# Lecture 3
## $A^*$ Search and Search in Complex Environments

TDT4136: Introduction to Artificial Intelligence

Xavier F. C. Sánchez Díaz

Department of Computer Science
Faculty of Information Technology and Electrical Engineering
Norwegian University of Science and Technology

September 5, 2024

# Outline

# Recap on Uninformed Search

- Uninformed search strategies systematically navigate the search space blindly—not questioning where the goal may be in the space.
- The search space is often very large.

# Recap on Uninformed Search

- Uninformed search strategies systematically navigate the search space blindly—not questioning where the goal may be in the space.
- The search space is often very large.
- We can be smarter about it using a heuristic (guess estimate)
- We covered (Greedy) Best First, where you pick the option with the best estimate
- We also covered $A^*$, which uses both the cost and the estimate

# $A^*$ search

## Informed search strategies

What if we consider the cost *and* the heuristic?

# $A^*$ search

## Informed search strategies

What if we consider the cost *and* the heuristic?

Our new **heuristic function** will consider both things:

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

where

# $A^*$ search

What if we consider the cost *and* the heuristic?

Our new **heuristic function** will consider both things:

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

where

- $g(n)$ is the cost we have paid so far to reach $n$

# $A^*$ search

## Informed search strategies

What if we consider the cost *and* the heuristic?

Our new **heuristic function** will consider both things:

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

where

- $g(n)$ is the cost we have paid so far to reach $n$
- $h(n)$ is the estimated cost of the node (to the goal)

# $A^*$ search

## Informed search strategies

What if we consider the cost *and* the heuristic?

Our new **heuristic function** will consider both things:

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

where

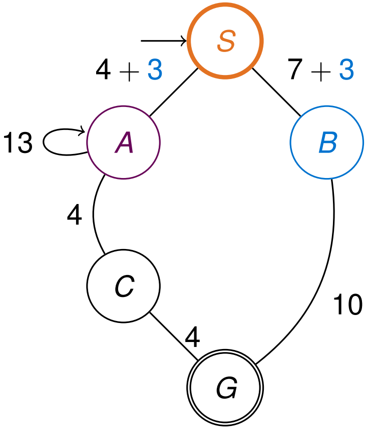
- $g(n)$ is the cost we have paid so far to reach $n$
- $h(n)$ is the estimated cost of the node (to the goal)
- $f(n)$ is then the estimated cost of the cheapest solution through $n$ to the goal

# $A^*$ search

## Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

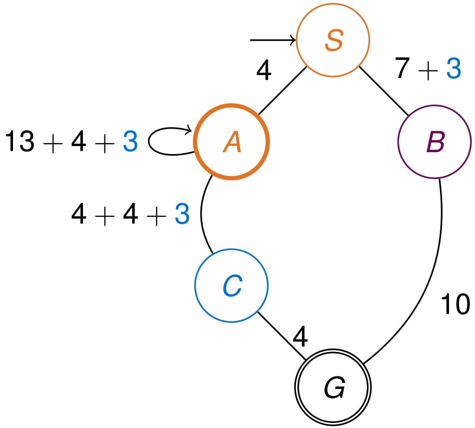


# $A^*$ search

## Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

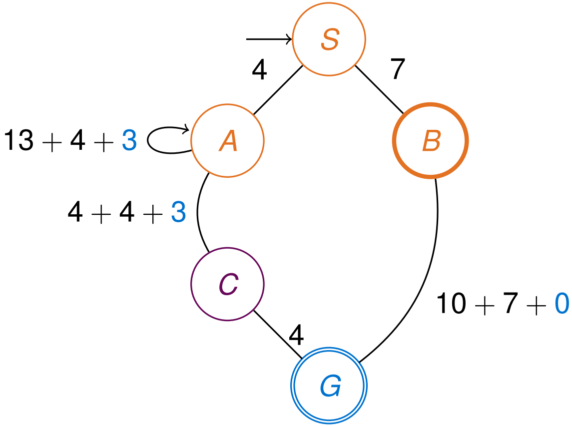


# $A^*$ search

## Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

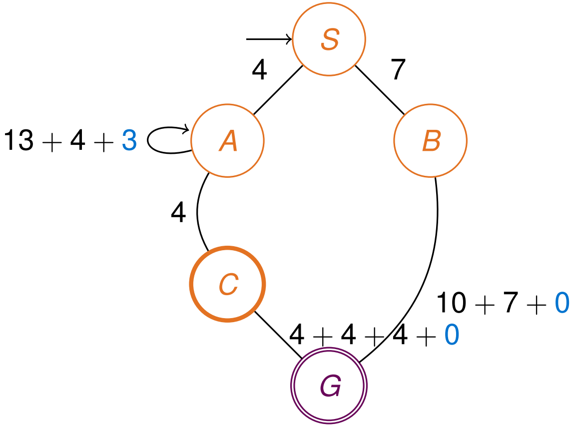


# $A^*$ search

## Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

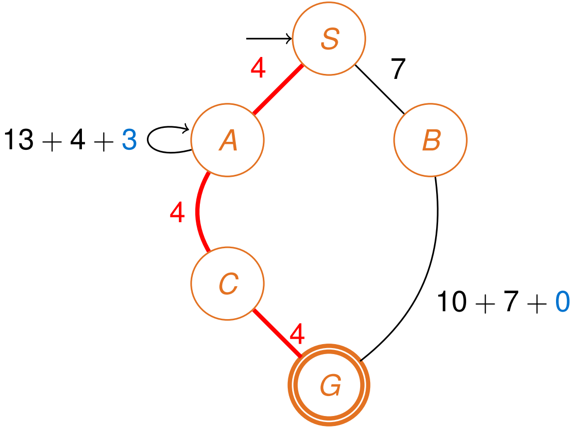


# $A^*$ search

## Informed search strategies

With the those estimated distances to the goal:

- We have found the goal!

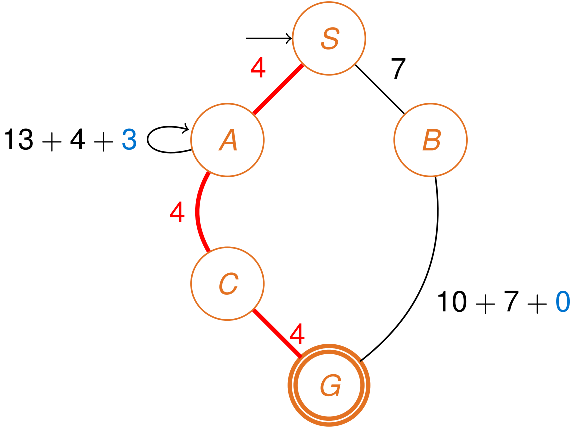


# $A^*$ search

## Informed search strategies

With the those estimated distances to the goal:

- We have found the goal!
- It is **Complete** for positive costs, within a finite state space and an existing solution.

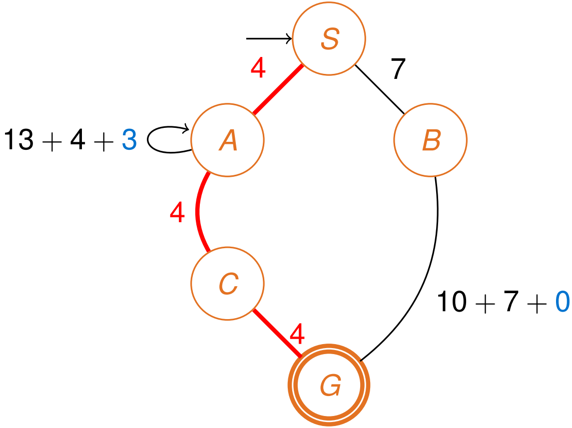


# $A^*$ search

## Informed search strategies

With the those estimated distances to the goal:

- We have found the goal!
- It is **Complete** for positive costs, within a finite state space and an existing solution.
- It is **Cost optimal** if **certain conditions are met**



# $A^*$ optimality

## More on $A^*$

$A^*$ is **cost optimal** if **certain conditions are met.** What are these conditions?

[1] They usually are.

# $A^*$ optimality
## More on $A^*$

$A^*$ is **cost optimal** if **certain conditions are met.** What are these conditions?

- Arc costs need to be positive[1]
- The heuristic function needs to be **admissible** and non-negative.

[1]They usually are.

# Admissibility

## More on $A^*$

### Admissibility of $h$

We say a heuristic $h$ is **admissible** if it **never overestimates** the cost from a node to the goal node.

# Admissibility

## More on $A^*$

### Admissibility of $h$

We say a heuristic $h$ is **admissible** if it **never overestimates** the cost from a node to the goal node.

An admissible heuristic means that for every node $n$:

- $h(n) >= 0$, and
- $h(goal) = 0$

# Admissibility

## More on $A^*$

### Admissibility of $h$

We say a heuristic $h$ is **admissible** if it **never overestimates** the cost from a node to the goal node.

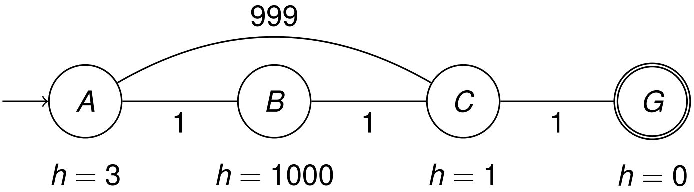
An admissible heuristic means that for every node $n$:

- $h(n) >= 0$, and
- $h(goal) = 0$

An admissible heuristic is *optimistic!*

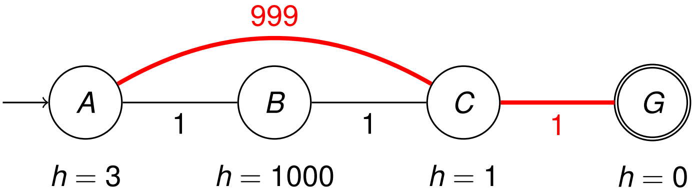
# A crazy example

## More on $A^*$



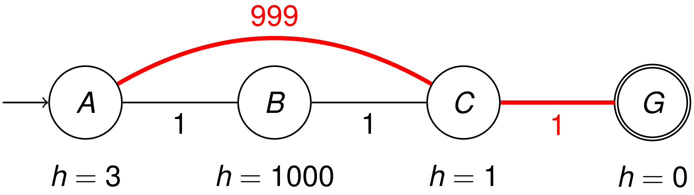
# A crazy example

## More on $A^*$



# A crazy example

## More on $A^*$




We would not choose the optimal path due to $h(B)$ being overestimated of the actual cost!

# Consistency

## More on $A^*$

Another important (and even stronger) property of a heuristic $h$ is **consistency**.

### Consistency of $h$

A heuristic $h$ is **consistent** if for every node $n$ and <u>all</u> of its successors $n'$ generated by an action $a$, we have

$$h(n) \leq c(n, a, n') + h(n')$$

# Consistency

## More on $A^*$

Another important (and even stronger) property of a heuristic $h$ is **consistency**.

### Consistency of $h$

A heuristic $h$ is **consistent** if for every node $n$ and <u>all</u> of its successors $n'$ generated by an action $a$, we have

$$h(n) \leq c(n, a, n') + h(n')$$

In other words, the estimate of a node should be less or equal than the the estimate of a descendant plus the cost of reaching there.

# Consistency: an example

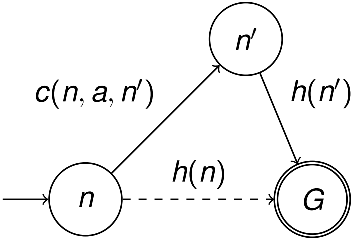
## More on $A^*$

### Consistency of $h$

A heuristic $h$ is **consistent** if for every node $n$ and all of its successors $n'$ generated by an action $a$, we have

$$h(n) \leq c(n, a, n') + h(n')$$

- A triangle inequality helps picturing it!
- Moving through $h(n)$ **has** to be cheaper than going to $G$ via the successor $n'$
- This **must** be true for every successor $n'$ of $n$
  - Think of an euclidean grid



# Consistency and admissibility

## More on $A^*$

Why is this important?

- A heuristic that is **consistent** is **always** **admissible**

# Consistency and admissibility

## More on $A^*$

Why is this important?

- A heuristic that is **consistent** is **always** **admissible**
  - Not necessarily the other way around!

# Consistency and admissibility

## More on $A^*$

Why is this important?

- A heuristic that is **consistent** is **always** **admissible**
  - Not necessarily the other way around!
- Since a consistent heuristic is admissible, then a consistent heuristic is also **always optimal**

# Consistency and admissibility

## More on $A^*$

Why is this important?

- A heuristic that is **consistent** is **always** **admissible**
  - Not necessarily the other way around!
- Since a consistent heuristic is admissible, then a consistent heuristic is also **always optimal**
- A consistent heuristic $h(n)$ ensures that the cost function $f(n) = g(n) + h(n)$ is **monotonic nondecreasing**

# Consistency and admissibility

## More on $A^*$

Why is this important?

- A heuristic that is **consistent** is **always** **admissible**
  - Not necessarily the other way around!
- Since a consistent heuristic is admissible, then a consistent heuristic is also **always optimal**
- A consistent heuristic $h(n)$ ensures that the cost function $f(n) = g(n) + h(n)$ is **monotonic nondecreasing**
  - That means that $f(n)$ is non-decreasing along any path

# Optimality and efficiency

## More on $A^*$

- $A^*$ is **optimally efficient** with a consistent heuristic
- This means that any other search algorithm with the same heuristic values must expand all nodes that $A^*$ expanded

# Optimality and efficiency

## More on $A^*$

- $A^*$ is **optimally efficient** with a consistent heuristic
- This means that any other search algorithm with the same heuristic values must expand all nodes that $A^*$ expanded

However, the main issue of $A^*$ lies on its memory use. Some ways to reduce it:

- Reference count – remove a state from *reached* when there are no more ways to reach it

# Optimality and efficiency

## More on $A^*$

- $A^*$ is **optimally efficient** with a consistent heuristic
- This means that any other search algorithm with the same heuristic values must expand all nodes that $A^*$ expanded

However, the main issue of $A^*$ lies on its memory use. Some ways to reduce it:

- Reference count – remove a state from *reached* when there are no more ways to reach it
- Beam search – limit size of *frontier* to $k$-best candidates

# Optimality and efficiency

## More on $A^*$

- $A^*$ is **optimally efficient** with a consistent heuristic
- This means that any other search algorithm with the same heuristic values must expand all nodes that $A^*$ expanded

However, the main issue of $A^*$ lies on its memory use. Some ways to reduce it:

- Reference count – remove a state from *reached* when there are no more ways to reach it
- Beam search – limit size of *frontier* to $k$-best candidates
- Iterative deepening $A^*$ – gradually increase the $f$-cost *cutoff*.

# Optimality and efficiency

## More on $A^*$

- $A^*$ is **optimally efficient** with a consistent heuristic
- This means that any other search algorithm with the same heuristic values must expand all nodes that $A^*$ expanded

However, the main issue of $A^*$ lies on its memory use. Some ways to reduce it:

- Reference count – remove a state from *reached* when there are no more ways to reach it
- Beam search – limit size of *frontier* to $k$-best candidates
- Iterative deepening $A^*$ – gradually increase the $f$-cost *cutoff*.
- Memory-bounded $A^*$ – expand until memory is ful, and then drop the worst candidate from *frontier*

# A generalised heuristic search

## More on $A^*$

### Generalised heuristic search

$$f(n) = g(n) + w \cdot h(n)$$

where $w$ is a *weight* defining how important the heuristic $h(n)$ is.

In most other applications, we usually have $w_1$ and $w_2$, one for $g(n)$ and one for $h(n)$. The book uses only $w$ for $h(n)$.

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
    - Choose the cheapest!
    - This is called **uniform-cost search** and it's an uninformed search.
    - It is also known as Dijkstra's algorithm.
- With $w = \infty$ you only care about the estimate

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.
- With $w = \infty$ you only care about the estimate
  - Choose the one that *seems* the cheapest

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.
- With $w = \infty$ you only care about the estimate
  - Choose the one that *seems* the cheapest
  - This is Greedy Best-First search

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.
- With $w = \infty$ you only care about the estimate
  - Choose the one that *seems* the cheapest
  - This is Greedy Best-First search
- with $w = 1$ you care equally about the path cost and the estimates

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.
- With $w = \infty$ you only care about the estimate
  - Choose the one that *seems* the cheapest
  - This is Greedy Best-First search
- with $w = 1$ you care equally about the path cost and the estimates
  - This is $A^*$

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.
- With $w = \infty$ you only care about the estimate
  - Choose the one that *seems* the cheapest
  - This is Greedy Best-First search
- with $w = 1$ you care equally about the path cost and the estimates
  - This is $A^*$

# A generalised heuristic search

## More on $A^*$

$$f(n) = g(n) + w \cdot h(n)$$

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!
  - This is called **uniform-cost search** and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.
- With $w = \infty$ you only care about the estimate
  - Choose the one that *seems* the cheapest
  - This is Greedy Best-First search
- with $w = 1$ you care equally about the path cost and the estimates
  - This is $A^*$

Of course you can set $w$ to something else, depending for example if there is *uncertainty* on your heuristic (but this then becomes a whole other course :^))

# Building heuristics

## More on $A^*$


How far are we from solving this sliding puzzle?

- $h_1(n)$ will be the number of misplaced tiles
- $h_2(n)$ will be the **total** Manhatttan distance[a]

[a] number of squares away from the desired location

# Building heuristics

## More on $A^*$


How far are we from solving this sliding puzzle?

- $h_1(n)$ will be the number of misplaced tiles
- $h_2(n)$ will be the **total** Manhatttan distance[a]

[a] number of squares away from the desired location

Remember that each configuration is a state!

# Other ideas for building heuristics

## More on $A^*$

- Consider relaxations of the problem
- Consider creating the heuristic by looking **backwards from the goal**.
- Consider dividing into subproblems!
  - For example, instead of solving the whole sliding puzzle at once, consider getting in place four tiles only
  - Then store all these solutions in a DB. Create an admissible heuristic for this subproblem
  - Combine the subproblems to choose the best heuristic

The process of choosing the appropriate representation, data structures and heuristics for a problem is known as **modelling** and is crucial for AI developers and researchers!

# Dominance: comparing heuristics



**Which of the heuristics is better?**

# Dominance: comparing heuristics

## More on $A^*$

**Which of the heuristics is better?**
Admissible heuristics can be compared by looking at their values.

### Heuristic Domination

An admissible heuristic $h_2$ it is said to **dominate** another admissible heuristic $h_1$ if **for all** nodes $n$ if $h_2(n) \geq h_1(n)$.
This will reflect in $A^*$ expanding fewer nodes on $h_2$, and thus find an optimal solution, faster.

# Dominance: comparing heuristics

## More on $A^*$

**Which of the heuristics is better?**
Admissible heuristics can be compared by looking at their values.

### Heuristic Domination

An admissible heuristic $h_2$ it is said to **dominate** another admissible heuristic $h_1$ if **for all** nodes $n$ if $h_2(n) \geq h_1(n)$.
This will reflect in $A^*$ expanding fewer nodes on $h_2$, and thus find an optimal solution, faster.

A generalisation of this would then be

$$h_{best}(n) = \max(h_a(n), h_b(n), \ldots)$$

Section 3

# Search in Complex Environments

# Searching in complex environments

- Both informed and uninformed searching strategies are designed to explore search spaces systematically
- They keep one or more paths in memory, and record which alternatives have been explored at each point along the path
- The path to that goal constitutes a solution
- But in most problems in the real world, the path to a solution might be **irrelevant**

# Searching in complex environments

- Both informed and uninformed searching strategies are designed to explore search spaces systematically
- They keep one or more paths in memory, and record which alternatives have been explored at each point along the path
- The path to that goal constitutes a solution
- But in most problems in the real world, the path to a solution might be **irrelevant**

If we only care about finding a solution, then there are better ways to search the space!

# Local Search

- It uses a single **current node** and move to **neighbouring** nodes

[2] as in most real world applications

# Local Search

- It uses a single **current node** and move to **neighbouring** nodes
- It eases up on the completeness and optimality in the interest of improving time and space complexity[2]

[2] as in most real world applications

# Local Search

- It uses a single **current node** and move to **neighbouring** nodes
- It eases up on the completeness and optimality in the interest of improving time and space complexity[2]
- Local Search algorithms use “little” memory (usually a constant amount)

[2] as in most real world applications

# Local Search

- It uses a single **current node** and move to **neighbouring** nodes
- It eases up on the completeness and optimality in the interest of improving time and space complexity[2]
- Local Search algorithms use “little” memory (usually a constant amount)
- They can often find reasonable solutions in very large (or infinite) state spaces

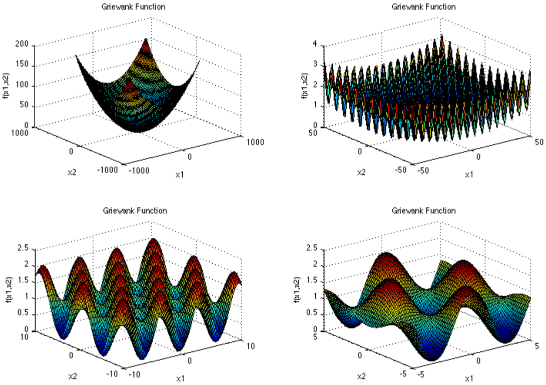
[2] as in most real world applications

# The search landscape

## Search in complex environments

Usually, the **state space** is referred to as the **search space**. We can **visualise** this space by looking at the heuristic function!

$$f(x) = \sum_{i=1}^{d} \frac{x^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

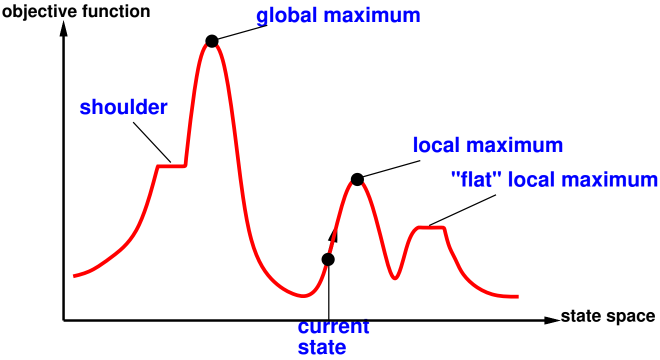



The Griewank function. Image from Surjanovic & Bingham

https://www.sfu.ca/~ssurjano/griewank.html

# The search landscape

## Search in complex environments

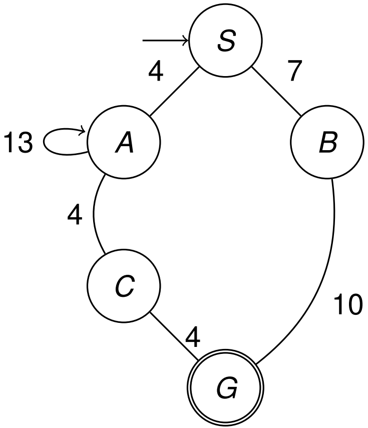



- Each point in the landscape represents a state in the search space and has “an elevation” (its $h(n)$)
- If the elevation corresponds to an objective function, then the aim is to find the highest peak (or **maximum**)
- If the elevation corresponds to a cost function, then we look for the lowest valley (or **minimum**)

# The search landscape

## Search in complex environments

Recall our search problems.




- $A$ is a neighbour of $S$, $C$ and itself because those are the states than can be reached from $A$.
- The **neighbourhood** of $A$ is then $\{A, C, S\}$.
- This concept of neighbourhood is very important for **local search**, as we decide *where to move next* by looking around us!

Section 4

# Local Search Algorithms

As the last time with algorithms, please check the full details on the book!

# Hill climbing and gradient descent

## Local search algorithms

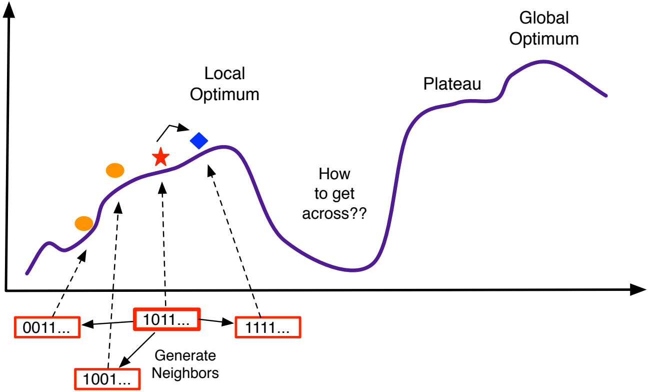
Idea: Go to the best spot you see now.

# Hill climbing and gradient descent

## Local search algorithms

Idea: Go to the best spot you see now.

- Assume you are doing **maximisation**
- You then want to **climb** the tallest peak
- This is called **hill-climbing**!




If you are **minimising** instead, then the procedure is called **gradient descent** as we want to move towards the direction where the difference in "height" is largest.

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions
  - This is the key to the **simulated annealing** algorithm

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches
  - This is the key idea behind **population-based optimisation**

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches
  - This is the key idea behind **population-based optimisation**
- Idea 4: Increase the **neighbourhood** *size*

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches
  - This is the key idea behind **population-based optimisation**
- Idea 4: Increase the **neighbourhood** *size*
  - For example, consider 2-moves-away adjacency instead

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches
  - This is the key idea behind **population-based optimisation**
- Idea 4: Increase the **neighbourhood** *size*
  - For example, consider 2-moves-away adjacency instead
- Idea 5: Jump!

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such “bad” decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches
  - This is the key idea behind **population-based optimisation**
- Idea 4: Increase the **neighbourhood** *size*
  - For example, consider 2-moves-away adjacency instead
- Idea 5: Jump!
  - Either via *long* jumps when you are not doing very good

This line of research is usually referred to as **metaheuristics.**

# How to get across?

Both **hill-climbing** and **gradient descent** get stuck in **local optima**. How do we get out of this mess?

- Idea: take some *not so good* decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such "bad" decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches
  - This is the key idea behind **population-based optimisation**
- Idea 4: Increase the **neighbourhood** *size*
  - For example, consider 2-moves-away adjacency instead
- Idea 5: Jump!
  - Either via *long* jumps when you are not doing very good
  - Or doing *short* hops when you are in a promising state (you do not want to miss it)

This line of research is usually referred to as **metaheuristics.**

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

1. Start with a population of $k$ randomly generated states

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

1. Start with a population of $k$ randomly generated states
2. Randomly choose two parent states weighted by their fitness (objective function)

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

1. Start with a population of $k$ randomly generated states
2. Randomly choose two parent states weighted by their fitness (objective function)
3. Generate child states by combining parent states randomly

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

1. Start with a population of $k$ randomly generated states
2. Randomly choose two parent states weighted by their fitness (objective function)
3. Generate child states by combining parent states randomly
4. Add child states to the population

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

1. Start with a population of $k$ randomly generated states
2. Randomly choose two parent states weighted by their fitness (objective function)
3. Generate child states by combining parent states randomly
4. Add child states to the population
5. Replace the old population by the new

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

1. Start with a population of $k$ randomly generated states
2. Randomly choose two parent states weighted by their fitness (objective function)
3. Generate child states by combining parent states randomly
4. Add child states to the population
5. Replace the old population by the new

# Genetic Algorithms

## Local search algorithms

A well-known metaheuristic in the family of **population-based** optimisers is the **genetic algorithm**.

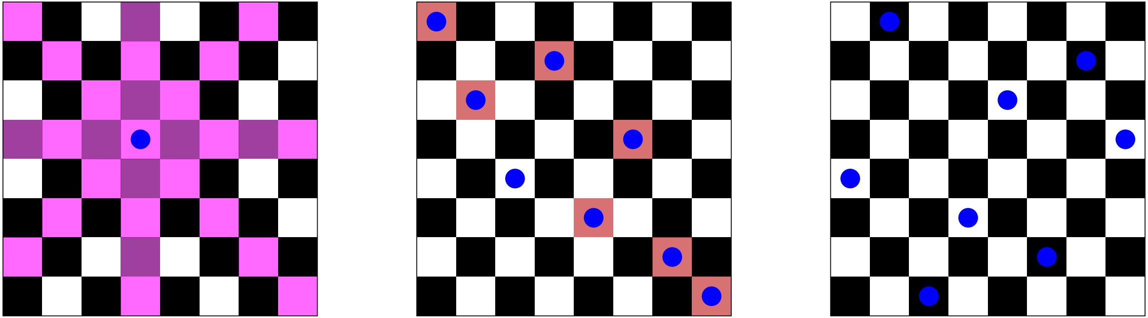
1. Start with a population of *k* randomly generated states
2. Randomly choose two parent states weighted by their fitness (objective function)
3. Generate child states by combining parent states randomly
4. Add child states to the population
5. Replace the old population by the new

This process will be repeated until a solution has been found, or until enough *generations* have been replaced.

We have a whole course on evolutionary computation methods during the spring semester: IT3708 Bio-Inspired AI!

# The 8-queens problem

Place 8 queens in a chess board such that no queen checks each other.



(a) Queen constraints (b) Generated conflicts (c) Possible solution

Figure: The 8-queens problem. 1a shows the constraints (in pink) imposed by the placement of a single queen piece (in blue). 1b highlights the conflicts arising from a possible configuration of the board. 1c illustrates one possible solution with no conflicts.

See a worked example in `https://ntnu-ai-lab.github.io/EvoLP.jl/stable/tuto/8_queens.html`

Section 5

# Nondeterministic and partially observable environments

# Searching with Nondeterminism

- So far, we have assumed that actions are deterministic

# Searching with Nondeterminism

- So far, we have assumed that actions are deterministic
  - That our intended action will **always** yield the result we expect

# Searching with Nondeterminism

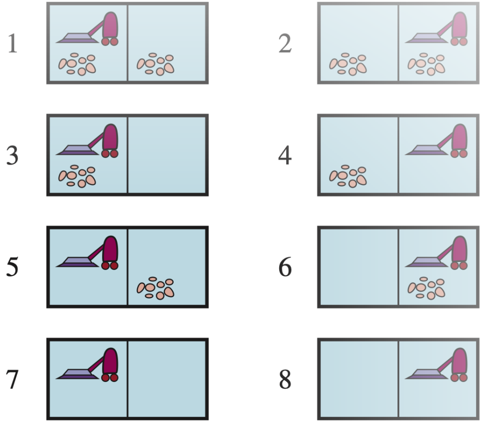
- So far, we have assumed that actions are deterministic
  - That our intended action will **always** yield the result we expect
- In the real-world, things do not always go as expected

# Searching with Nondeterminism

- So far, we have assumed that actions are deterministic
  - That our intended action will **always** yield the result we expect
- In the real-world, things do not always go as expected
- To account for different possible outcomes, we need to come up with a contingency plan instead of a single path of actions

# Example: The Erratic Vacuum World

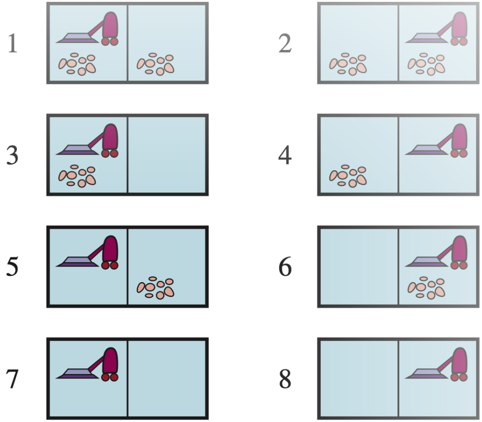
## Searching with Nondeterminism




- Nondeterministic *suck* action:
  $suck(s_1) = \{s_5, s_7\}$

# Example: The Erratic Vacuum World

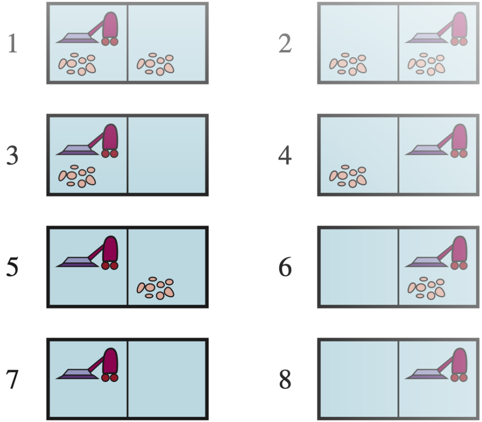
## Searching with Nondeterminism




- Nondeterministic *suck* action:
  $suck(s_1) = \{s_5, s_7\}$
  - which means both states $s_5$ and $s_7$ are possible outcomes of executing a *suck* action on state $s_1$

# Example: The Erratic Vacuum World

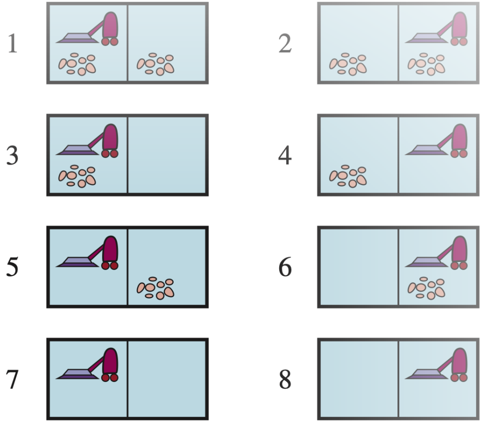
## Searching with Nondeterminism




- Nondeterministic *suck* action: $\mathit{suck}(s_1) = \{s_5, s_7\}$
  - which means both states $s_5$ and $s_7$ are possible outcomes of executing a *suck* action on state $s_1$
- $\mathit{suck}(s_7) = \{s_3, s_7\}$

# Example: The Erratic Vacuum World

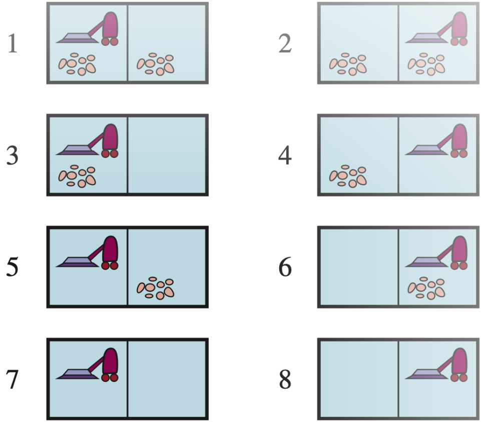
## Searching with Nondeterminism




- Nondeterministic *suck* action: $suck(s_1) = \{s_5, s_7\}$
  - which means both states $s_5$ and $s_7$ are possible outcomes of executing a *suck* action on state $s_1$
- $suck(s_7) = \{s_3, s_7\}$
  - Which means both $s_3$ and $s_7$ are possible outcomes of *suck* on $s_7$

# Example: The Erratic Vacuum World

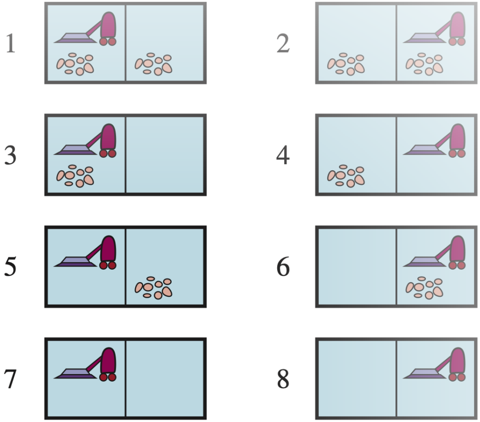
## Searching with Nondeterminism




- Nondeterministic *suck* action: $suck(s_1) = \{s_5, s_7\}$
  - which means both states $s_5$ and $s_7$ are possible outcomes of executing a *suck* action on state $s_1$
- $suck(s_7) = \{s_3, s_7\}$
  - Which means both $s_3$ and $s_7$ are possible outcomes of *suck* on $s_7$

# Example: The Erratic Vacuum World

## Searching with Nondeterminism




- Nondeterministic *suck* action: $suck(s_1) = \{s_5, s_7\}$
  - which means both states $s_5$ and $s_7$ are possible outcomes of executing a *suck* action on state $s_1$
- $suck(s_7) = \{s_3, s_7\}$
  - Which means both $s_3$ and $s_7$ are possible outcomes of *suck* on $s_7$

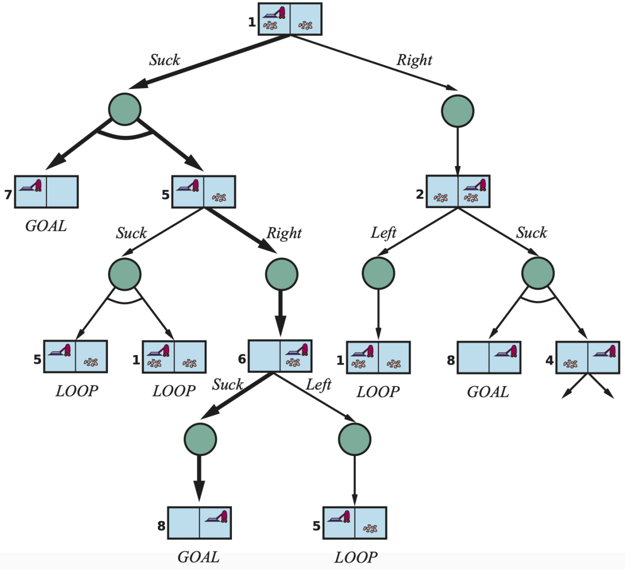
Nondeterminism can happen with other actions like *moveRight*! See the *slippery vacuum world* in the book!

# AND-OR search trees

## Searching with Nondeterminism

One way to handle these, is to consider *compound nodes*, made up of the possible states after a given action

- **OR** nodes represent actions
- **AND** nodes represent outcomes
- Since it is a **tree**, we can **search** in it
  - This is called **AND-OR search**
  - It is recursive, with a base case of either failure or an empty plan



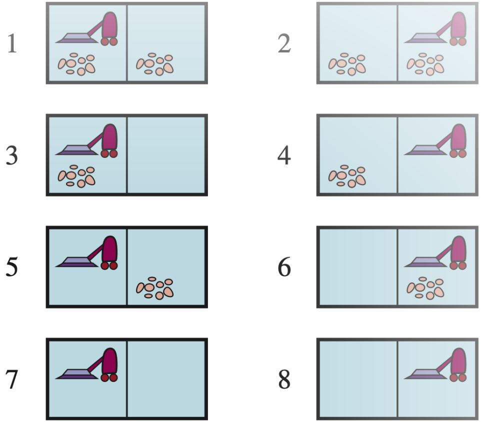
# Searching in Partially Observable Environments

- So far, we have assumed that the agent knows exactly the state of its environment
- In reality, an agent receives partial (and possibly noisy) observations
- Therefore, the state can only be *estimated* through a “belief”

# Sensorless deterministic vacuum world

## Searching in partially observable environments

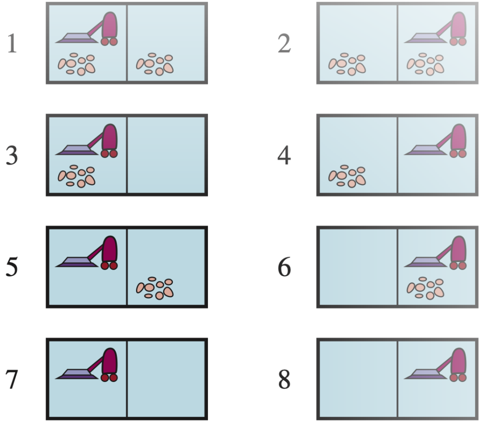
- $$Result(\{1,2,3,4,5,6,7,8\}, moveRight) = \{2,4,6,8\}$$



# Sensorless deterministic vacuum world

## Searching in partially observable environments

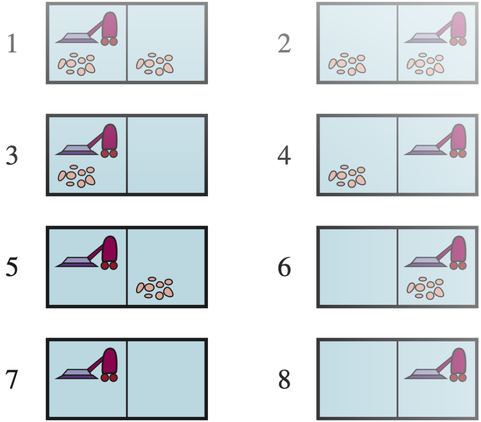
- $\mathit{Result}(\{1, 2, 3, 4, 5, 6, 7, 8\}, \mathit{moveRight}) = \{2, 4, 6, 8\}$
  - Which means that executing *moveRight* on any state $s \in S$ will yield a result in $\{2, 4, 6, 8\}$



# Sensorless deterministic vacuum world

## Searching in partially observable environments

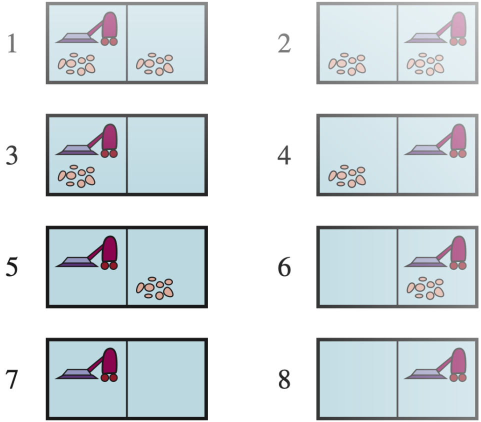
- $$Result(\{1,2,3,4,5,6,7,8\}, moveRight) = \{2,4,6,8\}$$
  - Which means that executing *moveRight* on any state $s \in S$ will yield a result in $\{2,4,6,8\}$
- $Result(\{2,4,6,8\}, Suck) = \{4,8\}$



# Sensorless deterministic vacuum world

## Searching in partially observable environments

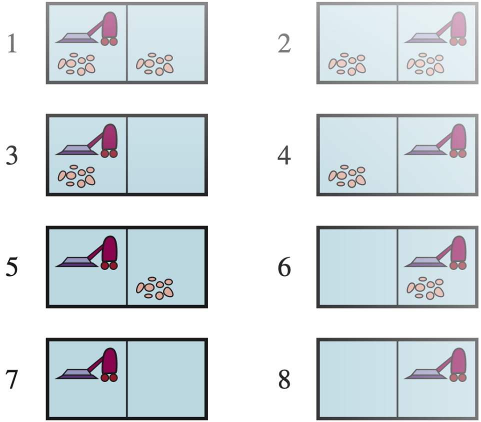
- $Result(\{1, 2, 3, 4, 5, 6, 7, 8\}, moveRight) = \{2, 4, 6, 8\}$
  - Which means that executing *moveRight* on any state $s \in S$ will yield a result in $\{2, 4, 6, 8\}$
- $Result(\{2, 4, 6, 8\}, Suck) = \{4, 8\}$
- $Result(\{4, 8\}, Left) = \{1, 7\}$



# Sensorless deterministic vacuum world

## Searching in partially observable environments

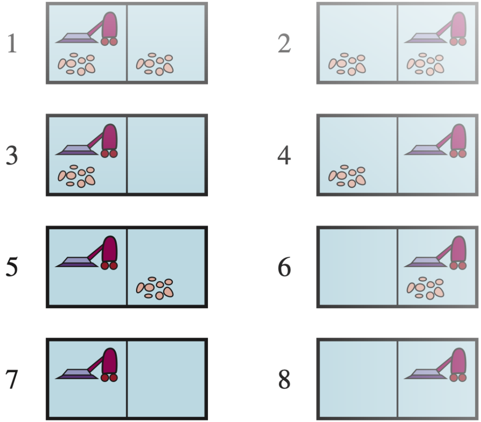
- $Result(\{1,2,3,4,5,6,7,8\}, moveRight) = \{2,4,6,8\}$
  - Which means that executing *moveRight* on any state $s \in S$ will yield a result in $\{2,4,6,8\}$
- $Result(\{2,4,6,8\}, Suck) = \{4,8\}$
- $Result(\{4,8\}, Left) = \{1,7\}$
- $Result(\{1,7\}, Suck) = \{7\}$



# Sensorless deterministic vacuum world

## Searching in partially observable environments

- $Result(\{1, 2, 3, 4, 5, 6, 7, 8\}, moveRight) = \{2, 4, 6, 8\}$
  - Which means that executing *moveRight* on any state $s \in S$ will yield a result in $\{2, 4, 6, 8\}$
- $Result(\{2, 4, 6, 8\}, Suck) = \{4, 8\}$
- $Result(\{4, 8\}, Left) = \{1, 7\}$
- $Result(\{1, 7\}, Suck) = \{7\}$

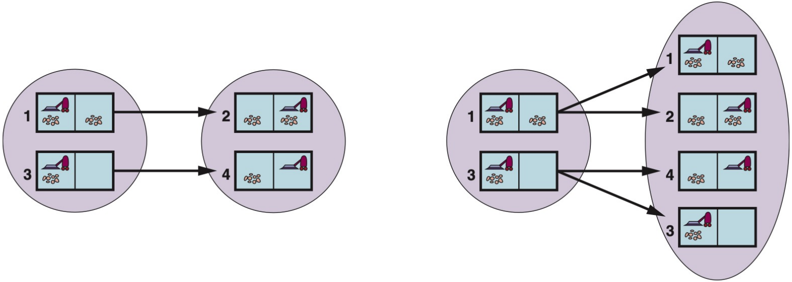



Think of 5D-chess: you solve the problem on multiple paths at the same time!

# Predicting the next state with sensorless agents

## Searching in partially observable environments

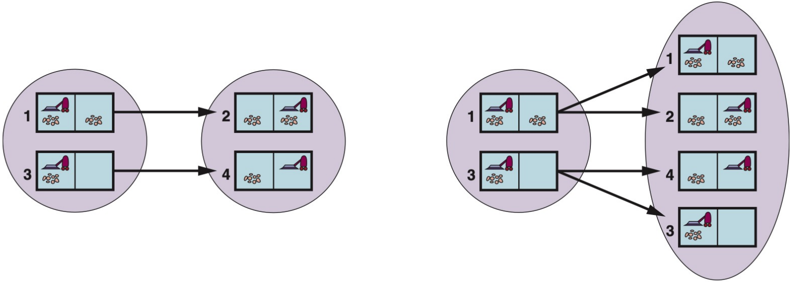
We are, in a way, making compound nodes with multiple outcomes in, where some of our actions lead to specific environment settings inside those belief states.



# Predicting the next state with sensorless agents

## Searching in partially observable environments

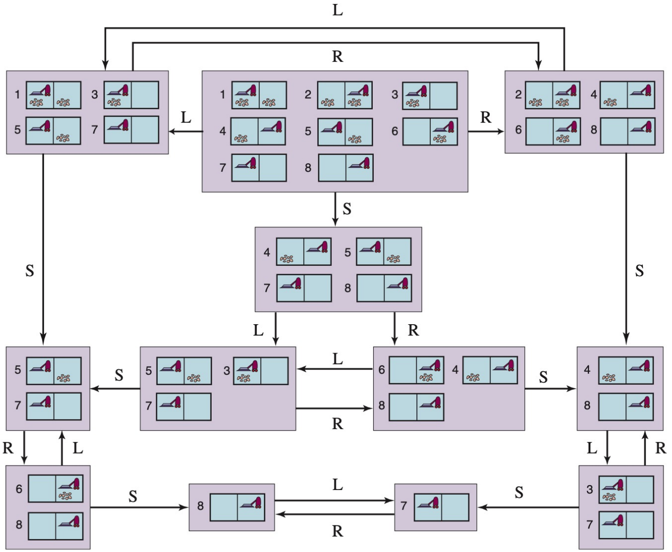
We are, in a way, making compound nodes with multiple outcomes in, where some of our actions lead to specific environment settings inside those belief states.




Of course it can be <u>**both**</u> nondeterministic and partially observable!

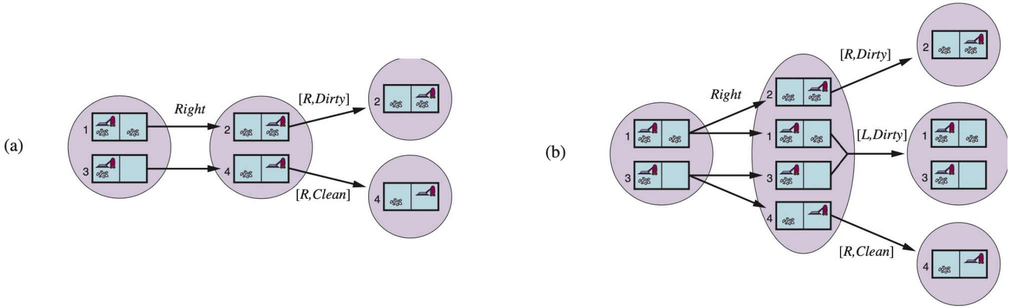
# Searching through the belief space in deterministic environments

If we have a deterministic setting, we can use an ordinary search algorithm.



# Searching through the belief space in partially observable environments

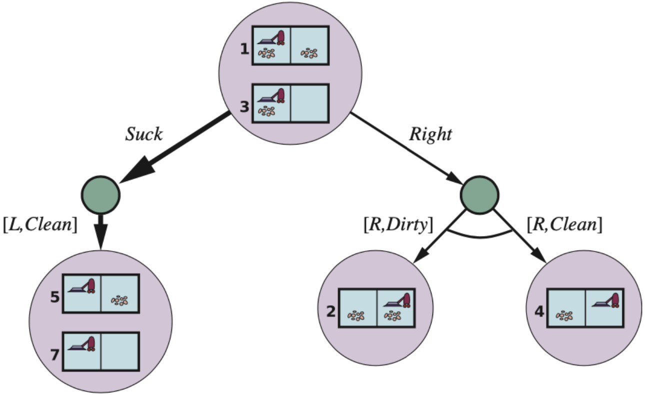
## With sensors




- The agent knows where it is and see the dirt (if any) on its spot
- The **transition model** becomes a **function** of a belief state, an action, and a **another** belief state
    - In case of nondeterminism (right), we do like Dr. Strange and consider possible outcomes on different universes. **How**?

# Seaching through the belief space in partially observable environments

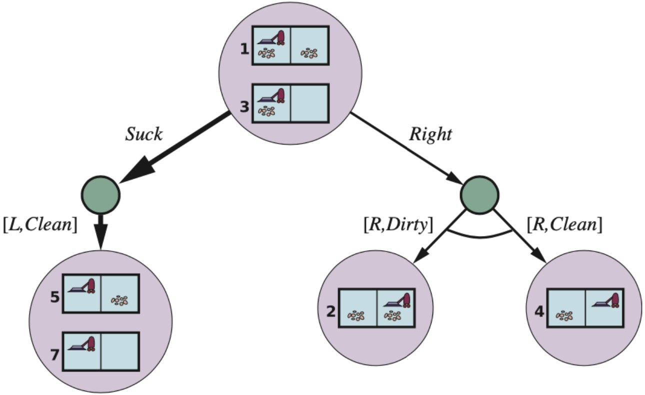
## With sensors, in a nondeterministic world




- Using an **AND-OR tree**

# Seaching through the belief space in partially observable environments

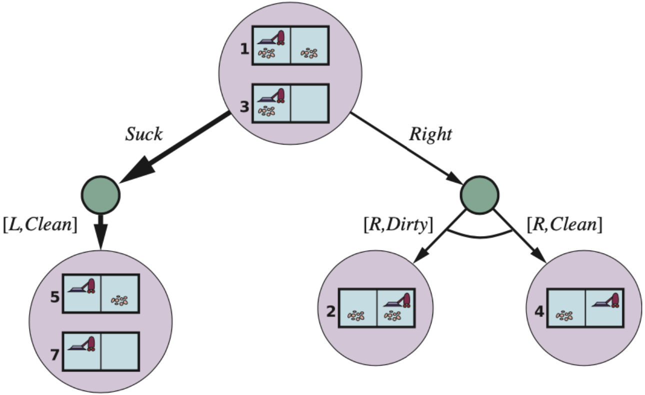
## With sensors, in a nondeterministic world




- Using an **AND-OR tree**
- Notice how the nodes are now **belief states**

# Seaching through the belief space in partially observable environments

## With sensors, in a nondeterministic world




- Using an **AND-OR tree**
- Notice how the nodes are now **belief states**
- The solution is a conditional **plan**