Dynamic Factored Particle Filtering for Context-Specific Correlations

by

Dimitri Mostinski

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.
I understand that my thesis may be made electronically available to the public.

Dimitri Mostinski

Abstract

In order to control any system one needs to know the system's current state. In many real-world scenarios the state of the system cannot be determined with certainty due to the sensors being noisy or simply missing. In cases like these one needs to use probabilistic inference techniques to compute the *likely* states of the system and because such cases are common, there are lots of techniques to choose from in the field of Artificial Intelligence.

Formally, we must compute a probability distribution function over all possible states. Doing this exactly is difficult because the number of states is exponential in the number of variables in the system and because the joint PDF may not have a closed form. Many approximation techniques have been developed over the years, but none ideally suited the problem we faced.

Particle filtering is a popular scheme that approximates the joint PDF over the variables in the system by a set of weighted samples. It works even when the joint PDF has no closed form and the size of the sample can be adjusted to trade off accuracy for computation time. However, with many variables the size of the sample required for a good approximation can still become prohibitively large.

Factored particle filtering uses the structure of variable dependencies to split the problem into many smaller subproblems and scales better if such decomposition is possible. However, our problem was unusual because some normally independent variables would become strongly correlated for short periods of time.

This dynamically-changing dependency structure was not handled effectively by existing techniques. Considering variables to be always correlated meant the problem did not scale, considering them to be always independent introduced errors too large to tolerate. It was necessary to develop an approach that would utilize variables' independence whenever possible, but not introduce large errors when variables become correlated.

We have developed a new technique for monitoring the state of the system for a class of systems with context-specific correlations. It is based on the idea of caching the context in which correlations arise and otherwise keeping the variables independent. Our evaluation shows that our technique outperforms existing techniques and is the first viable solution for the class of problems we consider.

Acknowledgments

I would like to thank my supervising professors Pascal Poupart and Charles Clarke for all the help with this work. I received excellent feedback and we have had many interesting discussions, which resulted in this work and some very exciting findings. Thanks to you, I'm proud of my accomplishments in my graduate career so far and have much more to expect in the future. Thank you!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work is motivated by the Real-Time Programming course at the University of Waterloo. A major part of this course is constructing a real-time control system for a model train track with multiple trains and switches that can be controlled by a computer. A few years back the hardware used for the course was changed from a custom-made track to off-the-shelf components from Marklin – a major German manufacturer of model train sets. This transition made it easier to maintain the lab, but introduced a major problem into the development of the control system that we have set out to fix.

With the old hardware, the system was able to know the locations of trains and states of the switches, so routing the trains to desired locations was purely a control problem. With the new hardware, most of the time, neither the switch states, nor the train positions can be known for certain. There are sensors that get hit when a train passes over them, but if the trains are not moving there is simply no way to know where they are. We discuss the complications more in Section 4.1, and for now, simply say that tracking the positions of trains has become a much more complicated problem than routing the trains ever was. The change in hardware therefore introduced a whole other component into the course that had to be dealt with in a structured and formal way, which was the goal of this work.

In more general terms, the problem we are facing is monitoring the state of a stochastic process based on noisy and incomplete data. Because we do not know the exact state of the system, we must maintain a probability distribution (a.k.a. belief) over the possible system states.

Belief monitoring for stochastic processes is a common task in Artificial Intelligence. Unfortunately, exact computation in most real-world scenarios is intractable, so approximations must be used. A classic problem of trading off accuracy for computation time therefore applies to belief monitoring.

A popular method for approximating and monitoring the belief using a weighted set of samples is called Particle Filtering, or Condensation [5]. Its simplicity (i.e., samples are easy to manipulate), and versatility (samples can approximate any arbitrary PDF) has permitted its deployment in a wide range of applications. Real-time applications also benefit from the fact that the size of the sample can be decreased to fit the computation of the next belief state within the required time constraints. However, the size of the sample required for an acceptable approximation tends to be exponential in the number of variables in the system [2]. For a system with more than a few variables it too becomes intractable.

Fortunately, in many applications, the set of all variables can be split into weakly-correlated clusters [7]. If such a decomposition is possible, the size of the sample required can be dramatically reduced for a small cost to approximation accuracy. This method breaks the correlations between clusters at the end of every time step, which reduces the variance of the sampling process and concentrates on most likely samples for each cluster independently.

However, in some problems, variables are considered weakly-correlated if they evolve independently most of the time, but may get strongly correlated for short periods of time. An example of this type of correlation is evident in our application where one train's position is normally independent of another switch's state unless the train has just passed the switch. In the latter case the switch determines which branch the train will follow.

These context-specific correlations cannot be handled adequately by existing techniques. Going with a factored approach means that once in awhile the error introduced by factoring may be unacceptably large, while not factoring the variables would mean incurring a considerable unnecessary computational overhead most of the time. In solving the belief monitoring problem for trains control we were able to design a solution for these specific types of correlations.

The basic idea was to keep the variables factored, but remember the event that triggered

the correlation in what we called a cache. When a non-zero observation is received, a cache can be used to reconstruct the true joint PDF of correlated variables and correctly adjust all correlated variables' marginals.

Our method does not increase the variance of the sampling process compared to the factored approach (which is the minimum variance achievable) and captures all the correlations that existed (which is the maximum accuracy achievable). We hypothesized that with the problem at hand we would get the accuracy of plain Particle Filtering for the price of Factored Particle Filtering, and our results confirm this hypothesis. Our approach also facilitated the use of Rao-Blackwellisation [4] which is an approach to further reduce variance by calculating exactly some marginals whenever possible. As such, our method is the first for this class of context-specific correlations.

We have also investigated the possibility of dynamically creating clusters of correlated variables when correlations do occur. This has been tried previously by Ng et al. [6] for the Mars rover. In that problem, orientation of the rover was used for both determining when to create a cluster as well as when to break it. In our case we created a cluster when a train passed a switch and broke it some time afterwards, when keeping the cluster around presented little benefit to the accuracy of the approximation.

There were other problems with dynamically creating clusters in our application. The clusters had a tendency to grow as trains passed more switches. They became overlapping as different trains passed the same switches. These aspects have had a significant impact on the complexity of the sampling process, and introduced unnecessary variance into the system, and in our testing was not as successful as the factored particle filtering with caches approach.

Our contributions mainly lie in three areas.

- CS 452: We developed the necessary theoretical background to approach the problem of model train tracking in a principled way. This problem has to be solved by students every term in order to complete their projects. Without knowing the necessary background it is practically impossible to develop a reliable system.

- Particle Filtering: We applied a well-accepted technique to a real-world problem with some unusual properties. We have identified the limitations of existing theory for context-specific correlations.

- AI Research: We developed a new technique for the class of problems with context-specific correlations. We have evaluated and compared our technique to existing techniques in AI and showed that ours performs the best results both in scaling and accuracy. To our knowledge this is the first viable solution for the class of problems we consider.

The rest of the thesis is structured as follows: in Chapter 2 we outline the environment that we deal with, we introduce the general goals of the complete system, which our belief monitoring is a part of; in Chapter 3 we model that environment, define variables and correlations; in Chapter 4 we present the methods for monitoring the belief state of the system with Section 4.4 targeting directly the context-specific correlations. We present the results in Chapter 5 and conclude in Chapter 6 outlining the directions for future research.

Chapter 2

Environment

In this work, we facilitate the development of a real-time control system for a Marklin rail-road model. We do not develop the control system itself, but lay out a design that can be implemented using our technique and show how effective our technique is.

The rail-road model consists mainly of straight and curved track pieces, switches, and trains. Straight and curved track pieces can be equipped with sensors to detect train movements. The rail-road (trains and switches) can be controlled by a computer via digital interface. The model is characterized by high degree of uncertainty in its state. For one, the switch state cannot be directly queried, secondly the train positions cannot be queried either and have to be inferred from a series of discrete sensor hits, which themselves need to be localized in time. The control system will thus have to possess a high degree of Artificial Intelligence.

2.1 Application Level Track Interface

We would like application level interface to be

- Intuitive – Easy to use
- Deterministic – Contain no uncertainty or probabilistic aspects
- Real-Time – Support real-time constraints

In particular we want the interface to support the creation of three classes of applications commonly implemented as part of the Real-Time Programming course of the University of Waterloo. They are "joystick" control, delivery service, and "follow the leader". Detail descriptions of these problems as well as outlines of sample solutions are given near the end of this section. Of course, while completing their tasks, the trains must behave intelligently – avoid collisions, not create dead-locks, and get to their destinations as soon as possible.

A computer interface is only able to send commands to the track at the fairly low rate of about 5 commands per second. This quickly becomes a bottleneck when multiple trains are active. The bandwidth must also be utilized intelligently to maximize the system's throughput.

2.1.1 Some terminology

In discussing the railroad track layout it is useful to refer to a switch's branches in a consistent way. In most cases, the switch looks similar to the one depicted on Figure 2.1. As illustrated we refer to the three end points of a switch as left branch, right branch, and a trunk.

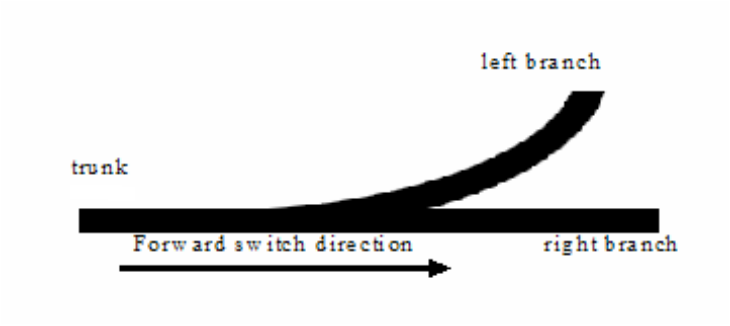


Figure 2.1: Typical switch

When going over a switch in the forward direction a train can end up leaving along either the right branch or the left branch depending on the switch state. When going over

the switch in the opposite, or backwards, direction a train will always leave the switch along the trunk whether it came from the left or the right branch.

We note that a more complicated type of switch can be represented by a group of typical switches. For example Figure 2.2 shows how a single three-way switch can be represented by two regular switches.

Figure 2.2: Representation of a three-way switch

2.1.2 Coordinates

If the track has at least one switch and the track is connected, every point on the track can be addressed relative to some switch. We assume the track has this property and define the following rules for addressing points on the track. The address is a tuple $< S, B, D >$ where $S \in \aleph$ is switch number, $B \in \{L, R, T\}$ is a switch branch (left, right, or trunk), and $D \in \Re^+$ is the distance measured in centimeters. We define the set of such tuples as COORD for future reference. Figure 2.3 illustrates this scheme with few examples.

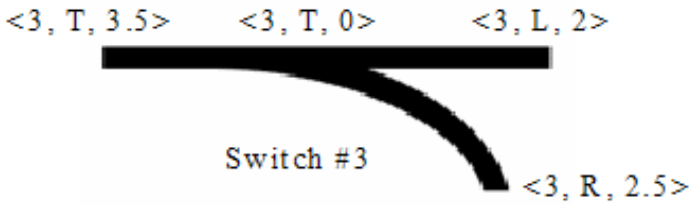


Figure 2.3: Examples of coordinates

To find a point $< s, b, d >$, we find the switch s, and travel the distance d along the branch b. If while traveling we encounter another switch going in forward direction, we proceed along the right branch as a heuristic. If we encounter another switch in the backward direction we proceed along the trunk. Figure 2.4 illustrates examples of such addresses.

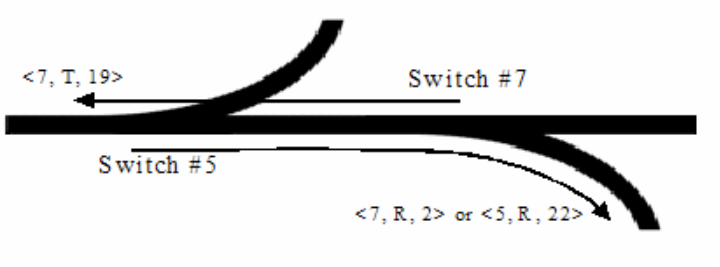


Figure 2.4: More examples of coordinates

We note that the same point may have multiple addresses.

2.1.3 Tasks

Trains will perform real-time tasks. For each train we want to specify a destination and the deadline by which to get there. In order to control the speed we also specify a maximum speed allowed while completing the task. The task is thus a tuple $< T, C, S, D >$ where $T \in \aleph$ is the train number, $C \in COORD$ is a coordinate as described above, $S \in \Re^+$ is maximum allowed speed measured in centimeters per second, and $D \in \aleph$ is the time the task must be completed by. D is measured in ticks since the start of the system, and can be converted to and from the conventional time units.

The set of such commands comprises a schedule. The schedule must be finite and can change over time as new tasks are added to the system and old ones leave the system.

For each submitted task, the system will produce an output tuple $< S, T >$ where $S \in \{success, late, failure\}$ is the end result of executing this task and $T \in \aleph$ is the time the task left the system. For a successful task t an output tuple o would be such that $o.s = success$ and the time the task left the system $o.t \leq t.d$ the deadline set for the task.

For a failed task $o.s = failure$ and $o.t$ will be the time the task was terminated, for an invalid command, for example, it would be the time the task was submitted.

The following set of rules defines the semantics of task execution.

- The train will always try to go to its destination by the shortest path and will only take detours if the shortest way is blocked by other traffic.
- If the train's path is blocked, the train will attempts the alternate route only if it believes it can still meet the deadline.
- If there is no time to take a detour the train will wait for the way to become clear until the deadline passes.
- If the deadline passes while the train is blocked the task will fail, whereas if the deadline passes while the way is clear the train will still go to its destination.
- Once the train reaches its destination it will stop unless another task is given to it.
- A train that is stopped or blocked while waiting for the way to become clear, will move out of another train's way, if necessary. In the case of two trains blocking each other, the train whose task has a higher chance of succeeding will assume priority.

In order to follow such rules, the state of the system has to be estimated accurately. The goal of this thesis is to provide a solution for tracking the state of the system well enough, to allow implementing such a policy for train control.

2.1.4 Protocol

In order to show, how common problems in Real-Time Programming can be solved, we design a general protocol for supplying commands to the system. We then present solutions to relevant problems using our protocol.

The protocol describes the means of inputting the schedule into the system and receiving the results. The protocol must also allow us to specify more complicated behaviours to the system rather than merely going from place to place, "follow the leader" for example (to be discussed in detail later).

We could simply enter the task tuples into the system on-line. However if we want a train to follow some route and not stop at every intermediate point we would like to know when the train is only approaching its destination. For that purpose the protocol allows another parameter to be sent to the system along with the input tuple. This parameter is progress report request r where $0\% \leq r \leq 100\%$.

Once the train completes the amount of journey specified by report parameter the system sends an acknowledgement to the user stating the train identifier and its current position. The task for which the report is sent is understood to be the last task given to that train.

Once the task is complete (or terminated) the system sends another report, well refer to it as complete report, containing the train identifier, the status of the task, and the time at which the task was completed. This is basically the output tuple discussed before plus the train identifier.

If a task arrives for the train while it is in the process of executing an earlier task, the train will abandon the task it is currently doing and proceed with the new one. In this case the complete report is not sent as the previous status of the train is assumed to be known to the user.

Another useful feature of the protocol is to simply query the position, speed, and destination of the train. We define a poll command for that purpose which contains a train identifier only.

In response to the poll command the system sends an update message. The update message contains the train number, its coordinate, speed, and the coordinate of where the train is going if it currently has a task.

Table 2.1 summarizes the protocol.

2.1.5 Application examples

Here we discuss how the three classes of problems mentioned in the beginning of this section can be solved using this interface.

Command	**Parameters**	**Disposition**	**Description**
go	train, coordinate, maximum speed, deadline, report	input	submits a task to the system
report	train, coordinate	output	reports the train's progress in completing a task
complete	train, status, time	output	reports on the final status of a task
poll	train	input	queries the system about the train status
update	train, coordinate, speed, target	output	updates the user of the train status

Table 2.1: Protocol summary

Joystick Control

The idea of the joystick is simple: having a joystick (or the four directional arrows present on most keyboards) we assign meaning to each direction for a particular train as follows.

- Forward - The trains should speed up if not at the maximum speed.
- Backwards - The train should slow down or stop if it is at the minimum speed. If the train is already stopped the train should reverse and start moving at the minimum speed in the reverse direction.
- Right/Left - The train should follow right/left branch the next time it encounters a switch going in forward direction.

We keep the train going by adding new tasks to the system whenever one is near completion. To keep the speed at the desired level we set the maximum allowed speed to the desired speed and set the deadline to current time. This ensures that if the way is clear the train will go to its destination at maximum allowed speed. We note that all the tasks are going to be late or cancelled (in case of blocking traffic), but being on time is not an aspect of the "Joystick" problem.

To implement turning whenever a left or right signal arrives we choose a target along the specified branch of the next switch.

Delivery Service

Delivery service is the one most closely related to the suggested protocol. There are producers and consumers of resources positioned somewhere along the track. Once in a while consumers produce orders for certain producers. The train must pick the ordered goods at the producer and deliver them to the consumer by the specified time. Each order therefore yields two tasks: going to a producer and going to the consumer. The application must simply manage the time to set appropriate deadlines. Under normal circumstances translating orders to these tasks is all that the application has to do.

One variation of the delivery service is a simple random movement. Here we simply generate random places to go and go there. Only one task is necessary whenever we decide to go somewhere and we give this task a very loose deadline.

Follow the Leader

This application requires more care in developing. For the sake of example we'll assume the track has the simple layout depicted on Figure 2.5.

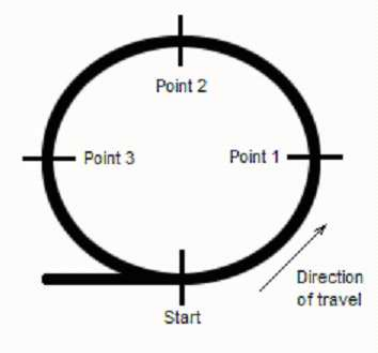


Figure 2.5: Follow the leader on a simple track

The basic idea is for one train to traverse a route (circle in the example) counter-

clockwise (Start → Point 1 → Point 2 → Point 3 → Start etc). This train is the leader. Then we add another train. It approaches the switch from the left branch and waits for the leader to come around. Then the follower positions itself after the leader and continues to traverse the circle as well. If the leader is too far ahead the follower increases its speed, if the leader is too close the follower slows down.

Later more trains can be added in the same manner each following the train added before it. The track can also be more complicated but the idea is the same: the leader chooses some route while the follower follows.

We note that we force the leader to traverse the loop counter-clockwise because the deadline we set is tight forcing the train to choose the shortest path.

We present pseudo code for the two routines one for the leader and one for the follower in Figure 2.6.

2.2 Hardware Level Track Interface

So far we have described a high level interface presented to the application developer. The control system on the other hand needs to communicate with the track using the native interface defined by Marklin, the manufacturer of the track and trains used in this work. We now describe this low level interface that is communicated from the system to the track via computer serial port.

Marklin trains have 14 speed settings, or *gears*, and can go both forward and backward. The train is given one byte for its speed which can range from 0 to 14. If this byte is set to 15 the train stops and reverses its direction (the next time it will be given a positive speed it will move in the opposite direction). Along with speed another byte needs to be sent that identifies the train. A train command is thus two bytes long.

A Marklin train's gear is a discrete number from 0 to 14. For different trains this number translates to a different actual velocity on the track. Furthermore trains do not always keep their velocity constant. The same is true for the acceleration and deceleration of trains. The system will need to learn these properties for every train.

Switches can be told to throw either way by sending a two byte command. One byte specifies the desired position of the switch and the other one identifies the switch. Switches

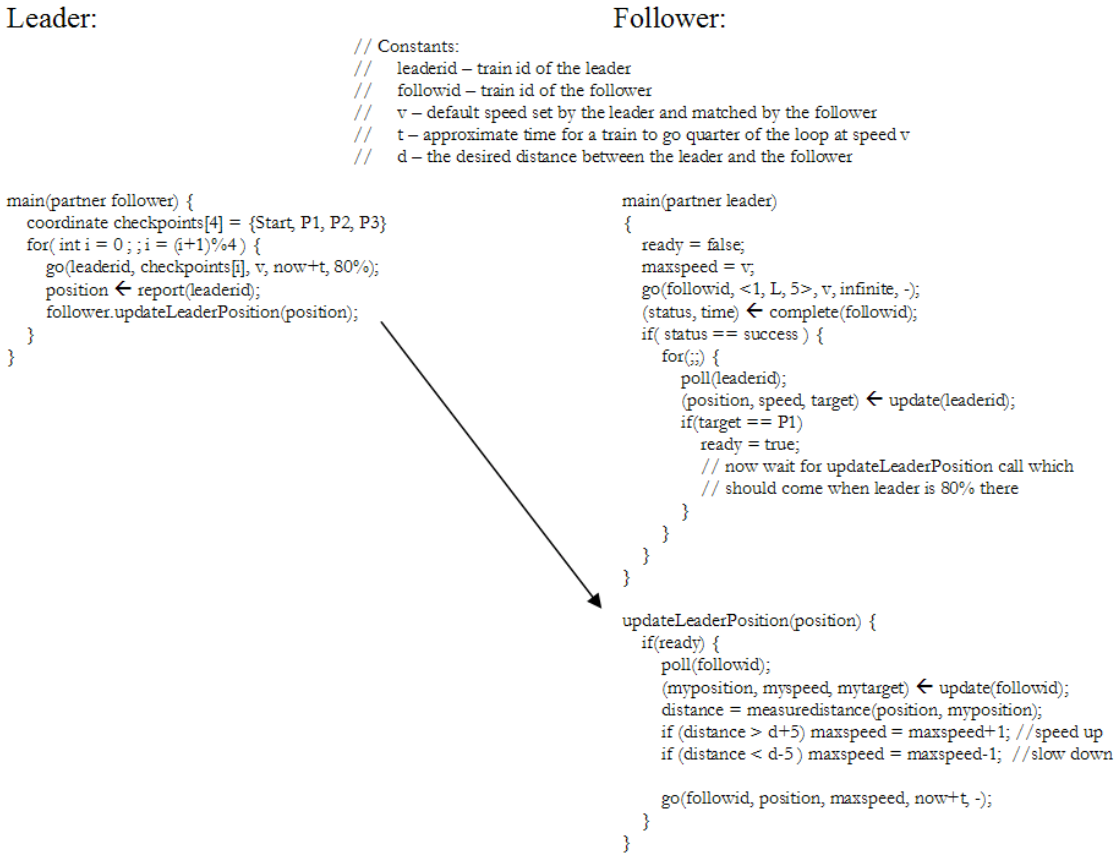


Figure 2.6: Follow the leader on a simple track

are thrown by miniature electromagnets which do not automatically turn off once the operation is complete. Another one-byte command needs to be sent after an approximate 150 ms delay to turn off these electromagnets.

In order to track the train movements the track is equipped with sensors. The sensor registers a direction of the train when it passes. The sensor hit is recorded in memory and needs to be queried by the computer with a one byte command sent to the track. In response to the query the track dumps into the computer all the sensor hits that happened since the last time these sensors were queried. The time of the hit is therefore some time in between the sensor queries and the train identifier needs to be inferred from the system state.

After every command the computer interface needs to take a pause approximately 200 ms long. The interface can therefore send about 5 commands a second to the track.

2.3 Connecting Interfaces

As one can see, there is a substantial difference in hardware and application interfaces, that is between what we have and what we want to achieve. This work is about facilitating the creation of application interface while controlling the system through the hardware interface.

There is a lot of uncertainty in trains' positions, controlling this uncertainty is the key aspect of this work. Once we have a good handle on the system's state higher-level control problems can be solved.

We have provided information about the desired control system in order to define the context of our work. The development of the control system is outside the scope of this thesis.

Chapter 3

Modeling

We model the system by a set of variables. The system undergoes transitions in a discrete sequence of steps called the time steps. Values assigned to the variables at the next time step depend on the their values at current time step, observation received from the system, and action communicated to the system. If knowing variable's current value, observation, and action allows us to determine variable's next value deterministically, we call that variable observable. If this information only allows us to construct a probability distribution function over the possible next values for the variable, we call that variable unobservable. Variables may be correlated such that the value of a variable at the next time step depends on the values of all correlated variables at the current time step. Dynamic Bayesian Networks, or DBNs, are an intuitive way of presenting such correlations.

In the following sections we first define the variables for the problem of train control in CS 452. We then provide and explain those variables' correlations by means of a DBN.

3.1 State Variables

T Time at the beginning of the time step. This variable is observable.

$\mathbf{Vel_t} \in \Re$ The velocity train t has. This variable is unobservable.

$\mathbf{Pos_t} \in \mathbf{COORD}$ The position of train t. This variable is unobservable.

$\mathbf{Gear_t} \in \{\mathbf{0}, \mathbf{1}, .., \mathbf{14}\}$ The gear assigned to train t. This variable is observable through actions.

$\mathbf{Dir_t} \in \{-\mathbf{1}, \mathbf{1}\}$ The direction of train t. The direction of the train is relative to Pos_t. In essence if $Dir_t = 1$ the coordinate of train t is increasing, and decreasing otherwise. This variable is observable through actions.

$\mathbf{Sw_s} \in \{\mathbf{L}, \mathbf{R}\}$ The state of switch s. The switch is thrown left if $Sw_s = L$, and right if $Sw_s = R$. This variable is unobservable. Although actions imply a value for this variable, switches may fail to throw, they can also be thrown manually without notifying the system.

3.2 Observation

An observation at step x, $O_x \equiv \{Hit_i | Hit_i \in \{0, 1\}$ for each installed sensor $i\}$. Sensors are installed at various positions on the track and register a train passing in a certain direction. Therefore Hit_i implies a pair of coordinate and direction relative to that coordinate (Pos_t, Dir_t) where the train has passed.

Sensors are commonly installed in pairs such that positions of two sensors are the same, but they register trains passing in different directions. There is an off-the-shelf component that implements such a pair of sensors available from manufacturer.

3.3 Action

Action at step x, A_x is an ordered set of atomic actions Act_j such that each Act_j is one of the three operations that can be performed on the track.

$Tr(t, g)$ Tell train t to move at gear g

$Rev(t)$ Tell train t to reverse. The train changes direction and stops (gear is set to 0)

$Sw(s, p)$ Tell switch s to throw to specified position p (throw left if $p = L$ and throw right if $p = R$)

The order of Act_j is the order in which the actions are communicated to the track. Depending on the order the two operations may have different effects. For example a Rev command followed by a Tr command causes the train to reverse and proceed at a given gear. Submitting these actions in another order causes the train to stop completely after changing the direction.

Formally $A_x \equiv \{Act_j | Act_j \in \{Tr(train, gear) \forall train, gear\} \cup \{Rev(train) \forall train\} \cup \{Sw(switch, position) \forall switch, position\}\}$

3.4 Dependency relations of state variables

We represent dependency relations between the state variables as a Dynamic Bayesian Network on Figure 3.1. Arrows represent the direct dependencies of the variables and the absence thereof represents conditional independence. State variables at the next time step are marked with primes. What this DBN illustrates is what variables at the current time step effect what variables at the next time step.

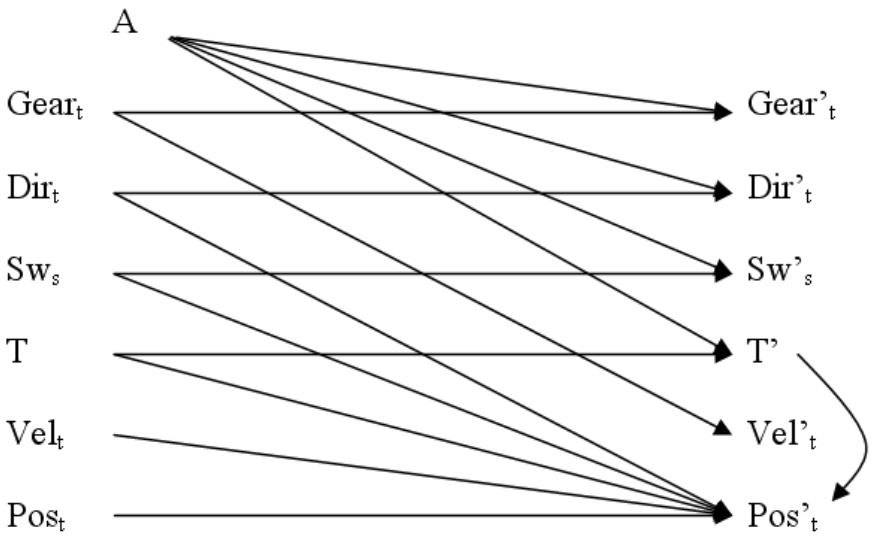


Figure 3.1: Variable dependencies

We now quantify these dependency relations. Parameters of the system we must learn are designated by θ.

$$Vel'_t \sim N(\theta_{sp_{t,g}}, \theta_{var_{t,g}}) \tag{3.1}$$

velocity is assumed to have normal distribution where $\theta_{sp_{t,g}}$ is approximate velocity train t has when traveling at gear g, and $\theta_{var_{t,g}}$ is the variance of train t velocity around the mean value $\theta_{sp_{t,g}}$, when traveling at gear g.

$$Pos'_t \sim \delta(Pos'_t, Pos_t \oplus Dir_t * Vel_t * (T' - T)) \tag{3.2}$$

position is Dirac delta function where $\oplus$ is the operator that evaluates the new coordinate a given distance away from current coordinate given Sw_s for every relevant switch s.

$$T' \sim \delta(T', T + T_{obs} + T_A) \tag{3.3}$$

time is Dirac delta function where T_A is the time necessary to communicate the generated action A to the track and T_{obs} is the time necessary to produce an observation. The time it takes to communicate an action to the track is dependent on the type of action only – T_{tr}, T_{sw}, and T_{rev} are the times for a single Tr, Sw, and Rev commands to be processed. A generated action may contain multiple atomic actions as described in Section 3.3. T_a is thus the sum of times required to communicate all the atomic actions contained in A.

Table 3.1 presents the probability of Sw'_s given Sw_s and action A. It is presented as a value for every relevant combinations of prior variable assignments. Here θ_{r_s} is the probability that switch s will correctly change to the *right* position when such command is given to it and similarly θ_{l_s} is the probability that the switch will correctly change to the *left* position.

$$Gear'_t \equiv \begin{cases} g & if Tr(t,g) \in A \\ 0 & if Rev(t) \in A \\ Gear_t & otherwise \end{cases} \tag{3.4}$$

$$Dir'_t \equiv \begin{cases} -1 * Dir_t & Rev(t) \in A \\ Dir_t & otherwise \end{cases} \tag{3.5}$$

$\mathbf{Sw_s}$	$\mathbf{Sw'_s}$	$\mathbf{Sw(s,R)} \in \mathbf{A}$	$\mathbf{Sw(s,L)} \in \mathbf{A}$	$\mathbf{Pr(Sw'_s \mid Sw_s, A)}$
L	L	0	0	1
L	R	0	0	0
R	L	0	0	0
R	R	0	0	1
L	L	1	0	$1 - \theta_{r_s}$
L	R	1	0	θ_{r_s}
R	L	1	0	0
R	R	1	0	1
L	L	0	1	1
L	R	0	1	0
R	L	0	1	θ_{l_s}
R	R	0	1	$1 - \theta_{l_s}$

Table 3.1: Conditional probability distribution for Sw'_s

3.5 Learning System Parameters

3.5.1 Train Speeds

For every train t and gear g we need to know two parameters: $\theta_{sp_{t,g}}$ and $\theta_{var_{t,g}}$. These parameters were learned offline in a series of experiments ran on a oval track with two different locomotives t_{12} and t_{20}.

The oval track (see Figure 3.2) was equipped with three sensors set a known distance apart. Each time a train would pass from one sensor to the next a measurement of the speed was taken by recording the travelling time and dividing by the distance between sensors. Three measurements were therefore made each time a train would complete a lap.

While the train was assigned a certain gear, 50 measurements were taken. After the 50 measurements were complete the train's gear was changed in a different manner depending on the experiment. The gears were either gradually decreased from 14 to 1 in a series referred to as *decelerating* experiments, or increased from 1 to 14 in a series referred to as *accelerating* experiments. The trains were also ran forwards in a series referred to as

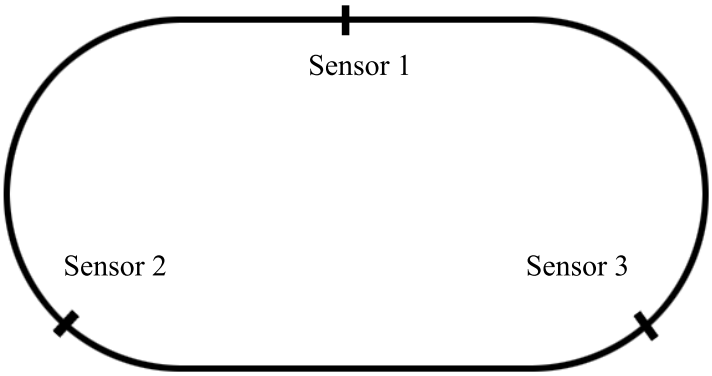


Figure 3.2: Oval Track

forwards experiments or backwards in a series referred to as *backwards* experiments, with one of the directions arbitrarily labeled forwards and the other one backwards.

Three experiments were run for every train totaling 150 measurements for every $\theta_{t,g}$.

- *forwards decelerating*
- *forwards accelerating*
- *backwards decelerating*

An average was calculated over the three tests ran and variance was calculated. A sample set for every train and every gear contained 150 values. In certain cases however it proved to be impossible to get the total of 150 values because one of the trains behaved highly irregularly at slow speeds – it would stop and had to be pushed to continue, or would fail to move at all. So samples for those gears and that train were not collected.

Figures 3.3 and 3.4 show separately the averages for the three tests ran as well as the average over three experiments with standard deviation represented as bounds around the average value.

As one can see the behaviours of the two tested trains are dramatically different. Train 12 shows a linear dependency of the velocity vs. gear setting. This train also behaves

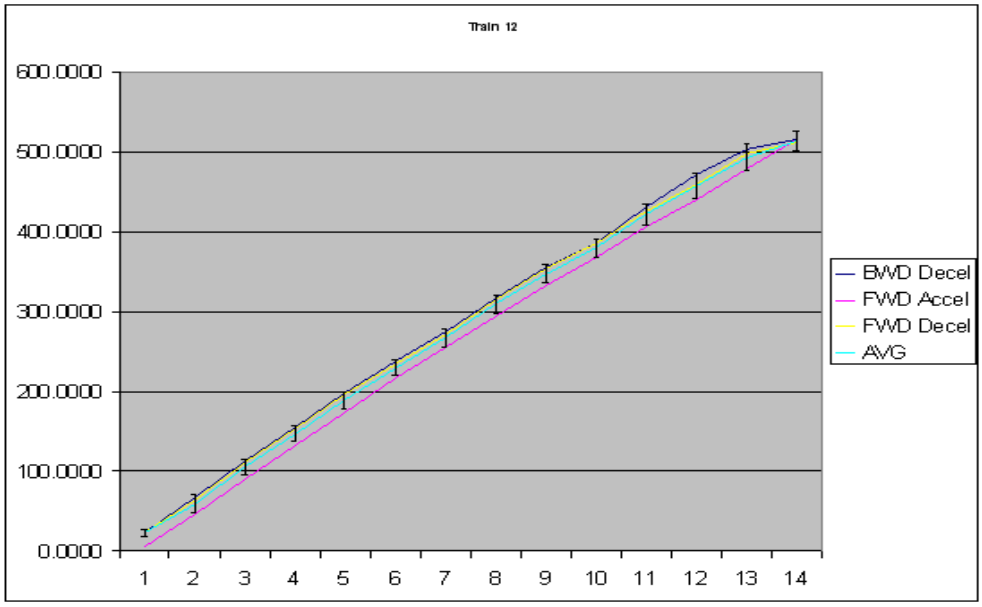


Figure 3.3: t_{12} speed parameters in mm/sec

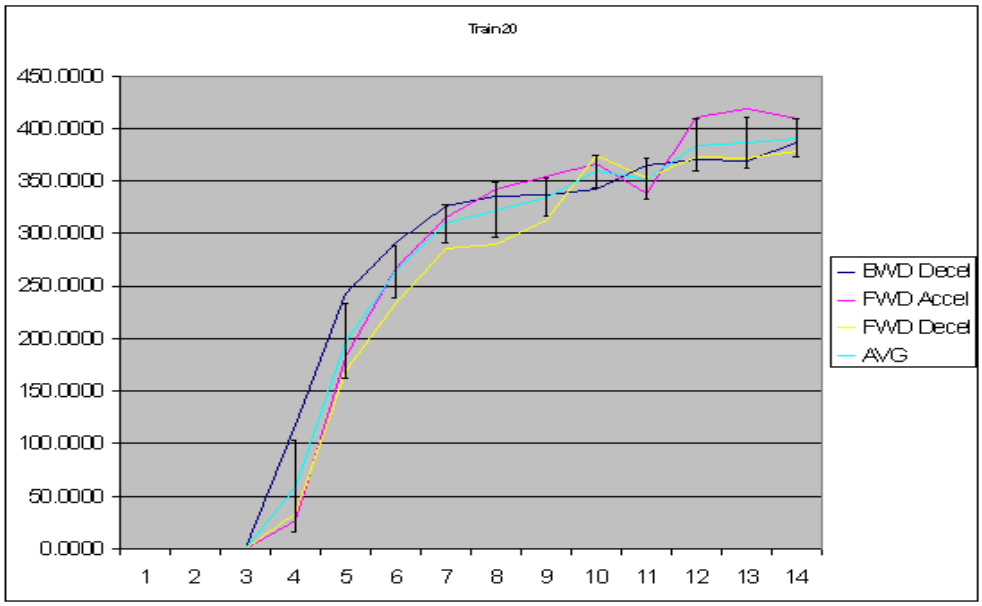


Figure 3.4: t_{20} speed parameters in mm/sec

predictably at low speeds. It can also be seen that the velocity for the same gear seems to be lower if the train is accelerating and higher if it is decelerating. The variance in speed is also fairly small.

For the train 20, all of those assertions fail. It is a simpler model of the locomotive than train 12 and behaves worse. It does not move or gets stuck on the smaller gear settings and there are no visible trends in the velocity vs. gear dependency other than generally higher gear meaning higher velocity. Nevertheless, with large enough variance, we can represent the velocity distribution as a function of the gear.

Chapter 4

Tracking

The high-level problems of train control such as collision detection and following the schedule can be solved if the system knows the positions of the trains on the track. The basic mechanism available for ascertaining the trains' positions is polling the status of sensors, positioned at various points along the track. When polled, the sensor reports a hit if the train passed that sensor since the last time it was polled. To be detected the train would have to move in the direction the sensor is set up to detect trains in. We wish to approximate the trains' actual positions at any time to the best of our ability.

We discuss the difficulties of tracking trains in our domain and first, attempt to solve them using existing techniques. We show that existing techniques are not suitable because they either don't scale or provide inaccurate results. We then develop our own technique based on existing techniques discussed in this section. Our technique can be applied to a new class of problems and solves the problems of scaling and accuracy. Extensive evaluation of our technique is presented in the next section.

4.1 Sources of uncertainty

4.1.1 Sensors

The sensors, which can malfunction and are therefore unreliable, can only tell us if they were hit since the last time they were polled. In addition to dealing with false hits and

missed real hits we also have a time and space uncertainty about the train's position when we get a real sensor hit. The time uncertainty translates into the space uncertainty given the train's speed and can be bounded only by querying the sensors on a regular basis.

4.1.2 Switches

The switches on the track can also be faulty. Sometimes they can fail to throw when directed to. The switches can also be thrown manually by hand. In the latter case the system has no way of knowing that the switch was flipped. There is no way of querying the switch state; so additional uncertainty about a train's position is introduced when a train is passing over a switch.

4.1.3 Trains

The trains can be directed to move at one of 14 gears. However the trains do not keep their actual velocity constant very well. The actual velocities to which these 14 gears correspond (θ_{sp}) vary from train to train and some trains keep their actual velocity constant better than others (θ_{var}). As a result of this, as time passes, the uncertainty of a moving train's position grows and can only be decreased when a sensor hit is reported.

4.2 Dealing with uncertainty

With all those complications it becomes apparent that the system has no hope of knowing the position of the train exactly. We therefore must settle for a probabilistic approach that would ideally give us the train position plus some bound on the error in train's position approximation. Alternatively we can, and as it turns out must, learn the probability density function (PDF) of a train's position to have an idea of where the train is. We can then ask queries like what is the probability that a given train is within 10 cm of the specified location.

What kind of a PDF are we likely to deal with? On the straight piece of track a normal distribution would likely do the trick, however when a train moves over a switch our PDF should intuitively split and have two local maxima, one on each of the two switch

branches. Reversing near the switch can also present a problem and so can faulty sensors. It is therefore best to not make assumptions about the characteristics of the PDF.

To keep track of the trains' positions we choose to use a method by Michael Isard and Andrew Blake [5] called Condensation, also known as Particle Filtering. Originally developed for vision this method allows us to keep track of general PDFs corresponding to trains' positions and describe the system dynamics in an intuitive way.

The basic idea is to represent, and more importantly track the dynamics of a PDF as a set of weighted snapshots drawn from that PDF – a weighted sample. By "snapshot" we mean an assignment of some values to every state variable. Weight assigned to the snapshot is a measure of how certain we are that the world looks like this snapshot, in other words the likelihood that the state variables have the values specified in the snapshot.

The method iterates through a number of steps.

1. **Sample the PDF.** We sample our tracked PDF by choosing, with replacement, elements of our weighted set such that each element has a probability of being chosen equal to its weight (Note: the weights are normalized). In doing so we get a set of snapshots, or a sample, that represent the likely states of the system after a previous iteration of the algorithm.

2. **Apply system dynamics.** Each of the chosen snapshots undergoes transformations according to system dynamics. For example, a train's position variable may be assigned a value a certain distance away depending on the time passed since last iteration and the speed of the train. Such transformations are performed for every variable in a snapshot to create a new snapshot, and the set of new snapshots forms a new sample. If the system dynamics are not deterministic, for example if the speed of the train is not known for certain, a snapshot may yield different snapshots after the system dynamics are applied.

3. **Assign weights to new snapshots.** Now that we have a new sample we consider the received observation. Snapshots in the new sample are assigned weights proportional to the probability of receiving an observation, equal to the one we have received, given the snapshot itself and its predecessor snapshot. The weights are then

normalized and we end up with a weighted sample to be used in step 1 at the next time step.

4.3 Applying particle filtering

4.3.1 Defining particles

In Particle Filtering terms a snapshot of the world is called a particle. As stated previously a particle must contain assignments of all state variables and the set of all particles and their weights represents the joint PDF of all the state variables. In the most basic terms for two trains, and two switches our particle might contain the following values

$$\left\{\begin{array}{rcl} T: & 10:50:12.325 & \text{- Current time} \\ Vel_1: & 125cm/s & \text{- Train 1 speed} \\ Vel_2: & 80cm/s & \text{- Train 2 speed} \\ Pos_1: & 3R100 & \text{- Train 1 position} \\ Pos_2: & 1T50 & \text{- Train 2 position} \\ Gear_1: & 10 & \text{- Train 1 gear} \\ Gear_2: & 6 & \text{- Train 2 gear} \\ Dir_1: & 1 & \text{- Train 1 direction} \\ Dir_2: & -1 & \text{- Train 2 direction} \\ Sw_1: & R & \text{- Switch 1 state} \\ Sw_2: & L & \text{- Switch 2 state} \end{array}\right\}$$

However, to adequately represent the joint distribution of these state variables the number of particles necessary in the sample would be prohibitively large. We are therefore interested in reducing the number of state variables in our particles.

We note, first of all, that some of those variables can be inferred from the actions or the environment and projection of joint PDF onto those variables is the delta function around specific values. $Gear_t$ is known from actions and T is known from the environment. It is therefore convenient to store those as separate variables outside of our particles.

Next we note that according to Equation 3.1 a projection of joint probability onto Vel_t values is a normal distribution with mean $\theta_{sp_{t,g}}$ and variance $\theta_{var_{t,g}}$. This is an assumption

that we have made. We assume the velocity distribution does not change with time and therefore requires no tracking. We can thus keep Vel_t values out of the particles since their PDF is known given a value for $Gear_t$.

Finally, since Dir_t values are relative to, and fully determined by Pos_t values and performed actions, we can keep track of Dir_t values in the background. Whenever we update the Pos_t value or process $Rev(t)$ command we're going to update the corresponding Dir_t values deterministically. This means that keeping the Dir_t values in the particles will require no more particles to represent the joint PDF than if they were not included – all that matters is keeping track of Pos_t values and Dir_t values come for free. In further discussion in this section we won't concentrate on those values anymore and simply imagine them attached to Pos_t values.

We are left with Pos_t and Sw_s values in our simplified particles. These are the variables whose PDFs, not just values, change with time and require tracking. As in the previous example for two trains and two switches, a particle might look like this:

$$\left\{ \begin{array}{rcl} Pos_1: & 3R100 & \text{- Train 1 position} \\ Pos_2: & 1T50 & \text{- Train 2 position} \\ Sw_1: & R & \text{- Switch 1 state} \\ Sw_2: & L & \text{- Switch 2 state} \end{array} \right\}$$

We are able to take state variables out of the particles because they have no uncertainty and therefore there is no need to keep track of them in every particle - a single value is enough. One exception is Vel_t values. We assume that their distribution is fixed and completely represented by its mean and variance for different $Gear$ values.

4.3.2 Factored Particles

The variables left inside the particles are correlated. Switch states imply where the train is likely to go when it passes this switch. Alternatively if an observation suggests that a certain train has followed a certain path this would affect our belief in how switches along that path are set.

We argue that although a probabilistic dependence between Pos_t and Sw_s exists it is weak. For the majority of the time one switch's state does not effect another train's

position unless the train is very close to the switch. Also weak are the dependencies of Pos_t values for different trains and Sw_s values for different switches. A technique proposed by Brenda Ng, Leonid Peshkin, and Avi Pfeffer [7] called "Factored Particle Filtering" allows us to take advantage of the fact that variables' interdependencies are weak and further simplify the particles by organizing the remaining variables into clusters.

Clusters are subsets of variables such that interdependencies between variables in different clusters are weak and can be broken with little effect on the quality of approximation. The quality can even be improved if the variables are indeed weakly correlated and the number of particles required for classical particle filtering is prohibitively large. We basically put every variable into its own cluster – one for every train containing that train's position and one for every switch containing that switch's state. As in the previous example for two trains and two switches the clusters would be $Ct_1 \equiv \{Pos_1\}$, $Ct_2 \equiv \{Pos_2\}$, $Cs_1 \equiv \{Sw_1\}$, and $Cs_2 \equiv \{Sw_2\}$.

Variables in every cluster can then be stored separately in what is referred to as factored particles. Still with the same example factored particles corresponding to our clusters might look like

$$\left\{ \; Pos_1 : \;\; 3R100 \; \right\} \left\{ \; Pos_2 : \;\; 1T50 \; \right\} \left\{ \; Sw_1 : \;\; R \; \right\} \left\{ \; Sw_2 : \;\; L \; \right\}$$

The advantage of factored particles is that the total number of particles is reduced because each sample of factored particles approximates only the marginal PDF of variables in the corresponding cluster and not the total joint PDF. Whereas classical factored particle filtering normally requires a number of particles exponential in the number of variables to maintain an acceptable quality of joint probability distribution, factored particle filtering maintains each cluster separately and therefore requires a number of particles linear in the number of clusters. This improvement means that we can even get a better approximation with less particles provided the variables in different clusters are really weakly correlated.

Factored Particles technique adds two more steps to the algorithm presented in Section 4.2 between Steps 1 and 2, let's refer to them as Steps 1a and 1b. These two steps together break the weak interdependencies that exist between clusters we've defined.

1. (a) **Project** operation begins when step 1 ends – that is when we have just created a sample of complete particles. For each particle and each cluster we simply

take those values from the complete particles that belong to the cluster and store these variables as a new factored particle. The treatment in [7] draws an analogy to the database project operations where complete particles define a relation and factored particles are projections of this relation on the subsets of attributes in the relation. Identical rows in the projection relation, however, are not merged.

(b) **Join** operation creates new complete particles from the factored particles by combining factored particles from every cluster while making sure that the factored particles picked from different clusters are consistent. For example if we consider that a train has certain length, we cannot pick another train's position such that the two trains overlap. Such conditions are made possible because Projection operation broke the conditional dependencies between different trains' positions. Again the analogy to database cross-product operation on multiple relations is made where each factored particle represents a relation.

As suggested in [7] it is convenient to think of these steps as a single Project-Join operation. It takes as input a sample of complete particles where conditional dependencies between variables exist and returns another sample of complete particles where these dependencies are removed.

There are a couple more things to say about this technique. A Project-Join operation creates an approximation of the joint PDF it took as input. By doing this approximation we introduce bias into the system. Suddenly even if we use infinitely many particles the approximated PDF will never match the true PDF because the variables are weakly correlated. So why is this still desirable?

To answer this question, we note that simply by having few particles we're introducing correlations between variables. For example, consider a system of two binary variables A and B. Suppose we only use 2 particles to approximate their PDF and the particles are: $\{0, 0\}$ and $\{1, 1\}$. This sample suggests a very strong correlation in that $Pr(A = 1|B = 1) = 1$ while $Pr(A = 1|B = 0) = 0$! Now if we knew that A and B were not correlated, if they were two coin flips for example, what we should have concluded from the sample is that $Pr(A = 1) = 0.5$ independent of B. This is the conclusion a Project-Join operation would allow us to make.

We first project the sample into two clusters to get two samples of factored particles $A : \{\{0\}, \{1\}\}$ and $B : \{\{0\}, \{1\}\}$. Once we join those factored particles we would get a new sample $\{0, 0\}, \{0, 1\}, \{1, 0\}$, and $\{1, 1\}$. The output sample of the Project-Join operation generally consists of more particles than the input sample. To keep the number of complete particles constant Ng et al. [7] propose a technique called importance sampling, which we are going to use as well.

The basic idea is to construct a complete particle in steps. Start with all variables in complete particle unset and then pick clusters one by one, sample a factored particle for that cluster, and assign values to corresponding variables in complete particle until all variables are set. This process is repeated until we get the required number of complete particles.

This method becomes difficult when clusters overlap. In a case like this, when picking a factored particle some of its variables may have already been set in the complete particle. We must only pick factored particles that are consistent with earlier choices made for the complete particle that is being constructed.

To correct for the fact that we were forced to only pick certain particles, we multiply the initial weight of the resulting complete particle by the fraction of factored particles in the current cluster that are consistent with the choices already made.

We now take yet another view of the process, and treat the cycle as starting with step 1b, following steps 2, 3, 1, and ending at step 1a. In this view, what we actually store are factored particles. We begin by creating complete particles and taking them through the Condensation process. Once the weighted sample is acquired we sample from it, project the new sample into factored particles, and store the result.

Now, what do we end up storing? What we store are the samples for the variables in the corresponding clusters. Why do we store the samples? Samples allow us to estimate the marginal probability distributions for variables in the cluster. We estimate the marginal distributions because it is computationally intractable to calculate them exactly. However, not all variables are the same. While Pos_t values are continuous Sw_s variables can only be one of the two values L or R. Therefore to store the exact marginal distributions of Sw_s variables we need one and only one value $r_s = Pr(Sw_s = R)$ for every switch s. And as it turns out we can efficiently maintain r_s values.

The technique of marginalizing out variables in the state space exactly is called Rao-Blackwellisation [4]. We are going to marginalize out the Sw_s values and maintain their marginals r_s throughout the algorithm. This means we will not have factored particles for Cs clusters because we can efficiently do better than sampling.

4.3.3 Creating complete particles

To start step 2 of the Condensation algorithm we need to create complete particles from factored particles. We use Importance Sampling method described in [7] with special treatment given to assigning values to Sw_s variables. This section describes how we produce a fixed number N of complete particles p_i with weights w_i.

$$\left\{ \begin{array}{rcc} w_1 \ , \ p_1 : & \{Pos_1, Pos_2, .., Pos_T & Sw_1, Sw_2, .., Sw_S\}_1 \\ w_2 \ , \ p_2 : & \{Pos_1, Pos_2, .., Pos_T & Sw_1, Sw_2, .., Sw_S\}_2 \\ .. & .. & .. \\ w_N \ , \ p_N : & \{Pos_1, Pos_2, .., Pos_T & Sw_1, Sw_2, .., Sw_S\}_N \end{array} \right\}$$

We first scan the action A generated at the previous step to update the deterministic parameters $Gear_t$, Dir_t. We also update the r_s value, setting it to

$$\left\{ \begin{array}{rll} r_s + (1 - r_s) * \theta r_s & \text{if } Sw(s, R) \in A & \text{(prob. } s \text{ was right plus prob. it was left and changed)} \\ r_s * (1 - \theta l_s) & \text{if } Sw(s, L) \in A & \text{(prob. } s \text{ was right and did not change)} \\ (1 - \gamma) r_s + \gamma(1 - r_s) & \text{otherwise} & \text{(note } r_s \text{ goes to 0.5 over time)} \end{array} \right. \tag{4.1}$$

where $0 \leq \gamma \leq 1$ is the probability that switch gets thrown when no action is given for that switch. To clarify:

- If the switch was directed to throw right the probability that it is right afterwards is equal to the probability that it was already right plus the probability it was left and changed when it was told to.
- If the switch is directed to throw left the probability it is right afterwards is equal to the probability it was right and failed to switch.

- Finally, if the switch was not told to throw either way we become less certain of its state with every iteration of the algorithm. Passing trains may affect the mechanism as well as people can be throwing switches manually. We assign constant probability γ to the event of switch throwing without the system's knowledge. The value of r_s gets degraded in the absence of actions for switch s. Note that as time goes by r_s tends to 0.5 meaning switch s is as likely to be thrown right as is to be thrown left.

Now for each i we start with $w_i = 1$ and all variables in p_i unset. For each cluster Ct_t in turn, we randomly pick a value from associated sample for Pos_t that is consistent with values already chosen $\{Pos_1, .. , Pos_{t-1}\}$ for other variables. We multiply w_i by the fraction of factored particles in Ct_t sample consistent with variables already chosen. By the end of this process we end up with particles p_i where all Pos_t variables are assigned. We do not initialize the Sw_s variables and simply remember, that should we wish to initialize those values we would pick a value R with probability r_s and L otherwise.

We next pick values for Vel_t according to Equation 3.1 for every Pos_t value in every particle p_i. Then if in some particle p_i for some train t moving a distance $Dir_t{*}Vel_t{*}(T'-T)$ from position Pos_t involves going over a switch s in forward direction, and Sw_s is not yet assigned a value in p_i, we set Sw_s to R with probability r_s in particle p_i.

This concludes the setting of variables and calculating the initial weights of complete particles p_i. We now need to apply the system dynamics to get the new sample of complete particles p'_i.

4.3.4 Applying the system dynamics

We are now ready to apply Equation 3.2 to every Pos_t value in every particle to get new positions Pos_t^+. And that would complete step 2 of Condensation algorithm if the trains could pass through each other. We applied the system dynamics in such a way that each train moved a certain distance away from their current positions. We did not take into account that in the same particle a train cannot go the distance if another train is in the way. Note, that a train in one particle may pass through a train in another particle because those are two different versions of the world.

We need to detect any collision that might have happened between the trains. We should note that the collisions we are looking for do not imply that the trains necessarily

collide, or even that it is likely they would collide. All we are concerned with is that in a single particle all $Pos_t \rightarrow Pos_t^+$ transitions are feasible when considered together.

For example if train 1 goes from point A to point B while train 2 goes from point B to point A when there is only one path from A to B we understand that both transitions are not possible at the same time. We therefore must modify the final destinations for both trains to get a feasible transition for the particle. We do this by finding a point C between A and B where the collision would occur. We have selected speeds Vel_1 and Vel_2 already and we know the initial locations of the trains $Pos_1 \equiv A$ and $Pos_2 \equiv B$ and we know the time they start moving is T. We can therefore easily solve this problem and also find time T_{col} where both trains reach the point C. We resolve the collision by setting $Pos_1^+ = Pos_2^+ = C$.

For every pair of positions Pos_l and Pos_m in a particle p_i we need to determine if going to positions Pos_l^+ and Pos_m^+ means the trains would collide. If they would, we need to calculate time $Tcol_{l,m}$ when these trains will be at the same spot. After we sort all $Tcol$ values we can start resolving corresponding collisions starting with the one that is going to occur first. Particles involved in collisions would back off covering smaller distance then they were originally assigned. After all collisions are resolved Pos_t^+ values become Pos_t' values and there will be one for every Pos_t in every particle p_i.

We have now completed Step 2 of Condensation and proceed to Step 3.

4.3.5 Processing an observation

The dynamics of our system at current step are defined as going from particle p_i to particle p_i'. Each transition is a hypothesis of what the world looked like and how it has changed. We now use the observation to evaluate our hypotheses. In terms of our previous example a hypothesis might be

$$p_i : \left\{ \begin{array}{ll} Pos_1 : & 3R100 \\ Pos_2 : & 1T50 \end{array} \right\} \rightarrow p_i' : \left\{ \begin{array}{ll} Pos_1' : & 3R120 \\ Pos_2' : & 1L10 \end{array} \right\}$$

Note that we do not specify the modifications to Sw_s values because observations we receive do not directly affect these variables. Given an observation O we need to assign weights

$$w_i' = w_i * Pr(O \mid p_i, p_i')$$

We must then normalize w'_i so that $\sum w'_i = 1$.

Recall that weights w_i were created because we sampled the factored particles in order to create complete particles. These weights are therefore the prior probability of each particle. $Pr(O \mid p_i, p'_i)$ are the means of testing the hypothesis. Let's see how it is being calculated.

Transition from p_i to p'_i by itself implies that certain sensors were hit. For example, if there was a sensor z_1 at $3R110$ facing in the direction of increasing $3R$ coordinate, the case above would imply that $z_1 \in O$. On the other hand if there was a sensor z_2 at $3T10$ facing any direction, the case above would imply that $z_2 \notin O$.

Let O_e be the observation implied by the $p_i \rightarrow p'_i$ transition and U be the set of all sensors then

$$Pr(O \mid p_i, p'_i) = PrH^{|O \cap O_e|} * PrF^{|O - O_e|} * PrM^{|O_e - O|} * (1 - PrF)^{|U - O \cup O_e|} \tag{4.2}$$

Where PrH is the probability that a train passes a sensor and the sensor correctly reported a hit. Each sensor hit that is in both O and O_e multiplies the weight by PrH. PrF is the probability that a train did not pass a sensor, but a false hit was reported by the sensor. This could happen if a human tripped a sensor manually. Each hit that is in the O, but not in O_e multiplies the weight by PrF. PrM is the probability that train passed the sensor, but the sensor failed to report a hit. Each sensor hit that is in O_e, but not in O multiplies the weight by PrM. The last term in the equation corresponds to the sensors that were not hit and are not reported.

Step 3 completes when weights w'_i are calculated and normalized. We proceed to Step 1.

4.3.6 Resampling and factoring particles

We sample new complete particles with replacement from p'_i with probabilities w'_i to get N new particles $\widetilde{p}_i$ and proceed to Step 1a. We project the Pos_t values into their respective clusters, however Sw_s values require special care. For each switch s we sum up the weights of all particles where $Sw_s = R$ as w^r_s and sum up the weights of all particles where $Sw_s = L$ as w^l_s. Note, that because some particles would have Sw_s unassigned $(1 - w^r_s - w^l_s) \geq 0$

is the total probability of all particles that don't have Sw_s assigned. Then new marginals are computed as follows

$$r'_s = 1 * w^r_s + 0 * w^l_s + r^*_s * (1 - w^r_s - w^l_s) \tag{4.3}$$

where r^*_s is the value of r_s after applying the system dynamics function – Equation 4.1 to r_s value from previous time step.

We have now finished an iteration of the tracking logic and wait for the next time step to repeat it all over again.

4.4 Dynamic particle filtering with context-specific correlations

Very early in the experimental stage we started seeing problems with the factored particle filtering approach presented so far. This lead to revisit the way we characterized variables' interdependencies. We realized that interdependencies were unusual in a specific way that was not handled adequately by the presented algorithm based on existing techniques. As such we ended up creating our technique based on factored particle filtering.

In this section we explain the shortcomings of existing techniques and present our technique to deal with specific types of correlations that exist in our system and other systems that share their structure with ours.

4.4.1 Preliminary experimental results

We first say a few words about results of running the proposed factored particle algorithm on an experimental layout and on a simulator. We hoped that the system would be able to determine with good accuracy the states of the switches by monitoring train movements over time. Through experiments, it has become evident that although this does happen to some extent, it cannot be considered satisfactory.

We found that posterior distribution of a switch state variable changes with respect to its prior distribution in response to a train movement if and only if the following conditions are met: A train passes this switch in the forward direction and hits a sensor on either the right or the left branch of the switch in a single time step.

Consider Figure 4.1. If we had a complete particle p_i transitioning into particle p'_i where a train went form point A to point C

$$p_i : \left\{ \begin{array}{ll} Pos_1 : & A \end{array} \right\} \rightarrow p'_i : \left\{ \begin{array}{ll} Pos'_1 : & C \\ Sw_1 : & R \end{array} \right\}$$

the switch's state would be initialized to R position. This transition implies that the sensor on the path from A to C should have been hit. The weight w'_i assigned to the particle p'_i would then depend on whether this sensor hit was in the observation or not. If an observation told us that the sensor on path from A to C was hit, w'_i would grow, and when

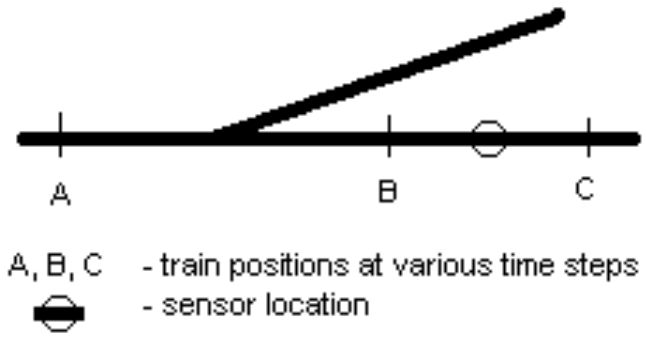


Figure 4.1: Possible state transitions

the complete particle got refactored the marginal probability $r_s = Pr(Sw_s = R)$ would grow as well.

If, on the other hand, our particle corresponded to a move from A to B, then the switch state cannot be confirmed by an observation. What we mean is that this particle would be weighted independently of the switch assignment because we have nothing to tell us which branch the train is on.

Logically, this makes sense. If a train passed from A to C we do know that the switch is thrown R. If the train passed from B to C we have no indication of the switch's state. In order to make conclusions about the switch state in the latter case, we would have to remember that before B the train was at A. With the switch variables decoupled from train variables factored particle filtering does not allow us to do this.

One way to overcome this problem would be to introduce persistent interdependencies between switch states and train positions. If we do this we can no longer factor the complete particles into separate train and switch particles. Every train position would have to be coupled with all switch states.

This would present a number of other complications. If we still factor particles such that every train has its own cluster (train's position plus all switch states) only train positions with the same switch assignments will be consistent. This is a severe limitation as the

trains can be in different parts of the track and have absolutely no effect on each other.

Further, the set of factored particles for each cluster would have to approximate the join probability of corresponding train plus all the switches, and as a result, the required number of factored particles would grow exponentially in the number of switches. The number of switches is the physical parameter of the track, we therefore introduce a computational limit on the size of the track, which is especially undesirable.

Finally, this approximation is too crude because only transitions like the one from A to C in the earlier example really tell us something about the switch position and not subsequent transitions. Yet, we'd still like the system to infer the switch states from train movements better that it does with the present algorithm.

4.4.2 Context-specific correlations

While designing the factored particle filtering approach we have made an assumption that train position variables and switch state variables are weakly correlated to the point that they can be split into different clusters. In the previous section we've shown that in particular circumstances train positions are strongly correlated to switch states. However it is a specific train that is correlated with a specific switch and for a very short period of time. As such, this correlation does not lend itself easily to any of the techniques we've discussed so far. We call them context-specific correlations (analogous to context-specific independence [1]), meaning that they exist in the specific context eg. a switch is correlated with a train if the train has just passed that switch.

We wish to be able to capture context-specific correlations such that for a short period of time a train would be correlated to the switches it just passed. We note, however, that every particle in a train's factored particle set moves independently, so each particle can potentially pass different switches at different points in time. We therefore have to link each factored particle to a particular switch state assignment independently.

As time passes we want those assignments to expire ending the correlation between the corresponding variables. We also want to discount the assignments that were made a few time steps back.

4.4.3 Factored particle filtering with caches

Caches

When a train passes a switch in the forward direction we initialize the Sw_s variable using the marginal r_s as described in Section 4.3.3. We now wish to store the value assigned to that switch and associate this value with the new position of the train Pos'_t – the position where the train ends up in *because* of the assigned Sw_s value.

Consider the train factored particle set. Over the course of a few time steps each particle follows some route. Two particles can correspond to positions far apart, and their routes can be very different, passing different switches at different points in time. We therefore extend not the cluster as a whole, but particles themselves. So, every particle in the train factored particle set would contain a position value, plus zero or more switch state assignments, let's call them *Caches*. A cache stores an assignment of switch variable Sw_s and carries it over into the next time step linked with the specific position value Pos_t. We also wish to remember when the cache was created and therefore store the time step when the assignment was made as part of our cache. So for a train factored particle set with three elements we may store something like that:

$$\left\{ \begin{array}{ll} Pos_1 : 1R30 & \{Sw_1 : R \text{ at } T_{100}\} \\ Pos_1 : 1L50 & \{Sw_1 : L \text{ at } T_{101},\ Sw_2 : L \text{ at } T_{98}\} \\ Pos_1 : 2R10 & \{\} \end{array} \right\}$$

An important aspect to cover is when do the caches expire. A cache is valuable as long as it can say something about the present state of the switch better, than r_s value. An old cache tells us the state the switch was in some time ago. At every time step, according to our system dynamics, the switch has probability γ of flipping the other way. Therefore as time passes, cache's value decreases and r_s, which is updated every time step, becomes a better indicator of the switch's position.

Other events that significantly decrease cache's value are commands given to the corresponding switch. The command will affect the r_s value according to Equation 4.1 and because actions are expected to be more or less reliable, caches can be dropped at that point in time.

In this work we chose to delete the cache after a fixed number of time steps or sooner,

if a command is given to the corresponding switch. Generalizing, we can say that the aging of cache should apply to other domains and commands correspond to events that significantly changes the marginal distribution of variable corresponding to a cache.

Factored particle filtering with caches

We must now modify step 1b of our algorithm. When sampling factored particles for different trains we must make sure we don't pick particles that have inconsistent caches – that is different switch value assignments at the same time step. Suppose, for example, that we pick the following particle for train 1 $\{Pos_1 : 1T30\ \{Sw_1 : R \text{ at } T_{100}\}\}$. Then a particle for train 2 $\{Pos_2 : 2R10\ \{Sw_1 : L \text{ at } T_{100}\}\}$ is inconsistent and needs to be resampled. Note that if the time stamps are different then the caches are consistent and the most recent cache represents the most likely assignment of the switch variable.

Although factored particle filtering algorithm prescribes that the initial weight of a particle must be multiplied by the ratio of all values consistent with values already chosen, we argue that the probability of a particle being inconsistent is very low. Only if two trains pass the same switch in a single time stamp would we get an inconsistency. Most likely this would never happen. We therefore assume that all particles are consistent and if we do find a case of inconsistency we would simply pick a different value without adjusting the initial weight.

When sampling a new complete particle it is also a convenient time to detect any old caches. If a factored particle contains an old cache, or a cache that was invalidated by a switch command this cache needs to be ignored and not considered in the resulting complete particle. We set the maximum number of steps for a cache to be valid in advance based on the discount factor and common sense.

Let's look at an example. Suppose the maximum age for a cache is 3 and we are at time step T_{10}. We have two trains and the following factored particles:

$$\left\{ \begin{array}{ll} Pos_1 : 1T30 & \{\} \\ Pos_1 : 1T50 & \{Sw_1 : R \text{ at } T_6,\ Sw_2 : L \text{ at } T_9\} \\ Pos_1 : 2R10 & \{\} \end{array} \right\}$$

$$\left\{ \begin{array}{ll} Pos_2 : 3T10 & \{Sw_1 : R \text{ at } T_8,\ Sw_2 : R \text{ at } T_7,\ Sw_5 : R \text{ at } T_9\} \\ Pos_2 : 4L90 & \{\} \\ Pos_2 : 2T10 & \{\} \end{array} \right\}$$

suppose we picked the second particle from first cluster and first particle from the second cluster. Suppose that action at previous time step A_9 contained a command for switch 5. We would create the following complete particle:

$$\left\{ \begin{array}{lll} Pos_1 : 1T50 & \{ & , Sw_2 : L \text{ at } T_9\} \\ Pos_2 : 3T10 & \{Sw_1 : R \text{ at } T_8 & , Sw_2 : R \text{ at } T_7\} \end{array} \right\}$$

we see that one of Pos_1 caches got invalidated due to its age. Furthermore one of Pos_2 caches got invalidated because a command was given to the corresponding switch. These caches do not exist anymore.

We now need to initialize those Sw_s values that have valid caches. Knowing the cache age we can re-create the proper PDF for the corresponding switch. In each time step, a switch has probability γ of changing its state. We can easily calculate the probability that the switch changed its value since the cache was created and with that probability assign Sw_s value that is different from cache. Otherwise Sw_s gets the same value as the one in cache.

However, we are going to use a more simple scheme and show that it has the same effect on the r_s values as the one we've just described. We are going to assign Sw_s variable the value specified in the most recent cache, and when calculating r_s value, compensate for this, by adjusting the weights used in computation. Let us see how our simplified scheme would work by going back to the previous example. Our complete particle would look like

this.

$$p_i \left\{ \begin{array}{lll} Pos_1 : 1T50 & \{ & , Sw_2 : L \text{ at } T_9\} \\ Pos_2 : 3T10 & \{Sw_1 : R \text{ at } T_8 & , Sw_2 : R \text{ at } T_7\} \\ Sw_1 : R & \text{Due to } Pos_2 & \\ Sw_2 : L & \text{Due to } Pos_1 & \end{array} \right\}$$

First of all, we note that what we are really interested in is maintaining the proper r_s values. These values contain the knowledge we have about each switch state. Sw_s values exist only temporarily, until the particles are factored, and are really just means of tracking r_s values – the encoding of switch PDFs.

Recall Equation 4.3 that specifies how we calculate r_s values based on the weighted set of complete particles. The assignment of Sw_s variable will determine if the particle's weight is counted towards w_s^r or w_s^l term. Let us go back to our example and apply system dynamics to the sample particle to obtain its weight.

$$p_i \rightarrow p_i' \left\{ \begin{array}{lll} Pos_1' : 1T80 & \{ & , Sw_2 : L \text{ at } T_9\} \\ Pos_2' : 3R20 & \{Sw_1 : R \text{ at } T_8 & , Sw_2 : R \text{ at } T_7,\ Sw_3 : R \text{ at } T_{10}\} \\ Sw_1' : R & \text{Due to } Pos_2 & \\ Sw_2' : L & \text{Due to } Pos_1 & \\ Sw_3' : R & \text{Due to } Pos_2 \rightarrow & Pos_2' \text{ transition} \end{array} \right\}$$

In this transition, train 2 passed switch 3 in forward direction ending up on the right branch of switch 3. We have initialized Sw_3' value by picking R with probability r_3 and created a new cache and attached it to Pos_2' value.

We process the observation as we normally do and obtain a weight w_i' for our new complete particle p_i'. Now we are about to apply Equation 4.3 while refactoring the complete particle. Let us show that we can calculate r_s value correctly despite the simplified scheme we have used to assign Sw_s value in p_i.

In our example Sw_2 is assigned the value of L due to a cache that is one time step old. If we were to properly assign Sw_2 its value then with probability γ we would assign Sw_2 a value R. This is because γ is the probability the switch state changed in one time step. This means that when *properly* calculating r_2 using Equation 4.3, w_i' could be counted towards w_s^r with probability γ and otherwise it is counted towards w_s^l term. Well we can

still make w'_i be counted correctly if we add $\gamma * w'_i$ to w^r_s and $(1-\gamma) * w'_i$ to w^l_s. The same argument works for older caches with the only difference that the portion going to each of the two sums (w^r_s and w^l_s) is harder to calculate.

We note that the portion of the weight going to each of two sums is determined by γ and the age of cache only. We can therefore pre-compute these portions for all valid ages of the cache. Let us denote by γ^x the portion of the particle's weight going to the term where we would normally sum the weights for particles with the opposite Sw_s value assignments. In our previous example $\gamma^1 * w_i$ is added towards w^r_s. We note that $\gamma^1 = \gamma$, but $\gamma^2 \neq \gamma * \gamma$. Also newly created caches (new switch assignments made due to train movements like Sw_3 in previous example) must not be discounted therefore $\gamma^0 = 0$. In general $\gamma^x = (1-\gamma^{x-1}) * \gamma + \gamma^{x-1} * (1-\gamma)$.

To calculate r_s values correctly despite our simplified scheme we don't even change Equation 4.3, but we do change how we calculate w^r_s and w^l_s values.

$$w^r_s = \sum_i (1-\gamma^{x(i,s)}) * w'_i + \sum_j \gamma^{x(j,s)} * w'_j$$

$$w^l_s = \sum_j (1-\gamma^{x(j,s)}) * w'_j + \sum_i \gamma^{x(i,s)} * w'_i$$

for all i such that $Sw_s = R$ in p'_i and j such that $Sw_s = L$ in p'_j and $x(i,s)$ is the age of the cache that was used to initialize each Sw_s value in particle i.

Generalizing to other domains

The solution that was presented is highly tailored to our application's domain. Our results, presented later, indicate that this method produces the best results over the set of proposed solutions. We would like to understand the specifics of this solution and see where else Factored Particle Filtering With Caches may be applicable.

Let us first consider the Dynamic Bayesian Network that represents our problem on Figure 4.2.

The most important aspect is that position and switch variables really evolve completely independently most of the time. A single event that is stored in a cache completely specifies the correlation of the two variables. This condition is necessary for us to be able to keep the correlated variables in separate clusters.

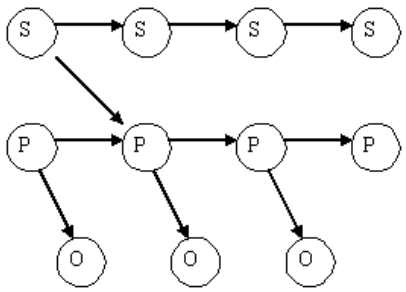


Figure 4.2: DBN for our problem's domain

Another necessary condition is that dynamics of the cached variable must be such that knowing a value a few time steps back, can be used to derive the variable's current value better, that by considering current observation-action pair. In the trains domain, for example, we simply can't infer a current state of the switch from our observations, and actions are noisy. We can only infer a position of a switch at the time the train passed over it and cache captures exactly this information.

If these two conditions are met, then a weight given to a complete particle after processing an observation gives us the marginal probability of cached variable's value at the time the cache was created. If we can use this information to construct a PDF at present time step, the cache is sufficient for tracking the corresponding variable.

We thought about other problems that would resemble ours. A completely unrelated, but structurally similar problem may be that of tracking infectious disease outbreaks in different areas of the world by monitoring patients in one country. When a person comes to a doctor with a rare disease it is sufficient to ask which countries that person has been to in the last little while to update our belief about where the outbreak may be going on. In that sense each person is a variable in its own cluster and so are countries. Not factoring these variables means all people and all countries belong to the same giant cluster.

This problem is not as dynamic as trains and particle filtering may not be the approach to take, but disease monitoring shares a lot of the structure with trains. For our purposes countries are effectively unobservable and similar to switches observations only tell us about

variable's past value when it was correlated with the observable variable. This tells us that the structure is not unique to the trains domain and factored particle filtering with caches is likely applicable to other problems.

4.4.4 Dynamic factored particle filtering

A more general approach to handling context-specific correlations is simply create clusters dynamically. If we know an event that triggers a correlation, like train passing over a switch, we can create a cluster with just that one train and the switch it has just passed. If that train passes another switch, that switch would have to be added to the cluster. This point describes how a cluster can grow.

Another unfortunate side-effect is that if a train passes a switch that is already in the cluster with some other train, clusters would become overlapping, which in turn will make the joint operation more complex. Clusters may overlap by more than one variable and importance sampling becomes exponentially more complex with the number of overlapping variables.

We can battle the problems of growing and overlapping clusters by breaking them up. If we knew when correlations ended, this would be a good time to break a cluster. In her work Brenda Ng [6] considered a problem of mars rover. Wheels on the two sides of the rover would generally move independently, except when the rover would be on the incline. The cluster was created while the rover was on the incline and broken once the rover was level again.

But in trains domain and possibly other domains, correlations do not end, but rather gradually fade away. For comparison reasons we've implemented a dynamic factored particle filtering approach for our trains problem and set the same rules for breaking a cluster up as for deleting our caches.

This approach is more general and makes fewer assumptions about the problem's structure, yet it increases the variance of the tracking process. The clusters must maintain the approximate joint PDF for all the switches in it, while caches allowed us to calculate the marginals exactly.

Chapter 5

Results

We have implemented and tested four solutions to the trains problem. They are: Plain Particle Filtering (PF), Factored Particle Filtering (FPF), Factored Particle Filtering With Caches (CFPF), and Dynamic Factored Particle Filtering (DFPF). We have run a number of experiments on a simulator and on a real experimental track. In this chapter we report the results of those experiments.

The track layout used for these experiments is depicted on Figure 5.1. The simulator was made to match the layout of the physical track. Figure 5.1 also shows bidirectional sensors as black squares, 10 most likely positions for every train as numbered circles, and switch numbers followed by corresponding r_s values near the locations of switches.

5.1 Simulation-Based Evaluation of Tracking Accuracy

For the first series of experiments we have used a simulator. It gives us a unique ability, knowing the true simulated trains' locations and switches' states, compare them against the algorithm predictions.

Three simulated trains traversed a path along the track. Keeping the trains on correct routes required flipping switches and reversing the trains. Avoiding collisions required stopping trains and setting them back in motion at various points along the route.

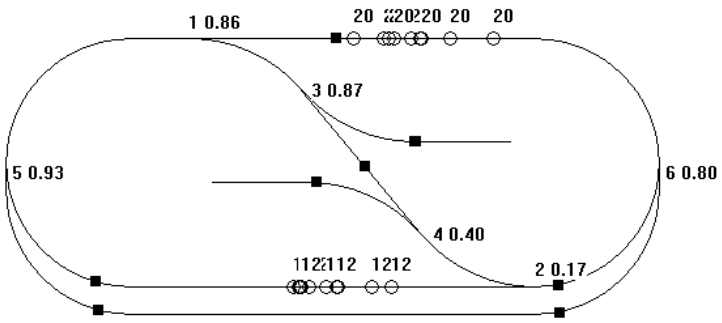


Figure 5.1: Sample system's output and track layout

For trains we calculate the expected error by multiplying each particle's weight by the distance from predicted train's location to the simulated location and sum over all particles. The average expected absolute error is reported for 1000, 500, and 250 particles in Table 5.1. The average is taken over the duration of test runs.

Number of particles	1000	500	250
Plain (PF)	79.45	143.78	227.10
Factored (FPF)	273.92	252.65	305.07
Dynamic (DFPF)	112.29	149.15	178.60
Caches (CFPF)	84.26	103.70	136.06

Table 5.1: Average Expected Error in Train Position in **mm**

We can see that while for 1000 particles PF gives the best results, its performance is significantly degraded when the number of particles is decreased to 500. For 500 and 250 particles CFPF gives the best results.

For switches we report the probability assigned to the correct simulated state of the switch. Average probability assigned to the correct simulated switch status over the duration of test run are reported for 1000, 500, and 250 particles in Table 5.2.

The number of particles does not seem to affect the switch prediction accuracy by much.

Number of particles	1000	500	250
Plain (PF)	0.81	0.81	0.84
Factored (FPF)	0.58	0.60	0.61
Dynamic (DFPF)	0.77	0.76	0.76
Caches (CFPF)	0.80	0.81	0.80

Table 5.2: Switch State Tracking Accuracy

We attribute this to the fact that switches are binary variables, and the sample necessary to accurately track their distribution can be quite small. The reason we cannot decrease the number of particles even further is because we need at least that many to track the train positions.

We note that CFPF comes in close second to PF, and DFPF is not far off. With sufficient number of particles PF represents the best, unbiased approximation to the true joint PDF. CFPF, DFPF, and FPF are all approximations of PF in that sense and CFPF comes closest.

It is interesting to see that PF can do quite well for a subset of variables, while other variables in the same run are approximated much worse.

5.2 Observation Prediction Accuracy

The next set of experiments were run using a real track with real trains. A playback file that contained the times the sensors were hit and actions given to the track, was recorded and used for evaluation of all four techniques off-line. The playback file contained the two trains that were run in various add-hoc paths around the track. Again the trains were often reversed or stopped to avoid collisions and switches were thrown to make trains behaviour interesting. Most of the time both trains were in motion.

When a sensor hit was observed, we calculated how many particles were consistent with that hit and divided by the total number of particles. This metric gave us the accuracy of observation prediction. It is a number equivalent to the percentage of sensor hits accurately predicted by our system. We took an average value for the duration of the run and made

50 runs. We report the average over the 50 runs for 5000, 1000, 500, and 250 particles in Table 5.3.

Number of particles	5000	1000	500	250
Plain (PF)	0.47	0.26	0.14	0.08
Factored (FPF)	0.39	0.37	0.36	0.30
Dynamic (DFPF)	0.49	0.46	0.43	0.29
Caches (CFPF)	0.47	0.47	0.45	0.34

Table 5.3: Observation Prediction Accuracy

We note that 100% in this metric is not the goal for the system to achieve. The only way to have a 100% observation prediction accuracy is to have an absolutely deterministic system. The maximum achievable value for this metric is highly dependent on the variance inherent in the system. To know if we are performing well, we ran plain particle filtering algorithm with 25,000 particles. This took much more time to compute, but the accuracy we got was only 49%. This tells us that we are very close to the maximum achievable observation prediction accuracy given our system's variance.

Because the real track was characterized by a higher degree of variance than the simulator, we found that PF requires 5000 particles to compare with the rest. For 1000, 500, and 250 particles CFPF gives the best results with DFPF being second. Also CFPF degrades the slowest as the number of particles is decreased. A 47% accuracy in this case means that 47 per cent of all particles predicted a hit while others cover the system's variance.

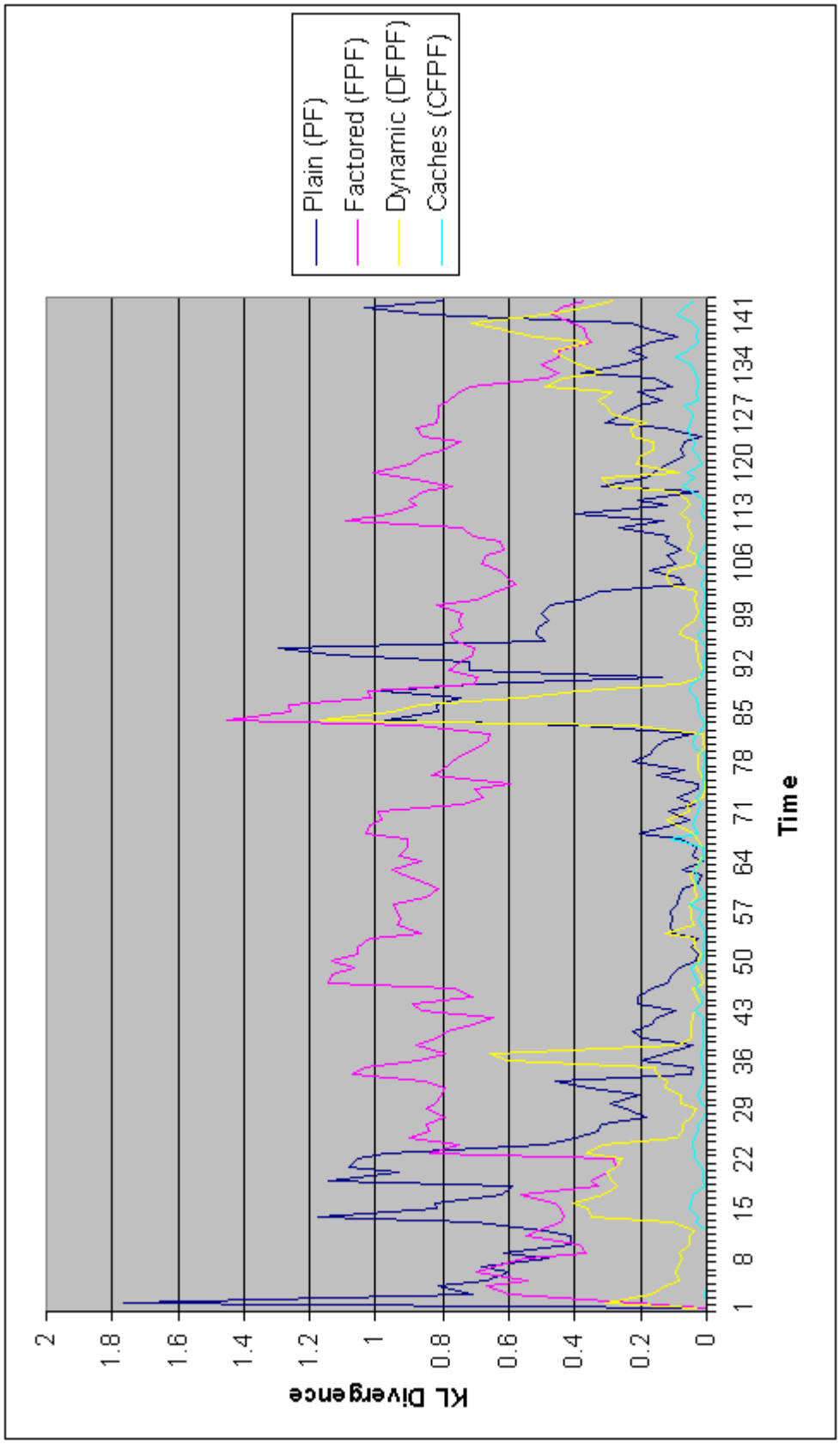
5.3 KL Divergence with respect to the Gold Standard

For the last set of experiments we wanted to evaluate the quality of tracking switch PDFs. Because we did not know the true states of switches we have used the data received by running PF with 25,000 particles as a gold standard. We then ran PF, FPF, DFPF, and CFPF with 1000 particles each and computed the KL divergence of those four runs versus the gold standard. We have used the same playback file that was used in the previous experiments.

KL divergence (Equation 5.1) [3] is a general metric that tells us the difference between two PDFs. KL divergence is non-negative, and zero if the two PDFs are equivalent.

$$D_{KL}(P \mid\mid Q) = \sum_{i} P(i) \log \frac{P(i)}{Q(i)} \tag{5.1}$$

Figure 5.2 shows the KL divergence for switch PDFs. We can see that KL divergence for CFPF is closest to zero out of the four techniques. CFPF also presents no spikes evident in other techniques. The spikes are likely caused by events that are comparably harder to track than the general dynamics of the process.


Plain (PF)
Factored (FPF)
Dynamic (DFPF)
Caches (CFPF)
KL Divergence
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
127
134
141

Chapter 6

Conclusions

We have considered a real world problem of train monitoring, which is a necessary part of controlling the train system in CS 452. We have presented a principled solution for this problem using the latest techniques in AI. The theory contained in this work will become a part of material taught in CS 452 in the future.

Due to interesting properties of the problem, we were able to create a functionally new approach in the framework of Factored Particle Filtering for tracking stochastic processes. If strong correlations occur between otherwise independent clusters of variables for short periods of time, our technique allows capturing and using the information about the triggering event in a "cache" for proper inference, without adjusting the decomposition of variables. For a class of problems that to our knowledge has not been considered so far, our technique allows tracking context-specific correlations with precision of unbiased plain Particle Filtering while keeping all variables factored.

We have compared our technique against other techniques some of which, like plain Particle Filtering, have been around for awhile, and others, like Dynamic Factored Particle Filtering, were only recently proposed. In a series of experiments involving simulation and real track data we have shown that our technique provides the best results in terms of both accuracy and scaling.

Our experiments show, that for the trains problem Factored Particle Filtering With Caches gives the best overall results. Dynamic Factored Particle Filtering, which comes in second, proved to be a lot harder to implement. The resampling process becomes very

complex due to cluster sizes and overlaps. Recall that we keep variables in clusters for the same amount of time we would otherwise hold a cache.

Factored Particle Filtering with the same decomposition as our technique provides inaccurate results, and plain Particle Filtering requires much more particles to keep up with our technique, in other words plain Particle Filtering does not scale.

For future work we would like to consider how the generated PDFs can best be used for making decisions for controlling the trains. Recall that what we end up with is a weighted sample of possible train positions. This information needs to be aggregated so that the train can successfully be routed. A most simple approach is to calculate a weighted mean and variance of our sample, but perhaps there are better ways.

Bibliography

[1] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Uncertainty in Artificial Intelligence*, pages 115–123, 1996.

[2] X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In *Uncertainty in Artificial Intelligence*, pages 33–42, 1998.

[3] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley and sons, 1991.

[4] A. Doucet, N. de Freitas, K. Murphy, and S. Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *Uncertainty in Artificial Intelligence*, pages 176–183, 2000.

[5] M. Isard and A. Blake. CONDENSATION—conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–18, 1998.

[6] B. Ng. *Factored inference for efficient reasoning of complex dynamic systems*. PhD thesis, Harvard University, Cambridge, MA, 2006.

[7] B. Ng, L. Peshkin, and A. Pfeffer. Factored particles for scalable monitoring. In *Uncertainty in Artificial Intelligence*, pages 370–377, 2002.