Artificial Intelligence Techniques for Mobile Robots

Teacher:	Alessandro Saffiotti Room: T-2224 Email: `alessandro.saffiotti@oru.se`
Lab assistant:	Ali Abdul Khaliq Room: T-1121 Email: `ali-abdul.khaliq@oru.se`

Course home page:

`http://aass.oru.se/~asaffio/Teaching/AIMR/`

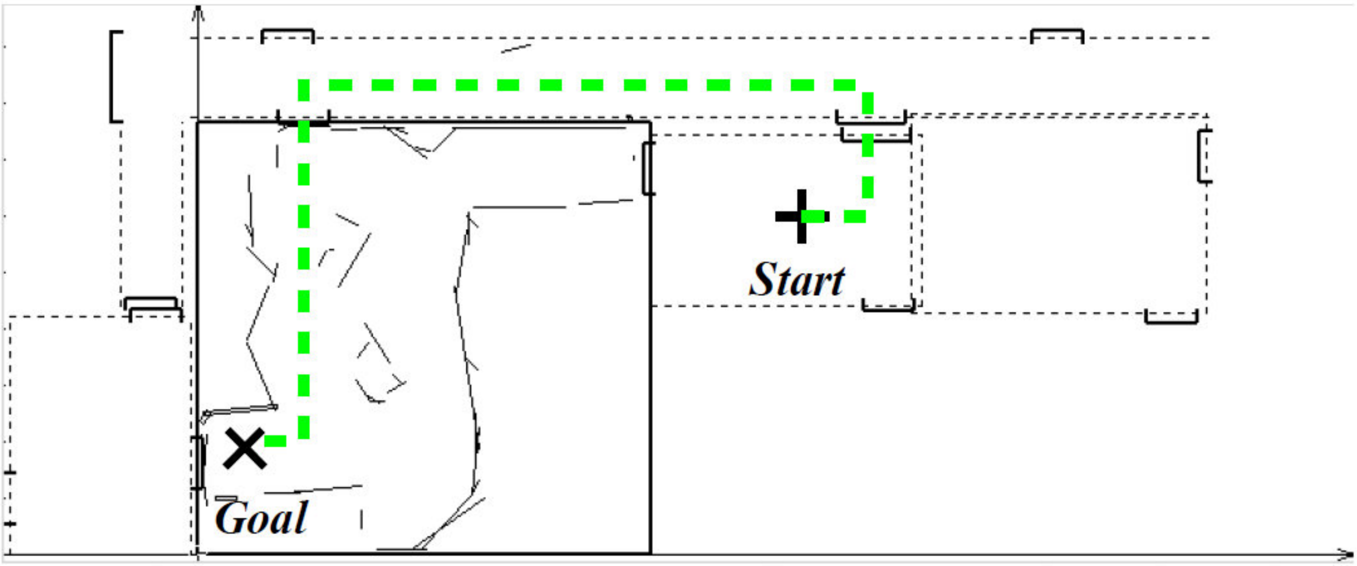
Path planning

Outline

- Searching
  - *the path planning problem*
  - *the search problem in Artificial Intelligence (AI)*
  - *path planning seen as a search problem*
- Breadth-first search
  - *intuition*
  - *the search algorithm* $\Leftarrow$ Lab
  - *path generation* $\Leftarrow$ Lab
- Other types of search
  - *depth-first search*
  - *heuristic search*
  - *A* search*
- Summary of types of search

Search

Recall: the planning problem

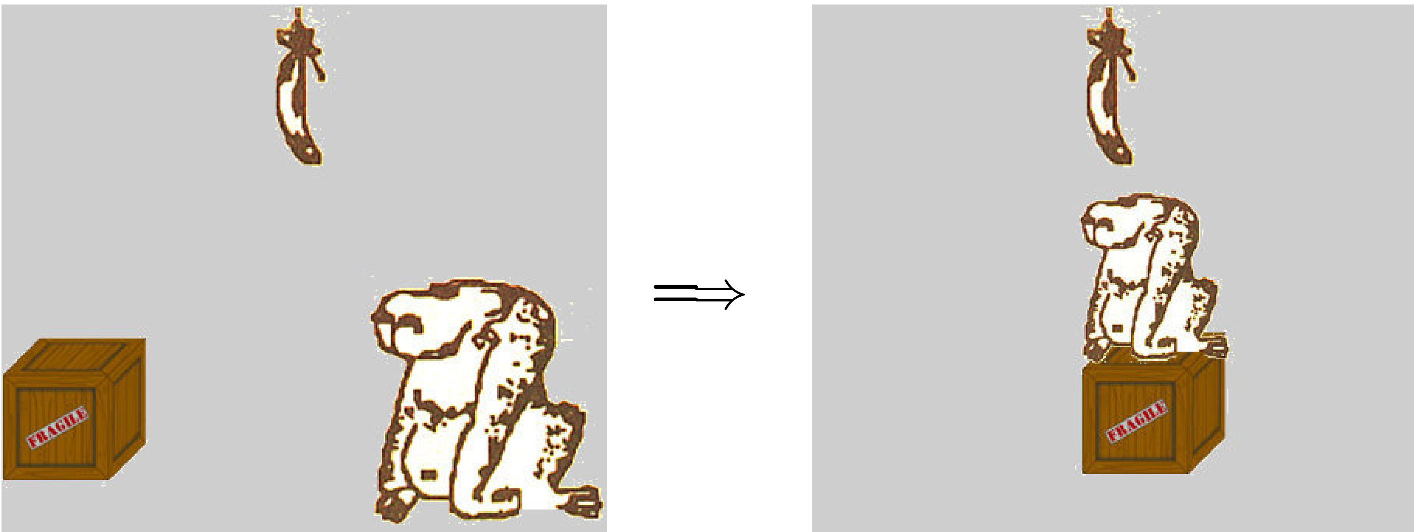


- Find a trajectory in the map that
  - *goes from start position to goal position*
  - *is collision free*
  - *has minimum length*

- In this lesson:
  - *assume full geometric decription of the environemnt*
  - *ignore the kinematic and dynamic contraints*
  - *length is only quality parameter (no time, energy, . . .)*
  - *ignore uncertainty*

The search problem

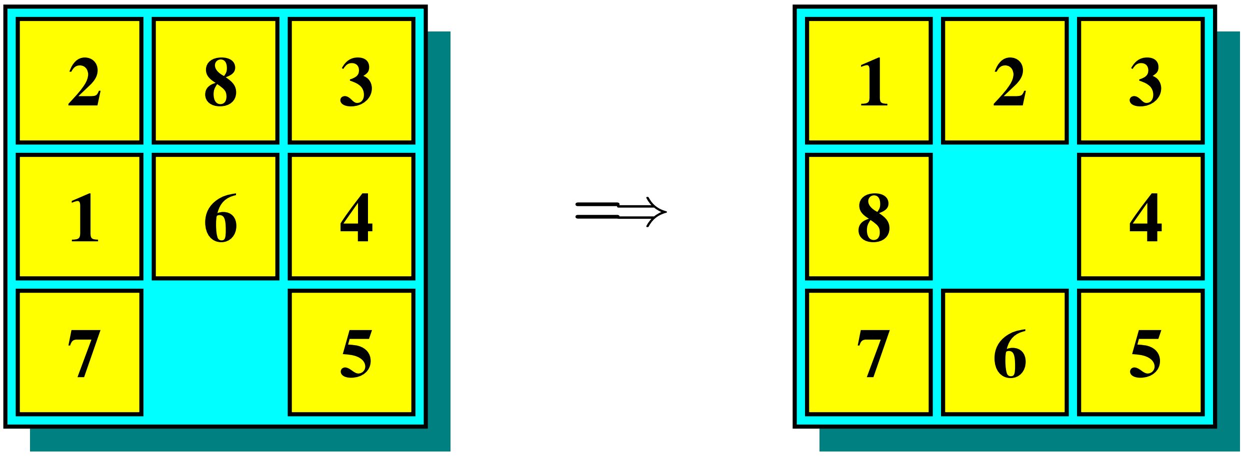
Problem solving as search in state space



- We want to reach a given <u>state</u> of the world...
- ...starting from an <u>initial</u> one...
- ...by applying state-tranforming <u>actions</u>
  - *'push box', 'climb box', 'grab banana'*

The search problem

Problem solving as search in state space



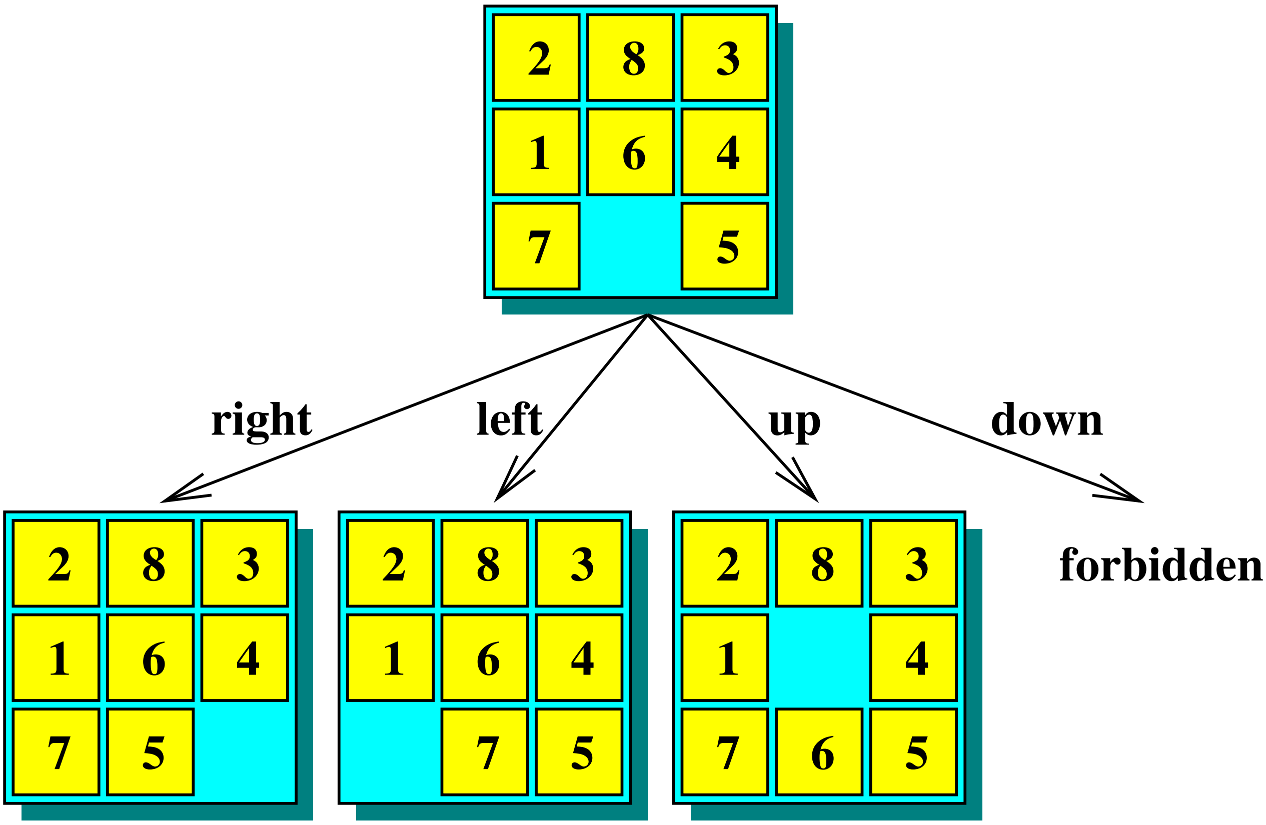
- We want to reach a given state of the world...
- ...starting from an initial one...
- ...by applying state-tranforming actions
  - *'move tile 6 down' , 'move tile 4 left' ...*

Formulating the Problem

- States: descriptions of the state of the world
  - *Must mention all and only the relevant aspects*
- Operators: how we move from state to state
  - *Must specify legal moves in each state*
  - *Must use abstract actions*
  - *Actions may have a cost*
- Goal: a set of states
  - *Can be one specific state ...*
  - *... or all the states satisfying a desired condition*
- Solution: a path from initial state to a goal state
  - *Encodes the sequence of actions to perform*
  - *We may want to find a minimal cost path*

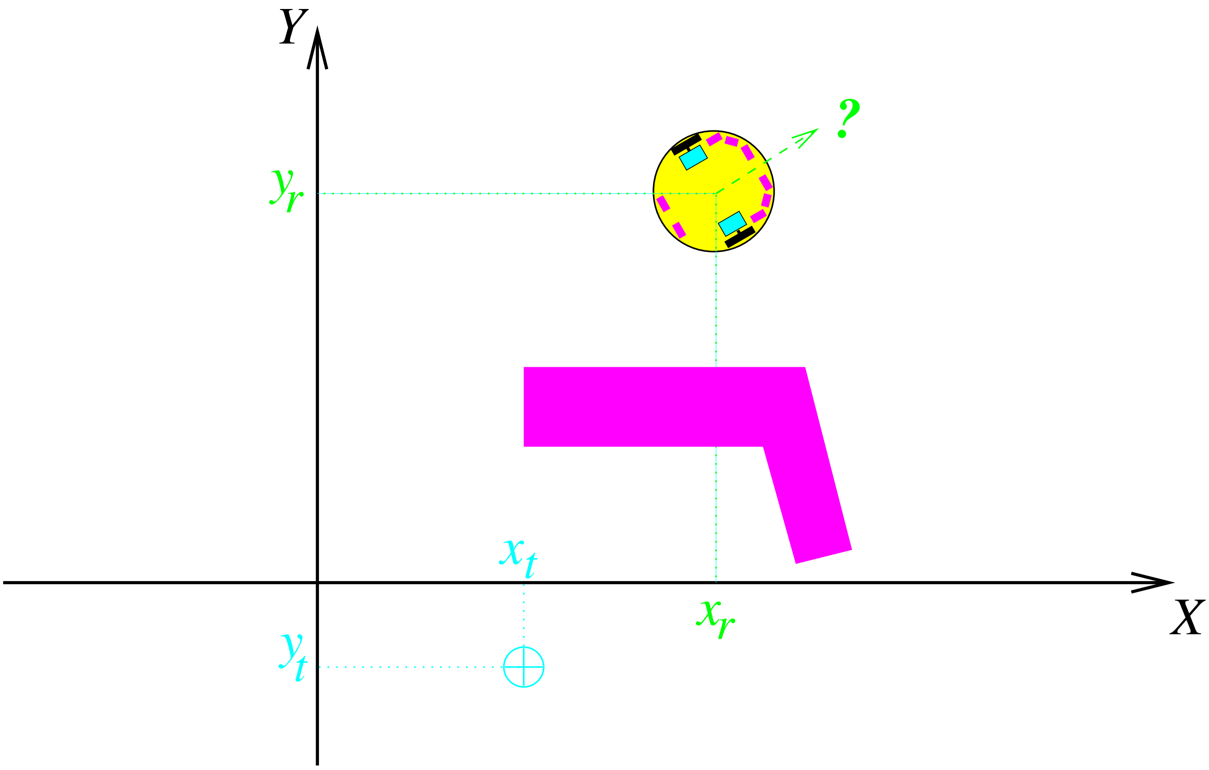
Search example

- States: 3×3 array or integers (0 for blank)
  - *# of nodes:* $9! = 362,880$ *nodes*
- Operators: 'move-1-up', 'move-1-left', ...
  - *branching factor:* $8 \times 4 = 32$
  - *better: 'move-blank-up', 'move-blank-left', ...*



- Goal: reach the given state

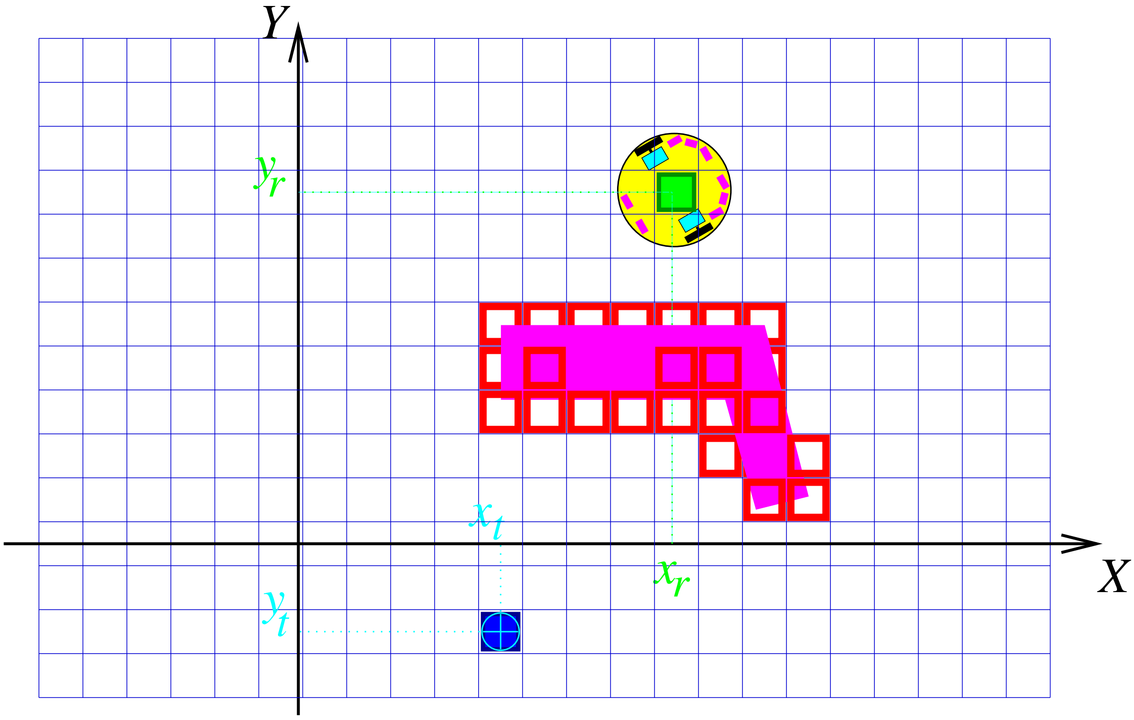
Search for robot navigation



- States:
  - *the* (x_r, y_r) *position of the robot*
- Operators:
  - *the possible atomoc moves of the robot*
  - *"possible" = do not collide with obstacles*
- Goal:
  - *the* (x_t, y_t) *position of the target*

But $\Rightarrow$ the state space is continuous!

Path planning as a search problem

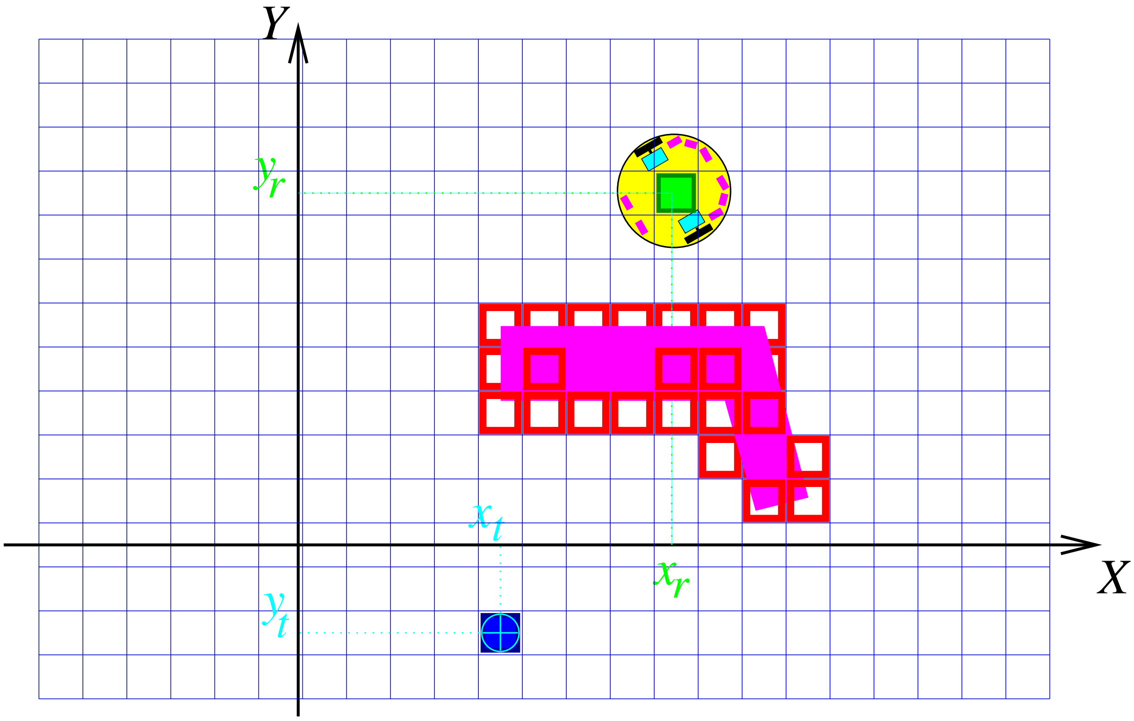


- States:
  - *grid decomposition of space*

- Operators:
  - *single step between cells in the grid*
  - *move according to 4-connectivity or to 8-connectivity*
  - *cells (partially) occupied by obstacle are forbidden*
  - *note: obstacles should be "grown" by one robot radius*

- Goal:
  - *reach a given cell in the grid . . .*
  - *. . . in a minimum number of steps*

Path planning: general strategy

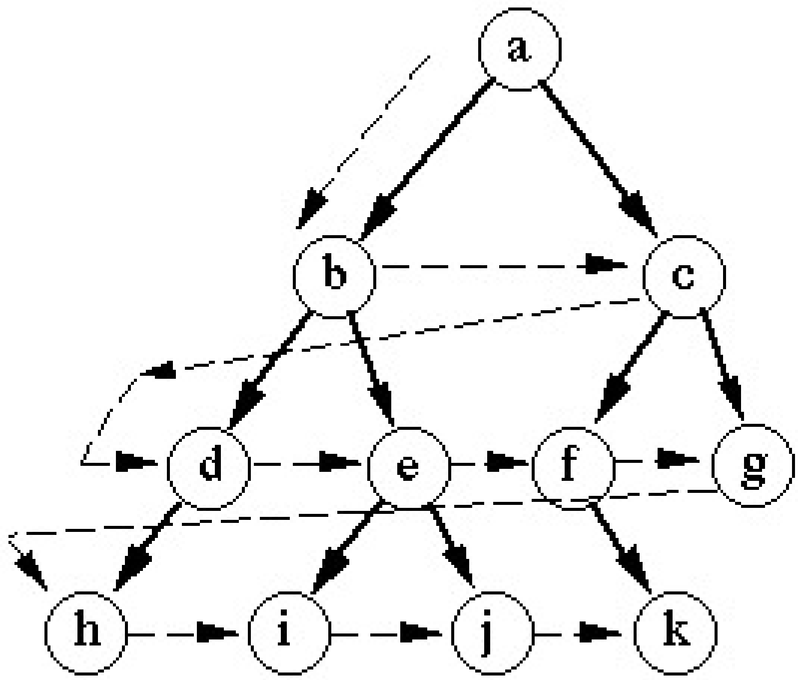


1. See what are your possible moves in current state
2. Chose and simulate execution of one move
3. If you are at the goal ⇒ stop
4. If you are in trouble ⇒ backtrack

Planning means to simulate execution in your head by trials-and-errors, before you go and execute the actions in the world.

Breadth-first search

Breadth-First Search

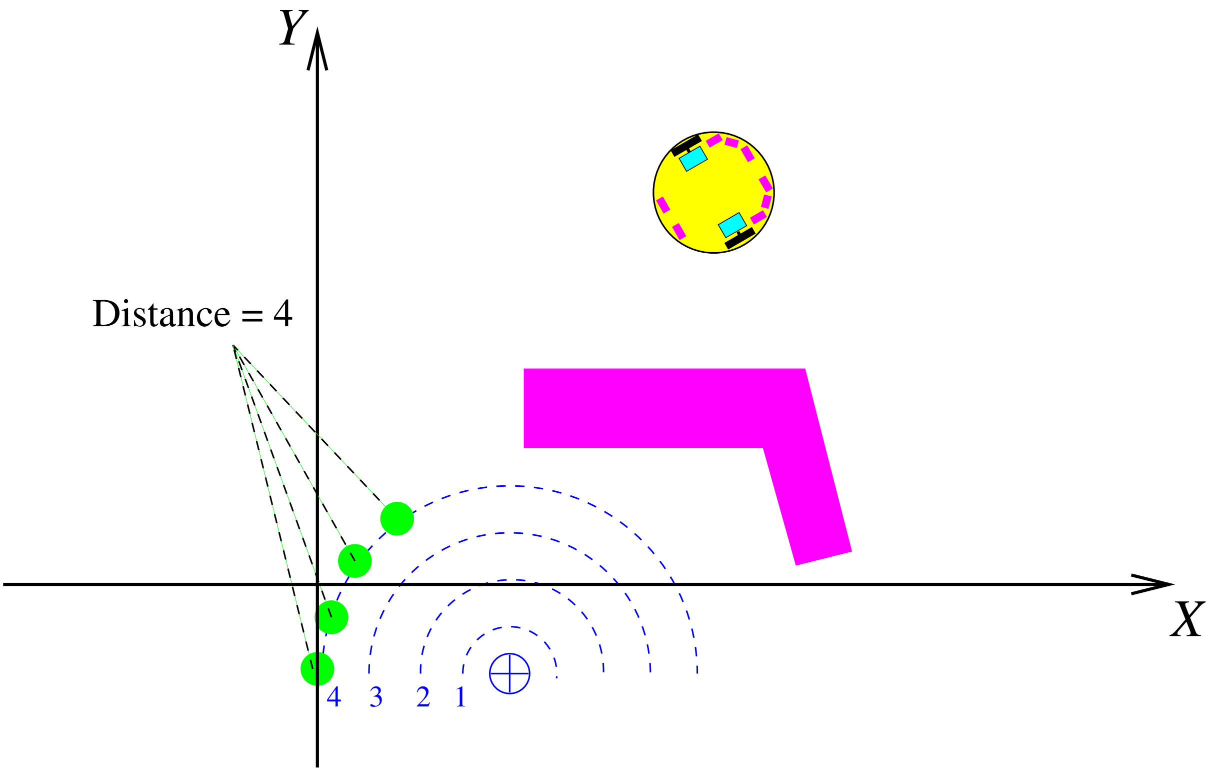


- Explore the space by layers
  - *all the nodes that can be reached in 1 move*
  - *then all the nodes that can be reached in 2 moves*
  - *then . . .*

- All nodes in a layer have same distance from root

- Can be done in either direction
  - *start from initial state until you hit the goal*
  - *start from goal until you hit initial state*
  - *or both (bi-directional) until the searches meet*

In our case . . .

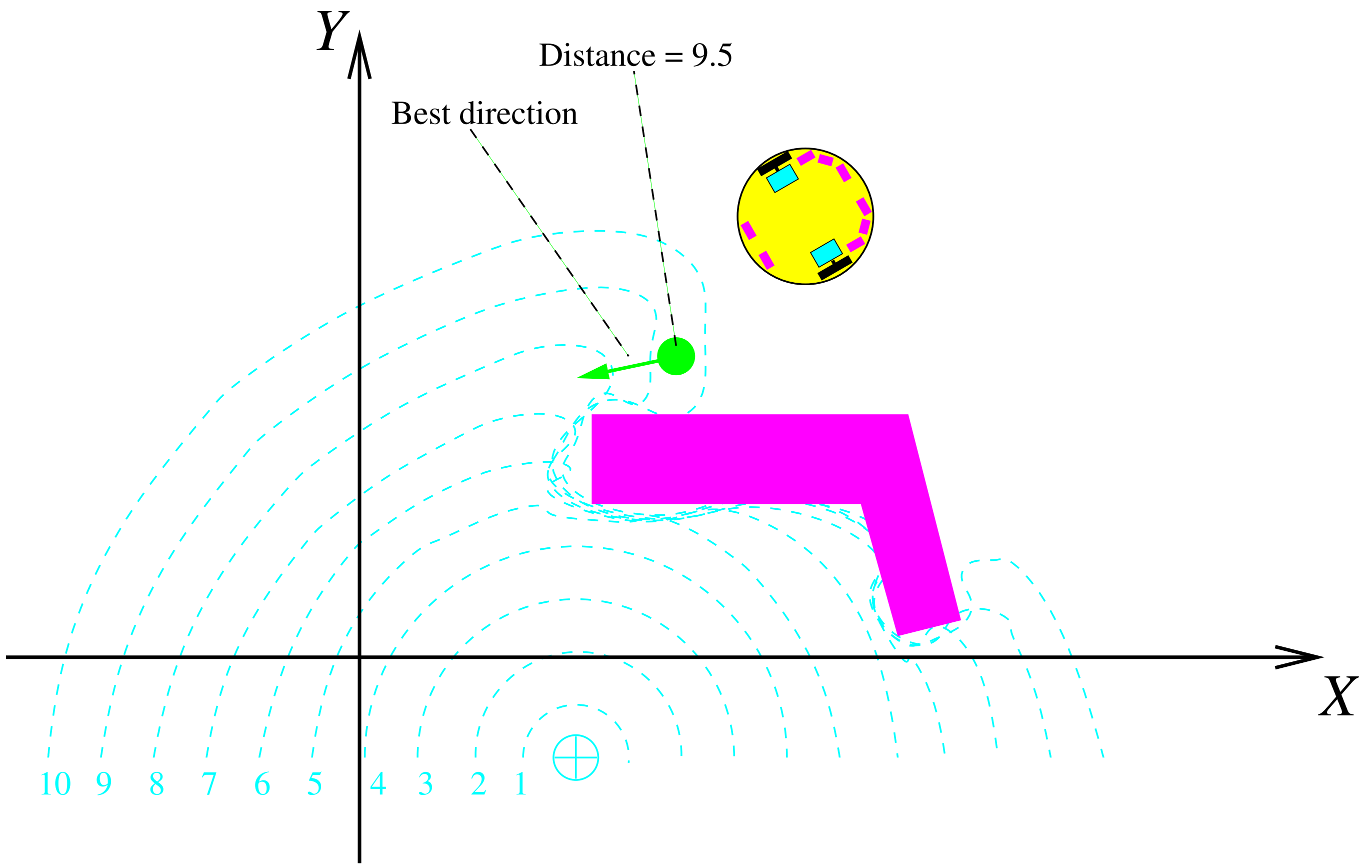


- Explore the space by layers
  - *all the cells that can be reached in 1 move*
  - *then all the cells that can be reached in 2 moves*
  - *then . . .*

- All cells in a layer have same distance from start

- Can be done in either direction
  - *start from robot until you hit the goal*
  - *start from goal until you hit the robot*
  - *or both (bi-directional) until the waves meet*

The Wavefront algorithm

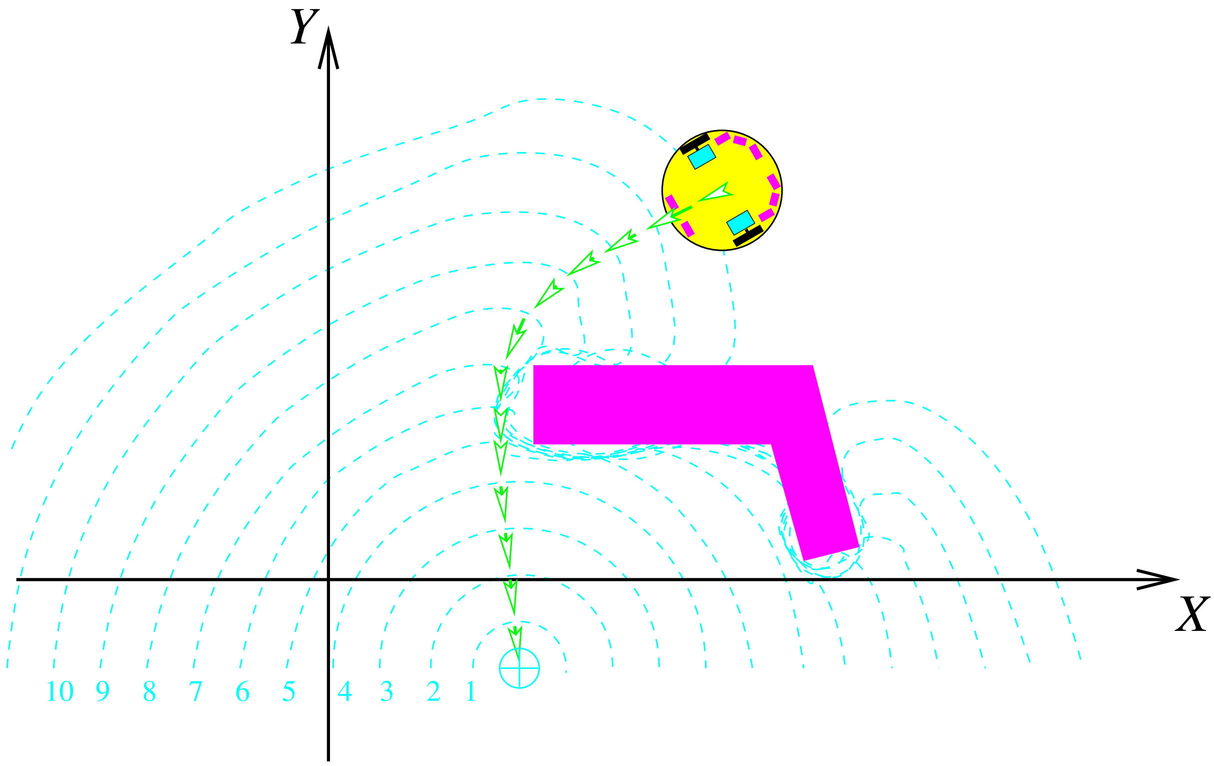


- Special case of Breadh-First Search
  - *propagate a "wave" from the goal*
  - *the wave goes around obstacles*
  - *the wave leaves a timestamp at each point*
  - *when the wave hits the robot, stop*

- Now, at each point:
  - *we know how far is the goal*
  - *we know which direction is best to go*

(the one with decreasing distance)

The result of the search



- We can keep the full <u>distance matrix</u>

- Or we can extract a <u>path</u>
  - *by looking at decreasing values*

- Discussion:
  - *what are the pros and cons of both?*

Implementation – data structures

cell: a 2-array (i,j) of indexes in the discrete grid

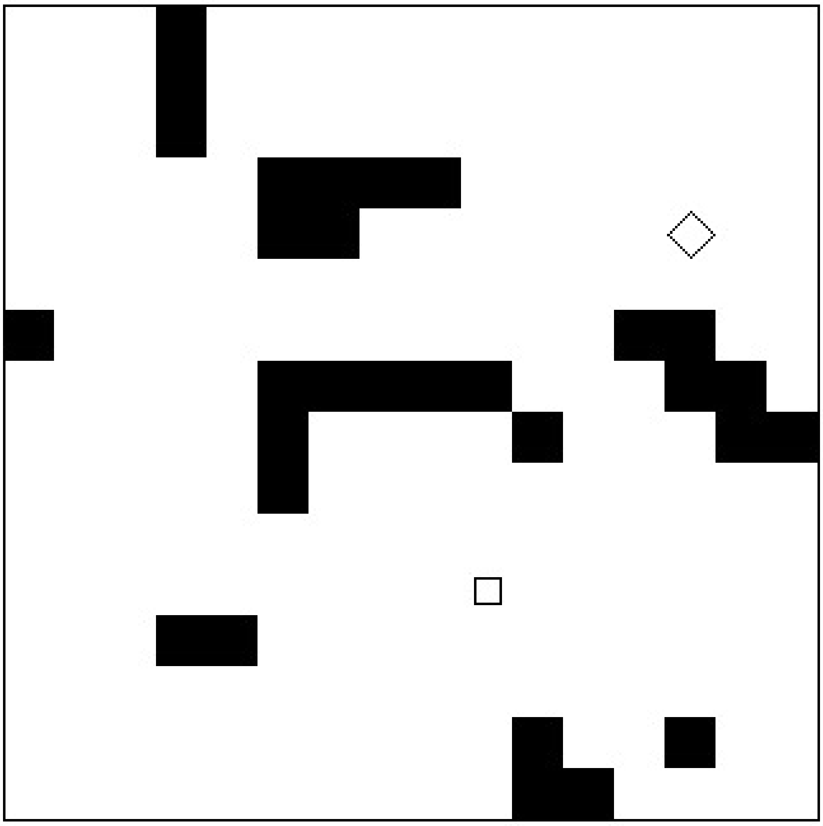
grid: N×N array of values for cells; each value in the grid is an integer with meaning:

$$\begin{cases} +x & \text{distance to goal is } x \text{ steps} \\ 0 & \text{goal cell} \\ -1 & \text{cell out of boundaries} \\ -2 & \text{cell has not been computed yet} \\ -3 & \text{obstacle cell} \\ -4 & \text{initial position} \end{cases}$$

(Note: markers in the files maps.c / maps.h might be different)

queue: list of cells that must be explored next details on this in a few moments . . .

The grid at start

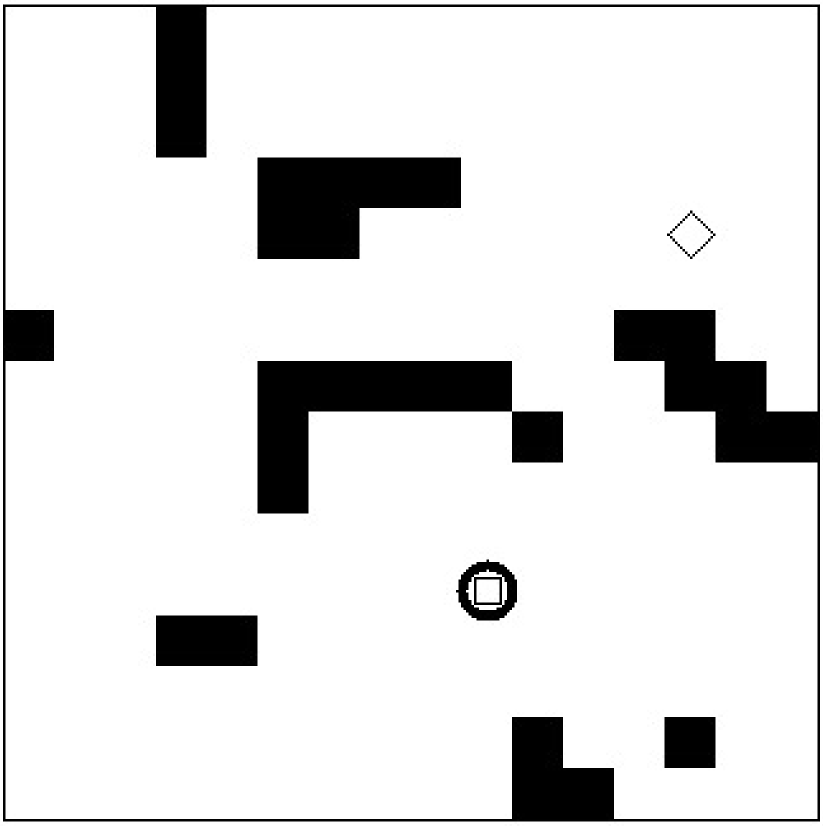


```
[0,0]->    -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1    <-[0,15]
           -1 -2 -2 -3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -2 -2 -3 -3 -3 -3 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -2 -2 -3 -3 -2 -2 -2 -2 -2 -2 -4 -2 -1
           -1 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3 -3 -2 -1
           -1 -2 -2 -2 -2 -3 -3 -3 -3 -3 -2 -2 -2 -3 -3 -1
           -1 -2 -2 -2 -2 -3 -2 -2 -2 -2 -3 -2 -2 -2 -3 -1
           -1 -2 -2 -2 -2 -3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -2 -2 -2 -2 -2 -2  0 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -3 -3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -1
           -1 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3 -2 -2 -3 -2 -1
[15,0]->   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1    <-[15,15]
```

Implementation – main concept

procedure Search ()
    put the goal cell in the queue
    repeat until *the queue is empty*
[*]   *take a cell* c *from the queue*
        if c *is the robot cell* then return(success)
        foreach n *which is a neighbor cell of* c
            if n *is not an obstacle* and n *has label* -2
                label n *by* grid(c) + 1
[*]   *insert* n *in the queue*
    if *the queue is empty* then return(failure)
end

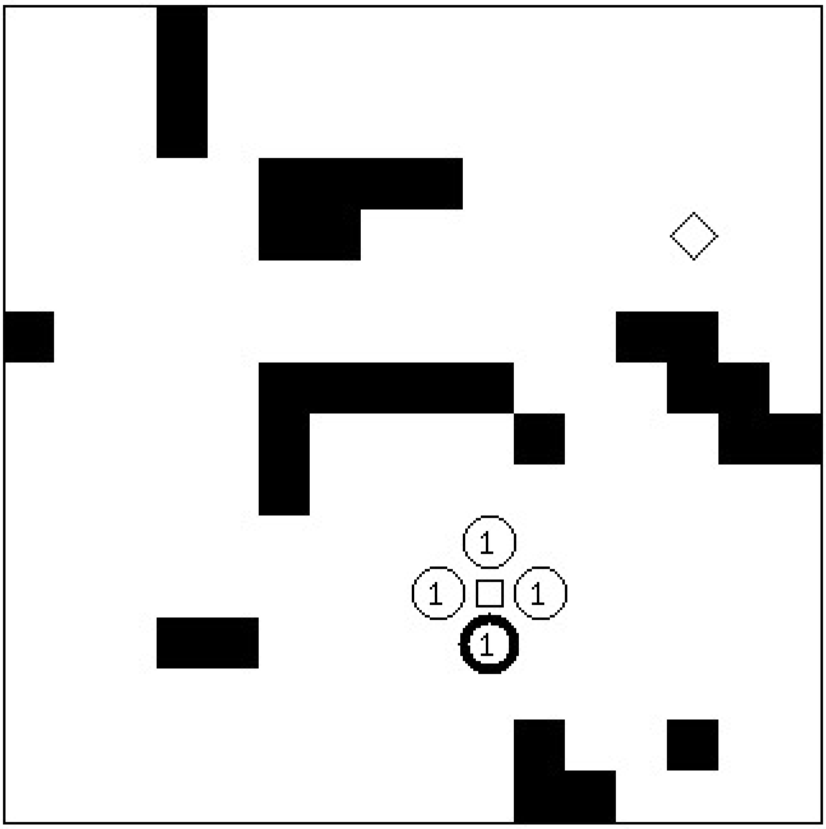
[*] depending on how we perform these operations we obtain search strategies



Example – step 2

procedure Search ()
    put the goal cell in the queue
    repeat until *the queue is empty*
[*]   *take a cell c from the queue*
        if c *is the robot cell* then return(success)
        foreach n *which is a neighbor cell of* c
            if n *is not an obstacle* and n *has label* -2
                label n *by* grid(c) + 1
[*]   *insert* n *in the queue*
    if *the queue is empty* then return(failure)
end

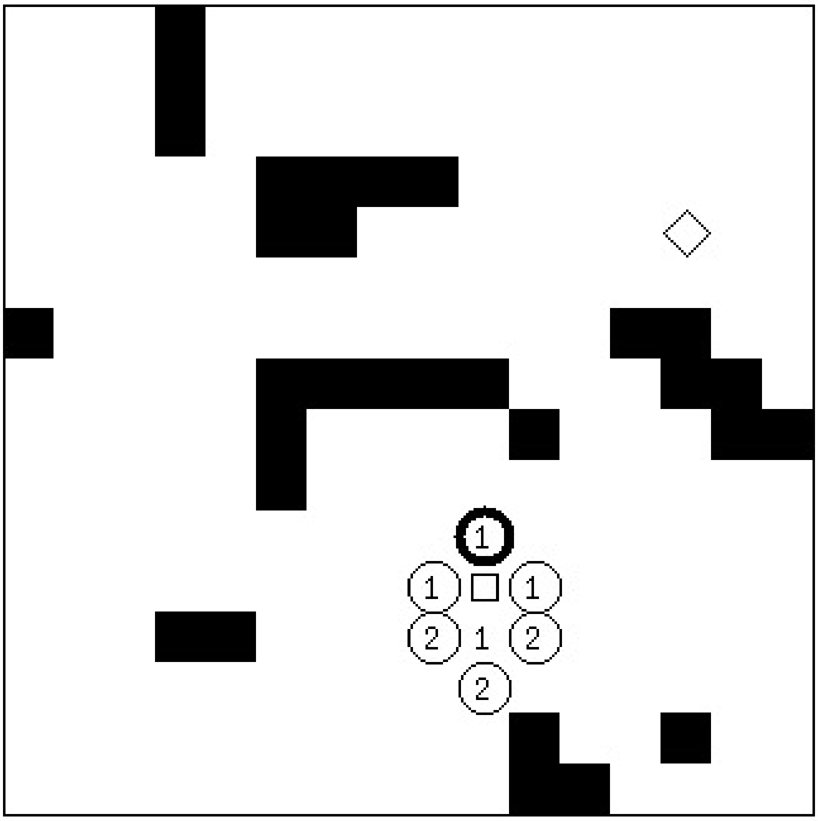
[*] depending on how we perform these operations we obtain search strategies



Example – step 3

procedure Search ()
    put the goal cell in the queue
    repeat until *the queue is empty*
[*]   *take a cell c from the queue*
        if *c is the robot cell* then return(success)
        foreach n *which is a neighbor cell of* c
            if n *is not an obstacle* and n *has label* -2
                label n *by* <u>grid</u>(c) + 1
[*]   *insert* n *in the queue*
    if *the queue is empty* then return(failure)
end

[*] depending on how we perform these operations we obtain <u>search strategies</u>



Implementation – functions

```
procedure Search ([goal_i,goal_j])
  clear the queue
  push the goal cell into the queue
  while (queue is not empty)
     [i,j] := pop_queue(queue)     // take a cell from queue
     if ([i,j] is robot's start position)
        return
     // MarkCell also does insertion into the queue //
     dist := grid([i,j]) + 1
     MarkCell([i,j-1], dist)
     MarkCell([i,j+1], dist)
     MarkCell([i-1,j], dist)
     MarkCell([i+1,j], dist)
  end while
end
```

```
procedure MarkCell (Cell [i,j], int value)
  case grid([i,j]) of
    -1: // cell out of boundaries: do nothing //
        return
    -2: // unexplored cell: mark it and put it into queue //
        grid([i,j]) := value
        push [i,j] onto the queue
        return
    -3: // obstacle cell: do nothing //
        return
    -4: // initial position: put it into queue //
        push [i,j] onto the queue
        return
  else: // already explored cell, do nothing //
        return
  end case
end
```

Implementation – the queue

```
structure Q_Element
  int i, j;
  Q_Element *next;
end

structure Queue
  Q_Element *head
  Q_Element *tail
  Q_Element *hog
  Q_Element element[QMaxNum]
end

procedure clear_queue (Queue q)
  for i from 0 to (QMaxNum-1)
    q.element[i] := &(q.element[i+1])
  q.element[QMaxNum-1] := null
  q.head := null
  q.tail := null
  q.hog  := &(q.element[0])
end

procedure get_el_queue (Queue q)
  p := q.hog
  q.hog  := p->next
  return(p)
end

procedure free_el_queue (Q_Element el, Queue q)
  el.next := q.hog
  q.hog := &el
end
```

Implementation – the queue

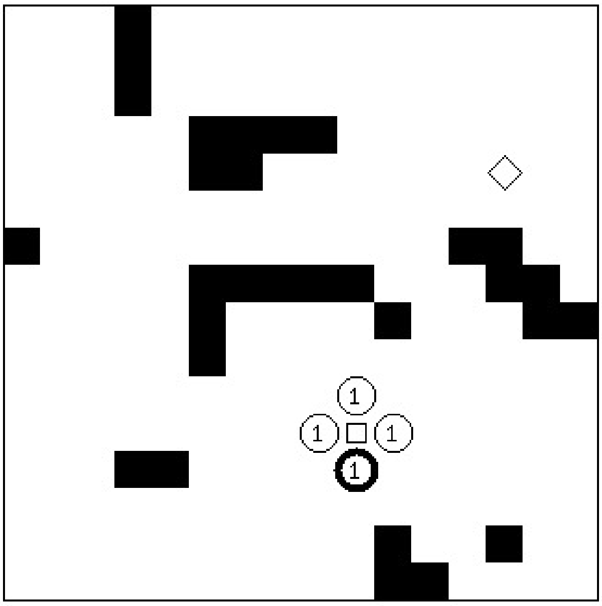
```
procedure empty_queue (Queue q)
  return(q.head = null)
end

procedure push_queue (Cell [i,j], Queue q)
  el := new_el_queue(q)
  if (el = null) return(-1)
  el.i := i
  el.j := j
  el.next := nil
  if (q.tail) q.tail->next := &el
  q.tail := &el
  if (q.head = nil) q.head := &el
end

procedure pop_queue (Queue q)
  el := q.head
  if (el = null) return(-1)
  q.head := el->next
  if (q.head = nil) q.tail := nil
  free_el_queue(el)
  return(el)
end
```

Note: $\boxed{\text{First-In-First-Out (FIFO) policy.}}$

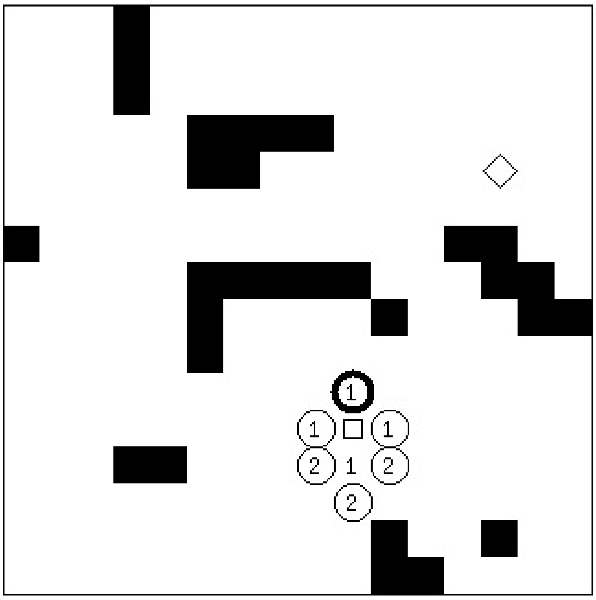
The example again



Queue: { [12,9] [10,9] [11,8] [11,10] }

-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-3	-3	-3	-3	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-3	-3	-2	-2	-2	-2	-2	-2	-4	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-3	-2	-2
-2	-2	-2	-2	-2	-3	-3	-3	-3	-3	-2	-2	-2	-3	-3	-2
-2	-2	-2	-2	-2	-3	-2	-2	-2	-2	-3	-2	-2	-2	-3	-3
-2	-2	-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	1	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	1	0	1	-2	-2	-2	-2	-2
-2	-2	-2	-3	-3	-2	-2	-2	-2	1	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-2	-2	-3	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-3	-2	-2	-2	-2

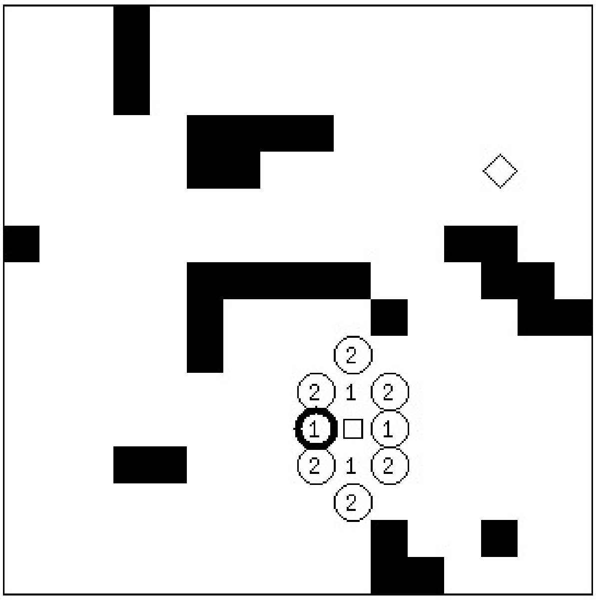
Step 2



Queue: {[10,9] [11,8] [11,10] [13,9] [12,8] [12,10]}

-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-3	-3	-3	-3	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-3	-3	-2	-2	-2	-2	-2	-2	-4	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-3	-2	-2
-2	-2	-2	-2	-2	-3	-3	-3	-3	-3	-2	-2	-2	-3	-3	-2
-2	-2	-2	-2	-2	-3	-2	-2	-2	-2	-3	-2	-2	-2	-3	-3
-2	-2	-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	1	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	1	0	1	-2	-2	-2	-2	-2
-2	-2	-2	-3	-3	-2	-2	-2	2	1	2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-2	-2	-3	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-3	-2	-2	-2	-2

Step 3

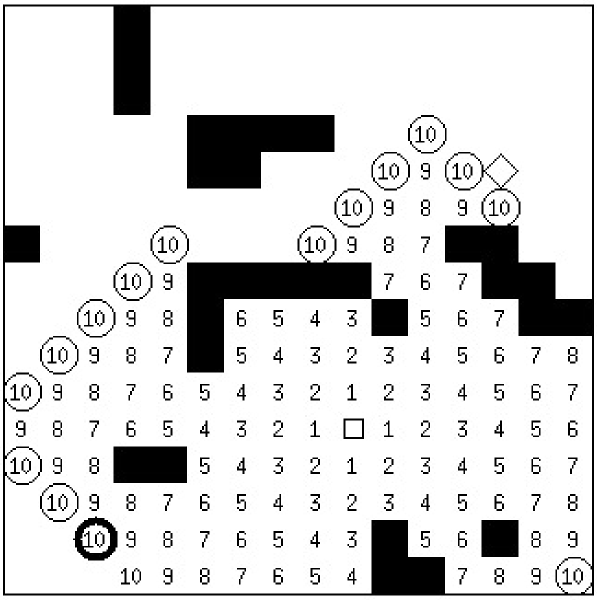


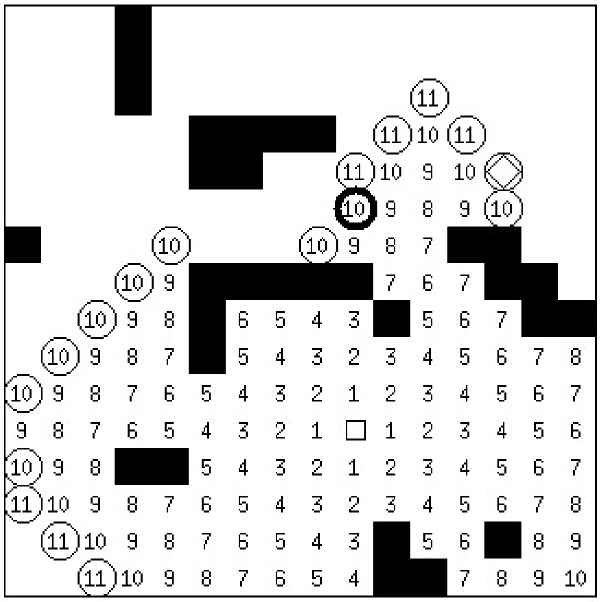
Queue: {[11,8] [11,10] [13,9] [12,8] [12,10] [9,9] [10,8] [10,10]}

-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-3	-3	-3	-3	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-3	-3	-2	-2	-2	-2	-2	-2	-4	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-3	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-3	-2	-2
-2	-2	-2	-2	-2	-3	-3	-3	-3	-3	-2	-2	-2	-3	-3	-2
-2	-2	-2	-2	-2	-3	-2	-2	-2	-2	-3	-2	-2	-2	-3	-3
-2	-2	-2	-2	-2	-3	-2	-2	-2	2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	2	1	2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	1	0	1	-2	-2	-2	-2	-2
-2	-2	-2	-3	-3	-2	-2	-2	2	1	2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-2	-2	-3	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-3	-3	-2	-2	-2	-2

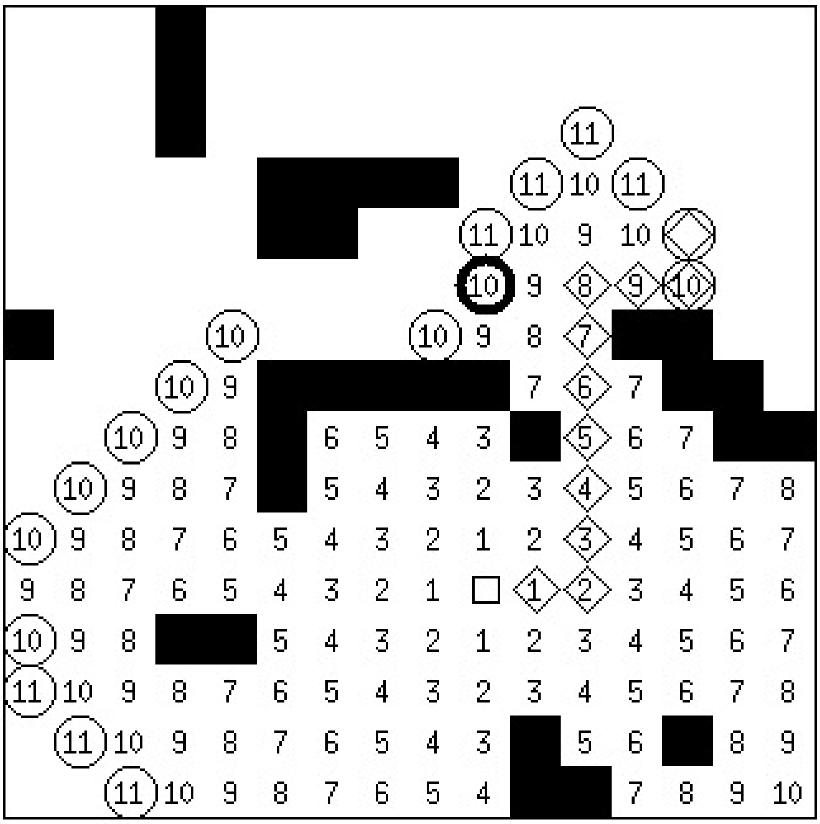
The next steps . . .

And the final ones





The found path



The algorithm – top level

```
procedure Plan (Cell [start_i,start_j], Cell [goal_i,goal_j])
  begin

    // we first do the search //
    read_the_environment_grid();
    Search([goal_i,goal_j]);

    // we now generate the trajectory //
    // begin with the start point //
    [i,j] := [start_i,start_j];
    print([i,j]);

    repeat
      // search the neighbor with the lowest value //
      bestvalue := maxinteger;
      for [i',j'] in neighbours([i,j]) do
        if (grid([i',j']) < bestvalue) then
           bestvalue := grid([i',j']);
           [best_i,best_j] := [i',j'];
           end if
        end for

      // print it as the next step in the trajectory //
      print([best_i,best_j]);

      // move to that point //
      [i,j] := [best_i,best_j];

    // keep going until we reach the goal //
    until grid([i,j]) = 0;

  end
```

Properties of BFS

Pros in general

- *Always finds a solution (if branching factor b is finite)*
- *Finds optimal solution (if costs are all equal)*

Cons in general

- *Does not take different costs into account*
- *Time: $O(b^d)$, with d = depth*
- *Keeps all nodes in memory: $1 + b + b^2 + \cdots + b^d$*

And in our case?

If you want to execute the path

- You need to store the path
  - *you can use the queue:*
  - *clear the queue before after the search*
  - *replace "print" by "push"*

- You need to transform $[i, j]$ into $[x, y]$
  - *decide the size of a cell, e.g., 20 mm*
  - *decide reference system (start point of robot)*

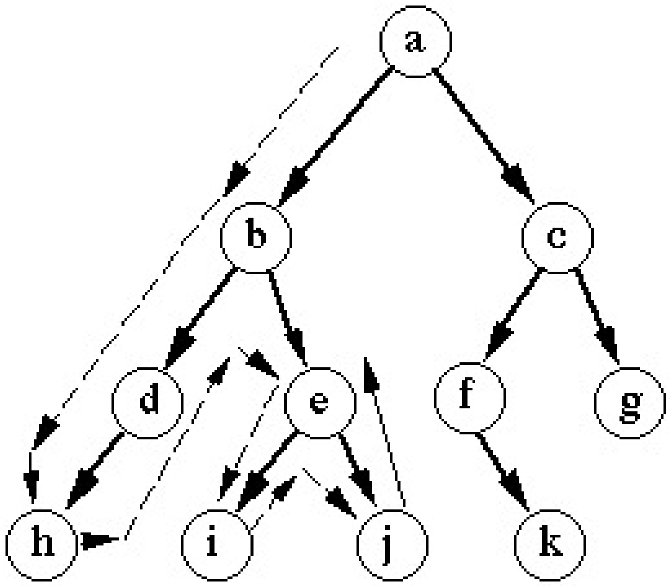
- You give the path to your robot
  - *use GoTo on all points one after the others*
  - *may need to make the way-points more sparse*

- Example:

Planned path	Way-points
$[10, 13]$	$(0, 0)$
$[10, 12]$	
$[10, 11]$	$(-40, 0)$
$[9, 11]$	
$[8, 11]$	$(-40, 40)$
$[7, 11]$	
$[6, 11]$	$(-40, 80)$
$[5, 11]$	
$[5, 12]$	$(-20, 100)$

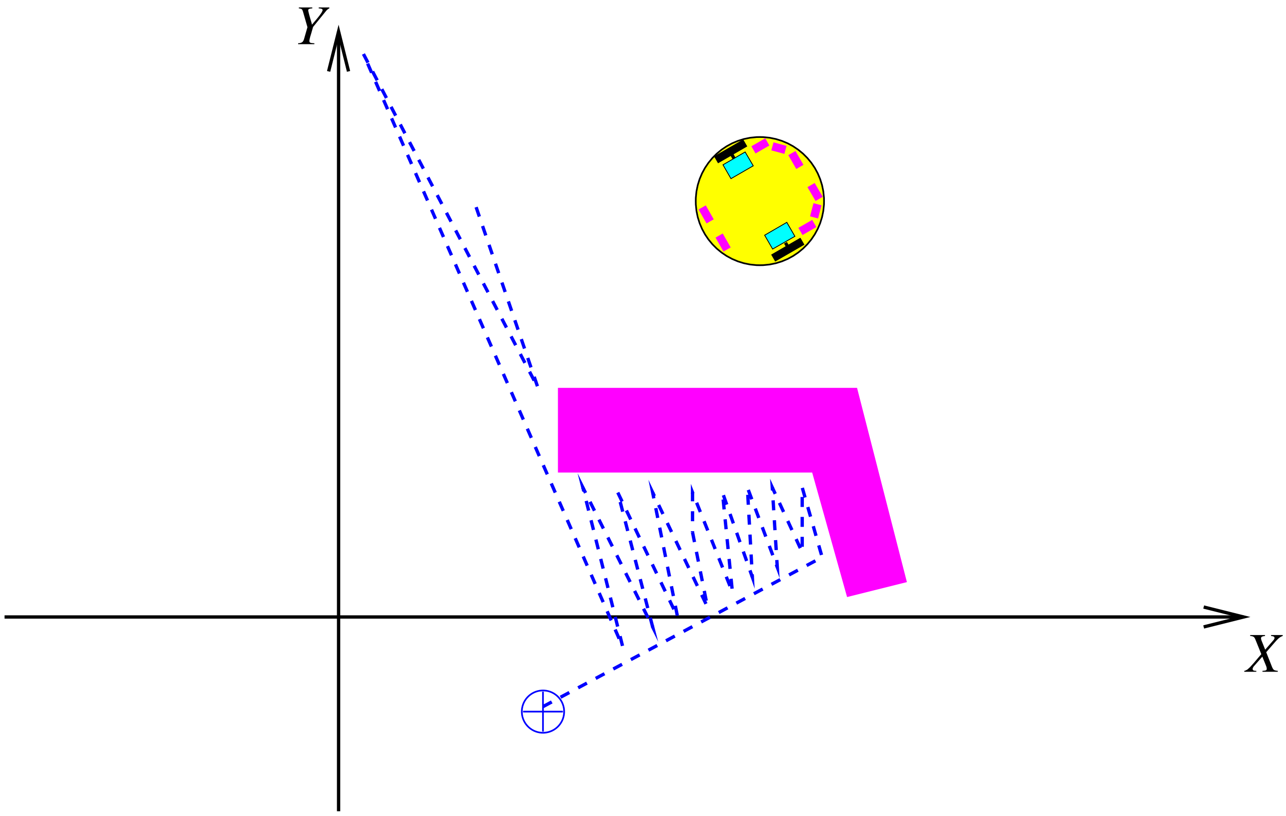
Other Types of Search

Depth-first search



- Can start from goal or from initial state
- Explore the space head-on until in trouble
  - *go all the way down a branch*
  - *when cannot go further, backtrack to most recent choice*
- "Artistic" behavior
  - *if lucky, it will go quickly to the goal*
  - *if unlucky, it will wander for ages before getting there*
- If space is infinite, it may never stop

Depth-first search



- Can start from goal or from robot
- Explore the space head-on until in trouble
  - *go all the way down in a direction*
  - *when cannot go further, return to most recent alternative*
- "Artistic" behavior
  - *if lucky, it will go quickly to the robot (or goal)*
  - *if unlucky, it will wander for ages before getting there*
- Eventually, it will hit the robot

Depth-First Search algorithm

procedure Search ()
  push the goal cell onto the **stack**
  repeat until *the* **stack** *is empty*
    pop a cell c *from the* **stack**
    if c *is the robot cell* then return(success)
    foreach n *which is a neighbor cell of* c
      if n *is not an obstacle* and n *has label* -2
        label n *by* grid$(c) + 1$
        push n *onto the* **stack**
  if *the* **stack** *is empty* then return(failure)
end

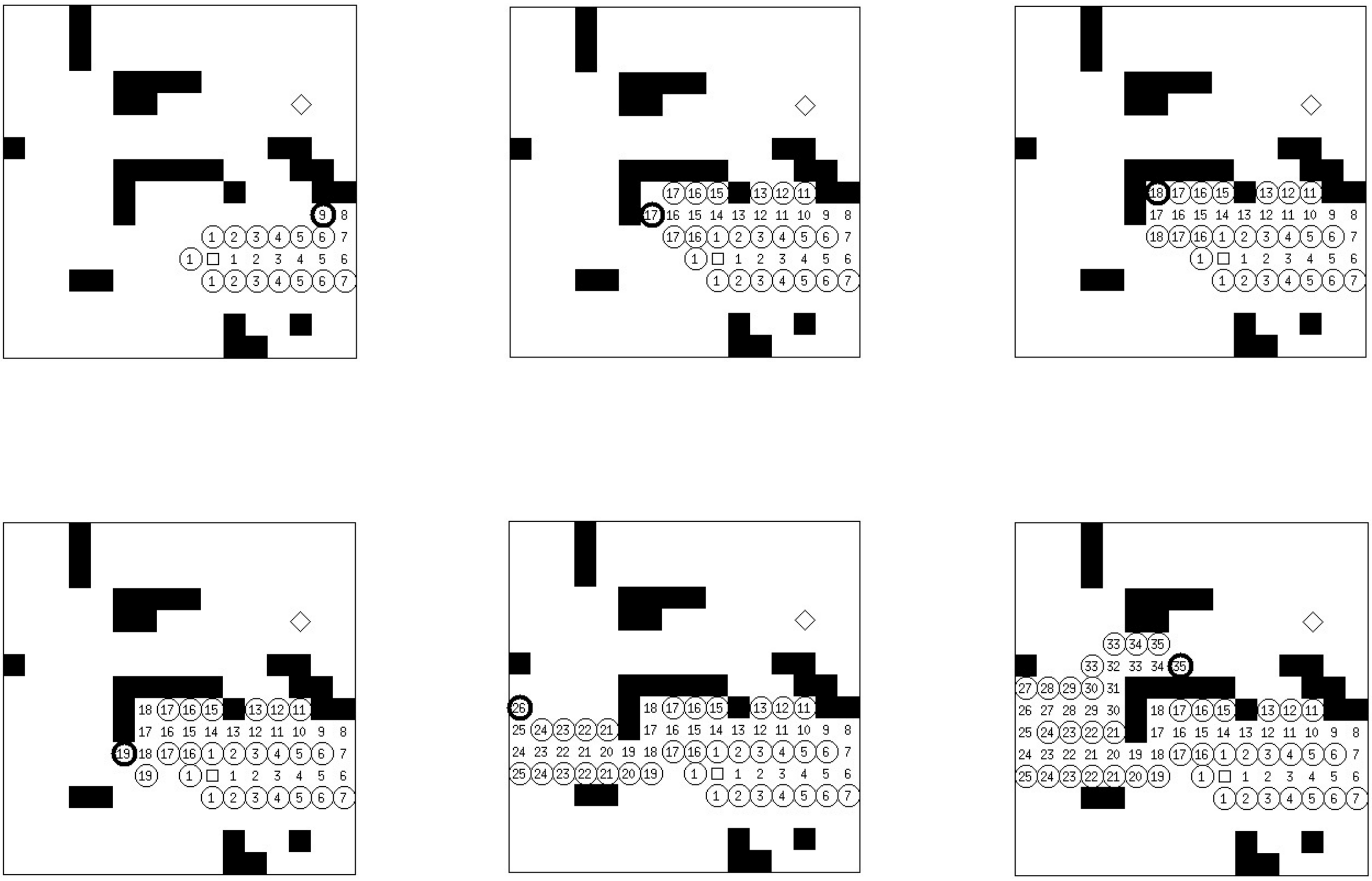
Note: We use a Last-In-First-Out (LIFO) policy.

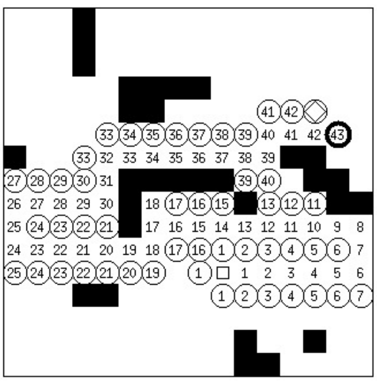
We need to make only one change to our program: use a different "push_queue"

```
procedure push_queue_lifo (Cell [i,j], Queue q)
  el := new_el_queue(q)
  if (el = null) return(-1)
  el.i := i
  el.j := j
  el.next := q.head
  q.head := &el
  if (q.tail = nil) q.tail := &el
end
```

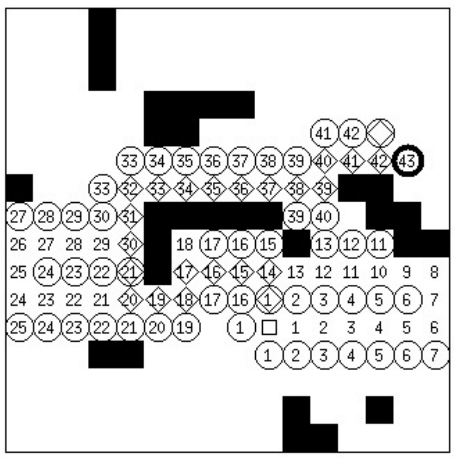
Example of depth-first search

Example (cont'd)





and the final path:



Properties of DFS

Pros in general

– *If m = max depth of search tree...*

– *Keeps only a $O(bm)$ nodes in memory (linear space)*

– *Always finds a solution (if exists) <u>in finite graphs</u>*

Cons in general

– *Time: $O(b^m)$ (very bad if $m \gg$ depth of solution)*

– *Can be trapped in infinite loops*

– *Can go astray on a wrong branch in infinite graphs*

And in our case?

Improvement: Iterative-Deepening Search

– *Strategy: do not expand nodes beyond depth D*

– *Increase D when the subtree is fully explored*

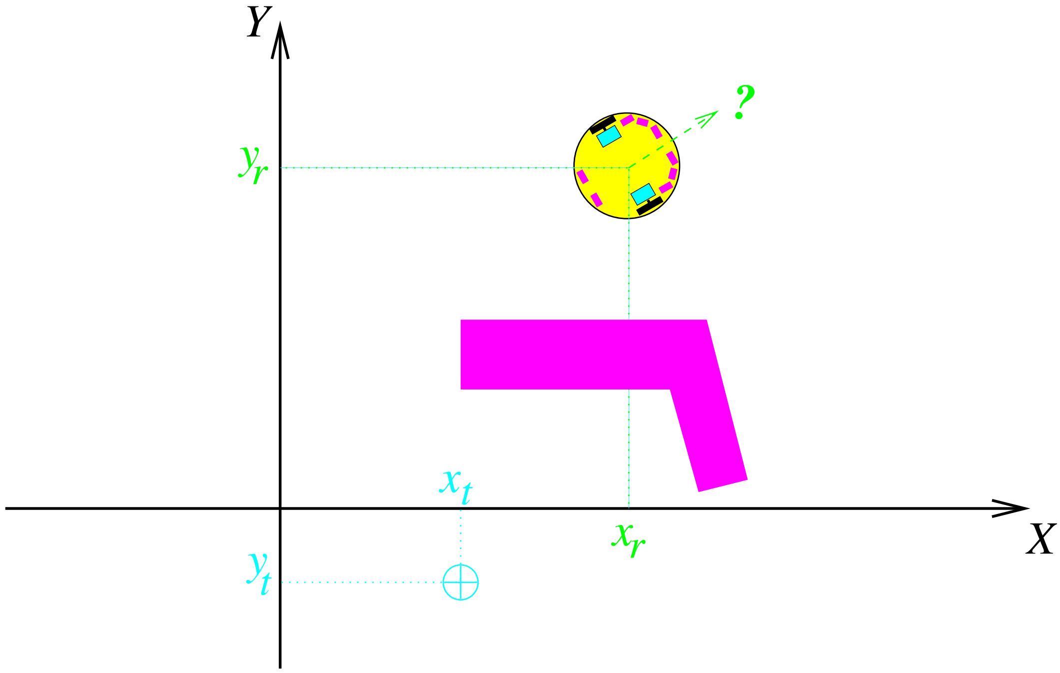
– *Is complete*

– *Time: $O(b^d)$ with d depth of solution (nice)*

– *Space: $O(b^d)$ (nice)*

– *Produces optimal solution if costs are all equal*

Limits of “uninformed” search



- Both BFS and DFS explore the space “blindly”
  - *do not use any information about the goal*
  - *except recognizing when we are at the goal*

- Breadth-First Search
  - *explores all directions in parallel*
  - *tends to explore a huge number of cells*

- Depth-First Search
  - *explores one direction at the time “to the end”*
  - *needs luck in selecting a good direction*

Heuristic search

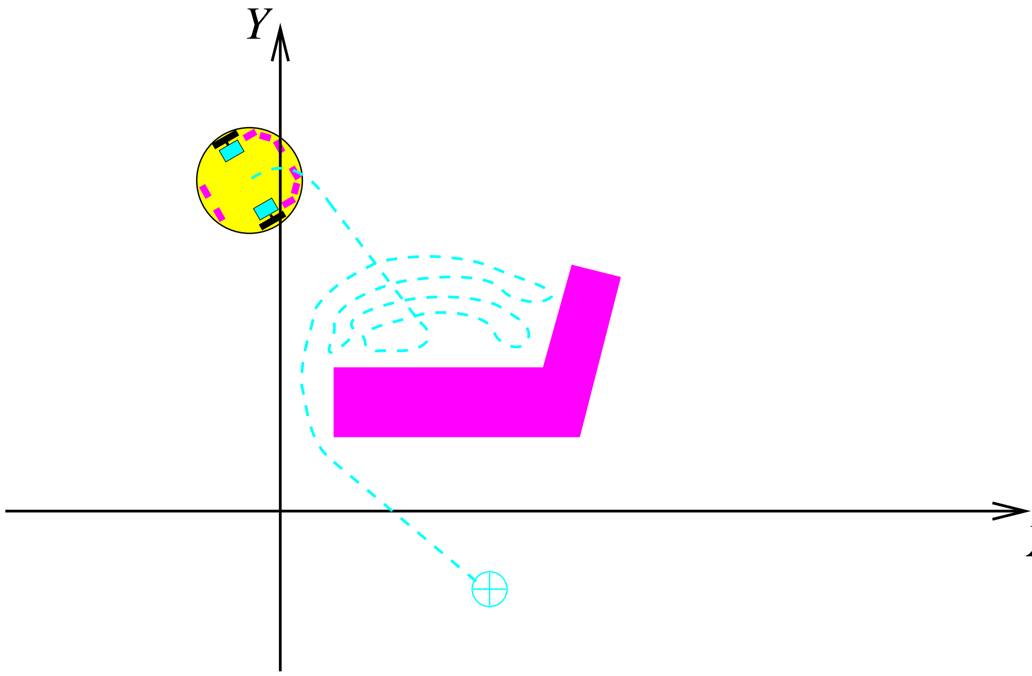
- General idea
  - *guide search towards most promising directions*
  - *use information about the domain to chose the most promising node to visit next*
  - *"promising" = value of heuristic function $h(n)$*

- The "heuristic" word
  - *something in our experience suggests that we should do it in this way, but we don't know why*
  - *cf. Rhys Kealley's definition of a hack:*

  > A hack is a heuristically appealing idea or approach which seems to work well in the absence of any sound background theory.

- Examples of heuristic functions in our domain:
  - h = *line-of-flight distance to goal*
  - h = *greater between x and the y distance to the goal*

Greedy search



- Heuristic function $h(c)$ = distance from c to goal
  - *in our case: Euclidean distance*

- Search goes depth-first
  - *first explore cells c that reduce $h(c)$*
  - *when blocked by an obstacle, explore cells around*

- Pros
  - *always find a solution (if exists)*
  - *can be fast*

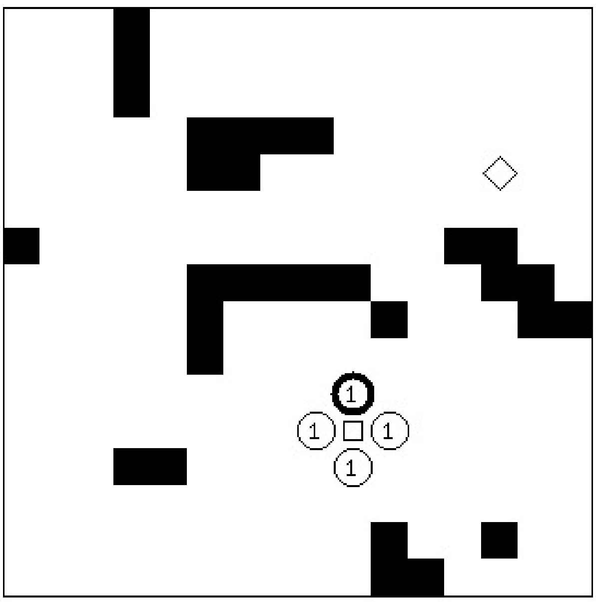
- Cons
  - *efficiency critically depends on the choice of h*

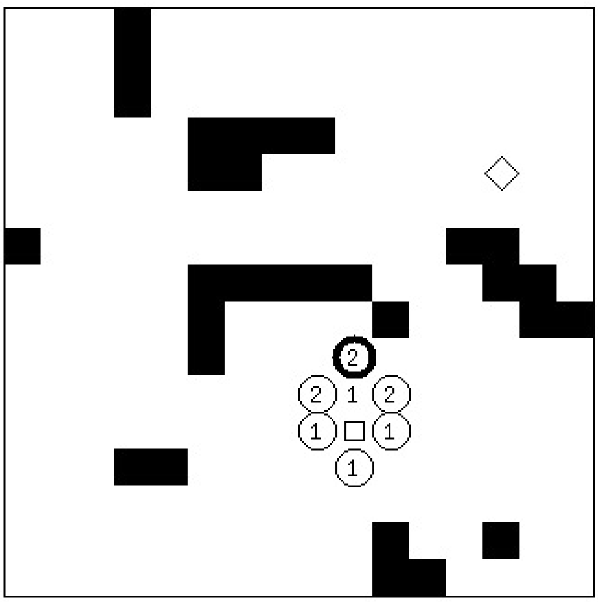
Heuristic Search algorithm

procedure Search ()
    put the goal cell in the queue
    repeat until *the queue is empty*
        take a cell c from the queue
        if *c is the robot cell* then return(success)
        foreach *n which is a neighbor cell of c*
            if *n is not an obstacle* and *n has label* -2
                label n by grid(c) + 1
(!)   *insert n in the queue ordered by $h(n)$*
    if *the queue is empty* then return(failure)
end

Note: $\boxed{\text{Elements in the queue are ordered according to } h.}$

Example: $h(n) =$ Manhattan distance to goal





Heuristic search – Implementation

Main missing ingredient: ordered queue

```
structure Q_Element
  int i, j, key;   // queue is ordered by increasing keys
  Q_Element *next;
end

procedure push_queue_ordered (int k, Cell [i,j], Queue q)
  el := new_el_queue(q);
  if (el = null) return(-1);
  el.k := k;
  el.i := i;
  el.j := j;
  if (q.head = nil)
     el.next := nil;
     q.head := &el;
     q.tail := &el;
     return();
  else
  if (k < q.head->k)
     el.next := q.head;
     q.head := &el;
     return();
  else
  for (p := q.head; p = q.tail; p := p.next)
    if (k < p.next->k)
       el.next := p.next;
       p.next  := &el;
       return();
  el.next := nil;
  p.next  := &el;
  q.tail  := &el;
end
```

We need to replace the call `push_queue([i,j], queue)`
by `push_queue_ordered([i,j], h([i,j]), queue)`,
where `h([i,j]) = |i-goal_i| + |j-goal_j|`

Example

A* search

- Limitation of greedy search
  - *$h(c)$ does not consider effort already made to get to c*
  - *$h(c)$ of current node and past nodes are not compared*

- Idea: use heuristic function $g(h) + h(c)$:
  - *$h(c)$ = distance from c to goal (still to go)*
  - *$g(c)$ = distance from start to c (already done)*
  - *Note: h is heuristic, g is known*
  - *Note: grounded on Bellman's principle*

- Search is a mixture of depth- and breadth-first
  - *priority to cells that "should" lay on a shorter path*

- Pros
  - *always find a solution (if exists)*
  - *always find the optimal solution (if h admissible)*
  - *can be very fast*

- Cons
  - *efficiency depends on the choice of h*

- How would you implement it?

Summary

- If you want a simple path planner
  - *use Breadth-First Search*
  - *it always finds the best solution*

- If you want an efficient path planner
  - *use Heuristic Search*
  - *and work out your h function very carefully*

- If you want a really efficient path planner
  - *use more sophisticated forms of heuristic search*
  - *e.g.: A**
  - *even better: IDA*, D* (search the web!)*

- If you have uncertainty in your domain
  - *use other A.I. techniques*
  - *e.g.: Markov decision processes, POMDP*