Foundations of AI

4. Informed Search Methods

Heuristics, Local Search Methods,
Genetic Algorithms

Bernhard Nebel and Luc De Raedt

Contents

Best-First Search

Search procedures differ in the way they determine the next node to expand.

Uninformed Search: Rigid procedure with no knowledge of how "good" a node is.

Informed Search: Knowledge of the quality of a given node in the form of an *evaluation function h*, which assigns a real number to each node.

Best-First Search: Search procedure that expands the node with the "best" (smallest) *h*-value.

General Algorithm

function BEST-FIRST-SEARCH(*problem,* EVAL-FN) **returns** a solution sequence
  inputs: *problem*, a problem
    Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(*problem, Queueing-Fn*)

When *Eval-Fn* is always correct, we don't need to search!

Greedy Search

A possible way to judge the "worthiness" of a node is to estimate its distance to the goal.

h(n) = estimated distance from *n* to the goal

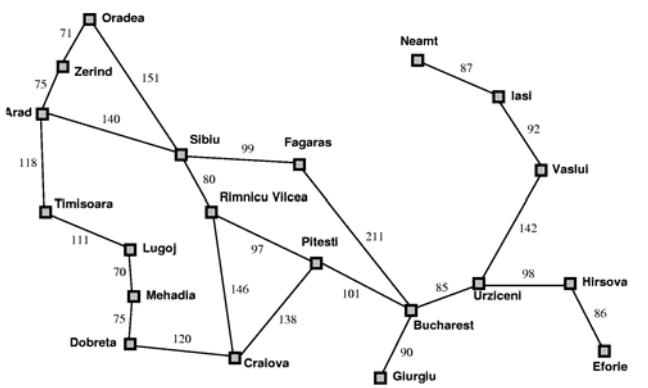
The only real restriction is that *h(n) = 0* if *n* is a goal.

A best-first search with this function is called a *greedy best-first search*.

Example for *route-finding* problem: *h* = straight-line distance between two locations.

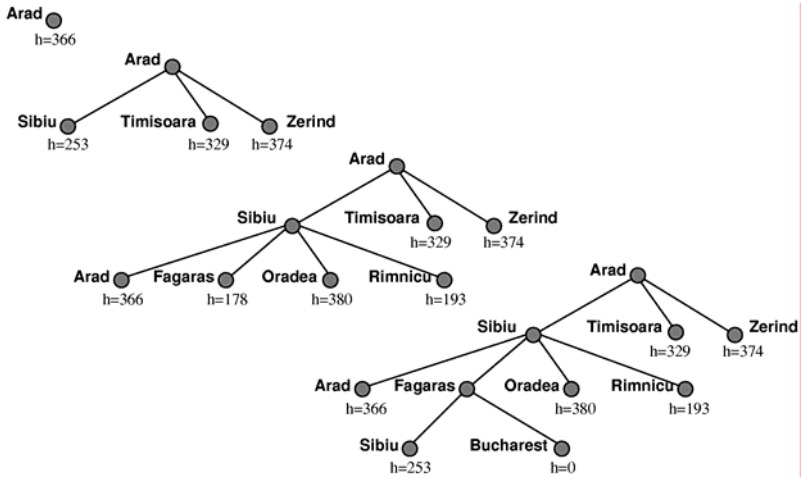
Greedy Search Example:

From Arad to Bucharest



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Search from *Arad* to *Bucharest*



Problems with Greedy Search

- Does find suboptimal solutions
  - Would be *Arad – Sibiu – Rimnicu Vilcea – Pitesti – Bucharest*
- Can be misleading
  - What happens if we want to go from *Iasi* to *Fagaras*?
- Can be incomplete (if we do not detect duplicates) in the above case

Heuristics

The evaluation function *h* in greedy searches is also called a *heuristic function* or simply a *heuristic*.

- The word *heuristic* is derived from the Greek word ἡυρισκειν (note also: *H*ευρεκα!)
- The mathematician Polya introduced the word in the context of problem solving techniques.
- In AI it has two meanings:
  - Heuristics are fast but in certain situations incomplete methods for problem-solving [Newell, Shaw, Simon 1963]
  - Heuristics are methods that focus the search without leading to incompleteness.

→ In all cases, the heuristic is *problem-specific* and *focuses* the search!

A*: Minimization of the Total Estimated Path Costs

A* combines the greedy search with the uniform-search strategy.

g(n) = actual cost from the initial state to *n*.

h(n) = estimated cost from *n* to the closest goal.

f(n) = g(n) + h(n), the estimated cost of the cheapest solution through *n*.

Let h*(n) be the ***actual cost*** of the optimal path from *n* to the closest goal.

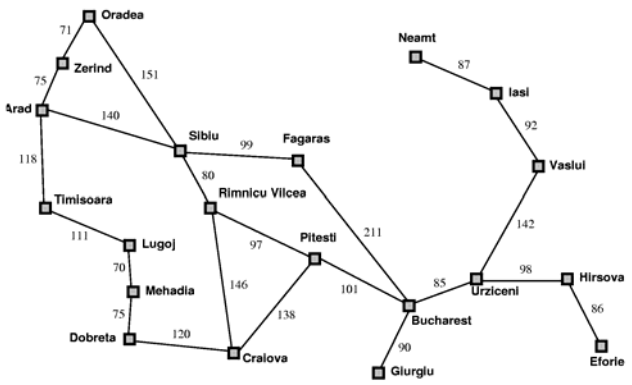
h is ***admissible*** if the following holds for all *n* :

$$h(n) \leq h^*(n)$$

We require that for A*, *h* is admissible.

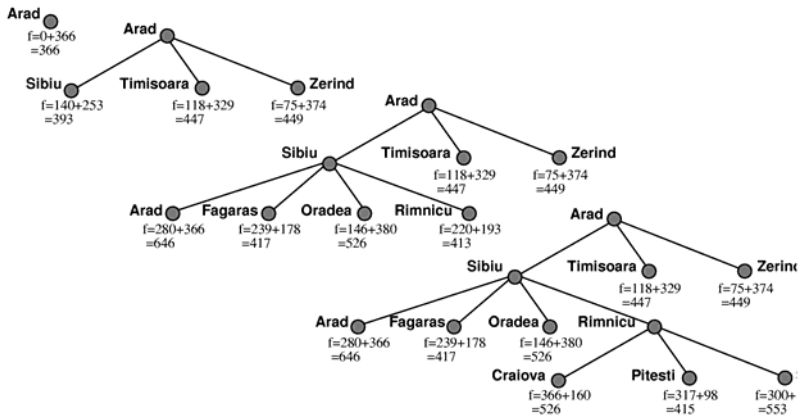
(Straight-line distance is admissible)

A* Search Example



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search from *Arad* to *Bucharest*

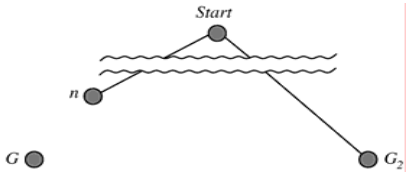


Optimality of A*

Claim: The first solution found in *tree search* has the minimum path cost (for *graph search* it is more difficult)

Proof: Suppose there exists a goal node G with optimal path cost C^*, but A* has found first another node G_2 with $g(G_2) > C^*$, i.e. $f(G_2) > C^*$.

Let *n* be a node on the path from the start to G that has not yet been expanded.



Since *h* is admissible, we have

$f(n) = g(n) + h(n) \leq C^*$.

Since

$f(n) \leq C^* < f(G_2)$,

n should have been expanded first!

Graph Search

Two remedies:

•discard the more expensive path when revisiting node

•ensure that first path found to node is optimal (cf. uniform-cost)

Monotonicity/Consistency: a heuristic is consistent iff

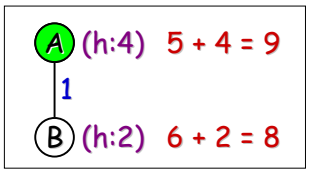
$$\forall \text{nodes } n, n' \in succ(n) : h(n) \leq h(n') + c(n, a, n')$$

Every consistent heuristic is admissible (exercise)

A* using graph search is optimal if heuristic is consistent

Monotonicity

If h consistent then the values of f along a path are non-decreasing:

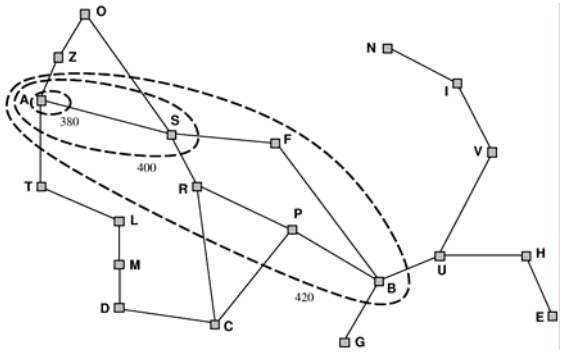


This throws away information !!

We already knew that total cost on this path to the goal is at least 9 (knowledge in node A)

Contours in A*

Within the search space, contours arise in which for the given *f*-value all nodes are expanded.



Contours at f = 380, 400, 420

Completeness and Complexity

Completeness: If a solution exists, A* will find one, provided that (1) every node has a finite number of successor nodes, and (2) there exists a positive constant δ such that every operator has at least cost δ.

→ Only a finite number of nodes *n* with $f(n) \leq f^*$.

Complexity: In the case where $|h^*(n) - h(n)| \leq O(\log(h^*(n)))$, only a sub-exponential number of nodes will be expanded.

Normally, growth is exponential because the error is proportional to the path costs. So, modify to look for suboptimal solutions and allow non-admissible heuristics!

Iterative Deepening A* Search (IDA*)

Idea: A combination of IDS and A*. All nodes inside a contour are searched in a DFS manner

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f- COST limit
          root, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f- COST(root)
  loop do
    solution, f-limit ← DFS-CONTOUR(root, f-limit)
    if solution is non-null then return solution
    if f-limit = ∞ then return failure; end

function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f- COST limit
  inputs: node, a node
          f-limit, the current f- COST limit
  static: next-f, the f- COST limit for the next contour, initially ∞

  if f- COST[node] > f-limit then return null, f- COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
    solution, new-f ← DFS-CONTOUR(s, f-limit)
    if solution is non-null then return solution, f-limit
    next-f ← MIN(next-f, new-f); end
  return null, next-f
```

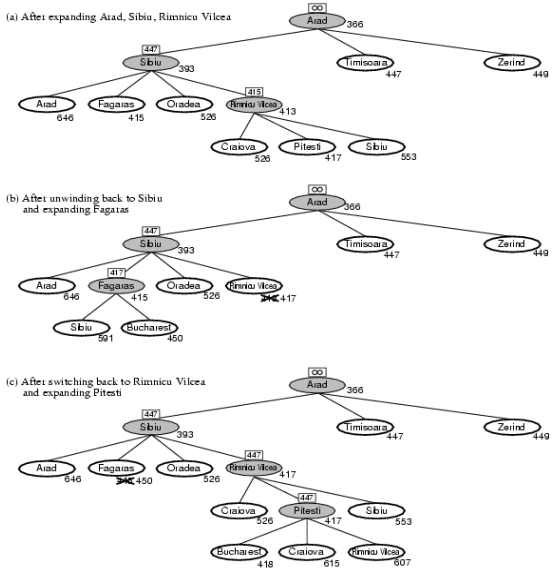
RBFS: Recursive Best-First Search

Avoid re-evaluation of nodes but keep only *O(bd)* nodes in memory

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](state) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        if f[best] > f_limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

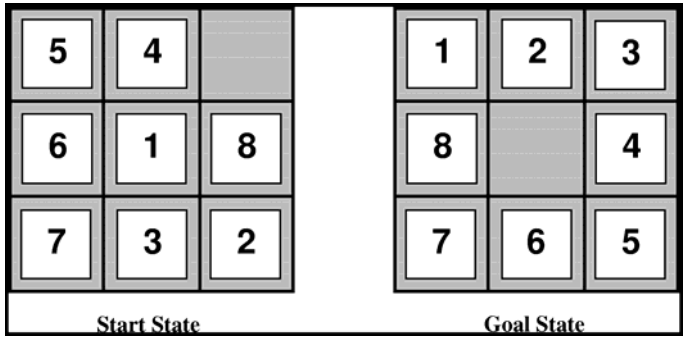
RBFS Example



How to Design a Heuristic

- Simplify the problem (by removing restrictions), creating a **relaxation:**
  - so that it becomes easy to solve
  - usually leading to shorter solutions
  - and making it easy to determine optimal solutions for the relaxation
- Examples:
  - straight line distance
  - simplify movement restrictions in multi-body problems (ignore collisions)
  - ignore negative effects

Example Heuristics



h_1 *= the number of tiles in the wrong position*
h_2 *= the sum of the distances of the tiles from their goal positions (Manhatten distance)*

Empirical Evaluation for IDS vs. A*

- *d* = distance from goal
- Average over 100 instances

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

How to choose among heuristics?

If $\forall n : h_2(n) \geq h_1(n)$ and h_1, h_2 both admissible
Then h_2 dominates h_1 and is better for search

- Holds for our illustration
- The more dominant the heuristic the better it approximates the real cost.
- Therefore, given 2 admissible heuristics,

Define $\forall n : h(n) = \max(h_1(n), h_2(n))$
h will dominate h_1, h_2

Local Search Methods

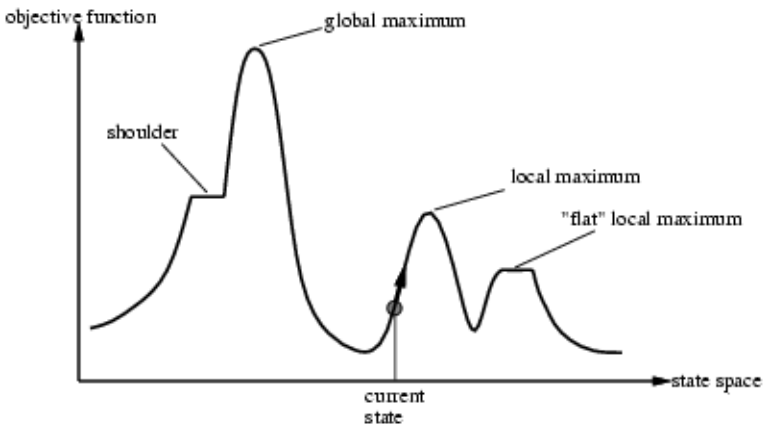
- In many problems, it is not possible to explore the search space systematically.
- If a quality measure (or objective function) for states is given, then **local search** can be used to find solutions.
- Begin with a randomly-chosen configuration/state and improve on it stepwise → **Hill Climbing**.
- Incomplete, but works for very large spaces.
- Has been used for IC design, scheduling, network optimization, … , 8-queens, …

Hill Climbing

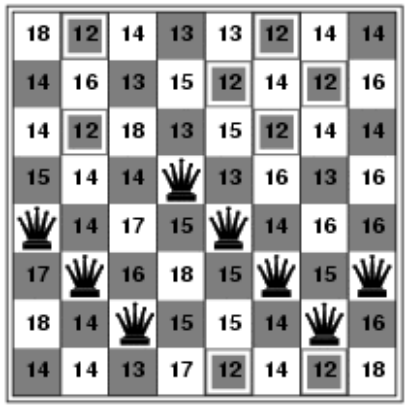
```
function HILL-CLIMBING(problem) returns a solution state
   inputs: problem, a problem
   static: current, a node
           next, a node

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

The Landscape: 2D Example



Example: 8 Queens



An 8-queens state with evaluation value 17 (violations), showing the value for all successors (when moving a queen in its column)

Problems with Local Search Methods

- *Local maxima*: The algorithm finds a sub-optimal solution.
- *Plateaus (shoulders, flat local maxima)*: Here, the algorithm can only explore at random (or exhaustively)
- Ridges: Similar to plateaus.

Solutions:

- *Restart randomly* when no progress is being made.
- "Inject noise" → random walk
- Tabu search: Do not apply the last *n* operators.

Which strategies (with which parameters) prove successful (within a problem class) can usually only empirically be determined.

Simulated Annealing

In the simulated annealing algorithm, "noise" is injected systematically: first a lot, then gradually less.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability 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^(ΔE/T)
```

Has been used since the early 80's for VSLI layout and other optimization problems.

Genetic Algorithms

Evolution appears to be very successful at finding good solutions.

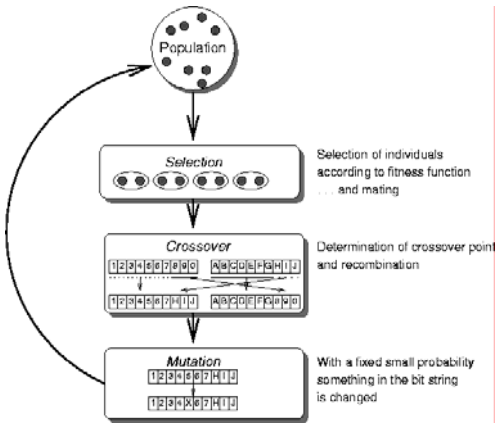
Idea: Similar to evolution, we search for solutions by "cross over", "mutation", and "selection" successful solutions.

Ingredients:

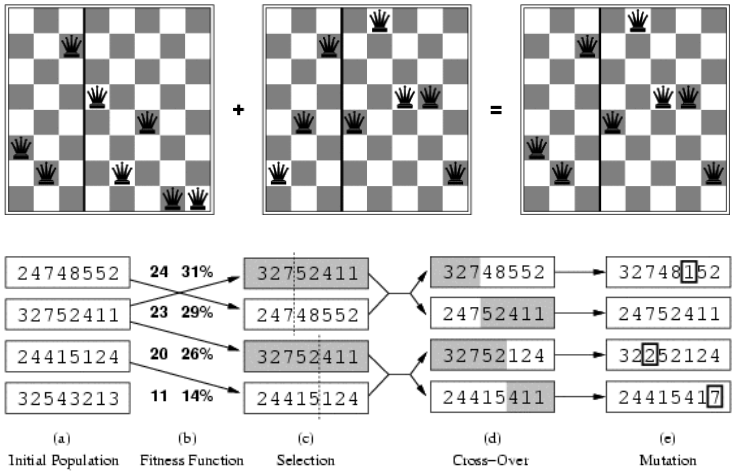
- Coding of a solution into a string of symbols or bit-string
- A fitness function to judge the fitness of configurations
- A population of configurations

Example: 8-queens problem as a chain of 8 numbers. Fitness is judged by the number of non-attacks. The population consists of a set of arrangements of queens.

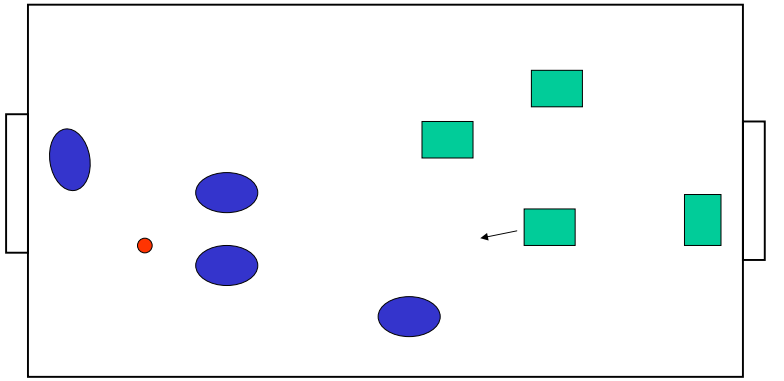
Selection, Mutation, and Crossover



Example: 8-Queens



Case-Study: Path Planning in Robotic Soccer



Possible Approaches

- Reactive: Compute a motor control command based on current observation and goal location
  - try to move towards the goal in a straight line and drive around obstacles
  - May get stuck in local optima
- Deliberative: Generate a (optimal) path plan to the goal location

Simplifying Assumptions

- We do not want to / cannot solve the continuous control problem
- Discretization: 10 cm, $\pi/16$, …
- Movements of other objects are known (or assumed to be irrelevant)
- Adaptation to dynamic change is achieved by continuous re-planning

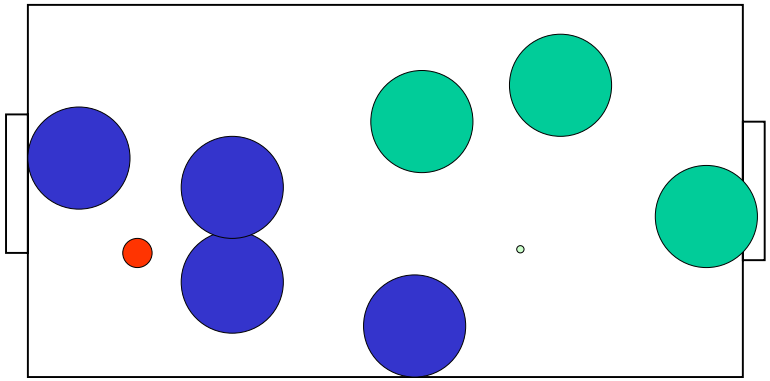
Searching in 5D

- Consider the space generated by
  - location (x,y)
  - orientation (θ)
  - translational velocity (v)
  - Rotational velocity (ω)
- Search in this space using A* in order to find the fastest way to the goal configuration
  - Computationally too expensive even on current hardware (250 msec for a 2m path, while we needed around 10 msec on a 100 MHz Pentium)

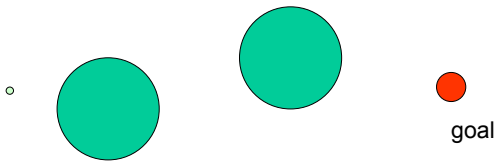
Further simplifications

- Consider only 2D space (location) and search for shortest path (ignoring orientation)
- Assume regular shape: circle
- Reduce robot to point and use obstacle growing
- Apply visibility graph method
- Solve by using A*

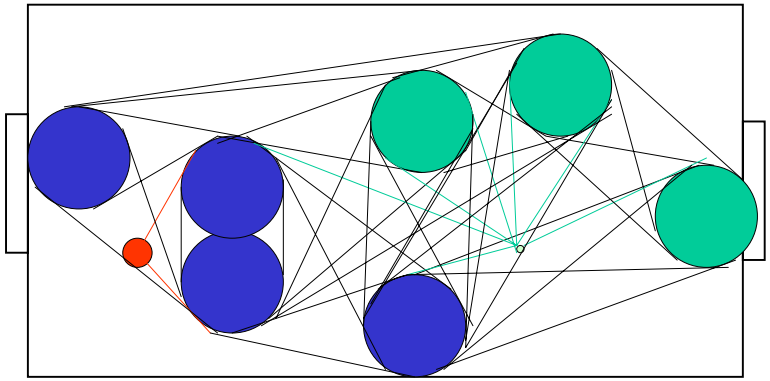
Obstacle Growing



Navigating Around Circles



The Visibility Graph: Compute all common visible tangents



Searching in the Visibility Graph

- The visibility map can now be searched as we can search in a road map using straight line distance as the heuristic estimate
- Note:
  - State space is very limited
  - Optimal solution is not necessarily an optimal solution for the original problem
  - Shortest path is neither the most safe nor the fastest path

Summary (1)

- **Heuristics** focus the search
- **Best-first search** expands the node with the highest worth (defined by any measure) first.
- With the minimization of the evaluated costs to the goal *h* we obtain a **greedy search**.
- The minimization of *f(n) = g(n) + h(n)* **combines uniform and greedy searches**. When *h(n)* is **admissible**, i.e. *h** is never overestimated, we obtain the **A* search, which is complete and optimal**.

Summary (2)

- There are many variations of **A***
- **Local search methods** only ever work on one state, attempting to improve it step-wise.
- **Genetic algorithms** imitate evolution by combining good solutions. General contribution not clear yet.
- There are no turnkey solutions, you always have to try and tweak