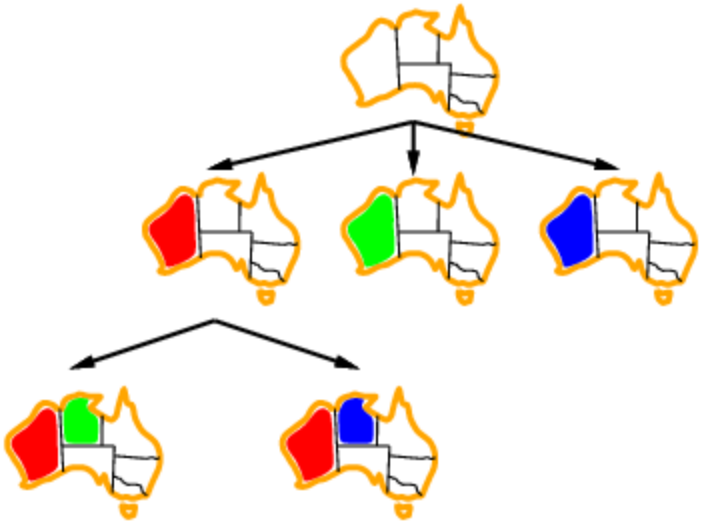
Backtracking search

- In CSP's, variable assignments are **commutative**
  - For example, *[WA = red then NT = green]* is the same as *[NT = green then WA = red]*
- We only need to consider assignments to a single variable at each level (i.e., we fix the order of assignments)
  - Then there are only m^n leaves
- Depth-first search for CSPs with single-variable assignments is called **backtracking search**

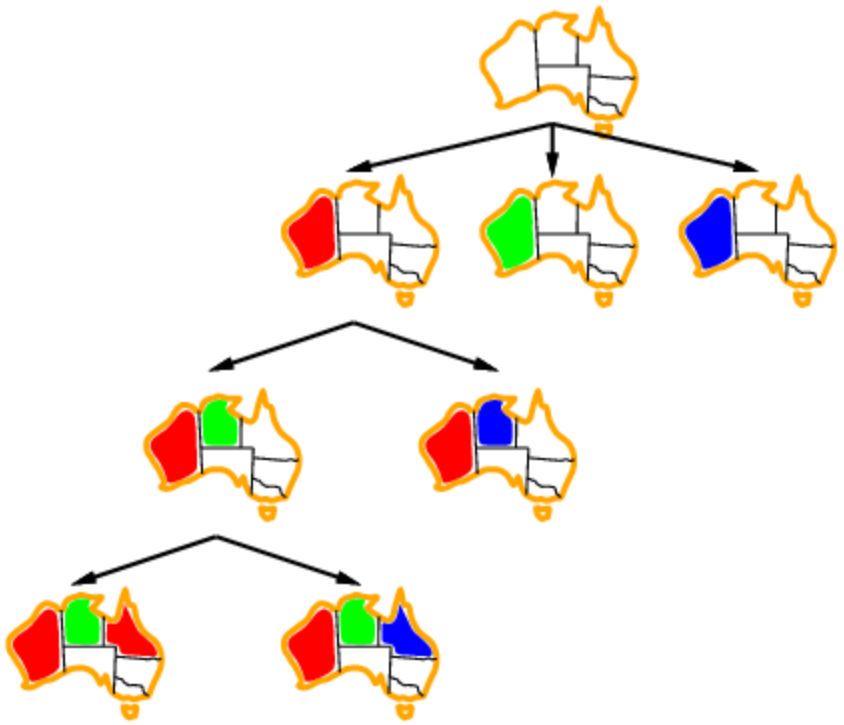
Example

Example

Example



Example



Backtracking search algorithm

```
function RECURSIVE-BACKTRACKING(assignment, csp)
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
      if value is consistent with assignment given CONSTRAINTS[csp]
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

- Making backtracking search efficient:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the *fewest* legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

Given a variable, in which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

Given a variable, in which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

Which assignment for Q should we choose?

Early detection of failure

```
function RECURSIVE-BACKTRACKING(assignment, csp)
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
        if value is consistent with assignment given CONSTRAINTS[csp]
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure

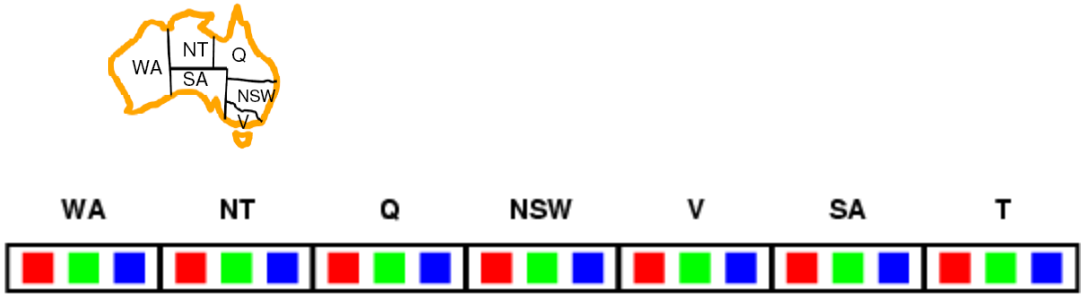
Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

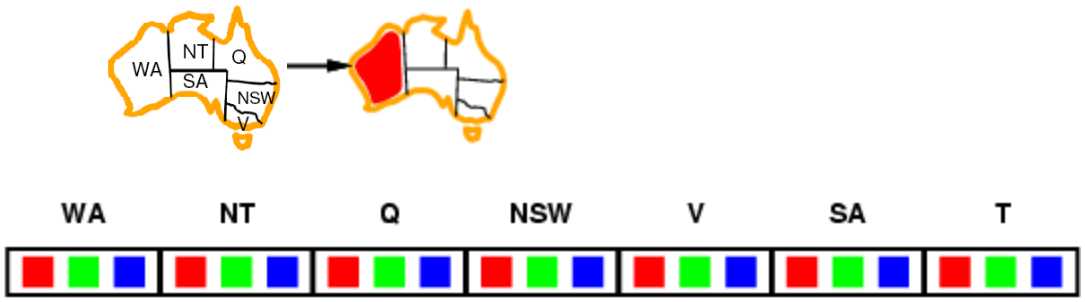
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



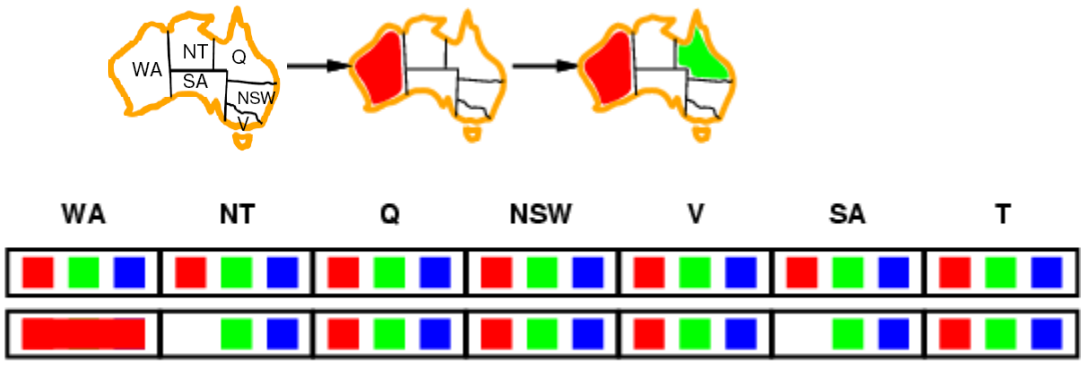
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



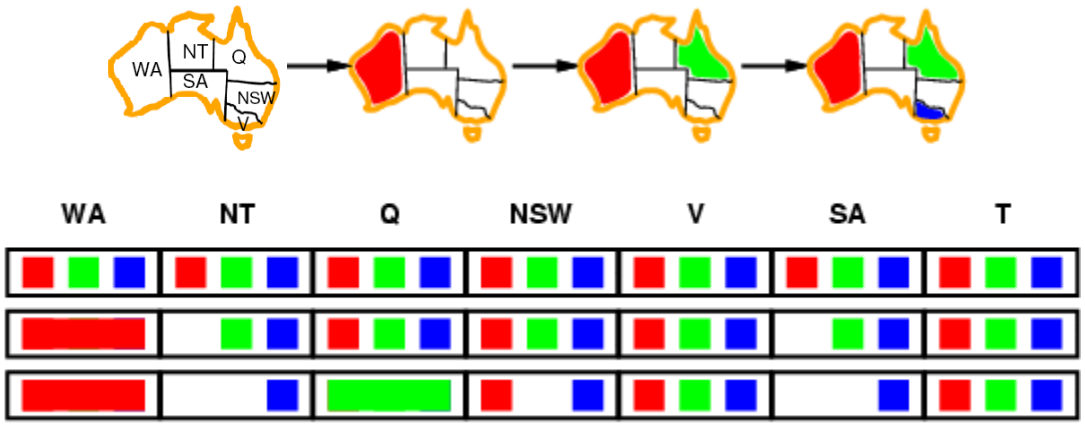
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



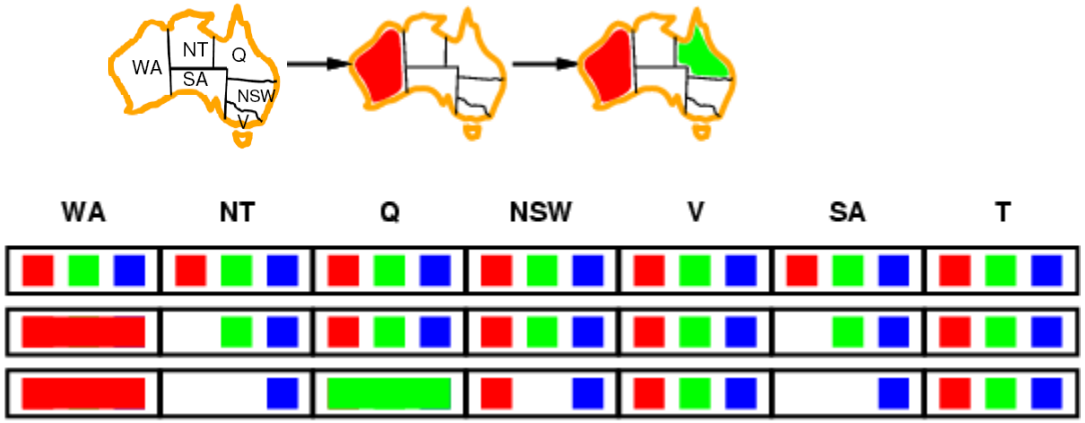
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



Constraint propagation

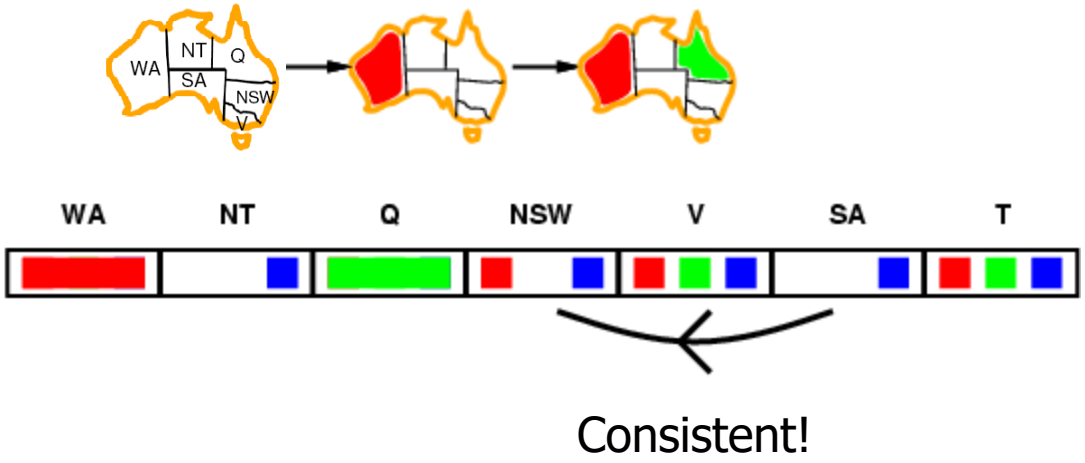
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures



- NT and SA cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints *locally*

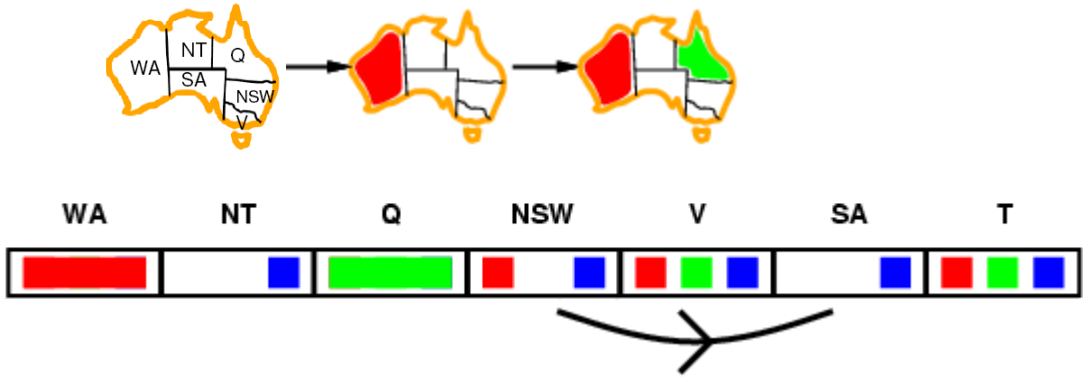
Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*



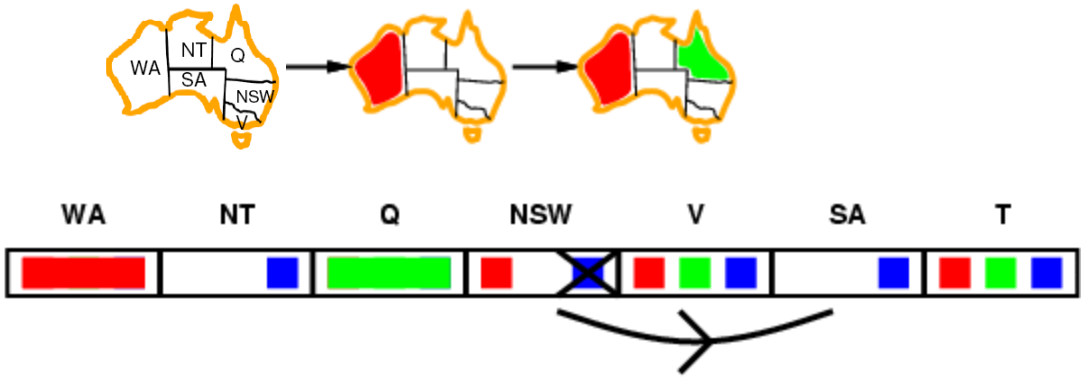
Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*



Arc consistency

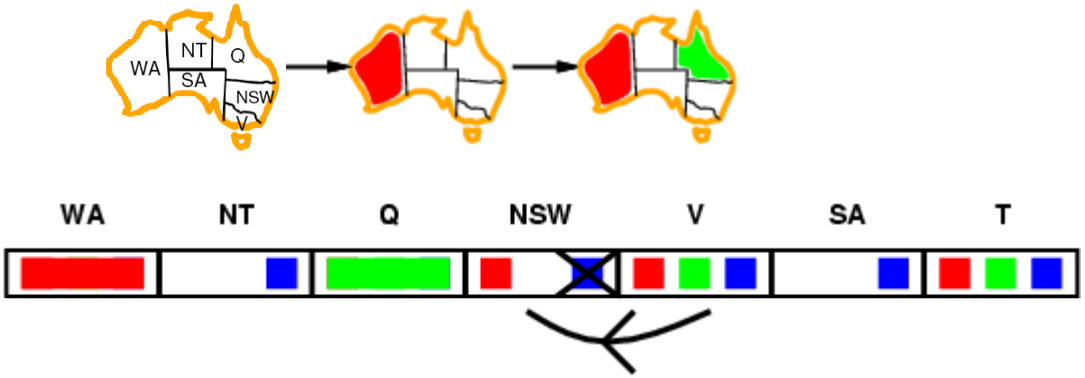
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

Arc consistency

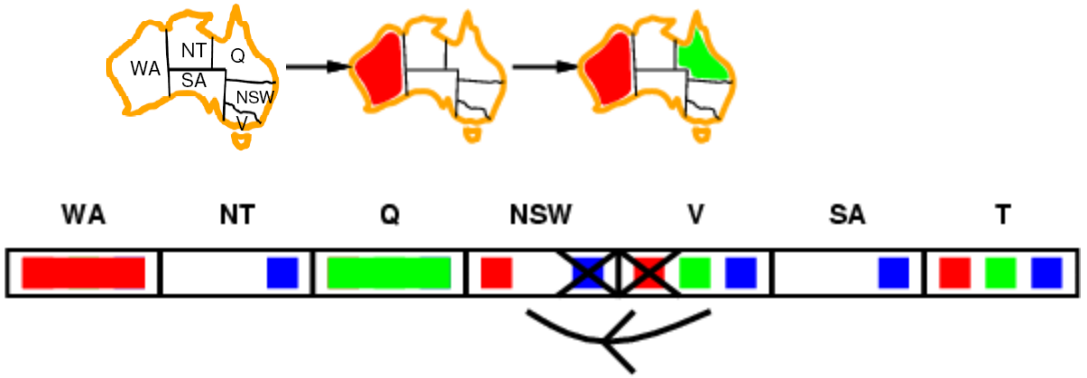
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

Arc consistency

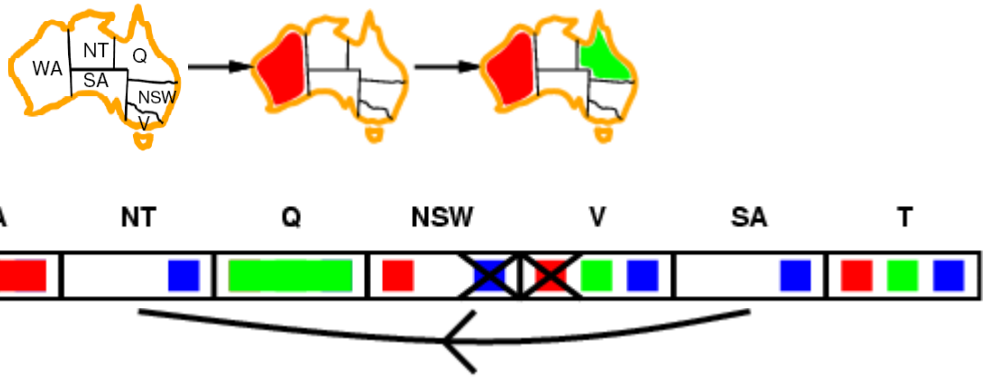
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

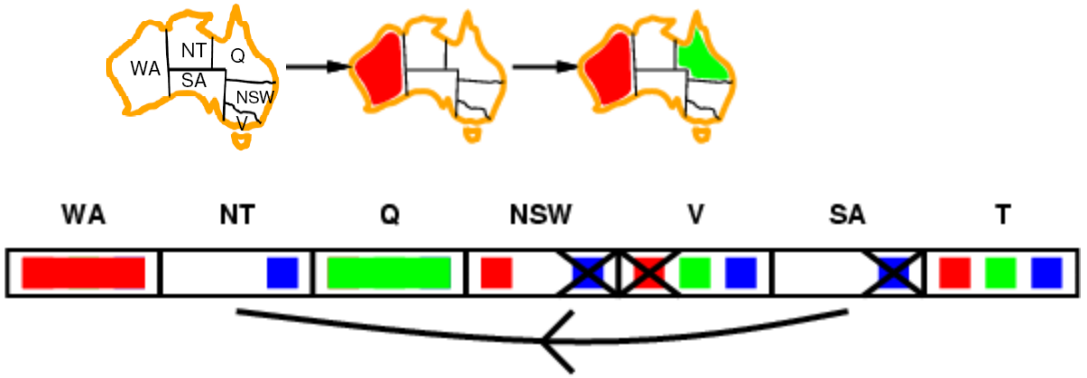
Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment

Arc consistency algorithm AC-3

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains

inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$

local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

while *queue* is not empty

$(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)

if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**

for each X_k **in** NEIGHBORS[X_i] **do**

add (X_k, X_i) to *queue*

---

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds

removed $\leftarrow$ *false*

for each x **in** DOMAIN[X_i]

if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$

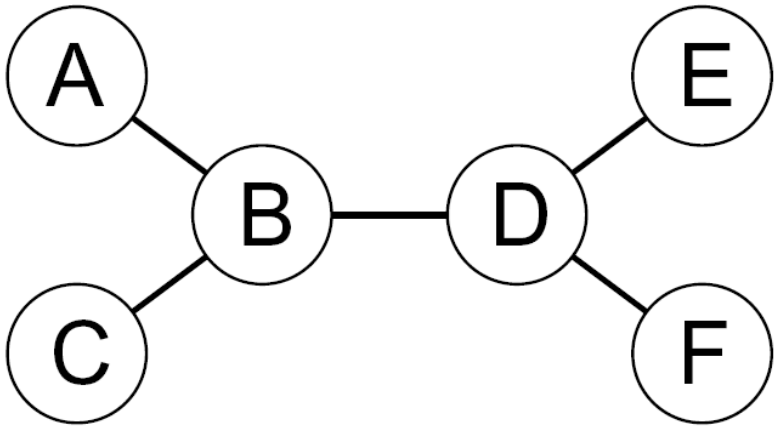
then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*

return *removed*

Tree-structured CSPs

- Certain kinds of CSPs can be solved without resorting to backtracking search!

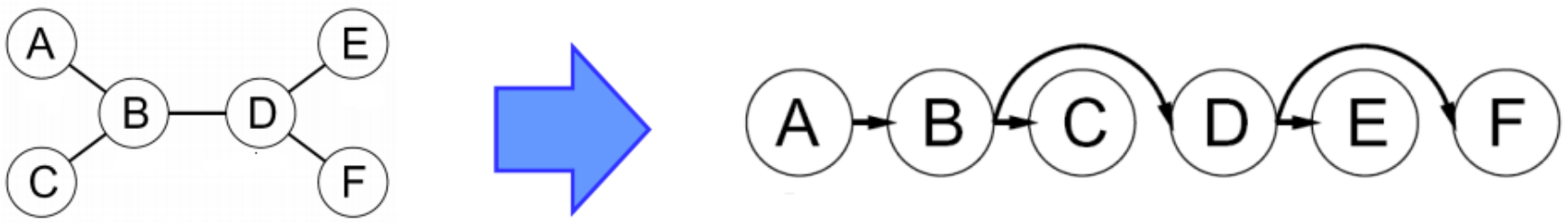
- *Tree-structured CSP*: constraint graph does not have any loops



Source: P. Abbeel, D. Klein, L. Zettlemoyer

Algorithm for tree-structured CSPs

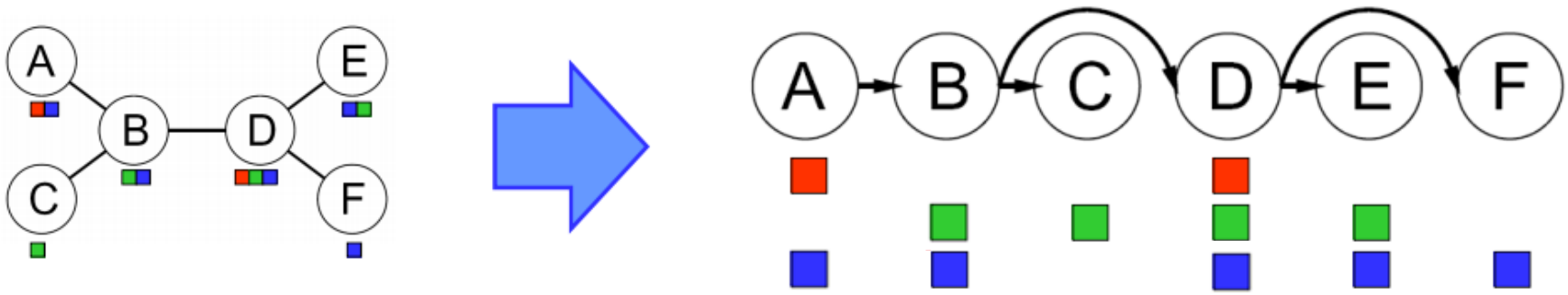
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

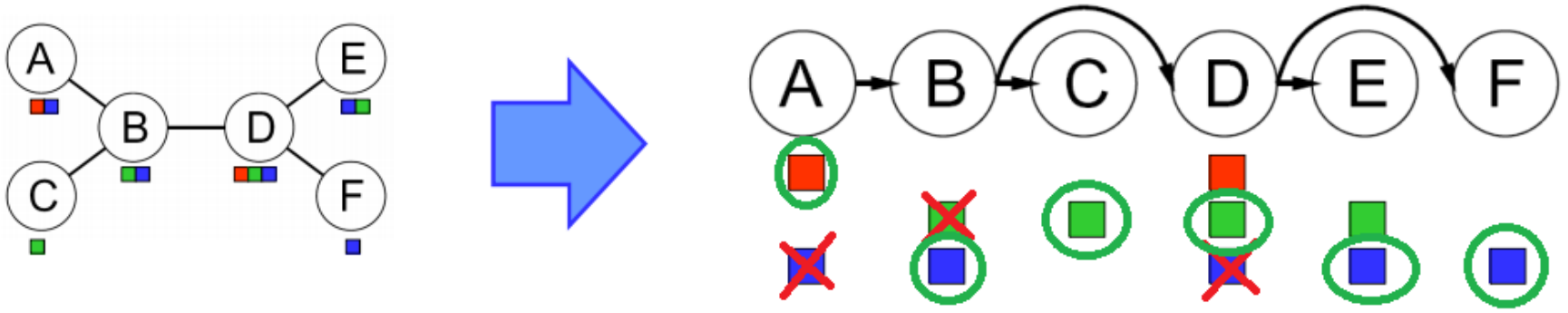
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- **Backward removal phase**: check arc consistency starting from the rightmost node and going backwards



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- **Backward removal phase**: check arc consistency starting from the rightmost node and going backwards
- **Forward assignment phase**: select an element from the domain of each variable going left to right. We are guaranteed that there will be a valid assignment because each arc is consistent

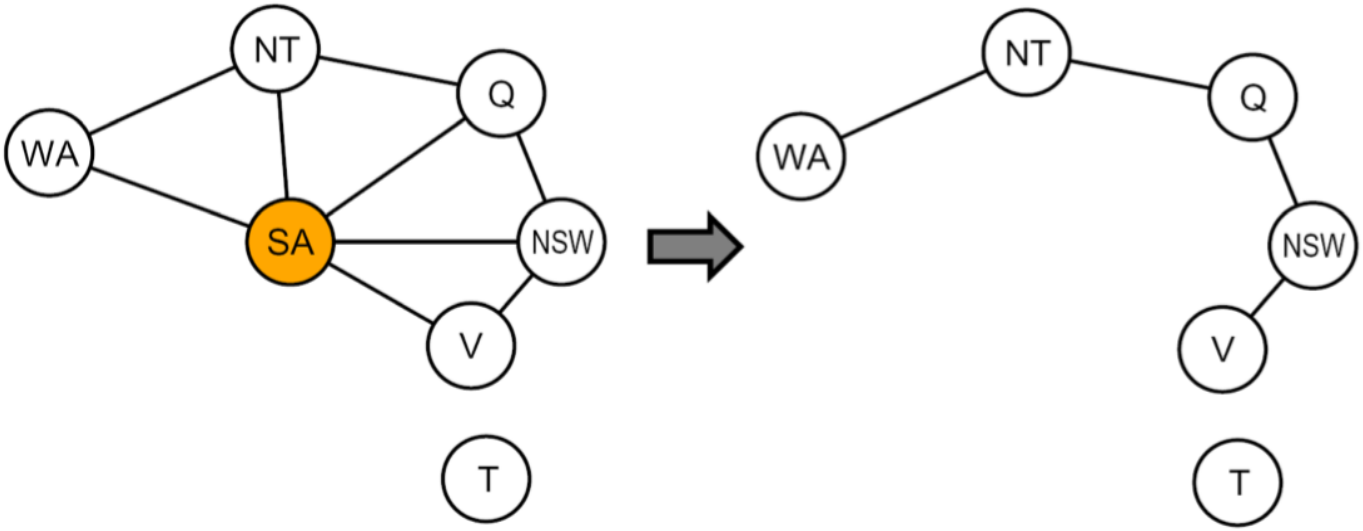


http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

- If n is the number of variables and m is the domain size, what is the running time of this algorithm?
  - $O(nm^2)$: we have to check *arc consistency* once for every node in the graph (every node has one parent), which involves looking at pairs of domain values
- What about backtracking search for general CSPs?
  - Worst case $O(m^n)$

Nearly tree-structured CSPs

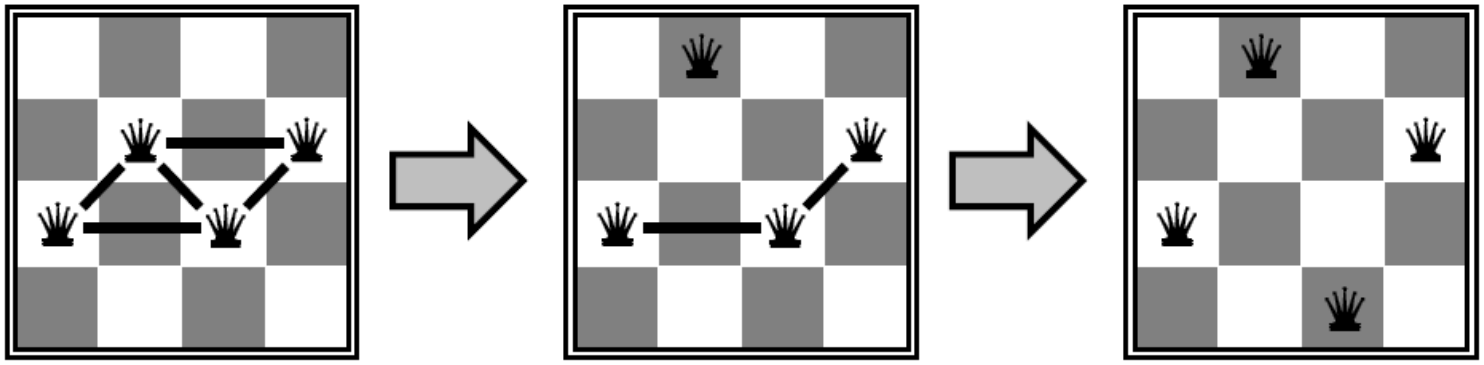


- **Cutset conditioning:** find a subset of variables whose removal makes the graph a tree, instantiate that set in all possible ways, prune the domains of the remaining variables and try to solve the resulting tree-structured CSP
- Cutset size c gives runtime $O(m^c (n - c)m^2)$

Source: P. Abbeel, D. Klein, L. Zettlemoyer

Local search for CSPs

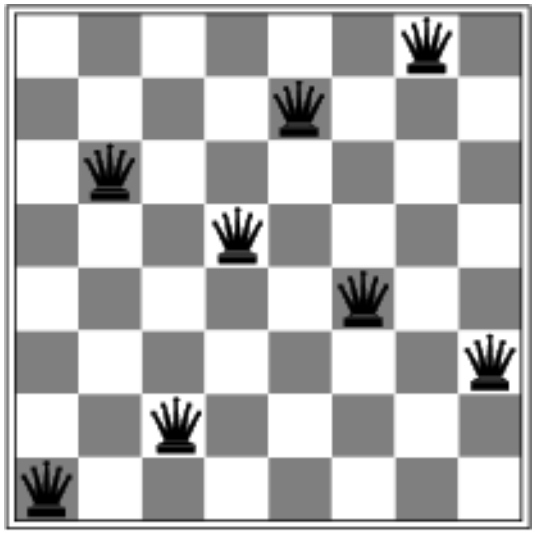
- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints



h = number of conflicts

Local search for CSPs

- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints
  - Problem: *local minima*



$h = 1$

Local search for CSPs

- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints
  - Problem: *local minima*
- For more on local search, see ch. 4

隨機選擇任何衝突變數，
並選擇違反最小約束的值

Computational complexity of CSPs

- Tree-structured CSPs can be solved in $O(nm^2)$ time (n is the number of variables and m is the domain size)
- Backtracking search for general CSPs has worst-case complexity $O(m^n)$
  - Equivalent to brute-force search
  - Can we do better?

Review: CSPs

- CSPs are a special kind of search problem:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values
- **Backtracking search** = DFS where successor states are generated by considering assignments to a single variable
  - **Variable ordering** and **value selection** heuristics can help significantly
  - **Forward checking** prevents assignments that guarantee later failure
  - **Constraint propagation** (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- Alternatives to backtracking search
  - Local search

Readings

- 5.1
- 5.2
- 5.3
- 5.4
- 5.5
- Summary

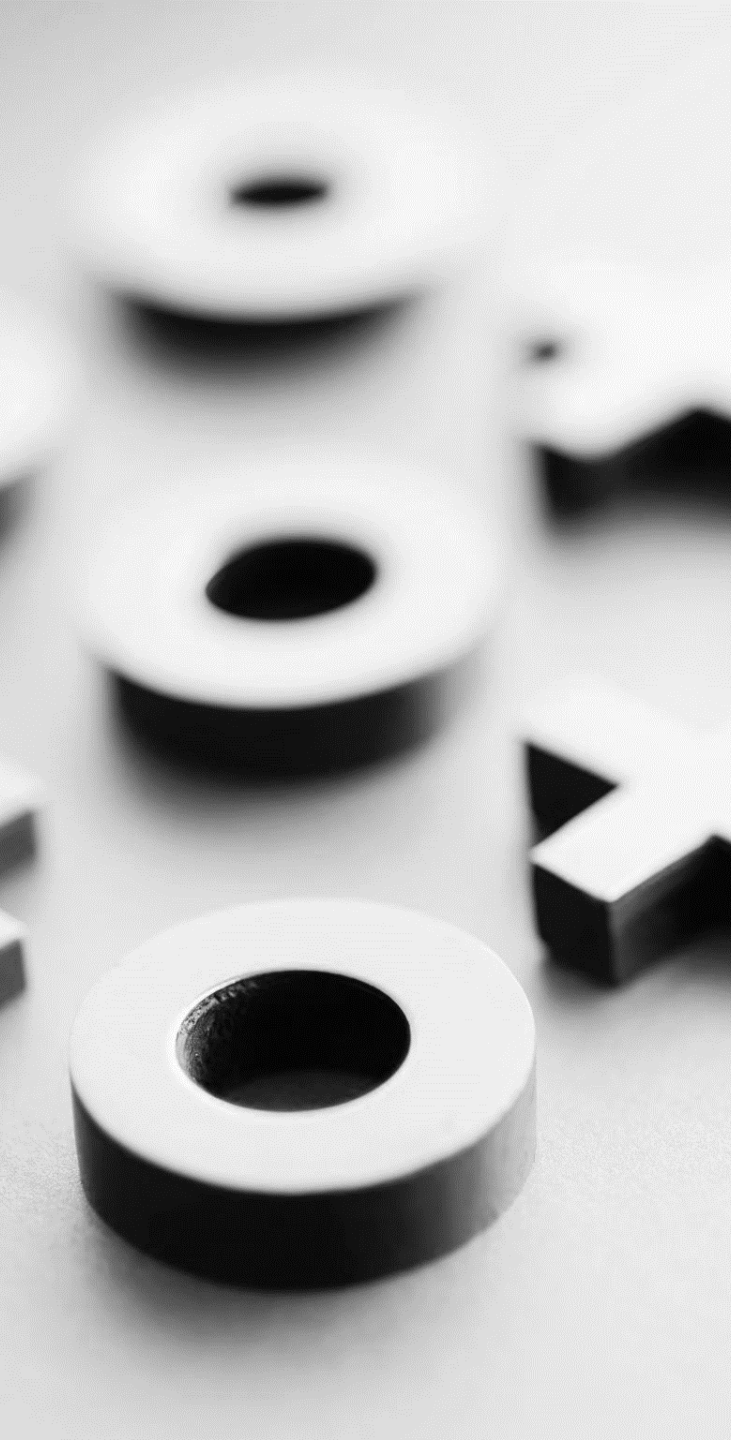
Artificial Intelligence

Adversarial Search and Games

AIMA Chapter 6

Slides by Michael Hahsler
with figures from the AIMA textbook

"Reflected Chess pieces" by Adrian Askew



Games

- Games typically confront the agent with a competitive (adversarial) environment affected by an opponent (strategic environment).

- We will focus on planning for
  - two-player zero-sum games with
  - deterministic game mechanics and
  - perfect information (i.e., fully observable environment).

- We call the two players:
  1) **Max** tries to maximize his utility.
  2) **Min** tries to minimize Max's utility since it is a zero-sum game.

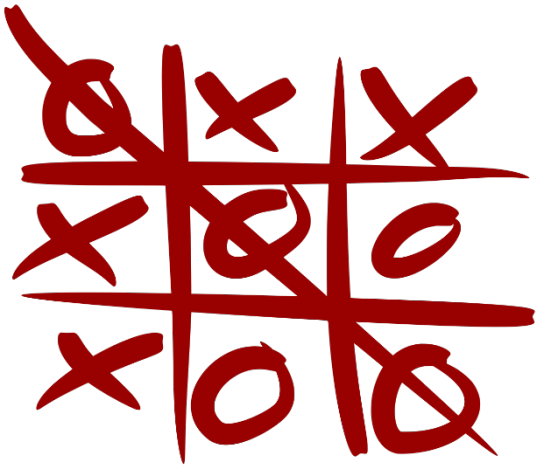
Definition of a Game

- **Definition:**

s_0	The initial state (position, board).
$Actions(s)$	Legal moves in state s.
$Result(s, a)$	Transition model.
$Terminal(s)$	Test for terminal states.
$Utility(s)$	Utility for player Max.

- **State space**: a graph defined by the initial state and the transition function containing all reachable states (e.g., chess positions).
- **Game tree**: a search tree superimposed on the state space. A complete game tree follows every sequence from the current state to the terminal state (the game ends).

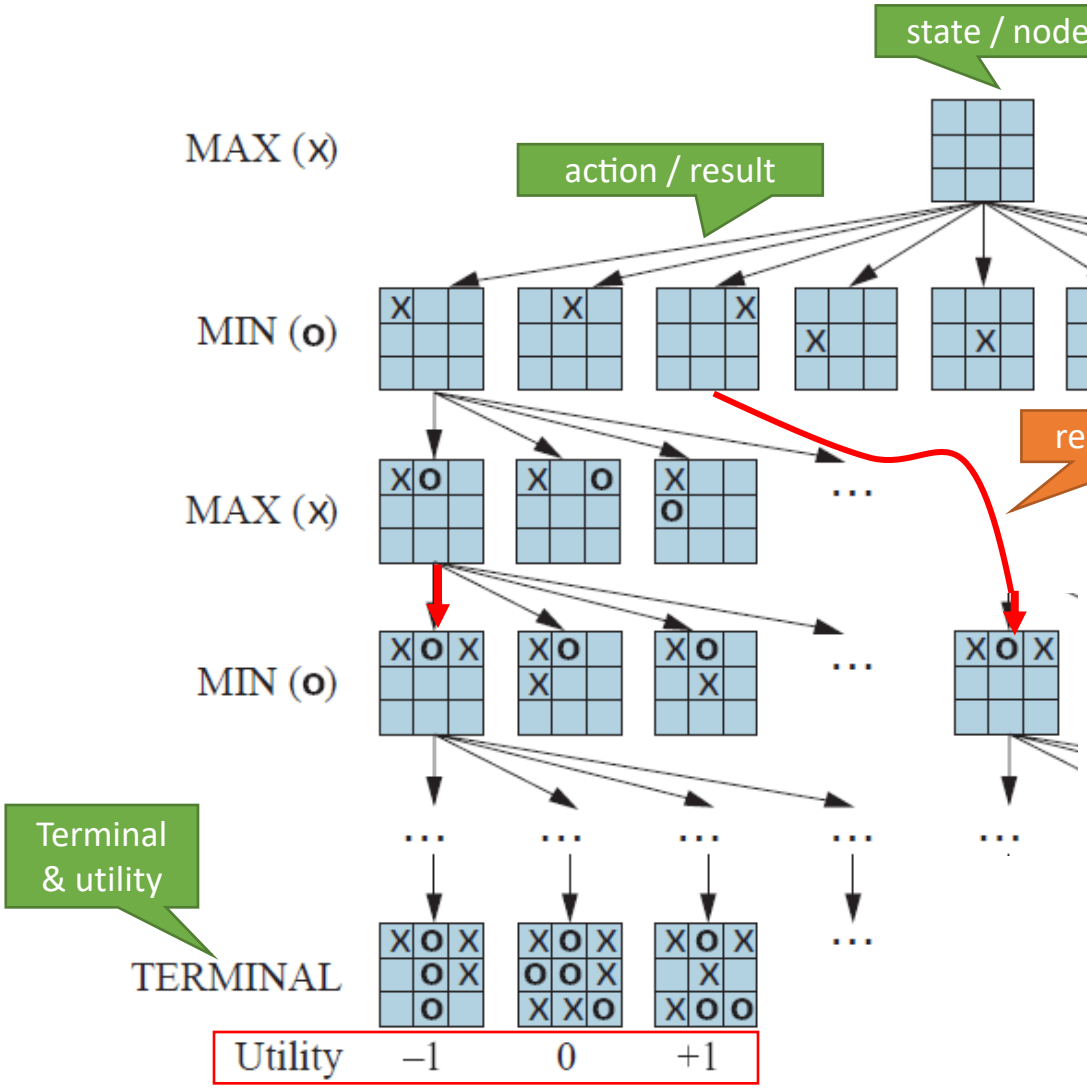
Example: Tic-tac-toe



s_0	Empty board.
$Actions(s)$	Empty squares.
$Result(s, a)$	Place symbol (x/o) on empty square.
$Terminal(s)$	Did a player win or is the game a draw?
$Utility(s)$	+1 if x wins, -1 if o wins and 0 for a draw.
	Utility is only defined for terminal states.

Here player x is Max and player o is Min.

Tic-tac-toe: Partial Game Tree



# of nodes

1

9

9×8

The **state space** size (number of possible boards) is much smaller than:

$$3^9 = 19{,}683 \text{ states.}$$

However, the complete **game tree** is much larger because the same state (board) can be reached in different subtrees (**redundant paths**). The game tree here is a little smaller than:

$$9 + (9\times 8) + (9\times 8\times 7) + \cdots (9!)$$
$$= 986{,}409 \text{ nodes}$$

Methods for Adversarial Games

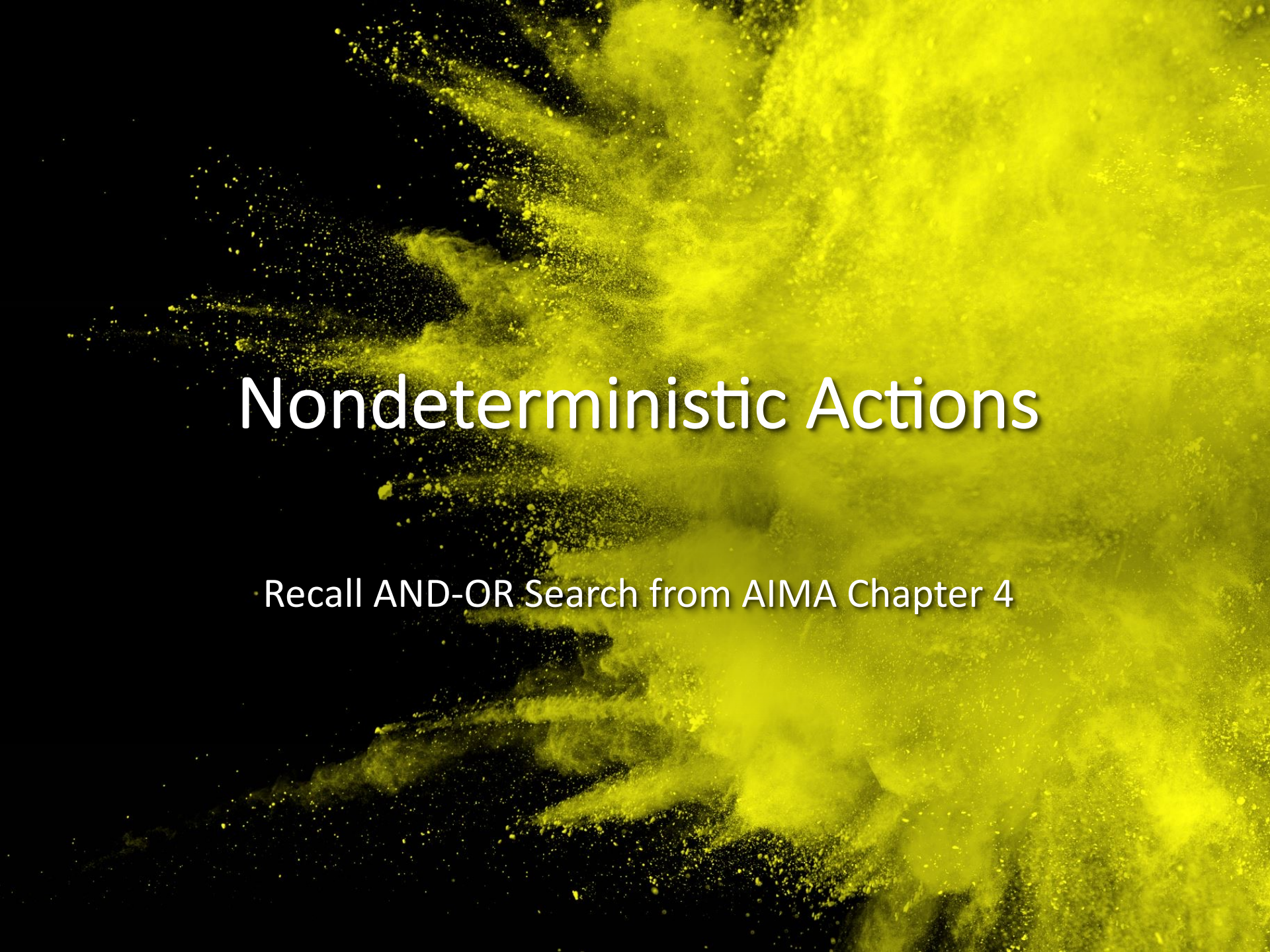
Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: *Minimax search* and *Alpha-Beta pruning* where **each player plays optimal** to the end of the game.

Heuristic Methods

(game tree is too large)

- **Heuristic Alpha-Beta Tree Search**:
    a. Cut off game tree and use heuristic for utility.
    b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.


Nondeterministic Actions
Recall AND-OR Search from AIMA Chapter 4

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods
(game tree is too large)

- **Heuristic Alpha-Beta Tree Search**:
    a. Cut off game tree and use heuristic for utility.
    b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

Recall: Nondeterministic Actions

For **planning**, we do not know what the opponents moves will be. This is the same situation as not being to sense the opponents moves during a real game which we have already modeled using nondeterministic actions.

Each action consists of the move by the player and all possible (i.e., nondeterministic) responses by the opponent.

Outcome of actions in the environment is nondeterministic = **transition model need to describe uncertainty about the opponent's behavior.**

Example transition:

$$Results(s_1, a) = \{s_2, s_4, s_5\}$$

i.e., action a in s_1 can lead to one of several states (which is called a belief state of the agent).

Recall: AND-OR DFS Search Algorithm

= nested If-then-else statements

```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
  return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
  if problem.IS-GOAL(state) then return the empty plan
  if IS-CYCLE(path) then return failure          // don't follow loops
  for each action in problem.ACTIONS(state) do   // check all possible actions
      plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
      if plan ≠ failure then return [action] + plan]
  return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
  for each s_i in states do
                                                 // check all possible current states
      plan_i ← OR-SEARCH(problem, s_i, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

my moves

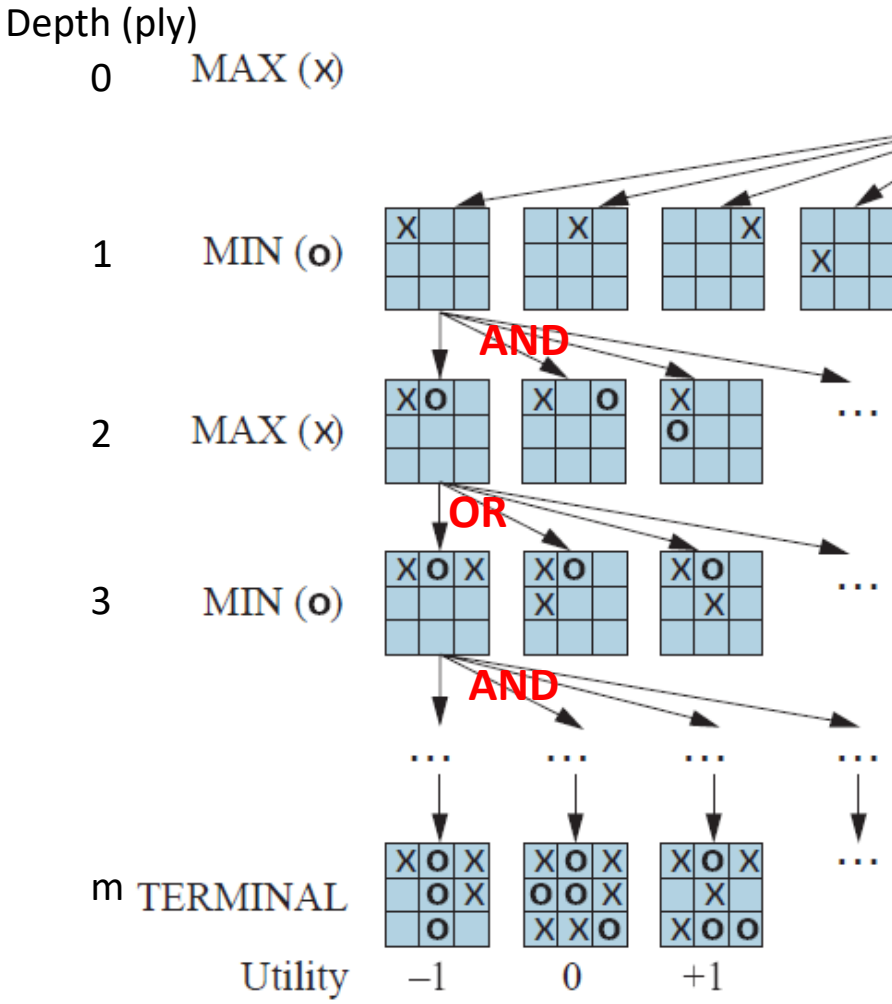
all states that can result from opponent's moves

abandon subtree if a loss is found

Go through opponent moves

Tic-tac-toe: AND-OR Search

We play MAX and decide on our actions (OR).
MIN's actions introduce non-determinism (AND).



Pick an action that leads to a subtree with only win leaves.

Objective: Find a subtree that has only win leaf nodes (utility +1). We can abandon a subtree if we find a single loss (utility -1).

We call playing always the best move **playing optimally**. Since we consider all the opponent's moves in the AND stage, we also includes MIN's best move. This means we consider MIN playing optimally.

Optimal Decisions

Minimax Search and Alpha-Beta Pruning

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: *Minimax search* and *Alpha-Beta pruning* where **each player plays optimal** to the end of the game.

Heuristic Methods
(game tree is too large)

- **Heuristic Alpha-Beta Tree Search**:
    a. Cut off game tree and use heuristic for utility.
    b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

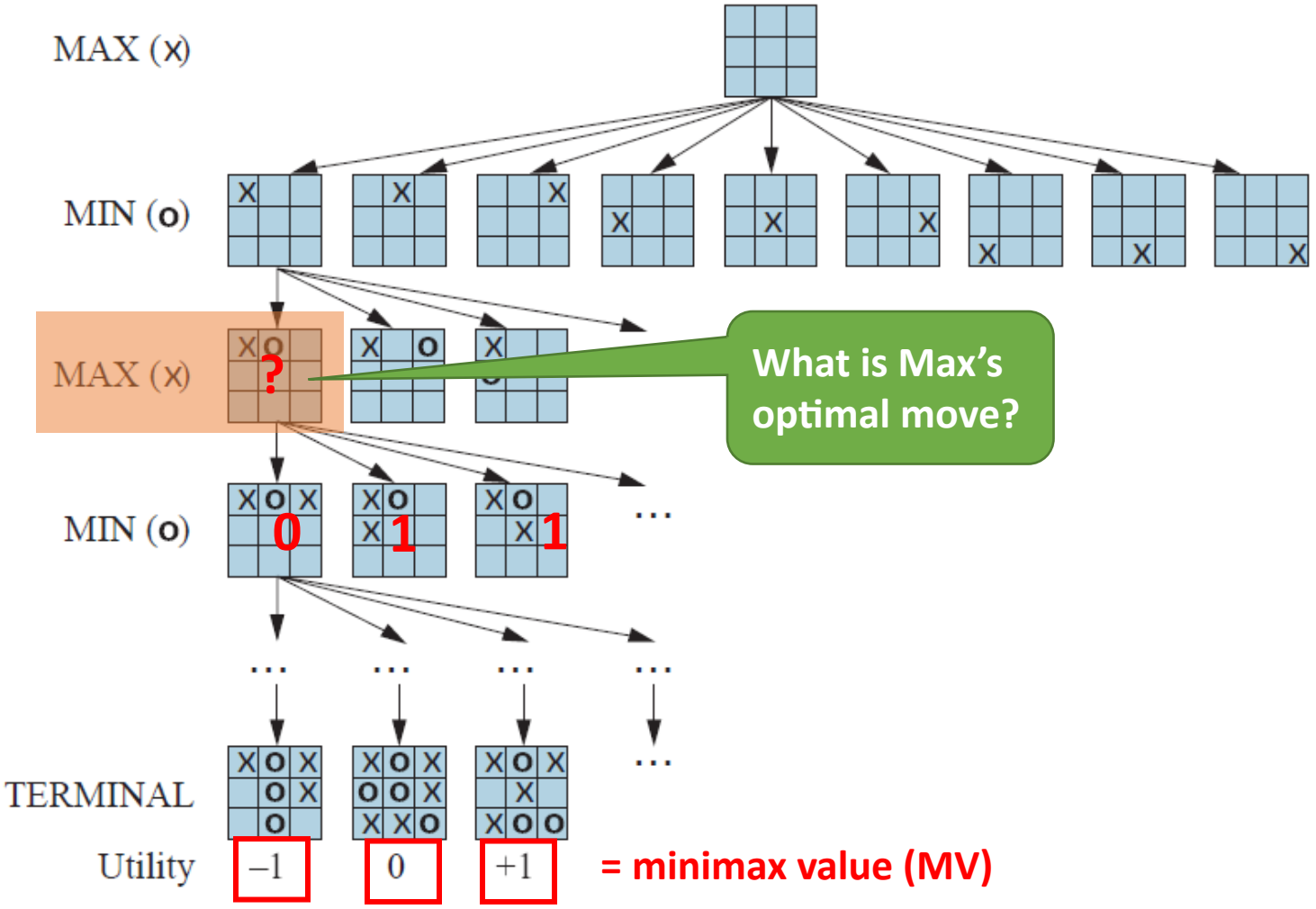
Idea: Minimax Decision

- Assign each state a **minimax value** that reflects how much Max prefers the state (= Min dislikes the state).

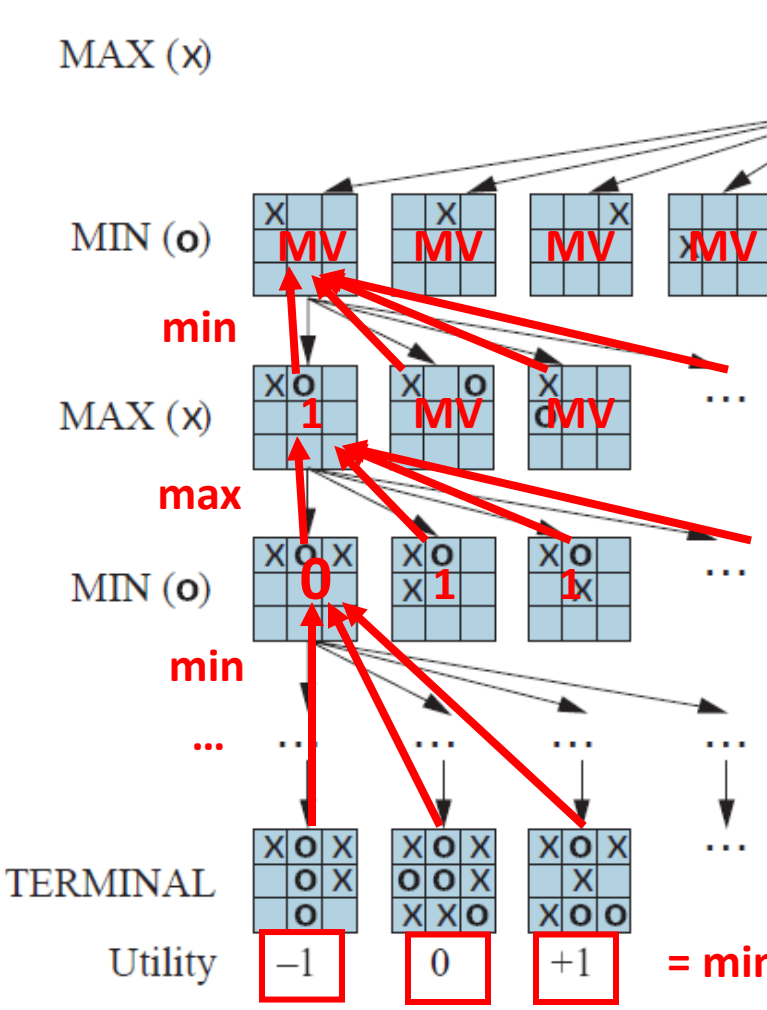
$$Minimax(s) = \begin{cases} Utility(s) & \text{if } terminal(s) \\ \max_{a \in Actions(s)} Minimax\big(Result(s,a)\big) & \text{if } move = Max \\ \min_{a \in Actions(s)} Minimax\big(Result(s,a)\big) & \text{if } move = Min \end{cases}$$

- The minimax value is the utility for Max in state s assuming that **both players play optimally** from s to the end of the game written as a recursion.
- The **optimal decision** for Max is the action that leads to the state with the largest minimax value.

Minimax Search:
Determining MV Values



Minimax Search: Back-up Minimax Values

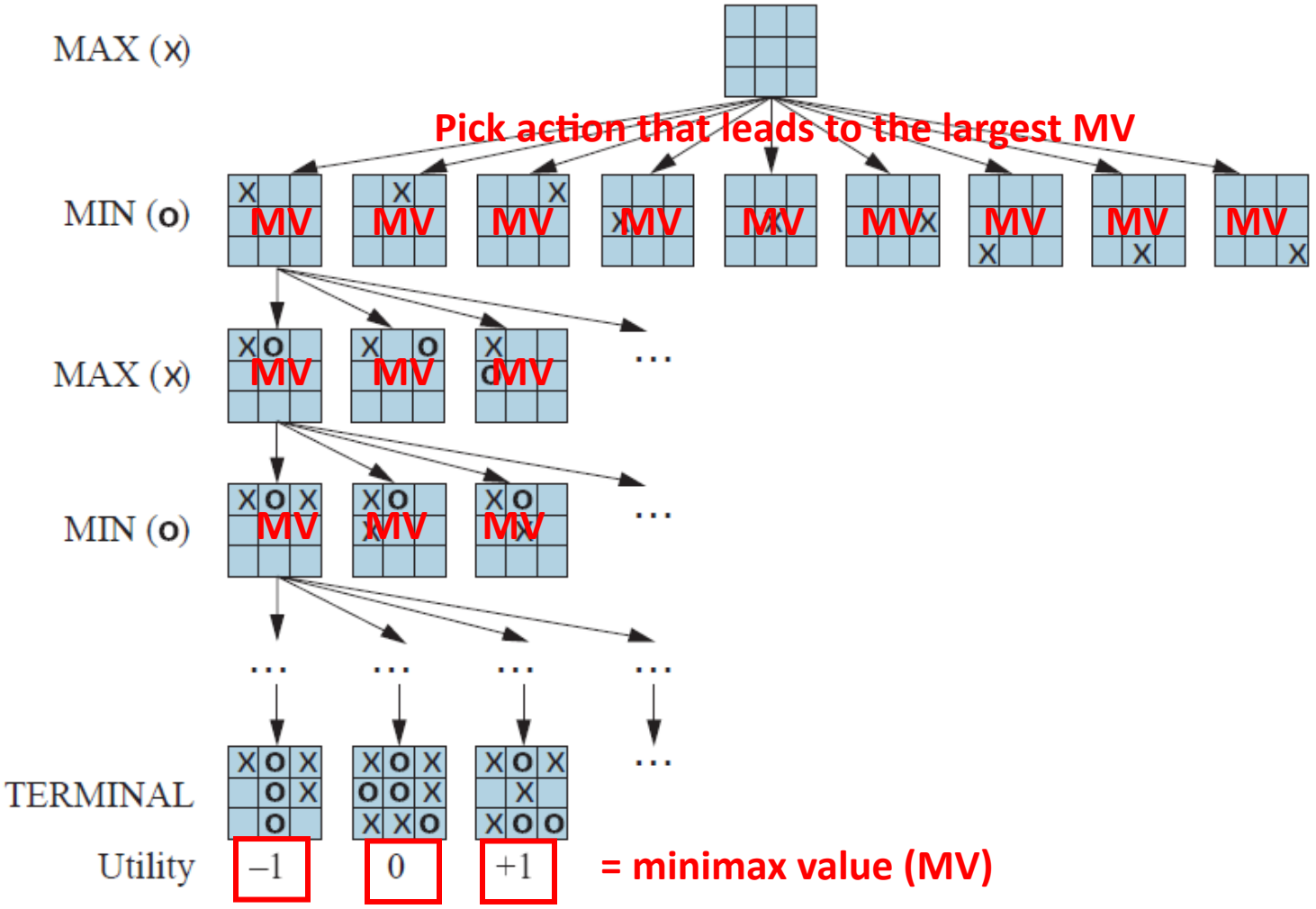


Determine MVs using a bottom-up strategy

- **Max** always picks the action that has the largest value.
- **Min** always picks the action that has the smallest value.

= minimax value (MV)

Minimax Search: Decision



Approach: Follow tree to each *terminal node* and back up minimax value.

Note: This is just a generalization of the AND-OR Tree Search and returns the first action of the conditional plan.

```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

Found a better action?

Found a better action?

Represents OR Search

Represents AND Search

Issue: Game Tree Size

b: max branching factor
m: max depth of tree

- **Minimax search traverses the complete game tree using DFS!**

Space complexity: $O(bm)$

Time complexity: $O(b^m)$

- Fast solution is only feasible for very simple games with small branching factor!

- Example: Tic-tac-toe

$$b = 9, m = 9 \rightarrow O(9^9) = O(387{,}420{,}489)$$

b decreases from 9 to 8, 7, … the actual size is smaller than:

$$1 + (9) + (9\times 8) + (9\times 8\times 7) + \cdots + (9!) = 986{,}409 \text{ nodes}$$

- We need to reduce the search space! → **Game tree pruning**

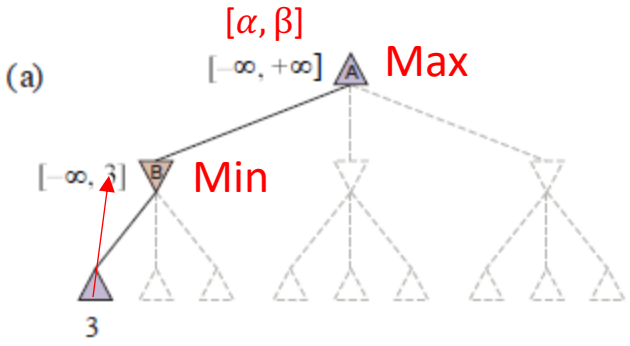
Alpha-Beta Pruning

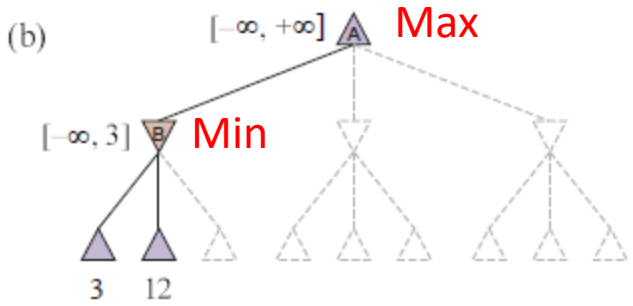
- **Idea**: Do *not* search parts of the tree if they do not make a difference to the outcome.

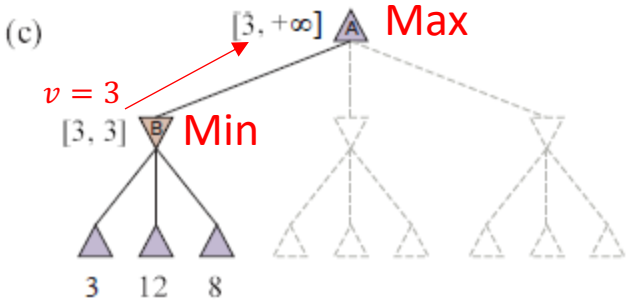
- **Observations**:
  - $\min(3, x, y)$ can never be more than 3
  - $\max(5, \min(3, x, y, \ldots))$ does not depend on the values of x or y.
  - Minimax search applies alternating min and max.

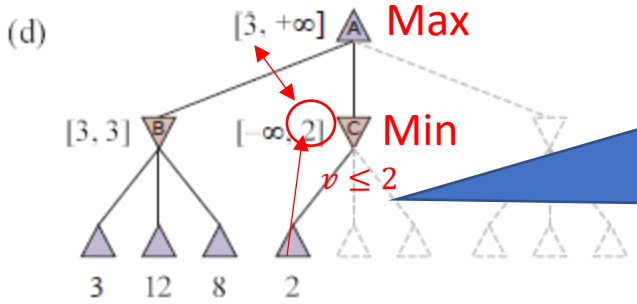
- **Approach**: maintain bounds for the minimax value $[\alpha, \beta]$ and prune subtrees (i.e., don't follow actions) that do not affect the current minimax value bound.
  - Alpha is used by Max and means "$Minimax(s)$ is at least α."
  - Beta is used by Min and means "$Minimax(s)$ is at most β."

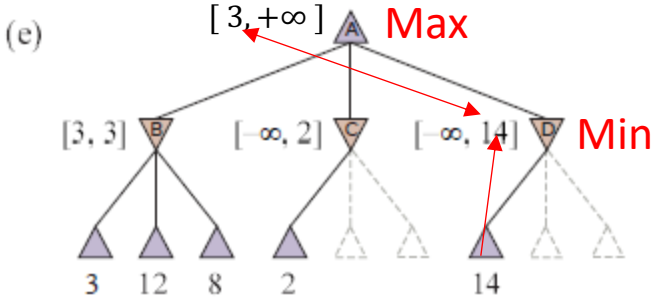
Example: Alpha-Beta Search

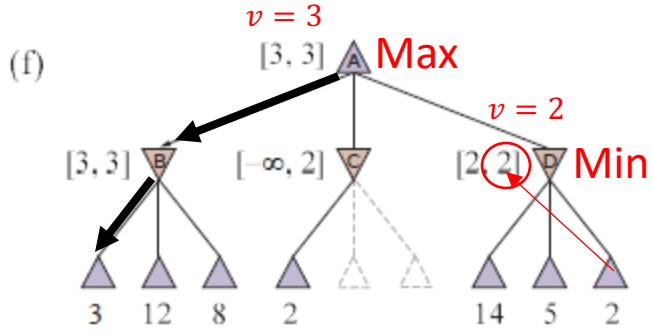












Max updates α (utility is at least)

Min updates β (utility is at most)

Utility cannot be more than 2 in the subtree, but we already can get 3 from the first subtree. Prune the rest.

Once a subtree is fully evaluated, the interval has a length of 0 ($\alpha = \beta$).

= minimax search + pruning

```
function ALPHA-BETA-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞        // v is the minimax value
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
        if v2 > v then                                          Found a better action?
            v, move ← v2, a
            α ← MAX(α, v)
        if v ≥ β then return v, move                            Abandon subtree if Max finds an actions that has more value than the best known move Min has in another subtree.
    return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
        if v2 < v then                                          Found a better action?
            v, move ← v2, a
            β ← MIN(β, v)
        if v ≤ α then return v, move                            Abandon subtree if Min finds an actions that has less value than the best known move Max has in another subtree.
    return v, move
```

Move Ordering for Alpha-Beta Search

- **Idea:** Pruning is more effective if good alpha-beta bounds can be found in the first few checked subtrees.
- **Move ordering for DFS** = Check good moves for Min and Max first.
- We need expert knowledge or some heuristic to determine what a good move is.
- **Issue:** Optimal decision algorithms still scale poorly even when using alpha-beta pruning with *move ordering*.


Heuristic Alpha-Beta
Tree Search

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods
(game tree is too large or search takes too long)

- **Heuristic Alpha-Beta Tree Search**:
    a. Cut off game tree and use heuristic for utility.
    b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

Cutting off search

Reduce the search cost by restricting the search depth:

1. Stop search at a *non-terminal node*.
2. Use a heuristic evaluation function $Eval(s)$ to *approximate* the utility for that node/state.

Needed properties of the evaluation function:

- Fast to compute.
- $Eval(s) \in [Utility(loss), Utility(win)]$
- Correlated with the actual chance of winning (e.g., using features of the state).

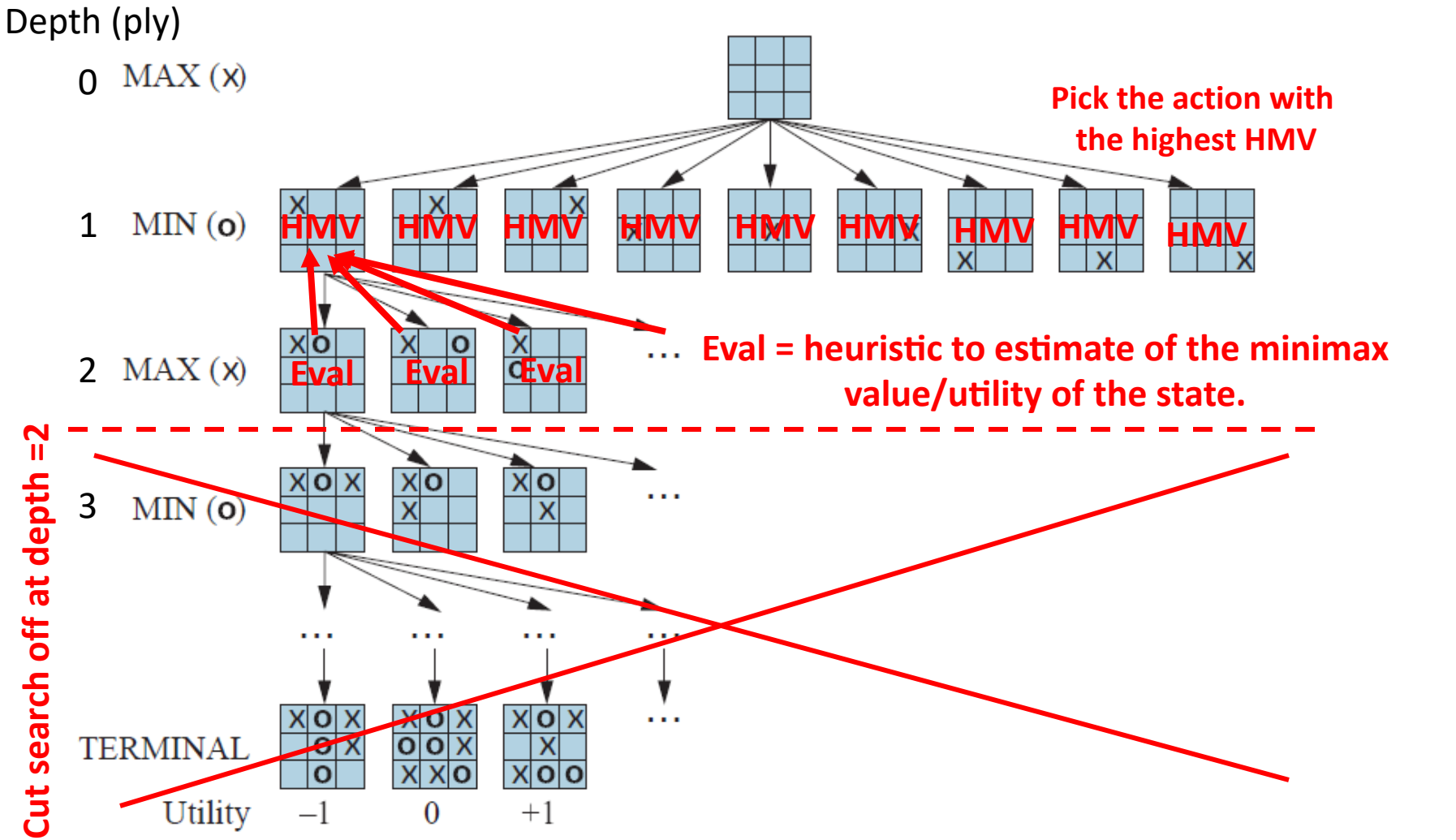
Examples:

1. A weighted linear function

   $$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

   where f_i is a feature of the state (e.g., # of pieces captured in chess).
2. A deep neural network trained on complete games.

Heuristic Alpha-Beta Tree Search: Cutting off search



Forward pruning

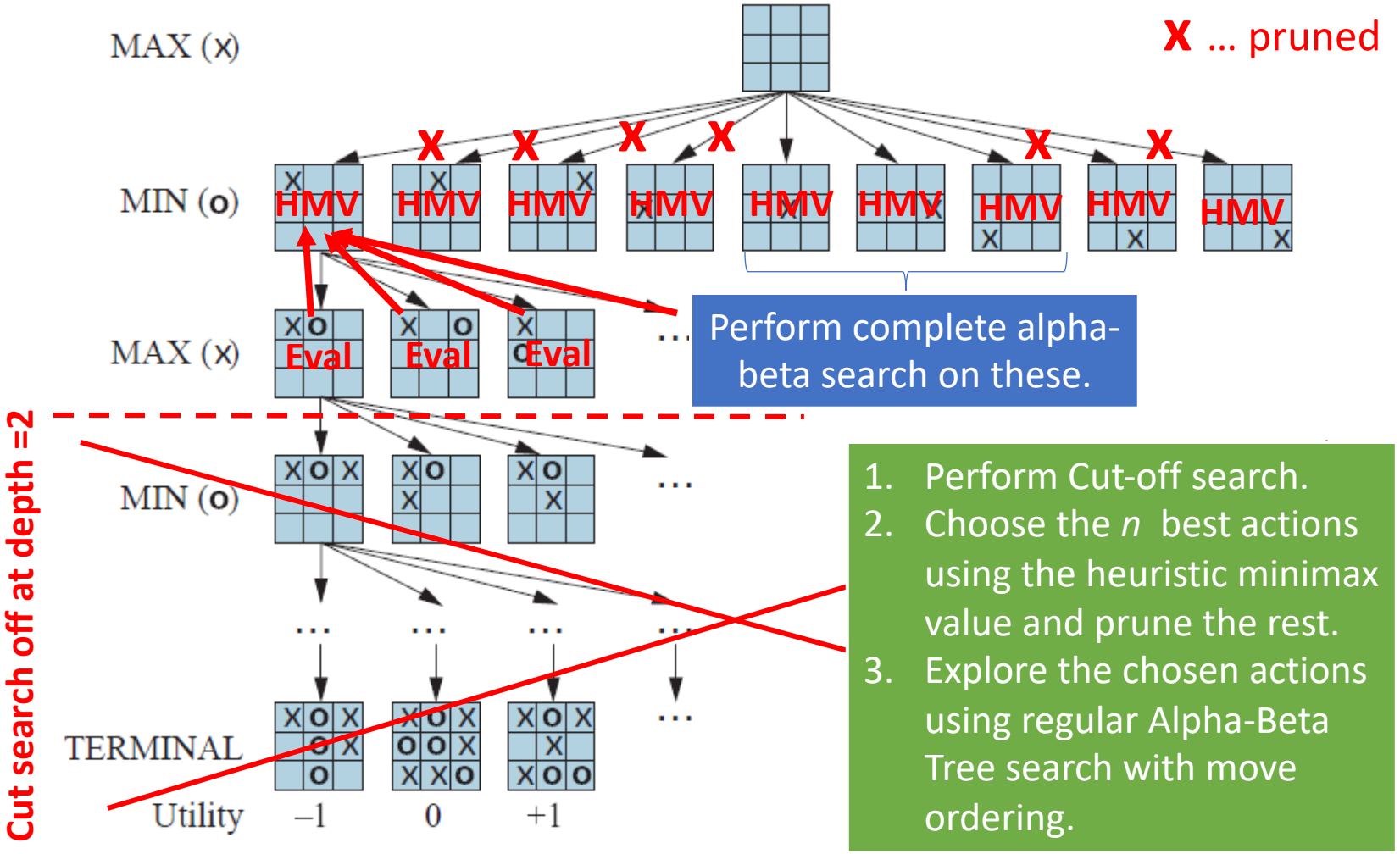
To save time, we can prune moves that appear bad.

There are many ways *move quality* can be evaluated:

- Low heuristic value.
- Low evaluation value *after* shallow search (cut-off search).
- Past experience.

Issue: May prune important moves.

Heuristic Alpha-Beta Tree Search: Example for Forward Pruning



Monte Carlo Tree Search (MCTS)

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods
(game tree is too large or search takes too long)

- **Heuristic Alpha-Beta Tree Search**:
    a. Cut off game tree and use heuristic for utility.
    b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

Idea

- **Approximate $\boldsymbol{Eval(s)}$** as the average utility of several simulation runs to the terminal state (called *playouts*).

- **Playout policy**: How to choose moves during the simulation runs? Example policies:
  - *Random*.
  - Heuristics for good moves developed by experts.
  - Learn good moves from self-play (e.g., with deep neural networks). We will talk about this when we talk about "Learning from Examples."

- Typically used for problems with
  - High branching factor (many possible moves).
  - Unknown or hard to define good evaluation functions.

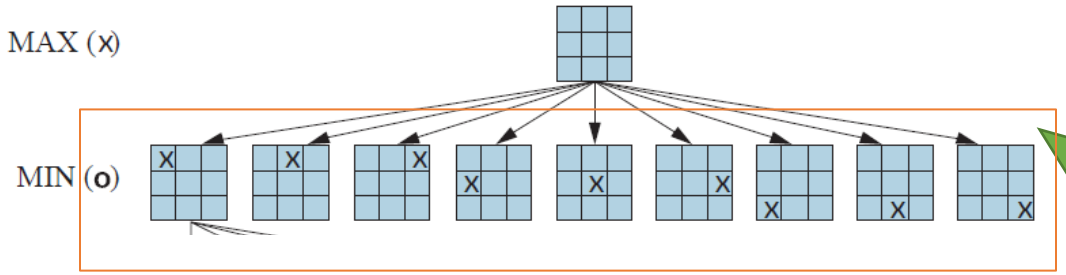
Pure Monte Carlo Search

Find the next best move.

- Method
  1. Simulate N playouts from the **current state**.
  2. Select the move that leads the highest win percentage.

- **Guarantee**: Converges to optimal play for stochastic games as N increases.
- **Do as many playouts as you can** given the available time.

Playout Selection Strategy



Max can start each playout at any of these states. Which one should it choose?

Issue: Pure Monte Carlo Search spends a lot of time to create playouts for bad move.

Better: Select the starting state for playouts to focus on important parts of the game tree.

This presents the following tradeoff between:

Exploration: perform more playouts from states that currently have no or few playouts.

Exploitation: more playouts for states that have done well to get more accurate estimates.

Selection using Upper Confidence Bounds (UCB1)

Tradeoff constant $\approx \sqrt{2}$
can be optimizes using experiments

$$UCB1(n) = \frac{U(n)}{N(n)} + C\sqrt{\frac{\log N(Parent(n))}{N(n)}}$$

Average utility (=**exploitation**)

High for nodes with few playouts relative to the parent node (=**exploration**). Goes to 0 for large $N(n)$

n … node in the game tree
$U(n)$ … total utility of all playouts going through node n
$N(n)$ … number of playouts through n

Selection strategy: Select node with highest UCB1 score.

Monte Carlo Tree Search

We do not need to always start playouts from the current node, we can build a **partial game tree** and simulate from any node in that tree.

Important considerations:

- We can use UCB1 as the **selection strategy** to decide what part of the tree we should focus on for the next playout.
- We can only store a small **part of the game tree**, so we do not store the complete playout runs.

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*

tree ← NODE(*state*)

while IS-TIME-REMAINING() **do**

leaf ← SELECT(*tree*)

child ← EXPAND(*leaf*)

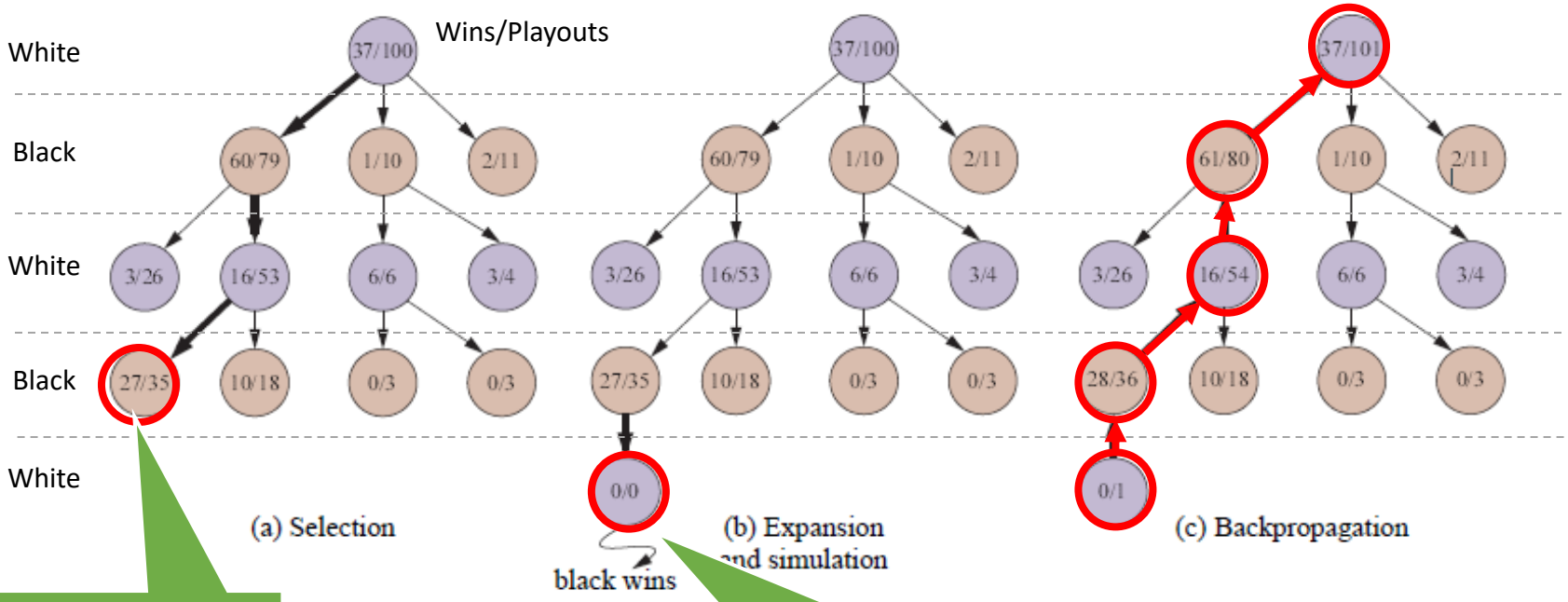
result ← SIMULATE(*child*)

BACK-PROPAGATE(*result*, *child*)

return the move in ACTIONS(*state*) whose node has highest number of playouts

Highest UCB1 score

UCB1 selection favors win percentage more and more.



Select highest UCB1 score node

Note: the simulation path is not recorded to preserve memory!

Conclusion

Nondeterministic actions:

- The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. *All possible moves are considered*.

Optimal decisions:

- Minimax search and Alpha-Beta pruning where *each player plays optimal* to the end of the game.
- Choice nodes and Expectiminimax for stochastic games.

Scale only for tiny problems!

Heuristic Alpha-Beta Tree Search:

- Cut off game tree and use *heuristic evaluation function* for utility (based on state features).
- Forward Pruning: ignore poor moves.
- Learn heuristic from data using MCTS

Monte Carlo Tree search:

- Simulate complete games and calculate proportion of wins.
- Use modified UCB1 scores to expand the partial game tree.
- Learn playout policy using self-play and deep learning.

State of the Art

Readings

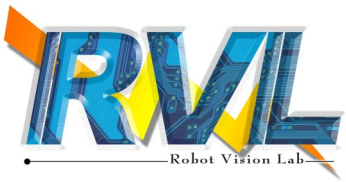
- 6.1
- 6.2
- 6.3
- 6.4
- Summary

Artificial Intelligence

Robotics

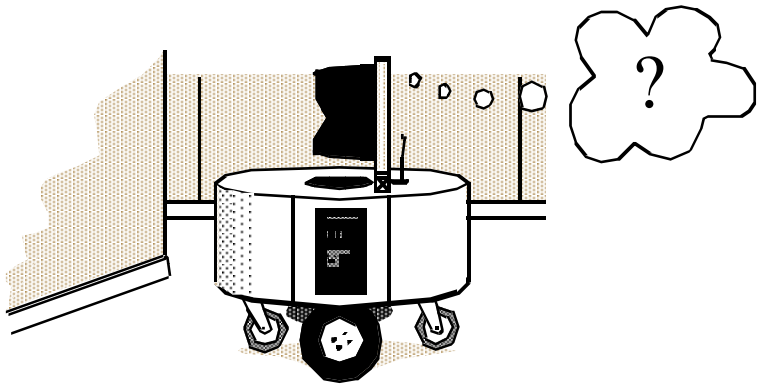
Robot Vision Lab

Huei-Yung Lin



Intelligent Robots

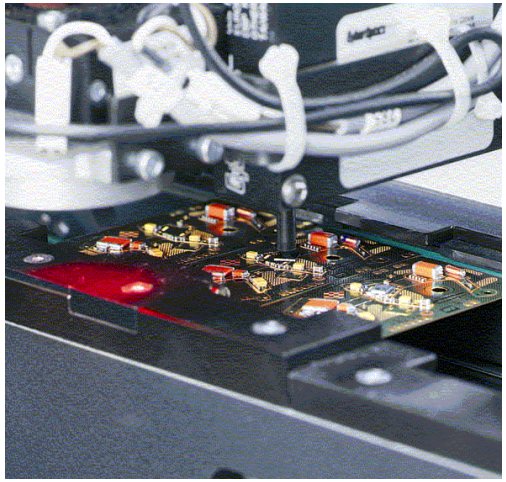
- Three key questions in Mobile Robotics
  - Where am I?
  - Where to go?
  - How to get there?



Intelligent Robots

- To answer these questions the robot has to
  - have a model of the environment (given or autonomously built)
  - perceive and analyze the environment
  - find its position within the environment
  - plan and execute the movement
- We will deal with Locomotion and Navigation
  - Perception
  - Localization
  - Planning and motion generation

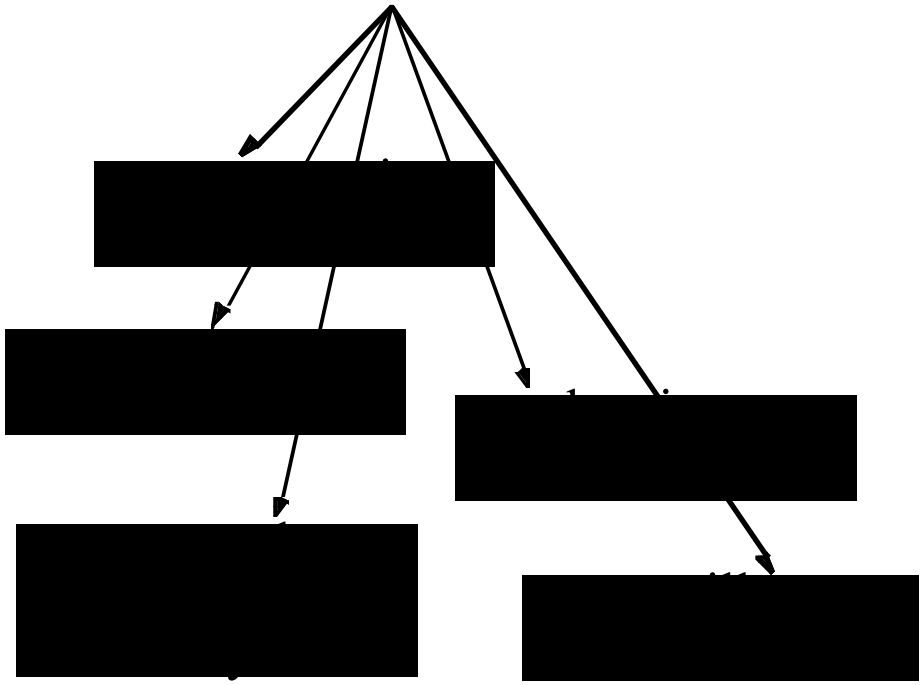
From Manipulators to Mobile Robots



Applications of Mobile Robots

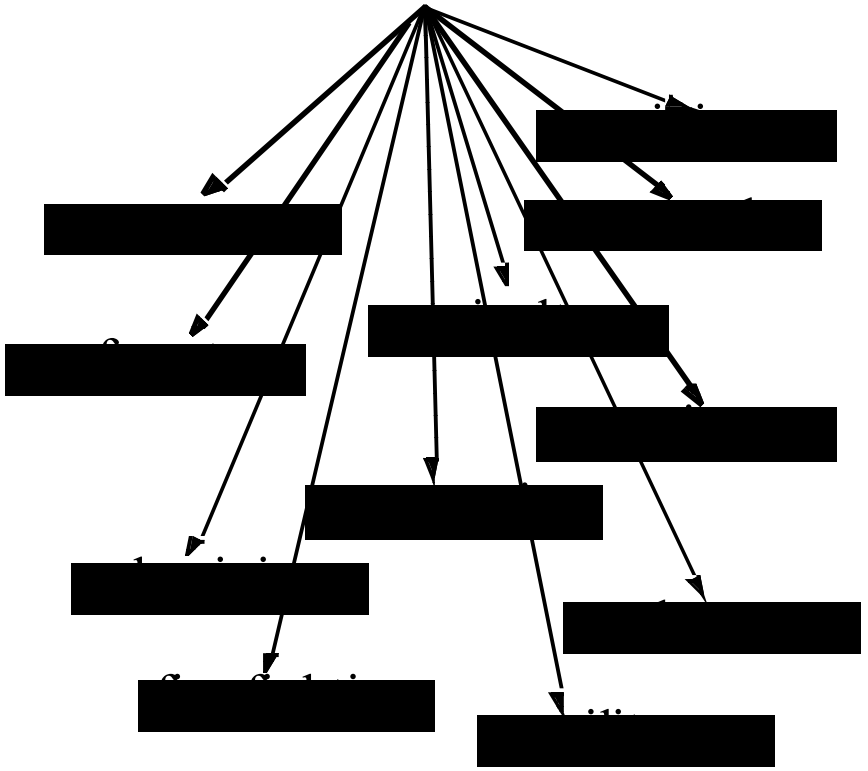
Indoor

Structured Environments



Outdoor

Unstructured Environments



Automated Guided Vehicle (AGV)

In March 2012, Amazon acquired robotics company Kiva Systems. Kiva manufactures squat, orange, ottoman-sized automated guided vehicles, capable of lifting custom shelfs and transporting them across the warehouse.

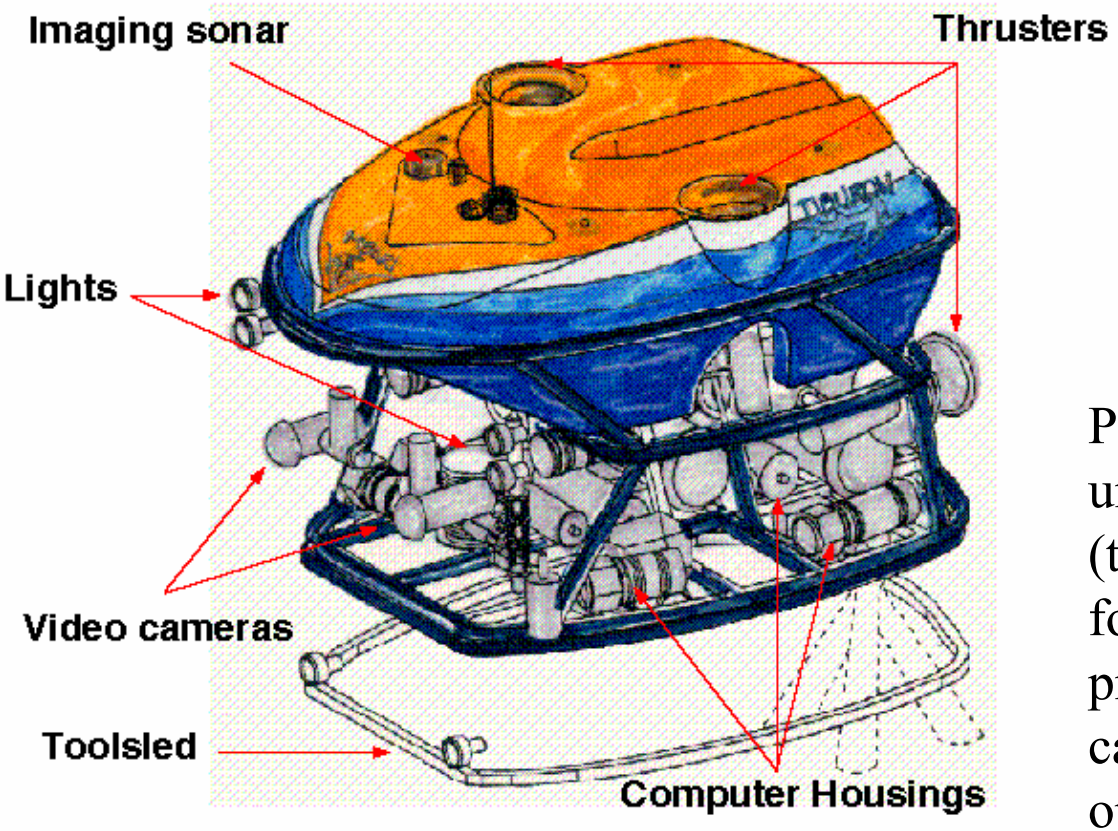
Hospital Robot

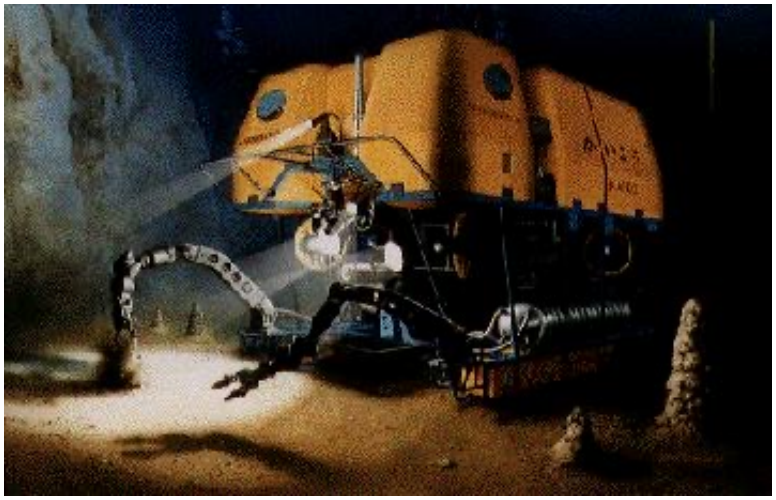
Aethon TUG Robot: Automating Internal Logistics

Roomba Cleaning Robot

The iRobot Roomba i7+ robot vacuum takes convenience to a whole new level with the first-of-its-kind Clean Base Automatic Dirt Disposal, which automatically empties the contents of the Roomba i7+ dust bin into the Clean Base.

ROV Tiburon Underwater Robot



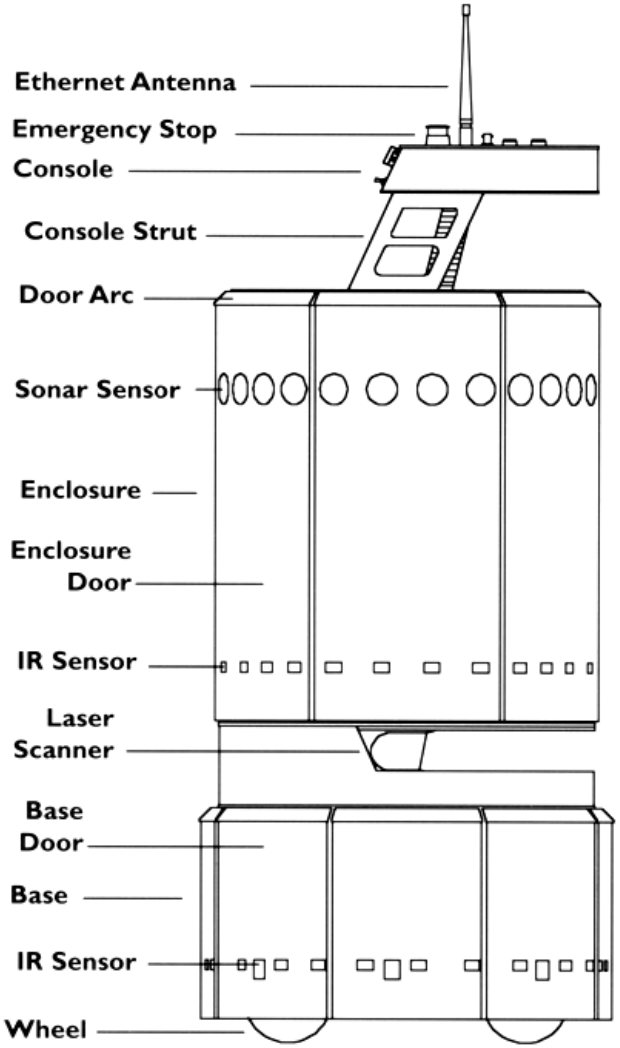


Picture of robot ROV Tiburon for underwater archaeology (teleoperated)- used by MBARI for deep-sea research, this UAV provides autonomous hovering capabilities for the human operator.

The Pioneer

Picture of Pioneer, the teleoperated robot that is supposed to explore the Sarcophagus at Chernobyl

The B21 Robot

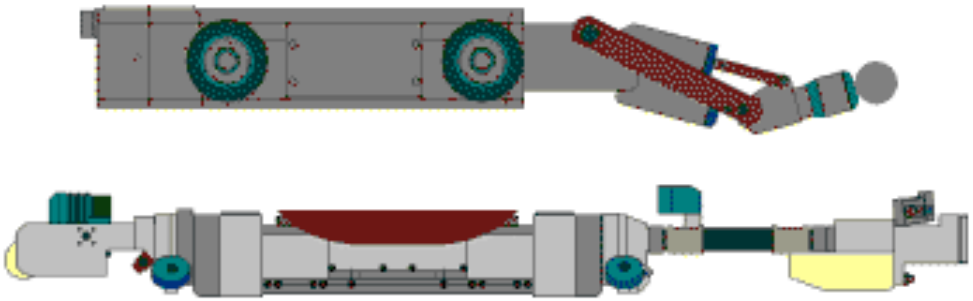


B21 of Real World Interface is a sophisticated mobile robot with up to three Intel Pentium processors on board. It has all different kinds of on board sensors for high performance navigation tasks.
http://www.rwii.com

Khepera IV Robot, K-Team

The robot includes an array of 8 Infrared Sensors for obstacle detection with 4 more for fall avoidance or line following as well as 5 Ultrasonic Sensors for long range object detection.

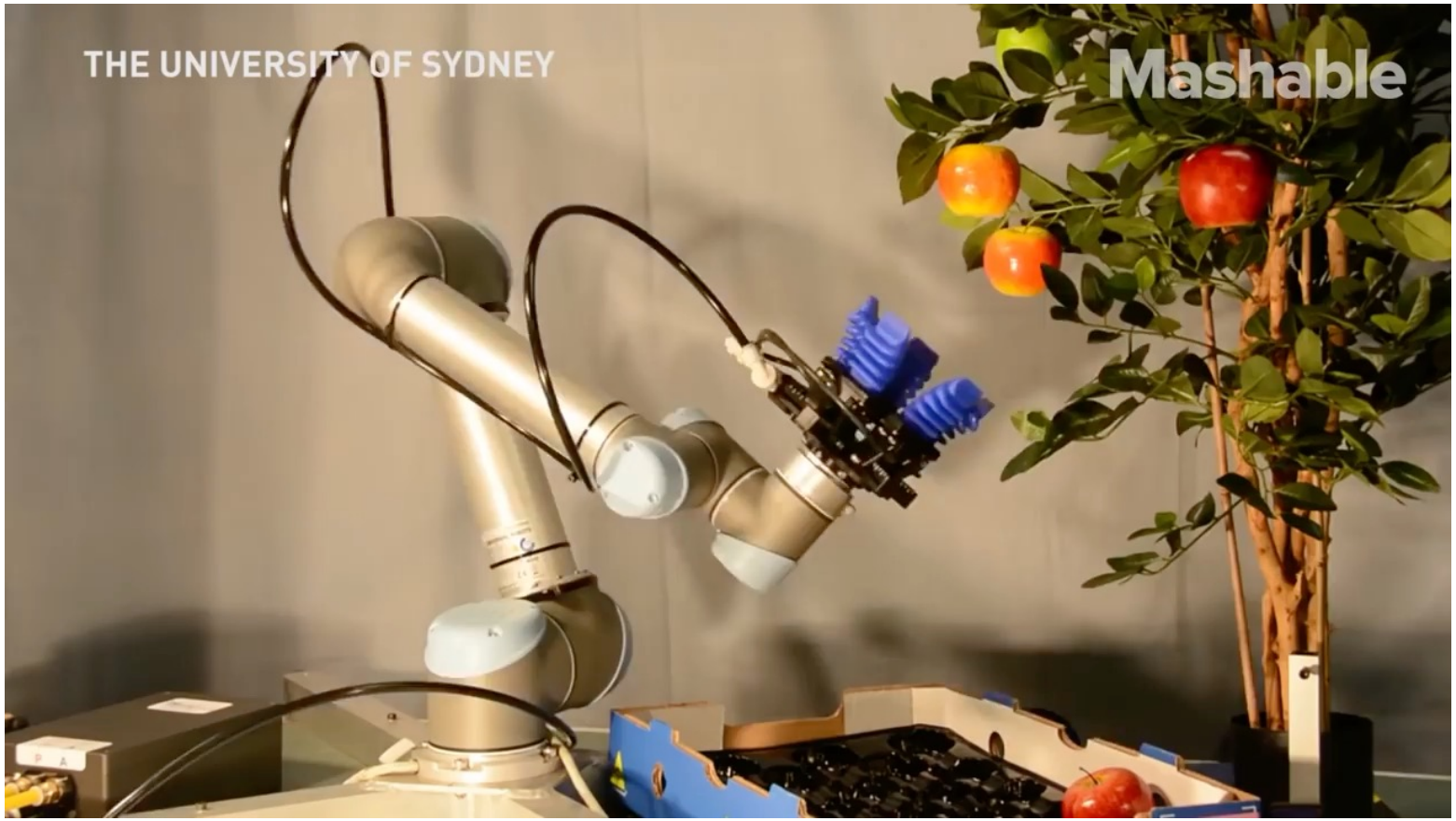
Robots for Tube Inspection



HÄCHER robots for sewage tube inspection and reparation. These systems are still fully teleoperated. http://www.haechler.ch

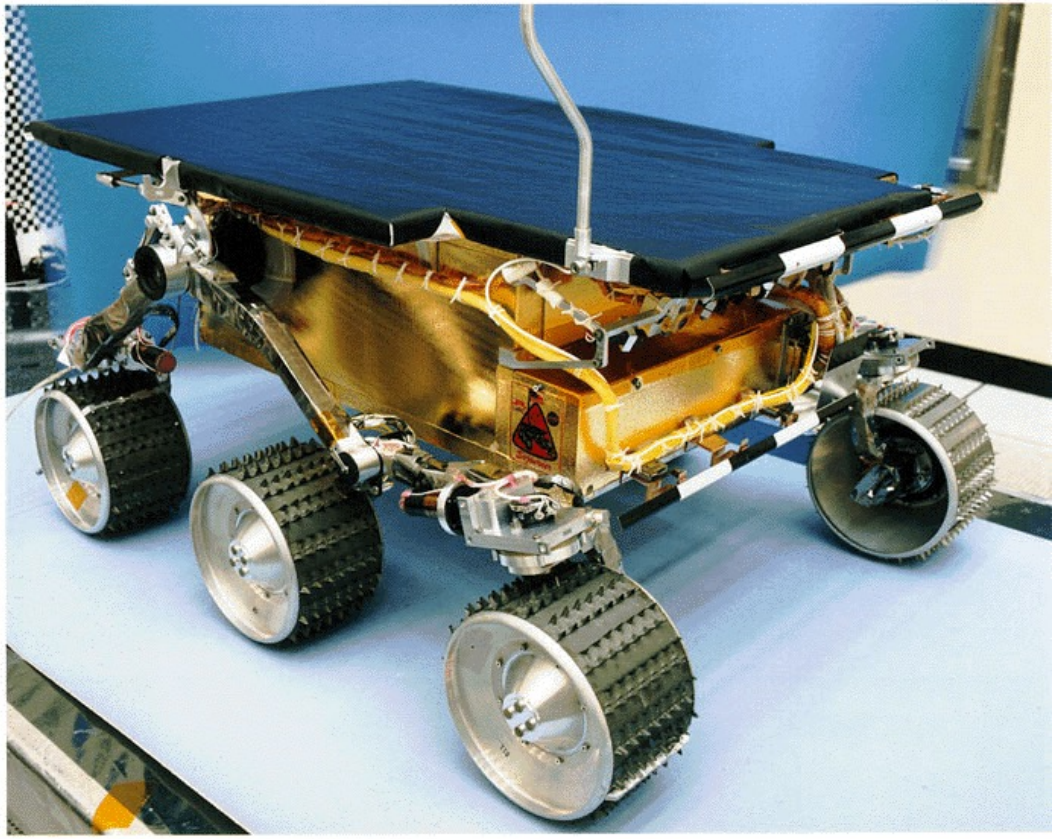
EPFL / SEDIREP: Ventilation inspection robot

Robotic Farming



The University of Sydney's Australian Centre for Field Robotics are pioneers when it comes to robotic farming. They give us a sneak peek of how future farms and orchards will operate in the era of mass automation.

Sojourner, First Robot on Mars



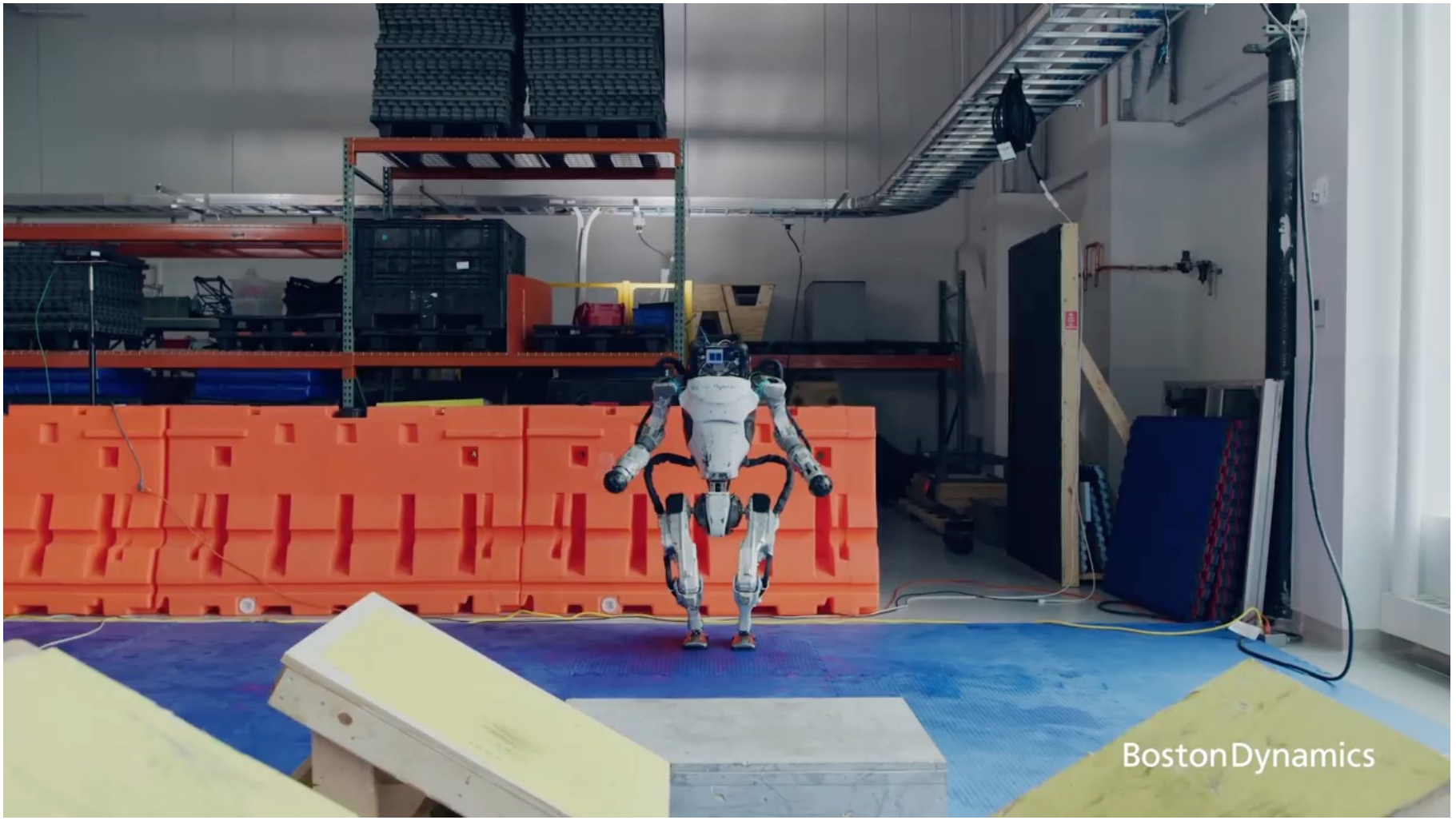
2003 Mars Rover

Press Release Animation

Dan Maas
dmaas@dcine.com

The mobile robot Sojourner was used during the Pathfinder mission to explore the mars in summer 1997. It was nearly fully teleoperated from earth. However, some on board sensors allowed for obstacle detection.
http://ranier.oact.hq.nasa.gov/telerobotics_page/telerobotics.shtm

Atlas | Partners in Parkour

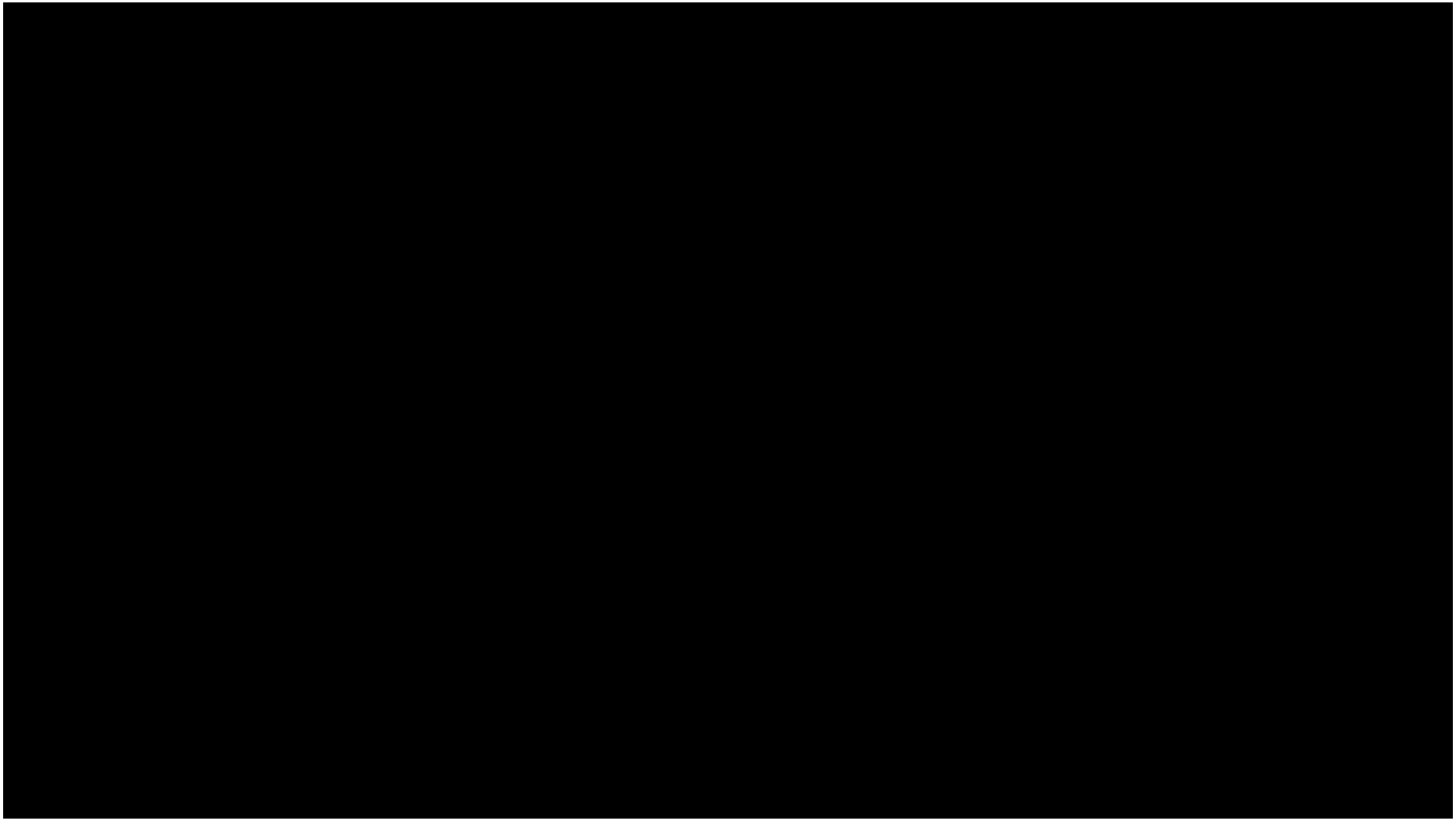


Boston Dynamics

Unmanned Aerial Vehicle (UAV)

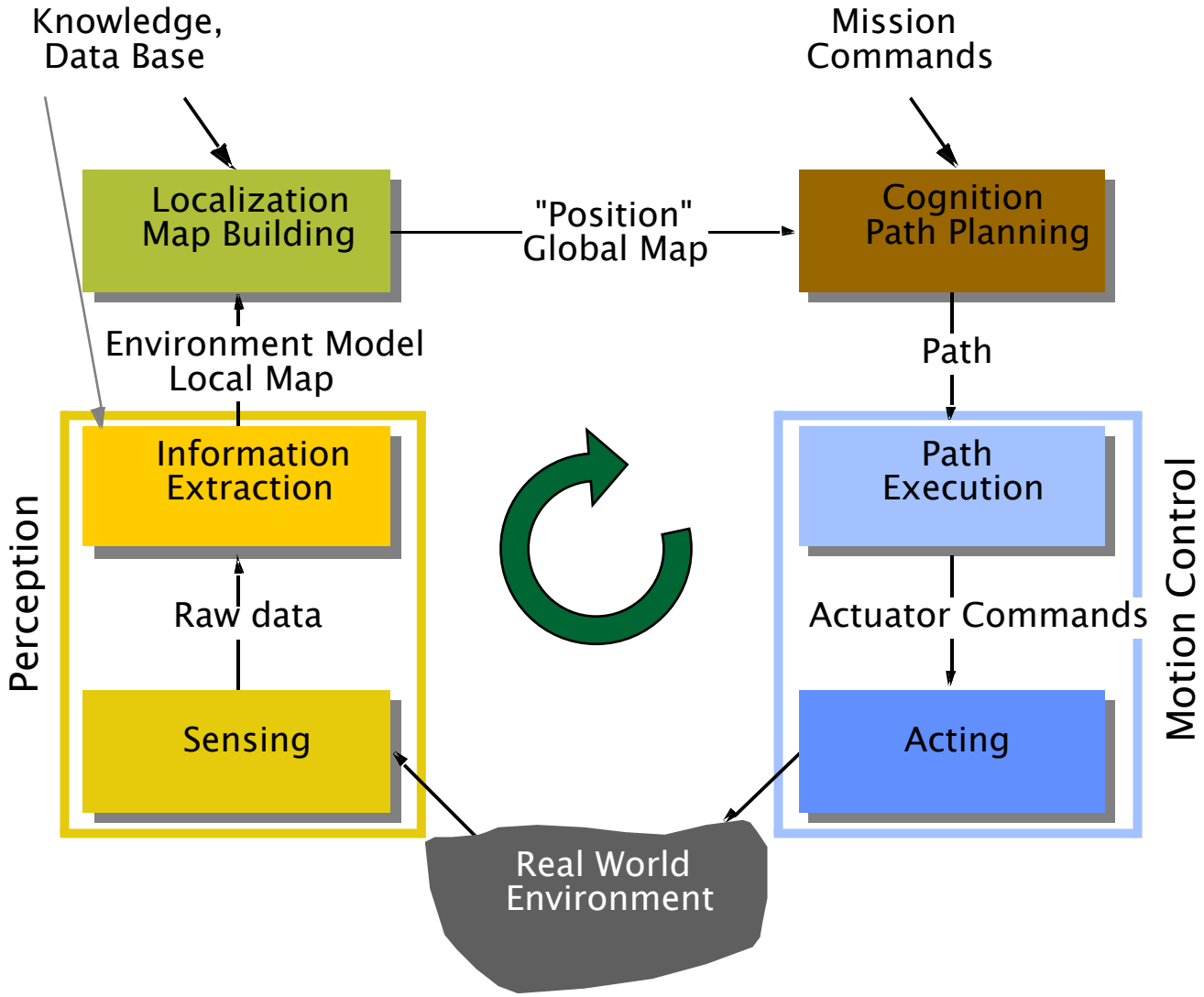
Unmanned Aircraft Systems & UAV Autonomy
https://www.swri.org/unmanned-aircraft-systems

Insect Drone



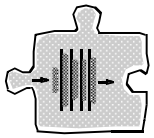
Researchers introduce a new generation of tiny, agile drones

Control Scheme for Mobile Robot System

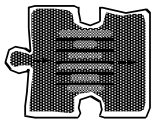


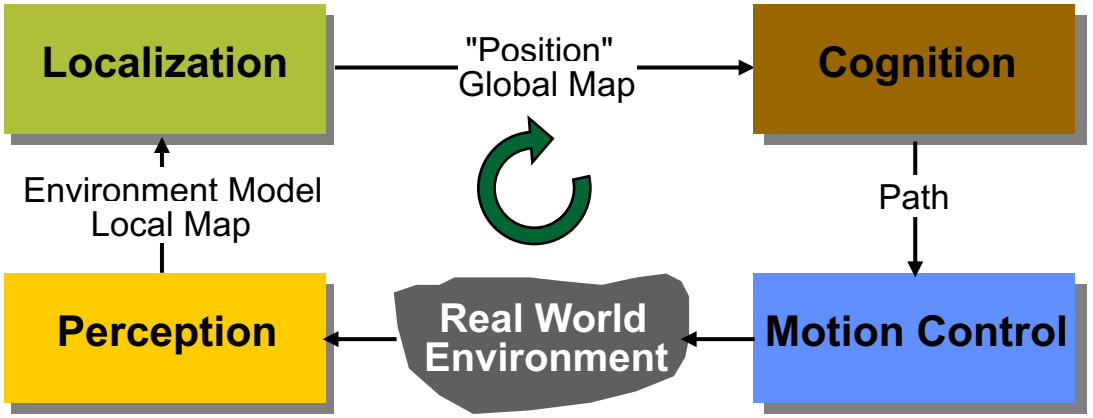
Control Architectures/Strategies

- Control Loop
  - Dynamically changing
  - No compact model available
  - Many sources of uncertainty
- Two approaches
  - Classic AI

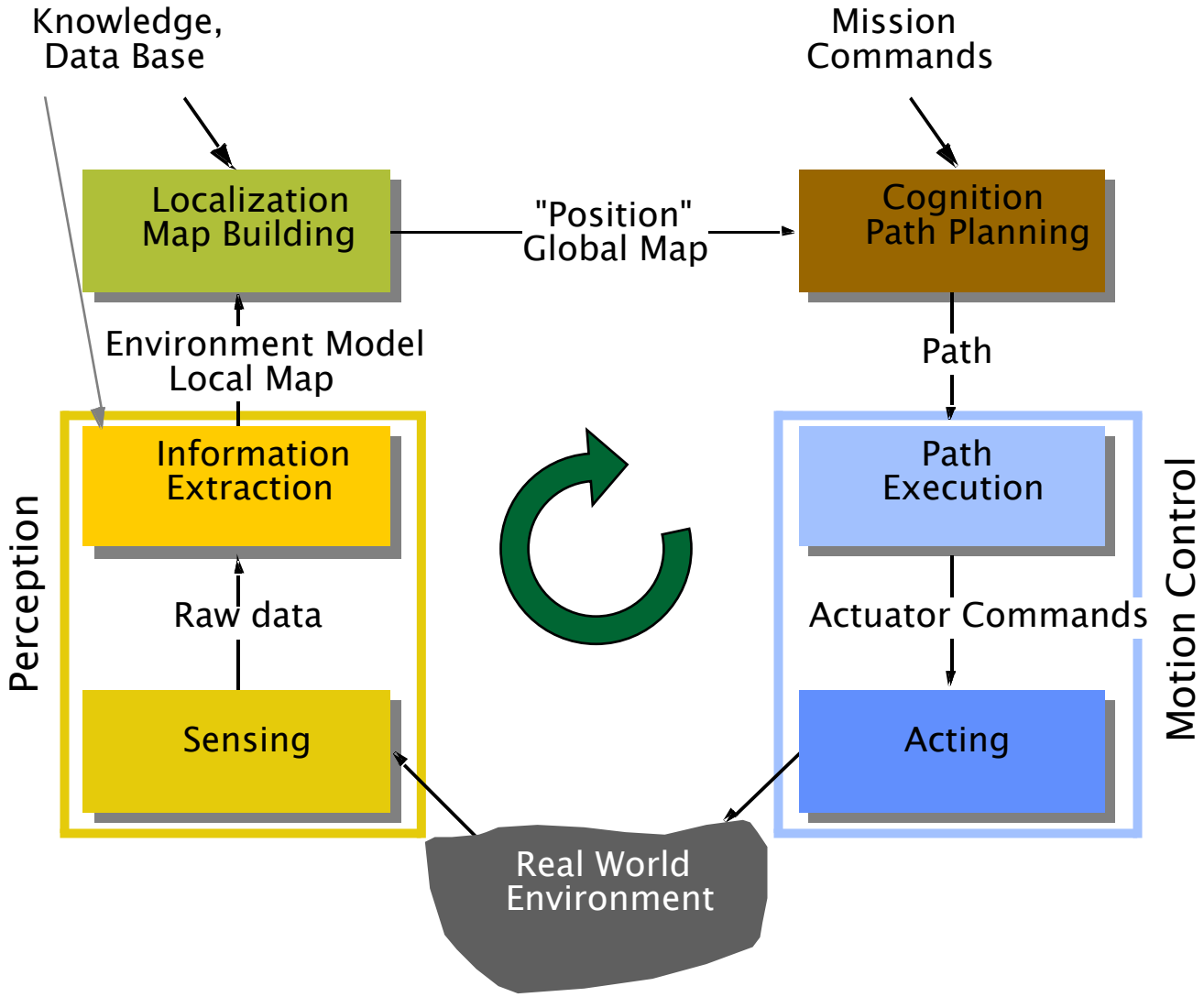


  - New AI



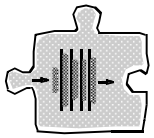


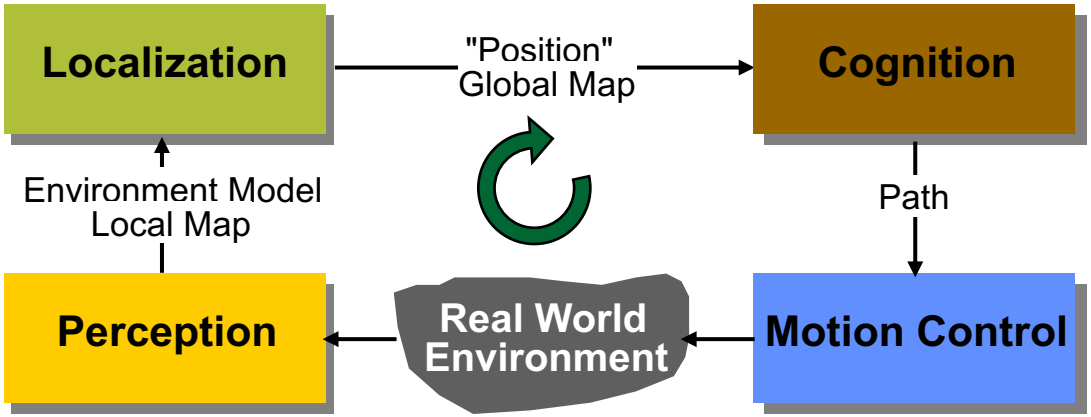
Control Scheme for Mobile Robot System



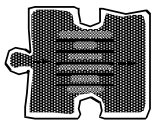
Control Architectures/Strategies

- Control Loop
  - Dynamically changing
  - No compact model available
  - Many sources of uncertainty
- Two approaches
  - Classic AI



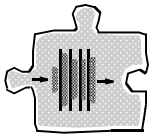


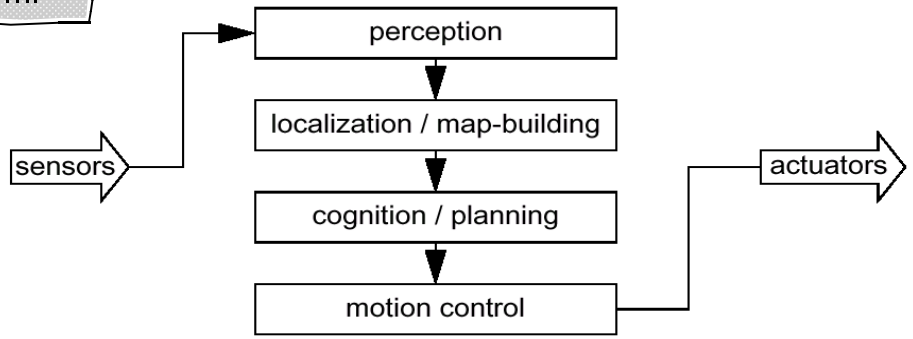
  - New AI

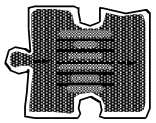


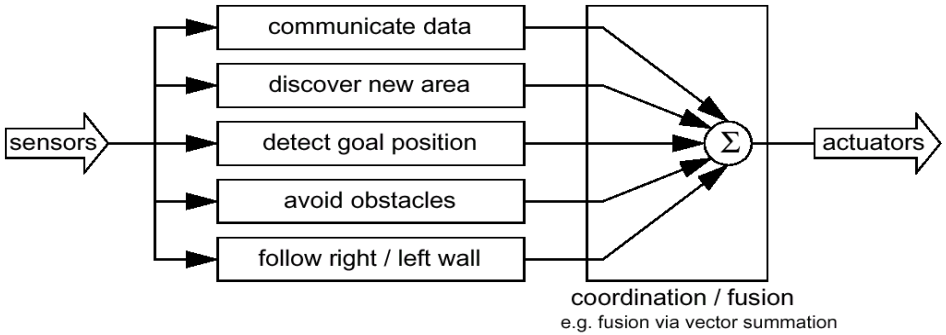
Two Approaches

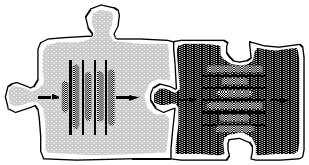
- Classical AI
  - Model-based navigation
  - Complete modeling
  - Function-based
  - Horizontal decomposition
- New AI
  - Behavior-based navigation
  - Sparse or no modeling
  - Behavior-based
  - Vertical decomposition
  - Bottom up
- Possible solution
  - Combined approaches





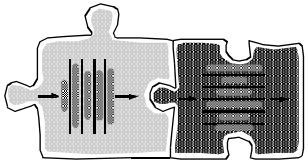


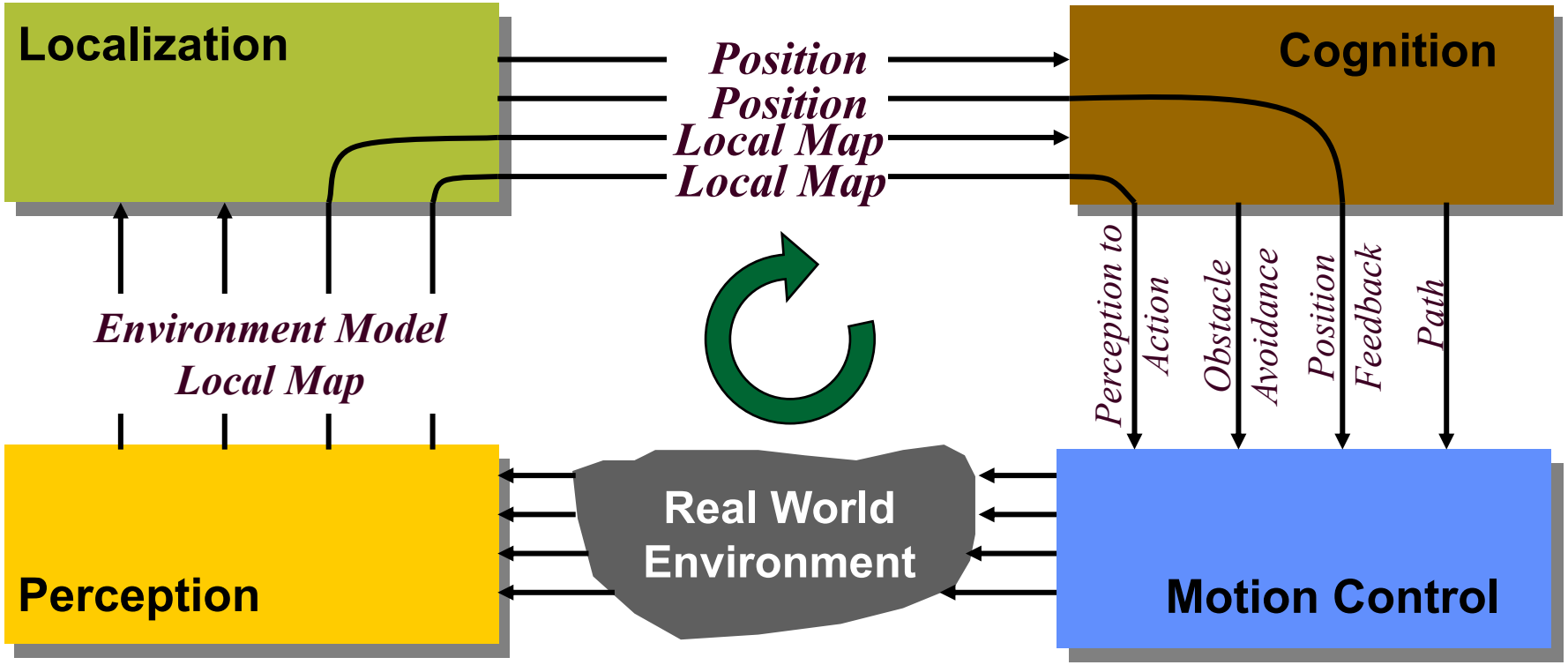




Mixed Approach

- Mixed approach depicted into the general control scheme





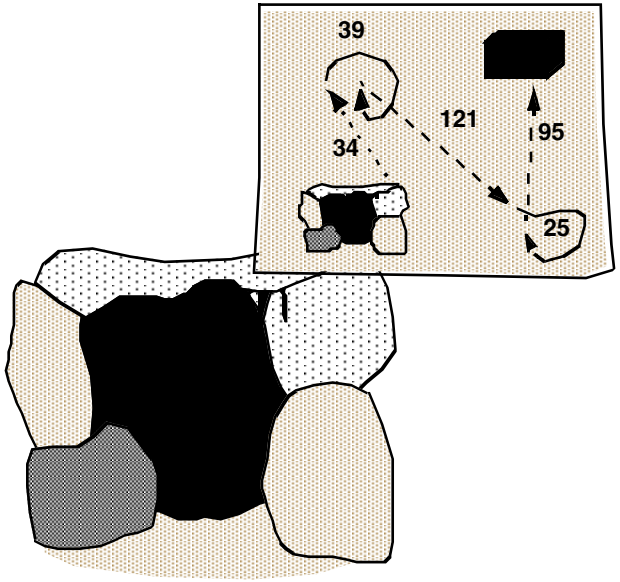
Keys for Autonomous Navigation

- Environment Representation
  - Continuous metric: x, y, θ
  - Discrete metric: metric grid
  - Discrete topological: topological grid
- Environment Modeling
  - Raw sensor data (laser range data, grayscale images)
    - Large volume of data, low distinctiveness
    - Make use of all acquired information
  - Low level features (corners, lines, other geometric features)
    - Medium volume of data, average distinctiveness
    - Filters out the useful information, still ambiguities
  - High level features (doors, a car, the Eiffel tower)
    - Low volume of data, high distinctiveness
    - Filters out useful information, few/no ambiguities, not enough info.

Environment Representation & Modeling

- Environment representation and modeling: How to do?

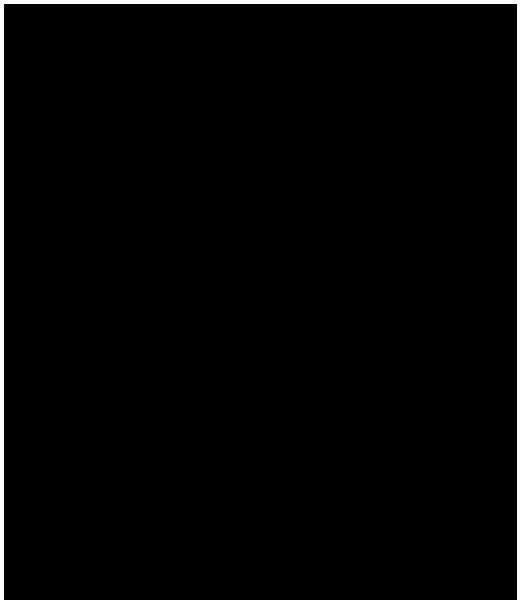
Odometry



How to find a treasure

Not applicable

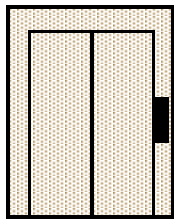
Modified Environment



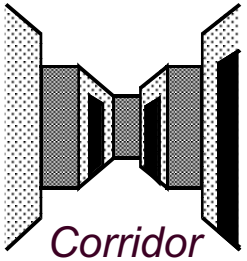
Landing at night

Expensive, inflexible

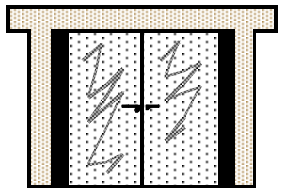
Feature-based Navigation



Elevator door



Corridor crossing

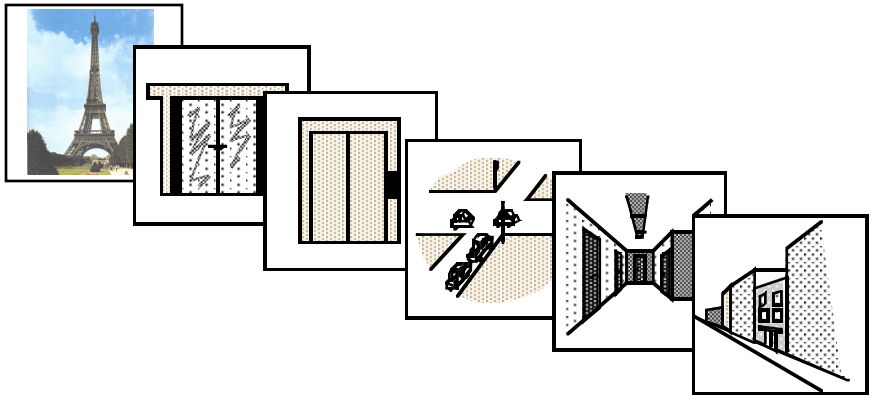


Entrance

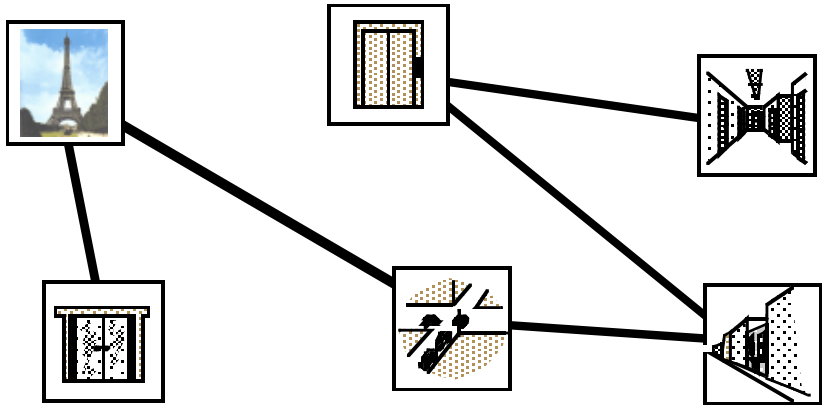
Still a challenge for artificial systems

Environment Map Representations

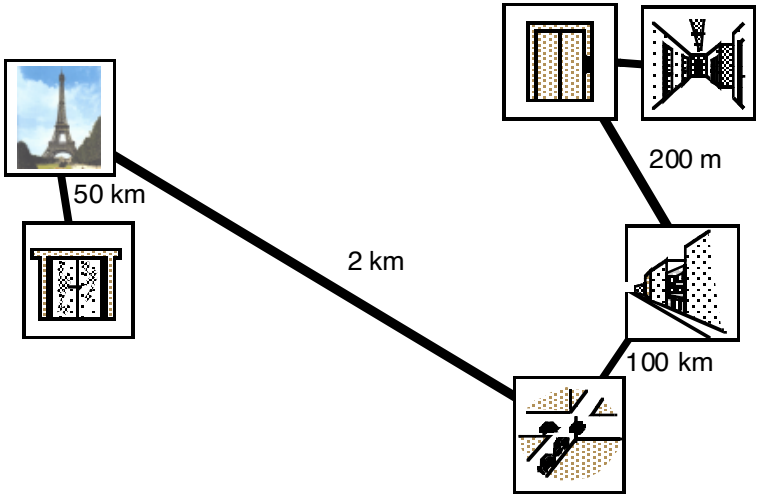
- Recognizable locations



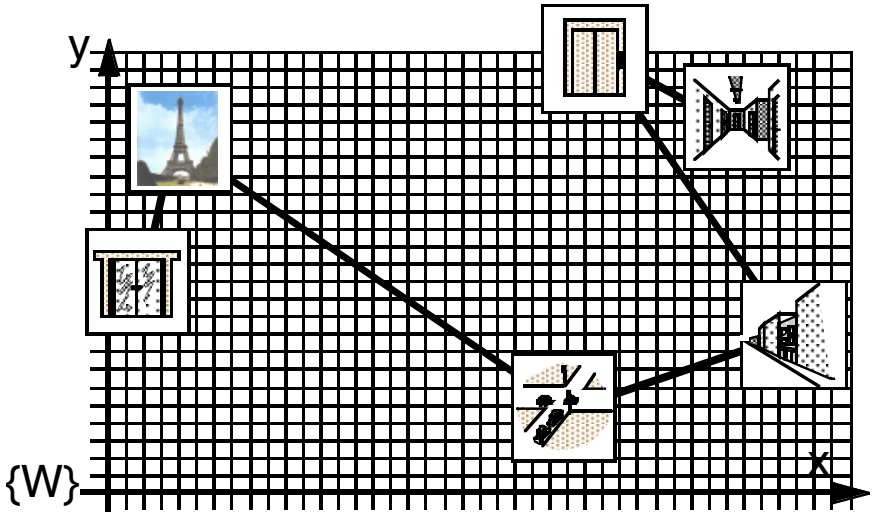
- Topological maps



- Metric topological maps



- Fully metric maps (cont. or disc.)

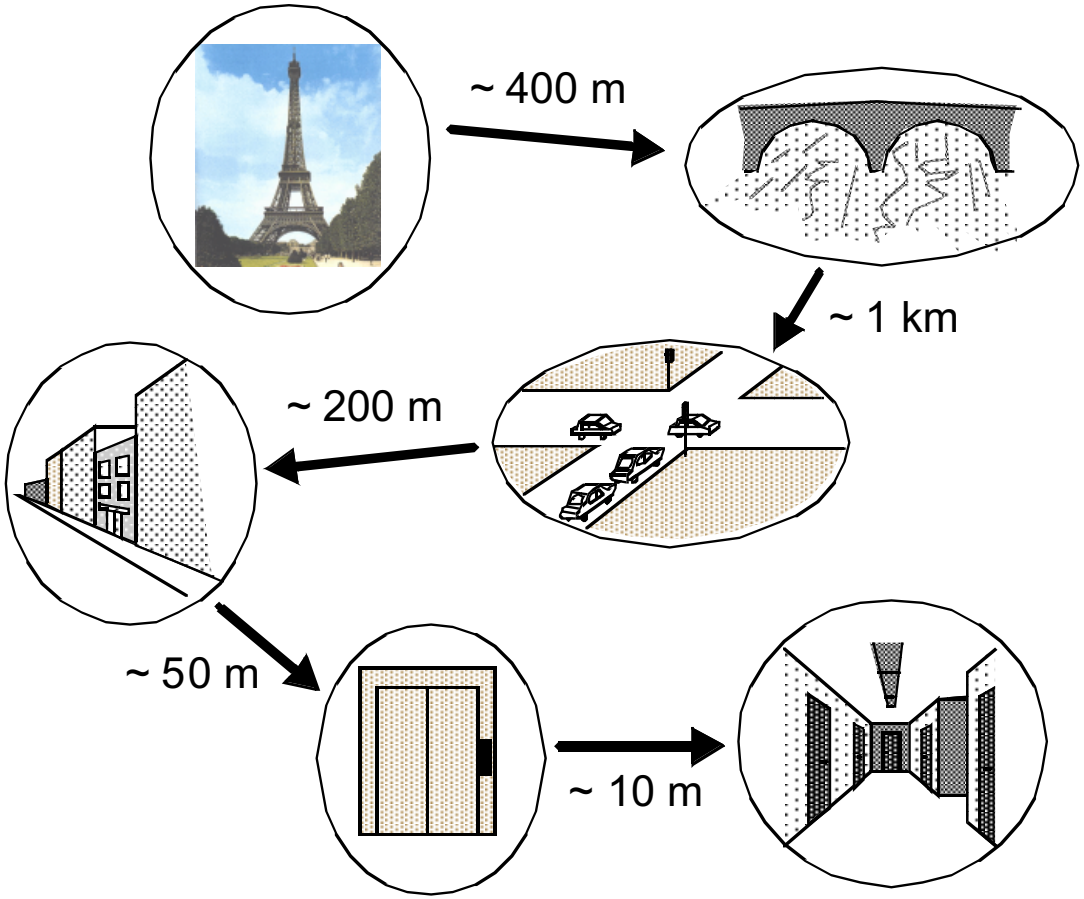


Environment Model

- Continuous
  - Position in x, y, θ
- Discrete
  - Metric grid
  - Topological grid
- Raw data
  - As perceived by sensors
- Features
  - Environmental structures which are static, always perceptible with the current sensor system and locally unique
  - e.g., geometric elements (lines, walls, columns, …), a railway station, a river, the Eiffel tower, skyscraper

Human Navigation

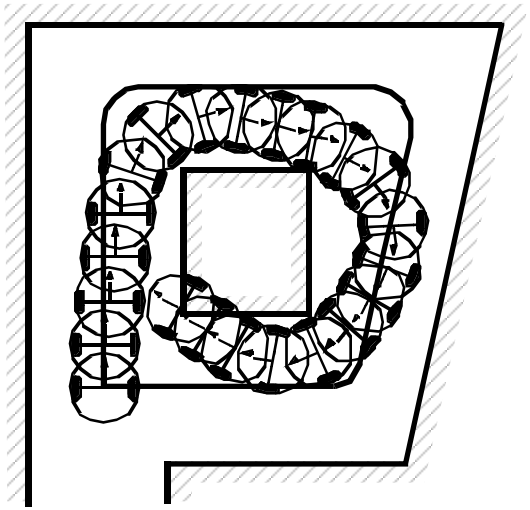
- Topological with imprecise metric information



Method for Navigation

Approaches with Limitations

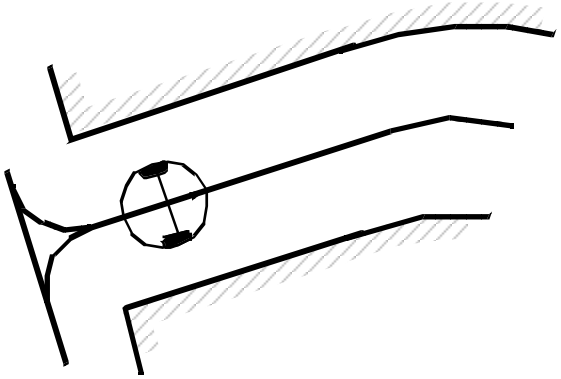
- Incrementally (dead reckoning)



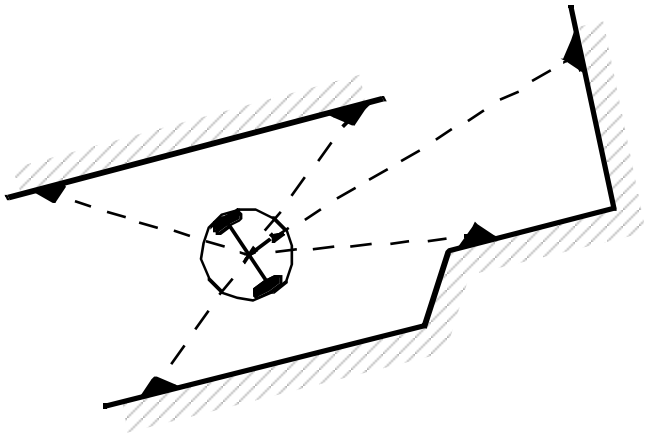
Odometric or inertial sensors (gyro)

Not applicable

- Modifying the environment (artificial landmarks/beacons)



Inductive or optical tracks (AGV)



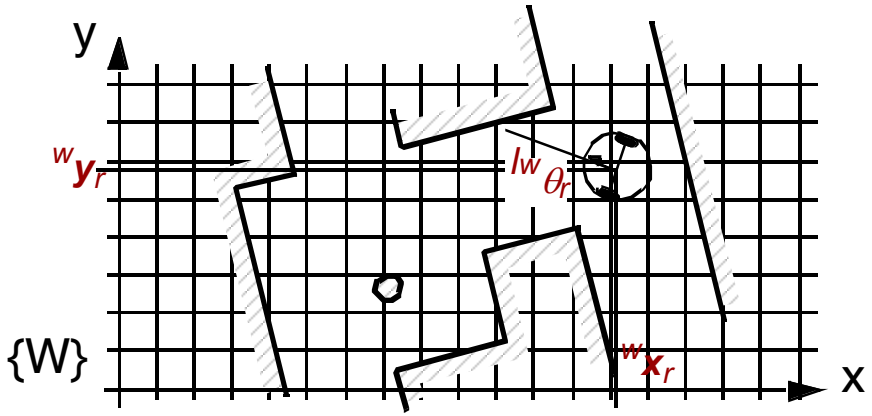
Reflectors or barcodes

Expensive, inflexible

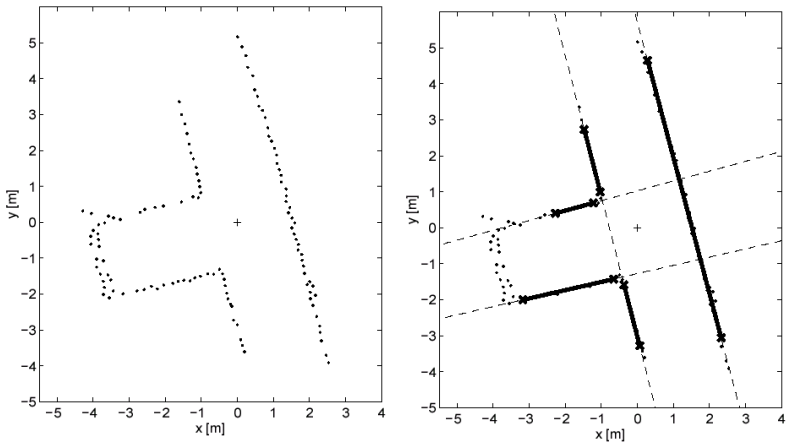
Method for Localization

The quantitative metric approach

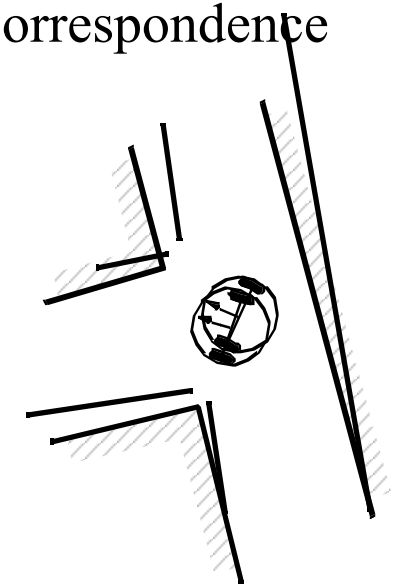
1. A priori map: graph, metric



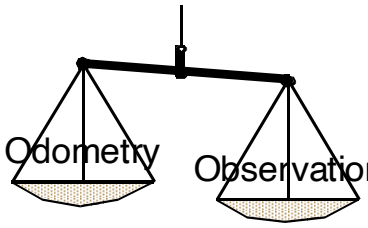
2. Feature extraction



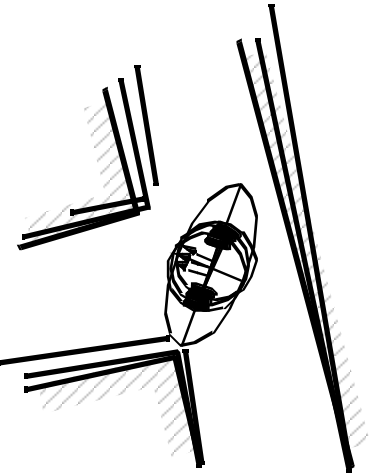
3. Matching: Find correspondence of features



4. Position estimation (e.g., Kalman filter, Markov)

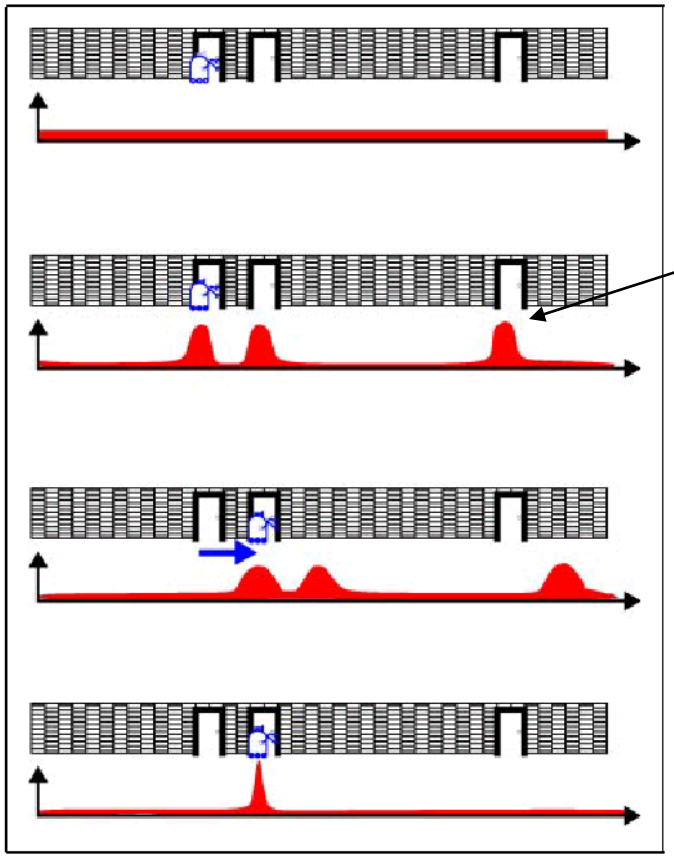


- Representation of uncertainties
- Optimal weighting according to a priori statistics



Gaining Information through Motion

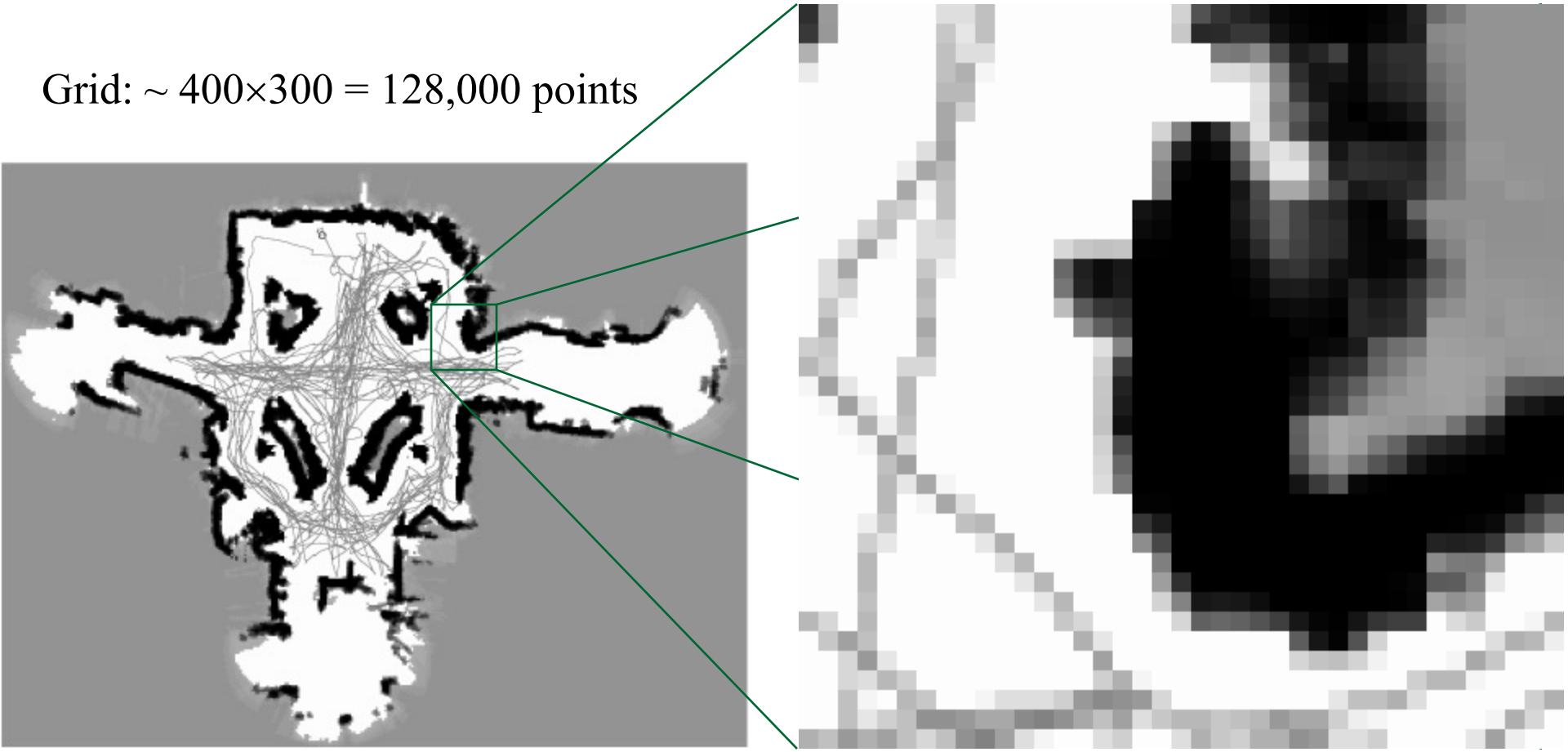
- Multi-hypotheses tracking



Grid-Based Metric Approach

- Grid map of the Smithsonian Museum of American History in Washington D.C.

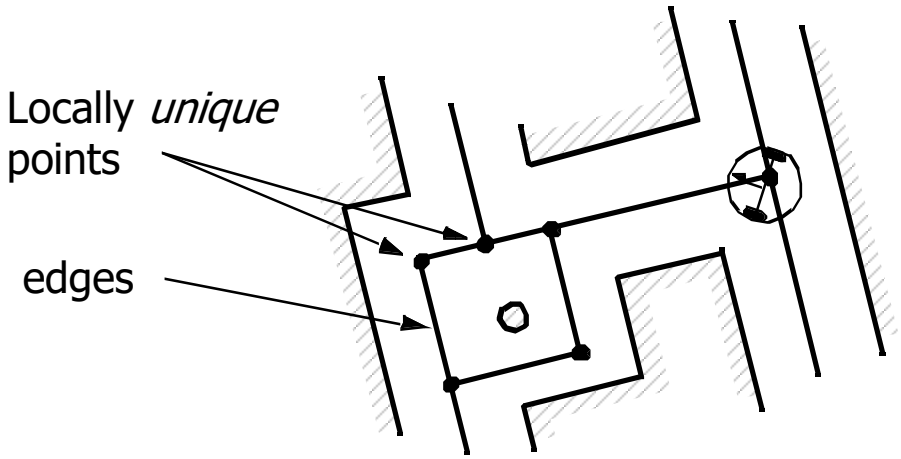
Grid: ~ 400×300 = 128,000 points



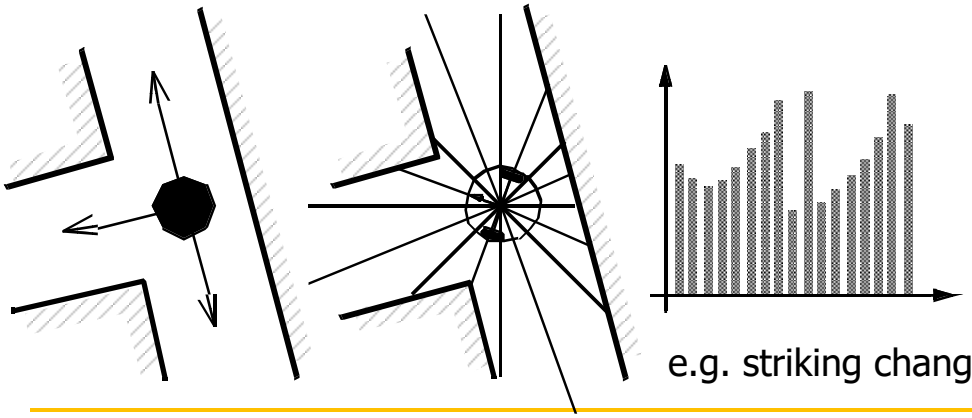
Methods for Localization

The quantitative topological approach

1. A priori map: graph



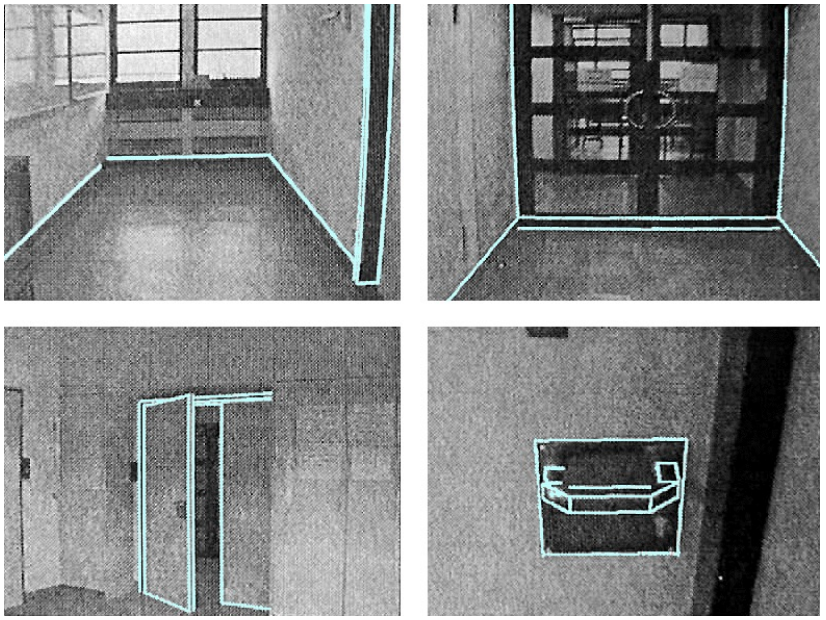
2. Method for determining the *local uniqueness*



e.g. striking changes on raw data level or highly distinctive features

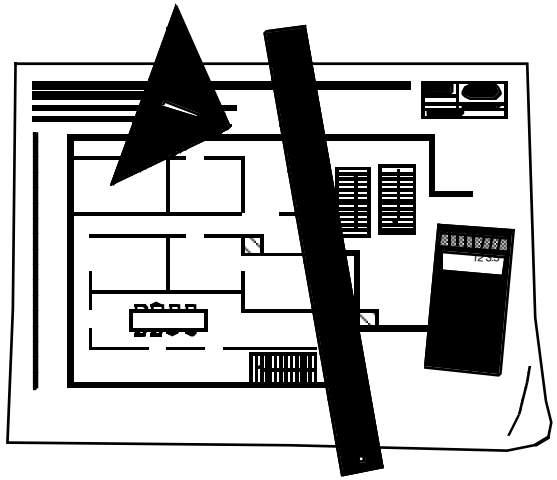
3. Library of *driving behaviors*

e.g., wall or midline following, blind step, enter door, application specific behaviors. Example: video-based navigation with nature landmarks



Map Building: How to Establish

- By hand



- Automatically: Map Building
  - The robot *learns* its environment
  - Motivation:
    - By hand: hard and costly
    - Dynamically changing environment
    - Different look due to different perception

Map Building: How to Establish

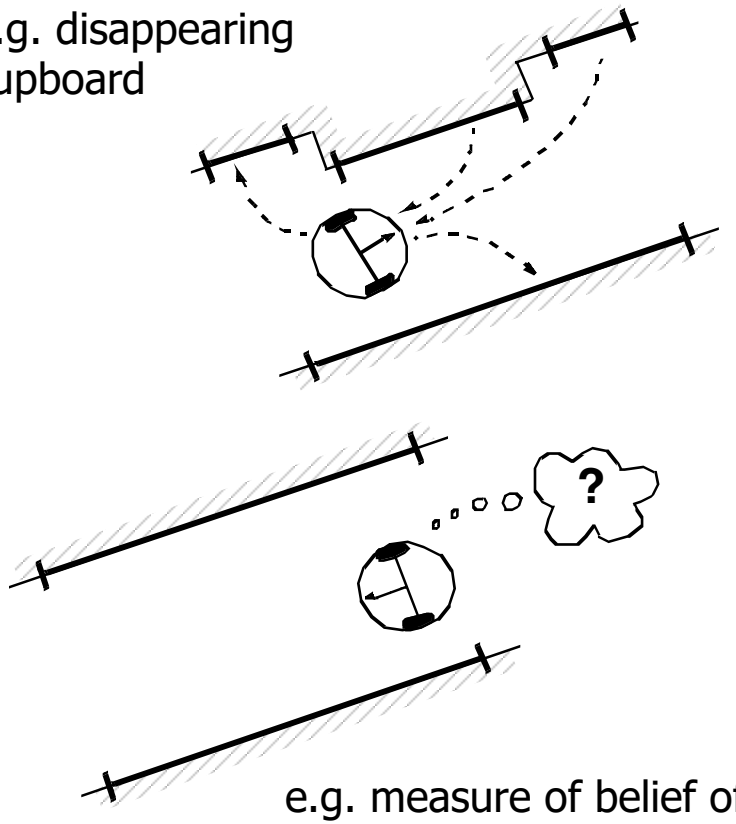
- Basic requirements of a map
  - A way to incorporate newly sensed information into the existing world model
  - Information and procedures for estimating the robot's position
  - Information to do path planning and other navigation task (e.g. obstacle avoidance)
- Measure of quality of a map
  - Topological correctness
  - Metrical correctness

- But: Most environments are a mixture of predictable and unpredictable features → hybrid approach
  - Model-based vs. behavior-based

Map Building: The Problems

- Map maintaining: Keep track of changes in the environment

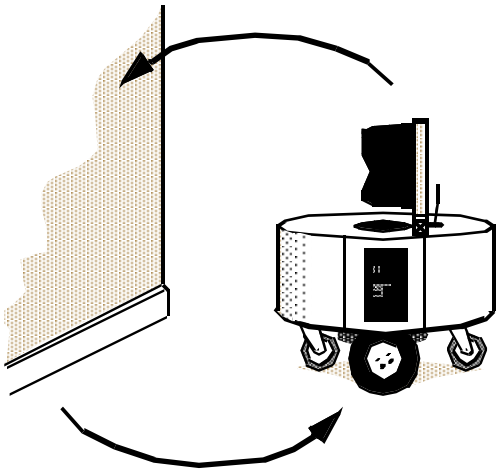
e.g. disappearing cupboard



e.g. measure of belief of each environment feature

- Representation and reduction of uncertainty
  - Probability density for feature positions
  - Additional exploration strategies

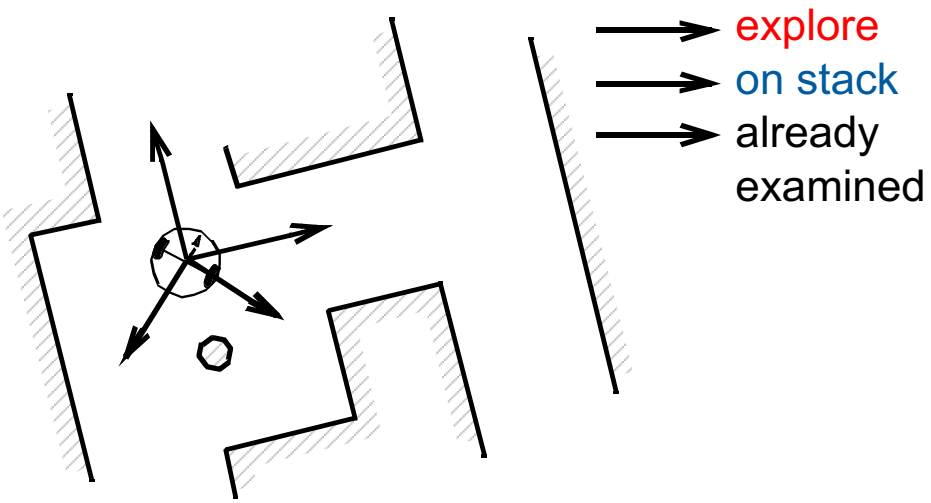
Position of robot → position of wall



Position of wall → position of robot

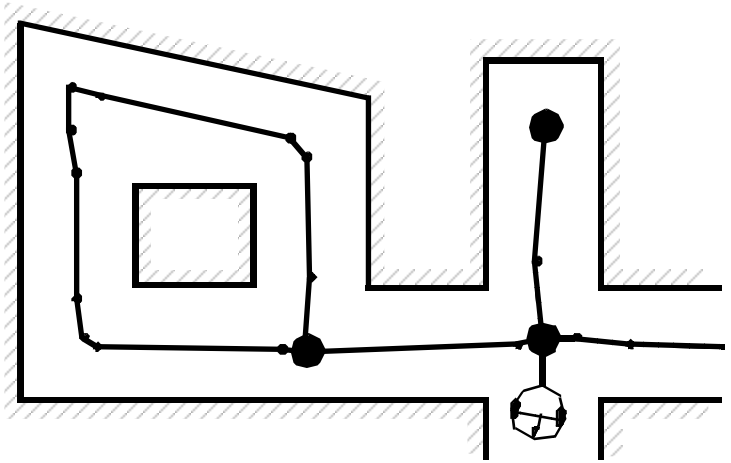
Exploration and Graph Construction

- Exploration

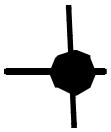


  - Provide correct topology
  - Must recognize already visited locations
  - Backtracking for unexplored openings

- Graph construction

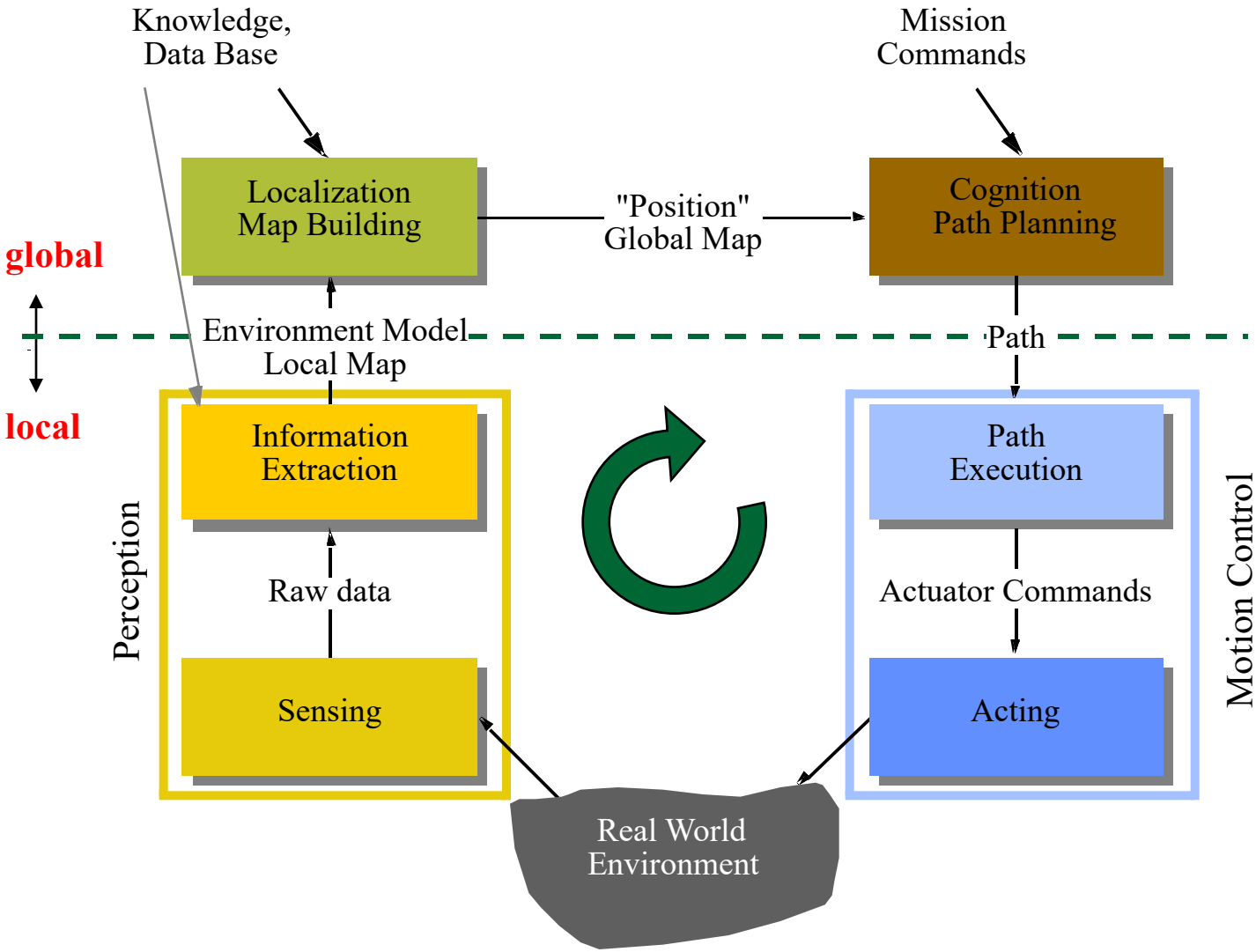


  - Where to put the nodes?
    - Topology-based: at distinctive locations



    - Metric-based: where features disappear or get visible

Control of Mobile Robot

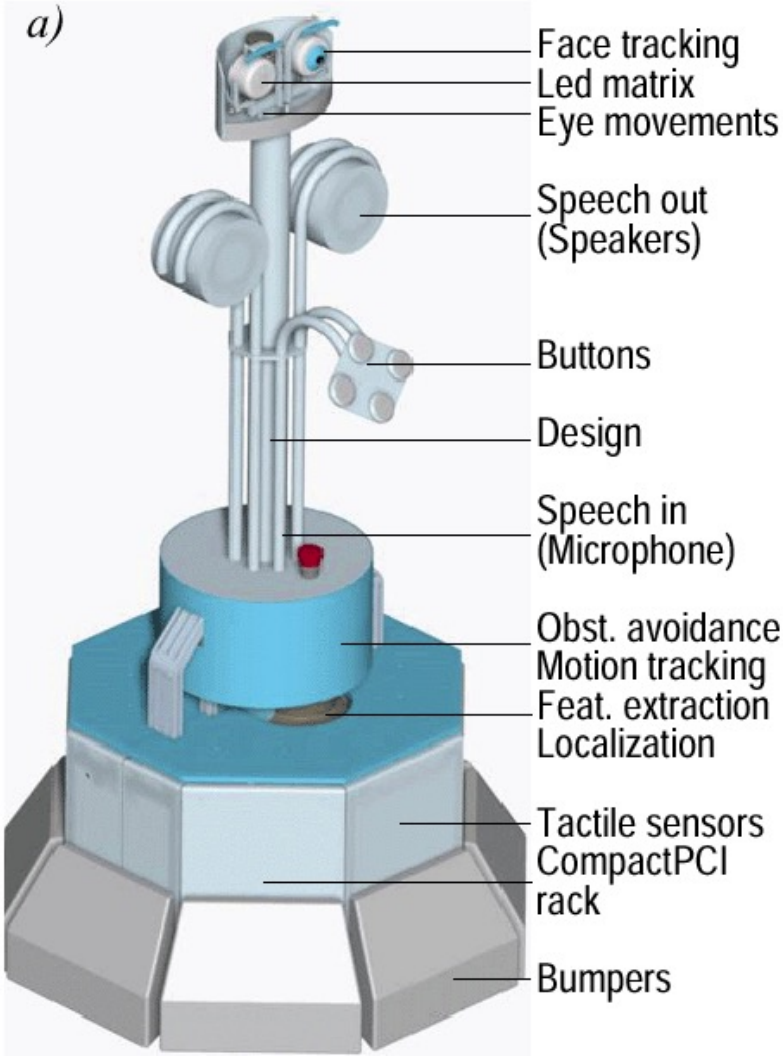


- Most functions for navigation are 'local' not involving localization and cognition
- Localization and global path planning → slower update rate, only when needed
- This approach is pretty similar to what human beings do

Demo Videos

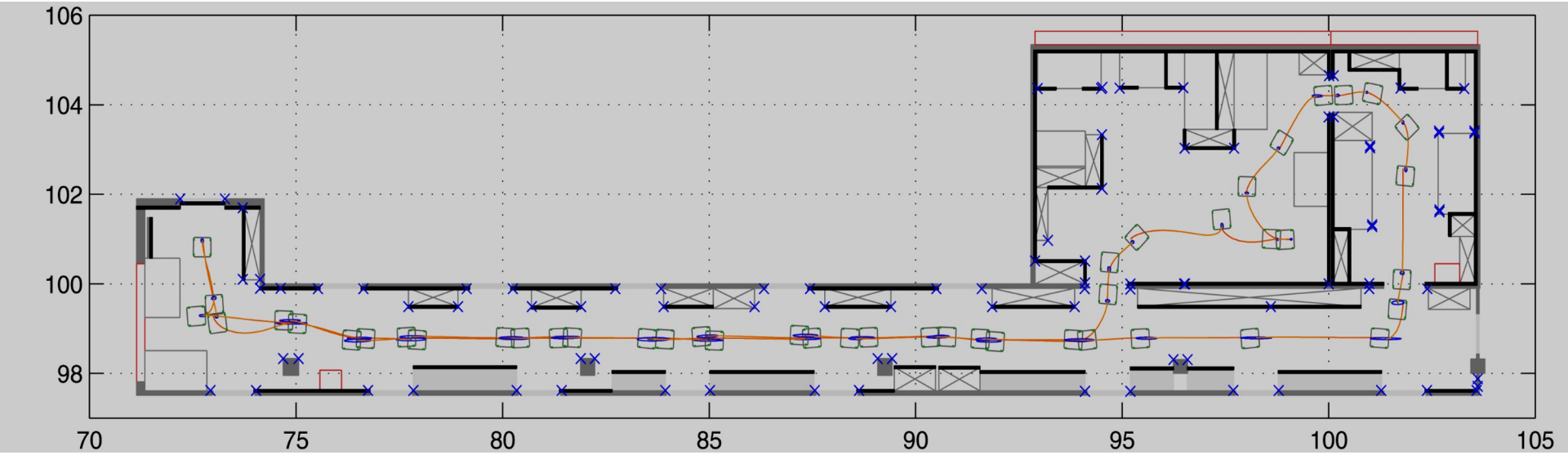
- Tour-Guide Robot (Nourbakhsh, CMU)
- Autonomous Indoor Navigation (Thrun, CMU)

Tour-Guide Robot (EPFL@expo.02)



Autonomous Indoor Navigation

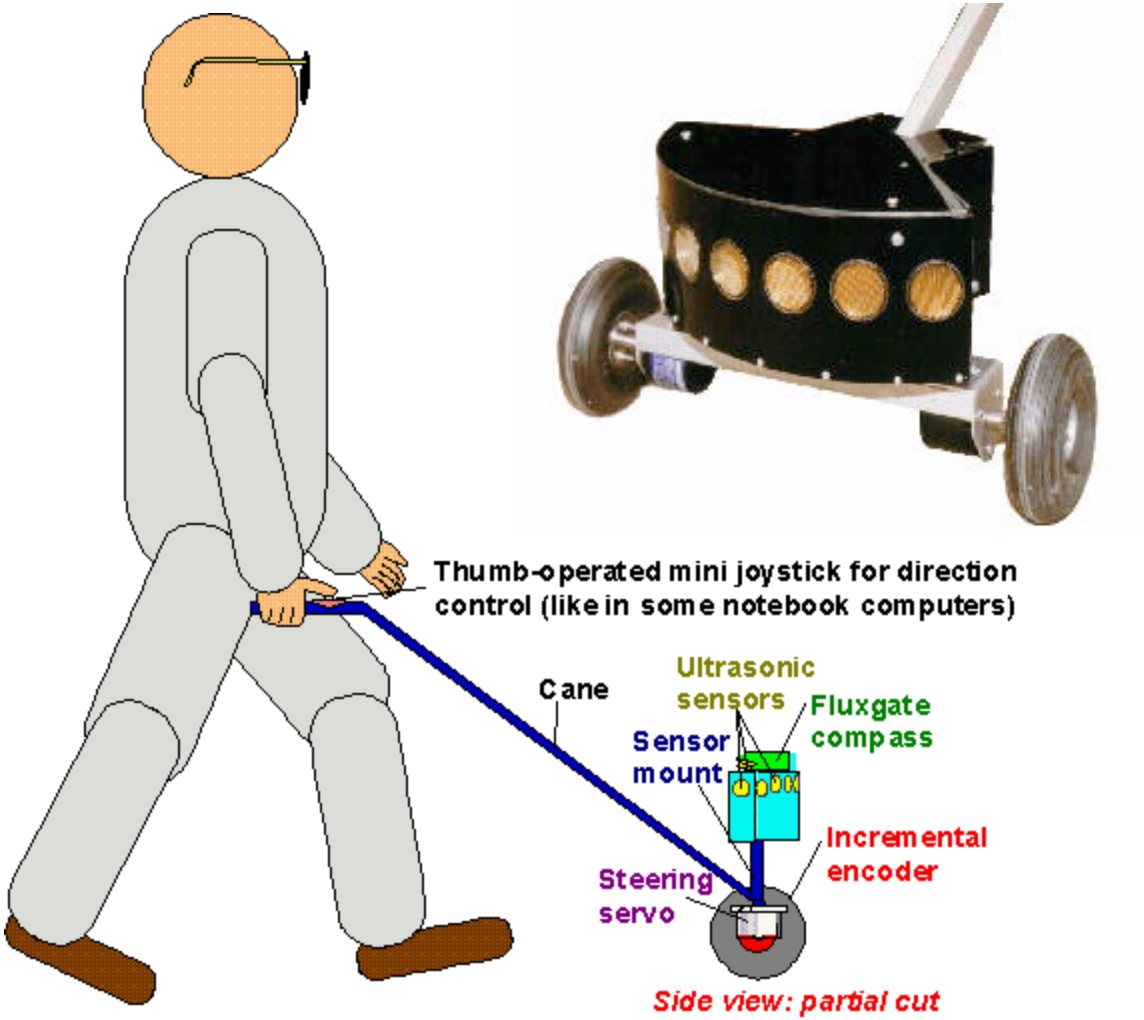
- Pygmalion EPFL
  - Very robust on-the-fly localization
  - Conducted 16 times > 1 km overall distance
  - 47 steps, 78 meter length, realistic office environment
  - One of the first systems with probabilistic sensor fusion
  - Partial difficult surface (laser), partial vertical edges (vision)



Demo Video

- Autonomous Robot for Planetary Exploration (ASL – EPFL)

GuideCane (U of Michigan)



LaserPlans Architectural Tool

- ActiveMedia Robotics

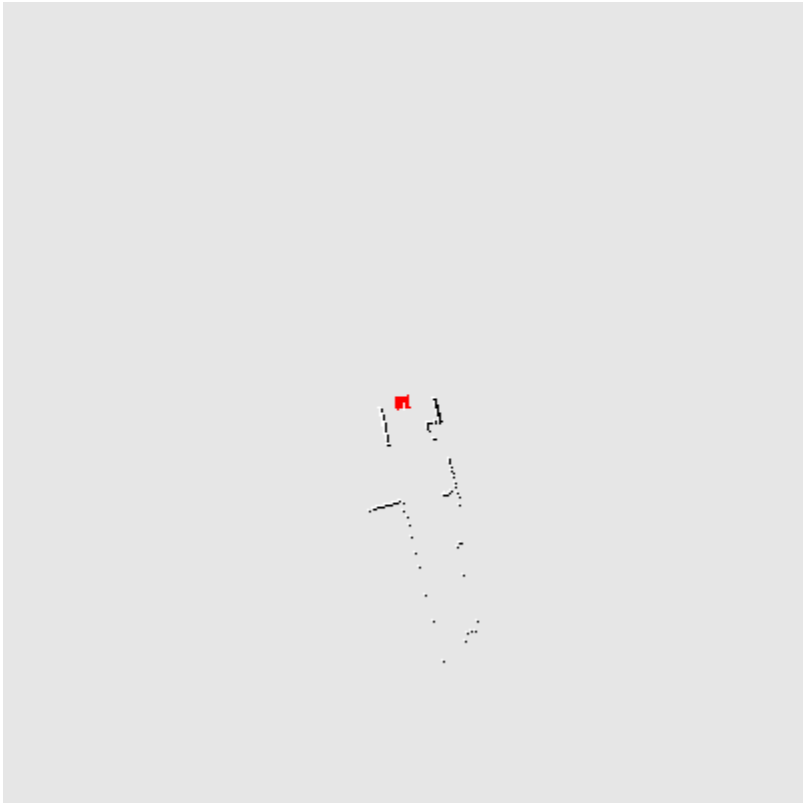
Demo Videos

- High-Speed Exploration and Mapping

- Turning Real Reality into Virtual Reality

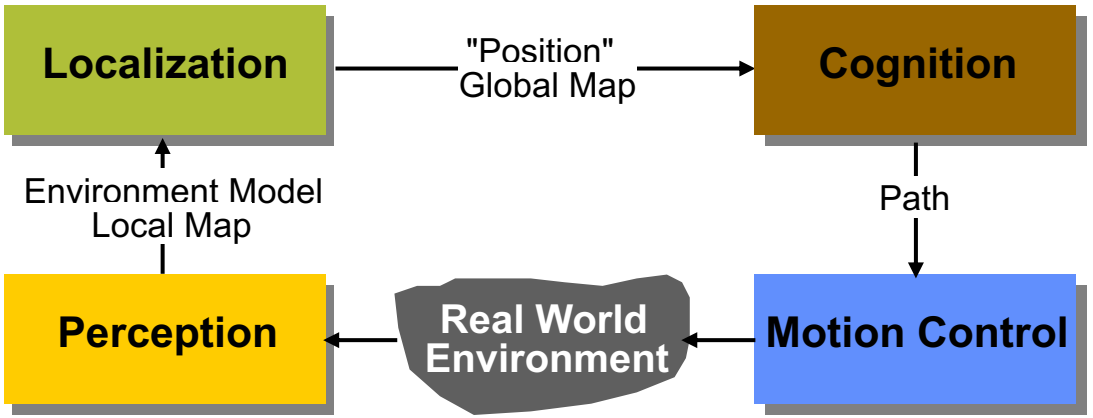
Outdoor Mapping (no GPS)

Real-Time Multi-robot Exploration



Mobile Robot Navigation

- Navigation is one of the most challenging competence for a mobile robot
- Success in navigation requires success at four building blocks
  - Perception
  - Localization
  - Cognition
  - Motion control

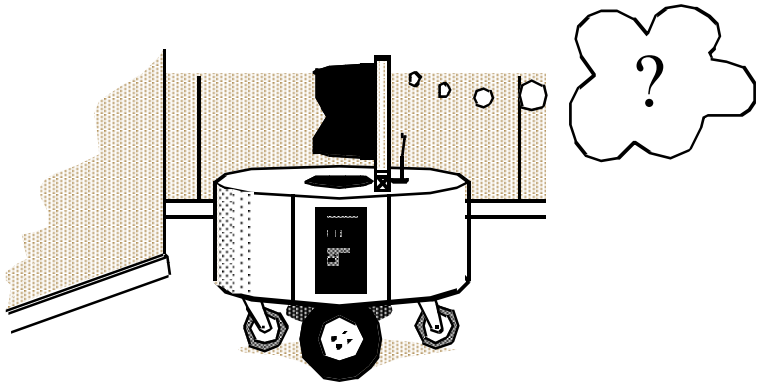


Mobile Robot Navigation

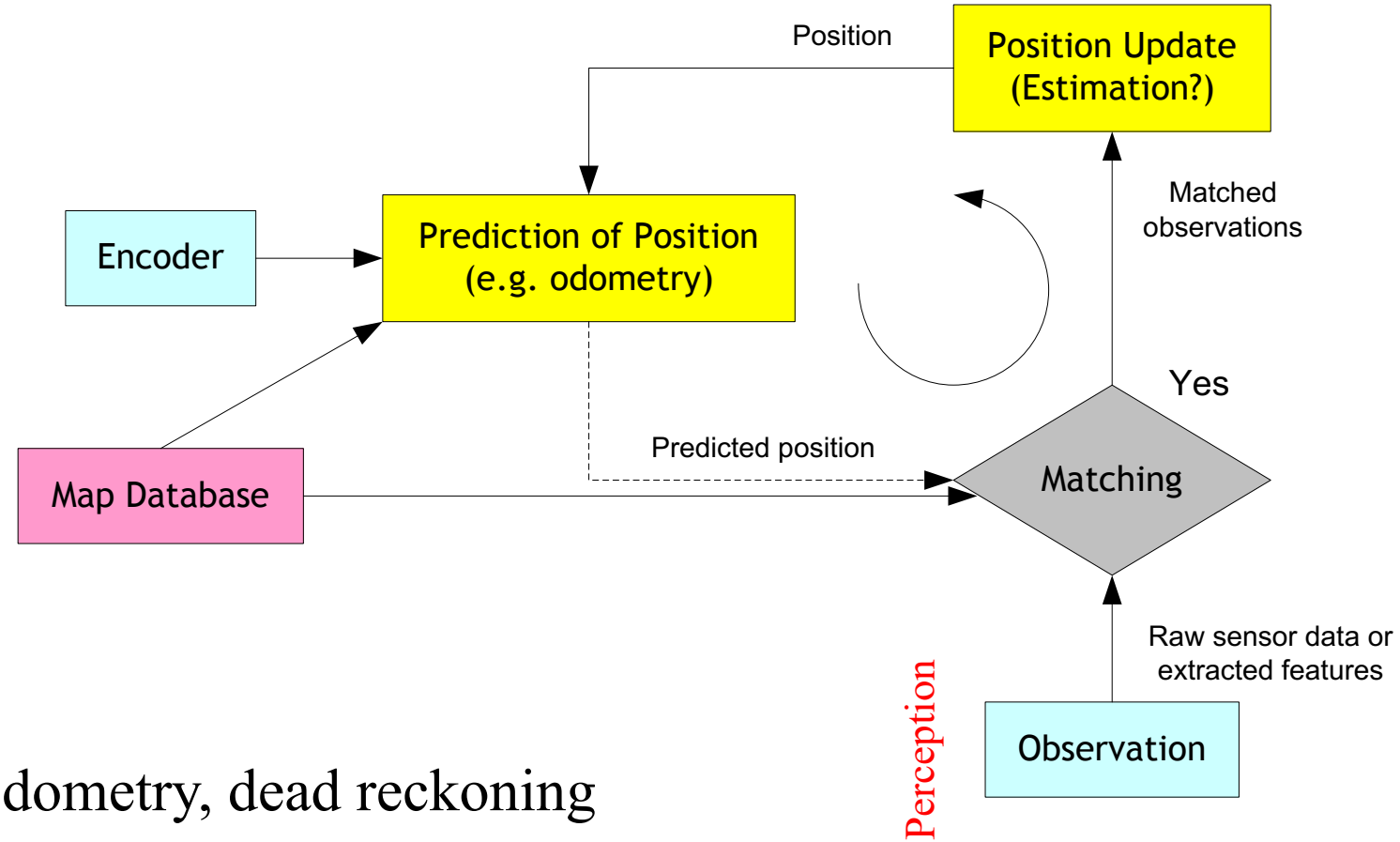
- Perception –
  - The robot must interpret its sensors to extract meaningful data.
- Localization –
  - The robot must determine its position in the environment.
- Cognition –
  - The robot must decide how to act to achieve its goal.
- Motion control –
  - The robot must modulate its motor outputs to achieve the desired trajectory.

Localization and Map Building

- Noise and aliasing
- Odometric position estimation
- To localize or not to localize
- Belief representation
- Map representation
- Probabilistic map-based localization
- Other examples of localization system
- Autonomous map building



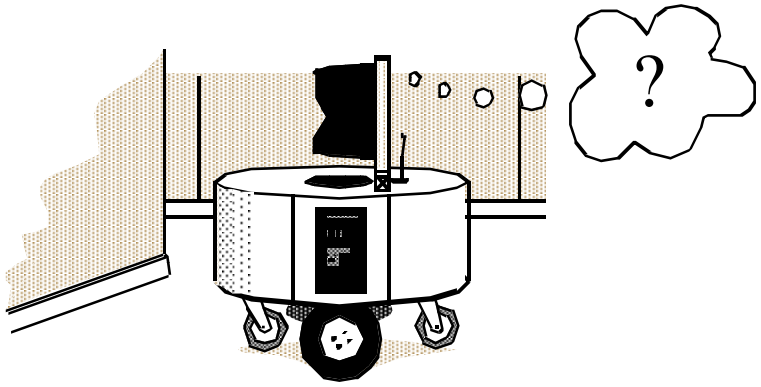
General Scheme for Robot Localization



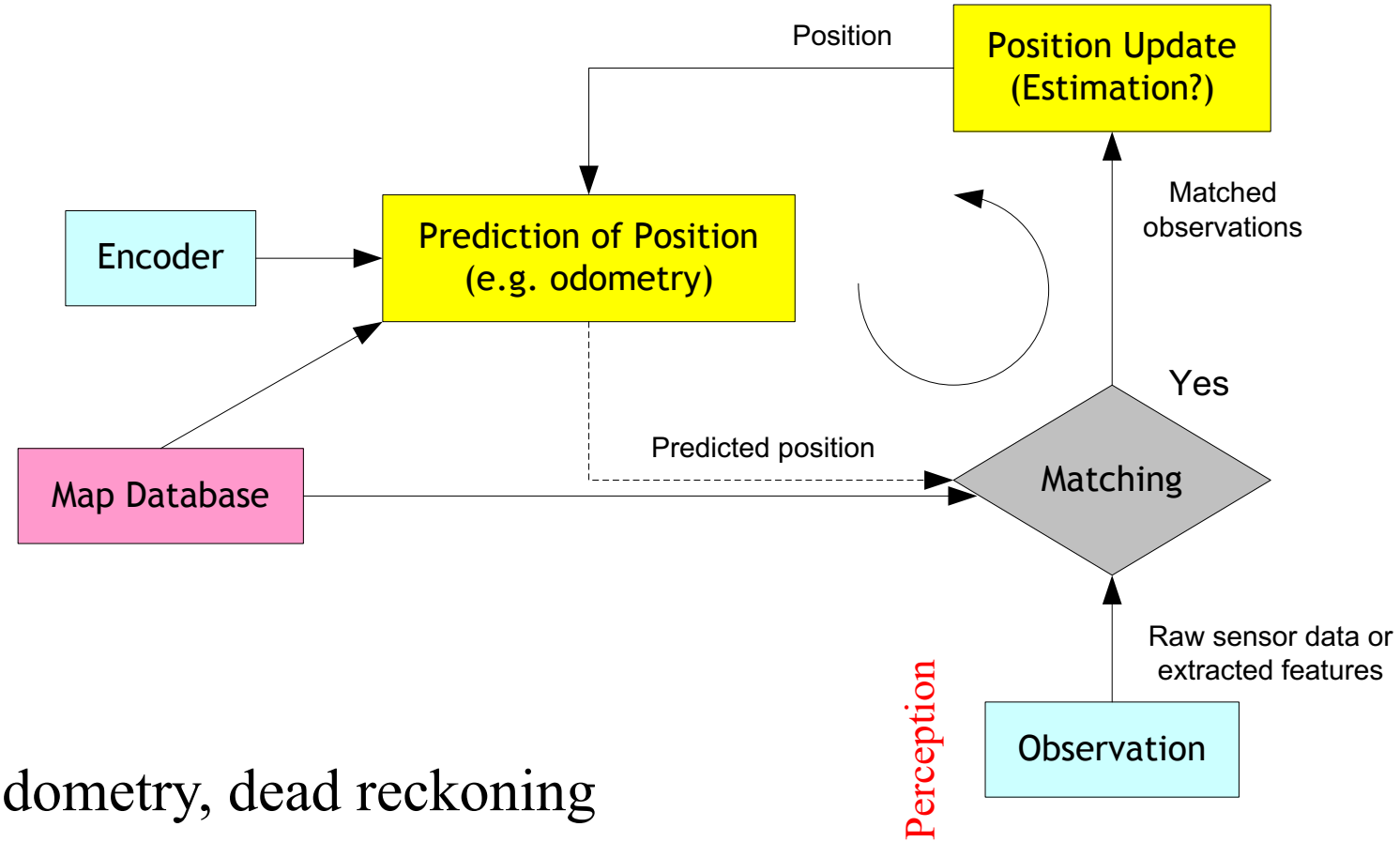
- Odometry, dead reckoning
- Probabilistic map based localization
- Localization base on external sensors, beacons or landmarks

Localization and Map Building

- Noise and aliasing
- Odometric position estimation
- To localize or not to localize
- Belief representation
- Map representation
- Probabilistic map-based localization
- Other examples of localization system
- Autonomous map building



General Scheme for Robot Localization



- Odometry, dead reckoning
- Probabilistic map based localization
- Localization base on external sensors, beacons or landmarks

Challenges of Localization

- Knowing the absolute position (e.g. GPS) is not sufficient
- Localization in human-scale in relation with environment
- Planning in the cognition step requires more than only position as input
- Perception and motion plays an important role
  - Sensor noise
  - Sensor aliasing
  - Effector noise
  - Odometric position estimation

Sensor Noise

- Sources of sensor noise
  - Influence of the environment (surface, illumination …)
  - The measurement principle itself (interference between ultrasonic sensors)
- Sensor noise drastically reduces the useful information of sensor readings
- Solutions:
  - Take multiple reading into account
  - Employ temporal and/or multi-sensor fusion

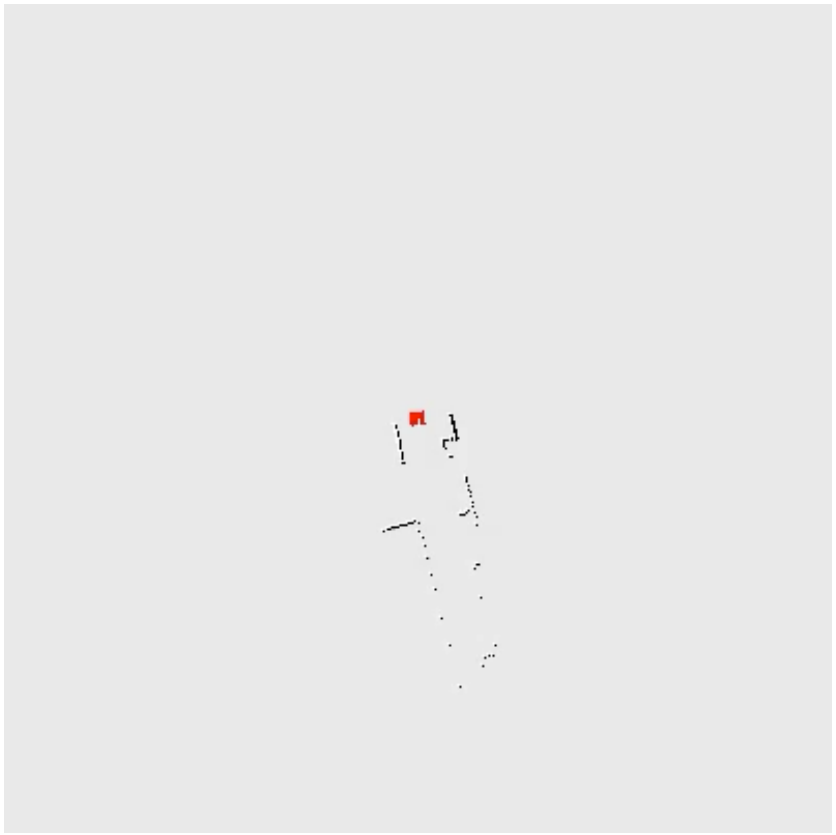
Sensor Aliasing

- In robots, non-uniqueness of sensors readings is the norm
- Even with multiple sensors, there is a many-to-one mapping from environmental states to robot's perceptual inputs
- Even with noise-free sensors, the amount of information perceived by the sensors is generally insufficient to identify the robot's position from a single reading
  - Robot's localization is usually based on a series of readings
  - Sufficient information is recovered by the robot over time

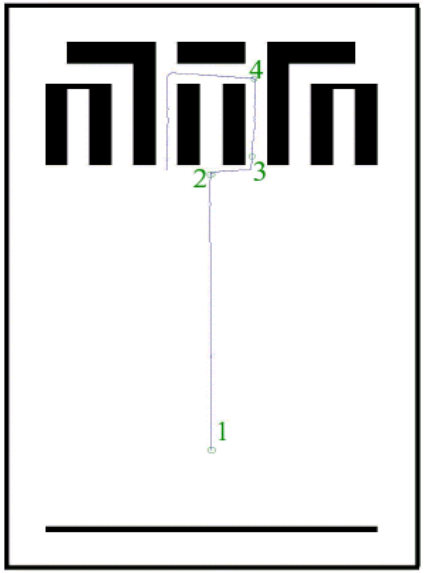
Single-Hypothesis Representation

- Advantage
  - No position ambiguity – facilitates decision-making at the robot's cognitive level (e.g., path planning)
  - Updating the robot's belief regarding position is also facilitated
- Disadvantage
  - In fact, robot motion often induces uncertainty due to effector and sensor noise – it is not possible to generate a single hypothesis of position

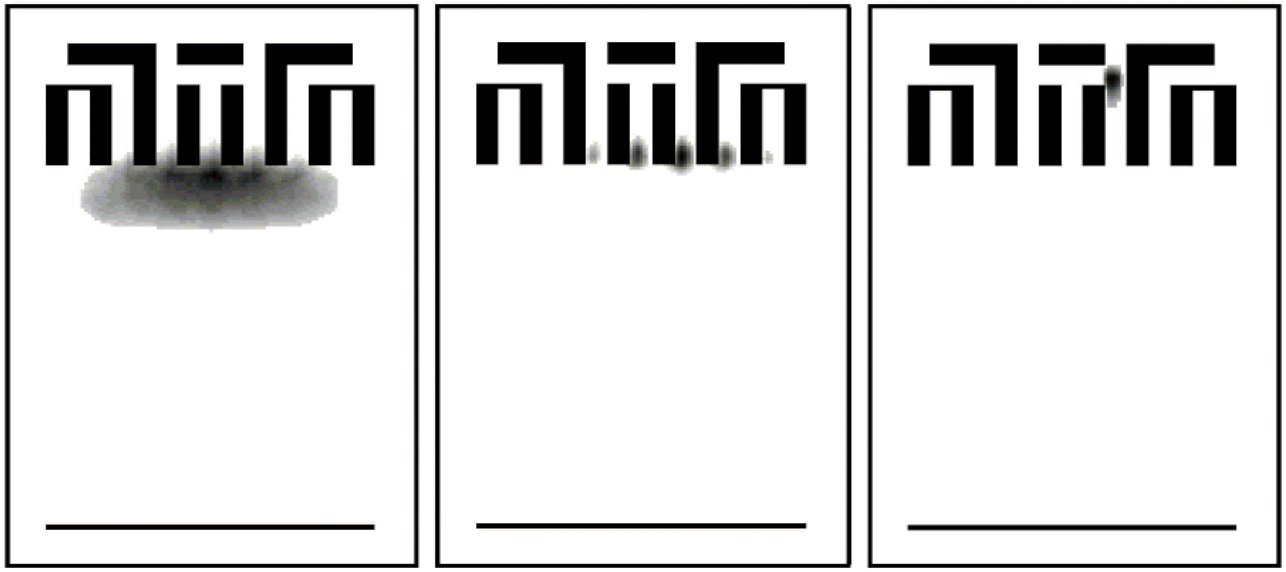
Examples



Multiple-Hypothesis Belief



Path of the robot



Belief states at positions 2, 3 and 4

Multiple-Hypothesis Representation

- Advantage
  - The robot can explicitly maintain uncertainty regarding its position
  - Partial information from sensors and effectors can be incorporated in an update belief
  - A robot may choose paths that minimize its future position uncertainty
  - Limited sensory information are enough in some cases
- Disadvantage
  - Decision-making – too many location possibilities!!!
  - Motion trajectory might not be consistent for different possible positions
  - Computational expensive (high-dimensional)

Map Representation

- Representation of the environment ↔ representation of the robot position
- Fidelity of the position presentation ↔ fidelity of the map
- Fundamental relationships for choosing a map representation
  - Map precision vs. application
  - Features precision vs. map precision
  - Precision vs. computational complexity
- Two different types of representation
  - Continuous representations
  - Decomposition (Discretization)

Representation of the Environment

- Environment Representation
  - Continuous Metric → x, y, θ
  - Discrete Metric → metric grid
  - Discrete Topological → topological grid
- Environment Modeling
  - Raw sensor data, e.g. laser range data, grayscale images
    - large volume of data, low distinctiveness on the level of individual values
    - makes use of all acquired information
  - Low level features, e.g. lines, other geometric features
    - medium volume of data, average distinctiveness
    - filters out the useful information, still ambiguities
  - High level features, e.g. doors, a car, the Eiffel tower
    - low volume of data, high distinctiveness
    - filters out the useful information, few/no ambiguities, not enough information

Continuous Representation

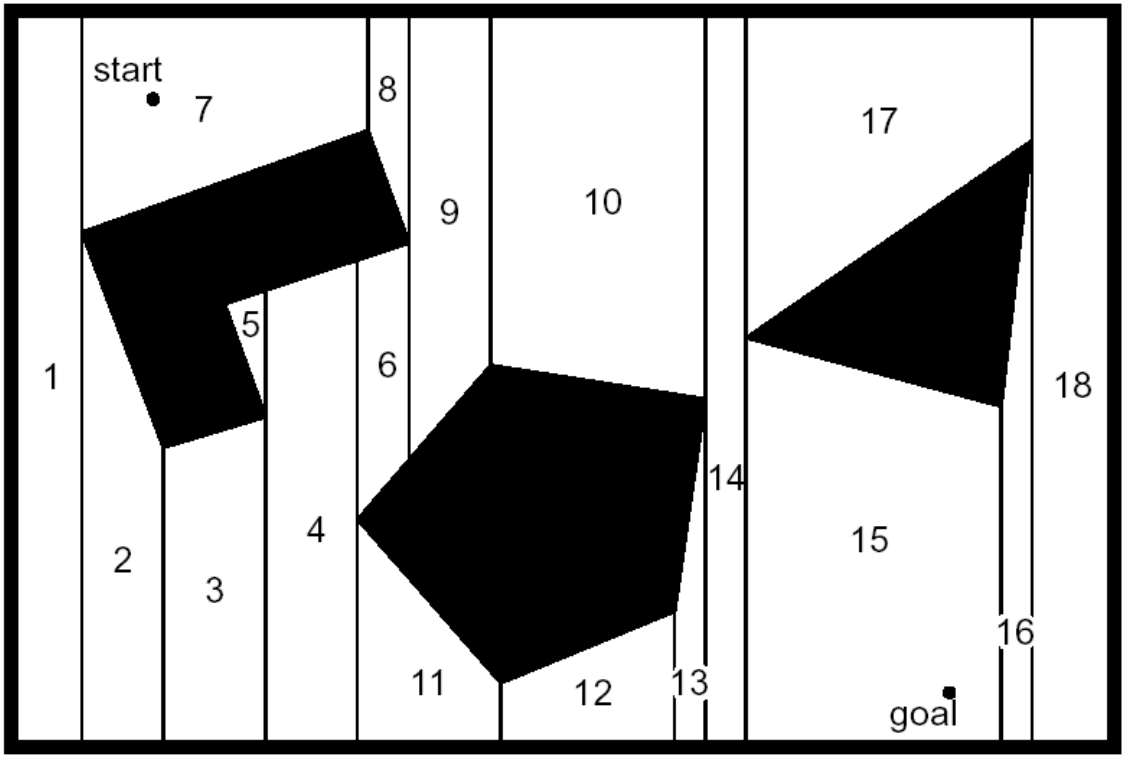
- A continuous-valued map is one method for exact decomposition of the environment
- To combine the exactness of a continuous representation with the compactness of the closed-world assumption
- Total storage needed in the map is proportional to the density of objects in the environment
- A sparse environment can be represented by a low-memory map

Decomposition Strategies: Abstraction

- A general decomposition and selection of environmental features
- Disadvantage
  - Loss of fidelity between the map and the real world
  - A highly abstract map does not compare favorably to a high-fidelity map
- Advantage
  - The map representation can be potentially minimized
  - Planning is computationally efficient for hierarchical representation

Exact Cell Decomposition

- Decomposition by selecting boundaries between discrete cells based on geometric criticality

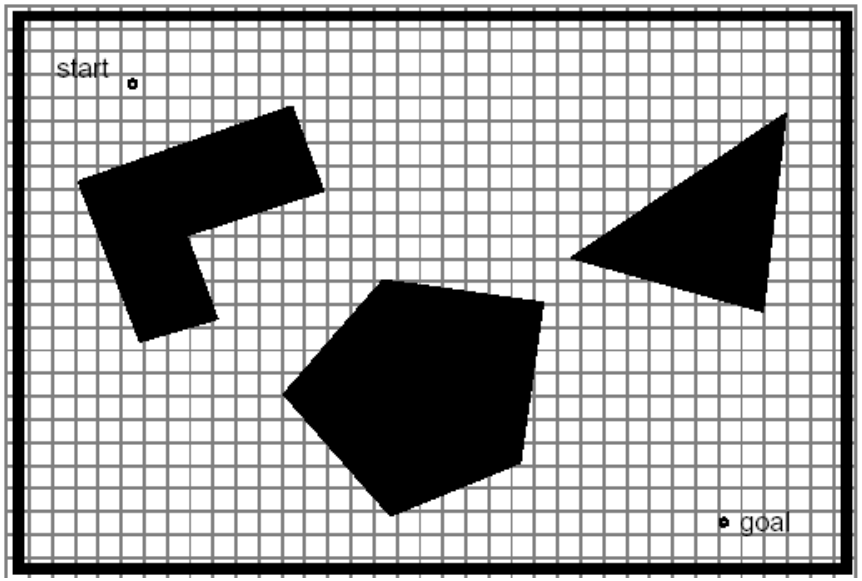


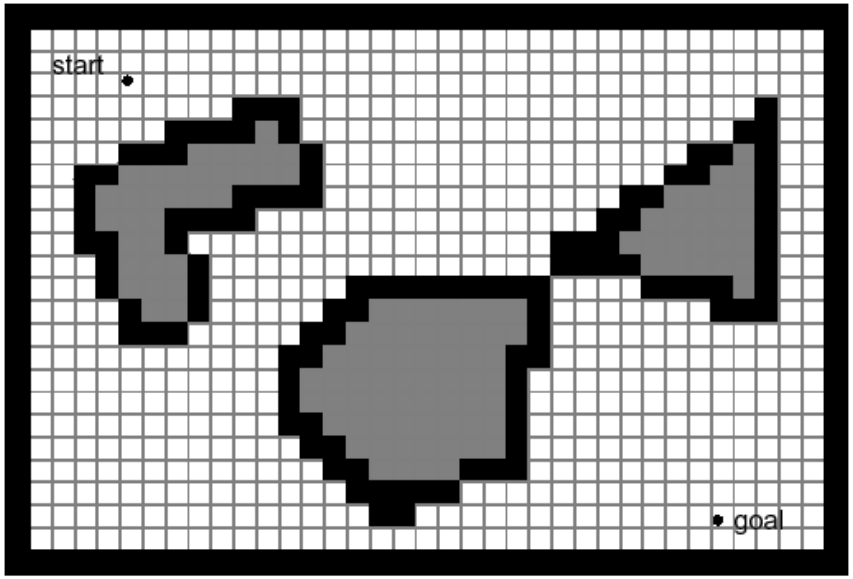
Exact Cell Decomposition

- We do not care about the particular position of a robot within each area of free space
- We care only the robot's ability to traverse from each area of free space to the adjacent areas
- Problems
  - Exact decomposition is a function of the particular environment obstacles and free space
  - It is not feasible if this information is expensive to collect or unknown

Fixed Cell Decomposition

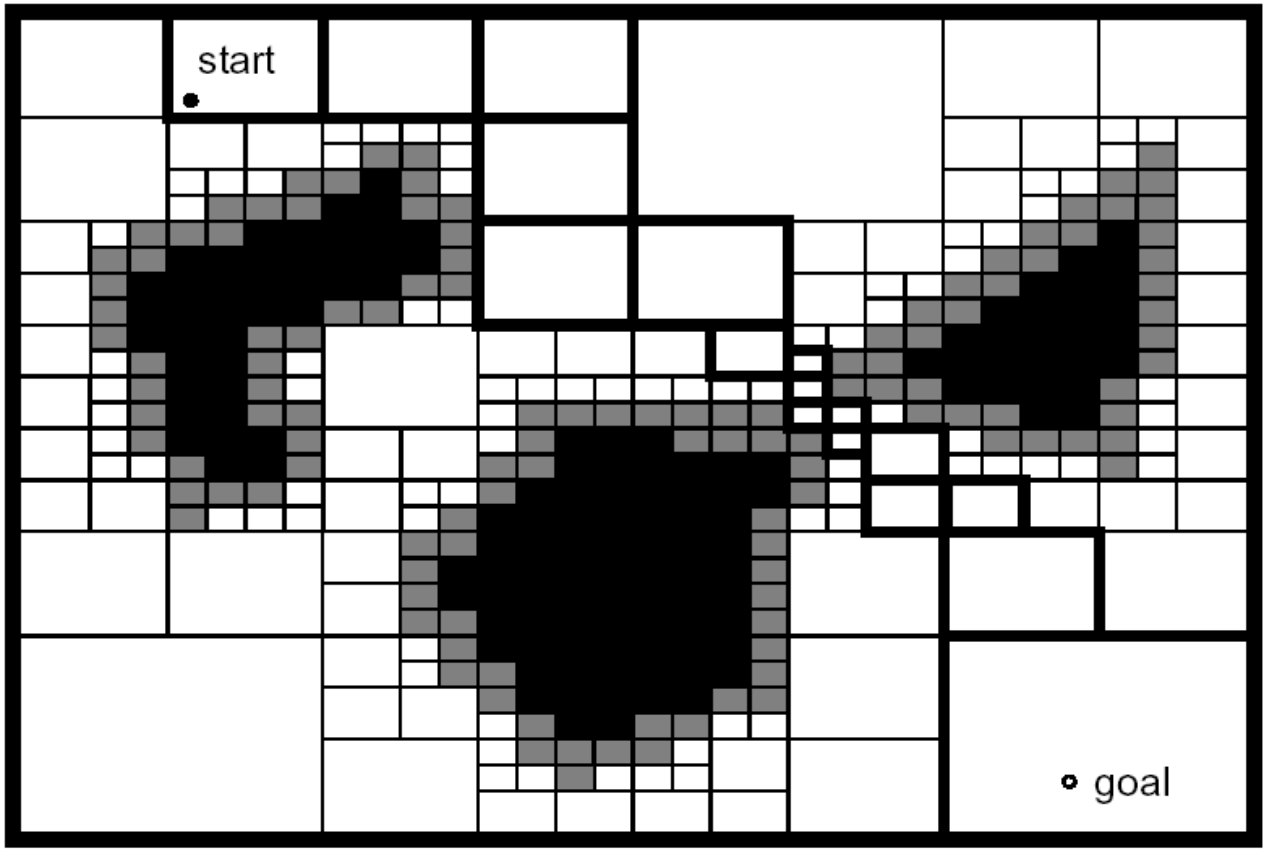
- Transform the continuous real environment into a discrete approximation for the map
- Disadvantage
  - Inexact: e.g., narrow passages disappear





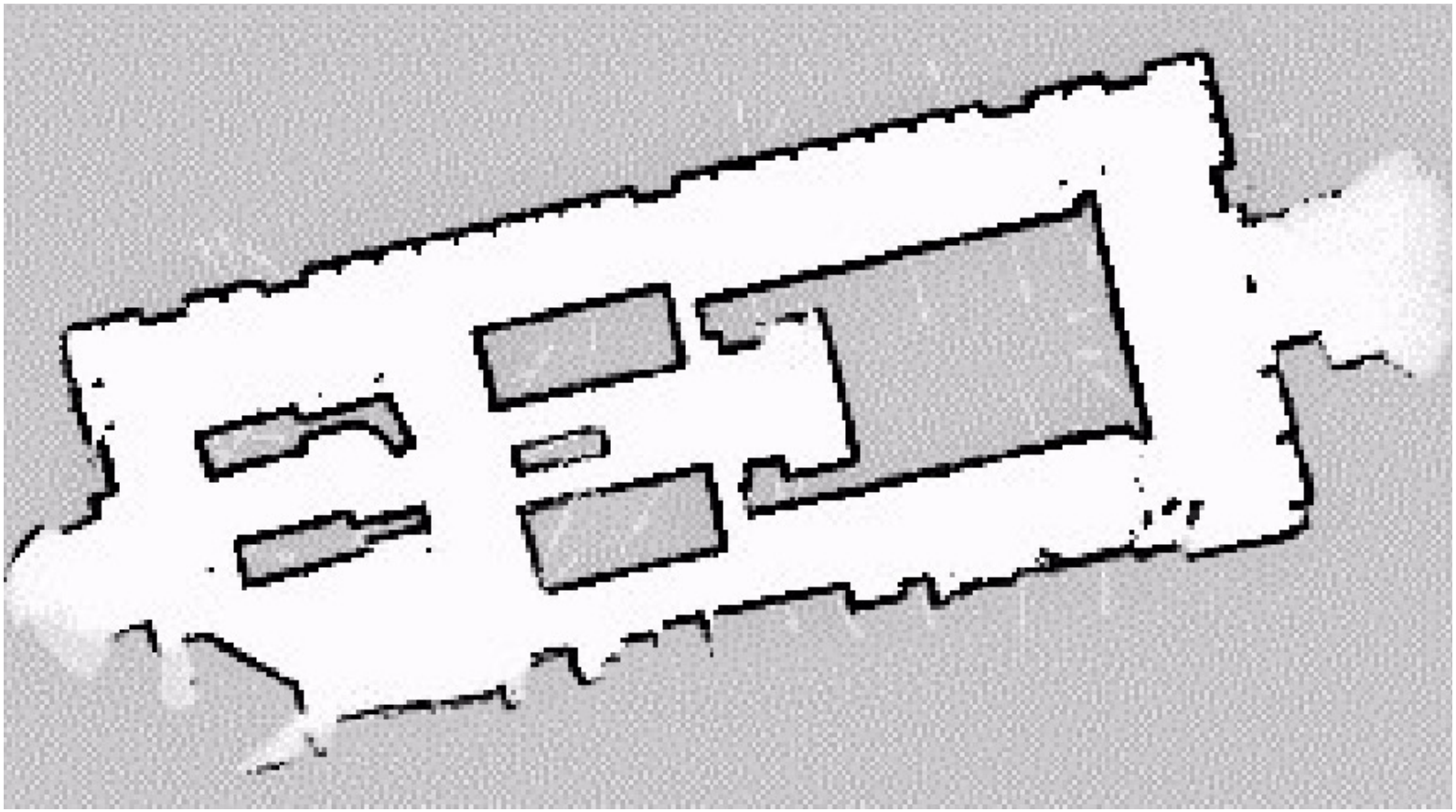
Adaptive Cell Decomposition

Adaptive cell decomposition



Occupancy Grid Representation

- Fixed cell decomposition – Example with very small cells

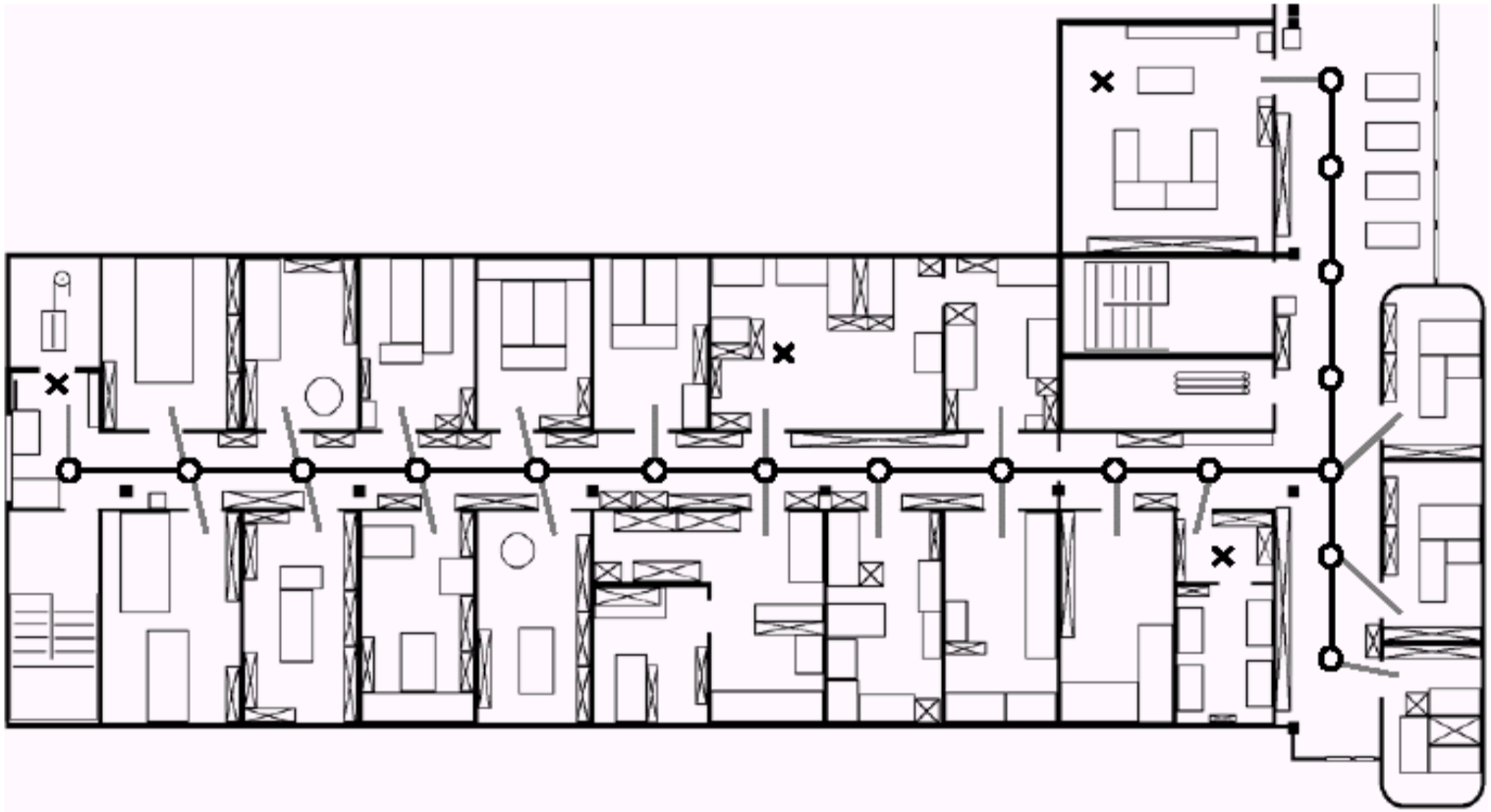


Occupancy Grid Representation

- Disadvantages
  - Small cell size implies large memory requirement
  - Not compatible with the close-world assumption
  - Might record unnecessary details (pre-allocate the whole space)

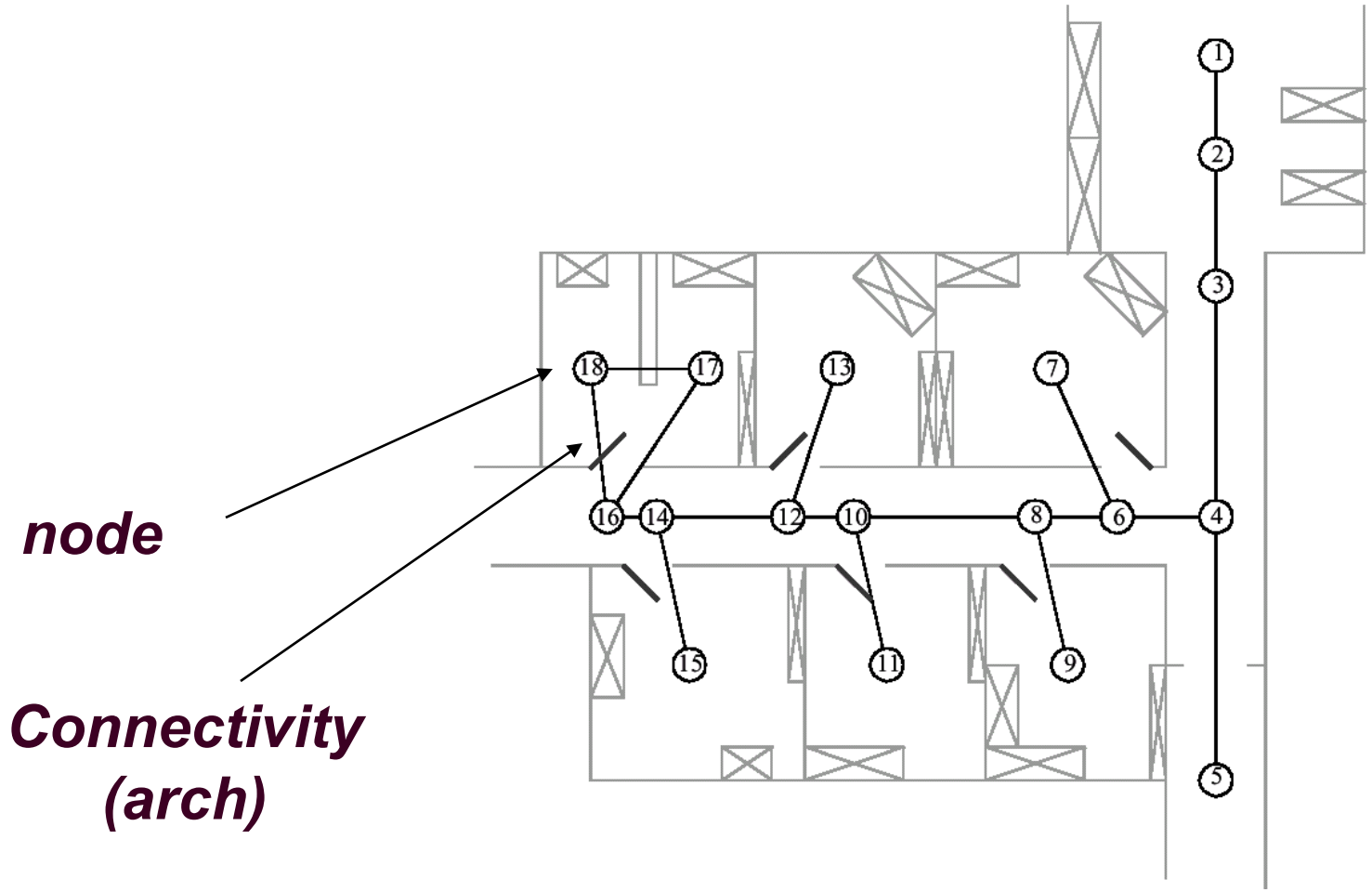
Topological Decomposition

- Avoid direct measurement of geometric quantities
- Concentrate on characteristics of the environment



Topological Decomposition

- Adjacency is the heart of the topological approach

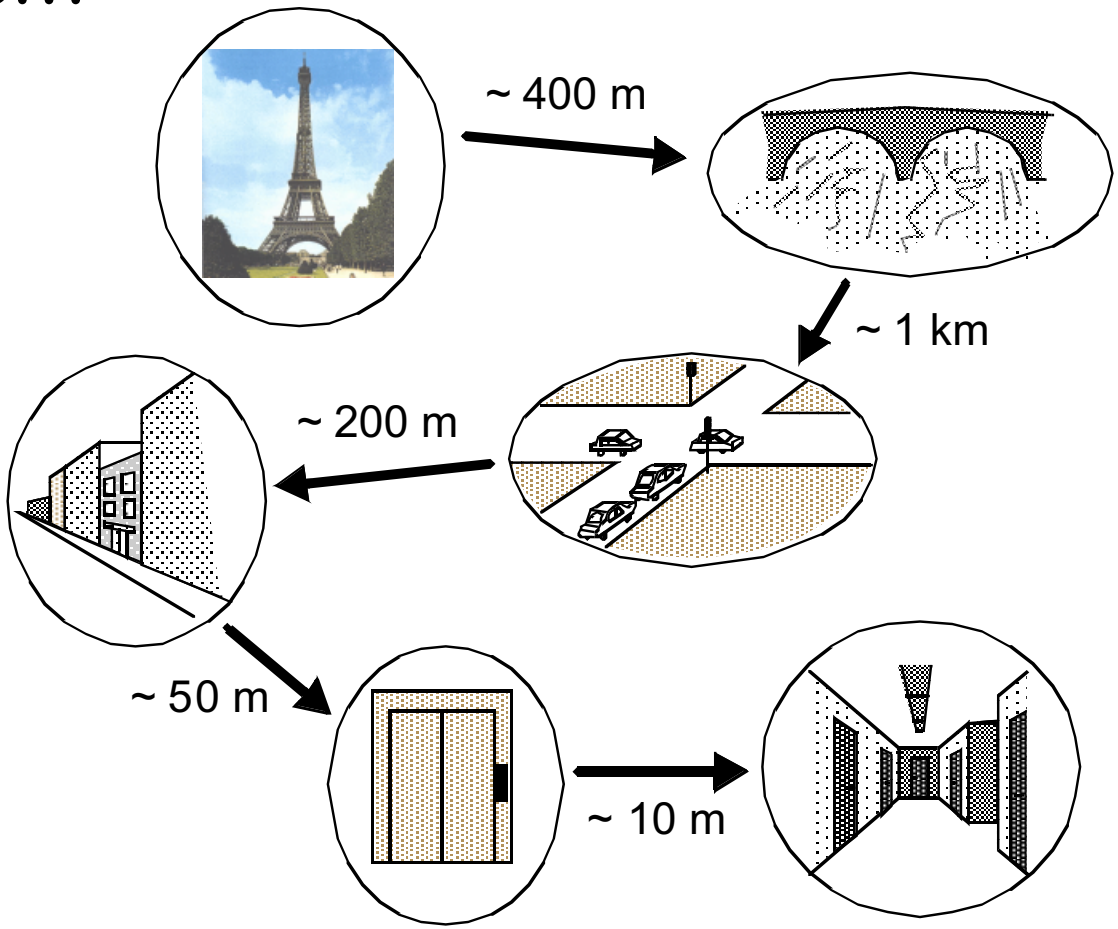


Topological Decomposition

- To navigate using a topological map robustly, a robot must be able to
  - Detect its current position in terms of the nodes of the topological graph
  - Travel between nodes using robot motion
- The node size and particular dimensions must be optimized to match the sensory discrimination of the mobile robot hardware

Topological Decomposition

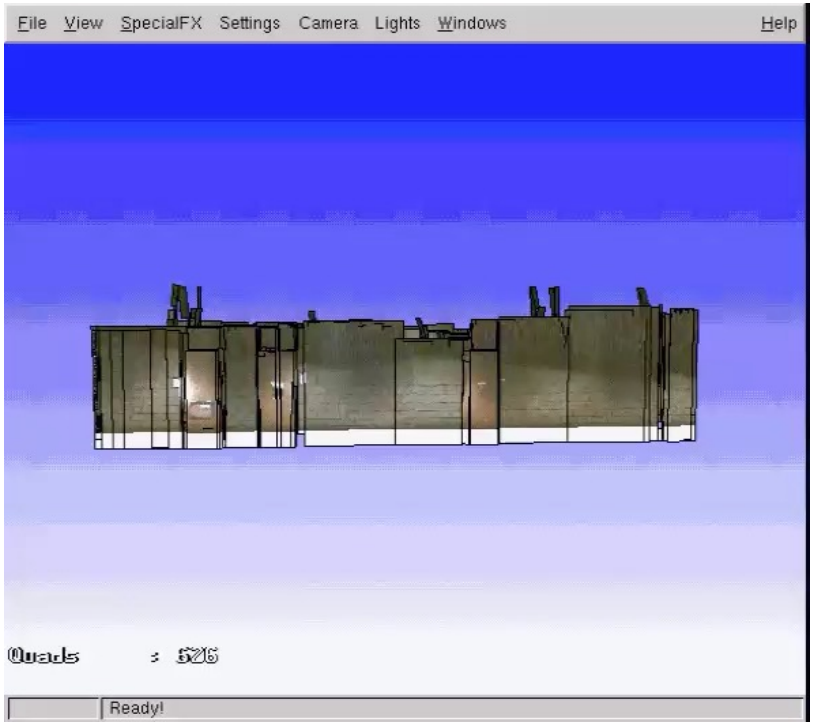
- The environment may contain important non-geometric features…



Summary

- Range is not the only measurable and useful environmental value for a mobile robot
- How to choose a map representation?
  - Understanding the sensors available on the mobile robot
  - Understanding the mobile robot's functional requirements (e.g., required goal precision and accuracy)

Examples


File View SpecialFX Settings Camera Lights Windows Help
Quads : 526
Ready!

Current Challenges in Map Representation

- Real world is dynamic
  - Distinguish between permanent and transient obstacles
  - Vision may or should be used
- Perception is still a major challenge
  - Error prone
  - Extraction of useful information difficult
- Localization involves the traversal of open space
- How to build up topology (boundaries of nodes)
  - Assumptions regarding spatial locality
- Sensor fusion – neural network classifier

Probabilistic Map-Based Localization

- There are two classes of probabilistic localization
  - Markov localization – uses explicitly specified probabilistic distribution across all possible robot positions
  - Kalman filter localization – uses a Gaussian probability density representation of robot position and scan matching for localization
- Unlike Markov localization, Kalman filter localization does not independently consider each possible pose in the robot's configuration space

Probabilistic Map-Based Localization

- Consider a mobile robot moving in a known environment
- As it start to move, say from a precisely known location, it might keep track of its location using odometry
- However, after a certain movement the robot will get very uncertain about its position
- Update using an observation of its environment
- Observation leads also to an estimate of the robot's position which can then be fused with the odometric estimation to get the best possible update of the robot's actual position

Probabilistic Map-Based Localization

- The information provided by the robot's odometry, plus the information provided by exteroceptive observation, can be combined to enable localization
- The processes of updating based on proprioceptive sensor values and exteroceptive sensor values are often separated logically, leading to a general two-step process for robot position update

Probabilistic Map-Based Localization

- Action update
  - Action model *Act*

$$s'_t = Act(o_t, s_{t-1})$$

with o_t : encoder measurement, s_{t-1} : prior belief state

  - *Increases uncertainty*
- Perception update
  - Perception model *See*

$$s_t = See(i_t, s'_t)$$

with i_t : exteroceptive sensor inputs, s'_t : updated belief state

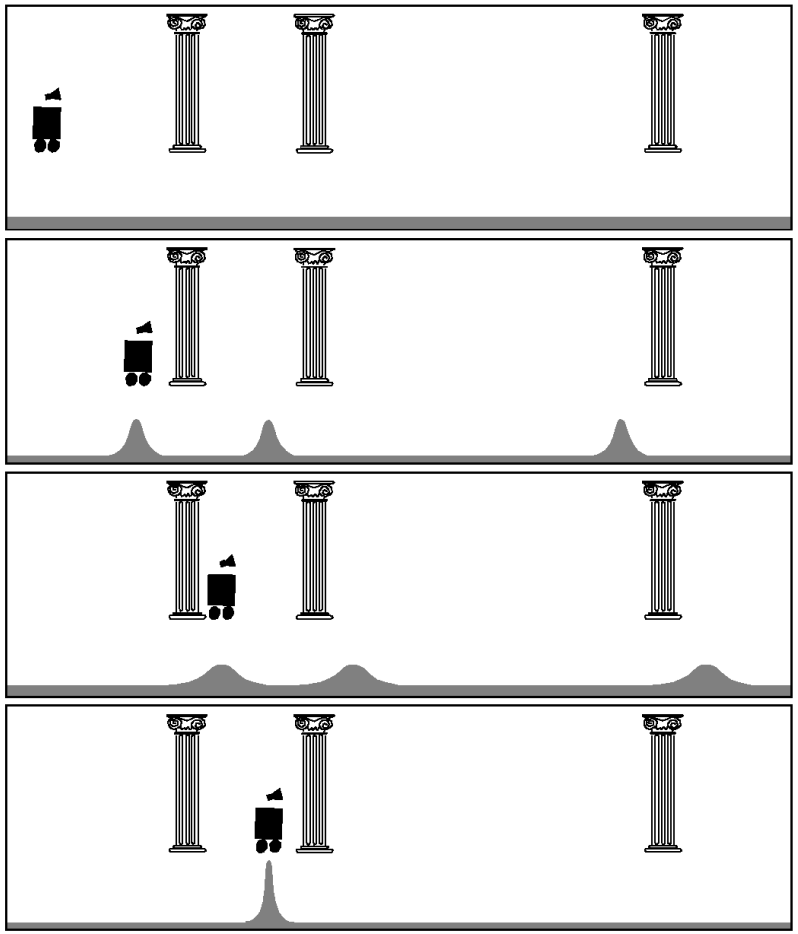
  - *Decreases uncertainty*

Probabilistic Map-Based Localization

- The *action update process* contributes uncertainty to the robot's belief about position
- *Perception update* generally refines the belief state
- Markov localization
  - The robot's belief state is represented as *separate probability assignments* for every possible robot pose
  - The action and perception update processes update the probability of every cell
- Kalman filter localization
  - The robot's belief state is represented by a *single well-defined Gaussian pdf* – only the updates of μ and σ are required

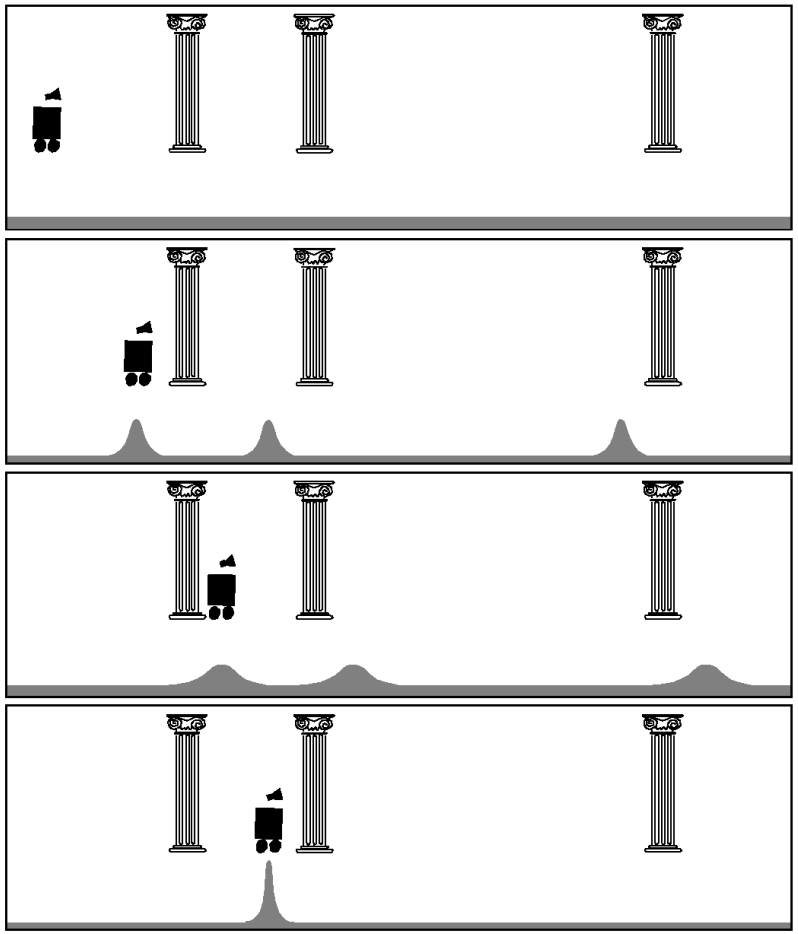
Probabilistic Map-Based Localization

- Improving belief state by moving
- Assuming 1-D



Probabilistic Map-Based Localization

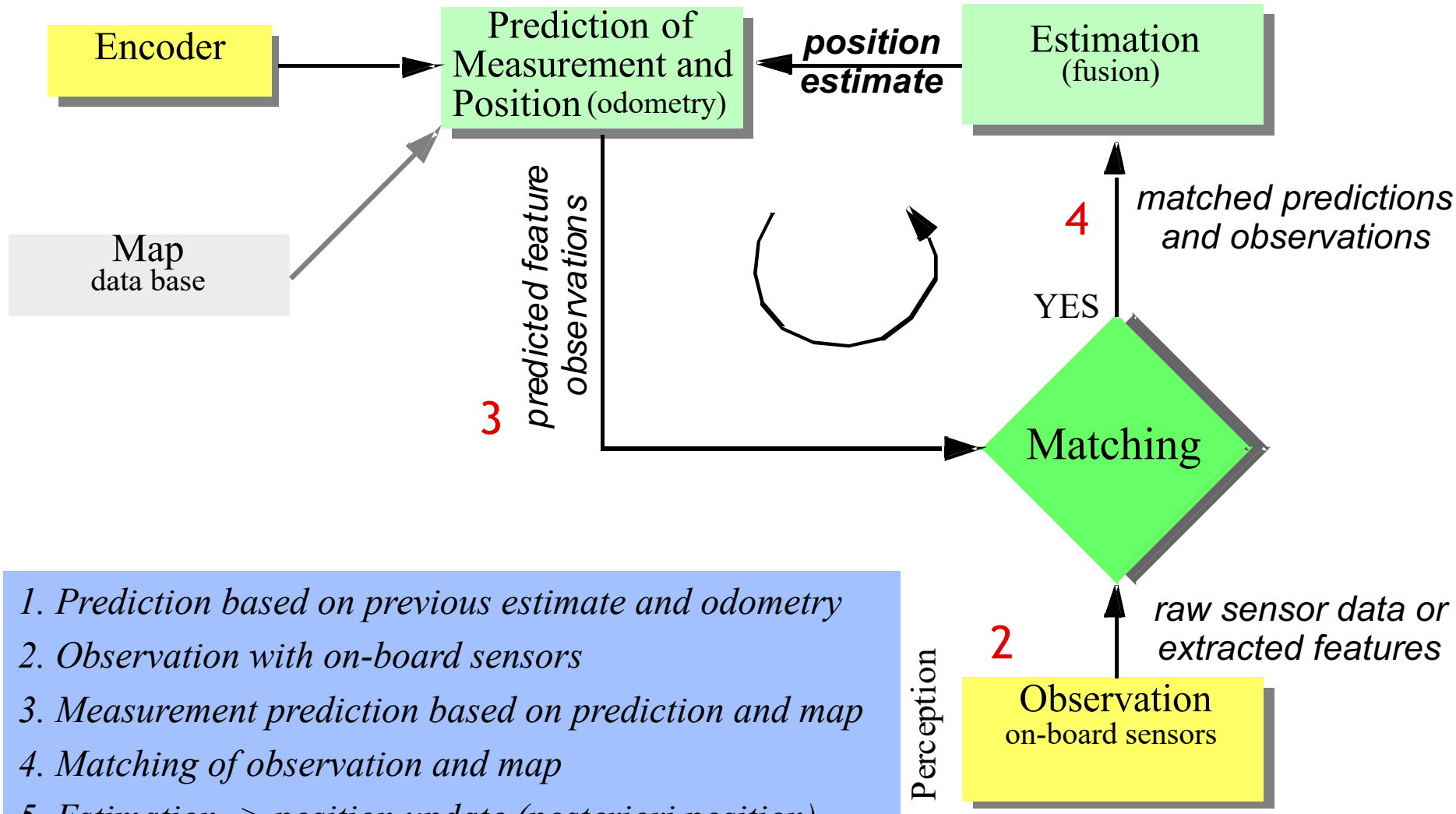
- Improving belief state by moving
- Assuming 1-D



Probabilistic Map-Based Localization

- Given
  - The position estimate $p(k|k)$
  - Its covariance $\Sigma_p(k|k)$ for time k
  - The current control input $u(k)$
  - The current set of observations $z(k+1)$
  - The map $M(k)$
- Compute
  - The new (posteriori) position estimate $p(k+1|k+1)$
  - Its covariance $\Sigma_p(k+1|k+1)$
- Such a procedure usually involves five steps

Five Steps for Map-Based Localization



1. *Prediction based on previous estimate and odometry*
2. *Observation with on-board sensors*
3. *Measurement prediction based on prediction and map*
4. *Matching of observation and map*
5. *Estimation -> position update (posteriori position)*

Markov ⇔ Kalman Filter Localization

- Markov localization
  - Localization starting from any unknown position
  - Recovers from ambiguous situation
  - To update the probability of all positions within the whole state space at any time requires a discrete representation of the space (grid)
  - The required memory and calculation power can thus become very important if a fine grid is used
- Kalman filter localization
  - Tracks the robot and is inherently very precise and efficient
  - If the uncertainty of the robot becomes too large (e.g. collision with an object) the Kalman filter will fail and the position is definitively lost

Markov Localization

- Markov localization uses an explicit, discrete representation for the probability of all positions in the state space
- This is usually done by representing the environment by a *grid* or a *topological graph* with a finite number of possible states (positions)
- During each update, the probability for *each state* (element) of the entire space is updated

Markov Localization

- $P(A)$: Probability that A is true
  - e.g. $p(r_t = l)$: the probability that the robot r is at position l at time t
- We wish to compute the probability of each individual robot position given actions and sensor measurements
- $P(A \mid B)$: Conditional probability of A given that we know B
  - e.g. $p(r_t = l \mid i_t)$: probability that the robot is at position l given the sensors input i_t
- Product rule: $p(A \wedge B) = p(A|B)p(B)$

  $p(A \wedge B) = p(B|A)p(A)$
- *Bayes rule*: $p(A|B) = \dfrac{p(B|A)p(A)}{p(B)}$

Markov Localization

- Bayes rule:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

- Map from a belief state and a sensor input to a refined belief state (*See*):

$$p(l|i) = \frac{p(i|l)p(l)}{p(i)}$$

- $p(l)$: belief state before perceptual update process
- $p(i \mid l)$: probability to get measurement i when being at position l
- Consult robots map, identify the probability of a certain sensor reading for each possible position in the map
- $p(i)$: normalization factor so that sum over all l equals 1

Markov Localization

- Bayes rule:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

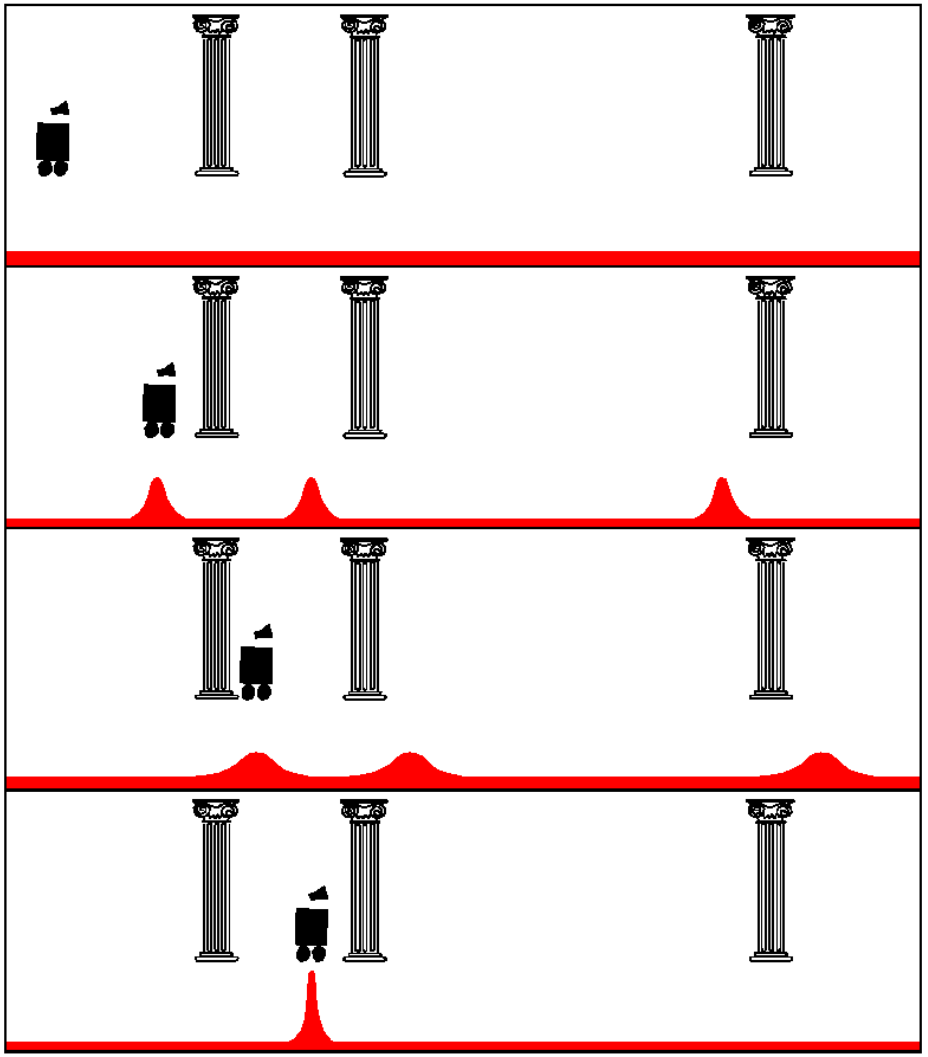
  - Map from a belief state and an action to new belief state (*Act*):

$$p(l_t|o_t) = \int p(l_t|l'_{t-1}, o_t)p(l'_{t-1})dl'_{t-1}$$

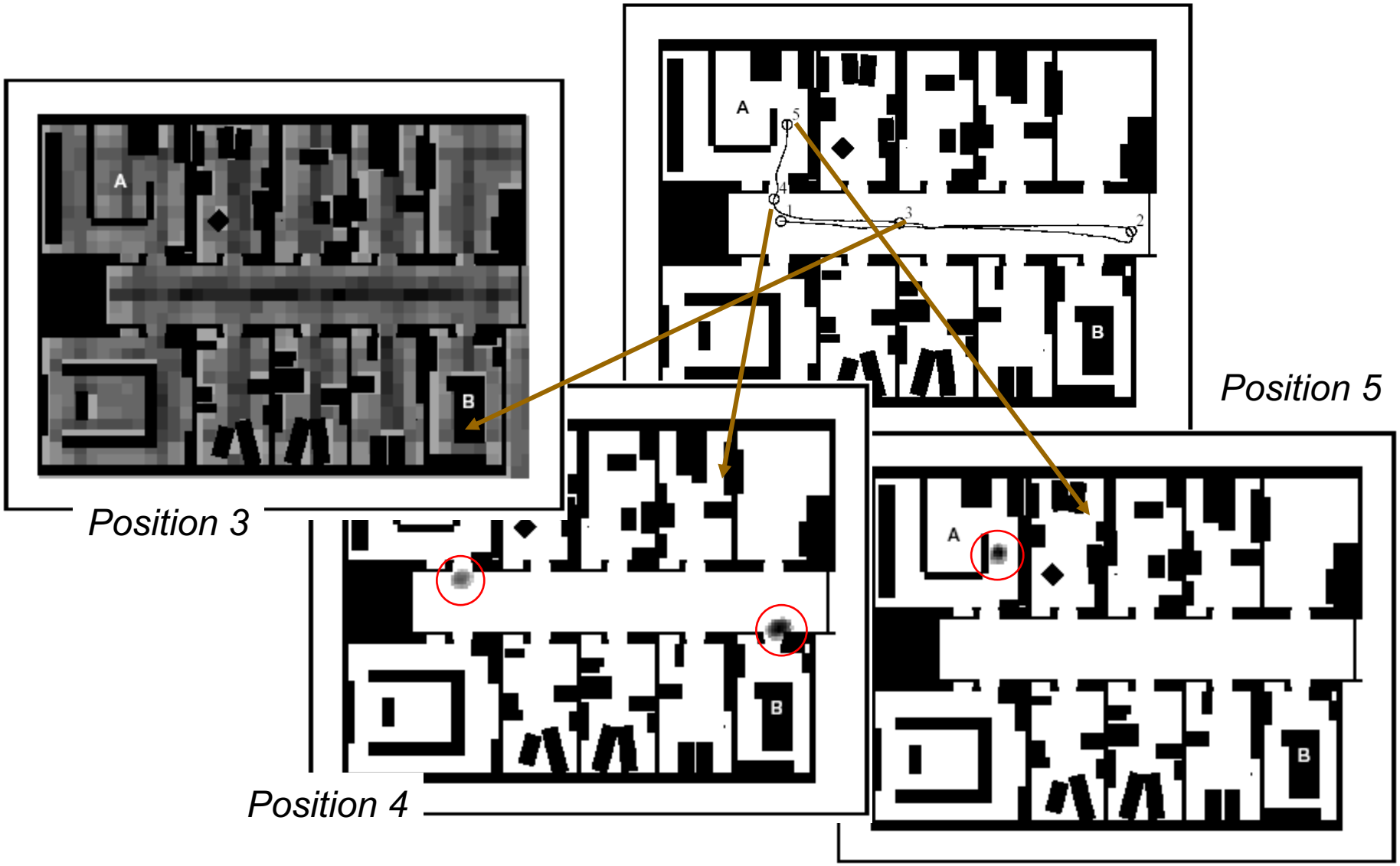
  - *Summing over all possible ways* in which the robot may have reached *l*
- Markov assumption:
  - Update only depends on *previous state* and its most recent actions and perception

Case Study – Grip Map (1D Case)

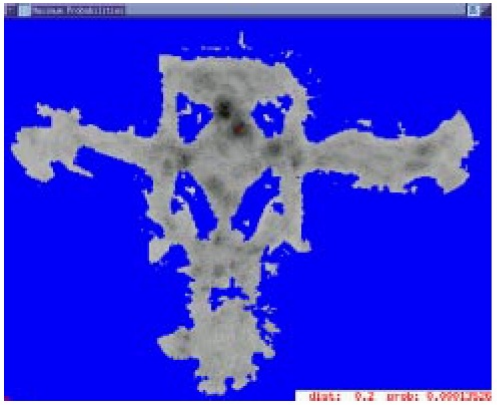
- Start
  - No knowledge at start, thus we have an uniform probability distribution.
- Robot perceives first pillar
  - Seeing only one pillar, the probability being at pillar 1, 2 or 3 is equal.
- Robot moves
  - Action model enables to estimate the new probability distribution based on the previous one and the motion.
- Robot perceives second pillar
  - Base on all prior knowledge the probability being at pillar 2 becomes dominant

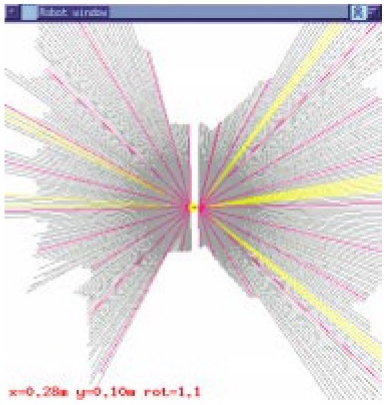


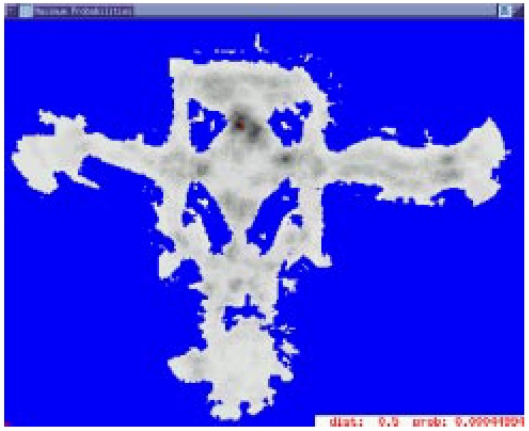
Example – Office Building

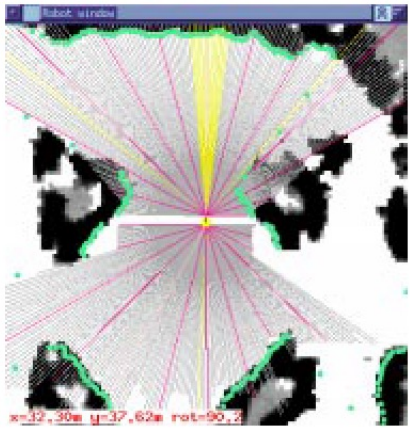


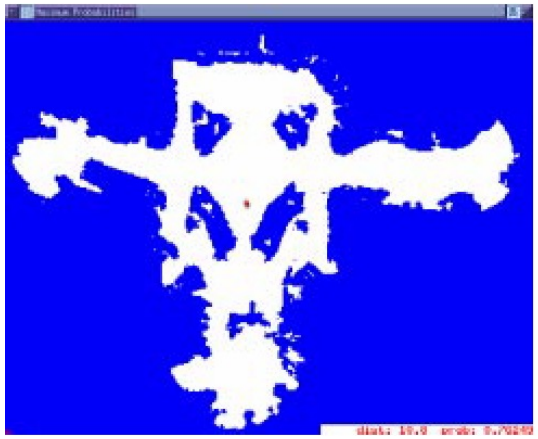
Example – Museum

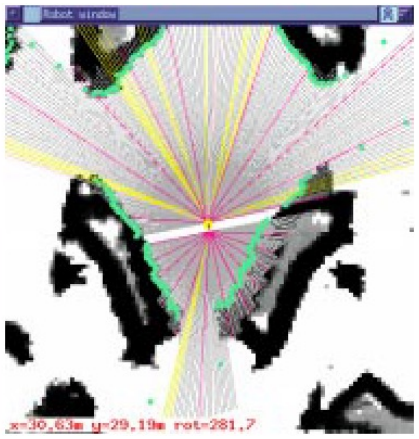










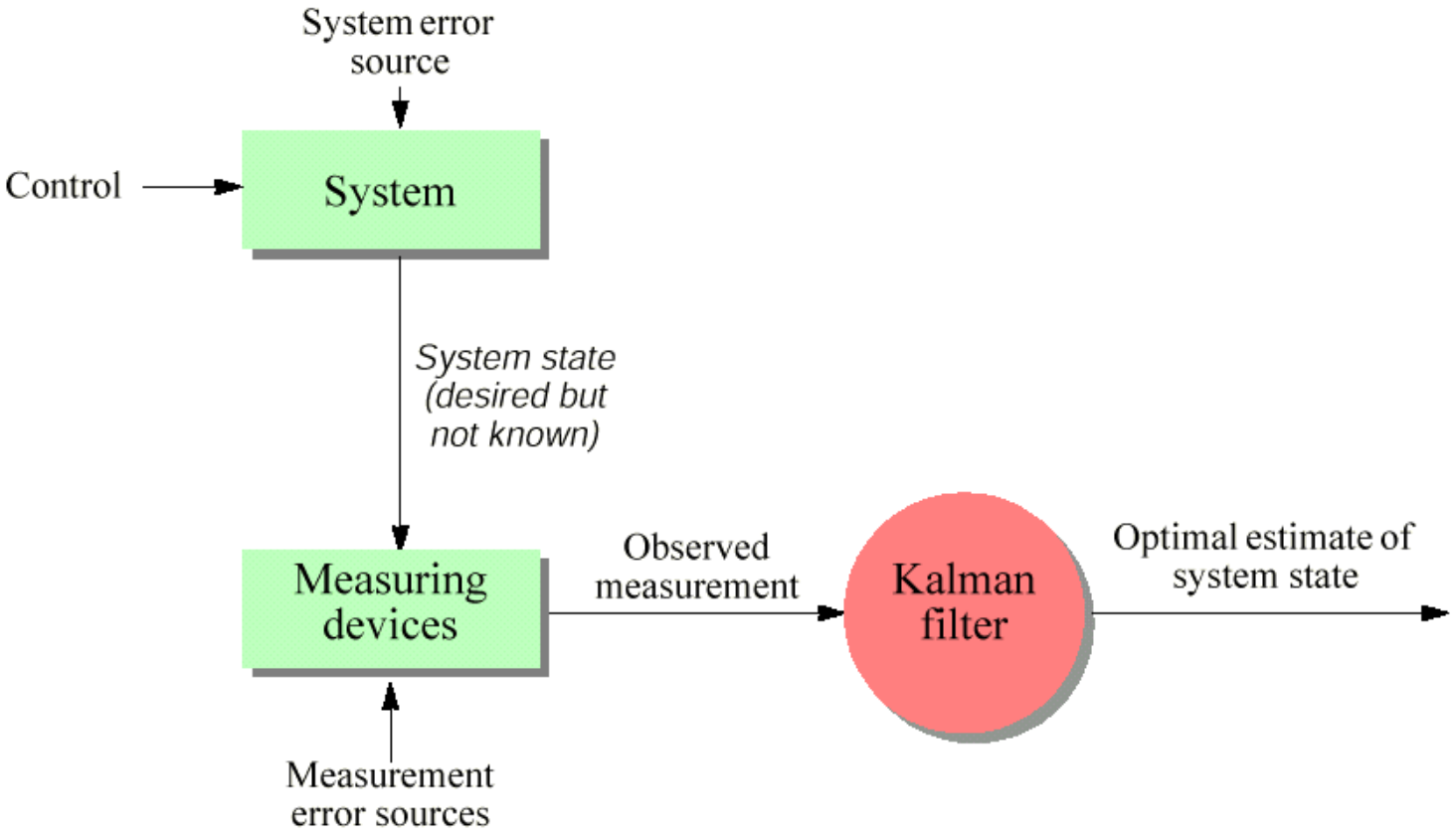


Kalman Filter Localization

- Robots usually include a large number of heterogeneous sensors, each providing clues as to robot position and each suffering from its own failure modes
- Optimal localization should take into account the information provided by all of these sensors
- The Kalman filter is a mathematical mechanism for producing an optimal estimate of the system state based on the knowledge of
  - The system and the measuring device
  - The description of the system noise and measurement errors
  - The uncertainty in the dynamic models
- Thus, the Kalman filter fuses sensor signals and system knowledge in an optimal way

Kalman Filter Localization

- The system is assumed to be *linear* with *white Gaussian noise*



Kalman Filter Theory

- The basic Kalman filter method allows multiple measurements to be incorporated optimally into a single estimate of state
- The *inputs* of Kalman filter are means and variances of Gaussian probability density curve
- The fused estimate of robot position is *again a Gaussian distribution*
- Kalman filter provides a compact and simplified representation of *uncertainty*
- It is efficient for combining heterogeneous estimates to yield a new estimate of robot position

Kalman Filter – Static Estimation

- Suppose a robot with two different sensors, say an ultrasonic range sensor and a laser rangefinder
- Is it possible to combine two sensor sources (*sensor fusion*) and get better information?
- Kalman filter enable such fusion efficiently under one assumption:
  - The error characteristics of these sensors are *uni-modal, zero-mean Gaussian noise*
- Suppose the robot remain static, and we take two measurements
  - One with sonar at time k
  - One with laser rangefinder at time k+1

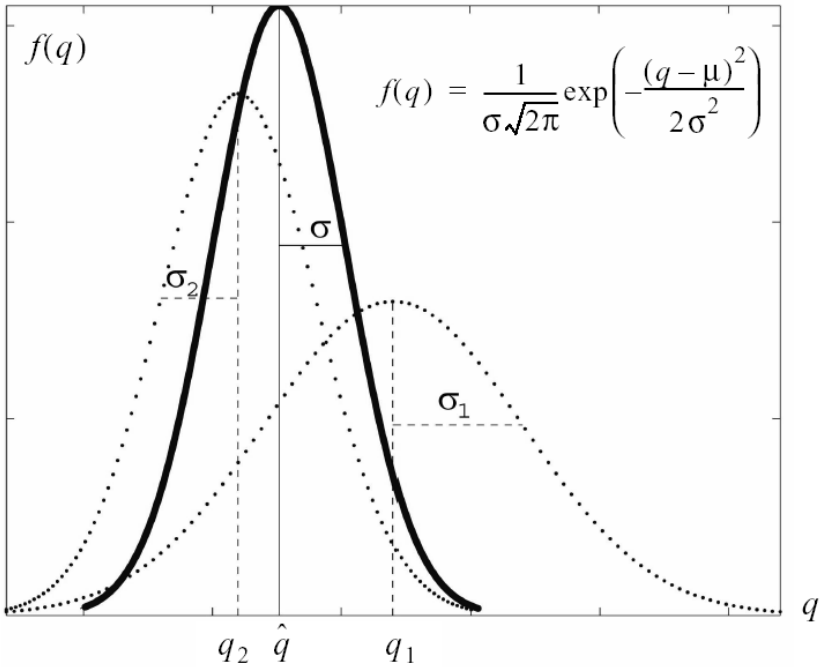
Kalman Filter – Static Estimation

- Suppose the two measurements are

$$\hat{q}_1 = q_1 \text{ with variance } \sigma_1^2$$

$$\hat{q}_2 = q_2 \text{ with variance } \sigma_2^2$$

- How to get best estimate $\hat{q}$ by fusing the above data?



Kalman Filter – Static Estimation

- Since there is no robot motion, weighted least-squares technique can be applied directly:

$$S = \sum_{i=1}^{n} w_i (\hat{q} - q_i)^2$$

- To find for the minimum error:

$$\frac{\partial S}{\partial \hat{q}} = \frac{\partial}{\partial \hat{q}} \sum_{i=1}^{n} w_i (\hat{q} - q_i)^2 = 2 \sum_{i=1}^{n} w_i (\hat{q} - q_i) = 0$$

- Thus,

$$\hat{q} = \frac{\sum_{i=1}^{n} w_i q_i}{\sum_{i=1}^{n} w_i}$$

Kalman Filter – Static Estimation

- If we take the weight $w_i = \frac{1}{\sigma_i}$
- Then $\hat{q} = q_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}(q_2 - q_1), \quad \sigma^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}$
- The resulting variance is less than all the variances of the individual measurements (*see the previous figure*)
- The uncertainty of the position estimate is *decreased* by combining the two measurements

Kalman Filter – Static Estimation

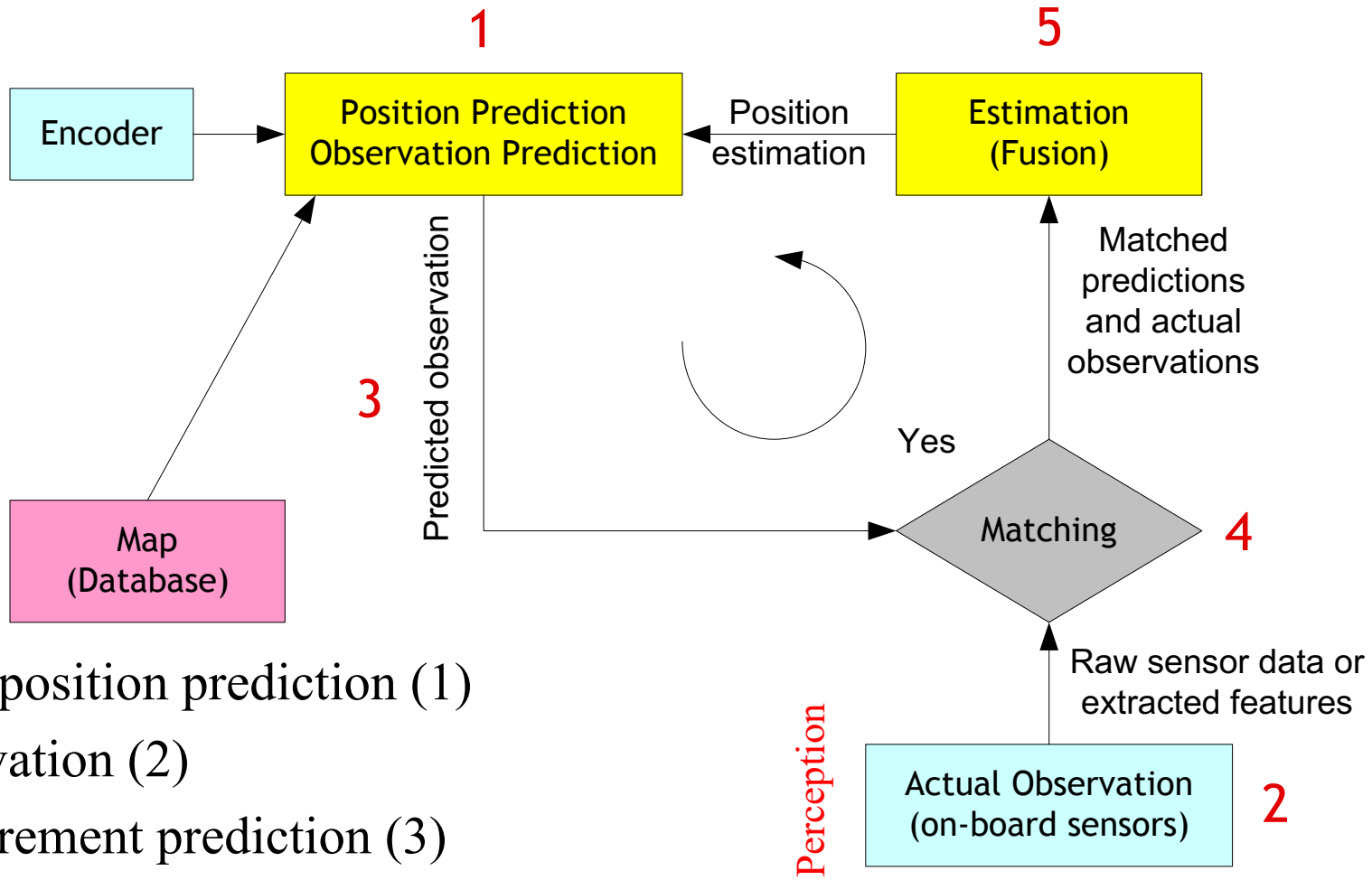
- In Kalman Filter notation

$$\hat{x}_{k+1} = \hat{x}_k + K_{k+1}(z_{k+1} - \hat{x}_k)$$

$$K_{k+1} = \frac{\sigma_k^2}{\sigma_k^2 + \sigma_z^2} \; ; \; \sigma_k^2 = \sigma_1^2 \; ; \; \sigma_z^2 = \sigma_2^2$$

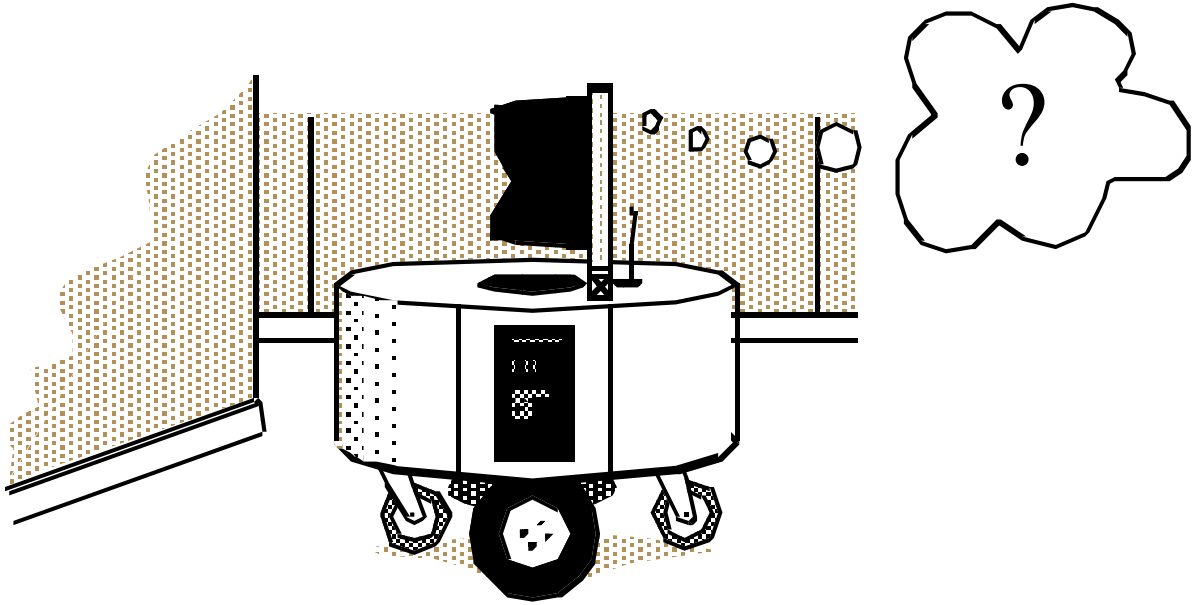
- The best estimate $\hat{x}_{k+1}$ of the state x_{k+1} at time k+1 equals
  - the best prediction of the value $\hat{x}_k$ before the new measurement z_{k+1} is taken,
  - plus a correction term of an optimal weighting value times the difference between z_{k+1} and the best prediction $\hat{x}_k$ at time k+1
- The variance of the state $\hat{x}_{k+1}$ is $\sigma_{k+1}^2 = \sigma_k^2 - K_{k+1}\sigma_k^2$

Kalman Filter for M.R. Localization



- Robot position prediction (1)
- Observation (2)
- Measurement prediction (3)
- Matching (4)
- Estimation: Applying the Kalman filter (5)

Where am I going? How do I get there?



Competencies for Navigation I

- Cognition / Reasoning :
  - The ability to decide what actions are required to achieve a certain goal in a given situation (belief state)
  - Decisions ranging from what path to take to what information on the environment to use.
- Industrial robots can operate without cognition (reasoning) because their environment is static and very structured

Competencies for Navigation II

- In mobile robotics, cognition and reasoning is primarily of geometric nature
  - e.g., Picking safe path or determining where to go next
  - It has been largely explored in literature for cases in which complete information about the current situation and the environment exists (e.g. *salesman problem*)
  - However, in mobile robotics the knowledge of about the environment and situation is usually only *partially known* and is *uncertain*
    - The task is much more difficult
    - It requires multiple tasks running in parallel, some for planning (global), some to guarantee "survival of the robot"

Competencies for Navigation III

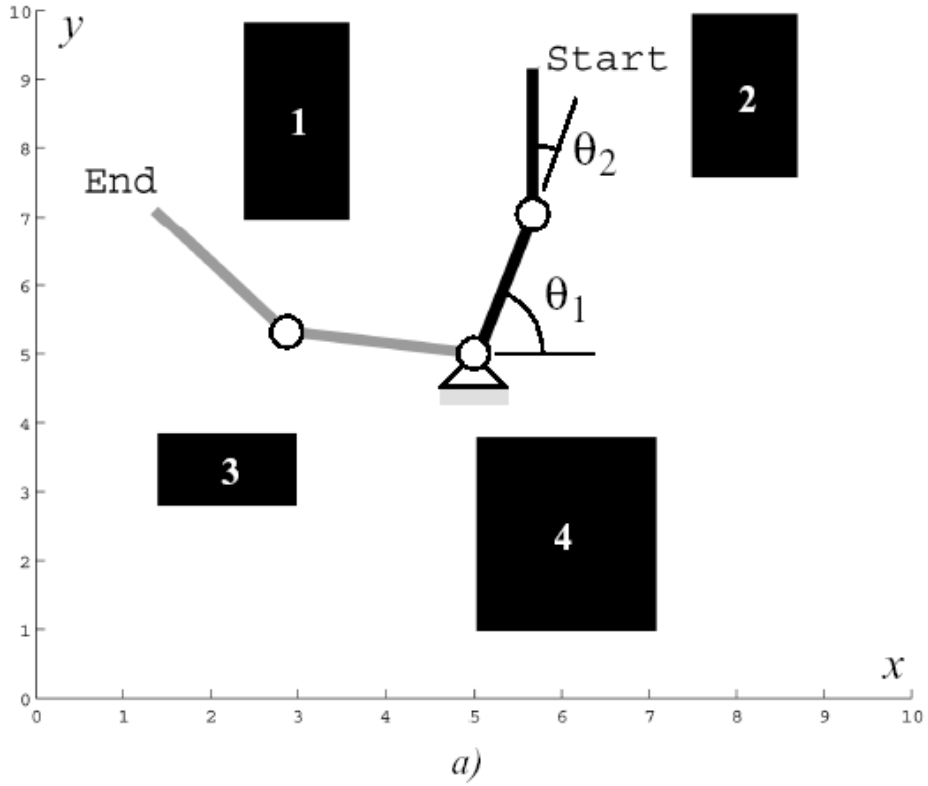
- Robot control can usually be decomposed in various behaviors or functions
  - Wall following
  - Localization
  - Path generation
  - Obstacle avoidance
- We are concerned with path planning & navigation, except low level motion control and localization
- We can generally distinguish between (global) path planning and (local) obstacle avoidance

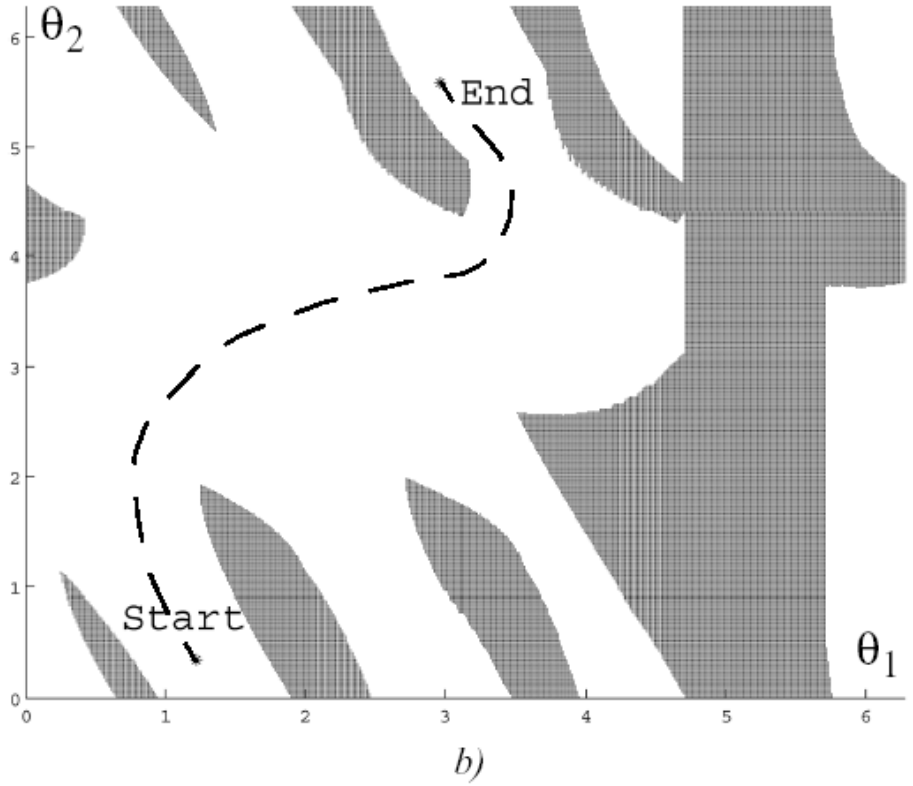
Global Path Planning

- Assumption:
  - Exist a good enough map of the environment for navigation
  - *Topological* or *metric* or a *mixture* between both
- First step:
  - Representation of the environment by a road-map (graph), cells or a potential field
  - The resulting *discrete locations* or *cells* allow then to use standard planning algorithms
- Examples:
  - Visibility Graph
  - Voronoi Diagram
  - Cell Decomposition -> Connectivity Graph
  - Potential Field

Path Planning: Configuration Space

- *State* or *configuration q* can be described with *k* values q_i

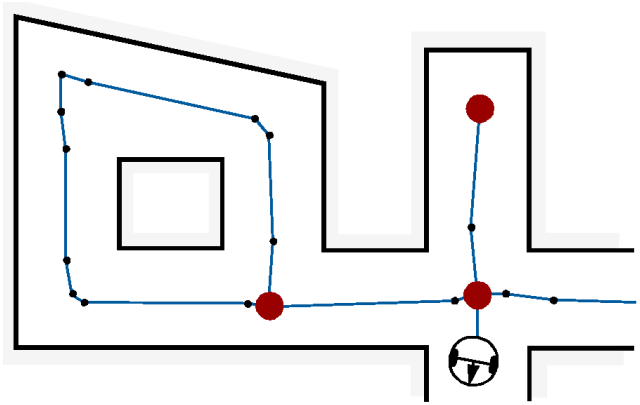




- What is the configuration space of a mobile robot?

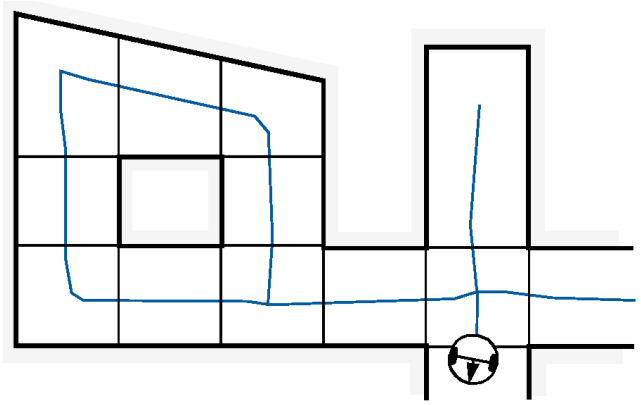
Path Planning Overview

- *Road Map, Graph construction*
  - Identify a set of routes within the free space



- Where to put the nodes?
- Topology-based:
  - at distinctive locations
- Metric-based:
  - where features disappear or get visible

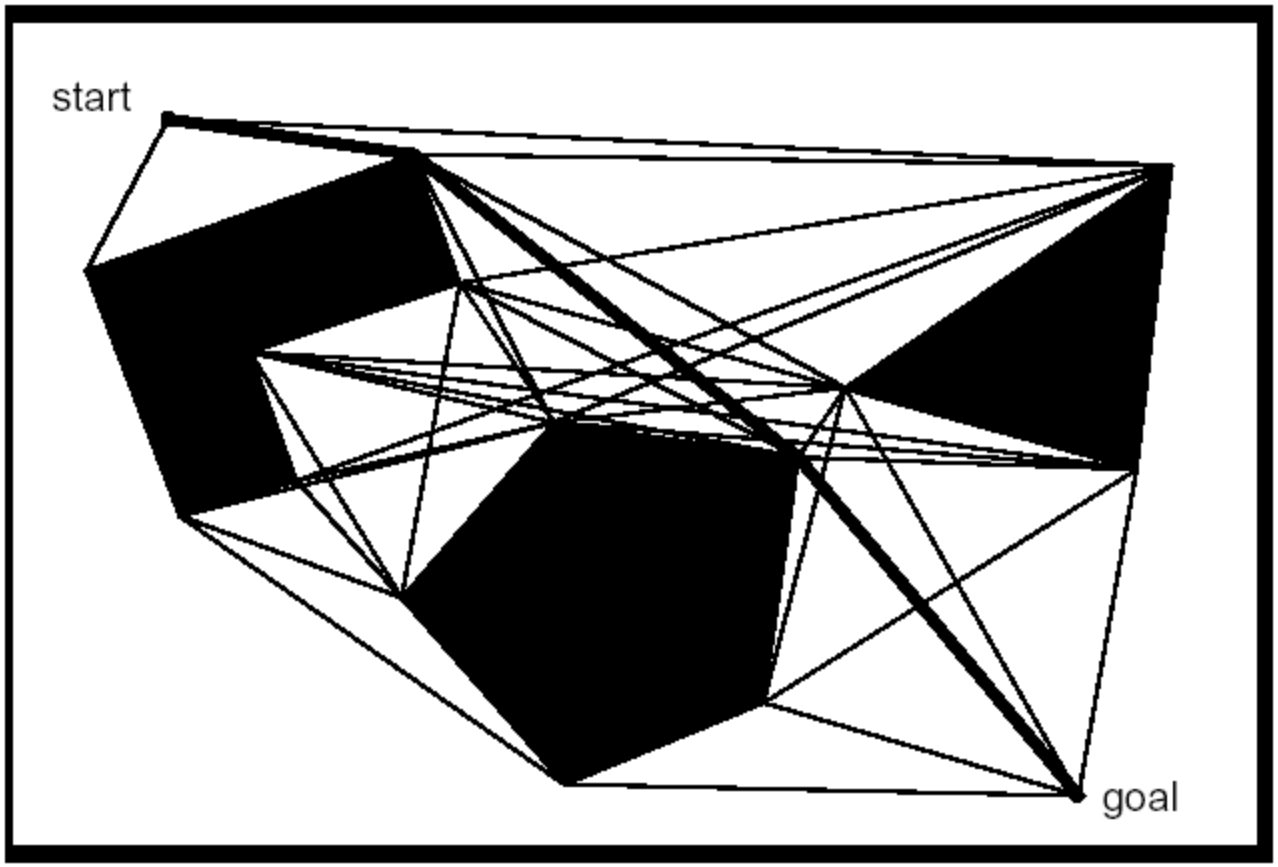
- *Cell decomposition*
  - Discriminate between free and occupied cells



- Where to put the cell boundaries?
- Topology- and metric-based:
  - where features disappear or get visible
- Potential Field:
  - Imposing a mathematical function over the space

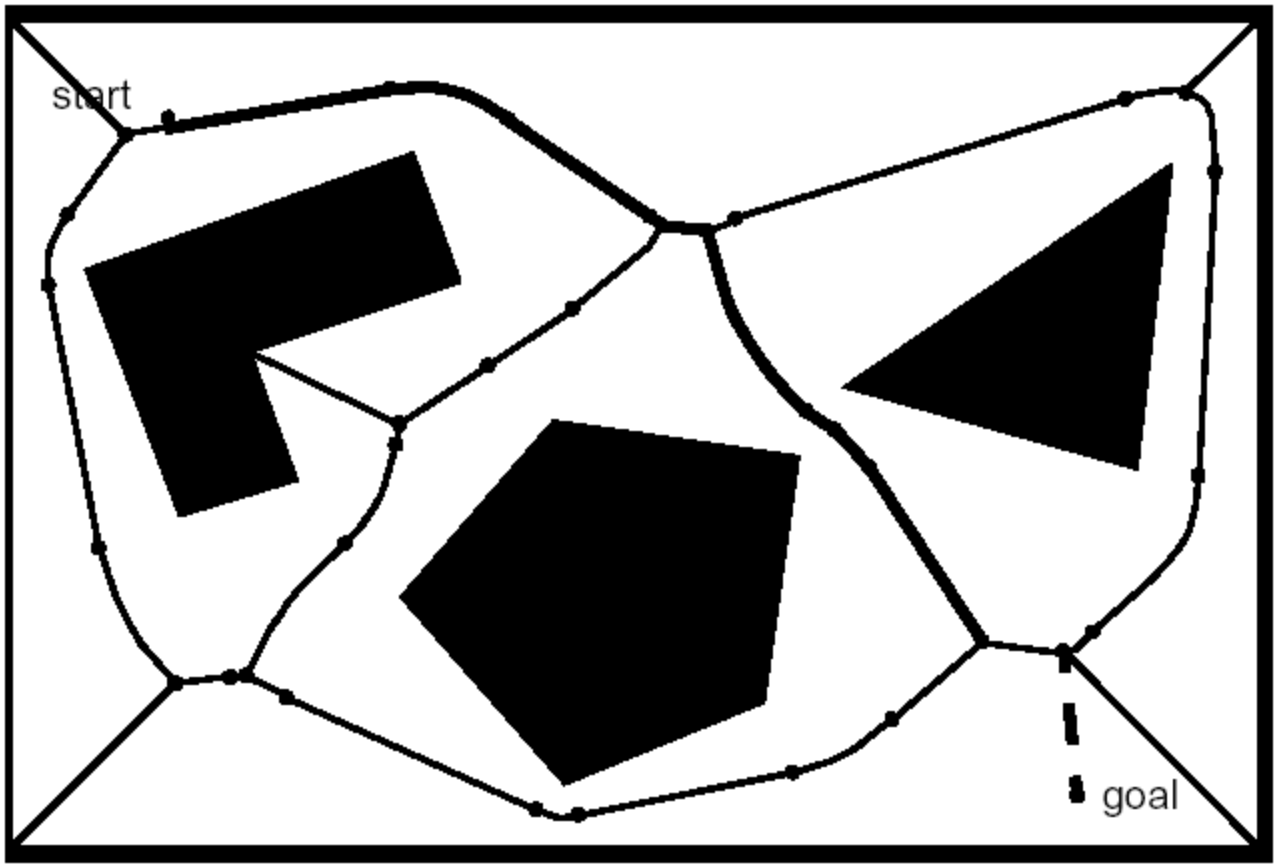
Road-Map Path Planning: Visibility Graph

- *Shortest path length*
- Grow *obstacles* to avoid collisions



Road-Map Path Planning: Voronoi Diagram

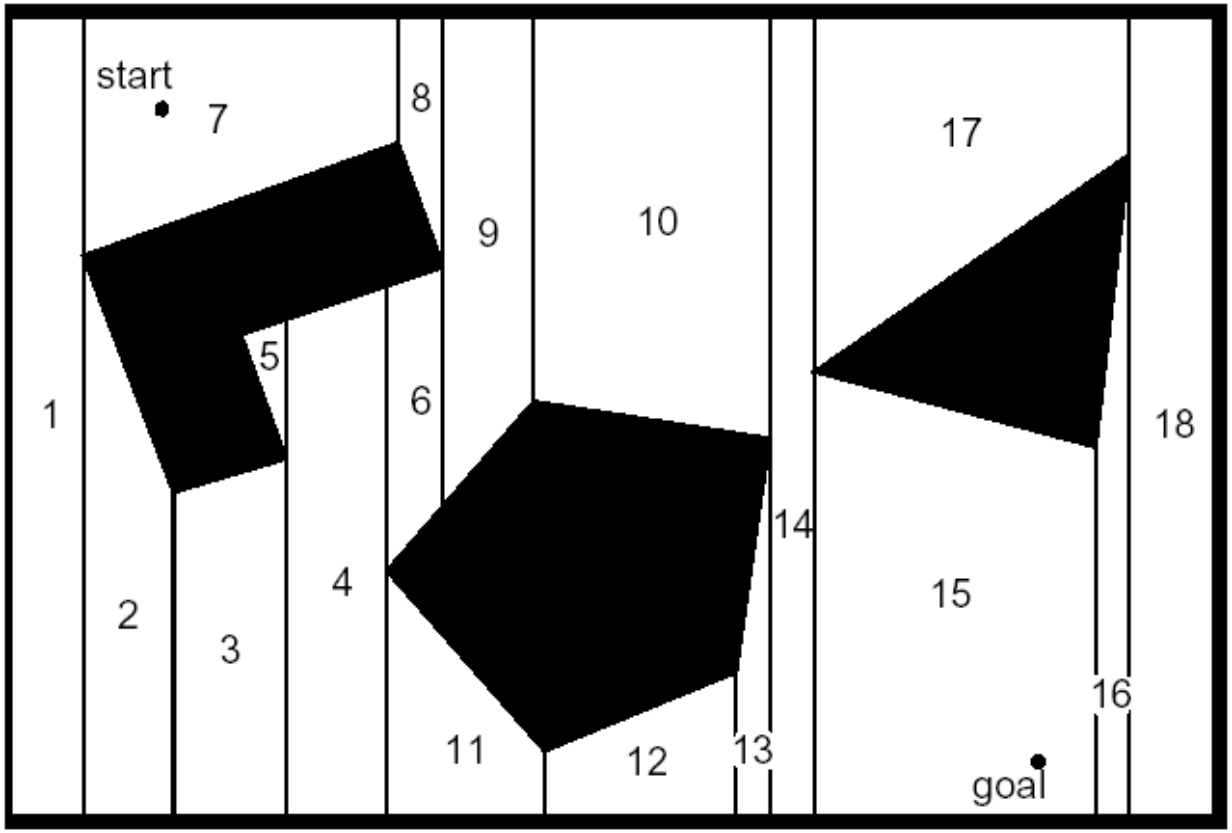
- Easy executable: Maximize the sensor readings
- Works also for map-building: *Move on the Voronoi edges*

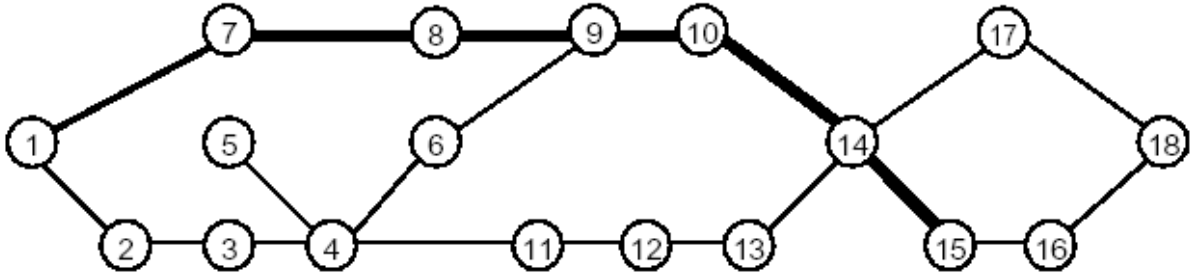


Cell Decomposition

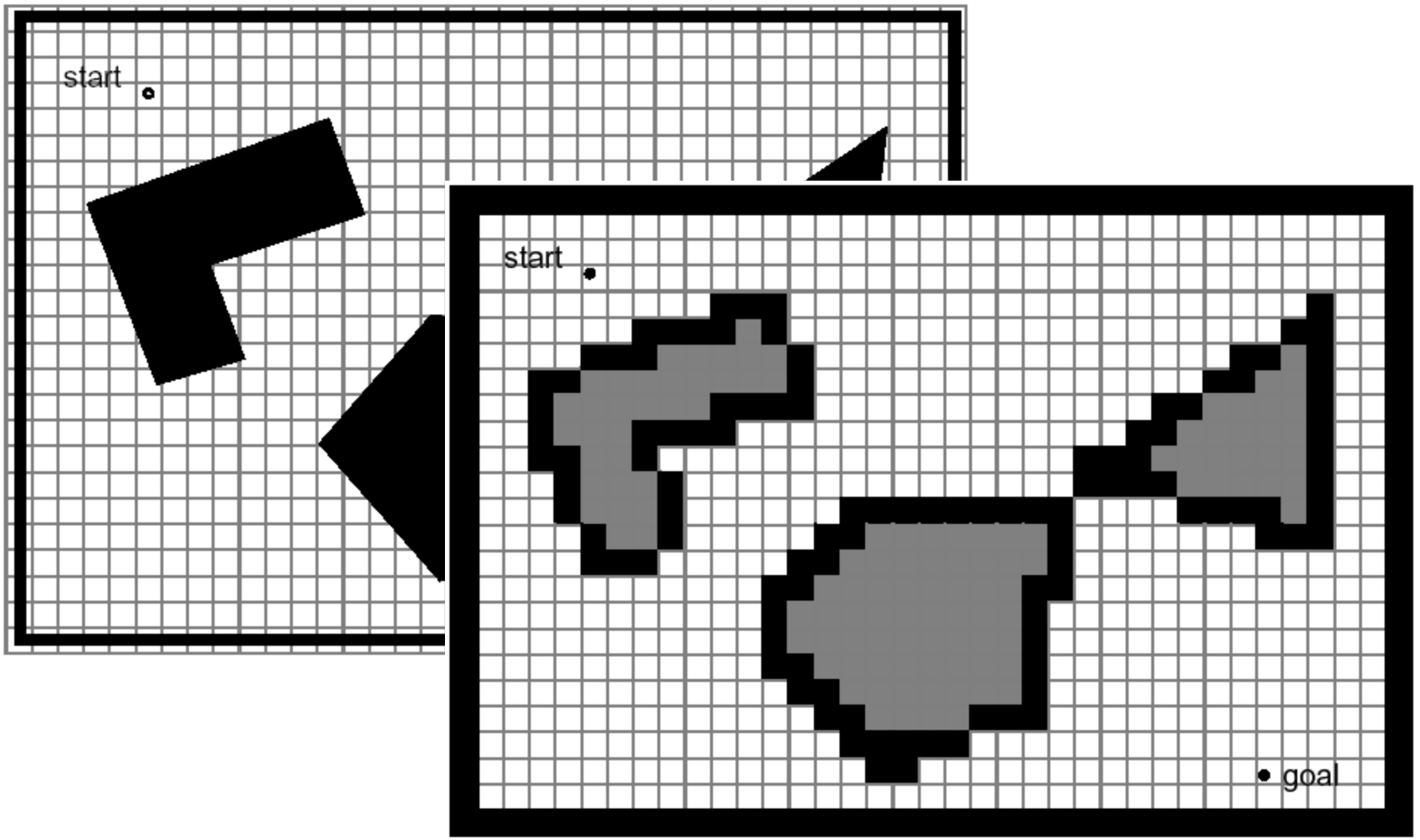
- Divide space into simple, connected regions: cells
- Determine which open cells are *adjacent* and construct a connectivity graph
- Find cells in which the *initial and goal configuration* (state) lie and search for a path in the connectivity graph to join them
- From the *sequence of cells* found with an appropriate search algorithm, *compute a path* within each cell
  - e.g. passing through the midpoints of cell boundaries or by sequence of wall following movements

Exact Cell Decomposition

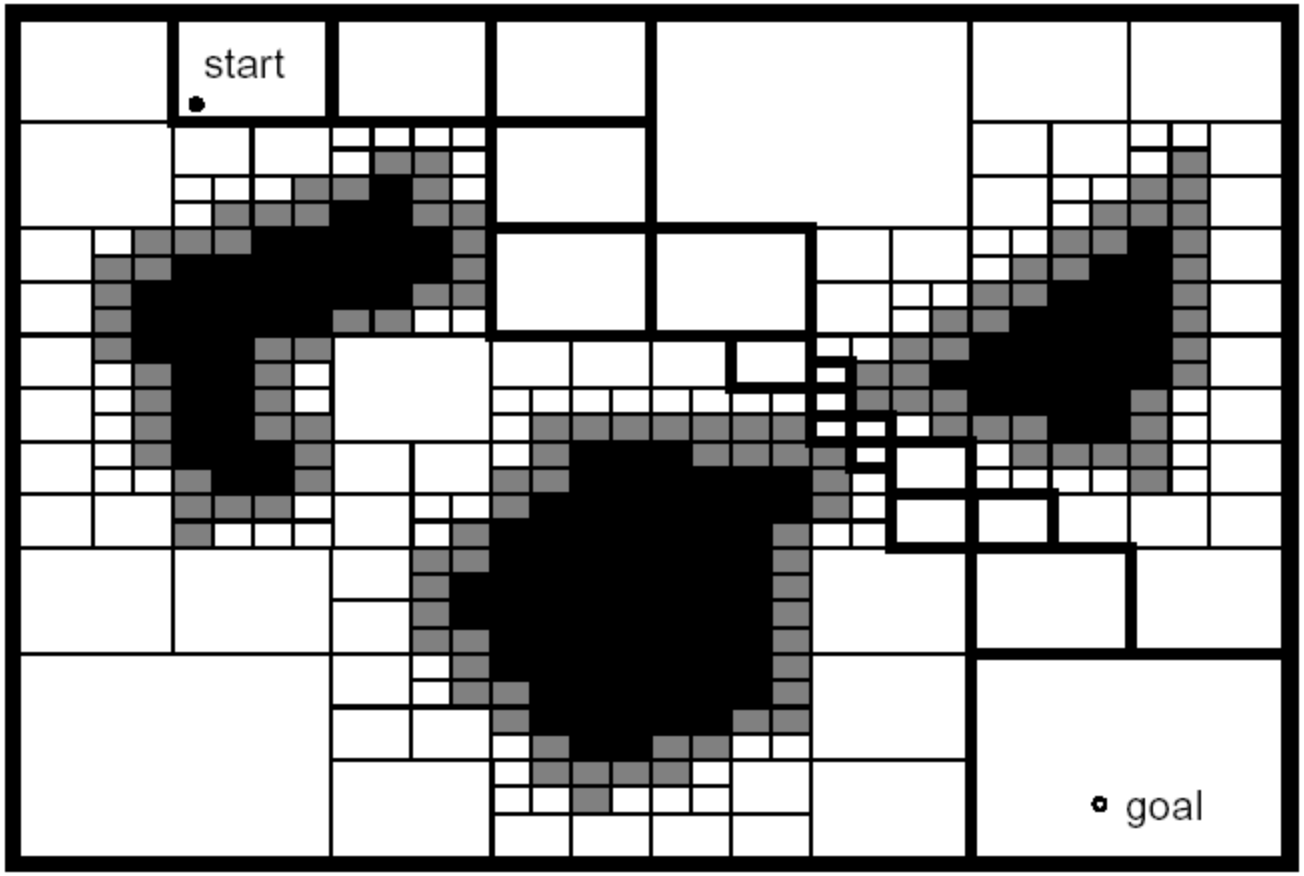




Approximate Cell Decomposition

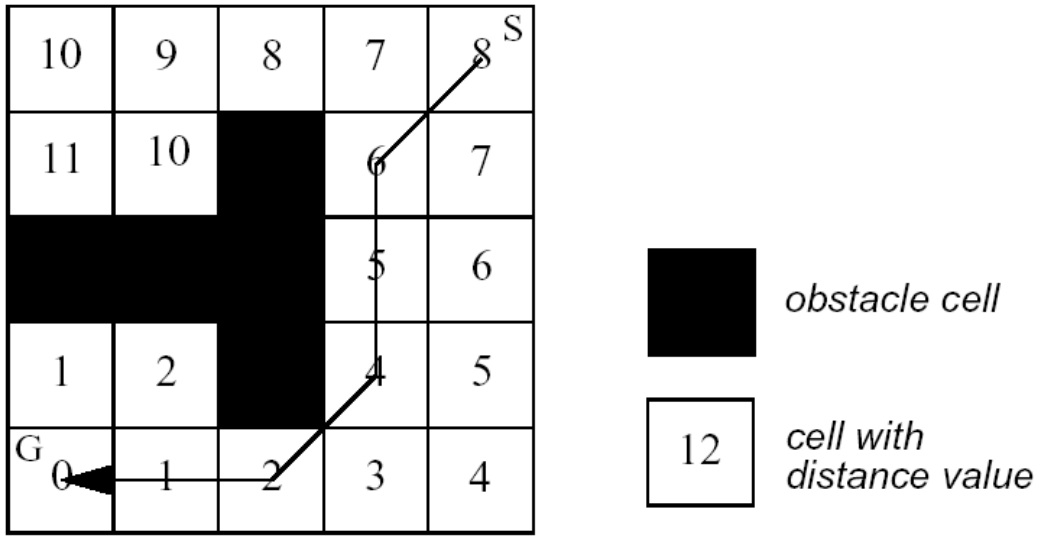


Adaptive Cell Decomposition



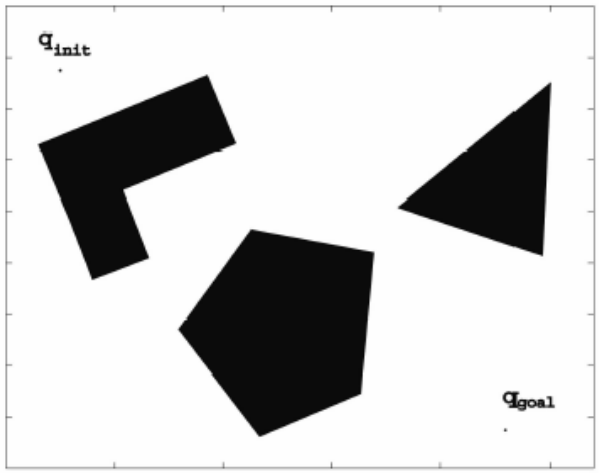
Path/Graph Search Strategies

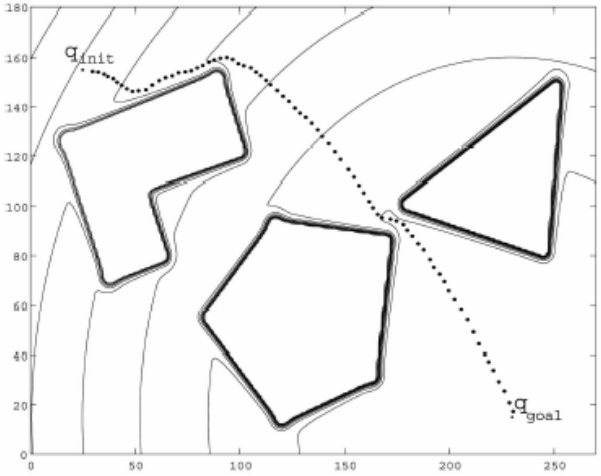
- *Wavefront Expansion* NF1

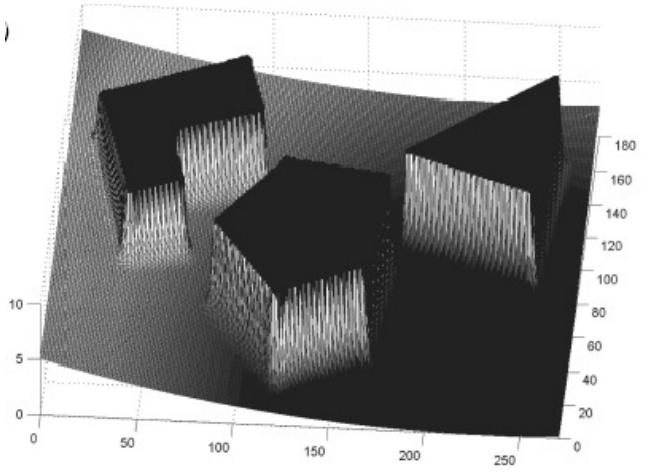


Potential Field Path Planning

- Robot is treated as a point under the influence of an artificial potential field
  - Generated robot movement is similar to *a ball rolling down the hill*
  - Goal generates attractive force
  - Obstacle are repulsive forces







Potential Field Generation

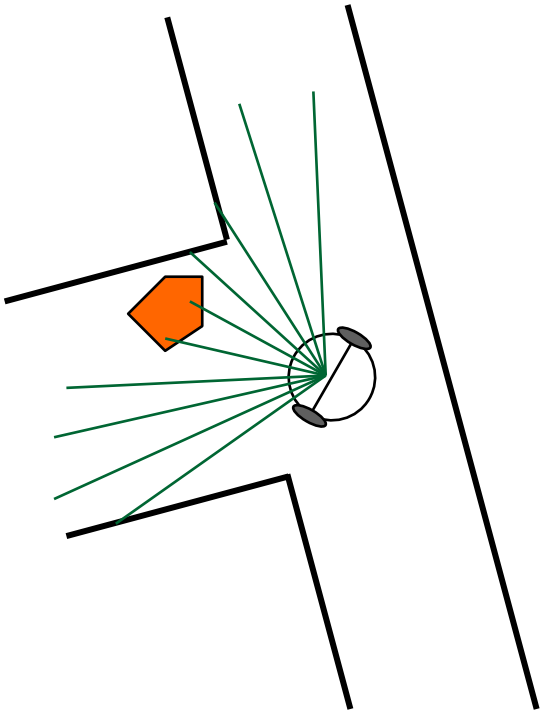
- Generation of potential field function $U(q)$
  - *attracting* (goal) and *repulsing* (obstacle) fields
  - *summing up the fields*
  - functions must be differentiable
- Generate artificial force field $F(q)$

$$F(q) = -\nabla U(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) = \begin{bmatrix} \frac{\partial U}{\partial x} \\ \frac{\partial U}{\partial y} \end{bmatrix}$$

- Set robot speed (v_x, v_y) proportional to the force $F(q)$ generated by the field
  - the force field drives the robot to the goal
  - if robot is assumed to be a point mass

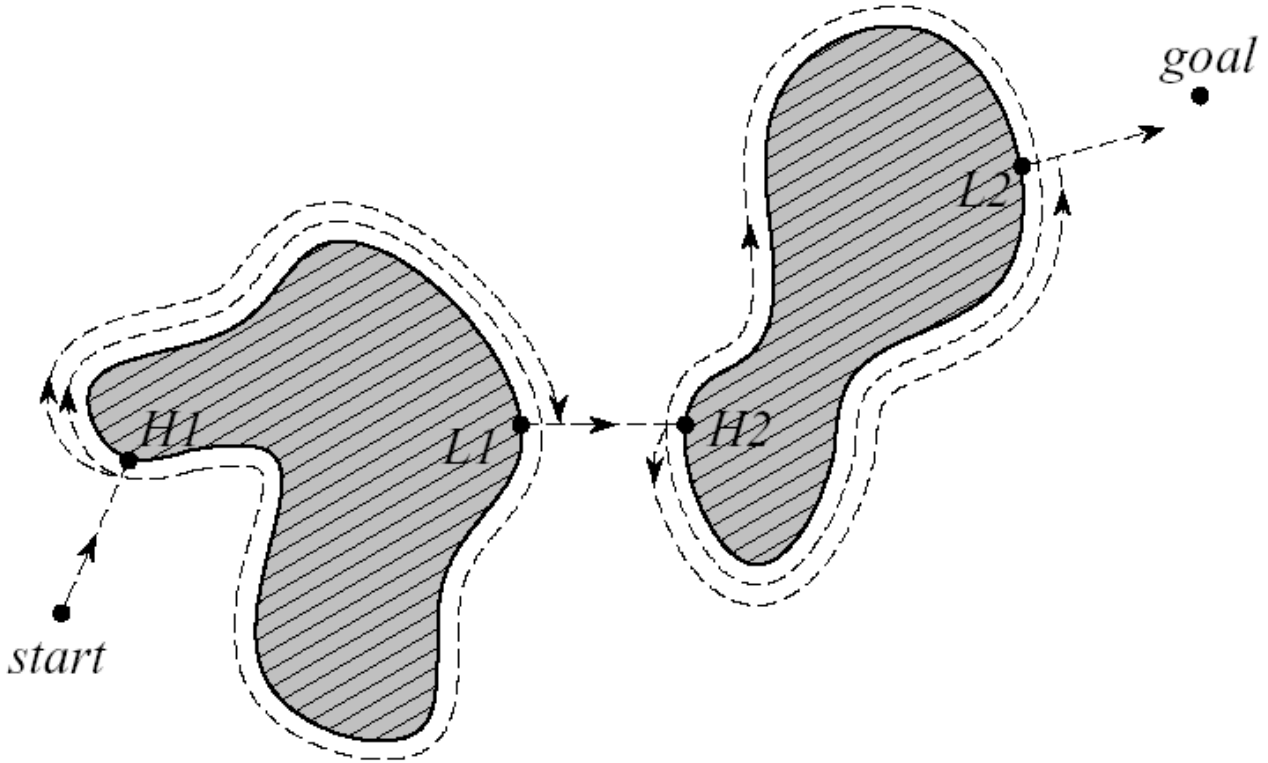
Obstacle Avoidance

- The goal of the obstacle avoidance algorithms is to *avoid collisions* with obstacles
- It is usually based on local map
- Often implemented as a more or less *independent task*
- However, efficient obstacle avoidance should be optimal with respect to
  - the overall goal
  - the actual speed and kinematics of the robot
  - the on boards sensors
  - the actual and future risk of collision



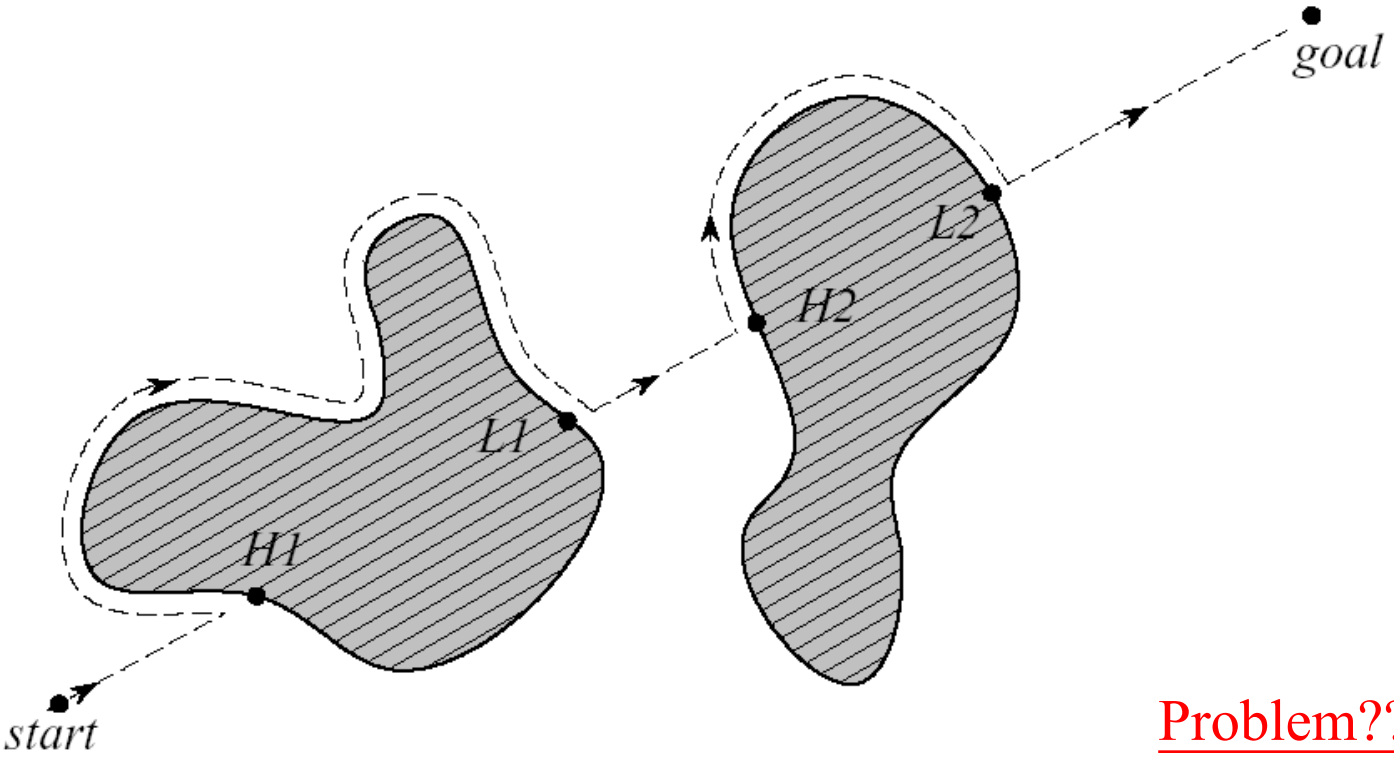
Obstacle Avoidance: Bug1

- *Following* along the obstacle to avoid it
- Each encountered obstacle is once fully circled before it is left at the point closest to the goal

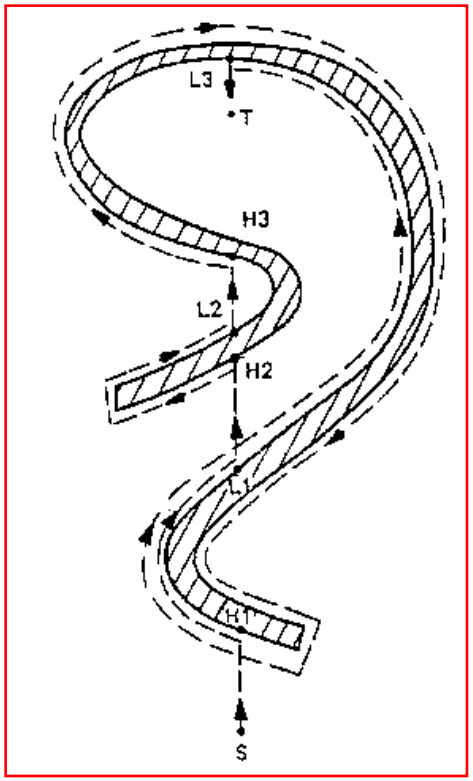


Obstacle Avoidance: Bug2

- *Following* the obstacle always on the left or right side
- Leaving the obstacle if the direct connection between start and goal is crossed



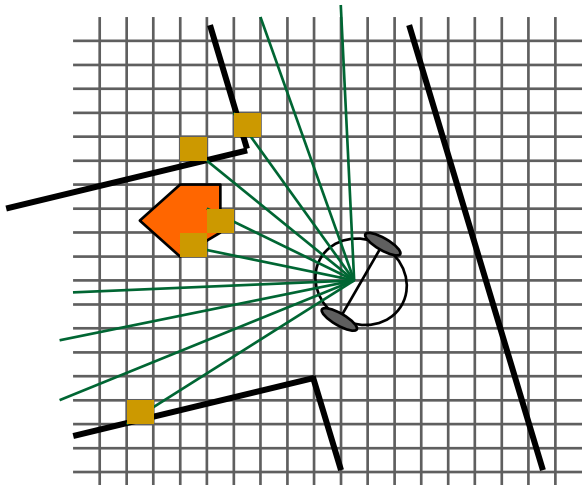
Problem??

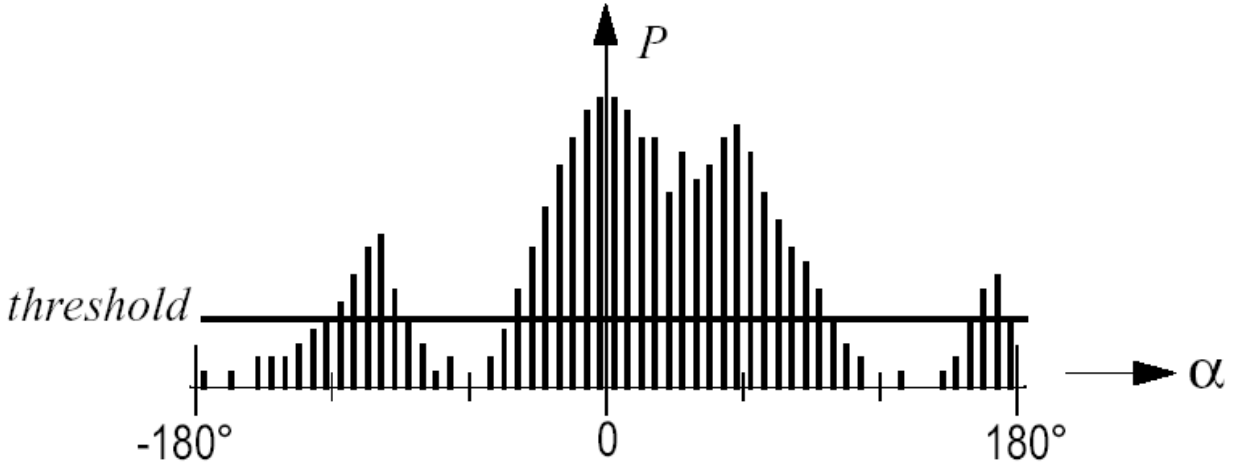


Vector Field Histogram (VFH)

- Environment represented in a grid (*2 DOF*)
  - cell values equivalent to probability that there is an obstacle
- Reduction in different steps to a 1 DOF histogram
  - calculation of *steering direction*
  - all openings for the robot to pass are found
  - the one with *lowest cost function* G is selected

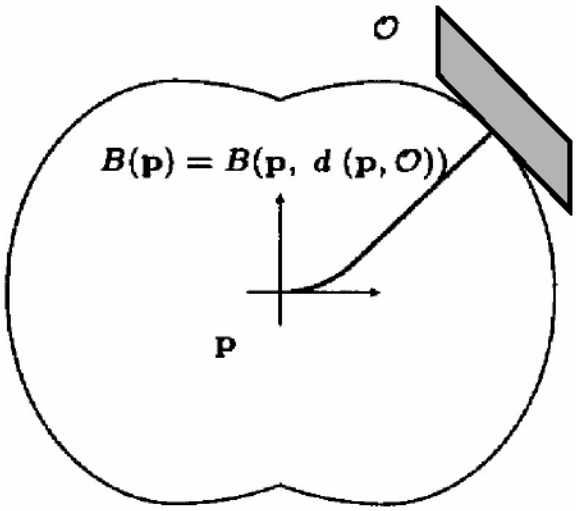
$$G = a \cdot \text{target\_direction} + b \cdot \text{wheel\_orientation} + c \cdot \text{previous\_direction}$$

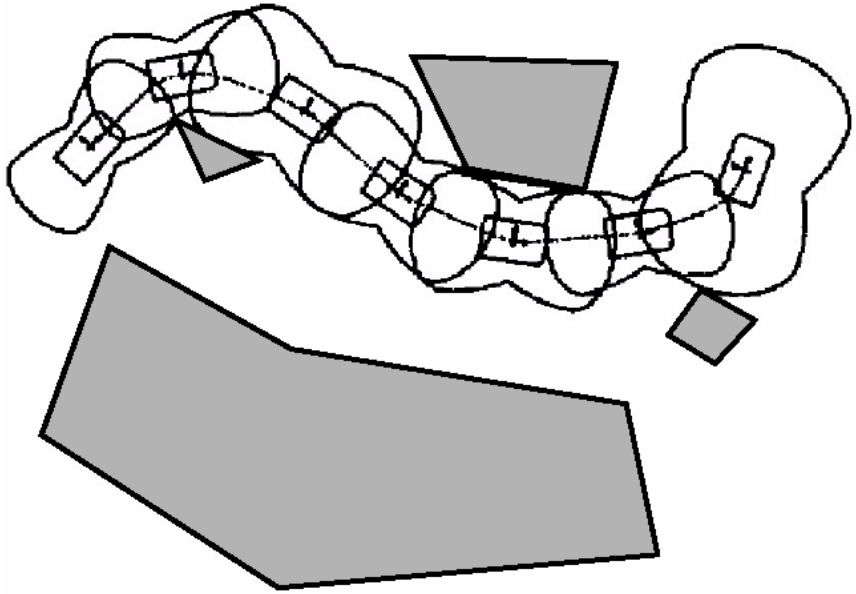




The Bubble Band Concept

- Bubble = Maximum free space which can be reached without any risk of collision
  - generated using the distance to the object and *a simplified model of the robot*
  - bubbles are used to form a band of bubbles which *connects the start point with the goal point*





Testing a Model

- After the model is selected, the final model is evaluated against the test set to **estimate the final model accuracy**.
- Very important: never “peek” at the test set during learning!

How to Split the Dataset

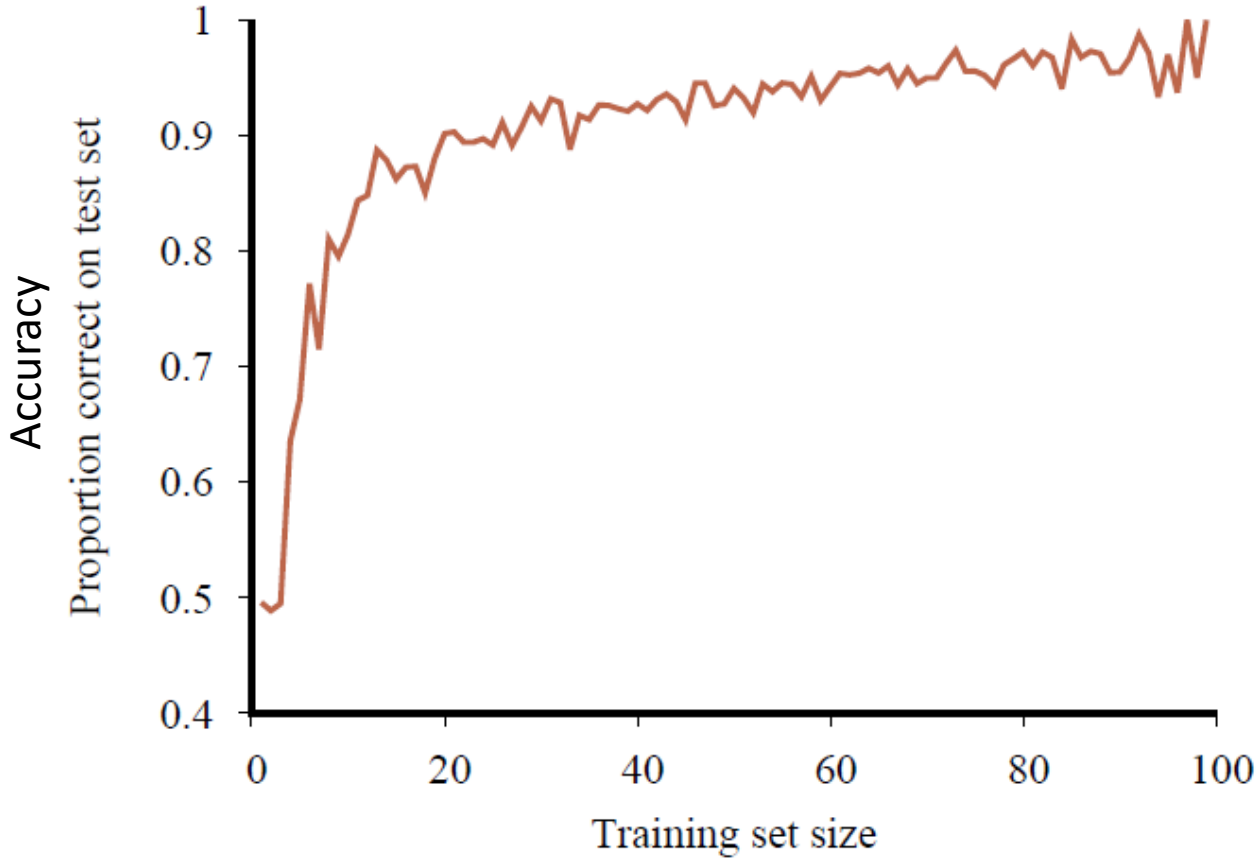
- **Random splits:** *Split* the data randomly in, e.g., 60% training, 20% validation, and 20% testing.

- **Stratified splits:** Like random splits, but balance classes and other properties of the examples.

- ***k*-fold cross validation:** Use training & validation data better
  - Split the training & validation data randomly into *k* folds.
  - For *k* rounds hold 1 fold back for testing and use the remaining $k - 1$ folds for training.
  - Use the *average* error/accuracy as a better estimate.
  - Some algorithms/tools do this internally.

- **LOOCV** (*leave-one-out cross validation*): $k = n$ used if very little data is available.

Learning Curve:
The Effect the Training Data Size



Accuracy of a classifier when the amount of available training data increases.

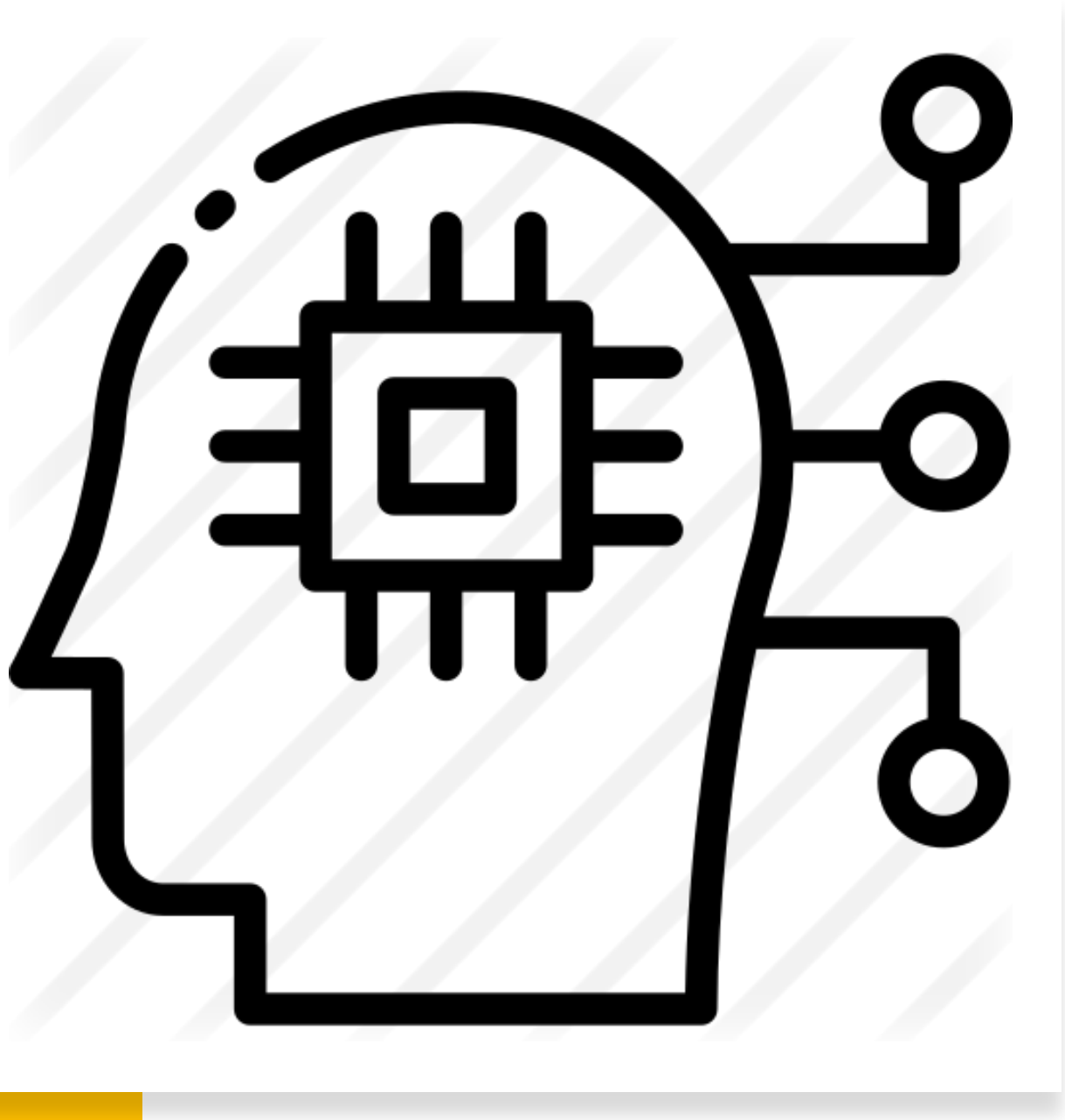
More data is better!

Comparing to a Baselines

- First step: get a **baseline**
  - Baselines are very simple straw man model.
  - Helps to determine how hard the task is.
  - Helps to find out what a good accuracy is.

- **Weak baseline**: The most frequent label classifier
  - Gives all test instances whatever label was *most common* in the training set.
    - Example: For spam filtering, give every message the label "ham."
  - Accuracy might be very high if the problem is skewed (called class imbalance).
    - Example: If calling everything "ham" gets already 66% right, so a classifier that gets 70% isn't very good…

- **Strong baseline**: For research, we typically compare to previous work as a baseline.



Types of Models

Regression: Predict a number

Classification: Predict a label

Regression: Linear Regression

Model: $h_{\boldsymbol{w}}(\boldsymbol{x}_j) = w_o + w_1 x_{j,1} + \cdots + w_n x_{j,n} = \sum_i w_i x_{j,i} = \boldsymbol{w}^T \boldsymbol{x}_j$

Empirical Loss: $L(\boldsymbol{w}) = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2$

Squared error loss over the whole data matrix $\boldsymbol{X}$

Gradient: $\nabla L(\boldsymbol{w}) = 2\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$

The gradient is a vector of partial derivatives

$$\nabla L(\boldsymbol{w}) = \left[\frac{\partial L}{\partial w_1}(\boldsymbol{w}), \frac{\partial L}{\partial w_2}(\boldsymbol{w}), \ldots, \frac{\partial L}{\partial w_n}(\boldsymbol{w})\right]^T$$

Find: $\nabla L(\boldsymbol{w}) = 0$

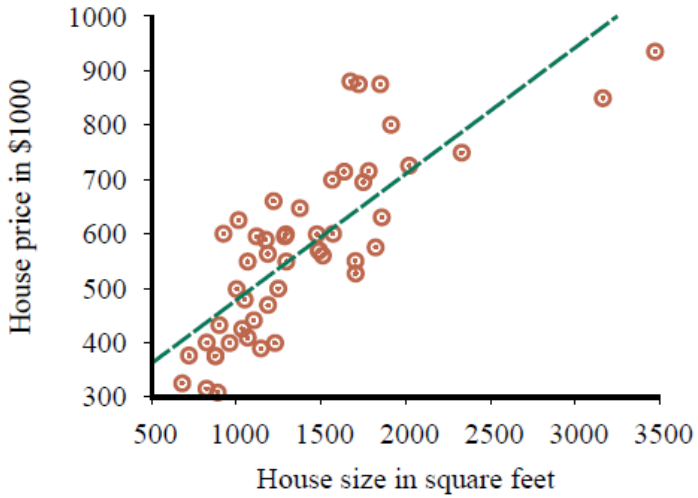
Gradient descend:

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla L(\boldsymbol{w})$$

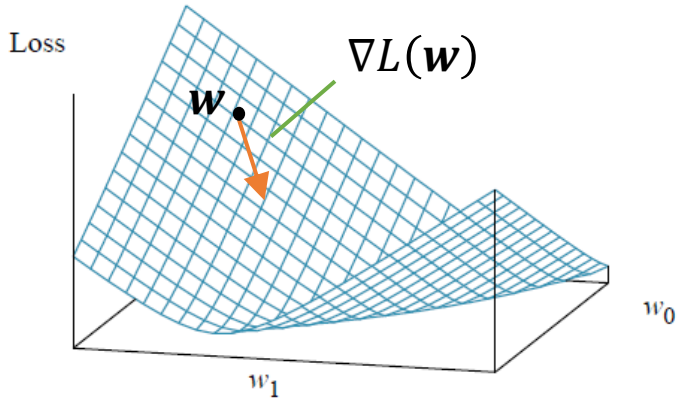
Analytical solution:

$$\boldsymbol{w}^* = \underbrace{(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T}\boldsymbol{y}$$

Pseudo inverse



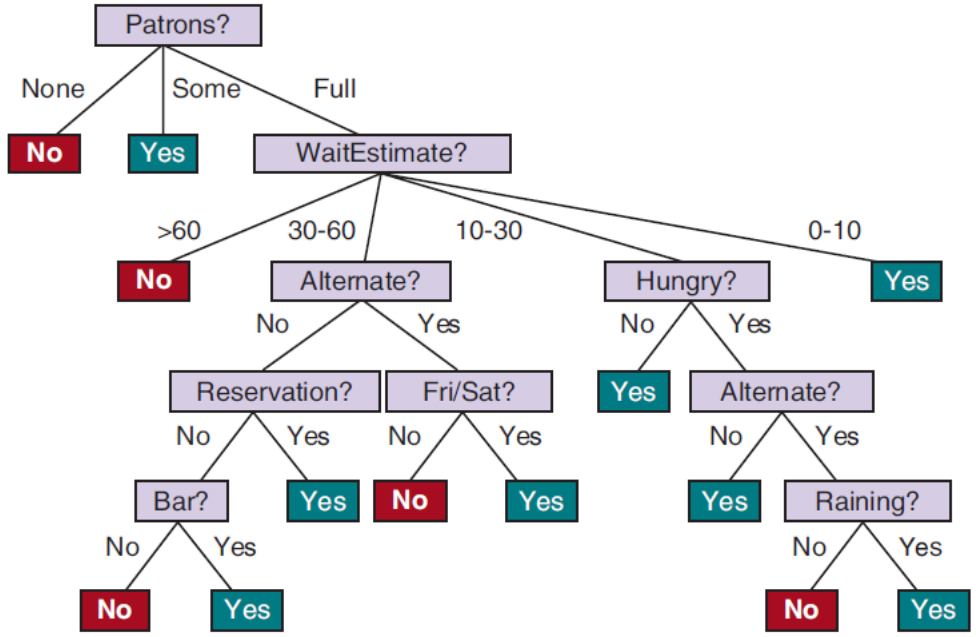
(a)



(b)

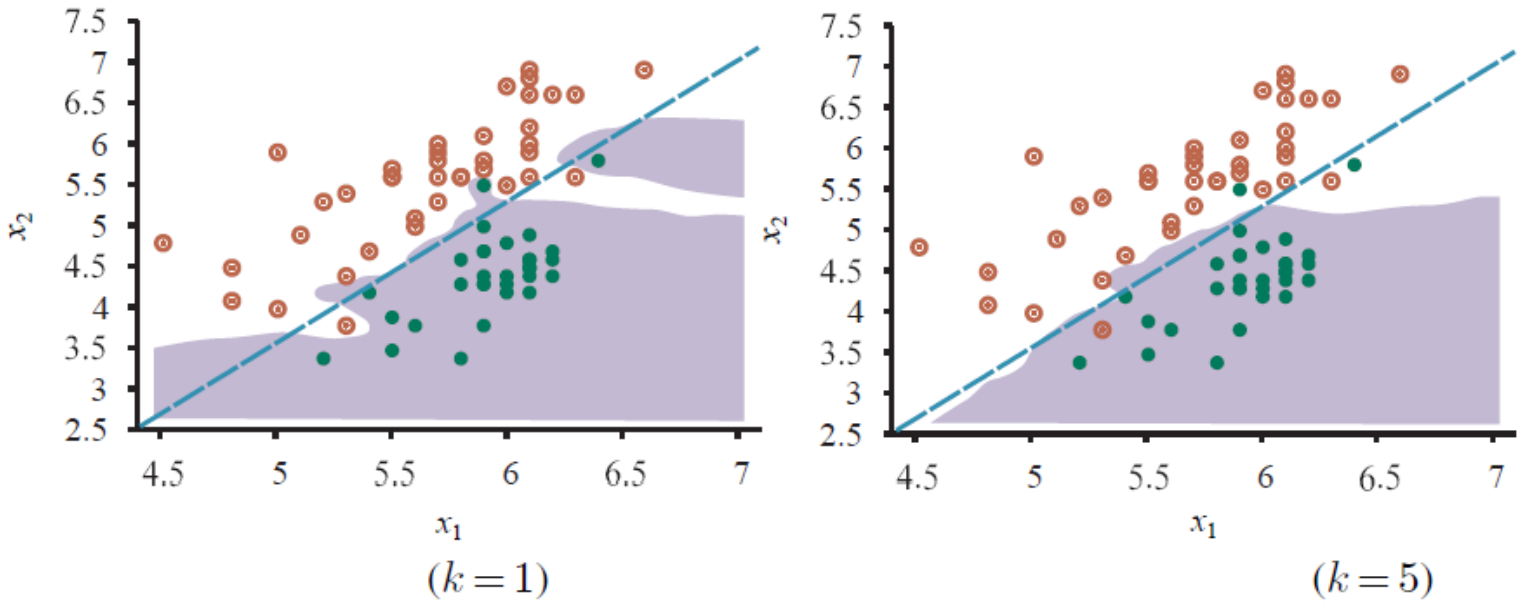
Decision Trees

Example	Input Attributes										Output
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$



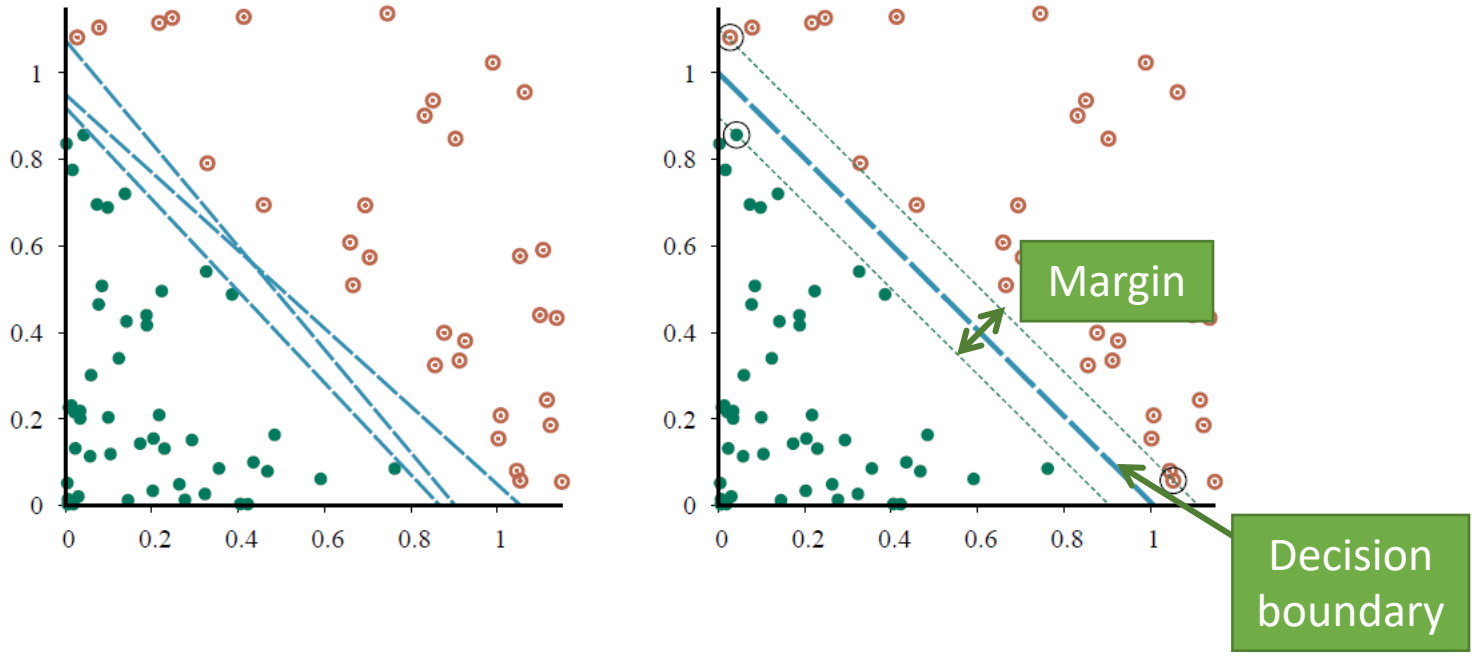
- A **sequence of decisions** represented as a tree.
- Many implementations that *differ* by
  - How to select features to split?
  - When to stop splitting?
  - Is the tree pruned?
- Approximates a Bayesian classifier by $h(x) = \underset{y}{\operatorname{argmax}} P(Y = y \mid \text{leafNodeMatching}(x))$

K-Nearest Neighbors Classifier



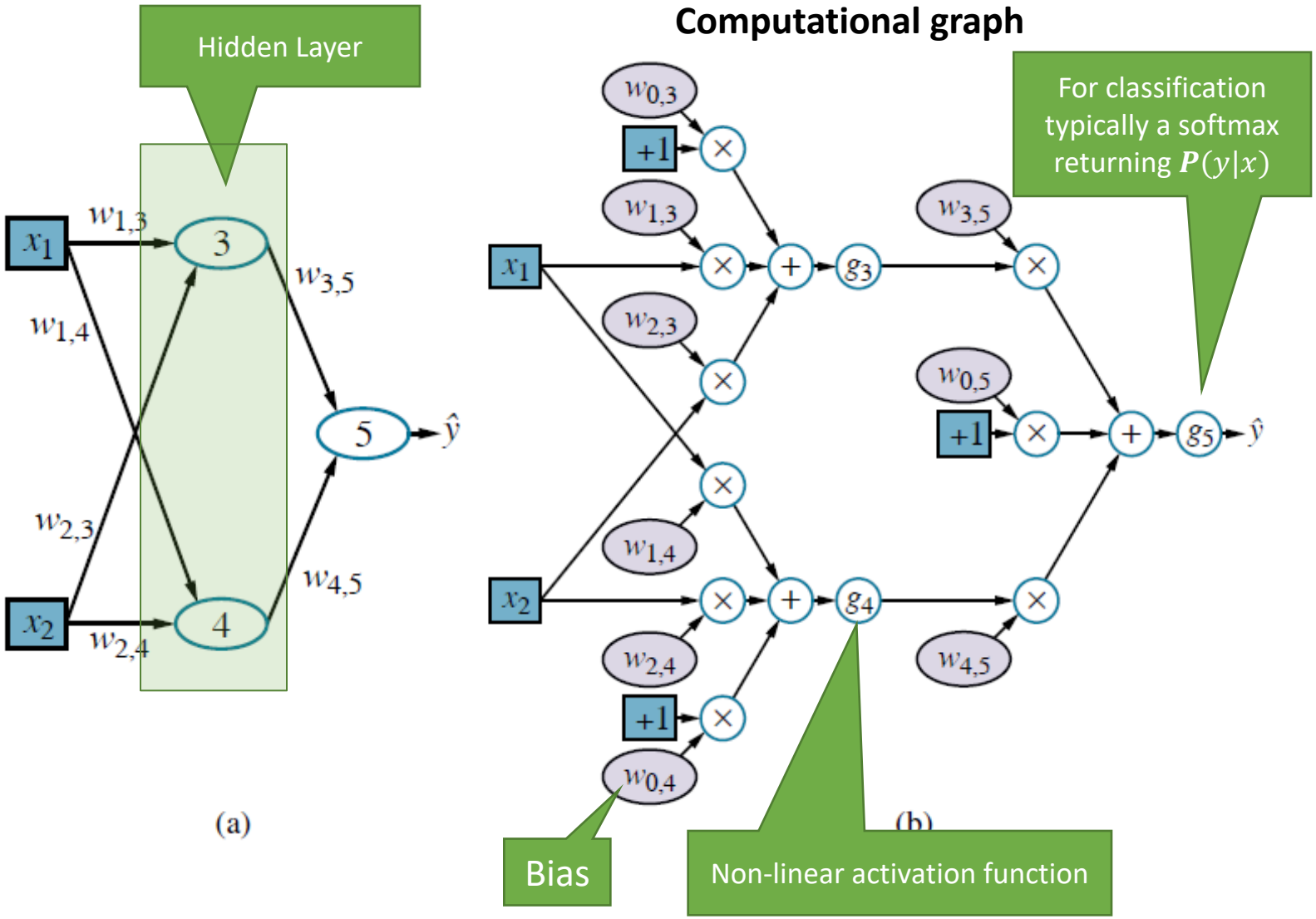
- Class is predicted by looking at the majority in the set of the *k* nearest **neighbors**. *k* is a hyperparameter. Larger *k* smooth the decision boundary.
- Neighbors are found using a distance measure (e.g., Euclidean distance between points).
- Approximates a Bayesian classifier by $h(x) = \underset{y}{\operatorname{argmax}} P(Y = y \mid \text{neighborhood}(x))$

Support Vector Machine (SVM)



- Linear classifier that finds **the maximum margin separator** using only the points that are “support vectors” and quadratic optimization.
- The kernel trick can be used to learn non-linear decision boundaries.

Artificial Neural Networks/Deep Learning



- Represent $\hat{y} = h(x)$ as a network of weighted sums with non-linear activation functions (e.g., logistic, ReLU).
- Learn weights **w** from examples using **backpropagation** of prediction errors $\mathrm{L}(\hat{y}, y)$ (gradient descend).
- ANNs are **universal approximators**. Large networks can approximate any function (no bias). **Regularization** is typically used to avoid overfitting.
- **Deep learning** adds more hidden layers and layer types (e.g., convolution layers) for better learning.

Other Popular Models and Methods

Many other models exist

- **Generalized linear model (GLM):** This important model family includes **linear regression** and the classification method **logistic regression.**

Often used methods

- **Regularization:** enforce simplicity by using a penalty for complexity.
- **Kernel trick:** Let a linear classifier learn non-linear decision boundaries (= a linear boundary in a high dimensional space).
- **Ensemble Learning:** Use many models and combine the results (e.g., random forest, boosting).
- **Embedding and Dimensionality Reduction:** Learn how to represent data in a simpler way.

Example Use of ML for Intelligent Agents

Learn Actions

- Directly learn the best action from examples.

$$action = h(state)$$

- This model can also be used as a **playout policy** for Monte Carlo tree search with data from self-play.

Learn Heuristics

- Learn evaluation functions for states.

$$eval = h(state)$$

- Can learn a **heuristic** for minimax search from examples.

Perception

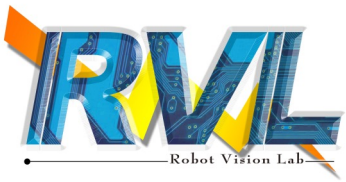
- **Natural language processing:** Use deep learning / word embeddings / language models to understand concepts, translate between languages, or generate text.
- Speech recognition: Identify the most likely sequence of words.
- Vision: Object recognition in images/videos. Generate images/video.

Artificial Intelligence

Computer Vision

Robot Vision Lab

Huei-Yung Lin



What is Computer Vision?

- Consider the problem associated with extracting information from image data
- What do we mean by information?
  - 3D structure of a scene
  - Presence and location of moving objects
  - Identity of a person
- Inverse Problem of Image Formation
  - To recover useful information about a scene from its two-dimensional projections

Vision as Inverse Problem

- Given information about the geometry of the scene, surface reflectance properties, lighting configuration and camera model we can generate an image

- The problem of computer vision is to infer these properties of the scene from the image data (a 2D array of numbers)
- Inverse problems are notoriously difficult

What is Computer Vision?

- Target problem of computer vision
  - Computing properties of the 3-D world from one or more digital images
  - Including geometric (shape and position of an object) and dynamic (object velocities)
  - We will focus more on 2-D computer vision and image processing in this course
- Image Analysis, Scene Analysis, Image Understanding
- Vision = Geometry + Measurement + Interpretation

Why Study Computer Vision?

- Images and movies are everywhere
- Fast-growing collection of useful applications
  - building representations of the 3D world from pictures
  - automated surveillance (who's doing what)
  - movie post-processing
  - face finding
- Various deep and attractive scientific mysteries
  - how does object recognition work?
- Greater understanding of human vision

Computer Vision as a Sensor

- Information about distant objects
- Passive sensor (as opposed to sonar, laser)
- High bandwidth
  - 1 picture = ? Words
- Corresponds to the most complex human sensory function
  - Eat it? Run from it? Mate with it? …

Defined by Image and Model

Output \ Input	Image	Model
Image	Image Processing	Computer Graphics
Model	Computer Vision	Computational Geometry

Main Topics

- Shape (and motion) recovery

  "What is the 3D shape of what I see?"
- Segmentation

  "What belongs together?"
- Tracking

  "Where does something go?"
- Recognition

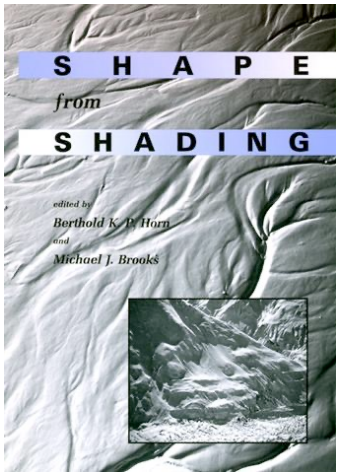
  "What is it that I see?"

Main Topics

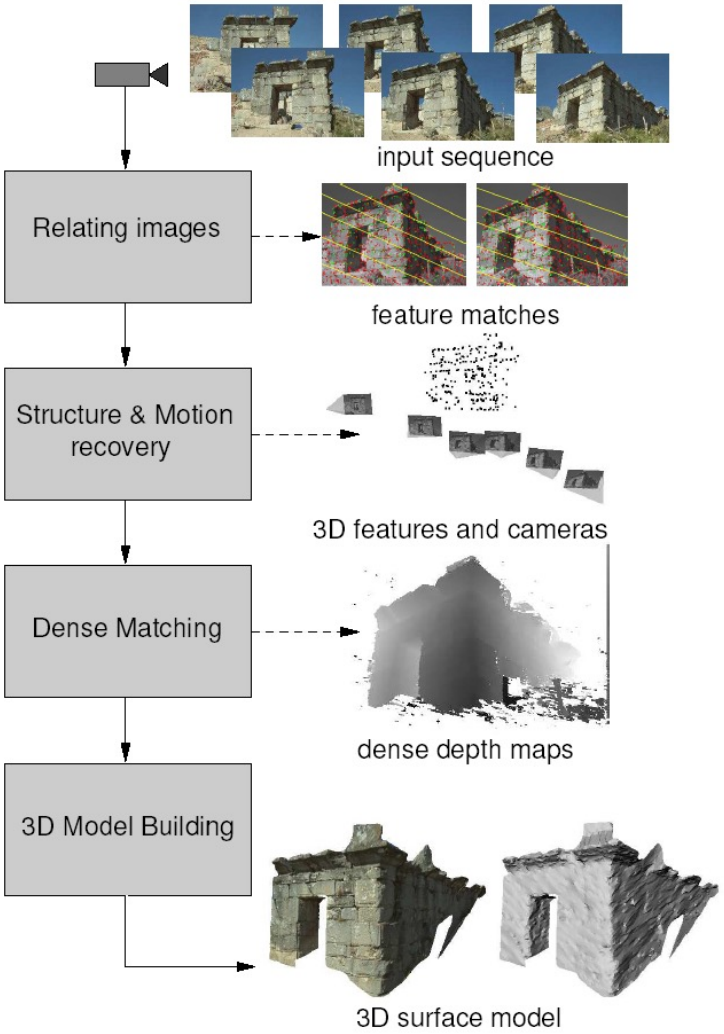
- Camera & Light
  - Geometry, Radiometry, Color
- Digital images
  - Filters, edges, texture, optical flow
- Shape (and motion) recovery
  - Multi-view geometry
  - Stereo, motion, photometric stereo, ...
- Segmentation
  - Clustering, model fitting, probalistic
- Tracking
  - Linear dynamics, non-linear dynamics
- Recognition
  - templates, relations between templates

Shape from …

- Many different approaches/cues

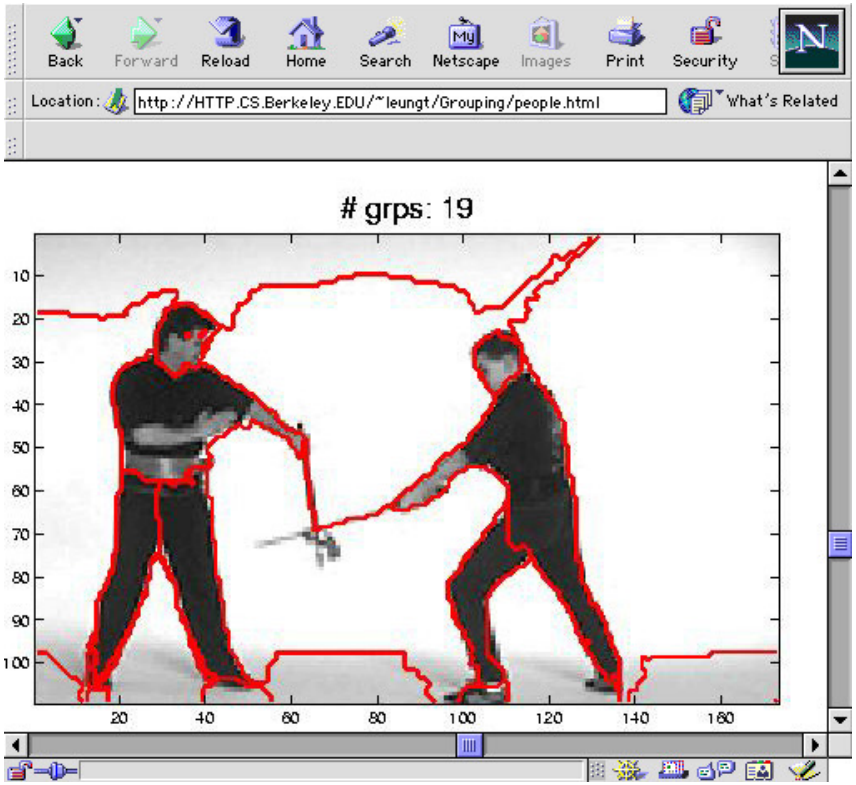


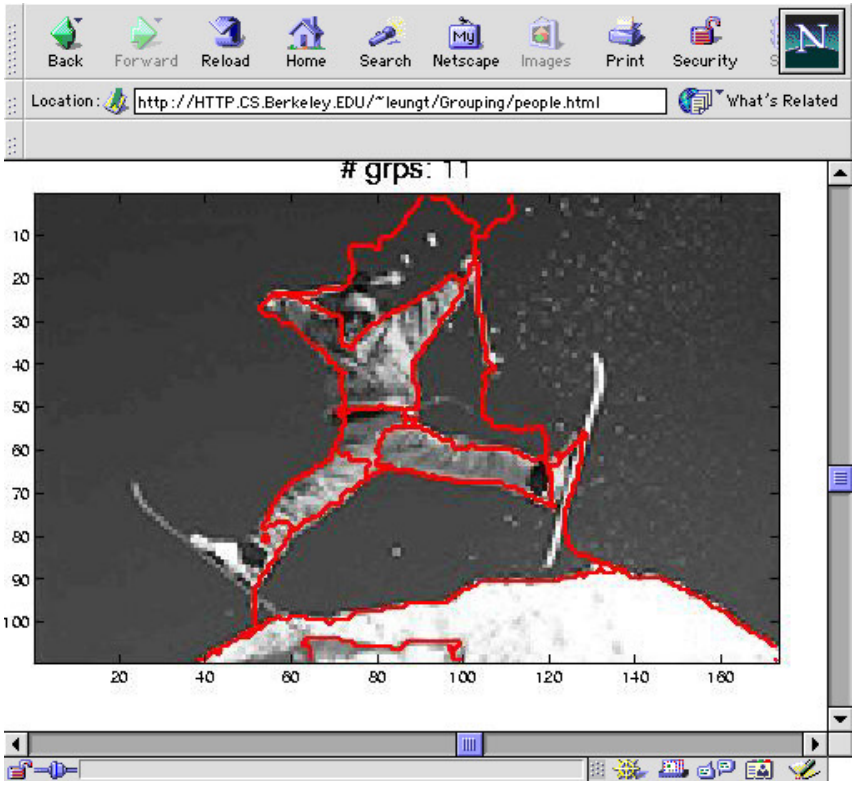
Structure from Motion



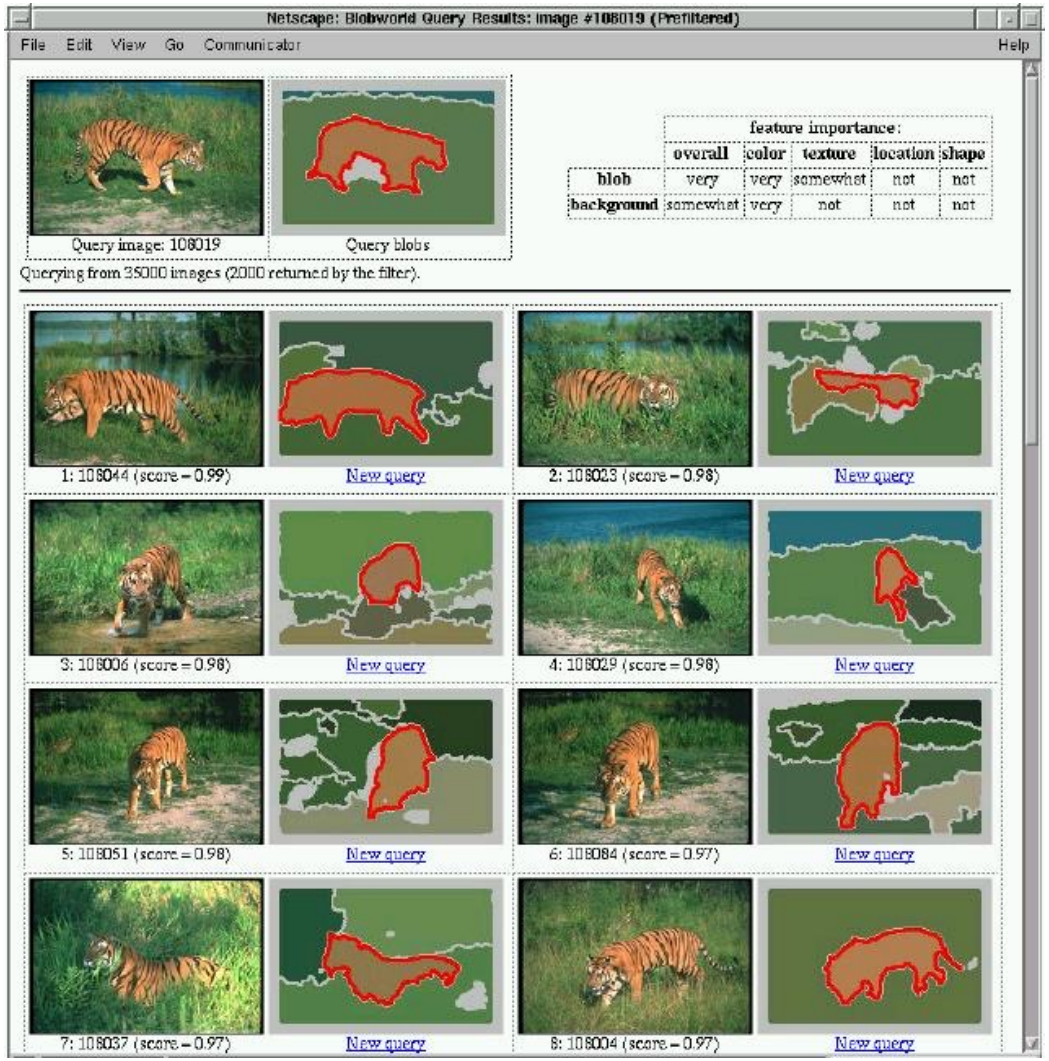
Segmentation

- Which image components "belong together"?
- Belong together = lie on the same object
- Cues
  - similar color
  - similar texture
  - not separated by contour
  - form a suggestive shape when assembled


Back
Forward
Reload
Home
Search
Netscape
Images
Print
Security
Location: http://HTTP.CS.Berkeley.EDU/~leungt/Grouping/people.html
What's Related
grps: 19
10
20
30
40
50
60
70
80
90
100
20
40
60
80
100
120
140
160

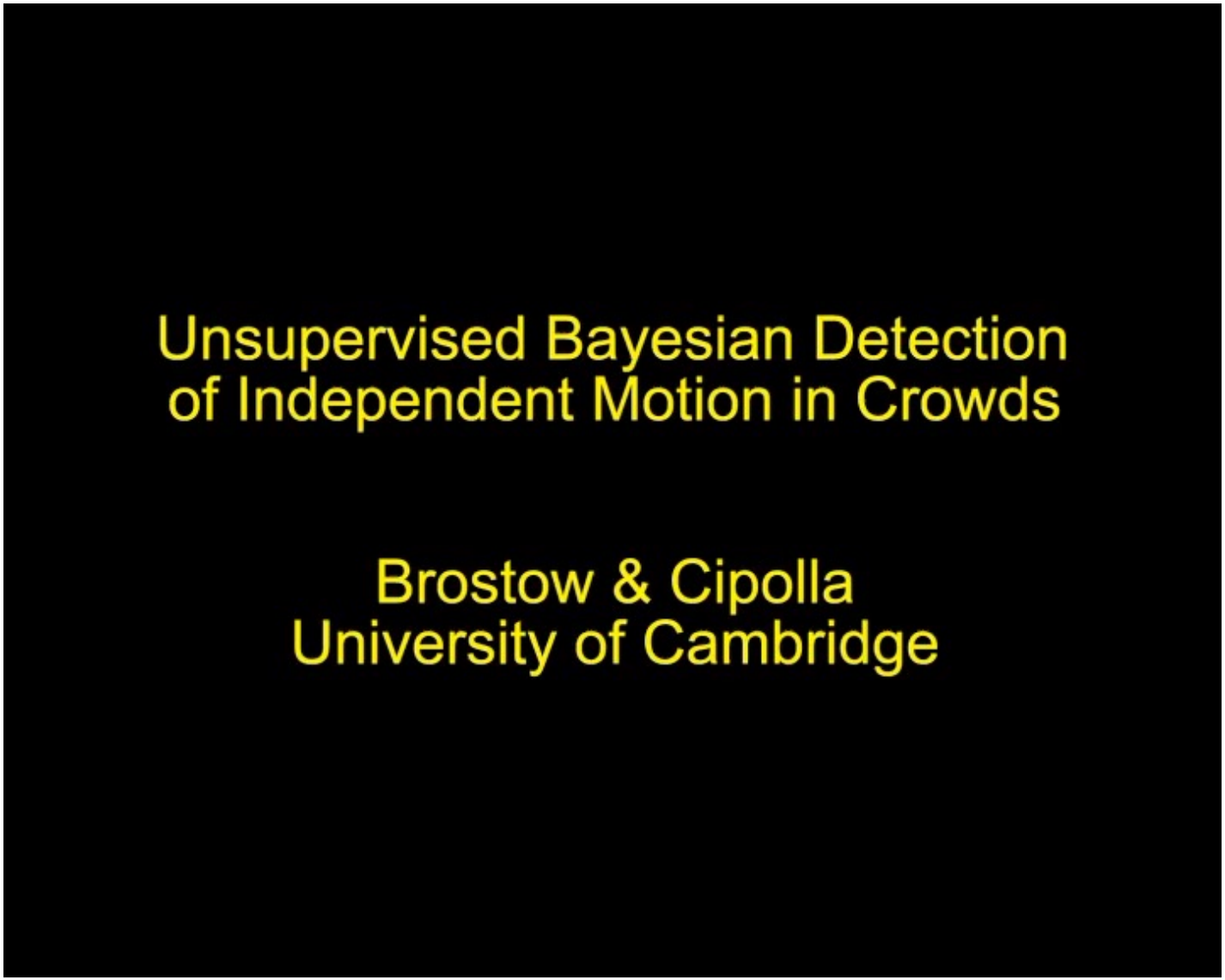

Back
Forward
Reload
Home
Search
Netscape
Images
Print
Security
Location: http://HTTP.CS.Berkeley.EDU/~leungt/Grouping/people.html
What's Related
grps: 11
10
20
30
40
50
60
70
80
90
100
20
40
60
80
100
120
140
160

Content Based Image Retrieval

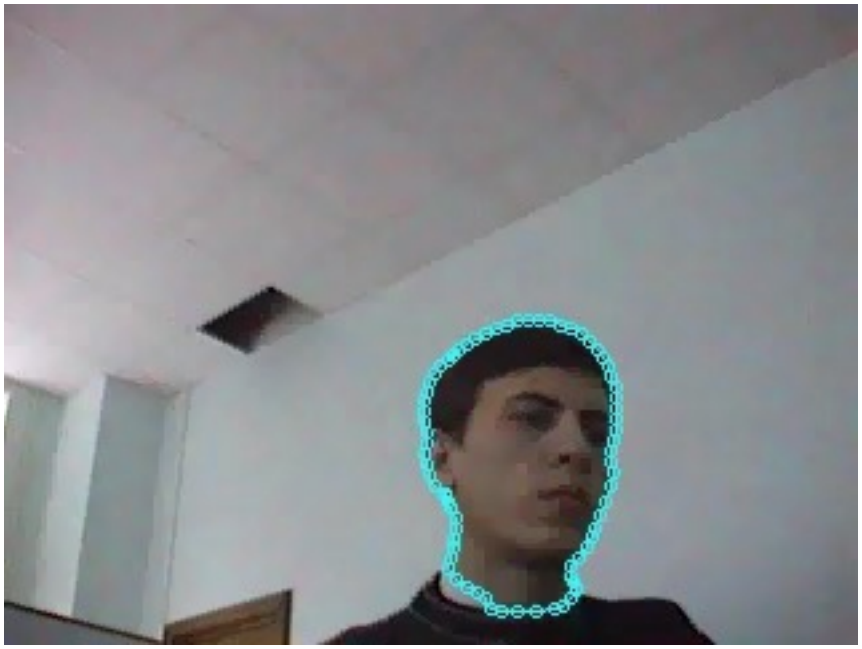


Tracking

Detection and Tracking



Tracking (Snake)



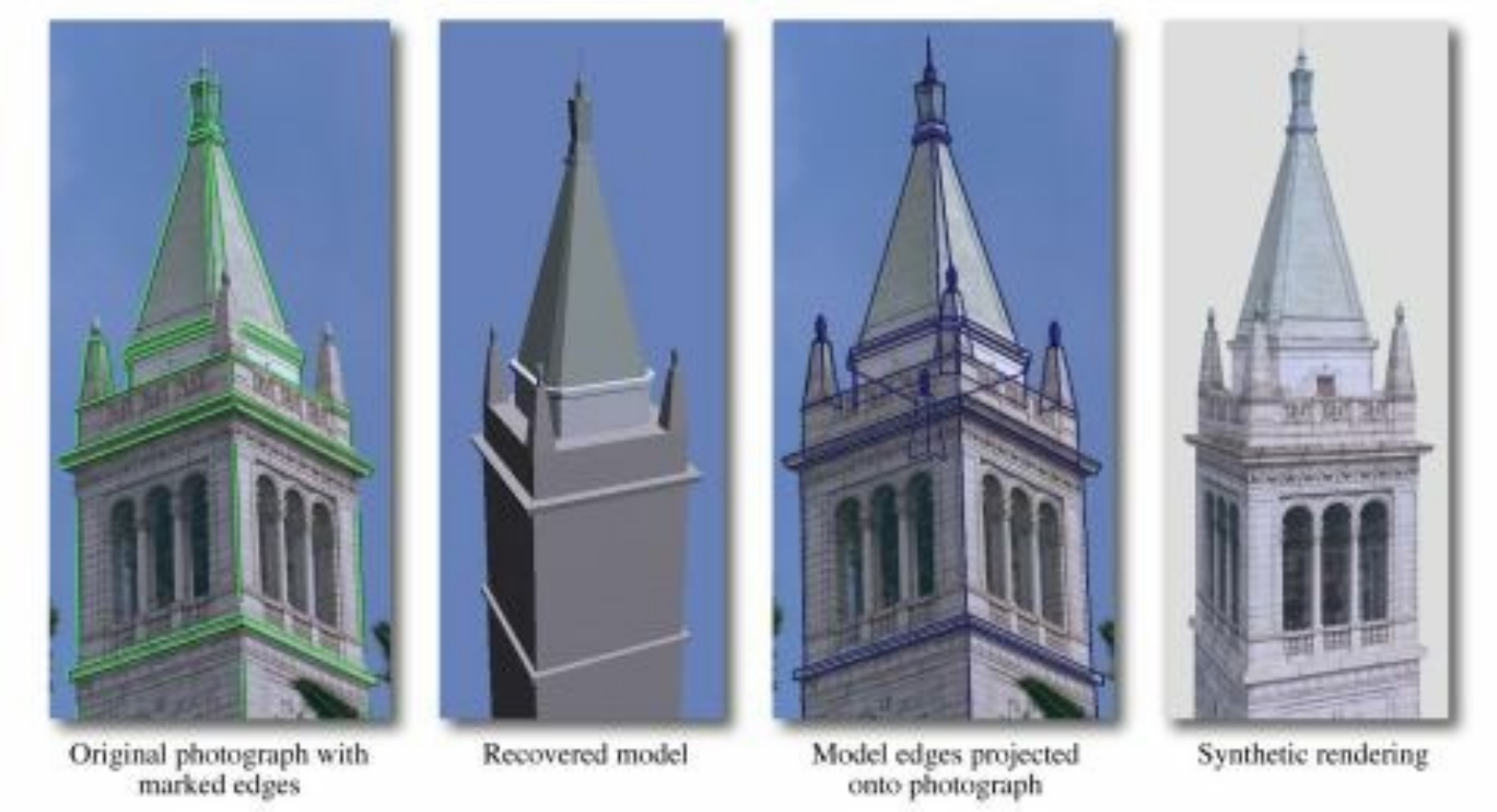
HCI

Soccer Robot

Image-Based Recognition

Surveillance

Vision for 3D Reconstruction



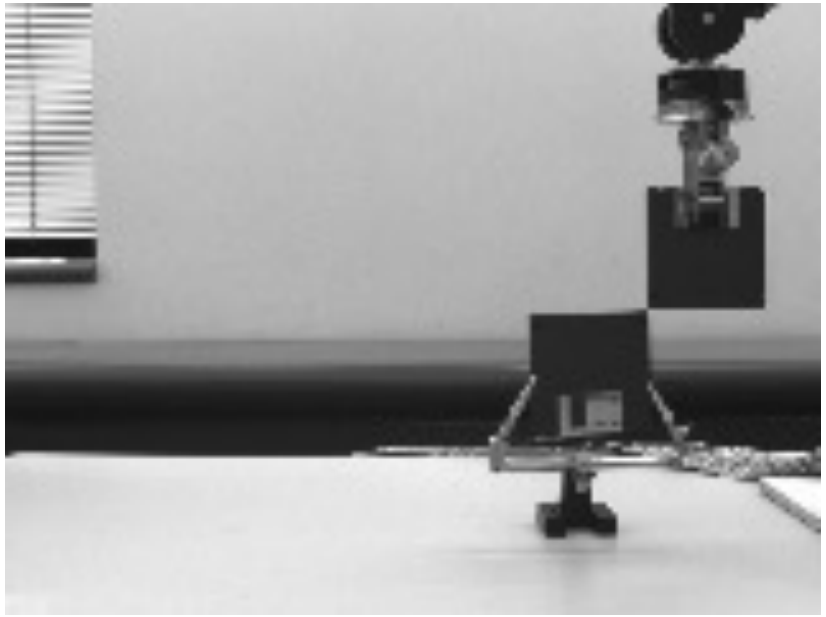
Vision for 3D Reconstruction

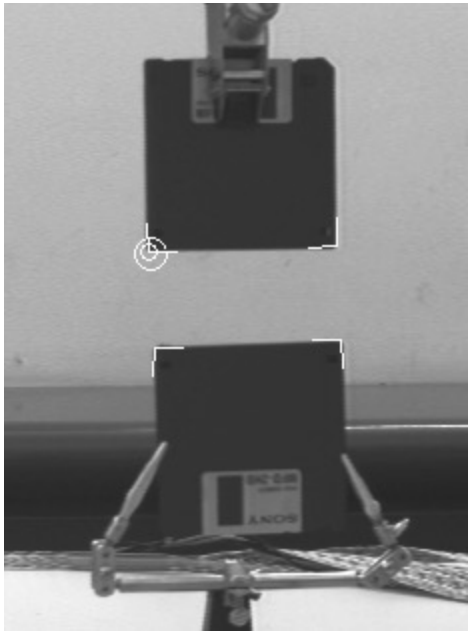
- http://www.debevec.org/Campanile/

More Reconstructions

Vision for Control

More Vision for Control





Opportunities for Computer Vision

- Thanks to Moore's law – desktop machine have gotten faster and cheaper – many already come equipped with cameras

- Computer Vision as the input device of the future – your computer will be able to:
  - Recognize you by sigh
  - Watch what you do and respond accordingly
  - Build model of the world from image data

Related Fields

- Image Processing
  - Consider image properties, image-to-image transformation
  - The task of information recovery is left to users
  - Image enhancement, image recovery, image compression, etc.
  - Emphasis in computer vision is on recovering information automatically, with minimal interaction
  - Image processing algorithms are useful in early stages of computer vision tasks (image enhancement, noise suppression, etc.)

Related Fields

- Computer Graphics
  - Methods and techniques for converting data to or from graphic displays via computer
  - To generate images from geometric primitive (line, circles, etc.)
  - Inverse problem to computer vision problems
    - Computer graphics – synthesis of images
    - Computer vision – analysis of images
  - Computer vision and computer graphics fields are merging in recent years (visualization, VR, etc.)

Related Fields

- Artificial Intelligence
  - Design intelligence systems
  - Analyze scene by computing a symbolic representation of the scene content
  - Three stages: perception, cognition, action
  - Computer vision is often considered as a subfield of artificial intelligence

- Computation Geometry
  - Algorithms for solving geometric problems on a computer

Related Fields

- Pattern Recognition
  - Recognize and classify objects using digital images
- Photogrammetry
  - Obtain reliable and accurate measurements from non-contact imaging, higher accuracy
- Robotics
- Neuroscience
- Physics
- Applied Mathematics

Application Areas

- Industrial inspection and quality control
- Reverse engineering
- Surveillance and security
- Face recognition
- Gesture recognition
- Autonomous vehicles
- Road monitoring
- Hand-eye robotics systems
- Medical image analysis (MRI, CT, X-rays, sonar scan)
- Image databases
- Virtual reality, tele-applications
- Space & Military
- Image based graphics

12/30/2022

- Final Project due TODAY.
- Project presentation starts next Tuesday (1/3 and 1/5).
- Final exam on 1/10/2023.

Stereo Vision

- What is Stereo Vision?
  - The ability to infer information on the 3D structure and distance of a scene from *two or more images* taken from different viewpoints

What Are Stereo Vision Problems?

- Correspondence problem:
  - Determining which pixel in the left image corresponds to which pixel in the right image
  - Binocular fusion of features observed by the eyes
  - Occluded regions in both images should not be matched

- Reconstruction problem:
  - Given a number of correspondence pairs and camera geometry information, find location and 3D structure of the observed objects
  - Recovering the geometry of the scene and/or relative camera positions

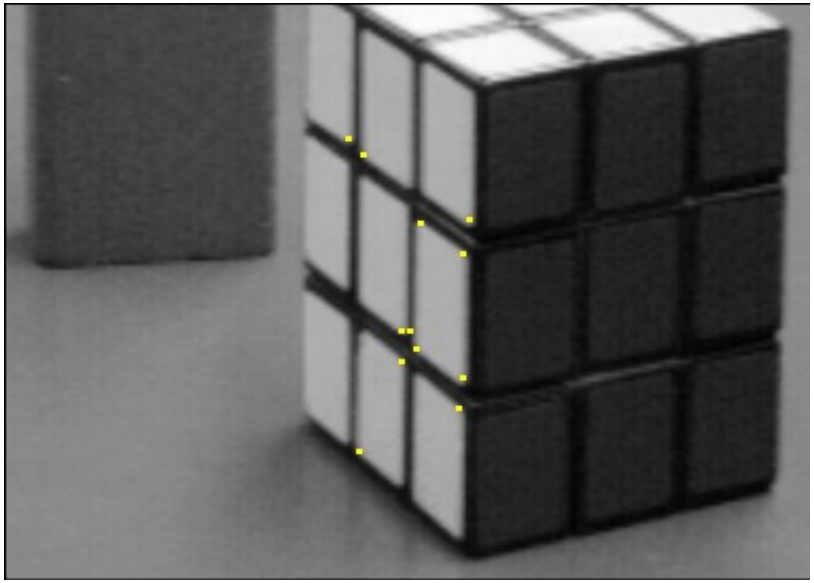
Disparity

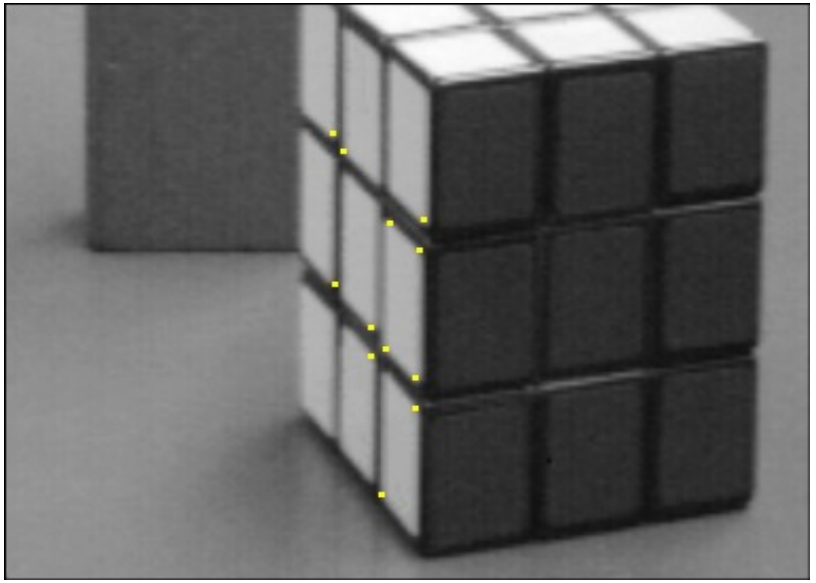
- Disparity:
  - The difference (in pixel) of the corresponding points in both images
- Disparity map:
  - An (intensity) image contains the disparity for each pixel

- If the geometry of the stereo system is known, the disparity map can be converted to a 3D map of the view scene

Example

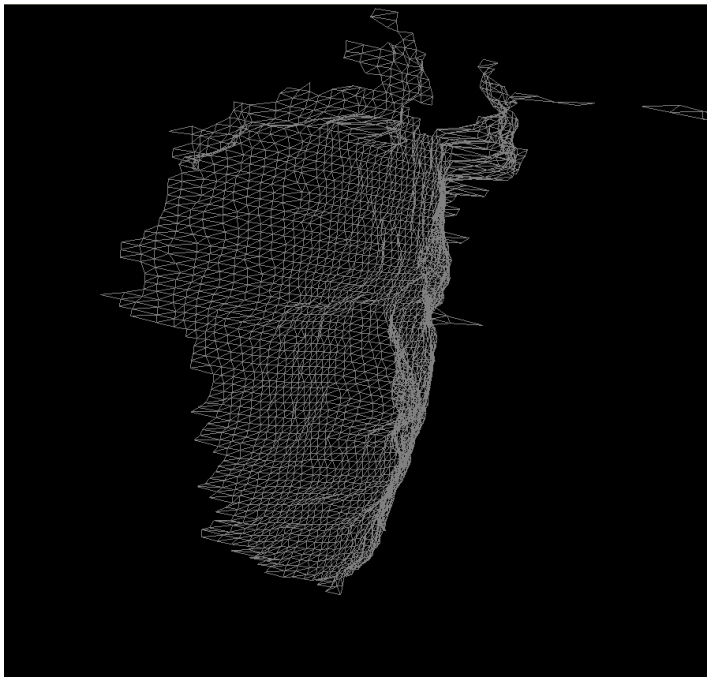
- An illustration of the correspondence problem





Example

- One image of a stereo pair and the 3-D rendering of stereo reconstruction

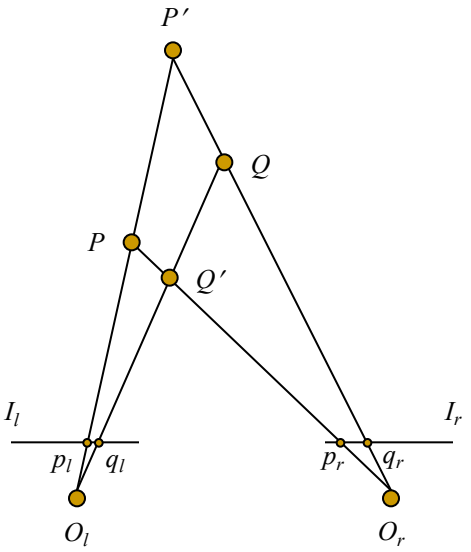


Stereo Vision System

- In a stereo system we have two images of a scene and the usual goal is to recover:
  - The 3D structure of the scene
  - The relative positions of the two cameras
  - Both of the above

A Simple Stereo System

- The top view of a stereo system with two pinhole cameras
  - Fixation point: the point of intersection of the optical axes
  - 3D reconstruction depends on the solution of the correspondence
  - Depth is estimated from the disparity of corresponding points
  - 3D position is determined by triangulation
    - P determined by p_l, p_r
    - Q determined by q_l, q_r,
    - P' determined by p_l, q_r
    - Q' determined by q_l, p_r
  - *Different pairs of corresponding points gives different depth estimations*
  - Correspondence problem is important!

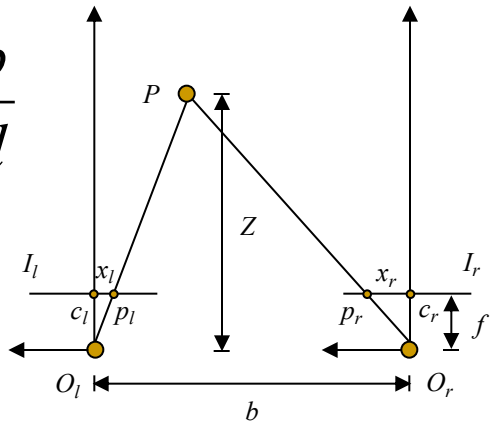


A Simple Stereo System

- Assume the correspondence problem has been solved
  - The stereo baseline b: the distance between the centers of projection O_l and O_r
  - x_l and x_r are the coordinates of p_l and p_r with respect to the principal points c_l and c_r
  - f is the focal length of the camera
  - Z is the distance between P and the baseline
- The depth of P is uniquely determined by*

$$\frac{b + x_l - x_r}{Z - f} = \frac{b}{Z}$$

- Let the disparity $d = x_r - x_l$, then* $Z = f\frac{b}{d}$
  - *Depth is inversely proportional to disparity**



Concluding Remarks

- The quantities f, b, c_l, c_r are the parameters of the stereo system, finding those values is the stereo calibration problem
- The parameters of a stereo system
  - Intrinsic parameters:
    - Characterize the transformation mapping an image point from camera to pixel coordinates, in each camera
    - The coordinates of principal point, focal length in pixel, skew factor
  - Extrinsic parameters:
    - Describe the relative position and orientation of the two cameras
    - Rotation matrix R, translation vector $\mathbf{T}$

Concluding Remarks

- In many cases the parameters are unknown, thus reconstruction is often a calibration problem
- There also exists uncalibrated stereo (epipolar geometry, etc.)*
- Stereo system with *converging* cameras, the disparity increases with the distance of the objects from the fixation point

The Correspondence Problem

- Assumptions
  - Most scene points are visible from both viewpoints
  - Corresponding image regions are similar
- The assumptions hold *if the fixation point from the cameras is much larger than the baseline*
  - Thus, the correspondence problem is a search problem:
    - Given an element in the left image, we search for the corresponding element in the right image. Two decisions:
      - *Which image element to match*? *Which similarity measure to adopt*?
    - Not all the elements of one image have necessarily a corresponding elements in the other image
- Two classes of correspondence algorithms: correlation-based and feature-based methods
  - Similar concepts, different implementations
    - All image points for correlation-based, sparse sets for feature-based

Correlation-Based Methods

- Two decisions:
  - The elements to match are image windows of fixed size
  - The similarity criterion is a measure of the correlation between windows in the two images
- The corresponding element is given by the window that *maximize the similarity criterion within a search region*