Imperative Programming

(week 13)

http://www.math.ru.nl/knopen/art_gallery.html

Recap last week

- Solving search problems: investigate all possible solutions that have certain properties
- Recursive algorithms
  - base cases detect solutions and non-solutions
  - recursive cases develop solutions
- Typical for these algorithms:
  - bookkeeping actions before and after recursive call(s)
  - large, if not huge, search space of algorithm
- Case study: the coins problem

Solving search problems

- Search problem solving algorithm **solve**:
  - path $P = A_0 \ldots A_n$ keeps track of attempt-so-far
  - base cases:
    - P is a solution: process P
    - P is senseless: abandon P
  - recursive cases:
    - for each attempt A' that extends A_n ($A_n \to A'$) **solve** $P' = P\,A'$

solve (P):
 if solution (P): process (P)
 else if senseless (P): stop
 else for each $A_n \to A'$: **solve** (P')

Solving search problems

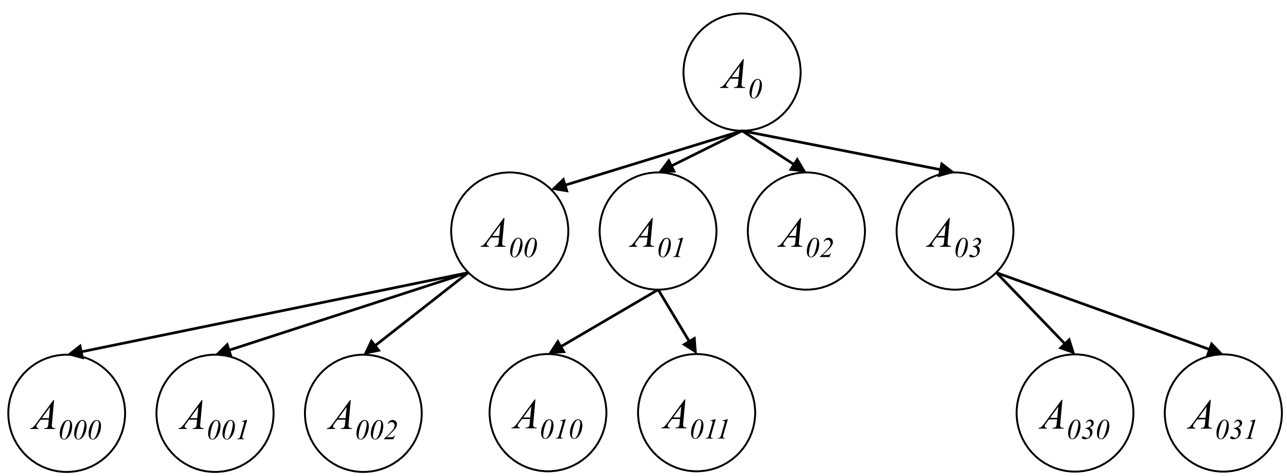
- Search problem solving algorithm **solve**:
  - path $P = A_0 \ldots A_n$ keeps track of attempt-so-far and best-solution-so-far B
  - base cases:
    - P is a solution: process P and improve B
    - P is senseless: abandon P
  - recursive cases:
    - for each attempt A' that extends A_n ($A_n \to A'$) **solve** $P' = P\,A'$ with B

solve (P):	
if solution (P):	process (P)
else if senseless (P):	stop
else for each $A_n \to A'$:	**solve** (P')

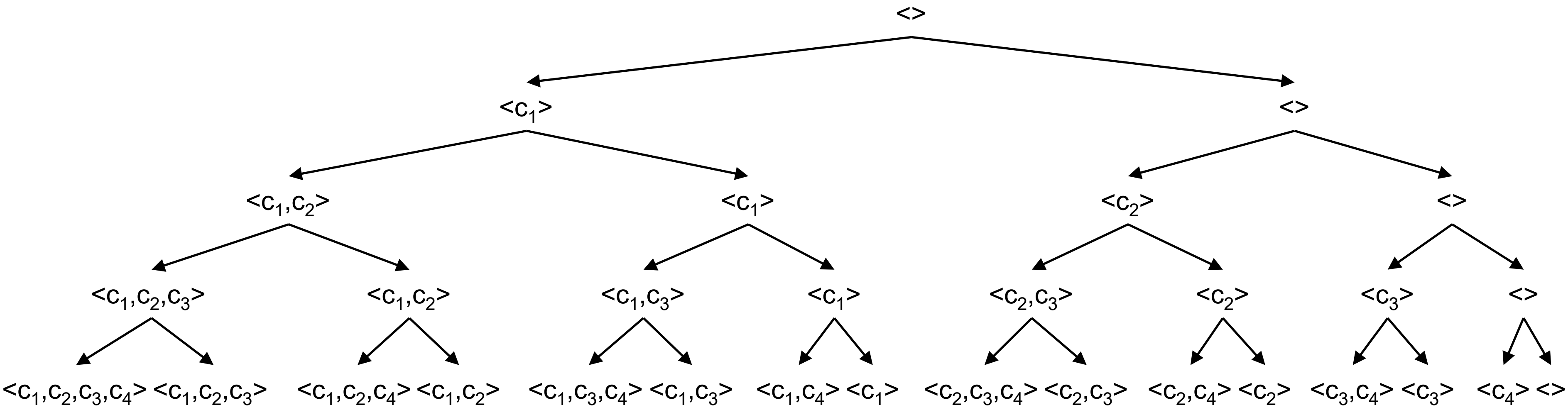
solve (P, B):	
if solution (P):	process (P, B)
else if senseless (P, B):	stop
else for each $A_n \to A'$:	**solve** (P', B)

Search space of algorithm

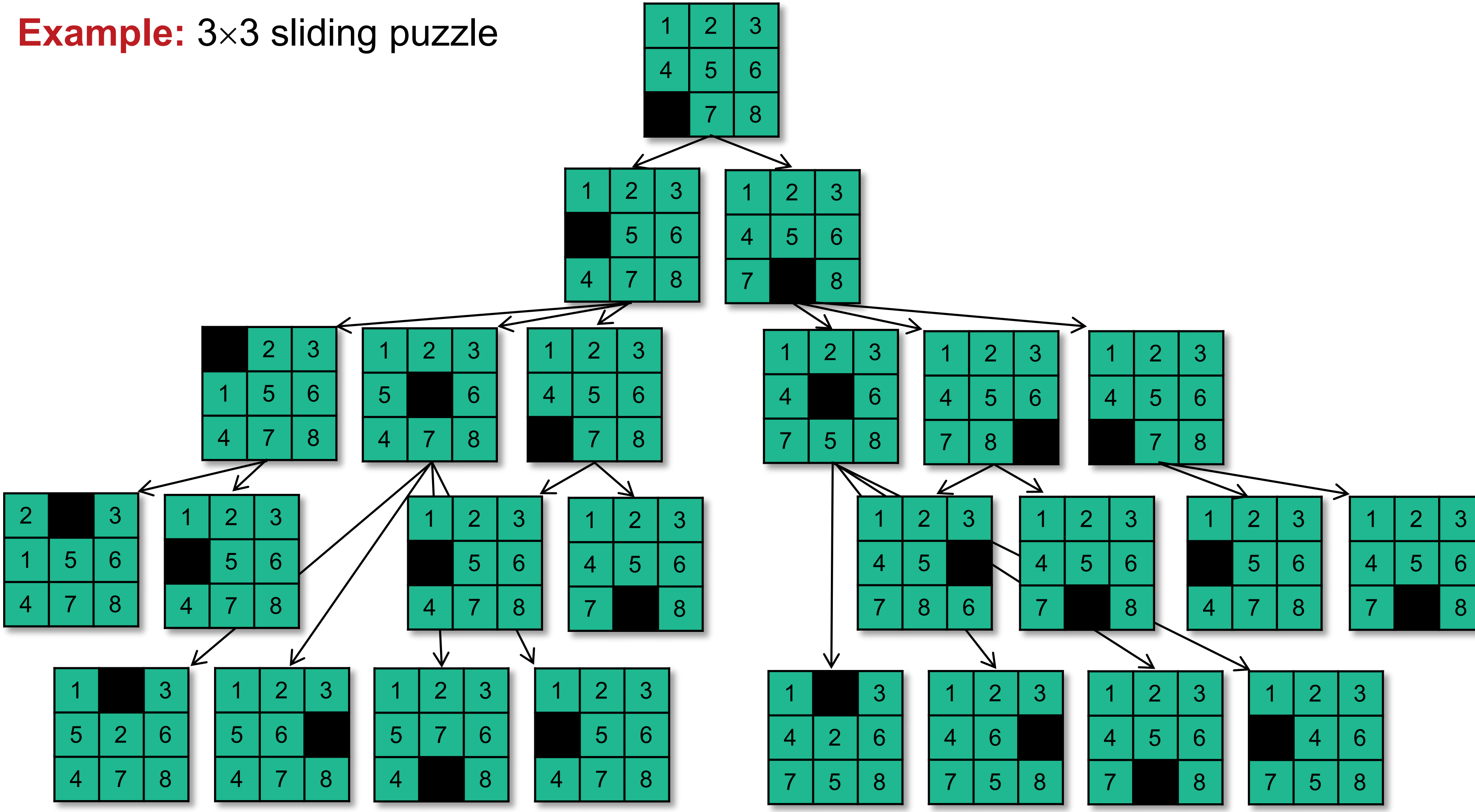
- Tree structure:
  - node A: possible attempt
  - edge $A \to A'$: A' is a computed extension of A



Example: 4 coins (c_1,c_2,c_3,c_4)



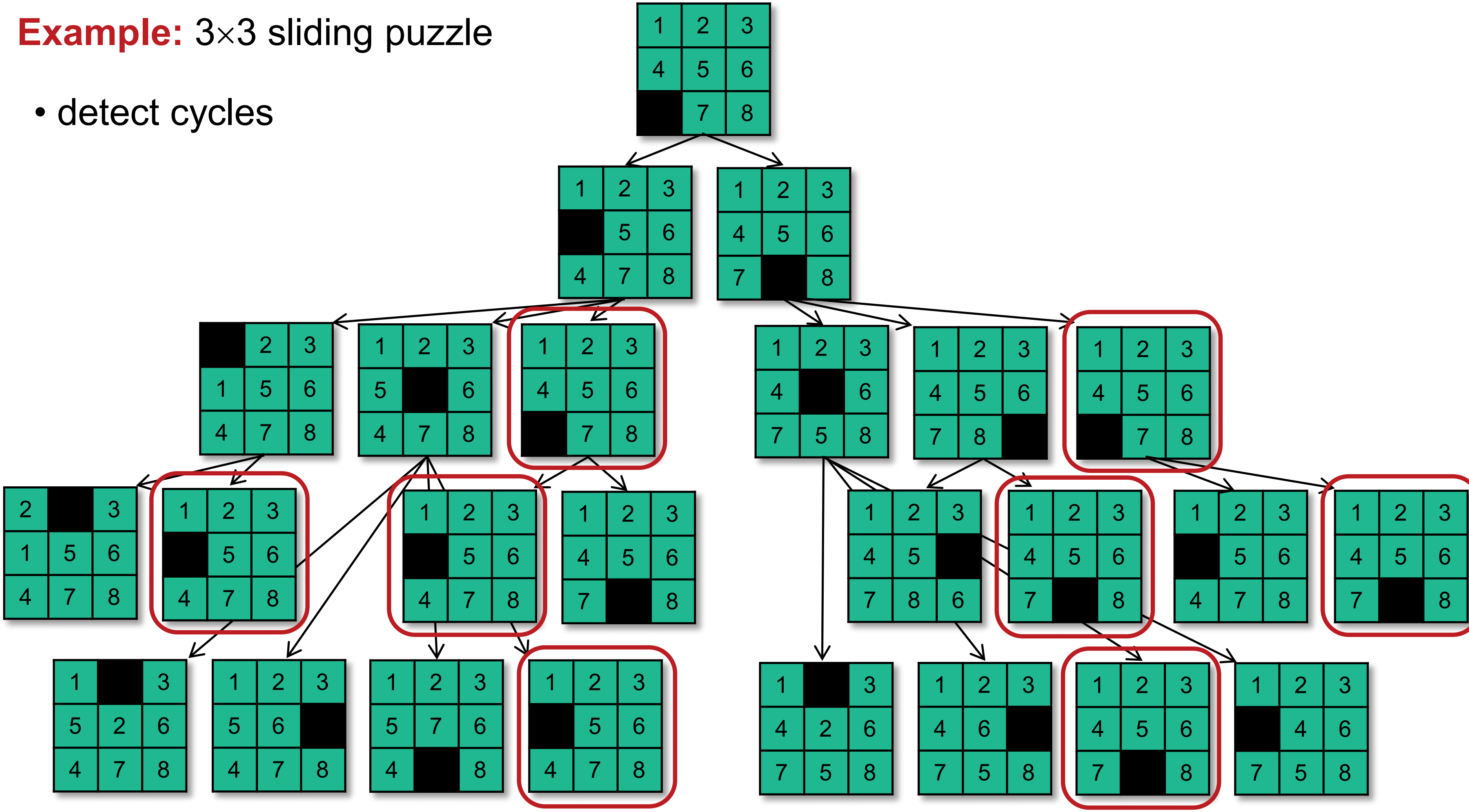
Example: 3×3 sliding puzzle



etc…

Example: 3×3 sliding puzzle

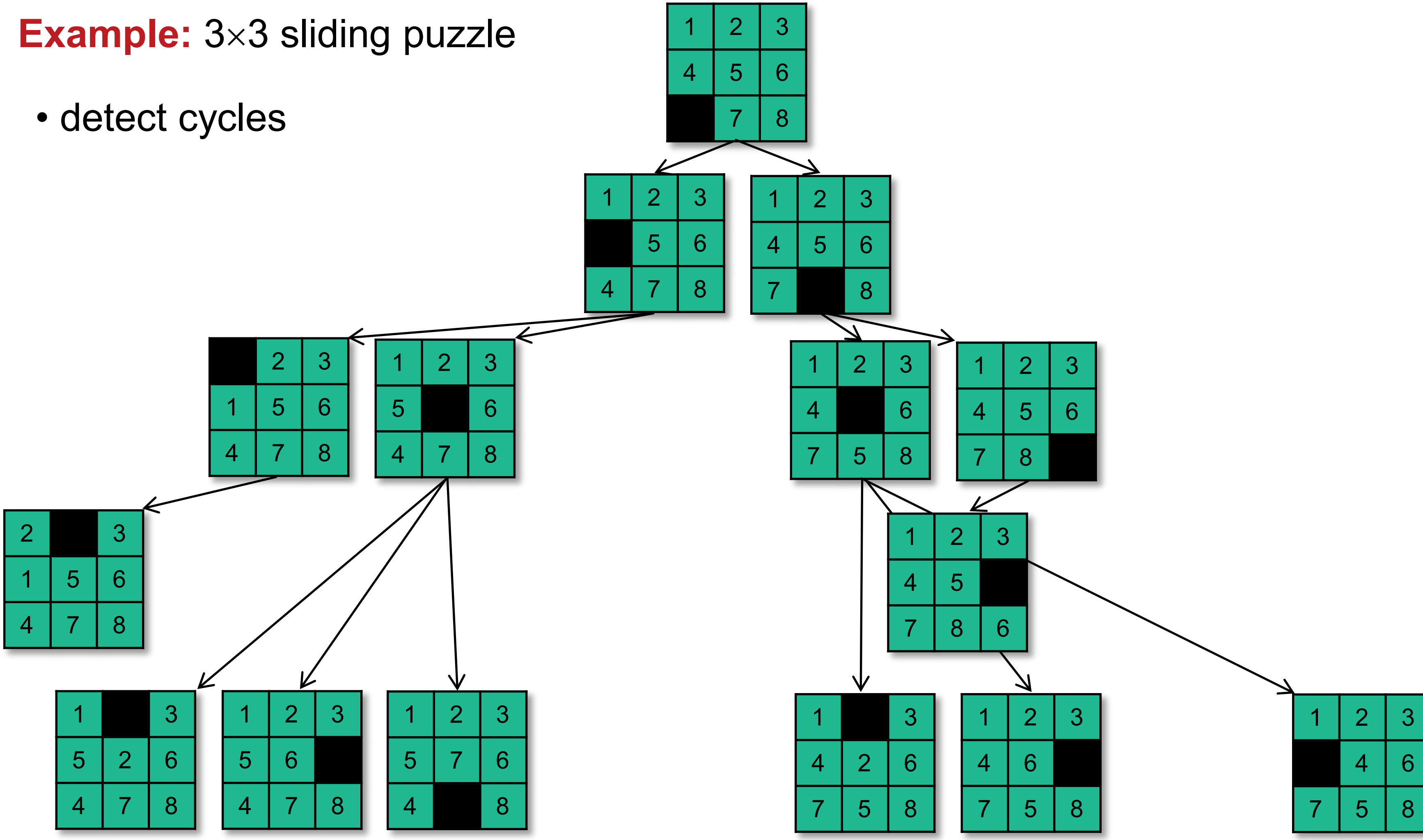
- detect cycles



etc…

Example: 3×3 sliding puzzle

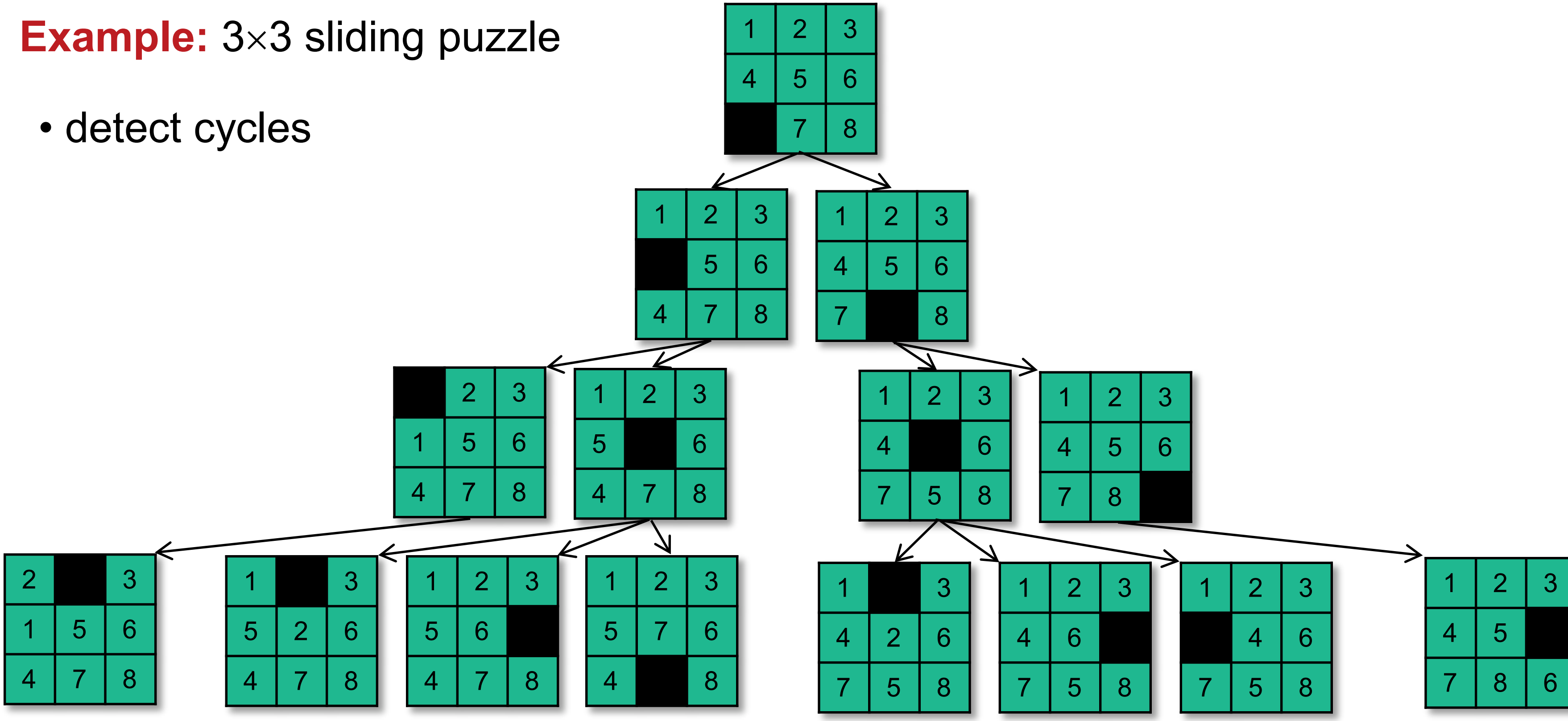
- detect cycles



etc…

Example: 3×3 sliding puzzle

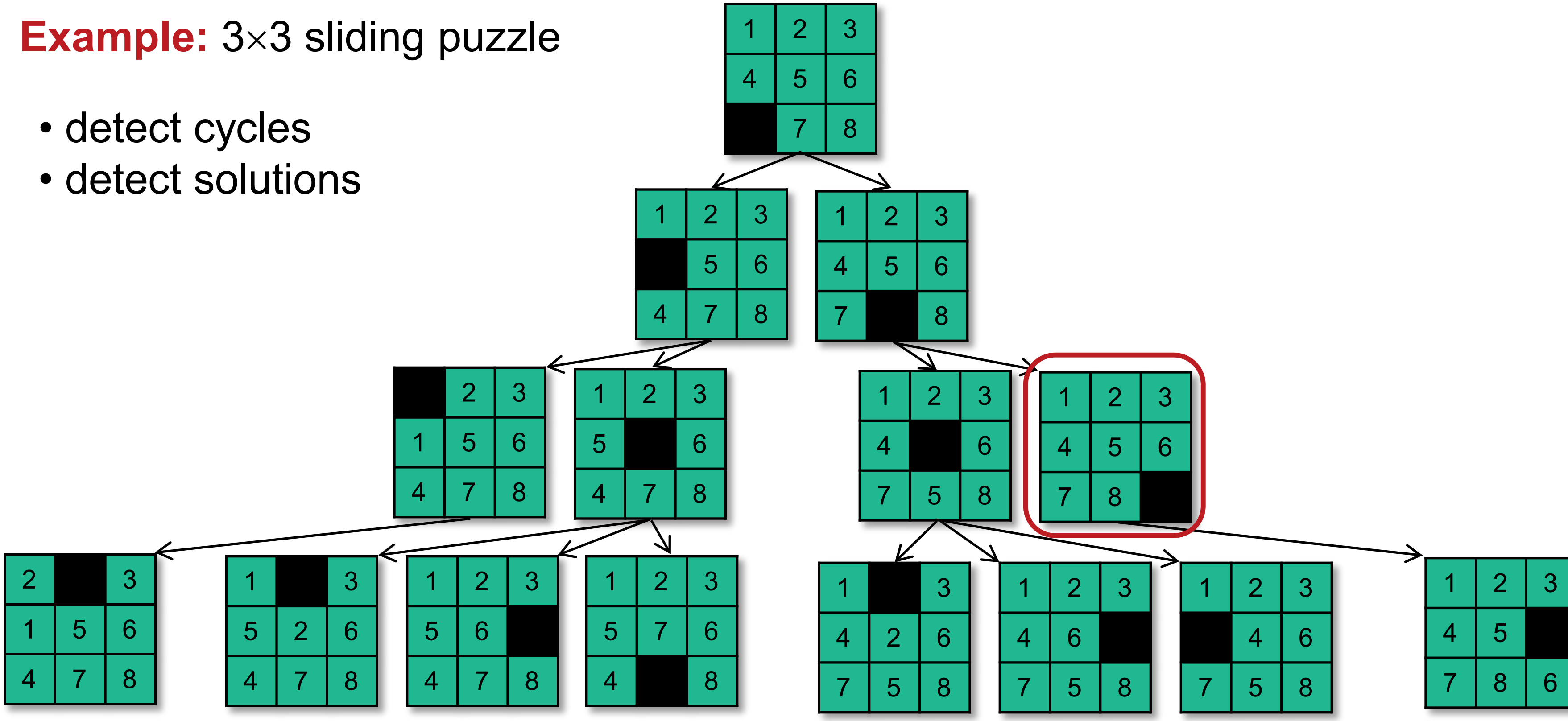
- detect cycles



etc…

Example: 3×3 sliding puzzle

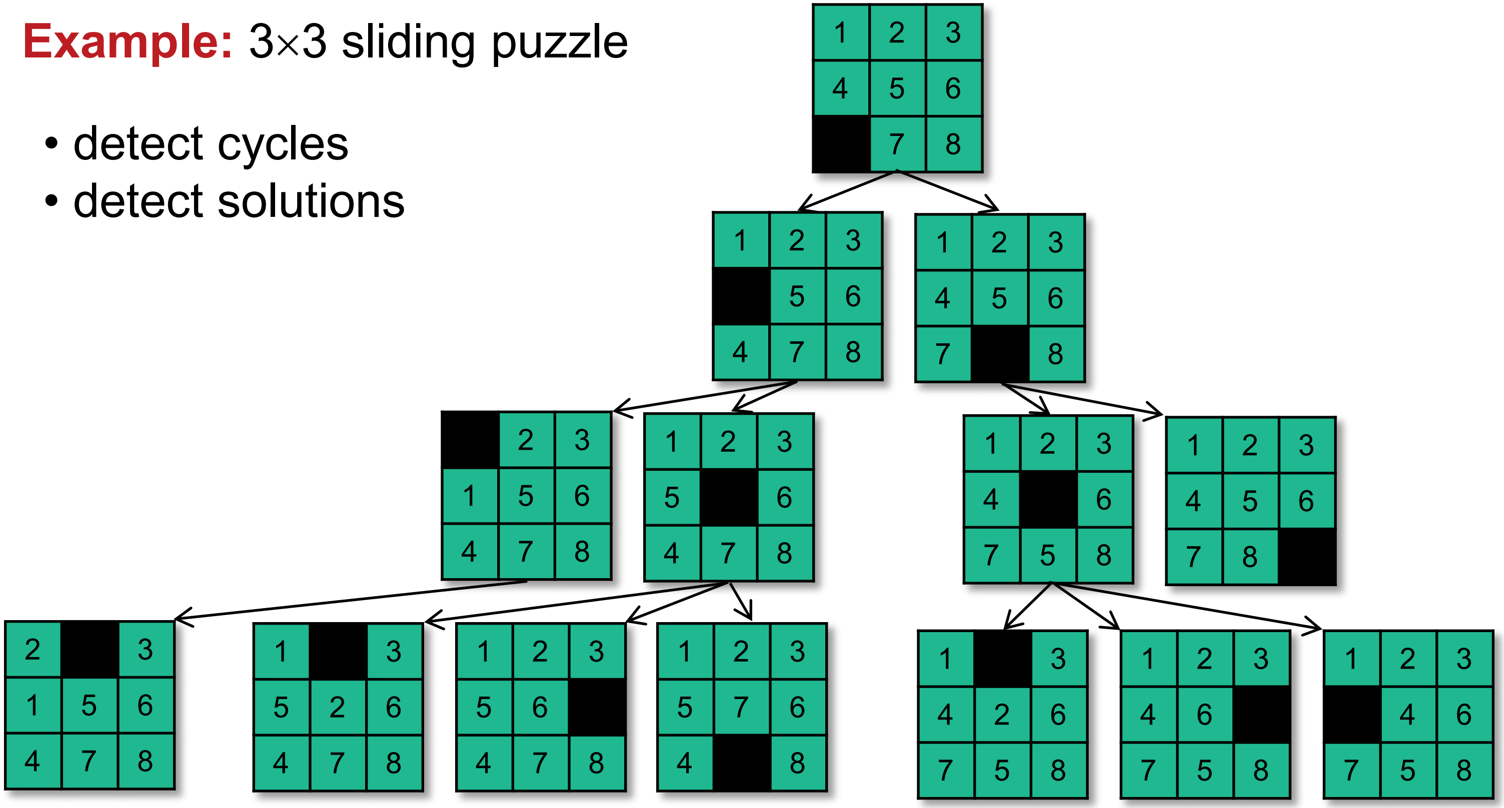
- detect cycles
- detect solutions



etc…

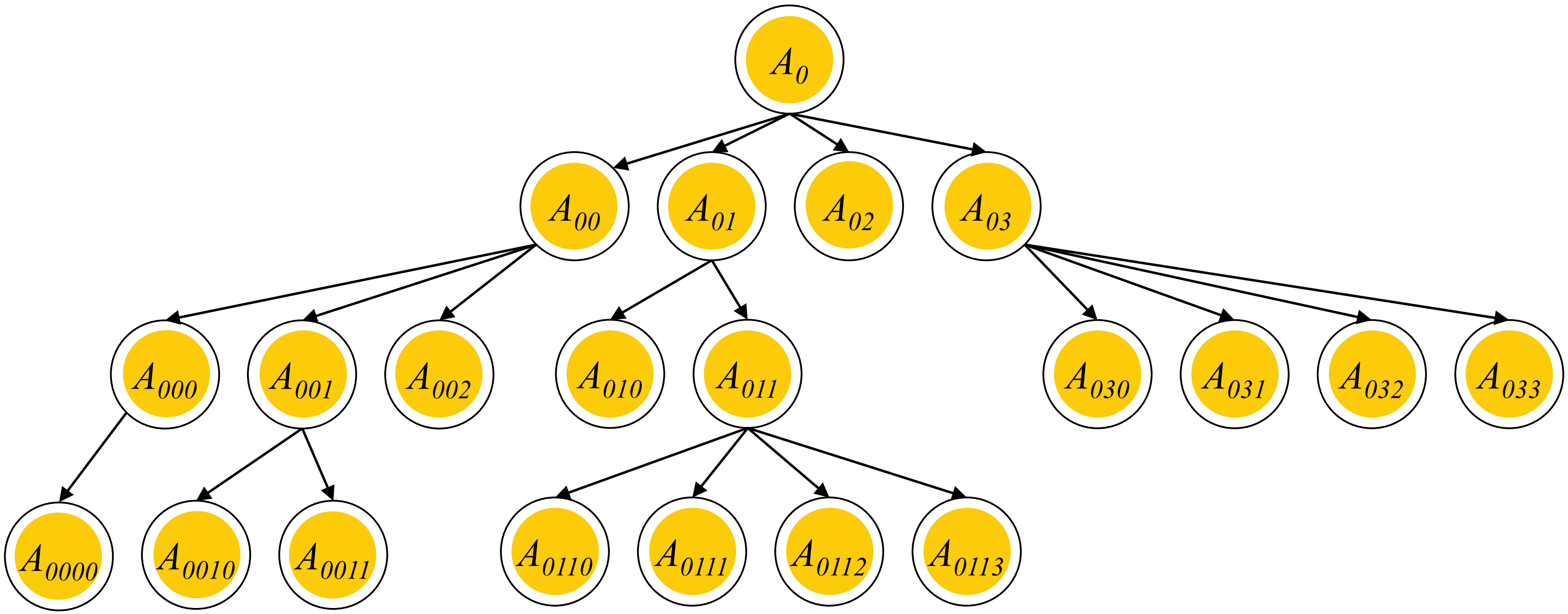
Example: 3×3 sliding puzzle

- detect cycles
- detect solutions



etc…

Solving search problems

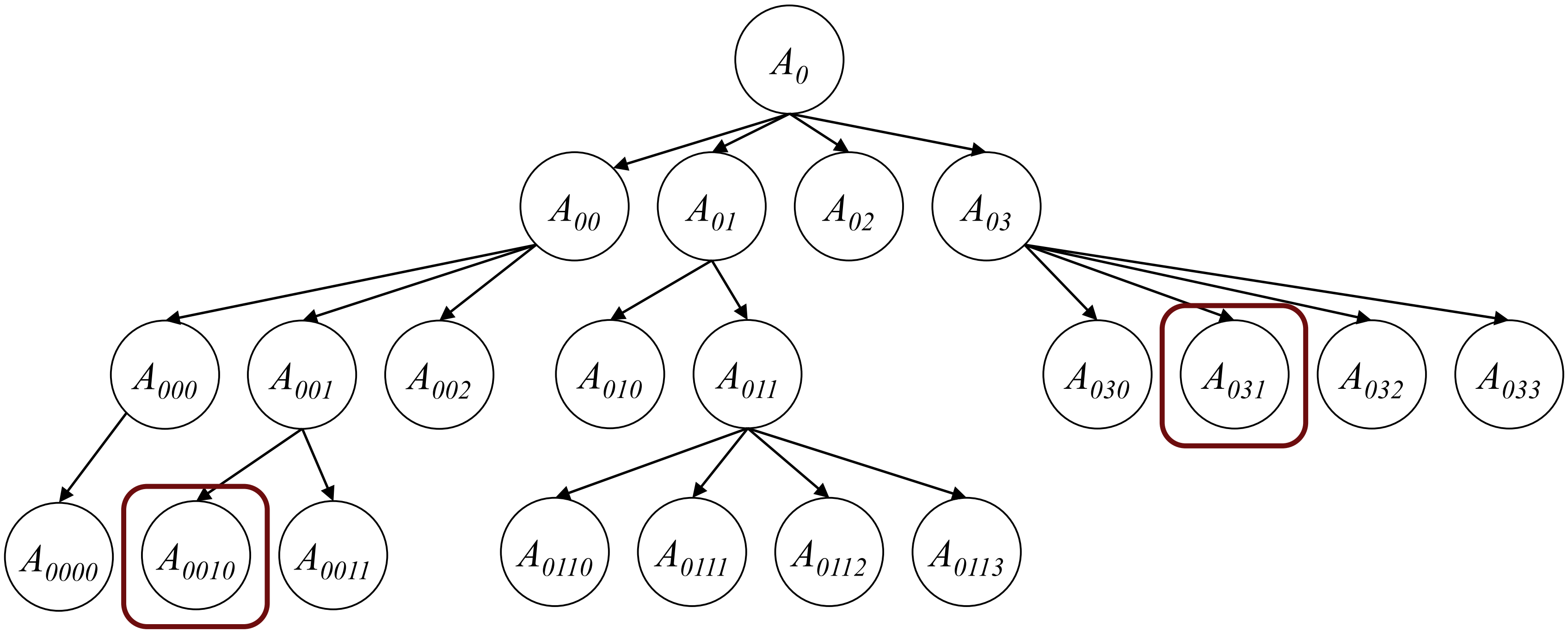


etc…

solve (P):
if solution (P): process (P)
else if senseless (P): stop
else for each $A_n \to A'$:
if $A' \notin P$: **solve** (P')

solve (P , B):
if solution (P): process (P, B)
else if senseless (P,B): stop
else for each $A_n \to A'$:
if $A' \notin P$: **solve** (P', B)

Solving search problems: depth-first search

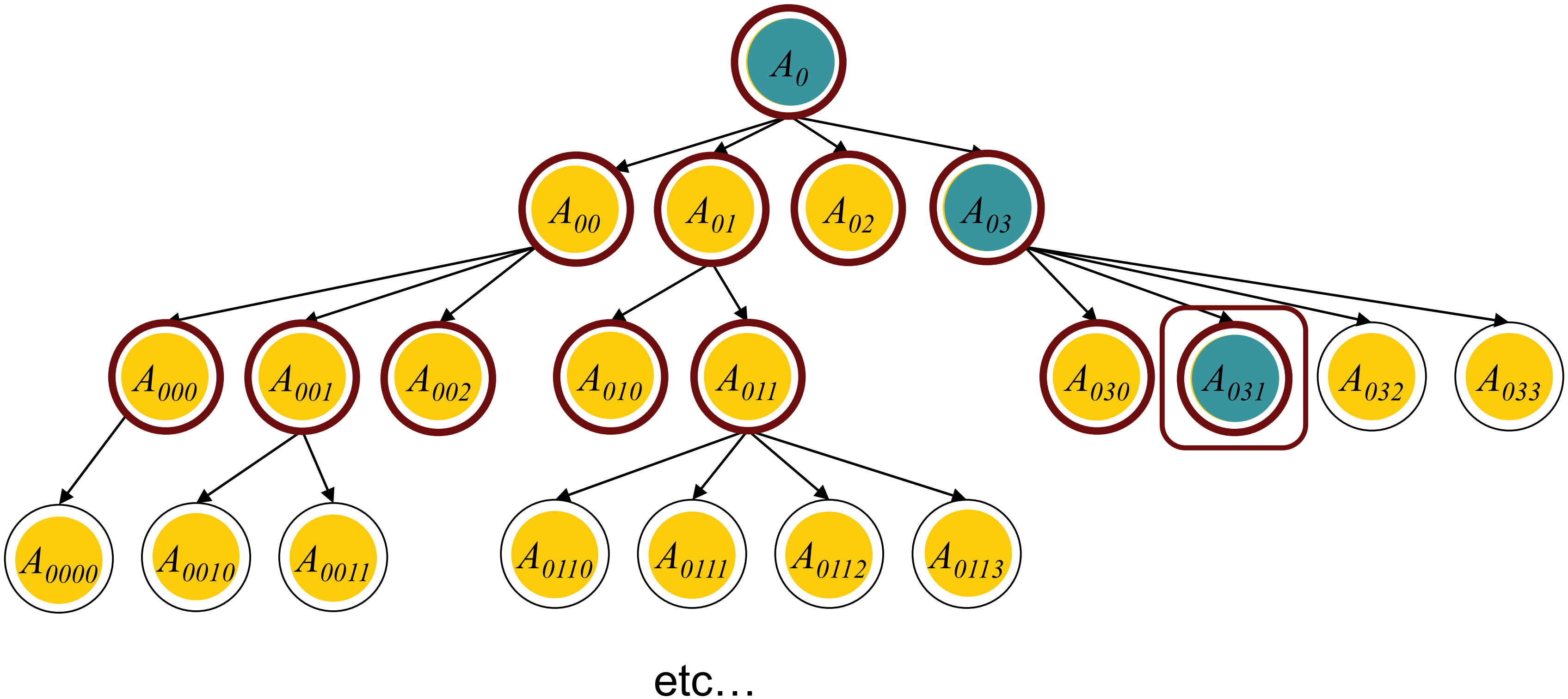


etc…

Advantages:
- finds ‘deep’, ‘left’ solutions early
- recursive structure
- relatively space-efficient
- solution path is readily available

Disadvantages:
- finds ‘undeep’, ‘right’ solutions too late
- might do double work

Solving search problems: breadth-first search



Advantages:
- finds 'undeep' solutions first
- can avoid double work
- iterative structure

Disadvantages:
- relatively space-inefficient
- solution path is not readily available

Solving search problems:
breadth-first search

- Keep track of attempts and index of their parent:

```
struct Candidate
{ A   candidate ;
  int parent_candidate ;
} ;
vector<Candidate> c = {{A0,-1}} ;
```

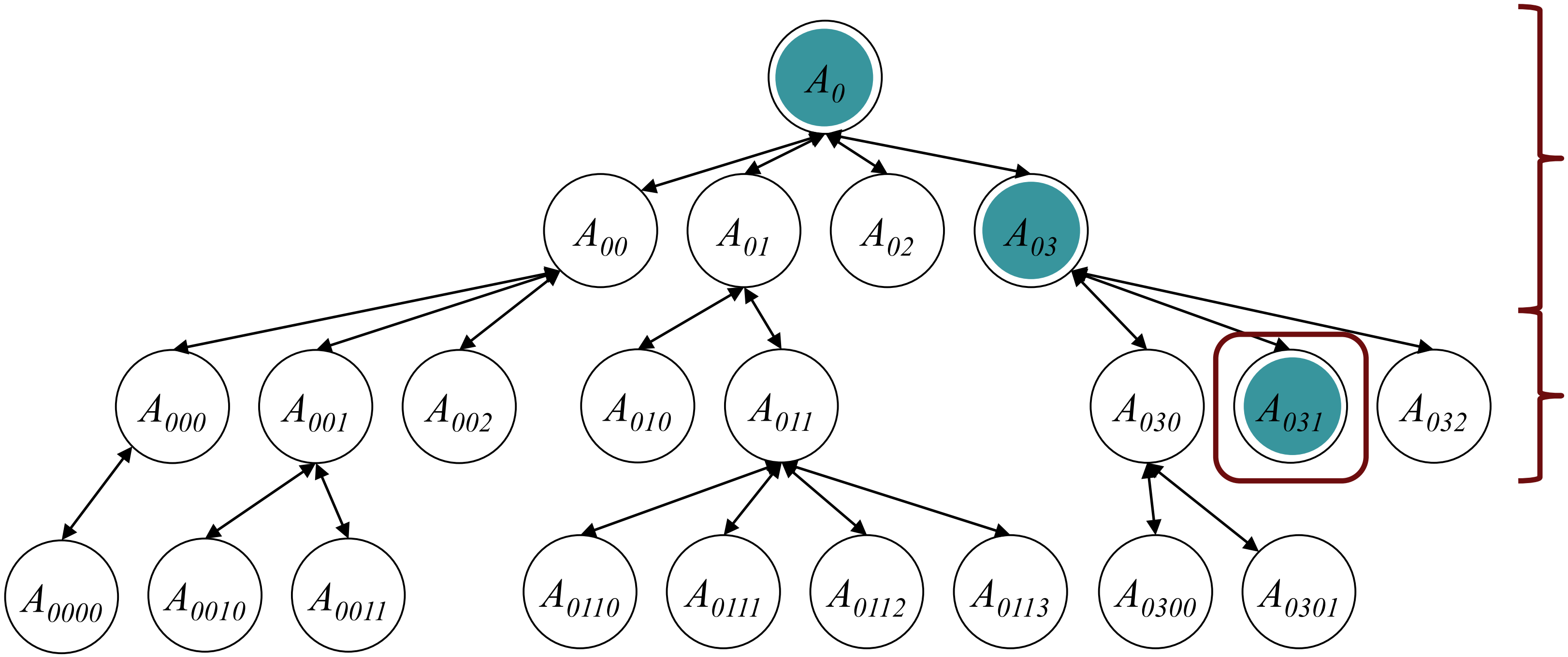
- Keep inspecting candidates as long as there are some

Solving search problems: breadth-first search

- Search problem solving algorithm **solve**:
  - keep track of attempts-so-far (C is initially $\{(A_0,-1)\}$) and index i of current attempt (i is initially 0)
  - while there still are attempts ($i <$ size (C)) and current attempt ($A_i = C[i]$.candidate) is no solution:
    - push every A' that extends A_i to C as (A',i)
    - increment i
  - if a solution is found: process it (using C and i)

solve (A_0):
  C = $\{(A_0,\text{-}1)\}$; i = 0 ;
  while (i < size (C) and $C[i]$.candidate is no solution):
    [(**for each** $C[i]$.candidate $\rightarrow A'$: push (A', i) to C) ; i = $i+1$; **]**
  if (i < size (C)) **then** **show path** (C, i) ;

Solving search problems: <u>breadth-first search</u>



show path (C, i):
if i < 0 : ready, do nothing
else
[**show path** (C, C[i].parent_candidate) ; show (C[i].candidate) **]**

Case study: sliding puzzle

- Given m by n puzzle
- Slides are numbered $1,\ldots,m\cdot n\text{-}1$
- Move one slide at a time to empty slot
- Ready when slides are ordered:
  - left-to-right, top-to-bottom: $1,\ldots,m\cdot n\text{-}1$
- Compute shortest solution
- Breadth-first search
- Depth-first search

Case study: sliding puzzle

- Design data structures:

```
const int WIDTH  = ... ;   // WIDTH > 0
const int HEIGHT = ... ;   // HEIGHT > 0
typedef int Cell ;         // 1 ≤ value ≤ WIDTH*HEIGHT
const Cell EMPTY = WIDTH * HEIGHT ;
struct Pos
{ int col, row ;           // 0 ≤ col < WIDTH and
} ;                        // 0 ≤ row < HEIGHT
struct Puzzle
{ Cell board [WIDTH][HEIGHT] ;
  Pos open ;
} ;
```

fixed size, so array is appropriate

redundant, $\mathcal{O}(1)$ access to EMPTY

Case study: sliding puzzle breadth-first search

- Keep track of all candidates:

```
struct Candidate
{ Puzzle candidate ;
  int parent_candidate ;
} ;
```

Case study: sliding puzzle breadth-first search

```
void solve (Puzzle start)
{   vector<Candidate> c = {{start,-1}} ;
    int i = 0 ;
    while (i < size (c) &&
              !puzzle_ready (c[i].candidate))
    {   Puzzle p = c[i].candidate ;
        if (can_go_north (p)) tries (c, i, north (p)) ;
        if (can_go_south (p)) tries (c, i, south (p)) ;
        if (can_go_west  (p)) tries (c, i, west  (p)) ;
        if (can_go_east  (p)) tries (c, i, east  (p)) ;
        i = i+1 ;
    }
    if (i < size (c))
        show_path (c, i) ;
}
```

```
void tries (vector<Candidate>& c, int i, Pos next)
{   Puzzle p = c[i].candidate ;
    Puzzle newp = move_empty (p, next);
    Candidate newc = {newp, i} ;
    if (!puzzle_present (c, i, newp))
        c.push_back (newc) ;
}
```

Case study: sliding puzzle breadth-first search

- Remaining programming tasks:

```
bool   puzzle_ready   (Puzzle p)
bool   puzzle_present(vector<Candidate>& c, int i, Puzzle p)
Puzzle move_empty     (Puzzle p, Pos next)
void   show_path      (vector<Candidate>& c, int i)
bool   can_go_north   (Puzzle p)
Pos    north          (Puzzle p)
bool   can_go_south   (Puzzle p)
Pos    south          (Puzzle p)
bool   can_go_west    (Puzzle p)
Pos    west           (Puzzle p)
bool   can_go_east    (Puzzle p)
Pos    east           (Puzzle p)
```

Case study: sliding puzzle
depth-first search

- Compute shortest solution:
  - use an upper bound on depth
  - keep track of attempt (initially challenge)
  - keep track of shortest solution (initially empty)

```
void solve ( vector<Puzzle>& attempt
           , vector<Puzzle>& shortest, int max_depth )
```

- Pre-condition:
  - attempt is not empty and max_depth ≥ 0
- Post-condition:
  - shortest contains a shortest solution that starts with attempt and is not longer than (max_depth+1)

Case study: sliding puzzle depth-first search

```
void solve ( vector<Puzzle>& attempt
           , vector<Puzzle>& shortest, int max_depth )
```

Base cases:

- size (shortest) > 0 $\wedge$ size (attempt) $\geq$ size (shortest): stop
- size (attempt) > max_depth+1: stop
- attempt is a solution: copy attempt to shortest

Recursive cases:

- for every possible direction:
  - if it does not occur in attempt:
    - *prepare:* add it to attempt
    - *recurse*
    - *repair:* remove it from attempt

Case study: sliding puzzle depth-first search


```
void solve ( vector<Puzzle>& attempt
           , vector<Puzzle>& shortest, int max_depth )
{  const int CURRENT = size (attempt) ;
   const int BEST    = size (shortest) ;
   Puzzle p          = attempt[CURRENT-1] ;
   if (BEST > 0 && CURRENT >= BEST)      { return ; }
   else if (CURRENT > max_depth+1)       { return ; }
   else if (puzzle_ready (p)) { shortest = attempt ; return ; }
   if (can_go_north (p)) tries (attempt, north (p), shortest, max_depth) ;
   if (can_go_south (p)) tries (attempt, south (p), shortest, max_depth) ;
   if (can_go_west  (p)) tries (attempt, west  (p), shortest, max_depth) ;
   if (can_go_east  (p)) tries (attempt, east  (p), shortest, max_depth) ;
}
```

```
void tries ( vector<Puzzle>& attempt,  Pos next
           , vector<Puzzle>& shortest, int max_depth )
{  Puzzle p    = attempt[size (attempt)-1] ;
   Puzzle newp = move_empty (p, next) ;
   if (!puzzle_present (newp, attempt))
   {   attempt.push_back (newp) ;   // prepare
       solve (attempt, shortest, max_depth) ;
       attempt.pop_back () ;        // repair
   }
}
```

Case study: sliding puzzle
depth-first search

- Remaining programming tasks:

```
✓ bool    puzzle_ready   (Puzzle p)
  bool    puzzle_present (Puzzle p, vector<Puzzle>& c)
✓ Puzzle  move_empty     (Puzzle p, Pos next)
  void    show_solution  (vector<Puzzle>& c)
✓ bool    can_go_north   (Puzzle p)
✓ Pos     north          (Puzzle p)
✓ bool    can_go_south   (Puzzle p)
✓ Pos     south          (Puzzle p)
✓ bool    can_go_west    (Puzzle p)
✓ Pos     west           (Puzzle p)
✓ bool    can_go_east    (Puzzle p)
✓ Pos     east           (Puzzle p)
```

already solved in breadth-first search solution

Solving search problems: aftermath[1]

- Solvable sliding puzzle:
  - inversion sum of all slides is even
  - inversion of a slide with value **c**:
    - the number of slides after this slide with value < **c**
  - inversion of the empty slide:
    - its row number (top row is 1)
- Known upperbound for 4×4 puzzle: 80

---

[1] http://mathworld.wolfram.com/15Puzzle.html

What have we done?

- Depth-first search versus breadth-first search
  - depth-first search is a recursive algorithm
  - breadth-first search is an iterative algorithm
  - share a similar collection of functions
- Case study: sliding puzzle