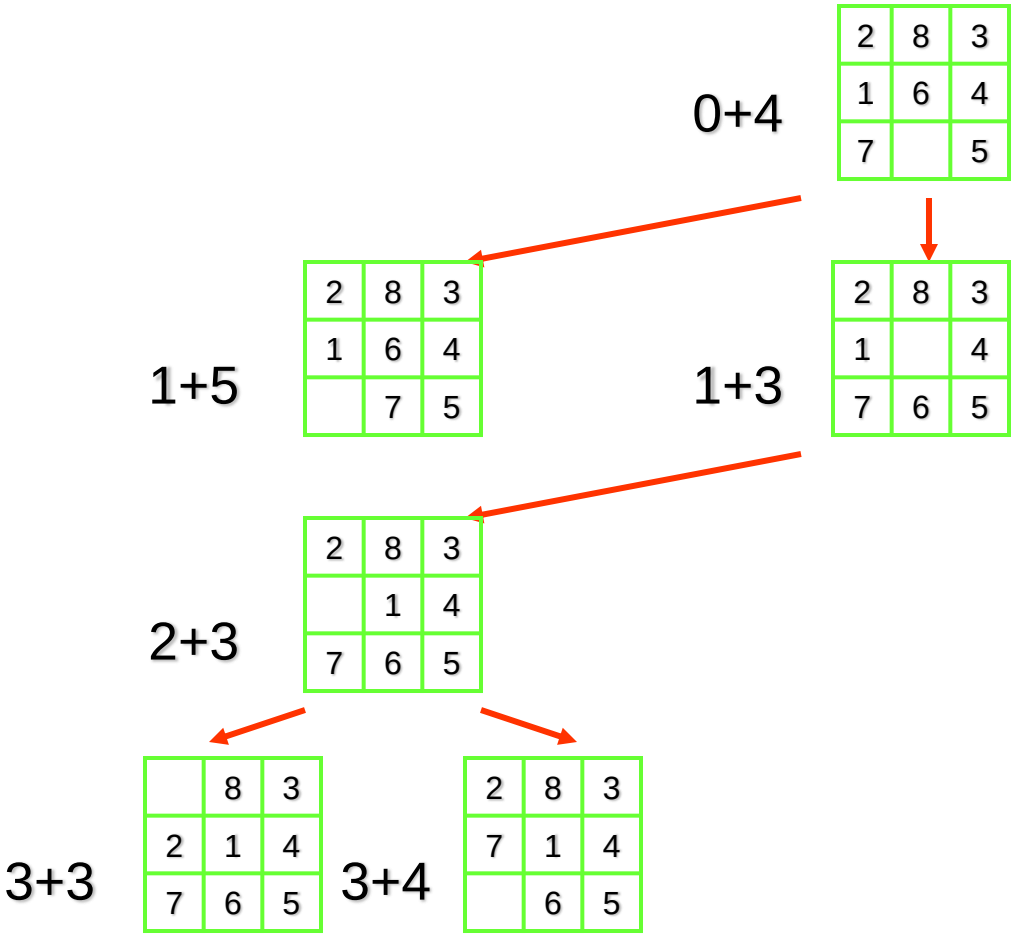
IDA*: Iteration 2, Limit = 5



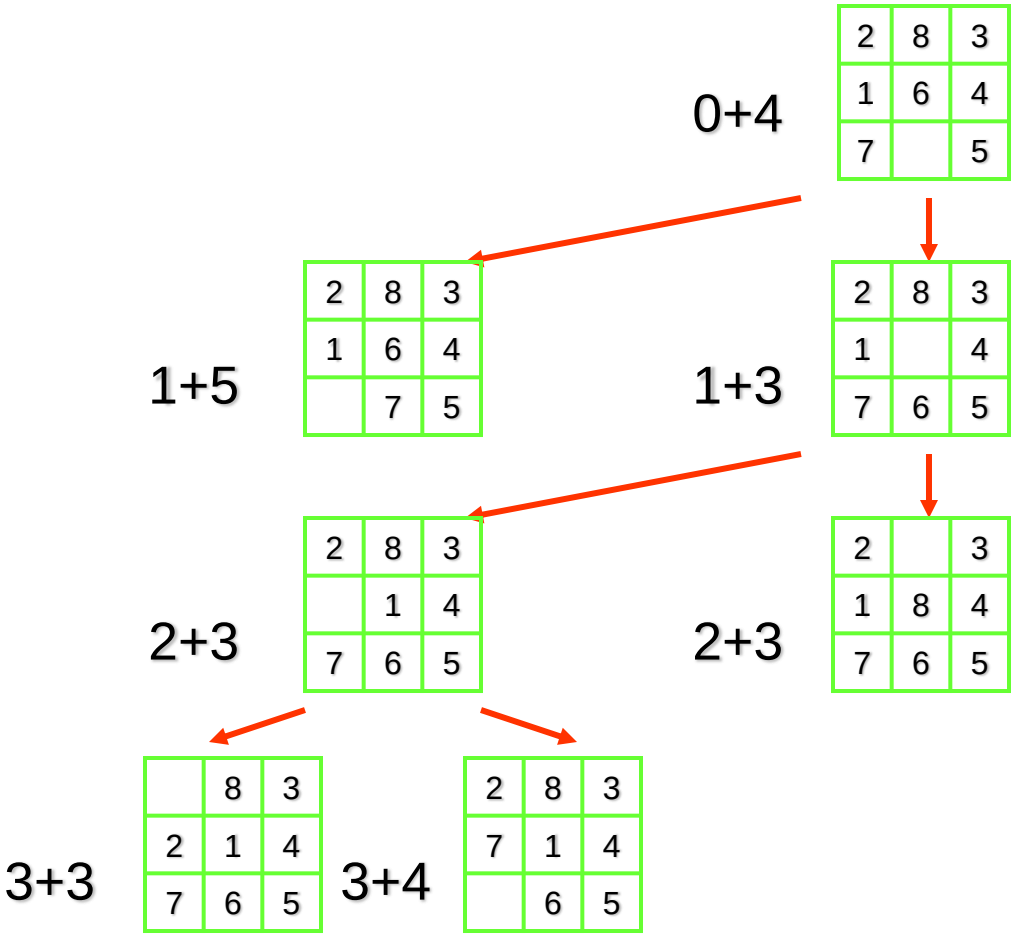
IDA*: Iteration 2, Limit = 5



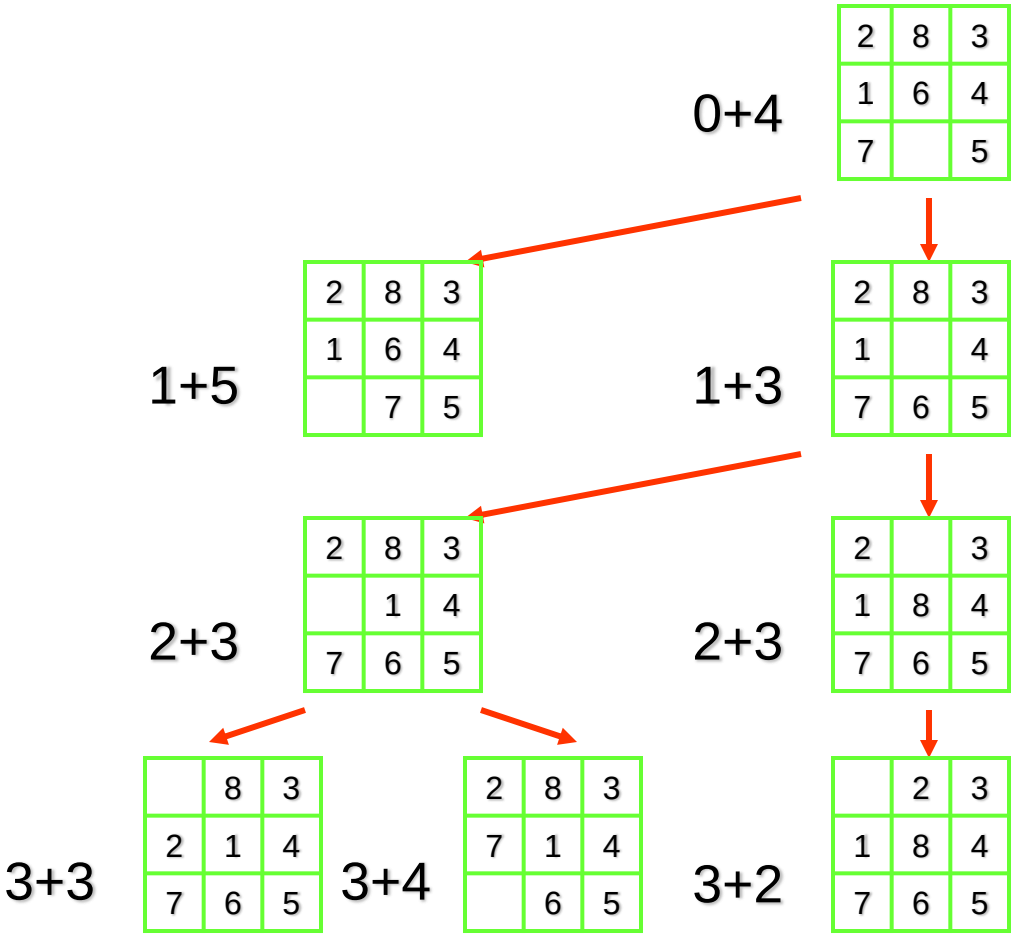
IDA*: Iteration 2, Limit = 5



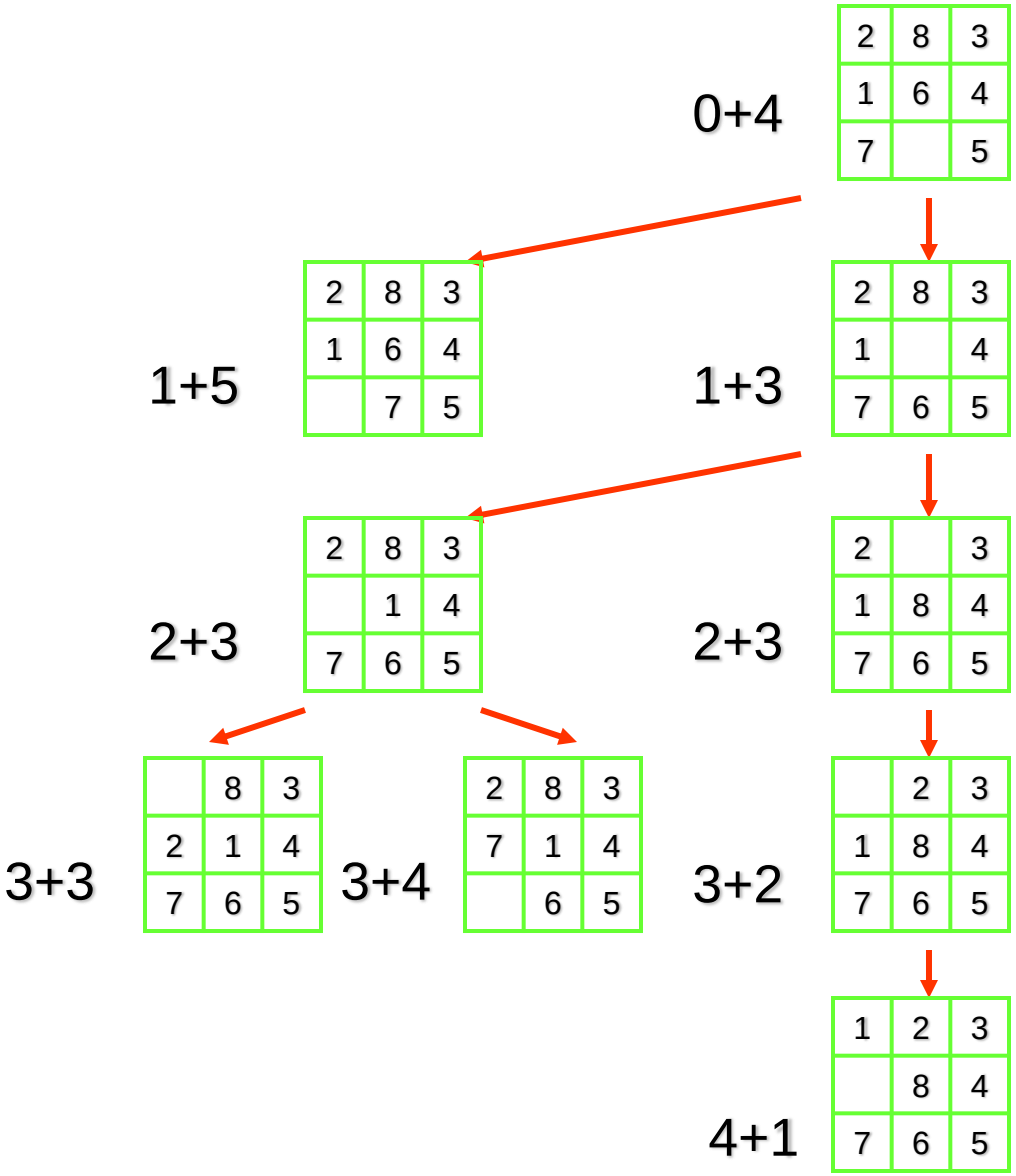
IDA*: Iteration 2, Limit = 5



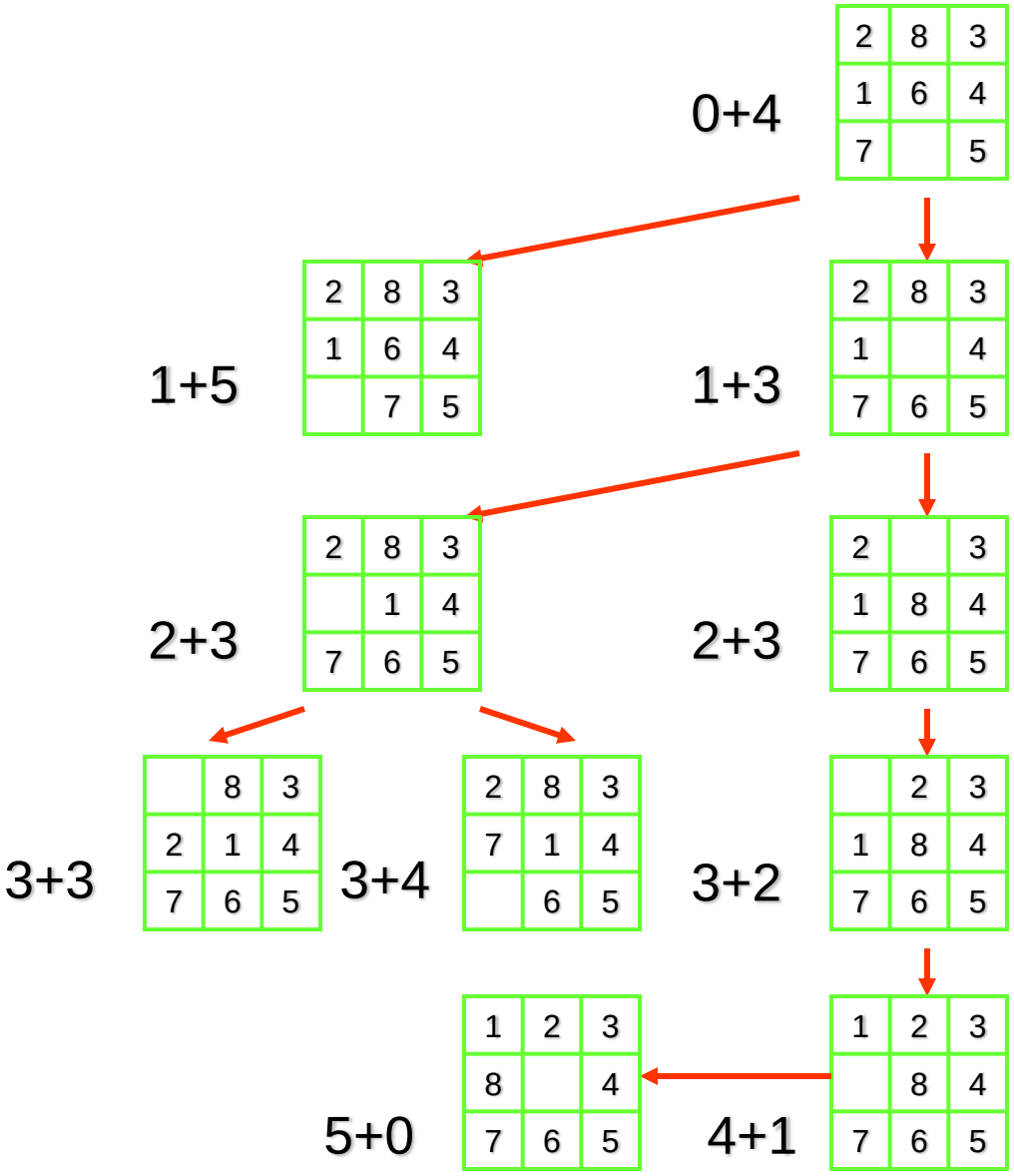
IDA*: Iteration 2, Limit = 5



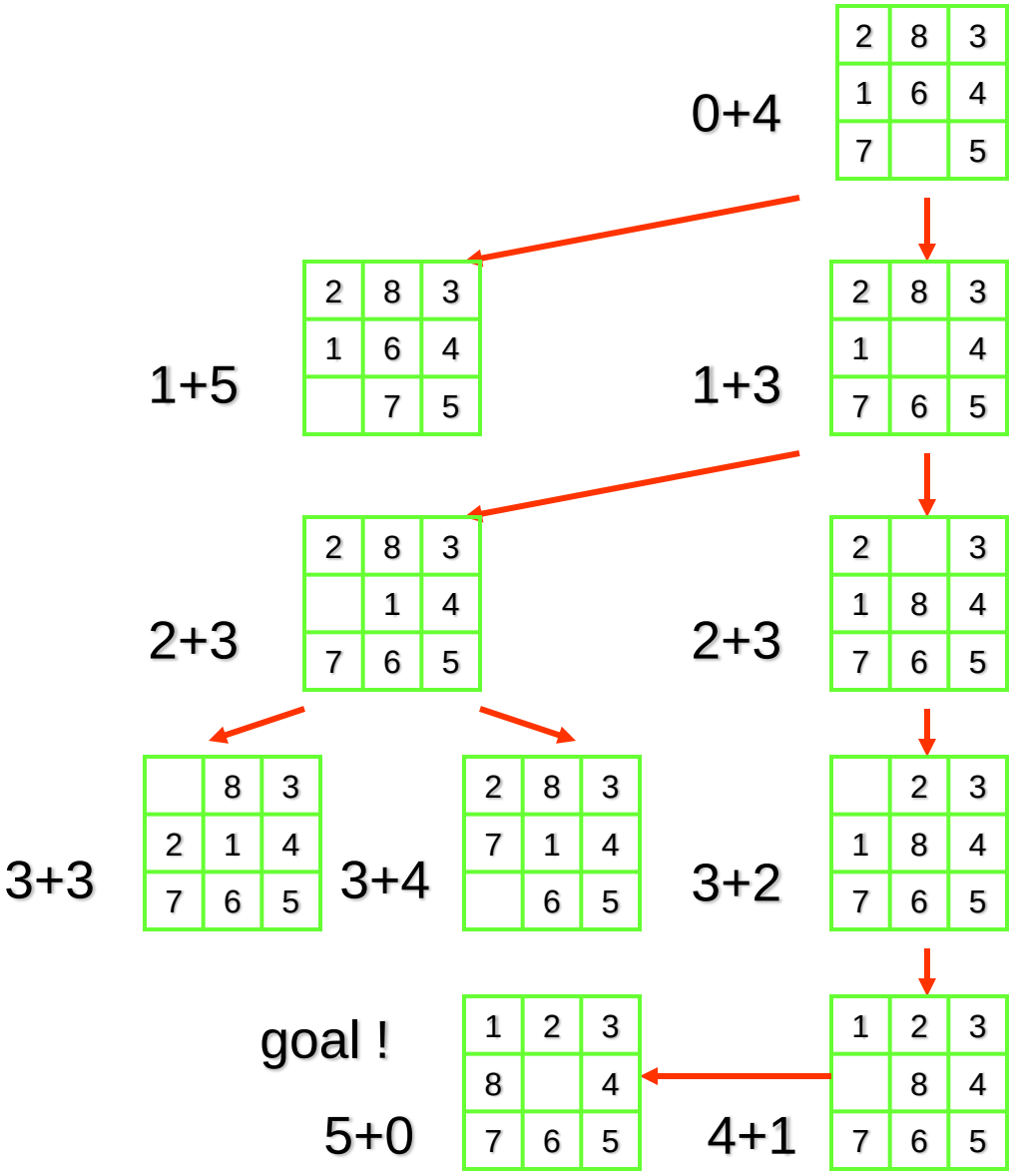
IDA*: Iteration 2, Limit = 5



IDA*: Iteration 2, Limit = 5

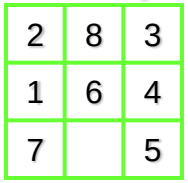


IDA*: Iteration 2, Limit = 5

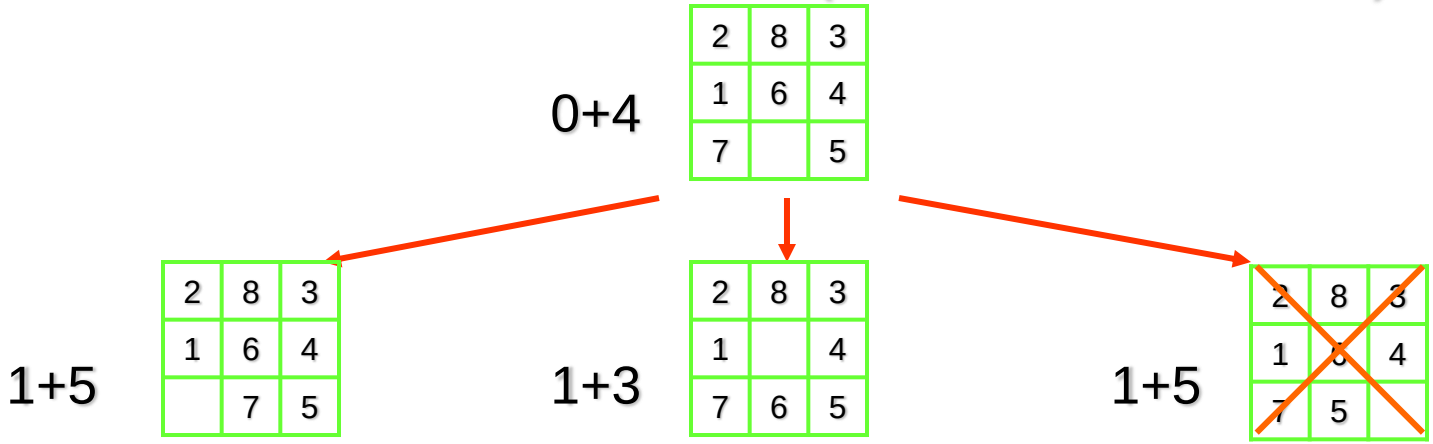


Beam Search (with limit = 2)

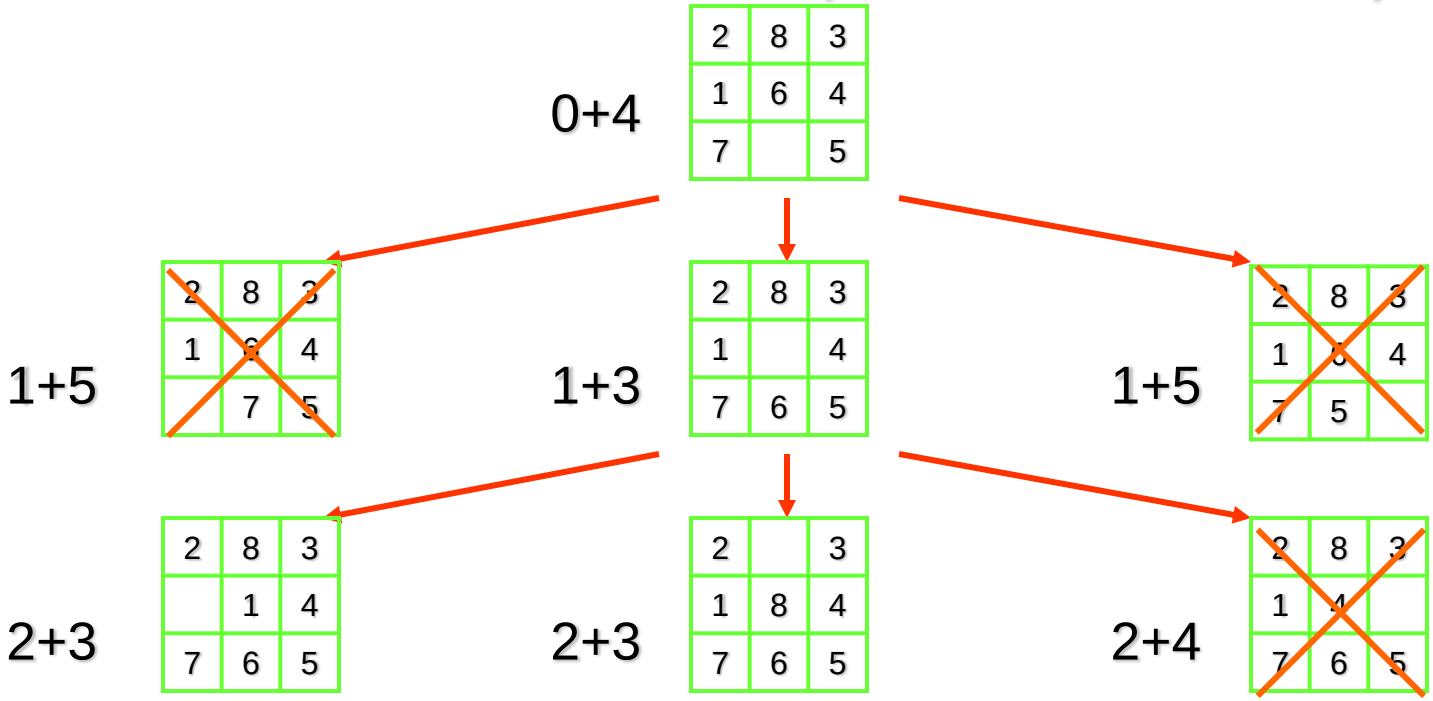
0+4



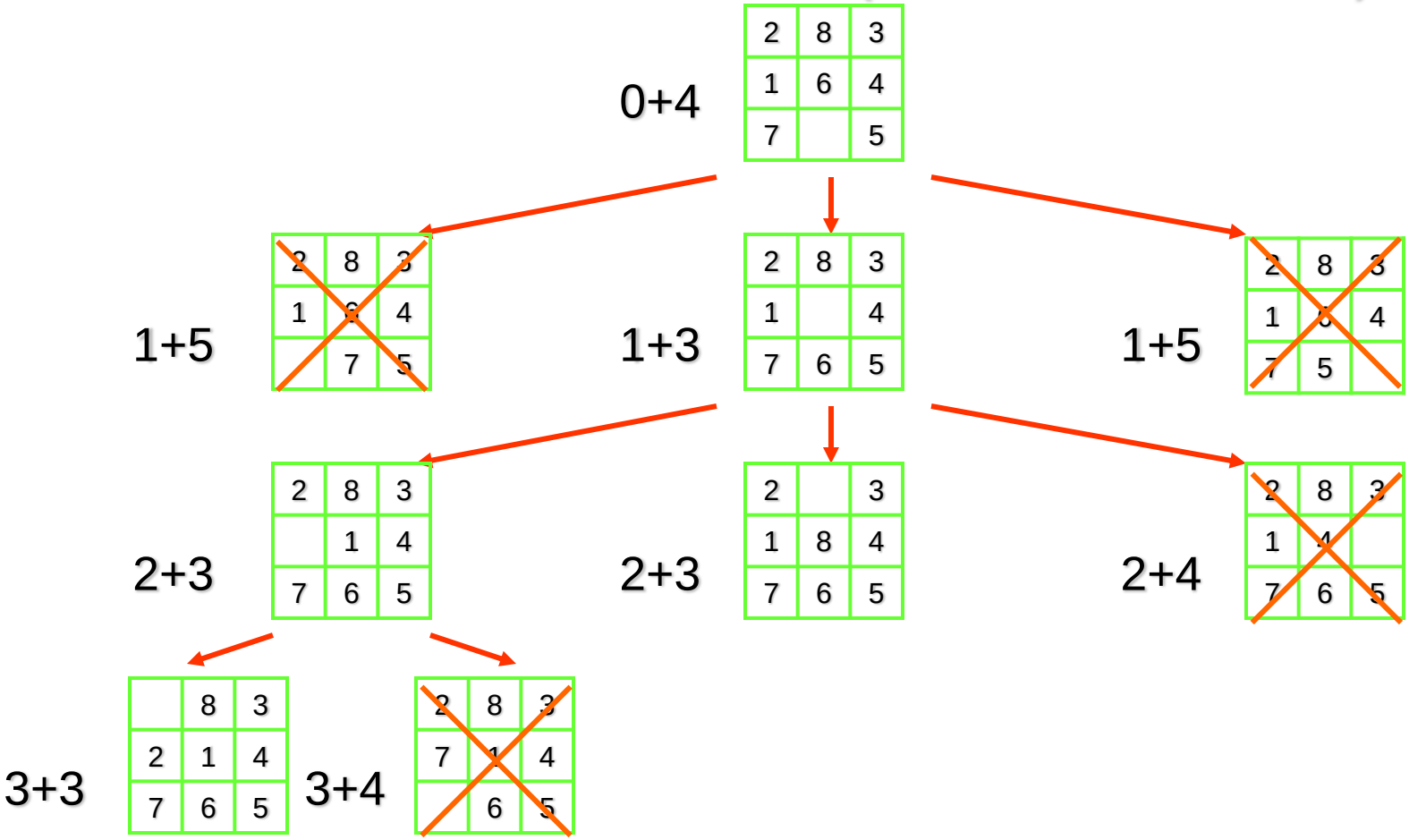
Beam Search (with limit = 2)



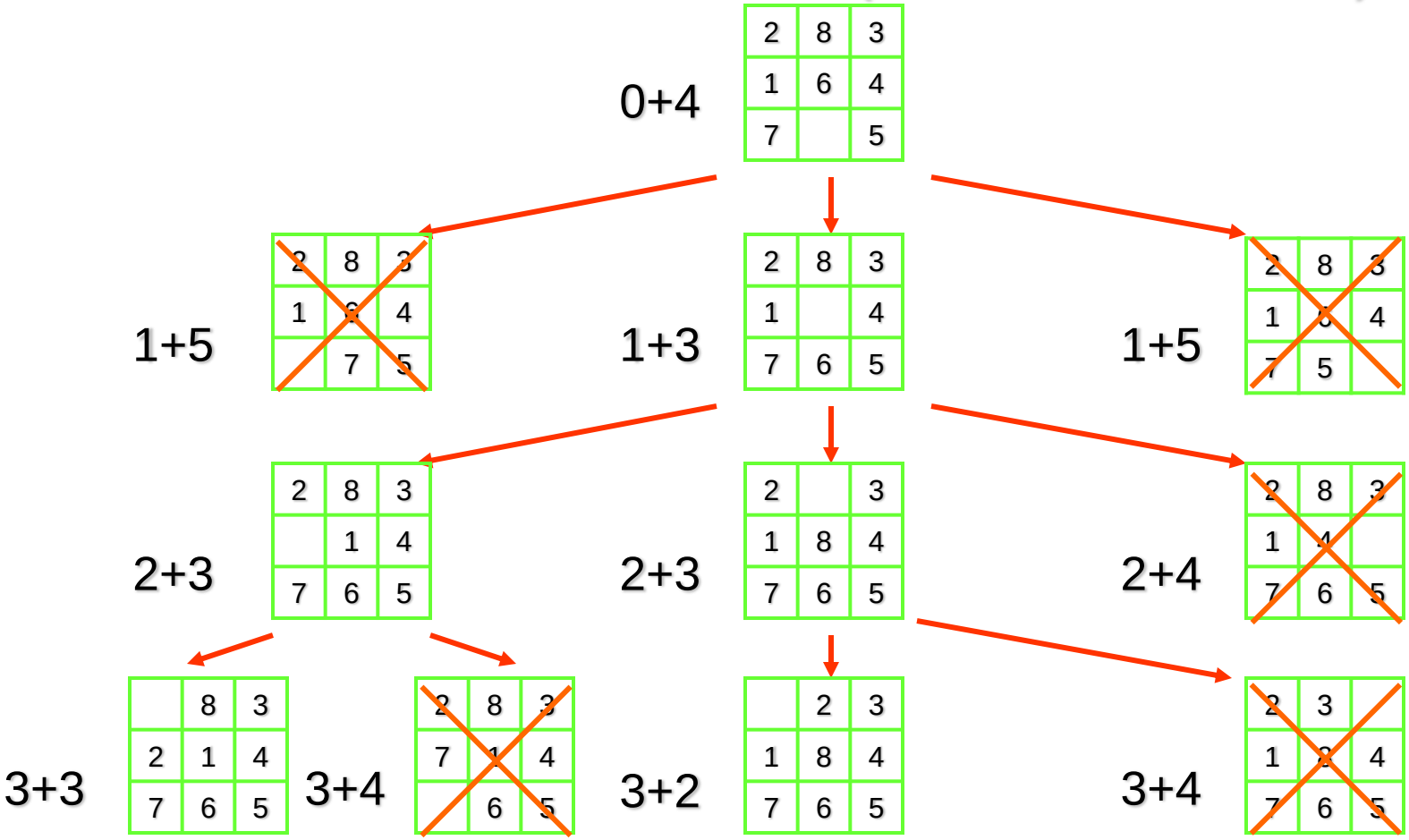
Beam Search (with limit = 2)



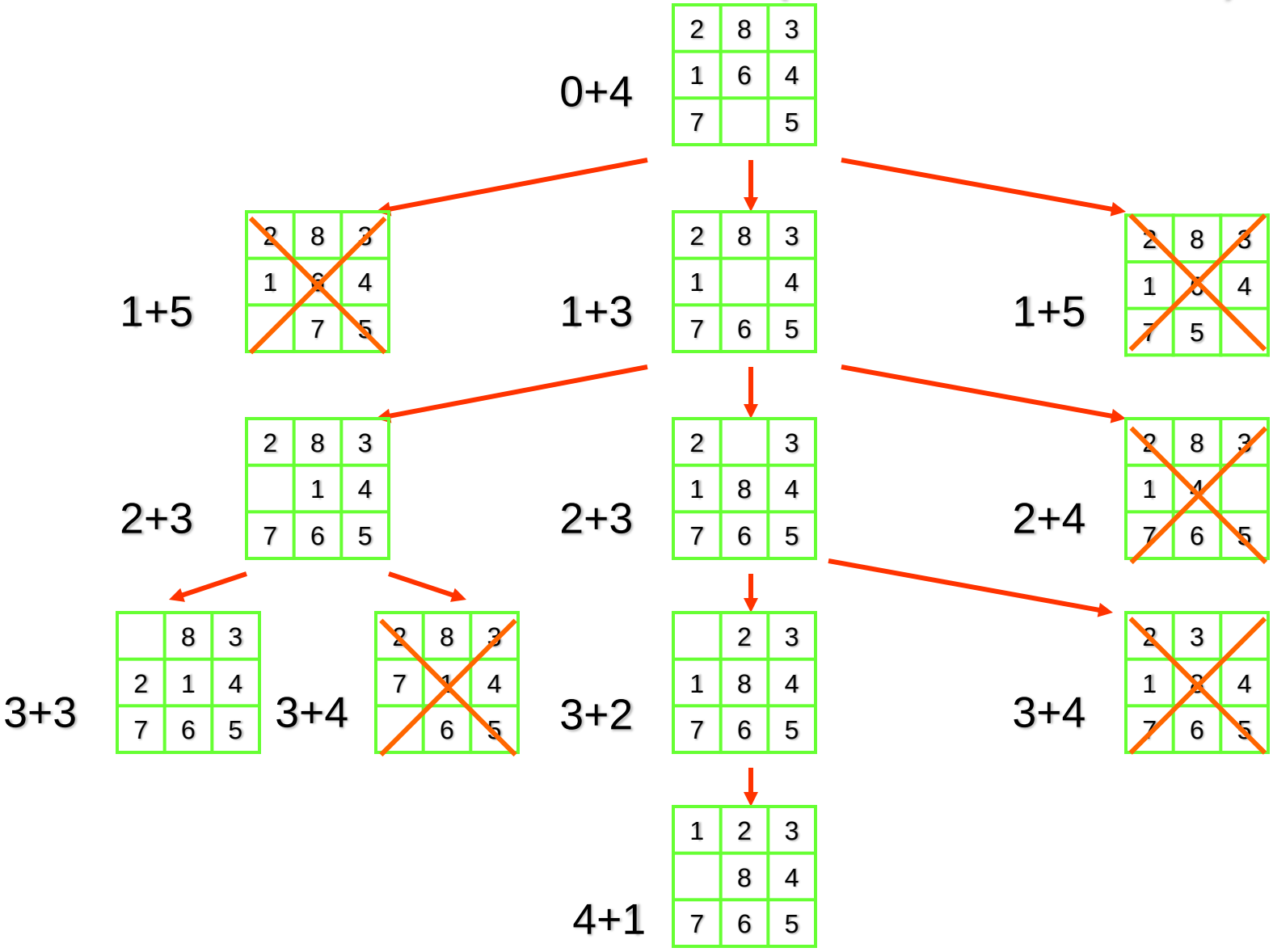
Beam Search (with limit = 2)



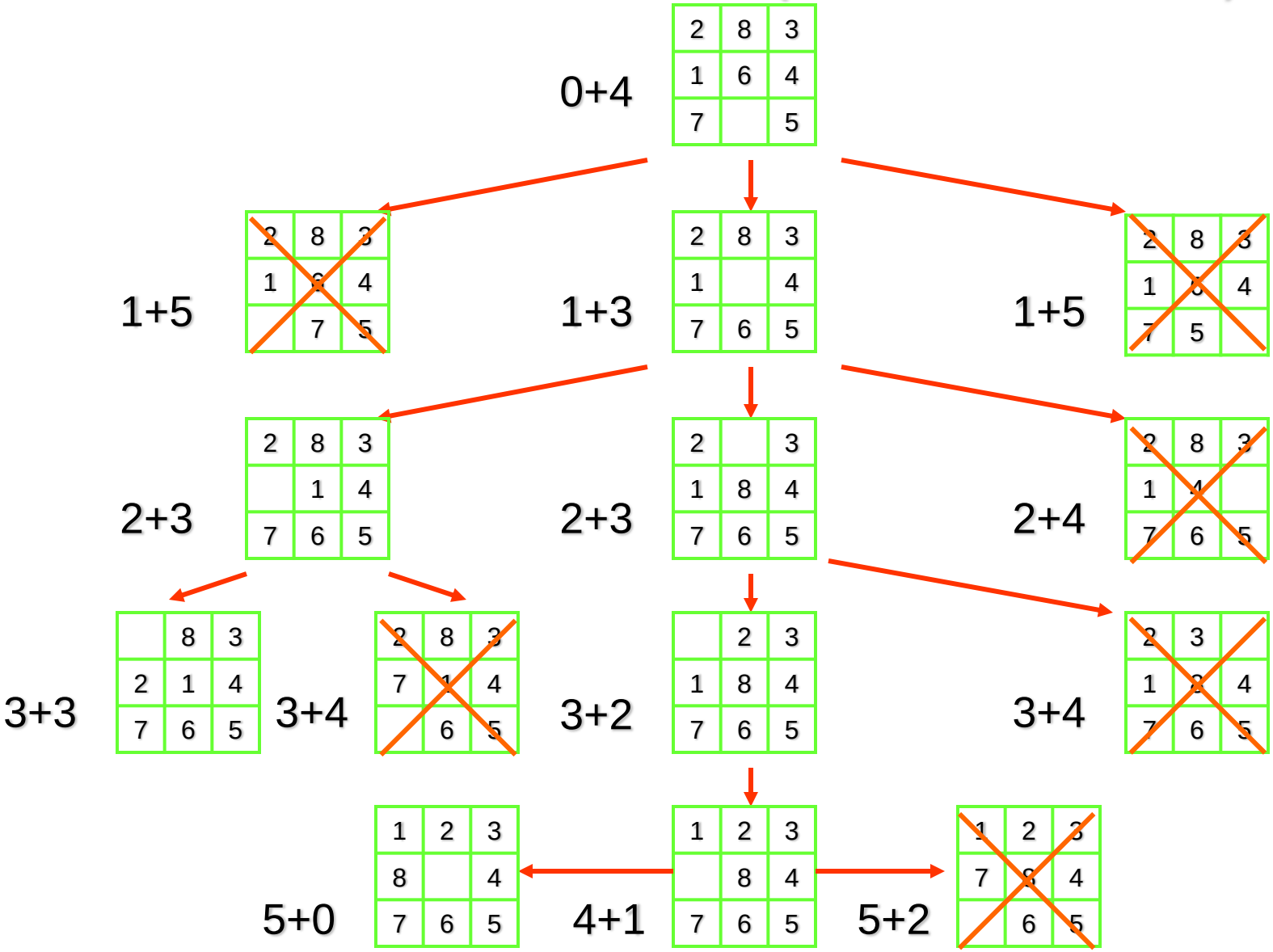
Beam Search (with limit = 2)



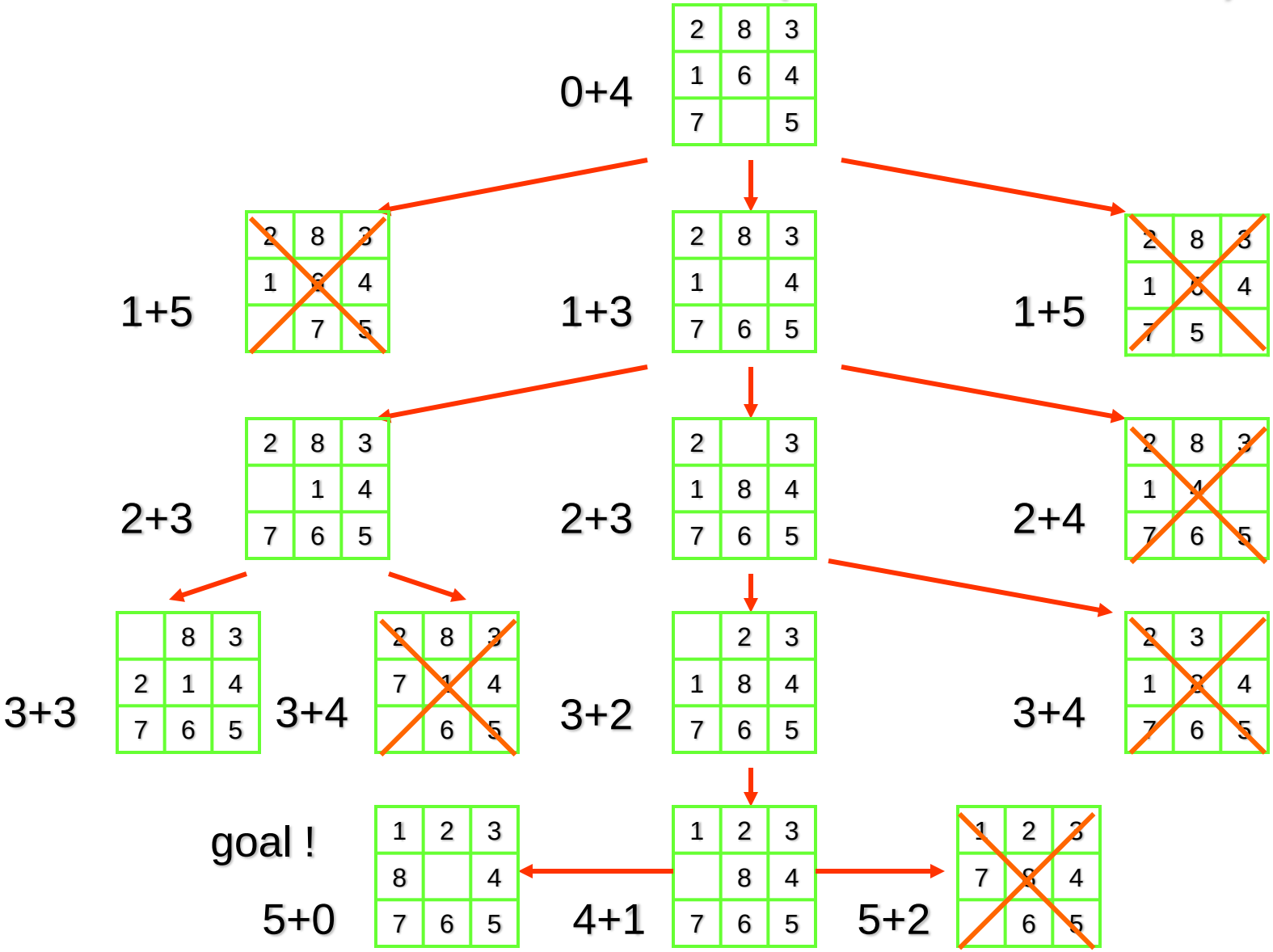
Beam Search (with limit = 2)



Beam Search (with limit = 2)



Beam Search (with limit = 2)



Generated and Test

- Algorithm
  1. Generate a (potential goal) state:
     - Particular point in the problem space, or
     - A path from a start state
  2. Test if it is a goal state
     - Stop if positive
     - go to step 1 otherwise
- Systematic or Heuristic?

Generated and Test

- Algorithm
  1. Generate a (potential goal) state:
     - Particular point in the problem space, or
     - A path from a start state
  2. Test if it is a goal state
     - Stop if positive
     - go to step 1 otherwise
- Systematic or Heuristic?
  - It depends on “Generate”

Local search algorithms

- In many problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Hill Climbing

- Simple Hill Climbing
  - expand the current node
  - evaluate its children one by one (using the heuristic evaluation function)
  - choose the FIRST node with a better value

- Steepest Ascend Hill Climbing
  - expand the current node
  - Evaluate all its children (by the heuristic evaluation function)
  - choose the BEST node with the best value

Hill-climbing

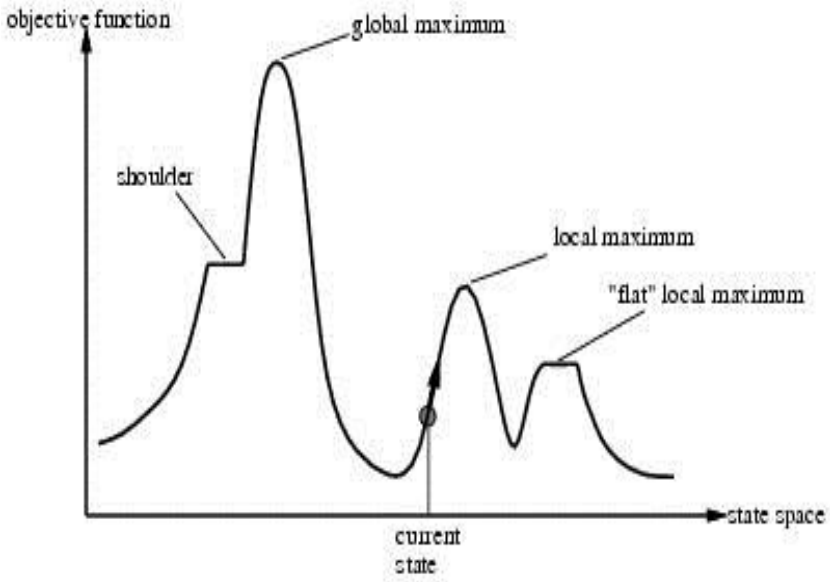
- Steepest Ascend

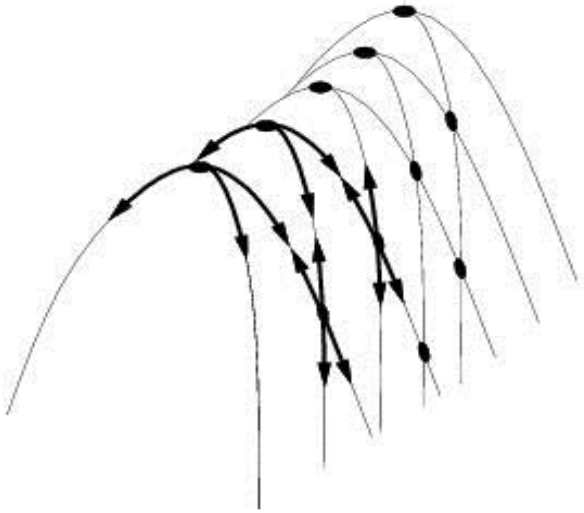
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Hill Climbing

- Problems:
  - local maxima problem
  - plateau problem
  - Ridge

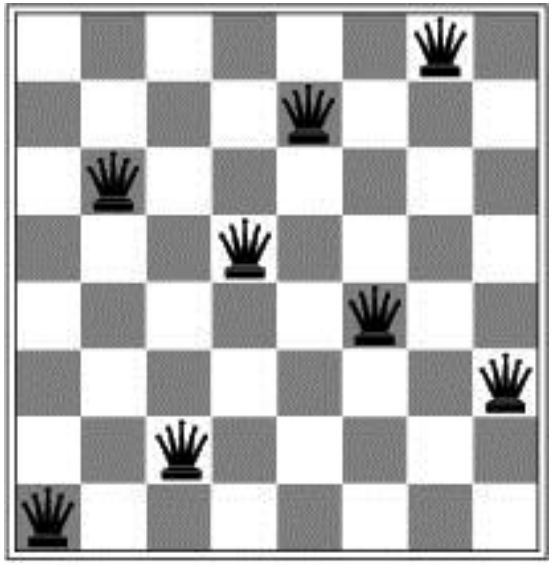
Drawbacks





- Ridge = sequence of local maxima difficult for greedy algorithms to navigate
- Plateaux = an area of the state space where the evaluation function is flat.

Hill-climbing example



a) shows a state of h=17 and the h-value for each possible successor.

b) A local minimum in the 8-queens state space (h=1).

Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
  - A series of Hill Climbing searches from randomly generated initial states
- Simulated Annealing
  - Escape local maxima by allowing some "bad" moves but gradually decrease their frequency

Simulated Annealing

- T = initial temperature
- x = initial guess
- v = Energy(x)
- Repeat while T > final temperature
  - Repeat n times
    - $X' \leftarrow$ Move(x)
    - V' = Energy(x')
    - If $v' < v$ then accept new x [$x \leftarrow x'$]
    - Else accept new x with probability $\exp(-(v' - v)/T)$
  - T = 0.95T /* Annealing schedule*/
- At high temperature, most moves accepted
- At low temperature, only moves that improve energy are accepted

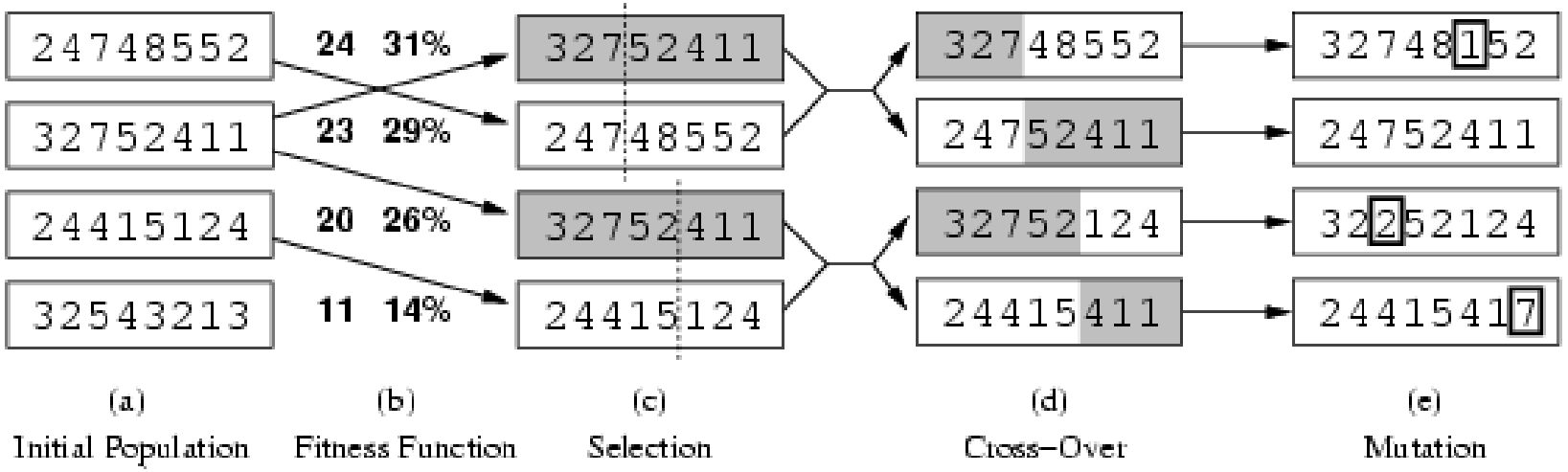
Local beam search

- Keep track of *k* “current” states instead of one
  - Initially: *k* random initial states
  - Next: determine all successors of the *k* current states
  - If any of successors is goal $\rightarrow$ finished
  - Else select *k* best from the successors and repeat.
- Major difference with *k* random-restart search
  - Information is shared among *k* search threads.
- Can suffer from lack of diversity.
  - Stochastic variant: choose *k* successors at proportionally to the state success.

Genetic algorithms

- Variant of local beam search with *sexual recombination.*
- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

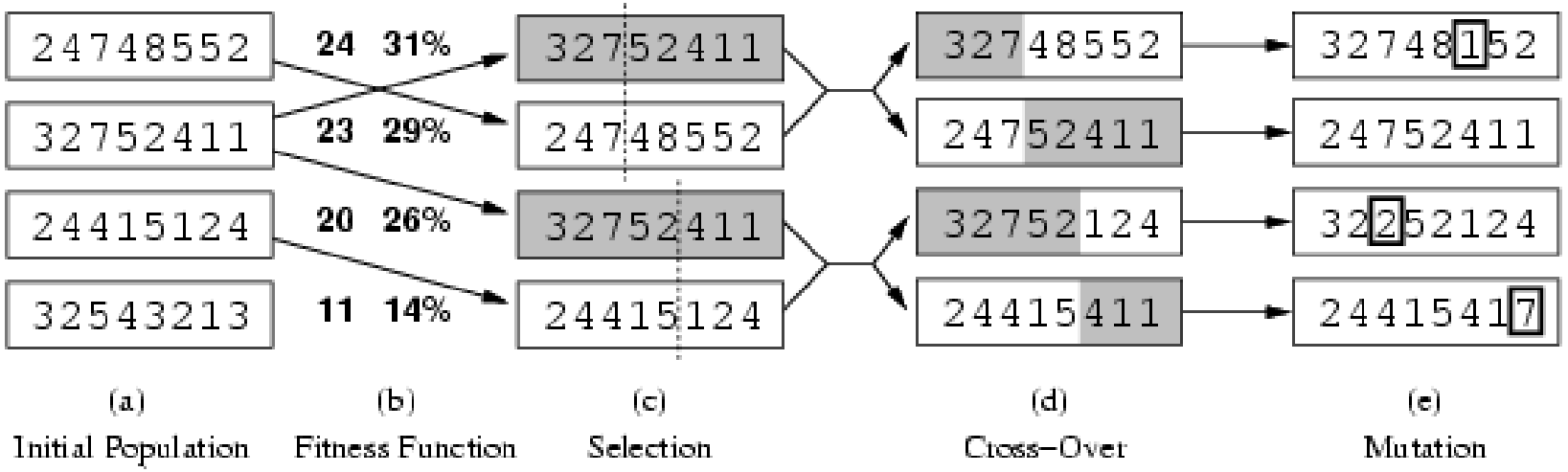
Genetic algorithms

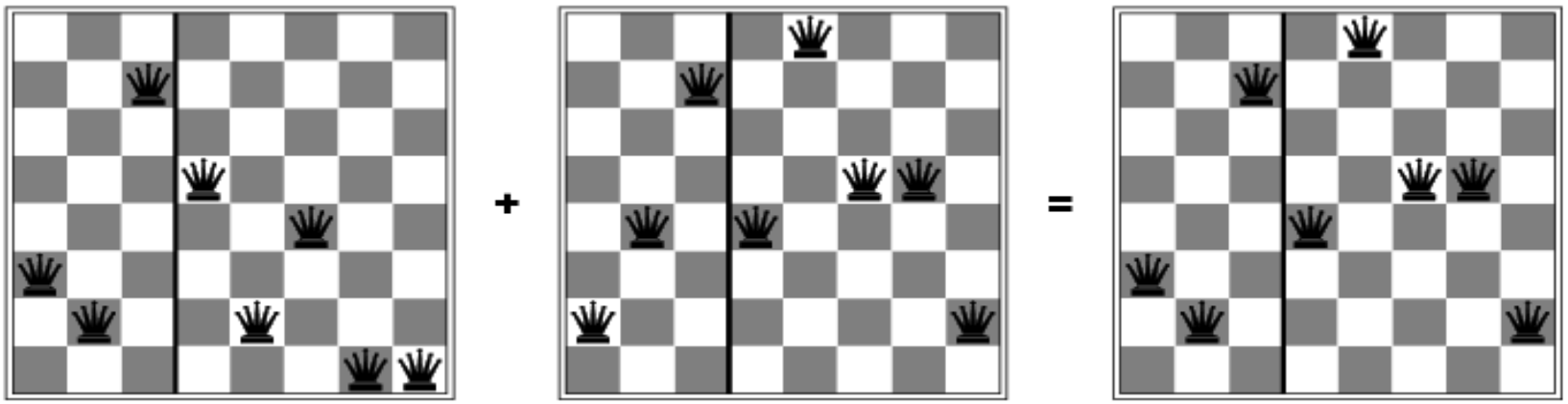


- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)

- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Genetic algorithms





A Genetic algorithm

function GENETIC_ALGORITHM(*population,* FITNESS-FN) **return** an individual
    input: *population*, a set of individuals
        FITNESS-FN, a function which determines the quality of the individual
    repeat
        new_population ← empty set
        loop for i **from** 1 **to** SIZE(*population*) **do**
            x ← RANDOM_SELECTION(*population*, FITNESS_FN)
            y ← RANDOM_SELECTION(*population*, FITNESS_FN)
            child ← REPRODUCE(*x,y*)
            if (small random probability) **then** *child* ← MUTATE(*child*)
            add *child* to *new_population*
        population ← *new_population*
    until some individual is fit enough or enough time has elapsed
    return the best individual

A Genetic Algorithm (Cont.)

function REPRODUCE(*x, y*) **return** an individual

input: *x, y*, parent individuals

$n \leftarrow$ LENGTH(*x); c* $\leftarrow$ random number from 1 to *n*

return APPEND(SUBSTRING(*x*, 1, *c*), SUBSTRING(y, *c* + 1, *n*))

In this more popular version of GA, from each two parents, only one offspring is produced, not two.