

Automatic Planning

Jörg Hoffmann

University of Innsbruck
DERI

Masterstudium Informatik Vertiefungsmodul (WM 9)

Part I. State Space Search



Jörg Hoffmann

Automatic Planning

Automatic Planning
3. State Space Search Algorithms

- ▶ Progression & Regression
- ▶ Search Algorithms
- ▶ Heuristic Functions
- ▶ Informed Search Algorithms
- ▶ Relaxations



Jörg Hoffmann

Automatic Planning



Jörg Hoffmann

Automatic Planning

Search Schemes

Definition

A *search scheme* is a tuple $(S, s_0, succ, Sol)$:

1. the set S of all *search states* $s \in S$,
2. the *start state* $s_0 \in S$,
3. the *successor state function* $succ : S \mapsto 2^S$, and
4. the *solution states* $Sol \subseteq S$.

- ▶ $succ(s)$ is finite for all s
- ▶ Solution paths $s_0 \rightarrow, \dots, \rightarrow s_n \in Sol$ (easily) correspond to solutions to our problem
- ▶ The *search space* is the directed graph implicitly given by s_0 and $succ$



Jörg Hoffmann

Automatic Planning

Progression:

- ▶ Also called **forward search**
- ▶ Start at the initial state, apply transition rules, stop when solution state is reached

Regression:

- ▶ Also called **backward search**
- ▶ Start at the goal formula, step backwards over transition rules that can make the formula TRUE, stop when initial state satisfies the formula

- ▶ **State space:** all world states that are reachable from the initial state by applying transition rules
- ▶ In progression, *search space* coincides with *state space*
- ▶ (Well, most of the time – in Timed Automata, e.g., the state space is continuous and forward search takes place in the space of *regions* of world states)
- ▶ Progression explores only world states that are **reachable** from I ; they may not be **relevant** for G

Progression in STRIPS

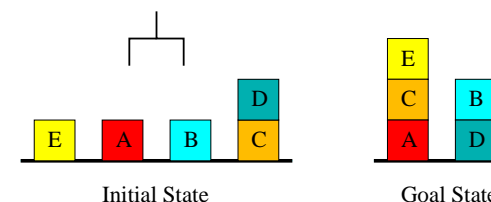
Definition

Let (P, A, I, G) be a STRIPS task. Progression is the tuple $(S, s_0, succ, Sol)$ where

1. $S = 2^P$ is the set of all subsets of P ,
2. $s_0 = I$,
3. $succ : S \mapsto 2^S$ is defined by
 $succ(s) = \{s' \in S \mid \exists a \in A : pre(a) \subseteq s, s' = result(s, \langle a \rangle)\}$
4. $Sol = \{s \in S \mid G \subseteq s\}$

Remember: in STRIPS “shorthand”, fact sets are conjunctions, effect instructions, world states.

Progression in STRIPS, Example Blockworld



- ▶ $s_0 = I = \{on(E, Table), clear(E), \dots, on(C, Table), on(D, C), clear(D), holding(NIL)\}$
- ▶ Applicable actions: $pickup(E), pickup(A), pickup(B), unstack(D, C) \Rightarrow |succ(s_0)| = 4$
- ▶ E.g. $result(s_0, \langle unstack(D, C) \rangle) = \{on(E, Table), clear(E), \dots, on(C, Table), clear(C), holding(D)\}$
- ▶ Initially, only ??? is relevant

- ▶ Search states are **formulas** ϕ ; start with the goal formula
- ▶ For search state ϕ , for all transition rules t , generate a new formula ψ such that ϕ holds when applying t to a world state in which ψ holds
- ▶ “If I have ψ , then, using t , I can achieve ϕ ”
- ▶ “Regress ϕ through t ”
- ▶ When ψ holds in the initial state, we can stop
- ▶ Regression explores only world states that are **relevant** for G ; they may not be **reachable** from I

- ▶ Search states (formulas) represent *sets* of world states
- ▶ E.g. the start state (goal formula) represents the set of goal states
- ▶ Regression is not as non-natural as you might think: if you plan a trip to Thailand, do you start with thinking about the train to Frankfurt?
- ▶ Regression was predominant in Planning until 1998
- ▶ In general, regression involves dealing with arbitrary formulas, and that can be hard...

Regression in STRIPS I

Definition

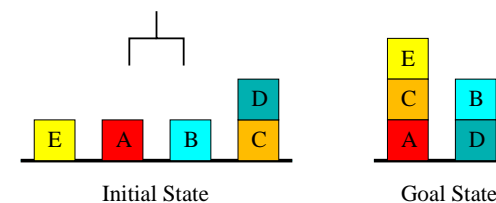
Let P be a set of facts, $s \subseteq P$, and a a STRIPS action. The **regression** $\text{regress}(s, a)$ of s through a is

$$\text{regress}(s, a) = \begin{cases} (s \setminus \text{add}(a)) \cup \text{pre}(a) & \text{add}(a) \cap s \neq \emptyset, \\ & \text{del}(a) \cap s = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the first case, s is said to be **regressible** through a .

- ▶ If $s \cap \text{add}(a) = \emptyset$ then a contributes nothing
- ▶ If $s \cap \text{del}(a) \neq \emptyset$ then we can't use a to achieve $\bigwedge_{p \in s} p$

Regression in STRIPS I, Example Blockworld



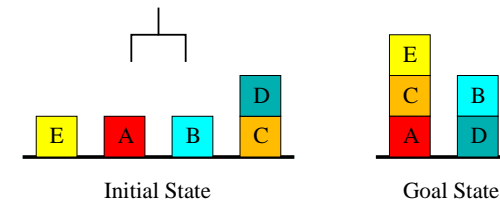
- ▶ $G = \{on(E, C), on(C, A), on(B, D)\}$
- ▶ $stack(B, D) : (\{holding(B), clear(D)\}, \{on(B, D), holding(nil), clear(B)\}, \{holding(B), clear(D)\})$
- ▶ $\text{regress}(G, stack(B, D)) = ???$

Definition

Let (P, A, I, G) be a STRIPS task. Regression is the tuple $(S, s_0, succ, Sol)$ where

1. $S = 2^P$ is the set of all subsets of P ,
2. $s_0 = G$,
3. $succ : S \mapsto 2^S$ is defined by $succ(s) = \{s' \in S \mid \exists a \in A : add(a) \cap s \neq \emptyset, del(a) \cap s = \emptyset, s' = regress(s, a)\}$
4. $Sol = \{s \in S \mid s \subseteq I\}$

Remember: in STRIPS “shorthand”, fact sets are conjunctions, effect instructions, world states.



- ▶ $G = \{on(E, C), on(C, A), on(B, D)\}$
- ▶ G can be regressed through ???
- ▶ $regress(G, stack(C, A)) = ???$

Automatic Planning

3. State Space Search Algorithms

- ▶ Progression & Regression
- ▶ **Search Algorithms**
- ▶ Heuristic Functions
- ▶ Informed Search Algorithms
- ▶ Relaxations

Search Algorithms

- ▶ Start at s_0 and explore $succ(s)$ until $s \in Sol$
- ▶ Differ mainly in terms of the *order* in which they explore the search states
- ▶ **Expand** a search state: insert its successor states into the “open list”
- ▶ **Open list**: those states that have yet to be expanded
- ▶ **Closed list**: there is also the issue of keeping visited states in memory; we’ll skip that here
- ▶ (There may be exponentially many different *paths* to the same search state – e.g., doing n actions in any order.)

```

Bool generic-search(search scheme,queuing-function)
1  open-list := ⟨s0⟩
2  while TRUE do
3      if open-list is empty then return FALSE
4      s := remove-front(open-list)
5      if s ∈ Sol then return TRUE
6      insert-queue(open-list,queuing-function,succ(s))
7  end

```

Algorithms with a “fixed” expansion order

- ▶ Breadth-first search
- ▶ Depth-first search
- ▶ Iterative-deepening Depth-first search

Also called **uninformed**, or brute-force search algorithms

We are interested in the following properties of a search algorithm:

- ▶ Its **time complexity**: in the worst case, how many search states are expanded?
- ▶ Its **space complexity**: in the worst case, how many search states are kept in the open list at any point in time?
- ▶ Is it **complete**, i.e. is it guaranteed to find a solution if there is one?
- ▶ Is it **optimal**, i.e. is it guaranteed to find an optimal solution?

Loop $m = 0, 1, \dots$ around generic search with queuing-function == enqueue-at-front, maximal depth m

- ▶ Time complexity: $O(b^d)$ ($\sum_{m=0}^d \sum_{i=0}^m b^i$)
- ▶ Space complexity: $O(bd)$
- ▶ Complete and optimal
- ▶ Most search nodes lie in the deepest layer anyway, particularly when the branching factor is high
- ▶ E.g. for $b = 10, d = 5$, worst case BrFS generates 111111 search states, 100000 in memory; ID-DFS generates 123456 search states, less than 50 in memory

- ▶ Progression & Regression
- ▶ Search Algorithms
- ▶ **Heuristic Functions**
- ▶ Informed Search Algorithms
- ▶ Relaxations

Definition

Let $(S, s_0, succ, Sol)$ be a search scheme. A **heuristic function** is a function $h : S \mapsto \mathbf{N}_0 \cup \{\infty\}$ from search states into the natural numbers including 0 and the ∞ symbol.

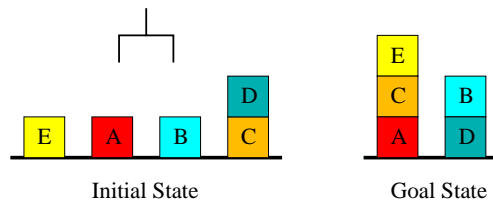
Heuristic functions, or **heuristics**, are efficiently computable functions that estimate a state's "solution distance".

Definition

Let $(S, s_0, succ, Sol)$ be a search scheme. The **solution distance** $sd(s)$ of a state $s \in S$ is the length of a shortest **succ** path from s to a state in Sol , or ∞ if there is no such path.

Computing the real solution distance is as hard as solving the problem itself.

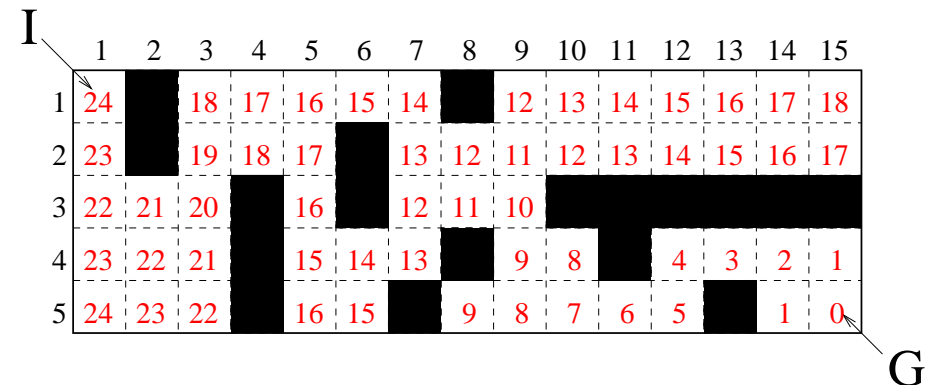
Solution Distance, Example Blocksworld



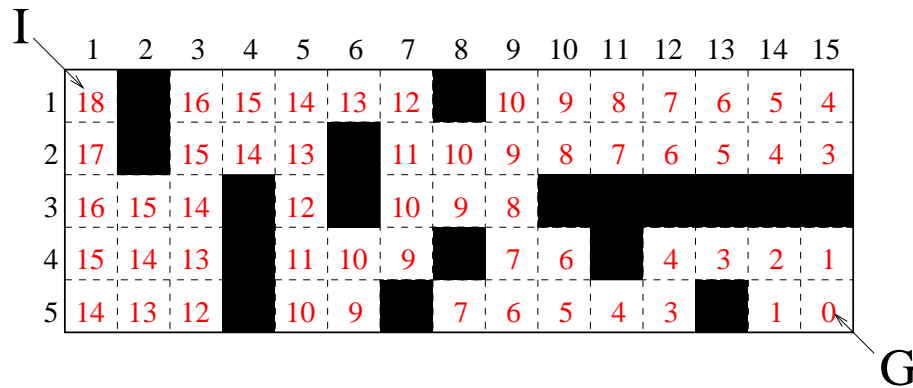
Progression search: $sd(I) = 8$, $sd(result(I, \langle unstack(D, C) \rangle)) = 7$, $sd(s) = ???$ for all other $s \in succ(I)$

Regression search: $sd(G) = 8$, $sd(regress(G, stack(E, C))) = 7$, $sd(regress(G, stack(B, D))) = 7$, $sd(regress(G, stack(C, A))) = ???$

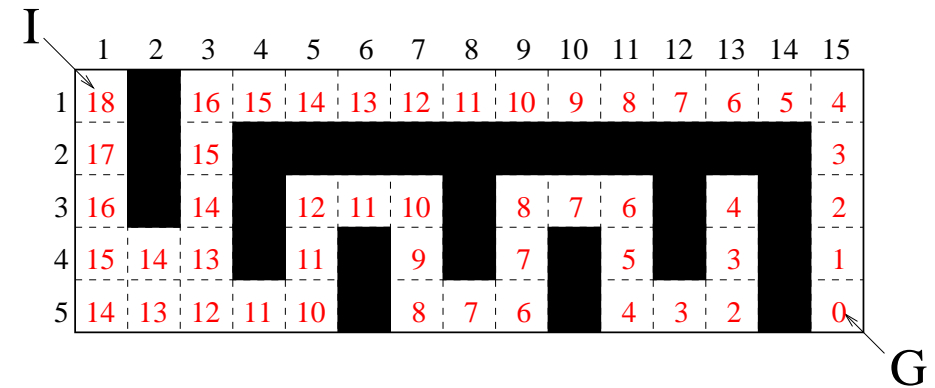
Solution Distance, Example Grid



"Manhattan Distance":



"Manhattan Distance":



Admissible Heuristic Functions

Definition

Let $(S, s_0, succ, Sol)$ be a search scheme. A heuristic function h is **admissible** if $h(s) \leq sd(s)$ for all $s \in S$.

- ▶ Also called *lower bounds*
- ▶ E.g. the Manhattan Distance in the Grid example

Main topic of the lecture:

How to derive (admissible) heuristic functions in general?

Automatic Planning 3. State Space Search Algorithms

- ▶ Progression & Regression
- ▶ Search Algorithms
- ▶ Heuristic Functions
- ▶ **Informed Search Algorithms**
- ▶ Relaxations

Complete/global algorithms:

- ▶ A*
- ▶ Weighted A*, and greedy best-first
- ▶ Iterative Deepening A*

Incomplete/local algorithms:

- ▶ Hill-climbing
- ▶ Simulated Annealing

- ▶ By $g(s)$, denote the number of steps made to s
- ▶ Define the f-cost as $f(s) = g(s) + h(s)$
- ▶ A*: **generic search with queuing-function == enqueue-by-increasing-f-cost**

- ▶ A* is complete: only finitely many states with $g(s) \leq g^*$ where g^* is optimal solution length
- ▶ With admissible heuristics, A* is also *optimal*:

Theorem

Let $(S, s_0, succ, Sol)$ be a search scheme, h an admissible heuristic function. If there is a solution, then the solution path identified by A* is optimal.

Proof.

Sketch: see Blackboard. Details – see [Pearl, "Heuristics", 1984] or [Russel&Norvig]. \square

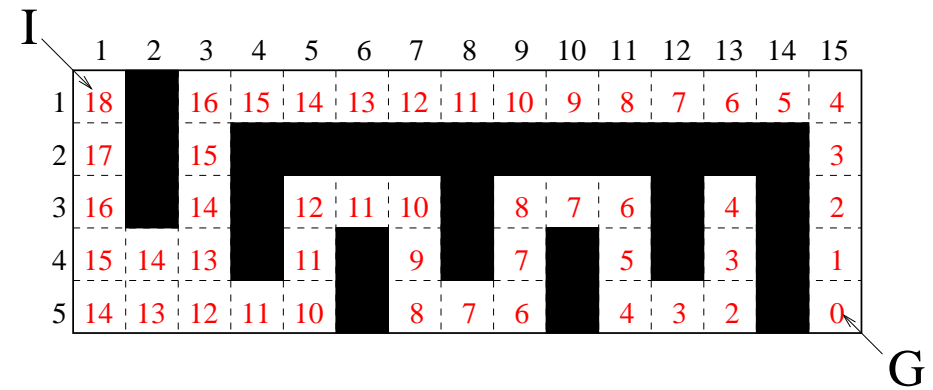
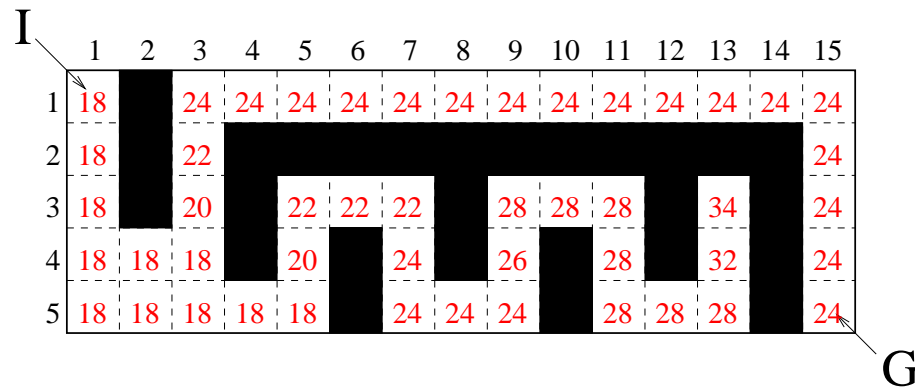
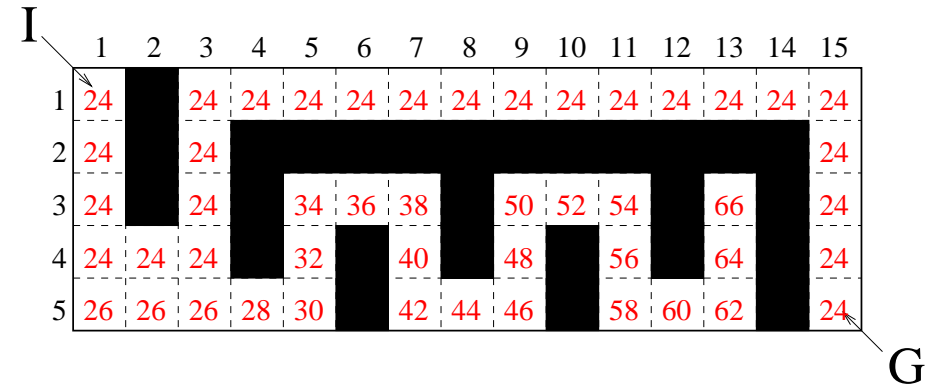
Weighted A*:

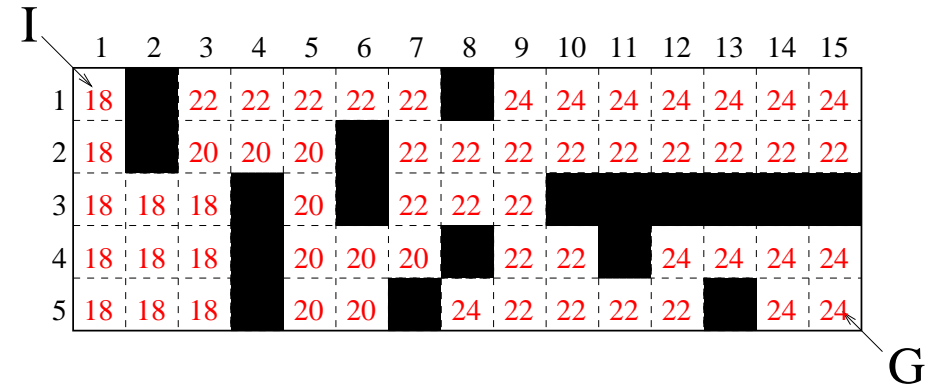
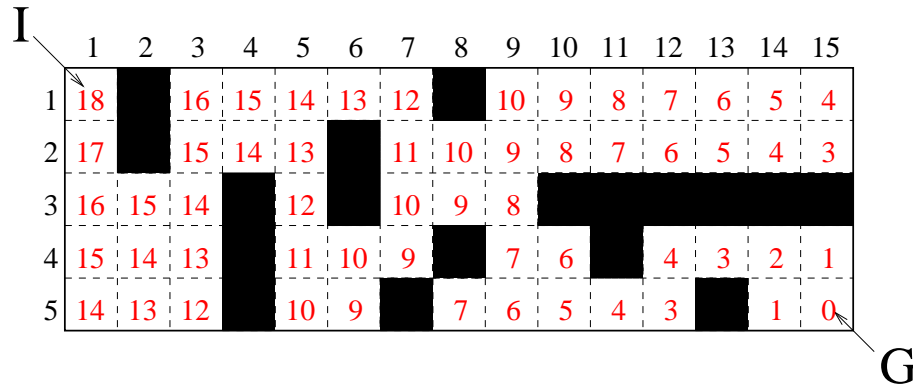
- ▶ Like A*, but f-cost is $f(s) = g(s) + w * h(s)$, for a $w \in \mathbf{N}$
- ▶ $w = 0$ is breadth-first search
- ▶ $w > 1$ is more greedy and may be faster
- ▶ Solution length guaranteed $\leq w * g^*$

Greedy best-first:

- ▶ Like A*, but f-cost is $f(s) = h(s)$
- ▶ Path cost is not considered, pure greediness
- ▶ No guarantee on solution length

- ▶ A* may still be too space-consuming
- ▶ Iterative deepening depth-first search where the length bound increases with the encountered f-costs
- ▶ Precisely:
 - ▶ Initial bound $b_1 := h(s_0)$
 - ▶ In DFS, prune nodes with f-cost $> b$
 - ▶ Bound b_{i+1} is the minimum f-cost of any state pruned in iteration i





Trade Offs in Informed Search

Search time:

- ▶ Reasonable quality heuristic: greedy faster
- ▶ Very bad quality heuristic: A* faster
- ▶ Usually, greedy is faster

Solution quality:

- ▶ The greedier, the (potentially) worse

So: usually, trade-off between runtime and solution quality

*Sometimes the heuristic is so bad that any global search is just like blind search; use a **local** search instead:*

Hill-Climbing

Bool hill-climbing(search scheme, h)

$s := s_0$

while TRUE **do**

if $s \in \text{Sol}$ **then return** TRUE

$s :=$ an element of $\text{succ}(s)$ with minimal h value

end

- ▶ Tie breaking, restarts, ... \Rightarrow tuning may be needed!
- ▶ Intuition: "Try your luck at various places"
- ▶ E.g.: searching satisfying assignments to randomly generated CNF formulas with $h = \#$ unsatisfied clauses

Bool simulated-annealing(search scheme, h)

$s := s_0, T := \text{some initial value} > 0$

while TRUE **do**

if $s \in \text{Sol}$ **then return** TRUE

if $T = 0$ **then return** FALSE

$s' := \text{a random element of } \text{succ}(s)$

if $h(s') < h(s)$ **then** $s := s'$

else $s := s'$ with probability $e^{(h(s)-h(s'))/T}$

 Decrease T

end

- ▶ T is the “temperature” of the process; high T , uphill moves are likely; T towards 0, uphill moves increasingly unlikely
- ▶ Supposed to be similar to physical processes (water turns to ice)

- ▶ Progression & Regression
- ▶ Search Algorithms
- ▶ Heuristic Functions
- ▶ Informed Search Algorithms
- ▶ Relaxations



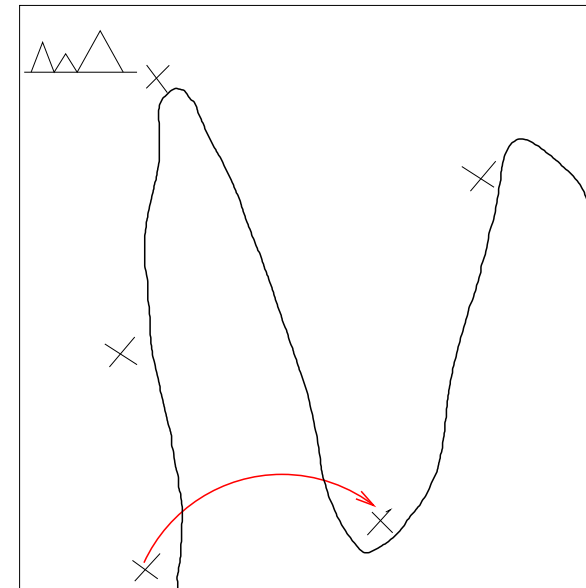
Relaxations

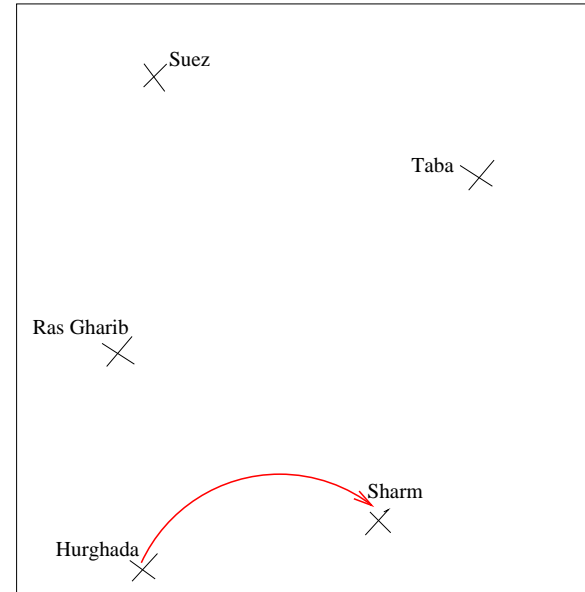
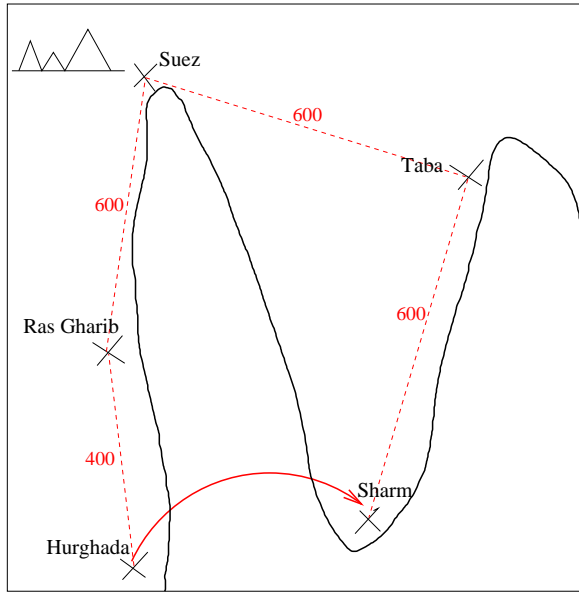
Q: How do we design heuristic functions?

A: Define a **relaxation**, i.e. a simplified version, of the problem. In every search state, solve the relaxed problem, and take the length of the relaxed solution as the heuristic value.

- ▶ Normally, the relaxation would be specific to the application (see next slides)
- ▶ For *automatic* planning, the relaxations are generic on the description language
- ▶ We will spend the next 5 weeks looking at different relaxations of STRIPS (and some extended formalisms)

An Example





“Solve the Relaxed Problem in Every Search State”

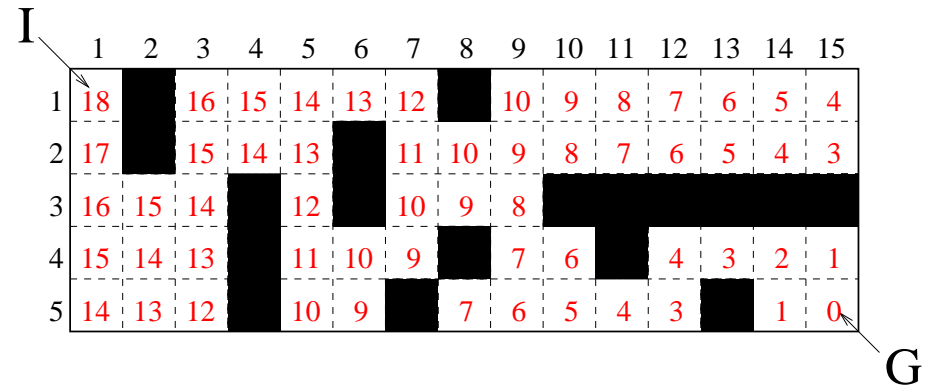
Relaxations, Example Grid: Manhattan Distance ...

So: there are two “search modes” between which we switch back and forth

1. Search states are generated in “real mode”
2. Heuristics are computed in “relaxed mode”

Concretely: define heuristics h that, given a task (P, A, I, G) , solve $relax(P, A, I, G)$ to estimate distance from I to G

1. Forward search: for each search state s , consider the task $relax(P, A, s, G)$
2. Backward search: for each search state s , consider the task $relax(P, A, I, s)$



... is based on the relaxation that **<fill in here>**!

1	2	3	→	1	2	3
6	7	9		4		6
4	8			7	8	9

1	2	3	→	1	2	3
6	7	9		4		6
4	8			7	8	9

- ▶ “A tile can move from square A to square B if A is adjacent to B and B is blank” ⇒ solution distance
- ▶ “A tile can move from square A to square B if A is adjacent to B” ⇒ manhattan distance heuristic h^{MD}
- ▶ “A tile can move from square A to square B” ⇒ misplaced tiles heuristic h^{MT}
- ▶ Here: $sd(s_0) = ???$, $h^{MD}(s_0) = ???$, $h^{MT}(s_0) = ???$

- ▶ An admissible heuristic h **dominates** another admissible heuristic h' if $h(s) \geq h'(s)$ for all s
- ▶ Intuition: h' is “more relaxed” than h
- ▶ Here: sd dominates h^{MD} dominates h^{MT}
- ▶ Later: sd dominates h^+ dominates h^{MD} dominates h^{MT}
- ▶ h^+ is a generic heuristic developed in the STRIPS context!



- ▶ S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, section on Search Strategies, chapter on Informed Search Methods
- ▶ J. Pearl, *Heuristics*, Morgan Kaufmann, 1983

