Planning in Continuous Domains with MDPs

Neil Bacon, Supervisor: Scott Sanner

October 29, 2008

Abstract

Markov Decision Processes (MDPs) are typically applied to finite discrete domains. Here we focus on combining some recently proposed techniques to apply MDPs to planning in continuous domains. Extensions to Algebraic Decision Diagrams are proposed as an efficient representation of real valued functions of both real and Boolean arguments, amenable to the mathematical operations required to solve MDPs. An implementation is provided and applied to a simple traffic control planning task.

Contents

1 Background

1.1 Markov Decision Process

Sutton and Barto [10] provide an easily accessible treatment of MDP models and solution techniques applied to Reinforcement Learning and Planning. In this work we focus on planning which uses a model of the environment to arrive at an optimal policy without interaction with the actual environment.

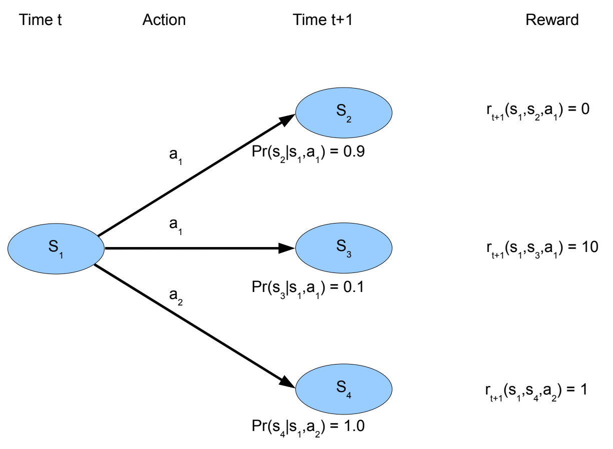


Figure 1: Example Markov Decision Process

A finite discrete Markov Decision Process (MDP) is characterised by

$$M = \langle S, A, \mathcal{P}, \mathcal{R} \rangle \tag{1}$$

where

- S is the finite set of states,
- A is the finite set of actions,
- $\mathcal{P} = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the stochastic Markov state transition model and
- $\mathcal{R} = \mathbb{E}(r_{t+1} | s_t = s, a_t = a, s_{t+1} = s')$ is the expected value of the next reward.

The transition probabilities satisfy the Markov property: that is that given the current state and action, the next state is conditionally independent of all earlier states. Many planning problems can be formulated as MDPs [10].

The long term return of an episodic process from time step t to final time step T is

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{2}$$

For a continuing process ($T = \infty$) we need to use discounted rewards to avoid infinite returns

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned} \tag{3}$$

where $0 \leq \gamma \leq 1$ is the discount rate. The episodic and continuing cases can be unified by considering episode termination to be the entering of a special absorbing state that transitions only to itself and that generates only rewards of zero.

A policy π defines the probability $\pi(s, a)$ of taking action a when in state s. An optimal policy $\pi^*(s)$ is a policy that maximises some definition of long term reward, e.g. the expected value of the sum of discounted future rewards. The *state-value function for policy* π is the expected return starting from s and following π thereafter

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left\{ R_t | s_t = s \right\} \\ &= \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \end{aligned} \tag{4}$$

The *action-value function for policy* π is the expected return starting from s, taking action a and following π thereafter

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi \left\{ R_t | s_t = s, a_t = a \right\} \\ &= \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \end{aligned} \tag{5}$$

The Bellman equation for V^π is easily derived ([10] eqn 3.10) and shows the recursive relation

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} \mathcal{P} \left[\mathcal{R} + \gamma V^\pi(s') \right] \tag{6}$$

1.2 Optimal Policies and Value Functions

In MDPS there is always at least one deterministic policy π that is better than or equal to all other policies π'. $\pi \geq \pi'$ iff $V^\pi(s) \geq V^{\pi'}(s), \forall s \in \mathcal{S}$ This is an *optimal policy.*

All optimal policies share the same *optimal state-value function*

$$V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S} \tag{7}$$

and the same *optimal action-value function*

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \\ &= \mathbb{E}\left\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right\} \end{aligned} \tag{8}$$

The Bellman optimality equations are easily derived ([10] eqn 3.15)

$$\begin{aligned} V^*(s) &= \max_a Q^*(s,a) \\ &= \max_a \mathbb{E}\left\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right\} \\ &= \max_a \sum_{s'} \mathcal{P}\left[\mathcal{R} + \gamma V^*(s')\right] \end{aligned} \tag{9}$$

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\left\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right\} \\ &= \mathbb{E}\left\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a\right\} \\ &= \sum_{s'} \mathcal{P}\left\{\mathcal{R} + \gamma \max_{a'} Q^*(s', a')\right\} \end{aligned} \tag{10}$$

1.3 Time-Dependent MDPs

Boyan and Littman [2] propose the Time-Dependent MDP, a finite discrete MDP extended with a continuous time dimension included in the state space. Given a number of restrictions, including value functions and rewards that are piecewise linear in the time variable and a discrete state transition model, there is an exact solution.

1.4 Structured Continuous MDPs

Feng et al. [5] examined more general continuous state MDPs and observed that many problems, for example from the Mars Rover domain, feature:

- continuous state space;
- finite set of goals with positive utility; and
- resource constraints;

that result in an optimum value function V^* with various humps and plateaus, each representing a region of state-space where a particular goal (or set of goals) can be reached. The minimum resource required for some actions leads to sharp transitions. This structure can be exploited in an approximate solution by aggregating states with similar values.

1.5 Continuous State Value Representation

Feng et al. [5] used kd-trees to recursively divide the space by hyperplanes perpendicular to the axes to represent Rectangular Piece-wise Constant (RPWC) or Linear (RPWL) functions. A Bellman backup creates non-linear regions when it performs the maximization step on linear functions, but such functions can be incorporated into the model. The value functions are now modelled by piece-wise functions where each rectangular region of the state space is modelled by the maximum of a set of linear functions. Pruning is performed to remove dominated linear functions. As in POMDPs, dominance is computed by solving a linear program.

1.6 Lazy approximation for solving continuous finite-horizon MDPs

When the transition function has the form of a continuous relative PDF

$$T(x', x, a) = \Pr(x' - x | a) \tag{11}$$

where $x, x' \in X$ and X is the continuous state space, the Bellman backup integral

$$V^{n+1}(x) = \max_{a \in A} \left\{ R(x, a) + \int_X T(x', x, a) V^n(x') dx' \right\} \tag{12}$$

becomes a convolution. Assuming T and R are RPWC, each backup step increases the polynomial order of V^{n+1} by one, i.e. if V^n is RPWC then V^{n+1} is RPWL and V^{n+2} is rectangular piece-wise quadratic (RPWQ). If we approximate a RPWL function V^{n+1} with a RPWC function $\bar{V}^{n+1}$ at each step we can limit the order of the result to RPWL [7].

Rather than discretizing a continuous transition function and thereafter applying a method designed for a discrete transition function, this method can vary discretization (i.e. the region definitions) as the value function evolves, providing finer discretization only where the function is changing quickly.

1.7 Algebraic Decision Diagrams

A Binary Decision Diagram (BDD) [3] is a tree representing a function $\mathbb{B}^n \to \mathbb{B}$ (see figure 2).

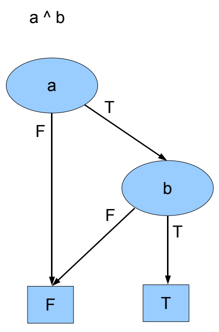


Figure 2: Binary Decision Diagram

An Algebraic Decision Diagram (ADD) [1] is obtained by replacing the Boolean values in the terminal nodes of a Binary Decision Diagram with real values. An

Algebraic Decision Diagram is a tree representing a function $\mathbb{B}^n \rightarrow \mathbb{R}$ (see figure 3).

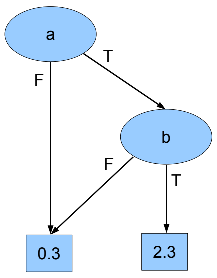


Figure 3: Algebraic Decision Diagram

1.8 Canonical Binary and Algebraic Decision Diagrams

A diagram is transformed into canonical form by applying the following node ordering and reduction rules.

1.8.1 Variable Ordering

Along any path from root to leaf:

1. no variable appears more than once; and
2. the variables always appear in the same order.

1.8.2 Reduction

A reduced diagram has:

1. any duplicate (same label and same children) nodes merged; and
2. any node with both children the same removed (as the decision is redundant).

BDDs and ADDs have several useful properties [6]. *"First, for a given variable ordering, each distinct function has a unique reduced representation. In addition, many common functions can be represented compactly because of isomorphic-subgraph sharing. Furthermore, efficient algorithms (e.g., depth-first search with a hash table to reuse previously computed results) exist for most common operations, such as addition, multiplication, and maximization."*

2 Extended Algebraic Decision Diagrams

This project extends Algebraic Decision Diagrams by adding two new types of nodes.

- non-terminal nodes that perform boolean valued tests on a single real valued variable; and
- terminal nodes that are polynomial functions of real valued parameters.

Such an Extended Algebraic Decision Diagram (XADD) is a tree representing a function $\langle \mathbb{B}^n, \mathbb{R}^m \rangle \to \mathbb{R}$ (see figure 4).

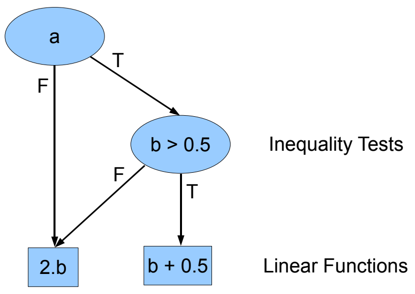


Figure 4: Extended Algebraic Decision Diagram

XADDs present addition redundancies over ADDs so their reduction rules are an augmentation of those presented in section 1.8.2. Figures 5 and 6 show XADDs with a redundancy on the left side and the equivalent reduced form on the right.

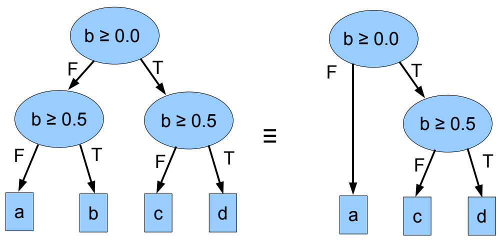


Figure 5: XADD Reduction. The leftmost $b \geq 0.5$ node in the left XADD is redundant because it can never be true.

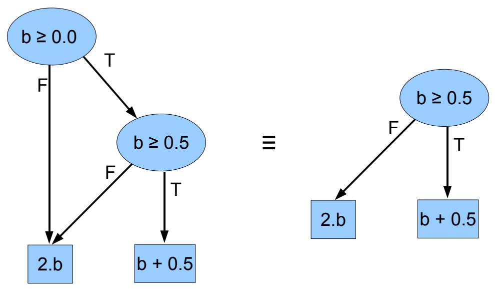


Figure 6: XADD Reduction. The test at the root of the left XADD is redundant because it and its true child share the same false child.

Extended Algebraic Decision Diagrams (XADDs) represent a useful special case of piece-wise polynomial functions, where the pieces are restricted to hyper-rectangles. They provide a compact representation that is amenable to computation. XADDs combine the computational advantages of BDDs/ADDs discussed in the previous section with the features of previously proposed methods of treating MDPs with continuous variables.

- XADDs allow time and other continuous variables to be naturally combined with Boolean variables as in section 1.3.
- The granularity of discretization can vary to suit the structure of the function being represented as in section 1.4
- Approximating a higher order polynomial with one of a lower order in each value iteration will adjust the discretization to suit the structure of the new function as in section 1.6.

A terminal node type for the maximum of a set of linear functions, as in section 1.5, could also be added.

3 Software Implementation of XADDs

The UML class diagrams in this section were produced using ArgoUML [11].

3.1 Nodes

A concrete class is defined for each of the terminal node types (see figure 7):

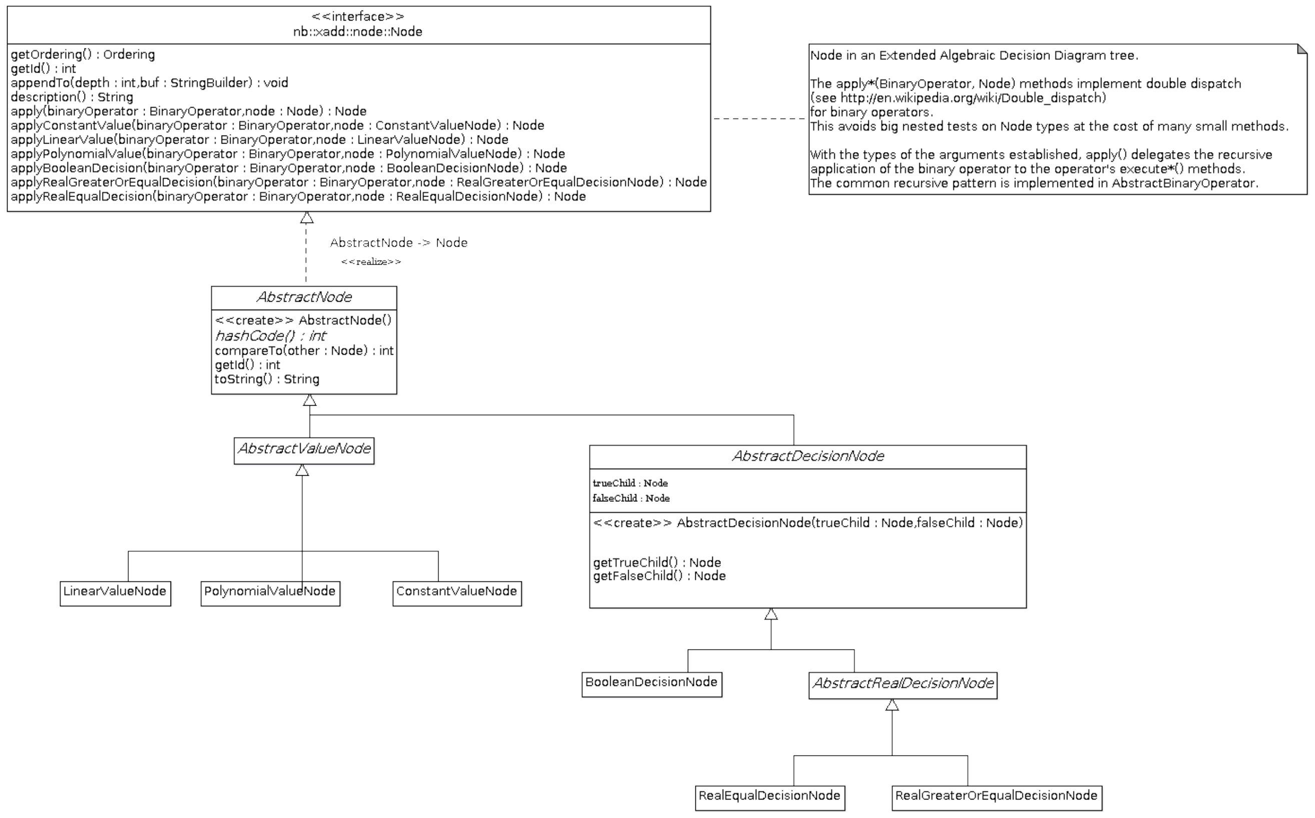
1. constant real values;
2. linear functions of real variables; and
3. arbitrary order polynomial functions of real variables;

and for each of the non-terminal node types:

1. test a Boolean variable;
2. test a real variable for equality to a fixed value;
3. inequality comparison (>=) of a real variable to a fixed value.

The *apply*()* methods implement triple dispatch [12] for binary operators. This avoids large nested tests on Node types at the cost of many small methods. With the types of the arguments established, *apply()** delegates the recursive application of the binary operator to the operator's *BinaryOperator.execute*()* methods. The common recursive pattern for the application of binary operators [6, 8] is implemented in *AbstractBinaryOperator*.

Nodes are created by a Factory as described in the following section. Constructors are not publicly accessible to enforce use of the Factory. Nodes are immutable to facilitate object reuse.



```
getOrdering() : Ordering
getId() : int
appendTo(depth : int,buf : StringBuilder) : void
description() : String
apply(binaryOperator : BinaryOperator,node : Node) : Node
applyConstantValue(binaryOperator : BinaryOperator,node : ConstantValueNode) : Node
applyLinearValue(binaryOperator : BinaryOperator,node : LinearValueNode) : Node
applyPolynomialValue(binaryOperator : BinaryOperator,node : PolynomialValueNode) : Node
applyBooleanDecision(binaryOperator : BinaryOperator,node : BooleanDecisionNode) : Node
applyRealGreaterOrEqualDecision(binaryOperator : BinaryOperator,node : RealGreaterOrEqualDecisionNode) : Node
applyRealEqualDecision(binaryOperator : BinaryOperator,node : RealEqualDecisionNode) : Node
```

Node in an Extended Algebraic Decision Diagram tree.

The apply*(BinaryOperator, Node) methods implement double dispatch
(see http://en.wikipedia.org/wiki/Double_dispatch)
for binary operators.
This avoids big nested tests on Node types at the cost of many small methods.

With the types of the arguments established, apply() delegates the recursive
application of the binary operator to the operator's execute*() methods.
The common recursive pattern is implemented in AbstractBinaryOperator.

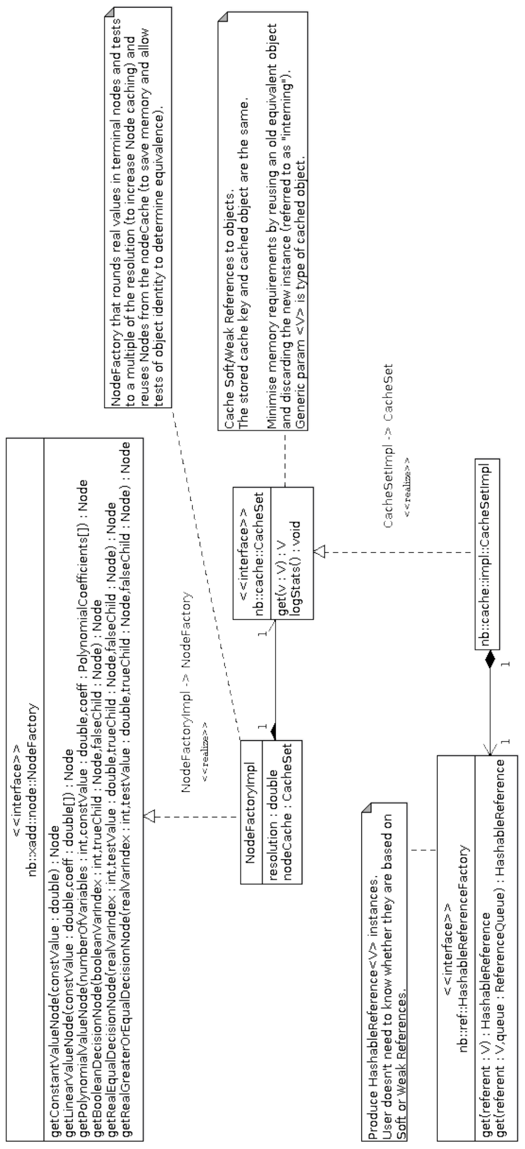
```
<<create>> AbstractNode()
hashCode() : int
compareTo(other : Node) : int
getId() : int
toString() : String
```

```
trueChild : Node
falseChild : Node
<<create>> AbstractDecisionNode(trueChild : Node,falseChild : Node)
getTrueChild() : Node
getFalseChild() : Node
```

Figure 7: XADD Node type hierarchy. Terminal nodes descend from *Abstract-ValueNode* and non-terminal nodes descend from *AbstractDecisionNode*.

3.2 NodeFactory

The implementation of the NodeFactory caches Nodes that it has produced (see figure 8). When requested to produce a Node equivalent to one in the cache, it returns the cached object (this is refered to as *interning* [13]). Interning conserves memory and allows testing for object identity (Java: $a == b$) to substitute for testing for equivalence (Java: *a.equals(b)*). Importantly interning facilitates the construction of reduced XADDs by the NodeFactory.

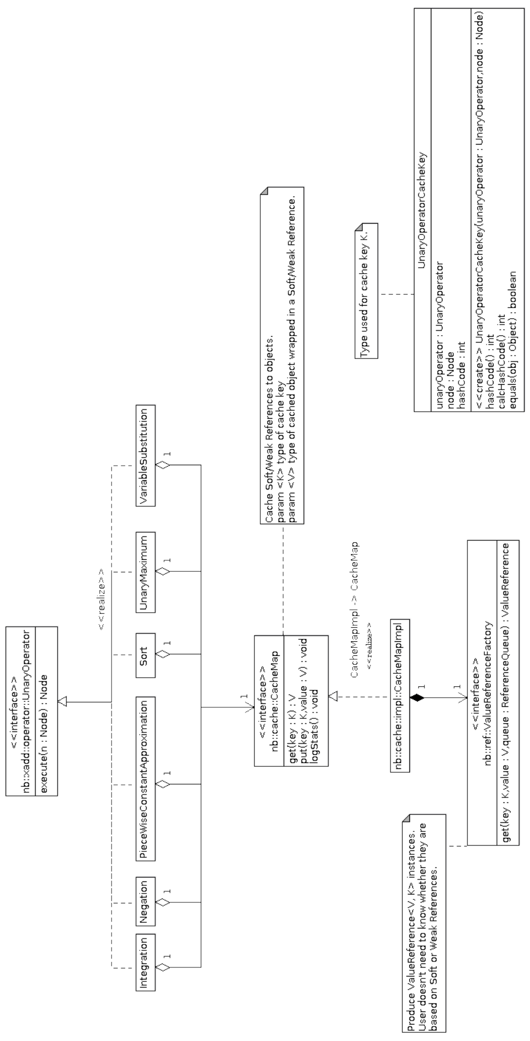


NodeFactory that rounds real values in terminal nodes and tests to a multiple of the resolution (to increase Node caching) and reuses Nodes from the nodeCache (to save memory and allow tests of object identity to determine equivalence).
Cache Soft/Weak References to objects. The stored cache key and cached object are the same.
Minimise memory requirements by reusing an old equivalent object and discarding the new instance (referred to as "interning"). Generic param <V> is type of cached object.
<<interface>> nb::xadd::node::NodeFactory
getConstantValueNode(constValue : double) : Node
getLinearValueNode(constValue : double,coeff : double[]) : Node
getPolynomialValueNode(numberOfVariables : int,constValue : double,coeff : PolynomialCoefficients[]) : Node
getBooleanDecisionNode(booleanVarIndex : int,trueChild : Node,falseChild : Node) : Node
getRealEqualDecisionNode(realVarIndex : int,testValue : double,trueChild : Node,falseChild : Node) : Node
getRealGreaterOrEqualDecisionNode(realVarIndex : int,testValue : double,trueChild : Node,falseChild : Node) : Node
NodeFactoryImpl -> NodeFactory
<<realize>>
NodeFactoryImpl
resolution : double
nodeCache : CacheSet
1
1
<<interface>> nb::cache::CacheSet
get(v : V) : V
logStats() : void
CacheSetImpl -> CacheSet
<<realize>>
nb::cache::impl::CacheSetImpl
1
1
Produce HashableReference<V> instances. User doesn't need to know whether they are based on Soft or Weak References.
<<interface>> nb::ref::HashableReferenceFactory
get(referent : V) : HashableReference
get(referent : V,queue : ReferenceQueue) : HashableReference

Figure 8: NodeFactory Caching

3.3 Operators

As for Nodes, Operators have non-publicly accessible constructors to enforce use of a Factory (not shown in figures) and are immutable to facilitate object sharing. Operators cache their results to avoid recomputation (see figures 9 and 10). Operators require canonical (see section 1.8) XADDs as arguments and use the NodeFactory to build their result, also in the form of a canonical XADD. An exception is the *Sort* unary operator that accepts a non-canonical argument and produces its canonical form. The implementation of many Operators is incomplete in that polynomial and/or linear function terminal nodes are not always handled. In such cases a *NotYetImplemented* exception is thrown.



<<interface>>
nb::xadd::operator::UnaryOperator
execute(n : Node) : Node
<<realize>>
Integration
Negation
PieceWiseConstantApproximation
Sort
UnaryMaximum
VariableSubstitution
1
<<interface>>
nb::cache::CacheMap
get(key : K) : V
put(key : K,value : V) : void
logStats() : void
Cache Soft/Weak References to objects.
param <K> type of cache key
param <V> type of cached object wrapped in a Soft/Weak Reference.
CacheMapImpl -> CacheMap
<<realize>>
nb::cache::impl::CacheMapImpl
Produce ValueReference<V, K> instances.
User doesn't need to know whether they are
based on Soft or Weak References.
<<interface>>
nb::ref::ValueReferenceFactory
get(key : K,value : V,queue : ReferenceQueue) : ValueReference
Type used for cache key K.
UnaryOperatorCacheKey
unaryOperator : UnaryOperator
node : Node
hashCode : int
<<create>> UnaryOperatorCacheKey(unaryOperator : UnaryOperator,node : Node)
hashCode() : int
calcHashCode() : int
equals(obj : Object) : boolean

Figure 9: Unary Operators

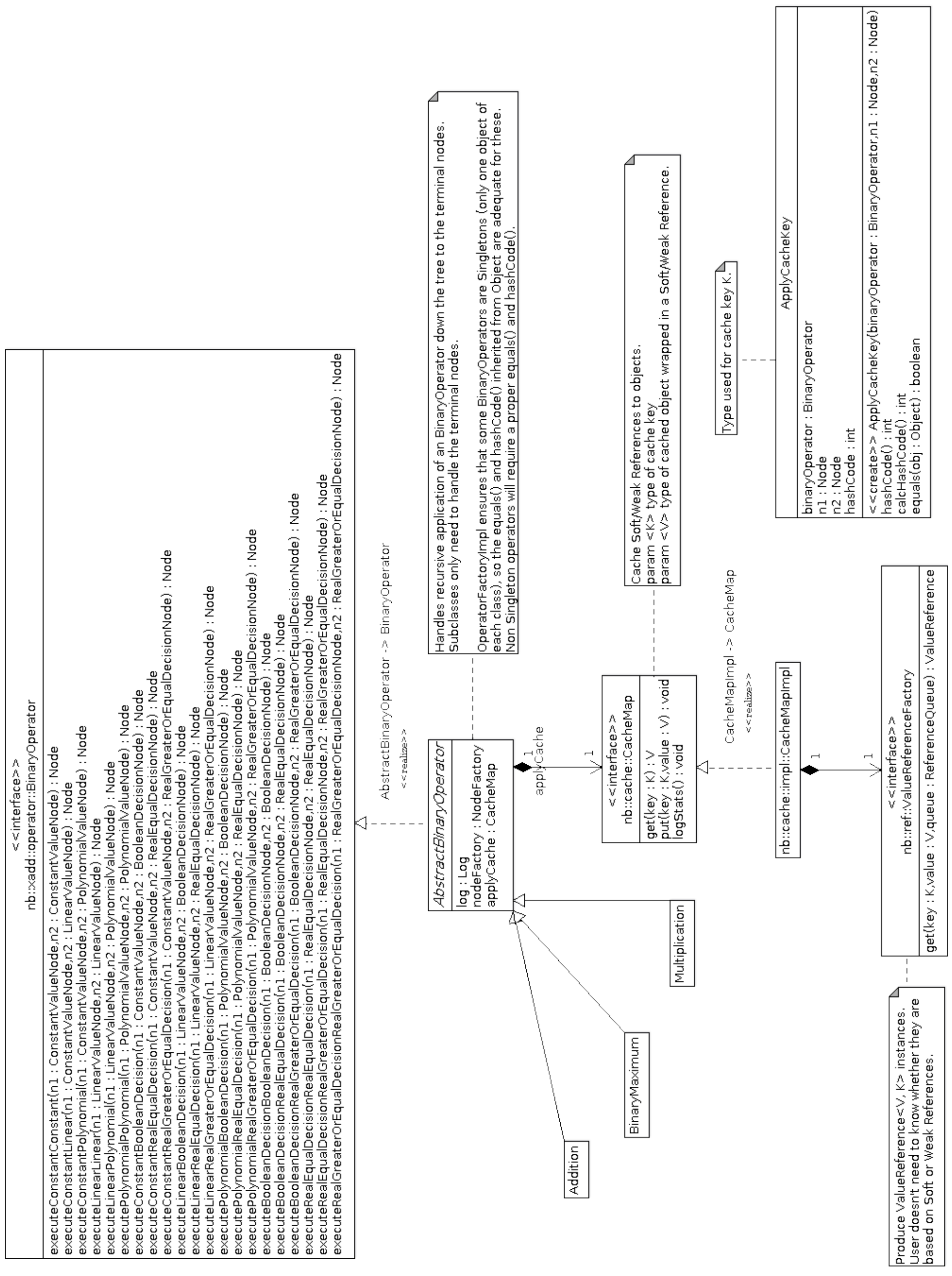

<<interface>>
nb::xadd::operator::BinaryOperator
executeConstantConstant(n1 : ConstantValueNode,n2 : ConstantValueNode) : Node
executeConstantLinear(n1 : ConstantValueNode,n2 : LinearValueNode) : Node
executeConstantPolynomial(n1 : ConstantValueNode,n2 : PolynomialValueNode) : Node
executeLinearLinear(n1 : LinearValueNode,n2 : LinearValueNode) : Node
executeLinearPolynomial(n1 : LinearValueNode,n2 : PolynomialValueNode) : Node
executePolynomialPolynomial(n1 : PolynomialValueNode,n2 : PolynomialValueNode) : Node
executeConstantBooleanDecision(n1 : ConstantValueNode,n2 : BooleanDecisionNode) : Node
executeConstantRealEqualDecision(n1 : ConstantValueNode,n2 : RealEqualDecisionNode) : Node
executeConstantRealGreaterOrEqualDecision(n1 : ConstantValueNode,n2 : RealGreaterOrEqualDecisionNode) : Node
executeLinearBooleanDecision(n1 : LinearValueNode,n2 : BooleanDecisionNode) : Node
executeLinearRealEqualDecision(n1 : LinearValueNode,n2 : RealEqualDecisionNode) : Node
executeLinearRealGreaterOrEqualDecision(n1 : LinearValueNode,n2 : RealGreaterOrEqualDecisionNode) : Node
executePolynomialBooleanDecision(n1 : PolynomialValueNode,n2 : BooleanDecisionNode) : Node
executePolynomialRealEqualDecision(n1 : PolynomialValueNode,n2 : RealEqualDecisionNode) : Node
executePolynomialRealGreaterOrEqualDecision(n1 : PolynomialValueNode,n2 : RealGreaterOrEqualDecisionNode) : Node
executeBooleanDecisionBooleanDecision(n1 : BooleanDecisionNode,n2 : BooleanDecisionNode) : Node
executeBooleanDecisionRealEqualDecision(n1 : BooleanDecisionNode,n2 : RealEqualDecisionNode) : Node
executeBooleanDecisionRealGreaterOrEqualDecision(n1 : BooleanDecisionNode,n2 : RealGreaterOrEqualDecisionNode) : Node
executeRealEqualDecisionRealEqualDecision(n1 : RealEqualDecisionNode,n2 : RealEqualDecisionNode) : Node
executeRealEqualDecisionRealGreaterOrEqualDecision(n1 : RealEqualDecisionNode,n2 : RealGreaterOrEqualDecisionNode) : Node
executeRealGreaterOrEqualDecisionRealGreaterOrEqualDecision(n1 : RealGreaterOrEqualDecisionNode,n2 : RealGreaterOrEqualDecisionNode) : Node
AbstractBinaryOperator -> BinaryOperator
<<realize>>
AbstractBinaryOperator
log : Log
nodeFactory : NodeFactory
applyCache : CacheMap
Handles recursive application of an BinaryOperator down the tree to the terminal nodes.
Subclasses only need to handle the terminal nodes.
OperatorFactoryImpl ensures that some BinaryOperators are Singletons (only one object of each class), so the equals() and hashCode() inherited from Object are adequate for these.
Non Singleton operators will require a proper equals() and hashCode().
Addition
BinaryMaximum
Multiplication
1
applyCache
1
<<interface>>
nb::cache::CacheMap
get(key : K) : V
put(key : K,value : V) : void
logStats() : void
Cache Soft/Weak References to objects.
param <K> type of cache key
param <V> type of cached object wrapped in a Soft/Weak Reference.
CacheMapImpl -> CacheMap
<<realize>>
nb::cache::impl::CacheMapImpl
1
1
<<interface>>
nb::ref::ValueReferenceFactory
get(key : K,value : V,queue : ReferenceQueue) : ValueReference
Produce ValueReference<V, K> instances.
User doesn't need to know whether they are based on Soft or Weak References.
Type used for cache key K.
ApplyCacheKey
binaryOperator : BinaryOperator
n1 : Node
n2 : Node
hashCode : int
<<create>> ApplyCacheKey(binaryOperator : BinaryOperator,n1 : Node,n2 : Node)
hashCode() : int
calcHashCode() : int
equals(obj : Object) : boolean

Figure 10: Binary Operators

3.4 Caching

Java provides the classes *java.lang.ref.WeakReference*, *java.lang.ref.SoftReference* and *java.lang.ref.ReferenceQueue* to allow applications to create memory sensitive caches that interact with the garbage collector. References (both Weak and Soft) hold a reference to an object (the *referent*) that may be cleared by the garbage collector under different circumstances. WeakReferences are cleared as soon as there are no hard or soft references to the referent. SoftReferences are cleared only when there are no hard references to the referent and the garbage collector is running short of memory. In this case some internal policy determines which SoftReferences are cleared.

References may be associated with a ReferenceQueue, in which case the Reference is added to the ReferenceQueue at some time after the garbage collector has cleared it.

Each of the caches discussed in previous sections can be configured to use either weak or soft references. The type of reference used is determined by the implementation of the *HashableReferenceFactory* or *ValueReferenceFactory* given to the cache (see figure 11). The caches use a ReferenceQueue to remove entries for items that have been cleared.

The unit tests for the caches use WeakReferences because these provide predictable clearing behaviour independent of memory usage and garbage collector policies. Application code may prefer to use SoftReferences to keep cache entries for as long as possible in case they are needed again.

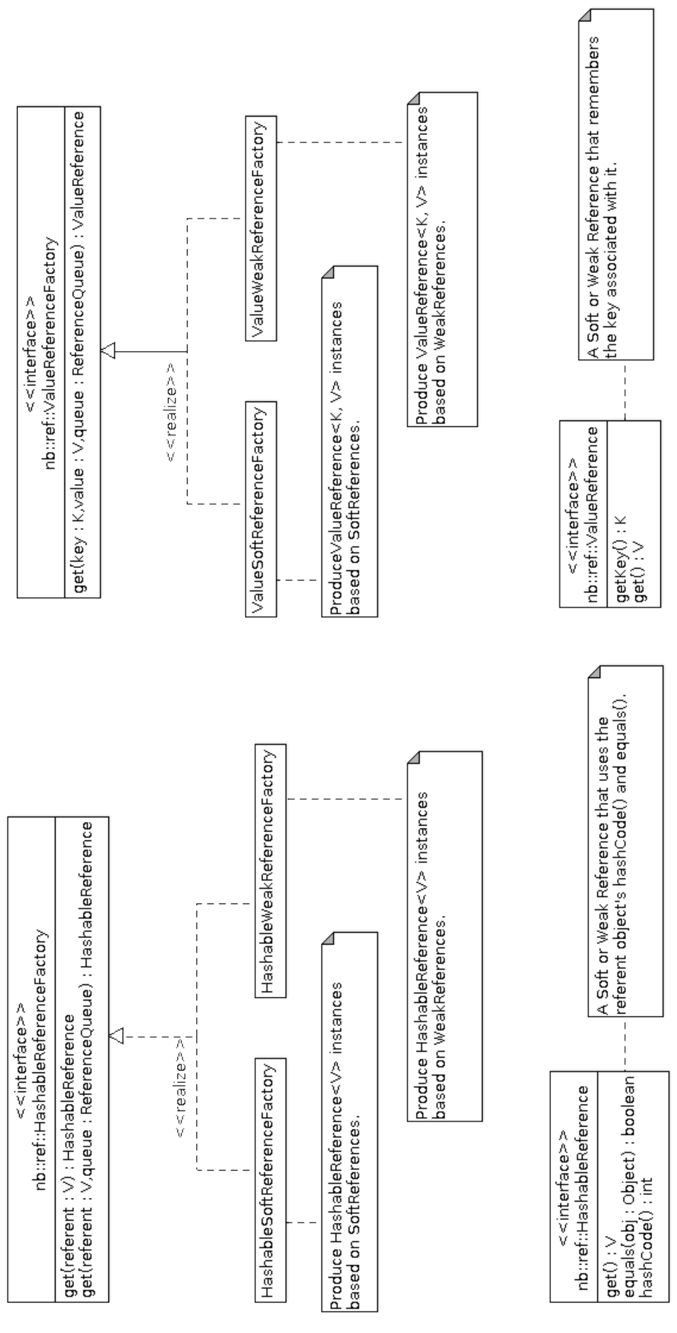

<<interface>>
nb::ref::ValueReferenceFactory
get(key : K,value : V,queue : ReferenceQueue) : ValueReference
<<realize>>
ValueSoftReferenceFactory
ValueWeakReferenceFactory
ProduceValueReference<K, V> instances based on SoftReferences.
Produce ValueReference<K, V> instances based on WeakReferences.
<<interface>>
nb::ref::ValueReference
getKey() : K
get() : V
A Soft or Weak Reference that remembers the key associated with it.
<<interface>>
nb::ref::HashableReferenceFactory
get(referent : V) : HashableReference
get(referent : V,queue : ReferenceQueue) : HashableReference
<<realize>>
HashableSoftReferenceFactory
HashableWeakReferenceFactory
Produce HashableReference<V> instances based on SoftReferences.
Produce HashableReference<V> instances based on WeakReferences.
<<interface>>
nb::ref::HashableReference
get() : V
equals(obj : Object) : boolean
hashCode() : int
A Soft or Weak Reference that uses the referent object's hashCode() and equals().

Figure 11: Factories produce either Weak or Soft References for use in caches.

3.5 Testing

A large suite of unit tests is run each time the software is built. The tests cover all major classes and provide a framework for quickly adding new tests to demonstrate bugs discovered or new classes and methods added. Test code has the valuable side benefit of providing example usage code or *executable documentation* that the build process guarantees is current.

4 Graphics

The data plotted in the following figures is extracted from the XADD representations used in the computations. The data is extracted by the Java class *ExportVisualizationData*. When a value is bounded on both sides by inequality tests, this class outputs the mid-point between the bounds, but when it is only bounded on one side then that bound is output. The resulting distortion at the edges is apparent in figure 17. Similar distortions may be less apparent but still present in other figures. This distortion is not present in the XADD representation and so does not affect calculations.

The 2D plots were produced with Python's *matplotlib* [9]. The 3D plots were produced using the *mlab* Python interface to *Mayavi2* [4]. Lack of time and familiarity has prevented the addition of grids, dimensions, bounding boxes etc., however the axes have been placed closest to the corner representing the origin. Mayavi2 was installed on Ubuntu linux according to the instructions at `http://debs.astraw.com/hardy/` (the version in the standard Ubuntu repositories was not current).

5 Traffic Control Domain

The initial application domain is a traffic controller for a single intersection. The intersection model has been kept as simple as possible. The state model $\langle s, \vec{q} \rangle$ consists of

- the cycle state $s \in \{s_1, \ldots, s_m\}$ determines which directions get a green light and which get a red,
- the number of cars in each of n lane queues $\vec{q} = \langle q_1, \ldots, q_n \rangle$ where each $q_i \in [0, \infty]$ (we make the simplifying assumption that each lane queue is fully observable).

The model could be extended to include:

- multiple intersections (including simple models of traffic as they travel between intersections),
- pedestrian push puttons,
- skipping a cycle if the queue is short in order to clear a longer queue on another road (as long as fairness is ensured).

5.1 Control Model

We assume the only control action at any stage in the cycle is how long to remain in that cycle stage. Thus, we model the action as a continuous waiting time t on the interval $t \in [\mathit{MinCycleTime}, \mathit{MaxCycleTime}]$.

5.2 Traffic Model

The traffic model updates the queue lengths at each stage as the difference between an *arrival model* and a *clearance model*.

The proposed arrival model is Poisson, with the probability of k cars arriving in the interval t given by:

$$\Pr(k|t) = \frac{(\lambda \frac{t}{l})^k e^{-\lambda \frac{t}{l}}}{k!} \tag{13}$$

where λ is the mean number of arrivals in the interval l. We model the queue length as a continuous real variable, so our arrival model is actually a continuous distribution that matches the Poisson distribution at integer values and interpolates between these values.

The proposed clearance model for the queue whose light is green is represented as a deterministic function

$$f_{clr}(s,t) = \begin{cases} \text{light for } i\text{th queue is green in } s: & \text{see graph below} \\ \text{light for } i\text{th queue is not green in } s: & 0 \end{cases} \tag{14}$$

where we model the clearance rate when the appropriate light is green as the following piecewise linear function of action duration t:

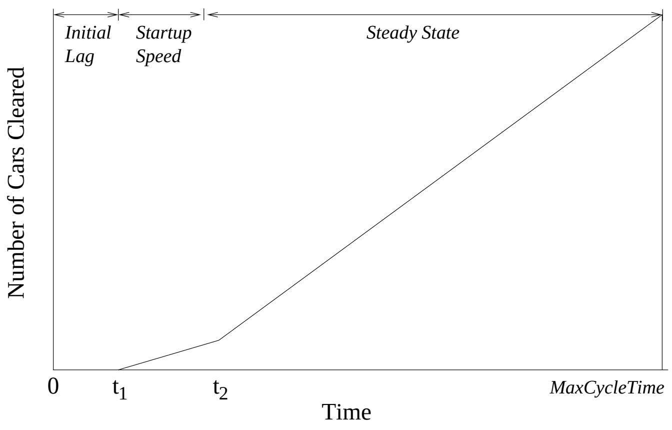


Figure 12: Cars cleared as a function of green light duration

With this, we arrive at the following traffic transition model specifying the probability of the length of the ith queue as a result of an action t:

$$P(q_i'|q_i,s,t) = \int_{q_i^a=-\infty}^{\infty} P(q_i', q_i^a|q_i,s,t)dq_i^a \tag{15}$$

$$= \int_{q_i^a=-\infty}^{\infty} P(q_i'|q_i^a,s,t)P(q_i^a|q_i,t)dq_i^a \tag{16}$$

In (15) and its factored form in (16), we break down the transition model for each queue length q_i in terms of an arrival quantity q_i^a (dependent upon the cycle duration t) and the final queue length q_i' (after the number of cars cleared in time t and state s) has been subtracted from q_i^a subject to positive queue constraints. These two probabilities are respectively defined using the arrival model P_{arr} defined previously in (13) and Dirac delta functions as follows:

$$P(q_i^a|q_i,t) = P_{arr}(k = q_i^a - q_i|t) \tag{17}$$

$$P(q_i'|q_i^a) = \begin{cases} q_i^a - f_{clr}(s,t) \geq 0: & \delta[q_i' - (q_i^a - f_{clr}(s,t))] \\ q_i^a - f_{clr}(s,t) < 0: & \delta[q_i'] \end{cases} \tag{18}$$

where equation 18 encodes the constraint that the number of cars cleared is limited to the sum of the number of cars in the queue and the number that arrive.

Noting that $q_i^a \geq q_i \geq 0$ we have

$$\begin{aligned} P(q_i'|q_i,s,t) &= \int_{q_i^a=q_i}^{f_{clr}(s,t)} P(q_i'|q_i^a)P_{arr}(k = q_i^a - q_i|t)dq_i^a \\ &\quad + \int_{q_i^a=f_{clr}(s,t)}^{\infty} P(q_i'|q_i^a)P_{arr}(k = q_i^a - q_i|t)dq_i^a \\ &= \int_{q_i^a=q_i}^{f_{clr}(s,t)} \delta[q_i']P_{arr}(k = q_i^a - q_i|t)dq_i^a \\ &\quad + \int_{q_i^a=f_{clr}(s,t)}^{\infty} \delta[q_i' - (q_i^a - f_{clr}(s,t))]P_{arr}(k = q_i^a - q_i|t)dq_i^a \\ &= \delta[q_i'] \int_{k=0}^{f_{clr}(s,t)-q_i} P_{arr}(k|t)dk \\ &\quad + P_{arr}(k = q_i' - q_i + f_{clr}(s,t)|t) \\ &= \begin{cases} q_i' = 0: & \int_{k=0}^{f_{clr}(s,t)-q_i} P_{arr}(k|t)dk \\ q_i' > 0: & P_{arr}(k = q_i' - q_i + f_{clr}(s,t)|t) \end{cases} \end{aligned} \tag{19}$$

This represents the intuitive idea that all parts of the arrival distribution that would result in a negative queue length (were that allowed) are piled up at $q' = 0$. The following figures show data extracted from the XADD representation of this distribution.

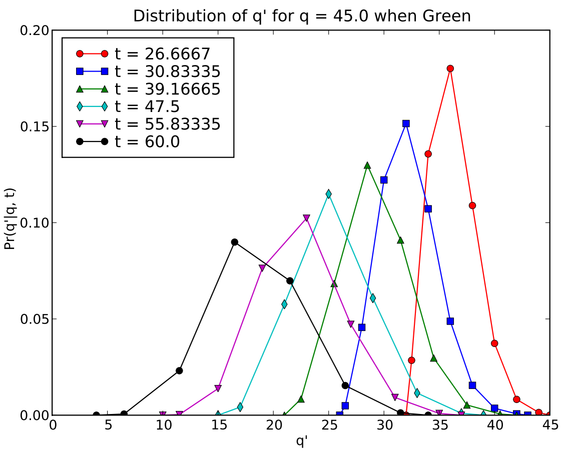


Figure 13: Distribution of final queue length q' for a fixed initial length $q = 45$ cars and various green light durations t sec. Other parameters are as described in section 6 (the mean arrival rate is 0.25 cars/sec). The dominance of clearance over arrivals is greater for higher values of t.

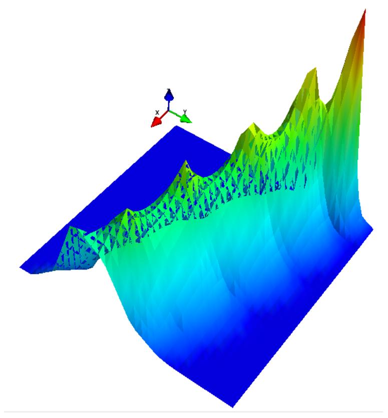


Figure 14: The same data as figure 13 plotted in 3D: X(red axis) $= t$, Y(green axis) $= q'$, Z(blue axis) $= Pr(q'|q,t)$. The origin is at the bottom centre back corner. The troughs between the peaks, which correspond to actual data points, are an artifact of the 3D interpolation between the data points.

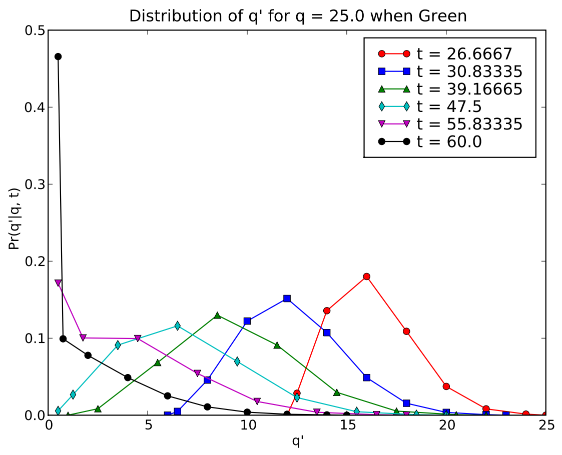


Figure 15: Distribution of final queue length q' for a shorter fixed initial length q and various green light durations t. In this case the shorter initial q allows the queue to be emptied ($q' = 0$) for the larger values of t. The accumulation of probability at $q' = 0$ is the result of the first line of equation 19

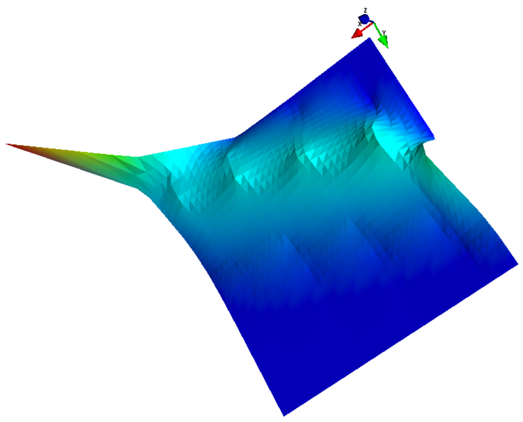


Figure 16: The same data as figure 15 plotted in 3D: X(red axis) $=$ t, Y(green axis) $= q'$, Z(blue axis) $= Pr(q'|q,t)$. The origin is at the bottom centre back corner (at the top of the figure).

We must also represent the cycle transition, which is independent of the cycle duration. We do this using the Kronecker delta function and modulo arithmetic:

$$P(s'|s = s_i, t) = P(s'|s) = \delta[s' = s_{(i \bmod m)+1}] \tag{20}$$

The cumbersome mod calculation is required because state indices start at s_1.

The Bellman backup integral for our formulation is

$$\begin{aligned}
V^{n+1}(s,\vec{q}) &= \max_t \sum_{s'} \int_{\vec{q'}} T(s',\vec{q'}|s,\vec{q},t) \left[R(s',\vec{q'},t,s,\vec{q}) + \gamma V^n(s',\vec{q'})\right] ds' d\vec{q'} \\
&= \max_t \left[R(t,s,\vec{q}) + \sum_{s'} \int_{\vec{q'}} T(s',\vec{q'}|s,\vec{q},t)\gamma V^n(s',\vec{q'}) ds' d\vec{q'}\right] \\
&= \max_t \left[R(t,s,\vec{q}) + \int_{\vec{q'}} \delta[s' = s_{(i \bmod m)+1}] \prod_{i=1}^{n} P(q'_i|q_i,s,t)\gamma V^n(s',\vec{q'}) ds' d\vec{q'}\right] \\
&= \max_t \left[R(t,s,\vec{q}) + \int_{\vec{q'}} \prod_{i=1}^{n} P(q'_i|q_i,s,t)\gamma V^n(s_{(i \bmod m)+1},\vec{q'}) d\vec{q'}\right]
\end{aligned} \tag{21}$$

assuming a reward function R independent of $s', \vec{q'}$, so it can come out of the integral.

5.3 Reward

A possible choice for the immediate reward $R(t,s,\vec{q})$ is a function proportional to the negative sum of the queue lengths multiplied by the action duration t:

$$R(t,\vec{q}) \propto -t\sum_i q_i \tag{22}$$

This penalises longer queues and especially those held for long durations, however it unfairly penalises the queue with a green light in the case where the queue is largely cleared during the green light duration. To correct for this we linearly discount the green queue to zero at time $t_{i,clr}$ given by the inverse of equation 14. The reward then becomes

$$R(t,\vec{q}) \propto = \sum_i \begin{cases} q_{i,t} \text{ has red light} & : -tq_i \\ q_{i,t} \text{ has green light and } t_{i,clr} \leq t & : -\frac{t_{i,clr}}{2} q_i \\ q_{i,t} \text{ has green light and } t_{i,clr} > t & : -t(q_{i,t} + \frac{q_i - q_{i,t}}{2}) \end{cases} \tag{23}$$

where $q_{i,t} = \dfrac{t_{i,clr} - t}{t_{i,clr}} q$.

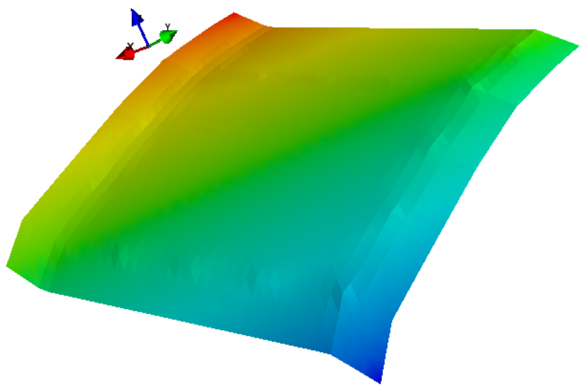


Figure 17: Reward for $t = 35$: X(red axis) $= q_{North}$, Y(green axis) $= q_{West}$, Z(blue axis) $= R(t, \vec{q})$. The origin is at bottom centre back corner (below the corner at the top centre of the figure). The kink near the edges is due to the edge distortion in the visualization of XADDs, discussed in section 4.

6 Results

The reward function and the state transition PDF are represented by XADDs using real valued terminal nodes, i.e. a piece-wise constant representation. Plots of these XADDs were presented in sections 5.2 and 5.3. The value iteration defined by equation 21 requires the unary operations:

- maximisation (over t);
- integration (over q'); and
- variable substitution $(V(s, \vec{q}) \to V(s_{(i \bmod m)+1}, \vec{q'}))$;

and the binary operations:

- addition; and
- multiplication;

all of which have been implemented as XADD operators. Value iteration is performed on the XADDs to find the optimal value function. The PDF and reward are generated with range $[0, 1]$. At each iteration the value function is divided by its maximum to maintain its range within $[0, 1]$. The maximum is observed to be close to 2, consistent with the range of the PDF and reward.

We consider a scenario with only two queues, which we label q_{North} and q_{West}. The cycle state s can then be represented by a single Boolean variable $isNorthGreen$, when $true$, q_{North} has a green light and q_{West} has a red and *vice-verse*. The variable substitution $s \to s_{(i \bmod m)+1}$ then becomes a test negation, which has also been implemented as an XADD operator.

The parameters for the presented results are:

$qMax = 60 cars$ the maximum queue length considered

$t_{min} = 10sec$ the minimum cycle time considered

$t_{max} = 60sec$ the maximum cycle time considered

$arr_{North} = 0.25cars/sec$ mean arrival rate for q_{North}

$arr_{West} = 0.10cars/sec$ mean arrival rate for q_{West}

$resolution = 0.0001$ XADD values are rounded to a multiple of this

$discount = 1.0$ the discount rate γ in equation 21

$maxPieces = 6$ the maximum number of pieces for each dimension

$maxHeap = 800Mb$ the maximum heap size to be made available to the application (Java *-Xmx800M*)

$iterations = 10$ the maximum number of iterations

After completing the specified number of iterations, the program logs the following cache statistics. Expunging (Exp.) refers to the cache clearing entries because references have been cleared by the garbage collector and subsequently added to the caches reference queue (see section 3.4).

Cache	**Entries**	**Hits**	**Misses**	**Exp. Runs**	**Exp. Entries**
Node	751,031	13,143,606	4,556,446	12	3,805,415
Binary Op.	5,972,721	2,803,846	12,070,323	41,282	6,092,839
Unary Op.	8,073	1,534	8,073	0	0

The largest XADD constructed was for the integrand of equation 21 which depends on all the variables $(s, t, q_{North}, q_{West}, q'_{North}, q'_{West})$ and attained 136,585 nodes with a maximum depth of 56.

The following figures show the value function after 10 iterations. I believe the precipitous drop as either queue length goes to zero is an as yet unresolved error. The shape of the top surface is consistent with the reward definition, in particular the highest value region (red) extends further along the q_{North} axis when this queue has a green light, as the longer queue lengths are discounted by equation 23.

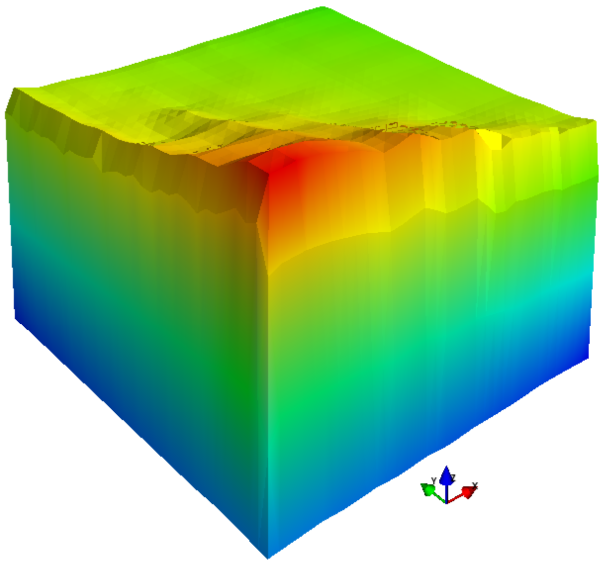


Figure 18: Value function for north light green after 10 iterations: X(red axis) = q_{North}, Y(green axis) = q_{West}, Z(blue axis) = $V(s = \text{North Green}, \vec{q})$. The origin is at the front bottom centre.

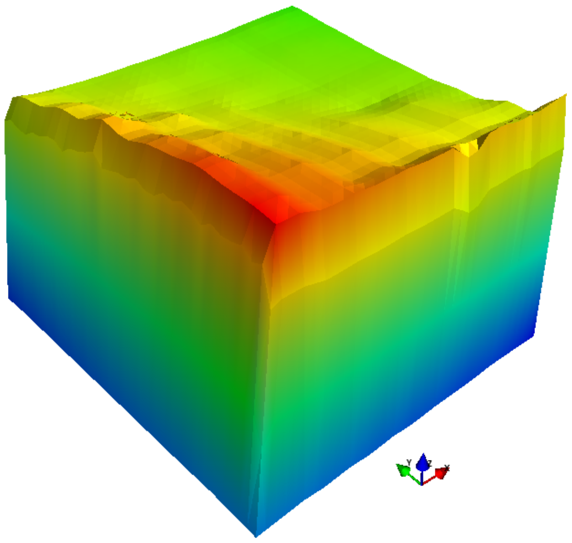


Figure 19: Value function for north light red after 10 iterations: X(red axis) = q_{North}, Y(green axis) = q_{West}, Z(blue axis) = $V(s = \text{North Red}, \vec{q})$. The origin is at the front bottom centre.

Further work is required to establish why the value function drops to low values when either queue length is zero and then to tune the traffic model to produce useful policies. The current model produces policies that always favour short cycle durations. Perhaps cycle duration will need to be included in the state.

Possible sources of error include:

- the model
  - reward
  - state transition PDF
- XADD representation
  - sufficiently accurate discretization
  - software bug in approximation or operators
- conversion from XADD to graphic representation

7 Conclusions

We have proposed XADDs as an efficient means of computation with continuous functions and successfully demonstrated their application in a continuous state MDP setting. The current software implementation includes incomplete support for terminal nodes that are linear functions and its scalability is limited due to its current reliance on piecewise constant representations. The following section proposes many enhancements that would allow XADDs to be applied to larger problems.

8 Future Directions

Traffic application:

- discover the cause of the sharp drop in value as either queue becomes empty;
- tune model to obtain useful policies.

XADD implementation:

- implement the *argmax* operator to facilitate extraction of policies;
- implement handling of *LinearValueNodes* by all operators. A RPWL approximation could represent complex functions using far fewer segments than a RPWC approximation and drastically increase scalability;
- create a terminal node that represents a linear function of the difference between variables in order to efficiently represent relative functions;
- consider implementing handling of *PolynomialValueNodes* by all operators or remove this class from the code;
- consider adding a terminal node type for the maximum of a set of linear functions, as discussed in section 1.5.

3D Visualisation (see section 4):

- improve exporting of data for visualizations;
- improve interpolation to avoid creating troughs between data points;
- add axes, grids, bounding boxes.

9 Acknowledgements

I would like to thank my supervisor Scott Sanner for proposing this project and for his support, help and encouragement; NICTA for allowing me to visit during this project and for the fine lectures their staff have given at ANU; Cambia, my employer for being very flexible; and Kerry, my recent bride for her love and patience.

References

[1] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 188–191, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[2] Justin A. Boyan and Michael L. Littman. Exact solutions to time-dependent MDPs. In *NIPS*, pages 1026–1032, 2000.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[4] Enthought. Mayavi2 - visualization of 3d data. `http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/`.

[5] Z. Feng, R. Dearden, N. Meuleau, and R. Washington. Dynamic programming for structured continuous markov decision problems, 2004.

[6] Jesse Hoey, Robert St-aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic Planning Using Decision Diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288. Morgan Kaufmann, 1999.

[7] Lihong Li and Michael L. Littman. Lazy approximation for solving continuous finite-horizon MDPs. In *National Conference on Artificial Intelligence*, 2005.

[8] S. Sanner. *First-order decision-theoretic planning in structured relational environments*. PhD thesis, University of Toronto, 2007. Also available as `http://users.rsise.anu.edu.au/~ssanner/Papers/Sanner_Scott_P_200803_PhD_thesis.pdf`.

[9] Sourceforge. matplotlib - a python 2d plotting library. `http://matplotlib.sourceforge.net/`.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. Also available as `http://www.cs.ualberta.ca/~sutton/book/ebook/the-book.html`.

[11] Tigris.org. Argouml - open source uml modeling tool. `http://argouml.tigris.org/`.

[12] Wikipedia. `http://en.wikipedia.org/wiki/Double_dispatch`.

[13] Wikipedia. `http://en.wikipedia.org/wiki/String_interning`.