Agents in the Brickworld

Gregory D. Weber
Computer Science, Indiana University East
2325 Chester Boulevard, Richmond, Indiana 47374-1289, U.S.A.
gdweber@indiana.edu

Abstract

Brickworld is a simulated environment which has been developed as a testbed for learning and planning—in particular, for learning and using knowledge of causal relations. The environment is both *dynamic*—there are other "agents" whose actions affect "the" agent's performance—and *stochastic*—future states can be predicted only with uncertainty. The task, building and maintaining a wall, has been formulated as a reinforcement learning problem. The ultimate goal of the Brickworld project is to develop a relational reinforcement learning agent that will learn a causal model of the environment representing both its own causal powers and those of the other "agents." The term "agents" is used here in the broadest possible sense, including not only intelligent agents but brute animals and even natural forces such as wind and rain—anything that can be a cause of environmental change.

This paper describes seven implemented agents—a quasi-reactive agent, four non-learning rule-based agents, and two (non-relational) reinforcement learning agents—and compares their performance. The experiments show that a reasonable knowledge representation for the environment results in a state-value function which has local optima, making greedy and ϵ-greedy policies inappropriate. Deeper search is required, leading to problems of inefficiency, which may be alleviated through hierarchical problem spaces. The paper raises questions about the legitimacy of programmer-designed hierarchies in the framework of reinforcement learning and suggests a principled solution.

Introduction

This paper describes a simulated environment, called *Brickworld*, developed as a testbed for learning and planning—in particular, for learning and using knowledge of causal relations—and some agents that have been developed for it. The Brickworld environment is both *dynamic*—there are other "agents" whose actions affect "the" agent's performance—and *stochastic*—future states can be predicted only with uncertainty. The task, building and maintaining a wall, has been formulated as a reinforcement learning (RL) problem; and some of the agents have been designed as preliminary versions of a relational reinforcement learning agent that will learn a causal model of the environment representing both its own causal powers and those of the other "agents." The term "agents" is used here in the broadest possible sense, including not only intelligent agents but brute animals and even natural forces such as wind, sun, and rain—anything that can be a cause of environmental change.

The rest of this paper proceeds as follows. The next section describes the Brickworld environment and formulates the agent's task as a reinforcement learning problem. The section after that specifies the design of an agent, not yet fully implemented, based on ideas from relational reinforcement learning. The next three sections describe several agents that have been implemented and the results of experiments with them. A simple, "quasi-reactive" agent, QRW, is provided as a baseline for comparisons. Four other agents implement aspects of and variations on the relational reinforcement learning agent design, without learning. They use a rule-based model of the environment for predicting the next state, a state-value function based on selected high-level features of states, and various policies (decision procedures) that are, in some sense, as close to greedy as possible. Although these agents do not learn, they establish the basic adequacy of the knowledge representation and decision algorithms to be used by the learning agents. The last two agents add state-value function learning to the design features of the best rule-based agents. They out-perform the non-learning rule-based agents, and perform about as well as the quasi-reactive agent. The final section presents the conclusions of this study. Although the knowledge representation appears to be as adequate as possible, the resulting state-value function exhibits local optima. As a result, "shallow" planning (as in greedy and near-greedy policies) is inadequate; deeper search is required. Search can be done either in the agent's perceptual apparatus or in the decision procedure. The former seems unattractive; the latter leads to the idea of hierarchical problem spaces to limit search. We inquire why the apparently simple QRW agent is so hard to beat; again, the answer suggests hierarchical problem spaces. This

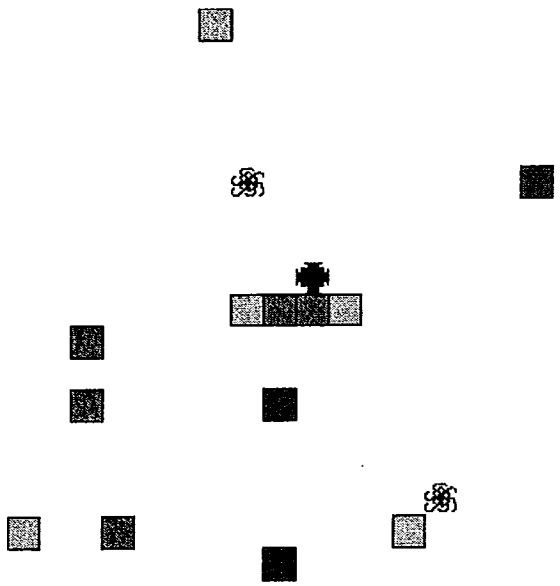

Figure 1: The Brickworld. At center, QRW agent (with four arms) has completed a wall of four bricks. Two raiders have six arms each; boxes represent red, green, and blue bricks.

leads to questions and speculations about the design and justification of hierarchical problem spaces in the reinforcement learning framework.

Brickworld Environment and Task

The Brickworld is an 18 × 18 grid. (See Figure 1.) At any time, each of its 324 cells may be occupied by at most one of the following: an agent; a red, blue, or green brick; or a raider. The agent's task is to build and maintain a wall. Raiders move in a two-dimensional random walk, except that when they are adjacent to red or blue bricks, they eat them. Raiders cannot eat green bricks. Blue bricks "decompose" (self-destruct) after a random period of time, but their supply is replenished by a process of spontaneous generation of blue bricks, which appear at random times and places. Red and green bricks are stable, being neither generated nor destroyed except for the red bricks being eaten. The Brickworld is highly indeterministic, because of the random movement of the raiders and the random generation and destruction of blue bricks.

The agent's task is to build and maintain a wall of bricks. The agent is given a specification of where the wall is to be, for example, a wall occupying the four cells (9, 7), (9, 8), (9, 9), and (9, 10); and at each time step, the agent earns one point of reward for each brick that is "in place" according to the specification. In principle, the task is a continuing task: the agent should build its wall and then maintain it forever after; however, for the sake of having computations that halt, the task is terminated after 200 time steps.[1] The *measure of performance* for the agent is the average reward accumulated per time step during an episode of 200 time steps. In reinforcement learning terms, the *return* which is to be maximized is the average (not discounted) reward received during an episode.

Operator	Description
`move(dir)`	Agent moves to the adjacent cell in direction *dir*.
`pickup(dir)`	Agent picks up object from adjacent cell in direction *dir*.
`putdown(dir)`	Agent puts down object it is holding into adjacent cell in direction *dir*.
`wait`	Agent does nothing.

Table 1: Operators for the Brickworld Agent. Argument *dir* is one of the directions east, west, north, south.

To accomplish its task, the agent may employ the operators `move`, `pickup`, `putdown`, and `wait`, as specified in Table 1.

A long-term goal of the Brickworld project is to understand how agents can learn about and exploit knowledge of causal powers. The causal powers of "the" agent are related to its operators: the power to change its position by moving, and the power to change the position of other objects by picking them up, moving, and putting them down. The causal powers of other agents include: the power of raiders to move and to eat bricks, the power of green bricks to resist being eaten, the power of blue bricks to spontaneously "self-destruct," and the power of the environment itself to spontaneously produce new blue bricks. A causal power is a capacity of an entity to produce or to inhibit change.

Design of a Reinforcement Learning Agent

Reinforcement learning (RL) has been described as any method of learning how to behave so as to maximize a reward function; a reinforcement learner *discovers* which actions are appropriate, rather than *being told* what to do as in supervised learning (Sutton & Barto 1998). Although RL does not prescribe what methods of learning are to be used, certain methods are in common use. The following agent architecture is based on some of them:

1. The agent learns a model, M, of the environment's state-to-state transition function. For any state s and action a, $M(s, a)$ is a prediction of the next state obtained by taking action a in state s. The model

[1] That is, the task is treated as an episodic task even though it is by nature a continuing task. The performance on the episodic task gives an estimate of performance on the continuing task: an agent that performs well for 200 time steps will generally do well for a longer interval as well.

may be deterministic, predicting a single state, or it may predict a probability distribution of states.

2. The agent learns a state-value function V. For any state s, $V(s)$ is an estimate of the *return* (i.e., future rewards) to be obtained by following an optimal policy from state s.
3. The agent makes decisions according to an *ϵ-greedy policy*. Most of the time, the agent makes a "greedy" decision by choosing an action a which maximizes $V(M(s,a))$.[2] However, with a small probability ϵ (e.g., 0.1), the agent will choose a random action. An ϵ-greedy policy is one way of managing the tradeoff between *exploiting* knowledge already gained and *exploring* the environment to improve one's knowledge.

The basic architecture described above is attractive for agents in the Brickworld, because it allows for a combination of shallow planning with probabilistic knowledge. Shallow planning—looking few steps ahead—is appropriate both because of interference from other agents and the intrinsic uncertainty of an indeterministic world. Human problem solving, in order to cope with uncertainty and the limitations of short-term memory, follows a cycle of shallow planning, acting, and replanning (Anderson 1990). The ϵ-greedy decision procedure uses the shallowest possible planning, looking only one step ahead. The use of probabilistic models also matches the indeterminism of Brickworld.

However, most research in RL has used a tabular form of knowledge representation which is inappropriate for Brickworld. That is, M and V are represented as tables with one entry per state in V and one entry per state-action pair in M. Each of the 324 cells in Brickworld can be in at least 6 configurations, so there are at least 6^{324}, or about 1.3×10^{252}, distinct states, far too many for tabular representation. A relational knowledge representation such as predicate logic would be more appropriate. In addition, an important aim of the Brickworld project is for agents to discover and exploit knowledge of causal relations. Such knowledge is naturally expressed in the form of causal laws which are relational expressions. The RL agents for Brickworld will therefore use a relational form of probabilistic rules to represent M, and perhaps V. It is hoped, however, that the value of a state depends on only a few high-level features which can be abstracted from it, such as the number of bricks in place and information concerning the distances between objects. This information would not need a relational representation; it could be encoded as a set of numerical parameters.

These considerations lead to the following refinements of the basic RL architecture described above:

1. The model, M, consists of relational probabilistic rules. The rules could be learned by methods similar to those used in EXPO (Gil 1994) and LIVE (Shen 1994), with extensions to allow for uncertainty and the actions of other agents.
2. The value function, V, is assumed to be a linear combination of high-level features of states, which can be learned by gradient descent in combination with temporal difference methods such as TD(0) or TD(λ) (Sutton & Barto 1998). If linear methods are inadequate, more sophisticated techniques such as neural networks and decision trees can be tried.

The Brickworld project is closely related to recent work by Dzeroski, De Raedt, and Blockeel (Dzeroski, Raedt, & Blockeel 1998). They have developed a relational regression tree inducer, TILDE-RT, and applied it to reinforcement learning for planning in the blocks world. Some key differences are: (1) The blocks world is deterministic; Brickworld is stochastic. (2) The blocks world has a single agent; Brickworld has multiple causal agents (primarily raiders and bricks, but there could also be multiple intelligent agents). (3) Their system learned an optimal state-action-value function (Q function) using Q-learning, with function approximation by relational tree induction. The Brickworld agents will use a relational rule learning algorithm to learn a model of the environment, and use TD(0) or TD(λ) to estimate the optimal state-value function, with function approximation by linear or other methods. These two approaches—learning the state-value function V and learning the state-action-value function Q—are complementary in RL.

The next two sections describe a quasi-reactive agent which will serve as a baseline for performance comparisons, and some "pilot" versions of the RL architecture which has been described here. The pilot versions do not learn, but they incorporate a relational model M, a linear state-value function V, and a variety of decision procedures including ϵ-greedy. They were developed in order to test the adequacy of the basic knowledge representation and decision procedures prior to work on actual model learning and value function learning.

A Quasi-Reactive Agent

The first agent, QRW, is a quasi-reactive wall-building and repairing agent. The main decision algorithm of QRW can be sketched as follows:

Decide-QRW

1. If there are no chinks, wait.
2. Else if I'm holding a brick, go to the nearest chink and put the brick down.
3. Else if there is a brick that is not in place, go after the nearest such brick and pick it up.
4. Else wait.

A brick is "in place" if it is part of the wall. The operations "go to the nearest chink" and "go after the nearest such brick" are implemented by first planning a path to the chink or brick, using best-first search, then taking

[2] If M is a probabilistic model, the agent maximizes the expected value.

Agent	Average Reward			
	Mean	Std. dev.	Std. err.	Range
Idle	0.06	.22	.05	0.00–1.00
Random	0.07	.23	.05	0.00–1.00
QRW	3.33	.30	.07	2.57–3.77
RBW-1	2.48	.96	.21	0.46–3.65
RBW-2	2.60	.77	.17	0.92–3.70
RBW-3	3.09	.21	.05	2.80–3.59
RBW-4	3.21	.29	.06	2.66–3.66

Table 2: Average reward accumulated by seven agents. For each agent, the table shows descriptive statistics for 20 test episodes of 200 time steps each. Each episode begins with 1 agent, 2 raiders, and 12 bricks (5 red, 5 green, 2 blue). For all but the random agent, the mean performance is significantly greater than the mean performance of the idle agent as judged by a one-sided, matched pairs **t** test with significance level 5%. For all of the RBW agents *except* RBW-4, the mean performance is significantly different from the mean performance of QRW, as judged by a two-sided, matched pairs **t** test with significance level 5%.

the first step in the plan. (This limited amount of planning, using only the `move` operator, seems necessary for effective performance.) After each step, the decision algorithm is called again. If circumstances warrant, QRW replans for movement toward the same chink or brick; but changed conditions may call for moving toward a different chink or brick or for waiting, so QRW is also able to abandon the plan. Because of its limited use of planning, QRW is called a "quasi-reactive" agent.

QRW is an effective agent for its task and environment. Table 2 shows the average reward per time step accumulated by QRW and several other agents. In this section we are concerned only with QRW and two of the other agents, which are "straw men": an idle agent, which does nothing, and a random agent, which performs randomly chosen actions. QRW obviously does better than the idle and random agents.

Although QRW is an effective agent, it is not an optimal agent. It has no preference for green bricks, which are immune from being eaten, over red or blue, and no preference for red bricks, which are immune from self-destruction, over blue. Introducing a preference for better bricks would improve QRW's performance, but it is not obvious how this preference should be implemented in a reactive agent. It is clear enough that if the distances are equal, a green brick is better than a red one, but what if the green brick is farther away? A planning agent is probably better able to handle the trade-off between the inherent quality of a green brick and the accident of its being farther off than a red brick. Although QRW is not optimal, it is a reasonable "baseline" agent for performance comparisons, because it balances simplicity with effectiveness: developing a better quasi-reactive agent would require significantly more work.[3]

move-empty If I am at cell (r_1, c_1) and I am holding nothing and I apply the operator `move(dir)` and (r_2, c_2) is the adjacent cell in direction `dir` and there is nothing in (r_2, c_2), then I will be at (r_2, c_2), and I will not be at (r_1, c_1).

move-holding If I am at cell (r_1, c_1) and I am holding a brick b and b is at (r_1, c_1) and I apply the operator `move(dir)` and (r_2, c_2) is the adjacent cell in direction `dir` and there is nothing in (r_2, c_2), then I will be at (r_2, c_2), b will be at (r_2, c_2), I will not be at (r_1, c_1), and b will not be at (r_1, c_1).

pickup-normal If I am at (r_1, c_1) and I am holding nothing and I apply the operator `pickup(dir)` and (r_2, c_2) is the adjacent cell in direction `dir` and there is a brick b at (r_2, c_2), then I will be holding b, b will be at (r_1, c_1), and b will not be at (r_2, c_2).

putdown-normal If I am at (r_1, c_1) and I am holding a brick b and b is at at (r_1, c_1) and I apply the operator `putdown(dir)` and (r_2, c_2) is the adjacent cell in direction `dir` and there is nothing at (r_2, c_2), then I will be holding nothing, b will not be at (r_1, c_1), and b will be at (r_2, c_2).

Table 3: Rules for the RBW agents.

Some Rule-Based Planning Agents

The next four agents, RBW-1, -2, -3, and -4, are "rule-based wallbuilder agents," based on the RL agent architecture described above, with the differences that M and V are hand-crafted, not learned, and with a variety of decision procedures as alternatives to the ϵ-greedy procedure.

The model M is expressed in the form of deterministic rules, which are paraphrased in Table 3. It is only a crude approximation of the true state-to-state transition function for the Brickworld environment, because it ignores the activities of raiders and the creation and destruction of bricks.

The state-value function V is based on the intuitions that (1) more bricks in place are better, and (2) for each brick that is not in place, it is better if it can be placed into a chink more easily. Let $N(s)$ be the number of bricks in place in state s, and let $E(x, s)$ be an estimate of the effort required to put brick x in place, starting in state s. Then

$$V(s) = aN(s) + b(1/\min(E(x, s))), \qquad (1)$$

[3] Additional improvements are possible. QRW uses a simple greedy strategy of getting the nearest brick and taking it to the nearest chink. Its performance could probably be improved by using a more sophisticated greedy strategy: going for the brick which minimizes the sum of the distances from agent to brick and from brick to chink. This might be improved further by globally minimizing the path length of a plan to put all bricks in place.

Figure 2: Local Optima in Brickworld. A greedy agent (RBW-1) carrying a brick is stuck in a local maximum of the state-value function. The agent needs to move to a cell adjacent to the chink, but two bricks prevent it from moving along a direct path. All unblocked paths lead first to a state with lower value, and at least three consecutive moves are required to get to a better state.

where a and b are constants, and the minimization is computed over all bricks x which are not in place in state s.[4] The estimated effort to place a brick is computed as follows. Let $D(x, y, s)$ be the Manhattan distance between objects x and y in state s. Let r be the robot (our agent), and let $H(r, x, s) = 1$ if r is holding x in state s, 2 otherwise. Then

$$E(x, s) = D(r, x, s) + \min(D(x, y, s)) + H(r, x, s), \quad (2)$$

the minimization being computed over all chinks y. The first term is the distance the robot needs to travel to pick up x. The second term is the distance from x to the nearest chink. The addition of $H(r, x, s)$ represents the need to pick up x, if the robot is not already holding it, and to put x down; it also prevents division by zero in (1). Because of the nested minimization, the value function is relatively hard to compute, taking $O(n^2)$ time, where n is the number of objects in the environment.

All of the RBW agents described here share the same model M and the same value function V. We turn now to their differences in design and performance.

RBW-1 is a greedy planner. In each decision cycle, it evaluates the state which is predicted to result from each possible action, looking only one step ahead, and chooses an action which maximizes the value of the predicted resulting state, breaking ties randomly. RBW-1 behaves quite effectively at first. But after a while, it usually starts to act very strangely: when it has built part of a wall and is carrying the next brick to the wall, if it reaches the cell just beyond either of the two endpoints of the wall, it gets stuck. (See Figure 2) It will perform all kinds of apparently silly actions, doing anything but move away. The reason for this peculiar behavior is that the two cells just beyond the endpoints of the wall are local maxima of the value function. Any movement away from these cells leads to a state with a lower estimated value than the present state.

RBW-2 is an ϵ-greedy agent with $\epsilon = 0.1$. The rationale for an ϵ-greedy decision procedure, in reinforcement learning, is to force the agent to explore the whole space of states and actions even though the value function it has learned so far would restrict its behavior more narrowly if a strictly greedy policy were followed. It seemed that the same technique, by forcing the agent to behave randomly at times, would enable it to "jump out" of the local optima. However, RBW-2 does not perform much better than RBW-1. Analysis shows that at least three moves away from the local optima are required to get far enough away so that a resumption of the greedy policy does not draw the agent immediately back in. With $\epsilon = 0.1$, the chance of three consecutive moves away from the local optimum is less than 0.001, too small to make the agent an effective performer for this task.

RBW-3 is a planner which uses depth-limited iterative deepening (DLID) search. In DLID, the agent performs depth-limited depth-first search with maximum depths of 1, 2, and so on up to some fixed limit N. In RBW-3, $N = 3$, and the goal of the search is any state with a better value than the current state. When RBW-3 finds a plan, it saves the plan and executes all steps of the plan before replanning.[5] The combination of a stored plan and a search depth of 3 enables RBW-3 to break out of local optima, and consequently it performs much more smartly than its predecessors. It is still reasonably fast (average time 0.1 second per decision) and usually finds a 1-step plan, but there are noticeable pauses (1–2 seconds) when deeper search is required.[6] In its original form, it would search to maximum depth when the wall was complete, only to find that there was no way to improve on the current state. In the present form, when there are no chinks, RBW-3 just waits without planning.

RBW-4 is a planner which uses an optimizing form of iterative deepening, IDOPT search. Like RBW-3, it just waits without planning if there are no chinks. It differs from RBW-3 as follows:

1. Whereas DLID finds any state at depth d which improves upon the present state, no matter how slightly, IDOPT finds the *best* state at depth d. Thus where RBW-3 might find a 3-step plan leading to any improved state, RBW-4 would find an optimal 3-step plan.

2. Instead of always using the stored plan, RBW-4 checks to see if there is a better plan of the same length. This does not increase the search time significantly, because the stored plan's length is always one less than the previous search depth (since the first action has been executed and removed from the plan).

3. Instead of always starting with a depth of 1, RBW-4 starts with the length of the stored plan, if it is non-empty.

[4] As implemented, $a = 1$ and $b = 1/2$.

[5] Without a saved plan, the next search, after a step away from a local optimum, would return a one-step plan that led directly back to it.

[6] The Brickworld environment simulator is implemented as a server in STk (Gallesio 1999). Agents run as clients in Petite Chez Scheme (Cadence Research Systems 1999). Both server and client run on the same 233-MHz AMD K-6 processor, with 64 MB RAM, under Linux.

The decision algorithms for RBW-4 are as follows:

Decide-RBW-4

1. If there are no chinks, execute the default action.
2. Else call IDOPT(initdepth, maxdepth), where initdepth is the length of the stored plan, if it is non-empty, initdepth = 1 otherwise.

IDOPT(lower, upper)

1. Let d = lower. If d > upper, return the single-step plan consisting of the default action.
2. Else find the best plan of length d, and return it if it is better than the present state.
3. Else return IDOPT(d + 1, upper)

For RBW-4, the default action is wait.

RBW-4 achieves the best performance of all of the RBW agents, and although its measured performance is slightly lower than that of QRW, the difference is not statistically significant. Like RBW-3, its average time per decision is about 0.1 second, and there are noticeable pauses, up to about 6 seconds, during deep planning. Details of the agents' performance are shown in Table 2.

Some Reinforcement Learning Agents

Several reinforcement learning agents have been implemented, using the same basic decision procedure as RBW-4, except that with probability ϵ the agent takes a random exploratory action. The RL agents use a six-featured value function which refines equation (1) by providing color-specific information, enabling the agents to learn some preferences for specific colors of bricks:

$$\begin{aligned} V(s) = \; & a_r N_r(s) + b_r(1/\min(E_r(x,s))) + \\ & a_g N_g(s) + b_g(1/\min(E_g(x,s))) + \\ & a_b N_b(s) + b_b(1/\min(E_b(x,s))), \end{aligned} \tag{3}$$

where the subscripts r, g, b designate the three colors of bricks. The coefficients are learned by using linear gradient descent, with TD(0) updates as the target values.

Two of the relatively successful RL agents will be described here. Both use a diminishing learning step size parameter $\alpha = 1/k$, where k is the number of learning steps taken so far. For one of the agents, RLA-2, the default action is `wait`; for the other, RLA-1RB, the default action is to take a random exploratory action. (The default action is the one taken when the planner can find no plan to improve on the present state.)

Both agents were trained and tested using the following experimental procedure. The untrained agent is tested, with learning off, using the same set of 20 test problems previously used for the QRW and RBW agents. Then the agent is incrementally trained and retested with 10, 20, 30, 40, and 50 training episodes. Each episode is 200 time steps. The training episodes use problems not in the test set. The agent's performance, after each increment of training and testing, is compared with its own performance before training, and with the performance of QRW and RBW-4, using matched-pairs, two-tailed **t** tests, with significance level 5%.

Agent	Average Reward after N Training Episodes			
	$N = 0$	$N = 10$	$N = 20$	$N = 50$
QRW	3.33			
RBW-4	3.21			
RLA-1RB	3.25	3.30	3.31	3.31
RLA-2	3.15	3.28	3.30	3.30

Table 4: Performance of two reinforcement learning agents. QRW and RBW-4 are shown for comparison.

The results are shown in Table 4. Each of the RL agents, after 20 training episodes (4,000 training steps), shows performance better than and significantly different from RBW-4's, but not significantly different from QRW's performance. Each RL agent's performance is also better than and significantly different from its own pre-training performance. After 20 training episodes, performance levels off, with no further improvement evident through 50 training episodes (10,000 training steps), although the coefficients in the V function continue to change slightly. Both agents begin with a coefficient vector $(a_r, a_g, a_b, b_r, b_g, b_b) = (1.0, 1.0, 1.0, 0.5, 0.5, 0.5)$. After 20 training episodes, RLA-1RB's coefficient vector has become $(8.9, 10.3, 1.2, 1.2, 1.4, 0.7)$, indicating a preference for having green, red, and blue bricks in place, in that order, and similarly for having green, red, and blue bricks "close to" in place, in that order. RLA-2's learned coefficients are similar. We can sometimes see these preferences in the agent's behavior; for example, it will pick up a green brick rather than an equally close blue brick. However, it is not a strong preference: the agent has been seen picking up a blue brick, rather than a green one only two units farther away.

Conclusions

We have seen three families of agents in the Brickworld. The quasi-reactive agent QRW is among the best. Among the non-learning rule-based agents, only RBW-4 does not perform significantly worse than QRW. Of the two RL agents, both learn color preferences which improve their performance, making them significantly better than RBW-4, and not significantly different from QRW. But why are the RL agents not *better* than QRW? There are at least three possibilities:

1. Poor choice of features. The features may be poor predictors of V.
2. Poor choice of functional form. V may be a non-linear function of the features.
3. Lack of hierarchical structure. QRW's main decision algorithm uses higher-level operators, such as "go to the nearest chink," which are implemented using best-first search in a limited problem space. This

combination of abstract operators, problem spaces, and a powerful search technique may be hard to beat.

In fact, the six features used in (3) are *not* entirely adequate, yet it would be difficult to find a more satisfactory set of features, and consequently we are faced with problems of local optima, requiring search, leading to problems of efficiency, for which the hierarchical organization of problem spaces may be a solution. However, the legitimacy of such hierarchical organizations within the RL framework is questionable.

The presence of local optima in the state-value function creates challenging problems for the design of a reinforcement learning agent. If the knowledge representation is adequate, i.e., if M and V are based on appropriate features of states, then the ϵ-greedy policy is inadequate. Now, the features in V represent information that seems necessary but insufficient to determine the exact, *true* value of a state. Certainly, the number of bricks in place is relevant, and the distances from agent to brick to chink are relevant. Moreover, distance information is often important for causal inference—and not just in Brickworld (Pazzani 1991). The problem here is that there are two positions for the agent holding a brick which have the same chink-distance relations but different values. They have different values because, from one position, the agent faces an obstacle on the path to the chink, so that it must take an indirect path. The only way to discover the presence of an obstacle is to plan a path, which requires search. Thus, improving the V function would require a more expensive computation to deliver more adequate features—in effect, putting the search into the agent's (high-level) perceptual apparatus, which delivers features for the V function. On the other hand, if search is not built into the perceptual system, it must be provided in the decision procedure.

Building search into perception does not seem a natural or attractive solution. The concepts built into the V function are already two levels higher than the raw facts or sense data which are the basis of perception. These raw facts have the form `(type id color row column holding)`, for example, `(agent agent1 black 5 4 brick19)`. These may be considered "first-level" concepts. Distance relations between objects, such as `(distance brick22 chink3 5)`, are second-level facts, computed from the raw facts.[7] Minimum distance information, such as `(easiest-brick-chink agent1 brick23 chink4 12)`, would be a third-level fact, computed from the second-level facts. It is this kind of information which is required for computing the value function V according to (1) or (3). The level of conceptual sophistication required here is already high: could a constructive induction system plausibly come up with such concepts? How can such concepts be explained? Adding path search to the picture would make it even more difficult.

On the other hand, search for path planning can use a restricted set of operators (the four `move` operators), so it can be far less costly than search in the decision procedure, which uses the full set of 13 operators. For example, an exhaustive search to depth 3 requires examining $13^3 = 2197$ plans for the full set, but only $4^3 = 64$ plans for the restricted set. Unrestricted search is expensive. A hierarchy of goals and subgoals, with problem spaces that restrict search to a subset of the operators, could alleviate this problem. Such methods have been studied in the context of reinforcement learning, but only with human-engineered decompositions of goals into subgoals and problem spaces (Dietterich 1998; Sutton, Precup, & Singh 1998).

However, "telling" the system what subgoals and problem spaces to use seems to be "cheating" from the point of view of reinforcement learning, where the aim is for the agent to learn what to do without being told. For example, programming the system to use a "movement planning" problem space with only the four `move` operators amounts to telling the system *not* to use the other nine operators. It is not quite as bad as saying, "In state s_{24}, you should move east"; still, it seems that the ideal in RL should be to let the system discover which operators are worth considering, rather than telling it. Yet, the reality of present-day reinforcement learning is that *tabula rasa* learning methods don't scale up to complex problems without tricks such as hierarchies of problem spaces, shaping (training that progresses from easy to hard problems), local reinforcement (rewards for getting closer to goals), and even imitation (which doesn't seem to be RL at all!) (Kaelbling, Littman, & Moore 1996).

Similarly, telling the system to cut off search at a depth of 3, and telling the system not to search at all when the wall is complete, seem to be "cheating" in RL. They could also be poor design choices. After all, there is no guarantee that a depth of 3 will always be sufficient for Brickworld or related environments; and it is simply false that the agent can never be doing anything useful when it has a complete wall.

Stephen Pepper has suggested that purposive behavior emerges from reactive, "chain-reflex" behavior when some links of the chain drop out and have to be filled in by learning through trial and error (Pepper 1970). If this is so, a principled and useful strategy for designing hierarchies might be to start with "reactive agents," such as QRW, remove some or all of the control at the top level (but keep the lower-level implementations of the top-level operators), and let RL fill in the gaps. Then do the same thing, recursively, at the lower levels. In the real world, chain reflexes are developed through evolution. Following nature's example, we could take this principled strategy a step further by starting with a genetic algorithm, evolving the reactive agent, then breaking down the chain reflexes and applying reinforcement learning.

[7] They would still be second-level facts, of course, if the raw sense data reported relative locations, i.e., the directions and distances of objects from the robot, instead of the objects' coordinates in some absolute reference frame.

Future research on Brickworld agents should be directed towards learning the probabilistic model M, exploring the use of additional features and alternative functional forms (neural networks, decision trees, etc.) for the state-value function V, and using hierarchical methods for problem solving. The open questions involving hierarchical methods include where to search (i.e., in "perception" or in the decision procedure), how deeply to search, and how to design and justify hierarchical problem spaces.

References

Anderson, J. R. 1990. *The Adaptive Character of Thought.* Lawrence Erlbaum Associates.

Cadence Research Systems. 1999. Chez Scheme. http://www.scheme.com/.

Dietterich, T. G. 1998. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 118–126.

Dzeroski, S.; Raedt, L. D.; and Blockeel, H. 1998. Relational reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning.*

Gallesio, E. 1999. STk. http://kaolin.unice.fr/STk/.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*, 87–95.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

Pazzani, M. 1991. A computational theory of learning causal relationships. *Cognitive Science* 15:401–424.

Pepper, S. 1970. *The Sources of Value.* University of California Press.

Shen, W.-M. 1994. *Autonomous Learning from the Environment.* Computer Science Press.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction.* MIT Press.

Sutton, R.; Precup, D.; and Singh, S. 1998. Intra-option learning about temporally abstract actions. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 556–564.