Beyond classical search

In which we relax the simplifying assumptions of the previous lecture, therby getting closer to the real world

Piotr Fulmański

Instytut Nauk Ekonomicznych i Informatyki,
Państwowa Wyższa Szkoła Zawodowa w Płocku, Polska

November 12, 2013

Spis treści

Goal of this lecture

Previous lecture addressed a single category of problems. Problems which are

- observable,
- deterministic,
- with known environments
- where the solution is a sequence of actions.

Now we look at what happens when these assumptions are relaxed.

Goal of this lecture: part I

We begin with fairly simple case: first part cover algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. **These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it.** The family of local search algorithms includes methods inspired by statistical physics (simulated annealing) and evolutionary biology (genetic algorithms).

Goal of this lecture: part II

Then in the next part, we examine what happens when we relax the assumptions of determinism and observability. The key idea is that if an agent cannot predict exactly what percept it will receive, than it will need to consider what to do under each contingency that its percepts may reveal.

Goal of this lecture: part III

With partial observability, the agent will also need to keep track of the states it might be in.

Goal of this lecture: part II

Then in the next part, we examine what happens when we relax the assumptions of determinism and observability. The key idea is that if an agent cannot predict exactly what percept it will receive, than it will need to consider what to do under each contingency that its percepts may reveal.

Goal of this lecture: part III

With partial observability, the agent will also need to keep track of the states it might be in.

Goal of this lecture: part IV

Last part investigates online search, in which the agent is faced with a state space that is initially unknown and must be explored.

Part I: Local search algorithms and optimization problems

Irrelevant path

The search algorithm that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path. When a goal is found, the path to that goal also constitutes a solution to the problem.

Irrelevant path

Irrelevant path

In many problems, however, the path to the goal is irrelevant.

Irrelevant path

For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added. The some general property holds for many important applications such as integrated-circuit design, factory-floor layout, job-shop sheduling, automatic programming etc.

Different class of algorithms

Different class of algorithms

If the path to the goal does not matter, we might conside a different class of algorithms, ones that do not worry about path at all.

Local search – advantages

Local search algorithms operate using a single current node (rather than miltiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, thay have two key advantages

- they use very little memory – usualy a constant amount,
- they can often find resonable solution in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

State-space landscape

To understand local search, we find it useful to consider **state-space landscape**. Landscape has both

- „location" (defined by the state) and
- „elevation" (defined by the value of the heuristic cost function or objective function).

Typically, if elevation corresponds to cost, than the aim is to find lowest valley – a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak – a **global maximum**. Of course you can convert from one to the other just by inserting a minus sign. Local search algoithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists. An **optimal** algorithm always find a global minimum/maximum.

Various topographic features

- global maximum
- local maximum
- flat local maximum
- shoulder

Hill-climbing search

Description

The hill-climbing search algorithm is simply a loop that continually moves in the direction of increasing value – that is, uphill. It terminates when it reaches a „peak" where no neighbour has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbours of the current state. **This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.** Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

Hill-climbing search

Description

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbour state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state. Unfortunately, hill climbing often gets stuck for the following reasons: local maxima (minima), plateaux.

Variants

Many variants of hill climbing have been invented.

- Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many of successors.

Hill-climbing search

Complete hill-climbing algoriyhm

The hill-climbing algorithms described so far are incomplete – they often fail to find a goal when one exists because they can get stuck on local maxima. Random-restart hill climbing conducts a series of hill-climbing seach from randomly generated initial states (although generating a random state from an implicitly specified state space can be a hard problem in itself), until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state. It was proved that it is best, in some cases, to restart a randomized seach algorithm after a particular, fixed amount of time and that this can be much more efficient than letting each search continue indefinitely.

Hill climbing (HC)

```
function HC(problem) return a state that is a local maximum
{
  current := MAKE_NODE(problem.INITIAL_STATE)
  loop
  {
    neighbour := a highest-valued successor of current
    if (neighbour.VALUE <= curren.VALUE) then
      return current.STATE
    current := neighbour
  }
}
```

Simulated annealing

Idea

A hill-climbing algorithm that never makes downhill moves toward states with lower value (or higher cost) is guarenteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk – that is, moving to a successor choosen uniformly at random from the set of successors – is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill-climbing with a random walk in some way that yields both efficiency and completeness. **Simulated annealing** is such an algorithm. In metallurgy annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.

Simulated annealing

Idea

The innermost loop of he simulated annelaing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move. **If the move improves the situation, it is always accepted.** Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the „badness" of the move. The probability also decreases as the „temperature" goes down: bad moves are more likely to be allowed at the start when „temperature" is high, and they become more unlikely as „temperature" descreases. If the schedule lowers „temperature" slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing (SA)

```
function SA(problem,schedule) return a solution state
{
  current := MAKE_NODE(peoblem.INITIAL_STATE)
  for t=1 to infty do
  {
    T := schedule(t)
    if (T = 0) then
      return current
    next := a randomly selected successor of current
    delta_E := next.VALUE - current.VALUE
    if ( delta_E > 0 ) then
      current := next
    else
      current := next only with probability P := exp(delta_E/T)
  }
}
```

Genetic algorithm

Terms

population, individual, fitness function, selection, crossover, mutation

Genetic algorithm (GA)

```
function GA(population,FITNESS_FUNCTION) return an indyvidual
{
  do
  {
    newPopulation := EMPTY_SET
    for i=1 to SIZE(population)
    {
      x := RANDOM_SELECTION(population,FITNESS_FUNCTION)
      y := RANDOM_SELECTION(population,FITNESS_FUNCTION)
      child := REPRODUCE(x,y)
      if (small random probability) then
        child := MUTATE(child)
      newPopulation.ADD(child)
    }
    population := newPopulation
  }
  while ( NOT(some individual is fit enough, or enough time has elapsed) )
  return the best individual in population, according to FITNESS_FUNCTION
}
```

Continuous space

Yet none of the algorithms we have described (except for first-choice hill climbing) can handle continuous state and action space, because they have infinite branching factors. This part provides a very brief introduction to some local search rechniques for finding optimal solution in continuous spaces.

Gradient

Many methods attempt to use gradient of the landscape to find a maximum. The gradient of the objective function is a vector that gives the magnitude and direction of the steepest slope.

Source of information

Jan Kusiak, Anna Danielewska-Tułecka, Piotr Oprocha, *Optymalizacja. Wybrane metody z przykładami zastosowań*, Wydawnictwo Naukowe PWN, Warszawa 2009.

Part II: Searching with nondeterministic actions

Percepts

In previous lecture we assume that

- the environment is fully observable and deterministic
- and that the agent knows what the effects of each action are.

Therefore, the agent

- can calculate exactly which state results from any sequence of actions
- and always knows which state it is in.

Its percepts provide no new information after each action, although of course they tell the agent the initial state. When the environment is either partially observable or nondeterministic (or both), percepts become useful.

Nondeterministic action

Percepts

In previous lecture we assume that

- the environment is fully observable and deterministic
- and that the agent knows what the effects of each action are.

Therefore, the agent

- can calculate exactly which state results from any sequence of actions
- and always knows which state it is in.

Its percepts provide no new information after each action, although of course they tell the agent the initial state. When the environment is either partially observable or nondeterministic (or both), percepts become useful.

Percepts

In previous lecture we assume that

- the environment is fully observable and deterministic
- and that the agent knows what the effects of each action are.

Therefore, the agent

- can calculate exactly which state results from any sequence of actions
- and always knows which state it is in.

Its percepts provide no new information after each action, although of course they tell the agent the initial state. When the environment is either partially observable or nondeterministic (or both), percepts become useful.

Percepts

In previous lecture we assume that

- the environment is fully observable and deterministic
- and that the agent knows what the effects of each action are.

Therefore, the agent

- can calculate exactly which state results from any sequence of actions
- and always knows which state it is in.

Its percepts provide no new information after each action, although of course they tell the agent the initial state. When the environment is either partially observable or nondeterministic (or both), percepts become useful.

Percepts

In previous lecture we assume that

- the environment is fully observable and deterministic
- and that the agent knows what the effects of each action are.

Therefore, the agent

- can calculate exactly which state results from any sequence of actions
- and always knows which state it is in.

Its percepts provide no new information after each action, although of course they tell the agent the initial state. When the environment is either partially observable or nondeterministic (or both), percepts become useful.

Percepts

In previous lecture we assume that

- the environment is fully observable and deterministic
- and that the agent knows what the effects of each action are.

Therefore, the agent

- can calculate exactly which state results from any sequence of actions
- and always knows which state it is in.

Its percepts provide no new information after each action, although of course they tell the agent the initial state. When the environment is either partially observable or nondeterministic (or both), percepts become useful.

Nondeterministic action

Percepts

In previous lecture we assume that

- the environment is fully observable and deterministic
- and that the agent knows what the effects of each action are.

Therefore, the agent

- can calculate exactly which state results from any sequence of actions
- and always knows which state it is in.

Its percepts provide no new information after each action, although of course they tell the agent the initial state. When the environment is either partially observable or nondeterministic (or both), percepts become useful.

Nondeterministic action

Percepts

- In a **partial observable** environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.
- When the environment is **nondeterministic**, percepts tell the agent which of the possible outcomes of its actions has actually occurred.

In both cases, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts. So the solution to a problem is not a sequence but **contingency plan** (**strategy**) that specifies what to do depending on what percepts are received. Now we examine the case of nondeterminism.

Nondeterministic action

Percepts

- In a **partial observable** environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.
- When the environment is **nondeterministic**, percepts tell the agent which of the possible outcomes of its actions has actually occurred.

In both cases, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts. So the solution to a problem is not a sequence but **contingency plan** (**strategy**) that specifies what to do depending on what percepts are received. Now we examine the case of nondeterminism.

Nondeterministic action

Percepts

- In a **partial observable** environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.
- When the environment is **nondeterministic**, percepts tell the agent which of the possible outcomes of its actions has actually occurred.

In both cases, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts. So the solution to a problem is not a sequence but **contingency plan** (**strategy**) that specifies what to do depending on what percepts are received. Now we examine the case of nondeterminism.

Nondeterministic action

Percepts

- In a **partial observable** environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.
- When the environment is **nondeterministic**, percepts tell the agent which of the possible outcomes of its actions has actually occurred.

In both cases, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts. So the solution to a problem is not a sequence but **contingency plan** (**strategy**) that specifies what to do depending on what percepts are received. Now we examine the case of nondeterminism.

Nondeterministic action

Assumptions

As an example, we use the vacuum world. As we know, the state space has eight states, there are three action (left, right, suck) and the goal is to clean up all the dirts. If the environment is observable, deterministic and completely known, then the problem is solvable by any of the algorithms from last lecture and the solution is an action sequence. Now suppose that we introduce nondeterminism and the suck action works as follows

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the floor.

Nondeterministic action

Assumptions

- To provide a precise formulation of this type problems, we need to **generalize the notion of a transition model**. Instead of defining the transition model by a result function that returns a single state, we use a results function that returns a **set** of possible outcome states.
- We also need to **generalize the notion of a solution** to the problem. There is no single sequence of actions that solves the problem. Instead we need a contingency plan such as the following: `do action and if the result is ... then do ...`

Nondeterministic action

Assumptions

- To provide a precise formulation of this type problems, we need to **generalize the notion of a transition model**. Instead of defining the transition model by a result function that returns a single state, we use a results function that returns a **set** of possible outcome states.
- We also need to **generalize the notion of a solution** to the problem. There is no single sequence of actions that solves the problem. Instead we need a contingency plan such as the following: `do action and if the result is ... then do ...`

Nondeterministic action

Assumptions

- To provide a precise formulation of this type problems, we need to **generalize the notion of a transition model**. Instead of defining the transition model by a result function that returns a single state, we use a results function that returns a **set** of possible outcome states.
- We also need to **generalize the notion of a solution** to the problem. There is no single sequence of actions that solves the problem. Instead we need a contingency plan such as the following: `do action and if the result is ... then do ...`

Why AND and OR? – OR nodes

To find contingent solution to nondeterministic problem, we begin by constructing search trees, but now trees have a different character. In a deterministic environment, the only branching is introduced by the agent's own choices in each state. We call this nodes **OR nodes**.

In the vacuum world, for example, at an OR node the agents chooses left, right or suck.

Why AND and OR? – OR nodes

To find contingent solution to nondeterministic problem, we begin by constructing search trees, but now trees have a different character. In a deterministic environment, the only branching is introduced by the agent's own choices in each state. We call this nodes **OR nodes**.

In the vacuum world, for example, at an OR node the agents chooses left, right or suck.

Why AND and OR? – AND nodes

In a nondeterministic environment, branching is also introduced by the environment's choice of outcome for each action. We call these nodes **AND nodes**. For example, if te action A in state S_1 leads to a state in the set $\{S_2, S_3\}$, so the agent would need to find a plan for state S_2 **and** for state S_3. These two kinds of nodes alternate, leading to an AND-OR tree.

And-or search tree

Solution

A solution for an AND-OR search problem is a subtree that

- has a goal node at every leaf,
- sppecifies one action at each of its OR nodes,
- includes every outcome branch at each of its AND nodes.

And-or search tree

What is important, an AND-OR tree specifies only the search space for solving a problem. Different search strategies for searching the space are possible.

AND-OR graph search (AO)

```
function AO(problem) return a conditional plan, or failure
{
  OR_SEARCH(problem.INITIAL_STATE,problem,[])
}

function OR_SEARCH(state,problem,path) return a conditional plan, or failure
{
  if (problem.GOAL_TEST(state)) then
    return the empty plan
  if (state is on path) then /* prevent from infinite loop */
    return failure
  for each action in problem.ACTIONS(state)
  {
    plan := AND_SEARCH(RESULTS(state,action),problem,[state|path])
    if (plan != failure) then
      return [action|plan]
  }
}

function AND_SEARCH(states,problem,path) return a conditional plan, or failure
{
  for each s_i in states
  {
    plan_i := OR_SEARCH(s_i,problem,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]
}
```

Part III: Searching with no/partial observations

Searching with partial observations

Key concept: belief

We now turn to the problem of partial observability, where the agent's percepts do not suffice to pin down the exact state. The key concept required for solving partially observable problems is the **belief state**, representing the agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point.

Searching with partial observations

We begin with the simplest scenario for studying belief states, which is when the agent has no sensors at all; than we add in partial sensing as well as nondeterministic actions.

Searching with partial observations

We begin with the simplest scenario for studying belief states, which is when the agent has no sensors at all; than we add in partial sensing as well as nondeterministic actions.

Part III: Searching with no/partial observations
Subpart I: Searching with no observation

Searching with no observation

Sensorless – is this a problem?

When the agent's percepts provide no information at all, we have what is called a **sensorless** problem. At first, one might think the sensorless agent hase no hope of solving a problem if has no idea what state it's in; in fact, sensorless problems are quite often solvable. Moreover, sensorless agent can be surprisingly useful, primarily because they don't rely on sensors working properly.

Sensorless version of vacuum world

We can make a sensorless version of vacuum world. Assume that the agent knows the geography of its world, but doesn't know its location or the distribution of the dirt. In that case, its initial state could be any element of the set of all states $\{1, 2, 3, 4, 5, 6, 7, 8\}$. The sequence `[right, suck, left, suck]` is guaranteed to reach the goal state (7).

Belief states

Belief states

To solve sensorless problems, we search in the **space of belief** states rather than physical states. Notice that in belief-state space, the problem is fully observable because the agent always knows its own belief state. Furthermore, the solution (if any) is always a sequence of actions. This is because, as in the ordinary problems of previous part (*Solving problems by searching*), the percepts received after each action are completely predictable – they're always empty! So there are no contingencies to plan for. This is true even if the environment is nondeterministic.

Sensorless problem definition (part I)

It is instructive to see how the belief state search problem is constructed. Suppose the underlying physical problem P is defined by `ACTIONS(P)`, `RESULT(P)`, `GOAL-TEST(P)` and `STEP-COST(P)`. Than we can define the corresponding sensorless problem as follows:

- **Belief states.** The entire belief-state space contains every possible set of physical states. If P has N states, then the sensorless problem has up to 2^N states, although many may be unreachable from the initial state.
- **Initial state.** Typically the set of all states in P, although in some cases the agent will have more knowledge than this.

Sensorless problem definition (part II)

- **Actions.** Suppose the agent is in belief state $b = \{s_1, s_2\}$, but ACTIONS(P,s_1)$\neq$ACTIONS(P,s_2); than the agent is unsure of what actions are legal. If we assume that illegal actions have no effect on the environment, than it is safe to take the union of all the actions in any of the physical states in the current belief state b

$$ACTIONS(b) = \bigcup_{s \in b} ACTIONS(P, s)$$

On the other hand, if an illegal action might be the end of the world, it is safer to allow only the interseaction, that is, the set of actions legal in all the states.

Example: for the vacuum world, every state has the same legal actions, so both methods give the same result.

Sensorless problem definition (part III)

- **Transition model.** The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state. For deterministic actions, the set of states that might be reached is

$$b' = RESULT(b, a) = \{s' : s' = RESULT(P, s, a) \quad \text{and} \quad s \in b\}$$

With deterministic actions, b' is never larger than b. With nondeterminism, we have

$$b' = \bigcup_{s \in b} RESULTS(P, s, a),$$

which may be larger than b. The process of generating the new belief state after the action is called the **prediction** step; the notation $b' = PREDICT(P, b, a)$ will come in handy.

Sensorless problem definition (part IV)

- **Goal test.** The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if all the physical states in it satisfy `GOAL-TEST(P)`. The agent may accidentally achieve the goal earlier, but it won't know that it has done so.
- **Path cost.** If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values. For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

Sensorless problem solving

The preceding definitions enable the automatic construction of the belief state problem formulation from the definition of the underlying physical problem. Once this is done, we can apply any of the search algorithm from previous part (*Solving problems by searching*). In „ordinary" graph search, newly generated states are tested to see if they are identical to existing states. This works for belief states, too. If belief state X has already been generated and found to be solvable, than any subset of X is guaranteed to be solvable. This extra level of pruning may dramatically improve the efficiency of sensorless problem solving.

Part III: Searching with no/partial observations
Subpart II: Searching with partially observable problem

Searching with partially observable problem

PERCEPT function

For a general partially observable problem, we have to specify how the environment generates percepts for the agent. For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares. The formal problem specification includes a `PERCEPT(s)` function that returns the percept received in a given state.

- If sensing is nondeterministic, then we use a `PERCEPT` function that returns a set of possible percepts.
- Fully observable problems are a special case in which `PERCEPT(s)=s` for every state s.
- Sensorless problems are a special case in which `PERCEPT(s)=null`.

Partially observable problem definition

Partially observable problem definition

When observations are partial, it will usually be the case that several states could have produced any given percept. The `ACTIONS`, `STEP-COST`, and `GOAL-TEST` are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated.

Partially observable problem definition

Partially observable problem definition – transitions

We can think of transitions from one belief state to the next for a particular actions as occuring in three stages:

- The **prediction** stage is the same as for sensorless problems: given the action a in belief state b, the predict belief state is

$$\hat{b} = PREDICT(b, a).$$

- The **observation prediction** stage determines the set of percepts o that could be observed in the predicted belief state

$$POSSIBLE-PERCEPTS(\hat{b}) = \{o : o = PERCEPT(s) \quad \text{and} \quad s \in \hat{b}\}.$$

Partially observable problem definition

Partially observable problem definition – transitions

- The **update** stage determines, for each possible percept, the belief state that would result from the percept. The new belief state b_0 is just the set of states in $\hat{b}$ that could have produced the percept

$$b_0 = UPDATE(\hat{b}, o) = \{s : o = PERCEPT(s) \quad \text{and} \quad s \in \hat{b}\}.$$

Notice that each updated belief state b_0 can be no larger than the predicted belief state $\hat{b}$; observations can only help reduce uncertainty compared to the sensorless case. Moreover, for deterministic sensing, the belief states for the different possible percepts will be disjoint, forming a partition of the original predicted belief state.

Partially observable problem definition

Possible belief states resulting from a given action

Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:

$$RESULT(b,a) = \{b_0 : b_0 = UPDATE(PREDICT(b,a),o) \text{ and } o \in POSSIBLE - PERCEPTS(PREDICT(b,a))\}.$$

Again, the nondeterminism in the partially observable problem comes from the inability to predict exactly which percept will be received after acting; underlying nondeterminism in the physical environment may contribute to this inability by enlarging the belief state at the prediction stage, leading to more precepts at the observation stage.

Solving partially observable problems

We showed how to derive the `RESULT` function for a partially observable problem from an underlying physical problem and the `PERCEPT` function. Given such a formulation, the AND-OR search algorithm can be applied directly to derive a solution.

Part IV: Online search agents and unknown environments

Online search

- So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution **before** setting foot in the real world and then execute the solution.
- In contrast, an **online search** agents interleaves computation and action:
  - first it takes an action,
  - then it observes the environment and computes the next action.

Online search

- So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution **before** setting foot in the real world and then execute the solution.
- In contrast, an **online search** agents interleaves computation and action:
  - first it takes an action,
  - then it observes the environment and computes the next action.

Online search

- So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution **before** setting foot in the real world and then execute the solution.
- In contrast, an **online search** agents interleaves computation and action:
  - first it takes an action,
  - then it observes the environment and computes the next action.

Online search

- So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution **before** setting foot in the real world and then execute the solution.
- In contrast, an **online search** agents interleaves computation and action:
  - first it takes an action,
  - then it observes the environment and computes the next action.

Online search

- So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution **before** setting foot in the real world and then execute the solution.
- In contrast, an **online search** agents interleaves computation and action:
  - first it takes an action,
  - then it observes the environment and computes the next action.

Online search

- So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution **before** setting foot in the real world and then execute the solution.
- In contrast, an **online search** agents interleaves computation and action:
  - first it takes an action,
  - then it observes the environment and computes the next action.

Online search

Online search is a good idea in dynamic or semidynamic domains – domains where there is a penalty for sitting around and computing too long. Online search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won't. Of course, there is a tradeoff: the more an agent plans ahead, the less often it will find itself up the creek without a paddle.

Unknown environments

Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an exploration problem and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following

- `ACTIONS(s)`, which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$ – note that this cannot be used until the agent knows that s' is the outcome;
- `GOAL-TEST(s)`.

Note in particular that the agent cannot determine `RESULT(s,a)` except by actually being in s and doing a. Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following

- `ACTIONS(s)`, which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$ – note that this cannot be used until the agent knows that s' is the outcome;
- `GOAL-TEST(s)`.

Note in particular that the agent cannot determine `RESULT(s,a)` except by actually being in s and doing a. Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following

- `ACTIONS(s)`, which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$ – note that this cannot be used until the agent knows that s' is the outcome;
- `GOAL-TEST(s)`.

Note in particular that the agent cannot determine `RESULT(s,a)` except by actually being in s and doing a. Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following

- `ACTIONS(s)`, which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$ – note that this cannot be used until the agent knows that s' is the outcome;
- `GOAL-TEST(s)`.

Note in particular that the agent cannot determine `RESULT(s,a)` except by actually being in s and doing a. Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following

- `ACTIONS(s)`, which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$ – note that this cannot be used until the agent knows that s' is the outcome;
- `GOAL-TEST(s)`.

Note in particular that the agent cannot determine `RESULT(s,a)` except by actually being in s and doing a. Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following

- `ACTIONS(s)`, which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$ – note that this cannot be used until the agent knows that s' is the outcome;
- `GOAL-TEST(s)`.

Note in particular that the agent cannot determine `RESULT(s,a)` except by actually being in s and doing a. Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following

- `ACTIONS(s)`, which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$ – note that this cannot be used until the agent knows that s' is the outcome;
- `GOAL-TEST(s)`.

Note in particular that the agent cannot determine `RESULT(s,a)` except by actually being in s and doing a. Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state.

Online search problems

Typically, the agent's objective is to reach a goal state while minimizing cost. Another possible objective is simply to explore the entire environment. For the cost (which is the total path cost) it is common to compare it with the cost of the path the agent would follow if it knew the search space in advance – that is, the actual shortest path. This is called the **competitive ratio**; we would like it to be as small as possible.

Online search agents – augmenting map of the environment

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithm we have seen previously.

Online search agents – local order of expanding nodes

For example, **offline algorithms** such as A* **can expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real action**. An online algorithm, on the other hand, can discover successors only for a node that is physically occupies. To avoid traveling all the way across the tree too expand the next node, it seems better to expand nodes in a local order. Depth first search has exactly this property because the next node expanded is a child of the previous node expanded.

An online depth-first search agent

```
s' - a percept that identifies the current state
persistent:
result - a table indexed by state and action, initially empty
untried - a table that lists, for each state, the actions not yet tried
unbacktracked - a table that lists, for each state, the backtracks not yet tried
s, a - the previous state and action, initially null

function ONLINE-DFS-AGENT(s') return an action
{
  if(GOAL-TEST(s')) then return stop
  if(s' is a new state (not in untried)) then untried[s'] := ACTIONS(s')
  if(s is not null) then
  {
    result[s,a] := s'
    add s to the front of unbacktracked[s']
  }
  if(untried[s'] is empty)then
  {
    if(unbacktracked[s'] is empty) then
      return stop
    else
      a := an action b such that result[s',b]=POP(unbacktracked[s'])
  }
  else
    a := POP(untried[s'])

  s := s'
  return a;
}
```

An online depth-first search agent

An online depth-first search agent – description

This agent stores its map in a table, `RESULT[s,a]`, that records the state resulting from executing action *a* in state *s*. Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack **physiacally**. In depth first search, this means going back to the state from which the agent most recently entered the current state. To achieve that, the algorithm keeps a table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, than its search is complete.

An online depth-first search agent

Note: reversible actions

Notice that because of its method of backtracking, online-dfs-agent works only in state space where the actions are reversible. There are slightly more complex algorithms that work in general state space, but no such algorithm has a bounded competitive ratio.

Solving online problem

Online local search

- hill-climbing
- random-walk

LRTA* (learning real-time A*) (idea)

LRTA* (learning real-time A*) (idea) The basic idea is to store a current best estimate $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic estimate $h(s)$ and is updated as the agent gains experience in the state space.

Five iterations of LRTA* on a one-dimensional state space

```
                            -->
<-1-> 8 <-1-> 9 <-1-> [2] <-1-> 2 <-1-> 4 <-1-> 3 <-1->

                           U
<-1-> 8 <-1-> 9 <-1-> 2 <-1-> [2] <-1-> 4 <-1-> 3 <-1->

                                 <--
<-1-> 8 <-1-> 9 <-1-> 3 <-1-> [2] <-1-> 4 <-1-> 3 <-1->

                                    U
<-1-> 8 <-1-> 9 <-1-> [3] <-1-> 2 <-1-> 4 <-1-> 3 <-1->

                               -->
<-1-> 8 <-1-> 9 <-1-> [3] <-1-> 4 <-1-> 4 <-1-> 3 <-1->

                              U
<-1-> 8 <-1-> 9 <-1-> 3 <-1-> [4] <-1-> 4 <-1-> 3 <-1->

                                       -->
<-1-> 8 <-1-> 9 <-1-> 5 <-1-> [4] <-1-> 4 <-1-> 3 <-1->

                                    U
<-1-> 8 <-1-> 9 <-1-> 5 <-1-> 4 <-1-> [4] <-1-> 3 <-1->

<-1-> 8 <-1-> 9 <-1-> 5 <-1-> 5 <-1-> [4] <-1-> 3 <-1->
```

LRTA*-AGENT

```
inputs: s' - a percept that identifies the current state
persistent:
result - a table, indexed by state and action, initially empty
H - a table of cost estimates indexed by state, initially empty
a,s - the previous state and action, initially null

function LRTA*-AGENT(s) return an action
{
  if(GOAL-TEST(s')) then return stop
  if(s' is a new state (not in H)) then H[s'] := h(s')
  if(s is not null)
  {
    result[s,a] := s'
    H[s] := min(LRTA*-COST(s,b,result[s,b],H), b in ACTIONS(s))
  }
  a := an action b in ACTIONS(s') that minimizes LRTA*-COST(s',b,result[s',b],H)
  s := s'
  return a
}

function LRTA*-COST(s,a,s',H) return a cost estimate
{
  if(s' is undefined) then
    return h(s)
  else
    return c(s,a,s')+H[s']
}
```