

Project 3a: Domination

Robust, state-dependent, event-driven robot applications

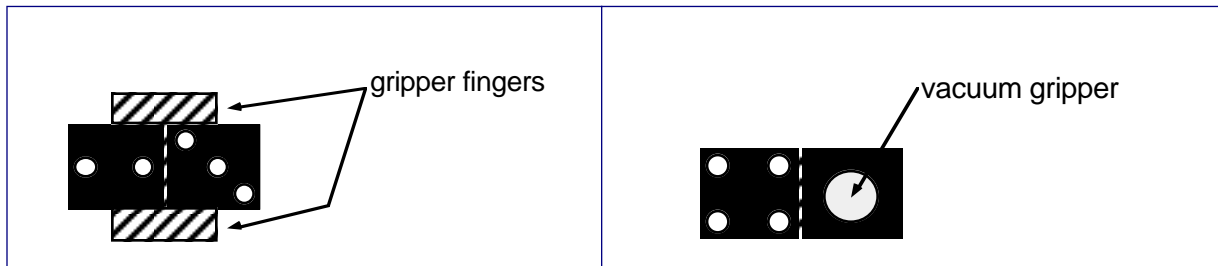
**paste
newspaper
clipping here**

Now that you are masters of robot and vision programming it is time to try some event/state-driven programming. The challenge is to construct a program that can handle a variety of events & states gracefully. In robot cells, the states are typically associated with numbers of parts in queues, inspection results, positions of movable devices etc. and the events are usually triggered by sensors.

To minimize the amount of setup, we have devised the following state-driven exercise where the *state* is determined by an arrangement of dominos and *events* are triggered by random part inputs.

Procedure:

1. **Input:** You will use modern “flexible feeding” techniques. The input conveyor will simulated by a 3" by 3" input area which you will designate. Dominos will arrive within this area in random orientation and with random timing. Occasionally one might be misplaced... (see notes below).
2. The robot must identify the domino and grasp it. Dominos will be black with 0-12 white dots. You can grasp the dominos using either a regular gripper or a vacuum gripper. If you use a regular gripper you will need to think about interference between the gripper fingers and adjacent dominos in the assembly area.



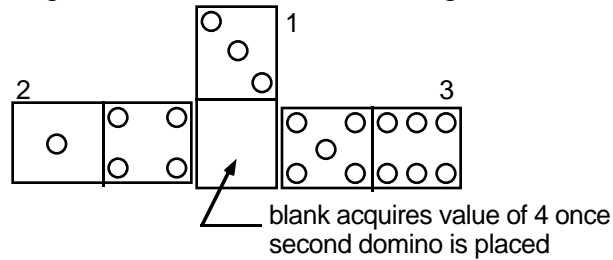
3. Place the first domino at the center of a 6x6¹ assembly area of your choosing. Then for all subsequent dominos:
4. IF either side of the domino matches the number of dots on another domino that has already been assembled AND a "valid placement" (see note 3 below) can be made within the assembly area THEN assemble it to the existing dominos. ELSE put the domino in a reject area of your choosing. Steps 3-4 continue with dominos added by instructor at random until at least 12 dominos have been submitted or the program fails – whichever comes first.

Domino Theory:

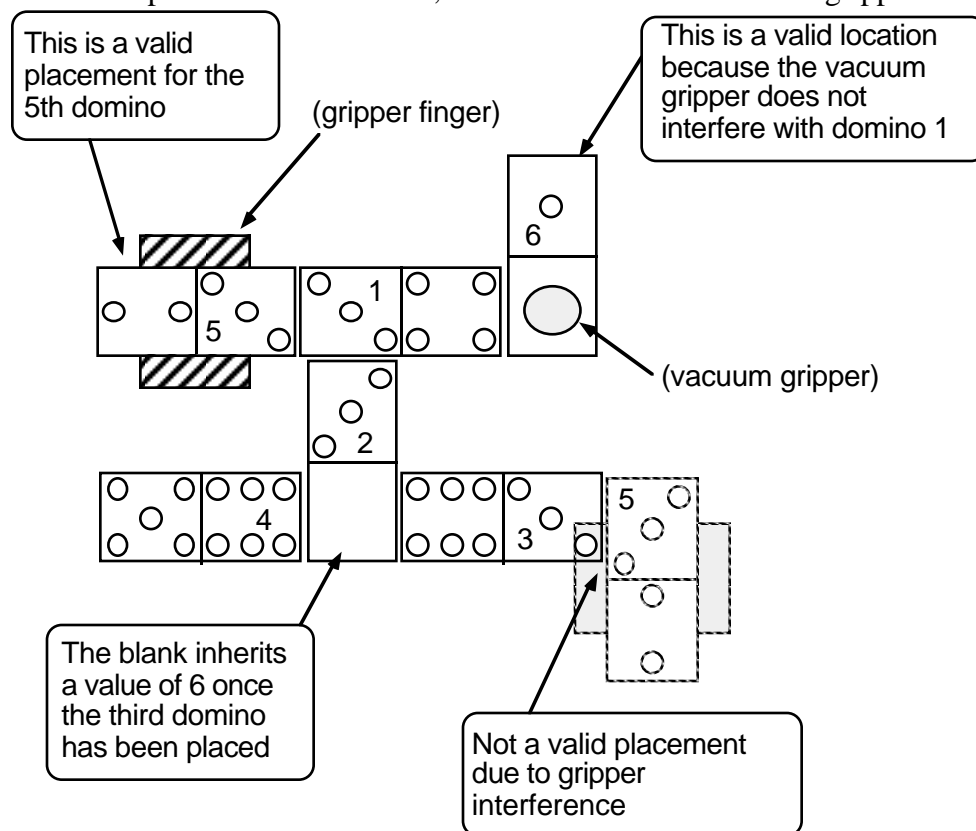
1. To reduce logic programming, this is a simplified version of the dominos game. (We care about states and events, not strategy.) All domino-domino joints with matching numbers of dots are legal (right angle, left angle, butt-joint). Forget about "Tee-ing". A blank half-domino is treated as a wild card and can match any number of dots. However, once the blank half-domino has been placed in the play area, it "inherits" the number of the domino it match-

1. That's 6 dominos or 12 half-dominos.

es. Thus, the following combination would *not* be legal:



- The dominos must be placed precisely onto the assembly, not dropped. This means that you will have to consider interference between the gripper fingers and previously placed dominos.
- Any valid domino placement is equally good. The following example illustrates some issues that may arise. The dominos have been placed in the order indicated. As domino 5 is added, there are several possibilities. However, some of them would result in gripper interference if



you use a gripper with fingers.

- IF there is no way to match numbers OR no way to place a domino without gripper interference OR no way to place a domino within the 12 x12 area THEN a valid placement cannot be found and the domino should be rejected.
- Obviously, dominos cannot be stacked on top previously placed dominos.

Hints:

- You may wish to construct a state-table or matrix to represent the configuration of previously placed dominos with various values to represent empty, unusable and/or played cells.
- Think about developing modular functions for (i) finding the position and orientation of an input domino (and checking if it is really a domino and is fully within the input area); (ii)

counting dots; (iii) getting the right gripper and grasp location; (iv) checking if there is a possible placement location in the play area (v) checking possible placement locations for interference problems. This is a good opportunity for division of labor in your group. The logic aspects can be worked on pretty much independently of the robot motion and vision parts. Two years ago one group tested all their logic off-line using MATLAB! You can also edit programs off-line and transfer to the machines for debugging if terminal access becomes difficult. On the Adept we also have V+ for Windows, which lets you do syntax checking as well as editing.

3. It's better for your program to miss some opportunities or be inefficient than to crash (either in terms of software or hardware). See scoring below for details.
4. Watch out for “defective” dominos. What if a domino is only partly within the 3” x 3” input area? What if someone accidentally sticks a roll of black electrical tape in the input area; will your program try to grasp it? (See figure below for input errors that we may simulate.)
5. As always, be sure to make the placement gentle, just in case there is an error in your logic and the robot does try to mash one domino on top of another.
6. The vacuum gripper is attractive from the standpoint of interference, but you may have trouble getting a reliable grip. Feel free to experiment with different suction cup ideas if you decide to use it.
7. You may wish to define two sets of speeds: “testing_speed” and “demo_speed”. Then in your programs set speeds to “testing_speed” as the default value, unless the user explicitly specifies demo_speed for that particular run. This will avoid the chance of any crashes at high speed when making final program adjustments. Note that when changing speeds, trajectory accuracy may vary a bit, especially with the Adept.

Game scoring (Score = 50 + bonus - cost)

Cost:

If a physically infeasible move occurs (grripper/domino or domino/domino crash, block placed outside 12 x 12 area) -10 points per event

If an invalid move occurs (numbers don't match or domino does not join with previously placed dominos) -5 points per event

Placement opportunity missed (program fails to place a domino when a domino could have been placed legally) -2 points per event

Sloppy placement (greater than 3mm maximum gap between adjacent dominos)-5 points total

Bonus:

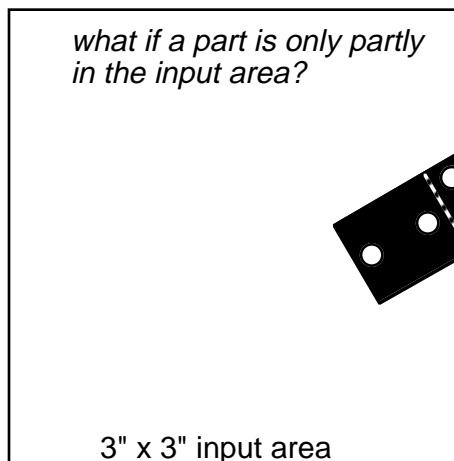
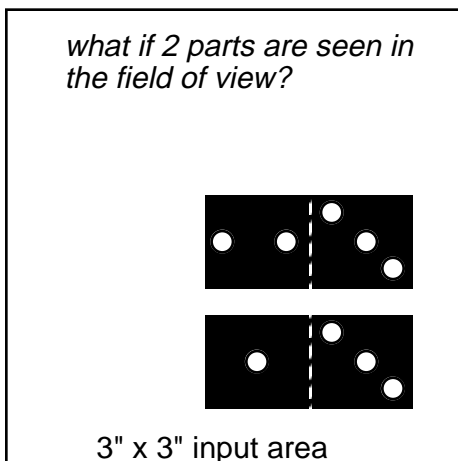
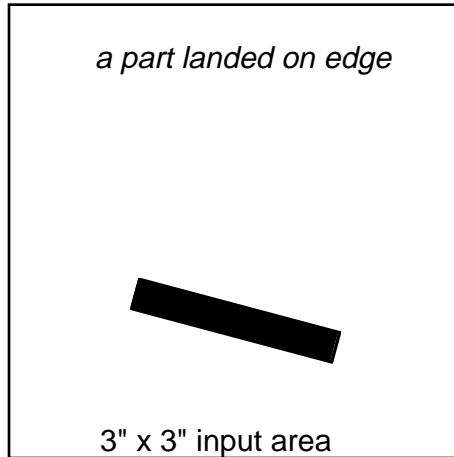
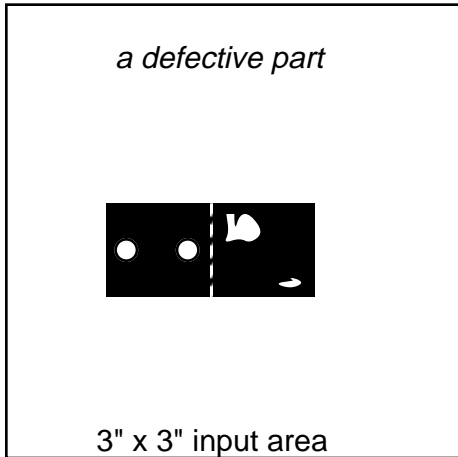
Nice user interface, fast smooth execution, general “coolness” of the approach up to +5 points

Especially robust at rejecting faulty dominos & other input errors up to +10 points

(See figure on next page)

Defective dominos

Here are 4 types of input errors that we might simulate:



Project 3b Domino Dendrites

NEW for '97. This is an experimental alternative project that I have been thinking about. It's not any harder than 3a (maybe easier) and I hope at least one group will try it.

dendrite : the branched part of a nerve cell that carries impulses toward the cell body. From Greek *dendron* (tree).

There's a nice quote from Herb Simon² to the effect that complex behavior need not arise from complicated systems; it can also arise from a simple system in a complex environment). One interesting way to elicit complex (i.e., not easily predicted) behavior from a robot is to use a combination of event-driven programming, state-dependence and recursion.

The game

The idea behind this game is to construct "dendrites" (recursively branching trees) of dominos that grow in an interesting way. The playing field is a rectangular square, 6 dominos wide by 10 dominos long.

Begin by placing a domino at the center of left end of the playing field, with the long side parallel to the long axis. This begins the main trunk of the tree (see figure on page 21). The right end of the domino is an "open branch" waiting to have more dominos added to it.

As in Project 3a, dominos will be placed into an input area and, if the number of either side matches an open branch, they should be used to extend the branch.

If a domino has matching left and right sides, and it can match the open end of a branch, then it becomes a T, starting two new branches at right angles to the parent one.

Blanks are wild cards and can match anything. So if one side of a domino matches a branch tip, and the other side is a blank, you should do a T. The blank side inherits the value of whatever it matches once it has been played (same as rules in Project 3a).

The Tee-ing is what makes a recursive algorithm attractive for this application. In pseudo-code, the logic is as follows:

```
function growbranch(branch_loc)
begin

while branch not terminated {
  get next domino ; get a non-defective domino

  if (Tmatch) && (Tplaceable) then {
    Tee branch, create 2 new locations, left_end and right_end.
    growbranch(left_end) ;here is the recursion!
    growbranch(right_end)
  }

  if (regular_match) then
    if (placeable)
      extend branch, update branchtip_loc
    else terminate branch

  if (nomatch) then
    reject domino ; and wait for the next one...
  }
end
```

² (Nobel Laureate and a father of the field of Artificial Intelligence)

Notes

1. `branch_loc` gives the position and orientation of a branch tip. So it is probably convenient to make it a transformation. It basically keeps track of the branch.
2. `left_end` and `right_end` are the new branch tip locations formed when Tee-ing a branch. They can be computed by knowing the domino size and previous branch location.
3. `placeable` is `false` if the branch runs out of the play area or would interfere with dominos already played.
4. L joints and butt joints are both acceptable as in Project 3a.
5. Basically, each branch continues to grow as long as there is a "run" of placeable dominos input in succession. If a domino can Tee a branch then it should (to create as many branches as possible). If the branch runs out of room the run is terminated and the algorithm pops out of whatever level of `while()` loop it was working on.
6. Basic issues of checking for interference and defective dominos are the same as in Project 3a.
7. About recursion and re-entrant code: Notice how compact the main logic in the above pseudo code is, because it exploits recursion. The Adept supports recursion, with a few caveats. Read the section in Volume 1 of the manual on "Re-entrant Code." The RAIL manual does not say anything specific recursion, but I think it supports it. If not, you can use the stack idea discussed below.
8. The above algorithm rejects a domino if it cannot be used to extend the current branch. But what if that domino could have been used to extend another branch that has not been terminated yet? This might make for more interesting play. To do this, we would need to have a way of recording started, but unterminated, branches. One could put unterminated "branch tips" on a stack (where "branch tip" stores location, orientation, matching dots number) and then test input dominos against any tips on the stack. A matching branch would then be modified and put back on the stack. Branches would be removed from the stack when they were terminated (due to space limitations).

Game scoring (Score = 50 + bonus - cost)

Basic scoring can be similar to Project 3a. But because this is a new experiment there will be some lenience for bugs. The real objective is to obtain "interesting" behavior, without a huge amount of tedious programming.

Cost:

If a physically infeasible move occurs (gripper/domino or domino/domino crash, block placed outside play area) up to -10 points per event

If an invalid move occurs (numbers don't match or domino does not join with previously placed dominos) up to -5 points per event

Placement opportunity missed (program fails to place a domino when a domino could have been placed legally) -2 points per event

Sloppy placement (greater than 3mm maximum gap between adjacent dominos)-5 points total

Bonus:

General "coolness" of the implementation up to +5 points

Especially interesting behavior obtained up to +10 points

.

Paste landscape figure of domino dendrites here

(see dendrites.pdf if pulling the electronic version of this file)