Artificial Intelligence 1

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

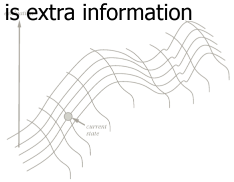
Problem Solving: Local and On-line Search Techniques

Introduction

- We know how to use **heuristics in search**
  - BFS, A*, IDA*, RBFS, SMA*
- **Today**:
  - **What if the path is not important?**
    - Local search: HC, SA, BS, GA
  - **What if the world changes?**
    - on-line search, LRTA*

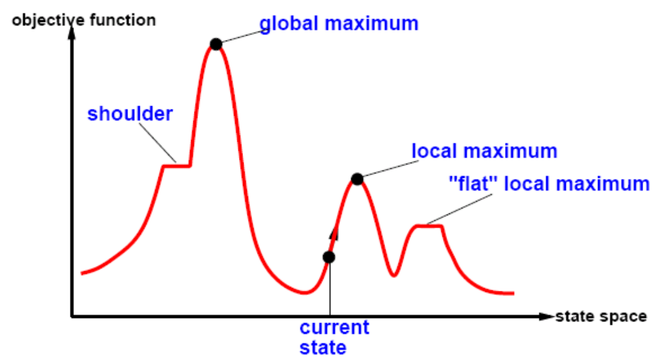
Local search

- So far we systematically explored all paths possibly going to the goal and the path itself was a part of the solution.
- For some problems (e.g. 8-queens) the path is not relevant to the problem, only the goal is important.
- For such problems we can try **local search** techniques.
  - we also keep a single state only (constant memory)
  - in each step we slightly modify the state
  - usually, the path is not stored
  - the method can also look for an **optimal state**, where the optimality is defined by an objective function (defined for states).
    - For example, for the 8-queens problem the objective function can be defined by the number of conflicting queens– this is extra information about the quality of states.



Local search - terminology

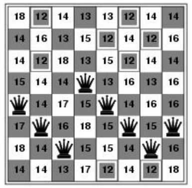
- Local search can be seen as a move in the state-space **landscape**, where coordinates define the state and elevation corresponds to the objective function.



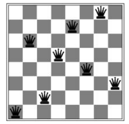
- From the neighboorhood the algorithm selects the state with the best value of the objective function and moves there (**hill climbing**).
  - knows only the neighborhood
  - the current state is forgotten

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

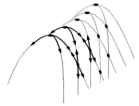
- **# conflicts = 17**
- **state change = change row of a queen**
- **random selection among more best neighbours**



HC is a **greedy algorithm** – goes to the best neighbour with looking ahead



- **local optimum** (the state such that each neighbour is not better)
  - HC cannot escape local optimum
- **ridges** (a sequence of local optima)
  - difficult for greedy algorithms to navigate

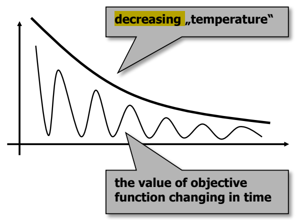


- **plateaux** (a flat area of the state-space landscape)
  - shoulder – progress is still possible
  - HC may not find a solution (cycling)

- **stochastic HC**
  - chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move
  - usually converges more slowly than steepest ascent
  - in some landscapes, it finds better solutions
- **first-choice HC**
  - implements stochastic HC until a successor better than the current state is generated
  - a good strategy when a state has many (thousands) of successors
- **random-restart HC**
  - conducts a series of HC searches from randomly generated initial states (restart)
  - can escape from a local optimum
  - if HC has a probability p of success then the expected number of restarts required is 1/p
  - a very efficient method for the N-queens problem (p ≈ 0.14 i.e. 7 iterations to find a goal)

- **HC** never makes the "downhill" moves towards states with lower value so the algorithm is not complete (can get stuck on a local optimum).
- **Random walk** – that is moving to a successor chosen randomly – is complete but extremely inefficient.
- **Simulated annealing** combines hill climbing with random walk
  - motivation in metallurgy – process to harden metals by heating them to a high temperature and then gradually cooling them (allowing the material to reach a low-energy crystalline state
  - the algorithm picks a random move and accepts it if:
    - it improves the situation
    - it worsens the situation but this is allowed with a probability given by some temperature value and how much the state worsens; the temperature is decreasing according to a **cooling scheme**

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```



Local beam search

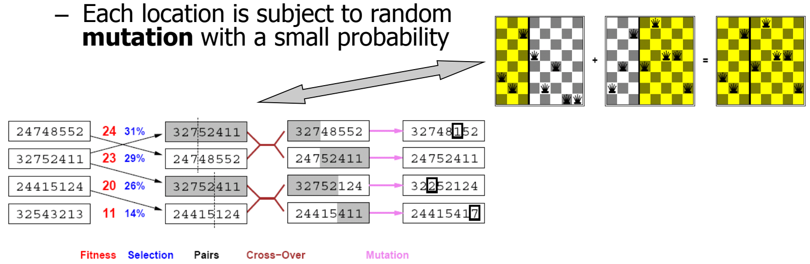
Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.

Can we exploit available memory better?

- **Local beam search** algorithm
  - keeps track of *k* states rather than just one
  - at each step, all the successors of all *k* states are generated
    - if any one is a goal, the algorithm halts
  - otherwise, it selects the *k* best successors and repeats
- **This is not running *k* restarts of HC in parallel!**
  - useful information is passed among the parallel search threads
  - The algorithm quickly abandons unfruitful searches and moves its resources to where the most is being made
  - can suffer from a lack of diversity
    - stochastic beam search helps alleviate this problem (*k* successors are chosen at random with the probability being an increasing function of state value)
    - resemble the process of natural selection

Genetic algorithms

- A variant of stochastic beam search in which successors are generated by combing two parent states (sexual reproduction)
  - Begin with a set of *k* randomly generated states – **population**
    - Each state is represented as a string over a finite alphabet (DNA)
    - **fitness** function evaluates the states (objective function)
  - Select a pair of states for reproduction (probability of selection is given by the fitness function)
  - For each pair choose a **crossover** point from the positions in the string
  - Combine **offsprings** to a new state
  - Each location is subject to random **mutation** with a small probability



Genetic algorithm

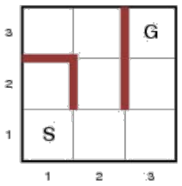
```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     loop for i from 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Offline vs. online

- So far we have concentrated on **offline search**
  - compute a complete solution
  - then execute the solution without assuming percepts
- **Online search** is different in
  - interleave computation and action
    - select an action
    - execute an action
    - observe the environment
    - compute the next action
  - a good idea in dynamic and semidynamic environments
  - helpful in nondeterministic domains (unknown actions and unknown results of actions)

On-line search

- Online search is useful **for agents executing actions** (useless for pure computation).
- The agent knows only the following **information**:
  - **Actions(s)** – a list of actions allowed in state *s*
  - **c(s,a,s')** – the step cost function (cannot be used until the agent knows state s')
  - **Goal-Test(s)** – identifying the goal state
- We assume the following:
  - agent can **recognize a visited state**
  - (agent can build a world map)
  - actions are deterministic
  - agent has an **admissible heuristic** h(s)



Evaluating on-line algorithms

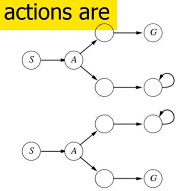
Quality of online algorithms can be measured by **comparing with the offline solution** (knowing the best solution in advance).

- **Competitive ratio**
  = quality if online solution / quality of the best solution
  - can be ∞, for example for a **dead**-end state (if some actions are **irreversible**).

Claim: No algorithm can avoid dead ends in all state spaces.

Proof (adversary argument)

Agent has visited states S and A must make the same decision in both, but in one situation, the agent reaches dead-end.



Assume that the state space is **safely explorable** (some goal state is reachable from every reachable state).

- No bounded competitive ratio can be guaranteed if there are paths of unbounded cost.
- Adversary argument can be used to arbitrarily extend any path.

Hence it is common to describe the performance of online search algorithms in **terms of the size of the entire state space** rather than just the depth of the shallowest goal.

Online DFS

- Opposite to offline algorithms such as A* online algorithms can discover **successors only** for a node that the agent physically occupies.
- It seems better to expand nodes in a **local order** as done for example by DFS.

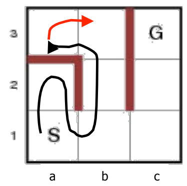
```
function ONLINE-DFS-AGENT(s') returns an action
   inputs: s', a percept that identifies the current state
   static: result, a table, indexed by action and state, initially empty
           unexplored, a table that lists, for each visited state, the actions not yet tried
           unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
           s, a, the previous state and action, initially null

   if GOAL-TEST(s') then return stop
   if s' is a new state then unexplored[s'] ← ACTIONS(s')
   if s is not null then do
      result[a, s] ← s'
      add s to the front of unbacktracked[s']
   if unexplored[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
   else a ← POP(unexplored[s'])
   s ← s'
   return a
```

learns outcome of actions

The state can be visited more times in a single journey (it is leaved to different states) so we need to remember where to go back – **Ariadna thread**

to go back we need a reverse action to the actions used to reach the state

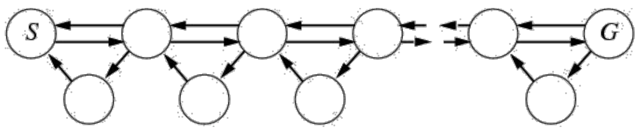
Online DFS



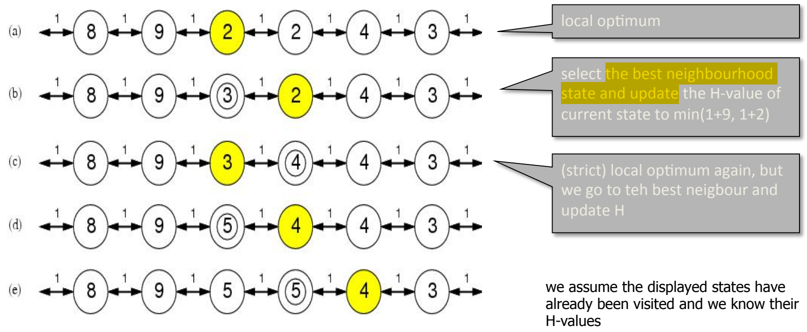
stav	unEX	unBT	rUP	rDN	rLF	rRG
(1,a)	{}	(2,a)	(2,a)	-	-	(1,b)
(2,a)	{}	(1,a)	-	(1,a)	-	-
(1,b)	LF,RG	(1,a)	(2,b)	-		
(2,b)	DW	(1,b)	(3,b)		-	-
(3,b)	DW	(3,a), (2,b)	-		(3,a)	-
(3,a)		(3,b)	-	-	-	(3,b)

- In the worst case, every link is traversed exactly twice (forward and backward).
- This is optimal for exploration, but for finding a goal, the agent's competitive ratio could be arbitrarily bad.
  - an online variant of iterative deepening solves this problem
- On-line DFS works only in state spaces where **actions are reversible**.

- **Hill climbing is an on-line algorithm.**
  - keeps a **single state** (the current physical state)
  - does **local steps** to the neighbouring states
  - in its simplest form **cannot escape local optima**
    - Beware! **Restarts cannot be used in online search!**
    - We can still use **random work**.
      - A random walk will **eventually find a goal** or complete its exploration, provided that the space is finite.
      - The process **can be very slow**.
      - In the following example, a random walk will take exponentially many steps to find the goal (at each step, backward progress is twice as likely as forward progress).



- We can exploit available memory to remember visited states and hence leave local optima.
  - **H(s)** – the current best estimate of path length from *s* to the goal (equals h(s) at the beginning)



- Algorithm LRTA* makes local steps and learns the result of each action as well as a better estimate of distance to the goal (H).

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          H, a table of cost estimates indexed by state, initially empty
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  unless s is null
      result[a, s] ← s'
      H[s] ←     min      LRTA*-COST(s, b, result[b, s], H)
             b∈ ACTIONS(s)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[b, s'], H)
  s ← s'
  return a

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

improve the h-function in the previous state

select the next action with the best cost cile (can also go back); prefer not-yet explored states

if the action has not been applied yet, we optimistically assume that it leads to state with the best cost, i.e. h(s)

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz