APPENDIX A

GLOBAL PATH PLANNING COMPUTATIONAL COSTS

This appendix provides some typical timings for the cartographic and navigator components of the AuRA architecture. All CPU times are from a moderately loaded VAX-11 750 (8 MB memory and floating point hardware). It is estimated that the times cut be cut by 67% to 75% by using a 68000 based workstation (e.g. SUN).

§1. Mapbuilder

Recognizing that the mapbuilder code is far from optimized, still some basic interpretations of the performance of the underlying algorithms can be made. Each component of the mapbuilding algorithm will be discussed in turn. The results of this section are based on a total of 72 different decompositions on 8 maps.

§1.1 *Border Growing*

The border growing algorithm is dependent on the number of vertices to be grown. The time required to grow the border for figure 16 was approximately 5.2 CPU seconds (28 vertices). A larger border (57 vertices) took approximately 22 seconds.

§1.2 *Obstacle attachment*

The obstacle attachment component of the mapbuilding algorithm is currently the least efficient part of the code. Massive improvements in CPU speed can be made. This phase of the mapbuilding is the most time consuming and appears to be exponential in order. A significantly more efficient algorithm for this phase could and should be developed if this system was to be used for other than experimental testing. The total

time to grow and attach 1 typical obstacle is 0.05 sec. This increases to 28 seconds for 7 obstacles and is dependent on the number of vertices of the border, number of obstacles, and number of obstacle vertices. Efficiency is currently not a prime factor during the long term memory mapbuilding phase as it is only compiled once at the start of the run. Nonetheless, for more complex environments this part of the code should be cleaned up.

§1.3 *Decomposition Times*

Perhaps of more interest is the time required for the recursive decomposition of the single region output by the obstacle attachment algorithm. Surprisingly rapid results were obtained. For Figure 18, computational times from 3.2 to 12.2 CPU seconds were observed (including merge). The general trends were as expected; choosing the first concave vertex consistently outperformed the least or most concave modes (an example timing is provided in Table 6). Of greater significance was the choice of the victim vertex: the opposite vertex performed best, followed by leftmost then rightmost victim. In many cases choice of the most opposite vertex resulted in decomposition times about 1/4 of the time required for rightmost victims.

§2. Navigational path finder

Fifteen different paths using four different start and end point pairs and two different safety margins were computed using the decomposition shown in Figure 22a. Many of these paths are shown in Figures 26-32.

No claim is made for the efficiency of the implementation. Significant time savings could probably be achieved (especially in the path improvement section) if the code was rewritten. These should be considered relative figures, not those that indicate the lower bounds of the algorithm.

§2.1 *Search*

As would be expected the search time for the A*-3 algorithm is considerably higher than that for the A*-1 method. The total number of possible nodes to be expanded triples, while the path solution space is cubed. Although the A*-3 method is definitely

Table 6: Typical mapbuilding timings

Concave Mode	Victim Mode	Decompose time	Clean-up merge time	Total time	Number regions
Most	Opposite	2.06	1.10	39.3	29
Most	Leftmost	3.08	3.95	43.1	38
Most	Rightmost	6.15	6.06	48.3	35
Least	Opposite	2.56	0.95	39.6	37
Least	Leftmost	3.50	5.06	44.7	37
Least	Rightmost	4.00	5.35	45.5	38
First	Opposite	2.00	1.86	40.0	31
First	Leftmost	2.76	5.20	44.1	38
First	Rightmost	3.55	6.88	46.5	34

These timings are for the decomposition of the map shown in Figure 18.

The time required to grow the border region (configuration space) is 5.2 seconds.

The time required to grow and attach the obstacles to the border is 30.9 seconds.

more costly, it is not prohibitive as the data below confirms.

Range for A*-1 (search) : 0.46-1.38 sec (avg: 0.95 sec)

Range for A*-3 (search) : 1.51-6.15 sec (avg: 4.07 sec)

Percentage Increase for A*-3 over A*-1 (search):

- Range: 188% - 781%
- Average: 455%

§2.2 *Path Improvement*

A*-3 will perform better than A*-1 in the path improvement component of the algorithm as it can completely bypass the initial tautness part. This results in significant time savings which offset some of the additional computation as described above.

Using the same paths as in the search component, these results follow:

Range for A*-1 (path imp.): 0.76-6.91 sec (avg: 3.18 sec)

Range for A*-3 (path imp.): 0.43-4.20 sec (avg: 1.46 sec)

Percentage of A*-3 over A*-1 (path improvement):

- Range: 12% - 83%
- Average: 58%

§2.3 *Total Path time (simple terrain - no relaxation)*

In all but one case the A*-1 algorithm outperformed the A*-3 method using CPU time as the indicator. The results are not too widely separated due to the shorter path improvement times required for A*-3. This shows the feasibility of this method in certain instances. It should also be noted that in every case (except one) the A*-3 algorithm came up with a path equal to or lower in cost to the one produced by the A*-1 algorithm. Only when high safety factors were involved did A*-1 derive a lower cost path. This is misleading as the goal was to produce safe, not short paths. If the straightening was turned off the A*-3 algorithm would have outperformed A*-1 as expected.

Range for A*-1 (total) : 1.22-7.59 sec (avg: 4.13 sec)

Range for A*-3 (total) : 2.14-8.66 sec (avg: 5.54 sec)

Percentage Increase per run for A*-3 over A*-1 (total):

- Range: 75% - 271%
- Average: 162%

On the average it took 62% longer to find a path using A*-3 than A*-1 when all path improvement techniques are in use.

§2.4 *Path cost savings*

Of the three paths produced by the A*-3 method that were less costly than those produced using the A*-1 method, the first was 8% less costly, the second was 7% less costly, and the third was 2% less costly. These are rather paltry cost savings for the mobile robot domain in which dynamic obstacles and/or uncertainty can render any precomputed path useless.

§2.5 *Multi-terrain transition zone relaxation*

Path relaxation figures were harder to collect and are highly dependent on the relaxation step size. The increment was set to 3.0 feet, about the diameter of the robot. Obviously, the number of transition zones to be relaxed also has a profound effect on the relaxation time.

For a single relaxation zone, (the longest concrete-grass border: see Figure 36), relaxation times observed ranged from a minimum of 0.40 CPU sec to a maximum of 4.13 sec. For two zones (grass across gravel path to other grassy section), times from 3.06 to 5.23 seconds resulted.

The relaxation for Figure 37 (two zones) took 0.38 seconds (this was overhead cost only as the path produced by the improvement strategies was already at a minimum cost). Figure 38 (2 zones) took 3.73 seconds and Figure 39 (3.5 zones) took 10.7 seconds. The longest observed relaxation time (4 zones) was 22.81 seconds. It is anticipated that most paths for the mobile robot will not involve numerous transition zone crossings.

APPENDIX B

ROBOT VEHICLE INTERFACE C LIBRARY

The routines listed below are used to establish communications, execute translation and orientation changes of the vehicle, control sensor data collection processes, and other miscellaneous functions. Fortunately, many of the functions existed within the DRV terminal emulation software and the major function for several of the routines below was to communicate the command from the host VAX to the robot. In other cases, the DRV software had no parallel. The Denning documentation set [38] and the UMASS DRV's Software Development Guide [12] serve as further references.

If AuRA is to be ported to another vehicles, these routines would need to be recoded to accommodate the new vehicle. As long as the vehicle's locomotion system is not that different from the DRV's, most of the other subsystems of AuRA can be considered insulated from a vehicle change and thus support the concept of robot vehicle independence.

§1. Robot Primitives

- Translation commands:
  - move(distance,velocity,acceleration)
    move a given distance then stop.
  - drive(velocity,acceleration)
    initiate translational motion.
  - wait_for_drive_to_stop()
    signal when drive motor has stopped.
  - stopdrive()
    terminate translation normally (smooth deceleration).

- killdrive()

  emergency stop.

- stop()

  stop both translation and rotation.

- Rotation commands:

  - turn(angle,angular_velocity,angular_acceleration)

    turn a given angle then stop.

  - steer(angular_velocity, angular_acceleration)

    initiate rotation.

  - home(angular_velocity,angular_acceleration)

    turn wheels to known orientation relative to body.

  - wait_for_steer_to_stop()

    signal when robot has stopped turning.

  - stopsteer()

    terminate rotation normally.

  - killsteer()

    emergency rotation stop.

- Sensor

  - Ultrasonics:

    * range_start()

      start ultrasonic sensors firing.

    * range_stop()

      stop firing of ultrasonic sensors.

    * range_read(sonar_data)

      acquire ultrasonic data.

  - Shaft encoders:

    * initxy()

      set xy position coordinates to x = 0, y = 0, theta = 0.

  * getxy(x,y,theta)

    acquire positional information from DRV.

  * setxy(x,y,theta)

    set positional information from DRV.

  * drivepos()

    acquire shaft encoder data from drive motor.

  * steerpos()

    acquire shaft encoder data from steering motor.

  * drivevel()

    acquire velocity of drive motor.

  * steervel()

    acquire velocity of steering motor.

  * drivestat(status)

    read the status of the drive motor controller.

  * steerstat(status)

    read the status of the steering motor controller.

- Communications

  - open_robot_channel()

    Establish communications with the robot.

  - send_robot_message(channel,message,length)

    send a command to the robot.

  - flush_to(channel,character)

    remove garbage output from robot.

  - read_robot_character(channel,character)

    read a character from robot.

  - find_first_non(channel,character,buffer)

    strips data up to character.

  - close_tty_line(channel)

    terminate communications with robot.

- Miscellaneous
  - set_terminal_raw()

    configure the DRV's communication port.
  - set_parameter(name,value)

    set a DRV internal parameter to a specified value.
  - monitor_move(sensor,spread,limit)

    safe translation using ultrasonic data.

APPENDIX C

VISION ALGORITHMS

This appendix contains outlines of two of the vision algorithms referred to in chapter 6. References are cited where appropriate to direct the reader to more information on their details.

§1. Fast line finder

The outline of this algorithm, developed by Kahn, Kitchen and Riseman, is reproduced from [64]. The interested reader is referred to that paper for additional details. It is based on a previous algorithm by Burns, Hanson, and Riseman [31].

1.0 PASS 1 over the image

1.1 **for** each image pixel **do**

1.1.1 Compute the gradient direction and magnitude

1.1.2 Threshold on gradient magnitude

1.1.3 Use gradient direction to classify suprathreshold pixels into buckets. (Pixels below threshold and pixels whose gradient is not in a direction of interest are specially labeled and ignored in all subsequent processing.)

1.1.4 Perform connected components analysis to group adjacent pixels that share identical bucket labels into region fragments, and build a *fragment equivalence list*

1.2 **for** every unprocessed region fragment **do**

1.2.1 Use the *fragment equivalence list* to merge fragments into line support regions and set the region pixel count to the sum of the fragment pixel counts

1.2.2 Eliminate regions whose pixel count is below a certain threshold by tagging their constituent fragments "insignificant" so they will be ignored in subsequent processing

1.2.3 Assign region labels to fragments which are part of these "significant" regions

2.0 PASS 2 over the fragment labeled image and gradient magnitude image

2.1 **for** each image pixel in a "significant" region **do**

2.1.1 For the region within which the pixel is contained, accumulate statistics needed to compute the scatter matrix and endpoints (for line fitting)

2.1.2 If desired, build a region labeled image

2.2 **for** each "significant" region **do**

2.2.1 Compute the scatter matrix from the accumulated statistics

2.2.2 Compute the best-fit line orientation and centroid anchor position

2.2.3 Compute the line endpoints

2.2.4 Put the fitted line and associated statistics into a line list to be output by the FLF program

§2. Fast Region Segmenter

This algorithm closely resembles the fast line finder on which it is based. See [64] for additional information.

1.0 PASS 1 over the image

1.1 **for** each image pixel **do**

Using a previously loaded look-up table and an arbitrary image, produce the bucket-labeled image based on gradient magnitude

1.1.2 Perform connected components analysis to group adjacent pixels that share identical bucket labels into regions and build a *fragment equivalence list*

1.2 **for** every unprocessed region fragment **do**

1.2.1 Use the *fragment equivalence list* to merge fragments into regions of similar intensity and set the region pixel count to the sum of the fragment pixel counts

1.2.2 Eliminate regions whose pixel count is below a certain threshold by tagging their constituent fragments "insignificant" so they will be ignored in subsequent processing

1.2.3 Assign region labels to fragments which are part of these "significant" regions

2.0 PASS 2 over the region labeled image and gradient magnitude image

2.1 **for** each image pixel in a "significant" region **do**

2.1.1 For the region within which the pixel is contained, accumulate region statistics.

2.1.2 Build a region labeled image

2.2 **for** each "significant" region **do**

2.2.1 Compute the scatter matrix from the accumulated statistics

2.2.4 Put the realm statistics into a list to be output by the FRS program