EXERCISES 4

SEARCH IN COMPLEX ENVIRONMENTS

4.1 Local Search and Optimization Problems

Exercise 4.ASTF

For each of the following assertions, say whether it is true or false and support your answer with examples or counterexamples where appropriate.

a. A hill-climbing algorithm that never visits states with lower value (or higher cost) is guaranteed to find the optimal solution if given enough time to find a solution.

b. Suppose the temperature schedule for simulated annealing is set to be constant up to time N and zero thereafter. For any finite problem, we can set N large enough so that the algorithm is returns an optimal solution with probability 1.

c. For any local-search problem, hill-climbing will return a global optimum if the algorithm is run starting at any state that is a neighbor of a neighbor of a globally optimal state.

d. In a nondeterministic, partially observable problem, improving an agent's transition model (i.e., eliminating spurious outcomes that never actually occur) will never increase the size of the agent's belief states.

e. Stochastic hill climbing is guaranteed to arrive at a global optimum.

f. In a continuous space, gradient descent with a fixed step size is guaranteed to converge to a local or global optimum.

g. Gradient descent finds the global optimum from any starting point if and only if the function is convex.

a. False. Such an algorithm will reach a local optimum and stop (or wander on a plateau).

b. False. If the temperature is fixed the algorithm always has a certain amount of randomness, so when it stops there is no guarantee it will be in an optimal state. Cooling slowly towards zero is the key to optimality.

c. False. The intervening neighbor could be a local minimum and the current state is on another slope leading to a local maximum. Consider, for example, starting at the state valued 5 in the linear sequence 7,6,5,0,9,0,0.

d. True. For any given sequence of actions and observations, removing spurious action outcomes can only eliminate states from the final belief state. (The formal proof is straightforward.)

e. False. Stochastic hill climbing chooses from the uphill moves at random, but, unlike simulated annealing, it stops if there are none, so it still gets trapped in local optima.

f. False. While convexity implies that gradient descent will find a global optimum, the converse is not true. A function can have no local optima without being convex—e.g., in one dimension, it just needs to be strictly decreasing/increasing to the left/right of the global optimum.

Exercise 4.SPCA

Give the name of the algorithm that results from each of the following special cases:

a. Local beam search with $k = 1$.

b. Local beam search with one initial state and no limit on the number of states retained.

c. Simulated annealing with $T = 0$ at all times (and omitting the termination test).

d. Simulated annealing with $T = \infty$ at all times.

e. Genetic algorithm with population size $N = 1$.

a. Local beam search with $k = 1$ is hill-climbing search.

b. Local beam search with one initial state and no limit on the number of states retained, resembles breadth-first search in that it adds one complete layer of nodes before adding the next layer. Starting from one state, the algorithm would be essentially identical to breadth-first search except that each layer is generated all at once.

c. Simulated annealing with $T = 0$ at all times: ignoring the fact that the termination step would be triggered immediately, the search would be identical to first-choice hill climbing because every downward successor would be rejected with probability 1. (Exercise may be modified in future printings.)

d. Simulated annealing with $T = \infty$ at all times is a random-walk search: it always accepts a new state.

e. Genetic algorithm with population size $N = 1$: if the population size is 1, then the two selected parents will be the same individual; crossover yields an exact copy of the individual; then there is a small chance of mutation. Thus, the algorithm executes a random walk in the space of individuals.

Exercise 4.BRIO

Exercise BRIO-EXERCISE considers the problem of building railway tracks under the assumption that pieces fit exactly with no slack. Now consider the real problem, in which pieces don't fit exactly but allow for up to 10 degrees of rotation to either side of the "proper" alignment. Explain how to formulate the problem so it could be solved by simulated annealing.

Despite its humble origins, this question raises many of the same issues as the scientifically important problem of protein design. There is a discrete assembly space in which pieces are chosen to be added to the track and a continuous configuration space determined by the “joint angles” at every place where two pieces are linked. Thus we can define a state as a set of oriented, linked pieces and the associated joint angles in the range $[-10, 10]$, plus a set of unlinked pieces. The linkage and joint angles exactly determine the physical layout of the track; we can allow for (and penalize) layouts in which tracks lie on top of one another, or we can disallow them. The evaluation function would include terms for how many pieces are used, how many loose ends there are, and (if allowed) the degree of overlap. We might include a penalty for the amount of deviation from 0-degree joint angles. (We could also include terms for “interestingness” and “traversability”—for example, it is nice to be able to drive a train starting from any track segment to any other, ending up in either direction without having to lift up the train.) The tricky part is the set of allowed moves. Obviously we can unlink any piece or link an unlinked piece to an open peg with either orientation at any allowed angle (possibly excluding moves that create overlap). More problematic are moves to join a peg and hole on already-linked pieces and moves to change the angle of a joint. Changing one angle may force changes in others, and the changes will vary depending on whether the other pieces are at their joint-angle limit. In general there will be no unique “minimal” solution for a given angle change in terms of the consequent changes to other angles, and some changes may be impossible.

Exercise 4.TSPL

In this exercise, we explore the use of local search methods to solve TSPs of the type defined in Exercise TSP-MST-EXERCISE.

a. Implement and test a hill-climbing method to solve TSPs. Compare the results with optimal solutions obtained from the A* algorithm with the MST heuristic (Exercise TSP-MST-EXERCISE).

b. Repeat part (a) using a genetic algorithm instead of hill climbing. You may want to consult Larranaga *et al.* (1999) for some suggestions for representations.

Here is one simple hill-climbing algorithm:

- Connect all the cities into an arbitrary path.
- Pick two points along the path at random.
- Split the path at those points, producing three pieces.
- Try all six possible ways to connect the three pieces.
- Keep the best one, and reconnect the path accordingly.
- Iterate the steps above until no improvement is observed for a while.

Exercise 4.HILL

Generate a large number of 8-puzzle and 8-queens instances and solve them (where possible) by hill climbing (steepest-ascent and first-choice variants), hill climbing with random restart, and simulated annealing. Measure the search cost and percentage of solved problems and graph these against the optimal solution cost. Comment on your results.

Code not shown.

4.2 Local Search in Continuous Spaces

Exercise 4.SADD

In a continuous state space, a **saddle point** is a point at which all partial derivatives are zero but the point is neither a minimum nor a maximum; instead, it is a minimum in some directions but a maximum in others. Explain how the various continuous-space local search algorithms in the chapter function if started at such a point.

Continuous state space local search requires either gradient or empirical gradient methods.

Empirical methods involve discretizing the continuous state space using some constant δ such that local search algorithms function as before. Nonetheless, too large a choice of δ will change the state space such that it no longer resembles the continuous space and even for a very small δ a solution may be intractable or otherwise infinitesimally distant from the optimum. Thus, as in the discrete case, empirical gradient methods will get stuck on saddle points depending on their dimensionality.

Line search and other local gradient methods attempt to navigate to the local minimum by stepping according to the product of a local gradient and a step size parameter. These methods are affected by ridges or saddle points given that they cannot exhaustively consider the effect of the step size on every possible partial gradient and thus may ”miss” the ridge (or valley) leading to the global maximum (or minimum).

Analytic gradient methods (even for only the local gradient), such as that of Newton-Raphson, should conceivably distinguish saddle points and should therefore be able to follow the appropriate ridge or valley, but in high dimensional spaces such analyses as computing the Hessian matrix become intractable and thus suffer the fate of other local gradient search methods.

4.3 Search with Nondeterministic Actions

Exercise 4.CONP

The AND-OR-GRAPH-SEARCH algorithm in Figure 4.11 checks for repeated states only on the path from the root to the current state. Suppose that, in addition, the algorithm were to store *every* visited state and check against that list. (See BREADTH-FIRST-SEARCH in Figure 3.9 for an example.) Determine the information that should be stored and how the algorithm should use that information when a repeated state is found. (*Hint*: You will need to distinguish at least between states for which a successful subplan was constructed previously and states for which no subplan could be found.) Explain how to use labels, as defined in Section 4.3.3, to avoid having multiple copies of subplans.

See Figure S4.1 for the adapted algorithm. For states that OR-SEARCH finds a solution for it records the solution found. If it later visits that state again it immediately returns that solution.

When OR-SEARCH fails to find a solution it has to be careful. Whether a state can be solved depends on the path taken to that solution, as we do not allow cycles. So on failure

```
function AND-OR-GRAPH-SEARCH($problem$) returns $a\ conditional\ plan,\ or\ failure$
  OR-SEARCH($problem$.INITIAL-STATE, $problem$, [ ])

function OR-SEARCH($state, problem, path$) returns $a\ conditional\ plan,\ or\ failure$
  if $problem$.GOAL-TEST($state$) then return the empty plan
  if $state$ has previously been solved then return RECALL-SUCCESS($state$)
  if $state$ has previously failed for a subset of $path$ then return $failure$
  if $state$ is on $path$ then
      RECORD-FAILURE($state$, $path$)
      return $failure$
  for each $action$ in $problem$.ACTIONS($state$) do
      $plan \leftarrow$ AND-SEARCH(RESULTS($state, action$), $problem$, [$state$ | $path$])
      if $plan \neq failure$ then
          RECORD-SUCCESS($state$, [$action$ | $plan$])
          return [$action$ | $plan$]
  return $failure$

function AND-SEARCH($states, problem, path$) returns $a\ conditional\ plan,\ or\ failure$
  for each $s_i$ in $states$ do
      $plan_i \leftarrow$ 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 $plan_n$]
```

Figure S4.1 AND-OR search with repeated state checking.

OR-SEARCH records the value of *path*. If a state is which has previously failed when *path* contained any subset of its present value, OR-SEARCH returns failure.

To avoid repeating sub-solutions we can label all new solutions found, record these labels, then return the label if these states are visited again. Post-processing can prune off unused labels. Alternatively, we can output a direct acyclic graph structure rather than a tree.

Exercise 4.CONL

Explain precisely how to modify the AND-OR-GRAPH-SEARCH algorithm to generate a cyclic plan if no acyclic plan exists. You will need to deal with three issues: labeling the plan steps so that a cyclic plan can point back to an earlier part of the plan, modifying OR-SEARCH so that it continues to look for acyclic plans after finding a cyclic plan, and augmenting the plan representation to indicate whether a plan is cyclic. Show how your algorithm works on (a) the slippery vacuum world, and (b) the slippery, erratic vacuum world. You might wish to use a computer implementation to check your results.

The question statement describes the required changes in detail, see Figure S4.2 for the modified algorithm. When OR-SEARCH cycles back to a state on *path* it returns a token *loop* which means to loop back to the most recent time this state was reached along the path to it. Since *path* is implicitly stored in the returned plan, there is sufficient information for later

```
function AND-OR-GRAPH-SEARCH($problem$) returns $a$ $conditional$ $plan$, $or$ $failure$
  OR-SEARCH($problem$.INITIAL-STATE, $problem$, [ ])

function OR-SEARCH($state, problem, path$) returns $a$ $conditional$ $plan$, $or$ $failure$
  if $problem$.GOAL-TEST($state$) then return the empty plan
  if $state$ is on $path$ then return $loop$
  $cyclic - plan \leftarrow None$
  for each $action$ in $problem$.ACTIONS($state$) do
    $plan \leftarrow$ AND-SEARCH(RESULTS($state, action$), $problem$, [$state$ | $path$])
    if $plan \neq failure$ then
      if $plan$ is acyclic then return [$action$ | $plan$]
      $cyclic - plan \leftarrow [action \mid plan]$
  if $cyclic - plan \neq None$ then return $cyclic - plan$
  return $failure$

function AND-SEARCH($states, problem, path$) returns $a$ $conditional$ $plan$, $or$ $failure$
  $loopy \leftarrow True$
  for each $s_i$ in $states$ do
    $plan_i \leftarrow$ OR-SEARCH($s_i, problem, path$)
    if $plan_i = failure$ then return $failure$
    if $plan_i \neq loop$ then $loopy \leftarrow False$
  if not $loopy$ then
    return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else ... if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
  return $failure$
```

Figure S4.2 AND-OR search with repeated state checking.

processing, or a modified implementation, to replace these with labels.

The plan representation is implicitly augmented to keep track of whether the plan is cyclic (i.e., contains a *loop*) so that OR-SEARCH can prefer acyclic solutions.

AND-SEARCH returns failure if all branches lead directly to a *loop*, as in this case the plan will always loop forever. This is the only case it needs to check as if all branches in a finite plan loop there must be some And-node whose children all immediately loop.

Algorithm results

Traces of the algorithm running follow. The state is encoded by specifying whether the squares are clean (`c`) or dirty (`d`), with the letter in capitals if the vacuum cleaner is at that spot e.g., `Cd` means the left square is clean, the right is dirty, and the vacuum is on the left square. Indentation follows nesting of function calls. `OR` and `AND` lines are printed at the start of OR-SEARCH and AND-SEARCH, respectively. `trying` and `testing` lines are printed at the top of the loops within OR-SEARCH and AND-SEARCH, respectively.

Exercise 4.NPPA

We can turn the navigation problem in Exercise PATH-PLANNING-EXERCISE into an environment as follows:

- The percept will be a list of the positions, *relative to the agent*, of the visible vertices. The percept does *not* include the position of the robot! The robot must learn its own position from the map; for now, you can assume that each location has a different "view."
- Each action will be a vector describing a straight-line path to follow. If the path is unobstructed, the action succeeds; otherwise, the robot stops at the point where its path first intersects an obstacle. If the agent returns a zero motion vector and is at the goal (which is fixed and known), then the environment teleports the agent to a *random location* (not inside an obstacle).
- The performance measure charges the agent 1 point for each unit of distance traversed and awards 1000 points each time the goal is reached.

a. Implement this environment and a problem-solving agent for it. After each teleportation, the agent will need to formulate a new problem, which will involve discovering its current location.

b. Document your agent's performance (by having the agent generate suitable commentary as it moves around) and report its performance over 100 episodes.

c. Modify the environment so that 30% of the time the agent ends up at an unintended destination (chosen randomly from the other visible vertices if any; otherwise, no move at all). This is a crude model of the motion errors of a real robot. Modify the agent so that when such an error is detected, it finds out where it is and then constructs a plan to get back to where it was and resume the old plan. Remember that sometimes getting back to where it was might also fail! Show an example of the agent successfully overcoming two successive motion errors and still reaching the goal.

d. Now try two different recovery schemes after an error: (1) head for the closest vertex on the original route; and (2) replan a route to the goal from the new location. Compare the performance of the three recovery schemes. Would the inclusion of search costs affect the comparison?

e. Now suppose that there are locations from which the view is identical. (For example, suppose the world is a grid with square obstacles.) What kind of problem does the agent now face? What do solutions look like?

The student needs to make several design choices in answering this question. First, how will the vertices of objects be represented? The problem states the percept is a list of vertex positions, but that is not precise enough. Here is one good choice: The agent has an orientation (a heading in degrees). The visible vertexes are listed in clockwise order, starting straight ahead of the agent. Each vertex has a relative angle (0 to 360 degrees) and a distance. We also want to know if a vertex represents the left edge of an obstacle, the right edge, or an interior point. We can use the symbols L, R, or I to indicate this.

The student will need to do some basic computational geometry calculations: intersection of a path and a set of line segments to see if the agent will bump into an obstacle, and visibility calculations to determine the percept. There are efficient algorithms for doing this on a set of line segments, but don't worry about efficiency; an exhaustive algorithm is ok. If this seems too much, the instructor can provide an environment simulator and ask the student only to program the agent.

To answer (c), the student will need some exchange rate for trading off search time with movement time. It is probably too complex to make the simulation asynchronous real-time; easier to impose a penalty in points for computation.

For (d), the agent will need to maintain a set of possible positions. Each time the agent moves, it may be able to eliminate some of the possibilities. The agent may consider moves that serve to reduce uncertainty rather than just get to the goal.

4.4 Search in Partially Observable Environments

Exercise 4.CONF

In Section 4.4.1 we introduced belief states to solve sensorless search problems. A sequence of actions solves a sensorless problem if it maps every physical state in the initial belief state b to a goal state. Suppose the agent knows $h^*(s)$, the true optimal cost of solving the physical state s in the fully observable problem, for every state s in b. Find an admissible heuristic $h(b)$ for the sensorless problem in terms of these costs, and prove its admissibilty. Comment on the accuracy of this heuristic on the sensorless vacuum problem of Figure 4.14. How well does A* perform?

A sequence of actions is a solution to a belief state problem if it takes every initial physical state to a goal state. We can relax this problem by requiring it take only *some* initial physical state to a goal state. To make this well defined, we'll require that it finds a solution for the physical state with the most costly solution. If $h^*(s)$ is the optimal cost of solution starting from the physical state s, then

$$h(S) = \max_{s \in S} h^*(s)$$

is the heuristic estimate given by this relaxed problem. This heuristic assumes any solution to the most difficult state the agent things possible will solve all states.

On the sensorless vacuum cleaner problem in Figure 4.14, h correctly determines the optimal cost for all states except the central three states (those reached by $[suck]$, $[suck, left]$ and $[suck, right]$) and the root, for which h estimates to be 1 unit cheaper than they really are. This means A* will expand these three central nodes, before marching towards the solution.

Exercise 4.BELS

This exercise explores subset–superset relations between belief states in sensorless or partially observable environments.

a. Prove that if an action sequence is a solution for a belief state b, it is also a solution for any subset of b. Can anything be said about supersets of b?

b. Explain in detail how to modify graph search for sensorless problems to take advantage of your answers in (a).

c. Explain in detail how to modify AND–OR search for partially observable problems, beyond the modifications you describe in (b).

a. An action sequence is a solution for belief state b if performing it starting in any state $s \in b$ reaches a goal state. Since any state in a subset of b is in b, the result is immediate.
Any action sequence which is *n*ot a solution for belief state b is also not a solution for any superset; this is the contrapositive of what we've just proved. One cannot, in general, say anything about arbitrary supersets, as the action sequence need not lead to a goal on the states outside of b. One can say, for example, that if an action sequence solves a belief state b and a belief state b' then it solves the union belief state $b \cup b'$.

b. On expansion of a node, do not add to the frontier any child belief state which is a superset of a previously explored belief state.

c. If you keep a record of previously solved belief states, add a check to the start of OR-search to check whether the belief state passed in is a subset of a previously solved belief state, returning the previous solution in case it is.

Exercise 4.MVLS

On page 128 it was assumed that a given action would have the same cost when executed in any physical state within a given belief state. (This leads to a belief-state search problem with well-defined step costs.) Now consider what happens when the assumption does not hold. Does the notion of optimality still make sense in this context, or does it require modification? Consider also various possible definitions of the "cost" of executing an action in a belief state; for example, we could use the *minimum* of the physical costs; or the *maximum*; or a cost *interval* with the lower bound being the minimum cost and the upper bound being the maximum; or just keep the set of all possible costs for that action. For each of these, explore whether A* (with modifications if necessary) can return optimal solutions.

Consider a very simple example: an initial belief state $\{S_1, S_2\}$, actions a and b both leading to goal state G from either initial state, and

$$\begin{aligned} c(S_1, a, G) &= 3\,; \qquad & c(S_2, a, G) &= 5\,;\\ c(S_1, b, G) &= 2\,; \qquad & c(S_2, b, G) &= 6\,. \end{aligned}$$

In this case, the solution $[a]$ costs 3 or 5, the solution $[b]$ costs 2 or 6. Neither is "optimal" in any obvious sense.

In some cases, there *will* be an optimal solution. Let us consider just the deterministic case. For this case, we can think of the cost of a plan as a mapping from each initial physical state to the actual cost of executing the plan. In the example above, the cost for $[a]$ is $\{S_1{:}3, S_2{:}5\}$ and the cost for $[b]$ is $\{S_1{:}2, S_2{:}6\}$. We can say that plan p_1 *weakly dominates* p_2 if, for each initial state, the cost for p_1 is no higher than the cost for p_2. (Moreover, p_1 *dominates* p_2 if it weakly dominates it *and* has a lower cost for some state.) If a plan p weakly dominates all others, it is optimal. Notice that this definition reduces to ordinary optimality in the observable case where every belief state is a singleton. As the preceding example shows, however, a problem may have no optimal solution in this sense. A perhaps acceptable version of A* would be one that returns any solution that is not dominated by another.

Principle of optimality

To understand whether it is possible to apply A* at all, it helps to understand its dependence on Bellman's (1957) **principle of optimality**: *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.* It is important to understand that this is a restriction on performance measures designed to facilitate efficient algorithms, not a general definition of what it means to be optimal.

In particular, if we define the cost of a plan in belief-state space as the minimum cost of any physical realization, we violate Bellman's principle. Modifying and extending the previous example, suppose that a and b reach S_3 from S_1 and S_4 from S_2, and then reach G from there:

$$\begin{array}{llll} c(S_1,a,S_3)=6\,; & c(S_2,a,S_4)=2\,; & & \\ c(S_1,b,S_3)=6\,; & c(S_2,b,S_4)=1\,.c(S_3,a,G)=2\,; & c(S_4,a,G)=2\,; & \\ c(S_3,b,G)=1\,; & c(S_4,b,G)=9\,. & & \end{array}$$

In the belief state $\{S_3, S_4\}$, the minimum cost of $[a]$ is $\min\{2,2\}=2$ and the minimum cost of $[b]$ is $\min\{1,9\}=1$, so the optimal plan is $[b]$. In the initial belief state $\{S_1, S_2\}$, the four possible plans have the following costs:

$$[a,a]:\min\{8,4\}=4\,;[a,b]:\min\{7,11\}=7\,;[b,a]:\min\{8,3\}=3\,;[b,b]:\min\{7,10\}=7\,.$$

Hence, the optimal plan in $\{S_1, S_2\}$ is $[b,a]$, which does *not* choose b in $\{S_3, S_4\}$ even though that is the optimal plan at that point. This counterintuitive behavior is a direct consequence of choosing the minimum of the possible path costs as the performance measure.

This example gives just a small taste of what might happen with nonadditive performance measures. Details of how to modify and analyze A* for general path-dependent cost functions are give by Dechter and Pearl (1985). Many aspects of A* carry over; for example, we can still derive lower bounds on the cost of a path through a given node. For a belief state b, the minimum value of $g(s)+h(s)$ for each state s in b is a lower bound on the minimum cost of a plan that goes through b.

Exercise 4.VACL

Consider the sensorless version of the erratic vacuum world. Draw the belief-state space reachable from the initial belief state $\{1,2,3,4,5,6,7,8\}$, and explain why the problem is unsolvable.

The belief state space is shown in Figure S4.3. No solution is possible because no path leads to a belief state all of whose elements satisfy the goal. If the problem is fully observable, the agent reaches a goal state by executing a sequence such that $Suck$ is performed only in a dirty square. This ensures deterministic behavior and every state is obviously solvable.

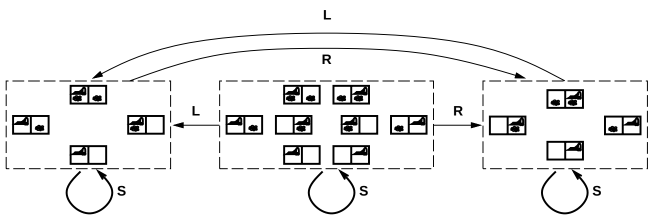

Figure S4.3 The belief state space for the sensorless vacuum world under Murphy's law.

4.5 Online Search Agents and Unknown Environments

Exercise 4.ONOF

Suppose that an agent is in a 3×3 maze environment like the one shown in Figure 4.19. The agent knows that its initial location is (1,1), that the goal is at (3,3), and that the actions Up, $Down$, $Left$, $Right$ have their usual effects unless blocked by a wall. The agent does *not* know where the internal walls are. In any given state, the agent perceives the set of legal actions; it can also tell whether the state is one it has visited before. Assume that the agent's percept is exactly the set of unblocked directions (i.e., blocked directions are illegal actions).

a. Explain how this online search problem can be viewed as an offline search in belief-state space, where the initial belief state includes all possible environment configurations. How large is the initial belief state? How large is the space of belief states?

b. How many distinct percepts are possible in the initial state?

c. Describe the first few branches of a contingency plan for this problem. How large (roughly) is the complete plan?

Notice that this contingency plan is a solution for *every possible environment* fitting the given description. Therefore, interleaving of search and execution is not strictly necessary even in unknown environments.

There are 12 possible locations for internal walls, so there are $2^{12} = 4096$ possible environment configurations. A belief state designates a *subset* of these as possible configurations; for example, before seeing any percepts all 4096 configurations are possible—this is a single belief state.

a. Online search is equivalent to offline search in belief-state space where each action in a belief-state can have multiple successor belief-states: one for each percept the agent could observe after the action. A successor belief-state is constructed by taking the previous belief-state, itself a set of states, replacing each state in this belief-state by the successor state under the action, and removing all successor states which are inconsistent with the percept. This is exactly the construction in Section 4.4.2. AND-OR search can be used to solve this search problem. The initial belief state has $2^1 0 = 1024$ states in it, as we know whether two edges have walls or not (the upper and right edges have no walls) but nothing more. There are $2^{2^{12}}$ possible belief states, one for each set

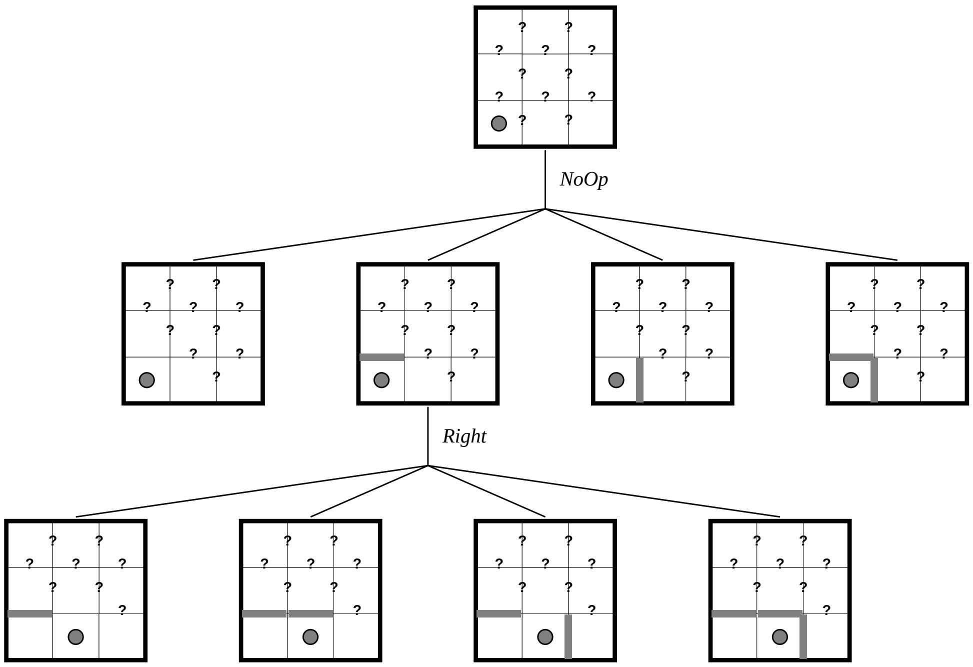

Figure S4.4 The 3×3 maze exploration problem: the initial state, first percept, and one selected action with its perceptual outcomes.

of environment configurations.

We can view this as a contingency problem in belief state space. After each action and percept, the agent learns whether or not an internal wall exists between the current square and each neighboring square. Hence, each reachable belief state can be represented exactly by a list of status values (present, absent, unknown) for each wall separately. That is, the belief state is completely decomposable and there are exactly 3^{12} reachable belief states. The maximum number of possible wall-percepts in each state is 16 (2^4), so each belief state has four actions, each with up to 16 nondeterministic successors.

b. Assuming the external walls are known, there are two internal walls and hence $2^2 = 4$ possible percepts.

c. The initial null action leads to four possible belief states, as shown in Figure S4.4. From each belief state, the agent chooses a single action which can lead to up to 8 belief states (on entering the middle square). Given the possibility of having to retrace its steps at a dead end, the agent can explore the entire maze in no more than 18 steps, so the complete plan (expressed as a tree) has no more than 8^{18} nodes. On the other hand, there are just 3^{12} reachable belief states, so the plan could be expressed more concisely as a table of actions indexed by belief state (a **policy** in the terminology of Chapter 16).

Exercise 4.PPHC

In this exercise, we examine hill climbing in the context of robot navigation, using the environment in Figure **??** as an example.

a. Repeat Exercise PATH-PLANNING-AGENT-EXERCISE using hill climbing. Does your agent ever get stuck in a local minimum? Is it *possible* for it to get stuck with convex obstacles?

b. Construct a nonconvex polygonal environment in which the agent gets stuck.

c. Modify the hill-climbing algorithm so that, instead of doing a depth-1 search to decide where to go next, it does a depth-k search. It should find the best k-step path and do one step along it, and then repeat the process.

d. Is there some k for which the new algorithm is guaranteed to escape from local minima?

e. Explain how LRTA* enables the agent to escape from local minima in this case.

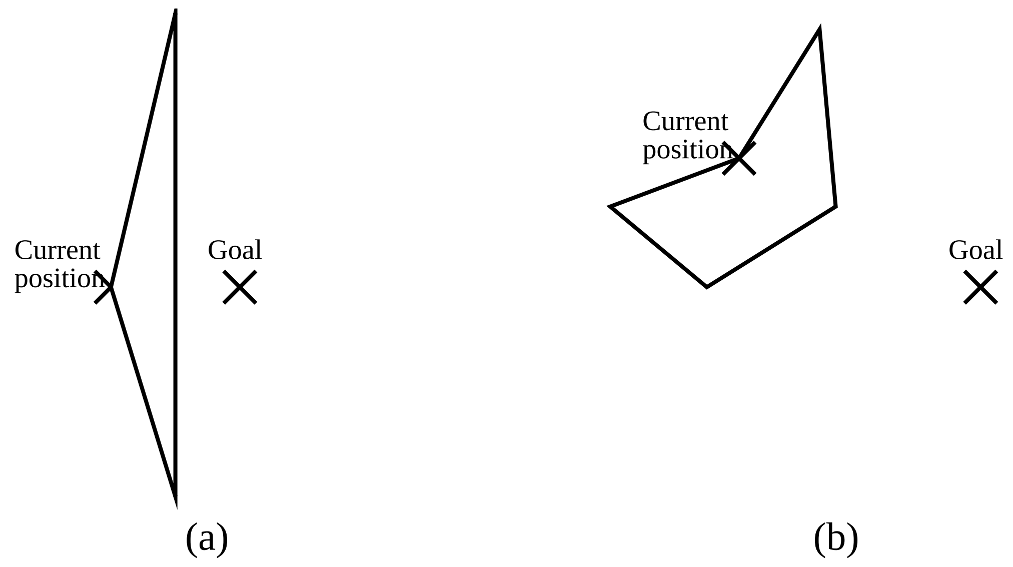

Figure S4.5 (a) Getting stuck with a convex obstacle. (b) Getting stuck with a nonconvex obstacle.

Hillclimbing is surprisingly effective at finding reasonable if not optimal paths for very little computational cost, and seldom fails in two dimensions.

a. It is possible (see Figure S4.5(a)) but very unlikely—the obstacle has to have an unusual shape and be positioned correctly with respect to the goal.

b. With nonconvex obstacles, getting stuck is much more likely to be a problem (see Figure S4.5(b)).

c. Notice that this is just depth-limited search, where you choose a step along the best path even if it is not a solution.

d. Set k to the maximum number of sides of any polygon and you can always escape.

e. LRTA* always makes a move, but may move back if the old state looks better than the new state. But then the old state is penalized for the cost of the trip, so eventually the local minimum fills up and the agent escapes.

Exercise 4.ODFS

Like DFS, online DFS is incomplete for reversible state spaces with infinite paths. For example, suppose that states are points on the infinite two-dimensional grid and actions are unit vectors $(1, 0)$, $(0, 1)$, $(-1, 0)$, $(0, -1)$, tried in that order. Show that online DFS starting at $(0, 0)$ will not reach $(1, -1)$. Suppose the agent can observe, in addition to its current state, all successor states and the actions that would lead to them. Write an algorithm that is complete even for bidirected state spaces with infinite paths. What states does it visit in reaching $(1, -1)$?

Since we can observe successor states, we always know how to backtrack from to a previous state. This means we can adapt iterative deepening search to solve this problem. The only difference is backtracking must be explicit, following the action which the agent can see leads to the previous state.

The algorithm expands the following nodes:
Depth 1: $(0, 0)$, $(1, 0)$, $(0, 0)$, $(-1, 0)$, $(0, 0)$
Depth 2: $(0, 1)$, $(0, 0)$, $(0, -1)$, $(0, 0)$, $(1, 0)$, $(2, 0)$, $(1, 0)$, $(0, 0)$, $(1, 0)$, $(1, 1)$, $(1, 0)$, $(1, -1)$

Exercise 4.CLRT

Relate the time complexity of LRTA* to its space complexity.

The space complexity of LRTA* is dominated by the space required for $result[a, s]$, i.e., the product of the number of states visited (n) and the number of actions tried per state (m). The time complexity is at least $O(nm^2)$ for a naive implementation because for each action taken we compute an H value, which requires minimizing over actions. A simple optimization can reduce this to $O(nm)$. This expression assumes that each state–action pair is tried at most once, whereas in fact such pairs may be tried many times, as the example in Figure 4.22 shows.